Modern API Delivery

Kevin Chisholm

Andrew Rohn

CMSC 495 6381

University of Maryland Global Campus

Contents

[Problem 3](#_Toc19466678)

[Solution 5](#_Toc19466679)

[Project 5](#_Toc19466680)

[Roles 6](#_Toc19466681)

[Course Dictated Milestones 6](#_Toc19466682)

[Project Goals 6](#_Toc19466683)

[Technology Stack 7](#_Toc19466684)

[Stretch Goals 7](#_Toc19466685)

[Entity Relationship Diagram 8](#_Toc19466686)

[Site Map 9](#_Toc19466687)

[Implementation 10](#_Toc19466688)

[REST API User Guide 10](#_Toc19466689)

[GraphQL API User Guide 11](#_Toc19466690)

[Test Plan 12](#_Toc19466691)

[User Roles 14](#_Toc19466692)

[References 15](#_Toc19466693)

## Problem

Traditional REST API’s have been the de facto method to share serialized data for the use of front-end applications.  Due to its popularity, there have been many issues discussed around the use and implementation of REST APIs.

The first issue with this default method is the reliance on multiple endpoints to retrieve the information one may require. As the consumer of the API, imagine we wanted to find out what job specialization someone held, we would first have to find the specific person by hitting the user endpoint then take the relevant information from that endpoint to hit the job endpoint and search by the ID. In figures 1 and 2 we demonstrate what the API calls, and the returned data would look like.

The second issue we run into is handling the data that we receive from the API. There are best practices, but it is not sure that all APIs will follow these practices. In a front end application, we may be expecting certain types of data. In the examples provided in figures 1 and 2, the job\_id field returns a string in the first API call and a number in the second API call. In JavaScript, this may be a problem that can be worked around, but there is always the need to avoid unnecessary side-effects. In implementations such as TypeScript, this would blow up completely due to the expected data types, not corresponding to the data received.

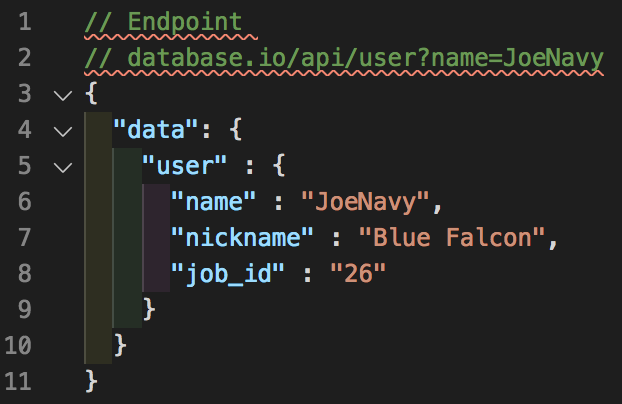


Figure 1

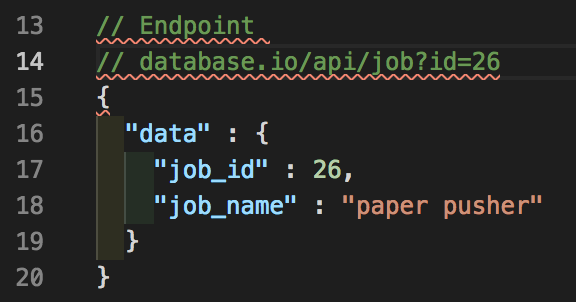


Figure 2

## Solution

In 2015, Facebook announced a new technology that was lauded as the replacement for traditional REST APIs, a technology known as GraphQL. GraphQL is a query language for APIs that aim to solve both of the problems listed in the section above. A GraphQL query reaches out to a single endpoint to retrieve only the requested data, solving the first problem discussed. GraphQL is also a typed system, requiring the data types of the data queried and returned to be specified, solving problem number two.

Figure 3 [1]

## Project

We will implement a REST API and GraphQL API that serves trivia questions to demonstrate the benefits of GraphQL over a traditional REST API.  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.

