Bachelor's thesis

Comparison of REST web architecture and GraphQL

Zarif Abdalimov

Department of Computer Graphics and Interaction
Supervisor: Bc. Petr Huťák

May 21, 2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Abdalimov** Jméno: **Zarif** Osobní číslo: **445379**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Srovnání webové architektury REST a GraphQL

Název bakalářské práce anglicky:

Comparison of REST web architecture and GraphQL

Pokyny pro vypracování:

Každý rok je trh IT naplněn novými technologiemi pro vytváření
webového API. Jednou z těchto nových a populárních technologií je
GraphQL. Cílem této práce je porovnat webové architektury REST a
GraphQL a zjistit výhody jednoho řešení oproti druhého.
Pro tento projekt je potřeba vytvořit webový server poskytující funkce,
které umí moderní webové servery, jako jsou CRUD operace nad entitami
systému, autorizace a autentizace, přenos souborů (dokumentů a obrázku), interakce mezi prohlížeči v reálném čase,
použiti technologií pro
zrychlení interakce mezi serverem a klientem (optimistic response a persisted queries).
Na základě zkušenosti z implementace funkčních požadavku web serveru
bude provedena analýza z pohledu rychlosti implementace, bezpečnosti a
celkové konzistence řešeni (dokumentace, možnost škálování a zpracování chyb).
Požadované výstupy BP:
1. Web engine, který používá jak GraphQL tak REST API.
2. React web a React native klientská aplikace splňující požadavky uvedené výše
3. Analyza výhod/nevýhod REST a GraphQL na základě zušeností z implementace.

Seznam doporučené literatury:

[1] – GraphQL dokumentace
[2] – Apollo GraphQL dokumentace
[3] - Web Services Architecture

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Bc. Petr Hurťák, katedra počítačové grafiky a interakce FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2020** Termín odevzdání bakalářské práce: _____

Platnost zadání bakalářské práce: **30.09.2021**

Bc. Petr Hurťák podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

Datum převzetí zadání

Podpis studenta

Acknowledgements

I would like to thank my supervisor Bc. Petr Huťták who guided me throughout this work.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 21, 2020 .....

Czech Technical University in Prague
Faculty of Electrical Engineering

Citation of this thesis

Abdalimov, Zarif. *Comparison of REST web architecture and GraphQL*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Electrical Engineering, 2020.

Abstrakt

Každý rok je trh IT naplněn novými technologiemi pro vytváření webového API. Jednou z těchto nových a populárních technologií je GraphQL. Cílem této práce je porovnat webové architektury REST a GraphQL a zjistit výhody jednoho řešení oproti druhého.

Klíčová slova web api, klientská aplikace

Abstract

Every year, the IT market is filled with new technologies for creating a web API. One of these new and popular technologies is GraphQL. The aim of this work is to compare REST and GraphQL web architectures and find out the advantages of one solution to another.

Keywords web api, client application

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Goal

The goal of this work is to compare web architectures GraphQL and REST and find out the advantages of one solution over another.

1.2 Outputs

The outputs of this work are:

1. Definition of the requirements
2. Analysis of GraphQL and REST web architecture
3. Express.js server using GraphQL and REST API, web application.

1.3 Web API

A web application (or web app) is an internet technology term used to describe a computer software program that is run on a web server, unlike computer-based software programs that are stored locally on the Operating System (OS) of a device. Web applications are accessed by the user through a web browser with an active internet connection. These applications are programmed using a client–server modeled structure—the user ("client") is provided services through an off-site server that is hosted by a third-party. Examples of commonly-used, web applications, include: web-mail, online retail sales, online banking, and online auctions.[2]

REST

Representational state transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services. Web services that conform to the REST architectural style, called RESTful Web services, provide interoperability between computer systems on the Internet. RESTful Web services allow the requesting systems to access and manipulate textual representations of Web resources by using a uniform and predefined set of stateless operations.[3]

GraphQL

GraphQL is an open-source data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data. GraphQL was developed internally by Facebook. It provides an approach to developing web APIs, and has been compared and contrasted with REST and other web service architectures. It allows clients to define the structure of the data required, and the same structure of the data is returned from the server, therefore preventing excessively large amounts of data from being returned, but this has implications for how effective web caching of query results can be. It consists of a type system, query language and execution semantics, static validation, and type introspection.[4]

1.4 Motivation

Over the past decade, REST has become the standard for designing web APIs. It offers some great ideas, such as stateless servers and structured access to resources. However, REST APIs have shown to be too inflexible to keep up with the rapidly changing requirements of the clients that access them.

GraphQL follows the same set of constraints as REST APIs, but it organizes data into a graph using one interface. Objects are represented by nodes (defined using the GraphQL schema), and the relationship between nodes is represented by edges in the graph. Each object is then backed by a resolver that accesses the server's data.

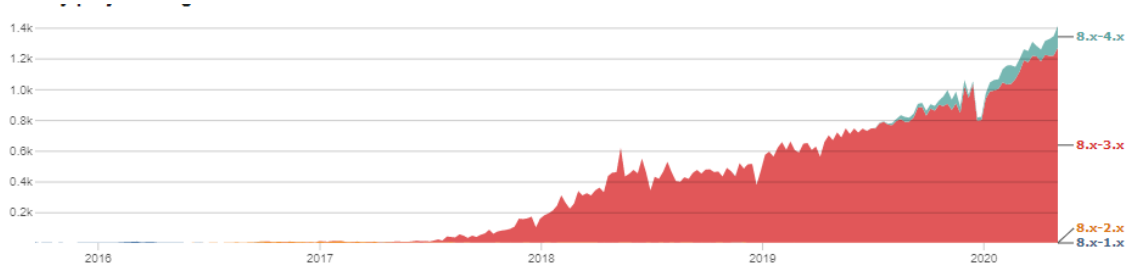

Figure 1.1: GraphQL weekly project usage chart [1]

Chapter 2

Background

This chapter introduces information about the context of this project.

2.1 Node.js

Node.js is an open-source, cross-platform, JavaScript runtime environment that executes JavaScript code outside of a web browser. Node.js lets developers use JavaScript to write command line tools and for server-side scripting—running scripts server-side to produce dynamic web page content before the page is sent to the user's web browser. Consequently, Node.js represents a "JavaScript everywhereen" paradigm, unifying web-application development around a single programming language, rather than different languages for server- and client-side scripts.[5]

2.2 REST

Representational state transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services. Web services that conform to the REST architectural style, called RESTful Web services. It was designed for distributed systems to address architectural properties such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability. [6] REST architectural style is defined by 6 principles/architectural constraints:

1. Client-server
2. Uniform interface
3. Stateless interactions
4. Cacheable
5. Layered system
6. Code on demand

2.3 GraphQL

GraphQL is a query language for your API, and a server-side runtime for executing queries by using a type system you define for your data. GraphQL isn't tied to any specific database or storage engine and is instead backed by your existing code and data.[7] A GraphQL service is created by defining types and fields on those types, then providing functions for each field on each type.

GraphQL schema example

```
type User {
  id: String
  email: String
  firsName: String
  lastName: String
}

type Query {
  me: User
}

type Mutation {
  createUser (input: UserInput!): User
}

input UserInput {
  email: String!
  firsName: String!
  lastName: String!
  password: String!
}
```

Listing 2.1: GraphQL schema example

GraphQL query and response example:

```
// Query
{
  me {
    id
    email
    firstName
    lastName
  }
}

// Response
{
  "me" {
    "id": "5de120339d402e485cc04211"
    "email": "john.doe@gmail.com"
    "firstName": "John"
    "lastName": "Doe"
  }
}
```

Listing 2.2: GraphQL query definition and query result example

2.4 Technology Stack

Technologies used in this project:

1. Express.js [8] - Node.js framework for processing http requests
2. MongoDB [9] - Document oriented database
3. Mongoose [10] - ORM for MongoDB
4. Google storage [11] - CDN for storing documents
5. Redis [12] - Library for creating JWT tokens. Used for authentication
6. Apollo client [13] - library for creating GraphQL requests and managing client state
7. Apollo server [14] - library for creating the GraphQL server.
8. Yarn [15] - Package manager for JavaScript

Express.js

Express.js is Node.js framework for building API. In this project Express.js used for creating web API with Graphql and REST endpoints.[6]

MongoDB

MongoDB is an open-source database management system (DBMS) that uses a document-oriented database model that supports various forms of data. It is one of the numerous nonrelational database technologies which arose in the mid-2000s under the NoSQL banner for use in big data applications and other processing jobs involving data that does not fit well in a rigid relational model. Instead of using tables and rows as in relational databases, MongoDB architecture is made up of collections and documents.[9]

JWT

JWT token is a library for decoding, verifying, and generating JWT. In this project, JWT used for authentication and authorization. The token is stored in cookies, and the client will be sent with every request. The express server will verify and decode token data. Based on the token data system will know what sources are allowed to the client. [12]

Apollo server

Apollo Server is an open-source, spec-compliant GraphQL server that's compatible with any GraphQL client, including Apollo Client. It's the best way to build a production-ready, self-documenting GraphQL API that can use data from any source. [14]

Apollo client

Apollo Client is a complete state management library for JavaScript apps. Simply write a GraphQL query, and Apollo Client will take care of requesting and caching your data, as well as updating your UI. Fetching data with Apollo Client guides you to structure your code in a predictable, declarative way consistent with modern React best practices. With Apollo, you can build high-quality features faster without the hassle of writing data plumbing boilerplate. [13]

Chapter 3

Design

In order to compare REST and GraphQL it will need to build a web application that using both GraphQL and REST APIs. The web server will provide functionality that modern web servers can do. This chapter will introduce the business and functional requirements.

3.1 Business requirements

For this project, I'm creating a web application for recruiting programmers. There will be three user types: programmer, company and admin. Programmers (in future I will call them members) may create a profile where they can add information about themselves and their technical knowledge. Companies may create a profile where they can create positions that they need to hire. After creating a profile, member can start the auction for 30 days, where registered companies may add the offer to this auction. In offer, they will let the members know at what position they want to hire them and what salary they can give. After 30 days of the auction, the member may contact the company with an integrated chat or by contact that the company provides. Administrators will have an administrator interface where they can control the status of companies, add new programming fields, languages, positions to the system.

