Problem 1 - Fix This Code! 💻

What 3 questions would you ask to be able to fix this code?

1. Do I understand what is happening? What are the input/outputs?

- What functionality do the imported packages provide?

`bluebird` is a third party promise library for easy handling of asynchronous code

`promise-throttle` limits the amount of promises given a certain amount of requests/time and implementing the Bluebird promise library

`sequelize` is an ORM tool for node and must be imported and initialized which is missing from this file (this file represents one model for a larger MVC pattern and it is possible that sequelize is initialized in another file and passed as a parameter when the model is called but for my purposes I have initialized the local db connection). The first part of the exported file uses the `sequelize` package to define a table 'geopositions' with the appropriate fields. To fully integrate the package I initialized a database connection and removed the function parameters `(sequelize, Datatypes)` changing each `Datatype` to the `Sequelize` field import type. In order to actually query the table in the options object you have to create a model name by setting the `sequelize.define` method to a variable.

The `classMethods` defined in the option object of the model format the information from the API call. Logging out the `[table].options.classMethods`:

```
{ associate: [Function: associate], geocode: [AsyncFunction: geocode] }
```

The next big method is `classMethods.geocode()`. The query parameter is 'sanitized' into the correct format using regex (e.g. `/[^\x00-\x7F]/` is a negated set that makes sure no character from NULL to DEL is present). In the final regex the goal is to reduce any repeated whitespace to a single space so it should be: `...sanitizedAddress.replace(/\s/, '');`

To test this model I have been using the following function call from a separate file:

```
const geopositions = require('./geopositions_error');
const query = 'Testing QUERY 123tESt';
let result = geopositions.options.classMethods.geocode(query);
console.log('result:', result);
```

Eventually, what is returned is a new address object ready to be inserted as an entry into the `geopositions` table with all of the fields generated from the result of the Google API call.

2. When a bug is found, what is its impact?

> The 80/20 rule states: '80% of the bugs are found in 20% of the code"

- Initial flags: asynchronous code!
    - the `geoCode` class method needs to be declared an asynchronous function in order to await the result of the API call. Each `await` is a promise unable to resolve unless preceded by `async` on its function declaration.
- Do the fields defined in the table match the keys in the returned object?
    - The table definition is missing the field of `premise` LINE 38

3. Are there any JavaScript language errors related to syntax/lexical/operator placement?

- Most notable are the multiple improper uses of the comparison operator versus the assignment operator:

```
if ((result.status = 'OVER_QUERY_LIMIT')) {
  throw new Error('OVER_QUERY_LIMIT');
}

// this is a comparison statement - change to:

if (result.status === 'OVER_QUERY_LIMIT') {
  throw new Error('OVER_QUERY_LIMIT');
}

// also:
if (result.status != 'OK')
// should be:
if (result.status !== 'OK')
```

and also in the `address_components` assignment forEach():

```
 if (component.types.some(type => (type = 'premise'))) {
    address.premise = component.long_name;

    // should be for each field:
  type => (type === 'premise')
```

- The use of `.find()` on line 115 is a bit strange because `.find()` takes a callback function to find the position of a matched condition in an array. When `allowPartial` is set to false this cached query is skipped over straight to the Api call. Since there is only a `findAll()` method in sequelize I am guessing it has something to do with not being able to properly refer to the defined table and make the query. Change `this` to the table name `geopositions`. It looks as though `.find()` is deprecated in the sequelize package.

```
// SELECT * FROM geopositons WHERE query = coalescedAddress;
let cached = await geopositions.findAll({
  where: {
    query: coalescedAddress
```

```
  }
});
```

- Also: in these final instances the `this` is referring to the `classMethods` object which does not contain a `find()` or `create()` method (es5 function declaration of the methods) `<rejected> TypeError: this.find is not a function`. I have been able to get around this error by defining the Model in the initial `sequelize.define` call and using this Model as the query target for both `Model.find()` and `Model.create()`.

See the Sequealize Documentation on Model Definition and Querying

What assumptions did you make to complete?

- I assume that in the `../config/google-apis` file not provided the api is initialized with url and key and returns the API call url to which the coalescedAddress is concatenated to make the `geocodeAsync` call. This returns a result that will be checked and parsed into the address returned by the file.

- That `cached.get()` actually returns a value. I am only familiar with the `.get()` where it accepts a key or string as a parameter

