jamk.fi

The Web service development with React, GraphQL and Apollo

Dmitry Sklyarov

Bachelor's thesis
October 2020
Information and Communications Technology
Degree Programme in Media Engineering

jamk.fi

Description

Author(s) Sklyarov, Dmitry	Type of publication Bachelor's thesis	Date October 2020
		Language of publication English
	Number of pages 110	Permission for web publication: x
Title of publication **The Web service development with React, GraphQL and Apollo**		
Degree programme Media Engineering		
Supervisor Rantala, Ari		
Assigned by Movya Oy		

Abstract

The goal of the project, assigned by Movya Oy, was to develop a Web service that facilitates the creation of business proposals and subsequent contracts between the company and the customer.

An essential aspect of software development was using a modern server, and client technologies React, GraphQL, and Apollo.

The various stages of developing a web service included database design and management, responsive application design building, and implementing access control, pagination, sorting, filtering large amounts of data, grouping items, editing documents, etc. The thesis covers both the most critical aspects of developing these features, including the theoretical part before.

The main result of the thesis was the product itself - the Web service called *Moffers*. In addition to this specific result, a lot of useful experience was gained, as well as knowledge of modern technologies and methods in JavaScript programming.

Keywords/tags (subjects)
Web service, JavaScript, React, GraphQL, Apollo, Ant Design, Mongoose, MongoDB

Miscellaneous (Confidential information)

jamk.fi

Kuvailulehti

Tekijä(t) Sklyarov, Dmitry	Julkaisun laji Opinnäytetyö, AMK	Päivämäärä Lokakuu 2020
		Julkaisun kieli Englanti
	Sivumäärä 110	Verkkojulkaisulupa myönnetty: x

Työn nimi
The Web service development with React, GraphQL and Apollo

Tutkinto-ohjelma
Mediatekniikka

Työn ohjaaja(t)
Rantala, Ari

Toimeksiantaja(t)
Movya Oy

Tiivistelmä

Opinnäytetyön toimeksiantajana oli Movya Oy. Opinnäytetyön tavoitteena oli kehittää verkkopalvelu, joka auttaa myyntitiimiä arvioimaan ja luomaan räätälöityjä tarjouksia nopeasti.

Oleellinen osa ohjelmistokehitystä oli modernien asiakastekniikoiden kuten React, GraphQL ja Apollo käyttäminen.

Verkkopalvelun kehittämisen eri vaiheisiin sisältyi mm. tietokannan suunnittelu, pääsynvalvonta, suurten tietomäärien suodattaminen, kohteiden ryhmittely ja asiakirjojen muokkaaminen.

Opinnäytetyö kattaa laajan tietoperustan, kriittisen arvioinnin valituille menetelmille ja ominaisuuksien kehittämiselle.

Opinnäytetyön päätulos oli itse tuote - verkkopalvelu nimeltä Moffers. Tämän nimenomaisen tuloksen lisäksi saatiin paljon hyödyllistä kokemusta sekä tietoa nykyaikaisista tekniikoista ja menetelmistä JavaScript-ohjelmoinnissa. Toimeksiantaja oli erittäin tyytyväinen tuloksiin.

Avainsanat (asiasanat)
Web service, JavaScript, React, GraphQL, Apollo, Ant Design, Mongoose, MongoDB

Muut tiedot (salassa pidettävät liitteet)

Contents

Figures

Terminology

API

Application Programming Interface is a particular interface of a program or application (class and procedure libraries), with which one program/application can interact with another.

Babel

Babel is a tool used to convert modern JavaScript (ES2015) code into a backward-compatible version in current and older browsers or environments.

CLI

A command-line interface is a text-based interface used for entering commands.

CPQ

Configure, Price, Quote is a system that helps the sales team to estimate and create customized quotes fast.

CRUD

CRUD is an acronym that stands for four primary functions used when interacting with databases: *create*, *read*, *update,* and *delete*.

DOM

Document Object Model is a programming interface that defines HTML elements as objects, properties of all HTML elements, methods to access all HTML elements, and events for all HTML elements.

Express

Express is a lightweight Node.js web application server framework that provides features for building web and mobile applications.

FQL

Facebook Query Language is a query language that allows querying Facebook user data by using a SQL-style interface.

GIT

Git is an open-source version control system designed for fast and efficient development.

GraphiQL

GraphiQL is an IDE for interacting with the GraphQL API that supports syntax highlighting, code completion, error warnings, and viewing query results.

GraphQL Playground

GraphQL Playground is an in-browser GraphQL IDE based on GraphiQL with several advanced features such as automatic schema reloading, support for GraphQL Subscriptions, query history, and HTTP headers configuration.

Hook

A hook is a particular function that allows manipulating to React state and lifecycle from functional components.

HOC

A higher-order component is a function that implements one of the more advanced ways to reuse logic by getting a component and returning a new component.

HTTP

Hypertext Transfer Protocol is an application-level protocol for transferring arbitrary data based on the "client-server" technology.

HTTPS

Hypertext Transfer Protocol Secure is an extension of the HTTP protocol to support encryption for increased security.

IDE

Integrated Development Environment is a software system used by programmers to develop software.

JavaScript

JavaScript is an object-oriented programming language with a prototypal inheritance that creates dynamically updating content, mainly used in web development.

JSON

JavaScript Object Notation is a JavaScript-based textual data exchange format commonly used for transferring data in web applications as one of two structures: a set of key/value pairs or an ordered set of values.

MongoDB

MongoDB is an open-source document-based database management system that does not require a table schema description and uses JSON-like documents with optional schemas.

Mongoose

Mongoose is an object data modeling library that provides a vast set of functionalities for creating and working with strongly typed schemas conforming to the MongoDB document.

MPA

A multi-page application is a web application that includes more than one page and requires an entire page to be reloaded even with minor content changes.

MVC

Model-View-Controller is an architectural pattern that separates an application into three main logical components (Model, View, and Controller) so that each component can be modified independently.

Node.js

Node.js is an open-source server environment that executes JavaScript code outside of a web browser.

NPM

Node Package Manager is a package manager with Node.js, used by JavaScript developers to exchange tools, install various modules, and manage their dependencies.

NPX

NPX is a tool designed to help standardize the NPM package experience, making it easy to install and manage dependencies located in the registry, as well as using CLI utilities and other executables.

REST

Representational State Transfer is an HTTP resource-based architectural style/model whose principles govern HTTP and URLs.

RESTful

RESTful is a term used for REST-based Web services.

SDL

Schema Definition Language is a part of the GraphQL specification, which includes human-readable syntax for defining a schema and storing it as a string.

SOAP

Simple Object Access Protocol is a structured messaging protocol in a distributed computing environment used to exchange arbitrary messages in XML format and is an extension of the XML-RPC protocol.

SPA

A single-page application is a web application or website that uses a single HTML document as a wrapper for all web pages and organizes user interaction through dynamically loaded HTML, CSS, JavaScript, usually through AJAX.

SQL

Structured Query Language is a programming language used to access and manage a relational database.

UI

The user interface is an interface that provides information transfer between a human user and software and hardware components of a computer system.

URL

Uniform Resource Locator is a standard for representing the address of a resource on the Internet, including information of protocol, domain name, file subdirectory, and the requested resource's file name.

UX

User Experience determines the impression a person gets from interacting with a digital product.

XML

Extensible Markup Language is a format for describing data as a set of tags, nested tags, and attributes for storing and transmitting data.

XML-RPC

XML Remote Procedure Call is a remote procedure call standard/protocol that uses XML to encode its messages and HTTP as the transport mechanism.

Webpack

Webpack is a module builder that allows compiling JavaScript modules into a single JavaScript file.

WebSocket

WebSocket is a communication protocol over a TCP connection, designed to exchange messages between a browser and a web server in real-time.

Yarn

Yarn is a package manager that is a replacement for package management in web projects with some features missing from the NPM command-line interface.

1 Introduction

1.1 Client presentation

Movya is an awarded digital agency offering digital media services to industrial forerunners. Movya produces visualized multilingual content, builds up digital service concepts, delivers exceptional results for customers, and throws a deep understanding of their needs, customers, and target audience, and then combines it with an insight-driven strategy and creativity with the highest level of execution. Movya, as a name, is connected to high-quality and cost-efficient productions. (Rautvuori 2019.)

Movya's clients are international industrial companies such as ABB, Airbus, Daikin, Ensto, JYU, Kemppi, Kone, Metso, Outotec, Paroc, Ruukki, Thermo, Valmet Automotive, Valmet, Valtra, to name a few. Movya gives its customers both a voice for their innovative products and services, as well as tools for active marketing and sales. Movya integrates the client's product planning, customer training, digital user and maintenance guidelines, and marketing into a uniformed competitive, profitable entity. (What we do n.d.)

1.2 Project and thesis objectives

Movya does hundreds of projects every year. It is essential for their business that the process of creating and managing quotes is simple and efficient. For that purpose, Movya is developing CPQ (Configure, Price, Quote system). At the time, most existing CPQ systems were aimed at large companies. Movya is looking for a simple, easy to use solution and customized for the company's needs.

The project's main objective was to create the system's basic functionalities, which formed a Web service basis. An essential aspect of software development was the requirement to use a combination of modern server and client-side technologies. All technologies were selected by the customer and presented in the background research of the work.

The main result of the thesis was the product itself - the Web service called *Moffers*, which stands for *Movya Offers*. The purpose of the written part of the thesis work was to study modern software development and the review of problems occurring in the process of development and ways of their solution.

2 Web services

2.1 What are Web services?

First of all, Web services is technology. Like any other technology, it has a reasonably well-defined application environment. If Web services are looked at in the context of the network protocol stack, it can be seen that, in the classical case, this is nothing more than another add-on on top of the HTTP protocol. (McIlraith, Son, & Zeng 2001; Web services in theory and practice for beginners 2019.)

On the other hand, the Internet can be hypothetically divided into several technological layers. At least two conceptual types of applications can be distinguished: computing nodes that implement non-trivial functions and applied web resources. Also, the latter is often related to the services of the former. But the Internet itself is not homogeneous. There are different applications on different network nodes. They work on various hardware and software platforms, use different technologies and languages. Web services were invented to tie it all together and enable some applications to communicate with others. They provide interfaces for exchanging data between different applications, used in different languages and distributed across other nodes. (Richardson & Ruby 2008; Web services in theory and practice for beginners 2019.)

2.2 Difference between Web services and APIs

Both APIs and Web services are communication tools. A particular case of such communication as client-server interaction is shown in Figure 1.

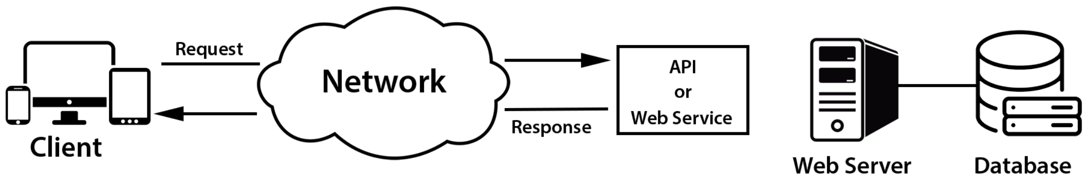


Figure 1. Client and server communication

Web services are often confused with APIs. Despite the similarity in purpose, there are some minor differences between them: while a Web service facilitates communication between two devices over a network, an API acts as an interface between two different applications to communicate with each other. The Web service is, as it were, a particular case of the API: it works over the HTTP protocol (network), supports JSON data and XML format, is limited to REST, SOAP, and XML-RPC for communication. (Samer 2017.)

3 React

In recent years, the SPA has become increasingly popular in modern web development. Unlike MPA, they allow smoothly and quickly changing web content without reloading the page, thanks to various JavaScript libraries and frameworks. One of these libraries is React - an open-source JavaScript library for building user interfaces. (ReactJS by Example-Building Modern Web Applications with React 2016.)

React was created by Facebook in 2011, and in May 2013, the project's source code was published on the JSConf US website. (React.js history, 2019.)

React was developed to solve problems related to developing complex user interfaces from the View layer in any MVC frameworks. (ReactJS by Example-Building Modern Web Applications with React 2016.)

3.1 DOM and Virtual DOM

The entire structure of a web page is represented using the DOM as HTML elements that can be manipulated (change, delete or add new ones). JavaScript is used to interact with DOM. However, manipulating HTML elements with JavaScript causes performance degradation, significantly when changing a large number of elements, which will inevitably affect the user experience. The concept of virtual DOM was created to solve the performance problem. Virtual DOM is a lightweight copy of the regular DOM, and the distinctive feature of React is that this library works with virtual DOM, and not the regular one. (Čto takoe Reakt. Pervoe priloženie 2017.)

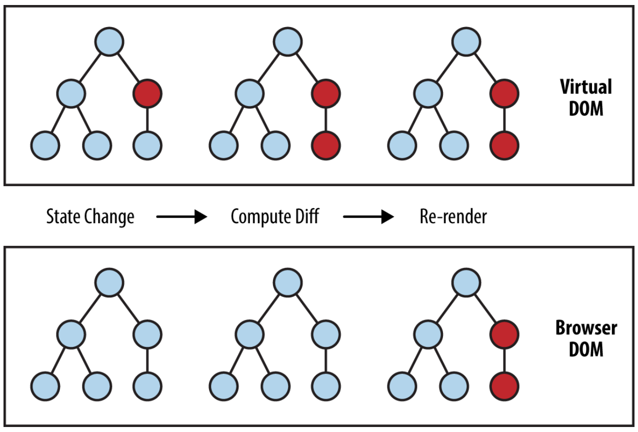


Figure 2. Virtual and real DOM comparison (Hamedani 2018a)

If an application needs information about the state of elements, then virtual DOM is accessed. If elements of a web page need to be changed, the changes are first made to the virtual DOM. Then the new state of the virtual DOM is compared with the current state. And if these states are different, React finds the minimum amount of manipulation necessary before updating the real DOM to a new state and performs them (Figure 2). As a result, this interaction scheme with a web page's elements works much faster and more efficiently than with the DOM directly. (Aggarwal 2018.)

3.2 JSX

JSX is a JavaScript language extension representing a way to describe visual code through a combination of JavaScript code and HTML markup. Figure 3 shows how JSX allows any valid JavaScript expression to be included in elements using parentheses. JSX expressions are transpiled into pure JavaScript, allowing JSX elements, for example, inside loops or as variables. (Znakomstvo s JSX n.d.)

```
const name = 'Dmitry'
const element = <h1>Hello, {name}!</h1>
```

Figure 3. JSX syntax

React can be used without JSX, but JSX is the recommended way to create a UI because it makes it more descriptive and allows React to display more useful error messages. (Znakomstvo s JSX n.d.)

3.3 Rendering elements

The smallest blocks of a React application are elements. The definition of the simplest element is shown in Figure 3. Elements in React are regular JavaScript objects that are faster to work with than common elements on a web page. (Rendering èlementov n.d.).

React uses *ReactDOM.render()* method to render elements. This method takes three parameters: the element to render, DOM-element to add the element to, the optional callback function (Figure 4). (Rendering èlementov n.d.)

```
const element = <h1>Hello, world</h1>
ReactDOM.render(element, document.getElementById('root'))
```

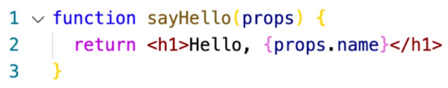
Figure 4. *ReactDOM.render()* method

React elements are immutable. After an element is created, it is not possible to change its attributes or child elements. And the only way to change the interface defined in an element is to create a new element and pass it to *ReactDOM.render()* function. (Rendering èlementov n.d.)

3.4 Components and props

React allows creating elements that embed in a web page. However, this is poorly suited for creating complex HTML markup. React element objects are difficult to reuse in similar situations and harder to maintain. React uses components to solve this problem. Components are easier to update and reuse. Components are similar to JavaScript functions. They store state through properties and return React elements, which then appear on the web page. (Komponenty 2017.)

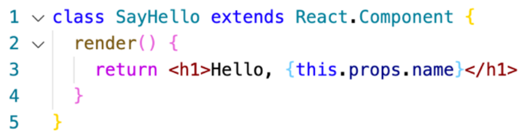
Components can be defined in two equivalent ways: functional and class. A functional component is a regular JavaScript function that receives data as an object called "props" and returns a React element (Figure 5). (Komponenty i propsy n.d.)



```
function sayHello(props) {
  return <h1>Hello, {props.name}</h1>
}
```

Figure 5. Functional component

The definition of a class component assumes the use of ES6 classes and the *render()* method's mandatory use, which returns the generated element in JSX (Figure 6). (Komponenty i propsy n.d.)



```
class SayHello extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

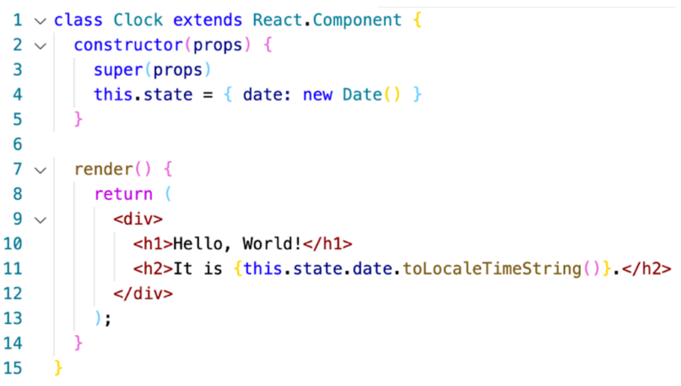
Figure 6. Class component

Props represent a collection of values that are associated with a component. These values allow creating dynamic components that do not depend on hard-coded static data. (Props 2017.)

3.5 State

React components can have a state. A state is an object that describes the local state of a component and gives it interactivity. Unlike props (that represent input data) passed to a component from outside, the state stores objects created directly in the component, entirely depending on it, and can be changed. (State 2017.)

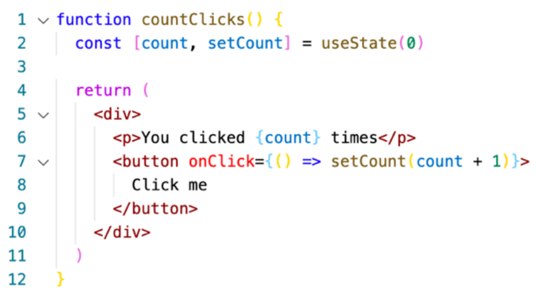
The only place where the state is set in a class component is in the class constructor. Figure 7 shows how a component's state is first initialized in the constructor function and then implemented inside the class component. Updating the local state triggers reactive rendering by re-rendering the component and its children. (Sostoyanie i žiznennyj cikl n.d.)



```
class Clock extends React.Component {
  constructor(props) {
    super(props)
    this.state = { date: new Date() }
  }

  render() {
    return (
      <div>
        <h1>Hello, World!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Figure 7. Class component initialization and using with state

In functional components, the state is set using the *useState()* hook in the function body at the top level (Figure 8). (Kratkij obzor hukov n.d.)



```
function countClicks() {
  const [count, setCount] = useState(0)

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  )
}
```

Figure 8. Functional component initialization and using with state

3.6 Component lifecycle

In the working process, the component goes through several lifecycle phases: mounting, updating and unmounting. At each of the phases, a specific function is called in which it possible to define any actions. The most common lifecycle functions are shown in bold in Figure 9. (Hamedani 2018b.)

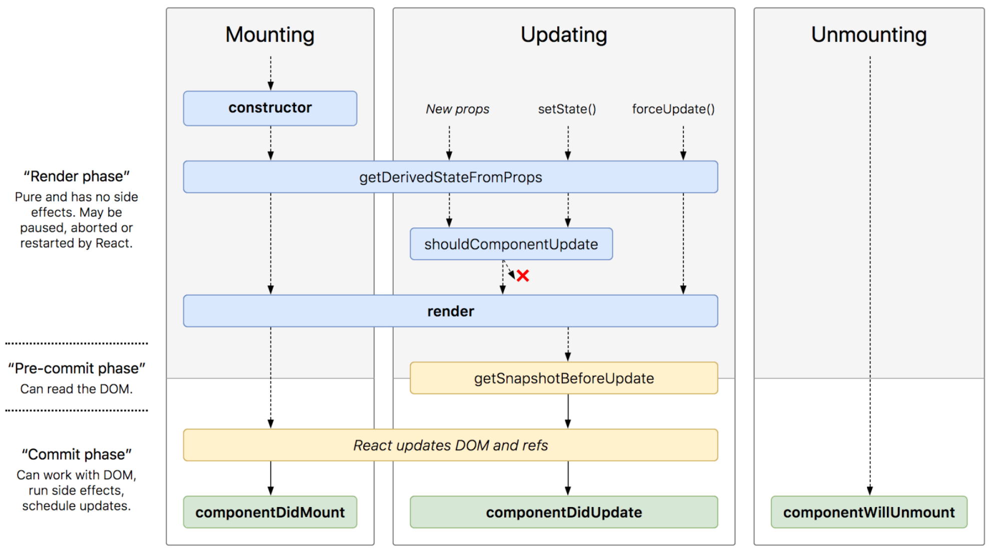
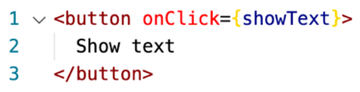


Figure 9. Component lifecycle and lifecycle functions (Hamedani 2018b)

For all class components, *the render()* function is required. It is launched at the stage of the initial mounting of the component, as well as during the update. Immediately after the component is mounted, the *componentDidMount()* function is run, in which it is recommended to handle side effects, for example, make API calls. As soon as changes are detected in the DOM structure (as a result of props or state changes), *the componentDidUpdate()* function runs. When the component finally exits, it is time to do the cleanup actions using the *componentWillUnmount()* function (Hamedani 2018b.)

3.7 Event handling

The main differences between event handling in React elements and DOM elements are that events in React use camelCase syntax, and with JSX, a function is passed as an event handler instead of a string. Figure 10 shows an example of a handling click event.



```
<button onClick={showText}>
  Show text
</button>
```

Figure 10. Handling *onClick()* event

3.8 Context API

React passes data from parent to child components via props from top to bottom. As the application grows, the component tree becomes layered, and data passing becomes cumbersome and painful. This problem of passing data along the component chain is called "prop drilling" (Figure 11). (Spukas 2019.)

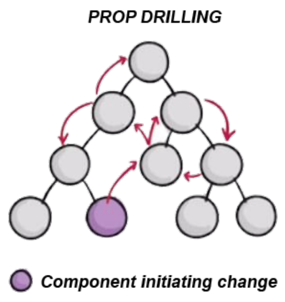


Figure 11. Prop drilling

Context API solves the prop drilling problem by passing data to all components without explicitly passing it through each level of the tree (Figure 12). (Kontekst n.d.)

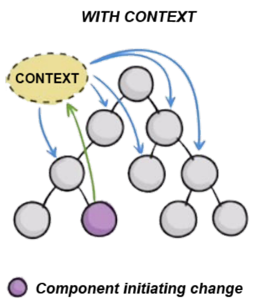
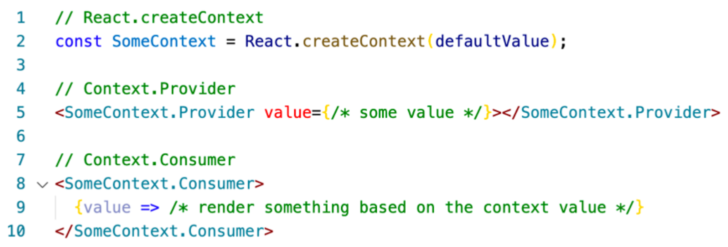


Figure 12. Context API

React provides ready-made interfaces for creating and using context. The context object is created using the *createContext()* function, into which a default value can be passed if needed. Each context created includes a Provider component that allows the context to be distributed among its child Consumer components. (Kontekst n.d.)

A short example of creating and using a context using the above functions is shown in Figure 13



```
// React.createContext
const SomeContext = React.createContext(defaultValue);

// Context.Provider
<SomeContext.Provider value={/* some value */}></SomeContext.Provider>

// Context.Consumer
<SomeContext.Consumer>
  {value => /* render something based on the context value */}
</SomeContext.Consumer>
```

Figure 13. Context creating and using

4 GraphQL

In 2012, because the performance of Facebook's mobile applications that used RESTful and FQL was questionable, applications often crashed. The company's engineers decided to improve the way data is passed and created a query language for APIs. This made describing the capabilities and requirements of data models for client-server applications possible. In July 2015, programmers released the initial GraphQL specification and reference implementation of GraphQL in JavaScript called graphql.js. In September 2016, GraphQL specification passed the technical preview stage. This meant that it was officially ready for release, although Facebook had used it for many years. (Bènks & Porsello 2019.)

When GraphQL specification was released, some positioned it as a replacement for REST. It solves REST architecture's common issues such as over-fetching, under-fetching, endpoint management, versioning, schema definition, documentation, and subscriptions. (Bènks & Porsello 2019; Newby 2019.)

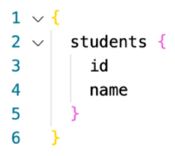
4.1 Solving REST problems using GraphQL

4.1.1 Over-fetching

With a REST request, there is no easy way to get a specific limited set of fields. The client always receives all data from the resource. And this can lead to over-fetching, which means fetching too much data that is often not used. (Bènks & Porsello 2019.)

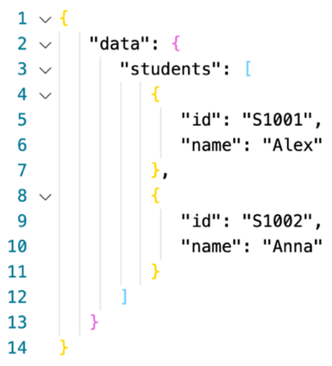
Using GraphQL, it is possible to "ask for what you need, get exactly that". (The GraphQL Foundation 2020a.)

This issue can be demonstrated with the following example. Suppose there is a *student* object with fields *id*, *name*, *email*, *phone*. Web application only needs to get the *id* and *name* fields. GraphQL query shown in Figure 14 will return the required values only for *id* and *name* fields and will not retrieve values for the other fields of the object (Figure 15).



```
{
  students {
    id
    name
  }
}
```

Figure 14. GraphQL query with required data



```
{
  "data": {
    "students": [
      {
        "id": "S1001",
        "name": "Alex"
      },
      {
        "id": "S1002",
        "name": "Anna"
      }
    ]
  }
}
```

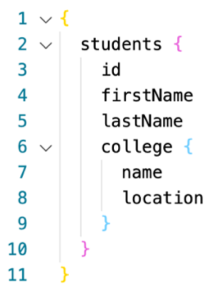
Figure 15. Server response with required object data

4.1.2 Under-fetching

Sometimes the opposite problem occurs with REST: to fetch two different resources, two separate requests have to be sent to the server. (Bènks & Porsello 2019.)

GraphQL allows getting "many resources in a single request" to solve the under-fetching issue. (The GraphQL Foundation 2020a.)

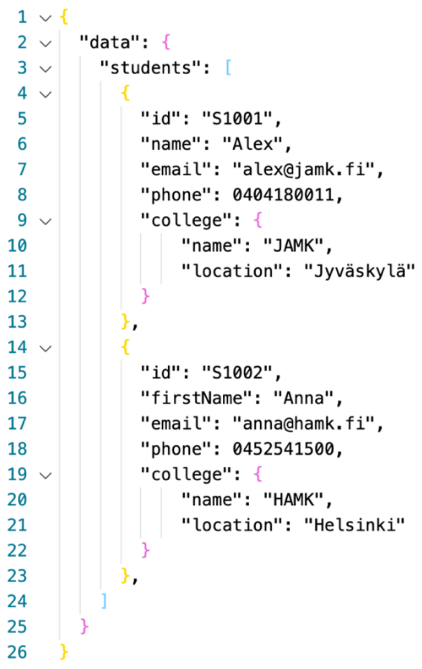
Suppose there is also an object *college*, which has attributes: *name* and *location*. Object *student* has a relation with the object *college*. Using REST to get students and their college data, two separate requests have to be performed to the server, such as "/api/students" and "/api/colleges" endpoints. However, an application can retrieve student and college objects data in a single request using GraphQL, shown in Figure 16.



```
{
  students {
    id
    firstName
    lastName
    college {
      name
      location
    }
  }
}
```

Figure 16. Single GraphQL query to get multiple objects data

An example of the result of such a query is shown in Figure 17.



```
{
  "data": {
    "students": [
      {
        "id": "S1001",
        "name": "Alex",
        "email": "alex@jamk.fi",
        "phone": 0404180011,
        "college": {
          "name": "JAMK",
          "location": "Jyväskylä"
        }
      },
      {
        "id": "S1002",
        "firstName": "Anna",
        "email": "anna@hamk.fi",
        "phone": 0452541500,
        "college": {
          "name": "HAMK",
          "location": "Helsinki"
        }
      },
    ]
  }
}
```

Figure 17. Server response with multiple objects data

4.1.3 Endpoint management

Unlike REST, where new endpoints are created as an application grows (and their number can quickly get out of hand), in GraphQL, a typical architecture includes one endpoint (Figure 18). A single endpoint can act as a gateway and manage multiple data sources, and a single endpoint simplifies data organization. (Bènks & Porsello 2019.)

GraphQL Endpoint http://localhost:4000/graphql

Figure 18. Querying data to "/graphql" endpoint in GraphiQL

4.1.4 Schema definition

"Describe what's possible with a type system". (The GraphQL Foundation 2020a.)

GraphQL is strongly typed, queries are based on fields and their associated data types. If there is a type mismatch in a GraphQL query, server applications return understandable and helpful error messages. It helps in debugging and error detection by client applications. GraphQL also provides client libraries that can help to reduce explicit data transformation and analysis. (*GraphQL – kratkoe rukovodstvo* 2019.)

An example of the *Student* and *College* data types is shown in Figure 19.

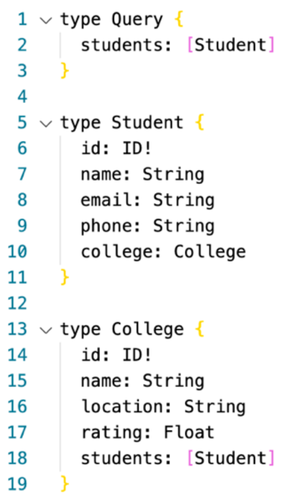
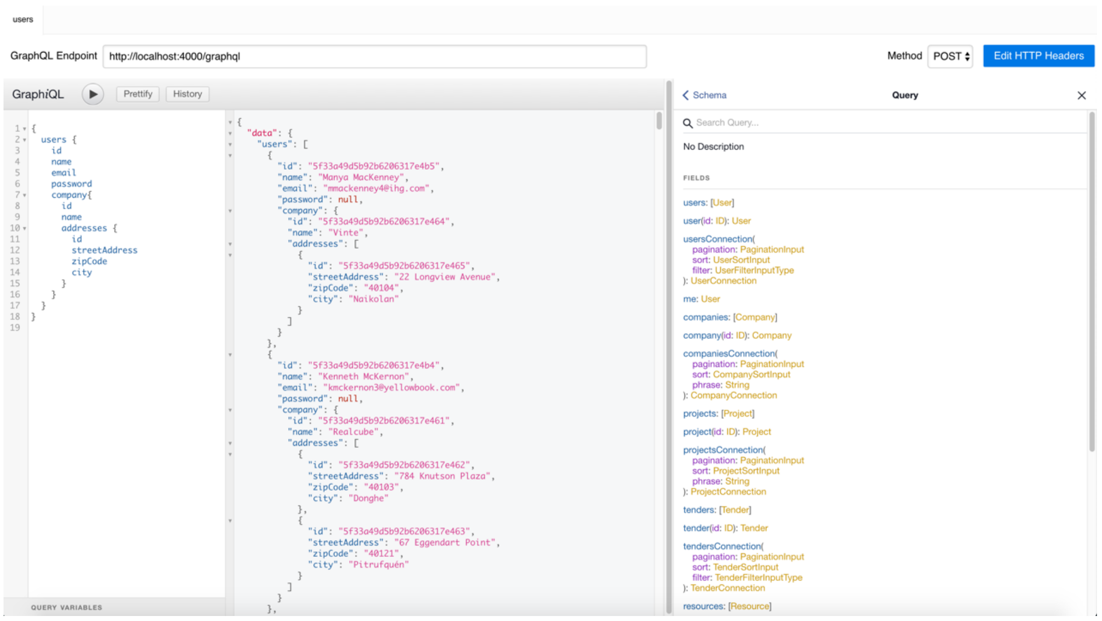

```
type Query {
  students: [Student]
}

type Student {
  id: ID!
  name: String
  email: String
  phone: String
  college: College
}

type College {
  id: ID!
  name: String
  location: String
  rating: Float
  students: [Student]
}
```

Figure 19. Data types used in GraphQL

4.1.5 Documentation

"Move faster with powerful developer tools". (The GraphQL Foundation 2020a.)

GraphQL makes it easy to document APIs without leaving the editor, using a strongly-typed schema as documentation. Figure 21 shows an example of using the Documentation Explorer in GraphiQL. (The GraphQL Foundation 2020a; Newby 2019.)



```
{
  users {
    id
    name
    email
    password
    company{
      id
      name
      addresses {
        id
        streetAddress
        zipCode
        city
      }
    }
  }
}
```

```
{
  "data": {
    "users": [
      {
        "id": "5f33a49d5b92b6206317e4b5",
        "name": "Manya MacKenney",
        "email": "mmackenney4@ihg.com",
        "password": null,
        "company": {
          "id": "5f33a49d5b92b6206317e464",
          "name": "Vinte",
          "addresses": [
            {
              "id": "5f33a49d5b92b6206317e465",
              "streetAddress": "22 Longview Avenue",
              "zipCode": "40104",
              "city": "Naikolan"
            }
          ]
        }
      },
      {
        "id": "5f33a49d5b92b6206317e4b4",
        "name": "Kenneth McKernon",
        "email": "kmckernon3@yellowbook.com",
        "password": null,
        "company": {
          "id": "5f33a49d5b92b6206317e461",
          "name": "Realcube",
          "addresses": [
            {
              "id": "5f33a49d5b92b6206317e462",
              "streetAddress": "784 Knutson Plaza",
              "zipCode": "40103",
              "city": "Donghe"
            },
            {
              "id": "5f33a49d5b92b6206317e463",
              "streetAddress": "67 Eggendart Point",
              "zipCode": "40121",
              "city": "Pitrufquén"
            }
          ]
        }
      },
```

< Schema   Query   ×

Search Query...

No Description

FIELDS

```
users: [User]
user(id: ID): User
usersConnection(
  pagination: PaginationInput
  sort: UserSortInput
  filter: UserFilterInputType
): UserConnection
me: User
companies: [Company]
company(id: ID): Company
companiesConnection(
  pagination: PaginationInput
  sort: CompanySortInput
  phrase: String
): CompanyConnection
projects: [Project]
project(id: ID): Project
projectsConnection(
  pagination: PaginationInput
  sort: ProjectSortInput
  phrase: String
): ProjectConnection
tenders: [Tender]
tender(id: ID): Tender
tendersConnection(
  pagination: PaginationInput
  sort: TenderSortInput
  filter: TenderFilterInputType
): TenderConnection
resources: [Resource]
```

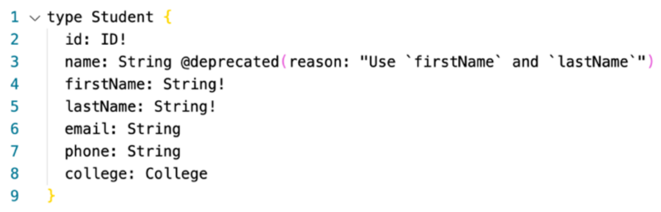
Figure 20. Documentation Explorer in GraphiQL

4.1.6 Versioning

"Evolve your API without versions" (The GraphQL Foundation 2020a.)

As an application develops and needs change, API also needs to evolve, changing its schema. In REST, it is quite normal to offer, in this case, several versions of the same API, for example, indicated at the endpoint as "/api/v1", "/api/v2". (Losoviz 2020.)

