Tri Tran

Build a GraphQL application with Node.js and React

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

10 April 2019

Metropolia

Author	Tri Tran
Title	Build a GraphQL application with Node.js and React
Number of Pages	38 pages
Date	10 April 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software Engineering
Instructors	Janne Salonen, Head of ICT Department

REST has been popular for building an API for a long time. In spite of its popularity and usage, it still represents several challenges when building a REST server. A new technology called GraphQL was introduced to tackle these issues. GraphQL has gained a significant amount of support from the developer community and it has been adopted by many enterprises.

The main objective of the paper is to explain how GraphQL works and discuss the benefits and drawbacks of using GraphQL. Documentations of the library, a book, and a few reliable online articles were used to describe the implementation of a GraphQL server and a GraphQL client. This analysis will illustrate if using GraphQL is a great solution and show how straightforward it is to implement GraphQL on the back-end and on the front-end. While the API will be built with Node.js, the client will be built with React.

Keywords	GraphQL, Node.js, backend, server, JavaScript, API, REST

Contents

List of Abbreviations

REST	Representational State Transfer
API	Application Programming Interface
URL	Uniform Resource Locator
SOAP	Simple Object Access Protocol
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
IDE	Integrated Development Environment
SDL	Schema Definition Language
CRUD	Create, Read, Update, and Delete

1 Introduction

Since Representational State Transfer (REST) was introduced in 2000 by Roy Fielding, it has become the de facto standard for network requests between client and server applications [1]. Most applications written today was implemented in REST. REST is easier to understand than SOAP (Simple Object Access Protocol) and it is also less verbose than SOAP. In a REST API, every resource is accessed by a URL (Uniform Resource Locator). Moreover, REST helps client and server to communicate with each other by using HTTP (Hypertext Transfer Protocol) with URLs and HTTP methods [1].

Since REST uses the HTTP protocol, the request and response mechanism is simple in REST. The client is not tied to the server anymore. Therefore, it is easier to build a large and scalable infrastructure of the organization without worrying about the technology stack used on the client-side. Additionally, the server supports caching and it is stateless thanks to REST architecture.

Even though REST was popular for a long time, managing data in modern applications is still challenging. Data has to be aggregated from multiple sourced and distributed to multiple clients. In addition, front-end developers need to manage data on the client besides executing features related to the user interface. A technology called GraphQL was introduced by Facebook in 2012 to ease these paint points. Since its release, GraphQL has gained significant momentum and has been adopted by an increasing number of large companies such as GitHub, Twitter, and PayPal. Thanks to its declarative data fetching, data transfer between applications becomes more efficient.

2 Summary of GraphQL

GraphQL is a query language created by Facebook in 2012 to support the mobile application infrastructure of the company. Facebook started to rethink mobile application data-fetching from the perspective of product designers and developers, which led to the creation of GraphQL. GraphQL moved the focus of development to the client applications from the server, where developers and designers spend their time and attention. It was then open-sourced in 2015. [1] While REST is an architecture, GraphQL is not since GraphQL is just a data fetching specification. Moreover, it is also an API query language.

[2] GraphQL is also documentation for its own API because it serves a thorough and comprehensible description of the data in the API. It helps clients to write an API that can evolve over time and an API which is supported by powerful developer tools. GraphQL has three principle fundamental characteristics. The first characteristic is that the client can request exactly the data it needs. Second, aggregating data from multiple sources in GraphQL is easier than aggregating in a REST API. Last but not least, GraphQL uses a type system to describe data, which makes GraphQL strongly typed.

3 How GraphQL solves REST issues

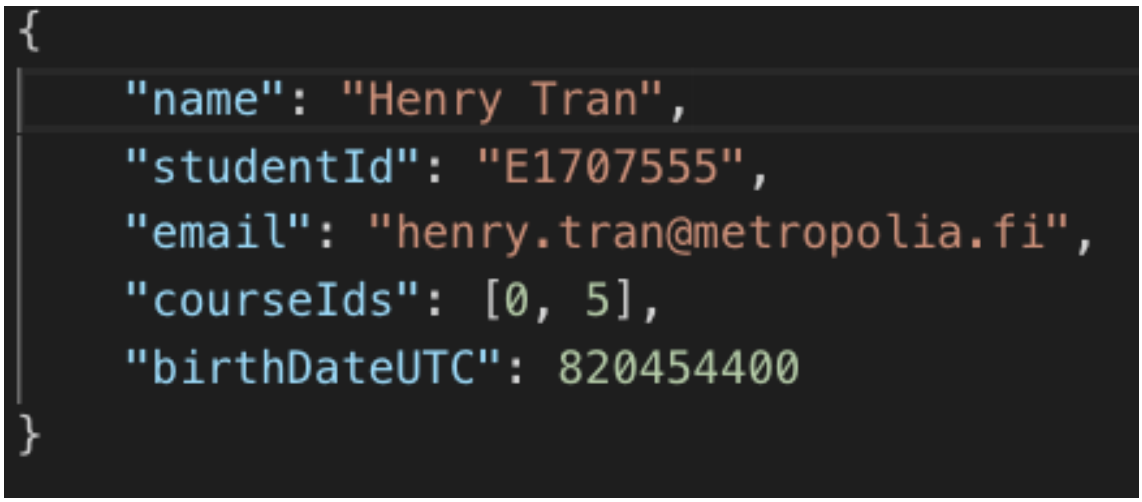
3.1 Over-fetching and under-fetching

Over-fetching and under-fetching are the most common issues that a REST API has. GraphQL tackles these issue by fetching data declaratively and only sends back the properties which the client explicitly specifies. [3] This issue can be demonstrated by the following example. A web application needs to show the name of a student and also the name of courses that they are taking. The following figure explains the data that the client needs from the server.

Figure 1. Data the clients need from the server

As shown in figure 1, even though the data that the client needs from the server is simple, the client still has to make three requests to get all of the information it needs. First, the

client must make a GET request to the REST API through the endpoint that the server exposes “/student/{id}”. The response of the server includes “name”, “studentId”, and other information of the student. The API will also provide the client with an array of IDs of all the classes that the student takes. The response can be seen in the below figure.



```
{
    "name": "Henry Tran",
    "studentId": "E1707555",
    "email": "henry.tran@metropolia.fi",
    "courseIds": [0, 5],
    "birthDateUTC": 820454400
}
```


Figure 2. Information of a student returned from a REST API

As can be seen from figure 2, In spite of the fact that the client only demands the name of the student and an array of IDs of all courses, other information related to the student will still be sent back to the client. Therefore, the client receives a large payload despite only querying for “name” and “courseIds”. Moreover, the client does not have the information about the courses in the first request. Therefore, more requests must be sent to the server due to under-fetching. After getting an array of IDs of all courses that the student participates in, the client still has to make two more requests to the server to get the information of each course. Two requests will be sent through the endpoint “/course/{id}” and the response of these requests can be described in the following figure.