3.2 Functional requirements

The system will have 3 roles:

1. Member
2. Company
3. Admin

Functional requirements for each role

Member

The system will allow to create member profile. Member registration will consist of two stages. In each stage, the user will complete information about himself. In the first stage member will complete information:

- About his work position (the member must choose the work position that will be provided by the system)
- Information about his technological knowledge (the member must choose technologies that will be provided by the system)
- Phone number, email, first name, surname, and password
- Accept the license agreement

After completing the first stage, the server sent the member an email, with which the member can confirm his email address. After confirming the email address, the member will complete the information:

- Current place of work, working position, date of start, employment form (contract, employee), place of residence, employment time (full-time, part-time), date of birth (optional)
- Fill out a special form. The questionnaire will have six questions; for some questions, the user can choose from the suggested answers.
- Fill out a special form about member education (University/school name, education start/end, language knowledge)

After registration, the member will have access to the profile. In the profile, the member will be able to change the avatar, upload examples of his code, write a text about himself. Also, the member will be able to update the information from the previous steps except for the email address and password. After registration, the member will be able to launch an auction, which will last 30 days. After creating, the auction will appear in the search list in the profile of the company. If the company makes an offer to the member, the member will receive a notification. The member will be able to log out and log in using the email address and password

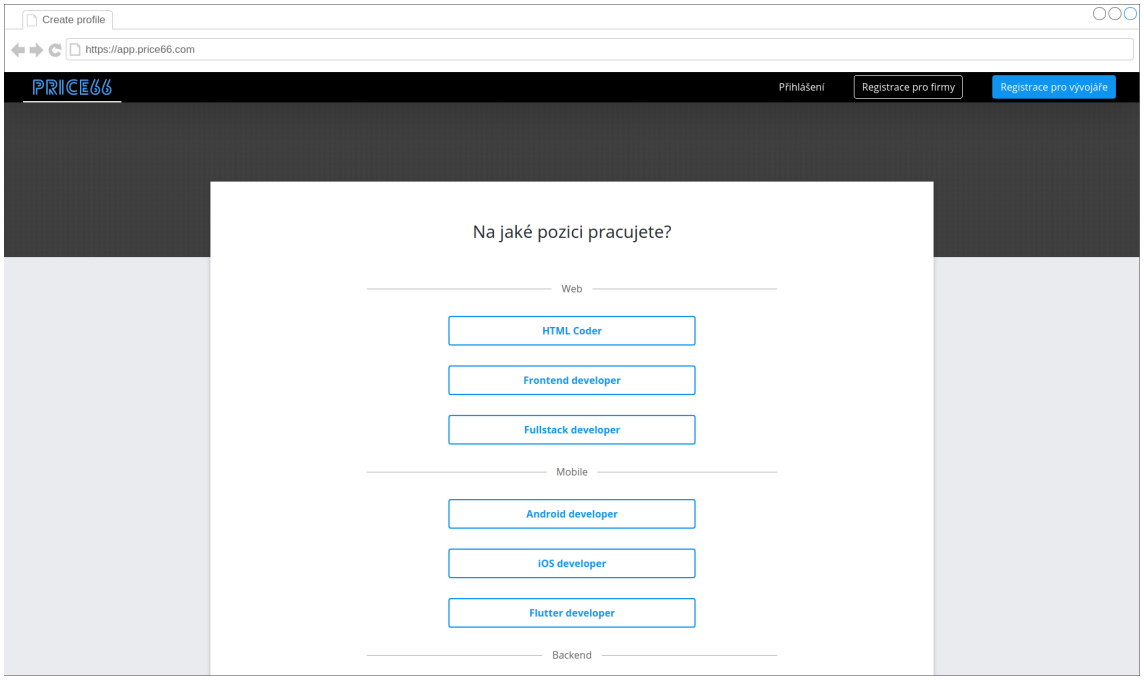

Figure 3.1: UI select position example

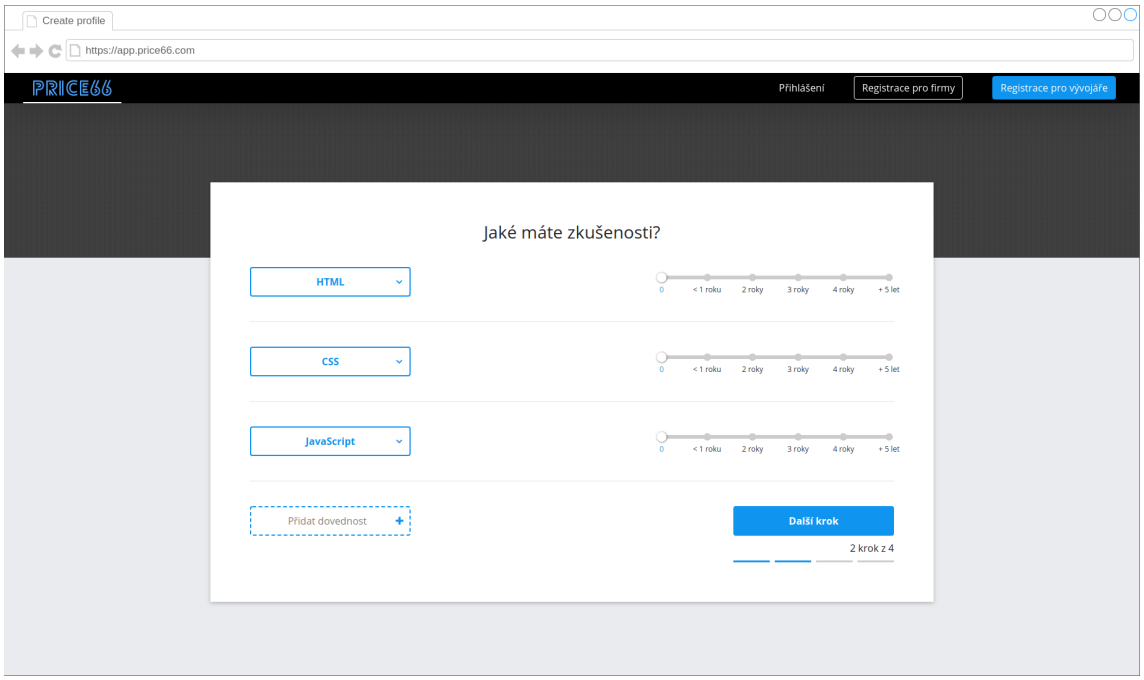

Figure 3.2: UI select skills example

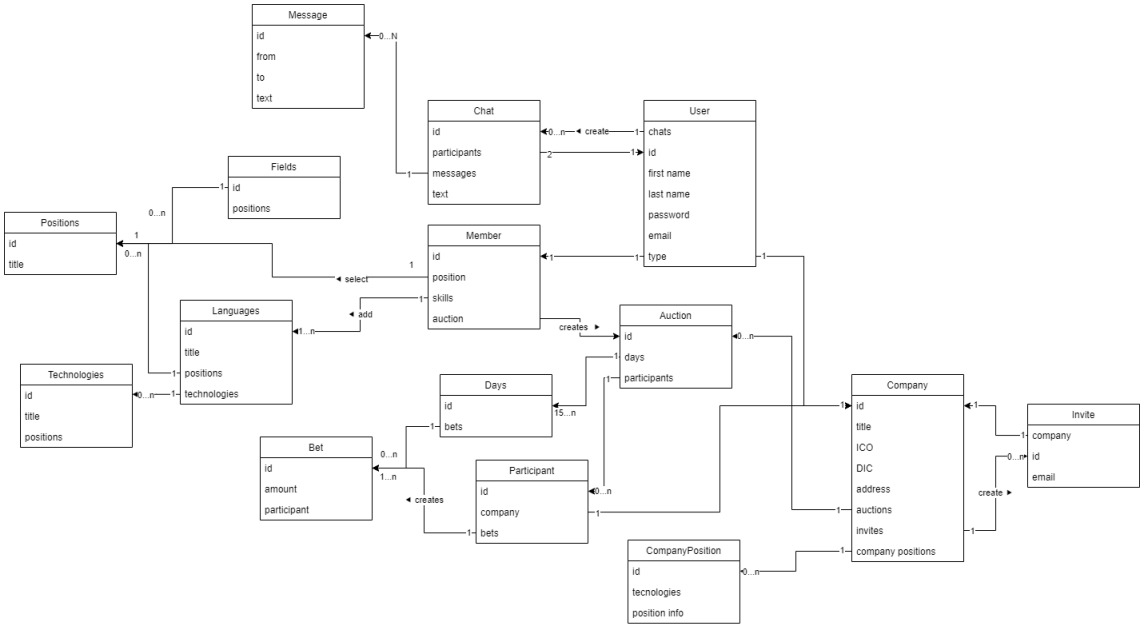

Figure 3.3: System UML diagram

Company

The system will allow to create company profile. To create a company profile, company CEO will complete information:

- Email address and password
- Company name, full time employee, contractor (optionally), company web page, company LinkedIn profile.

After the registration, the server sent an email, with which the company CEO can confirm his email address. After confirming the email address, the company will have access to the company profile. In the company profile, the user can create a job position. A job position will be created with the following information:

- Position title
- Company description
- Company team description
- Company project description
- Technologies that programmer will use on the project
- Offer provider contact (name, company position, email address, phone number, web page link, LinkedIn profile link)

In the company profile, the user will be able to browse active auctions. The company will be able to participate in member auction. To participate in the auction, the user will need to click on the auction page and complete the following information.

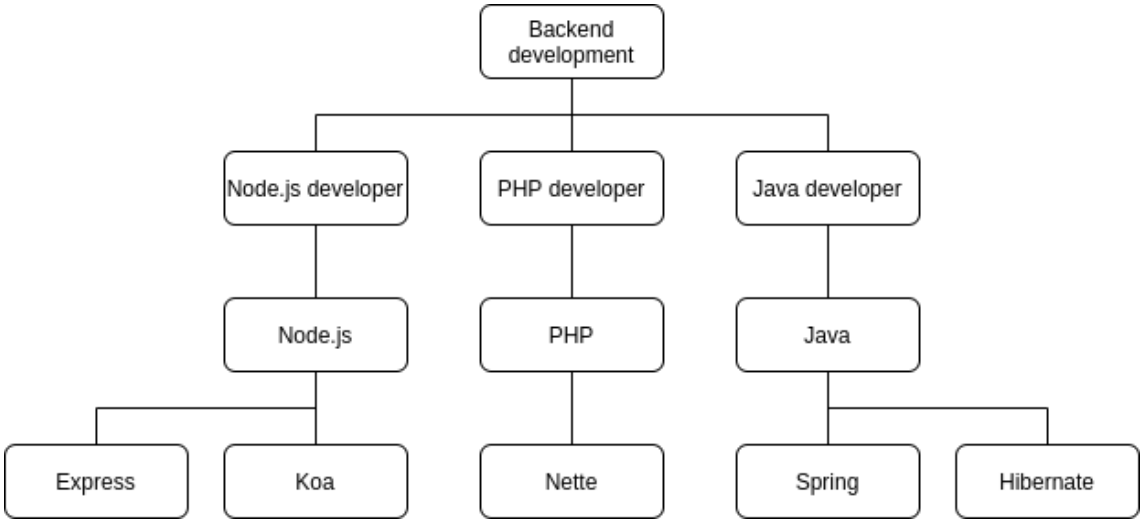

Figure 3.4: Programming field example

- Select position from created positions
- Offer salary
- Employment type(contract, full-time employee)
- Message for member

After creating an offer, the member will get the notification email. In the company profile, the user will be able to invite new users to the company profile. The company will have a page where will be a list of all invited users. In the company profile, the user will be able to update any company information except company ID.

