# **Uma imagem com texto, ClipArt Descrição gerada automaticamente**

**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. Layers**

My solution consists in the following layers:

* **Route** – Each route consists of the pair of path and http method and matches the pair to the correspondent controller.
* **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 that is already implemented as a middleware that uses predefined schemas that the request needs to follow.
* **Service** – The service will execute the connection to the database and do the different operations, from fetching data to updating rows.

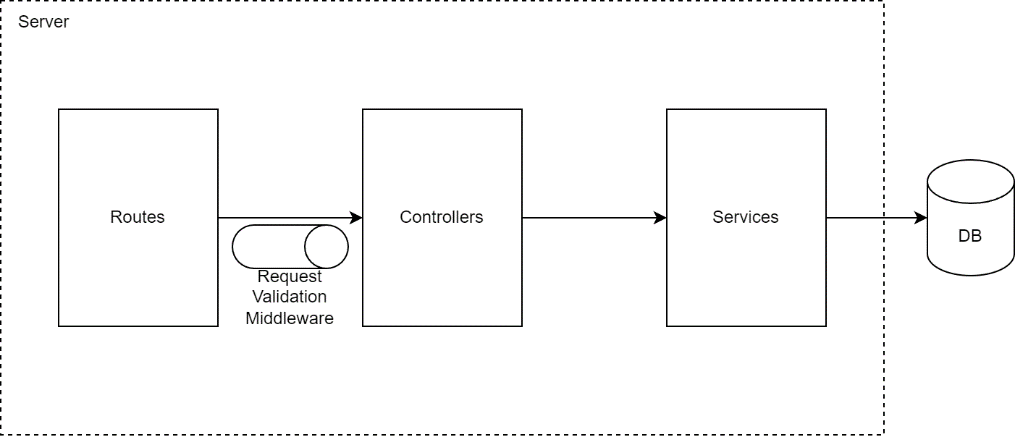


Figure 1 API Architecture

## **1.1 Modules**

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

* **Routes** – This file defines all endpoints and matches them with the correspondent controller.
* **Controllers** – Extracts data and executes the business logic, calling functions to obtain data found in the Service file and setting the result in the response.
* **Services** – Connects the server to the database, making the necessary queries and mapping the results.
* **Schemas** – 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.
* **Models** – Contains the definition of the entity will all its properties.

# **2. Data model**

From the prompt, the following entities can be extracted:

* **Route** [[1]](#Note1) – 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]](#Note2)
  + End Location – The end location for the route, represented by a string. [[2]](#Note2)
* **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]](#Note3)
  + Latitude – Numeric value standing for the latitude in which the landmark is located (part of the extra functionality of the second phase). [[3]](#Note3)
* **Route Landmark** – Represents the junction 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.

**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.

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

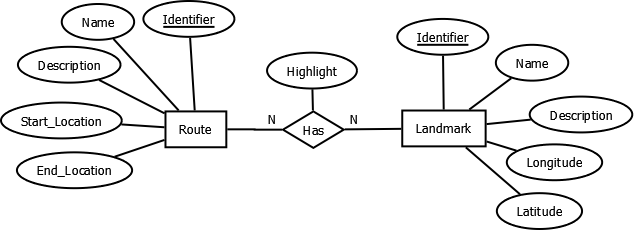


Figure 2 ER Model

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.