Implementing GraphQL in a Simple Web Application

Fanny Örte

Bachelor’s Thesis in Computer Science
at Stockholm University, Sweden, 2021

Implementing GraphQL in a Simple Web Application

Fanny Örte

Bachelor’s Thesis in Computer Science (15 ECTS credits)
Bachelor’s Programme in Computer Science
Stockholm University year 2021
Supervisor at the Department of Mathematics was Lars Arvestad
Examiner was Woosok Moon

Department of Mathematics
Stockholm University
SE-106 91 Stockholm, Sweden

Abstract

This report investigates the potential advantages and disadvantages with implementing the GraphQL specification for a small-scale web application. GraphQL is a new language used for client-to-server communication implemented to be an alternative to the more traditional RESTful technology used for the same purpose.

The purpose of this report is to give the reader some insight into the complexity of GraphQL. The aim is also to hopefully give an idea of how GraphQL can be implemented for, and influence, a web development project.

For this purpose a simple web application using the web technologies React, Node and Express was implemented. The client-side of the application was implemented to contain two features, one that allows users to fill out a contact form, and another that allows users to log in. The server-side was implemented to store and deliver the appropriate data for each of these features. The client to server communication was first done using RESTful technology and was later compared to using GraphQL instead. GraphQL was implemented using the Apollo server and Apollo client libraries.

The study found that there were more advantages than disadvantages. Both advantages and disadvantages will depend on how extensive the web application is.

Sammanfattning

Implementering av GraphQL i en enkel webbapplikation

Den här uppsatsen redogör för vilka potentiella fördelar och nackdelar det finns med att implementera och använda GraphQL i en småskalig webbapplikation. GraphQL är ett nytt språk som används för kommunikation mellan datorer över internet och är ett alternativ till teknologin RESTful som används för samma syfte.

Syftet med denna uppsats är att ge läsaren en inblick i hur komplext GraphQL är och hur GraphQL kan användas i ett webbutvecklingsprojekt.

För detta syfte implementerades en enkel webbapplikation med hjälp av ramverken React, Node och Express. På klient-sidan av applikationen implementerades två funktionaliteter, en som låter användaren av hemsidan fylla i ett kontaktformulär och en annan som tillåter användare att logga in till en personlig sida. Server-sidan av applikationen implementerades att lagra och leverera lämplig data för att stödja båda funktionaliteterna. Kommunikationen från klient till server implementerades först med RESTful och sedan användes GraphQL för att göra en jämförelse mellan de båda teknologierna. GraphQL implementerades i webbapplikationen med hjälp av Apollo-biblioteken.

Resultatet av studien visade att det finns fler fördelar med GraphQL än nackdelar. Fördelar och nackdelar kommer att bero på hur omfattande webbapplikationen är.

Contents

1 Introduction

Being able to share data between computer systems over the internet is and has always been fundamental. Since the introduction of the RESTful API in the year 2000 it has become the most popular technology used to implement the communication between two computers over the web. However, a new technology is spreading.

RESTful (Representational State Transfer), or just REST, is a set of recommendations on how to implement the communication between the client, the computer requesting data, and the server, the computer sending data. With the REST API (application programming interface) a request is made over HTTP from the client to a URL that is connected to a server. The URL is a path to the server application, called an endpoint.

For an endpoint a RESTful server can implement 4 different methods. GET for returning requested data, POST for accepting new data, DELETE and PUT for deleting and updating existing data. The client can make these requests. The response to a request is typically sent in JSON format (JSON is JavaScript Object Notation).

With REST a client often needs to make multiple requests in order to get all the data required. This can be a problem in terms of latency. From these multiple requests the client usually also gets more data than required.

As an example, a client wants to access information about the top 10 selling books in a bookstore and would also like to have the author names for each of the 10 books. To get a list of books containing author id the client might need to make a GET request at the "/book/" endpoint. In order to get the name of each author another 10 separate requests need to be made using the author id's from the previous result. In addition to this, making a GET request at the "/author" endpoint will result in data that is not intended to be used. It is possible for the REST API to include all, or just a small portion of, the information about the author. However this would most likely mean passing on unnecessary data to another clients request (1).

In 2012 Facebook experienced an optimisation problem with REST when using it for their mobile application servers. As a solution a new client-to-server communication technology, GraphQL, was created. GraphQL is a query language and was made public as open source in 2015 (2).

While REST leaves it up to the server to specify what data is passed from the server to the client GraphQL can be described as a client-driven architecture. On the server-side a GraphQL schema specifies what data exists and how this data is presented to the client when making the request. The client is in control of what data is returned from the server by executing a query against it. The response data from the server has the same format as the query (3).

1.1 Motivation and aim

FaceBook, together with many other companies, e.g. Twitter and Github make use of GraphQL in their products (4). Although REST is the most popular web technology for sharing data between client and server via HTTP, GraphQL seems to be a trending topic in the field of API technologies. GraphQL promises multiple advantages over REST but could also be excessive for small applications (3).

This study aims to contribute with knowledge about using and implementing GraphQL for a web application. For the purpose of this study a simple web application was created with aim to further investigate if and potentially how GraphQL can benefit or disadvantage a website.

1.2 Research Question

What are pros and cons when implementing GraphQL in a simple web application that communicates with a small scale back-end server?

1.3 Delimitation

In order to limit the scope of this study, the focus will be on comparing GraphQL to the RESTful technology.

2 Background

In this chapter essential concepts for this thesis will be presented.

2.1 Client- and server side

In web development there are mainly two parts to creating a website, the client-side and the server-side. Client- and server-side developers typically need to work closely together in order to maintain a well-designed and functioning product.