Admin

The system will allow using the admin interface. To get access to the admin interface, the user must be logged in to any profile (company profile or member profile) with an email from the white list. The admin interface will allow creating tree structures from programming fields, positions, programming languages, and technologies. The programming filed will be a root of the tree. Programming fields may have 0-N references to the working positions. Working positions will have reference to the single programming field and may have 0-N references to the programming languages. Programming languages will have reference to the single working position and may have 0-N references to the technologies. Technologies will have a single reference to programming language.

3.3 System architecture

System will have client-server architecture. Client Server Architecture is a computing model in which the server hosts, delivers and manages most of the resources and services to be consumed by the client. This type of architecture has one or more client computers connected to a central server over a network or internet connection. This system shares computing resources. Client/server architecture is also known as a networking computing

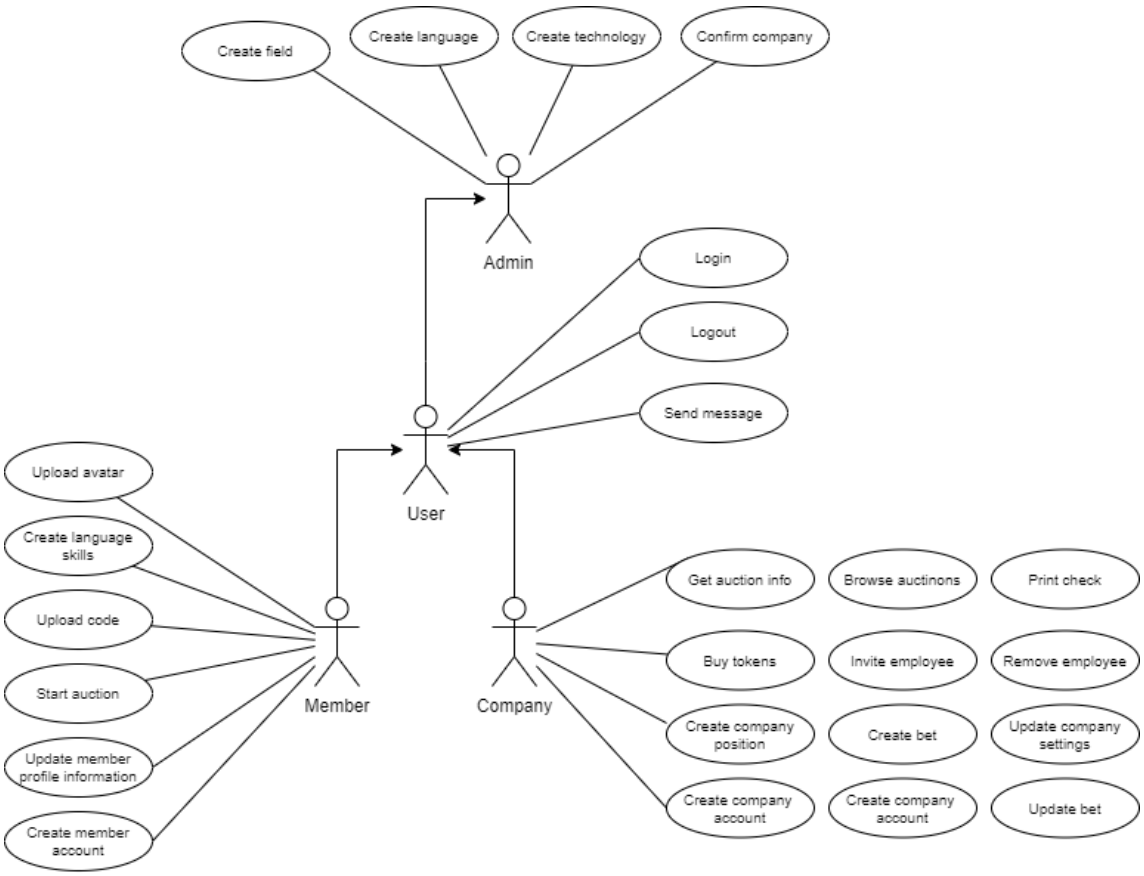

Figure 3.5: Use case model of the application

model or client/server network because all the requests and services are delivered over a network.[16] The server is Express.js server, which using GraphQL and REST API. The client is web application that running in the browser. The client and server will communicate with HTTP requests.

Chapter 4

Implementation

4.1 API setup

This section describes how to start with REST and GraphQL on the Express.js server and compare both ways.

GraphQL

To start with GraphQL on Express.js server, it will need to:

1. Install packages express and apollo-server-express.
2. Create GraphQL schema.
3. Add resolvers to the GraphQL schema.
4. Create GraphQL server.
5. Install GraphQL server to the Express.js.

Packages express and apollo-server-express can be installed with npm or yarn. After we can start with GraphQL on the server. Firstly it needs to define GraphQL schema. The basic components of a GraphQL schema are object types, which represent a kind of object that can be fetched from the server, and what fields it has[17]. In the GraphQL schema language, data might be represented like this:

```
type User {
  id: String
  email: String!
  firstName: String
  lastName: String
}
```

Listing 4.1: GraphQL schema Object type example

The language is pretty readable, but let's go over it so that we can have a shared vocabulary:

- "User" is a GraphQL Object Type, meaning it's a type with some fields. Most of the types in your schema will be object types.

- "firstName" and "lastName" are fields on the "User" type.
- String is one of the built-in scalar types - these are types that resolve to a single scalar object, and can't have sub-selections in the query. We'll go over scalar types more later.
- String! means that the field is non-nullable, meaning that the GraphQL service promises to always give you a value when you query this field. In the type language, we'll represent those with an exclamation mark.

Most types in the schema will just be normal object types, but there are two types that are special within a schema this is Query and Mutation types. These types are the same as a regular object type, but they are special because they define the entry point of every GraphQL query. Example of creating GraphQL server:

```
// 1. Create express server
const server = express()

// 2. Define schema
const types = gql`
  type User {
    id: String
    email: String
    firstName: String
    lastName: String
  }

  type Query {
    me: User
  }
`

// 3. Create executable schema for Apollo GraphQL server
const schema = makeExecutableSchema({
  typeDefs: [types],
  resolvers: {
    Query: {
      me: async (_, __, { req }) => {...},
    },
  }
});

// 4. Create apollo server
const apolloServer = new ApolloServer({schema})

// 5. Install Apollo GraphQL server to express server
apolloServer.applyMiddleware({app: server})

// 6. Start express server
server.listen({port: 3000})

// Apollo GraphQL server will be at http://localhost:3000/graphql
```

Listing 4.2: Add GraphQL server to the Express.js server example

REST

To start with REST on Express.js server, it needs to install express package and create REST resources. In REST, primary data representation is called Resource.

A resource can be a singleton or a collection. For example, "users" is a collection resource and "user" is a singleton resource. We can identify "users" collection resource using the URI "/users". We can identify a single "user" resource using the URI "/user/userId".[18]. In the Express.js server, usually, REST resources are called "controller". Controller implementation example:

```
// 1. Create express server
const server = express()

// 2. Declare controller
server.get("/me", (req, res) => {...})

server.listen({ port: 3000 })

// 3. Controller will be at http://localhost:3000/me
```

Listing 4.3: Express.js REST controller example

The name of REST controller should be self-descriptive. For example controller at URI "/user/userId" return user, "/users" returns a users collection and etc. In web development name "me" is used for controller that return authorized user.

REST and GraphQL comparison in terms of setting up the Express.js server

Implementation speed

In REST, it needs just defining controllers. REST does not need to define what data types controller return or which inputs does it have. Creating REST controllers is a far faster way to start with API than GraphQL. In GraphQL, it needs to create the schema. Inside schema, it must be defined what types the system may operate. Then it needs to create resolvers for schema and apply it to the server. This process is taking more time, and it needs to install more packages for the project.

Code Documentation

In REST, after defining controllers, it needs to use external services, for example, Swagger to create documentation to the API. In services like Swagger, it needs to define URI to the controller, what data types do controller accepts, and what data types controller returns. In GraphQL, we are getting documentation about the API after defining schema. GraphQL provides a tool for the API – GraphiQL. GraphiQL is a graphical interface where we can inspect what queries/mutations/subscriptions the API supports. For every query/mutation/subscription, we can see what inputs does it have and what data does it return. In this case, we are using apollo-graphql, so we do not need to install graphiql. GraphQL server provides a playground that is the same tool as GraphiQL, but we can use it inside the browser. After starting the server, the playground is available at:

```
http://localhost:<port>/graphql
```

Versioning

Often when consuming third-party REST APIs, we see stuff like v1, v2, v3 etc. which simply indicate the version of the REST API we are using. This leads to code redundancy and less maintainable code. With GraphQL, there is no need for versioning as we can easily add new fields and types to our GraphQL API without impacting existing queries. Also, we can easily mark fields as deprecated and the fields will be excluded from the response gotten from the server.

	Speed	**Code docs**	**Versioning**
REST	No need to install packages	non self-documenting	Using of different URIs for new versions
GraphQL	Install packages Define schema Create resolvers Create GraphQL server Add to express.js server	Self-documenting	No versioning just update schema

Table 4.1: Summary of initializing the REST and GraphQL on the express.js server

4.2 HTTP GET request

In the last chapter, we started express.js server with GraphQL and REST. In this chapter, we will look at how to fetch data from server. In other words - making of HTTP GET requests to the express.js server from the client application and how to proceed GET request on the server.

GET request

In REST, HTTP GET requests are requests that are fetching data from the server without modifying any sources. In GraphQL GET request equivalent is Query type.

GraphQL

Server

Firstly it needs to define the query in GraphQL schema.

```
type User {
  id: String
  email: String
  firstName: String
  lastName: String
}

type Query {
  me: User
}
```

Listing 4.4: GraphQL schema query example

The schema above has the query “me”, which returns an object of type “User”. After it will need to define resolver for this query. Resolver implementation example:

```
const resolvers = {
  Query: {
    me: async (_, __, { req }) => {
      return await models.User.findOne({id: req.user.id})
    }
  }
}
```

Listing 4.5: GraphQL query resolver example

Read more about GraphQL resolvers in the "REST controllers and GraphQL resolvers" section. That's all you need to do to create a query in the GraphQL server.