In GraphQL, it is possible to change APIs into deprecated fields at the field level (Figure 21). Therefore, when accessing a deprecated field, it receives a warning. After some time, the deprecated field can be excluded from the schema, and then no more clients will use it. This way, the GraphQL API can evolve without the need for versioning. (Wieruch 2018.)



```
type Student {
  id: ID!
  name: String @deprecated(reason: "Use `firstName` and `lastName`")
  firstName: String!
  lastName: String!
  email: String
  phone: String
  college: College
}
```

Figure 21. Using a deprecated directive in GraphQL

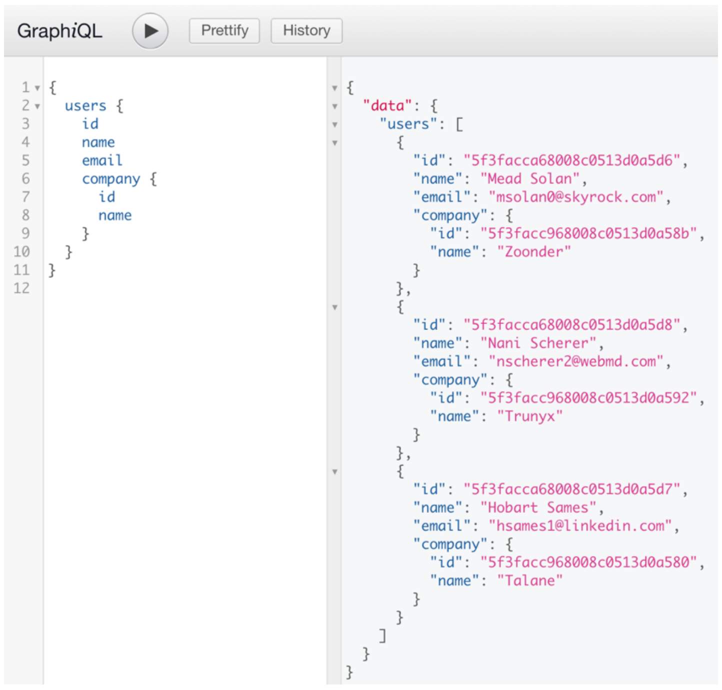
4.2 Queries

GraphQL implements ideas developed initially for queries against SQL databases: a GraphQL query can return related data. It is possible to use GraphQL queries to change or delete data. No wonder the abbreviation "QL" in their names means the same thing: query language. Despite this similarity, the GraphQL and SQL query languages are completely different. They have entirely different syntax and are designed for completely different environments: SQL queries are sent to the database, and GraphQL queries are sent to the API. (Bènks & Porsello 2019.)

GraphQL query is a simple string sent in the body of a POST request to a GraphQL endpoint. This line always looks the same and does not depend on the programming language used in the project. (Bènks & Porsello 2019.)

All GraphQL queries start at the root query, and what needs to be retrieved during the query is called a field. A query always has precisely the same shape as a result, so the client still gets what it expects, and the server knows exactly which fields are being requested. A query can be made using a shorthand syntax where the operation type and name are omitted. (The GraphQL Foundation 2020b).

Figure 20 shows an example of a root query in a shorthand syntax that asks for the field *users*, inside which a nested field *companies* is optionally defined.



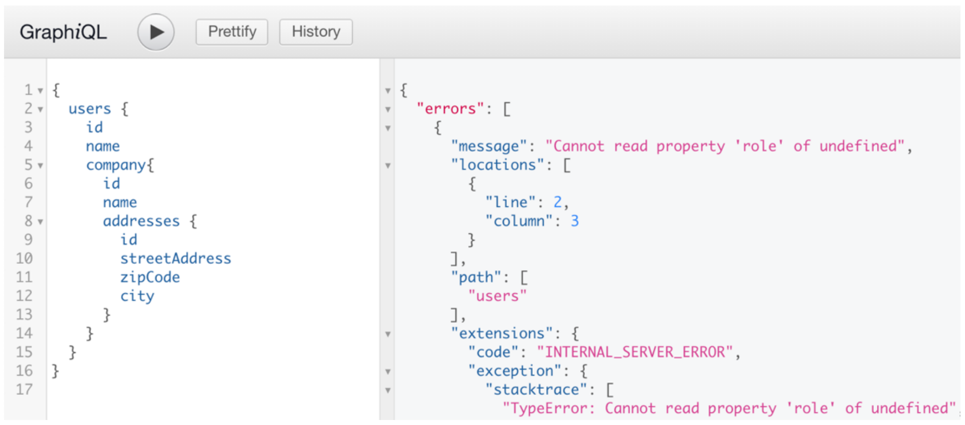
```
{
  users {
    id
    name
    email
    company {
      id
      name
    }
  }
}
```

```
{
  "data": {
    "users": [
      {
        "id": "5f3facca68008c0513d0a5d6",
        "name": "Mead Solan",
        "email": "msolan0@skyrock.com",
        "company": {
          "id": "5f3facc968008c0513d0a58b",
          "name": "Zoonder"
        }
      },
      {
        "id": "5f3facca68008c0513d0a5d8",
        "name": "Nani Scherer",
        "email": "nscherer2@webmd.com",
        "company": {
          "id": "5f3facc968008c0513d0a592",
          "name": "Trunyx"
        }
      },
      {
        "id": "5f3facca68008c0513d0a5d7",
        "name": "Hobart Sames",
        "email": "hsames1@linkedin.com",
        "company": {
          "id": "5f3facc968008c0513d0a580",
          "name": "Talane"
        }
      }
    ]
  }
}
```

Figure 22. GraphQL query and response data

GraphQL queries always return JSON data. If the query is successful, the JSON document displays the data in the *data* field, in case of an unsuccessful error in the *errors* field. (Bènks & Porsello 2019.)

An example of an authorization error is shown in Figure 23.



```
{
  users {
    id
    name
    company{
      id
      name
      addresses {
        id
        streetAddress
        zipCode
        city
      }
    }
  }
}
```

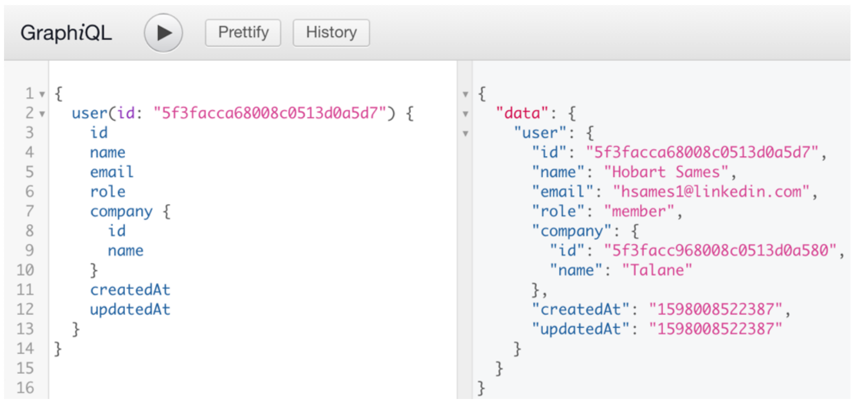
```
{
  "errors": [
    {
      "message": "Cannot read property 'role' of undefined",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "path": [
        "users"
      ],
      "extensions": {
        "code": "INTERNAL_SERVER_ERROR",
        "exception": {
          "stacktrace": [
            "TypeError: Cannot read property 'role' of undefined"
```

Figure 23. GraphQL query and response error

4.3 Arguments

GraphQL arguments are key-value pairs associated with the operation field and implemented to select data or filter the GraphQL operation results. (Bènks & Porsello 2019.)

Figure 24 shows an example of a query for a specific user using a known *id*.



```
{
  user(id: "5f3facca68008c0513d0a5d7") {
    id
    name
    email
    role
    company {
      id
      name
    }
    createdAt
    updatedAt
  }
}
```

```
{
  "data": {
    "user": {
      "id": "5f3facca68008c0513d0a5d7",
      "name": "Hobart Sames",
      "email": "hsames1@linkedin.com",
      "role": "member",
      "company": {
        "id": "5f3facc968008c0513d0a580",
        "name": "Talane"
      },
      "createdAt": "1598008522387",
      "updatedAt": "1598008522387"
    }
  }
}
```

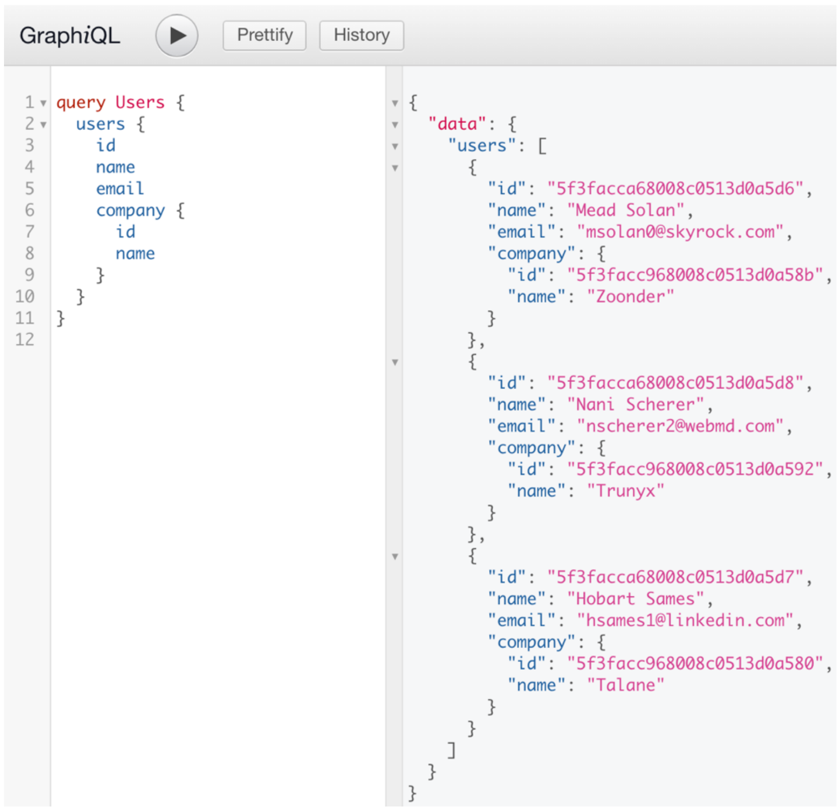
Figure 24. GraphQL query with an argument

In GraphQL, it is possible to pass arguments to fields and nested objects. This helps to avoid multiple round-trip cycles of data, tedious and cumbersome in the REST API. (Ravichandran 2019.)

4.4 Operation type and name

In real applications, it is useful to use the type and name of the operation when performing it. This makes the code easier to read and more comfortable to debug. (The GraphQL Foundation 2020b.)

An example of a query includes the keyword "query" as operation type and "users" as an operation name is shown in Figure 25.



```
query Users {
  users {
    id
    name
    email
    company {
      id
      name
    }
  }
}
```

```
{
  "data": {
    "users": [
      {
        "id": "5f3facca68008c0513d0a5d6",
        "name": "Mead Solan",
        "email": "msolan0@skyrock.com",
        "company": {
          "id": "5f3facc968008c0513d0a58b",
          "name": "Zoonder"
        }
      },
      {
        "id": "5f3facca68008c0513d0a5d8",
        "name": "Nani Scherer",
        "email": "nscherer2@webmd.com",
        "company": {
          "id": "5f3facc968008c0513d0a592",
          "name": "Trunyx"
        }
      },
      {
        "id": "5f3facca68008c0513d0a5d7",
        "name": "Hobart Sames",
        "email": "hsames1@linkedin.com",
        "company": {
          "id": "5f3facc968008c0513d0a580",
          "name": "Talane"
        }
      }
    ]
  }
}
```

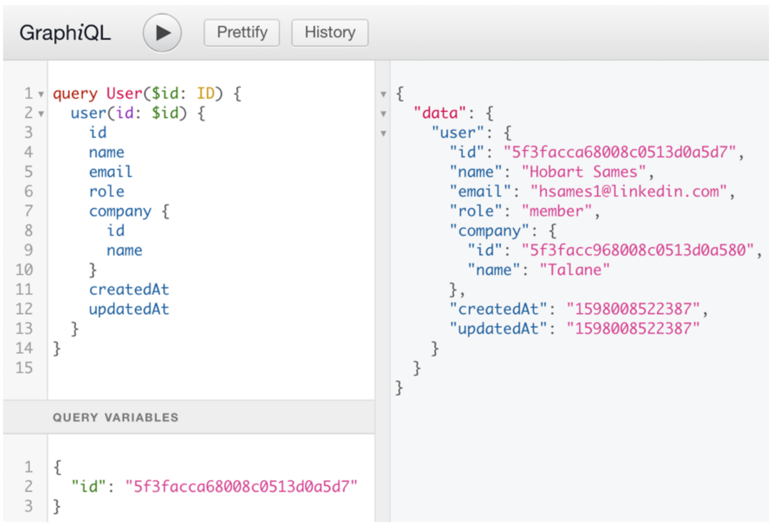
Figure 25. GraphQL query with the operation type and name

GraphQL uses the following types of operations: query, mutation, or subscription, each of which describes what type of operation is being performed. (The GraphQL Foundation 2020b.)

4.5 Variables

Since the arguments for fields are dynamic in most applications, it is not recommended to pass them directly in the query string. GraphQL offers to extract dynamic values from a query and pass them as variables. Variables in GraphQL use the "$" sign. Further, through the ":" sign, the data type of the variable is indicated, and the "!" sign can optionally be added, which suggests that the variable is required. (The GraphQL Foundation 2020b.)

Any GraphQL IDE has a "QUERY VARIABLES" window that specifies the values of variables in JSON format. An example of a query using variables is shown in Figure 26.



```
query User($id: ID) {
  user(id: $id) {
    id
    name
    email
    role
    company {
      id
      name
    }
    createdAt
    updatedAt
  }
}
```

```
{
  "id": "5f3facca68008c0513d0a5d7"
}
```

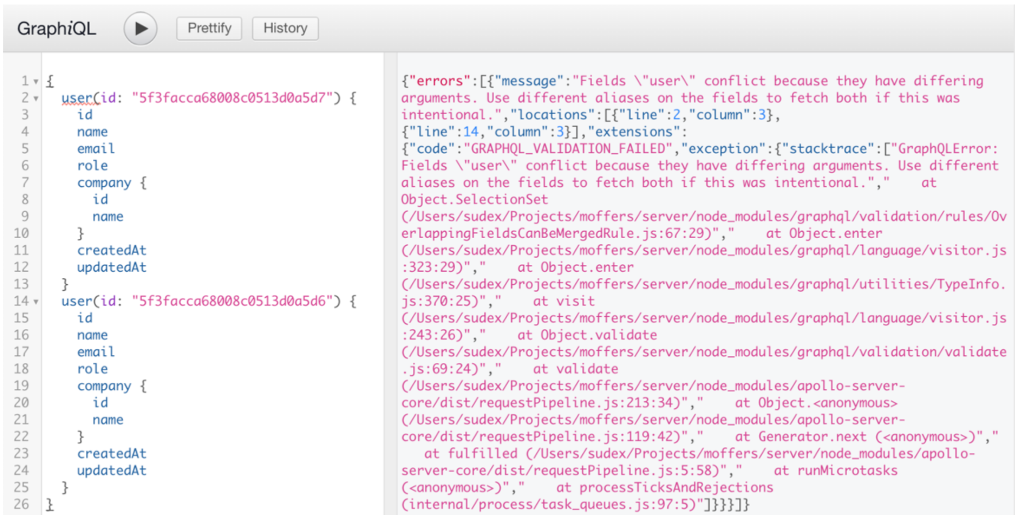
```
{
  "data": {
    "user": {
      "id": "5f3facca68008c0513d0a5d7",
      "name": "Hobart Sames",
      "email": "hsames1@linkedin.com",
      "role": "member",
      "company": {
        "id": "5f3facc968008c0513d0a580",
        "name": "Talane"
      },
      "createdAt": "1598008522387",
      "updatedAt": "1598008522387"
    }
  }
}
```

Figure 26. GraphQL query using variables

4.6 Aliases

In JSON, properties must be unique within a single object. (Osnovy GraphQL 2019.)

Figure 27 shows an example of an error query for an object with different arguments.



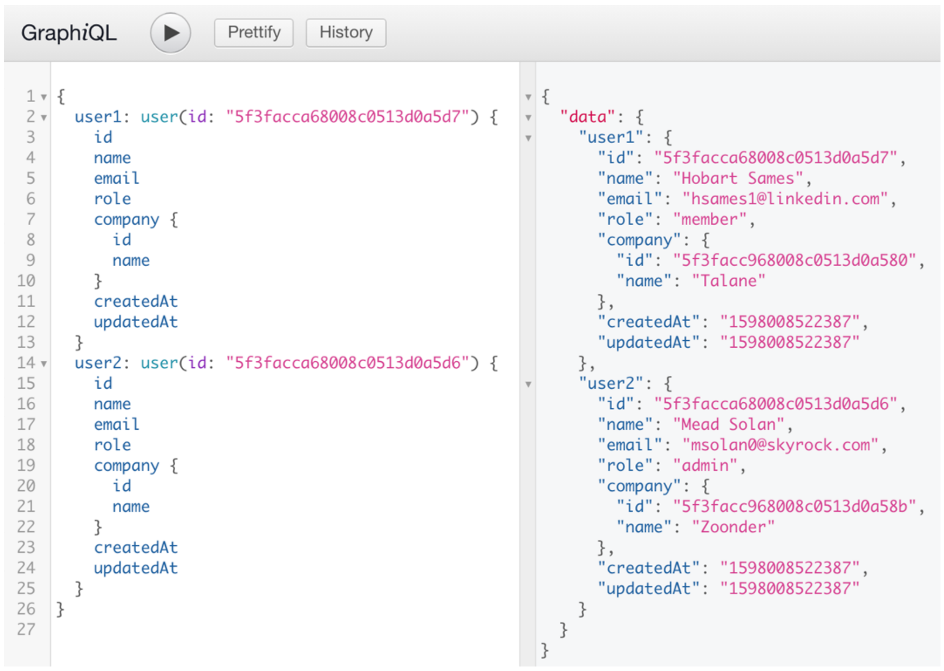
```
{
  user(id: "5f3facca68008c0513d0a5d7") {
    id
    name
    email
    role
    company {
      id
      name
    }
    createdAt
    updatedAt
  }
  user(id: "5f3facca68008c0513d0a5d6") {
    id
    name
    email
    role
    company {
      id
      name
    }
    createdAt
    updatedAt
  }
}
```

```
{"errors":[{"message":"Fields \"user\" conflict because they have differing
arguments. Use different aliases on the fields to fetch both if this was
intentional.","locations":[{"line":2,"column":3},
{"line":14,"column":3}],"extensions":
{"code":"GRAPHQL_VALIDATION_FAILED","exception":{"stacktrace":["GraphQLError:
Fields \"user\" conflict because they have differing arguments. Use different
aliases on the fields to fetch both if this was intentional.","    at
Object.SelectionSet
(/Users/sudex/Projects/moffers/server/node_modules/graphql/validation/rules/Ov
erlappingFieldsCanBeMergedRule.js:67:29)","    at Object.enter
(/Users/sudex/Projects/moffers/server/node_modules/graphql/language/visitor.js
:323:29)","    at Object.enter
(/Users/sudex/Projects/moffers/server/node_modules/graphql/utilities/TypeInfo.
js:370:25)","    at visit
(/Users/sudex/Projects/moffers/server/node_modules/graphql/language/visitor.js
:243:26)","    at Object.validate
(/Users/sudex/Projects/moffers/server/node_modules/graphql/validation/validate
.js:69:24)","    at validate
(/Users/sudex/Projects/moffers/server/node_modules/apollo-server-
core/dist/requestPipeline.js:213:34)","    at Object.<anonymous>
(/Users/sudex/Projects/moffers/server/node_modules/apollo-server-
core/dist/requestPipeline.js:119:42)","    at Generator.next (<anonymous>)","
   at fulfilled (/Users/sudex/Projects/moffers/server/node_modules/apollo-
server-core/dist/requestPipeline.js:5:58)","    at runMicrotasks
(<anonymous>)","    at processTicksAndRejections
(internal/process/task_queues.js:97:5)"]}}}]}
```

Figure 27. GraphQL error with the same fields

When it is needed to query data in the same field with different sets of arguments, GraphQL provides aliases. (Ravichandran 2019.)

An example of using aliases "user1" and "user2" to query users with different ids is shown in Figure 28.



```
{
  user1: user(id: "5f3facca68008c0513d0a5d7") {
    id
    name
    email
    role
    company {
      id
      name
    }
    createdAt
    updatedAt
  }
  user2: user(id: "5f3facca68008c0513d0a5d6") {
    id
    name
    email
    role
    company {
      id
      name
    }
    createdAt
    updatedAt
  }
}
```

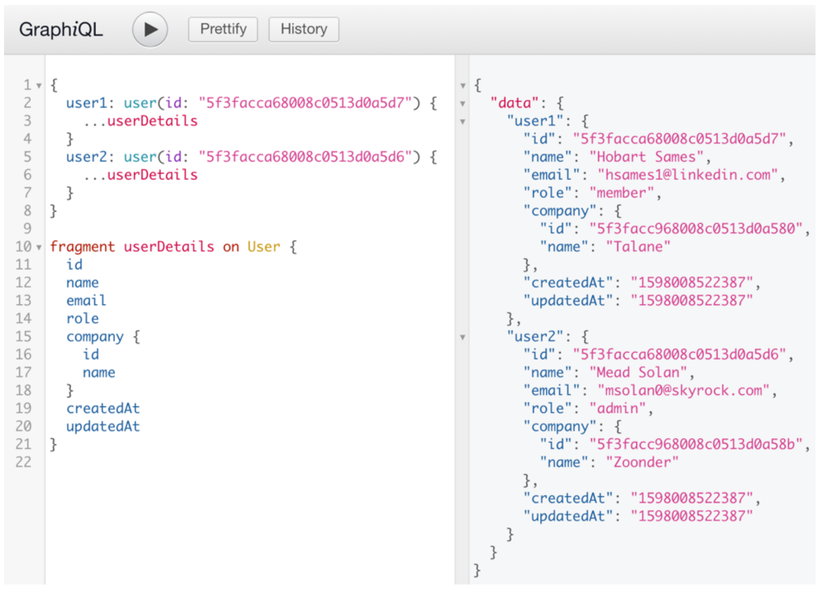
```
{
  "data": {
    "user1": {
      "id": "5f3facca68008c0513d0a5d7",
      "name": "Hobart Sames",
      "email": "hsames1@linkedin.com",
      "role": "member",
      "company": {
        "id": "5f3facc968008c0513d0a580",
        "name": "Talane"
      },
      "createdAt": "1598008522387",
      "updatedAt": "1598008522387"
    },
    "user2": {
      "id": "5f3facca68008c0513d0a5d6",
      "name": "Mead Solan",
      "email": "msolan0@skyrock.com",
      "role": "admin",
      "company": {
        "id": "5f3facc968008c0513d0a58b",
        "name": "Zoonder"
      },
      "createdAt": "1598008522387",
      "updatedAt": "1598008522387"
    }
  }
}
```

Figure 28. GraphQL query with using aliases

4.7 Fragments

To duplicate structures of the same type in queries, GraphQL provides a useful concept of fragments. This allows creating multiple fields and including them in multiple queries. A fragment is specified using the *fragment* keyword, followed by its name and type. (Osnovy GraphQL 2019.)

The improved syntax of the previous example using the *userDetails* fragment is shown in Figure 29.



```
{
  user1: user(id: "5f3facca68008c0513d0a5d7") {
    ...userDetails
  }
  user2: user(id: "5f3facca68008c0513d0a5d6") {
    ...userDetails
  }
}

fragment userDetails on User {
  id
  name
  email
  role
  company {
    id
    name
  }
  createdAt
  updatedAt
}
```

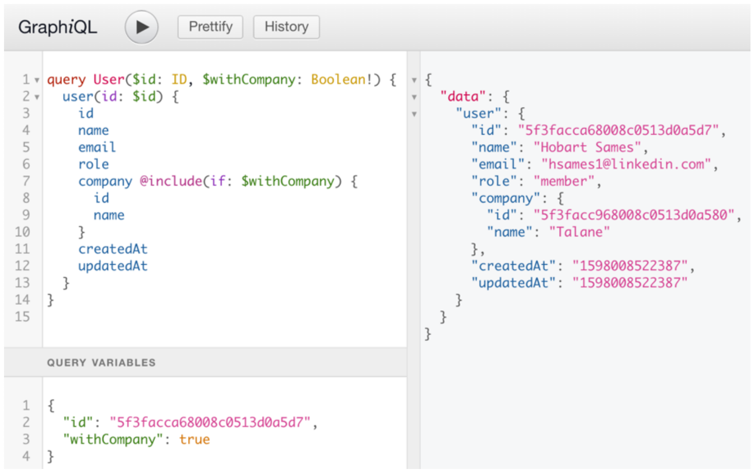
```
{
  "data": {
    "user1": {
      "id": "5f3facca68008c0513d0a5d7",
      "name": "Hobart Sames",
      "email": "hsames1@linkedin.com",
      "role": "member",
      "company": {
        "id": "5f3facc968008c0513d0a580",
        "name": "Talane"
      },
      "createdAt": "1598008522387",
      "updatedAt": "1598008522387"
    },
    "user2": {
      "id": "5f3facca68008c0513d0a5d6",
      "name": "Mead Solan",
      "email": "msolan0@skyrock.com",
      "role": "admin",
      "company": {
        "id": "5f3facc968008c0513d0a58b",
        "name": "Zoonder"
      },
      "createdAt": "1598008522387",
      "updatedAt": "1598008522387"
    }
  }
}
```

Figure 29. GraphQL query with using fragment

4.8 Directives

A directive is used in GraphQL in two cases: when required to return some value by condition (*@include*) or skip some value (*@skip*). (Osnovy GraphQL 2019.)

Figure 30 shows an *@include* directive example: the *withCompany* variable is declared, responsible for whether the *company* field should be included in the query.



```
query User($id: ID, $withCompany: Boolean!) {
  user(id: $id) {
    id
    name
    email
    role
    company @include(if: $withCompany) {
      id
      name
    }
    createdAt
    updatedAt
  }
}
```

```
{
  "id": "5f3facca68008c0513d0a5d7",
  "withCompany": true
}
```

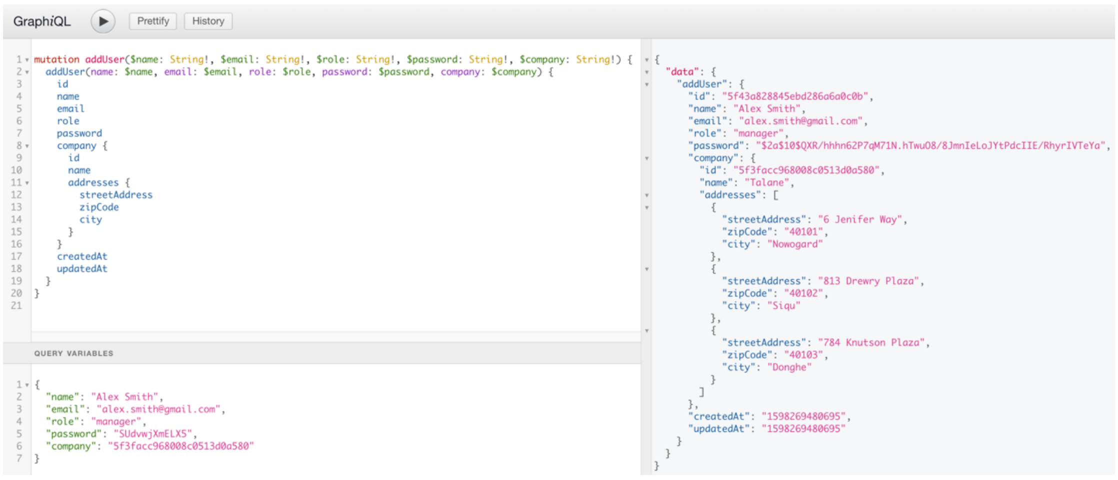
```
{
  "data": {
    "user": {
      "id": "5f3facca68008c0513d0a5d7",
      "name": "Hobart Sames",
      "email": "hsames1@linkedin.com",
      "role": "member",
      "company": {
        "id": "5f3facc968008c0513d0a580",
        "name": "Talane"
      },
      "createdAt": "1598008522387",
      "updatedAt": "1598008522387"
    }
  }
}
```

Figure 30. GraphQL query with an *@include* directive

4.9 Mutations

Mutations in GraphQL perform data changes that affect the original data's state, such as creating, updating, or deleting. When describing mutations, a syntax is similar to when forming queries but using a mutation operation type instead of a query. (Bènks & Porsello 2019.)

Figure 31 shows an example of the *addUser* mutation that returns the created user's fields.



```graphql
mutation addUser($name: String!, $email: String!, $role: String!, $password: String!, $company: String!) {
  addUser(name: $name, email: $email, role: $role, password: $password, company: $company) {
    id
    name
    email
    role
    password
    company {
      id
      name
      addresses {
        streetAddress
        zipCode
        city
      }
    }
    createdAt
    updatedAt
  }
}
```

```json
{
  "name": "Alex Smith",
  "email": "alex.smith@gmail.com",
  "role": "manager",
  "password": "SUdvwjXmELX5",
  "company": "5f3facc968008c0513d0a580"
}
```

```json
{
  "data": {
    "addUser": {
      "id": "5f43a828845ebd286a6a0c0b",
      "name": "Alex Smith",
      "email": "alex.smith@gmail.com",
      "role": "manager",
      "password": "$2a$10$QXR/hhhn62P7qM71N.hTwu08/8JmnIeLoJYtPdcIIE/RhyrIVTeYa",
      "company": {
        "id": "5f3facc968008c0513d0a580",
        "name": "Talane",
        "addresses": [
          {
            "streetAddress": "6 Jenifer Way",
            "zipCode": "40101",
            "city": "Nowogard"
          },
          {
            "streetAddress": "813 Drewry Plaza",
            "zipCode": "40102",
            "city": "Siqu"
          },
          {
            "streetAddress": "784 Knutson Plaza",
            "zipCode": "40103",
            "city": "Donghe"
          }
        ]
      },
      "createdAt": "1598269480695",
      "updatedAt": "1598269480695"
    }
  }
}
```

Figure 31. GraphQL mutation

4.10 Subscriptions

Subscriptions are a way of passing data from a server to clients who want to receive real-time updates. Subscriptions, like queries, specify a set of fields to be delivered to the client. Still, unlike queries, the client does not need to re-send the request - the result is automatically sent every time a specific event occurs on the server. Another difference from queries is that subscriptions are stateful and require the maintenance of the GraphQL document, variables, and context for the subscription duration. For example, in Figure 32, the server remembers the subscription, requested fields, and so on from the client and uses them to return a response to an event. (Vardhan 2020.)

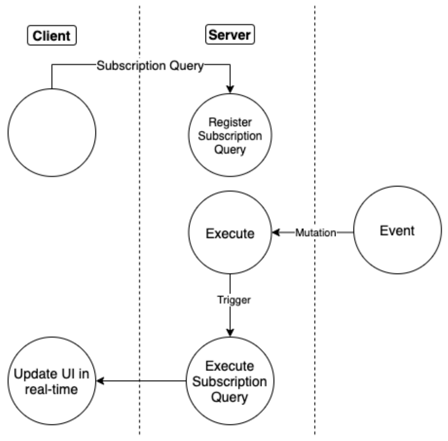
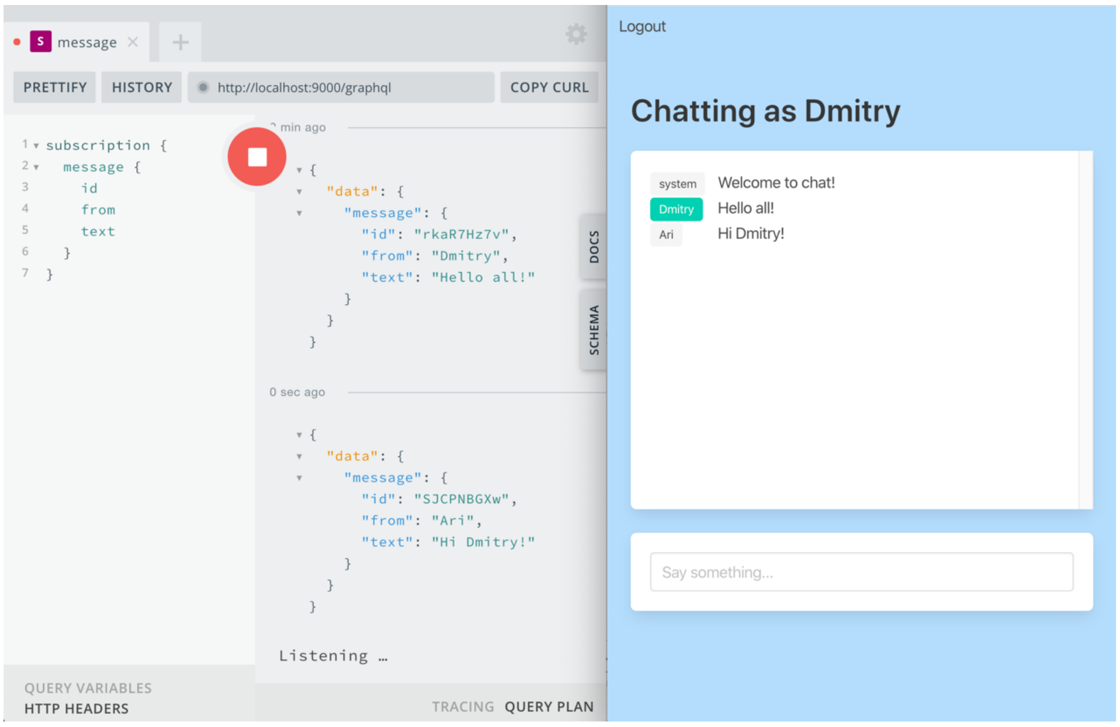


Figure 32. Scheme for implementing subscriptions in GraphQL (Vardhan 2020)

A classic example of using the subscriptions would be a chat implemented with Web-Sockets. Figure 33 shows an example of a chat application, in which a message is sent from the client (right) and received in listening mode using subscriptions on the server (left).



```
subscription {
  message {
    id
    from
    text
  }
}
```

```
? min ago
{
  "data": {
    "message": {
      "id": "rkaR7Hz7v",
      "from": "Dmitry",
      "text": "Hello all!"
    }
  }
}

0 sec ago
{
  "data": {
    "message": {
      "id": "SJCPNBGXw",
      "from": "Ari",
      "text": "Hi Dmitry!"
    }
  }
}

Listening …
```

Chatting as Dmitry

system	Welcome to chat!
Dmitry	Hello all!
Ari	Hi Dmitry!

Figure 33. A chat app implemented using the GraphQL subscriptions

4.11 GraphQL server

GraphQL server is required to execute queries. It provides a single endpoint for client interaction and recognizes the query before responding with the correct data (Figure 34). (Glover 2019; Huder 2019.)

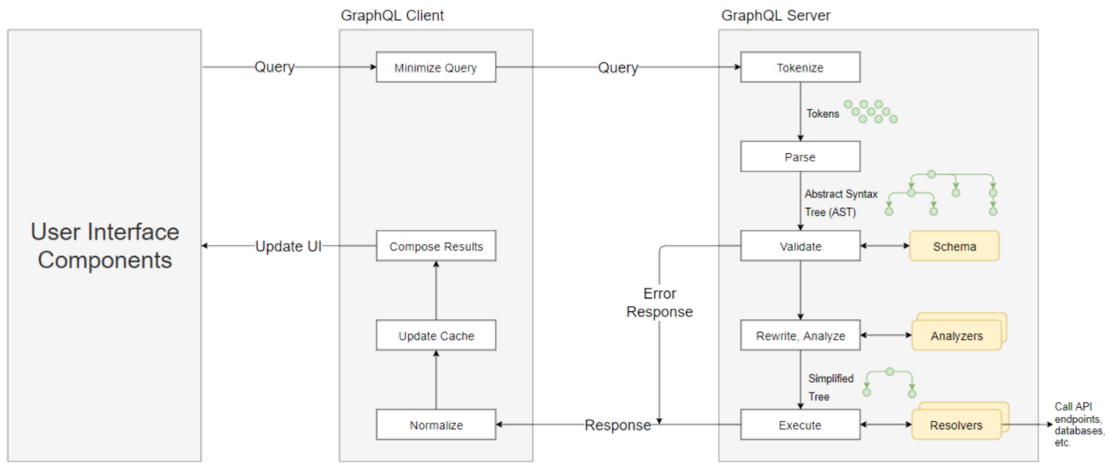


Figure 34. GraphQL general architecture (Huder 2019)

GraphQL service tokenizes the query string and parses the resulting set of tokens to create an abstract syntax tree. Then there is a validation process with the existing scheme. In case of an error, the corresponding response is returned to the client. If the query is valid, the tree can be reduced to a more straightforward form. If query analyzers are defined, they will be called. At runtime, all resolvers are called to get the actual data for each field. The resolver function is used to retrieve data for the corresponding field. This function queries a database or a third-party server. After returning all resolvers' results, the GraphQL server packages the data in the format described by the query and sends it back to the client. (Huder 2019.)

