Nguyen Tran Quang Khoi

Migrate to GraphQL: Rethinking application development

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor’s Thesis

10 May 2022

Abstract

Author: Nguyen Tran Quang Khoi
Title: Migrate to GraphQL: Rethinking application development
Number of Pages: 37 pages
Date: 10 May 2022

Degree: Bachelor of Engineering
Degree Programme: Information Technology
Professional Major: Mobile Solutions
Supervisors: Ilkka Kylmäniemi, Project Manager

The purpose of this thesis was to explore GraphQL as an option for API design and software development. Another goal for this study was to find out whether GraphQL can fully replace REST which is a flexible and scalable design architecture for API that enables communication with data resources through stateless methods such as GET, POST and DELETE.

As a result of this project, a comparison between two development approaches using REST and GraphQL was carried out. Respectively, two API servers were built to tackle the same set of use cases of a task-organizer project. Based on the comparison results and observations during development process, it was suggested that GraphQL could be the better option in many cases, but it was not perfect and should not be able to replace REST in any time soon. Developers and product managers need to take the specific requirements of their product into considerations to decide which architecture is the most suitable.

Keywords: GraphQL, REST API, software development, Javascript

Contents

List of Abbreviations

1 Introduction 1

2 Theoretical background 3

2.1 Client-server system 3
2.2 Javascript 3
2.3 NodeJS 4
2.4 ExpressJS 4
2.5 JSON, object, and array 4
2.6 REST 5
2.7 GraphQL 6
2.8 Postman 7
2.9 RFC 8

3 The limitations of REST and how GraphQL resolves them 9

3.1 The limitations of REST 9
3.2 How GraphQL resolves the limitations of REST 11

4 Comparison in implementation and usability of REST and GraphQL 15

4.1 Context for comparison and data service 15
4.2 The REST approach 17
4.3 The GraphQL approach 23
4.4 Comparison results 32

5 Conclusion 36

References 37

List of Abbreviations

API:	Application Programming Interface
REST:	Representational State Transfer
QL:	Query Language
SOAP:	Simple Object Access Protocol
HTTP:	Hypertext Transfer Protocol
URI:	Uniform Resource Identifier
IETF:	Internet Engineering Task Force
RFC:	Request for Comments
JS:	Javascript

1 Introduction

Nowadays the Internet has been such an essential part of human society that it is difficult to imagine what the modern world would look like without it. What started as an idea to create a connection between several universities, has now become a global network connecting billions of electronic devices from anywhere on Earth. Communication methods such as emails, instant messages or video calls are more effective every day. In agriculture, the real-time information about weather, soil, water, and air qualities are helping farmers make better plans for their crops. Banks and organizations are taking advantages from the fast and secured transactions, as well as the convenient and trustworthy electronic signings. The media industry is also blooming in the recent decades thanks to the Internet. Social media platform such as TikTok, YouTube, Twitter, or Instagram are now the new forms of entertainment that are familiar to people around the world.

The above examples are only a few among countless services that the Internet provides. These services are served over the World Wide Web on multiple platforms such as websites or mobile applications. The huge data transmissions between the components of a web system demand a flexible architecture. In the late nineteenth and early twentieth centuries, SOAP and REST were the two known API design schemes, then REST quickly claimed the popularity thanks to its low bandwidth usage and flexibilities in methods as well as returned data. These features allowed developers to modularize the application logics, manipulate multiple forms of resources with less effort. REST API has also proved to be adaptable to cloud platforms: all the big names in cloud technology such as Google, Amazon, Microsoft are leveraging REST API in their own web services. It was not too long before REST became the main approach to web development.

The amount of data a human can interact with is exponentially increasing every year. The rapid growth of social media platforms has established a need for

speed, in both software development and user experience. GraphQL was introduced to help address those issue. GraphQL applications are praised for cost effective and fast performance, along with its developer friendliness. Companies such as GitHub, Twitter, Starbuck, PayPal, or Airbnb have already started to adopt GraphQL in their products [1].

The thesis was carried out with the aim of understanding what is the optimal solution for API design in applications based on the characteristics of REST and GraphQL. Throughout the discussion, a web application was developed in order to demonstrate the difference between the two approaches and provide practical insights into working with REST and GraphQL. The application was inspired by Trello – a popular project management tool – with only a few basic functions to represent the most common API methods. The idea was to make an initial version with REST APIs, then another version with GraphQL and compare the two versions to find the pros and cons. Finally, a conclusion was made about API designing process including the improvements, restraints as well as considerations for software product stakeholders.

2 Theoretical background

2.1 Client-server system

Every second, innumerable amount of data is being transferred via Internet to serve people with their purposes: from business contracts, banking transactions, shopping receipts to events streaming, or online video games, etc. When a device, such as a computer, a mobile phone, or a fax machine, sends or receives data on network, that device is considered an entity in a client-server system. This is defined as a software architecture, in which the client requests and displays information on a user interface, while the server handles the data from its sources (database) to produce such information [2]. Figure 1 illustrates a simple example of a software architecture.

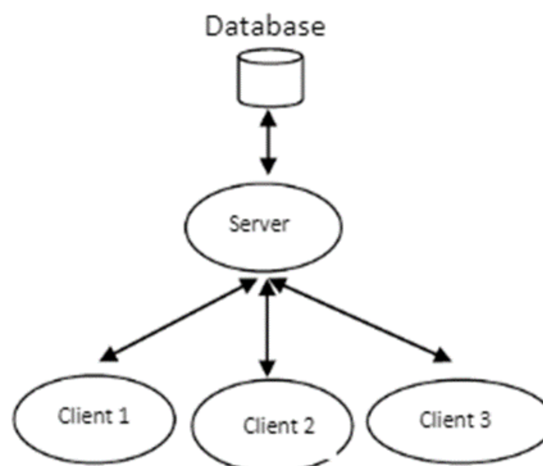

Figure 1 The communication process between entities in a client-server system [2, p. 1]

In practice, there are usually multiple clients communicating with a server, that queries data from the storage of the server itself or from a remote database.

2.2 Javascript

Javascript, or JS as abbreviation, is one of the most popular programming languages in the world. It has been mostly used for web browsers, and recently also been adopted by server and embedded environments. The features and

characteristics of Javascript include prototype-based, multi-paradigm, single-threaded and dynamic. It supports object-oriented, imperative, and declarative programming styles [3].

2.3 NodeJS

NodeJS is a server, or back-end, environment that is open-source, built on V8 Javascript engine and compatible with various platforms including MacOS, Windows, and Linux.

As it executes Javascript on the server, NodeJS brings the benefit of reusing development resources from front to back optimally, as well as lowering learning curve for developers, especially when the web development with Javascript has been spiking over the last decades. Some applications of NodeJS includes webapp, real time interactions, API server and IoT services. Every day, NodeJS powers the operations of millions of companies around the world such as eBay, Spotify, LinkedIn, Twitter, etc.

2.4 ExpressJS

ExpressJS is a framework for web server development in NodeJS. It provides a wide range of utilities and middleware for HTTP service, allowing for easy creation and maintenance of APIs. ExpressJS is trusted and the selection of many companies such as IBM, Uber, Yandex, etc.

2.5 JSON, object, and array

JSON stands for Javascript Object Notation and is a lightweight data-interchange format [4, p. 1]. It stores and transmit data with human-readable text based on 2 main structures: object and array.

