Project #03

Due: Novembre 06th, 11:59PM

Aims

The aims of this project are as follows:

- To expose you to `expressjs`.
- To make you design and implement REST web services.

Requirements & Submission

Only one submission per team is required. Please refrain from making multiple submissions per group.

You must upload a zip-archive to brightspace, such that when it is unpacked, it will **create** a `prj3-sol` directory, such that typing `npm ci` within that directory followed by `npm run build` will build your project. Once built, typing `./dist/index.js` within that directory should be sufficient to run your project.

Specifically, the unpacked `prj3-sol` directory must directly contain `package.json`, `package-lock.json` and `tsconfig.json` files. It should not contain `node_modules` or `dist` directories.

You are being provided with `index.ts` and `main.ts` which provide the required command-line behavior.

What you specifically need to do is add code to the provided `library-ws.ts` file to implement the following web services rooted at some ***BASE*** url (defaults to `/api`):

Assuming that your server is set up to run with all URLs rooted at ***BASE***, it will need to implement the following web services:

1. Get-Book: `GET` *`BASE`*`/books/`*`ISBN`*

This service should set the `result` of the success envelope to the book having `isbn` *ISBN*.

- The service should return an error envelope with HTTP status set to `NOT_FOUND` if there is no book having `isbn` *ISBN*.

2. Add-Book: `PUT BASE/books`

This service should add a book with parameters specified in the request body and set the `result` of the success envelope to all the information associated with the newly added book.

- The success envelope should have its HTTP status set to `201 CREATED` and `Location` header set to the URL for the added book.
- The service should return an error envelope with HTTP status set to `BAD_REQUEST` if any of the parameters provided for the book in the request body are invalid.
- Note that when the book being added already exists, this service should work similarly to the corresponding method from your earlier project; i.e., the number of copies for the book is merely incremented by `nCopies`. Of course, the web service should return a HTTP status of `BAD_REQUEST` if there is any inconsistency between the information of the book being added and that which is stored for that `isbn`.

3. Find-Books: `GET BASE/books?search=SEARCH`

- This service should set the `result` of the success envelope to a list of all the books which match the *SEARCH* query parameter in the above URL.

- This web service should work like the `findBooks()` method from the previous projects.
- The list should be returned as empty if there are no matching books.

4. Checkout-Book: `PUT BASE/lendings`

- This service should checkout the book having `isbn` *ISBN* to the patron having `patronId` *PATRON_ID* specified in the JSON body of the request. The service should return with HTTP status `BAD_REQUEST` if the checkout encounters errors, with the error code set as per earlier projects.

5. Return-Book: `DELETE BASE/lendings`

- This service should return the book having `isbn` *ISBN* by the patron having `patronId` *PATRON_ID* specified in the JSON body of the request. The service should return with HTTP status `BAD_REQUEST` if the return encounters errors, with the error code set as per earlier projects.

6. Clear: `DELETE BASE`

- Clear out all data for the lending library.
- All request and response bodies for the web services must use type JSON. Responses must always be enclosed within success or error envelopes as per the typescript definitions in `response-envelope.ts`.
- All success envelopes should have an HTTP status code of 200 **OK** unless noted otherwise. If any of the above services encounters an error, then the service should return a suitable error envelope. Error responses should contain a suitable HTTP status (often `BAD REQUEST 400`). The `options.code` in any error messages should be set as in previous projects.

- The behavior of the program is illustrated in this automatically generated `annotated log`.

Provided Files

The `prj3-sol` directory contains a start for your project. It contains the following files:

src/library-ws.ts
A skeleton file for your project. You should be doing all your development in this file.

config.mjs
A configuration file provided on the command-line when starting the server. It contains information like the port # for the server, as well as specifying the certificates used to run the server.

You should not need to modify this file if you are happy with its contents.

response-envelope.ts
Typescript definitions for the response envelopes.

index.ts and main.ts
These files provide the complete command-line behavior which is required by your program. It imports the code generated from `library-ws.ts`. You should not need to modify this file.

tsconfig.json
A configuration file for typescript. You may modify this file if necessary.

README
A `README` file which must be submitted along with your project. It contains an initial header which you must complete (replace the dummy entries with your team member names, B-numbers and one email address at which you would like to receive project-related email). After the header you may include any content which you would like to read during the grading of your project.

A slightly modified solution to your previous project is available as an additional library in `lending-library`.

The `test` directory contains tests. The web services tests in `test/library-ws.ts` use supertest.

The `extras` directory contains the following files:

LOG file
A log file which illustrates the operation of the project. Note that this is a fake log file produced non-interactively by running the `do-cmd-log.mjs` script.