Three architectures include GraphQL Service: GraphQL server with a database connected, GraphQL server that integrates existing systems, and a hybrid approach.

4.11.1 GraphQL server with a database connected

This type of architecture has a single web server that implements the GraphQL specification. The server reads the query's payload when it arrives and retrieves the information it needs from the database, i.e., resolves the query. It then creates a response object and returns it to the client (Figure 35). (Huder 2019; Big Picture (Architecture) n.d.)

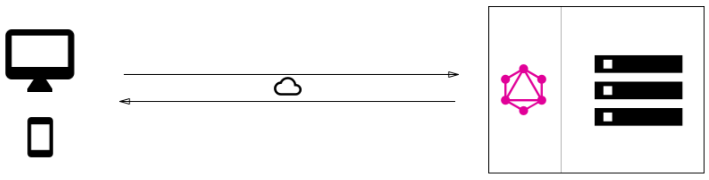


Figure 35. GraphQL server with a single database (Big Picture (Architecture) n.d.)

4.11.2 GraphQL server that integrates existing systems

In this type of architecture, the GraphQL server integrates several existing systems into a single consistent API (Figure 36). This architecture is relevant when working with many services with their API (Huder 2019; Big Picture (Architecture) n.d.)

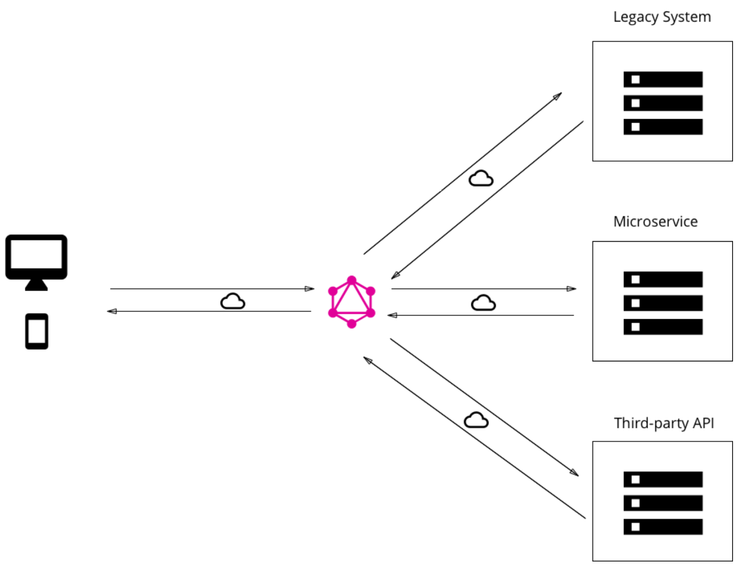


Figure 36. GraphQL server integrates systems into a single API (Big Picture (Architecture) n.d.)

4.11.3 Hybrid approach

By combining the two previous approaches, it is possible to build a GraphQL server with a connected database and communicate with third-party resources. When this server receives a query, it decides on the resource that provides the data (Figure 37). (Huder 2019; Big Picture (Architecture) n.d.)

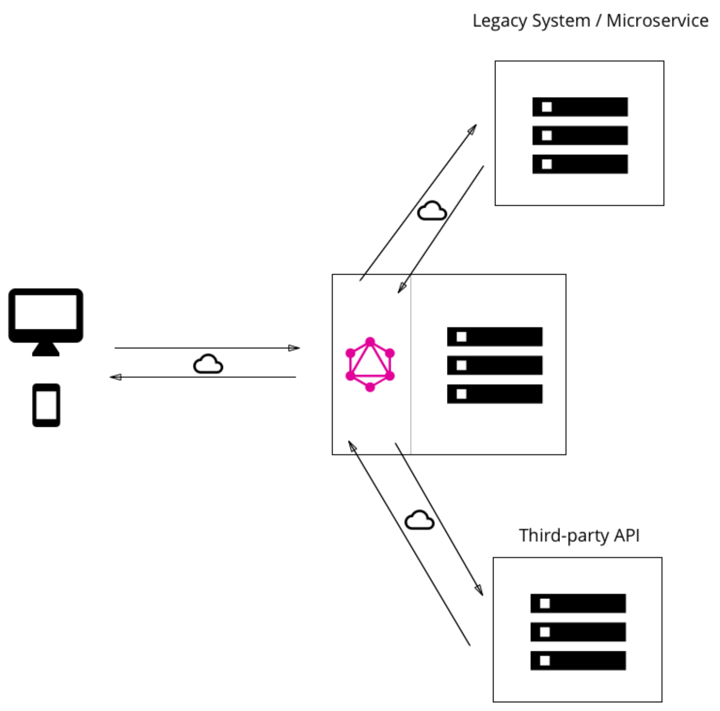
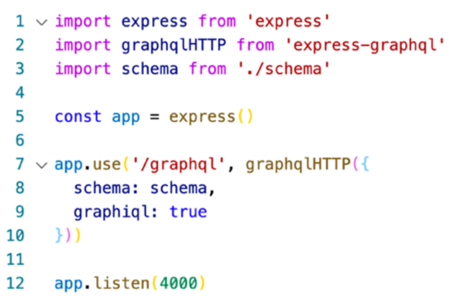


Figure 37. A hybrid approach to building architecture with GraphQL server (Big Picture (Architecture) n.d.)

4.11.4 Server building

GraphQL server implementation locally using modern libraries and frameworks does not require any special skills and can be implemented in a few code lines. Figure 38 shows an example of a GraphQL server built with Express.



```
import express from 'express'
import graphqlHTTP from 'express-graphql'
import schema from './schema'

const app = express()

app.use('/graphql', graphqlHTTP({
  schema: schema,
  graphiql: true
}))

app.listen(4000)
```

Figure 38. HTTP server built with Express

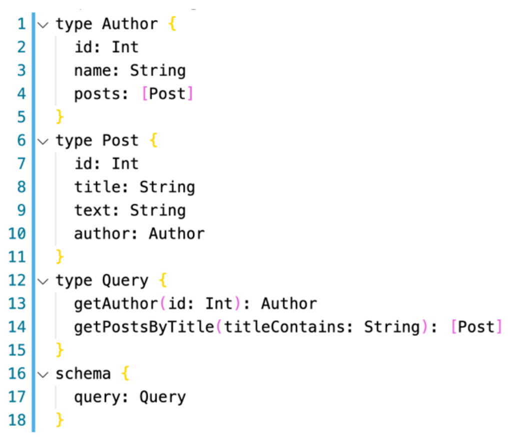
Using the *graphqlHTTP()* function from the *express-graphql* package, the scheme described in a separate file is "attached", and the server is started on port 4000. Thus, a local path "http://localhost:4000/graphql" is generated for the client, to which queries can be sent.

Every GraphQL server has two main parts that define how it works: schema and resolution functions. (Helfer 2016.)

4.11.5 Schema

GraphQL schema is at the heart of any GraphQL server. It defines the server's API, allowing clients to know what operations can be performed, what queries are allowed to make, what types of data can be retrieved from the server, and the relationships between these types. (Helfer 2016; Mbanugo 2019.)

Figure 39 shows the example of schema indicating that the application has three types – *Author*, *Post*, and *Query*. The third type, *Query*, is to mark the entry point into the schema. Every query has to start with one of its fields: *getAuthor* or *getPostsByTitle*. *Author* and *Post* refer to each other. (Helfer 2016.)



```
type Author {
  id: Int
  name: String
  posts: [Post]
}
type Post {
  id: Int
  title: String
  text: String
  author: Author
}
type Query {
  getAuthor(id: Int): Author
  getPostsByTitle(titleContains: String): [Post]
}
schema {
  query: Query
}
```

Figure 39. GraphQL schema (Helfer 2016)

A graphical representation of this schema is shown in Figure 40.

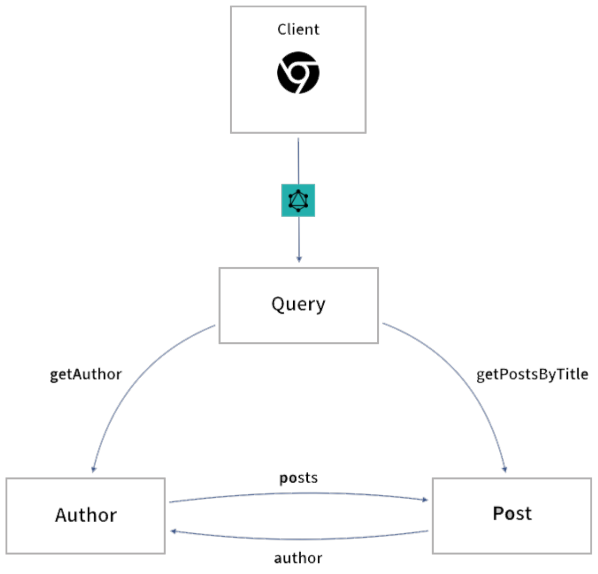


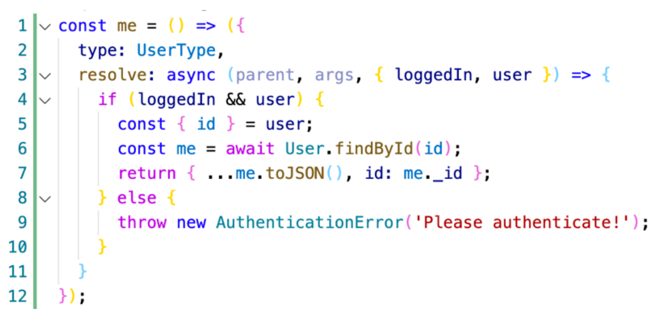
Figure 40. GraphQL schema: a graphical representation (Helfer 2016)

4.11.6 Resolvers

Schema tells the server what queries are allowed for clients and how the different types are related, but it does not contain information where the data for each type comes from. This is what resolvers (or resolve functions) are for. (Helfer 2016.)

Regardless of the implementation language, each resolver can take four arguments: *root*, *args*, *context,* and *info*. The *root* argument is an object that is used to pass data from parent to child resolvers. The *args* is an object with arguments passed to the field in the query. The *context* argument is a mutable object that is created and destroyed between each query. The *info* argument contains information about the query's status, including the field name, a path to the field from the root. (Stuart 2018.)

An example of an asynchronous resolver using the *context* argument to store authentication data is shown in Figure 41.



```
const me = () => ({
  type: UserType,
  resolve: async (parent, args, { loggedIn, user }) => {
    if (loggedIn && user) {
      const { id } = user;
      const me = await User.findById(id);
      return { ...me.toJSON(), id: me._id };
    } else {
      throw new AuthenticationError('Please authenticate!');
    }
  }
});
```

Figure 41. Auth function *me()* using asynchronous resolver

4.12 GraphQL client

The GraphQL client's purpose is to send queries to the server (without constructing HTTP requests). A typical GraphQL client also provides features like data caching, validation, and optimization of queries based on the schema. (Huder 2019; Advanced Tutorial – Clients n.d.)

4.12.1 Sending queries to the server

One of the benefits of GraphQL is that it allows receiving and updating data declaratively. In other words, it provides a higher-level abstraction over the API. GraphQL client is responsible for designing, optimizing, and sending the request over the network. The interface component only indicates what data is needed. (Huder 2019; Advanced Tutorial – Clients n.d.)

4.12.2 Integration with UI components

After processing the received data, the GraphQL client is responsible for updating the UI's necessary part. Depending on the platforms and framework, there are different approaches to how the process of updating the interface takes place as a whole. For example, the React library's GraphQL client uses the concept of HOCs and hooks (Apollo Client version >= 3.0) to obtain the required data and make it available in the interface component. (Huder 2019; Advanced Tutorial – Clients n.d.; Hooks n.d.)

4.12.3 Data caching

Most applications need to maintain a cache of data that was previously received from the server. Caching information at the local level is essential to ensure free user experience and load from the server. Each GraphQL client implements the caching process differently. There are simple mechanisms, the algorithm of which is to save a hash table in which the key is the query, and the value is the response from the server. This method of preservation can perform its function but does not do it efficiently. The main problem with this approach is data duplication, which leads to a rapid increase in cache size. A more cost-effective approach is based on the pre-normalization of data. This allows getting flat data and a set of individual records with unique identifiers. With IDs, it is possible to retrieve and modify data quickly. (Huder 2019; Advanced Tutorial – Clients n.d.)

4.12.4 Build-time schema validation and optimizations

Since the schema contains all the information about what the client could potentially do with the API, it is possible to check and optimize the client's requests during the project build process. GraphQL client can analyze all GraphQL queries in the project and compare them with the schema's information. This allows catching most of the errors during the product development phase. (Huder 2019; Advanced Tutorial – Clients n.d.)

5 Apollo

Apollo is a data graph building platform developed by Meteor Development Group Inc. The data graph is a communication layer that connects the client side of the applications to the internal services. (Documentation Home n.d.)

The Apollo platform helps to build, query, and manage a data graph. This is a unified data layer that enables applications to interact with data from connected data stores and external APIs. A data graph is positioned between application clients and backend services, facilitating data flow between them, as shown in Figure 42. (The Apollo platform n.d.)

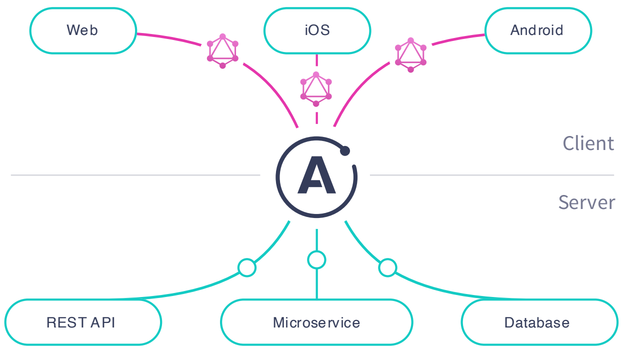


Figure 42. App data flow with Apollo and GraphQL (The Apollo platform n.d.)

Apollo data graph uses GraphQL to define and provide the structure of the data flow. (The Apollo platform n.d.)

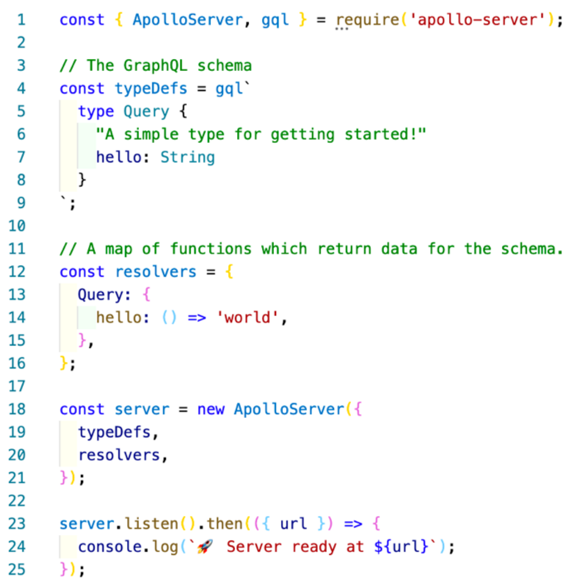
5.1 Apollo Server

The data graph needs a service that can handle GraphQL operations from application clients by interacting with internal data sources to retrieve and modify data. To build this service, Apollo Platform provides Apollo Server, an extensible open-source JavaScript GraphQL server that is used to define GraphQL schema and resolvers. (The Apollo platform n.d.)

Apollo Server can be used as a stand-alone GraphQL server, an add-on to existing Node.js application middleware (such as Express), or as a gateway for a federated data graph. It provides a fairly straightforward setup for fetching data, incremental implementation to add functionality as needed, universally compatible with any data source, build tool, and any GraphQL client. (Introduction to Apollo Server n.d.)

5.1.1 Stand-alone installation

Using Apollo Server as a stand-alone in a project requires installing *apollo-server* and *graphql* NPM dependencies and then creating an *index.js* file as an entry point that defines the schema and its functionality, i.e., resolvers (Figure 43). (Apollo Server 2020.)



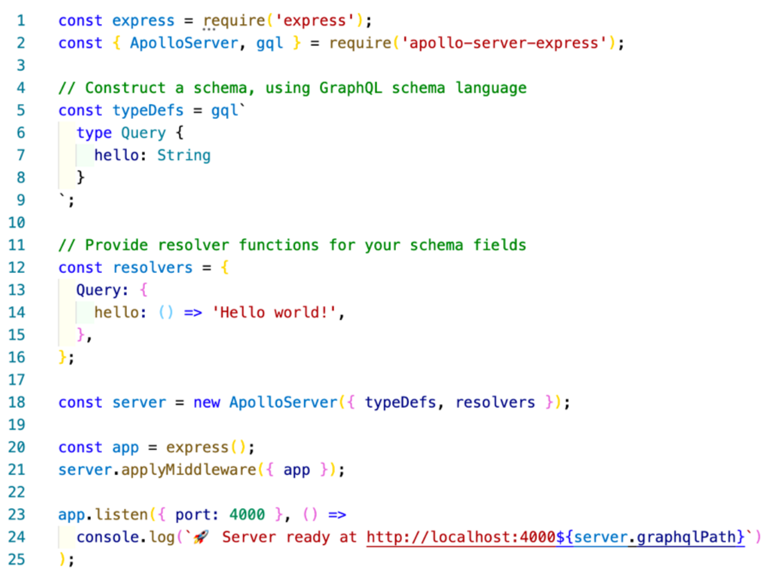
```
const { ApolloServer, gql } = require('apollo-server');

// The GraphQL schema
const typeDefs = gql`
  type Query {
    "A simple type for getting started!"
    hello: String
  }
`;

// A map of functions which return data for the schema.
const resolvers = {
  Query: {
    hello: () => 'world',
  },
};

const server = new ApolloServer({
  typeDefs,
  resolvers,
});

server.listen().then(({ url }) => {
  console.log(`🚀 Server ready at ${url}`);
});
```

Figure 43. Deploying Apollo Server using SDL (Apollo Server 2020)

5.1.2 Integration installation

Apollo Server integration packages can be paired with specific web frameworks, e.g., Express, Koa, Hapi, Fastify, and few others. Each of these connected integrations has its installation instructions and examples on its package *README.md*. (Apollo Server 2020.)

An example of the most popular Apollo Server and Express integration installed using the *apollo-server-express* NPM package is shown in Figure 44.

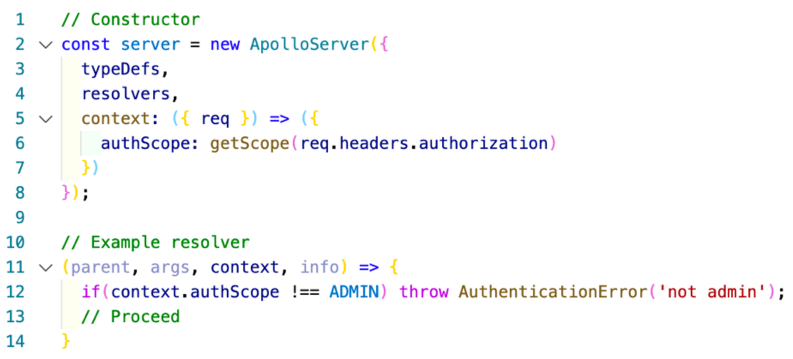


```
const express = require('express');
const { ApolloServer, gql } = require('apollo-server-express');

// Construct a schema, using GraphQL schema language
const typeDefs = gql`
  type Query {
    hello: String
  }
`;

// Provide resolver functions for your schema fields
const resolvers = {
  Query: {
    hello: () => 'Hello world!',
  },
};

const server = new ApolloServer({ typeDefs, resolvers });

const app = express();
server.applyMiddleware({ app });

app.listen({ port: 4000 }, () =>
  console.log(`🚀 Server ready at http://localhost:4000${server.graphqlPath}`)
);
```

Figure 44. Deploying Apollo Server with Express integration using SDL

5.1.3 Context

For each request, Apollo Server provides a context as an optional third argument in its constructor. The context is defined as a function, called on every request, and gets an object containing the *req* property representing the request itself (Figure 45). (Apollo Server 2020.)



```
// Constructor
const server = new ApolloServer({
  typeDefs,
  resolvers,
  context: ({ req }) => ({
    authScope: getScope(req.headers.authorization)
  })
});

// Example resolver
(parent, args, context, info) => {
  if(context.authScope !== ADMIN) throw AuthenticationError('not admin');
  // Proceed
}
```

Figure 45. Context using for passing authentication scope

Context can be especially useful for authentication data transfers, database connections, and custom fetch functions. (Resolvers n.d.)

5.2 Apollo Client

Apollo Client is an open-source state management library for JavaScript that allows defining queries directly in UI components, manage local and remote data with GraphQL, and retrieve, cache, and modify app data automatically when the UI is updated. (The Apollo platform n.d.; Introduction to Apollo Client n.d.)

Apollo Client includes official React support, so all React functionality is available out of the box with both *create-react-app* and React Native, including setting up Babel and other JavaScript tools. There are also community-supported libraries for other popular libraries and frameworks like Angular, Vue, Svelte, Ember. (View Integrations n.d.)

5.2.1 Apollo Client architecture

General Apollo Client architecture is shown in Figure 46.

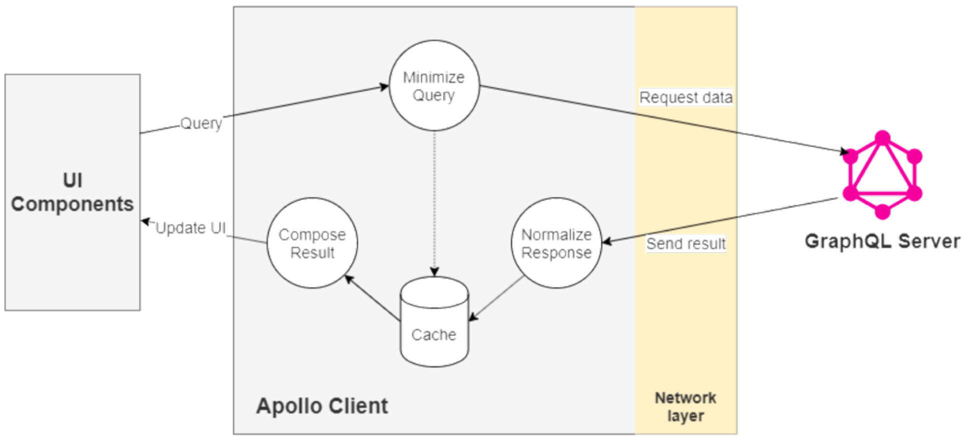


Figure 46. Apollo Client architecture (Huder 2019)

The main components of the Apollo Client are the cache and network layer. Cache in Apollo Client is implemented to store the results of its queries in the browser, avoid unnecessary network calls, and speed up the application. Depending on the specific fetch policy settings, a query can be fetching new data from the server or reading it from the cache. (Corey 2018.)

5.2.2 Cache layer

Apollo Client offers several data fetch policies: *cache-first*, *cache-and-network*, *network-only*, *no-cache,* and *cache-only*. (Huder 2019.)

Cache-first is Apollo's default fetch policy. If all data required to complete a query are in the cache, that data will be returned. Apollo Client requests the server only if the required data is missing in the cache. This fetch policy aims to minimize the number of network requests directed from the interface component. (Corey 2018.)

Using *cache-and-network* fetch policy Apollo Client checks the cache for the data and, if the data is in the cache, returns it. Regardless of whether any data was found or not, a request is passed to the server to get the most recent data and update the cache with it. This fetch policy allows the user to get a quick response and also keep the cached data consistent with server data through additional network requests. (Corey 2018.)

With *network-only* fetch policy, data is not read from the cache. Instead, the Apollo Client always makes a network request. The cache is updated after the server responds. This fetch policy solves the problem of data consistency at the expense of instant response to the user. (Corey 2018.)

No-cache fetch policy is similar to the *network-only* policy but skips the cache updating step. This may be appropriate if you do not need to store certain information in the cache. (Corey 2018.)

Cache-only fetch policy is the exact opposite of *no-cache,* avoiding any network requests. If the requested data is not available in the cache, an error will be thrown.

This can be useful if there is a need to display consistent information to the user while ignoring any server-side changes. (Corey 2018.)

5.2.3 Network layer

The main component for building a network layer in Apollo Client is the Apollo Link. This library is designed as a powerful way to compose actions around data processing using GraphQL. Each link is a subset of functionality combined with other links to create complex data management flows. At a basic level, Apollo Link is a function that receives an *Operation* and returns an *Observable,* as shown in Figure 47. (Concepts Overview n.d.)

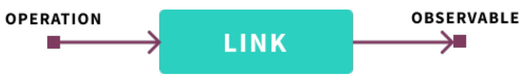


Figure 47. Apollo Link (Concepts Overview n.d.)

An *Operation* object contains the information about a query (*query*), operation name (*operationName*) and its variables (*variables*) being sent with, extension data (*extensions*) and functions such as *getContext()*, *setContext()* and *toKey()*. The *getContext()* returns a request context that can be used by links to determine the actions to take. The *setContext()* function takes either a new context object or a function that takes the previous context and returns a new one. The *toKey()* function converts the current operation to a string used as a unique identifier.

To support different application requirements, each GraphQL response is represented by an *Observable*. Observables provide a subscribe method that takes three callbacks: *next()*, *error()*, and *complete()*. The *next()* method can be called many times until either the *error()* or *complete()* callback is triggered. This callback structure is great for working with current and planned GraphQL results, including queries, mutations, subscriptions, and even live queries. (Hauser 2017.)

The network layer algorithm sequentially sends a request to each Apollo Link one by one, as shown in Figure 48. (Concepts Overview n.d.)

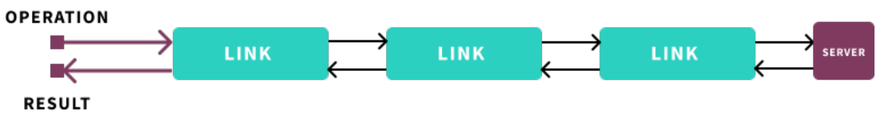
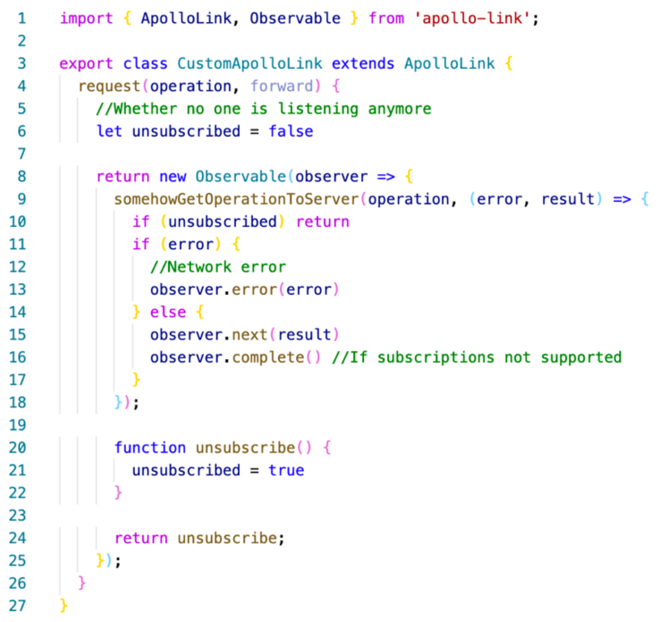


Figure 48. Apollo Client network layer (Concepts Overview n.d.)

Apollo Link is based on the *request()* method, which accepts as an argument the *operation* being passed through the Apollo Link and optionally a link to the next link in the chain (*forward*). (Concepts Overview n.d.) An example implementation of a basic custom network layer using Apollo Link is shown in Figure 49.

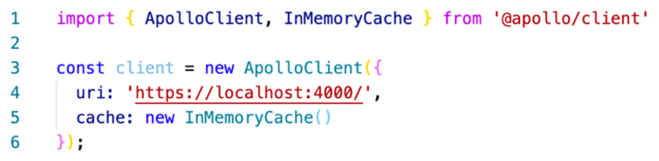


```
import { ApolloLink, Observable } from 'apollo-link';

export class CustomApolloLink extends ApolloLink {
  request(operation, forward) {
    //Whether no one is listening anymore
    let unsubscribed = false

    return new Observable(observer => {
      somehowGetOperationToServer(operation, (error, result) => {
        if (unsubscribed) return
        if (error) {
          //Network error
          observer.error(error)
        } else {
          observer.next(result)
          observer.complete() //If subscriptions not supported
        }
      });

      function unsubscribe() {
        unsubscribed = true
      }

      return unsubscribe;
    });
  }
}
```

Figure 49. Custom network implementation with Apollo Link

5.2.4 Configure Apollo Client with React

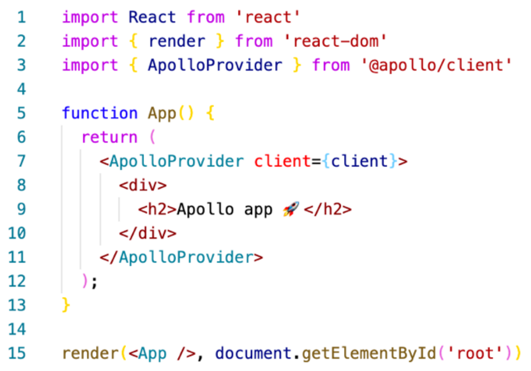
Apollo encapsulates all of the lower-level network logic and provides an interface to the GraphQL server. The first thing to do when using Apollo is to set up an *ApolloClient* instance that needs to know the GraphQL API endpoint to work with network connections (Figure 50). (Burk n.d.; Get started n.d.)



```
import { ApolloClient, InMemoryCache } from '@apollo/client'

const client = new ApolloClient({
  uri: 'https://localhost:4000/',
  cache: new InMemoryCache()
});
```

Figure 50. Creating an Apollo Client instance

To connect Apollo Client to React the *ApolloProvider* component is used. It wraps the React application and puts the client in a context, allowing access to it. Typically, *ApolloProvider* is located at the highest level in the component tree to access GraphQL data for any component (Figure 51). (Get started n.d.)

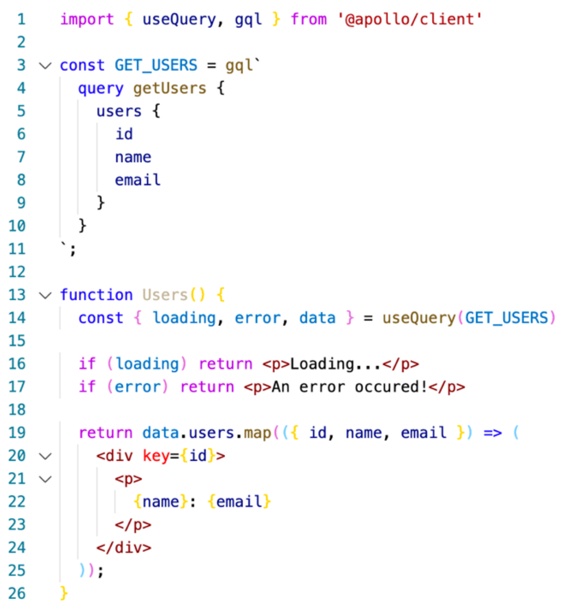


```
import React from 'react'
import { render } from 'react-dom'
import { ApolloProvider } from '@apollo/client'

function App() {
  return (
    <ApolloProvider client={client}>
      <div>
        <h2>Apollo app 🚀</h2>
      </div>
    </ApolloProvider>
  );
}

render(<App />, document.getElementById('root'))
```

Figure 51. Creating an Apollo Client instance

5.2.5 Data querying

Once the *ApolloProvider* is connected, it is possible to query data using the *use-Query()* React hook, which uses the Hooks API to exchange GraphQL data with the UI. A GraphQL query can be passed to the *useQuery()* hook using the *gql* function. When the component is rendered, and the *useQuery()* hook runs, a result object is returned that contains the *loading*, *error*, and *data* properties, which are used to handle the load state of the application, error in getting data that might occur, and the data itself received from the server (Figure 52). (Get started n.d.)



```
import { useQuery, gql } from '@apollo/client'

const GET_USERS = gql`
  query getUsers {
    users {
      id
      name
      email
    }
  }
`;

function Users() {
  const { loading, error, data } = useQuery(GET_USERS)

  if (loading) return <p>Loading...</p>
  if (error) return <p>An error occured!</p>

  return data.users.map(({ id, name, email }) => (
    <div key={id}>
      <p>
        {name}: {email}
      </p>
    </div>
  ));
}
```

Figure 52. Data query using *useQuery()* hook

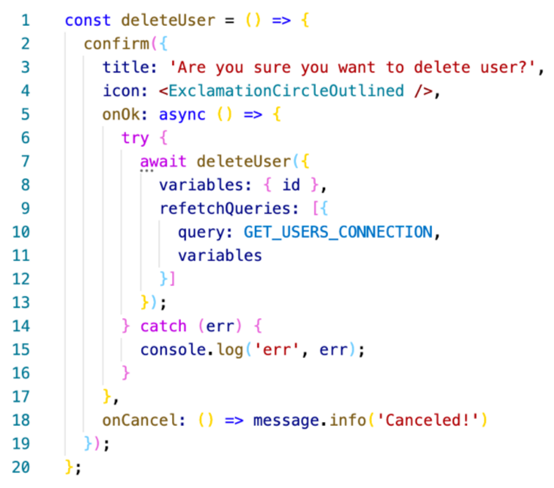
5.2.6 Updating the cache after a mutation

For a single entity update mutation, the Apollo Client automatically updates that entity's value in its cache when the mutation returns. To do so, the mutation must return the id of the modified entity along with the fields that were modified. (Mutations n.d.)

For all other cases, Apollo provides the feature to manage the contents of the cache manually. This is very convenient, especially after a mutation to update, create, or delete data has been made. It allows defining exactly how the cache should be updated. (More Mutations and Updating the Store n.d.)

There are two ways to sync cache each time a mutation operation performs: by refetching matching queries using the *refetchQueries* object property and modifying the cache data using the *update()* helper function. (Krofegha 2020.)

