**Final Project Report**

**Kevin Chisholm**

**Andrew Rohn**

**CMSC 495: Current Threads and Projects in Computer Science**

**Fall 2019 - Section 6381**

**University of Maryland Global Campus**

Table of Contents

[Overview 3](#_Toc21910680)

[Background Information 3](#_Toc21910681)

[Group Members 4](#_Toc21910682)

[*Kevin Chisholm* 4](#_Toc21910683)

[*Andrew Rohn* 4](#_Toc21910684)

[Project Plan 5](#_Toc21910685)

[Goal 5](#_Toc21910686)

[Requirements Specification 5](#_Toc21910687)

[System Specification 6](#_Toc21910688)

[User’s Guide 7](#_Toc21910689)

[Account Creation 7](#_Toc21910690)

[REST API Calls 8](#_Toc21910691)

[GraphQL API Calls 11](#_Toc21910692)

[Test Plan and Results 13](#_Toc21910693)

[Test Case Table 13](#_Toc21910694)

[Screen Captures 13](#_Toc21910695)

[Design and Alternate Designs 14](#_Toc21910696)

[Development History 15](#_Toc21910697)

[ Week 1 15](#_Toc21910698)

[ Week 2 15](#_Toc21910699)

[ Week 3 15](#_Toc21910700)

[ Week 4 15](#_Toc21910701)

[ Week 5 15](#_Toc21910702)

[ Week 6 15](#_Toc21910703)

[ Week 7 15](#_Toc21910704)

[ Week 8 15](#_Toc21910705)

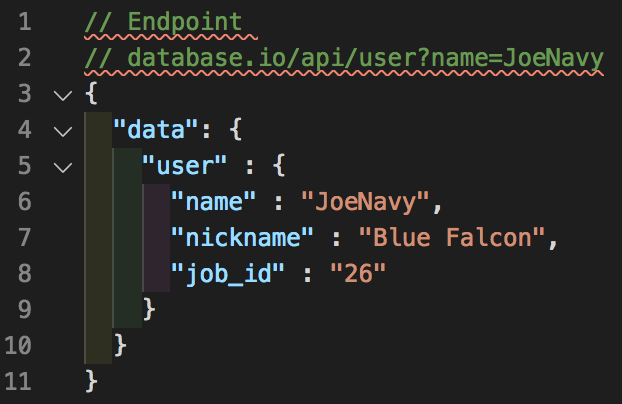
[Conclusion 16](#_Toc21910706)

# Overview

## Background Information

For this course’s final project, Group Alpha decided we would illustrate the difference between a traditional REST API, as opposed to the more recent GraphQL API. Traditional REST API’s have been the de facto method to share serialized data for the use of front-end applications across the web.  Due to its widespread adoption, many issues surrounding the use and implementation of REST APIs have been brought up in numerous discussions.

The first issue with REST APIs is its reliance on multiple endpoints to retrieve desired information for the user. Let’s imagine that we wanted to find out what job specialization someone held. With a REST API, we would first have to find the specific person by searching the user endpoint and then take the relevant information from that endpoint to use for an additional query of the job endpoint. In figures 1 and 2, we demonstrate what the returned data from separate API queries would look like.



**Figure 1 – User Endpoint Figure 2 – Job Endpoint**

The second issue with REST APIs is how we handle the data that we receive from queries. Our front-end application may be expecting data of certain types. In the examples provided in figures 1 and 2, the “job\_id” field return different data types. In the first API call, a string is returned; and in the second, an integer. While most front-end applications can be programmed to handle such situations, it still remains as an unnecessary obstacle to data retrieval. In implementations such as TypeScript however, this would blow up completely due to the expected data types not corresponding to the data received. There are best practices for API calls, but it is guaranteed that all APIs will take these into consideration.

In order to overcome both of the problems mentioned above, Facebook announced in 2015 a new technology that was lauded as the replacement for traditional REST APIs, a technology known as GraphQL. A GraphQL query reaches out to a single endpoint to retrieve only the requested data, solving the first problem. GraphQL is also a typed system, requiring the data types of the data queried and returned to be specified, thus preventing problem two (see Figure 3). It is this, which we wish demonstrate with our project.



Figure 3 - GraphQL API call

## Group Members

### Kevin Chisholm

Leading this project is Kevin Chisholm. He is a very experienced web developer and as such, served as the project manager. Kevin did most of the development for this project; creating the database, designing the website, and developing the REST and GraphQL API’s.

### Andrew Rohn

The other member of Group Alpha is Andrew Rohn. He assisted with the front-end development. He also contributed by handling database entries, testing, and documentation for the project.

