1 Project 4

Due: Nov 20 by 11:59p

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

This document first provides the aims of this project. It then lists the requirements as explicitly as possible. It then provides some background information. Finally, it provides some hints as to how those requirements can be met.

1.1 Aims

The aims of this project are as follows:

- To introduce you to building a dynamic web set using only server-side programming.
- To expose you to the use of *mustache templates*.
- To enable you to build a web site based on remote web services.

1.2 Requirements

You must check in a `submit/prj4-sol` directory in your gitlab project such that typing `npm install` within that directory is sufficient to run the project using `./index.js` with usage as follows:

```
$ ./index.js PORT WS_BASE_URL
```

where

`PORT` Specifies the port at which your program will listen for HTTP requests.

`WS_BASE_URL` The URL where the web services developed in your *previous project* are running.

Running the above command should start a web server running on `PORT`. The server should support the following pages:

Root Page Any attempt to access this page at URL `/` should simply do a redirect to the home page.

Home Page This page at URL `/docs` should contain a suitable heading and two links:

1. A link to the document search page. The link **must** have an `id` attribute set to `search`.

2. A link to the document add page. The link **must** have an `id` attribute set to `add`.

Document Search Page This page must display a form containing a text input widget having attribute `id` set to `query` and a submit button with `id` attribute set to `submit`. Typing in one or more space-separated search terms into the `query` widget and clicking the `submit` button should result in a search for the search terms in the underlying document collection.

After a successful search, the document search page should be redisplayed with the user input retained. Up to 5 matching documents should be displayed for a successful search with each result displayed in a paragraph having the following format.

- A line containing a link to the content page for the matching document. The link's contents must contain the name for the document in the underlying document collection and it must have a `class` attribute containing `doc-name`.
- One-or-more lines from the matching document which contain the search terms. Within each of these lines occurrences of search term words must have a `class` attribute containing `search-term`.

Following the results, there may be up to 2 links:

- A link to the document search page displaying the previous set of results for the same query. This link must have its `id` attribute set to `previous`.
- A link to the document search page displaying the next set of results for the same query. This link must have its `id` attribute set to `next`.

Any errors on submitting the form should be reported with suitable messages with `class` attribute containing `error`: Errors include the following:

- Submitting the form with an empty query field.
- No results for the query.
- Errors resulting from the underlying web services.

Document Add Page This page must display a form containing a file upload widget having attribute `id` set to `file` and a submit button with `id` attribute set to `submit`. Selecting a file using the file upload widget and clicking the submit button should result in the contents of the file being added to the underlying document collection. The name of the added document should be set to the base name of the file with any `.txt` extension removed.

Successful addition of a document should redirect to the content page for the newly added document.

Any errors on submitting the form should be reported with suitable messages with `class` attribute containing `error`: Errors include the following:

- Submitting the form without choosing a file.
- Errors resulting from the underlying web services.

Document Content Page This page must display the content of a document from the underlying document collection preceeded by a heading giving the name of document. The content must have a `class` attribute containing `content` and the name must have a `class` attribute containing `doc-name`. The displayed content should have the same line formatting as the underlying document.

Note that only the URLs for the root and home pages have been specified as `/` and `/docs` respectively. The URLs for all other pages may be chosen appropriately.

All of the above pages except the root and home pages must display a footer containing links to the home page, document add page and document search page with `id` attribute set to `home`, `add` and `search` respectively.

The browser back button should work without any errors.

All error messages should be reported with a `class` attribute containing `error`.

[It is important to meet the specifications regarding `class` and `id` attributes as meeting those specifications will allow automated testing of your project. If your project works when run manually but does not work when run under a script because you did not meet the `class` and `id` requirements, you will loose at least 10 points.]

1.3 Existing Servers

A slightly modified solution to *Project 3* is running on `zdu.binghamton.edu:1235` with the `snark` data files preloaded. It is restarted automatically every one hour.

The modification to the project 3 web services adds a `start` index to each of the links returned from the search results. This makes it possible to build links for this project without parsing the URLs returned in the links.

