1 Project 3

Due: July 23 by 11:59p

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in your letter grade for the entire course being reduced by one level.

This document first describes the aims of this project followed by a brief overview. It then lists the requirements as explicitly as possible. This is followed by a log which should help you understand the requirements. It describes the files with which you have been provided. Finally, it provides some hints as to how the project requirements can be met.

1.1 Aims

The aims of this project are as follows:

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

1.2 Overview

In this project, you will use the persistent model from your *Project 2* to implement JSON web services to:

- Search a read-only book catalog.
- Create, update and display shopping carts.

Each one of these will be modelled as a **collection** of **collection items**.

Each collection will be available behind a RESTful **collection URL**. A collection URL `rm` can typically be set up as follows:

- A `rm` to `rm` creates a new item in the collection with a 201 `rm` empty response. The `rm` header in the response should specify the **item-URL** of the newly created item which is typically along the lines of `rm`, where `rm` is an ID for the newly created item.

  Note that this is not supported for the books collection since the books catalog is read-only.

- A `rm` to `rm` enhanced with query-parameters will search for all items in the collection which satisfy the query parameters. The response will usually contain next/previous links to facilitate scrolling through the results.

  Note that the carts collection does not support search.

- A `rm` to an *item-URL* will return the details of that collection item.

- A `rm` to an *item-URL* will update that collection item with parameters specified in the request body with a 204 No Content HTTP status.

  Note that this is not supported for books since they are read-only.

Note that a request to an invalid item URL should result in a 404 Not Found HTTP response as usual.

1.3 Requirements

You must push a `rm` directory to your github repository such that typing `rm` within that directory followed by `rm` with usage:

```
./index.mjs PORT MONGO_DB_URaL [DATA_FILE...]
```

will start a web server listening on `rm` backed up by the database specified by `rm`.

The `rm` arguments should specify `rm` or `rm` files containing book data. If there is at least one `rm` argument, then the database is cleared of **all** data (both books and carts).

You are being provided with code which provides the necessary command-line behavior.

The responses of the web services you build can be of two types:

- An **error response**: The response must be a JSON object with two properties:

  `rm` This must be an integer and must match the HTTP status for the response.

  `rm` This must be a list of errors, where each error must be a JSON object containing the following properties:

    `rm` A string giving a code for the error.

    `rm` The error message which should be as detailed as possible.

    `rm` The internal name of the widget which is the proximate cause of the error (optional).

  Note that this must be the format for **all** error responses.

- A **success response**: All non-empty success response must be a JSON object, possibly containing the following two properties:

  `rm` The main result of the web service. Documented for each web services.

  `rm` This must be a list of links for the response, where each link must be a JSON object containing the following properties:

    `rm` An absolute URL for the linked resource.

`rm` The relationship between the response and the linked resource.

`rm` A description for the link.

This top-level `rm` property must always contain a *self-link* with `rm` and `rm` both set to `rm` and `rm` set to the URL which generated the response.

What you specifically need to do is add code to the provided ws-server.mjs source file to implement the following URLs served on `rm`:

`rm rm` There is no `rm` property, but the `rm` property must contain the following 3 links:

- A *self-link* as documented above.
- A *books-link* with `rm` set to `rm` and `rm` set to `rm` and `rm` set to a collection URL for books. The value for `rm` is entirely up to you as long as it is subordinate to `rm`; the sequel refers to this `rm` value as the *books-collection-url*.
- A *carts-link* with `rm` set to `rm` and `rm` set to `rm` and `rm` set to a collection URL for carts. The value for `rm` is entirely up to you as long as it is subordinate to `rm`; the sequel refers to this `rm` value as the *carts-collection-URL*.

`rm` ***books-collection-URL*** Conduct a search for books which match specified query parameters `rm`, `rm`, `rm` and `rm`. Return `rm` as a list of matching results sorted by book `rm`. The `rm` list can contain up to `rm` results (default 5), starting at `rm` (default 0). The `rm` list should be empty if there are no matching results.

Each individual book item in the `rm` list should be enhanced with a `rm` property containing a single link with `rm` set to `rm`, `rm` set to `rm` and `rm` set to the *book-item-URL* for that book item.