# Project Plan

## Goal

The goal of this project is to implement a REST API and GraphQL API that allows the user to fetch answers to trivia questions from a database using their respective syntax. The API queries will demonstrate the benefits of GraphQL over a traditional REST API in terms of the quality of results. For simplicity, the REST API portion of the application will only be used for demonstration purposes while the GraphQL API will be utilized to demonstrate our trivia question delivery service.

## Requirements Specification

In order for the program to meet our goal and to effectively illustrate the superiority of a GraphQL API call in comparison to a REST API call, several things had to be developed:

* **Database**

First, a database was needed to store information in. This database would be where API calls would retrieve their information from.

* **REST API**

Second, we would need to develop our own REST API back-end to interface with database and retrieve answers.

* **GraphQL API**

Third, we would need to develop a user-friendly interface to the already existing GraphQL architecture.

* **Website**

Finally, we wanted a user-friendly webpage where the user could manage an account. The website would serve as the front-end where the user can enter in API queries and see the fetched results from the database.

## System Specification

Each of the aforementioned aspects of the program was achieved through software development on multiple platforms, using various programming languages.

Software Used:

* **Database**

The database was created and hosted on Amazon Web Services.

* **REST API**

The REST API was developed in-house with the Laravel framework utilizing PHP.

* **GraphQL**

The GraphQL API was implemented in its on specific language, interfacing through an Apollo Server. An intuitive interface layer was added on top of it to facilitate legibility and ease of use.

* **Website**

The website front-end was developed utilizing the Bootstrap framework using a combination of HTML, CSS, and JavaScript. The website itself is hosted on Heroku, a cloud application platform.

This program has no unique hardware or software requirements and simply requires a stable internet connection on a modern web browser, making it accessible to virtually anyone.

# User’s Guide

## Account Creation

The first thing the user needs to do is access the Quizmate website at the following URL: <https://quiz-mate-api.herokuapp.com/>. This will bring the user to the homepage (Figure 4).

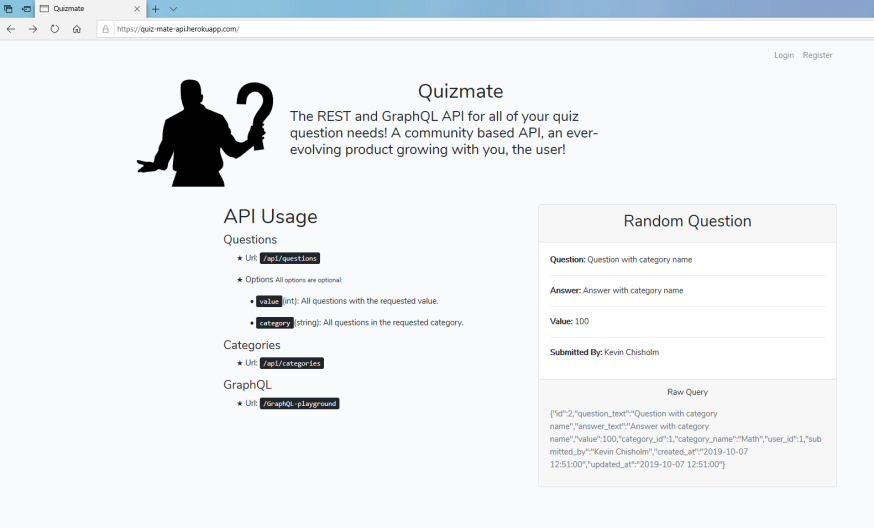


Figure 4 - Quizmate Homepage

From there, the user should click on ‘Register’ in the upper right-hand corner and create an account with an email address and a password (Figure 5).

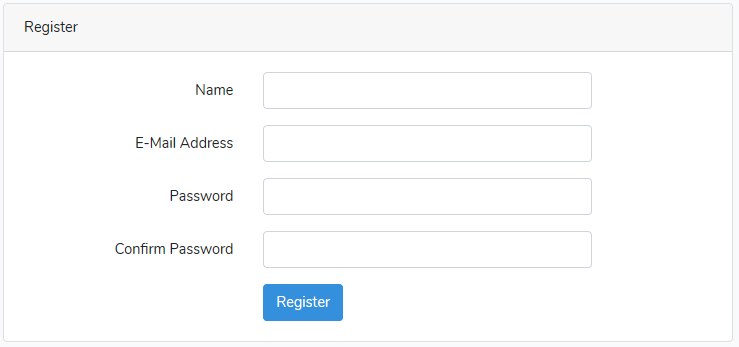


Figure 5 – Registration

After registration, the user can then click on ‘Login’ in the upper right-hand corner to login using the username and password they just created (Figure 6).

