University of Groningen

Web Engineering

Group 5

API Architecture

Authors:
Alina Boshchenko
Ai Deng
Henry Salas

March 21, 2019

Contents

1 Web API

Based on simple REST principles, we implemented the web API endpoints for the airports database according to the requirements document (included in the project's Github repository).

Every API endpoint we implemented has the 'API/' prefix in the URL. The API provides a set of endpoints, each with its own unique path.

1.1 Requests

The web API is based on REST principles and thus uses appropriate HTTP verbs for each action: GET, POST, PUT, DELETE.

1.2 Responses

The web API returns JSON objects as default. If desired, csv types can also be returned by adding 'text/csv' in the Content-Type key of the request header.

1.3 Response Status Codes

HTTP codes:

200: OK - The request has succeeded. The client can read the result of the request in the body and the headers of the response.

201: Created - The request has been fulfilled and resulted in a new resource being created.

400: Bad Request - The request could not be understood by the server due to malformed syntax.

404: Not Found - The requested resource could not be found. This error can be due to a temporary or permanent condition.

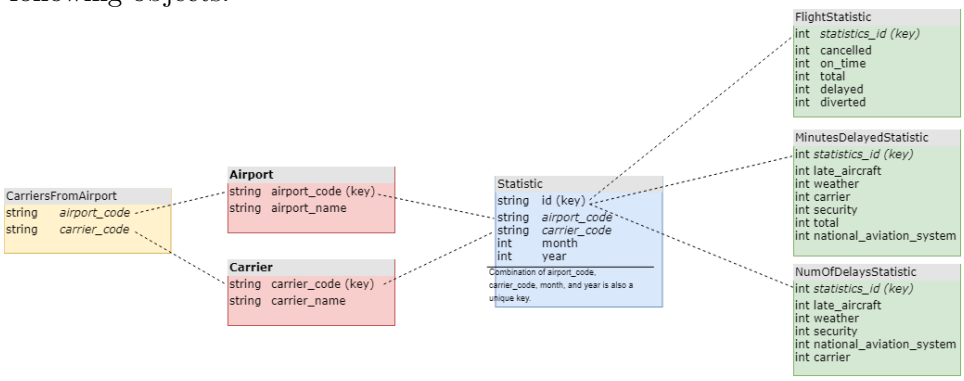
500: Internal Server Error. You should never receive this error because our clever coders catch them all ... but if you are unlucky enough to get one, please report it to us through a comment at the bottom of this page.

2 Object Models

We decided that for this project, we would utilize an SQL database, namely MariaDB. The reason we chose this over a no-SQL database or a caching engine like Redis is due to the (relatively) small amount of data we have to deal with (it is in the thousands). In other words, we are able to get

very fast response query responses by just sticking with MariaDB. Being a relational database, it also allowed us to directly map objects to tables.

In order to follow essential SQL-based database design principles, we used object role modeling. This allowed us to split each JSON in the file as individual objects to allow for a relational database. As a result, we have the following objects:



We now explain each object in more detail:

2.1 Airport

A table/model containing all airports. Its primary key is `airport_code`. It also has another property which is `airport_name`.

2.2 Carrier

A table/model containing all carriers. Its primary key is `carrier_code`. It also has another property which is `carrier_name`.

2.3 CarriersFromAirport

A table/model that stores which carriers are in what airports. For instance, 'KLM' and 'AA' are carriers in the airport 'LAX'. In this table, there are no uniqueness constraints.

2.4 Statistics tables

This table/model serves to identify for which combination of `airport_code`, `carrier_name`, `month`, and `year` we have a statistic for. This statistic row

then has an `id` primary key, which is used as a foreign key in the tables FlightStatistic, MinutesDelayedStatistic, and NumOfDelaysStatistic. This would then allow us to search for the `statistics_id` in the child tables to identify the specific kind of statistic we are looking for.

3 Technology stack

To implement this project, we used PHP as the programming language, Laravel as the web framework, and MariaDB as the database. We use an Apache server to host our web app based on the XAMPP stack.