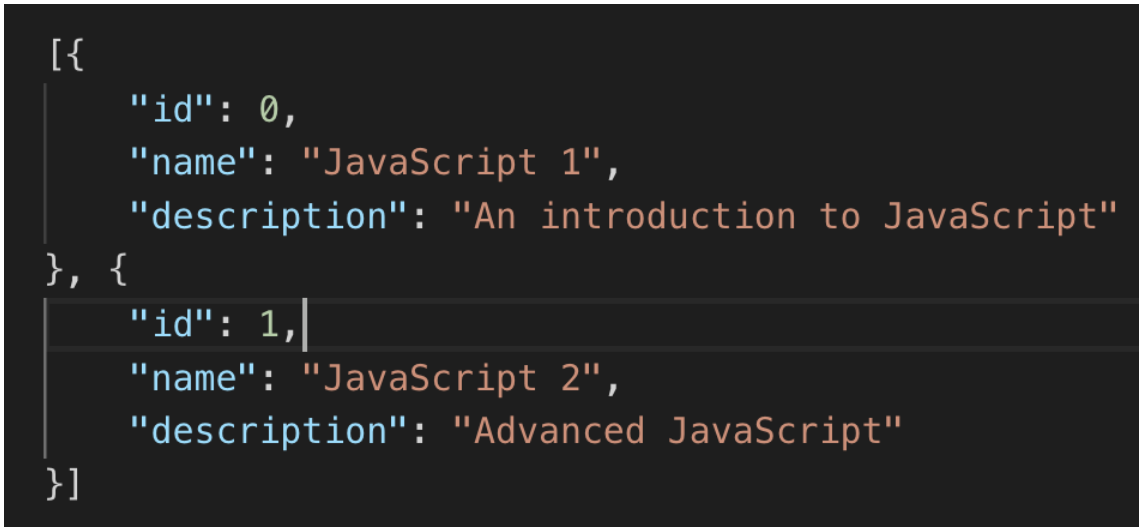


Figure 3. Information of courses a student participating in returned from a REST API

As shown in figure 3, an array of courses is returned from the server and each course has an id, a name, and a description. Once the client receives all three responses from the server, the data can be combined to satisfy the need of the application. Besides doing three round-trips to get the data for a simple user interface, the approach for getting the data is imperative. The client has to give an instruction to the server for how to fetch the data and how to process and combine it for the user interface. [4]

The REST API can overcome the issue mentioned above by implementing nested resources and understanding the relationship between a student and a course. For instance, the client can get all of the necessary data by making a request to "/student/{id}/course". However, this implementation for the REST API is not scalable because the server needs to keep adding custom endpoints to satisfy the need of the client. Additionally, developers have to maintain more custom endpoints [4]. To tackle this problem, the client uses a state manage library such as Redux or Mobx to aggregate the data from REST resources into the format that the user interface requires [5].

By using GraphQL, the client will not misuse the state management library and it can ask all the data it needs from the server in a single request. The client has to make just one request to the server with a GraphQL query through a single endpoint. On the other hand, the client needs to retrieve data from the server through multiple endpoints in a REST API.

Using GraphQL reduces points of access to data on the backend to one endpoint. Moreover, fetching all the initial data required by a view with a single round-trip to the server is always possible when using GraphQL. Therefore, the client can reduce the number of requests sent to the server. The response time from a GraphQL server can be significantly faster than a REST server because of fewer requests and not fetching the extraneous data, which brings a better user experience for the end users. The query and the response of GraphQL can be demonstrated in the following figure.

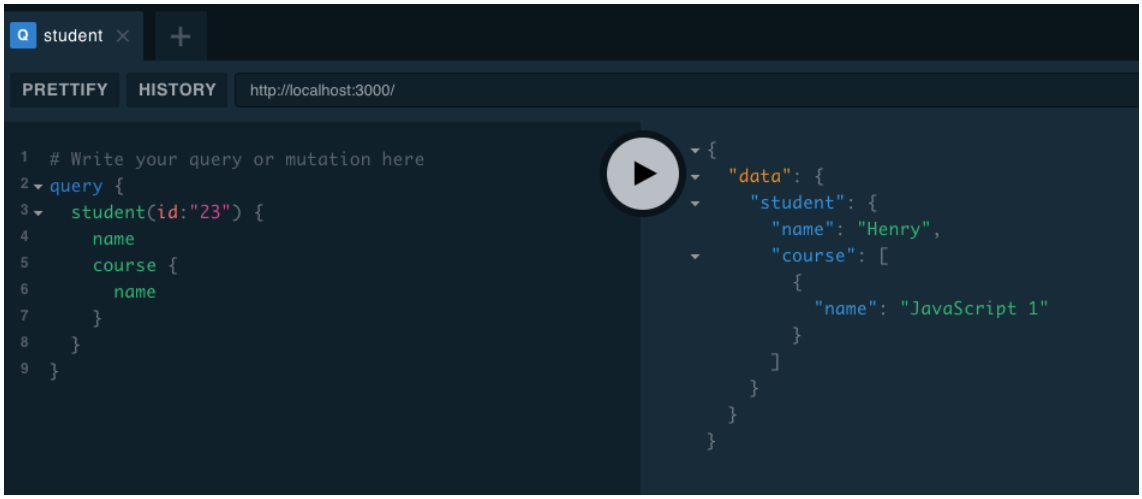


Figure 4. Query and response of getting student and course name

On the left of figure 4, a query for getting a student ID of 23 is issued. The name of the student and the name of the course are requested. On the right of figure 4, the response is displayed in JSON (JavaScript Object Notation) format. The response is in the exact shape of the data that the client asks for. Moreover, it does not include extra field, unlike REST. By switching to GraphQL, the complexity of the client-side state management is reduced. Therefore, the client can concentrate on rendering data in the user interface instead of managing and aggregating data.

3.2 Versioning

Versioning a REST API is essential when there are multiple clients which consume data from the REST API. Some clients require a specific version of the REST API as each client requires a different shape of data. Besides, a new version of an API is required when there are breaking changes. One of the common techniques to version a REST API is including a version number in the URL. [6] For instance, version one of the API

will have the following URL “https://metrpolia.com/api/v1/students”. And “https://metrpolia.com/api/v2/students” will be the URL of version two.

Another way of setting a version in a REST API is by using a custom HTTP header. For example, a request will include a header “X-API-Version” that will include a specific value of the version. However, it is more difficult to cache on the HTTP layer. Additionally, developers also need to handle requests which do not contain a version header, which can be quite complicated. [6]

Even though there are some benefits when versioning a REST API, developing a REST API with many versions is still an issue. Developers have to provide a new endpoint for each version. Furthermore, Versioning makes the code on both the frontend and backend application less maintainable. [4] Last but not least, developers need to maintain multiple endpoints and it can also lead to code duplication on the backend. Maintaining several versions of an API can make it more complicated to develop a scalable application.

Versioning a GraphQL service is possible. Nevertheless, developers can avoid versioning a GraphQL service. GraphQL server only returns the data that is explicitly requested from the client. Therefore, new fields and types can be added to the backend without affecting the client. [4] A GraphQL backend will help client applications to get continuous access to new features. In addition, the backend code becomes cleaner and more maintainable. For example, a field called “firstName” will be added to the student. If the client does not specify that it needs “firstName” in the query, the server will not return “firstName”. On the other side, a REST API will be affected as the server will send all of the recently added fields and types to the client.

In GraphQL, deprecating the API on a field level is a possibility. There is a directive which is an identifier preceded by a “@” character in GraphQL to mark a field is deprecated. The following listing shows how to use a directive to identify which field is going to be deprecated.

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

Listing 1. Deprecated directive in GraphQL

As can be seen from listing 1, “@deprecated” is used next to the field that will be deprecated. “@deprecated” is a directive provided in GraphQL to specify which field the client should not continue using. Besides, when the client queries a deprecated field, it will receive a deprecating warning from the server. When the field is not used by any client, it can be safely removed from the schema. GraphQL makes it easier to evolve an API without the need for versioning. On the other hand, a REST API developer has a difficult time figuring out which client is still using a deprecated field. Therefore, they have to continue supporting the deprecated fields.

3.3 Discoverability

Another advantage of using GraphQL is that the resources are discoverable by default, whereas REST API resources are not. In GraphQL, developers only need to create a schema. A schema is a representation of functionality such as query and mutation available to the client. [7] After creating the schema, the data structure will be discoverable by installing a developer tool called GraphiQL. GraphiQL is an in-browser IDE (Integrated Development Environment) for exploring a GraphQL server. The following figure shows how GraphiQL works.