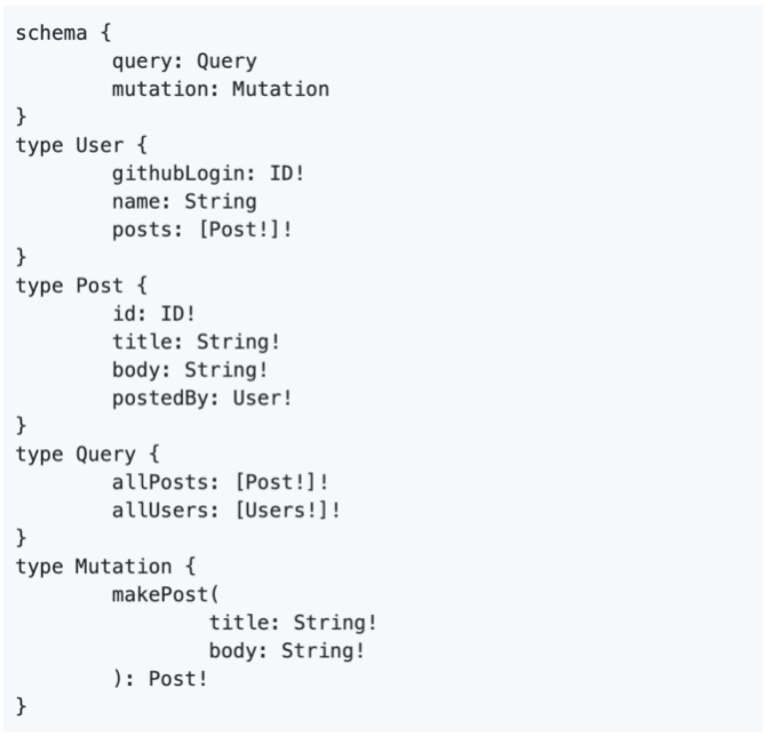
The client-side is the software that runs on a users web browser. The web browser can read Javascript, Html and Css code. Developing a website on the client-side means implementing every part of the website that a user sees and interacts with. The client-side is also referred to as the front-end of the application.

The server-side is the part of a website that run on the server. The client-side is a platform that displays and collects data from the user, the data is retrieved and sent to the server-side. The server-side, or the back-end, of the application is built to work behind the scenes to manage the data of the application that is typically stored in a database (5).

2.2 API gateway

An API gateway is a type of middleware that goes between the client and server. API gateways are especially useful when the back-end of an application is broken up into small independent services, in what is known as a microservice architecture. The Microservice architecture is popular since it offers plenty of benefits including that is makes an application easier to develop, deploy and maintain.

An API gateway serves as a central application interface. Instead of sending request directly to a service at the back-end, the client sends them to the API gateway who passes the request to the appropriate service. The API gateway can also be useful for a number of other things, for example to authenticate a user. If the user needs to fetch data from multiple services, they need only to be authenticated once. This reduces the time for data to travel from client to server and ensures the authentication process stays consistent across the application (6).



```
schema {
        query: Query
        mutation: Mutation
}
type User {
        githubLogin: ID!
        name: String
        posts: [Post!]!
}
type Post {
        id: ID!
        title: String!
        body: String!
        postedBy: User!
}
type Query {
        allPosts: [Post!]!
        allUsers: [Users!]!
}
type Mutation {
        makePost(
                title: String!
                body: String!
        ): Post!
}
```

Figure 2.1: Example of a GraphQL schem for an application that lets users log in and make posts.

2.3 GraphQL

GraphQL is described as a query language used in client-to-server communication. On the client-side the language is essentially used to request data from a server. On the server-side GraphQL specifies how to present data to a client (3).

The GraphQL specification serves as a guideline of how to implement client-to-server communication, describing the capabilities and characteristics of the language. Following these guidelines, it is possible to implement GraphQL using any programming language (7).

In the next three subsections the basics of GraphQL will be described.

Schema

A GraphQL schema is a text document on the server-side consisting of a collection of types. These types are the data that the server-side of the application needs to store and deliver. The defined types are used by both client and server to validate data requests.

As an example of a schema, consider an application, like twitter, that lets users log in and post a text that will be broadcasted.

This app will store information about users and posts and its schema is depicted in figure 2.1.

A type in a GraphQL schema represents an object. These objects have fields that returns a specific type of data. In the schema, the “User” type has three fields. The fields “githubLogin” and “name” return a scalar data type. The scalar type String and ID are part of the GraphQL language. Both String and the ID are strings but the ID will be a unique identifier for each user. It is also possible to define custom scalar types. The field “posts” will return a list of the object type “Post”. The exclamation mark means that the field can not return something that is empty.

The object type “Post” has four fields. The first three fields return a scalar data type and the last field return a “User” object type.

The ability to query multiple types of related data is an important feature of the GraphQL language. When a field of an object is specified to return another object, these objects become connected. A post must be made by a user so the field “postedBy” will return that user. Likewise the “posts” field for the “User” object will return a list of “Posts” that is associated with the user.

While the “User” and “Post” types are defined specifically for the application, the Mutation and Query types are part of the GraphQL language. The Query type defines what types of data requests can be made to the server. The “allPosts” query will return a list of posts and the “allUsers” query will return a list of users.

The Mutation type defines what kind of data the server accepts, in this case “makePost” makes it possible to create new posts. Adding these in the “schema” type makes them available in the GraphQL API (7).

Resolvers

A Resolver is a function that is responsible for getting and returning the correct data for a single field specified in the schema. Every field must have a corresponding resolver function with the same name that returns the specified datatype (7).

Queries and mutations

When making a request to a GraphQL server the query operation is used to fetch data. The query specifies what data to receive by including fields that maps to the field with the same name in the schema defined on the server. As an example, a client can send the following query to a GraphQL server (that has defined the schema in figure 2.1):

```
query {
   allPosts {
         title
         body
         postedBy {
               name
         }
   }
}
```

This is a request to get a list of all posts with their title and body and also information about the name of the user who posted it.