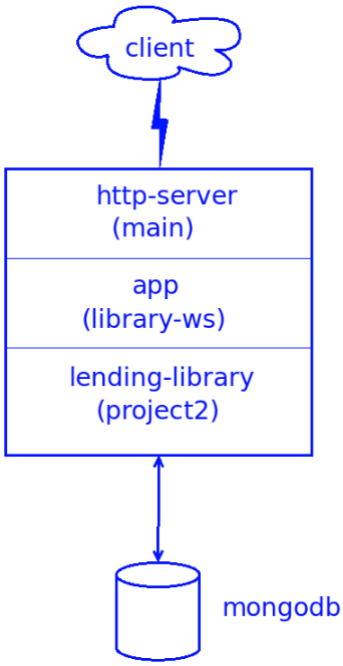
log-cmds.mjs
The commands used to generate the above `LOG`.

Changes from Project 2

A modified solution to **Project 2** - available in the `lending-library package` - provides an additional `getBook()` service in `lending-library.ts`.

Application Architecture

The overall application architecture is as shown in the following figure:



The expressjs `app` you will be implementing in `library-ws.ts` for this project will wrap the lending-library from `Project 2`, merely extracting parameters from the HTTP request and forwarding them to the lending-library. Finally, the provided main program in `main.js` starts a HTTPS server which embeds the `app`. It is this server which will be accessed by clients on the web.

Hints

The following points are worth noting:

- The JSON produced by the web services is not formatted. To format the output when accessing the web services using a command-line client like curl, pipe the output through a program like jq (pre-installed on your VM). If running directly within Chrome (not using some kind of REST client), use a Chrome extension like JSON Formatter.
- If you are using curl to test your project and also want to see the response headers on the terminal in addition to `jq` pretty-printed json, you cannot have curl output the headers to standard output as `jq` would choke on them. Hence they can be sent to standard error using `-D`

`/dev/stderr`. Unfortunately, a drawback of this solution is that it becomes impossible to redirect standard error.

To make development easy, it is useful to not have to manually restart the server after each source code change. Automated restarts can be achieved by using a program like nodemon. Once `nodemon` is installed as a development dependency within your `node_modules` directory, you can start your server from the `prj3-sol` directory using something like:

```
$ npx nodemon --watch ./dist ./dist/index.js config.mjs
```

and the server should be automatically restarted whenever you recompile your TypeScript code to the `dist` directory.
The above command assumes that all the library data has already been loaded into the database.
Note that you can also run the command `npm run watch` to automatically recompile your TypeScript code to the `dist` directory if you have set up your `package.json` as in previous projects.

As in the previous project, our error-handling convention is that a function indicates an error by returning a `Result` object having `isOk===false` with an `errors` property. We need to convert such errors into an HTTP error envelope as specified in the requirements. That is done by the provided `mapResultErrors()` utility function.

The code for each handler can be structured as follows:

```
async function(req: Express.Request, res: Express.Response)
{
       try {
         ... usually access a service on app.locals.model
       }
       catch(err) {
        const mapped = mapResultErrors(err);
        res.status(mapped.status).json(mapped);
       }
   }
```

Hence if an error occurs in an intermediate step of processing a request, it is sufficient to merely check for the error and `throw`:

```
...
const itermediateResult = ...;
if (!intermediateResult.isOk) {
   throw intermediateResult;
}
```

- with the `throw` caught by the surrounding `try-catch` block and converted into an `HTTP error`.
- All validation will be performed by the lending-library services available via `app.locals.model`. If a service returns errors, then those errors can be converted to an HTTP error envelope using code like the above.

If you get a `port in use` error when attempting to start your server, you probably have an instance of the server already running. Use `lsof` to discover the `pid` of that instance and then `kill` it:

```
$ lsof -i :2345
COMMAND     PID ...
node    1940488 ...
$ kill 1940488
```

The following steps are not prescriptive in that you may choose to ignore them as long as you meet all project requirements.

1. Read the project requirements thoroughly. Look at the `sample log` to make sure you understand the necessary behavior. Review the material covered in class including the `express-play` example as well as the `users ws slides` and `users-ws code`.
   Look at the expressjs docs. It is probably a good idea to have those docs open in a browser as you work; you will particularly be concerned with the req and res objects.
2. Your web service implementations will largely consist of accessing the web service parameters and forwarding those parameters onto an appropriate lending-library service. Hence make sure you understand

the difference between query parameters (`req.query`) and route parameters (`req.params`) and familiarize yourself with the services from `lending-library.ts`. An instance of those services is available within your Express `app` as `app.locals.model`.

3. Decide on how you will send HTTP requests to your server. You can do so using a command-line HTTP client like curl as in the provided `log`, an app like Postman, or a browser client like Talend API Tester or yarc.

4. Set up your `package.json` as per your previous project. Install the following **additional dependencies**:

   Runtime Dependencies

   `express`, `body-parser`, `http-status` and `cors`.