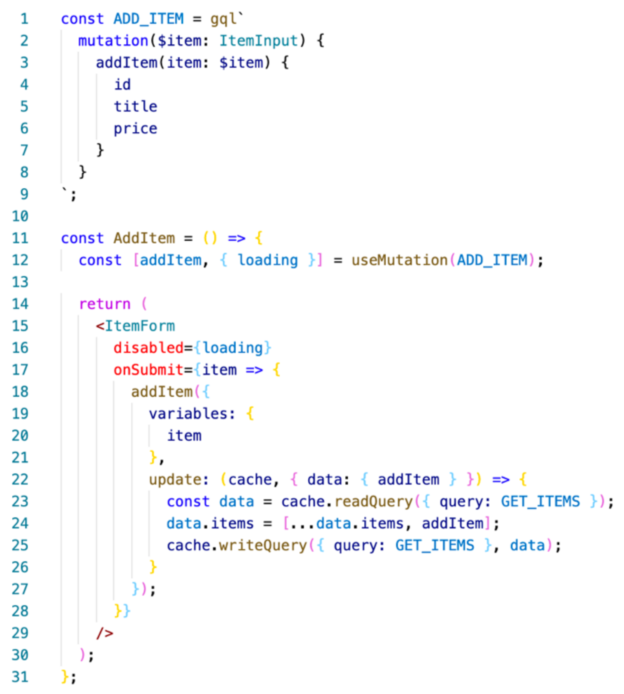
The easiest way to update the cache is by using the *refetchQueries* object property. It defines one or more queries that need to be run after a mutation is completed to refetch the store's parts that may have been affected by the mutation. An example of using this when deleting a user is shown in Figure 53. (Advanced topics on caching n.d.)



```
const deleteUser = () => {
  confirm({
    title: 'Are you sure you want to delete user?',
    icon: <ExclamationCircleOutlined />,
    onOk: async () => {
      try {
        await deleteUser({
          variables: { id },
          refetchQueries: [{
            query: GET_USERS_CONNECTION,
            variables
          }]
        });
      } catch (err) {
        console.log('err', err);
      }
    },
    onCancel: () => message.info('Canceled!')
  });
};
```

Figure 53. Deleting a user using *refetchQueries()* object property

Using the *update()* function is the recommended way to update the cache. It provides full control over the cache, allowing changes to the data model to be made in response to a mutation in any way. The *useMutation* call can include an *update()* function if the mutation modifies multiple entities, creates or deletes entities, as the Apollo Client cache is not automatically updated. (Advanced topics on caching n.d.) Figure 54 shows an example of defining an *update()* function in a *useMutation* call when adding item:



```
const ADD_ITEM = gql`
  mutation($item: ItemInput) {
    addItem(item: $item) {
      id
      title
      price
    }
  }
`;

const AddItem = () => {
  const [addItem, { loading }] = useMutation(ADD_ITEM);

  return (
    <ItemForm
      disabled={loading}
      onSubmit={item => {
        addItem({
          variables: {
            item
          },
          update: (cache, { data: { addItem } }) => {
            const data = cache.readQuery({ query: GET_ITEMS });
            data.items = [...data.items, addItem];
            cache.writeQuery({ query: GET_ITEMS }, data);
          }
        });
      }}
    />
  );
};
```

Figure 54. Updating an item using *update()* function

6 Project implementation

6.1 Development environment and tools

As a development environment, Visual Studio Code was used. It is a cross-platform lightweight and powerful code editor developed by Microsoft with built-in support for JavaScript, TypeScript, and Node.js and a rich ecosystem of extensions for other programming languages and runtimes. (Getting Started 2020.) Visual Studio Code allows developing console and GUI applications, websites, web applications, and services. The editor includes a built-in terminal and code debugger, tools for working with Git, refactoring tools, code navigation, contextual hints, syntax highlighting, options for customizing custom themes, keyboard shortcuts, and configuration files.

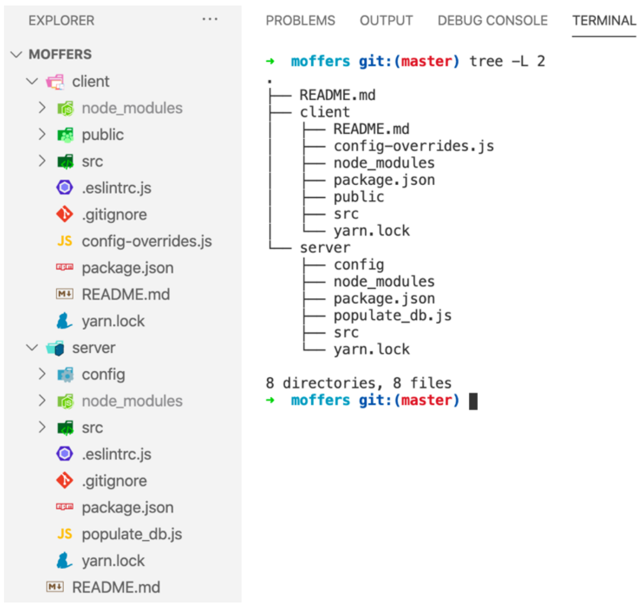
As a project versioning, Git and *Tower* were used. The *Tower* is a powerful native Git-client with a rich feature set, thoughtful design, and handy tools to make it easier to interact with Git (no command-line required).

As the main browser, Chrome was used. With built-in Chrome DevTools set and additionally installed React Developer Tools and Apollo Client Developer Tools extensions, it allows viewing and changing the DOM model, page styles (CSS), debugging JavaScript code, viewing messages, running JavaScript in the console, optimizing web application speed, checking network activity, working with Local Storage, React components and Apollo Cache.

For interacting with the GraphQL API and testing its endpoints were used GraphiQL (at the stage of using Express as a server) and GraphQL Playground (when Apollo Server installed instead).

6.2 Setting up the project

The project was initialized from scratch and included both client and server sides. The main (two-level) folder structure of the project is shown in Figure 55.



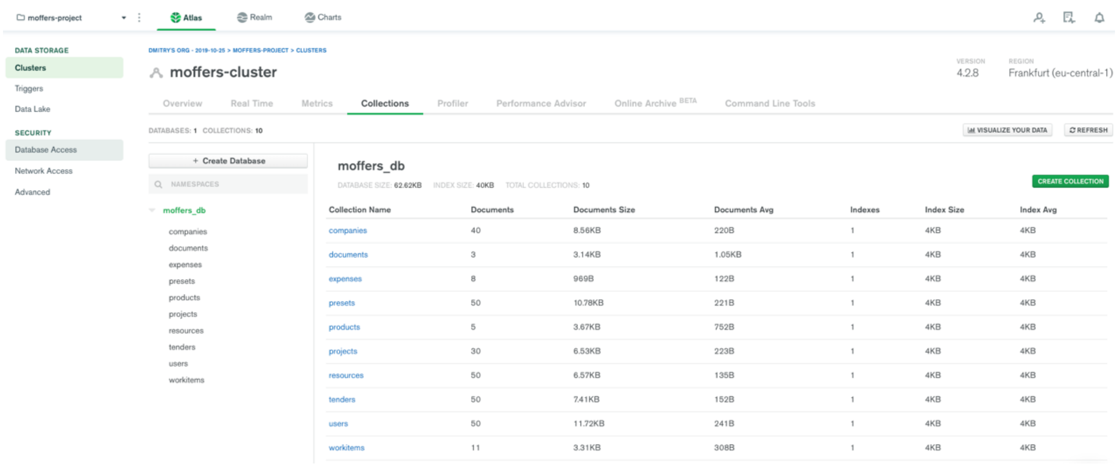
```
➜  moffers git:(master) tree -L 2
.
├── README.md
├── client
│   ├── README.md
│   ├── config-overrides.js
│   ├── node_modules
│   ├── package.json
│   ├── public
│   ├── src
│   └── yarn.lock
└── server
    ├── config
    ├── node_modules
    ├── package.json
    ├── populate_db.js
    ├── src
    └── yarn.lock

8 directories, 8 files
➜  moffers git:(master) 
```

Figure 55. The two-level folder structure of the project

All technologies for implementing the client-server architecture of the project are based on JavaScript - the primary programming language currently used by Movya for all their products.

6.2.1 Database

The database cluster creation was carried out in the cloud using the MongoDB Atlas service, which maximally simplifies working with MongoDB without the need for complicated configuration procedures and settings. The final collections of documents used in the project and stored in the MongoDB Atlas *moffers*-cluster are shown in Figure 56.



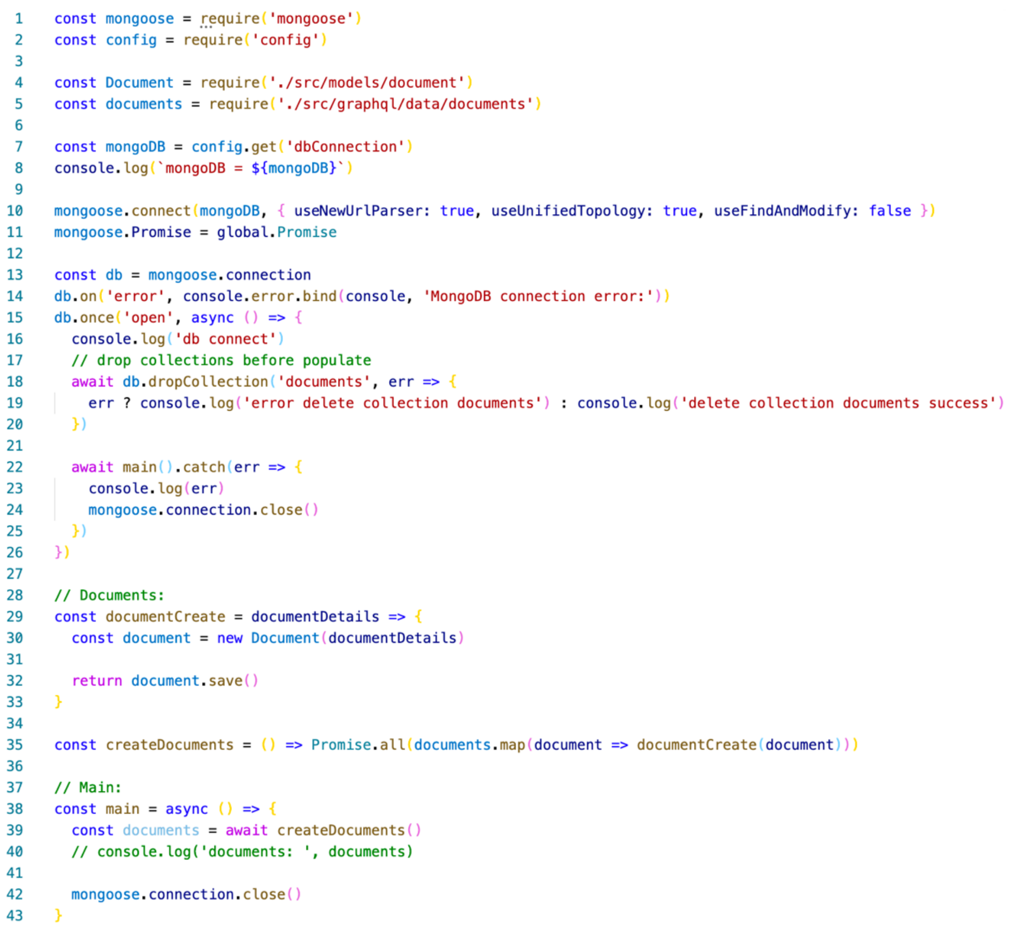
moffers_db

DATABASE SIZE: 62.62KB INDEX SIZE: 40KB TOTAL COLLECTIONS: 10

Collection Name	Documents	Documents Size	Documents Avg	Indexes	Index Size	Index Avg
companies	40	8.56KB	220B	1	4KB	4KB
documents	3	3.14KB	1.05KB	1	4KB	4KB
expenses	8	969B	122B	1	4KB	4KB
presets	50	10.78KB	221B	1	4KB	4KB
products	5	3.67KB	752B	1	4KB	4KB
projects	30	6.53KB	223B	1	4KB	4KB
resources	50	6.57KB	135B	1	4KB	4KB
tenders	50	7.41KB	152B	1	4KB	4KB
users	50	11.72KB	241B	1	4KB	4KB
workitems	11	3.31KB	308B	1	4KB	4KB

Figure 56. *Moffers*-cluster collections

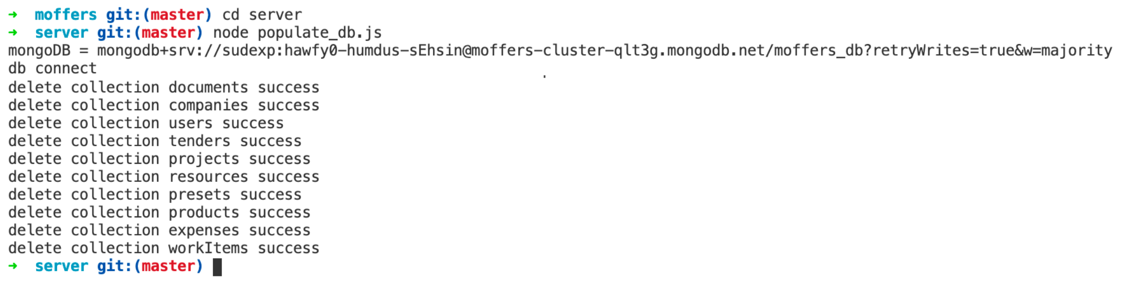
All project collections were populated with the data sufficient for testing the main frontend functions, such as displaying data in various UI-blocks, tables and pagination. Since collections can vary from complete emptying to significant sizes during CRUD operations testing, it makes sense to have an automatic mechanism to return to the initial dataset. For these purposes, a database population script was implemented. For demonstration purposes, the original population script of all the cluster collections, implemented in the project and containing more than 300 lines of code, has been minimized. Figure 57 shows the minimized example of this script for creating the *documents* collection.



```javascript
const mongoose = require('mongoose')
const config = require('config')

const Document = require('./src/models/document')
const documents = require('./src/graphql/data/documents')

const mongoDB = config.get('dbConnection')
console.log(`mongoDB = ${mongoDB}`)

mongoose.connect(mongoDB, { useNewUrlParser: true, useUnifiedTopology: true, useFindAndModify: false })
mongoose.Promise = global.Promise

const db = mongoose.connection
db.on('error', console.error.bind(console, 'MongoDB connection error:'))
db.once('open', async () => {
  console.log('db connect')
  // drop collections before populate
  await db.dropCollection('documents', err => {
    err ? console.log('error delete collection documents') : console.log('delete collection documents success')
  })

  await main().catch(err => {
    console.log(err)
    mongoose.connection.close()
  })
})

// Documents:
const documentCreate = documentDetails => {
  const document = new Document(documentDetails)

  return document.save()
}

const createDocuments = () => Promise.all(documents.map(document => documentCreate(document)))

// Main:
const main = async () => {
  const documents = await createDocuments()
  // console.log('documents: ', documents)

  mongoose.connection.close()
}
```

Figure 57. Minimized population script for the *documents* collection

At first, the script initiates a connection to the configured database using *mongoose* and *config* NPM packages. Then before the database population with new data, it deletes the old collection. The script is executed in the terminal command line by running the *node populate_db.js* command from the *server* folder in which it is located. As output in the terminal, the script displays data about the successful deletion of the collection (Figure 58) and newly populated data optionally (commented out).

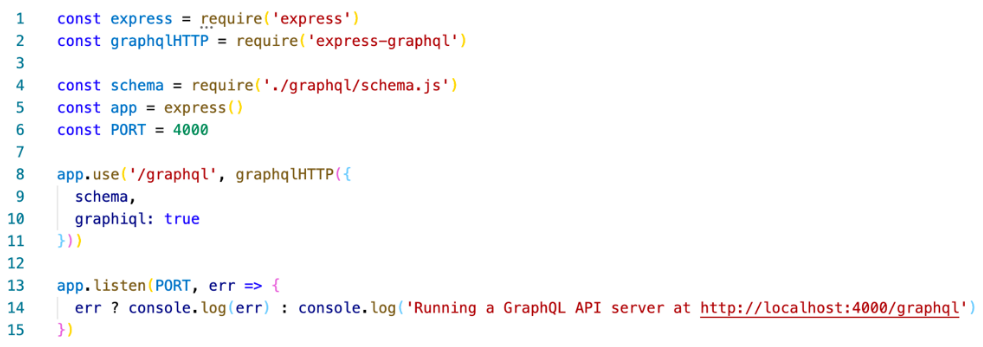


```
➜  moffers git:(master) cd server
➜  server git:(master) node populate_db.js
mongoDB = mongodb+srv://sudexp:hawfy0-humdus-sEhsin@moffers-cluster-qlt3g.mongodb.net/moffers_db?retryWrites=true&w=majority
db connect
delete collection documents success
delete collection companies success
delete collection users success
delete collection tenders success
delete collection projects success
delete collection resources success
delete collection presets success
delete collection products success
delete collection expenses success
delete collection workItems success
➜  server git:(master) █
```

Figure 58. Output after database populating in the terminal

6.2.2 Server

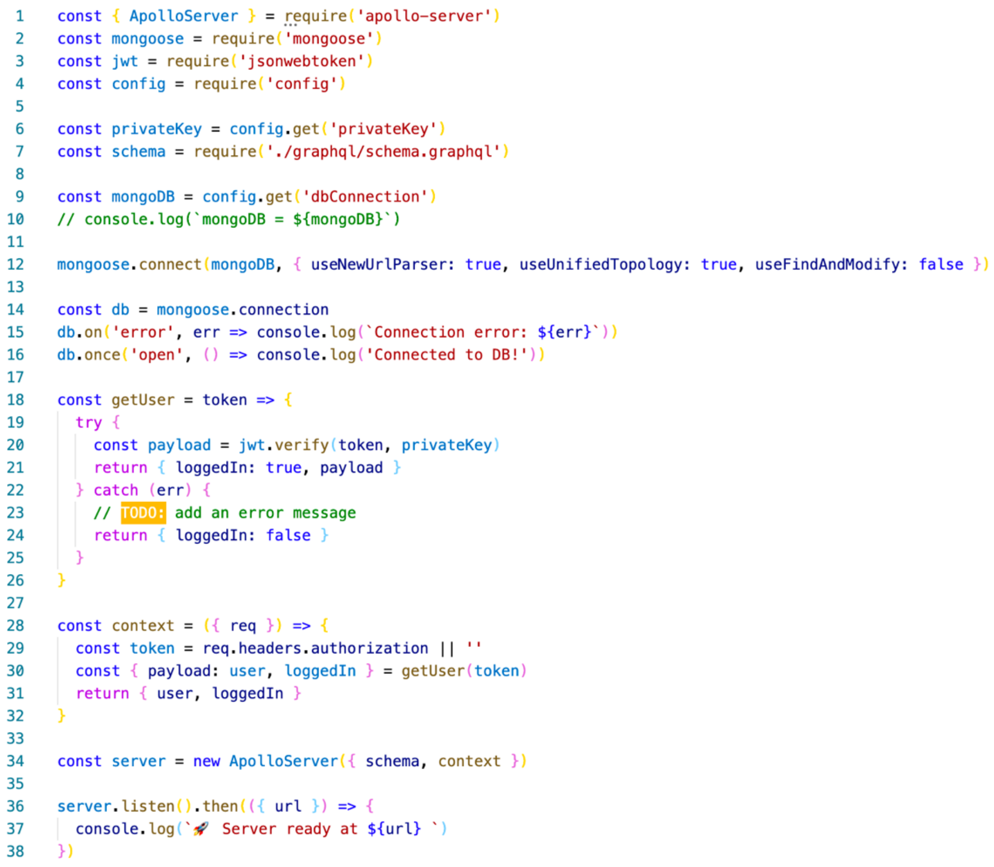
Initially, a GraphQL server's deployment was done using Express with the *express* and *express-graphql* NPM packages installed, as shown in Figure 59.



```
const express = require('express')
const graphqlHTTP = require('express-graphql')

const schema = require('./graphql/schema.js')
const app = express()
const PORT = 4000

app.use('/graphql', graphqlHTTP({
  schema,
  graphiql: true
}))

app.listen(PORT, err => {
  err ? console.log(err) : console.log('Running a GraphQL API server at http://localhost:4000/graphql')
})
```

Figure 59. Setting up an Express GraphQL server

The simplest way was to run a GraphQL server. (The GraphQL Foundation 2020c.) However, as the project grew and the requirements for it changed, it was decided to change the implementation of the server deployment from Express to Apollo Server due to the need to implement a user authentication system. As mentioned in paragraph 5.1.3, Apollo Server provides a context as an optional third argument in its constructor to be used for authentication data transfers. Figure 60 shows the deployment of the Apollo Server as a GraphQL server implementing a user authentication system.



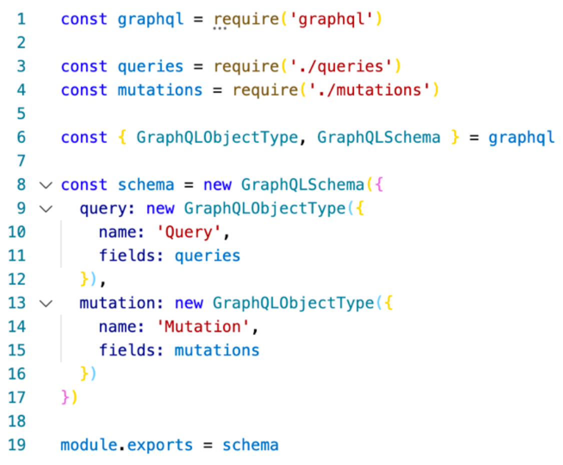
```
const { ApolloServer } = require('apollo-server')
const mongoose = require('mongoose')
const jwt = require('jsonwebtoken')
const config = require('config')

const privateKey = config.get('privateKey')
const schema = require('./graphql/schema.graphql')

const mongoDB = config.get('dbConnection')
// console.log(`mongoDB = ${mongoDB}`)

mongoose.connect(mongoDB, { useNewUrlParser: true, useUnifiedTopology: true, useFindAndModify: false })

const db = mongoose.connection
db.on('error', err => console.log(`Connection error: ${err}`))
db.once('open', () => console.log('Connected to DB!'))

const getUser = token => {
  try {
    const payload = jwt.verify(token, privateKey)
    return { loggedIn: true, payload }
  } catch (err) {
    // TODO: add an error message
    return { loggedIn: false }
  }
}

const context = ({ req }) => {
  const token = req.headers.authorization || ''
  const { payload: user, loggedIn } = getUser(token)
  return { user, loggedIn }
}

const server = new ApolloServer({ schema, context })

server.listen().then(({ url }) => {
  console.log(`🚀 Server ready at ${url} `)
})
```

Figure 60. Setting up a new GraphQL server with auth using Apollo Server

User authentication is based on passing a login token in an HTTP authorization header. The *context()* function looks at the request headers, pulls off the authorization header, and stores it to a variable. Then it calls a *getUser()* function with the *token* and expects a *user* to be returned as *payload* if the token is valid. After that, it returns a context object containing the (potential) user, for all of our resolvers to use. (Authentication n.d.)

The *getUser()* function in its implementation uses the *jwt.verify()* method of the *jsonwebtoken* library, which takes the *token* and the *privateKey* (the secret key defined in config) as arguments and returns the *payload* decoded if the signature is valid. This securely transferring information between parties as a JSON object is compact and self-contained and complies with the RFC 7519 standard. (Introduction to JSON Web Tokens n.d.)

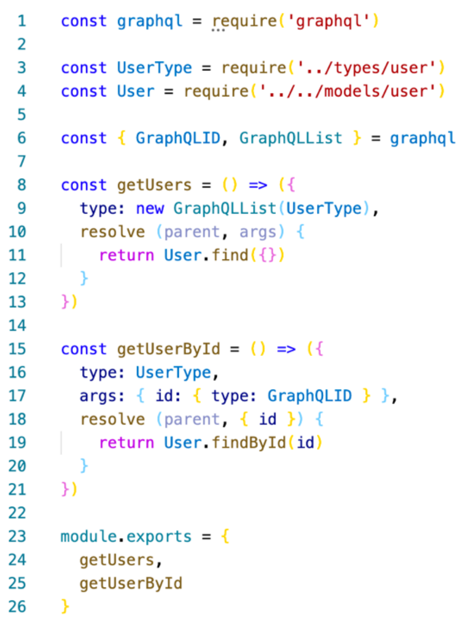
It is easy to see that regardless of how to set up the GraphQL server, the GraphQL schema is at the heart of its implementations. The schema describes the functionality available to client applications that connect to the server. To create a GraphQL schema and build an interface based on it, the project used the *graphql* module.



```
const graphql = require('graphql')

const queries = require('./queries')
const mutations = require('./mutations')

const { GraphQLObjectType, GraphQLSchema } = graphql

const schema = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: queries
  }),
  mutation: new GraphQLObjectType({
    name: 'Mutation',
    fields: mutations
  })
})

module.exports = schema
```

Figure 61. The schema defined

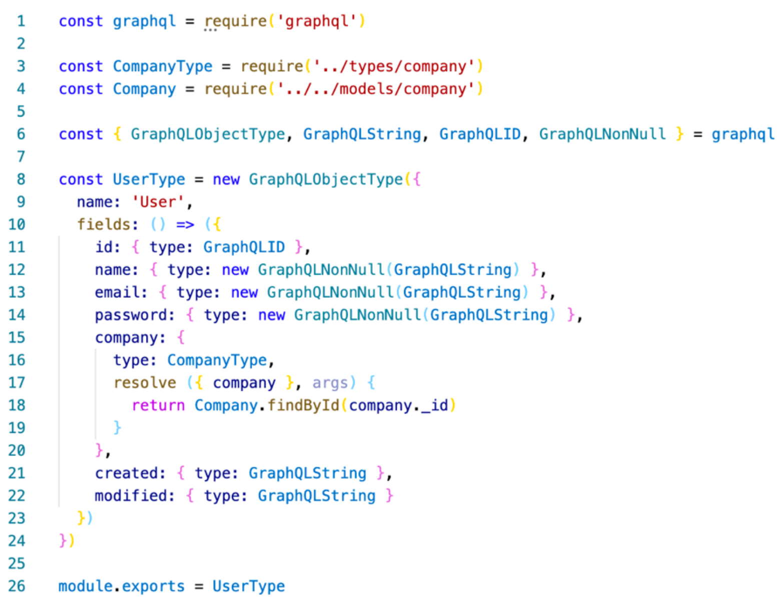
As seen in Figure 61, when creating a schema, the query and mutation fields are defined in its constructor, described using the *graphqlObjectType* class definition. The point is that it is not enough just to export the schema to the *server.js* file. When an application accesses GraphQL, the latter must request all the required data. Thus, it is necessary to create root types of each type of operation, query, and mutation (optional). (The GraphQL Foundation 2020d.) Inside the root query, all GraphQL queries implemented in the project are described in the *fields* field. A particular example of such queries during server configuration is shown in Figure 62.



```
const graphql = require('graphql')

const UserType = require('../types/user')
const User = require('../../models/user')

const { GraphQLID, GraphQLList } = graphql

const getUsers = () => ({
  type: new GraphQLList(UserType),
  resolve (parent, args) {
    return User.find({})
  }
})

const getUserById = () => ({
  type: UserType,
  args: { id: { type: GraphQLID } },
  resolve (parent, { id }) {
    return User.findById(id)
  }
})

module.exports = {
  getUsers,
  getUserById
}
```

Figure 62. Queries *getUsers(), getUserById()*

The *getUsers ()* and *getUserById()* queries are strongly typed with the *UserType* (Figure 63) and using the *resolve()* function, the *find()* and *findById() mongoose* library methods, return a specific user or all users from the database respectively.

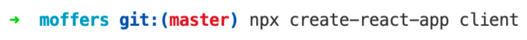


```
const graphql = require('graphql')

const CompanyType = require('../types/company')
const Company = require('../../models/company')

const { GraphQLObjectType, GraphQLString, GraphQLID, GraphQLNonNull } = graphql

const UserType = new GraphQLObjectType({
  name: 'User',
  fields: () => ({
    id: { type: GraphQLID },
    name: { type: new GraphQLNonNull(GraphQLString) },
    email: { type: new GraphQLNonNull(GraphQLString) },
    password: { type: new GraphQLNonNull(GraphQLString) },
    company: {
      type: CompanyType,
      resolve ({ company }, args) {
        return Company.findById(company._id)
      }
    },
    created: { type: GraphQLString },
    modified: { type: GraphQLString }
  })
})

module.exports = UserType
```

Figure 63. User fields type defining implemented with *UserType*

6.2.3 Client

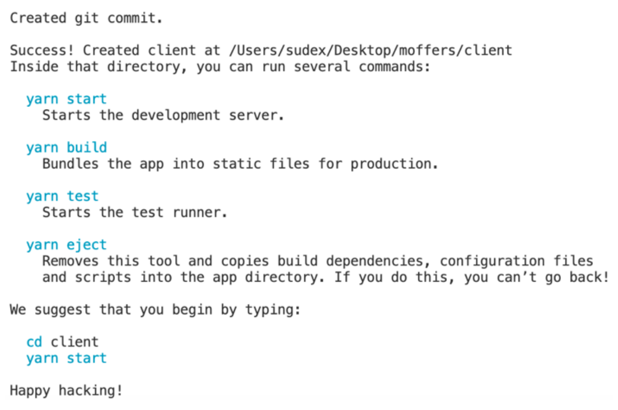
Client development with React has been deployed using a recommended command-line tool called Create React App, as shown in Figure 64. It sets up the development environment to take advantage of the latest JavaScript features, optimizes the production application, and provides comfort during development. One of this setup's benefits is not necessary to install or configure tools like webpack or Babel. They are preconfigured and hidden, allowing the developer to focus on the code. (Sozdaem novoe React-priloženie n.d.)



```
➜  moffers git:(master) npx create-react-app client
```

Figure 64. Creating the client using the Create React App tool and NPX utility

This command will install all the dependencies and packages required for the React project, including *react*, *react-dom,* and *react-scripts*. If the installation is successful, the terminal displays the following success message (Figure 65).



```
Created git commit.

Success! Created client at /Users/sudex/Desktop/moffers/client
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can’t go back!

We suggest that you begin by typing:

  cd client
  yarn start

Happy hacking!
```

Figure 65. A message after React successfully installed

The message describes the available commands for running, building, testing, and ejecting an application. The client is ready to start.

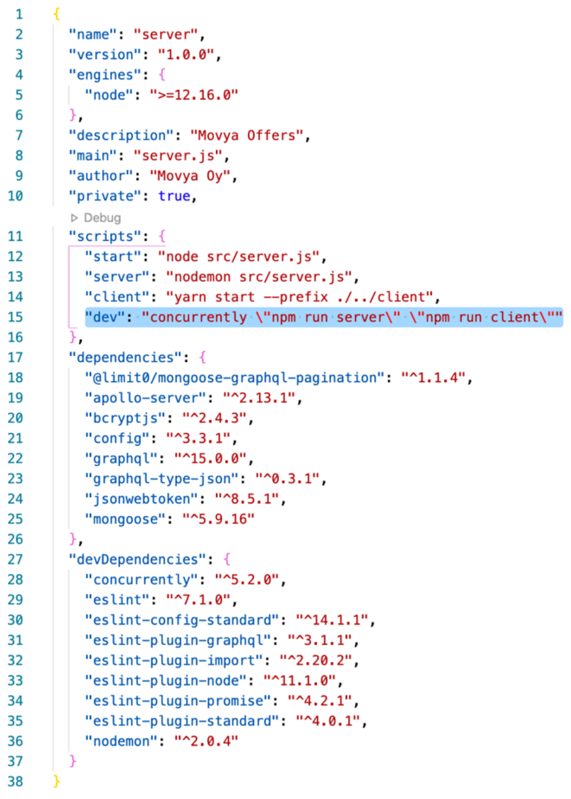
6.2.4 Nodemon

When developing Node.js applications, it is necessary to restart the server to see the changes made in action. This adds an extra step to workflow, which can be eliminated by using *nodemon*. *Nodemon* is a CLI utility that wraps a Node.js app, watches the file system any changes, and automatically restarts the server improving development productivity.

Nodemon was installed to the project locally as a development dependency using *yarn install nodemon --save-dev* command. After that, the *nodemon src/server.js* script was added to the *package.json* file in the server directory to start the server with the short command *npm server* (Figure 66).

6.2.5 Concurrently

After the server and client are configured, it is possible to run them using the *yarn start* or *npm server* commands, written using the *react-scripts start* and *nodemon src/server.js* scripts in the corresponding *package.json* files. It is needed for such running to use two separate terminals, which is inconvenient for quick development. Instead of opening two terminals and running development servers separately, the project used the NPM package *concurrently* and implemented the *concurrently \ "npm run server\" \ "npm run client\"* script that runs the servers with just one-line code *yarn run dev*. This script is also server-side in the *package.json* file (highlighted in Figure 66).

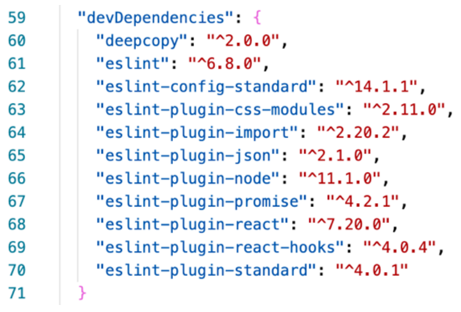


```
{
  "name": "server",
  "version": "1.0.0",
  "engines": {
    "node": ">=12.16.0"
  },
  "description": "Movya Offers",
  "main": "server.js",
  "author": "Movya Oy",
  "private": true,
  ▷ Debug
  "scripts": {
    "start": "node src/server.js",
    "server": "nodemon src/server.js",
    "client": "yarn start --prefix ./../client",
    "dev": "concurrently \"npm run server\" \"npm run client\""
  },
  "dependencies": {
    "@limit0/mongoose-graphql-pagination": "^1.1.4",
    "apollo-server": "^2.13.1",
    "bcryptjs": "^2.4.3",
    "config": "^3.3.1",
    "graphql": "^15.0.0",
    "graphql-type-json": "^0.3.1",
    "jsonwebtoken": "^8.5.1",
    "mongoose": "^5.9.16"
  },
  "devDependencies": {
    "concurrently": "^5.2.0",
    "eslint": "^7.1.0",
    "eslint-config-standard": "^14.1.1",
    "eslint-plugin-graphql": "^3.1.1",
    "eslint-plugin-import": "^2.20.2",
    "eslint-plugin-node": "^11.1.0",
    "eslint-plugin-promise": "^4.2.1",
    "eslint-plugin-standard": "^4.0.1",
    "nodemon": "^2.0.4"
  }
}
```

Figure 66. *Package.json* file in the server directory

6.2.6 Eslint

As seen in Figure 66, various *eslint* plugins are additionally installed to the server directory as development dependencies. Similar packages were also installed on the client-side (Figure 67).

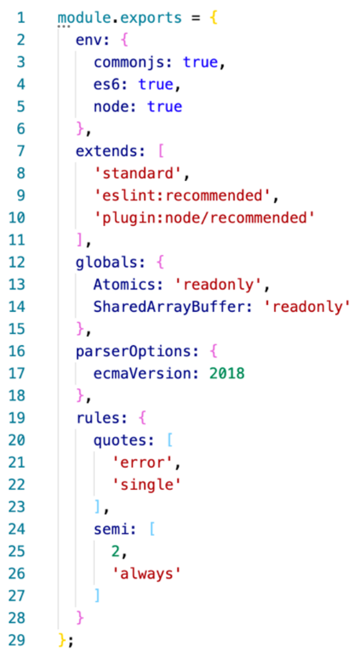


```
  "devDependencies": {
    "deepcopy": "^2.0.0",
    "eslint": "^6.8.0",
    "eslint-config-standard": "^14.1.1",
    "eslint-plugin-css-modules": "^2.11.0",
    "eslint-plugin-import": "^2.20.2",
    "eslint-plugin-json": "^2.1.0",
    "eslint-plugin-node": "^11.1.0",
    "eslint-plugin-promise": "^4.2.1",
    "eslint-plugin-react": "^7.20.0",
    "eslint-plugin-react-hooks": "^4.0.4",
    "eslint-plugin-standard": "^4.0.1"
  }
```

Figure 67. Development dependencies in the *package.json* file of the client

ESLint is a linter or static code analysis tool written in Node.js. Since JavaScript, being an interpreted programming language has no compilation step, and many errors can only be detected at runtime, using ESLint can significantly simplify development. It brings the code to a more uniform style, helps to find existing errors in the code and avoid them, can automatically fix many of the found problems and errors, and integrates perfectly with many development tools. ESLint is very flexible and customizable, and the developer can choose which rules to use or which style to apply. Many of the available rules are disabled by default, but it is possible to enable them in the *.eslintrc* configuration file, global or local to the project. (Begunov 2018.)

