Quote Service

Overview

It's time to write an API that serves an assortment of quotes from famous people! The quotes will be scraped from a mock website, http://quotes.toscrape.com.

Your API will scrape all of the quotes from the above website and serve them via the following route:

```
HTTP method | Query                   | Response data
------------+-------------------------+---------------------------------
GET         | /quotes                 | all available quotes
GET         | /quotes?author=:author | all quotes by a particular author
GET         | /quotes?tag=:tag        | all quotes with a specific tag
```

Provide an additional route to handle requests for author information, also extracted in full from the quotes website:

```
HTTP method | Query                | Response data
------------+----------------------+-----------------------------------------
GET         | /authors             | biographical information for all authors
GET         | /authors?name=:name | biographical information for one author
```

All parameter strings will be matched exactly--there is no need to lowercase or otherwise manipulate these strings.

Response format

All requests should respond with status code 200 on success. Responses will consist of the following JSON structure:

```
{
  "data": [...]
}
```

In this structure, the actual response data (objects with properties described below) will be contained inside of the `data` array (which may be empty in the case of no content).

Quotes response JSON

For requests to one of the `/quote` routes, each member of the `data` array should be an object containing three properties: `text`, the string text of the quote; `author`, the string author of the quote; and `tags`, an array of strings representing all of the tags associated with the quote (if any tags exist). Here's an example response for a query to the `/quote` route (`/quotes?tag=understanding`, specifically) containing one quote:

```
{
  "data": [
    {
```

```
      "author": "Albert Einstein",
      "text": "“Any fool can know. The point is to understand.”",
      "tags": [
        "knowledge",
        "learning",
        "understanding",
        "wisdom"
      ]
    }
  ]
}
```

Authors response body JSON

The other route group consists of author biographical information which can be obtained for all authors via `/authors`, or for a specific author via `/authors?name=:name`.

For requests to an `/authors` route, each member of the `data` array should be an object containing four properties: `name`, the string name of the author; `biography`, the biography of the author; `birthdate` the birth date of the author; `location`, the birth location of the author.

Here's a potential response for a GET request to the route `/authors?name=André%20Gide`:

```
{
  "data": [
    {
      "name": "André Gide\n    ",
      "biography": "\n        André Paul Guillaume Gide was a French author and winner of the Nobel Prize in literature in 1947. Gide's career ranged from its beginnings in the symbolist movement, to the advent of anticolonialism between the two World Wars.Known for his fiction as well as his autobiographical works, Gide exposes to public view the conflict and eventual reconciliation between the two sides of his personality, split apart by a straight-laced education and a narrow social moralism. Gide's work can be seen as an investigation of freedom and empowerment in the face of moralistic and puritan constraints, and gravitates around his continuous effort to achieve intellectual honesty. His self-exploratory texts reflect his search of how to be fully oneself, even to the point of owning one's sexual nature, without at the same time betraying one's values. His political activity is informed by the same ethos, as suggested by his repudiation of communism after his 1936 voyage to the USSR.    \n    ",
      "birthdate": "November 22, 1869",
      "location": "in Paris, France"
    }
  ]
}
```

Based on the above response, it's clear that some bodies will be quite large, which can make tracking down bugs and viewing diffs tedious. Additionally, the response's whitespace could use trimming. As a consequence, let's simplify responses for the purposes of this challenge.

Please call `.trim()` on every response field to remove leading and trailing whitespace. Additionally, call `.slice(0, 50)` on the `biography` and `text` fields (depending on whether we're returning an author biography or quote, respectively) to return only the first 50 characters. Specifically, use `.trim().slice(0, 50)` in that order for these two fields.

With this change, the above example response body becomes

```
{
```

```
  "data": [
    {
      "name": "André Gide",
      "biography": "André Paul Guillaume Gide was a French author and ",
      "birthdate": "November 22, 1869",
      "location": "in Paris, France"
    }
  ]
}
```

which is a bit more manageable.

Scraping

This section outlines guidelines for the data scraping component.

All data will be extracted from quotes.toscrape.com. Note that this site uses an unsecure `http` protocol rather than `https`. There should be 100 quotes and 50 author biographies, all of which should be made available by your API. A requests library, axios, is available for retrieving the HTML content.

HTML parsing modules which are available in your environment include:

- cheerio
- jsdom

This challenge does not presuppose familiarity with these libraries. Feel free to research and consult documentation or use any other libraries that are available in the environment.

Testing with chai-http

Here are a few examples to disambiguate the above requirements and offer some concrete goals.

The chai-http module will be used for testing. chai will import and run your app directly from its `module.exports` property, making calls to routes and evaluating the responses.

Example 1

```
// send a GET request to /quotes?tag=obvious
const req = "/quotes";
const res = await chai.request(app)
                      .get(req)
                      .query({tag: "obvious"});
const expected = {
  "data": [
    {
      author: 'Steve Martin',
      text: '“A day without sunshine is like, you know, night.”'
      keywords: ['humor', 'obvious', 'simile']
    }
  ]
};
```

```
expect(res.status).to.equal(200);
expect(res).to.be.json;
expect(res.body.data).to.be.a("array");
expect(res.body.data.length).to.equal(expected.data.length);
expect(validateResponse(res.body.data, expected.data)).to.be.true;
```

In this example, a GET request is made to the `/quotes?tag=obvious` route, which responds with all quotes which contain the keyword `"obvious"`. The response contains the HTTP status code 200/success, matches the correct JSON structure and has a `.data` property with length 1.

Lastly, `validateResponse` is invoked to verify that the response array matches `expected` satisfactorily. Primarily, order is ignored. Check `/test/helpers/validate.js` to see how this works.

Example 2

Here's an example of a 200 response for a query to `/quotes?tag=foo`, which still returns JSON albeit with an empty `data` array:

```
const req = "/quotes";
const res = await chai.request(app)
                      .get(req)
                      .query({tag: "foo"});
expect(res).to.have.status(200);
expect(res).to.be.json;
expect(res.body).to.be.a("object");
expect(res.body.data).to.be.a("array");
expect(res.body.data.length).to.equal(0);
```

Example 3

Here's a request to `/authors?name=C.S.%20Lewis` which retrieves biographical information for C.S. Lewis.

```
const req = "/authors";
const res = await chai.request(app).get(req).query({name: "C.S. Lewis"});
const expected = {
  data: [
    {
       name: 'C.S. Lewis',
       biography: 'CLIVE STAPLES LEWIS (1898–1963) was one of the int',
       birthdate: 'November 29, 1898',
       location: 'in Belfast, Ireland'
    }
  ]
};

expect(res.status).to.equal(200);
expect(res).to.be.json;
expect(res.body.data).to.be.a("array");
expect(res.body.data.length).to.equal(expected.data.length);
expect(validateResponse(res.body.data, expected.data)).to.be.true;
```

For additional examples, see the provided test cases. You'll likely need to write your own tests; a modifiable example file which will not be included with your final submission is located in `test/candidate.test.js`.