# Final Report
# Web Engineering 2019
# Group 13

Andrew Lalis *Lead Systems Architect and Web Engineer*
Tom den Boon *Database Domain Specialist*
George Rutherford Rakshiev *Documentation, English Language Specialist*

March 29, 2019

## Introduction

*Educating the mind without educating the heart is no education at all.*
-Aristotle

When we first began this course, our primary objective was simply to pass it, because we all had many other priorities. An easy-to-use framework would be chosen, and the majority of our labors would be spent in the writing of the API documentation and the report which you see before you today.

However, that is *not* what happened.

Instead, we took a rather radical turn and decided to carve yet one more trail along the well-beaten path of PHP framework design. Starting from just two things: PHP and SQLite3, we built a fully functioning, robust API and front-end representation in a way that few others in this course most likely have, and in doing so, we have undoubtedly spent more time than the majority, but perhaps the aforementioned quote can provide some motivation as to why we thought it necessary to discover for ourselves the complexities behind creating that which is so often taken for granted.

## Technology Stack

In today's work environment, it is quite commonplace to spend more time bickering over what technologies to use than actually using said technologies. We decided early on that this would indecision and gluttony of choice would not hinder our progress, so we chose technologies with which we had solid working experience. Andrew uses PHP regularly at work, Tom has made plentiful use of SQLite3 in past projects for his university career, and George has a typing speed that can rival that of a seasoned receptionist, and the linguistic capacity of a native English speaker.

Henceforth, the division of labor was quite clear. Tom would design the database schema and queries, Andrew would use PHP to create the main API system which depends on Tom's database, and George would work in parallel to keep documents up-to-date as changes were introduced with each iteration.

During the final iteration, notably when the front-end representation of the API was required, we opted to take advantage of two more technologies to aid in producing results faster.

JQuery provides extremely simple, intuitive manipulation of the document object model, and therefore was chosen as one of the javascript libraries to be included. In particular, we are quite fond of this library's asynchronous request system, which fulfills all our needs beautifully.

To display a collection of resources, it is not trivial to construct new DOM objects in javascript, even with JQuery, when compared to the simplicity that Handlebars offers. This library was chosen to allow us to design templates for each of the many API responses that would be shown on the front-end.

## Database Design

From the dataset provided, there are a few truths which are immediately self-evident. This includes the fact that there exists a list of airports, a list of carriers, and a list of statistical data which is identified by a many-to-many mapping between the aforementioned airports and carriers.

At first glance, one would assume that the optimal solution for this is three tables, one for each of the two entities, and one for the statistical data entries. However, upon further inspection, one would note that the statistical data is split into several categories, each of which could be neatly represented in its own table. Doing so reduces the complexity of queries to the database in many aspects, reduces the complexity for potential users of the API, and allows for partial data and more efficient division of resources when processing requests.

With these sentiments in mind we initially designed a database which stored airports, carriers, and the three types of statistics. Using this schema, we encountered no issues throughout development.

When it came time during the third iteration to add additional endpoints, we decided to add a Users endpoint (which will be discussed in more depth later), which required a slight modification to the schema: the addition of two new tables.

At the end of the development process, the final schema contained the following tables:

- airports
- carriers
- statistics
- statistics_flights
- statistics_delays
- statistics_minutes_delayed
- users
- user_requests

While this provides a good overview of the structure of the database, and by consequence, the API, more detail may be desired, in which case it is advised to read through the `SQLite_initializer.php` script which contains all schema definitions and initial data insertion from the provided dataset. Discussing the intricacies of this single script's behavior is beyond the scope of this document.

## API Design

Like many of our colleagues, we were at the start unsure of exactly what was required of us in terms of specificity and breadth of the API specification, but after attending the question session and through correspondence with our helpful teaching assistant, the task was quite clear.

Using the database schema as a guide, a rough listing of all endpoints was initially drafted, and each endpoint's syntax was discussed among the group until a concensus was reached. During these discussions, the question of pagination as a solution to large result sets was brought up, and it was then that we elected to implement pagination for all endpoints which represent a collection of resources.

The majority of the endpoints need only respond to GET requests, so designing the document itself was rather trivial, since formatting could be chosen for one endpoint and copied to all others. As the medium of the document, Latex-generated PDF was chosen, and in hindsight, would have been avoided due to the poor support for the display of large JSON data and inoportune page breaks, two things which markdown on the other hand, excels at.