- That this would mostly be a JavaScript syntax solving problem! I must admit I have used Knex as my main ORM for node/express and I have spent much of my time on this problem troubleshooting various versioning issues with the sequelize package in order to establish a local dev postgres database connection. Once established I am able to see that I am indeed querying the database but without the proper formatting of the gmAPI connection I will be unable to refactor the file further. This was an excellent exercise in troubleshooting errors with the implementation of a new package. My method of debugging involved working from the file 'top-down' in order to encounter errors in a rational order. This was helped by being able to initialize a db connection locally and, with my logging.js, start to drill into the returned object from the file (`geopositions.options.classMethods...`). The biggest 'trade-off' I made was to export the file as the created table/model name for import throughout the application. This was done to simplify the file and to ensure I could have a defined model name returned from `sequelize.define` method to use as my query tablename. My final sql log output shows that I was able to create the table on the PSQL database and formulate a formatted query. All I need now is a sample query and the Google Api key!

```
Executing: SELECT "id", "query", "formatted_address" AS
"formattedAddress", "lat", "lng", "premise", "subpremise", ... "modified"
AS "updatedAt" FROM "geopositions" AS "Geoposition" WHERE
"Geoposition"."query" = 'TESTINGQUERY 123TEST';
```

```
'use strict';

const Promise = require('bluebird');
const gmAPI = require('../config/google-apis');
const PromiseThrottle = require('promise-throttle');
```

```
const Sequelize = require('sequelize');

// initialize the database connection to the local dev db
const sequelize = new Sequelize(
  'postgres://artiefischer@localhost:5432/leedemo'
);

const geocodeThrottle = new PromiseThrottle({
  requestsPerSecond: 40,
  promiseImplementation: Promise
});

const geopositions = sequelize.define(
  'Geoposition',
  {
    id: {
      type: Sequelize.INTEGER,
      field: 'id',
      primaryKey: true
    },

    query: {
      type: Sequelize.STRING
    },
    formattedAddress: {
      type: Sequelize.STRING,
      field: 'formatted_address'
    },
    lat: {
      type: Sequelize.DECIMAL
    },
    lng: {
      type: Sequelize.DECIMAL
    },
    premise: {
      type: Sequelize.STRING
    },
    subpremise: {
      type: Sequelize.STRING
    },
    streetNumber: {
      type: Sequelize.STRING,
      field: 'street_number'
    },
    route: {
      type: Sequelize.STRING
    },
    locality: {
      type: Sequelize.STRING
    },
    adminAreaLevel2: {
      type: Sequelize.STRING,
      field: 'admin_area_level_2'
    },
```

```
      adminAreaLevel1: {
        type: Sequelize.STRING,
        field: 'admin_area_level_1'
      },
      postalCode: {
        type: Sequelize.STRING,
        field: 'postal_code'
      },
      viewportN: {
        type: Sequelize.FLOAT,
        field: 'viewport_n'
      },
      viewportS: {
        type: Sequelize.FLOAT,
        field: 'viewport_s'
      },
      viewportW: {
        type: Sequelize.FLOAT,
        field: 'viewport_w'
      },
      viewportE: {
        type: Sequelize.FLOAT,
        field: 'viewport_e'
      },

      createdAt: {
        type: Sequelize.DATE,
        field: 'created'
      },
      updatedAt: {
        type: Sequelize.DATE,
        field: 'modified'
      }
    },
    {
      tableName: 'geopositions',
      classMethods: {
        associate: db => {},
        geocode: async function(query, allowPartial) {
          //Default allowPartial to true
          if (!allowPartial) {
            allowPartial = true;
          }

          //Uppercase, remove invalid characters, coalesce repeated spaces
  into a single space
          const upperAddress = query.toUpperCase();
          const sanitizedAddress = upperAddress.replace(/[^\x00-\x7F]/, '');
          const coalescedAddress = sanitizedAddress.replace(/\s/, '');

          //Check if address is empty, if so return error
          if (/^\s+$/.test(coalescedAddress)) {
            return {
              status: false,
```

```
            statusCode: 'EMPTY_ADDRESS'
          };
        }

        if (allowPartial) {
          let cached = await geopositions.findAll({
            where: {
              query: coalescedAddress
            }
          });

          if (cached) {
            cached = cached.get();
            cached.status = true;
            cached.statusCode = 'CACHED';
            return cached;
          }
        }

        let result = await geocodeThrottle.add(
          gmAPI.geocodeAsync.bind(gmAPI, {
            address: coalescedAddress
          })
        );

        //If rate limit exceeded, throw error to force retry
        if (result.status === 'OVER_QUERY_LIMIT') {
          throw new Error('OVER_QUERY_LIMIT');
        }

        if (result.status !== 'OK') {
          return {
            status: false,
            statusCode: result.status
          };
        }

        if (allowPartial) {
          result = result.results[0];
        } else {
          //Filter result to disallow partial matches
          result = result.results.filter(row => {
            if (!row.partial_match) {
              return true;
            }
          });

          //If no results, return error
          if (!result) {
            return {
              status: false,
              statusCode: 'NO_EXACT'
            };
          }
```

```
        }

        const formattedAddress = result.formatted_address;
        const lat = result.geometry.location.lat;
        const lng = result.geometry.location.lng;

        //Generate model properties
        const address = {
          query: coalescedAddress,
          formattedAddress: formattedAddress,
          latitude: lat,
          longitude: lng,
          premise: '',
          subpremise: '',
          streetNumber: '',
          route: '',
          locality: '',
          adminAreaLevel2: '',
          adminAreaLevel1: '',
          postalCode: ''
        };

        //address_components ~ {types: string[], long_name: string}[]
        //Find relevant properties in address_components and assign to
        result.address_components.forEach(component => {
          if (component.types.some(type => type === 'premise')) {
            address.premise = component.long_name;
          } else if (component.types.some(type => type === 'subpremise'))
{
            address.subpremise = component.long_name;
          } else if (component.types.some(type => type ===
'street_number')) {
            address.streetNumber = component.long_name;
          } else if (component.types.some(type => type === 'route')) {
            address.route = component.long_name;
          } else if (component.types.some(type => type === 'locality')) {
            address.locality = component.long_name;
          } else if (
            component.types.some(type => type ===
'administrative_area_level_2')
          ) {
            address.adminAreaLevel2 = component.long_name;
          } else if (
            component.types.some(type => type ===
'administrative_area_level_1')
          ) {
            address.adminAreaLevel1 = component.short_name;
          } else if (component.types.some(type => type === 'postal_code'))
{
            address.postalCode = component.long_name;
          }
        });

        // add google api coordinates to the address object
```

```
          address.viewportN = result.geometry.viewport.northeast.lat;
          address.viewportE = result.geometry.viewport.northeast.lng;
          address.viewportS = result.geometry.viewport.southwest.lat;
          address.viewportW = result.geometry.viewport.southwest.lng;

          // INSERT INTO TABLE geopositions ... the new address object
          await geopositions.create(address);
          return address;
        }
      }
    }
  );

  module.exports = geopositions;
```