Junior Fullstack Engineer

Technical assessment

Name: Bernardo Filipe Martins Fragoso

Email: bernardo_fragoso91@hotmail.com

Github repository: https://github.com/BernardoFMF/indie-campers-challenge

August 2022

1. Important note

In the readme file found in the github repository you can find instructions on how to run the server and how to find documentation of the API.

2. Layers

My solution consists in the following layers:

- **Route** – Each route consists in the pair of path and http method and matches the pair to the correspondent controller. Also defines the middleware needed, which in this case is only a middleware to validate and parse requests.
- **Controller** – The controller will extract the necessary data from the request and will execute the logic of the route it corresponds. In the controller it is not necessary to verify the existence of parameters nor if they follow the convention defined, this is because of the middleware that was mentioned before.
- **Service** – The service will execute the connection to the database and do the different operations, which in this assessment consists of only fetching data.

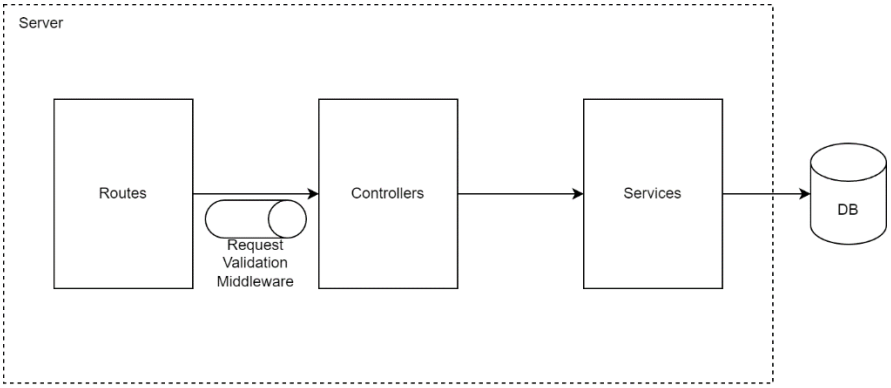

Figure 1 API Architecture

2.1. Modules

For each entity that will need its own routes a new module will be created for it. Each module will have the following files and organization:

- **Route** – This file defines all endpoints and matches them with the correspondent controller.
- **Controller** – Extracts data and executes the business logic, calling service functions to obtain data and setting the result in the response.
- **Service** – Connects the server to the database, making the necessary queries and mapping the results.
- **Schema** – Defines the schema which the request needs to follow, forcing it to have the necessary data in the path parameters, query parameters and body. Besides confirming the request, it also parses data to its correct type, for example parsing the id in a path parameter to a numeric value.
- **Model** – Contains the definition of the entity will all its properties.

2.2. Middleware

- **Middleware for validating and parsing requests** – This middleware guarantees that the request will always follow a predefined schema, where we define what properties the requests body, path and query should have. If it doesn't have a required property an error will be thrown and caught by the error middleware. This middleware also parses data to the correct type, for example an id path parameter is by default a string, but we can parse it to become a number. The schemas are defined using a schema validation package called Zod.
- **Middleware for handling errors** – This middleware provides a well-defined structure for how errors are supposed to be treated, enabling other components like controllers and services to throw a custom error with a status code and a message, which the middleware extracts and sets the status and message to the response. To make it so that errors are piped to this middleware, another component is needed, which is a wrapper on the controllers that calls the next middleware (in this case the error middleware) whenever there is an exception.

3. Data model

From the prompt, the following entities can be extracted:

- **Route** [1] – Represents a route. This entity has the following fields:
  - Identifier – Unique identifier.
  - Name – Represents the name of the route.
  - Description – Represents an in-depth explanation of the route.
  - Start Location – The initial location for the route, represented by a string. [2]
  - End Location – The end location for the route, represented by a string. [2]
- **Landmark** – Represents interesting points that will appear through the routes. As landmarks are not specific to a single route, this must be a separate table. This entity has the following fields:
  - Identifier – Unique identifier.
  - Name – Name of the landmark.
  - Description – Description of the landmark.
  - Longitude – Numeric value standing for the longitude in which the landmark is located (part of the extra functionality of the second phase). [3]
  - Latitude – Numeric value standing for the latitude in which the landmark is located (part of the extra functionality of the second phase). [3]
- **Route Landmark** – Represents the relation between Route and Landmark. Besides the Route and Landmark identifiers, it also has the following field:
  - Highlight – Defines whether a Landmark is a highlight for that Route.

Having identified what entities will exist, I can formulate the ER model:

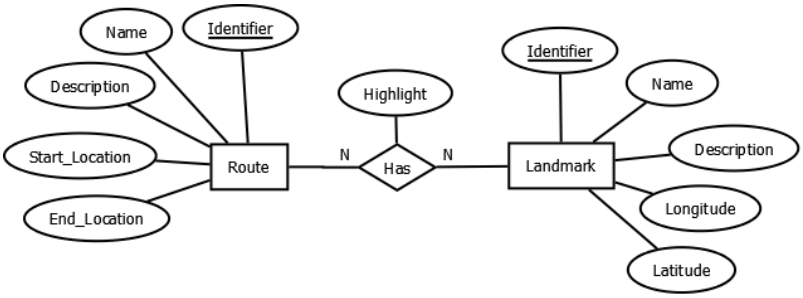

Figure 2 ER Model

Additional remarks:

1. It is assumed that there can be several routes with the same start and end locations, each having their respective landmarks.
2. The start location and end location of a route are defined as strings. An alternative would be to abstract these values, creating a table, manually adding predefined values and add these values as foreign keys in the Route table.
3. The longitude and latitude will have to be manually verified and inserted, since in this assessment there are no external APIs being used that could be utilized to fetch these values.

3.1. Database remarks

The database I have chosen to use is PostgreSQL. I've chosen an SQL database instead of NoSQL because the data model defined is already well-structured and having the ability to create database functions works very well in this case, for example to fetch the closest landmark which works around a custom ordering on the database side instead of server side.

In the server I have decided to use raw SQL with prepared statements instead of an ORM due to the inability of properly creating a custom order when querying, and since a common approach available in some ORMs is to execute raw SQL as well, there would be no advantage to use it in this case.

Due to the extra functionality in the second phase, there is a function in the database that calculates the distance between two geographic points (latitude and longitude). This is used when querying the closest landmark given a single geographic point by defining a custom order using said function and limiting the returned rows to a single row. This was a design choice made to avoid fetching unnecessary data and iterating it on the server.

4. API Endpoints

- **GET /api/routes/search** – This endpoint takes as query parameters the start location and end location and returns a list of routes that match these locations. Also includes a list of highlights, which are some of the chosen landmarks that the route has.
- **GET /api/routes/:id** – This endpoint fetches the route that has the id sent in the path parameter, including a list of all landmarks the route has, with a field that allows to differentiate which ones are highlights of the route.
- **GET /api/landmarks/geo** – This endpoint takes as query parameters the latitude and longitude and returns the closest landmark. This endpoint is independent from route since I've interpreted the functionality as wanting the closest landmark globally and not the closest landmark in a given route.