Overview

For the second practical we had to build the back-end for a web-based system. The application of the system is to support the allocation of dissertation topics to students. To do so the system would need to provide a RESTful service interface. In this practical we had to use a TDD approach to development and incrementally develop the required functionality guided by tests.

Design – Basic

First of all, I implemented the model for the application under js/model.js. This file contained classes for the elements that need to be modelled in this application and was implemented using the principles of OOP programming. The core functions are constructors, which construct "Dissertation", "Staff" and "Student" objects. Each constructor takes in the required attributes for the objects. In order to ensure only valid objects are created each constructor checks first a global variable "Valid" is true before returning a new object. Each constructor is called by another function, which is used to validate and verify the input JSON file used to create the objects. The first validator in the model.js file is the "DissertationFromJSON". This method first assigns the global variable "valid" too false. Next the function starts to read in the data from the input JSON file. For this practical I have decided to require an ID attribute for dissertations in the input JOSN files, rather than automatically creating them. An initial check is then made to first ensure there is an ID attribute in the file and an attempt it made to convert it to an integer, if successes full the function then using the "dataAccess" file, checks if a dissertation with a matching ID already exists on the system, if so a new "ValidationError" is thrown and the object is not created. If the ID is valid however the function continues on to check if all required components are in the file. Because some components of the input JSON file may be valid but aren't verified, I have also added checks, which first check if the proposer is a valid member on the system, if the proposer is a member of staff, the dissertation is set to available meaning it can be assigned to students. Next the function checks that the proposer role exists and is valid by checking if the proposer whether the prosper is actually a member of staff or is a student, preventing students from uploading dissertations claiming to be members of staff or vice versa. The final checks occur, if the dissertation has yet to be marked available and there is a supervisor whom is a member of staff and the proposer wasn't a student, if this criterion is met the dissertation is marked as available. Finally, if everything is valid and verified the global variable valid is set true and the function calls the "Dissertation" constructor creating and returning the new object with the input JSON file parameters.
The "StaffFromJSON" and "StudentFromJSON" functions have similar checks to those in the "DissertationFromJSON" function, using the "dataAccess" file, to check if there are any staff members or students with a matching ID's already existing in the system. Again If the ID is valid the function continues on to check if all required components are in the file. In the "StaffFromJSON" function I have use the "Validator" library to verify that the email address and phone number input also valid. Once the verification checks are complete, both the "StaffFromJSON" and "StudentFromJSON" function call a super constructor "User" which is used to create both objects. The rest of the "model" file contains prototypes for the objects, which mainly contain getters and setters with validation checks.

The next major file is the "dataAccess" file which is used to hold the data while the program is running. In this class there are 3 dictionaries one for storing "Dissertation" objects another for storing "Student" objects and finally one for storing "Staff" objects. In this file there are the appropriate function used to store and retrieve dissertations, staff members and student objects, using their ID. The substantial methods are the methods used to delete objects from the system. This is because the delete operations always cascade and delete references to themselves from other objects. The "deleteDissertation" iterates over all users and checks if any of them have the dissertation ID in their "Dissertation" array attribute. If so the ID removed from the array and the users Dissertation" attribute is updated. Similarly, if student or staff member is deleted whom proposed a dissertation that dissertation is also deleted. When a student is deleted a further check is made to see if the student was assigned to a dissertation, if so the dissertation is updated with the assigned to attribute being set to null again and if it valid to do so the dissertation will become available again.

The "api" file is the file used to provide the RESTful service interface, which by definition will use HTTP methods explicitly, is stateless and is able to transfer JSON files. The API will use GET requests to retrieve data, POST requests to create data. PUT requests to update or change data and DELETE requests to delete data. In this practical I have opted to use nodeJS and express 4.0. When the api file is run a new express instance is created, and set up to use variuous libraries such as "lodash" for merging JSON files and "body-parser" used to parse incoming request bodies in a middleware. Because I'm using express 4.0 I have used a router, which is used for determining how the application responds to a client request and routing them to a particular endpoint. Each router function, first checks gets the session access level and checks if the users authorization level.
When dissertations are returned the 'id', 'assignedTo' and 'available' attributes removed. Similarly, when staff members and students are returned their 'userid' and 'password' are removed. All the objects retunred are in JSON format.

The router includes functions to:

- Get all the available desertions in the dao object.
- Get specific dissertation in the dao, if available.
- Post a new dissertation on the system, which checks if the new dissertation is valid and if the proposer of the dissertation is the current user logged in. All users of the system are allowed to propose dissertation topics. A topic not proposed by an academic staff member will be available in the system but cannot be assigned to a student until it has been adopted by a member of academic staff to act as the supervisor. Topics proposed by students should not be visible to other students.
- Delete a dissertation if the currently logged on user is the proposer of the dissertation.
- Put an updated dissertation on the system. This method first of all creates a new dissertation with the update input and sets the ID of the new dissertation to -1, in order for it not to clash with any other dissertations, if the original supervisor of the dissertation is equal to the supervisor in the updated dissertation and the proposer is the same in both dissertations or if the user is a member of staff / admin. If all checks pass the lodash library is used to merge the update and the original and the update is saved.
- Get all users in the dao object.
- Get specific user from the dao.
- Get all staff members in the dao object.
- Get all students in the dao object.