As seen in Figure 55, the *.eslintrc* configuration files were added to both the client and the server's root directories. The simplest of them from the server folder is shown in Figure 68.



```
module.exports = {
  env: {
    commonjs: true,
    es6: true,
    node: true
  },
  extends: [
    'standard',
    'eslint:recommended',
    'plugin:node/recommended'
  ],
  globals: {
    Atomics: 'readonly',
    SharedArrayBuffer: 'readonly'
  },
  parserOptions: {
    ecmaVersion: 2018
  },
  rules: {
    quotes: [
      'error',
      'single'
    ],
    semi: [
      2,
      'always'
    ]
  }
};
```

Figure 68. *.eslintrc* configuration file in the server directory

6.2.7 Ant Design

In today's world, with tight time constraints, whether it is advisable to spend time building each React component from scratch always arises. Various component libraries have been created to address this issue. Developers either gradually replenish such a library themselves as needed, or use some of the ready-made solutions in a reasonably large assortment of NPM modules, which should be selected based on popularity and maintenance. However, the design task is often complicated because there is no such ready-to-use library with React components, agreed with the designers, and tested on real projects at the start of the project. And what if the project has no design, no designers, no design system? Usually, in such cases, before starting a project, it is chosen in favor of using ready-made component libraries to minimize the time spent on writing their own.

Given the popularity of React, its component approach, and the considerable hype around design systems in recent years, design libraries' choice is quite broad. After analyzing some of them and completing a small test project, Ant Design was chosen as the design system for the *Moffers* project.

According to official documentation, Ant Design is "a design system for enterprise-level products". (Ant Design n.d.) With its principles, style guides, and a library of components with an impressive list and functionality, this design system has two demanded features that make it stand out among similar libraries: tables and forms. And these were precisely the complex components that were mainly used in the project.

Tables

Built-in pagination. By default, Ant Design pagination is the client-side, but it is possible to implement server-side pagination.

Filtering and sorting. Out of the box, filtering by a drop-down selector with options is available. Sorting and filtering algorithms have to be described manually. By default, tables cannot filter records by the entered line, but it is possible to write your custom filter described in detail in the documentation.

Selection of lines. If needed to ensure that specific table rows are selected for further actions by the user, Ant Design tables provide a flexible API.

Nesting. If needed to make specific table rows expandable to hide additional information, Ant Design tables can do this out of the box.

Cells merging. Cells merging in the header and the rows are different, but in both cases, it is necessary to know in advance which cells are needed to merge and specify them explicitly. It complicates the processing of dynamic data somewhat, but in principle, does not make it impossible.

Column and heading fixation. A useful feature for rendering large amounts of data. You can fix both the left and right columns, the table header, and even all together.

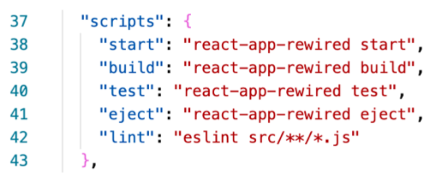
Editable cells. The API for tables is generally quite flexible and allows rendering cells in any way you want. So, the flexibility of editing cells is only a particular case of using the provided capabilities described in detail in the documentation.

Forms

The *Form* container component does not do much by itself: hide the asterisks of required fields, change the relative position of labels and fields, initialize the initial values, call the *onSubmit()*, *onFinish()*, and *onFinishFailed()* handlers. However, with the *Form.Item* component nested inside it, which takes over control of the form elements, a significant number of useful methods are added to its functionality. In the created form, you can add validation rules using simple objects, synchronize field values with the management system, store the default field values separately, and then be applied by calling just one method.

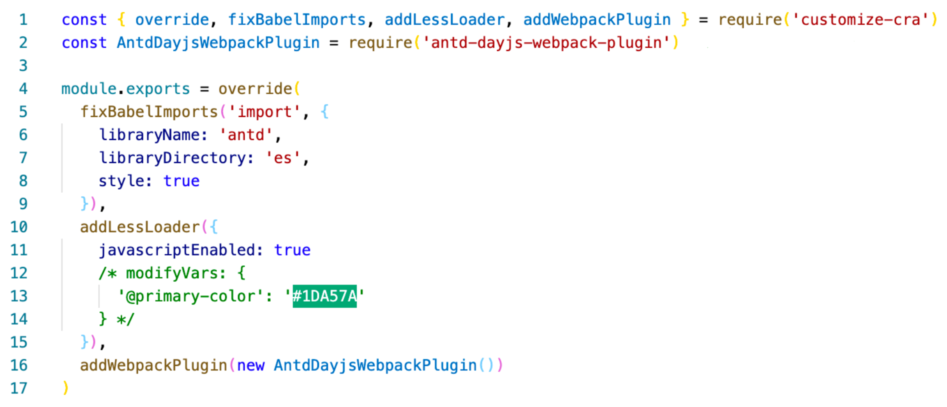
Connecting Ant Design to React

For React apps built using *Create React App* utility*,* Ant Design provides a way to connect the *antd* library to React and then modify the *webpack* config for some customizations. (Use in create-react-app n.d.) According to the documentation, the following things were installed: the *antd* NPM package, the *create-react-app* config utilities *react-app-rewired,* and *customize-cra*, the babel plugin for importing components on demand *babel-plugin-import*, and the compiler *less-loader* to customize the theme. Next, the *package.json* scripts and *config-overrides.js* were modified with these tools, as shown in Figure 69 and Figure 70.



```
"scripts": {
  "start": "react-app-rewired start",
  "build": "react-app-rewired build",
  "test": "react-app-rewired test",
  "eject": "react-app-rewired eject",
  "lint": "eslint src/**/*.js"
},
```

Figure 69. Main scripts of the *package.json* file on the client



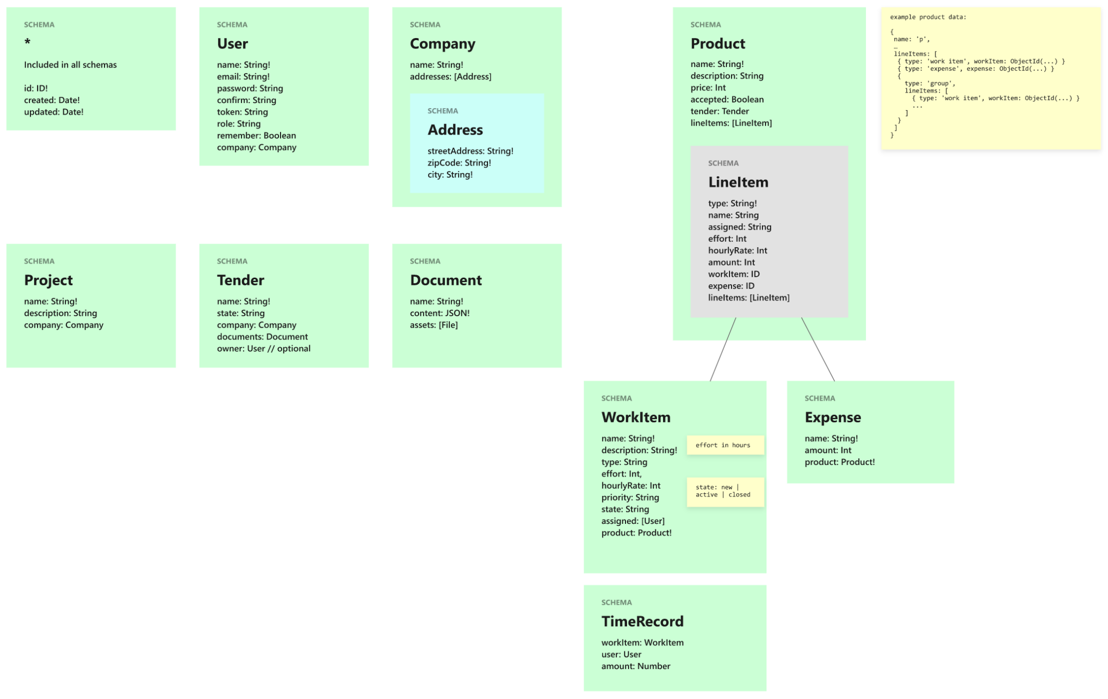
```
const { override, fixBabelImports, addLessLoader, addWebpackPlugin } = require('customize-cra')
const AntdDayjsWebpackPlugin = require('antd-dayjs-webpack-plugin')

module.exports = override(
  fixBabelImports('import', {
    libraryName: 'antd',
    libraryDirectory: 'es',
    style: true
  }),
  addLessLoader({
    javascriptEnabled: true
    /* modifyVars: {
      '@primary-color': '#1DA57A'
    } */
  }),
  addWebpackPlugin(new AntdDayjsWebpackPlugin())
)
```

Figure 70. Config settings of the *config-overrides.js file*

This sequence of actions allows setting up a React project with Ant Design, avoiding ejecting the application.

6.3 Data model

Figure 71 shows an abstract data model that describes data types, the project data-base schemas and standardizes their relationship.



```
example product data:

{
 name: 'p',
 …
 lineItems: [
  { type: 'work item', workItem: ObjectId(...) }
  { type: 'expense', expense: ObjectId(...) }
  {
    type: 'group',
    lineItems: [
      { type: 'work item', workItem: ObjectId(...) }
      ...
    ]
  }
 ]
}
```

Figure 71. Data model

In particular, the figure shows that all schema fields contain the date of creation and update, the user belongs to a specific company, which may have several addresses, and so on.

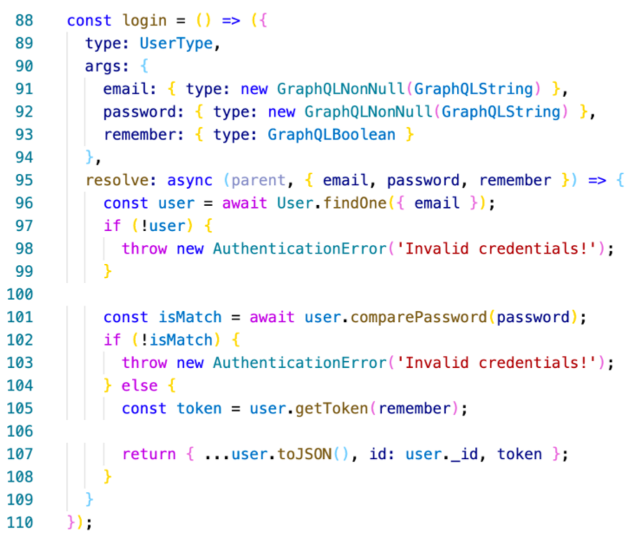
The significant volume of work put on project's implementation does not allow describing all its functionality within this thesis work. In this connection, the following sections will present the most significant parts and aspects of the project in a somewhat concise context.

6.4 User management

The web service provides a wide range of resources and access, which depends on users' access rights when registering through roles. However, before the specific user role is established, it is necessary to check whether the user is who he claims to be. Therefore, the project implemented authentication and authorization processes.

6.4.1 User authentication

To confirm the registered user's identity, token-based single-factor authentication was implemented based on checking the user's email credentials and password. Figure 72 shows the *login()* function, which implements server-side user authentication.

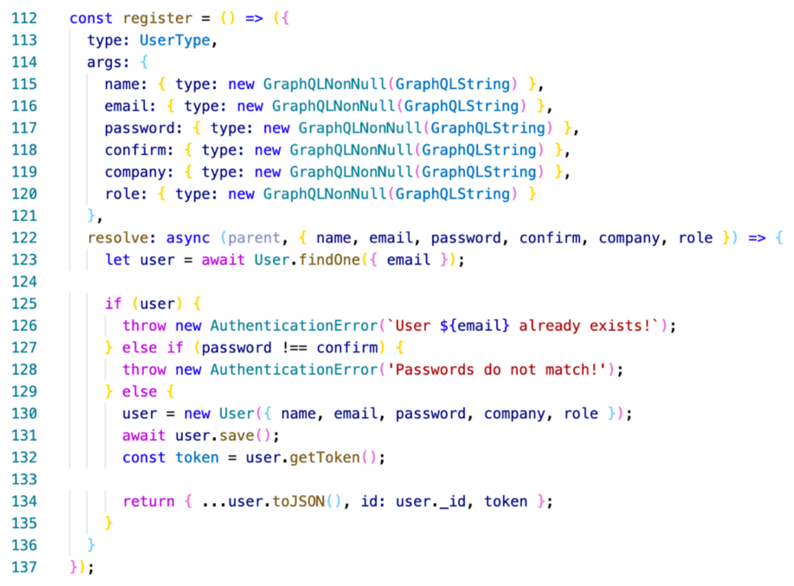


```
const login = () => ({
  type: UserType,
  args: {
    email: { type: new GraphQLNonNull(GraphQLString) },
    password: { type: new GraphQLNonNull(GraphQLString) },
    remember: { type: GraphQLBoolean }
  },
  resolve: async (parent, { email, password, remember }) => {
    const user = await User.findOne({ email });
    if (!user) {
      throw new AuthenticationError('Invalid credentials!');
    }

    const isMatch = await user.comparePassword(password);
    if (!isMatch) {
      throw new AuthenticationError('Invalid credentials!');
    } else {
      const token = user.getToken(remember);

      return { ...user.toJSON(), id: user._id, token };
    }
  }
});
```

Figure 72. User auth logic implemented in the *login()* function

This function takes *email*, *password*, and *remember* (optionally) from the context as destructured arguments and first checks if the user exists in the database (by the email). Depending on whether the user is found, the function will either return an *AuthenticationError* or check whether the entered password matches. In the end, depending on the password entry's correctness, the user will also receive an error message or be allowed to enter the system.

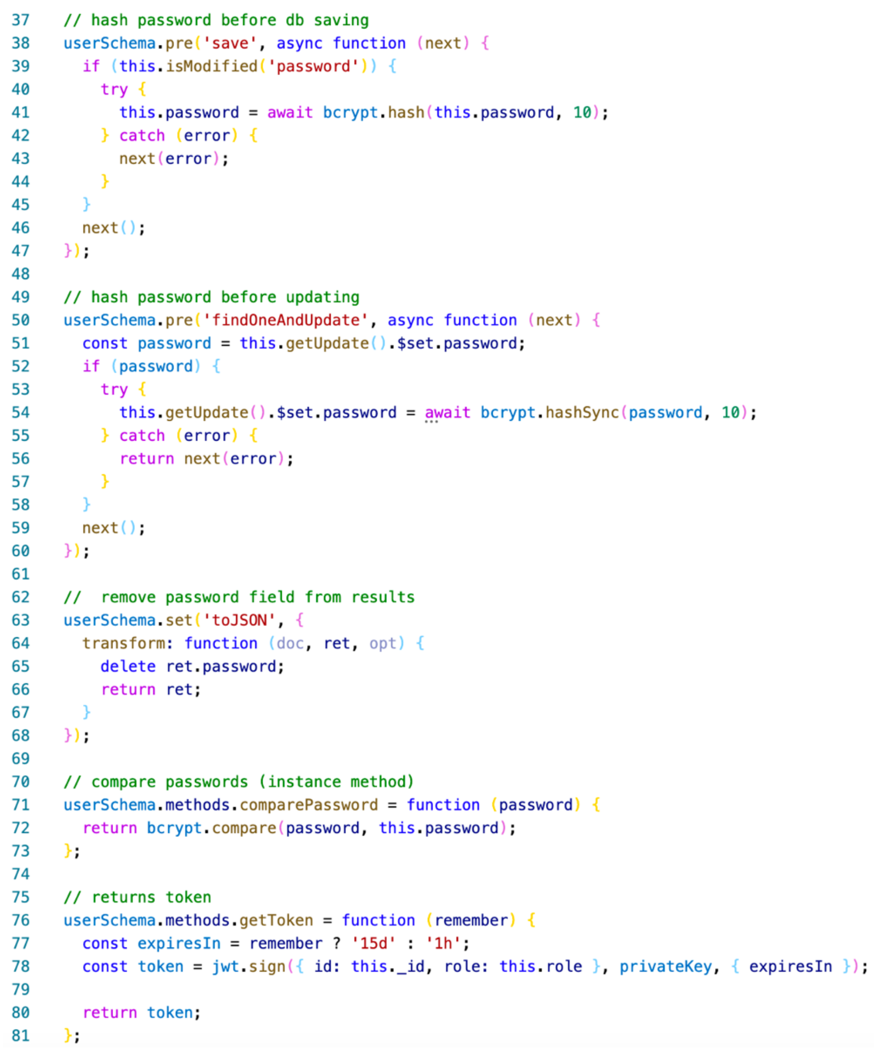
In case the user does not have credentials to enter the system, it is possible to register using the *register()* function. This function is implemented according to the principle similar to the *login()* function: first, it is checked whether there is a user with the same email in the database, then the procedure is carried out to check if the entered passwords match. If successful, the new user is saved in the database and automatically logged into the system (Figure 73).



```
const register = () => ({
  type: UserType,
  args: {
    name: { type: new GraphQLNonNull(GraphQLString) },
    email: { type: new GraphQLNonNull(GraphQLString) },
    password: { type: new GraphQLNonNull(GraphQLString) },
    confirm: { type: new GraphQLNonNull(GraphQLString) },
    company: { type: new GraphQLNonNull(GraphQLString) },
    role: { type: new GraphQLNonNull(GraphQLString) }
  },
  resolve: async (parent, { name, email, password, confirm, company, role }) => {
    let user = await User.findOne({ email });

    if (user) {
      throw new AuthenticationError(`User ${email} already exists!`);
    } else if (password !== confirm) {
      throw new AuthenticationError('Passwords do not match!');
    } else {
      user = new User({ name, email, password, company, role });
      await user.save();
      const token = user.getToken();

      return { ...user.toJSON(), id: user._id, token };
    }
  }
});
```

Figure 73. User register logic implemented in the *register()* function

The actions for accessing the database connected with finding a user, authenticating his password, and saving a new user are asynchronous and implemented using the *mongoose* and *bcryptjs* libraries' built-in methods. Additionally, before saving the user to the database or updating it, helper methods are implemented for hashing the password and removing the password field from results. Depending on the boolean value of the optional *remember* parameter, the user's session duration is determined. All helper methods are defined in the user model to facilitate interaction with the database (Figure 74).



```
// hash password before db saving
userSchema.pre('save', async function (next) {
  if (this.isModified('password')) {
    try {
      this.password = await bcrypt.hash(this.password, 10);
    } catch (error) {
      next(error);
    }
  }
  next();
});

// hash password before updating
userSchema.pre('findOneAndUpdate', async function (next) {
  const password = this.getUpdate().$set.password;
  if (password) {
    try {
      this.getUpdate().$set.password = await bcrypt.hashSync(password, 10);
    } catch (error) {
      return next(error);
    }
  }
  next();
});

//  remove password field from results
userSchema.set('toJSON', {
  transform: function (doc, ret, opt) {
    delete ret.password;
    return ret;
  }
});

// compare passwords (instance method)
userSchema.methods.comparePassword = function (password) {
  return bcrypt.compare(password, this.password);
};

// returns token
userSchema.methods.getToken = function (remember) {
  const expiresIn = remember ? '15d' : '1h';
  const token = jwt.sign({ id: this._id, role: this.role }, privateKey, { expiresIn });

  return token;
};
```

Figure 74. Helper methods of the user's *mongoose.Schema*

On the client-side, user authentication and registration are described in the corresponding form components: *LoginForm* and *RegisterForm*, the UI shown in Figure 75.

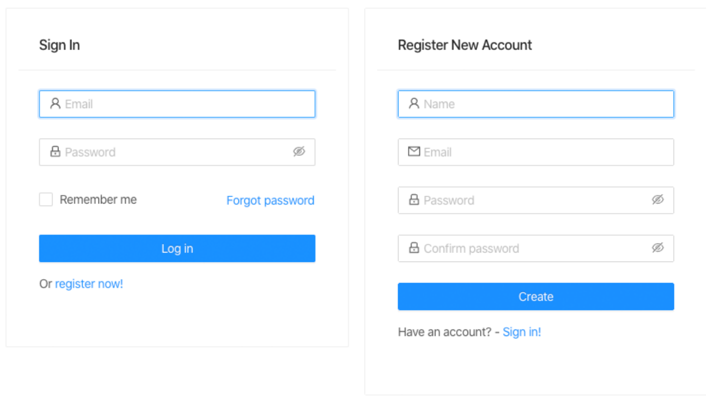
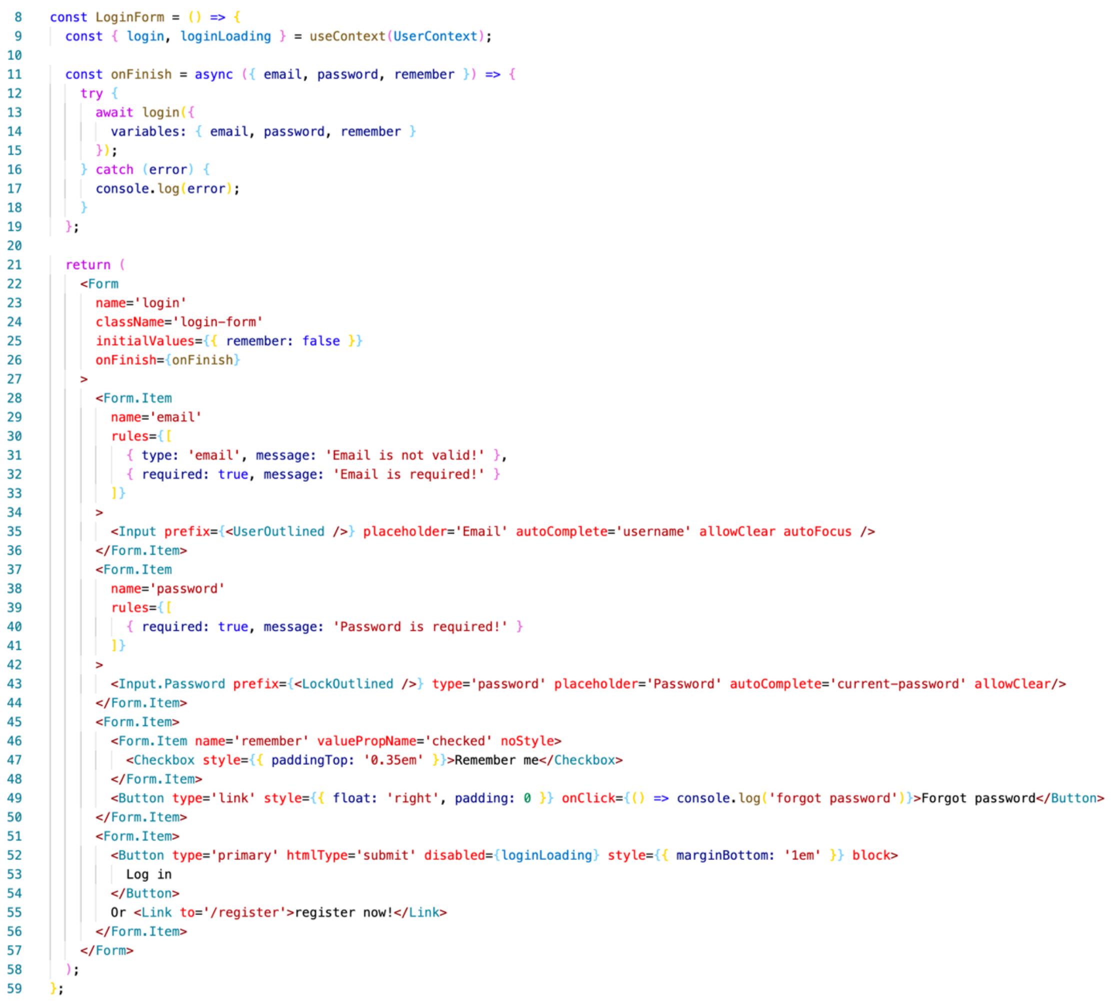


Figure 75. Login and register UI

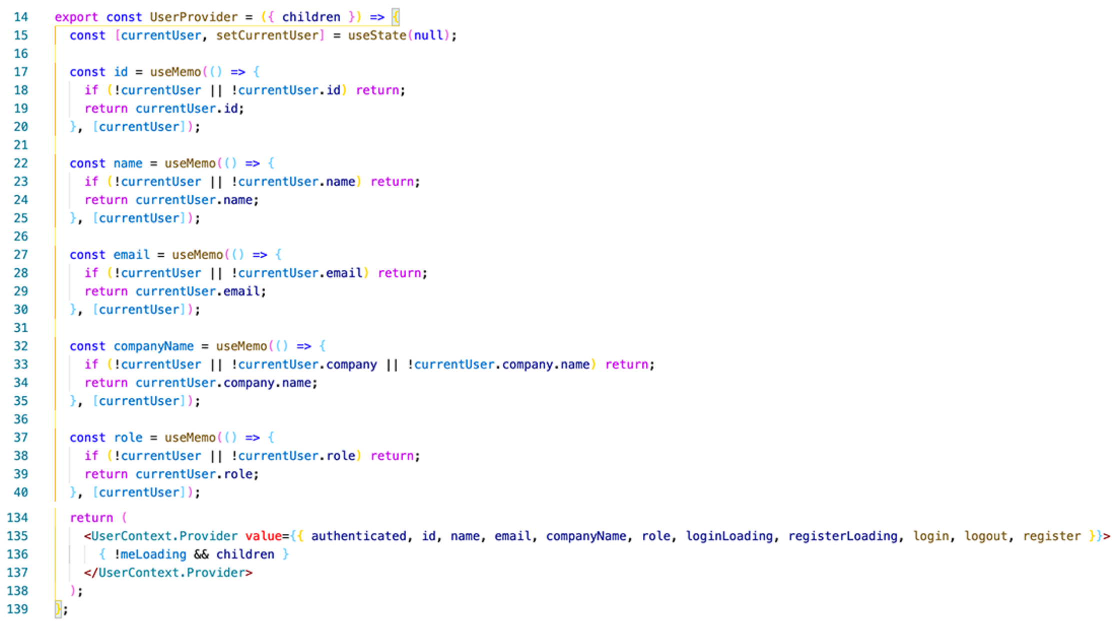
Switching between forms occurs by clicking on the appropriate link located under the button. The top input is focused by default. Form submit buttons have *onFinish()* events that take values from inputs and trigger login and registration mutations, as shown in Figure 76 for the login example.



```
const LoginForm = () => {
  const { login, loginLoading } = useContext(UserContext);

  const onFinish = async ({ email, password, remember }) => {
    try {
      await login({
        variables: { email, password, remember }
      });
    } catch (error) {
      console.log(error);
    }
  };

  return (
    <Form
      name='login'
      className='login-form'
      initialValues={{ remember: false }}
      onFinish={onFinish}
    >
      <Form.Item
        name='email'
        rules={[
          { type: 'email', message: 'Email is not valid!' },
          { required: true, message: 'Email is required!' }
        ]}
      >
        <Input prefix={<UserOutlined />} placeholder='Email' autoComplete='username' allowClear autoFocus />
      </Form.Item>
      <Form.Item
        name='password'
        rules={[
          { required: true, message: 'Password is required!' }
        ]}
      >
        <Input.Password prefix={<LockOutlined />} type='password' placeholder='Password' autoComplete='current-password' allowClear/>
      </Form.Item>
      <Form.Item>
        <Form.Item name='remember' valuePropName='checked' noStyle>
          <Checkbox style={{ paddingTop: '0.35em' }}>Remember me</Checkbox>
        </Form.Item>
        <Button type='link' style={{ float: 'right', padding: 0 }} onClick={() => console.log('forgot password')}>Forgot password</Button>
      </Form.Item>
      <Form.Item>
        <Button type='primary' htmlType='submit' disabled={loginLoading} style={{ marginBottom: '1em' }} block>
          Log in
        </Button>
        Or <Link to='/register'>register now!</Link>
      </Form.Item>
    </Form>
  );
};
```

Figure 76. *LoginForm* component

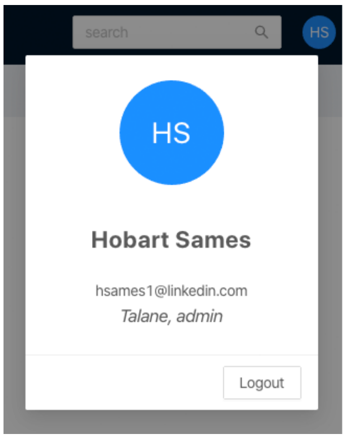
As seen from the figure, *LoginForm* gets the *login()* mutation and the *loginLoading* boolean argument, which is used to disable the button during a query to the database, from the *userContext*. All the logic for authentication, registration, and user logout is implemented using the React Context API. The current user data is provided to all components of the React application by the *userProvider*, which using the React hook *useMemo(),* calculates the memoized values, and stores it in its state (Figure 77).

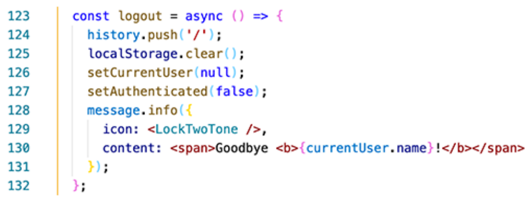


```
export const UserProvider = ({ children }) => {
  const [currentUser, setCurrentUser] = useState(null);

  const id = useMemo(() => {
    if (!currentUser || !currentUser.id) return;
    return currentUser.id;
  }, [currentUser]);

  const name = useMemo(() => {
    if (!currentUser || !currentUser.name) return;
    return currentUser.name;
  }, [currentUser]);

  const email = useMemo(() => {
    if (!currentUser || !currentUser.email) return;
    return currentUser.email;
  }, [currentUser]);

  const companyName = useMemo(() => {
    if (!currentUser || !currentUser.company || !currentUser.company.name) return;
    return currentUser.company.name;
  }, [currentUser]);

  const role = useMemo(() => {
    if (!currentUser || !currentUser.role) return;
    return currentUser.role;
  }, [currentUser]);
  return (
    <UserContext.Provider value={{ authenticated, id, name, email, companyName, role, loginLoading, registerLoading, login, logout, register }}>
      { !meLoading && children }
    </UserContext.Provider>
  );
};
```

Figure 77. Storing and passing data in the *userProvider* component

As shown in Figure 78, user data is displayed in the avatar on the header's right side, as two capital letters of his first and last name, obtained using the *getInitials()* script (Figure 79). By clicking on the avatar, a modal window opens, allowing the user to log out (Figure 78).

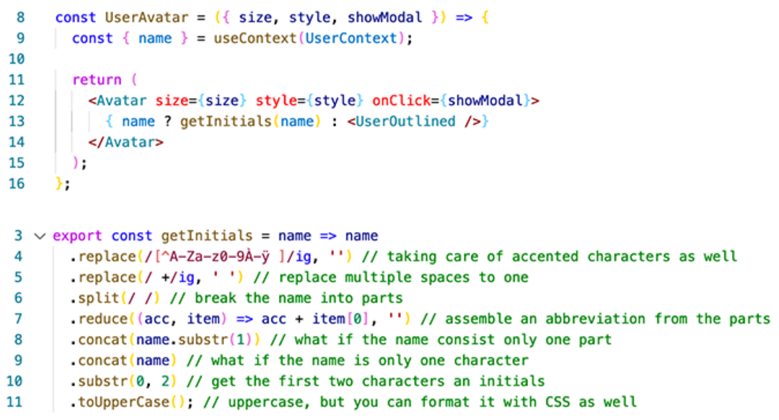




```
const logout = async () => {
  history.push('/');
  localStorage.clear();
  setCurrentUser(null);
  setAuthenticated(false);
  message.info({
    icon: <LockTwoTone />,
    content: <span>Goodbye <b>{currentUser.name}!</b></span>
  });
};
```

Figure 78. Logout: modal window and the implementing function

The *logout()* function clears the browser *localStorage* object and URL, removes all user data in the provider, and sets the user authentication flag to false.



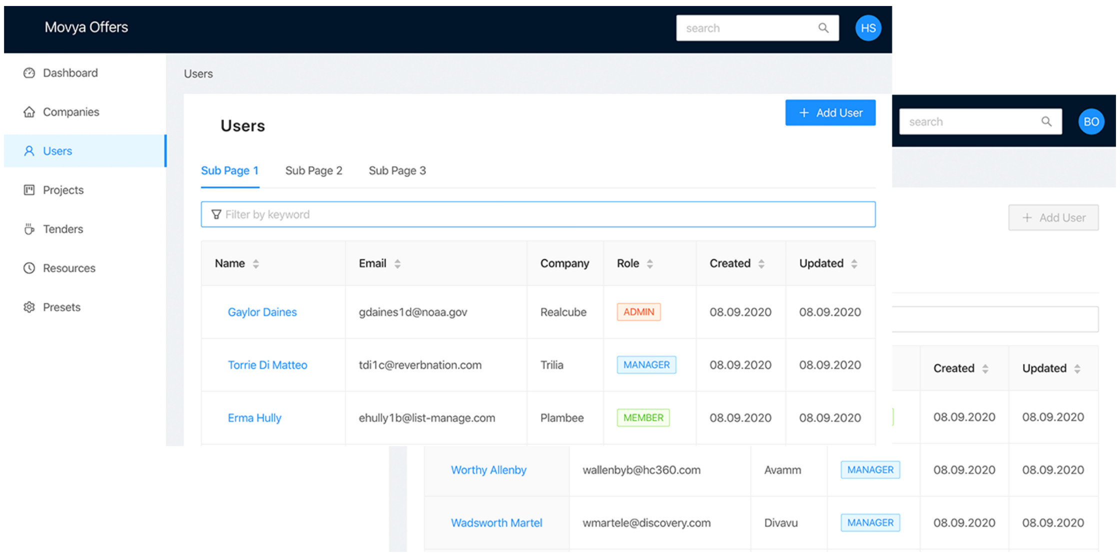
```
const UserAvatar = ({ size, style, showModal }) => {
  const { name } = useContext(UserContext);

  return (
    <Avatar size={size} style={style} onClick={showModal}>
      { name ? getInitials(name) : <UserOutlined />}
    </Avatar>
  );
};
```

```
export const getInitials = name => name
  .replace(/[^A-Za-z0-9À-ÿ ]/ig, '') // taking care of accented characters as well
  .replace(/ +/ig, ' ') // replace multiple spaces to one
  .split(/ /) // break the name into parts
  .reduce((acc, item) => acc + item[0], '') // assemble an abbreviation from the parts
  .concat(name.substr(1)) // what if the name consist only one part
  .concat(name) // what if the name is only one character
  .substr(0, 2) // get the first two characters an initials
  .toUpperCase(); // uppercase, but you can format it with CSS as well
```

Figure 79. *UserAvatar* component and *getInitial()* function logic

6.4.2 User authorization

After the system has successfully authenticated the user's identity, the authorization process determines whether the authenticated user has access to specific information resources and rights to perform certain actions. Any registered user of the web service has the role: *admin*, *manager*, *member*, or *guest* (by default). Depending on the given role, users have different capabilities. Figure 80 shows the user page UI for two different user types, *admin* and *member*.