## Roles

Kevin Chisholm:  Project Manager, Back-end Designer, Dev ops, Programmer

Andrew Rohn:  Front-end Designer, Documentation, Testing, Programmer

## Course Dictated Milestones

Week 1: Project Goal

Week 2: Requirements Analysis

Week 3: User's Guide and Test Plan

Week 4: Design

Week 5: Coding Plan: Phase I

Week 6: Coding Plan: Phase II

Week 7: Coding Plan: Phase III

Week 8: Final Evaluation

## Project Goals

Week 1:  Project idea, group formations and role designation.

Week 2:  Database schema design, tech stack choice.

Week 3:  REST API implementation and deployment

Week 4:  GraphQL implementation and deployment

Week 5:  Front-end implementation and deployment

Week 6:  Stretch goal work

Week 7:  Final testing, stretch goal work

Week 8:  Completion of project and documentation

## Technology Stack

Front-end:  Vue (Javascript library), Blade (PHP Templating Engine)

Back-end:  Laravel (PHP framework)

Database: PostgresQL

Version control:  Git (Github)

Deployment:  Heroku

Environment:  Mac

## Stretch Goals

Create quiz game

Make game multiplayer utilizing web sockets

## Entity Relationship Diagram

Figure 4

## Site Map

* [quizmate.com](http://test.com)

- /api

- /users

- /:id

- /:name

- /questions

- /:id

- /:value

- /categories

- /:id

- /:name

- /graphql

- /game

- /leaderboard

## Implementation

The goal of our project is to illustrate the benefits of using the GraphQL API query language over that of a traditional Representational State Transfer (REST) API. In order to do this, we will be creating a database, with which both query languages will communicate. The database will be populated with values that we ourselves will enter. Both the GraphQL and REST APIs will be used to access the same or similar data so the user can see how the two work and which returns more efficient results.

Since the GraphQL API and the REST API differ both in front-end and back-end design, a unique front and a unique back-end will need to be created for both. However, they will still access the same data from the same database. The results of both types of requests will be JavaScript Object Notation (JSON) files displayed in their respective output area. Since the user interfaces will be different for the two, they will need their own user guides.

## REST API User Guide

The REST API defines how applications communicate with each other over the Hypertext Transfer Protocol (HTTP) and thus uses HTTP request verbs for queries. Therefore, our REST API user interface will allow the user to select one of four HTTP request verbs (GET, POST, PUT, and DELETE) via a drop-down menu in order to manipulate data in the database; GET for retrieving data, POST for adding data, PUT for replacing data, and DELETE for removing data. There will be a text field where the user can input the HTTP request query. Once an HTTP request verb has been selected and the request query has been filled, the user will be able to send the request by pressing a “Send Request” button. Afterwards, the results of that request will be displayed in a text area. An example of such a user interface can be seen below in Figure 1.