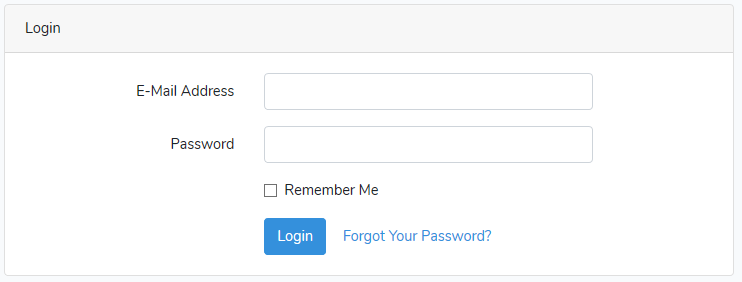


Figure 6 – Login

Should the user forget their password for any reason, they can simply click on the “Forgot Your Password?” hyperlink, where they will be given the opportunity to reset their password, provided they still remember the email address they used to create the account. One click of the “Send Password Reset Link” and the user will be able to reset their password by clicking a link in their email (Figure 7).

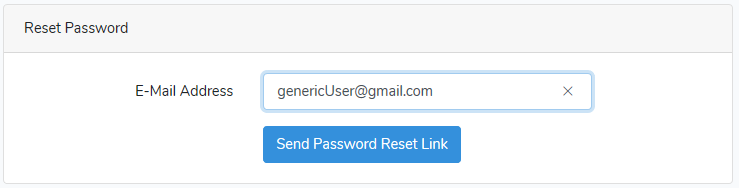


Figure 7 – Password Reset

## REST API Calls

Once the user is logged in, they will be redirected back to the home page, but this time they will be able to make API calls. If the user is an administrator, there will be buttons present that allow for the user to create new categories and questions that are entered into the database (Figure 8). If not, these buttons will be absent.

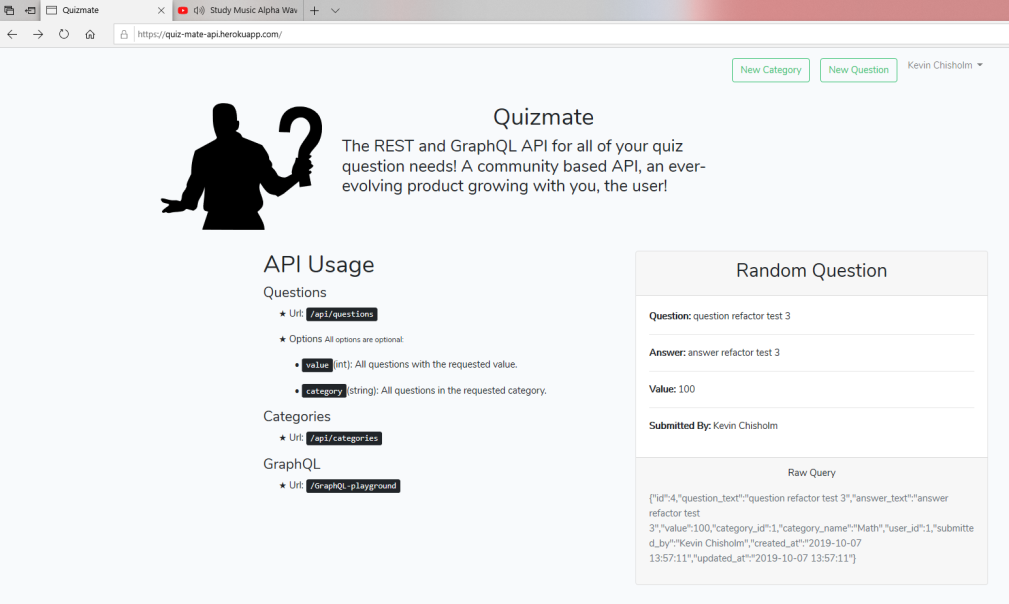


Figure 8 - Admin's Home Page

It needs to be mentioned that ordinary users will not be able to add categories or questions to the database. These rights are reserved for administrators. The user will only be able to make API queries, as intended for the purpose of this project. If they attempt to create a category or a question they will be met with a 403 error (Figure 9).



Figure 9 - 403 Error

In the left-hand corner of main page, there are several URL extensions specified. By appending the website’s URL with ‘/api/questions’ so that the URL is now ‘https://quiz-mate-api.herokuapp.com/api/questions’, the user will be able to see the questions that are saved in the database (Figure 10).

