Project Sprint 3 (D3) - Web Server + Frontend UI

Sprint 1 built a query engine to answer queries about UBC course sections.Sprint 2 extended the query engine to aggregate answers about actual courses and support details about course spaces on campus. Ultimately, you have built a robust backend for quickly and flexibly querying a diverse dataset. This is a great foundation, but is not particularly usable: how could your implementation actually be used in the real world? After all, software is of little value if it is never deployed and used, no matter how well it is designed and implemented.

In this deliverable you will complement your backend with a Web server and a frontend UI such that your software can be deployed and used as a Web application. Just as is common practice in Web development, you will be given a boilerplate UI implementation (consider us the "designers"). Most of the functionality has been removed from the sources and it will be your job to hook it all up with your Web server implementation.

This deliverable will extend your solutions for D1 and D2 so you must continue to work with the same partner using the same repository. D3 is worth 25% of your final grade, and becomes available once you have received at least 60% on D2. You will not have to hand anything in, though you will need to complete the open response assessment on the edX platform to receive your final deliverable grade.

Important notes:

- Autobot will create a pull request in your repos during the beginning of this sprint. It will contain some changes to your project (boilerplate frontend and server implementations) to transform your software into a Web app. It is a very good idea to carefully look at this pull request and what Autobot wants to merge into your projects. However, please do not merge the pull request before the final D2 submission as it may break your coverage scores.
- This spec should be considered a high level outline for this deliverable because it would be very easy to get lost in the details.
- There may be more pull requests from Autobot during this deliverable because we may need to change things on the go. Please keep an open eye on the discussion boards and merge all pull requests sooner rather than later to avoid conflicts.

Web Server

Bootstrap code

Once you merge the pull request from Autobot, you'll have three new files in your project associated with the implementation of a Web server:

- `/src/App.ts` contains the source code for starting the application and initializing the

server. This will be given to you for free.
- `/src/rest/Server.ts` contains the logic for your server.
- `/test/Server.spec.ts` contains the tests for your server.

Both the `Server.ts` and `Server.spec.ts` files will contain some sample code to point you in the right direction. We will use `restify` as a REST server library. Please refer to its documentation first whenever questions arise.

REST Endpoints

You will adapt your existing `InsightFacade` to also be accessed using REST endpoints. Both `InsightFacade` and the REST endpoints must continue to work independently. You will note that the REST descriptions below correspond closely to the values you are already surfacing from `InsightFacade`.

- `GET /` returns the frontend UI; this will already be implemented for you.

- `PUT /dataset/:id/:kind` allows to submit a zip file that will be parsed and used for future queries. The zip file content will be sent 'raw' as a buffer, you will need to convert it to base64 server side.

    - Response Codes and message formats:
        - `204` : Same as for 204 in `InsightFacade::addDataset(..)` .
        - `400` : Same as for 400 in `InsightFacade::addDataset(..)` .
- `DELETE /dataset/:id` deletes the existing dataset stored. This will delete both disk and memory caches for the dataset for the `id` meaning that subsequent queries for that `id` should fail unless a new `PUT` happens first.

    - Response Codes and message formats:
        - `204` : Same as for 204 in `InsightFacade::removeDataset(..)` .
        - `404` : Same as for 404 in `InsightFacade::removeDataset(..)` .
- `POST /query` sends the query to the application. The query will be in JSON format in the post body.

- NOTE: the server may be shutdown between the `PUT` and the `POST` . This endpoint should always check for a persisted data structure on disk before returning a missing dataset error.

- Response Codes and message formats:

    - `200` : Same as for 200 in `InsightFacade::performQuery(..)` .
    - `400` : Same as for 400 in `InsightFacade::performQuery(..)` .
- `GET /datasets` returns a list of datasets that were added.

Other `GET/*` endpoints will serve static resources. This will already be implemented in the bootstrap as well.

The `:id` and `:kind` portions above represent variable names that are extracted from the endpoint URL. For the PUT example URL `http://localhost:4321/dataset/mycourses/courses` , `mycourses` would be the `id` and `courses` would be the `kind`.

Testing

The same libraries and frameworks as before (`Mocha`, `Chai`) will be used for testing. This time, however, your tests will have to send requests to your backend and check the received responses for validity. The bootstrap code in `/test/Server.spec.ts` will point you in the right direction.