Object is a collection of key/value pairs [4, p. 3]. In Javascript, it is a data type to store a variety of collections and entities. Listing 1 below is an example of an object initializer in Javascript:

```
const example_object = { name: 'John', age: 30, cars: ['Toyota', 'Audi']};
```

Listing 1 Initialize an object in Javascript

Array is an ordered list of values. In Javascript, it enables storing a resizable collection of multiple items, which can be of different data types, under a single variable name. [5]. Listing 2 below showcases one of many ways to create an array in Javascript:

```
const example_array = [99, 'Javascript', {a: 'foo', b: 'bar'}, null];
```

Listing 2 Create an array in Javascript

2.6 REST

In 2000, Fielding introduced REST in his dissertation as a description of the Web's architectural style [6, p. 5]. Since then, a well-designed RESTful API has been one of the key features for a web service to compete with its rival [7, p. 3] and to maintain a high but stable performance. The term REST stands for "REpresentational State Transfer", which means that the client can ask for a "representation" (i.e., a description or information) of the "state" of a resource (i.e., resource data) from the server [7, p. 3]. For example, a student can use a laptop (the client) to find information about a book ("representational state" of that book) from a library's database (the server).

To enable the communication between client and server, a set of application programming interfaces (API) is designed, consisting of methods such as GET, POST, PUT, and DELETE. Each method is a HTTP request to predefined endpoint (URI) on the server. Thanks to these methods, a client can inform that it needs to display or update a resource, and the server can either return the requested data or execute the modification accordingly. Relying on HTTP

resources (such as query parameters, body payload, etc), the predefined endpoints are considered the low-level abstractions [8, p. 1] of the resource.

Listing 3 is an example of a REST API endpoint from GoRest (Online REST API service for Testing and Prototyping):

```
GET /public/v1/users
```

Listing 3 API endpoint to retrieve a list of users using GET method

This endpoint, when requested using HTTP method GET, returns data about users from a collection of sample data created by GoRest.

2.7 GraphQL

GraphQL is a query language, which means it is a language for querying resources from servers and does not address a specific programming language or system. GraphQL was internally developed and used by Facebook since 2012 until 2015 when the company decided to make it open-source [9, p. 2]. Thanks to its innovative philosophy and versatile syntax, GraphQL is growing in reputation as a new API design standard for client-server systems, competing with the traditional REST. Airbnb, Star-buck, GitHub, The New York Times and Twitter [1] are among many organizations and companies who have been using GraphQL effectively in their daily production.

GraphQL provides high-level abstractions to data resources via types, schemas, queries, mutations, and subscriptions. In its simplest form, GraphQL asks the server for specific fields in data objects by running a query like the example in Listing 4:

```
query GetUsers {
	users {
		first_name
		last_name
	}
}
```

Listing 4 GraphQL query to get first name and last name of all users

The above example is a “read” operation, which can be recognized by the keyword `query` as operation type and `GetUsers` is the operation name. For “write” actions, a `mutation` query will be used instead with parameters as in Listing 5:

```
mutation CreateNewUserMutation($name: String!) {
      createNewUser(name: $name) {
            name
            role
            created_date
      }
}
```

Listing 5 GraphQL mutation to create a new user with name

GraphQL server understands these queries and mutations thanks to the schemas and type system, which describe what data can be retrieved, which function to create/update data and how all of them look like in terms of data types.

2.8 Postman

Postman is an interactive tool for API testing. It provides an HTTP client that supports various types of requests, frameworks, and data payload. Developers use Postman to constructs requests to their APIs, and make sure the responses are what they expected. During the course of this document, Postman will be used as the primary tool for testing APIs in both REST and GraphQL approach. An example of user interface of the Postman application can be seen in Figure 2 below.

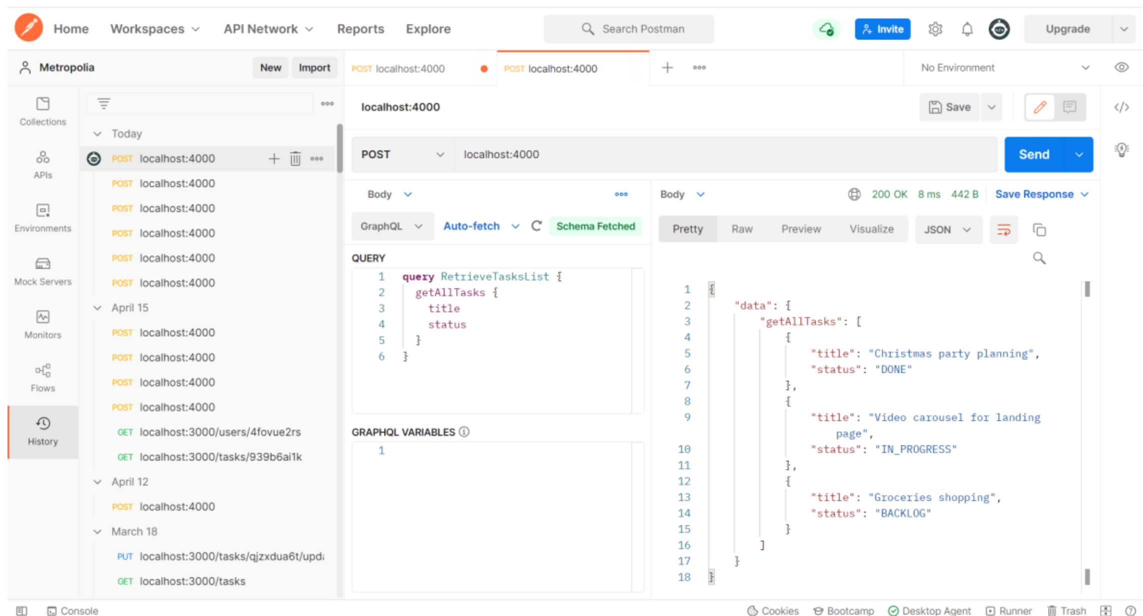

Figure 2 User interface of Postman application

2.9 RFC

Produced and stewarded by the IETF, RFC is a series of documents describing the Internet's technical foundations, such as addressing, routing, and transport technologies [10]. The first RFC document was published in 1969, specifying how packets are sent between computers as parts of a network. Nowadays there are more than 9000 RFCs, and they are considered the building blocks of computer networks and the Internet.

3 The limitations of REST and how GraphQL resolves them

3.1 The limitations of REST

The main idea of designing API with REST is treating everything as a resource, which is addressed by a Uniform Resource Identifier (URI). Interactions with a resource are made possible by the built-in HTTP methods like GET, POST, PUT and DELETE. For such a long time, this approach has been effective and flexible that it is arguably considered a standard when developing API for web services. But the world has been changing since then, and the data is growing with it, which also introduces new problems.

Consumption by round trips

As mentioned above, in REST architecture, each dataset or resource is referenced by an endpoint. For every action regarding the resource, such as retrieving or modifying the text content of a blog post, the client has to send a network request to the corresponding API endpoint. When the client receives the response from the server for that request, a round trip is completed between the client and server. In modern applications like a social media platform, the resources are usually relational in a high complexity. In other words, the data required by the client often comes from multiple different resources, which may lead to an excessive number of round trips on the network. As the system grows bigger, it be-comes more vulnerable to a degradation in performance, especially in situations like slow internet or obsolete devices.

Over-fetching and under-fetching

For all the requests to a REST API endpoint, the amount of data in the response will always be the same. This results in two contrasting problems: over-fetching and under-fetching. When the fetched data from server is more than required, it is called over-fetching. In the opposite, under-fetching is when not enough data is fetched when client requests it from a resource.