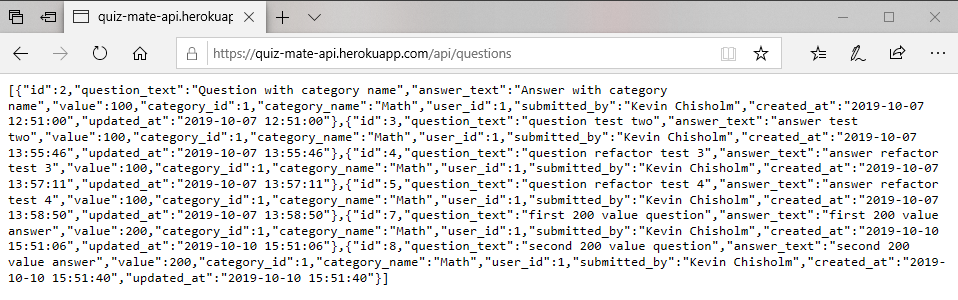


Figure 10 - Questions in Database

To see the categories in the database, the user can append the home page URL with ‘/api/categories’ so that the URL is now ‘https://quiz-mate-api.herokuapp.com/api/categories’. This will display the categories that are saved in the database (Figure 11).

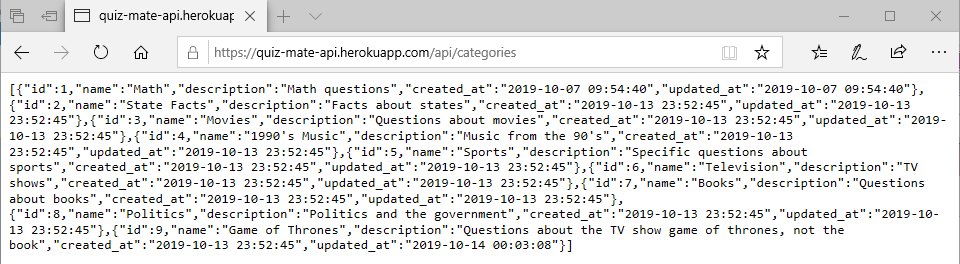


Figure 11 - Categories in Database

However, should the user want to narrow down the results; the user can add a query string at the end of the URL to find questions with specific values. For example, if the user wants to find all the questions with a value of 200 points; they will need to enter in ‘https://quiz-mate-api.herokuapp.com/api/questions?value=200’ into the address bar. This will return all the questions in the database with a value of 200 (Figure 12).

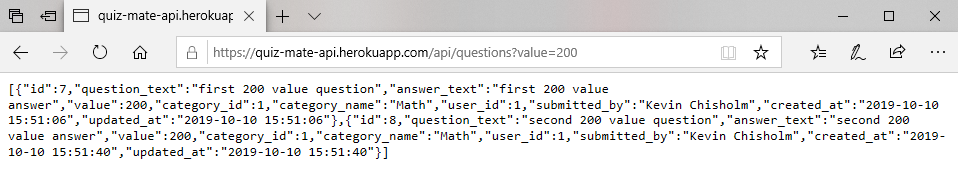


Figure 12 - Questions in Database w/ Value of 200

Also, the user can find all the questions that belong in a specific category. For example, they can use the query string ‘api/questions?category=Game of Thrones.’ to find all the questions in the database that belong to the category “Game of Thrones” (Figure 13).

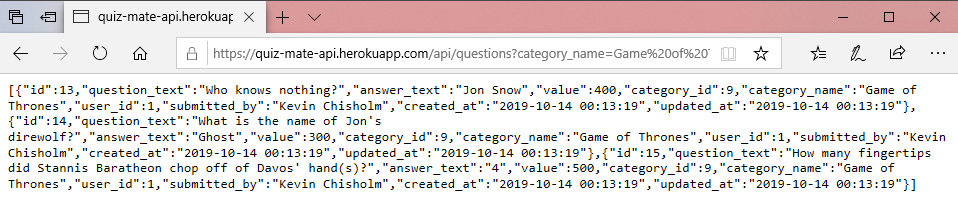


Figure 13 - Questions in Database that Belong to Category "Game of Thrones"

## GraphQL API Calls

To access the GraphQL portion of the website, the user needs to append the URL with ‘graphql-playground’ so that the URL is now ‘https://quiz-mate-api.herokuapp.com/graphql-playground’. This will bring the user to the GraphQL API front-end (Figure 14). The graphql playground is added for demonstration purposes. Unlike a REST API, a graphql query is conducted with a POST request, which seems unusual, as a REST API requires a GET request.