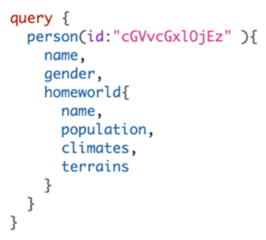
To write new data to the server the mutation operation is used. A mutation is written in a similar way as a query but also adds data to the server-side. To make a new post the title and body must be specified. For example, a user can make a new post and select information about the post that was just made with the following mutation (the id and information about the user will be automatically generated by the server) (7):

```
mutation {
   makePost (title: “Fun fact about elephants”,
               body: “Elephants are constantly eating”) {
         id
         postedBy {
               id
               name
         }
   }
}
```

As another example, figure 2.2 shows a query and the response sent from the Star Wars GraphQL API (8). In the query a request was made for the name, gender and also information about the name, population, climates and terrains of the home world for a person with the specified id.

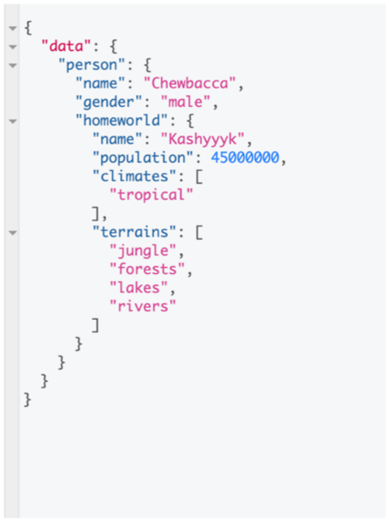
2.4 Node, Express and React

Node is a runtime environment that enables developers to create server-side applications in JavaScript. This is beneficial since it creates less of a gap between client-side and server-side developers when both sides write code in the same language.



```
query {
  person(id:"cGVvcGxlOjEz" ){
    name,
    gender,
    homeworld{
      name,
      population,
      climates,
      terrains
    }
  }
}
```

(b) *GraphQL query*



```
{
  "data": {
    "person": {
      "name": "Chewbacca",
      "gender": "male",
      "homeworld": {
        "name": "Kashyyyk",
        "population": 45000000,
        "climates": [
          "tropical"
        ],
        "terrains": [
          "jungle",
          "forests",
          "lakes",
          "rivers"
        ]
      }
    }
  }
}
```

(a) *Response object*

Figure 2.2: Query to, and response from the Star Wars GraphQL API

Node also has many other benefits, among other things it is optimised to be used in web application and encourages developers to take advantage of new improvements in language design. The node package manager (npm) also provides hundreds of thousands reusable packages that can be added to a project.

Express is a Node web framework that, among other things, enables developers to write handlers for HTTP (Hypertext Transfer Protocol) requests. These handlers make it easy to implement REST APIs that can work with cookies, URL parameters, sessions and many other things (9).

React is a JavaScript library for building user interfaces on the front-end of the application. React is component based, each component has its own state. React will efficiently update and render the right components when there are data changes to the application. Therefore it is a library that makes it easy to manage and update different views depending on the state of the application (10).

2.5 Apollo client and server

There are multiple libraries for implementing GraphQL into a project. For implementation on the server side the Apollo Server is the most popular one (3). With Apollo Server it is easy to build a production ready and self-documenting GraphQL server with Node, that can use data from any source. Apollo Server also provides the GraphQL playground on the same domain as the implemented Apollo GraphQL server. The playground is a graphical user interface (GUI) that can be used by the web browser during development (11).

Apollo client is a JavaScript library for fetching, cashing and modify application data with GraphQL and can be integrated in a React project (12).

The documentation for the Apollo libraries is extensive and provides many examples which makes the libraries easy to use.

3 Method

This study was done as a part of a bigger project implementing a full-stack web application for the company Value Delivery IT Consultancy. In order to find answers to the research question two website features where implemented, one that allows an end-user to contact the company and another that enables a user to log in to a personal account.

In order to completely investigate the benefits of using the GraphQL specification for the client to server communication, an implementation using the REST architecture was firstly done. With REST the client-side of the web app communicated directly to a service on the server-side by making post, get and delete requests to the exposed end-points. Secondly an API gateway was implemented using the GraphQL specification. With this implementation the client-side makes requests to the gateway using queries and mutations. The gateway works as a single end-point for the client-side and is responsible for populating these queries with data from the appropriate service.

3.1 The Web Application

The client-side (or front-end) of the web application was written using the user-interface library React.js. The server-side (or back-end) consists of two different services, the Contact Service and the Authentication Service. These services where implemented on different domains and with the web application framework Express.js. The server-side also consist of a service implemented with Express.js as an API gateway that uses the GraphQL specification to handle client-side requests.

In the next sections further details about how the web application was implemented around the above mentioned services will be presented.

3.2 Authentication Service

An end-user of the website for Value Delivery Consultancy is an employed consultant. A consultant of the company should be able to, among other things, review a personal account listing his or her technical competence and edit or upload a CV. As a first step to implement this feature an authentication service that allows users to log in and out was built for the server-side of the application.

The authentication service API provides four different end-points and communicates with a database that stores information about users and user sessions. An overview of the database can

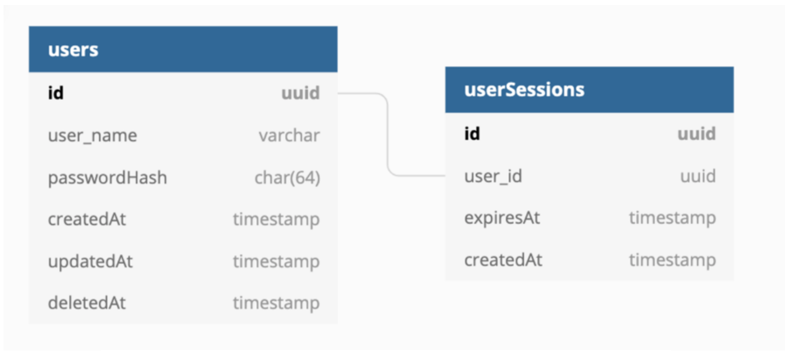


Figure 3.1: The Authentication Service database diagram

be seen in figure 3.1. This API is implemented using the REST architecture and enables client computers to make the following requests:

- A post request to create a new user in the database table "users" and a get request to fetch information about a given user.
- A post request to create a new user session in the database table "userSessions" and a get request to fetch information about a given user session.
- A delete request to remove a given user session from the database.