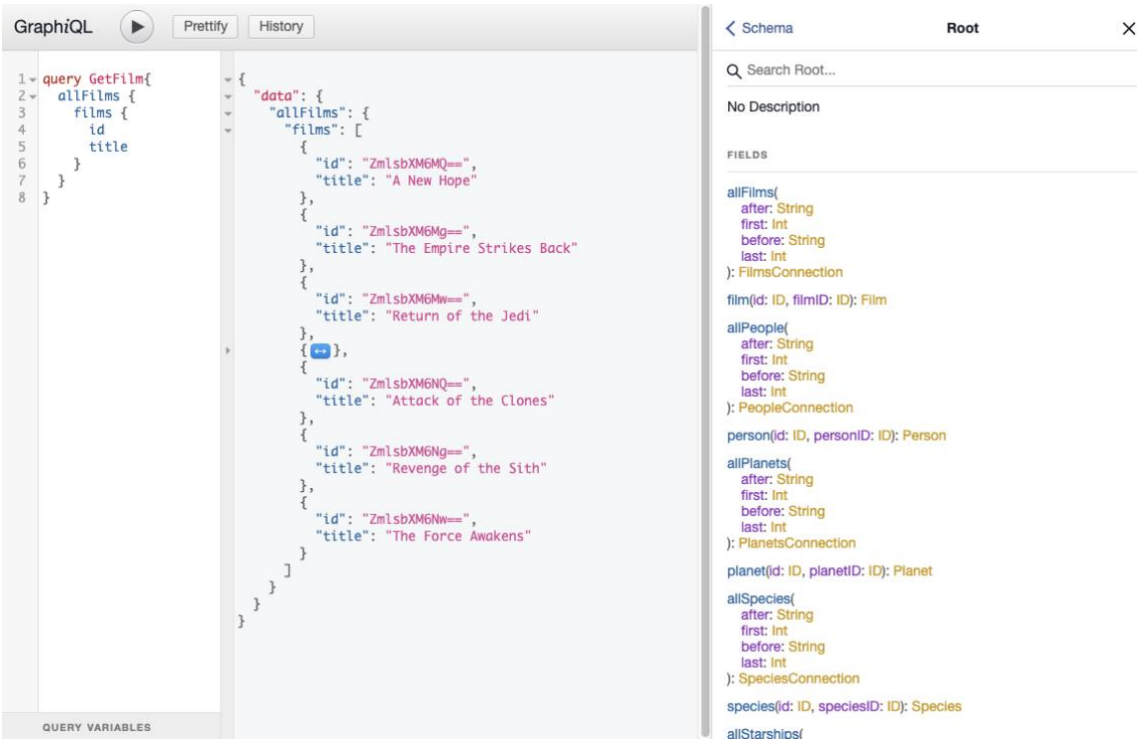


Figure 5. GraphiQL runs in a browser

On the left side of figure 5, users define the query to get the data they need. A query of all films is requested. Next, two fields of data: “id” and “title” are specified. In the middle of figure 5, GraphiQL shows the data returned from the query that users request. And there is the documentation in the right-hand corner of the IDE. The documentation shows all of the queries which users can use to get data from the server. GraphiQL is excellent at autogenerating documentation since all the information about the data which is available is defined in the schema.

When developing a REST API, if developers want to make the API discoverable, more libraries and efforts are required. First, a library such as Swagger has to be installed. Unlike GraphiQL, there is more configuration required and it can take several hours to set up the library. [3] In a REST API, developers need to manually mark which resources are available to the client. Therefore, if there are many endpoints, developers have to write documentation for each one.

4 Cons of GraphQL

4.1 Caching

First, implementing caching in GraphQL is more complex than implementing caching in a REST API. When using a REST API, re-fetching the same resources several times is easily avoided since REST uses HTTP caching. HTTP caching is available in REST because resources in REST are accessed through URL. Moreover, since the URL is a globally unique identifier, the client can take advantage of the URL to build a cache.

In GraphQL, every resource is accessed through the same URL. In addition, the query can be different even though the client requests the same resource from the GraphQL backend. For instance, a query for the first name and last name of a student called Henry is different from a query for only the first name of Henry. Last but not least, there is no built-in cache for the client, which means the clients have to cache on its end [3]. Nevertheless, if the clients use libraries for GraphQL such as Apollo or Relay, caching will be available out of the box.

4.2 File uploading

Second, the other major difference between GraphQL and REST is that while any type of file or content is supported in REST, only text or JSON is supported in GraphQL. GraphQL specification does not include file uploading. Consequently, developers need to implement this feature by themselves. There are several approaches to implement file uploading. Base64 encoding is one solution. Using a separate server that is not GraphQL is also an option. The third solution is using a library such as apollo-upload-server which follows GraphQL specification. [8]

4.3 Monitoring

Last but not least, GraphQL always returns 200 as an HTTP status code even though there is an error. On the other hand, a REST API can return other status codes such as 301, 404, and 500. These HTTP status codes carry the meaning of the error and give

developers a clue on how to fix it. Since an error is always presented as status code 200 in GraphQL, monitoring and error reporting tools are not leveraged.

5 Building a GraphQL server

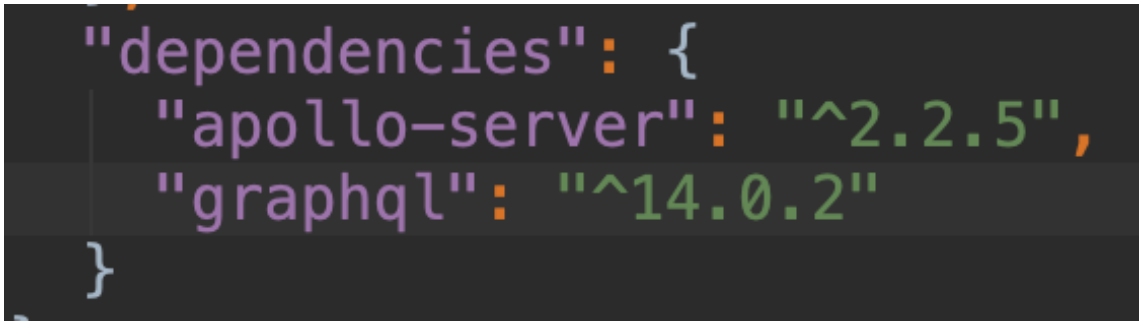
5.1 Setting up

This project is going to be written in JavaScript. The backend is a public API for an e-commerce application. The application is a CRUD (Create, Read, Update and Delete) API. The GraphQL server will provide a list of products, a product with a specific ID. It can also update the product and remove the product in the database. The GraphQL backend will be built on top of a Node.js server application. YARN, which is a package manager, will be installed to build and run the project. The following listing presents a list of development dependencies which are modules required only during development.

```
"devDependencies": {
  "@babel/cli": "^7.0.0",
  "@babel/core": "^7.0.0",
  "@babel/plugin-proposal-class-properties": "^7.0.0",
  "@babel/plugin-proposal-object-rest-spread": "^7.0.0",
  "@babel/preset-env": "^7.0.0",
  "babel-core": "7.0.0-bridge.0",
  "babel-eslint": "^8.2.1",
  "babel-jest": "^23.4.2",
  "cross-env": "^5.2.0",
  "eslint": "^4.15.0",
  "eslint-config-prettier": "^2.9.0",
  "eslint-config-standard": "^11.0.0",
  "eslint-friendly-formatter": "^3.0.0",
  "eslint-loader": "^1.7.1",
  "eslint-plugin-import": "^2.13.0",
  "eslint-plugin-jest": "^21.15.1",
  "eslint-plugin-node": "^7.0.1",
  "eslint-plugin-prettier": "^2.6.2",
  "eslint-plugin-promise": "^3.8.0",
  "eslint-plugin-standard": "^3.1.0",
  "graphql-codegen-core": "^0.14.5",
  "jest": "^23.6.0",
  "nodemon": "^1.18.3",
  "prettier": "^1.15.2",
  "rimraf": "^2.6.2",
  "supertest": "^3.3.0"
},
```

Listing 2. Development dependencies of the application

From this listing, it can be seen that Babel is used in this project to convert newer versions of ECMAScript such as ECMAScript 2017 and ECMAScript 2018 code into a backwards compatible version of JavaScript in current Node.js environments. Even though the current Node.js environment supports newer versions of ECMAScript, it still has not supported “import” and “export” yet. Therefore, Babel 7 needs to be installed. ESLint, which is a static code analysis, is installed to follow the coding style guidelines. Nodemon will also be installed to restart the Node.js application when code changes. Therefore, developers do not have to manually restart the server if there is any code change. Besides the development dependencies, dependencies which are modules required at runtime need to be installed. A list of required dependencies is demonstrated in the following listing.