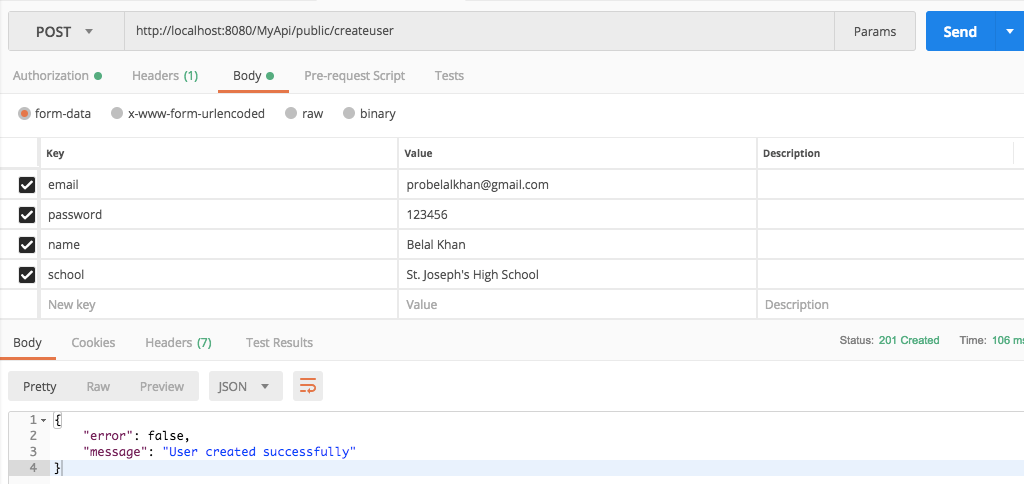


Figure 1 - REST API User Interface

## GraphQL API User Guide

The GraphQL API doesn’t utilize HTTP request verbs like REST API and instead uses three different operations: query, mutation, and subscription. A query operation is used for accessing data, a mutation for adding/removing/replacing data, and a subscription is used for opening a web socket to allow for dynamic request updates should values change in the database. We haven’t decided whether or not to add any functionality beyond a simple data query. Should we do so, a drop-down menu could be implemented in a similar fashion to the REST API’s HTTP request verb selection. Similarly a “Send Request” button will be pressed to process the query.

A GraphQL request query is formatted like typical JavaScript code with curly brackets, indentations, and multiple lines. As such, a large text area is needed for the input as well as the output. An example of a GraphQL user interface can be seen below in Figure 2.

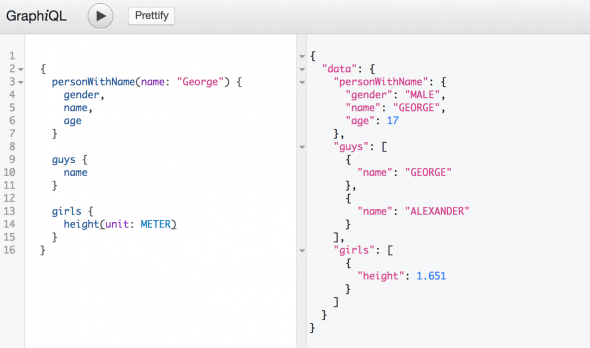


Figure 2 - GraphQL User Interface

## Test Plan

After the database has been populated and both the front-end and back-end designs have been completed, we will begin testing. Testing will reveal the quality of data returned from both APIs, so that the user can see the difference in the two API query languages side-by-side. We will be doing numerous test cases so that the difference between the two is evident. The user input, expected output, and actual outputs will be entered into a data table similar to the one below (see Figure 3). Screenshots will also be provided of each test case with a corresponding test case number so the difference in query results of the two API query languages can be seen side-by-side.

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **User Input** | **Expected Output** | **Actual Output** |
| **#1** | **API: REST**  **HTTP Verb:**  **Query:** | (Expected Data Values) | (Actual Data Values) |
| **API: GraphQL**  **Query:** | (Expected Data Values) | (Actual Data Values) |
| **#2** | **API: REST**  **HTTP Verb:**  **Query:** | (Expected Data Values) | (Actual Data Values) |
| **API: GraphQL**  **Query:** | (Expected Data Values) | (Actual Data Values) |
| **#3…** | **API: REST**  **HTTP Verb:**  **Query:** | (Expected Data Values) | (Actual Data Values) |
| **API: GraphQL**  **Query:** | (Expected Data Values) | (Actual Data Values) |

Figure 3 - Test Case Table

## User Roles

For this project, there will only be two roles when handling authentication and authorization. The two roles will be admin and user. The Admin role will have access to making multiple requests to the database using multiple HTTP verbs to manage the questions while the User role will only have access to the GET request on the REST API side of the application and the query request on the GraphQL side.

Another difference between the Admin and User side of the application will be the request limit. The Admin role will have no limit on requests made to the HTTP side of the application, while the User role and guest users will have their usage throttled.

For this application, we will not implement an API key for users to utilize the API for their own purposes.

## References

[1]. *Queries and Mutations*. graphql.org/learn/queries/.