In the previous example with GoRest API, the server returns all the fields related to a user, such as name, email, and gender, as shown in Figure 3. If the client only needs user's name, all the other fields become redundant. Over-fetching causes wasted data and consumes the network bandwidth as well as the server's computation time to deliver the response.

```
{
    id: 12,
    name: "Mr. Abhaya Panicker",
    email: "abhaya_panicker_mr@leffler.net",
    gender: "male",
    status: "active",
},
```

Figure 3 A fragment of the JSON response from https://gorest.co.in/public/v1/users, representing data of a single user.

On the other hand, if the client wants to read all the posts the user has created, along with all the comments in each of those posts, it becomes a case of under-fetching: a single request to the "user" can not provide sufficient information about the posts and the comments as they belong to different resources. In other words, the client must make another request to retrieve the list of posts by a user, then make as many requests to "comments" endpoint as the number of those posts. This issue largely contributes to the problem of round trips consumption above.

Slow iteration on the front-end

One common way of building API with REST is organizing endpoints according to the views in the application. The client can retrieve all the needed data for a specific view by sending a single request to the corresponding API endpoint. However, when a change is made to that view, it is likely to require more or less data than before, causing the backend (server) to change as well to meet the

new data requirements. This slows down the iteration rate of a product as more effort is needed for those changes. [11]

Little support for type-safe development

Poor typing is a known characteristic of REST API. Although there are solutions made by the community to enable typing pattern in RESTful applications, a dedicated documentation on database schema and API endpoints is still needed in most cases. Then even with a well-documented API system, the integration between the views and the APIs is still a pain point for productivity, due to the time consumption for testing, debugging and communication between front-end and back-end developers.

3.2 How GraphQL resolves the limitations of REST

Fewer round trips

Instead of making multiple requests to different endpoints, GraphQL collects and returns only the pieces of data that are specifically requested and presented in a standard format. This approach results in a high efficiency in data delivery since fewer round trips are needed in order for the client to receive the whole information.

Additionally, the simplicity of having only one common endpoint removes the risk of having multiple ad hoc endpoints and mitigates the Internet bandwidth for applications. Thanks to the graph-based schema, data can be considered as a package when being delivered by a GraphQL client. For example, Listing 6 is a GraphQL query for retrieving information of a user including their posts and comments in those posts:

```
query GetUserById($user_id: ID!){
	getUserById(id: $user_id) {
		first_name
		last_name
		posts {
			title
			post_content
			comments {
				author {
					first_name
				}
				Comment_content
			}
		}
	}
}
```

Listing 6 GraphQL query to get user data by user ID

For traditional client-server systems using REST, the whole dataset may take at least 5 API calls to different endpoints to gather the pieces: details of category "finance", list of books by category, list of authors for each book, list of references for each book and list of authors for each of those references. GraphQL only takes one request to send the query above so that the server knows exactly what is required and retrieve them all in a single package. As the number of requests is mitigated, the efficiency is improved and the relationship between client and server is restructured. The server specifies the common ground rules with data schema and the client makes queries base on that schema, therefore is well aware of the data format in advance.

The idea can be easier to grasp by illustrating a real-life situation, where a courier has to deliver to food box to a customer. If there are a dozen of routes leading to the house of that customer, each of which may have different conditions such as speed limits, traffic tolls, restricted vehicle types. These constraints can definitely affect the delivery time, the significance of the food (data) as well as customer satisfaction as the courier reaches the destination. It is obvious to see the efficiency of having a single route to the customer house, where all the traffic rules, street conditions and such are known beforehand. These are the benefits that GraphQL will help to achieve.

API stability

With great simplicity comes great stability – there are fewer risks of defects in the implementation iterations of API. Having a strong type system to support the clarity of the simple data queries, GraphQL allows for a more stable API process, from planning to implementation and testing.

This is accomplished by the one fundamental methodology of GraphQL: data is delivered in a pre-defined structure, which is agnostic of client types as the queries format and rules are controlled by GraphQL itself. The backend can freely manipulate the data and its internal implementation, without requiring according changes at the client. This has been impossible to do in traditional REST systems, as the client-server relationship is tightly coupled. Any changes to the database and API endpoints without concerning the data format and request methods from the client will result in an application breakdown.

From another point of view, GraphQL serves as a specification for API. It can be defined and implemented by any programming languages, as long as they follow GraphQL's methodology for data delivery. More importantly, GraphQL can integrate with ready-made REST systems by providing a wrapper or layer to abstract the multiple API endpoints into one. This means that GraphQL is flexible and easy to be adopted by developers with a low overhead of refactoring and integration.

Quick development iteration and organized type system

GraphQL provides the ability to create queries and mutations with high accuracy thanks to a complete type system. By defining a set of data types, also known as schema, for a GraphQL service, developers have a detailed documentation on what kind of data is available, what fields are included, and can always predict the result of the queries with-out much awareness of the database. The schema and type system also enables GraphQL to be easily

integrated with other typing languages such as Typescript and Swift, ensuring a type-safe environment.

With the schema as a single source of truth for data models, any changes or new requirements can be well accounted for, just by modifying the schema file. This is a clear advantage of GraphQL, which allows for rapid iteration in product development.

4 Comparison in implementation and usability of REST and GraphQL

For a practical comparison between REST API and GraphQL, an experiment is needed to illustrate how the two approaches would handle the same 4 use cases:

1. The client needs to show an overview a project board to the customer. In this board, there is a list of tasks with title and status.

2. When customer clicks on a task to see the details, the client displays title, description, status, and name of the assignee.

3. After a new task is created with minimal required information, the client shows the details of that new task.

4. Customer updates the new task by assigning a user or change the task description and status.

4.1 Context for comparison and data service

For the sake of clarity about the use cases, it is necessary to define some basic contexts and business rules. This also helps to construct a common and shared data service for the APIs implemented in REST and GraphQL.

Both approaches will be conducted in the same programming language - Javascript. The REST API server will be implemented with NodeJS and ExpressJS. The GraphQL server will be implemented with Apollo Server. Other dependencies and the installation steps are not mentioned in this experiment.

In all 4 use cases, the same project board is given, which means there is no need for a reference to this board throughout the implementation. Other than that, all the resource objects must have an “id” field for identification.

Aside from title and description, which are free texts, every task has a status and an assignee that come with specific typings. Status of a task can either be “Backlog”, “In progress”, “In review” or “Done”. An assignee is a user, who has email and name.

Title is required and description is optional for creating a new task. By default, task description, if not provided, is empty, the status will be “Backlog” and no user is assigned to it yet.

Database will not be a focus in the experiment as REST and GraphQL are basically database agnostic, they only concern with the “how” of API – the way of communication between the server and the external entities: web client, applications, or electronic devices, etc.