Listing 3. Runtime dependencies of the application

Since this is a GraphQL application, graphql library is required which can be shown in listing 3. The graphql module is a JavaScript reference implementation for GraphQL. graphql is installed to build a schema and perform queries based on that schema. Besides that, apollo-server which is a GraphQL server built and supported by the developer community is installed. apollo-server allows developers to concentrate on establishing the shape of your data and how to fetch it.

After installing all of the dependencies, the next step to run the application is creating the server. The following listing provides instruction on how to build the GraphQL server.

```
import { ApolloServer } from 'apollo-server'
import { loadTypeSchema } from './utils/schema'
import { merge } from 'lodash'
import config from './config'
import { connect } from './db'
import product from './types/product/product.resolvers'
import coupon from './types/coupon/coupon.resolvers'
import user from './types/user/user.resolvers'

const types = ['product', 'coupon', 'user']

export const start = async () => {
  const rootSchema = `
    schema {
      query: Query
      mutation: Mutation
    }
  `
  const schemaTypes = await Promise.all(types.map(loadTypeSchema))

  const typeDefs = [rootSchema, ...schemaTypes]

  const server = new ApolloServer( config: {
    typeDefs,
    resolvers: merge({}, product, coupon, user)
  })

  await connect(config.dbUrl)
  const { url } = await server.listen( opts: { port: config.port })

  console.log(`GQL server ready at ${url}`)
}

start()
```

Listing 4. Instruction to build and run GraphQL server

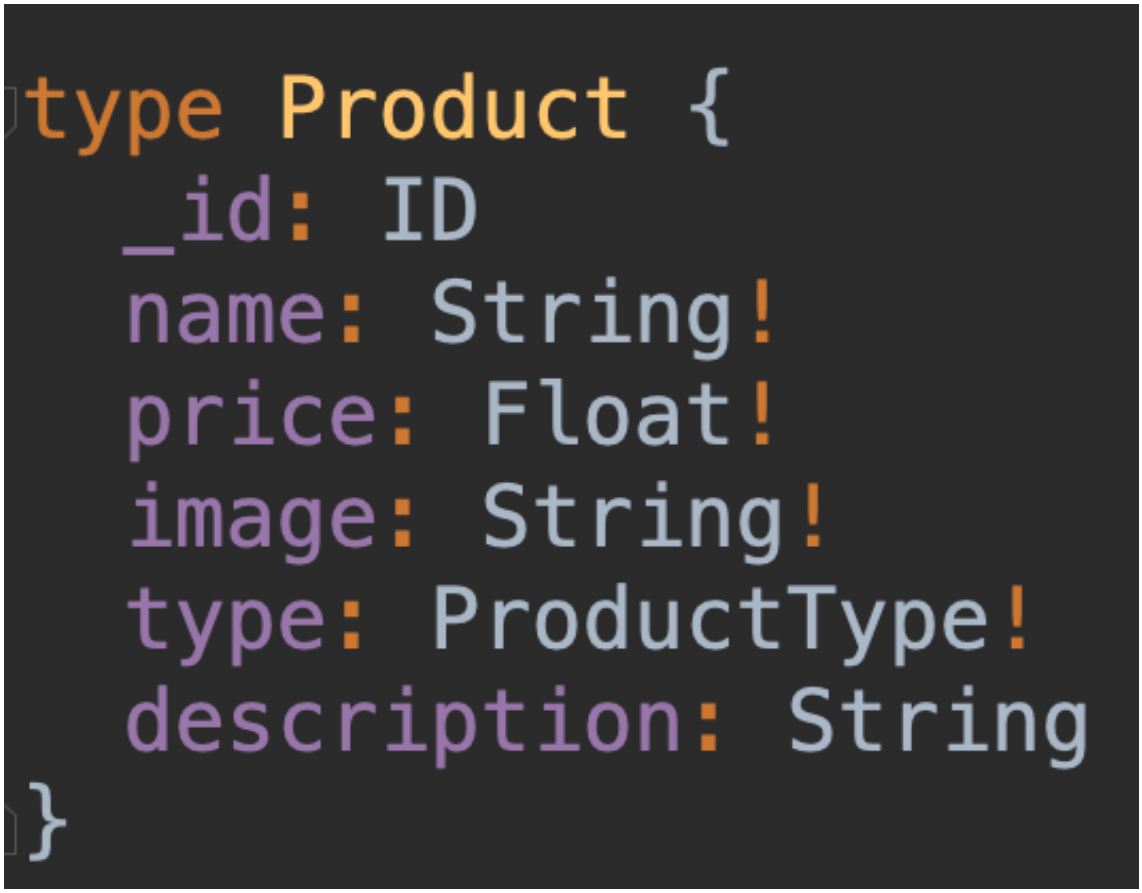
As follows from the figure shown above, dependencies that are required to set up the server are imported in “index.js”. After that, a function called “start” is defined. Then, a type definition, which specifies the shape of the data and indicates how to fetch the data from the GraphQL server, is initiated. The type definition consists of the root schema and other schemas in the application.

Schemas are split into different folders so that it is easier to maintain and it is more readable. Consequently, resolvers have to be defined to provide instruction on how to fetch the types specified in the schema. The resolvers need to exactly match the type definitions. Otherwise, a field or a mutation inside the schema will not be executed. After creating an instance of the server with “ApolloServer”, a connection to MongoDB database is established. Finally, “server.listen()” is called to launch the server.

5.2 Writing schema

A GraphQL schema is for expressing the types available in the schema. It is also used for defining the kind of resources are available for querying and, how they relate to each other, and how the client can query them. [2] On the other hand, a database schema is for keeping the data consistent when it enters the database. Both the database schema and the GraphQL schema can be the same. If a database schema is available, it will be a good starting point and reference for the GraphQL schema. Moreover, when there is an incoming request query, it will be validated against the schema before the resolver and the database run. In GraphQL, the validation is going to happen before the resolvers execute. The resolvers are only executed if the query is valid. Additionally, if there is an error before the resolvers run, GraphQL will point out the reason why the query is invalid.

There are several ways to create the schema. Developers can either use GraphQL SDL (Schema Definition Language) or GraphQL introspection query result, or GraphQLSchema object from the graphql.js library to create a schema. GraphQL SDL is the most popular way to define a GraphQL schema and it is also used in the GraphQL specification. An example of writing a schema with GraphQL SDL is presented in the following listing.



Listing 5. “Product” type is written with SDL

Despite the similarity between SDL and JavaScript, SDL must be stored as a string. Since SDL is just a string, developers can use GraphQL SDL to write down a GraphQL schema in a language-agnostic way. Furthermore, it is easy to read and it can be composable. Last but not least, it can be reused in other languages besides JavaScript as SDL is only a string.

In this application, multiple smaller schemas such as “Product”, “Coupon”, and “User” schemas are combined into one schema instead of creating only one large schema. The former approach makes the application more scalable than the latter one when there are going to be more types and fields. In the preceding schema, a type called “Product” is defined. The “Product” type is an object type which represents a group of fields. Each field type is a reference to another type which can either be an object type or a scalar type. Scalar types are built-in primitives and those are “String”, “Int”, “Float”, “Boolean”, and “ID”. For example, the types of “_id”, “name”, “price”, “image”, and “description” are scalar types. And the type of “type” is an object type. “type” returns a type called

“ProductType” which is an enumeration type. The following listing indicates how to write an enum definition.

```
enum ProductType {
  FOOD
  CLOTHES
  OTHER
}
```

Listing 6. Enum definition of ProductType

An enumeration type, which is also called as an enum, is a special kind of scalar type that only has a set of allowed values. Enums help to validate if any arguments of “ProductType” is either “FOOD”, “CLOTHES”, or “OTHER”.