Client to server communication using REST

The user interface for the log-in functionality was implemented with a new page "account" on the front-end of the website. The account page displays a log-in form and to log in the user enters a user name, and corresponding password, of a user that exists in the database table "users" on the server-side.

When a user submits the log-in form the front-end is triggered to firstly make a post request to the authentication service at the end-point "/api/v1/sessions", sending along the entered user name and password as a JSON object. This request is made so that the back-end can process the log-in request which means checking that the user exists and is authenticated (the right password was entered). The end-point is named v1 for version one because if changes need to be made to the REST API they will be added in a new version. A session is what is created in the database table "userSessions". When there exists a session for a user in the database then the back-end knows that this user is already authenticated and logged in to the application.

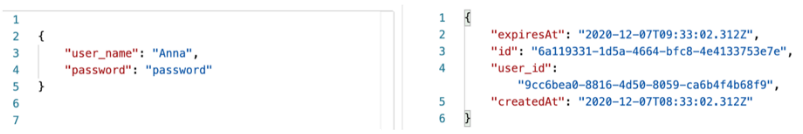
In order to handle the different HTTP request made to the authentication service, Express.js was used. First a new instance of express was created and set up to listen to port 8080. To handle incoming request when a user wants to log in, the method “post” of the express instance was used.

The post method accepts a string that is the end-point (in this case “/api/v1/sessions”) and a function as parameter. The function is a call-back function and will be called when an incoming post request is made at the end-point, in other words, when the user request to log in to the application. The function accepts three parameters req, res and next. The req parameter is the request that is being send and the res parameter is the response that will be returned. The next parameter is used to return an error from the authentication service. This is the standard way to, with Express.js, write functions that handles HTTP requests.

The body of the call-back function is where the code for authentication of a user was added. First the incoming request will be checked to contain a user name and a password. An error will be generated if this was not provided. Additional errors will be generated if the user name did not exists in the database table “users” or if the password did not match the corresponding “passwordHash”. The password check is done using the node package bcrypt. If no errors where generated the user was authenticated and a new user session will be created in the database table “userSessions”. The id for the user session is generated using the node package uuidv4, the user_id is the id of the authenticated user (found in the database table “users”) and the node package date-fns will be used to add the “expiresAt” timestamp, the “createdAt” timestamp will be automatically generated in the database.

The response that is then returned is the newly created user session as a JSON object. Also a cookie named “userSessionId” that stores the id of the user session (the id that was generated when creating the new user session). An example of a request and response is depicted in figure 3.2. The cookie acts as a secret key that is stored in the browser of the user and, when present, is sent along with every request to the server who in turn uses this key to authenticate the user.

When a user has been authenticated, a new user session has been created at the back-end and the front-end has received the response object and cookie, a second API call is made from the front-end to the back-end. The front-end wants not only to authenticate a user, it also wants to access information about the user in order to update the user interface for the account page. In order to do so a get request is made at the endpoint “/api/v1/users/:user_id”, where “:user_id” is replaced with the id of the user that was sent back as a result from the previous request to the server.



```json
{
    "user_name": "Anna",
    "password": "password"
}
```

```json
{
    "expiresAt": "2020-12-07T09:33:02.312Z",
    "id": "6a119331-1d5a-4664-bfc8-4e4133753e7e",
    "user_id":
        "9cc6bea0-8816-4d50-8059-ca6b4f4b68f9",
    "createdAt": "2020-12-07T08:33:02.312Z"
}
```

(a) Body sent with post request. *(b) Response object.*

Figure 3.2: Log-in communication from client to server, using REST.

In a similar way as for the log-in request, the authentication back-end service uses the get method on the express instance with the string "/api/v1/users/:user_id" and the same type of call-back function as parameters. The call-back function is executed when an incoming get request is made to this end-point and checks the database for the user with the given id, returning an error if user does not exist or responds with the data for the given user as a JSON object.

After these two separate calls the front-end selects the id of the user session and also the id and name of the user. This information is then stored in a global variable. This global variable is available on the entire domain during the run-time of the front-end application. The variable acts as the global state and allows the front-end to check if a user session is set, that is, if a user is logged in. If the user is not logged in the variable will be empty.

When a user is logged in, and the front-end has received the data about the user from the get request, the account page will display a view of personal information. As a start, a simple welcome message containing the user name and a log-out button.

When the user logs out a delete request is sent to the endpoint "/api/v1/sessions/:session_id", replacing "session_id" with the session id that is set in the global state. The authentication service uses the delete method on the express instance for this endpoint. The call-back function removes the user session with this id from the database and clears the cookie. The front-end clears the global variable so that the view with the log-in form will be displayed on the account page again.

In order to make sure that the front-end always knows when a user is logged in, two additional requests are sent to the back-end every time the application front-end re-renders. This is done in order for the account page to keep the view of the personal information as long as the user has not logged out or the session has expired on the server (in that case the user should also be logged out of the system).

This time the front-end wants to access information about the session set in the cookie in the browser. First a get request is send to the server endpoint "/api/v1/sessions", the server checks if a cookie containing a user session id was set. If a cookie was not set nothing will be returned and the front-end will clear the global state. If instead the user session, with the given id set in the cookie, has not expired it is returned from the server. Another call is made to the endpoint "/api/v1/users/:user_id". From the requests the id of the user session and also the id and user name of the user is set in the global state. At the back-end the functions that handles these request are implemented in the standard way with Express.js as previously mentioned.

Any errors occurred on the server-side when submitting the log-in from will be sent to the client-side who in turn will display an error message to the user.

3.3 Contact Service

Another end-user of the website is a potential client to Value Delivery Consultancy looking to hire a consultant. For this purpose the user should be able to contact the company.

The contact service REST API was implemented providing a single end-point "/api/v1/contact-service" which is associated with a post method.

Client to server communication using REST

The user interface was implemented with a new page "clients" and a contact form is displayed on this page. The client-side also validates that the entered value for all the form fields is in the correct format before the data is sent to the back-end as an object containing the data for all fields.