To serve as a single, and simple, data source for both approaches, a file named “data.js” will be used, which contains an array of objects for tasks, and an array of objects for users. These are the initial data for the two main resources. Then throughout the experiment, both servers will need to use the same service to interact with the data source, via read/write methods. These methods are implemented according to the 4 use cases, which includes retrieving all records, find a task or user by ID, creating a new data object or updating an existing one. A utility function “generateID” will help to generate identifiers for the new tasks and users created later. It can be found in Listing 7 below as an example when creating a new task with Task Service.

```
import { tasks } from "../data.js";
import { generateID } from "../utils.js";

export default class TaskService {
    constructor() {
        this.tasks = [...tasks];
    }

    getAllTasks() {
        return this.tasks;
    }

    findById(taskId) {
        return this.tasks.find((task) => task.id === taskId);
    }

    createTask(title, description) {
        const newTask = {
            id: generateID(),
            title,
            description: description ?? "",
            status: "Backlog",
            assignees: [],
        };
        this.tasks.push(newTask);
        return newTask;
    }

    updateTask(id, title, description, status, assigneesID) {
        let foundTask;
        const newAllTasks = this.tasks.map((task) => {
            if (task.id === id) {
                foundTask = {
                    id: task.id,
                    title: title || task.title,
                    description: description || task.description,
                    status: status || task.status,
                    assignees: assigneesID || task.assignees,
                };
                return foundTask;
            }
            return task;
        });
        if (foundTask) {
            this.tasks = newAllTasks;
        }
        return foundTask;
    }
}
```

Listing 7 Task service

4.2 The REST approach

As explained in section 2.6 (Theoretical background – REST), every different interaction with the resource requires an endpoint corresponding with a HTTP

method. It is a common practice in the industry to name the endpoints based on the type of resource they interact with, which, in this case, is the “tasks”.

Retrieve the tasks list

The API has only one simple job: to return the latest list of all tasks. With REST, it is defined by an URI “/tasks” and method GET. It is a standard practice that every forward slash character (/) indicates a hierarchical relationship between resources. In this case, it is the first level of interaction with resource “tasks”. These implementation details are reflected in Listing 8 below.

```
app.get("/tasks", (_, res) => {
    res.json(taskService.getAllTasks());
});
```

Listing 8 Implementation of GET ”/tasks” with Task Service

When the client sends a HTTP request with method GET to endpoint “/tasks”, the server responds with an array of all tasks in the JSON format as in Figure 4. Each item in the array contains everything about a task. However, the client does not need all these data, as in this use case the customer will only see the task title and status on the project board. This is the problem of over-fetching with REST.

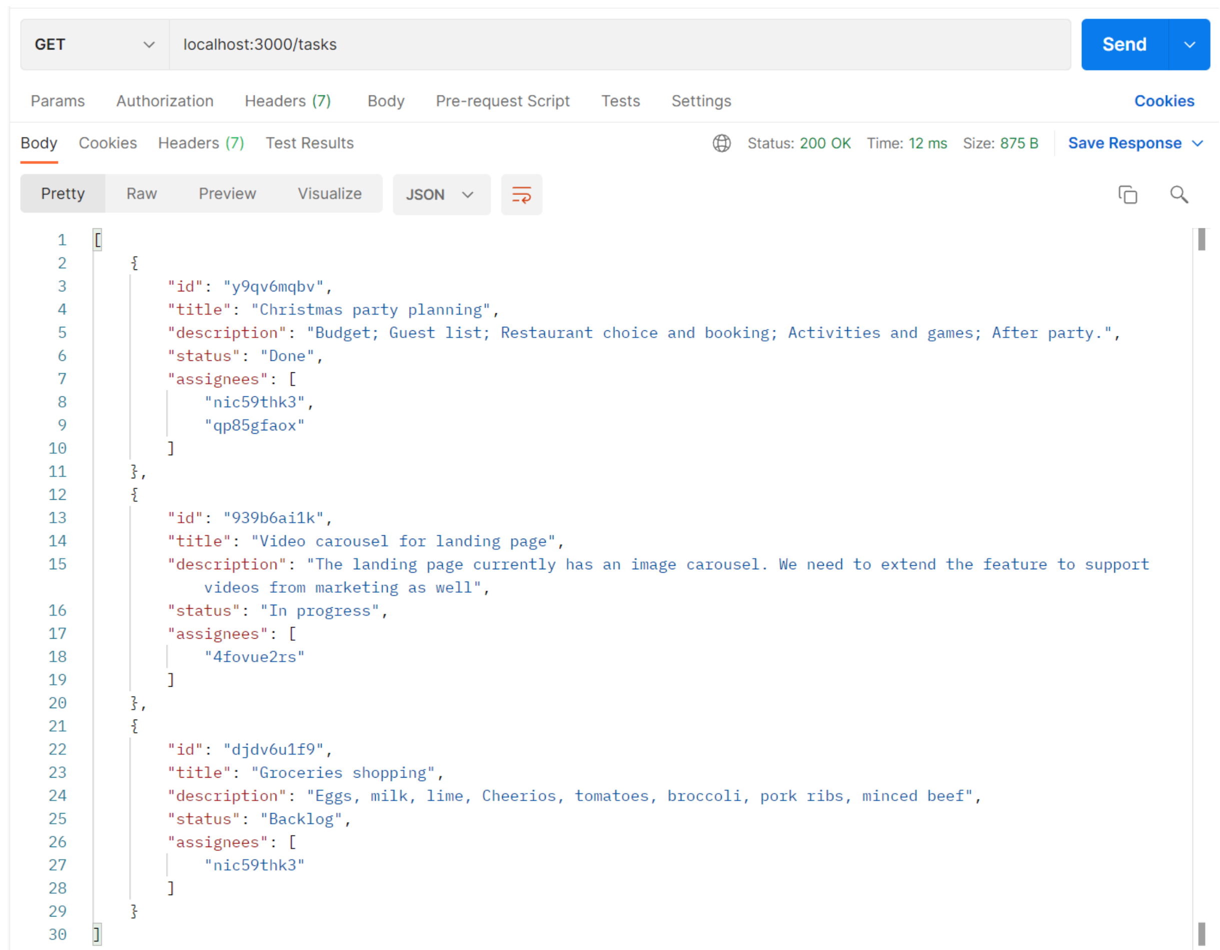

Figure 4 Retrieve the list of all tasks with GET /tasks

View task details

This use case, shown in Listing 9, demonstrates a deeper level in the “tasks” resource hierarchy. Every task is a subset of the list of all tasks. Therefore, the URI reflects this relationship with a second forward slash and a task ID as parameter: `/tasks/:taskId`. The parameter is a feature supported by ExpressJS that allows developers to define APIs pointing to a more specific node under the resource.

```
app.get("/tasks/:taskId", (req, res) => {
    res.json(taskService.findById(req.params.taskId));
});
```

Listing 9 Implementation of GET ”/tasks/:taskId”

The result that the client receives from the API, as shown in Figure 5, is a task object containing most of the data it needs for the customer, which are title, description and status. The assignees data, on the other hand, is an array of only ID.