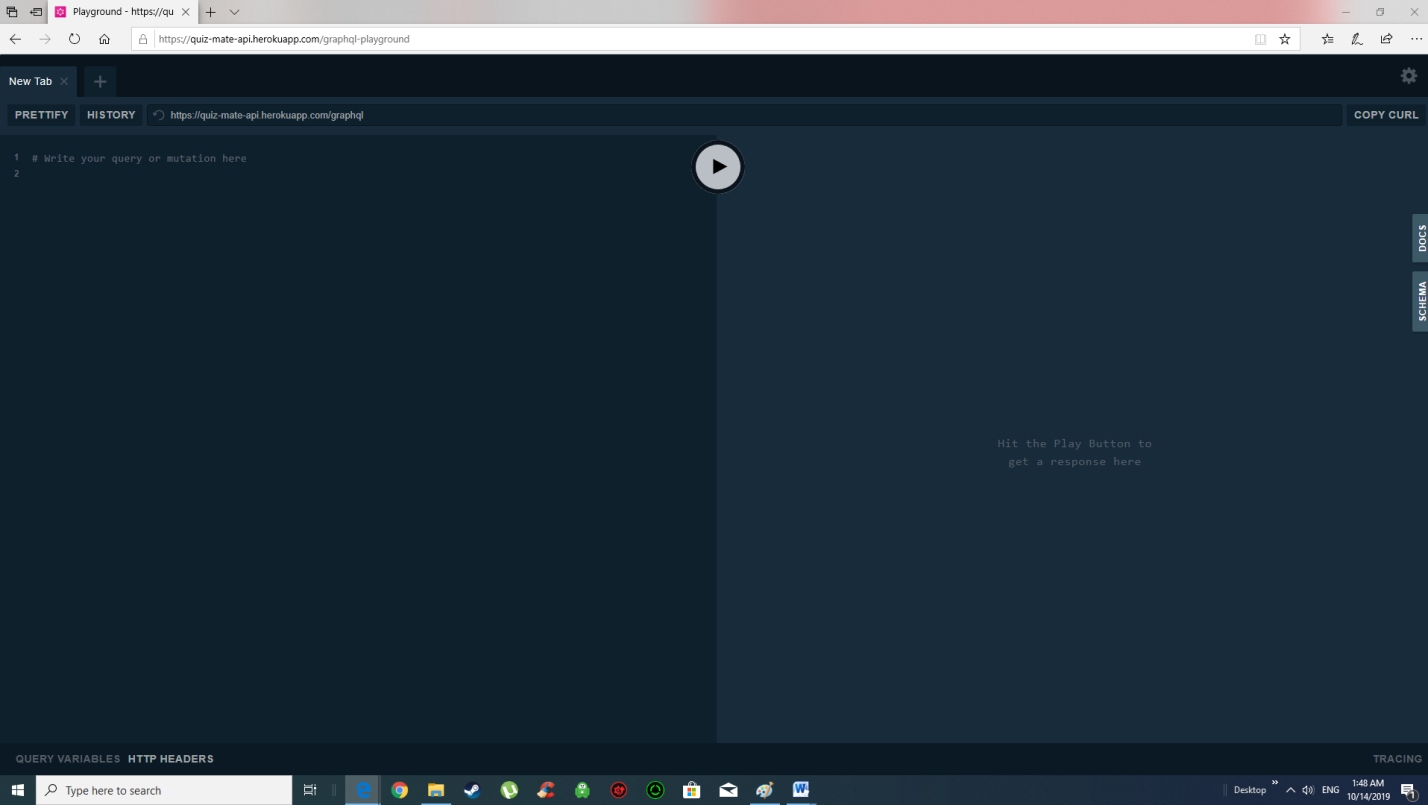


Figure 14 - GraphQL Layer

To make an API query, the user needs to enter the desired information on the left hand side of the screen. GraphQL API queries are formatted differently than REST API requests and resemble JavaScript code. An example of a GraphQL query would be:

{

allCategories {

id

name

description

}

}

Such a query will return all the values stored under “id”, “name”, and “description” for all categories on the right-hand side of the screen (Figure 15). To submit the API request, the user needs to press the button in the middle of the screen.