When the contact-service receives a post request, the received data object is firstly checked to contain all the required fields. Secondly an email containing all information is sent to contact@valuedelivery.se. The response to the client is a JSON object that contains information about if the mail was send successfully or not.

3.4 Client to server communication using GraphQL

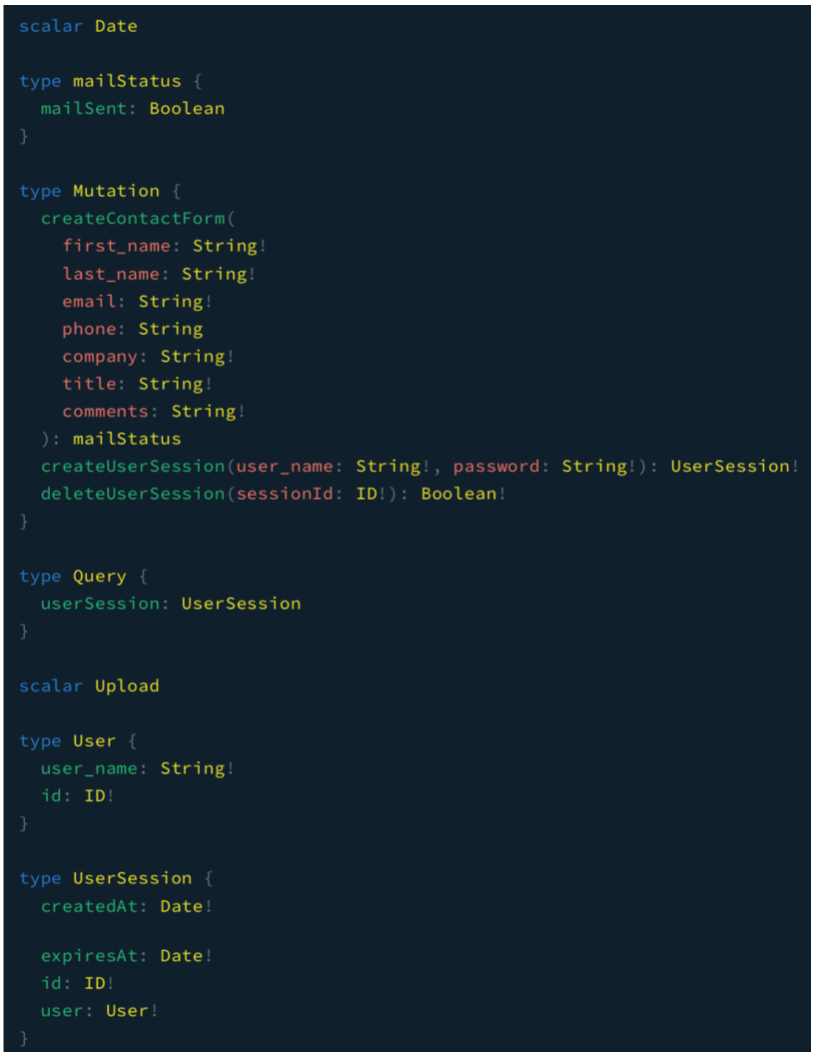
When implementing the client to server communication using GraphQL an additional service, an API gateway, was created. The gateway can be thought of as a part of the server-side, or back-end, of the web application, adding an additional layer between the two RESTful services. The front-end of the application was then changed to execute mutations and queries against the GraphQL gateway instead of making post, get and delete request directly to a back-end service. The back-end was changed to authenticate a user at the level of the gateway instead of at the individual authentication service. This gateway could also have been implemented to work as a middleware using RESTful instead of GraphQL. It was chosen not to do so for the purpose of this study since the company behind the website did not intend to use the gateway without GraphQL.

In order to use GraphQL the Apollo Server library was used in the API gateway service, a schema and resolvers where defined. Apollo Client was used to enable the client-side to execute queries and mutation against the GraphQL gateway server.

As the first step the different types needed for the application where identified. An overview of the schema can be seen in figure 3.3. Object types for the authentication service became "User" and "UserSession". Query and Mutation types for this service where "userSession", "createUserSession" and "deleteUserSession". The Contact service needed a "mailStatus" Object type and a "createContactForm" query.

When a user submits the log-in form the client-side of the application executes a mutation against the GraphQL gateway server, selecting the field "createUserSession". The return type for this field is the object type "UserSession" and the fields selected for this object type are "id" and "user". In turn the field "user" is of another object type "User" and the fields "id" and "user_name" are selected. An example request and response is depicted in figure 3.4.

The executed mutation is checked against the schema on the GraphQL server and the resolver for the field createUserSession is responsible for returning the correct data type. With Apollo Server library a post request is made to the authentication service in order to create a new user session in the database. The response from this request is an object containing all fields, with their corresponding value, of the newly created user session. With Apollo Server there is no need to have a custom resolver for each of the fields in the object type "UserSession", returning



```
scalar Date

type mailStatus {
  mailSent: Boolean
}

type Mutation {
  createContactForm(
    first_name: String!
    last_name: String!
    email: String!
    phone: String
    company: String!
    title: String!
    comments: String!
  ): mailStatus
  createUserSession(user_name: String!, password: String!): UserSession!
  deleteUserSession(sessionId: ID!): Boolean!
}

type Query {
  userSession: UserSession
}

scalar Upload

type User {
  user_name: String!
  id: ID!
}

type UserSession {
  createdAt: Date!

  expiresAt: Date!
  id: ID!
  user: User!
}
```

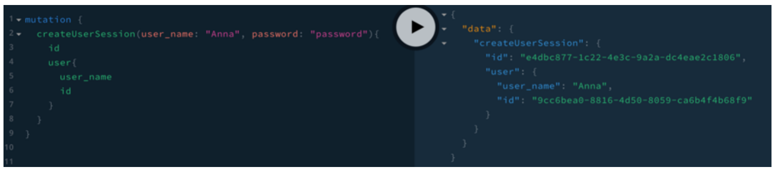
Figure 3.3: GraphQL schema for the API gateway

the object obtained from the post request to the authentication service will populate the corresponding fields of the "UserSession" object selected in the mutation.