Additionally, a solution to this project is running on *<http://zdu.binghamton.edu:1237>*. That server was started using the command:

```
$ ./index.js 1237 http://zdu.binghamton.edu:1235
```

1.4 Provided Files

The prj4-sol directory contains a start for your project. It contains the following files:

docs.js This skeleton file constitutes the guts of your project. You will need to flesh out the skeleton, adding code as per the documentation. You should feel free to add any auxiliary function or method definitions as required.

docs-ws.js This skeleton file should export a constructor function which instantiates an object having methods which provide wrappers for the remote web services as per your requirements.

index.js This file provides the complete command-line behavior which is required by your program. It requires docs.js docs-ws.js and . You **must not** modify this file.

Note that unlike your previous project, the provided `index.js` does not record the PID in a file; if you run it in the background, you will need to manually track its PID in order to kill it.

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.

Statics Directory A directory for static files. As provided, it merely contains a stylesheet which can be `<link>`'d into your HTML pages (you will need to add structure your HTML and/or add `class` attributes to take advantage of the stylesheet).

Templates Directory This directory is empty and can be used to hold your `*.ms` mustache templates which will be loaded into memory when your server starts up.

The course data directory has not changed since the previous project.

1.5 Hints

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. Play with the solution available on *<http://zdu.binghamton.edu:1237>*.
2. Study the *users site* code discussed in class. In particular, note the set up for static files, templates routes and action routines.

3. Ensure that your copy of the `cs580w` repository is up-to-date by pulling from the course repository.

   ```
   $ cd ~/cs580w
   $ git pull
   ```

4. Start your project by creating a `work/prj4-sol` directory in your gitlab project. Change into that directory and initialize your project by running `npm init -y`. This will create a `package.json` file; this file should be committed to your repository.

5. Copy the provided files into your project directory:

   ```
   $ cp -pr $HOME/cs580w/projects/prj4/prj4-sol/* .
   ```

   This should copy in the `README` template, the `index.js` command-line interface program, and the `docs.js` and `docs-ws.js` skeleton file into your project directory.

6. Install project dependencies. Minimally, you will need `axios` (a HTTP client used for accessing remote web services), `express.js` (for running the project server), `multer` (for handling file uploads) and `mustache` (for templating).

   ```
   $ npm install axios express multer mustache
   ```

7. You should now be able to run the project enough to get a usage message.

   ```
   $ ./index.js
   usage: index.js PORT WS_BASE_URL
   $
   ```

8. Set up a file in the statics directory to hold the contents of your home page. Verify that it works for the `/docs` URL specified for the home page.

9. Add a route for the root page. Set up its handler to merely redirect (using `res.redirect()`) to the home page. Test to ensure that the redirect works.

10. In the `docs-ws.js` skeleton file, implement wrappers for the web services. Specifically, add methods to access the following web services:

    Get Content Return content for a document having a particular name.

    Add Content Add content for a document under a particular name.

    Search Search document collection for some `searchTerms` string with results starting from some `start` index.

11. Add a route for the document content page. Set up its handler to use the get content web service to generate the page as per the project requirements. Test using a document which already exists in the underlying web services.

12. Add a route for the document add page. Set up its handler to generate the addition form using a suitable template. Be sure to specify the form's `enctype` attribute as `multipart/form-data`. Set up `multer` to upload a single file for that route as shown in the *multer docs*. If this is done correctly, you will be able to access the file details from the express.js `req` object. You can derive the document name from the file's `originalname` by stripping off any `.txt` extension. You can get the file's contents by simply using `buffer.toString('utf8')` on the file's `buffer` property. Once you have extracted these fields, it is a simple matter to add the document to the document collection using the add content web service. When the service is successful, send a redirect to a URL for the newly added document.

13. Implement the document search page. Set up its handler to generate the addition form using a suitable template. You will need to do extensive massaging on the results returned from the document search web service before you can mix them into the template.

14. 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 it is complete, please follow the procedure given in the *gitlab setup directions* to submit your project using the `submit` directory.