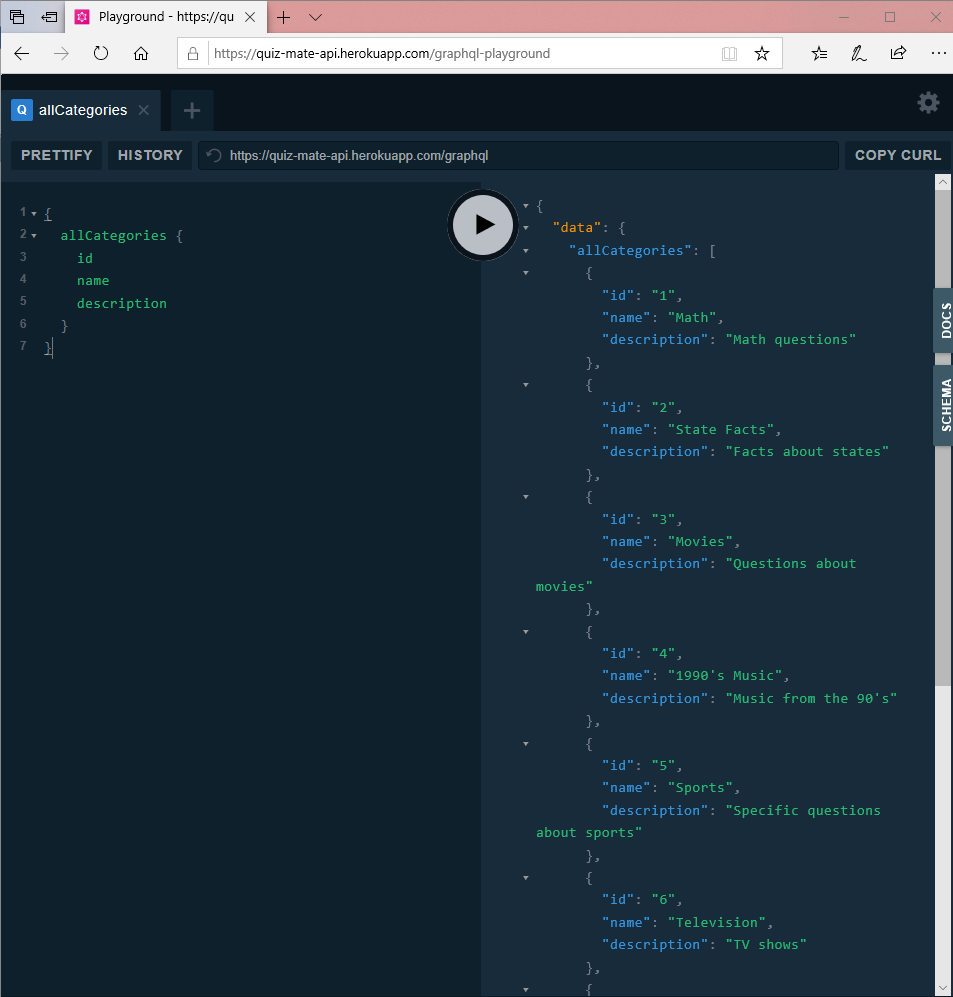


Figure 15 - GraphQL API Fetch Results

If the user wanted to find something more specific, such as the name of the category with an ID of 9, then they would enter in this GraphQL query:

{

category(id:9) {

name

}

}

The result of such a query is illustrated in Figure 16.

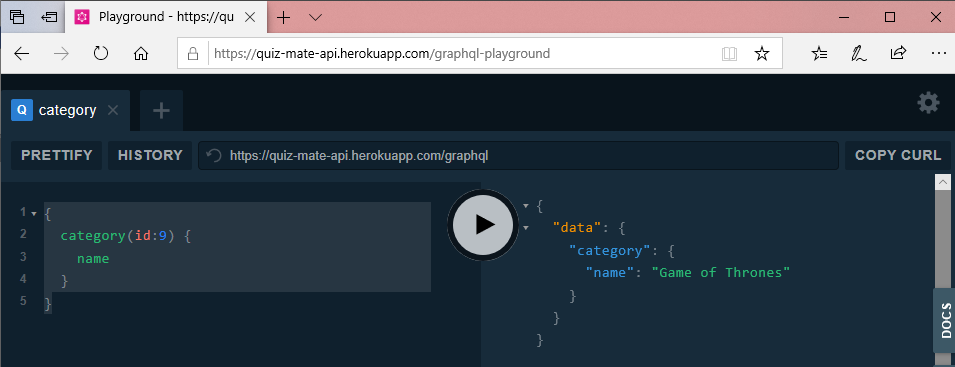


Figure 16 - Name of Category w/ ID of 9

# Test Plan and Results

## Test Case Table

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Expected Result | Actual Result | Pass? |
| #1  API: REST  Search Query:  /api/questions?value=100 | Returns all the questions in the database with a value of 100 | All the questions in the database with a value of 100 were returned  (Figure 17) | **Passed** |
| #2  API: REST  Search Query:  api/questions?value=400 | Returns all the questions in the database with a value of 400 | All the questions in the database with a value of 400 were returned  (Figure 18) | **Passed** |
| #3  API: REST  Search Query:  api/questions?category\_name=  State Facts | Returns all the questions in the database that are in the category “State Facts” | All the questions in the database that belonged in category “State Facts” were returned  (Figure 19) | **Passed** |
| #4  API: REST  Search Query:  api/categories | Returns all the information for all the categories in the database | Returned all the categories in the database  (Figure 20) | **Passed** |
| #5  GraphQL Query:  {  allCategories {  name  }  } | Returns the names of each category in the database | Returned all the category names in the database | **Passed** |
| #6  GraphQL Query:  {  allCategories {  id  name  }  } | Returns the id and name of each category in the database | Returned all the ids and category names in the database | **Passed** |
| #7  GraphQL Query:  {  category(id:9) {  name  }  } | Returns the name of the category with an id of 9 | Returned the name of the category with an id of 9 | **Passed** |
| #8  GraphQL Query:  {  category(id:9) {  name  questions {  question\_text  answer\_text  value  }  } | Returns the category name of the category with the id of 9 and returns all questions associated displaying the question, answer and value. | Returned the category name of the category with the id of 9 and returned all questions associated displaying the question, answer and value. | **Passed** |