The field "user" has a resolver and makes a get request, to the authentication service, with the user id obtained from the previous post request. The return object is used to populate the fields selected for the "User" object type.

To update the global user session state on the client-side a query, selecting the "userSession" field, is executed against the gateway. The gateway handles the request to fetch a user session with the id set in the cookie on the client. The fields selected on the UserSession type and User are the same as when executing the previous mentioned mutation. The difference is that this query could return the specified data or nothing at all.

When a user wants to log out from the app, the "deleteUserSession" field is selected on the mutation type using



```
mutation {
  createUserSession(user_name: "Anna", password: "password"){
    id
    user{
      user_name
      id
    }
  }
}
```

```
{
  "data": {
    "createUserSession": {
      "id": "e4dbc877-1c22-4e3c-9a2a-dc4eae2c1806",
      "user": {
        "user_name": "Anna",
        "id": "9cc6bea0-8816-4d50-8059-ca6b4f4b68f9"
      }
    }
  }
}
```

(b) Executed mutation on the client-side. *(a) Response object from the server-side.*

Figure 3.4: Log-in communication from client to server, using GraphQL.

the id of the current session. The resolver for this field returns true when this user session has been successfully removed from database on the server-side.

When a user submits the contact form, the client-side executes the field “createContactForm” of the mutation type.

4 Results

In this chapter the results of implementing GraphQL compared to REST will be presented.

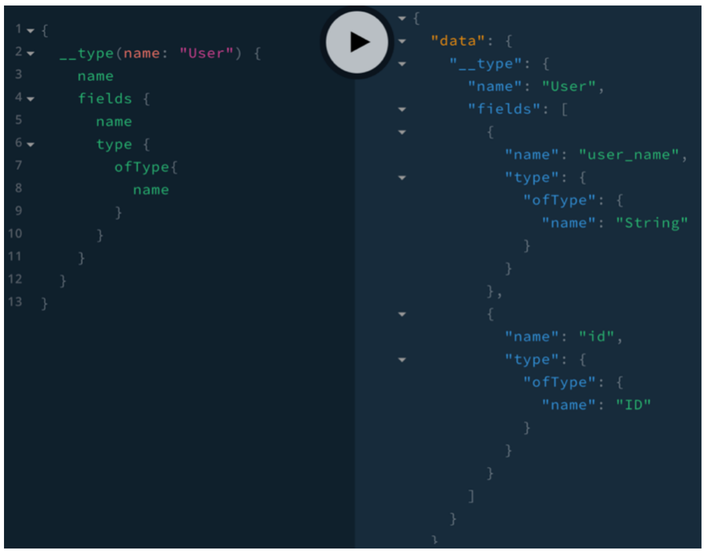
GraphQL set up

Implementing GraphQL into the web application involved learning about all the concepts of the GraphQL specification. It also required getting familiar with a library for setting up GraphQL, both on the client- and server-side of the application. Compared to the implementation using REST the GraphQL approach was more time consuming, although did not come with too many struggles thanks to the well documented Apollo libraries.

The study also found that, according to the GraphQL specification, a schema must contain a Query type. This was discovered when beginning the migration from REST to GraphQL with the contact service. To enable for the client-side to send data to the contact service, the API gateway need only to provide a mutation. This initial set up generated an error from the Apollo server library and resulted in adding a Query type that was later removed.

Introspection

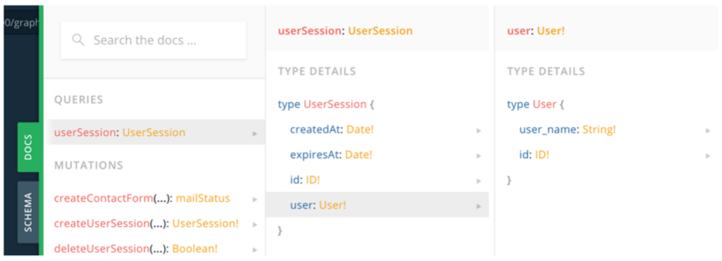
A beneficial feature found was that GraphQL supports introspection over the schema on the server-side. In figure 4.1 is an example of executing an introspecting query.



```
{
  __type(name: "User") {
    name
    fields {
      name
      type {
        ofType{
          name
        }
      }
    }
  }
}
```

```
{
  "data": {
    "__type": {
      "name": "User",
      "fields": [
        {
          "name": "user_name",
          "type": {
            "ofType": {
              "name": "String"
            }
          }
        },
        {
          "name": "id",
          "type": {
            "ofType": {
              "name": "ID"
            }
          }
        }
      ]
    }
  }
}
```

Figure 4.1: Example of a for type checking the schema, and a response



```
QUERIES
userSession: UserSession
MUTATIONS
createContactForm(...): mailStatus
createUserSession(...): UserSession!
deleteUserSession(...): Boolean!
```

```
userSession: UserSession
TYPE DETAILS
type UserSession {
  createdAt: Date!
  expiresAt: Date!
  id: ID!
  user: User!
}
```

```
user: User!
TYPE DETAILS
type User {
  user_name: String!
  id: ID!
}
```

Figure 4.2: Overview of the types using the GraphQL playground.

Apollo-server provides a UI tool, the GraphQL playground. With this tool developers can navigate into the types and discover the schema in a user friendly way. Figure 4.2 shows how the GraphQL playground can be used to get familiar with the types of the back-end.

There are UI tools for REST APIs, e.g. Postman, although it is not possible to check what kind of data is available on the server-side.

Data requests

For the log-in functionality the front-end of the application was simplified when using GraphQL. Instead of having to make two separate API calls and then extracting the data needed, only one data request was made and the data object returned had the desired format. With GraphQL there was one API call for every time the front-end re-renders and one call for when a user wants to log in, compared to REST when in those cases two separate calls and data-parsing was made.