Client

To create GraphQL requests from the client application, it needs to create an Apollo Client. To use Apollo, it needs to install packages apollo-client, apollo-cache-inmemory, and apollo-link-http the the client application. Apollo Client example:

```
const httpLink = createHttpLink({
  uri: `${CONFIG.url.api}/graphql`,
  credentials: "include"
})

const client = ApolloClient({
  cache: new InMemoryCache(),
  link: from([ httpLink ])
})
```

Listing 4.6: Create GraphQL client example

More options can be passed to the client, for example, authorization header, cookies, credentials, etc. These options will work for every GraphQL request that we make. After creating the client, we can do GraphQL queries like so:

```
const fetchData = async () => {
  const { data } = await client.query({
    query: gql`
      {
        me {
          id
          email
          firstName
          lastName
        }
      }
    `
  })

  return data.me
}

const data = await fetchData()
```

Listing 4.7: GraphQL client call query example

To create a GraphQL query, it needs to call function "query" on the client, and as a first parameter, it takes an object, which is an option for the query. For writing the query, the client using graphql-tag; this is the label gql before query that assists in the writing of GraphQL queries. Because GraphQL using schema definition language, the query result

is predictable, and it may autocomplete and suggest options for the query, for example, body parameters or query response variables. GraphQL returns only data that that is defined in the query. For example, in the query above, there are four variables:

- id
- email
- firstName
- lastName

If the client needs the only email, all variables except email can be removed, and the server will return the only email. After returning of queried data, Apollo will automatically save request data to cache. Next time when the client will need to view data, firstly, Apollo will try to find data in the cache. If Apollo finds data in the cache, it will return data from the cache, and the request will not be proceed. Read more about Apollo cache in "Data caching on the client application" section. Query variables can be sent as a JSON-encoded string in an additional query parameter called variables. If the query contains several named operations, an "operationName" query parameter can be used to control which one should be executed.

```
// GraphQL schema
type Query {
  user(id: String!): User // id is non-nullable
}

// Client query
const { data } = await client.query({
  query: gql`
    query GET_USER($id: String!) {
          user(id: $id) {
                email
      }
    }`,
  variables: {
    id: "5de120339d402e485cc04211"
  }
})
```

Listing 4.8: GraphQL query variables example

id: String! means that the field is non-nullable, meaning that the GraphQL service promises to always give you a value when you query this field.

Important to mention that the client application may create GraphQL request with native JavaScript "fetch" function or any function from libraries that allows creating HTTP GET request. It needs to pass the GraphQL query as a query parameter inside request URI. Example:

```
await fetch("http://localhost:4000/graphql?query={ me { id } }")
```

REST

Server

In REST it needs to create controller on the express.js server. To create REST controller it needs to call "get" function on the express.js server object. This function accepts two

positional arguments. First one is the URI, where the client make request, second is the handler for this controller. Read more about REST controllers in "REST controllers and GraphQL resolvers" section. Example of creating controller:

```
const server = express()

server.get("/me", (req, res) => {
  res.send(await models.User.findOne({id: req.user.id}))
})
```

Listing 4.9: REST get controller example

That is all that need to do to create the REST controller on the express.js server.

Client

To create the GET request to the REST controller, in the client application, there is a native JavaScript "fetch" function that is supported by most browsers. Also, we can install a package that allows making GET requests, for example, "axious". The pros of using such packages are that they automatically provide polyfill[19] for functionality that old browsers do not support. If we need to know what functionality does browser supports, we can use services such as `https://caniuse.com`. Here client will use the fetch function. Example of using fetch function on the client application:

```
const fetchData = async () => {
  return await fetch("http://localohost:3000/me")
}

const data = await fetchData()
```

Listing 4.10: REST GET request example

The first parameter of the function is URI to the server controller. The second parameter is query options, for example, headers that will be sent, request type, request body, etc. In REST, if we need to pass parameters to query, we can pass it to request URI.

```
const fetchData = async () => {
  return await fetch("http://localohost:3000/user?id=5
      de120339d402e485cc04211")
}

const data = await fetchData()
```

Listing 4.11: REST GET request with query variables example

REST and GraphQL comparison in terms of creating fetch data functions on the server and client application

Speed

To create a GET query to the REST controller, the client using the fetch function (or equivalent function from installed packages). As a first parameter, it will always require URI to the server controller, and optionally we can pass request options.

To create a GraphQL query, you can also fetch function or use GraphQL client. To use the client, it needs to set up an Apollo client. Then with this client, the client can call the query function and pass the GraphQL query as a parameter.

Data fetching

In REST, when we make the query, we do not know what data will be returned, unless we have got documentation. Also, we get all the data that the server returns. We cannot prevent fetching of unnecessary data.

In GraphQL, we always know what data will be returned from the server. And we can manipulate with query result. If we don't need data, we just do not query on it.

Query variables

In the REST, we can pass parameters inside URI as a query parameter. Cons of this approach are that on the server, this parameter will always seem like a string if we want to pass an integer, then on the server, we must parse integer query parameters. It's not secured way to transfer data to the server throw the internet and also query parameters has limited data length.

In GraphQL, we are passing variables in the request body. Also, we know what data types we are passing to request body. If we try to pass the wrong data type, for example, string instead of an integer, then GraphQL will throw an error.

Security

In some cases, we need to authorize the request to grant access to some server sources. For example, only authorized users may fetch profile data. To secure REST routes, we can do it in two ways. The first way is to control request headers (for example, cookies or authorization header) in the controller. Pros of this way that this is a fast way to secure the controller. Cons are that we will have duplicating code, and we do not have any documentation about what controllers are secured and what not. The second way is to create middleware. Middleware is a function that will always be called before proceeding request. In this function, we can check request headers, for example, cookies and, based on it, allow/forbid access to the controller. Pros of this way are that we can use one middleware for several controllers, and we do not need to duplicate our code. Cons are that if we have several middlewares and we need to apply it to different controllers, we will not have documentation on what routes are secure and what not.

In GraphQL, we can also use the manual way to check requests, we can also apply middleware for resolvers, but we can also use schema directives. A directive can be attached to a field or fragment inclusion, and can affect execution of the query in any way the server desires.[20] We can use schema directives not only for security reason but also, for example, to format or filter data. Read more about middelwares and directives in "Security middlewares and directives" section

	Speed	**Data fetching**	**Query variables**	**Security**
REST	Create GET controller	No control over fetching data	Pass variables inside URI as query parameters	Controllers control Middlwares
GraphQL	Update schema Create resolvers for new fields	Fetch only desired data	Pass variables as query options	Resolvers control Middlwares Directives

Table 4.2: Summary of creating GET queries for REST and GraphQL on the express.js server and client

4.3 HTTP POST, DELETE, PUT, UPDATE request

HTTP requests that are modifying server sources have a post, delete, put, or update method. REST using the same request types. GraphQL equivalent for all these request types is the Mutation type.

GraphQL

Server

When it needs to modify data on the server, in GraphQL, we are using Mutations. This is the unique GraphQL type. In GraphQL, mutate means to modify the data source (create, update, delete). Example of defining GraphQL mutation in the schema:

```
type User {
  id: String
  email: String
  firstName: String
  lastName: String
}

type Mutation {
  createUser(user: UserInput!): User
}

input UserInput {
  email: String
  firsName: String
  lastName: String
  password: String
}
```

Listing 4.12: GraphQL mutation example

In the example above, schema has mutation “createUser” that has one input “user” of type “UserInput” and this mutation returns the object of “User” type. ! sign after “UserInput” means that variable “input” cannot be nullable. To reduce repeating code and make GraphQL schema simpler, GraphQL has a special type called input. Instead of passing parameters separately, we can use inputs to pass an object to the resolver. It makes schema clearer and also simplifies work with parameters in the resolver. We can also use inputs to get rid of duplicate code, for example:

```
type Mutation {
  createUser(user: UserInput!): User
  updateUser(user: UserInput!): Boolean
```

```
}

input UserInput {
  email: String
  firsName: String
  lastName: String
  password: String
}
```

Listing 4.13: GraphQL input example

In the code above we are using input "UserInput" for "updateUser" and "createUser" mutations. After defining new mutations to the schema, we can create resolvers for it.

```
const resolvers = {
  Mutation: {
    createUser: async (_, {user}) => {
      const checkEmail = await models.User.findOne({email: user.email})

      if (checkEmail) {
        return new AlreadyExists()
      }

      const newUser = {
        ...user,
        password: hash(user.password)
      }

      return await models.User.create(newUser)
    },
  },
}
```

Listing 4.14: GraphQL mutation resolver example

In the example above, we create a resolver for "createUser" mutation. This resolver will check if user email is free. If it's free, then the server will create a new user. If it's not, then the server will throw "AlreadyExists" error. "AlreadyExists" is the custom error, GraphQL allows to create custom errors. That is all we need to do to create GraphQL mutation on the server.

Client

To make GraphQL mutations, we are using client the same way as we used it for making queries.

```
const createUser = async (user) => {
  const {data} = await client.mutation({
    query: gql`
      mutation CREATE_USER($user: UserInput!) {
        createUser(user: $user) {
          id
          email
          firstName
          lastName
        }
      }`,
      variables: {
        user: { ...user }
      }
```

```
  })
}
```

Listing 4.15: GraphQL client mutation example

Cache

In most cases, after a mutation, you need to update the interface and display the result of the mutation. For example, the system allows to admins creating of programming languages. After creating the programming language we wants to show created language inside the language table. There are two ways how can we do this. The first way - we will make a request to the server to save the language, and then we will make a second request to re-get all the languages that are recorded in the database.The second way is that we will make a request to the server to save the language, and in response data of this request, we will get the language ID that we created on the server. Then we modify the cache by combining the data from the client and data from the server (ID). Because we have already received a list of languages, we do not need to get all the languages again. It's enough to modify the cache in which these languages are stored. Apollo GraphQL provides a tool for interacting with cached data. To start with the cache update, we need to pass "update" property to the mutation options. "update" is the function that as the first parameter accepts cache, and the second parameter is data that we get from the server. Example of using Apollo to modify cached data:

```
createLanguage(language) {
  this.$apollo.mutate({
    mutation: removeLanguageMutation,
    variables: {
      language: { ...language }
    },
    update: (cache, {data: { createLanguage }}) => {

      // Read data from the cache. As the parameter, we are passing the
          query that we already make to the server
      const { languages } = cache.readQuery({
        query: languagesQuery
      })

      // Create new data by pushing new language to the array from the
          cache
      const newData = langugages.push(createLanguage)

      // Write new data to the cache
      cache.writeQuery({
        query: languagesQuery,
        data: { languages: newData }
      })
    }
  })
}
```

Listing 4.16: GraphQL client mutation with cache update example

This approach dramatically speed up the execution of queries that don't rely on real-time data. Read more about caching data in GraphQL in "Data caching on the client application" section.

REST

Server

Creating of REST endpoints for POST requests is the almost same as defining REST endpoint for GET requests. It need just call post function on the express server. As the first parameter it accepts URI, where REST endpoint will listen. As the second parameter it accepts function that handles POST request. Handle function accepting two parameters, request and response. Request is an object that contains information about request. Response is an object for creating response to the client. Read more about REST controllers at "REST controllers and GraphQL resolvers" section.

```
const server = express()

server.post("/createUser", (req, res) => {
  const user = req.body.user

  const checkEmail = await models.User.findOne({email: user.email})

  if (checkEmail) {
    return res.status(400).json({ error: 'User already exists' })
  }

  const newUser = {
    ...user,
    password: hash(user.password)
  }

  res.json(await models.User.create(newUser))
})
```

Listing 4.17: REST POST controller example

In the example above, we create a REST controller. This controller will check if user email is free. If it's free, then the server will create a new user. If it is not, then the server sends request status 400 with the error message.