Besides the above types, type modifier can be added to the schema to describe the validation of values in the query. The following figure shows how to define a Non-Null type modifier in the “Product” type.

```
name: String!
```

Figure 6. Non-nullable name in Product type

From this figure, it can be seen that the field name is a “String” type and it is marked as not nullable by adding an exclamation at the end. This means that the server always returns a non-nullable value for “name”. For example, if a null value is passed as an argument, GraphQL server will throw a validation error which can be seen in the following figure.

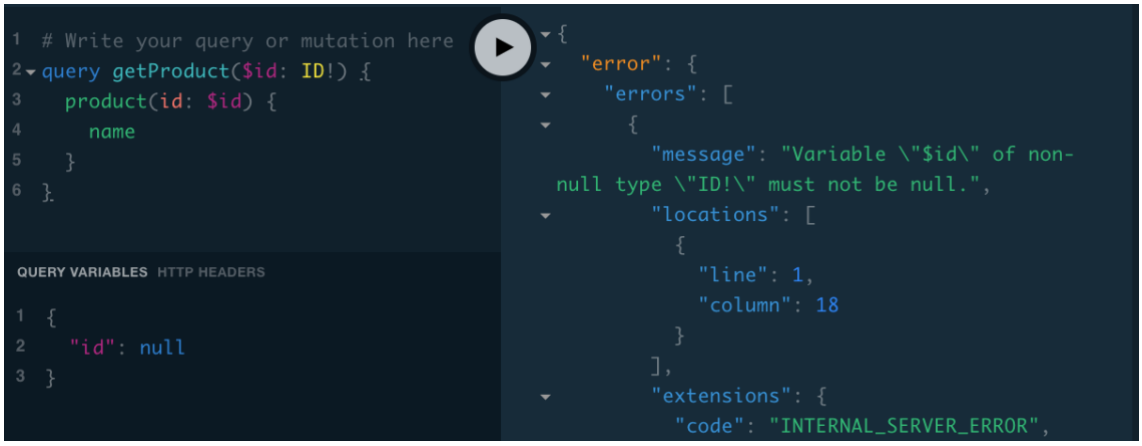


Figure 7. Validation error for the non-nullable value

As follows from the figure shown above, GraphQL will explicitly point out the error. In the "message" of "errors", it indicates which field is not supposed to be not null. In the above figure, GraphQL server throws an error because "id" is defined to be not nullable and a null value is passed as an argument.

Besides Non-Null type modifier, another type of modifier is Lists. A list reveals that this field will return an array of that type. A list is defined by wrapping the type in square brackets. The following figure describes how to define a Lists type modifier.

```
image: [String!]!
```

Figure 8. List of a non-nullable image

From this figure, it can be seen that "image" is an array of non-nullable string. Therefore, "image" cannot be a null value or it cannot contain a null value in a list either.

Besides normal object types, query and mutation are special types in a schema. The following listing shows how to define a Query type and a Mutation type.

```
const rootSchema = `
  schema {
    query: Query
    mutation: Mutation
  }
`
```

Listing 7. Root schema definition

In the preceding root schema definition, a Query type and a Mutation type are defined. Both describe what resources are available for clients to interact with the GraphQL API. And they would be similar to routes in a REST API. A Query type is the same as an object type and it is used to define a query which is for fetching data. However, it is a special type because it describes the entry point of every GraphQL query. Moreover, Query type specifies which queries the GraphQL service is capable of handling. [2] An instruction on how to define the Query type is described in the following listing.

```
extend type Query {
  products: [Product]!
  product(id: ID!): Product!
}
```

Listing 8. A Query type for Product

As listing 8 illustrates, the “Product” query needs to be prefixed with the “extend” keyword since a Query type is already created in the root schema. If there is no “extend" keyword, GraphQL will try to re-create a Query type and complain that a query already exists. Thanks to the extending functionality, each module can have its own Query type. In the

preceding Query type, there are two types of queries which are available on this “Product” module. If there is any kind of data that the public API needs to serve, a query for that has to be written. Otherwise, if a query is not defined for a resource, that resource will not be available to the clients. Therefore, only the “product” and “products” queries are available for the client to query.

The preceding Query type defines a query called “products” for getting a list of products and a query called “product” for getting a product with a specific id. “Product” query will return a type called “Product” and “Products” query will return an array of “Product” type. If the backend is implemented in REST architecture, these resources will be located on separate endpoints such as "/api/products" and "/api/product". Nevertheless, “product” and “products” can be queried at the same time and returned to the client at once in GraphQL. The following figure shows how a query for getting a list of products and a product with a specific id looks like.

Figure 9. A query to get product and products on the client side

As demonstrated in figure 9, the client can request a list of products and a specific product by sending only one single query. The “Product” type cannot be specified as the only field of the query since “Product” is an object type. Therefore, sub-selections of “Product” fields need to be provided as well. In this case, the client is only interested in the name of the product so only the field name is selected as a sub-field of the product.

In the preceding figure, the keyword “query” which is an operation type and the operation name can be omitted. There are three operation types in GraphQL: “query”, “mutation”, and “subscription”. Each type describes the operation which is going to be executed and

operation type is only required for “mutation” and “subscription” type. The operation name is encouraged to use since it is helpful for debugging and logging. Without the operation name, the operation will be anonymous. Consequently, it is more difficult to trace where the issue comes from.

Mutation type is similar to the Query type; however, the Query type is used just for reading the database while Mutation type is used for mutating the database. Mutations are responsible for creating, updating, or deleting data in the server. Verbs on a REST API such as PUT, POST, PATCH, and DELETE can be comparable to mutations [2]. Similar to how data-fetching operations on a GraphQL server are defined in the Query type, the root Mutation type specifies the entry-points for data-manipulation operations. A mutation called “newProduct” to create a product is defined in the following listing.

```
extend type Mutation {
  newProduct(name: String!, price: Float!, image: String!,
  description: String, type: ProductType!): Product!
}
```

Listing 9. “newProduct” mutation defined in a Mutation type

As shown in listing 9, the Mutation type of “Product” type has a “newProduct” mutation which accepts “name”, “price”, “image”, “description”, and “type” as arguments. “name”, “image”, and “description” field have String types. “price” has a Float type and “type” has an enum type. And this mutation will return the newly-created “Product” object. In the below figure, a “newProduct” mutation is sent to the GraphQL server and the response from the server can be seen on the right.

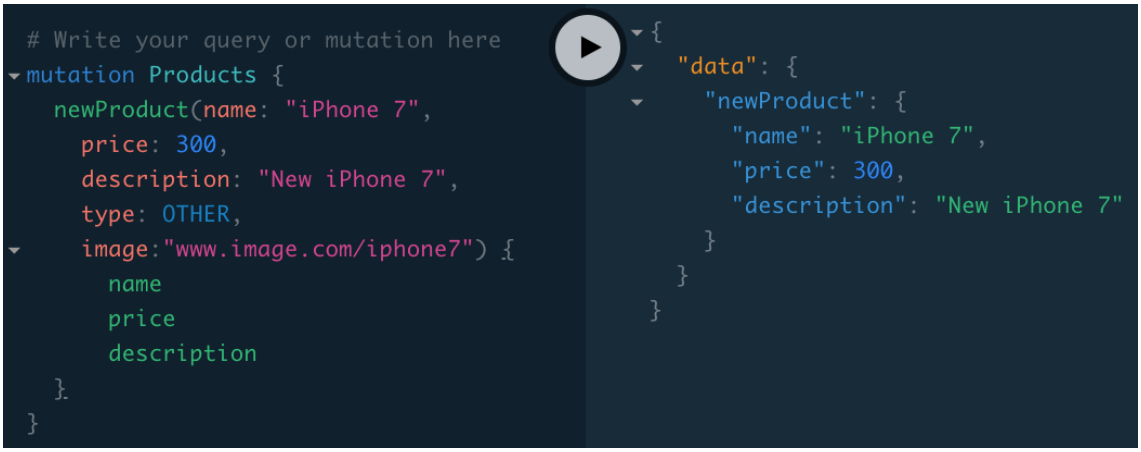


Figure 10. Mutation sent to the server and its response

As shown in figure 10, the “Product” object will match the previously-created “Product” type defined in listing 5. Similar to the Query type, fields of “Product” type need to be specified when sending a mutation to the server. The name of the product along with the price and description of the product is requested. Moreover, several mutations can be sent in the same request. Nevertheless, they will be executed in the exact order they are defined to avoid race-conditions within the operation.