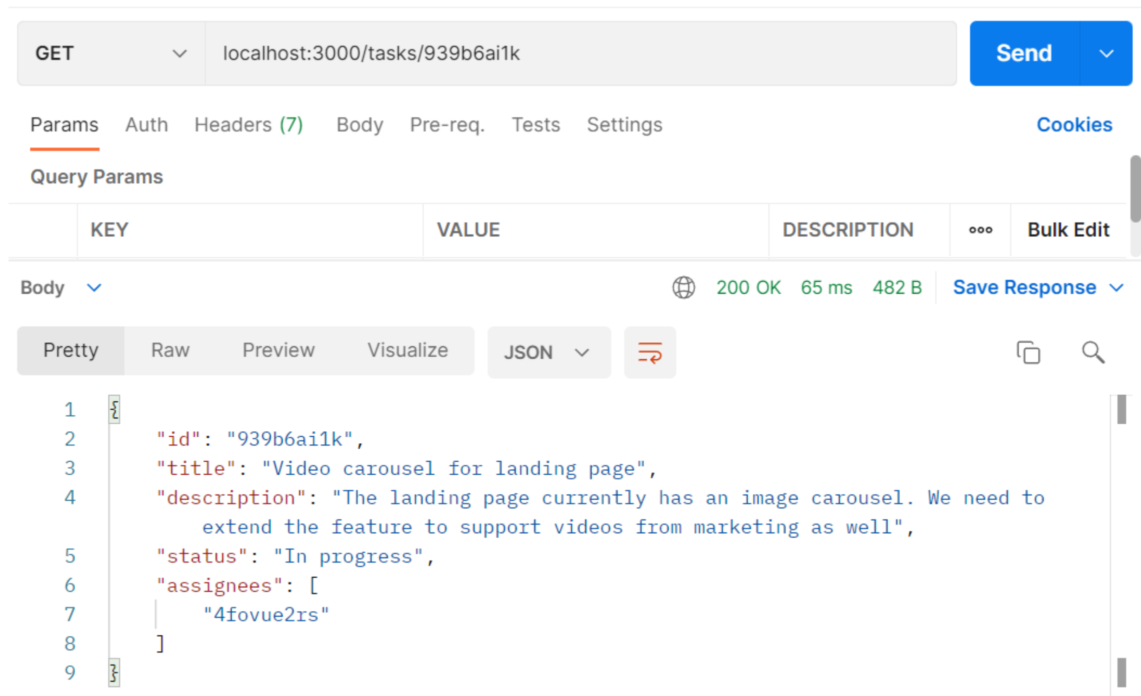

Figure 5 Retrieve data of a specific task with GET /tasks/:taskId

This is where the under-fetching problem comes up. Because the client now must make another request to retrieve data for each assignee. If the task has 5 assignees, for example, then the client will have to send 5 more GET /users/:userId requests to the server as in Figure 6, making it 6 requests in total only to show details of a task. After that, the client still has more work to do: mapping data of those 5 users to the list of assignees under the task data it received from the first request.

GET localhost:3000/users/4fovue2rs Send

Params Auth Headers (7) Body Pre-req. Tests Settings Cookies

Query Params

KEY	VALUE	DESCRIPTION	ooo	Bulk Edit

Body 200 OK 5 ms 296 B Save Response

Pretty Raw Preview Visualize JSON


```
1 {
2     "id": "4fovue2rs",
3     "email": "user1@test.com",
4     "name": "John Doe"
5 }
```


Figure 6 Getting assignee data with GET /users/:userId

Create a new task

As specified in Section 5.1, a title is required to create a new task, and the description is optional. When making a POST request to the server, the client sends along a payload containing data that the API needs.

```
app.post("/tasks/create", (req, res) => {
    const { title, description } = req.body;

    // title is the minimum requirement to create a task
    if (!title) {
        res.status(400).json({
            error: "Invalid request",
        });
        return;
    }

    res.json(taskService.createTask(title, description));
});
```

Listing 10 Implementation of POST "/tasks/create"

According to the implementation shown in Listing 10, if the title is missing from the payload, the API will respond with "Invalid request". Otherwise, a new task is created with a generated ID, the provided title and description, the default status as "Backlog" and an empty list of assignees as in Figure 7.

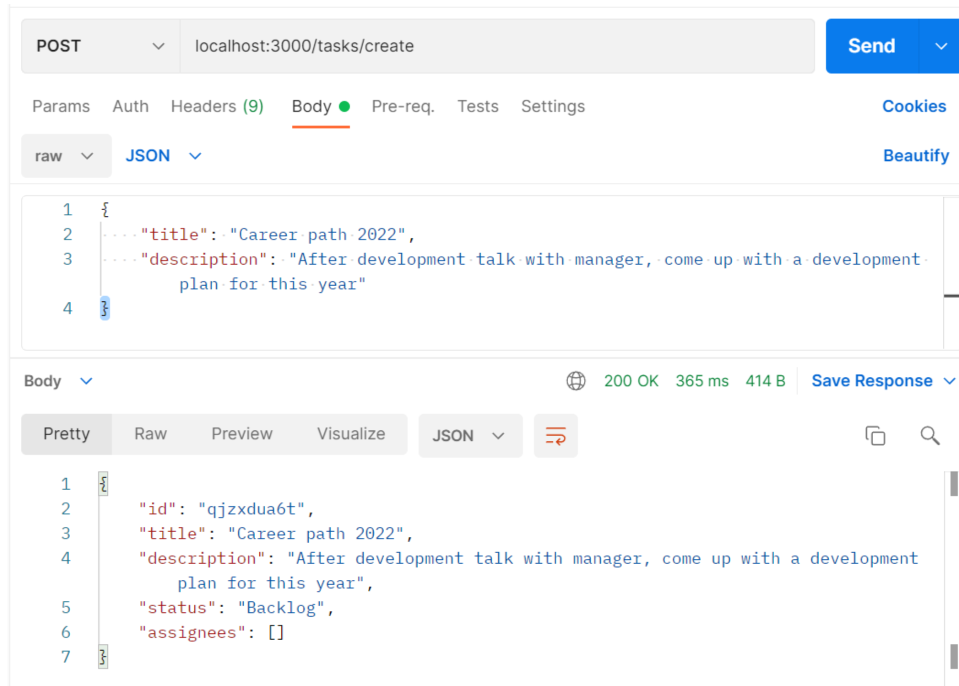

Figure 7 Create a new task with POST /tasks/create

Update a task

This API extends further the URI used for retrieving task details with an action “update” and uses method PUT to modify the data of a specific task. The whole implementation of this API is shown in Listing 11.

```
app.put("/tasks/:taskId/update", (req, res) => {
    const { taskId } = req.params;
    const { title, description, status, assigneesID } = req.body;

    // no task ID or no data provided to update the task
    if (!taskId || (!title && !description && !status && !assigneesID)) {
        res.status(400).json({
            error: "Invalid request",
        });
        return;
    }

    const updatedTask = taskService.updateTask(taskId, title, description,
status, assigneesID);
    if (!updatedTask) {
        res.status(400).json({
            error: "Task not found",
        });
        return;
    }
    res.json(updatedTask);
});
```

Listing 11 Implementation of PUT ”/tasks/:taskID/update”

Using the task ID from the query parameter, the API can now find the exact task object among the list of tasks, and then update it with the provided payload. At least one of the task's information is needed, otherwise an error will be responded. For example in Figure 8, to update the status of the task to "In progress" and keep everything else the same, the client needs to send only the desired status in the payload.