Name	Email	Company	Role	Created	Updated
Gaylor Daines	gdaines1d@noaa.gov	Realcube	ADMIN	08.09.2020	08.09.2020
Torrie Di Matteo	tdi1c@reverbnation.com	Trilia	MANAGER	08.09.2020	08.09.2020
Erma Hully	ehully1b@list-manage.com	Plambee	MEMBER	08.09.2020	08.09.2020
Worthy Allenby	wallenbyb@hc360.com	Avamm	MANAGER	08.09.2020	08.09.2020
Wadsworth Martel	wmartele@discovery.com	Divavu	MANAGER	08.09.2020	08.09.2020

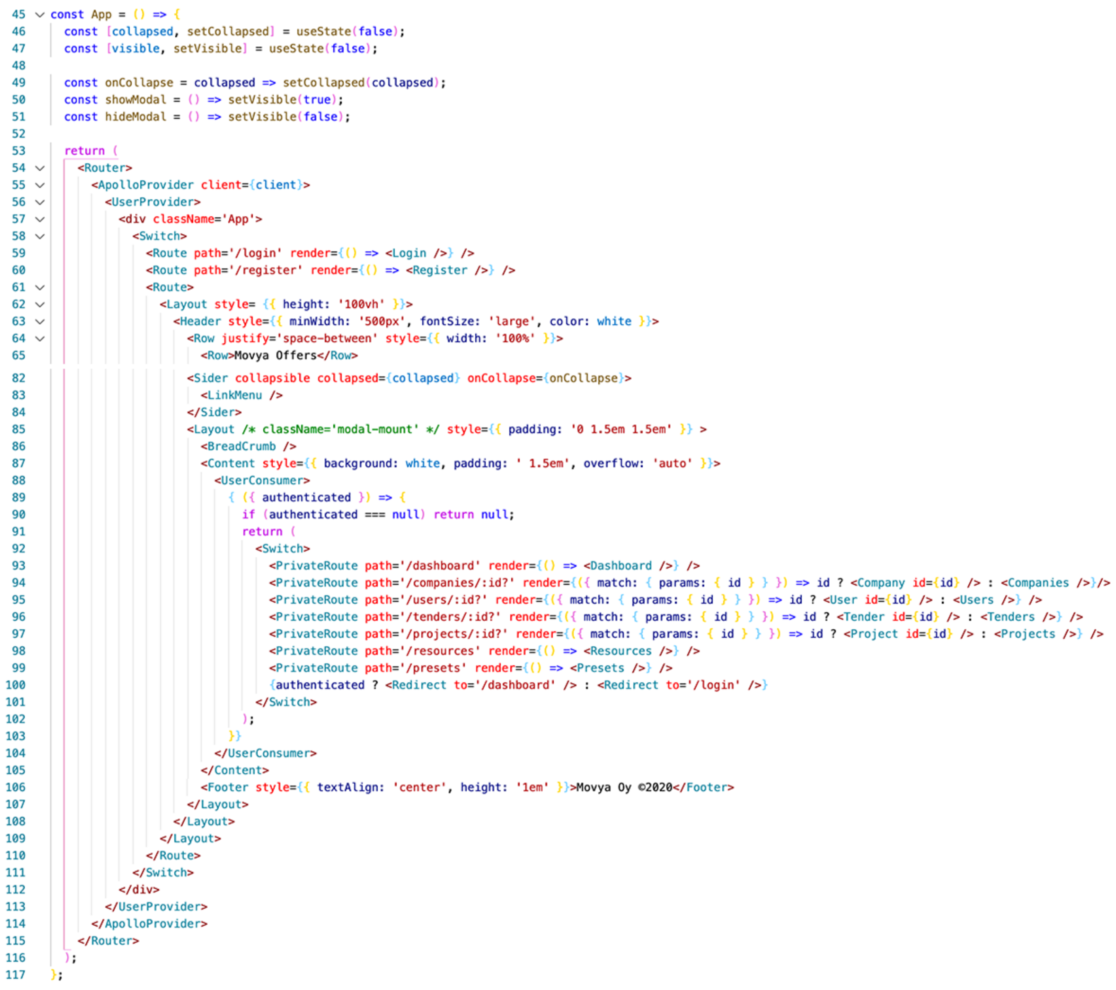
Figure 80. The user page UI for different roles

As seen in the figure, the *HS* user, unlike the *BO* user, whose *+ Add User* button is blocked, can add a user to the table.

6.5 Routing

Routing in a SPA application is used to load certain parts depending on the URL defined in the browser's address bar. React does not include a built-in implementation of routing; this is the task of special libraries. The project used the most popular solution for adding routing to a React application - React Router. It helps keep the UI in sync with the URL by using route components at any nesting level. When the URL changes, React Router will automatically mount and unmount the required components. Since React Router is an external library, it must be installed as an NPM dependency.

After adding the React Router, the *App.js* app's root React component was wrapped in a *Router* component that defines a set of routes and maps the request to the routes (Figure 81).



```
const App = () => {
  const [collapsed, setCollapsed] = useState(false);
  const [visible, setVisible] = useState(false);

  const onCollapse = collapsed => setCollapsed(collapsed);
  const showModal = () => setVisible(true);
  const hideModal = () => setVisible(false);

  return (
    <Router>
      <ApolloProvider client={client}>
        <UserProvider>
          <div className='App'>
            <Switch>
              <Route path='/login' render={() => <Login />} />
              <Route path='/register' render={() => <Register />} />
              <Route>
                <Layout style= {{ height: '100vh' }}>
                  <Header style={{ minWidth: '500px', fontSize: 'large', color: white }}>
                    <Row justify='space-between' style={{ width: '100%' }}>
                      <Row>Movya Offers</Row>
                    <Sider collapsible collapsed={collapsed} onCollapse={onCollapse}>
                      <LinkMenu />
                    </Sider>
                    <Layout /* className='modal-mount' */ style={{ padding: '0 1.5em 1.5em' }} >
                      <BreadCrumb />
                      <Content style={{ background: white, padding: ' 1.5em', overflow: 'auto' }}>
                        <UserConsumer>
                          { ({ authenticated }) => {
                            if (authenticated === null) return null;
                            return (
                              <Switch>
                                <PrivateRoute path='/dashboard' render={() => <Dashboard />} />
                                <PrivateRoute path='/companies/:id?' render={({ match: { params: { id } } }) => id ? <Company id={id} /> : <Companies />}/>
                                <PrivateRoute path='/users/:id?' render={({ match: { params: { id } } }) => id ? <User id={id} /> : <Users />} />
                                <PrivateRoute path='/tenders/:id?' render={({ match: { params: { id } } }) => id ? <Tender id={id} /> : <Tenders />} />
                                <PrivateRoute path='/projects/:id?' render={({ match: { params: { id } } }) => id ? <Project id={id} /> : <Projects />} />
                                <PrivateRoute path='/resources' render={() => <Resources />} />
                                <PrivateRoute path='/presets' render={() => <Presets />} />
                                {authenticated ? <Redirect to='/dashboard' /> : <Redirect to='/login' />}
                              </Switch>
                            );
                          }}
                        </UserConsumer>
                      </Content>
                      <Footer style={{ textAlign: 'center', height: '1em' }}>Movya Oy ©2020</Footer>
                    </Layout>
                  </Layout>
                </Layout>
              </Route>
            </Switch>
          </div>
        </UserProvider>
      </ApolloProvider>
    </Router>
  );
};
```

Figure 81. Main routing

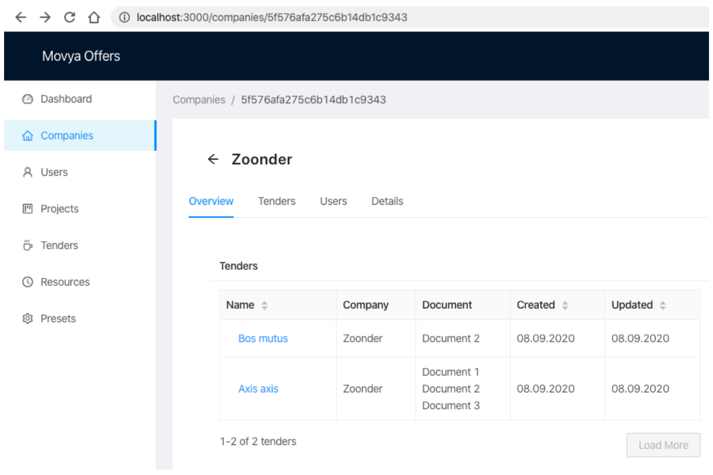
To select a route, a *Switch* object is defined, which allows choosing the first available route and using it for processing. Without this object, the *Router* can use multiple routes to process the same request if they match the request string. Each route represents a *Route* object. It has several attributes. In particular, two attributes were set here: *path* and *render*. The *path* argument is an address pattern against which the requested URL will be matched. The *render* argument renders the component responsible for processing the request along the given route.

The *LinkMenu* component is assembled from *Link* components used in the routing system to create links with a given *href* (Figure 82).

```
const LinkMenu = () => {
  const { pathname } = useLocation();

  const selectedKeys = () => {
    const pathSnippets = pathname.split('/').filter(i => i);
    return `/${[pathSnippets[0]]}`;
  };

  return (
    <Menu
      mode='inline'
      defaultSelectedKeys={['/dashboard']}
      selectedKeys={selectedKeys()}
      style={{ height: '100%' }}
    >
      <Menu.Item key='/dashboard'>
        <Link to='/dashboard'>
          <DashboardOutlined />
          <span>Dashboard</span>
        </Link>
      </Menu.Item>
      <Menu.Item key='/companies'>
        <Link to='/companies'>
          <HomeOutlined />
          <span>Companies</span>
        </Link>
      </Menu.Item>
      <Menu.Item key='/users'>
        <Link to='/users'>
          <UserOutlined />
          <span>Users</span>
        </Link>
      </Menu.Item>

      </Menu.Item>
      <Menu.Item key='/presets'>
        <Link to='/presets'><SettingOutlined />
          <span>Presets</span></Link>
      </Menu.Item>
    </Menu>
  );
};
```

Figure 82. *LinkMenu* component

This is not the only navigation method implemented in the project, but one of the main ones is through the sidebar (Figure 83).



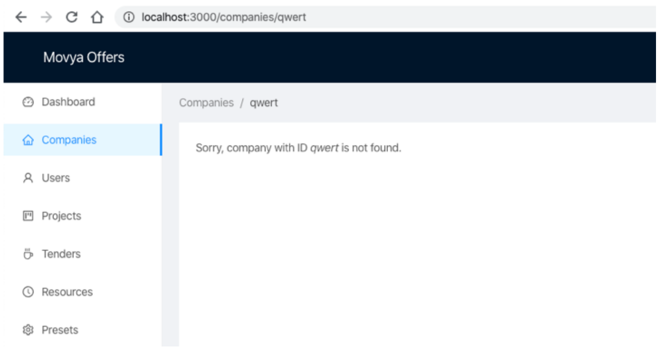
Tenders

Name	Company	Document	Created	Updated
Bos mutus	Zoonder	Document 2	08.09.2020	08.09.2020
Axis axis	Zoonder	Document 1 Document 2 Document 3	08.09.2020	08.09.2020

1-2 of 2 tenders

Figure 83. Specific company routing via the sidebar

Requests for all other URLs that do not match the sidebar routes are redirected to the *Dashboard* page (the Web Service home page). If it does not find an object by its identifier, the user will receive a corresponding message (Figure 84)



Sorry, company with ID *qwert* is not found.

Figure 84. Routing with incorrect ID

6.6 Content presentation and manipulation

Data presentation is often connected with procedures of pagination, sorting, and filtering. In a SPA, it is important to avoid, if possible, requesting the same data multiple times. A data caching mechanism solves this issue. Both data manipulation and presentation affect how the data is cached.

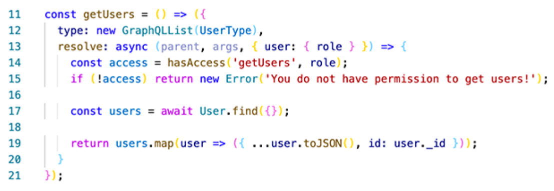
6.6.1 Pagination

The web service works with significant amounts of data, the size of which can reach, for example, a thousand rows in tables. It doesn't make sense to upload all of this data to the frontend since the user won't view most of it. Besides, loading and displaying a big block of information takes too long. In this connection, server-side pagination was implemented in the project, which implies loading data in parts and page-by-page display, while ensuring faster loading of the initial page.

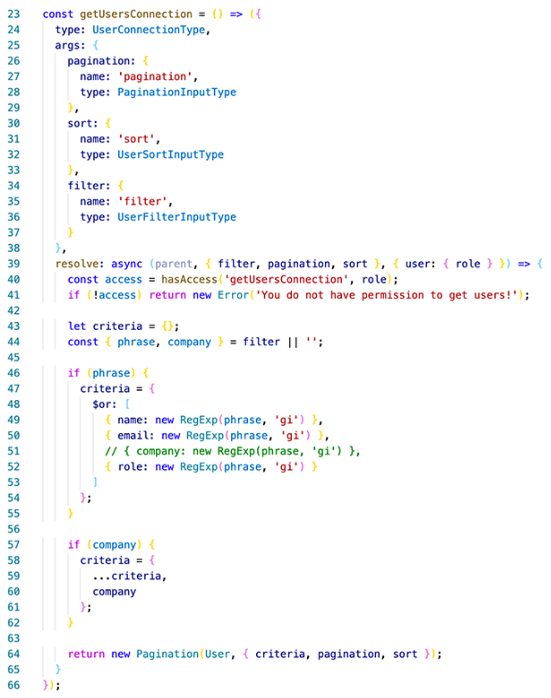
GraphQL provides various pagination models to enable different client capabilities: *Plurals*, *Slicing*, *Pagination and Edges*, *End-of-list, counts, and Connections*, *Complete Connection Model* (The GraphQL Foundation 2020e.) The project implemented the last of the above pagination models as the most complex and advanced in functionality.

This model's design allows the client to paginate through the list, ask for information about the connection itself, ask for information about the edge itself, and change how backend pagination since the user uses opaque cursors. (The GraphQL Foundation 2020e.)

Unfortunately, GraphQL does not provide a ready-made solution for implementing the *Complete Connection Model* of pagination, describing only its pattern in a standardized way. (GraphQL Cursor Connections Specification n.d.) Implementing such a solution from scratch is quite tricky. Therefore, a search was done for libraries that endow the application with cursor-based pagination functionality, as a result of which the NPM package *@limit0/mongoose-graphql-pagination* was added to the project. This library supports Relay style cursor pagination with Mongoose models/documents and also provides type-ahead (autocomplete) functionality using MongoDB regex queries. With its help, for example, the *getUsers()* query has changed as follows (Figure 85).



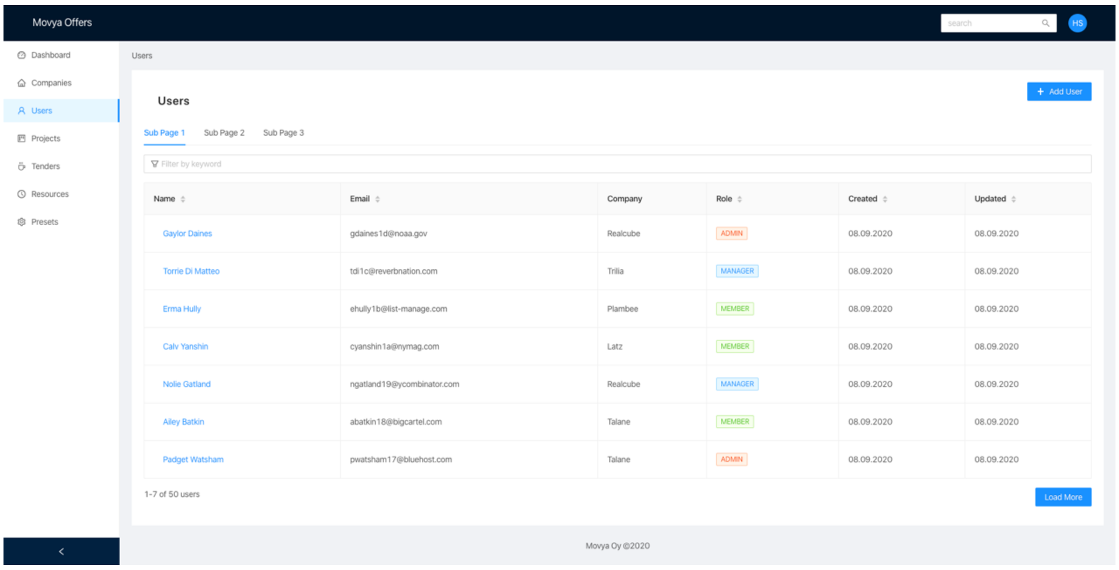
```
const getUsers = () => ({
  type: new GraphQLList(UserType),
  resolve: async (parent, args, { user: { role } }) => {
    const access = hasAccess('getUsers', role);
    if (!access) return new Error('You do not have permission to get users!');

    const users = await User.find({});

    return users.map(user => ({ ...user.toJSON(), id: user._id }));
  }
});
```



```
const getUsersConnection = () => ({
  type: UserConnectionType,
  args: {
    pagination: {
      name: 'pagination',
      type: PaginationInputType
    },
    sort: {
      name: 'sort',
      type: UserSortInputType
    },
    filter: {
      name: 'filter',
      type: UserFilterInputType
    }
  },
  resolve: async (parent, { filter, pagination, sort }, { user: { role } }) => {
    const access = hasAccess('getUsersConnection', role);
    if (!access) return new Error('You do not have permission to get users!');

    let criteria = {};
    const { phrase, company } = filter || '';

    if (phrase) {
      criteria = {
        $or: [
          { name: new RegExp(phrase, 'gi') },
          { email: new RegExp(phrase, 'gi') },
          // { company: new RegExp(phrase, 'gi') },
          { role: new RegExp(phrase, 'gi') }
        ]
      };
    }

    if (company) {
      criteria = {
        ...criteria,
        company
      };
    }

    return new Pagination(User, { criteria, pagination, sort });
  }
});
```

Figure 85. Users query before (right) and after (left), making changes

On the initial page load, for example, users' page, the React requests the API for an initial display of the data, with sort and filter parameters reset (Figure 86).



Name	Email	Company	Role	Created	Updated
Gaylor Daines	gdaines1d@noaa.gov	Realcube	ADMIN	08.09.2020	08.09.2020
Torrie Di Matteo	tdi1c@reverbnation.com	Trilia	MANAGER	08.09.2020	08.09.2020
Erma Hully	ehully1b@list-manage.com	Plambee	MEMBER	08.09.2020	08.09.2020
Calv Yanshin	cyanshin1a@nymag.com	Latz	MEMBER	08.09.2020	08.09.2020
Nolie Gatland	ngatland19@ycombinator.com	Realcube	MANAGER	08.09.2020	08.09.2020
Ailey Batkin	abatkin18@bigcartel.com	Talane	MEMBER	08.09.2020	08.09.2020
Padget Watsham	pwatsham17@bluehost.com	Talane	ADMIN	08.09.2020	08.09.2020

1-7 of 50 users

Figure 86. Users' page initially loaded

As a result, the standard pagination with numbered pages provided to the Ant Design library's client did not find applicability in the project. It may have problems redrawing the component after adding or removing data. The default number of entries for the users' table is set to 7 in the *defaultRows* variable. To request additional data, the UI provides the *Load More* button, clicking on which increases the *first* variable by the value of the *defaultRows* variable (Figure 87).

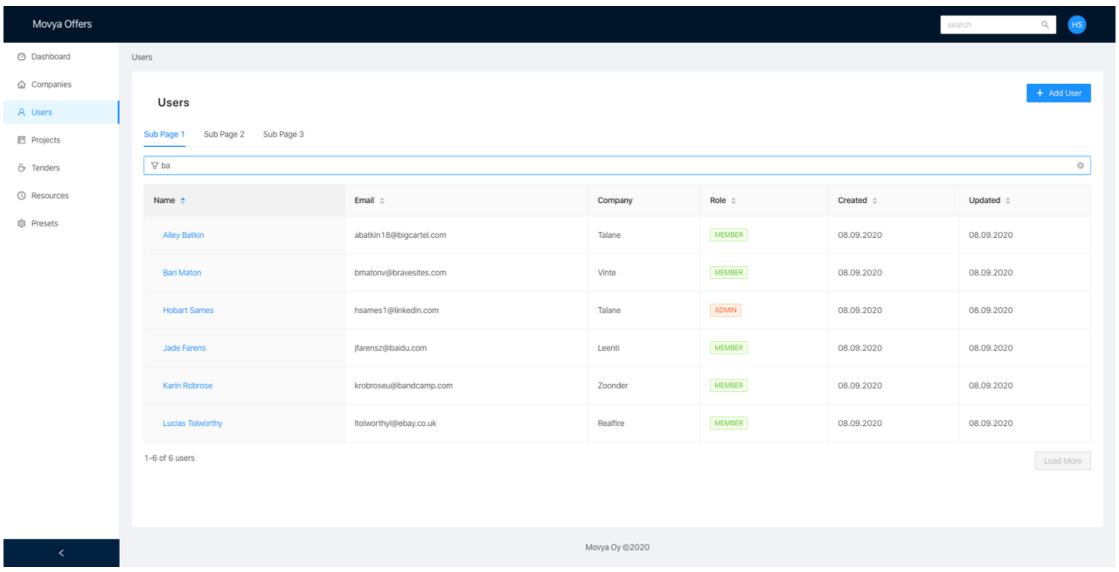
```
export const loadMore = (variables, connection, connectionName, fetchMore, setVariables) => {
  const { first, phrase } = variables;
  const { endCursor: after } = connection.pageInfo;

  fetchMore({
    variables: { after, phrase },
    updateQuery: (previousResult, { fetchMoreResult }) => {
      fetchMoreResult[connectionName].edges = [
        ...previousResult[connectionName].edges,
        ...fetchMoreResult[connectionName].edges
      ];
      return fetchMoreResult;
    }
  });

  setVariables({ ...variables, first: first + defaltRows, after });
};
```

Figure 87. Additional data loading function *loadMore()*

6.6.2 Sorting and filtering

An additional advantage of the modified *getUsersConnection()* query (Figure 85) to the implemented pagination is to sort and filter the data, using the parameters, which are optionally passed from the client in the *args* object.

The data sorting implementation on the UI is implemented using the Ant Design library tables' standard feature, the data filtering – using the input located above the table. Figure 88 shows an ascending alphabetical sorting on the *Name* column while filtering on the phrase "ba".

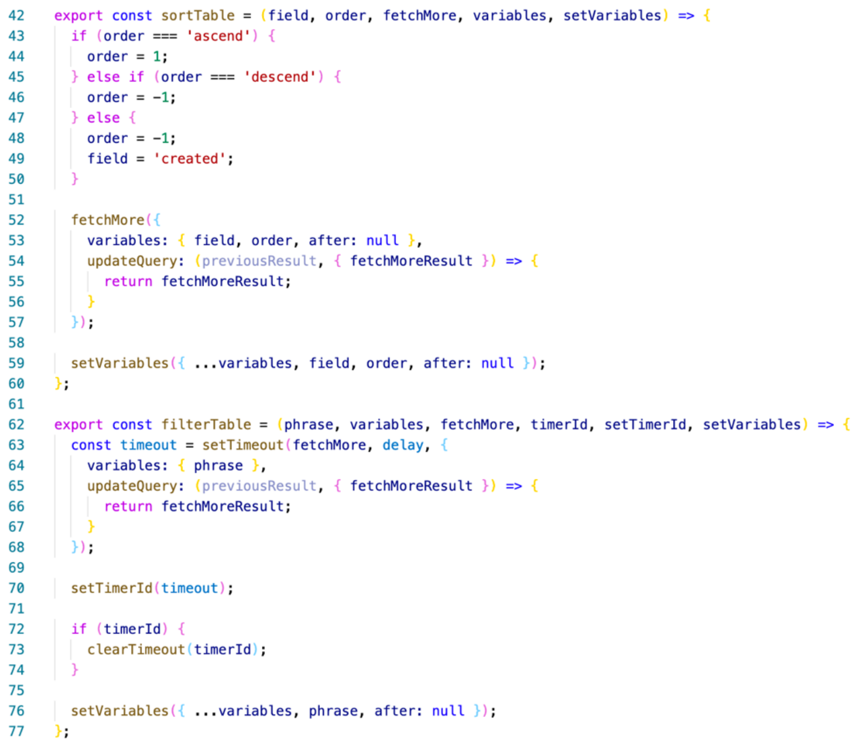


Name	Email	Company	Role	Created	Updated
Ailey Batkin	abatkin18@bigcartel.com	Talane	MEMBER	08.09.2020	08.09.2020
Bari Maton	bmatonv@bravesites.com	Vinte	MEMBER	08.09.2020	08.09.2020
Hobart Sames	hsames1@linkedin.com	Talane	ADMIN	08.09.2020	08.09.2020
Jade Farens	jfarensz@baidu.com	Leenti	MEMBER	08.09.2020	08.09.2020
Karin Robrose	krobroseu@bandcamp.com	Zoonder	MEMBER	08.09.2020	08.09.2020
Lucias Tolworthy	ltolworthyl@ebay.co.uk	Realfire	MEMBER	08.09.2020	08.09.2020

1-6 of 6 users

Figure 88. Sorting and filtering a table in UI

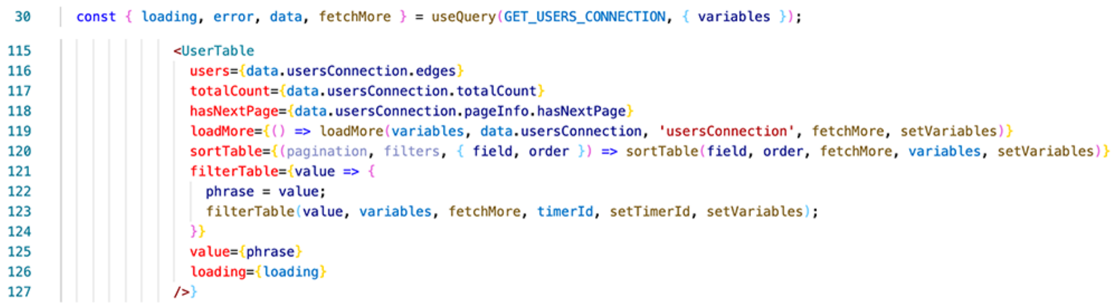
The table is filtered automatically after the user input a character or phrase. To reduce the number of unnecessary requests to the server, filtering is performed after an input pause of 250 milliseconds. This value is defined in the *delay* variable. Also, as seen in the figure, when the amount of requested data fits on one page, the *Load more* button disables. The implementation of sorting and filtering data functions is shown in Figure 85.



```
export const sortTable = (field, order, fetchMore, variables, setVariables) => {
  if (order === 'ascend') {
    order = 1;
  } else if (order === 'descend') {
    order = -1;
  } else {
    order = -1;
    field = 'created';
  }

  fetchMore({
    variables: { field, order, after: null },
    updateQuery: (previousResult, { fetchMoreResult }) => {
      return fetchMoreResult;
    }
  });

  setVariables({ ...variables, field, order, after: null });
};

export const filterTable = (phrase, variables, fetchMore, timerId, setTimerId, setVariables) => {
  const timeout = setTimeout(fetchMore, delay, {
    variables: { phrase },
    updateQuery: (previousResult, { fetchMoreResult }) => {
      return fetchMoreResult;
    }
  });

  setTimerId(timeout);

  if (timerId) {
    clearTimeout(timerId);
  }

  setVariables({ ...variables, phrase, after: null });
};
```

Figure 89. Sorting and filtering functions

As shown in Figures 87 and 89, the pagination, sorting, and filtering functions in their implementation use a specific function *fetchMore*(). This function is provided by the Apollo Client and included in the result object returned by the *useQuery()* hook. This allows executing a GraphQL query and merging the result with the original using specified variables (Figure 90).

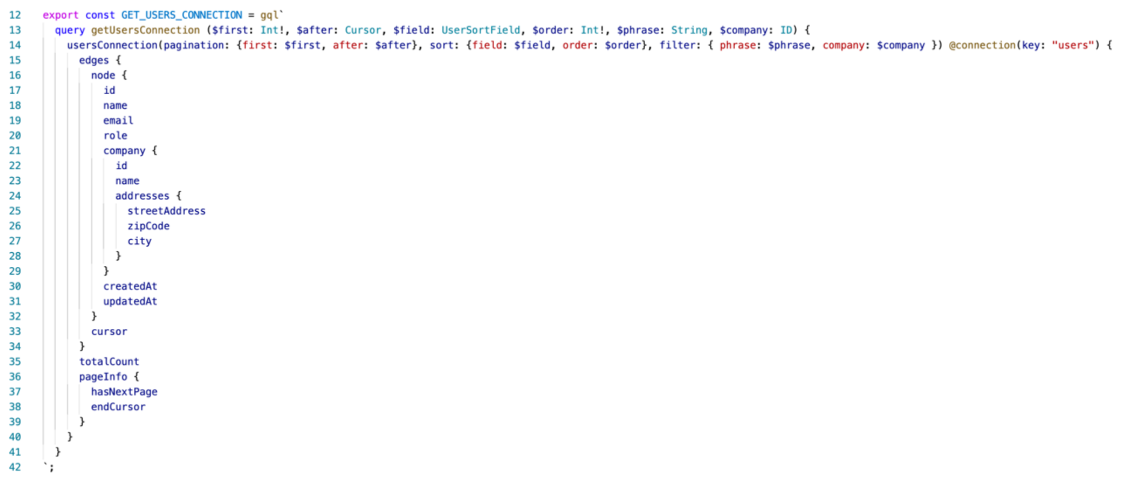


```
const { loading, error, data, fetchMore } = useQuery(GET_USERS_CONNECTION, { variables });

              <UserTable
                users={data.usersConnection.edges}
                totalCount={data.usersConnection.totalCount}
                hasNextPage={data.usersConnection.pageInfo.hasNextPage}
                loadMore={() => loadMore(variables, data.usersConnection, 'usersConnection', fetchMore, setVariables)}
                sortTable={(pagination, filters, { field, order }) => sortTable(field, order, fetchMore, variables, setVariables)}
                filterTable={value => {
                  phrase = value;
                  filterTable(value, variables, fetchMore, timerId, setTimerId, setVariables);
                }}
                value={phrase}
                loading={loading}
              />}
```

Figure 90. The *fetchMore()* function using

6.6.3 Apollo cache updating

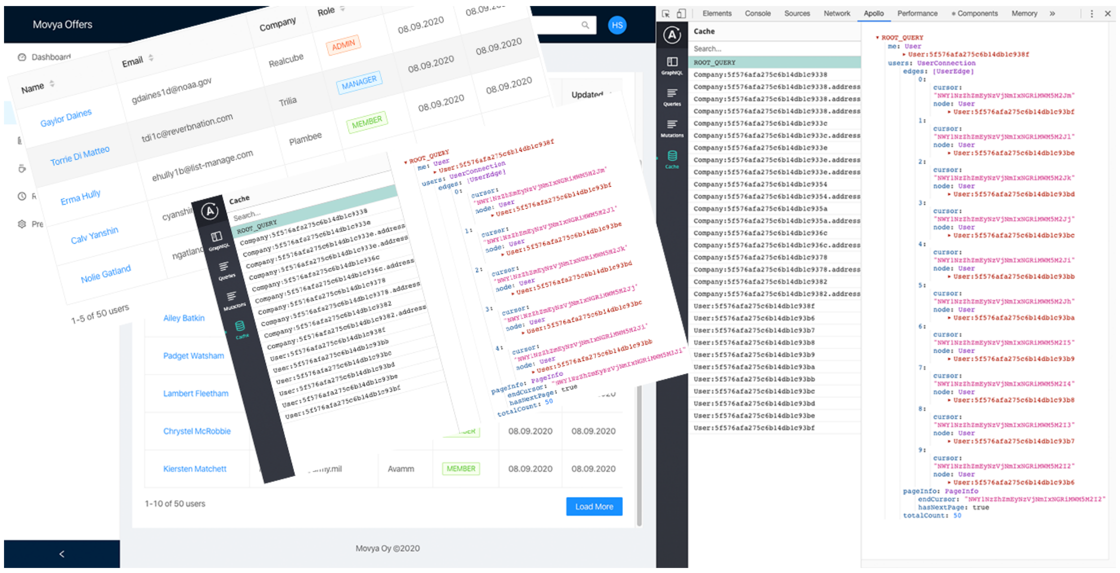
To ensure that the Apollo cache is updated correctly when additional data loads, sorts, or filters are performed, the query's stable storage key should be specified using the @connection directive (Figure 91).



```
export const GET_USERS_CONNECTION = gql`
  query getUsersConnection ($first: Int!, $after: Cursor, $field: UserSortField, $order: Int!, $phrase: String, $company: ID) {
    usersConnection(pagination: {first: $first, after: $after}, sort: {field: $field, order: $order}, filter: { phrase: $phrase, company: $company }) @connection(key: "users") {
      edges {
        node {
          id
          name
          email
          role
          company {
            id
            name
            addresses {
              streetAddress
              zipCode
              city
            }
          }
          createdAt
          updatedAt
        }
        cursor
      }
      totalCount
      pageInfo {
        hasNextPage
        endCursor
      }
    }
  }
`;
```

Figure 91. Using the *@connection* directive with the *users* key

This would result in the accumulated users in every query or more data loading placed in the cache under the *users* key, which could later be used for imperative cache updates (Figure 92).



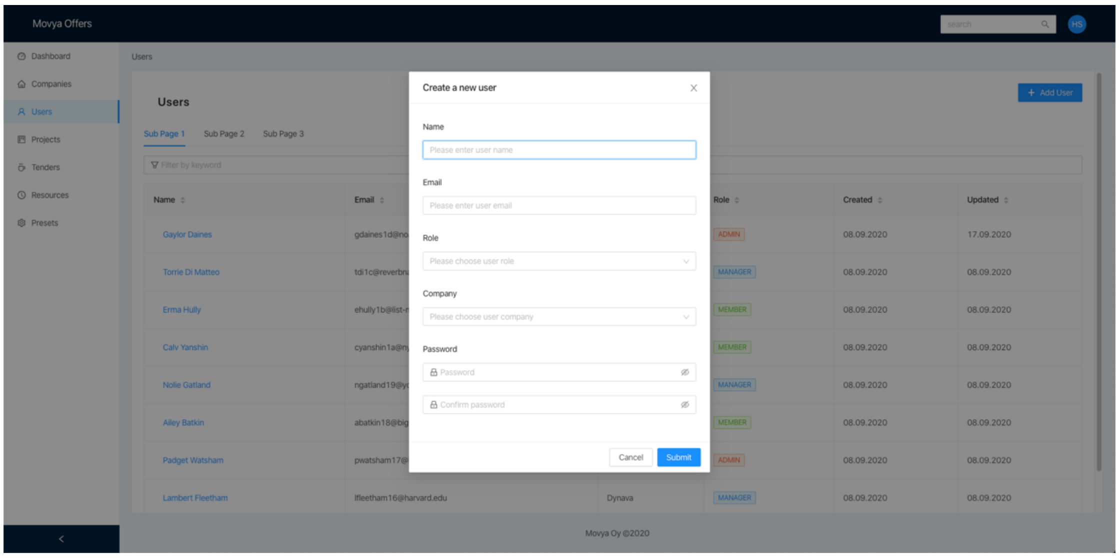
```
▾ ROOT_QUERY
  me: User
    ▸ User:5f576afa275c6b14dblc938f
  users: UserConnection
    edges: [UserEdge]
      0:
        cursor:
        "NWY1NzZhZmEyNzVjNmIxNGRiMWM5M2Jm"
        node: User
          ▸ User:5f576afa275c6b14dblc93bf
      ...
  pageInfo: PageInfo
    endCursor: "NWY1NzZhZmEyNzVjNmIxNGRiMWM5M2I2"
    hasNextPage: true
  totalCount: 50
```

Figure 92. The Apollo cache updating after loading more users

This figure shows the UI and Apollo cache states before loading additional users (in an inclined plane) and after (horizontally). It can be seen that as a result of the execution of the *loadMore()* function, the Apollo cache was updated correctly: the loaded user records were merged with the previous ones using the *cursor* IDs.

6.6.4 Creating a new item

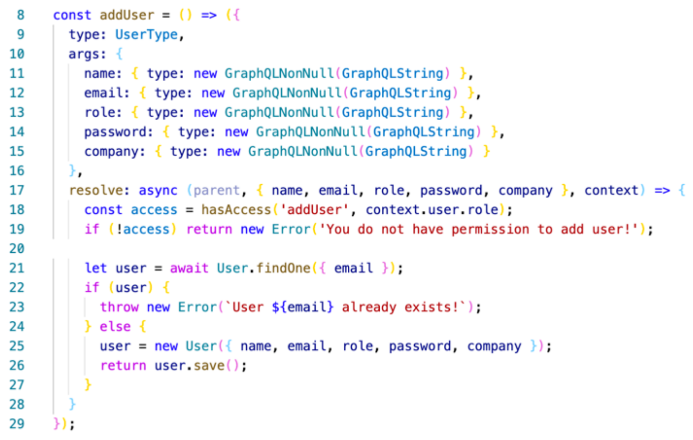
New item creation is implemented in the project for all pages and tables in the same way. The UI has an *Add* button, which opens a modal window with a form (Figure 93).