Client

To create POST request to the REST controller, similarly as in GET requests, the client application can use fetch function or any libraries that allows to create HTTP POST requests. Example of creating POST request to the REST endpoint:

```
const createUser = (user) => {
  const data = await fetch('http://localhost:3000/createUser', {
    headers: {
      ....
    },
    body: {
      user: {...user}
    }
  })
}
```

Listing 4.18: GraphQL client POST request example

REST and GraphQL comparison in terms of modifying data on the server

Speed

Creating of GraphQL mutations and REST POST controllers is the same as creating of GraphQL queries and REST GET controllers. In GraphQL, it needs to update the schema and implement resolvers for new fields. In the REST, it needs call "post" function on the express.js server object, define URI and controller handler.

POST request body

Passing the data to request the body is the important step of each request that is modifying server sources. The client should pass correct data types and transfer them to the server in a secure way. Both REST and GraphQL requests may run over HTTPS, so the request body is secured. However, the client can access the REST controller with any data in the body, so the server always needs to control received data. In GraphQL, we will never access the resolver if the client passes the wrong data to the mutation variables. GraphQL will throw an error if the client tries to pass the wrong data. So there is no need to control data types inside resolver because the server ensures that the resolver has correct data. Also GraphQL

Cached data

Both GraphQL and REST requests are cacheable but GraphQL provides efficient tool to manage, cached data. This helps to speed up client application and reduce load to the server.

Error handling

Both GraphQL and REST allows handling request errors. However, GraphQL allows us to create custom errors, with their options and custom types. Creating custom errors helps the client to present error information more efficient way than in the REST.

	Speed	**POST request body**	**Cached data**	**Error handling**
REST	Create POST controller	Server should controls data types	Cacheable, but no specific way how to update cached data	Send request status and error message
GraphQL	Update schema Create resolvers for new fields	Data in the body well formed and has correct types	Cacheable Provides tools for managing cached data	Custom errors

Table 4.3: Summary of creating of modifying requests for REST and GraphQL on the express.js server and client

4.4 REST controllers and GraphQL resolvers

In this section, I will describe in more details, how to define the GraphQL and REST endpoints.

GraphQL resolver

In order to respond to requests, a schema needs to have resolvers for all fields. Resolvers are per field functions that are given a parent object, arguments, and the execution context, and are responsible for returning a result for that field. Resolvers cannot be included in the GraphQL schema language, so they must be added separately. Every resolver in a GraphQL.js schema accepts four positional arguments:[21]

```
const resolvers = {
  Mutation: {
    async createLanguage(obj, args, context, info) {
      ...
    }
  }
}
```

Listing 4.19: GraphQL mutation resolver example

- obj - The object that contains the result returned from the resolver on the parent field, or, in the case of a top-level Query field, the rootValue passed from the server configuration. This argument enables the nested nature of GraphQL queries.
- args - An object with the arguments passed into the field in the query. For example, if the field was called with

  ```
  createLanguage(title: "JavaScript")
  ```

  the args object would be

  ```
  { "title": "JavaScript" }
  ```

- context - This is an object shared by all resolvers in a particular query, and is used to contain per-request state, including authentication information, dataloader instances, and anything else that should be taken into account when resolving the query. If you're using Apollo Server.
- info - This argument should only be used in advanced cases, but it contains information about the execution state of the query, including the field name, path to the field from the root, and more

REST controller

The REST controller is the endpoint that is listening on a specific URI and handle HTTP requests. To define the REST controller on the express.js server its needs to call post/delete/put/get function (based on request type) on the server object. post/delete/put/get function accepts two positional arguments.

```
const server = express()

server.get("/", (req, res) => {
  ...
})

server.post("/", (req, res) => {
  ...
})
```

Listing 4.20: REST controllers example

- URI - where the REST endpoint listening
- Controller handler function that handles the request

The second parameter function accepts two two positional arguments.

- req - this is an object that contains the information about request such as headers, cookies, body etc.
- res - this is the function that allows us to send response to the client.

The REST controllers are listening on the different URIs. All GraphQL requests are coming to the single URI "/graphql". This approach has its cons and pros. Pros of the single URIs is that the client has one URI where it sends all the request. Also, on the server, there is no need to think about securing different URIs. However, the single URI make limitations. For example, if the server needs to create API for a third party system, it should use REST for this purpose.

Another disadvantage of GraphQL resolvers is that the GraphQL request cannot be redirected. Each query field in the schema has defined results, and there is no nothing like "Redirect" type or redirect status in the GraphQL schema. For these purposes also better to use REST controllers.

4.5 Security middlewares and directives on the Express.js server

In the previous chapters, it was mentioned that the server uses middleware and GraphQL directive to authorize the client. In this chapter, I will describe in more detail the sequence of processing the request on the server.

Middleware

Middleware is a function that is called before GraphQL resolver or REST controller. middleware accepts 3 parameters:

1. req - this is an object that contains the information about request such as headers, cookies, body etc.
2. res - this is the function that allows us to send response to the client. For example

```
res.send("Response") // send plain text
res.redirect("http://www.fel.cvut.cz/en/") // send redirect to URI
```

Listing 4.21: REST response example

3. next - this is the function that allows us to pass request to the next stage

Create middleware example:

```
const server = express()

const newMiddleware = (req, res, next) => {
  ... // middleware body
  next()
```

```
}

server.use(newMiddleware, "/graphql")
```

Listing 4.22: Express.js middleware example

By calling use function on the express server, we are passing "newMiddleware" function as the first parameter, and at which URI should middleware be called, if the second parameter not passed, it will be called before every request.

Directive

A directive can be attached to a field or fragment inclusion, and can affect execution of the query in any way the server desires.[20]

```
directive @auth on FIELD_DEFINITION
directive @auctionMemberName on FIELD_DEFINITION
directive @confirmed on FIELD_DEFINITION
directive @company on FIELD_DEFINITION

type Query {
  me: User @auth
  auction(auction: String): Auction @auctionMemberName @confirmed @company
      @auth
}
```

Listing 4.23: GraphQL schema directives example

In the example above, we attached the "@auth" directive to the "me" query. If we are attaching several directives as we do for the "auction" query, then directives will be executed from right to left. After defining the directive, it needs to create resolver for it. "@auth" directive resolver implementation example.

```
class Auth extends SchemaDirectiveVisitor {
  visitFieldDefinition(field) {
    const { resolve = defaultFieldResolver } = field;
    field.resolve = async function(
      source,
      { format, ...otherArgs },
      context,
      info
    ) {
      if (context.req.authorized) {
        return await resolve.call(this, source, otherArgs, context, info);
      }
      else {
        return new Unauthorized();
      }
    };
  }
}

const schemaDirectives = {
  auth: Auth,
}

const schema = makeExecutableSchema({
  ...
  schemaDirectives
```

```
})
```

Listing 4.24: GraphQL directive resolver example

For the first time, you are looking at the directive. Its implementation might not look straightforward. It is true that directives still does not have clear syntax, unlike middleware. The central part of the example above is the body of the asynchronous anonymous function. All we do inside this function is controlling the request context if it has authorized property that we record in the middleware. If it has this property, then the directive will call resolve request, if not then it will throw an unauthorized error. Similarly, you can implement any logic for other directives.

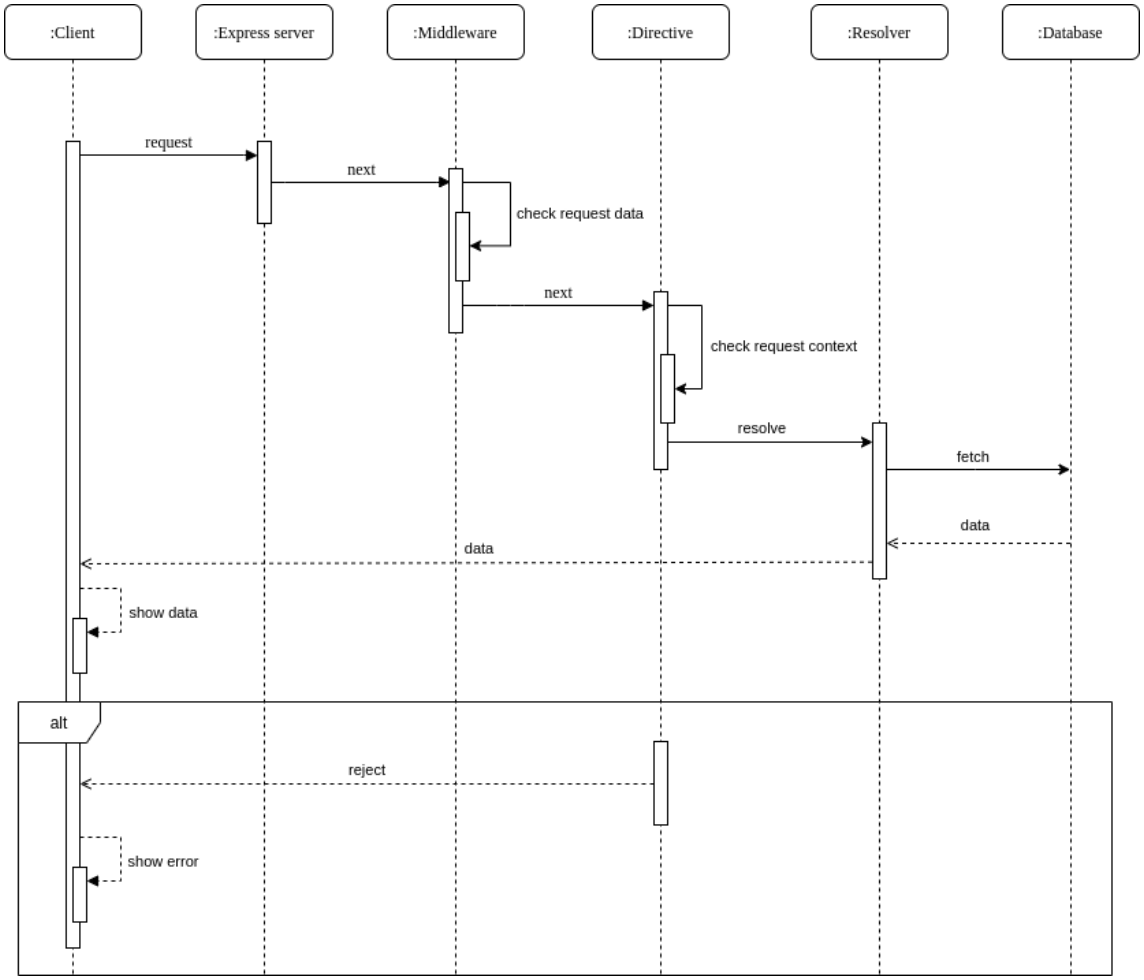

Figure 4.1: GraphQL request sequence diagram

The diagram above describes the sequence that the GraphQL request passes on the server.