We decided PHP to be the programming language as one of the team members had experience in it. Apart from that, PHP is an extremely popular web programming language, which meant that we were able to obtain plenty of documentation and resources for it.

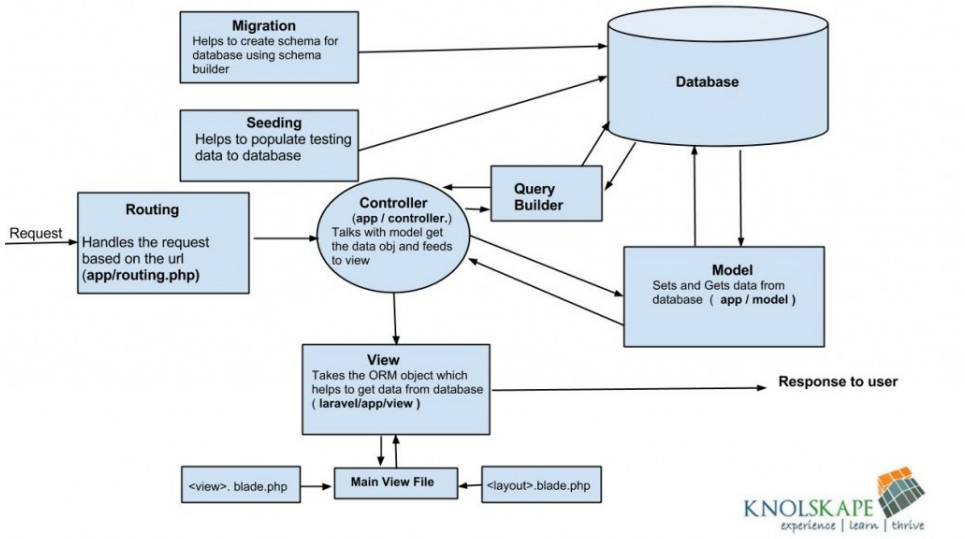
Following this, we based our project on the Laravel framework (version 5.8). The reason we went for a framework in the first place was because the team wanted to expose themselves to a web framework since no one had experience working with one previously. We chose Laravel because it is a very popular and renowned framework among the web development industry.

As for MariaDB, please refer to the Object Models section of this document.

4 Backend High-Level Architecture

* For details on the exact specifications of each endpoint, refer to our requirements document.

Our web app is based on Laravel's architecture. Because of this, we will discuss our app's structure alongside Laravel's architecture.



To begin with, Laravel is based on the Model-View-Controller design pattern. In our app's case, the Model is the set of all object models we have discussed previously. These are found in the `app` directory. The view is split in two:

1. Views for API responses: These are simply strings which are either in JSON or CSV format. In this case, it can be rather ambiguous to define it as a View (since the print is just a string).
2. Views for the web app: These are based on HTML, Javascript, and jQuery. These are for the front-end implementation of this project. These views are found in the `resources/views`

Before discussing Controllers further, we will mention briefly mention how routing works in our app. All our routes are found in the `app/routes/web.php` directory. These are according to the specifications given in the requirements document. Once a route is identified, we delegate the request to a controller.

Once a request has been delegated to a Controller, the controller takes full charge. These controllers are found in the `Http/Controllers` directory. We have two kinds of Controllers.

1. PagesController: This is a controller which takes care of all routing that is done to a Views page. It gathers all the data a View might need to build itself. It then returns a view as Response.
2. All other controllers are API controllers. These get all the resources we need (from the models) in order to serve the request and return a Response object containing a string body (a JSON or CSV). For API controllers, we do not technically have views. Laravel recognizes it is a Response object, and automatically prints the content string to the screen.

The last thing to discuss is our database migrations, which can be found in the `database/migrations` directory. Here we have implemented migrations for each of our tables in order to produce the tables as specified by the Object models section. We have implemented each according to the specification of Laravel's migrations: implementing an `up()‘` method which adds the data we want to the database and `down()` which reverts any changes we might want to do after running `up()`.