Name	Email		Role	Created	Updated
Gaylor Daines	gdaines1d@no		ADMIN	08.09.2020	17.09.2020
Torrie Di Matteo	tdi1c@reverbn		MANAGER	08.09.2020	08.09.2020
Erma Hully	ehully1b@list-r		MEMBER	08.09.2020	08.09.2020
Calv Yanshin	cyanshin1a@ny		MEMBER	08.09.2020	08.09.2020
Nolie Gatland	ngatland19@y		MANAGER	08.09.2020	08.09.2020
Ailey Batkin	abatkin18@big		MEMBER	08.09.2020	08.09.2020
Padget Watsham	pwatsham17@		ADMIN	08.09.2020	08.09.2020
Lambert Fleetham	lfleetham16@harvard.edu	Dynava	MANAGER	08.09.2020	08.09.2020

Figure 93. Adding a new user through a modal window form

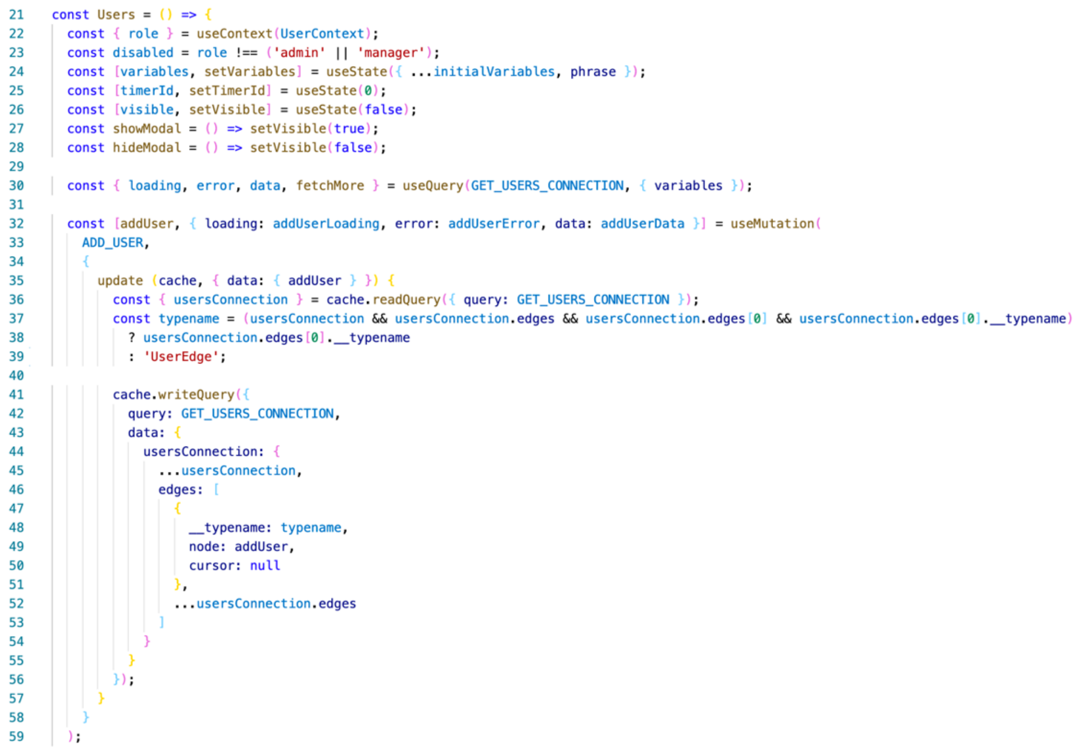
After the form fields are filled correctly, and the submit button is pressed, add mutation is performed (Figure 94).



```
const addUser = () => ({
  type: UserType,
  args: {
    name: { type: new GraphQLNonNull(GraphQLString) },
    email: { type: new GraphQLNonNull(GraphQLString) },
    role: { type: new GraphQLNonNull(GraphQLString) },
    password: { type: new GraphQLNonNull(GraphQLString) },
    company: { type: new GraphQLNonNull(GraphQLString) }
  },
  resolve: async (parent, { name, email, role, password, company }, context) => {
    const access = hasAccess('addUser', context.user.role);
    if (!access) return new Error('You do not have permission to add user!');

    let user = await User.findOne({ email });
    if (user) {
      throw new Error(`User ${email} already exists!`);
    } else {
      user = new User({ name, email, role, password, company });
      return user.save();
    }
  }
});
```

Figure 94. Mutation *addUser()*

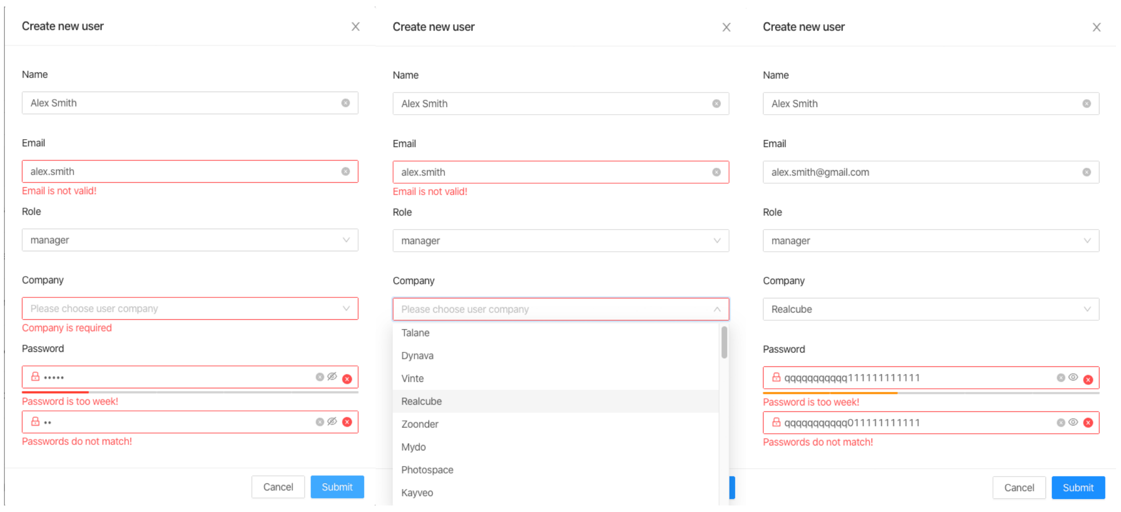
If it is successful, a new item will be rendered on the page, and the UI will be updated. For best performance, adding an item is done with updating the cache using the *update()* function (Figure 95).



```
const Users = () => {
  const { role } = useContext(UserContext);
  const disabled = role !== ('admin' || 'manager');
  const [variables, setVariables] = useState({ ...initialVariables, phrase });
  const [timerId, setTimerId] = useState(0);
  const [visible, setVisible] = useState(false);
  const showModal = () => setVisible(true);
  const hideModal = () => setVisible(false);

  const { loading, error, data, fetchMore } = useQuery(GET_USERS_CONNECTION, { variables });

  const [addUser, { loading: addUserLoading, error: addUserError, data: addUserData }] = useMutation(
    ADD_USER,
    {
      update (cache, { data: { addUser } }) {
        const { usersConnection } = cache.readQuery({ query: GET_USERS_CONNECTION });
        const typename = (usersConnection && usersConnection.edges && usersConnection.edges[0] && usersConnection.edges[0].__typename)
          ? usersConnection.edges[0].__typename
          : 'UserEdge';

        cache.writeQuery({
          query: GET_USERS_CONNECTION,
          data: {
            usersConnection: {
              ...usersConnection,
              edges: [
                {
                  __typename: typename,
                  node: addUser,
                  cursor: null
                },
                ...usersConnection.edges
              ]
            }
          }
        });
      }
    }
  );
```

Figure 95. Adding a new user with updating the cache

On the client-side, the form component implements various rules for validating inputs (Figure 96), on the server-side, for checking the existence of an item in the database (Figure 97).



Create new user

Name: Alex Smith

Email: alex.smith

Email is not valid!

Role: manager

Company: Please choose user company

Company is required

Password: •••••

Password is too week!

••

Passwords do not match!

Create new user

Name: Alex Smith

Email: alex.smith

Email is not valid!

Role: manager

Company: Please choose user company

Talane
Dynava
Vinte
Realcube
Zoonder
Mydo
Photospace
Kayveo

Create new user

Name: Alex Smith

Email: alex.smith@gmail.com

Role: manager

Company: Realcube

Password: qqqqqqqqqq11111111111

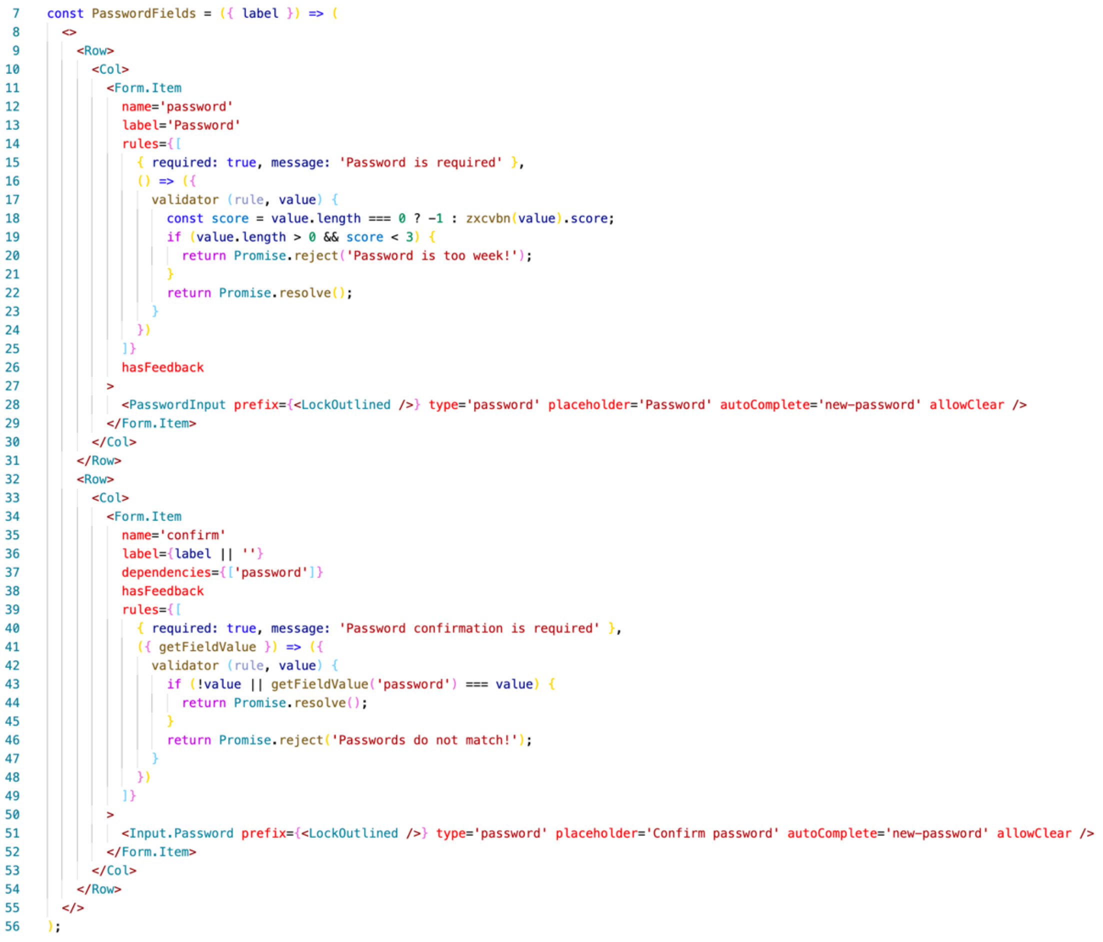
Password is too week!

qqqqqqqqqq01111111111

Passwords do not match!

Figure 96. Form for adding a new user

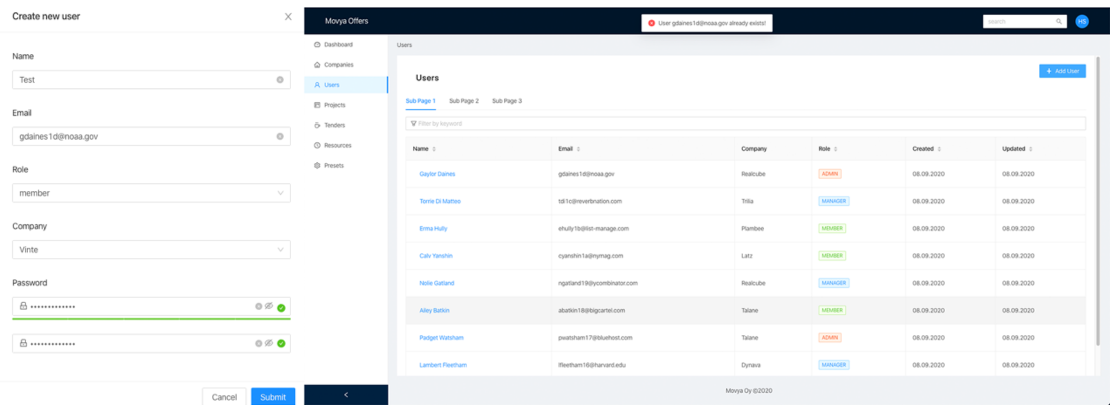
The figure shows some methods for validating form field data (Figure 96, left). Role and company inputs are implemented as a drop-down list with a dataset using the Select component of the *antd* library, which loads data only after clicking on the field (Figure 96, center) using the *useLazyQuery()* hook (Figure 101). Password validation is implemented using the *zxcvbn* library, a password strength estimator inspired by password crackers (Figure 97). It uses unique algorithms for password strength testing, which do not necessarily depend only on the password length (Figure 96, right).



```
const PasswordFields = ({ label }) => (
  <>
    <Row>
      <Col>
        <Form.Item
          name='password'
          label='Password'
          rules={[
            { required: true, message: 'Password is required' },
            () => ({
              validator (rule, value) {
                const score = value.length === 0 ? -1 : zxcvbn(value).score;
                if (value.length > 0 && score < 3) {
                  return Promise.reject('Password is too week!');
                }
                return Promise.resolve();
              }
            })
          ]}
          hasFeedback
        >
          <PasswordInput prefix={<LockOutlined />} type='password' placeholder='Password' autoComplete='new-password' allowClear />
        </Form.Item>
      </Col>
    </Row>
    <Row>
      <Col>
        <Form.Item
          name='confirm'
          label={label || ''}
          dependencies={['password']}
          hasFeedback
          rules={[
            { required: true, message: 'Password confirmation is required' },
            ({ getFieldValue }) => ({
              validator (rule, value) {
                if (!value || getFieldValue('password') === value) {
                  return Promise.resolve();
                }
                return Promise.reject('Passwords do not match!');
              }
            })
          ]}
        >
          <Input.Password prefix={<LockOutlined />} type='password' placeholder='Confirm password' autoComplete='new-password' allowClear />
        </Form.Item>
      </Col>
    </Row>
  </>
);
```

Figure 97. Password strength and matching check

The add mutation will not be sent if all form fields are not filled correctly. The mutation will not return a new item if such an item already exists in the database, and the user will receive a message about it from the server (Figure 98).



Create new user

Name: Test

Email: gdaines1d@noaa.gov

Role: member

Company: Vinte

Password: ••••••••••••• / •••••••••••••

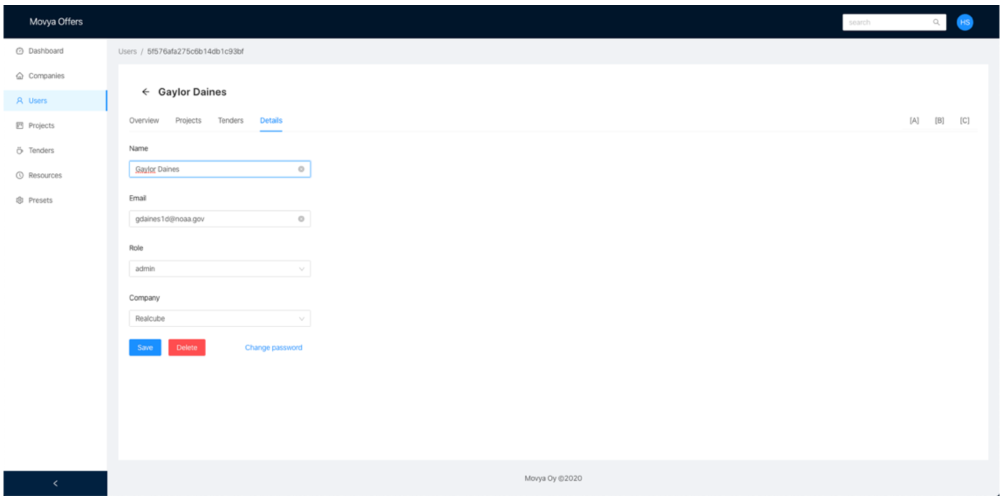
User gdaines1d@noaa.gov already exists!

Name	Email	Company	Role	Created	Updated
Gaylor Daines	gdaines1d@noaa.gov	Realcube	ADMIN	08.09.2020	08.09.2020
Torrie Di Matteo	tdi1c@reverbnation.com	Trilia	MANAGER	08.09.2020	08.09.2020
Erma Hully	ehully1b@list-manage.com	Plambee	MEMBER	08.09.2020	08.09.2020
Calv Yanshin	cyanshin1a@nymag.com	Latz	MEMBER	08.09.2020	08.09.2020
Nolie Gatland	ngatland19@ycombinator.com	Realcube	MANAGER	08.09.2020	08.09.2020
Alley Batkin	abatkin18@bigcartel.com	Talane	MEMBER	08.09.2020	08.09.2020
Padget Watsham	pwatsham17@bluehost.com	Talane	ADMIN	08.09.2020	08.09.2020
Lambert Fleetham	lfleetham16@harvard.edu	Dynava	MANAGER	08.09.2020	08.09.2020

Figure 98. If the user already exists in the database

6.6.5 Deleting an item or editing its data

Deleting or editing an item in its implementation is much like creating it. These actions are mainly carried out through the *details* tab, which contains the edit form and the button for deleting an item (Figure 99).



Users / 5f576afa275c6b14db1c93bf

Gaylor Daines

Name
Gaylor Daines

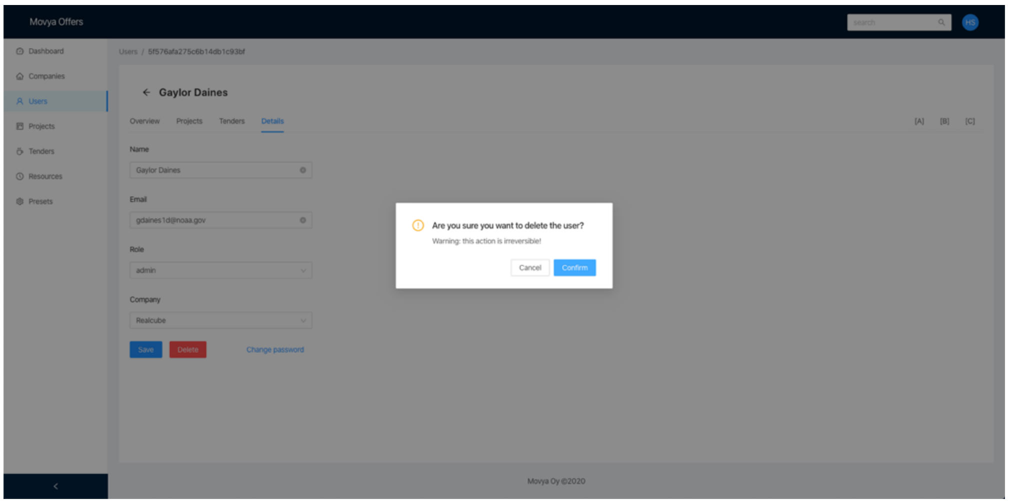
Email
gdaines1d@noaa.gov

Role
admin

Company
Realcube

Figure 99. Details tab

After the fields have been edited as required, the data can be saved to the database by clicking on the *Save* button. If it is necessary to delete an item using the *Delete* button, a modal window for confirming this irreversible operation opens (Figure 100).



Users / 5f576afa275c6b14db1c93bf

Gaylor Daines

Name
Gaylor Daines

Email
gdaines1d@noaa.gov

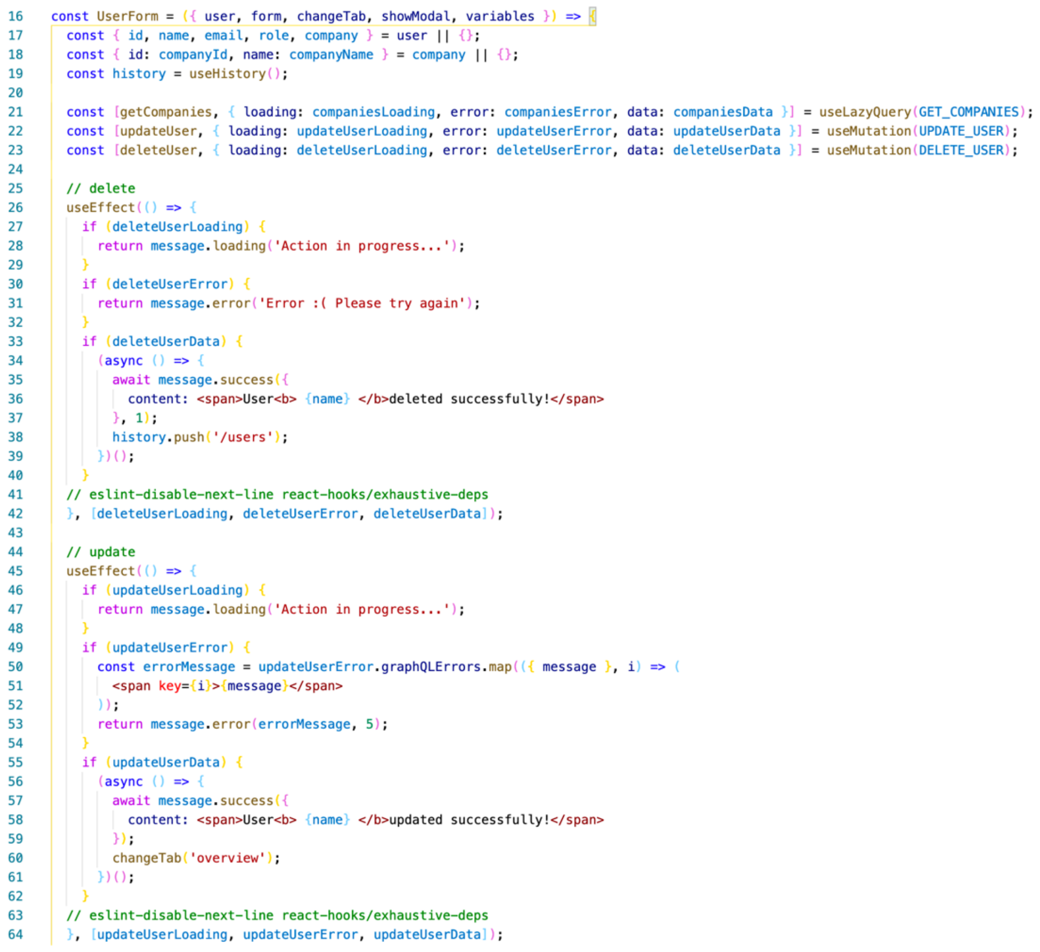
Role
admin

Company
Realcube

Are you sure you want to delete the user?
Warning: this action is irreversible!

Figure 100. The confirmation modal window

After successfully changing the item data or deleting the item (saving it to the database), the user will receive a confirm message, implemented on the client using the *useEffect()* React hook (Figure 101).

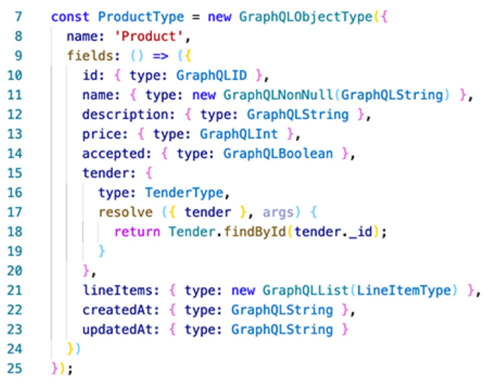


```
const UserForm = ({ user, form, changeTab, showModal, variables }) => {
  const { id, name, email, role, company } = user || {};
  const { id: companyId, name: companyName } = company || {};
  const history = useHistory();

  const [getCompanies, { loading: companiesLoading, error: companiesError, data: companiesData }] = useLazyQuery(GET_COMPANIES);
  const [updateUser, { loading: updateUserLoading, error: updateUserError, data: updateUserData }] = useMutation(UPDATE_USER);
  const [deleteUser, { loading: deleteUserLoading, error: deleteUserError, data: deleteUserData }] = useMutation(DELETE_USER);

  // delete
  useEffect(() => {
    if (deleteUserLoading) {
      return message.loading('Action in progress...');
    }
    if (deleteUserError) {
      return message.error('Error :( Please try again');
    }
    if (deleteUserData) {
      (async () => {
        await message.success({
          content: <span>User<b> {name} </b>deleted successfully!</span>
        }, 1);
        history.push('/users');
      })();
    }
  // eslint-disable-next-line react-hooks/exhaustive-deps
  }, [deleteUserLoading, deleteUserError, deleteUserData]);

  // update
  useEffect(() => {
    if (updateUserLoading) {
      return message.loading('Action in progress...');
    }
    if (updateUserError) {
      const errorMessage = updateUserError.graphQLErrors.map(({ message }, i) => (
        <span key={i}>{message}</span>
      ));
      return message.error(errorMessage, 5);
    }
    if (updateUserData) {
      (async () => {
        await message.success({
          content: <span>User<b> {name} </b>updated successfully!</span>
        });
        changeTab('overview');
      })();
    }
  // eslint-disable-next-line react-hooks/exhaustive-deps
  }, [updateUserLoading, updateUserError, updateUserData]);
```

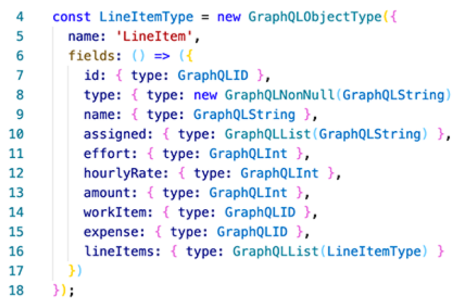
Figure 101. Updating UI using the *useEffect()* hook

6.6.6 Groupable list items

One of the critical functions implemented in the project was the grouping of product items. It had a rather complicated and non-trivial implementation, which involved creating an additional *ProductLineItem* schema in the product schema on the server-side (Figures 71, 102).



```
const ProductType = new GraphQLObjectType({
  name: 'Product',
  fields: () => ({
    id: { type: GraphQLID },
    name: { type: new GraphQLNonNull(GraphQLString) },
    description: { type: GraphQLString },
    price: { type: GraphQLInt },
    accepted: { type: GraphQLBoolean },
    tender: {
      type: TenderType,
      resolve ({ tender }, args) {
        return Tender.findById(tender._id);
      }
    },
    lineItems: { type: new GraphQLList(LineItemType) },
    createdAt: { type: GraphQLString },
    updatedAt: { type: GraphQLString }
  })
});
```

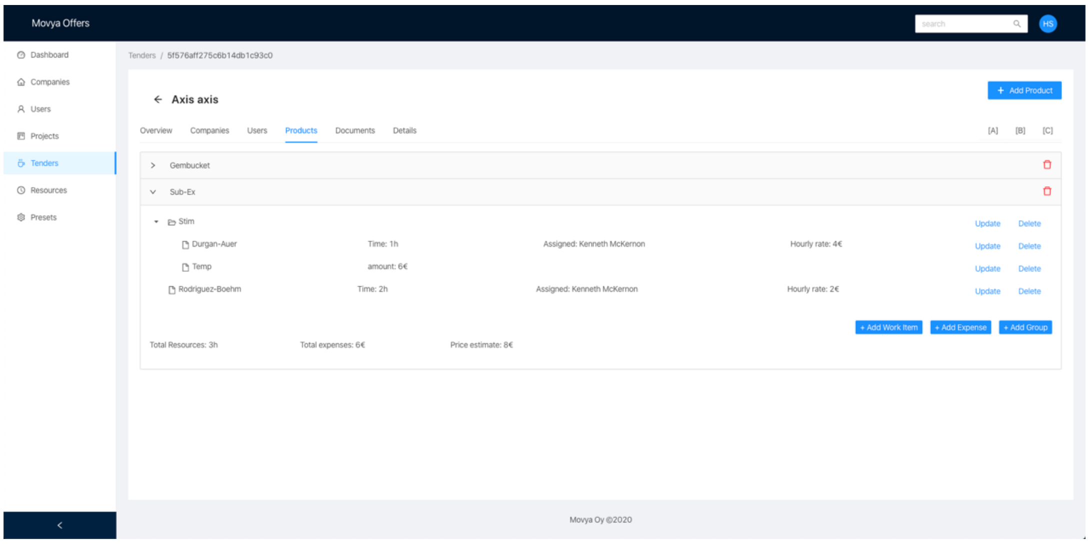


```
const LineItemType = new GraphQLObjectType({
  name: 'LineItem',
  fields: () => ({
    id: { type: GraphQLID },
    type: { type: new GraphQLNonNull(GraphQLString) },
    name: { type: GraphQLString },
    assigned: { type: GraphQLList(GraphQLString) },
    effort: { type: GraphQLInt },
    hourlyRate: { type: GraphQLInt },
    amount: { type: GraphQLInt },
    workItem: { type: GraphQLID },
    expense: { type: GraphQLID },
    lineItems: { type: GraphQLList(LineItemType) }
  })
});
```

Figure 102. *Product* and *LineItem* schemas

Each *lineItem* can include a *workItem*, expense, or *lineItem*, which can also have a *workItem*, expense, *lineItem*, etc. That is, any *lineitem* implementation is always recursive, which can be seen particularly in these figures.

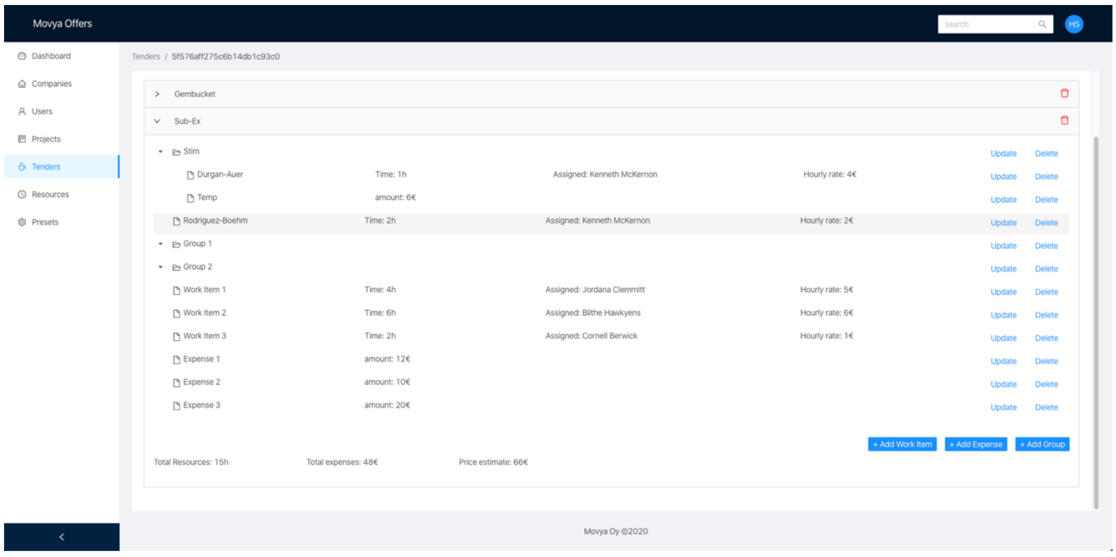
The groupable list implementation can be clearly demonstrated on the client-side. Figure 103 shows a particular *Sub-ex* product, which initially receives data from the server as a *Stim* items group and a *Rodriguez-Boehm* work item. The Stim group includes two items: a work item *Durgan-Auer* and an expense *Temp*.



```
> Gembucket
v Sub-Ex
  ▾ Stim
      Durgan-Auer        Time: 1h       Assigned: Kenneth McKernon    Hourly rate: 4€
      Temp               amount: 6€
    Rodriguez-Boehm      Time: 2h       Assigned: Kenneth McKernon    Hourly rate: 2€

Total Resources: 3h      Total expenses: 6€      Price estimate: 8€
```

Figure 103. Products tab on the tenders' page

Using the *Add* buttons on the right side of the item panel, test data was added as two groups, three work items, and three expenses (Figure 104).



Tenders / 5f576aff275c6b14db1c93c0

Gembucket

Sub-Ex

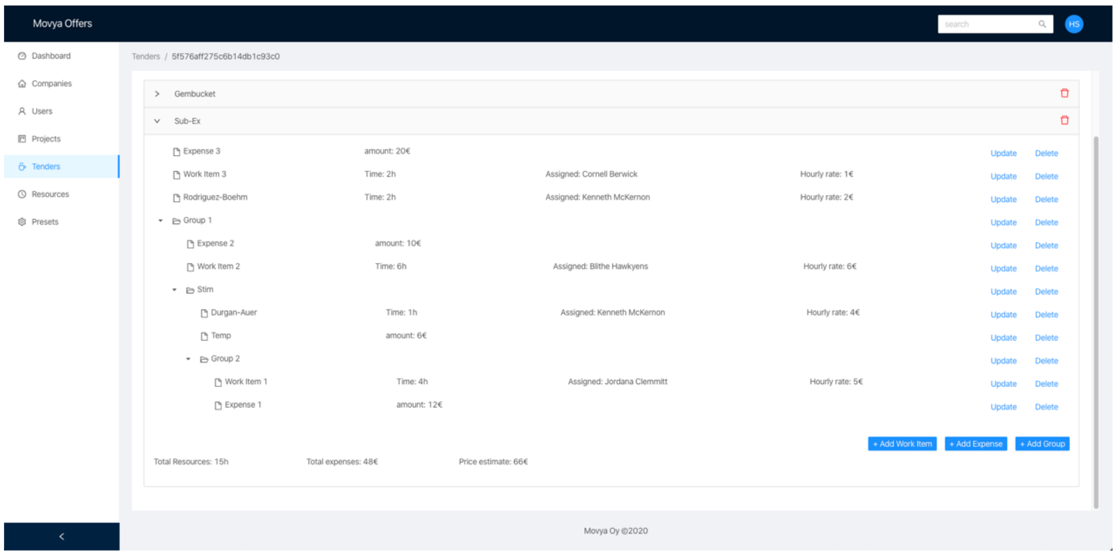
Stim			
Durgan-Auer	Time: 1h	Assigned: Kenneth McKernon	Hourly rate: 4€
Temp	amount: 6€		
Rodriguez-Boehm	Time: 2h	Assigned: Kenneth McKernon	Hourly rate: 2€
Group 1			
Group 2			
Work Item 1	Time: 4h	Assigned: Jordana Clemmitt	Hourly rate: 5€
Work Item 2	Time: 6h	Assigned: Blithe Hawkyens	Hourly rate: 6€
Work Item 3	Time: 2h	Assigned: Cornell Berwick	Hourly rate: 1€
Expense 1	amount: 12€		
Expense 2	amount: 10€		
Expense 3	amount: 20€		

Total Resources: 15h Total expenses: 48€ Price estimate: 66€

Movya Oy ©2020

Figure 104. Products tab on the tenders' page after items added

By default, all initially created groups and items are added to the root directory of the product. After the item or groups are created, it is possible to drag and drop from the product root directory to any of the groups, including any nesting level, for example, in the order shown in Figure 105: *Group 1* remained in the product root folder, the steam group was placed in *Group* 1, *Group* 2 is placed in the *Stim* group, work items and expenses of *Group* 1 and *Group* 2 are swapped.