Starting and accessing the app

A new yarn command `yarn start` will be available in your project through a change to `package.json`. It will essentially run `App.js` as a `node` application. Once you started the server, you'll be able to access the app in the browser at `http://localhost:4321`. Please note that your datasets must be available for the UI to work.

Frontend UI

Bootstrap code

Once you merge the pull request from Autobot, you'll have a new directory `/frontend` in your repo that contains the boilerplate frontend implementation. The subdirectory `public` contains the static sources that will be hosted by your Web app:

- `index.html` contains the HTML code for the UI. This file is hosted by the `GET /` endpoint of your REST server.
- `bundle.css` contains the styles for the UI.
- `bundle.js` contains the existing logic for the UI.
- `query-builder.js` will contain the logic for building queries from the UI.
- `query-sender.js` will contain the logic for sending queries from the UI to your Web server.
- `query-index.js` will contain the logic for the chain of building a query and sending it to the UI once the form in the UI is submitted.

Please note that you should only touch the `query-*.js` files for your frontend implementation. Each of the three files is described in *Implementation* section below. The other parts are given to you and should not be changed.

Implementation

The frontend part of this deliverable differs from your previous development in several ways. The two most significant are:

1. Plain JavaScript. While it is theoretically possible to develop in TypeScript on the frontend as well, it is not common practice and we will stick to plain JavaScript here. Please apply to this rule and don't try to make TypeScript work somehow within the `/frontend` directory. The advantage is that you won't have to build/compile your project when you work on the frontend.

2. Browser. You will dive into the world of browsers with your frontend implementation. Your frontend code will be run client-side in the browser and will communicate with your Web server via REST/Ajax calls. This means also that you will have the global `window`, `document` and `XMLHttpRequest` objects from the browser available anywhere in your code.

The source code of the UI merged into your repository will expose a global object `CampusExplorer` on the browser's `window` object that contains three methods:

- `CampusExplorer.buildQuery` builds queries from the current state of the UI. Information from the UI must be extracted using the browser-native global `document` object. The returned queries must be of the same format as the ones given to your `InsightFacade.performQuery` method.
- `CampusExplorer.sendQuery` sends an Ajax/REST request to the `POST /query` endpoint of your Web server, taking a query as produced by `CampusExplorer.buildQuery` as argument. You must use the browser-native `XMLHttpRequest` object and its `send` and `onload` methods to send requests because otherwise the Autobot tests will fail.
- `CampusExplorer.renderResult` renders a given result from the `POST /query` endpoint in the UI.

The last of the above methods (`CampusExplorer.renderResult`) will be already available for you. It will be your job to implement the other methods in the respective files `/frontend/public/query-builder.js` and `/frontend/public/query-sender.js`.

Once these methods are implemented, you will have to attach them to the submit button in the UI and call them in the right chain in `/frontend/public/query-index.js`. The sequence is as follows:

1. Click on submit button in the UI
2. Query is extracted from UI using global `document` object (`CampusExplorer.buildQuery`)
3. Query is sent to the `POST /query` endpoint using global `XMLHttpRequest` object (`CampusExplorer.sendQuery`)
4. Result is rendered in the UI by calling `CampusExplorer.renderResult` with the response from the endpoint as argument

More specific directions will be provided as comments in the bootstrap files.

There are a few important notes on `CampusExplorer.buildQuery`. Please consider these carefully because otherwise Autobot tests may fail.

- The UI will only be able to build a subset of all possible queries. Several complex structures (e.g. nesting) are not possible and this is intended.
- If no conditions are specified, the query will have no conditions.
- If only one condition is specified, no logic connector (and/or) should be used but only the single condition.
- The order of the keys in the order section is ignored and will not be tested by Autobot.

Important note: usage of any library not native to the browser is strictly prohibited in the frontend part of this deliverable. Please stick to the global objects `CampusExplorer`, `document` and `XMLHttpRequest` which are the only ones required. Autobot will fail if you violate this requirement.

Just to make sure: for the frontend implementation, please only touch the files `query-builder.js`, `query-sender.js` and `query-index.js` in your `/frontend/public` directory. The other parts are already implemented, hooked up and ready to go in the bootstrap sources.