The overall response should also have a top-level `rm` property containing up to 3 links:

- A *self-link* as described above. This must always be present and must point to the URL resulting in this response.
- A *next-link* with `rm` and `rm` both set to `rm` and `rm` set to a URL which can be used to get the next set of `rm` results. This link should be present only if there are subsequent results for the search.
- A *previous-link* with `rm` and `rm` both set to `rm` and `rm` set to a URL which can be used to get the previous set of `rm` results. This link should be present only if there are previous results for the search.

The last two links can be used for scrolling through search results.

rm ***book-item-URL*** The rm property is set to the book identified by *book-item-URL*. The rm property should be a list containing a single *self* link. The response should fail with a 404 error if there is no such book.

rm ***carts-collection-URL*** This should create a new shopping cart. The response should be empty with status 204 rm with a rm header set to the URL of the newly created cart. In the sequel, the URL for an individual cart is referred to as the *cart-URL*.

rm ***cart-URL*** Display the contents of the specified cart. A successful response should have the following properties:

- **'_lastModified'** A timestamp giving the time the cart was last modified.
- rm A list containing a single *self-link*.
- rm A list of the cart items. Each item should have rm and rm properties as well as a rm property specifying a single link with rm set to rm, rm set to rm and rm set to the *book-item-URL* for the book item corresponding to the rm.

rm ***cart-URL*** This request must have a JSON body giving the rm and rm to be updated. The cart specified by *cart-URL* is updated with the specified rm and rm with a return HTTP status of 204 rm. If the rm does not correspond to the ISBN of a book in the catalog, then the HTTP response should be a rm.

The behavior of the program is illustrated in this *annotated log*.

A working version of these web services can be accessed at the *<http://zdu.binghamton.edu:2345/api>*. Note that this URL is accessible only from within the CS network; you should be able to access it from your VM.

1.4 Provided Files

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

ws-server.mjs A skeleton file for your project. You should be doing all your development in this file.

index.mjs The file invoked on the command-line. It is a trivial wrapper which simply calls rm.

cli.mjs This file provides the complete command-line behavior which is required by your program. It requires model.mjs. You **must not** modify this file; this ensures that your server meets its command-line specifications.

model.mjs Identical to the skeleton file provided for your previous project. This file has been provided only to ensure that the code in the rm directory can be run out-of-the-box. In practice, you will replace the file with either that from your solution to Project 2, or the class solution for Project 2.

meta.mjs Meta-information about the different model object categories. Identical to that provided for your previous project.

model-error.mjs A trival class for application errors. Identical to that provided for your previous project.

validator.mjs Validation code for checking for local errors which depend only on a single object instance. Note that it provides generic validation based on types for input parameters. The model will need to perform additional validation for checking for global errors across objects. Identical to that provided for your previous project.

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 name, B-number and email address at which you would like to receive project-related email). After the header you may include any content which you would like read during the grading of your project.

The *course data directory* contains data files which can be used to test your project. It remains unchanged from your previous project.

1.5 Hints

The following points are worth noting:

- The provided code should pretty much take care of **most** error handling.
- When running your server in the background use bash's `rm` command to put it in the foreground. Once it is in the foreground, you can kill it using control-C.
- Remember that specifying the `rm` arguments when starting your server results in all data (including carts) being cleared and all books being reloaded from `rm`. So this is not something you want to do when you are working on carts.
- The JSON produced by the web services is not formatted. If the web services are run on the command line using a program like curl, the output can be piped through a program like json_pp (pre-installed on your VM). If running directly within Chrome (not using some kind of REST client), use a Chrome extension like *JSON Formatter*.
- 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 `rm` is installed within your `rm` directory, you can start your server from the `rm` directory using something like:

```
$ npx nodemon --watch src \
    ./index.mjs 2345 mongodb://localhost:27017/books
```

and the server will be automatically restarted whenever you make changes to JavaScript files in `rm`.

The above command assumes that the books have already been loaded into the database. It is usually a bad idea to provide the `rm` argument, as doing so will result in your database being cleared out on every automated restart; this can be rather disconcerting if you are attempt to debug shopping carts!