As you, the reader, probably now are aware of, the second API document was completely re-written in markdown, and this process took less than half the time that it took to write the first draft, even taking into account additional endpoints.

Inspired by SpringBoot's implementation of HATEOAS, our API implementation would have a strict format for responses and any navigation links to related resources. Each response would be split into a *content* and a *links* section, which makes the responses from this API very predictable. Furthermore, authentication and general security were ignored, because we as a group agreed that the focus of this project was not on security, but on designing the proper architecture and set of endpoints, and authentication could be added later if needed with very little modification to the existing codebase.

The next step for the project is the actual implementation of the API.

## API Implementation

By far the most difficult part of the whole project, creating a working API server from nothing but plain PHP files proved to be quite a formidable task. Nonetheless, as you the reader are aware, we in the end prevailed, and have a robust, fully capable API which serves its data elegantly.

From the start, a large focus was placed on modularity. *If it can be abstracted, it must be abstracted.* That was the general sentiment during the initial development phase. Each endpoint should be able to be defined with as few lines of code as possible, while still providing the flexibility that is often lacking in enterprise-scale solutions.

While the majority of the endpoints respond only to GET requests, a distinction is made between those which return one resource, and those which return a collection of resources. The latter would become known as *Paginated Endpoints*, which provide abstract functionality for fetching a subset of a collection of resources from the database and returning it in a response. This abstraction works so well that defining a new paginated endpoint is typically far less complex than defining a normal endpoint, even considering the fact that a normal endpoint needs only implement one function. Paginated endpoints automatically create a series of links to other subsets(pages) of the data, so as to guide users to easily discover all resources while retaining acceptable performance standards.

Due to the complexities of manipulating statistical data entries which have referential integrity constraints with each of the several sub-relations, each of the required POST, PATCH, and DELETE endpoints have their logic abstracted by the generic statistics endpoint, such that each sub-relation's endpoint need only declare that it is a child of the statistics endpoint, and the rest is trivial, once again perpetuating our principle of abstraction above all else.

During development, care was taken to account for as many erroneous situations as possible, and as a result, our API features specially-designed error message responses which provide descriptive answers to the user as to why their request could not be fulfilled.

Although the intricacies of creating a working API server, including a router, endpoint abstractions, pagination, database operation abstraction, and more, demanded a large amount of time, we feel that the journey of discovery was well worth the effort.

## Front-End Design

While every other section thus far could be handled masterfully, so much cannot be said about the front-end of the website. While every single one of the members of our group has ample experience with basic HTML and some CSS, that is just about all we have, putting us at a major disadvantage when compared to other groups who have some background knowledge of javascript frameworks.

Despite this challenging outlook, Andrew created a basic single-page interface which displays different endpoints exposed by the API. Using JQuery and Handlebars (as mentioned previously) for performing asynchronous requests to the API and formatting the results, a quick, unstyled skeleton of a page was formed, to which style could later be added.

Styling of the web page does in no way reflect the abilities of any of the individual group members with regard to programming or API design, but does reflect our abilities in graphic design. Regardless of how the website appears, it does in fact provide all of the data which it must, and also plays some music in the background, which we thought was a nice touch, especially for when the time comes to demonstrate the website and API.

## Reflection

This project was a learning experience for everyone involved. This can be interpreted to mean both good or bad, depending on one's perspective.

Initially, our strategy was to finish this course as easily and quickly as possible, yet we stayed the hand of fate and chose instead to build an API truly from the ground up. During this process, discoveries were made about the inner workings of a router, and how it uses pattern matching to find an endpoint to fulfill a user's request, or how primitive database abstraction layers work to remove SQL from the hands of the programmer. With this knowledge, we are all better equipped to work with, and more deeply understand, any existing framework on a more fundamental level than most users typically do. The experience we gained with this project will carry over to any (and most certainly there will be) new web engineering projects.

Despite the knowledge gained, sleep was lost, and time which could have been otherwise spent working the more difficult Computer Graphics course during this block. Knowing now what we do, the next time that an API needs to be developed, we will take full advantage of the rich ecosystem of web apps and frameworks that do all the hard work for us. While developing one's own custom solution to a problem is sometimes the cleanest or most elegant method, it is often not the most time efficient, and as the sands of time flow, so does the need for functioning software ever burgeon, so that one must eventually make a choice between ever diminishing improvement bordering on perfection, or delivering that which is both tangible and functional. We believe that we have struck a balance between the two.