CS5500: Foundations of Software Engineering

Spring 2025

Individual Project 2

~~Due: Friday, Feb 28 at 11:59 PM~~

Due: Friday, Mar 7 at 11:59 PM

Learning Outcomes

After completing this individual project, you will

1. learn how to implement REST-based services.
2. learn how to read and understand a formal specification.
3. use asynchronous programming concepts.
4. understand software architecture principles such as separation of concerns and layered architectures.
5. acquire experience in TDD based on e2e GUI tests.

Introduction

In this assignment, we will implement a client-server architecture. The server is designed as a layered architecture. The first layer is written in React, which serves up the web pages. The next layer is implemented on Node.js and offers several REST services, which are consumed by the React layer. The REST services communicate with a shared MongoDB database.

Why do we use Node.js to implement the REST services? Node.js allows us to maintain the entire application in one language, i.e., TypeScript. Further, it enables us to use a light-weight framework called Express to help us define the web services aligned with the REST protocol specification. Finally, Node.js has a single threaded event loop architecture, which allows us to handle multiple requests concurrently.

Additionally, we will use MongoDB as a non-relational database to store data that will persist. However, as MongoDB's native language differs from Node.js and Express, we will use Mongoose, an ORM/ODM framework, to conveniently manage the interaction between our web services and the underlying database.

Getting Started

We will use MongoDB as the NoSQL database to store data related to this application. Follow the instructions in the official MongoDB documentation to install the free community edition. Select the installation instructions for your local operating system. After you install it, follow the instructions to run MongoDB as a **background service**. The Mongo Shell (**mongosh**) should also have been installed when you installed MongoDB. If it wasn't, then follow the instructions here. The Mongo Shell provides a command line interface that is used to interact with the databases in MongoDB. If *mongosh* is successfully installed, then the command **mongosh** should connect to the local instance of MongoDB on your system and open an interpreter where we can type commands to interact with MongoDB. Try the command **show dbs;** you should see a list of existing databases in your local instance. **Note that by default, the MongoDB service will run on *127.0.0.1* (localhost), port *27017*. We recommend that you do not change these settings.**

Install the current LTS version of Node.js. We will use this to manage React and the packages needed to run our server. When you install Node.js, it will come with the **npm** package manager, which will also get installed. We will use **npm** to install dependencies and also to start the react application.

Download/clone the GitHub repository from where you arrived at this document. The repository has a *server* and *client* directory. Each directory has a *package.json* file which lists the dependencies of the *server* and *client* applications respectively. Use the following commands to install all dependencies from the project's root:

```
$ cd client/
$ npm install
```

```
$ cd server/
$ npm install
```

A successful install should install the packages for *typescript, eslint, mongoose, express, jest, eslint, axios, cypress, and cors* at the very least. Run the command ***npm ls*** from each directory on a terminal to view all packages that were installed.

A brief description of the relevant packages with links to their documentation is given below.

The express framework to write server-side code. Install express in the server directory using the command **npm install express**, if the above installation process failed. If you don't yet understand how to use express, start with the examples discussed in class. For more detailed guidance look at the official Express documentation.

The mongoose data modeling library. Mongoose will help us connect with a MongoDB instance and define operations to manage and manipulate the data according to the needs of our application. Install it in the server directory using **npm install mongoose**, if the previous installation process failed.

The axios library is used to send HTTP requests to REST services. Install it in the client directory using **npm install axios**, if the previous installation steps did not work.

We will use the cors middleware to enable CORS to enable seamless connection between the client and the server during the development process. The current cors configuration allows all connections to the server for convenience. This is fine for a development environment. In a production environment where the application is deployed on a cloud service, the CORS policy needs to be specified more strictly. Read more about CORS https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

Summary of the default host/port settings:

React Instance	**https://localhost:3000**
REST services Instance	**https://localhost:8000**
Database Instance	**mongodb://127.0.0.1:27017/fake_so**

Tasks to be completed

1. Read the Open API Specification for the server in *server/openApi.yaml*. The specification will define the REST URIs and the HTTP verbs that will need to be defined for this server. It also defines the expected request and the response for each path/URI. You should look at the examples to understand the expected formats.
2. View the Open API spec in the browser by starting the server and navigating to http://localhost:8000/api-docs in the browser. Look at the operations defined in specification. For each operation you will see a "Try it out" button, click on it and execute the operations. Initially you will see a 404 error response. This is expected as the paths haven't been defined in the server. Commands to start the server:

   ```
   $ cd server/
   $ npm start
   ```

3. Write asynchronous functions for the expected URIs and their corresponding HTTP verbs in server.ts. You can start with mock requests and responses. Click the "Try it out" button in the api-docs to see the expected response. In this initial stage send mock responses that match the formats specified in the OpenAPI spec file. In *server/types* you will a few types already defined for you. You can use these types to define the shape of the responses. Add additional types to define the expected shape of the requests.