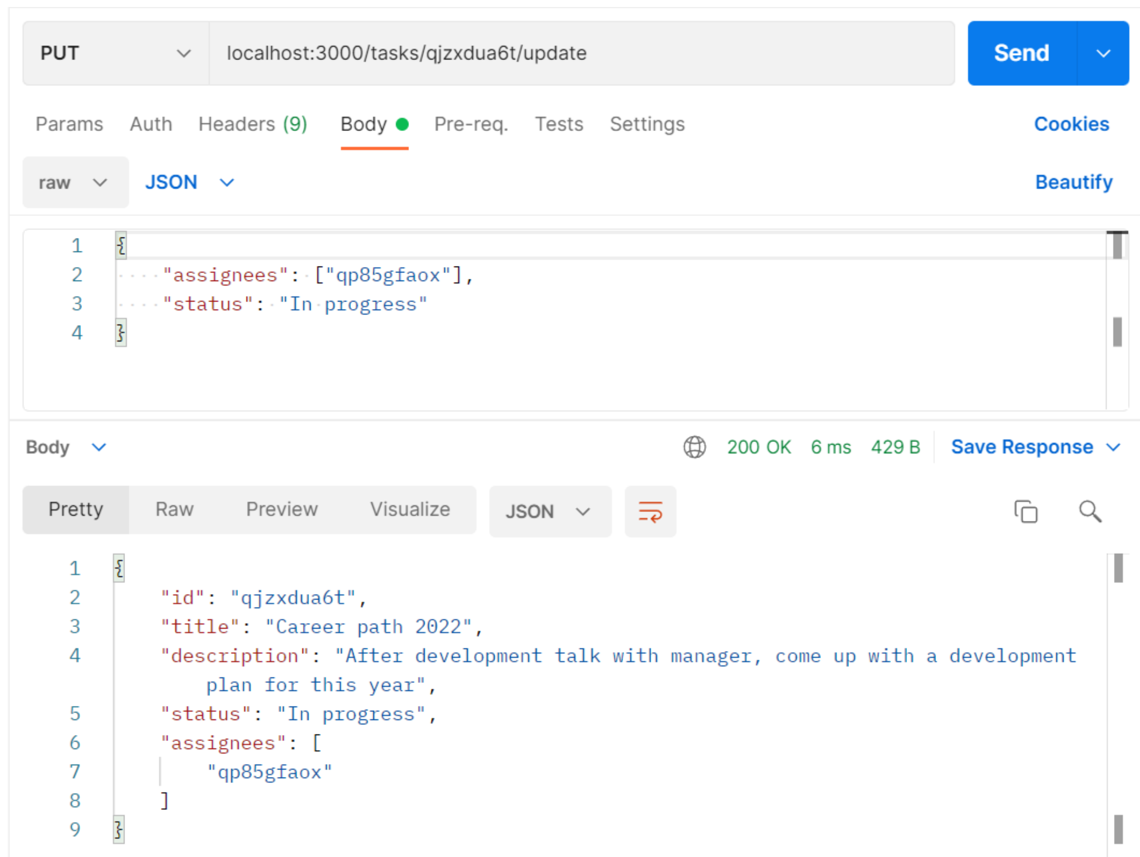

Figure 8 Update task details with PUT /tasks/:taskId/update

4.3 The GraphQL approach

Defining schemas

Every GraphQL server needs a schema, based on which the client can query necessary information from data source. The schema contains definitions of data types and the fields on those types. In this experiment, there are two main

data types: User and Task, corresponding to the two main resources in the sample data. The User schema is declared as in Listing 12 below.

```
import { gql } from "apollo-server";

export default gql`
  type User {
    id: ID!
    email: String!
    name: String!
  }

  type Query {
    getAllUsers: [User]
    getUserById(id: ID!): User
  }

  type Mutation {
    addUser(email: String!, name: String!): User
  }
`;
```

Listing 12 User schema

In REST, requests with any method might cause side-effects on the server, but by convention GET requests are used for data fetching, POST requests for creating new data, PUT requests for modification, and so on. Similarly with GraphQL, although a query could be implemented to create or update a record in data source, it is a convention that such data writing operations must be sent via mutations, as seen in both Listing 12 and Listing 13. Furthermore, a mutation can return an object type, which means the client can also ask for specific nested fields just like it does with query. This is demonstrated later with Figure 12 and Figure 13 in their respective use cases.

```
import { gql } from "apollo-server";

export default gql`
  type Task {
    id: ID!
    title: String!
    description: String
    status: TaskStatus
    assignees: [User]!
  }

  enum TaskStatus {
    Backlog
    In progress
    Done
  }

  type Query {
    getAllTasks: [Task]
    getTaskById(id: ID!): Task
  }

  type Mutation {
    createTask(title: String!, description: String): Task
    updateTask(id: ID!, title: String, description: String, status:
TaskStatus, assigneesID: [ID]): Task
  }
`;
```

Listing 13 Task schema

Although it is possible and acceptable to define schemas for the whole type system in a single place, it is a common practice to have a separate schema file for each resource, then combine them when providing to Apollo Server. Figure 9 describes how the array of user schema and task schema is constructed.

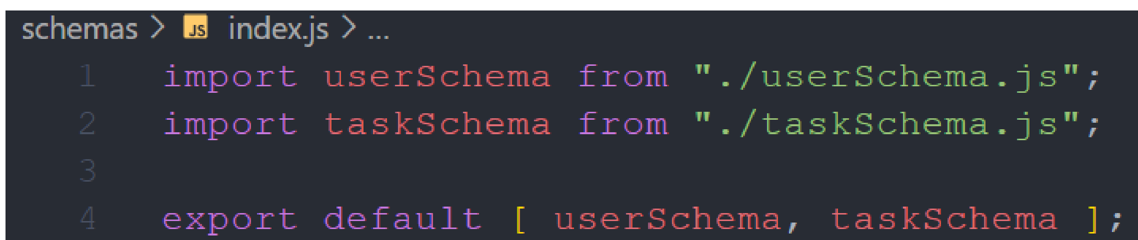

Figure 9 Combine all schemas into one array to later provide for GraphQL server

Defining resolvers

When receiving a request for a specific field in data (for example, name of a user), GraphQL server can now find that field in the schema, but it also needs to populate the data for it. This is where the resolvers step in. Each field on each

type must have a resolver function to produce the corresponding value for that field. For the sake of demonstration, all schema fields under type User and Task are simple enough that there is no need to explicitly define resolvers for them. Apollo Server, under the hood, will create default resolvers for them.

On the other hand, all schema fields under Query and Mutation types should be defined by developers. These resolvers can either use a database, services or an external API to accomplish their tasks. Listing 14 shows how the User Service is used to return corresponding data for User resolvers.

```
export default {
    Query: {
        getAllUsers: (_, _args, context) => context.userService.getAllUsers(),
        getUserById: (_, args, context) =>
context.userService.findById(args.id),
    },
    Mutation: {
        addUser: (_, args, context) => {
            const { email, name } = args;
            return context.userService.addUser(email, name);
        },
    },
};
```

Listing 14 Resolvers for every field under Query and Mutation types related to User

It is noticeable that the implementation of these GraphQL resolvers is similar to that of the API endpoints in REST approach. The main difference will be how these functions are executed when the client sends a request to the server. Listing 15 showcases the whole implementation of Task resolvers as a reference to compare with the REST approach.

```
import { UserInputError } from "apollo-server";
import { tasks } from "../data.js";
import { generateID } from "../utils.js";

const allTasks = [ ...tasks ];

export default {
    Query: {
        getAllTasks: () => allTasks,
        getTaskById: (_, args) => allTasks.find(task => task.id === args.id)
    },
    Mutation: {
        createTask: (_, args) => {
            const { title, description } = args;
            if (!title) {
                throw new UserInputError('Invalid request');
            }
            const newTask = {
                id: generateID(),
                title,
                description: description ?? '',
                status: 'Backlog',
                assignees: []
            };
            allTasks.push(newTask);
            return newTask;
        },
        updateTask: (_, args) => {
            const { id, title, description, status, assigneesID } = args;
            // no task ID or no data provided to update the task
            if (!id || (!title && !description && !status && !assignees)) {
                throw new UserInputError('Invalid request');
            }
            let foundTask;
            const newAllTasks = allTasks.map((task) => {
                if (task.id === id) {
                    foundTask = {
                        id: task.id,
                        title: title || task.title,
                        description: description || task.description,
                        status: status || task.status,
                        assignees: assigneesID || task.assignees
                    };
                    return foundTask;
                }
                return task;
            });
            if (!foundTask) {
                throw new UserInputError('Task not found');
            }
            allTasks = newAllTasks;
            return foundTask;
        }
    }
};
```

Listing 15 Resolvers for every field under Query and Mutation types related to Task