Even though a list of arguments can be passed to the mutation, it will be verbose. Moreover, if there are multiple mutations with the same input parameters, those fields will be repeated several times in each mutation. Fortunately, there is a type called Input type to resolve this issue. The following listing describes how to define an input type for creating a new product.

```
input NewProductInput {
  name: String!
  price: Float!
  image: [String!]
  type: ProductType!
  description: String
}
```

Listing 10. “NewProductInput” input type

As can be seen from listing 5 and listing 10, an input type is similar to an object type. However, the keyword is “input” instead of “type”. In GraphQL, an object type cannot be reused for an input type. The reason is that an object type can have circular references or references to interfaces and unions, neither of which is appropriate for an input type. The following figure explains how to use an input object type in a mutation.

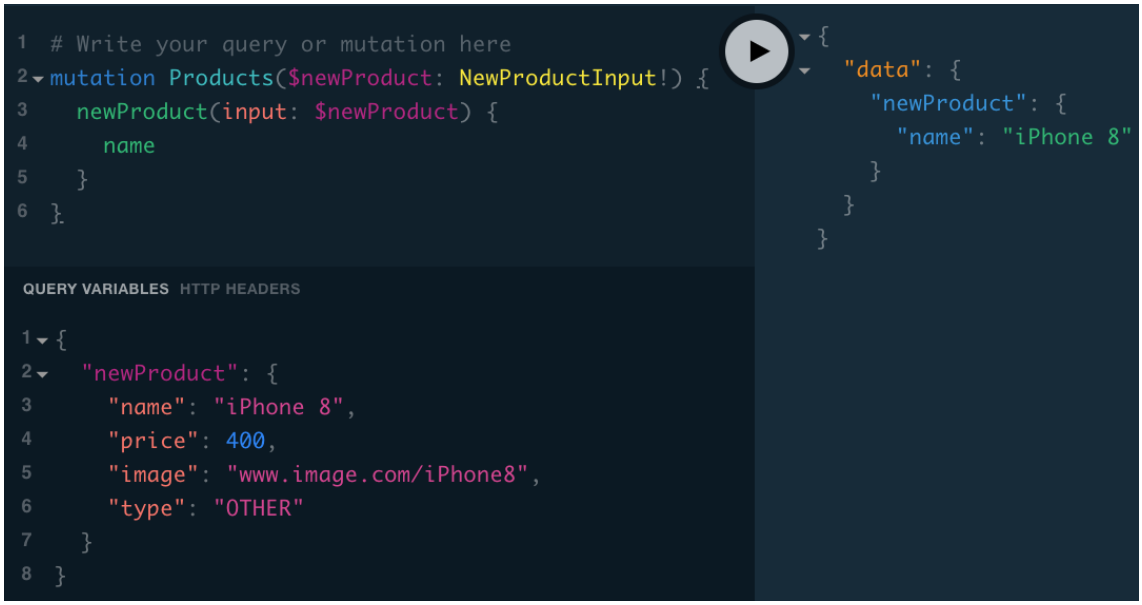


Figure 11. Mutation sent to the server with input type

The input is provided as a variable to the mutation. In the “Products” mutation, a variable “newProduct” of type “newProductInput” is required. And that variable will be the input of “newProduct” mutation.

“newProuct” mutation is similar to a POST request to create a product in a REST API. After refactoring “newProduct” mutation to use an input type, mutations for updating the product and deleting product are needed for the application. The following listing describes all the mutations of the Product type and the input for each mutation.

```
input NewProductInput {
  name: String!
  price: Float!
  image: String!
  type: ProductType!
  description: String
}

input UpdateProductInput {
  name: String
  price: String
  image: String
  description: String
}

extend type Mutation {
  newProduct(input: NewProductInput!): Product!
  updateProduct(id: ID!, input: UpdateProductInput!): Product!
  removeProduct(id: ID!): Product!
}
```

Listing 11. Mutations of “Product” type

As listing 11 illustrates, “updateProduct” is for updating a product with a specific id. “updateProduct” needs to be provided with the id of the product and the input of “UpdateProductInput” type. The input of “newProduct” mutation is different from the input of “updateProduct” mutation since there are some fields which are not allowed to be updated in “updateProduct” mutation. In this case, a field called “type” is not allowed to be updated. A PUT request to update a resource in a REST API is similar to the “updateProduct” mutation. And “removeProduct” is for deleting a product with a specific id and it is similar to a DELETE request in a REST API.

5.3 Resolvers

Similar to controllers in a REST API, resolvers are responsible for getting data from the database or other data sources. If there is a query or a mutation defined in a schema, there must be a resolver which returns the type specified in the schema. Resolvers provide the instructions for turning a GraphQL operation such as a query or a mutation into data. Moreover, resolvers return data which has the same type specified in the schema or a promise for that data is returned in the resolvers. The following figure shows what a resolver function looks like.

```
fieldName = (parent, args, context, info) => data
```

Figure 12. Example of a resolver

As figure 12 illustrates, a resolver function accepts four arguments. First, the “parent” argument receives the result returned from the previous resolver or the parent type. Second, the “args” argument is an argument provided to the query. Thirdly, the “context” argument is an object shared by all resolvers and it is often used to hold contextual information such as authentication information. Moreover, “context” is also used to access data sources. Last but not least, the “info” argument is an object contains the abstract syntax tree of the incoming query. It has information about the current query and the schema detail. Therefore, the “info” argument can be used for optimizing database calls. [2] The server can inspect the “info” argument to query the database for the exact fields that the client explicitly asks for. For example, if the client wants to get only the name of the product, “name” will be the only field queried from the database.

In a resolver, if there is an error, the server will not crash unlike a REST API built in Express.js. Developers do not have to worry about handling the error in a resolver to avoid crashing the server because all of the resolvers are wrapped in a try catch. Instead of crashing the server when there is an error in a REST API, partial data and the error will be sent back to the client in GraphQL.

In the “Product” module, there are two queries and three mutations. The following listing describes the queries and mutations available for the “Product” type.

```
extend type Query {
  products: [Product]!
  product(id: ID!): Product!
}

extend type Mutation {
  newProduct(input: NewProductInput!): Product!
  updateProduct(id: ID!, input: UpdateProductInput!): Product!
  removeProduct(id: ID!): Product!
}
```

Listing 12. Query and Mutation of “Product”

A resolver must be written for each query or mutation. Therefore, there will be five resolvers which need to be implemented. “products” query is for getting all of the products in the database and “product” is for getting a product with a specific ID. And there are three mutations created for creating, updating, and deleting the resources. The implementation for all resolvers of the “Product” module is illustrated in the following listing.

```
const product = (_, args, ctx) => {
  return Product.findById(args.id)
    .lean()
    .exec()
}

const products = (_, args, ctx) => {
  return Product.find({})
    .lean()
    .exec()
}

const newProduct = (_, args, ctx) => {
  return Product.create(args.input)
}

const updateProduct = (_, args, ctx) => {
  const update = args.input
  return Product.findByIdAndUpdate(args.id, update, options: { new: true })
    .lean()
    .exec()
}

const removeProduct = (_, args, ctx) => {
  return Product.findByIdAndRemove(args.id)
    .lean()
    .exec()
}
```

Listing 13. “Product” resolvers

As shown in listing 13, the first argument of all the resolvers is always empty since it refers to the root of the graph. In this project, the database is MongoDB and Mongoose is used to query the database. A product model is created with Mongoose and it exposes methods such as “findById”, “find”, “create”, “findByIdAndUpdate”, and “findByIdAndRemove” to access the database. These methods are used in the resolvers to query against the database. In the “product” resolver, the “id” is passed to the “args” argument and it is used to get, update, and delete a product. “products” resolver does not need any argument as it only needs to get all of the products in the database. Last but not least, “newProduct” resolver and “updateProduct” resolver require input from the query to create or update the product. “updateProduct” needs “id” in the “args” argument because it needs to find the id in the database.