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 users-ws example.

2. The requirements only specify a single URL `rm`. Decide on the *books-collection-URL* and the *carts-collection-URL*.

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, or a brower client like *Talend API Tester* or yarc.

4. Set up your `rm` branch and `rm` as per your previous projects.

5. Install necessary modules. You will need `rm`, `rm` and `rm` as runtime dependencies and `rm` as a development dependency.

6. You should be able to run the project with a usage message and the `rm` base URL returning a partial result:

```
$ ./index.mjs
966164
usage: index.mjs PORT MONGO_DB_URL [DATA_FILE...]
$ ./index.mjs 2345 mongodb://localhost:27017/books &
[2] 966196
$ 966196
listening on port 2345

$ curl -s http://localhost:2345/api | json_pp
{
   "links" : [
      {
        "href" : "http://localhost:2345/api",
        "name" : "self",
        "rel" : "self"
      }
   ]
}
```

7. Replace the `rm` entries in the `rm` template and commit your project to github.

8. Replace the skeleton `rm` file in your `rm` directory with one from a solution to Project 2. You can use one from your solution or that from the *provided solution*. Note that a model instance generated by the `rm` from the provided solution exports a `rm` property; this is a somewhat clumsy attempt to avoid violating DRY by providing access to the default value for the number of search results.

9. Complete the provided `rm` handler to add in links for the book and cart collections based on the URLs you have chosen.

10. Point a `rm` route to a handler for creating a new shopping cart.

    The handler will need to be `rm` since it will need to get the cart ID by making an asynchronous `rm` call to the model accessed as `rm`. The returned cart ID can be used to build up a URL for the newly created cart. Add a `rm` header giving the cart's URL using res.append(). Finally, set the response status as per the requirements and end an empty response.

    The code for the handler (and all subsequent handlers) should be enclosed within a `rm` similar to the `rm` provided for `rm`. Note that you could wrap each handler within an `rm` function as done in the users-ws example, but that is not necessary as long as you keep the `rm` block code simple enough to avoid exceptions.

    When testing, you can use mongo shell to verify that you have created a new cart in the database.

11. Set up a `rm` handler for updating a cart. Since different carts will have different cart IDs, use express's path patterns to allow arbitrary cart IDs. Extract the cart ID from the request URL using req.params. Since the body-parser has been set up as a JSON parser for all request bodies, you should find an object corresponding to the request parameters like `rm` and `rm` in `rm`. Combine these parameters with the cart ID extracted from the request path to make a call to the model's `rm` method.

    Verify your tests using mongo shell.

12. Set up a `rm` handler for an individual book. You can call the model's `rm` with an `rm` search parameter containing the book's isbn extracted from the request path. If `rm` does not return exactly one book, throw a suitable error. Build up the required response object containing a suitable `rm` property. Finally deliver the built-up response using res.json().

13. Set up a route and handler to display the contents of a cart. Retrieve the cart contents by calling `rm` on the model. Massage the contents into the required format, adding `rm` properties.

14. Set up a `rm` handler for searching for books by query parameters accessed via req.query. Extract `rm` and `rm` from the query parameters, defaulting them to the specified default values if they are not present. These enhanced query parameters can be used as parameters to the model's `rm` method except for one embellishment which facilitates the *next-link*.

    Recall that the requirements specify that a *next-link* should be present only if there are further results. Hence set things up so that the count provided to `rm` is one greater than the requested count. Then if `rm` returns this larger count of results, then the *next-link* must be generated, otherwise it should not be generated.

    Set up a response object to contain the returned books with each books enhanced with a `rm` property for the individual book. Set up a top-level `rm` property for the response object. It will always contain a self-link. It should contain a *previous-link* only if the requested `rm` was greater than 0. It will contain a *next-link* as described above.

15. Iterate until you meet all requirements.

It is a good idea to commit and push your project periodically whenever you have made significant changes. When complete, please follow the procedure given in the *git setup* document to merge your `rm` branch into your `rm` branch and submit your project to the grader via github.