Retrieve the tasks list

Just like with schemas, the resolvers for User and Task are defined in separate files and then combined into an array when providing for Apollo Server. As

shown in Listing 16, with both schemas and resolvers in place, together with the data services as the shared context across the resolvers, the GraphQL server is basically ready to accept requests from client.

```
import { ApolloServer } from "apollo-server";
import schemas from "./schemas/index.js";
import resolvers from "./resolvers/index.js";
import TaskService from "./services/taskService.js";
import UserService from "./services/userService.js";

try {
    const server = new ApolloServer({
        typeDefs: schemas,
        resolvers,
        context: {
            userService: new UserService(),
            taskService: new TaskService(),
        },
    });
    server.listen().then(({ url }) => {
        console.log(`Server ready at ${url}`);
    });
} catch (error) {
    console.log(error);
}
```

Listing 16 GraphQL server configurations in index.js file

The client asks GraphQL server to return exactly what it needs via a query. In this case, the query `getAllTasks` will be used to get an array of task objects, each containing ID, title, description status and an array of assignees' IDs. These fields are specified in a structure that the client expects to receive from the server. Below in Listing 17 is how the query is constructed.

```
query RetrieveTasksList {
  getAllTasks {
    id
    title
    description
    status
    assignees {
      id
    }
  }
}
```

Listing 17 GraphQL query sent by client to retrieve the tasks list

Looking at the query sent by the client, the server knows to return the data in the name `getAllTasks`, which is an array in this case. The query contains only what the client needs: title and status. And that is exactly the response from

server (see Figure 10 below). It is noticeable that the data fields in the result also match with the naming and ordering of those in the query. By allowing the client to specify the data structure that it needs, GraphQL does not have over-fetching problem as compared to REST approach.

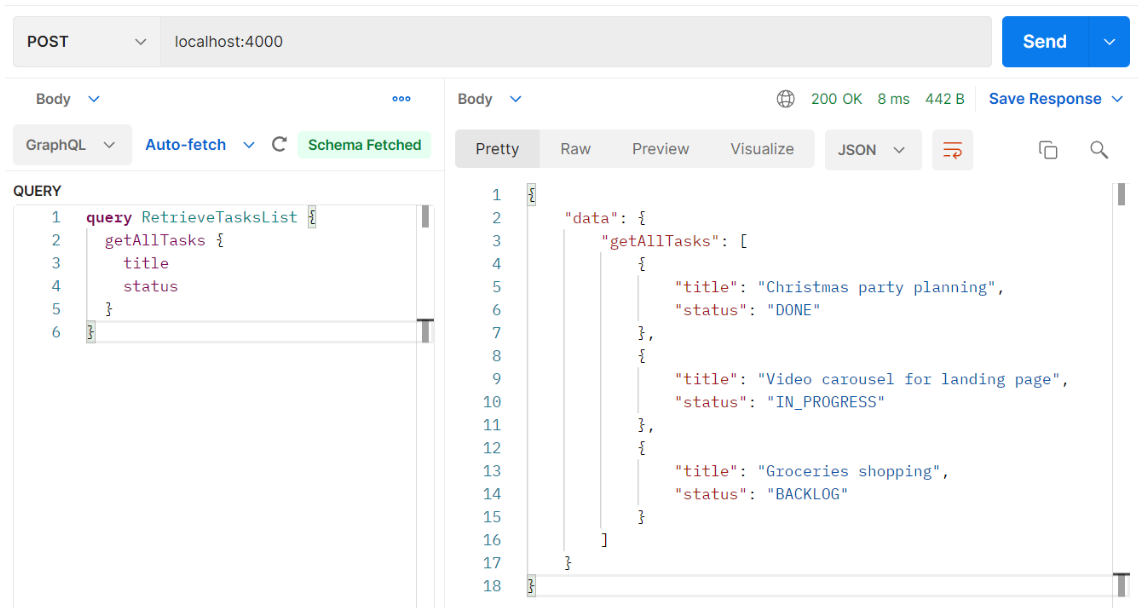

Figure 10 Retrieve the list of tasks with query “getAllTasks”

View task details

Similar to REST API, a GraphQL query or mutation accepts arguments as well. As shown in Listing 18, the `getTaskById` query takes a task ID as an argument and return all details of that task, including the assignee’s name.

```
query ViewTaskDetails($taskId: ID!) {
  getTaskById(id: $taskId) {
    title
    description
    status
    assignees {
      name
    }
  }
}
```

Listing 18 GraphQL query to get details of a task by ID

Because `getTaskById` refers to a Task object, and “assignees” field refers to User objects, GraphQL traverses these objects and their fields, returns all the related data that the client needs in one request (Figure 11 below), even if, for example, the task has multiple assignees. Therefore, under-fetching is not an issue with GraphQL either.

Figure 11 View task details with query “getTaskById”

Create a new task

GraphQL mutations work almost exactly like queries, except it is used for data writes as a convention. The `CreateTask` mutation takes “title” as a required argument, while “description” is optional. Listing 19 shows how it is constructed.

```
mutation CreateTask($title: String!, $description: String) {
  createTask(title: $title, description: $description) {
    title
    description
    status
    assignees {
      name
    }
  }
}
```

Listing 19 GraphQL mutation to create a new task

When the mutation is executed, the `createTask` field will return the same fields that client needs to display the details of the newly created task (Figure 12). Although the REST approach above was implemented to return the same data, it was for the sake of the use case, where the requirements were defined beforehand. Requests for write actions (add, update, delete) in REST are often implemented in such a way that they return only the ID of the object in action, and sometimes the status (for example: success or error). REST has such a separation of concerns that the action of retrieving object details is expected to be handled by a dedicated API endpoint. With GraphQL, it is possible to mutate and query the new data in the same request, which helps to save the network bandwidth and make a performant product.

POST localhost:4000

Body · GraphQL · Auto-fetch · Schema Fetched

QUERY

```
mutation Mutation($title: String!, $description: String) {
  createTask(title: $title, description: $description) {
    title
    description
    status
    assignees {
      name
    }
  }
}
```

GRAPHQL VARIABLES

```
{
  "title": "Send email",
  "description": "Testing123"
}
```

Body · 200 OK 9 ms 353 B · Pretty · Raw · Preview · Visualize · JSON

```
{
  "data": {
    "createTask": {
      "title": "Send email",
      "description": "Testing123",
      "status": "BACKLOG",
      "assignees": []
    }
  }
}
```

Figure 12 Create a new task with a mutation

Update a task

According to Listing 20, the mutation `UpdateTask` has task ID as a required argument to specify the task. Task data will be updated with other arguments provided for the mutation: title, description, status and assignees' IDs. If all of them are missing, an "Invalid request" error will be returned.

```
mutation UpdateTask($taskID: ID!, $title: String, $status: TaskStatus,
$assigneesId: [ID]) {
  updateTask(id: $taskID, title: $title, status: $status, assigneesID:
$assigneesId) {
    title
    description
    status
    assignees {
      name
    }
  }
}
```

Listing 20 GraphQL mutation to update details of a task

Like with the `CreateTask` mutation, after the task data is updated, server will return the new data so that the client can display the changes without making another API request (Figure 13).

Figure 13 Update task details with a mutation “UpdateTask”

4.4 Comparison results

REST has been the traditional method of API implementation and is still widely used by companies all over the world. It is even adopted in education programs in universities and colleges. Therefore, it strives to be an easier option when it comes to API design, as almost every developer is familiar with it. Although

being a relatively new name in the technologies landscape, GraphQL has quite a gentle learning curve, to some degrees, compared to REST. To build up a functional GraphQL server, developers need to define three main components: schemas, queries, and resolvers. The type system, as well as its own syntax in the schemas, can be easy to learn thanks to a well written documentation and good support from its growing community. With a good understanding about these components, it is not difficult to adopt GraphQL.

