Wildebeests and Where to Find Them

How to run the project:

There is a script, "init.bat" that launches both packages on different ports for Windows machines. On other machines: "cd ./frontend" -> "npm start". Then "cd ../backend" -> "npm start". View the frontend at "http://localhost:3000".

Tests

backend

Set an environment variable "mode" to "TEST" (all caps). Make sure the project isn't running. In powershell, the command is: '$env:mode = "TEST"'.

In your cli: "cd ./frontend" -> "npm test". "cd ../backend" -> "npm test".

Docs

"cd ./frontend" -> "npm run docs". "cd ../backend" -> "npm run docs".

Code structure

Broadly speaking, there are two folders, frontend and backend. Folder "frontend" has all the frontend code. Folder "backend" has the backend code. Each end is it's own node package to make package management easier and to prevent "create-react-app" from potentially modifying files in the backend, and vice versa.

Architecture

Because speed of coding, not scalability or computational efficiency is needed, I can most rapidly prototype my backend in Node.js using loopback. This will allow me to create CRUD actions for the wildebeests most rapidly and implement the retrieval of the wildebeests.

I will use "create-react-app" for the frontend, as it packages many useful technologies together. I will use React as the frontend language with its built-in server. I will use Jest for testing. I will not be using Redux as the app does not have to scale, and slows development time. In addition, the frontend state is very simple.

Design

Backend

The backend is a strongloop app which will have CRUD scaffold for the models. I will create a custom route for "getYourWildebeest". "getWildebeets" is a special case of the "GET" REST request, so I will not have to implement it.

Models

Because a wildebeest can carry an unlimited number of Oxpeckers, and the rest of the specification does not deal with Oxpeckers, I will not be modeling Oxpeckers.

There are no relations between wildebeests, so the model is simply an object with fields not referencing each other.

Wildebeests have a location, a name, and a direction.

The location will be two fields of type "string", "latitude" and "longitude". I am not using floating point numbers because floating points can lose precision, especially with the specificity we are using. I will convert them as needed.

Their name will be a field of type "string" called "name".

Their direction will be a field of type "string" called "direction" that will hold the name of a destination, either "Kenya" or "Tanzania". This makes it easy to add more destinations later versus a boolean data type. It will return the same error that no wildebeest are available if an incorrect destination is entered.

Methods

getWildebeests()

This will be scaffolded for me.

getYourWildebeest(latitude, longitude, destination) -> Name of wildebeest that will pick you up.

This looks like a variant of the k-nearest-neighbor-search problem (https://en.wikipedia.org/wiki/Nearest_neighbor_search), for k=1. Because there are only (up to) 10 beests, I will take the naïve approach, find the euclidean distance to each beest and keep track of the best so far and return that. I could spend more time and implement an optimized approach off of NPM, but I think you also want to see me implement an algorithm here.

Testing

I will write unit tests using supertest and mocha (https://github.com/visionmedia/supertest, https://mochajs.org/).

getWildebeests()

This is pretty straightforward. I will test if any wildebeest exist or is an empty array. If a wildebeest exists I will check if it has the right fields of the correct type and aren't null.

getYourWildebeest()

I will test that a given GPS coordinate and destination will return a wildebeest. I will test that the wildebeest it returns is the closest from a sample set of ten. I would test more than 10 wildebeest to make sure it can analyze large amounts of data but that isn't needed. I will make sure it gives an error message if there are no Wildebeest going in your direction.

Generating data

I will create a boot-script in loopback that will create the 10 wildebeest and randomize them every 3 seconds in a non-terminating loop.

frontend

I will use "create-react-app" with Jest. This will be a single page application, with the ui.

Because react-map-gl isn't compatible with "create-react-app" I will be using "react-google-maps" because it is compatible and has lots of documentation on StackOverflow. I'll encapsulate it in a React component and only expose a way to pass the wildebeest models.

I'll make a simple form on the left side that submits a request using the fetch api.

I'll set up jest snapshotting. I can check that the form component submits all the data as a proper JSON object.

I made a simple unit test for the form.

For UI design, get inspiration from: https://www.uber.design/#about

I chose a thin-line font, san-serif to reflect roughly what Uber's font looks like. I took the stylizing of the logo by how "uberX" has Uber in lowercase, so I de-emphasized "zoober" and emphasized the "X". I used the black and white style familiar from the app. I did not use any borders like the app does, and the inputs change background color like the location selection feature does.

I took the layout from Uber's fare estimator, with the location entry on the left and a huge map to the right. I used placeholders instead of labels like in the ui.

I used flex-box, so the entire ui is reasonably responsive. This would allow for easy embedding in frames.

Considered auto-fitting all markers as they are added to the map, too much work to implement.

Right clicking the map was the fastest to implement without having the user type in the values

themselves.

Google Maps isn't compatible with "enzyme" the shallow DOM renderer I'm using, so I didn't test that one. I'll be looking at the webpage for updates.