For GraphQL requests, the server uses both middlewares and directives. In the middleware server checking request cookies and if it has authorization token, then it will verify token and write token data to the request context. This middleware applied to all requests coming to the server. Then inside directives, we checking request context, and based on these parameters, the server resolves or reject GraphQL request. Code example:

```
const authorization = (req, res, next) => {
    const token = req.cookies[ "token-name" ]
    if (token) {
        try {
            const data = verifyToken(token)
            req.authorized = true
            req.user = data.user
        }
        catch (e) {
            req.authorized = false
        }
        next()
    }
    else {
```

```
        req.authorized = false
        next()
    }
}
```

Listing 4.25: Express.js server authorization middleware example

In the example above, we create middleware “authorization” that checking request cookies. If it has a cookie with a name token, then try to verify token and modify the request context. Notice that this middleware does not reject the request if it does not contain cookies, its just modifying context. Then the request will proceed to GraphQL. Firstly it will proceed to directive. GraphQL directive will check context and based on the context it will reject/resolver the request. Directive example above

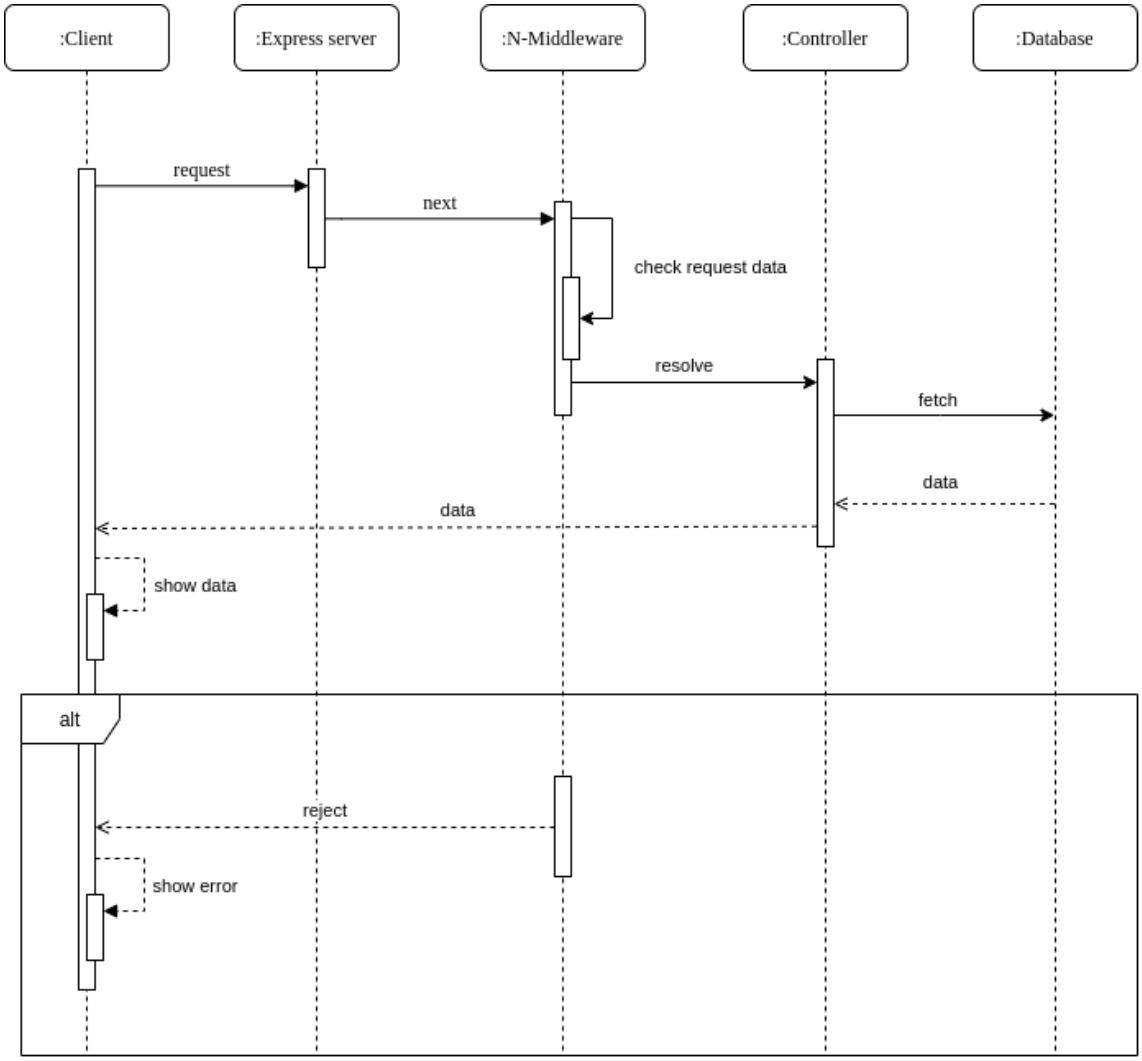

Figure 4.2: REST request sequence diagram

The diagram above describes the sequence that the REST request passes on the server. Because there is no directives in the REST we should use several middlewares to secure controllers. N-middleware means that request may pass several middlewares. Example:

```
const server = express()

server.use("*", authorization),
server.use("/user/*", userMiddleware)
server.use("/user/auction/*", userCompletedProfileMiddleware)
...
```

Listing 4.26: Express.js server middleware use example

In the example above, the server using several middlewares for different URIs. "*" means that "authorization" middleware applied to all requests, "/user/*" means that "userMiddleware" applied to all requests to "/user" URI and any source after it, etc.

GraphQL and REST comparison

Let's compare both ways of securing server sources on the express.js server.

```
const server = express()

server.use("/authorized", authorization),

server.use("/authorized/user/auction/*", memberMiddleware)
server.use("/authorized/company/auctions/*", companyMiddleware)
...
```

Listing 4.27: Several middlewares example

This is how we can secure REST controllers with the middlewares. The disadvantage of this approach is that we always need to define the URI where we need the middlewares to be used. If the server has a complex structure of source security, the syntax will become unclear. For example, we do not know what routes are listing at "/authorized/user/auction/*". It is important to mention that the middleware ways of securing REST controllers, works only on the Node.js servers. Other web servers like Java also using similar ways of securing REST controllers, but there is no standardized way of how we should secure API. However GraphQL provides schema directives. In GraphQL schema, we can see each field, and with what directives this field is protected.

```
directive @member
directive @company
directive @auth

type Query {
  myAuction: Auction @member @auth
  auctions: [Auction] @company @auth
}

type Mutation {
  createAuction(auction: MutationInput): Auction @member @auth
  createBet(bet: BetInput): Bet @company @auth
}
```

Listing 4.28: Directives example

In the example above, there is GraphQL schema, which is using several directives to different fields.

4.6 Data caching on the client application

In the previous chapters, it was mentioned that the client application uses Apollo GraphQL to manage cache memory during requests. In this chapter, I will describe in more detail how Apollo GraphQL works with cached data.

Apollo

Because the GraphQL is the type system, it has a loot of tools that helping us in development. One of this tools is Apollo. If you are using GraphQL, in most cases you will also use Apollo framework. Apollo is the industry-standard GraphQL implementation, providing the data graph layer that connects modern apps to the cloud.[22] In the client applications, Apollo helps us to create GraphQL requests, interacting with cached data and application state managment.

Mutation

GraphQL mutation is a special type that is used to modify the source on the server. Mutation schema definition example:

```
type Language{
  id: String
  title: String
}

type Mutation {
  createLanguage(title: String!): Language
}
```

Listing 4.29: GraphQL mutation schema example

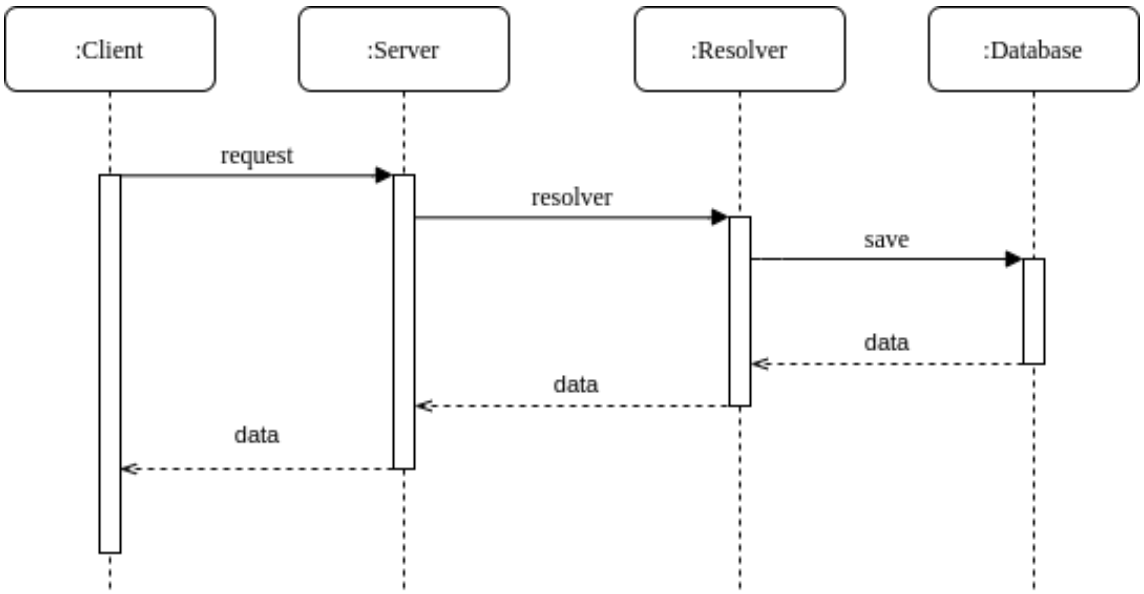

Figure 4.3: Simple GraphQL mutation

The diagram above shows the simple GraphQL mutation request sequence. Simple means that we are not using anything to display data from the server. We can proceed with the request in two more ways - with updating the cached data and with using of optimistic response. Firstly I will describe updating cached data, and then I will describe the optimistic response.