On a high level, the main distinction between REST and GraphQL is that REST is a concept of architectural constraints when designing APIs, while GraphQL itself is a query language, providing specifications for data manipulation processes. Going more into details about implementation, there are some clear differentiations found based on the experiment:

Request methods and performance

REST APIs leverage various methods such as GET, POST, PUT, DELETE to define different types of operation for multiple endpoints. GraphQL uses a single POST endpoint for all operations.

This single endpoint feature, together with the ability for the server to return exactly what the client needs, GraphQL resolves many long-standing problems of REST, giving it an advantage in terms of performance.

API specification between server and client

Server implemented in REST does not expose any information to the client side, except for the URI of the resources. Therefore, developers often need to write API specifications document alongside with the URIs or use third party automated documentation tools. With GraphQL, the introspection system enables the client to know in advance what queries, or mutations with arguments, it can request. GraphQL schema plays a critical role as an architecture specification for both server and client.

HTTP Status Codes

REST API uses HTTP Status Codes for classification of responses. For example, as specified in section 10 of RFC 1616 [12, p. 56 – p. 70], status codes range from 400-499 indicate client errors, while server errors can be recognized by status code from 500-599. This feature allows developers to set up monitoring and logging systems based on the error codes.

On the other hand, GraphQL always return the status code 200 OK in the response, while the error is only indicated by looking at the response body, where there is an `error` field containing details about the issues. For example, as shown in Figure 14 below, the mutation `UpdateTask` is executed without at least one parameter among title, description, status, and assignees' IDs. Therefore, an error is thrown with message "Invalid request" and an error code "BAD_USER_INPUT" is returned. Although the message can be more specific, and the error codes are provided out of the box by Apollo Server's error subclasses, it is not a straightforward task to build a systematically monitoring method, especially for large applications.

QUERY

```
1  mutation UpdateTask($taskID: ID!) {
2    updateTask(id: $taskID) {
3      title
4      description
5      status
6      assignees {
7        name
8      }
9    }
10 }
```

GRAPHQL VARIABLES

```
1  {
2    "taskID": "1fpexi3ev"
3  }
```

```
1  {
2    "errors": [
3      {
4        "message": "Invalid request",
5        "locations": [
6          {
7            "line": 2,
8            "column": 3
9          }
10       ],
11       "path": [
12         "updateTask"
13       ],
14       "extensions": {
15         "code": "BAD_USER_INPUT",
16         "exception": {
17           "stacktrace": [
18             "UserInputError: Invalid request",
19             "    at Object.updateTask (file:///E:/Metropolia/Bachelor%20thesis/code/graphql-server/resolvers/taskResolver.js:30:23)",
20             "    at field.resolve (E:\\Metropolia\\Bachelor thesis\\code\\graphql-server\\node_modules\\apollo-server-core\\dist\\utils\\schemaInstrumentation.js:56:26)",
21             "    at executeField (E:\\Metropolia\\Bachelor thesis\\code\\graphql-server\\node_modules\\graphql\\execution\\execute.js:479:20)",
```

Figure 14 Error when executing mutation "UpdateTask" due to missing parameters

Data format

GraphQL only supports sending text data out of the box. In complex cases like files upload, third-party libraries are required, which comes with various risks such as licenses, lack of maintenance or malicious codes. For responses, JSON is the only way for GraphQL to represent data. On the other hand, REST brings the convenience of supporting multiple data formats such as xml, octet-stream, audio or image formats, etc.

Caching mechanism

In modern applications where speed and accuracy are keys, data caching is needed more than ever so that the client can avoid fetching the same resource multiple times. REST works seamlessly with HTTP caching mechanism. REST API response header often contains information about caching allowance of a resource, as well as the duration of cache validity. Until the resource cache is expired, the client will use the cached data instead of sending the same requests again to the server.

For GraphQL, caching is not as simple. Because the same URI is used for all the requests, GraphQL doesn’t directly provide a globally unique identifier for a given data object. Libraries or frameworks for GraphQL server such as Apollo Server are now implementing caching mechanism under the hood. But for those who does not use them, they have to set up caching by themselves with a unique identifier of their own.

5 Conclusion

In general, both REST and GraphQL are approaches to designing API systems. It means that they share some similar concepts: they always must call one or many functions on the server to retrieve data and they are often dependent on frameworks and libraries to handle the network-related matters.

Having been popular in the industry for a long time, REST proves its flexibility in adapting to a wide range of use cases, and reliability backed by an ecosystem of tools and frameworks. REST has become a fundamental concept most likely known by all stakeholders in every IT system. For applications that require a monitoring system with intuitive error handling, or relies on HTTP caching mechanism, REST is trusted as the safer option.

While REST will still be indispensable for a long time in the future, there definitely is room for upgrade and improvements. And this is where GraphQL steps in to be most relevant. Products built with GraphQL will reap more benefits in the long run thanks to the high performance and cost saving processes, especially for those cross-platform applications that requires smooth user interactions and real-time presentation of data. GraphQL also has the potential of bringing better developer experience, by providing a type safe system and allowing independent process between front-end and backend. However, being still in its early days, GraphQL has a fairly steep learning curve and it is expected to change rapidly over the next few years.

It is still too early to say that GraphQL will replace REST, and it should not be considered as such. But the rising of GraphQL has been bringing about a new breeze of change in the technologies landscape. Now that there is another option beside REST, and the decision could be very much subjective, developers and product owners need to take them both into consideration, along with other business specific constraints, budget, schedule, and available resources, when designing API systems.

References

1 Who's using GraphQL? [Internet]. Graphql.org. 2020 [cited 07 October 2020]. Available from: https://graphql.org/users/

2 Haroon Shakirat Oluwatosin. 2014. Client-Server Model. IOSR Journal of Computer Engineering (IOSR-JCE), Issue 1, January 2014, p. 67.

3 Javascript – MDN Web Docs [Internet]. 2022 [last modified 27 April 2022]. Developer.mozilla.org. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript

4 ECMA-404 The JSON Data Interchange Standard. 2nd edition, December 2017. Available from: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/

5 Array in Javascript – MDN Web Docs [Internet]. 2022 [last modified 15 April 2022]. Developer.mozilla.org. Available from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

6 Mark Massé. 2012. REST API Design Rulebook. United States of America: O'Reilly Media, Inc.

7 Kenneth Lange. 2016. The Little Book on REST Services. Electronic book. Copenhagen: Kenneth Lange. https://www.kennethlange.com. Accessed 25 September 2020.

8 Gleison Brito, Marco Tulio Valente. February 2020. REST vs GraphQL: A Controlled Experiment. Electronic book. ResearchGate. https://www.researchgate.net/publication/339413273.

9 Robin Wieruch. 2018. The Road to GraphQL. Electronic book. Leanpub. http://leanpub.com. Accessed 28 September 2020.

10 Internet Standards - RFCs. IETF.org. Available from: https://www.ietf.org/standards/rfcs/

11 GraphQL is the better REST [Internet]. Howtographql.com. 2020 [cited 07 October 2020]. Available from: https://www.howtographql.com/basics/1-graphql-is-the-better-rest

12 R. Fielding. 1999. RFC 1616. The Internet Society. Available from: https://datatracker.ietf.org/doc/html/rfc2616