## Screen Captures

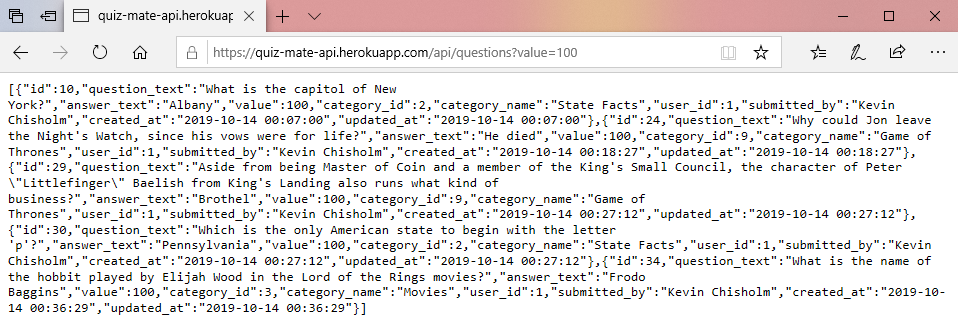


Figure 17 - Test Case 1 Results

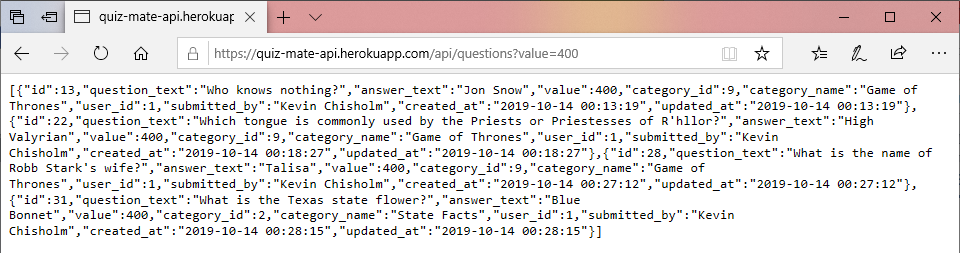


Figure 18 - Test Case 2 Results

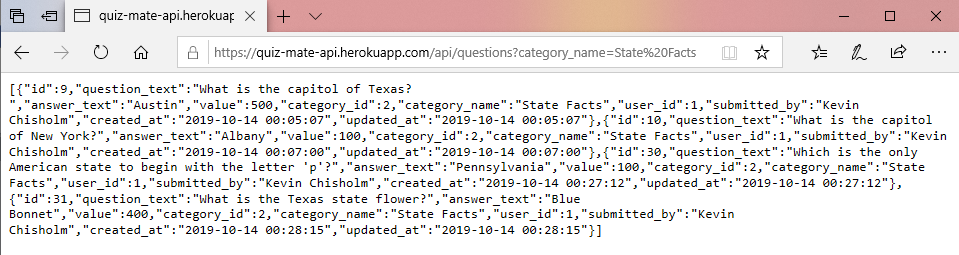


Figure 19 - Test Case 3 Results

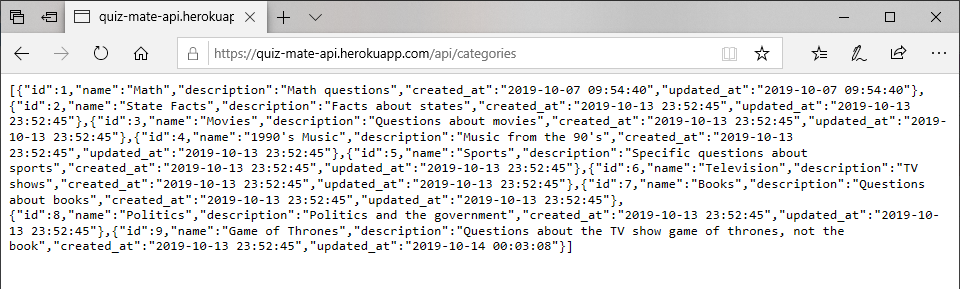


Figure 20 - Test Case 4 Results

A screen shot of a computer

Description automatically generated

Figure 21 - Test Case 5 Results

A screenshot of a computer

Description automatically generated

Figure 22 - Test Case 7 Results

A screenshot of a cell phone

Description automatically generated

Figure 23 - Test Case 7 Results

A screenshot of a computer screen

Description automatically generated

Figure 24 - Test Case 8 Results

# Design and Alternate Designs

Earlier on in the development cycle, we had planned on implementing a single-page web app with a user interface that interfaced with both the REST API and GraphQL API (see Figure 25). The user would be able to make REST and GraphQL API calls side-by-side, allowing the user to easily see the difference in how the queries are formatted and what type of information was returned.