5.4 Testing

Nowadays, testing is an essential part of software development and it is always a good practice to write tests for an application. It helps developers to prevent potential bugs and keep the application work normally while they work on new features or refractor an existing one. Fortunately, it is easy to write tests in GraphQL.

To set up for testing in GraphQL, a MongoDB database is created in every single block test. The database will be shut down and deleted after each test runs, which make the tests run in parallel and keep them fast.

The first thing to test in GraphQL is testing schema composition. Testing schema helps developers to avoid a situation when a developer changes a schema and the client applications do not work anymore. The following listing represents how to write a test for the schema definition.

```
import { buildSchema } from 'graphql'
import { schemaToTemplateContext } from 'graphql-codegen-core'
import { loadTypeSchema } from '../../../utils/schema'
import { mockServer } from 'graphql-tools'

describe('Product schema', () => {
  let schema, typeDefs
  beforeAll(async () => {
    const root = `
      schema {
        query: Query
        mutation: Mutation
      }

    `
    const typeSchemas = await Promise.all(['product'].map(loadTypeSchema))
    typeDefs = root + typeSchemas.join(' ')
    schema = schemaToTemplateContext(buildSchema(typeDefs))
  })

  it('Product has base fields', () => {
    let type = schema.types.find(t => {
      return t.name === 'Product'
    })

    expect(type).toBeTruthy()

    const baseFields = {
      _id: 'ID',
      name: 'String!',
      price: 'Float!',
      image: 'String!',
      type: 'ProductType!',
      description: 'String'
    }

    type.fields.forEach(field => {
      const type = baseFields[field.name]
      expect(field.raw).toBe(type)
    })
  })
```

Listing 14. Tests for schema and types

As listing 14 illustrates, a schema is created before all of the tests run. Creating a schema in a testing environment is the same as creating a schema in a development environment. After it is created, a variable that contains all of the expected fields is created. A variable called “baseFields” is initialized and it has all of the field names and field types of the “Product” type. Then “Product” type is verified to have the correct field names and types. These tests will save developers from bugs such as typos and type conflicts. Moreover, developers can avoid breaking the application when a field is renamed, deleted, or added.

Secondly, queries and mutations are essential to be tested. They can be tested by creating a mock GraphQL server. Developers do not have to mock the server by themselves since there is a module from graphql-tools to create a mock server. Only the schema, which is already created in listing 14, is needed to set up the mock server. Therefore, queries and mutations are easy to be tested, which can be demonstrated in the following listing.

```
it('products query', async () => {
  const server = mockServer(typeDefs)
  const query = `
    {
      products {
        name
        price
        type
      }
    }
  `
  await expect(server.query(query)).resolves.toBeTruthy()
  const { errors } = await server.query(query)
  expect(errors).not.toBeTruthy()
})

it('newProduct mutation', async () => {
  const server = mockServer(typeDefs)
  const query = `
    mutation CreateNewProduct($input: NewProductInput!) {
      newProduct(input: $input) {
        name
        price
        image
        type
        description
      }
    }
  `
  const vars = {
    input: {
      name: 'iPhone 4',
      price: 450,
      type: 'OTHER',
      image: 'www.image.com/iphone4',
      description: 'iPhone 4'
    }
  }
  await expect(server.query(query, vars)).resolves.toBeTruthy()
  const { errors } = await server.query(query, vars)
  expect(errors).not.toBeTruthy()
})
})
```

Listing 15. Tests for queries and mutations

As shown in listing 15, the mock server instance exposes a function called “query” to run the query and mutation. The query and mutation are defined in a string and then passed as the first argument of the query function. Since “products” query does not need any input, the query is executed with the query as the first argument. On the other hand, the “newProduct” mutation needs to be provided with an input. Hence, an input is passed as the second argument of the query function. As the server which runs in these tests is a mock server, the result does not need to be checked if the response is correct or not. Developers only need to verify if the query resolves and there is no error returned from the mock server.

The last part to test when developing a GraphQL server is resolvers. Testing a resolver is simpler than testing a schema because a resolver is just a pure JavaScript function and it does not depend on GraphQL. Therefore, writing tests for a resolver is similar to writing tests for a normal JavaScript function. Furthermore, there is no internal state and no mutable payload in the resolver. A resolver can be tested by mocking the shape of these arguments. The following listing gives an example of writing a test for a resolver.

```
import resolvers from '../product.resolvers'
import { Product } from '../product.model'

describe('Resolvers', () => {
  test('products gets all products', async () => {
    const products = await Product.create([
      {
        name: 'iPhone 7',
        price: 200,
        image: 'www.image.com/iphone7',
        type: 'OTHER',
        description: 'iPhone 7'
      },
      {
        name: 'iPhone XS',
        price: 500,
        image: 'www.image.com/iphonexs',
        type: 'OTHER',
        description: 'iPhone XS'
      }
    ])

    const result = await resolvers.Query.products( _: null,  args: {})

    expect(result).toHaveLength(2)
    products.forEach(p => {
      const match = result.find(r => `${r._id}` === `${p._id}`)
      expect(match).toBeTruthy()
    })
  })
})
```

Listing 16. Test for a resolver

As can be seen from listing 16, “resolvers” and the “Product” model are imported. First, two products are created in the MongoDB database. There will not be any duplicated product since the database is created on the fly and deleted after the test finishes. After creating the products, the resolver for querying every product is executed. Since the query is defined in the root schema, null is passed as the first argument which refers to the parent of the resolver. Moreover, there is no input for the query so an empty object is passed to the second argument which is the “args” argument. After waiting for the function to resolve, the response of the resolver is checked if it has the same length as the length of an array of products created in the database. Furthermore, the response of the products resolver is iterated to verify each product contains the same id as the one created in the database.

6 Building a GraphQL client

The front-end application is a web application and it is written in JavaScript. The client will communicate with the server built in GraphQL previously. The user interface of the front-end application can be shown in the following figure.

Figure 13. User interface of the front-end application

As figure 13 illustrates, the web application can show a list of products and it allows users to add a new product to the list. Since the client is simple and straightforward, React is chosen to be the user interface library. create-react-app is used to generate the React application because it does not require any build configuration. apollo-client is used to manage remote data in the React application since it has declarative data fetching and it does not need any configuration for caching.

First, there are packages need to be installed: apollo-boost, grahql, and react-apollo. apollo-boost is a package to set up Apollo Client. graphql library is installed to parse the GraphQL queries. Since the client is a React application, react-apollo is installed as the view layer integration for React. After all the dependencies are installed, an endpoint for the GraphQL server needs to be provided for the front-end application. The following listing demonstrates how to provide the endpoint to the client.

```
import ApolloClient from "apollo-boost";
import { ApolloProvider } from "react-apollo";

const client = new ApolloClient({
  uri: "http://localhost:3000"
});

class App extends Component {
  render() {
    return (
      <ApolloProvider client={client}>
        <div className="App">
          <Products />
          />
        </div>
      </ApolloProvider>
    );
  }
}
```

Listing 17. Connect React and Apollo Client

ApolloClient is imported from apollo-boost to create a configuration object for the client. Then, the endpoint for the GraphQL server is provided through the “uri” property of the configuration object. To connect Apollo Client to React, ApolloProvider is imported from apollo-boost and wrapped around the root component. If Redux or React Router is used, ApolloProvider will be outside of the redux Provider component and the root route component [9].