With the google chrome developer tool the total duration, from the start of the request to the receipt of the final byte in the response was evaluated. At five different occasions the total time for a log-in request to be send, and a response to be received, was tested. The result can be seen in figure 4.3. The graph shows a comparison between the request loading time for the implementation with GraphQL and the implementation with REST (where the two different API requests where summarised). The results where different each time. This is probably due to difference in network speed and to that the functions handling the requests are asynchronous. Asynchronous functions are executed in parallel to the rest of the programming code and may cause the functions to finish executing at different orders each time.

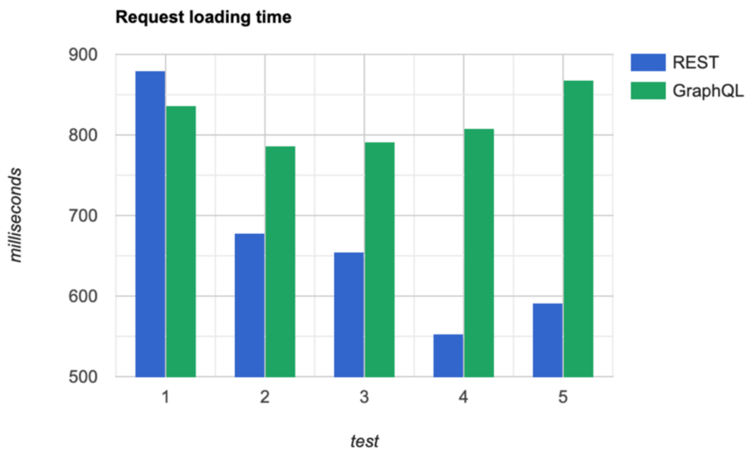


Figure 4.3: Request loading time on five different occasions, comparing REST with GraphQL.

API changes

With GraphQL it is possible to make some API changes to the server-side without effecting the client-side. The authentication service may at one point add additional fields to the database table “users”. With GraphQL these fields can be added to the “User” type in the schema and the existing Queries and mutations made from the client are not affected by this change since the client is in control of what is returned from the server. In other words this type of change will not cause the application to crash or give unexpected results to the client.

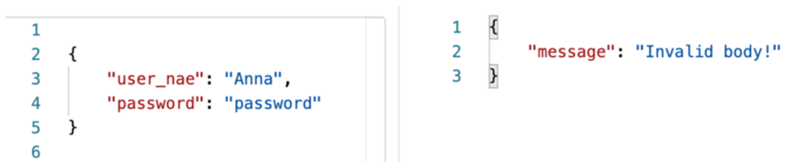
With REST, when client communicates directly to the endpoint “/api/v1/users/:user_id”, if similar changes are made, the client-side will not work as expected until these changes are handled.

Error handling

In some cases the implementation with GraphQL automatically presents the client with detailed error messages.

As an example, a common error is to send a request object in the wrong format to the server-side. With REST, code was added to check that the object sent with the post request to the end-point “api/v1/sessions” contained both the fields “user_name” and “password”. As shown in figure 4.4, if the client sends a request directly to the REST end-point without these exact fields the authentication service will send an error with an error message of “invalid body”.

No code was added to the GraphQL API gateway to inform the client of an invalid request body, although as shown in figure



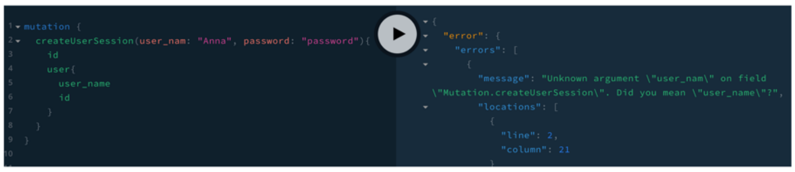
```

{
    "user_nae": "Anna",
    "password": "password"
}

```

```
{
    "message": "Invalid body!"
}
```

Figure 4.4: Error message using REST.



```
mutation {
  createUserSession(user_nam: "Anna", password: "password"){
    id
    user{
      user_name
      id
    }
  }
}

```

```
{
  "error": {
    "errors": [
      {
        "message": "Unknown argument \"user_nam\" on field
\"Mutation.createUserSession\". Did you mean \"user_name\"?",
        "locations": [
          {
            "line": 2,
            "column": 21
```

Figure 4.5: Error message using GraphQL.

4.5 the client, when making this kind of error, is presented with a detailed error message automatically, specifying exactly where things went wrong.

5 Discussion

This chapter contains a further discussion about the significance of each of the results. Finally, a reflection over the choice of method and a discussion about what can be done for further research will be presented.

GraphQL set up

A conclusion to be made from setting up GraphQL for the web application is that a disadvantage with GraphQL could appear in a small-scale project. If an application only needs their back-end to allow for some mutations then GraphQL will involve implementing features that are not intended to be used. Having code in a project that does not add anything to the rest of the program will most likely over-complicate the project and be misleading for anyone trying to understand and work on it.

The time spent on the implementation with GraphQL could also be a disadvantage for a project with a tight deadline. It is most likely of great value for a company that the end-users will like their products. If GraphQL does not add, in the perspective of the end-users, a valuable feature, it might not be beneficial delaying the release of a product because of it.

Introspection

With the introspective feature, little to no knowledge of the back-end is needed to start sending data to and requesting data from the services. Client-side developers can easily find out what kinds of data requests are available and what type of data is returned from making them.

For server-side development this feature acts as documentation for the API. The GraphQL API gateway in this sense can be thought of as a self-documented whereas with the REST implementation the server-side would need to provide some extra documentation. With this automatically generated documentation, GraphQL servers will have a somewhat consistent documentation. It is beneficial to have a consistent documentation for a back-end API. It could for example facilitate for front-end developers by abstracting the back-end implementation, in other words front-end developers do not need to know details about the implementation to start sending requests. Back-end developers can also spend less time writing commentary.