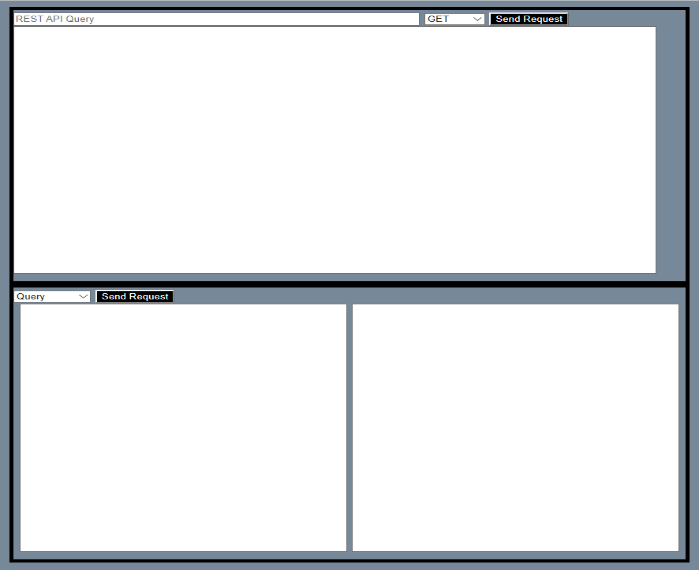


Figure 25 - Early User Interface Prototype

Also, the user would have been able to do more than make queries. The REST API would have been able to use HTTP request verbs such as POST, PUT, and DELETE to allow the user to add, modify, and delete information from the database. GraphQL has its own operations such Query, Mutation, Subscription that act in a very similar fashion.

Ultimately, it was decided we would not add this functionality for a few reasons. First, it would have added to the complexity of the program meaning a longer development time and more detailed user instructions and test cases. For the sake of time, we decided to keep it simple. Second, giving any user the ability to make alterations to the database would increase security risks. Finally, the whole point of an API is to serve as a standardized data retrieval tool. We didn’t want to stray away from this central theme. Had we gone in this direction, we would have had develop it on a platform such as VueJS or ReactJS to handle real time virtual DOM manipulations.

Another thing that we had considered implementing was additional account features, such as a dashboard that would display recent activity such as successful API calls and other user information. This was also gutted due to time restraints.

# Development History

## Week 1

As per the course, the project began with the formation of Group Alpha in the first week.

## Week 2

During the second week, we decided what we wanted our project to be about and developed a project plan outlining specific goals we wanted to achieve.

## Week 3

In the third week, we wrote a test plan explaining what the desired outcome of the project would be and how we could effectively test and document the results.

## Week 4

In the fourth week, we submitted a document detailing the intended design of the project, going into specifics about what types of platforms, programming languages, methods, fields, and data structures would be needed to develop the project. This was also the week that development of the project began in earnest.

## Week 5

Starting in week five, we began releasing updated segments of source code in weekly phases. The first phase was completed in week five and included the creation of the database. This was also when the front-end and back-end development began.

## Week 6

The second phase was released during the sixth week and saw the creation of the REST and GraphQL APIs, though further work still needed to be done. We began populating the database with entries at this point so that we could begin testing. This is also the point where we decided not to do a single-page web app, but rather interface the REST and GraphQL APIs separately on a website for simplicity and to cut down on development time.

## Week 7

The third phase introduced the website as well as some more testing and fine tuning of the APIs and database.

## Week 8

Finally, we arrive at the present. Everything has been fully implemented and all the testing has been completed. We are excited to say that we have met our goals and are ready for others to use the program.

# Conclusion

**Lessons Learned**

- We should have focused on test driven development with this project. Tests should have been implemented before the functionality was written. This could have helped us take care of all our problems before we reached the implementation phase. There were many times where we ran into 403 errors, 404 errors, SQL errors that could have been caught by the built-in unit tests.

- There was too much time spent on making each section perfect. The approach should have been getting out a minimum viable product where all the functionality linked up before refining any specific area.

**Design Strengths**

**-** A major design strength of this project is having a functional REST API and GraphQL API on a single server. Serving both and making considerations for both was a difficult task in itself.

**Limitations**

**-** This application has no severe limitations that a traditional REST API or GraphQL API would have because they are both implemented in a single project.

**Suggestions for Future Improvements**

**-** Create a front-end application to utilize the API such as a quiz game.

- Add the feature for users to submit questions and categories to be approved by the administrators and added to the database.

- Set up versioning for the REST API in case there are changes made later while users are already consuming it for their own apps.

- Flesh out the database of questions.