Once the React application is connected to Apollo Client, Query is imported to request data from the GraphQL server. Query component requires a query to fetch data and it passes the data to its children to render the user interface. First, Query will check if there is a result for the query in Apollo cache and then load the result. If there is no result in the cache, a request is sent to the server [10]. Moreover, the result is updated reactively since the Query component subscribes to the result. The following listing shows how to request data using the Query component.

```
export const GET_PRODUCTS = gql`
  query Products {
    products {
      name
      price
    }
  }
`;

export const Products = () => (
  <Query query={GET_PRODUCTS}>
    {({ loading, error, data }) => {
      if (loading) return <p>Loading...</p>;
      if (error) return <p>Error :(</p>;

      return (
        <div className="products">
          {data.products.map(({ name, price }) => (
            <div key={name}>
              {name}: {price}€
            </div>
          ))}
        </div>
      );
    }}
  </Query>
);
```

Listing 18. Query component

As illustrated in listing 18, a variable called “GET_PRODUCTS” is initialized with gql function to wrap the GraphQL query string. “GET_PRODUCTS” is a query for getting all of the products. Then, a function is provided to the children property of Query to determine what to render, which will have information about the loading state, error, and data returned from the client [10]. Thanks to Apollo Client, error and loading state are tracked automatically. On the other hand, developers have to manually trace error and loading state if they use redux-saga or redux-thunk to fetch data.

In the front-end application, it will first show “Loading…” if the application is fetching data. After checking the loading state, it will check for an error. If there is an error from the GraphQL server, it will show “Error :(“ to users. After fetching data is done and there is no error, an array of products will be available in data property. Finally, the name and price of these products are rendered in the web application.

Once reading data from the server, writing data to the server can be added to the client. First, a form that has “name”, “price”, “image”, “type”, and “description” is rendered to the client. The following listing explains how to initialize the values of the form and update them.

```
class App extends Component {
  state = {
    newProduct: {
      name: "",
      price: 0,
      image: "",
      type: "",
      description: ""
    }
  };

  handleChange = (event, type) => {
    const {
      target: { value }
    } = event;
    const newProduct = { ...this.state.newProduct };
    newProduct[type] = type === "price" ? parseFloat(value) : value;
    this.setState( state: { newProduct });
  };

  render() {
    return (
      <ApolloProvider client={client}>
        <div className="App">
          <Products />
          <ProductForm
            newProduct={this.state.newProduct}
            handleChange={this.handleChange}
          />
        </div>
      </ApolloProvider>
    );
  }
}
```

Listing 19. Using React form

As can be seen from listing 19, values of the form are stored in a variable called “newProduct”. “newProduct” is a React state and it is updated in “handleChange” function of App component. “handleChange” function is executed when any value of the form changes.

Mutation is imported to send mutations to the GraphQL server from the user interface. The Mutation component is created the same way as the Query component. The following list explains how to use Mutation to write data to the server.

```
import { Mutation } from "react-apollo";

export const ADD_PRODUCT = gql`
    mutation AddProduct($input: NewProductInput!) {
        newProduct(input: $input) {
            name
            price
        }
    }
`;

export const AddProduct = ({ newProduct }) => {
    return (
        <Mutation
            mutation={ADD_PRODUCT}
            variables={{ input: newProduct }}
            update={(cache, { data: { newProduct } }) => {
                const { products } = cache.readQuery({ query: GET_PRODUCTS });
                cache.writeQuery({
                    query: GET_PRODUCTS,
                    data: { products: products.concat([newProduct]) }
                });
            }}
        >
            {addProduct => <button onClick={addProduct}>ADD</button>}
        </Mutation>
    );
};
export const GET_PRODUCTS = gql`
  query Products {
    products {
      name
      price
    }
  }
`;
```

Listing 20. Mutation component

As listing 20 illustrates, “ADD_PRODUCT” contains the mutation string for adding product wrapped with gql function. Then, “ADD_PRODUCT” is passed to the mutation property of Mutation. The first argument of the render prop function provides the mutation function [11]. The mutation function called “addProduct” is used to submit the form.

“addProduct” is passed to the “onClick” handler of the button. The input of “AddProduct” mutation is provided through property “variables” of Mutation. “variables” of the mutation is “newProduct” and “newProduct” is a property passed from the root component in App.js. The root component passes “newProduct” to “ProductForm” from React state. “newProduct” is passed to “AddProduct” component since “AddProduct” is a child of “ProductForm” and “newProduct” is passed to “AddProduct”. The second argument of the render prop function is the mutation result. The mutation result has a loading state, error, and data.

After performing the mutation, the GraphQL server and the Apollo cache become out of sync because there are more products in the server than the cache [11]. Therefore, the query for getting all of the products needs to be updated through Apollo Client. By using the “update” function, the cache of the mutation can be updated [11]. First, “cache.readyQuery” is used to read the data from the cache. Then, the query for getting a list of products is provided for “cache.readQuery”. Next, the list of products is updated by using “cache.WriteQuery” to write data back to the cache. Finally, the user interface will update reactively with the new product once it comes back from the GraphQL server.

7 Conclusion

To summary, this thesis introduces briefly about GraphQL and discusses the benefits and drawbacks of using GraphQL. Over-fetching and under-fetching have been one of the pain points which developers face while developing a REST API. Thanks to the declarative data fetching implementation of GraphQL, it has solved this issue and eased other pain points. Even though there are still some cons of using GraphQL such as caching and file uploading, the pros still far outweigh the cons. Additionally, there are solutions that resolve these disadvantages of GraphQL.

A minimal CRUD API is implemented in this project to illustrate how to build and test a GraphQL application in JavaScript. The backend is built with Node.js and apollo-server. On the front-end, the application is built with React and apollo-client. The implementation of the back-end and the front-end shows how easy it is to adopt and develop a GraphQL application.

All in all, GraphQL is the potential successor for REST since it is easy to adopt and it brings multiple advantages compared to REST. The technology has become more matured and it also gets supported by the developer community and large companies.

References

1 Robin W. Why GraphQL: Advantages, Disadvantages & AlternativesURL: https://www.robinwieruch.de/why-graphql-advantages-disadvantages-alternatives/. Accessed: 9 April 2019.

2 Brian K. Beginning GraphQL. Birmingham: Packt Publishing; 2018.

3 Sheena. GraphQL versus REST; 6 December 2018. URL: https://www.codementor.io/sheena/graphql-versus-rest-pp19tr6zk. Accessed: 9 April 2019.

4 Samer B. REST APIs are REST-in-Peace APIs. Long Live GraphQL; 24 July 2017. URL: https://medium.freecodecamp.org/rest-apis-are-rest-in-peace-apis-long-live-graphql-d412e559d8e4. Accessed 30 March 2019.

5 Mark J. How GraphQL replaces Redux; 9 March 2018. URL: https://hackernoon.com/how-graphql-replaces-redux-3fff8289221d. Accessed 30 March 2019.

6 Jani T. Versioning an API in GraphQL vs. REST. URL: https://symfony.fi/entry/versioning-an-api-in-graphql-vs-rest. Accessed 7 April 2019.

7 Phil S. GraphQL vs REST: Overview; 24 January 2017. URL: https://philsturgeon.uk/api/2017/01/24/graphql-vs-rest-overview/. Accessed 30 March 2019.

8 Esteban H. 5 reasons you shouldn't be using GraphQL; 20 Sep 2018. URL: https://blog.logrocket.com/5-reasons-you-shouldnt-be-using-graphql-61c7846e7ed3. Accessed 31 March 2019.

9 Get started. URL: https://www.apollographql.com/docs/react/essentials/get-started. Accessed 10 April 2019.

10 Queries. URL: https://www.apollographql.com/docs/react/essentials/queries. Accessed 10 April 2019.

11 Mutations. URL: https://www.apollographql.com/docs/react/essentials/mutations. Accessed 10 April 2019.