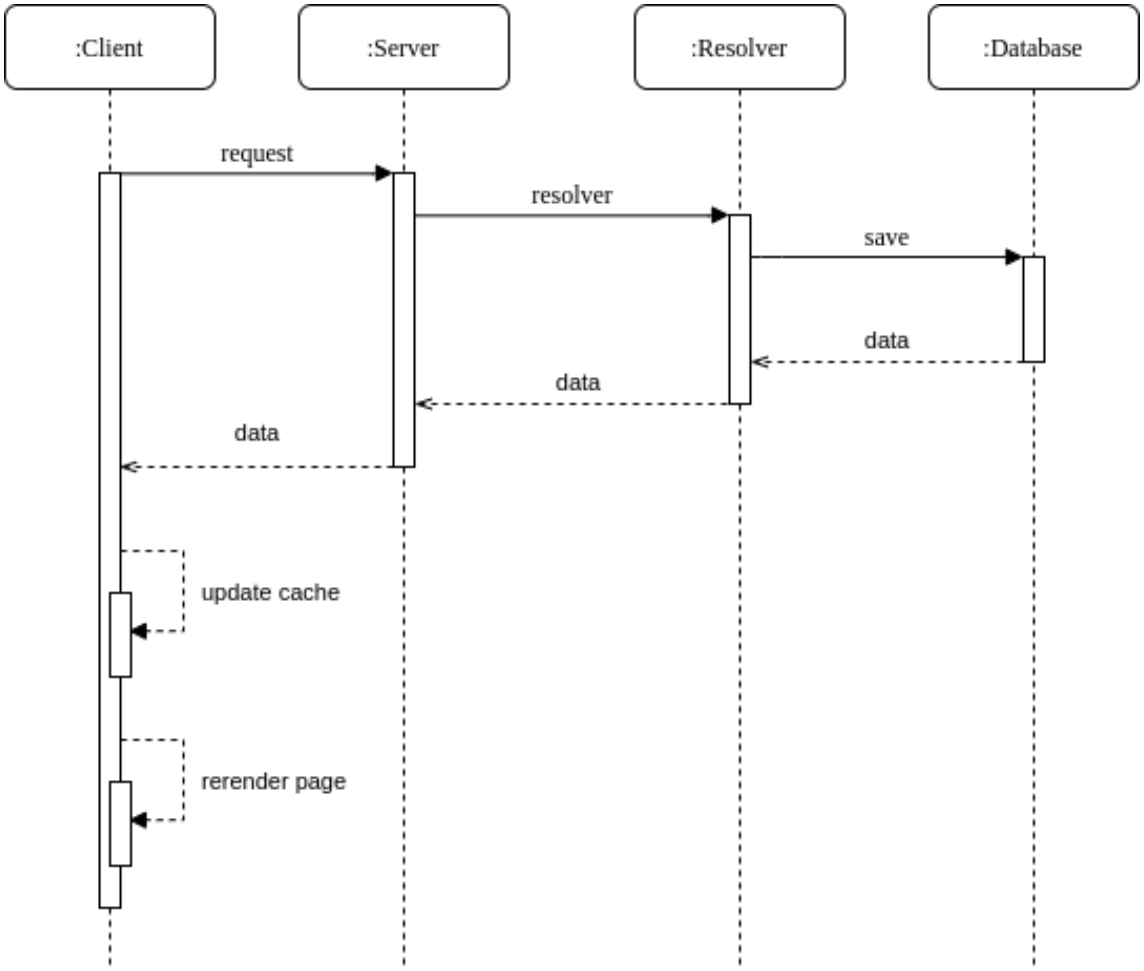

Figure 4.4: GraphQL mutation with updating the cached data

Updating the cached data

The diagram above shows the GraphQL mutation request and cache updating sequence. After receiving new data from the server, we are calling the "update" function. We are passing this function to mutation options.

```
const languagesQuery = gql‘
  query {
    languages {
      id
      title
    }
  }
‘

const createLanguageMutation = gql‘
  mutation CREATE_LANGUAGE($language: LanguageInput!) {
    createLanguage(language: $language) {
      id
    }
  }
‘
```

```
const fetchLanguages = async () => {
  const {data} = await this.$apollo.query({
    query: languagesQuery
  })

  return data
}

const createLanguage = (language) => {
  this.$apollo.mutate({
    mutation: createLanguageMutation,
    variables: {
      language: { ...language }
    },
    update: (cache, {data: { createLanguage }}) => {
      const { languages } = cache.readQuery({
        query: languagesQuery
      })

      const newData = langugages.push(createLanguage)

      cache.writeQuery({
        query: languagesQuery,
        data: { languages: newData }
      })
    }
  })
}

fetchData()
createLanguage({title: "JavaScript"})
```

Listing 4.30: GraphQL client caching data example

Firstly we are fetching the data from the server. It will fetch all the languages that we have in the database. After the query request completed, it will automatically save request data to the cache. So if the component that has this data will be unmounted, on the next mount, the request will not be sent to the server again, it will take data from the cache.

After fetching the data, we want to create a new language, so we called "createLanguage" mutation. The mutation "createLanguage" returns object of "Language" type so we can get any "Language" parameter from the server. In this case, we need language id only because we already have the language title in the client.

After the mutation request is completed, we need to modify the cache manually. To do it, we are passing update function to mutation options. The first parameter of the "update" function is cached data. A cache is an object that allows us to interact with cached data. It has four primary methods:

- readQuery - this method enables you to run GraphQL queries directly on your cache. If your cache contains all of the data necessary to fulfill a specified query, readQuery returns a data object in the shape of your query, just like a GraphQL server does. If your cache doesn't contain all of the data necessary to fulfill a specified query, readQuery throws an error. It never attempts to fetch data from a remote server. Example:

  ```
  const client = new ApolloClient(...)
  const cache = client.cache
  ```

```
const languagesQuery = gql`
  languages {
    id
    title
  }
`

const {data} = await client.query({
  query: languagesQuery
})

const {languages} = cache.readQuery({
  query: languagesQuery
})
```

Listing 4.31: GraphQL client read data from the cache example

In the example above, "data.languages" and "languages" will be almost the same fields that contain "Language" type objects. They will not be exactly the same because Apollo modifying fetched data with key property, so we can access this data in the cache with this key. Usually the key is the same as id property.

- readFragment - this method enables you to read data from any normalized cache object that was stored as part of any query result. Unlike readQuery, calls to readFragment do not need to conform to the structure of one of your data graph's supported queries.
- writeQuery and writeFragment - with method you can write arbitrary data to the cache. These methods have the same signature as their read counterparts, except they require an additional data variable. Example:

The second parameter of the "update" function is the data object. This is data that we are fetching after mutation. In this case, we are fetching "id" property, so after data will be an object called "createLanguage" with single property "id". Example of using the "update" function:

```
const update = (cache, {data: { createLanguage }}) => {
  const { languages } = cache.readQuery({
    query: languagesQuery
  })

  const newData = languages.push(createLanguage)

  cache.writeQuery({
    query: languagesQuery,
    data: { languages: newData }
  })
}
```

Listing 4.32: GraphQL client mutation update cache example

In the example above, firstly, we are fetching data from the cache with "readQuery" function. After fetching data, we are creating new data that we will push to cache. So "newData" will be a field with a new language. Then we are writing new data to cache with "writeQuery" function.

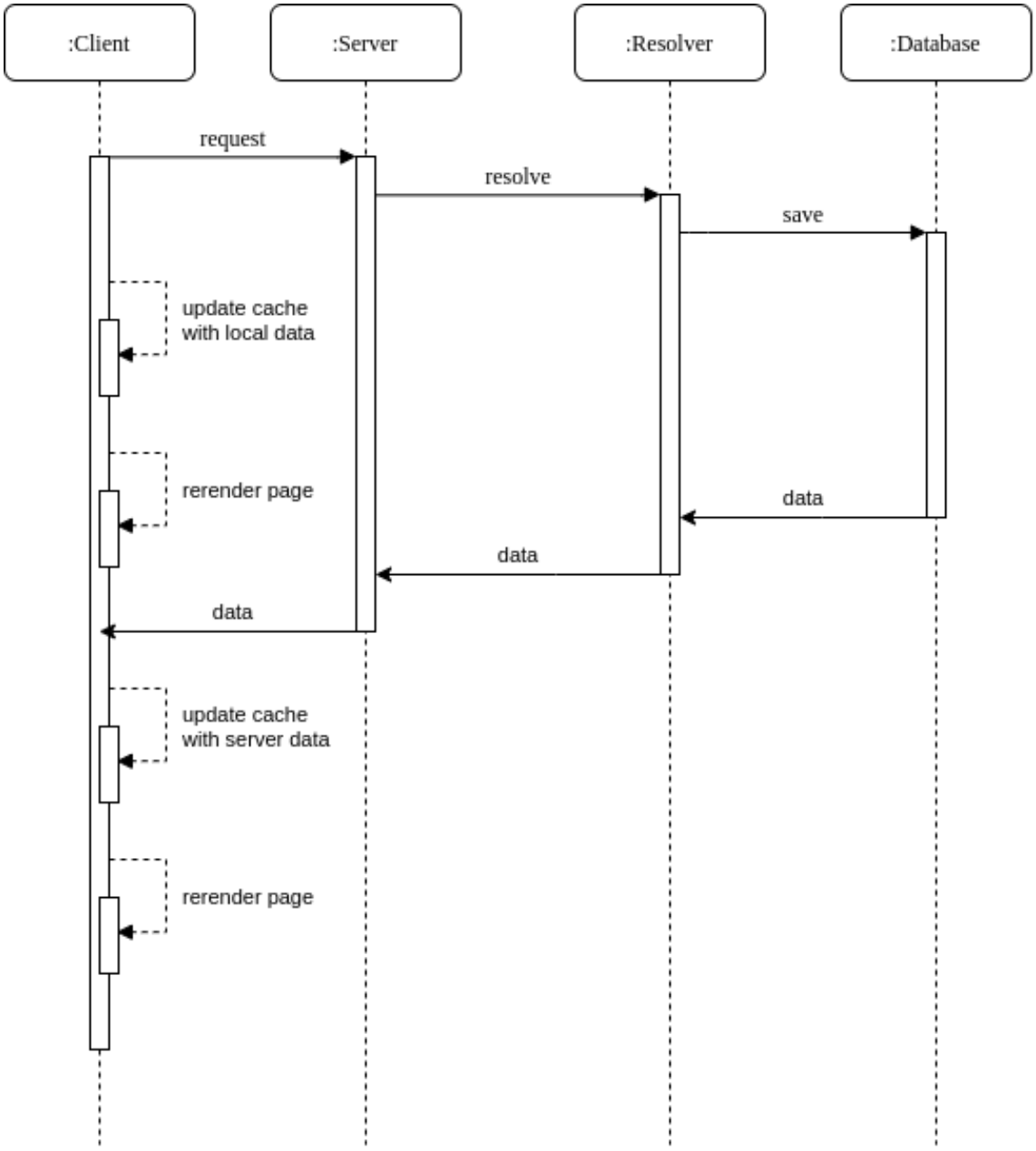

Figure 4.5: GraphQL mutation with optimistic response

Optimistic response

The diagram above shows the GraphQL mutation request with an optimistic response.