- Put a new member of staff on the system, which first checks if the currently logged on user is admin, next the JSON file containing the new staff member's details is validated.
- Delete a member of staff on the system, checks if the currently logged on user is admin.
- Put an updated staff member on the system. This method first of all creates a new staff member with the update input and sets the ID of the new staff member to -1, in order for it not to clash with any other users. Next the function searches for the user to be updated in the dao, if found checks are preformed to see if the userid in the update is the same as the id in the query parameter, if the userid in the update is the same as the currently logged in user and finally if the users access level is equal to 2. If all checks pass the lodash library is used to merge the update and the original and the update is saved.
- Put an updated student, this process is very similar to the put an updated staff member on the system.
- Delete a user which checks to see if the current logged in user is the admin user
- Post interest in a dissertation, which first finds the dissertation using the dao and then the user, using the query parameters, if both exist then then function checks if the user currently logged in is the same as user in the query parameter or if the user currently logged in is a member of staff / admin. Next, the function checks to see if the dissertation is available, if so the user is added to the "interests" array in the dissertation object.
- Post allocation of a dissertation to a student, which first checks if the user currently logged in is a member of staff / admin. Next it finds the dissertation using the dao and then the user, using the query parameters. If both exist, the function then checks the dissertation has a supervisor or if it was proposed by a member of staff and also if the dissertation is available. If all these conditions are met the dissertation is assigned to the user and users "dissertations" array is updated with the dissertation id added. Finally, the dissertation is marked as unavailable and the "assighnedTo" attribute is set the user.
- Post and update for a member of staff, which first validates the input JSON data, next it will search for the user in the dao, if found the function continues to check if the currently logged in user is trying to update their own profile and not someone else's, finally a check is made to ensure the user is not changing their id. If all validation checks pass the update is applied.
- Post and update for a member of student, which works in a similar manner to the post and update for a member of staff.

The next file id the utils file which is used by the api to authenticate users. This file uses the "basicAuth" library to take in a user name and password whenever a user of system tries to navigate to one the URIs, this is done in the API class by using the .use function, provided by express, so whenever a URI is entered it automatically uses authentication function in this file. Initially a global variable "authLevel" is set to -1 meaning no one is logged in. Then when a user tries to access the system the "basicAuth" function is called and checks first if the credentials match that of the admin user if so the "authLevel" variable is set to 1. If the credentials don't match the admins, the function searches for a member of staff with matching credentials if one is found the "authLevel" variable is set to 2. Othwerwise the function searches for a student with matching credentials and again if one is found the "authLevel" variable is set to 3. If no users are found with matching credentials the "authLevel" variable remains at -1 and the api returns "Unauthorized".

Design – Extension

For this practical I have deiced to implement a database system, and remove the dao file, meaning the data is now stored permanently. To do this I opted to use MongoDB as it interfaces well with express and nodeJS, moreover over, it's a no SQL database system and can store JSON objects with ease. However, because the lab machines don't have MongoDB installed I have used mLab which is a Database-as-a-Service for MongoDB. In order to set up a database I had to first set up a single node on the Amazon web services platform and then using a driver via the standard MongoDB URI connect to the database in the API.

In order to save the objects created in the "model" file I have also used the mongoose library to map objects using schemas. The schemas are contained in the files: "Staff", "Student" and "Dissertation".The mongoose library also provides functions for finding, deleteing and updating entities in a MongoDB database meaning the main differences in the api file is rather than calling function in the "dataAcsess" file, the api uses the mongoose functions to directly interact with the database. Initially after adding the database extensions I was encountering problems with the authentication methods in the utils class. I therefore has to update them and now instead of using a global variable, I send the userid and the authentication level in the request. When the api is started it maintains a connection to the database.

In order to test the database, I have had to re write the "api-spec" test file. Because all the calls made in the API class are asynchronous I have had to make sure the functions in the api file are called one after another, waiting for the last call to finish before the next one is called. For example, a user has to be created before a dissertation can be added by that user. As another extension, I have also decided to use the superTest and superAgent libraries to test my API. This has allowed to me to do end to end testing whilst testing the api class too. The superTest library takes in the api file and automatically fires it up at the start of the tests and then tear it down once the tests are complete. The library allows you to make calls to the api using the correct URI's it also allows you to send data in the request and authentication details. Each test receives the response from the live server and using the chai and mocha libraries, to check the content of the header and the body is equal to what was expected. Each test using superTest contains multiple subtests which include:

- Response status
- Response header
- Response text
- Response body

Testing

I have written unit tests for the model file and the api file and the dataAccess file. The dataAccess unit tests are quite rudimentary as this all the functions have been rigorously tested with expected, erroneous and extreme data, when testing the API. I do not have a separate file for the end to end tests as the API tests are end to end tests. The reason I chose not to test the API file directly is because when the server is live the results may be different to when just accessing the methods in the file. For the basic submission, the build.sh file can be used to install the required dependences and run all the tests. For the extension I have removed the model tests as the model is the same as in the basic submission. I have uploaded all the test files used in the tests. The files have expected, erroneous and extreme data in them in order to fully test the system. It should also be noted that because I'm using a free cloud based service to store the database, the response time varies and some tests can fail due to timeouts or racing bugs. However, I have run the tests with all them passing multiple times and I'm confident in saying all the functionality works as expected. Again the build sh file can be run to install the dependences and run the tests.

Below are the screen shots of all the tests passing.

Basic:

Extension:

Evaluation

After testing my program, I pleased to say it works as expected and completes all the required functionality with extensions. There are things if I had more time I would like to update and change about the program however. These include:

- Rather than storing the description as a string create a new object which stores the different fields that can be contained in the description
- Adding a proper web interface using JADE
- Improving the database efficiency and using bty-encrypt to secure passwords
- Using the passport library to do more rigorous authentication
- Refactoring the code and modulating it

Conclusion

In conclusion I'm satisfied with end program and I'm happy I attempted the database extension as although it wasn't perfect it was an interesting endeavor and I learnt more about no SQL databases. I'm also pleased I managed to avoid using curl by using the SuperTest library.