CSE183 Fall 2020 - Assignment 7

In this assignment you will write a Node.js & Express RESTful E-Mail API and associated tests to demonstrate it works as required.

This assignment is worth 7% of your final grade.

Submissions are due NO LATER than 23:59, Saturday November 21, 2020 (> 1 week)

Installation

See instructions in Assignment 3 and ensure you are running the current LTS version of Node.js.

Setup

Download the starter code archive from Canvas and expand into an empty folder. I recommend, if you have not already done so, creating a folder for the class and individual folders beneath that for each assignment.

The starter code archive contains two files that you will modify:

```
src/app.js
api/openapi.yaml
```

And some files that you should ***not*** modify by hand though your code for the Stretch Requirement will modify the data files programmatically:

```
data/inbox.json
data/trash.json
data/sent.json
src/server.js
test/invalid.url.test.js
package.json
.eslintrc.js
```

To setup the development environment, ***navigate to the folder where you extracted the starter code*** and run the following command:

```
$ npm install
```

This will take some time to execute as `npm` downloads and installs all the packages required to build the assignment.

To start the dev server, run the following command:

```
$ npm start
```

To execute tests run the following command:

```
$ npm test
```

To run the linter against your code, run the following command:

```
$ npm run lint
```


Background

E-Mails in this system have the following properties:

- `id` Universally Unique Identifier https://www.uuidtools.com/what-is-uuid
- `to-name` String
- `to-email` E-Mail Address
- `from-name` String
- `from-email` E-Mail Address
- `subject` String
- `received` ISO Format Date String
- `content` String

For example, the JSON representation of an E-Mail looks like this:

```
{
  "id": "591b428e-1b99-4a56-b653-dab17210b3b7",
  "to-name": "CSE183 Student",
  "to-email": "cse183-student@ucsc.edu",
  "from-name": "Cherye O'Loughane",
  "from-email": "coloughane0@nymag.com",
  "subject": "Broderskab (Brotherhood)",
  "received": "2020-07-07T00:18:37Z",
  "content": "Duis aliquam convallis nunc."
}
```

You are provided with three existing mailboxes: inbox, sent, and trash but will need to create new ones if asked to by code calling your API.

Note that your API must be constrained by an OpenAPI version 3.0.3 schema. You should endeavor to make this schema as restrictive as possible.

For example, your schema should specify the acceptable format of the e-mail id property. If an id in any other form is presented, your system should reject it simply by performing the validation provided in the starter code.

When you start the server, use a browser to visit http://localhost:3010/v0/api-docs/ where you can manually test your API as demonstrated in class for the books example.

Initially, the only operation available will be:

Which will not work until you implement the necessary Express route.

Once that is working, you will need to add additional endpoints in the OpenAPI schema and additional Express routes to service them.

If you have no idea what any of those terms represent, view the recording of lecture 18 “Node.js & Express”, and study the handouts from the same lecture before consulting with the TAs and/or Instructor.

Requirements

Basic:

- Write an OpenAPI compliant E-Mail RESTful API with the following informally described endpoints:
  - **GET /v0/mail**

    Return all the emails in the system as an array of named mailboxes containing arrays of e-mails, one per mailbox. For example:

    ```
    [
      {"name": "inbox", "mail": [<e-mails in inbox>]},
      {"name": "sent", "mail": [<e-mails in sent>]},
      {"name": "trash", "mail": [<e-mails in trash>]},
      ...
    ]
    ```

    Note that all returned e-mails should have had their ‘content’ property removed.

  - **GET /v0/mail?mailbox={mailbox}**

    Return all the emails in a specified mailbox as an array containing one named mailbox containing an array of e-mails in the mailbox. Throw a 404 error if the mailbox is unknown. For example, if the query parameter “mailbox” was equal to ‘trash”, return:

    ```
    [ {"name": "trash", "mail": [<e-mails in trash>]} ]
    ```

    Note that the returned e-mails should have had their ‘content’ property removed.

  - **GET /v0/mail/{id}**

    Return the e-mail identified by id, including its ‘content’ property. Throw a 404 error if id does not identify a known e-mail.

  - **POST /v0/mail**

    Save a new email sent in the HTTP request body into the sent mailbox and return the newly created e-mail. The id property should be undefined on submission and set on return. Throw a 500 error if the submitted e-mail has an id attribute.

  - **PUT /v0/mail/{id}?mailbox={mailbox}**

    Move the e-mail identified by id into the named mailbox. If the mailbox does not exist, create it and move the identified e-mail into it. Throw a 404 error if id does not identify a known e-mail. Throw a 409 error if the named mailbox is ‘sent’ and the mail identified by id is not already in the sent mailbox.

Advanced:

- Write tests to demonstrate your implementation satisfies the Basic Requirements.
- Show your tests achieve 100% line, statement, branch, and function code coverage.
- Demonstrate zero lint errors or warnings in your code.

Stretch:

- Save mailbox modifications to disk.
- When the sever re-starts the modified mailboxes (and any new ones created) continue to be available.
- Write tests to demonstrate this works.
- Continue to show zero lint errors and 100% code coverage.

What steps should I take to tackle this?

You should base your implementation on the books example from class. Start by implementing the simple GET endpoints then move on to the more complex POST and PUT operations.

There is a huge amount of information available on OpenAPI 3.0.3 at https://swagger.io/specification/ and http://spec.openapis.org/oas/v3.0.3. Also consult the Petstore example at https://petstore3.swagger.io/.

How much code will I need to write?

Much of this implementation will be in the OpenAPI schema in `api/openapi.yaml`. if you find yourself writing "defensively" in your JavaScript to check for incorrect data sent by the caller, you should instead make your OpenAPI schema more robust such that it can be used to reject invalid submissions.

Assuming you OpenAPI schema is sufficiently robust, your solution is likely to contain somewhere in the region of 200 to 300 lines of code.

Grading scheme

The following aspects will be assessed:

1. (100%) **Does it work?**

    a. Basic (4 points)
    b. Advanced (2 points)
    c. Stretch (1 point)

2. (-100%) **Did you give credit where credit is due?**

    a. Your submission is found to contain code segments copied from on-line resources and you failed to give clear and unambiguous credit to the original author(s) in your source code (-100%). You will also be subject to the university academic misconduct procedure as stated in the class academic integrity policy.

    b. Your submission is determined to be a copy of a past or present student's submission (-100%)

    c. Your submission is found to contain code segments copied from on-line resources that you did give a clear an unambiguous credit to in your source code, but the copied code constitutes too significant a percentage of your submission:

        - < 80% copied code — No deduction
        - 33% to 66% copied code — (-50%)
        - > 66% copied code — (-100%)

What to submit

Run the following command to create the submission archive:

```
$ npm run zip
```

****** UPLOAD** `Assignment7.Submission.zip` **TO THE CANVAS ASSIGNMENT AND SUBMIT ******