Optimistic UI is a pattern that you can use to simulate the results of a mutation and update the UI even before receiving a response from the server. Once the response is received from the server, the optimistic result is thrown away and replaced with the actual result. Optimistic UI provides an easy way to make your UI respond much faster, while ensuring that the data becomes consistent with the actual response when it arrives. [23]

So the idea of an optimistic response is to create a “fake” request response from the server by defining data that we expect to be returned from the server. In most cases,

we have all data that we are creating on the client application, and only data we need from the server is an id that will be generated on the server. For example, when we are creating a new language, we expect that server will return the object of type "Language" with properties:

- title - which is a string with the value that we entered on the client
- typename - which is a string with value "Language" that GraphQL will add on the server
- id - the property that will be generated on the server.

To use an optimistic response on the client application, we need to pass optimistic response property to the mutation options. In this property, we are defining the object that we expect that will be returned from the server. Example:

```
const createLanguage = (language) => {
  this.$apollo.mutate({
    mutation: createLanguageMutation,
    variables: {
      language: { ...language }
    },
    update: (cache, {data: { createLanguage }}) => {
      const { languages } = cache.readQuery({
        query: languagesQuery
      })

      const newData = langugages.push(createLanguage)

      cache.writeQuery({
        query: languagesQuery,
        data: { languages: newData }
      })
    },
    optimisticResponse: {
      __typename: "Mutation",
      createLanguage: {
        __typename: "Language",
        id: -1,
        title: language.title
      }
    }
  })
}
```

Listing 4.33: GraphQL client optimistic response example

In the example above, we defined that we are expecting an object "createLanguage" (the name is the same as mutation name). For property "id" we are using -1 value because we do not know what id will be generated on the server. After sending the mutation request, the client application will update the cached data with the optimistic response data, and then after fetching data from the server, the client application will update cached data again but with data from the server.

There are some use cases where the optimistic response may cause errors on the server. For example, if we are using optimistic response, after sending "createLanguage" request and then using "updateLanguage" (which is the mutation that updating language) or

"deleteMutation" (which is the mutation that deleting language) without waiting for the data from the server, then it will send a request with language id equals to -1. In that case, we need to handle this kind of thing on the client application. For example, make the delete button disabled if language id equals to -1.

4.7 Real time applications

In this chapter I will describe how to create real-time application with GraphQL. the GraphQL spec supports a third operation type, called subscription. GraphQL subscriptions are a way to push data from the server to the clients that choose to listen to real time messages from the server. Subscriptions are similar to queries in that they specify a set of fields to be delivered to the client, but instead of immediately returning a single answer, a result is sent every time a particular event happens on the server.

GraphQL subscription

A common use case for subscriptions is notifying the client side about particular events, for example the creation of a new object, updated fields and so on. [24] In our server, I am using a subscription for creating chat. After clients starts the application it will send subscription "newMessage" with parameter "targetId". That means that the client is subscribed to the new message event with one filter "targetId".

```
type Mutation {
  createMessage(message: InputMessage!): Message
}

input InputMessage {
  text: String!
  from: String!
  to: String!
}

type Message {
  id: String
  text: String
  from: String
  to:String
}

type Subscription {
  newMessage(targetId: String!): Message
}
```

Listing 4.34: GraphQL subscription schema example

In the example above, the server has a mutation "createMessage". This mutation accepts one argument "message". This argument has three fields:

- text - message text
- from - id of the user that is sending the message
- to - id of the user that will receive this message

Inside the resolver of this mutation, the server emits an event that the new message was created. Then the server will check current "newMessage" subscriptions. This subscription has one parameter "targetId". If "targetId" of one of subscription will be equal to "to" parameter, then the server will send data to the client that is subscribed with this parameter. This simple event-based principle helps to create real-time applications with GraphQL. REST web architecture do not provide any specific way how to create real-time applications. In REST client may send the request with interval to check if there is new data on the server, but this is the unoptimized way of creating such applications.

4.8 File transfer

REST

To upload the files to the storage (in our system we are using Google storage) it will need to create REST endpoints, because GraphQL does not supporting transferring binary data. GraphQL will need to use external libraries to make it work. In REST web architecture it will need to create endpoint and send data for example as Blob. The Blob object represents a blob, which is a file-like object of immutable, raw data; they can be read as text or binary data, or converted into a ReadableStream so its methods can be used for processing the data:

Blobs can represent data that isn't necessarily in a JavaScript-native format. The File interface is based on Blob, inheriting blob functionality and expanding it to support files on the user's system.[25] Example

```
const sendData = async () => {
  const file = new Blob(["<h1>HTML file</h1>"], { type: "text/html" });
  const formData = new FormData();

  formData.append("file", file);

  await fetch("http://localhost:3000/import-file", {
    headers: {
      "content-type": "multipart/form-data"
    },
    body: formData
  })
}
```

Listing 4.35: Example of sending data to the REST endpoint with Blob

Chapter 5

Conclusion

This chapter will shortly introduce a summary of each section from "Implmentation" chapter

API structure

Types

GraphQL uses SDL. Every GraphQL service defines a set of types that describe entirely the set of possible data that can query on that service. Then, when queries come in, they are validated and executed against that schema. So, you can always get predictable result. REST web architecture does not provide a type system. The resource naming principles is the only thing that can describe resource expect data.

Documentation

After defining the schema, GraphQL creating documentation for API, to let know what exactly API can do. With GraphiQL or GraphQL Playground, developers can inspect schema and even run queries and mutations to test out API. For REST, it needs to use external services like Swagger and manually create documentation for each REST controller.

Schema tools

GraphQL provides many tools, such as directives, enums, fragments, and inputs. Each of this tool helps to make the code clearer, reduce code repeating and make it more understandable for other developers

Data fetching/modifying

The most significant improvement that GraphQL introduced is data fetching. In a typical REST API, to fetch or retrieve data from a server, we might end up making requests to multiple endpoints. But with GraphQL, we only have one endpoint with which we access data on a server. With a single request, we can get an object and its related objects.

Security

GraphQL provides schema directives to secure resolvers. However, directives might be used not only for security reasons. In the REST web architecture, the usual way to secure resources is to use middlewares for defined URIs.

API endpoints

Because GraphQL using single URI for all requests, it will be problematically to create API for other systems. For this purpose, it is better to use the REST endpoint, which is running on different URIs.

Technologies

Because GraphQL is the type system, it has many tools that help the developers to build modern web applications with features such as "Optimistic response". Also, there are many tools like `https://graphql-code-generator.com/` that help developers to reduce manual writing of code and make it more automatized.

Real-time applications

One of the most significant advantages of GraphQL is Subscriptions. Subscriptions are a GraphQL feature that allows a server to send data to its clients when a specific event happens in real-time. GraphQL describes the easy way of creating such features. REST web architecture do not provide any specific way how to create real-time applications. In REST client may send the request with interval to check if there is new data on the server, but this is the unoptimized way of creating such applications.

Files transfer

If the webserver using GraphQL without any libraries, then transferring documents from the client to the server will be possible only with REST API. This is the place where REST web architecture taking a step forward.

Appendix A

Application Setup Instructions

In order to run application there are following software required to be installed on the machine:

- Yarn 1.22.4
- NPM 6.14.4
- Node.js 14.2.0

Run *yarn install* in the root folder from the terminal. It will install project dependencies. After installation, run *yarn run price66-dev* and *yarn run api-dev* from the root folder, it will start API and web application. After start, web application will be available at `http://localhost:8024` and server at `http://localhost:8380`. You can test GraphQL requests with GraphQL playground. This is the graphical interface for developers. After server launched, playground will be available at `http://localhost:8380/graphql`. To test REST endpoint you can use any tool for making HTTP requests, for example Postman

To start Cypress[26] end-to-end[27] tests, run *yarn run cypress:open* from the root folder. It will open a Chorium window where you can start tests.

Deployed version is running on `https://app.price66.com/`

Appendix B

List of Abbreviations

API Application Programming Interface

CRUD Create, Read, Update and Delete

REST Representational State Transfer

URI Uniform Resource Identifier

URL Uniform Resource Locator

ORM Object-relational mapping

SDL Schema definition language

JSON JavaScript Object Notation

HTTP Hypertext Transfer Protocol

UML Unified Modeling Language

Bibliography

[1] GraphQL usage chart. Available from https://www.drupal.org/project/usage/graphql.

[2] Web API definition. Available from https://en.wikipedia.org/wiki/Web_application.

[3] REST API definition. Available from https://en.wikipedia.org/wiki/Representational_state_transfer.

[4] GraphQL definition. Available from https://en.wikipedia.org/wiki/GraphQL.

[5] IBM. *JavaScript Everywhere and the Three Amigos (Into the wild BLUE yonder!)*. 2013, available from https://community.ibm.com/community/user/ibmcommunity/home.

[6] Ledvinka, M. HTTP, REST Web Services. 2018, available from https://cw.fel.cvut.cz/b181/_media/courses/b6b33ear/lectures/lecture-06-rest-s.pdf.

[7] Introduction to GraphQL. 2018, available from https://graphql.org/learn/.

[8] Express.js - Node.js framework. Available from https://expressjs.com/.

[9] MongoDB - document database. Available from https://www.mongodb.com/.

[10] Mongoose - ORM for MongoDB. Available from https://mongoosejs.com/.

[11] Google Storage - CDN for storing documents. Available from https://cloud.google.com/storage.

[12] JWT - decode, verify and generate JWT. Available from https://jwt.io/.

[13] Apollo client. Available from https://www.apollographql.com/docs/react/.

[14] Apollo server. Available from https://www.apollographql.com/docs/apollo-server/, keywords = "Apollo, server".

[15] Yarn - package manager for JavaScript. Available from https://yarnpkg.com/.

[16] Client Server Architecture definition. Available from https://cio-wiki.org/wiki/Client_Server_Architecture.

[17] GraphQL schema definition. Available from https://graphql.org/learn/schema/.

[18] REST Resource Naming Guide. Available from https://restfulapi.net/resource-naming/.

[19] Polyfill. Available from https://developer.mozilla.org/en-US/docs/Glossary/Polyfill.

[20] GraphQL directive definition. Available from https://graphql.org/learn/queries/.

[21] GraphQL resolver definition. Available from https://www.apollographql.com/docs/graphql-tools/resolvers/.

[22] Apollo definition. Available from https://www.apollographql.com/.

[23] Apollo Optimistic UI definition. Available from https://www.apollographql.com/docs/react/performance/optimistic-ui/.

[24] GraphQL subscription definition. Available from https://www.apollographql.com/docs/react/data/subscriptions/.

[25] Blob definition. Available from https://developer.mozilla.org/en-US/docs/Web/API/Blob.

[26] Cypress - JavaScript end-to-end testing framework. Available from https://www.cypress.io/.

[27] End to End. Available from https://www.valentinog.com/blog/cypress/.