Tenders / 5f576aff275c6b14db1c93c0

Gembucket

Sub-Ex

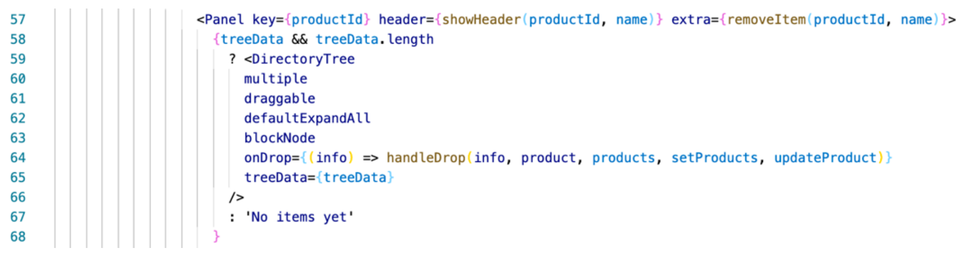
Expense 3	amount: 20€		
Work Item 3	Time: 2h	Assigned: Cornell Berwick	Hourly rate: 1€
Rodriguez-Boehm	Time: 2h	Assigned: Kenneth McKernon	Hourly rate: 2€
Group 1			
Expense 2	amount: 10€		
Work Item 2	Time: 6h	Assigned: Blithe Hawkyens	Hourly rate: 6€
Stim			
Durgan-Auer	Time: 1h	Assigned: Kenneth McKernon	Hourly rate: 4€
Temp	amount: 6€		
Group 2			
Work Item 1	Time: 4h	Assigned: Jordana Clemmitt	Hourly rate: 5€
Expense 1	amount: 12€		

Total Resources: 15h Total expenses: 48€ Price estimate: 66€

Movya Oy ©2020

Figure 105. Products tab on the tenders' page after items mixed

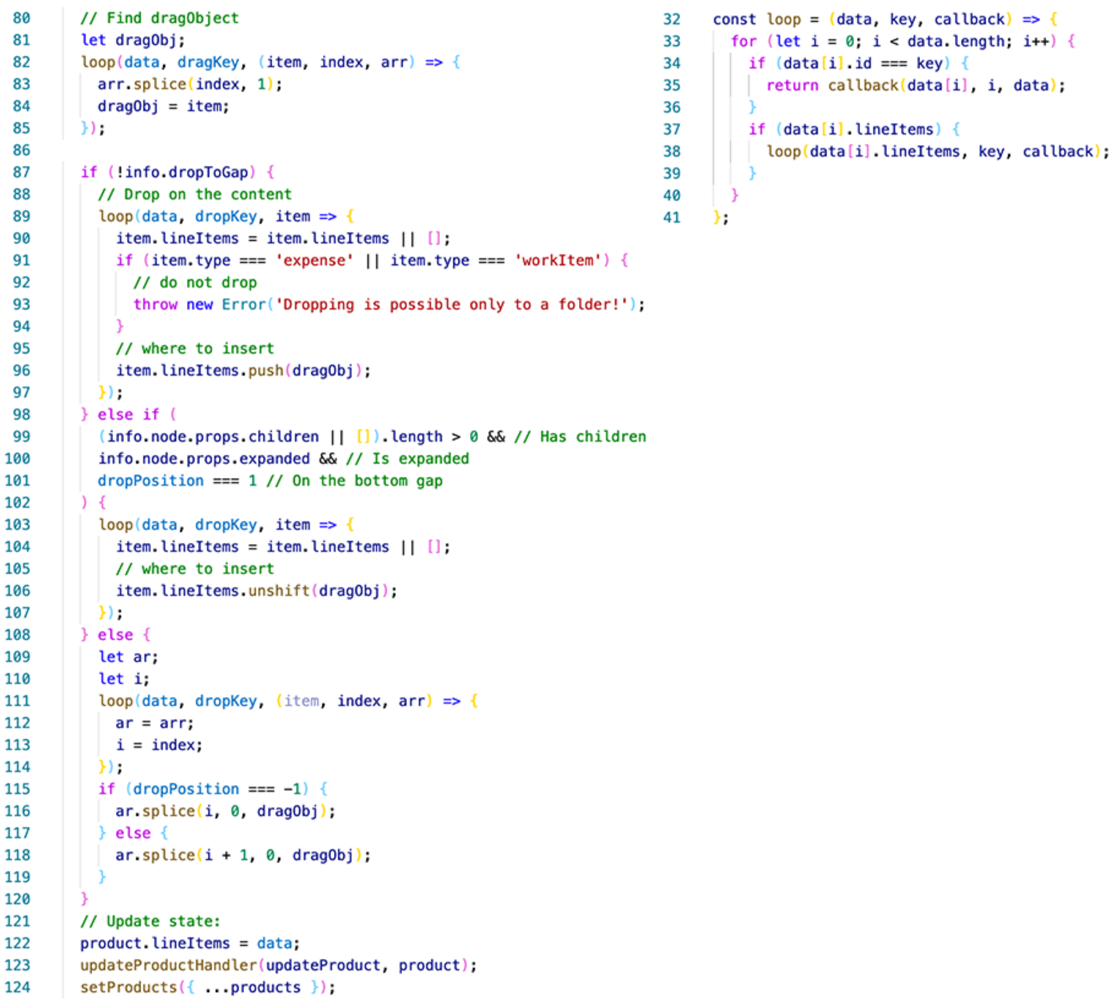
The client's implementation of this functionality was done using a hierarchical list structure component *DirectoryTree* of the *antd* library, whose API supports built-in drag-and-drop capabilities (Figure 106).



```
<Panel key={productId} header={showHeader(productId, name)} extra={removeItem(productId, name)}>
  {treeData && treeData.length
    ? <DirectoryTree
      multiple
      draggable
      defaultExpandAll
      blockNode
      onDrop={(info) => handleDrop(info, product, products, setProducts, updateProduct)}
      treeData={treeData}
    />
    : 'No items yet'
  }
```

Figure 106. The *DirectoryTree* component

This component receives several properties: *draggable*, *treeData*, and *onDrop*(). *Draggable* specifies whether this *DirectoryTree* is draggable. *treeData* contains the data needed to create the tree structure. The *onDrop()* function implements the main drag-and-drop logic: it finds the dropped object, gap, or object to insert into and updates the product state to re-render the UI. (Figure 107).



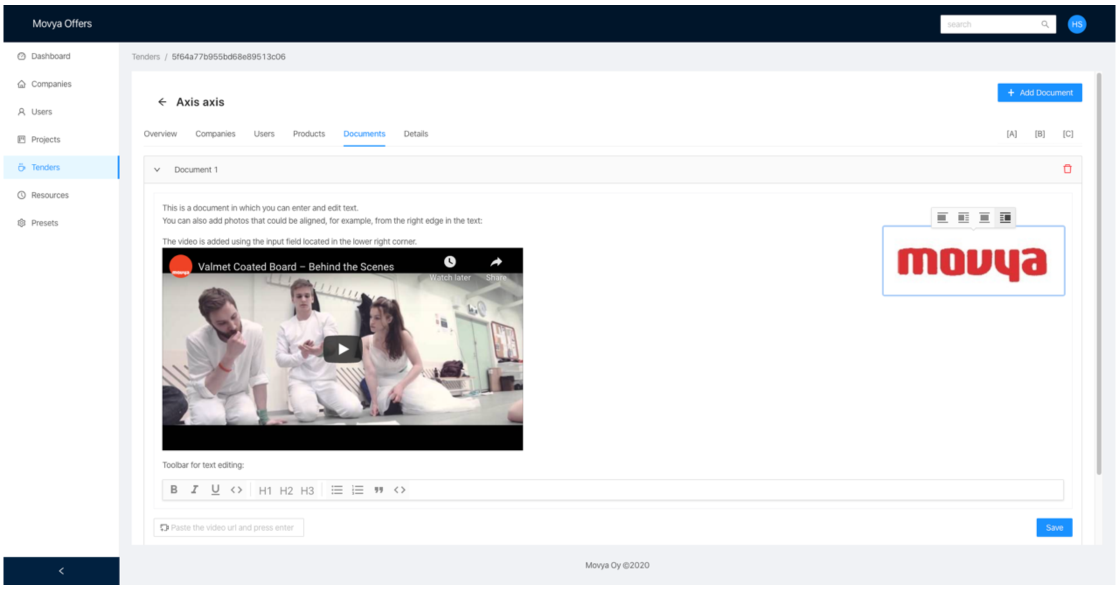
```
// Find dragObject
let dragObj;
loop(data, dragKey, (item, index, arr) => {
  arr.splice(index, 1);
  dragObj = item;
});

if (!info.dropToGap) {
  // Drop on the content
  loop(data, dropKey, item => {
    item.lineItems = item.lineItems || [];
    if (item.type === 'expense' || item.type === 'workItem') {
      // do not drop
      throw new Error('Dropping is possible only to a folder!');
    }
    // where to insert
    item.lineItems.push(dragObj);
  });
} else if (
  (info.node.props.children || []).length > 0 && // Has children
  info.node.props.expanded && // Is expanded
  dropPosition === 1 // On the bottom gap
) {
  loop(data, dropKey, item => {
    item.lineItems = item.lineItems || [];
    // where to insert
    item.lineItems.unshift(dragObj);
  });
} else {
  let ar;
  let i;
  loop(data, dropKey, (item, index, arr) => {
    ar = arr;
    i = index;
  });
  if (dropPosition === -1) {
    ar.splice(i, 0, dragObj);
  } else {
    ar.splice(i + 1, 0, dragObj);
  }
}
// Update state:
product.lineItems = data;
updateProductHandler(updateProduct, product);
setProducts({ ...products });
```

```
const loop = (data, key, callback) => {
  for (let i = 0; i < data.length; i++) {
    if (data[i].id === key) {
      return callback(data[i], i, data);
    }
    if (data[i].lineItems) {
      loop(data[i].lineItems, key, callback);
    }
  }
};
```

Figure 107. Main *onDrop()* and secondary *loop()* functions

6.6.7 Editable documents

Editable documents are the last feature of the project that is worth mentioning. The document is a rich text editor with advanced functionality, particularly adding photos and videos (Figure 108).



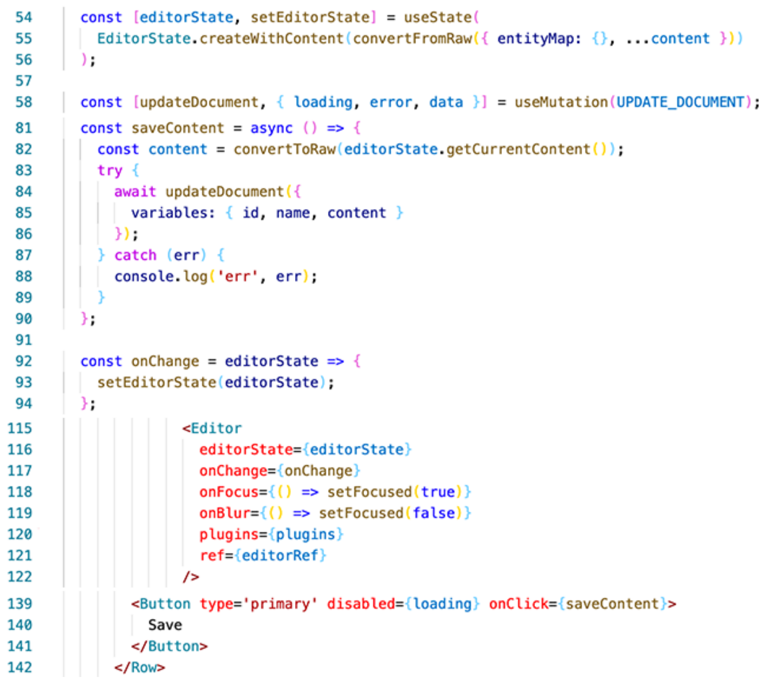
This is a document in which you can enter and edit text.
You can also add photos that could be aligned, for example, from the right edge in the text:

The video is added using the input field located in the lower right corner.

Toolbar for text editing:

Figure 108. Editable document

The text in the picture describes the main features implemented in the editor. They were built using the low-level *draft.js* framework, a tool for solving a very narrow range of text input control and text editing tasks, and the *draft-js-plugins-editor* library. The editor is connected to the page using the *Editor* React component (Figure 109).

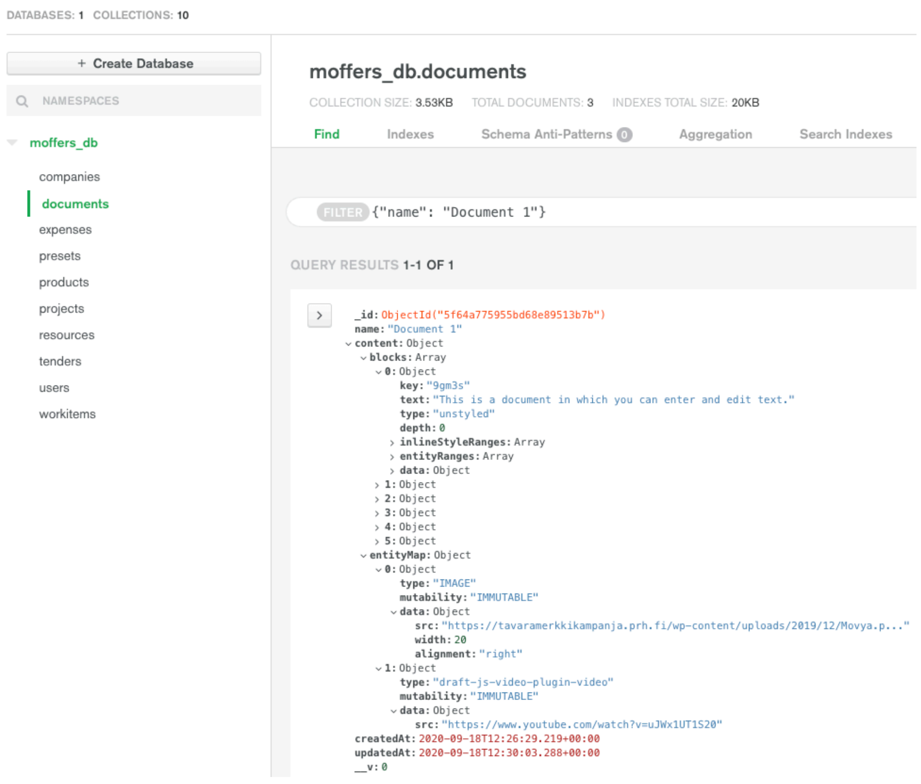


```
const [editorState, setEditorState] = useState(
  EditorState.createWithContent(convertFromRaw({ entityMap: {}, ...content }))
);

const [updateDocument, { loading, error, data }] = useMutation(UPDATE_DOCUMENT);
const saveContent = async () => {
  const content = convertToRaw(editorState.getCurrentContent());
  try {
    await updateDocument({
      variables: { id, name, content }
    });
  } catch (err) {
    console.log('err', err);
  }
};

const onChange = editorState => {
  setEditorState(editorState);
};
            <Editor
              editorState={editorState}
              onChange={onChange}
              onFocus={() => setFocused(true)}
              onBlur={() => setFocused(false)}
              plugins={plugins}
              ref={editorRef}
            />
        <Button type='primary' disabled={loading} onClick={saveContent}>
          Save
        </Button>
      </Row>
```

Figure 109. The *Editor* component

The main properties declared in the editor are *editorState* and *onChange()*. The *editorState* property defines the editor's current state. As seen from the code, *editorState* is stored in the parent component's state and updated using the *onChange()* method through the *setEditorState()* update function of the *useState()* React hook. The *onChange()* function is a function that is called upon any manipulation in the editor. Since the editor appears on the page with predefined content already, the *createWithContent()* method is used.

The *Save* button saves the document content, converted by the *convertToRaw()* function, into a formatted JavaScript object in the database. Figure 110 shows the complex data structure of an object stored in the database.

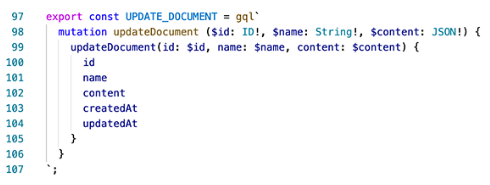


```
{"name": "Document 1"}
```

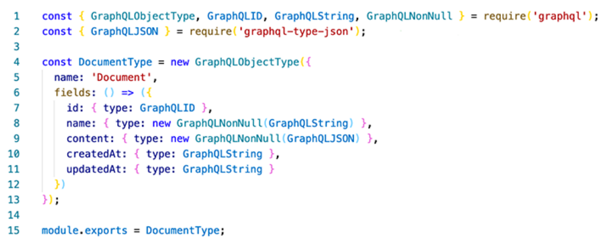
```
_id: ObjectId("5f64a775955bd68e89513b7b")
name: "Document 1"
⌄ content: Object
  ⌄ blocks: Array
    ⌄ 0: Object
        key: "9gm3s"
        text: "This is a document in which you can enter and edit text."
        type: "unstyled"
        depth: 0
      › inlineStyleRanges: Array
      › entityRanges: Array
      › data: Object
    › 1: Object
    › 2: Object
    › 3: Object
    › 4: Object
    › 5: Object
  ⌄ entityMap: Object
    ⌄ 0: Object
        type: "IMAGE"
        mutability: "IMMUTABLE"
      ⌄ data: Object
          src: "https://tavaramerkkikampanja.prh.fi/wp-content/uploads/2019/12/Movya.p..."
          width: 20
          alignment: "right"
    ⌄ 1: Object
        type: "draft-js-video-plugin-video"
        mutability: "IMMUTABLE"
      ⌄ data: Object
          src: "https://www.youtube.com/watch?v=uJWx1UT1S20"
createdAt: 2020-09-18T12:26:29.219+00:00
updatedAt: 2020-09-18T12:30:03.288+00:00
__v: 0
```

Figure 110. A document saved in the database

Content is an object with two top-level properties: *blocks* and *entityMap*. For this data to end up in the database as an embedded document it is necessary to define *content* data type in GraphQL. Thus, the NPM package *graphql-type-json* was used, which provides the JSON value scalar GraphQL type. By defining the content object type as JSON, the data is sent to the server using the *updateDocument* mutation (Figure 110).



```
export const UPDATE_DOCUMENT = gql`
  mutation updateDocument ($id: ID!, $name: String!, $content: JSON!) {
    updateDocument(id: $id, name: $name, content: $content) {
      id
      name
      content
      createdAt
      updatedAt
    }
  }
`;
```



```
const { GraphQLObjectType, GraphQLID, GraphQLString, GraphQLNonNull } = require('graphql');
const { GraphQLJSON } = require('graphql-type-json');

const DocumentType = new GraphQLObjectType({
  name: 'Document',
  fields: () => ({
    id: { type: GraphQLID },
    name: { type: new GraphQLNonNull(GraphQLString) },
    content: { type: new GraphQLNonNull(GraphQLJSON) },
    createdAt: { type: GraphQLString },
    updatedAt: { type: GraphQLString }
  })
});

module.exports = DocumentType;
```

Figure 111. The *updateDocument* mutation (left) and defining the *Document* type (right)

7 Conclusion

7.1 Discussion

During the last ten years, the JavaScript development industry made considerable progress, and it keeps emerging fast. This resulted in numerous development tools (scripts for project bootstrapping like *create-react-app* and automated code formatting like *eslint* used for this thesis project) and packages (NPM). Tools allow performing many development and code maintenance tasks, such as scanning code for quality issues and fixing incorrect formatting, start the project quickly, and skip creating boilerplate code. The variety of packages covers almost every possible problem or need that one can imagine.

However, these benefits need to be used with caution. For example, the automated project setup can sometimes be too complicated for a new developer or contain features that are not required for a project. E.g., for the frontend part, the utility *create-react-app* was used, which performed some comprehensive setup of configuration. And then, a similar step had to take place for adding Ant Design to the project. For this step, another utility updated some project configuration according to the Ant Design library's needs.

JavaScript packages usually are built with many dependencies, which often leads to a large dependency tree. This makes it pretty difficult to track the quality of all the packages that a project uses. As a rule of thumb, only the packages with high popularity and good maintenance support were used for this project. This approach allows building a reliable and robust software with the opportunity to update dependency packages later when any patches or bug fixes are released.

7.1.1 Solving REST problems

Traditional REST applications are robust and proved themselves with time as reliable solutions. However, some significant disadvantages can be addressed to improve both development and performance: over-fetching (as well as under-fetching) of data and maintenance-heavy endpoints.

Over-fetching and under-fetching are described in detail in the GraphQL section of the thesis. The GraphQL approach solves these problems very nicely. Having implemented the data layer with GraphQL allowed building the application that has an excellent performance. This allows using it on mobile devices as well as with low network latency. There were many requirements for fetching data in the project, and implementation-wise it all got down to writing GraphQL queries without any changes on the server-side. The query language is very flexible and allows fetching data that has any level of complexity. For example, there was a case with a recursively nested data structure in the project, and GraphQL allowed to make the implementation pretty quickly.

The second problem of building endpoints on the server-side for the REST API usually consists of a developer's responsibility to write many endpoints. If the data model of a project is complicated, it requires many RESTful endpoints to support it. Thus, the server side's development is both times consuming and might lead to a large codebase. Also, it is essential to consider the maintenance aspect of a project. E.g., when a new requirement for the data model comes in, it might require changes in multiple endpoints. This will then require updating tests, documentation, etc. If a new developer joins a project, it can take a lot of time for onboarding. This problem gets very nicely solved by GraphQL - there is only a single endpoint on the server-side that has to be deployed. Thus it can be said that this way, the server-side API is maintenance-free.

Apollo platform is a prevalent and mature open-source implementation of GraphQL client and server. It was very easy to adapt and even to customize to satisfy the project's requirements. Even the complex implementation issues of caching have been solved thanks to the library's high popularity and support of the wide community of Apollo developers.

7.1.2 Solving UX requirements

Any modern front-end application requires an MVC framework. React (as a "View" part of MVC) is the most popular solution. All sort of organizations and companies has adopted it. With this, the JavaScript community got the widest choice of UI components available out there for web-applications. React is both a performant and reliable library. It supports all browsers eliminating cross-browser issues for development completely. It is backed by Facebook and is being developed continuously. Also, the entry point and the learning curve for new developers is low. It promotes many development patterns and goes along with the development of JavaScript (ECMA) programming language.

The project has many UI components that are very interactive and require rapid updates of data. The VirtualDOM feature of React solved all the potential performance issues in the browser, even with rendering a lot of data (e.g., infinite paginated scrolling). There were no performance issues noticed during the testing of the application with a production set of data.

The design system Ant Design was a great compliment to React. It eliminates the need for a visual designer for the team. It provides many ready-to-use UI components (especially tables and forms since they are heavily used in the project). Both of these components offer very flexible and rich functionality. In the end, all the UX requirements were solved.

7.2 Results

As a result of the work, the primary goals were achieved, the web service was implemented. It satisfies main requirements, such as the modern stack of technologies that have been used for both the project's backend and frontend parts; most of the required features were implemented.

Also, one of the critical aspects was the analysis of using design systems to build the UI. After exploring different options, the Ant Design library was chosen, and a small proof-of-concept application was built to prove that it can satisfy the requirements. This design system saved an enormous amount of time and effort. It provided a large selection of ready-made components, endowed with excellent functionality for implementing the most complicated web service features.

Having had some working experience with the React library previously, I can note some positive aspects of its use combined with the GraphQL and Apollo, which worked well together in the project. In particular, these technologies allowed to reduce network traffic and increase the applications' speed by specifying the only data required in each request (preventing over-fetching), obtaining many resources in a single request (preventing under-fetching), and implementing data caching. The GraphQL type system provides a living form of self-documenting, which significantly improves developers' interaction when combined with GraphiQL or GraphQL Playground tools.

References

Advanced topics on caching. N.d. Documentation on the Apollographql website. Accessed 4.9.2020. Retrieved from https://www.apollographql.com/docs/react/caching/advanced-topics/#updating-after-a-mutation.

Advanced Tutorial – Clients. N.d. Article on the Howtographql website. Accessed 26.8.2020. Retrieved from https://www.howtographql.com/advanced/0-clients/.

Aggarwal, S. 2018. *Modern web-development using reactjs*. International Journal of Recent Research Aspects, *5*(1), 2349-7688.

Ant Design. N.d. Documentation on the AntDesign website. Accessed 10.9.2020. Retrieved from https://ant.design/.

Apollo Server. 2020. Documentation on the Github website. Accessed on 1.9.2020. Retrieved from https://github.com/apollographql/apollo-server.

Authentication. N.d. Documentation on the Apollographql website. Accessed 9.9.2020. Retrieved from https://www.apollographql.com/docs/apollo-server/security/authentication/.

Begunov, A. 2018. *ESLint. Znakomstvo*. [ESLint. Acquaintance.] Original publication: ESLint. Знакомство. Article on the Medium website. Accessed on 10.9.2020. Retrieved from https://medium.com/@catwithapple/eslint-знакомство-69ffc19edbf8.

Bènks, A., & Porsello, E. 2019. *GraphQL: yazyk zaprosov dlya sovremennyh veb-priloženij*. [GraphQL: query language for modern web applications.] Original publication: GraphQL: язык запросов для современных веб-приложений. Progress Kniga Ltd.

Big Picture (Architecture). N.d. Article on the Howtographql website. Accessed 26.8.2020. Retrieved from https://www.howtographql.com/basics/3-big-picture/.

Burk, N. N.d. *Getting Started*. Article on the Howtographql website. Accessed 3.9.2020. Retrieved from https://www.howtographql.com/react-apollo/1-getting-started/.

Concepts Overview. N.d. Documentation on the Apollographql website. Accessed 2.9.2020. Retrieved from https://www.apollographql.com/docs/link/overview/.

Corey, G. 2018. *Understanding Apollo Fetch Policies*. Article on the Medium website. Accessed 1.9.2020. Retrieved from https://medium.com/@galen.corey/understanding-apollo-fetch-policies-705b5ad71980.

Čto takoe Reakt. Pervoe priloženie. 2017. [What is React. First application.] Original publication: Что такое Реакт. Первое приложение. Article on the Metanit website. Accessed on 18.8.2020. Retrieved from https://metanit.com/web/react/1.1.php.

Documentation Home. N.d. Documentation on the Apollographql website. Accessed 26.8.2020. Retrieved from https://www.apollographql.com/docs/.

Get started. N.d. Documentation on the Apollographql website. Accessed 3.9.2020. Retrieved from https://www.apollographql.com/docs/react/get-started/.

Getting Started. 2020. Documentation on the Visual Studio Code website. Accessed 8.9.2020. Retrieved from https://code.visualstudio.com/docs.

Glover, R. 2019. *GraphQL in Depth: What, Why, and How*. Article on the Ponyfoo website. Accessed 25.8.2020. Retrieved from https://ponyfoo.com/articles/graphql-in-depth-what-why-and-how.

GraphQL – kratkoe rukovodstvo. 2019. [GraphQL – Quick guide.] Original publication: GraphQL – Краткое руководство. Manual on the Coderlessons website. Accessed on 20.8.2020. Retrieved from https://coderlessons.com/tutorials/veb-razrabotka/izuchite-graphql/graphql-kratkoe-rukovodstvo.

GraphQL Cursor Connections Specification. N.d. Documentation on the Relay website. Accessed 15.9.2020. Retrieved from https://relay.dev/graphql/connections.htm.

Hamedani, M. 2018a. *React Virtual DOM Explained in Simple English*. Article on the Programmingwithmosh website. Accessed 18.8.2020. Retrieved from https://programmingwithmosh.com/react/react-virtual-dom-explained/.

Hamedani, M. 2018b. *React Lifecycle Methods – A Deep Dive*. Article on the Programmingwithmosh website. Accessed 18.8.2020. Retrieved from https://programmingwithmosh.com/javascript/react-lifecycle-methods/.

Hauser, E. 2017. *Apollo Link: Creating your custom GraphQL client*. Article on the Apollographql website. Accessed on 2.9.2020. Retrieved from https://www.apollographql.com/blog/apollo-link-creating-your-custom-graphql-client-c865be0ce059/.

Helfer, J. 2016. *GraphQL explained*. Article on the Apollographql website. Accessed on 25.8.2020. Retrieved from https://www.apollographql.com/blog/graphql-explained-5844742f195e/.

Hooks. N.d. Documentation on the Apollographql website. Accessed 27.8.2020. Retrieved from https://www.apollographql.com/docs/react/api/react/hooks/.

Huder, K. N. 2019. *Modifikovanij sposib kešuvannya danih klients'koj biblioteki Apollo-Client dlya GraphQL*. [Modified method of caching Apollo-Client client library for GraphQL.] Original publication: Модифікований спосіб кешування даних клієнтської бібліотеки Apollo-Client для GraphQL. Master's thesis, Igor Sikorskij KPI.

Introduction to Apollo Client. N.d. Documentation on the Apollographql website. Accessed 27.8.2020. Retrieved from https://www.apollographql.com/docs/react/.

Introduction to Apollo Server. N.d. Documentation on the Apollographql website. Accessed 26.8.2020. Retrieved from https://www.apollographql.com/docs/apollo-server/.

Introduction to JSON Web Tokens. N.d. Documentation on the JWT website. Accessed 9.9.2020. Retrieved from https://jwt.io/introduction/.

Komponenty. 2017. [Components.] Original publication: Компоненты. Article on the Metanit website. Accessed on 18.8.2020. Retrieved from https://metanit.com/web/react/2.2.php.

Komponenty i propsy. N.d. [Components and props.] Original publication: Компоненты и пропсы. Documentation on the React website. Accessed 18.8.2020. Retrieved from https://ru.reactjs.org/docs/components-and-props.html.

Kontekst. N.d. [Context.] Original publication: Контекст. Documentation on the React website. Accessed 19.8.2020. Retrieved from https://ru.reactjs.org/docs/context.html.

Kratkij obzor hukov. N.d. [Hooks at a glance.] Original publication: Краткий обзор хуков. Documentation on the React website. Accessed 18.8.2020. Retrieved from https://ru.reactjs.org/docs/hooks-overview.html.

Krofegha, B. 2020. *Understanding Client-Side GraphQl With Apollo-Client In React Apps*. Article on the Smashingmagazine website. Accessed 3.9.2020. Retrieved from https://www.smashingmagazine.com/2020/07/client-side-graphql-apollo-client-react-apps/.

Losoviz, L. 2020. *Versioning fields in GraphQL*. Article on the LogRocket website. Accessed on 20.8.2020. Retrieved from https://blog.logrocket.com/versioning-fields-graphql/.

Mbanugo, P. 2019. *GraphQL: Schema, Resolvers, Type System, Schema Language, and Query Language*. Article on the Telerik website. Accessed on 25.8.2020. Retrieved from https://www.telerik.com/blogs/graphql-schema-resolvers-type-system-schema-language-query-language.

McIlraith, S. A., Son, T. C., & Zeng, H. 2001. *Semantic Web Services*. IEEE intelligent systems, 16(2), 46-53.

More Mutations and Updating the Store. N.d. Article on the Howtographql website. Accessed 3.9.2020. Retrieved from https://www.howtographql.com/react-apollo/6-more-mutations-and-updating-the-store/.

Mutations. N.d. Documentation on the Apollographql website. Accessed 3.9.2020. Retrieved from https://www.apollographql.com/docs/react/data/mutations/.

Newby, C. 2019. *A comparison of GraphQL and REST*. Article on the ITNEXT website. Accessed 20.8.2020. Retrieved from https://itnext.io/a-comparison-of-graphql-and-rest-e125d77fb329.

Osnovy GraphQL. 2019. [GraphQL basics.] Original publication: Основы GraphQL. Article on the Coldfox website. Accessed on 24.8.2020. Retrieved from http://www.coldfox.ru/article/5c5369b5779576192190cf1c/Основы-GraphQL.

Props. 2017. Article on the Metanit website. Accessed on 18.8.2020. Retrieved from https://metanit.com/web/react/2.3.php.

Rautvuori M. 2019. *Joulukuun tuore tavaramerkki: Tarinasi takana - Movya*. Article on the PRH website. Accessed on 12.8.2020. Retrieved from https://www.prh.fi/fi/uutislistaus/2019/P_19590.html.

Ravichandran, A. 2019. *GraphQL queries explained in simple terms*. Article on the LogRocket website. Accessed on 24.8.2020. Retrieved from https://blog.logrocket.com/graphql-queries-in-simple-terms/.

React.js history. 2019. Article on the Education-ecosystem website. Accessed on 17.8.2020. Retrieved from https://www.education-ecosystem.com/guides/programming/react-js/history.

Rendering èlementov. N.d. [Rendering elements.] Original publication: Рендеринг элементов. Documentation on the React website. Accessed 18.8.2020. Retrieved from https://ru.reactjs.org/docs/rendering-elements.html.

Resolvers. N.d. Documentation on the Apollographql website. Accessed 1.9.2020. Retrieved from https://www.apollographql.com/docs/apollo-server/data/resolvers/.

Richardson, L., & Ruby, S. 2008. *RESTful web services*. O'Reilly Media, Inc.

Samer, A. 2017. *Difference Between API and Web Service*. Article on the Medium website. Accessed on 14.8.2020. Retrieved from https://medium.com/@programmerasi/difference-between-api-and-web-service-73c873573c9d.

Sostoyanie i žiznennyj cikl. N.d. [State and Lifecycle.] Original publication: Состояние и жизненный цикл. Documentation on the React website. Accessed 18.8.2020. Retrieved from https://ru.reactjs.org/docs/state-and-lifecycle.html.

Sozdaem novoe React-priloženie. N.d. [Create a New React App.] Original publication: Создаём новое React-приложение. Documentation on the React website. Accessed 9.9.2020. Retrieved from https://ru.reactjs.org/docs/create-a-new-react-app.html.

Spukas, L. 2019. *Avoid Prop drilling in React with Context API*. Article on the Dev website. Accessed 19.8.2020. Retrieved from https://dev.to/spukas/avoid-prop-drilling-in-react-with-context-api-1ne5.

State. 2017. Article on the Metanit website. Accessed on 18.8.2020. Retrieved from https://metanit.com/web/react/2.4.php.

Stuart, M. 2018. *GraphQL Resolvers: Best Practices*. Article on the Medium website. Accessed on 25.8.2020. Retrieved from https://medium.com/paypal-engineering/graphql-resolvers-best-practices-cd36fdbcef55.

The Apollo platform. N.d. Documentation on the Apollographql website. Accessed 26.8.2020. Retrieved from https://www.apollographql.com/docs/intro/platform/.

The GraphQL Foundation. 2020a. *A query language for your API*. Documentation on the GraphQL website. Accessed 19.8.2020. Retrieved from https://graphql.org/.

The GraphQL Foundation. 2020b. *Queries and Mutations*. Documentation on the GraphQL website. Accessed 21.8.2020. Retrieved from https://graphql.org/learn/queries/.

The GraphQL Foundation. 2020c. *Running an Express GraphQL Server*. Documentation on the GraphQL website. Accessed 8.9.2020. Retrieved from https://graphql.org/graphql-js/running-an-express-graphql-server/.

The GraphQL Foundation. 2020d. *Graphql/type*. Documentation on the GraphQL website. Accessed 9.9.2020. Retrieved from https://graphql.org/graphql-js/type/.

The GraphQL Foundation. 2020e. *Pagination*. Documentation on the GraphQL website. Accessed 15.9.2020. Retrieved from https://graphql.org/learn/pagination/.

Use in create-react-app. N.d. Documentation on the AntDesign website. Accessed 11.9.2020. Retrieved from https://3x.ant.design/docs/react/use-with-create-react-app.

Vardhan, A. 2020. *How GraphQL Subscriptions Work*. Article on the Dgraph website. Accessed on 20.8.2020. Retrieved from https://dgraph.io/blog/post/how-does-graphql-subscription/.

View Integrations. N.d. Documentation on the Apollographql website. Accessed 27.8.2020. Retrieved from https://www.apollographql.com/docs/react/integrations/integrations/.

Vipul, A. M., & Sonpatki, P. 2016. *ReactJS by Example-Building Modern Web Applications with React*. Packt Publishing Ltd.

Web services in theory and practice for beginners. 2008. Article on the Sudonull website. Accessed 14.8.2020. Retrieved from https://sudonull.com/post/202174-Web-services-in-theory-and-practice-for-beginners.

What we do. N.d. Company presentation on the Movya website. Accessed on 12.4.2020. Retrieved from http://www.movya.fi/en/services.

Wieruch, R. 2018. *The Road to GraphQL: Your journey to master pragmatic GraphQL in JavaScript with React. js and Node. js*. Robin Wieruch.

Znakomstvo s JSX. N.d. [Introducing JSX.] Original publication: Знакомство с JSX. Documentation on the React website. Accessed 18.8.2020. Retrieved from https://ru.reactjs.org/docs/introducing-jsx.html.