4. **Complete the mongoose schema definitions.** The **server** directory defines a schema of the data model in the **models/schema** directory in files *answers.ts*, *questions.ts*, and *tags.ts*. These files are empty. You should complete them with the appropriate schema definition for each corresponding document in the relevant MongoDB collection. The schema definition for each collection's documents is specified below:

A Question document must have the following attributes:

- *title: String*
- *text: String*
- *asked_by: String*
- *ask_date_time: Date*
- view: Number {default: 0}
- answers: an array of Answer objects
- tags: an array of Tag objects

An Answer document must have the following attributes:

- *text: String* to store the text associated with an answer
- *ans_by: String* to store the username of the answer provider
- *ans_date_time:* Date to store the date and time the answer was posted.

A Tag document must have the following attributes:

- *Name: String* to store the name of the tag.

Once the schema is defined, you can run the script in server/populate_db.ts to verify that your document schema is defined correctly. Run the following command:

```
$ cd server/
$ npx ts-node populate_db.ts <mongo_url>
```

Read the instructions at the top of the script for `<mongo_url>`. If this script throws an error, you will know something is wrong with the schema definition and must fix it. Make sure the mongod process is up and running before you execute the script.

Verifying the test data by running a few MongoDB queries in the *mongosh* shell. See MongoDB Queries Documentation.

5. Complete the functions in *server/models/schema.* For each schema object, define the static and instance methods that will be used to run queries on the

corresponding collection. See the Mongoose examples discussed in the lecture examples for inspiration. These methods should essentially act as wrapper methods for the queries. You can start by returning mock results initially, eventually replacing them with actual queries.

6. **Back to the REST functions.** For each REST operation, interact with the wrapper methods in the schemas to insert, read, update, or delete the data. The results from this interaction should be processed and eventually returned as response from the REST functions. You can verify the REST functions by clicking on the "Try out" button for each operation in the api-docs Swagger UI for the server. Remember to populate the database with test data using the script in *server/scripts.* Run the following commands:

   ```
   $ cd server
   $ npm run populate_db <mongo_url>
   ```

   To delete all data and reset the database, run:

   ```
   $ cd server
   $ npm run remove_db <mongo_url>
   ```

7. Once you are satisfied with the server implementation. Start the server and the client in different terminals.

   ```
   $ cd server
   $ npm start
   $ cd client
   $ npm start
   ```

   Open the browser and open the client application. Verify manually if you see the correct expected results.

8. Run Cypress e2e tests. In the client you can run e2e Cypress tests to verify the server behavior and its integration with the client.

   $ cd client
   $ npx cypress open (Run the individual test files)

   Note the Cypress tests run the database scripts to insert test data and remove

test data before and after each test. Make sure to keep the server and the mongod processes running.

Optional: Use the following commands to run individual Cypress test files in headless mode (command-line environment). This is useful when you don't want to see the tests run in a browser.

```
$ cd client/
$ npm start
$ cd server/
$ npm start
$ cd client/
$ npx cypress run --spec </path/to/e2e/tests>
```

9. **Extra Credit.** The client codebase has specific code smells. Identify each code smell and use design patterns to refactor the codebase. You must detect and refactor at least two code smells for full credit. There is no partial credit. Further, this you will qualify for extra credit only if you meet the code quality criteria defined for the server.

Code quality

For the entire codebase, you must ensure that your code meets quality standards. Use the following checklist for the same.

a. Add descriptive comment in JSDoc style for the server. This includes the types, the modules, the schema definitions, and the functions in the models.
b. Separation of concerns. Wherever possible you must keep the data layer separate from the logic to process requests and responses.

c. Apply the relevant design patterns learned in class to enhance the maintainability of your code base.
d. Remove code smells such as long functions and long parameters.
e. Remove code smells based on complicated conditionals, especially where the decision is made on a categorical type.
f. There should be no style warnings. Use Eslint to detect such warnings and fix them.

Grading Rubric

1. Mongoose schema: 10%.
2. Cypress e2e tests: 45%.
3. Code quality: 40%.
4. Code style: 5%.

Useful Resources

1. Mongoose Queries: https://mongoosejs.com/docs/queries.html
2. Mongoose Documents: https://mongoosejs.com/docs/documents.html
3. Express Tutorial: https://expressjs.com/en/guide/routing.html
4. Open API Specification: https://swagger.io/specification/
5. What is Open API? https://www.openapis.org/what-is-openapi
6. MongoDB Queries Document: https://www.mongodb.com/docs/manual/tutorial/query-documents/