Testing

We will use the test runner `Karma` for frontend testing. `package.json` will change respectively to accomodate the required dependencies. Karma works in a way that it will run your tests in a browser environment. There will be a new command `yarn test:frontend` in your project that will run the frontend test suites with Karma.

The configuration for `Karma` is in `/karma.conf.js`. Please do not change this file because it may make Autobot fail running the tests. However, you can create your own configuration file (e.g. `karma.my.conf.js`) and run Karma explicitly with your file: `./node_modules/karma/bin/karma start karma.my.conf.js`. Karma is a powerful tool and can be run with different browsers, from the terminal or IDE and you can even hook up the WebStorm debugger with Chrome using the JetBrains IDE Support add-on. But you can also debug your frontend code simply using the developer tools of your browser.

There will be two test suites for D3:

- `query-builder.spec.js` contains the test suite for `CampusExplorer.buildQuery`.
- `query-sender.spec.js` contains the test suite for `CampusExplorer.sendQuery`.

Rather than editing these test suites directly, you will work with so-called HTML and JSON *fixtures* in the `/frontend/test/fixtures` directory. For testing your two methods `buildQuery` and `sendQuery` you should maintain a set of HTML files in `fixtures/html`, set of named query objects in `queries.json` and a set of descriptions in `descriptions.json`. `fixtures/html` should contain one HTML file named `[queryName].html` for each query with the HTML code of the currently active form in the UI as content. To create these HTML fixtures, the UI contains a `Copy HTML` button that will copy the current HTML of the active form element in the UI to your clipboard which you can then simply paste into your HTML files. `queries.json` should contain a `queryName => query` mapping with the queries that you expect for the HTML code in `[queryName].html`. The idea is this: "if the HTML in the UI looks like `[queryName].html`, then I expect the query with key `[queryName]` in queries.json to be returned by `CampusExplorer.buildQuery`". This way you can create your test scenarios by interacting with the UI and then create your HTML fixtures. We included an example in the bootstrap code to get a better understanding. The `query-sender.spec.js` test suite will then take every query fixture you specified in `queries.json` and check if your `CampusExplorer.sendQuery` method sends an Ajax request properly. `descriptions.json` should contain a description for each query with a mapping `queryName => description`.

While you must not edit the actual test suites `query-builder.spec.js` or `query-sender.spec.js` directly, we highly encourage you to look at the code. We implemented a few utility methods for testing on a global window object `TTT`, as well as two new `chai` assertions `equalQuery` and `sendAjaxRequest`. Understanding how the test suites work will help you understand the test framework better and will generally help you with the frontend part of this deliverable.

Just to make sure: for frontend testing in this deliverable, it is sufficient to add your HTML fixtures in `/frontend/test/fixtures/html`, your expected queries in `/frontend/test/fixtures/queries.json` and query descriptions in `/frontend/test/fixtures/descriptions.json`. No other files need to be changed.

Getting started

Our aim with this deliverable is to give you an impression on Web/frontend development without confronting you with extensive details and fiddling of implementing a user interface. That being said, hooking together the architecture and implementing the required parts will still be time consuming. This specification may seem rather unstructured to you. This is intended since things are often unstructured in the world of Web development and creating something structured out of an unstructured specification certainly makes a good developer. Most likely, you will spend the most time understanding how the pieces of the puzzle fit together in this deliverable.

There is no best way to get started, but a few hints may help you:

- Create the big picture of your project in your mind. Use pencil and paper and draw some diagrams. Where is what component and how do they interact with each other? What's on the client and what's on the server?

- The two parts in this deliverable (server, frontend) should be implemented and tested independent from each other. They are only hooked together with Ajax requests going from the frontend to the server but all parts are designed to be implemented and tested independently.

- The hosted reference UI will be updated with the implementation you will merge into your repos. Play around with the UI and inspect what queries are sent with what state of the UI by clicking around and sending queries.

- Make use of the querues you used for testing in the previous deliverables. You'll want to keep a clear thread throughout your whole project. (Note though that not all queries can be produced with the UI.)

- Carefully look at the bootstrap code and simply run the Web app with only the bootstrap sources. If this spec looks intimidating to you, the actual bootstrap sources might enlighten you how everything fits together.

Good luck and we sincerely hope you'll also have some fun implementing this deliverable!

Assessment

Please refer to the Project Description for more information on grading.