Data requests

The GraphQL implementation reduced the amount of data being passed from the server- to client-side. When the client-side execute a query for when a user wants to log in, there is no over-fetching of data. Nothing needs to be done to the response object to be in the desired format. This is beneficial since it simplifies the implementation, making it easier to understand. The query is in the same format as the result from executing it against the API gateway. In the same situation the client-side avoided under-fetching of data. With REST one separate API call did not result in the desired data, making it so that the client needed to make an additional request. With GraphQL only a single API call was made to fetch the data.

It is hard to make any firm conclusions from the performance tests made for this study. The result showed that the REST implementation reached the highest loading time but also had overall lower results than the GraphQL implementation. To get a better idea of how the two implementations affected the website performance, additional tests need to be done.

It is also hard to make any conclusions to whether the application performance benefited or not from having no over- and under-fetching of data. This because the back-end did not remove the RESTful implementation when implementing GraphQL, rather an additional service needs to be up and running in order for the client-side to execute queries and mutations with GraphQL. This most likely effected the result of the performance tests. It is most likely that the result had been different if an API gateway had not been used and instead the RESTful architecture was completely exchanged with GraphQL. Testing the performance when hosting the server-side on a more optimised web server, as supposed to running everything locally, would also most likely give different results.

API changes

For a small-scale project that has only one client that communicates with a server it might not be too much of an issue that changes made for the server API effects the runtime of the client. If however a back-end service is being used by multiple clients it will most likely be problematic that changes to the API could cause these clients to crash.

With the REST architecture, a possible solution to this is versioning, keeping old end-points up and running and creating new end-points that are updated with the newest changes. In this way clients can be made aware that certain versions are being deprecated. Clients can migrate to the newer version, making it "safer" to gradually remove older end-points.

An advantage with GraphQL is there is no need for versioning. Instead of having old end-points up and running, the fields, that are about to be deprecated because of the new changes, can be kept. This can be beneficial for saving up on resources and also for developers to spend less time versioning the API.

Error handling

With REST it is possible to generate custom errors and messages. As the example in the result chapter shows, we can generate an error from the authentication service if the log-in form did not contain the excepted fields. An advantage with GraphQL was that if the mutation to create a session does not contain the expected fields, a detailed error message will be generated automatically. GraphQL points out exactly which part of the mutation was at fault. Although it's possible to write code that will generate similar detailed errors, having it automatically generated is, again, beneficial since it means less time spend on writing them and they are consistent throughout the application.

Choice of method

The method chosen to answer the research question, made it somewhat difficult to test whether the performance of such a small web application would benefit from using GraphQL. This because the additional service, the API gateway was used to handle requests made with GraphQL. To get a better idea of the performance was affected, it could be easier to completely remove the REST architecture and have GraphQL as a single end-point for each of the back-end services. In this way the server-side can have resolvers that populates the fields in the request with data from the database, which reduces the overall number of requests.

The choices of web frameworks and language also had an impact on the complexity of setting up GraphQL in the project. Since there are different implementations for different frameworks and languages, further research about how GraphQL can be integrated with the chosen web tools needs to be done in order to determine the complexity level.

Further research

In order to further research the pros and cons of GraphQL, an implementation involving caching can be done. It can also be of interest to investigating how uploading of files are supported by the Apollo framework and how this differs from uploading files using the REST architecture. File uploads are not included in the GraphQL specification and it is mentioned in articles that this is one of the larger drawbacks to GraphQL (3).

Another topic for research is how the two implementations would be affected by having heavy traffic, in other words multiple requests. This was not done for this study since the application was only tested to run on a local computer. The results will depend on where the website is hosted and there are tools for testing a website once it can be accessed on a web server.

6 Conclusions

It was found that a disadvantage with GraphQL for a small-scale web application was increased complexity of the implementation and steeper learning curve. The complexity of setting up GraphQL also depends on which programming languages and web framework being used.

Advantages where also found. When considering implementing GraphQL for a simple web application that communicates with a small scale back-end server, it is a good idea to weight these advantages against the possible disadvantages. Depending on how simple the web application is, in other words how many features and resources it consists of, these advantages will be more or less effective.

Advantages found using GraphQL where:

- The ability for the client-side to check what types of data are available on the server-side. This is useful when front-end developers are not too familiar with the back-end implementation.
- No over- or under-fetching of data. The front-end is in control of what is returned from the server-side and related data can be fetched with a single API call.
- There is no need for versioning. Changes made to the back-end API does not affect the client-side in the same way as they do using the REST architecture. This means saving up on resources and time.
- More auto generated errors and detailed error messages.

Bibliography

1. Craig Buckler. *What Is a REST API?* https://www.sitepoint.com/developers-rest-api/. (Online; accessed 09-Dec-2020).

2. Lee Byron. *GraphQL: A data query language*. https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/. (Online; accessed 09-Dec-2020).

3. *GraphQL: Core Features, Architecture, Pros and Cons*. https://www.altexsoft.com/blog/engineering/graphql-core-features-architecture-pros-and-cons/. (Online; accessed 09-Dec-2020).

4. *Who's using GraphQL?* https://graphql.org/users/. (Online; accessed 09-Dec-2020).

5. Joshua Weinstein. *Client-Side vs Server-Side Web Development*. https://careerkarma.com/blog/client-vs-server-side-development/. (Online; accessed 09-Dec-2020).

6. Thomas Bush. *What Is an API Gateway?* https://nordicapis.com/what-is-an-api-gateway/. (Online; accessed 09-Dec-2020).

7. Eve Porcello and Alex Banks. *Learning GraphQL: Declarative data fetching for modern web apps*. O'Reilly Media, Inc., 2018.

8. swapi-graphql. https://github.com/graphql/swapi-graphql. (Online; accessed 09-Dec-2020).

9. *Express/Node introduction*. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express_Nodejs/Introduction. (Online; accessed 09-Dec-2020).

10. https://reactjs.org. (Online; accessed 09-Dec-2020).

11. *Introduction to Apollo Server*. https://www.apollographql.com/docs/apollo-server/. (Online; accessed 09-Dec-2020).

12. *Introduction to Apollo Client*. https://www.apollographql.com/docs/react/. (Online; accessed 09-Dec-2020).