   Install the package containing the modified solution to `Project 2` from `lending-library-0.0.1.tgz` in a manner similar to the `cs544-{js,node}-utils` packages.

   Development Dependencies

   `supertest`, `@types/supertest`, `@types/express`, `@types/cors`.

Create your own certificate for your VM by running the provided `gen-localhost-cert.sh` script:

```
$ ./bin/gen-localhost-cert.sh
```

5. This will generate the certificate files in `~/tmp/localhost-certs`. Ignore the characters output on the terminal.

After compiling your project using `npm run build`, you should be able to run the project with a usage message.

```
$ ./dist/index.js
usage: index.js CONFIG_MJS [BOOKS_JSON_PATH]
$ ./dist/index.js config.mjs
listening on port 2345
```

```
^C       #stop server
$
```

You can use the following command to start the server while resetting all data to the data in `data/books.json`.

```
$ ./dist/index.js config.mjs ~/cs544/data/books.json
```

6. Running `npm test` should result in all test suites being skipped.

   Quick Sanity Checks: To check whether you have everything installed correctly, try the following:

   Access `/` on the server:

```
# start server in background
$ ./dist/index.js config.mjs &
[1] NNNNNN
$ listening on port 2345

# GET /
$ curl -s -k https://localhost:2345 | jq .
{
  "status": 404,
  "errors": [
    {
      "options": {
           "code": "NOT_FOUND"
        },
         "message": "GET not supported for /"
      }
    ]
  }

# bring server back to foreground
$ fg
./index.js config.mjs
^C    #stop server
$
```

7. You should get the `404` error as the provided code does not provide any route for `/`.

8. Open up your copy of `library-ws.ts` in an editor. Look at the comments in the file. Note that it contains the top-level exported function, a function for setting up routes, a trace handler, default handlers for `404` and `500` errors and utility functions.

9. Implement the `Add-Book` web service for adding a book corresponding to the parameters in the request body. Set up a suitable route in the router. The handler should be a simple wrapper around the `addBook()` service provided by `app.locals.model` called with the book data from `req.body`. Use the provided `selfHref()` utility function to set the `Location` header and the provided `selfResult()` utility function to build the response envelope.
   Be sure you check for errors and convert them to HTTP errors using the `procedure` outlined earlier.
   You should now be able to run the *Add Book Web Service* test suite.

10. Implement the `Get-Book` web service for reading a book given its `isbn` in the route. Set up a suitable route in the router. The handler should be a simple wrapper around the `getBook()` service provided by `app.locals.model` called with the book `isbn` extracted from `req.param`. Use the provided `selfResult()` utility function to build the response envelope.
    Be sure you check for errors and convert them to HTTP errors using the `procedure` outlined earlier. Also be sure to set up your code to return a `404` error envelope if no results are found.
    You should be able to run the entire suite.

11. Implement the `Clear` web service. You should now be able to run the *Clear Web Service* suite of tests.

12. Implement the `Find-Books` web service for finding books matching the specified `search` query parameter. Set up a suitable route in the router. The handler should be a simple wrapper around the

`findBooks()` service provided by `app.locals.model`, called with the `search` parameter from `req.query`. Be sure to modify the query to request one more result than requested by the `count` query parameter so as to enable correct display of the `next` link after the last page of results. Note that the `query` parameters will always be strings. Be sure you check for errors and convert them to HTTP errors using the `procedure` outlined earlier.

Implement the remaining web service methods for `lendings` for the `Checkout-Book` and `Return-Book` services.

13. Enable all tests and ensure that your implementation passes all of them.

Test using the command-line as in the provided `LOG`.

You can produce the log automatically using the `do-cmd-log.mjs` script.

```
$ ~/cs544/bin/do-cmd-log.mjs
~/cs544/projects/prj3/extras/log-cmds.mjs
```

This output will differ from the provided `LOG` in the `Date` headers and possibly in the `ETag` headers. You can filter those out using `grep -v`:

Assuming that you have your server running with a fresh load of books data, and that you have a `~/tmp` directory:

```
$ cd ~/cs544/projects/prj3/extras
$ cat LOG \
    | grep -v '^Date:' \
    | grep -v '^ETag:' \
 > ~/tmp/t0
$ ~/cs544/bin/do-cmd-log.mjs log-cmds.mjs \
    | grep -v '^Date:' \
    | grep -v '^ETag:' \
 > ~/tmp/t1
$ diff ~/tmp/t0 ~/tmp/t1
```

14. If the last command returns silently, then your output matches the provided `LOG`.

15. Iterate until you meet all requirements.

Submit as per your previous projects. Before submitting, please update your README to document the status of your project:

- Document known problems. If there are no known problems, explicitly state so.
- Anything else which you feel is noteworthy about your submission.