SOFTWARE PROJECT - FALL 2020

BACK END DOCUMENTATION

This document is part of the documentation for the Back End service developed for our project for Signant Health as part of the 2020 fall semester course.

The structure of the documentation will be as follows:

In this file we give a brief overview of the Back End structure and its features, and is to be considered a reference guide. The practical implementation and instructions are given in the Readme files of the repository. Currently, there are such files for each part of the Back End (web-server, database and SSH -plus our test GraphQL server-) along with a main file with API use instructions. Particular details of the implementation are commented in the source code.

BASIC TECHNOLOGIES AND STRUCTURE

For our Back End server, we are using **Node.js** runtime environment (v14.15) and the **Express.js** framework (v4.17). We are using as well **Knex** (official website: https://knexjs.org/) for SQL building and handling connections to our set-up MariaDB, using the **mysql2** client (source: https://www.npmjs.com/package/mysql2).

Organizational structure for Node projects does not have strict guidelines and many "suggestions" can be found online. We have built our own approach, based on the specific needs and size of this project.

The project is structured with an index.js and the package.json in the root folder and subfolders divided by purpose. These include the following folders:

- Config, comprising any configuration files for our application. Currently only has one file including app and database settings.
- Routes, where JavaScript files handling the routing are stored. We are using express router for this purpose. Only the routing is defined here, any additional handling is done on files stored on the next folder.
- Controllers, for JS controller files handling the business logic. Handling requests and responses, as well as Knex calls are done here.
- Tests, where we store our API tests ran with the software Postman. It could also include unit tests, if those are created.
- Docs, for documentation, such as this file.

In addition to this is, of course, the node_modules folder, self-explanatory. Also, we have deemed unnecessary to create our own repository (handling the Data Access Layer) and Knex takes care of this and it would be simply redundant.

API

The API serves a few different endpoints, with **/api/tests** being the basic route.

GET ENDPOINTS

GET requests can be performed on the following:

/all

returns all the test suites entered in the database.

/type/{type}

returns all tests done for a determined type of test. For example, all "unit" or "system" tests can be fetched at once.

/component/{componentName}

returns all tests done for a set component.

In addition, all these routes can be extended by adding **/startDate/endDate** at the end. These endpoints provide the ability to filter the results between certain dates, which will be very helpful for the tasks we are intended to run. The endDate parameter is optional, and if let not set it will default to the current date. The date format should be ISO Standard, that is **YYYY-MM-DD**.

Metadata can be queried at following endpoints:

/component

returns a list of all components currently in the database with tests assigned to them.

/type

returns a list of all available test types.

All GET requests return status code 200 (OK) if successful or 500 (Internal Server Error) plus the error message in JSON format if they are not.

POST ENDPOINTS

POST requests are currently supported with two purposes, conducting custom queries and inserting data into the database.

/api/tests/custom

This is the endpoint used when performing custom queries. Returned tests match a custom set of parameters specified in the request. The set must be expressed in JSON format, following this model:

```
{
    "columns": ["col_1", "col_2", "col_3"],
    "types": ["type_1", "type_2", "type_3"],
    "components": ["Component_1", "Component_2"],
    "startDate": "YYYY-MM-DD",
    "endDate": "YYYY-MM-DD"
}
```

Fields "types" and "components" are the only ones mandatory, all the rest being optional. The field "columns" defaults to: *component name, test type, start date, total fail and total pass*. Date handling works the same way as in the previous example, if only a start date is given, the end is considered the present day.

/api/tests

The basic endpoint, is used in order to insert data into the database. Performing an insertion can be done with either a test object or an array of them in the request body, in JSON form. With the test(s) passed an insertion is attempted in the database, returning status code 201 (Created) and the row id if successful, or code 500 (Internal Server Error) and the error message from the database otherwise.

The model object for making these POST requests is:

```
}

    startTime: "YYYY-MM-DD hh:mm:ss.sss"
    endTime: "YYYY-MM-DD hh:mm:ss.sss"
    hasPassed: "0/1",
    totalPass: "num",
    totalFail:   "num",
    testType: "alpha",
    testtrigger: "alpha",
    componentName: "alpha",
    documentation: "alpha"

}
```

All non-matching endpoints -that is, those which have not been implemented yet- return a 404 (Not found) status code in the response.

VALIDATION

Validation is performed at web-server level in order to avoid having the database perform unnecessary checks.

All fields are required when creating an object, except for Documentation.

When querying, the fields must be in the correct format, for example with the dates entered as string having the ISO standard form. This is checked using a Regular Expression. As of the time of this documentation, only the string format is checked, and checks for date validity are not implemented; they are left to the database (for example, the number of days on a certain month being correct).