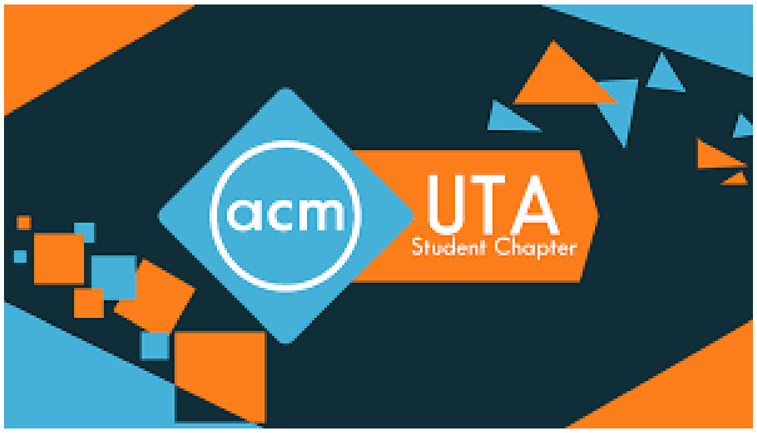
Department of Computer Science & Engineering
The University of Texas at Arlington

Detailed Design Specification
CSE 4317: Senior Design II
Spring 2021



Mav Blazers - Team 6
UTA ACM Website

Bojil Ivanov
Sanjeet Acharya
Kierra Thompson
Andy Sustaita

Revision History

Revision	Date	Author(s)	Description
0.1	1.10.2021	TEAM	document creation
0.2	1.15.2021	TEAM	document draft
1.1	1.18.2021	TEAM	draft revision 1
1.2	1.23.2021	TEAM	draft revision 2
2.1	2.15.2021	TEAM	document complete

Contents

List of Figures

List of Tables

1 Introduction

The UTA ACM website is a product dedicated to the UTA ACM chapter. It provides services for ACM members to view ACM news, updates, and upcoming events. Members can sign up and login to the website, giving them benefits which gives another incentive to join the chapter. These benefits include the features to upload blogs to their profile, sign up for email or calendar notifications, and download resources other users uploaded to their profiles. These resources could include class notes, lecture videos, or their resumes, which companies can request and view.

ACM chapter officers have more privileges to the website, allowing them to update the News, Home, and About pages. In addition, they can send out notifications and register the ACM members who signed up for the website. Website administrators have the highest level of control, can register ACM officers, and have direct access to the server and database. Other requirements for the website include support for multiple browsers and mobile devices and a professional look.

2 System Overview

The architecture of the website is split into four main layers. The View Layer handles the rendering of the current HTML page to the browser. The Resource Layer acts as an API that creates, reads, updates, and deletes information stored in the database. The Resource Layer will use this API sending and retrieving information from the browser to the database. The Authentication Layer is responsible for signing the user in, authenticating the user on the server, and storing the user role which determines how the user can interact with the website. The Database Layer acts a persistent storage for the website's data.

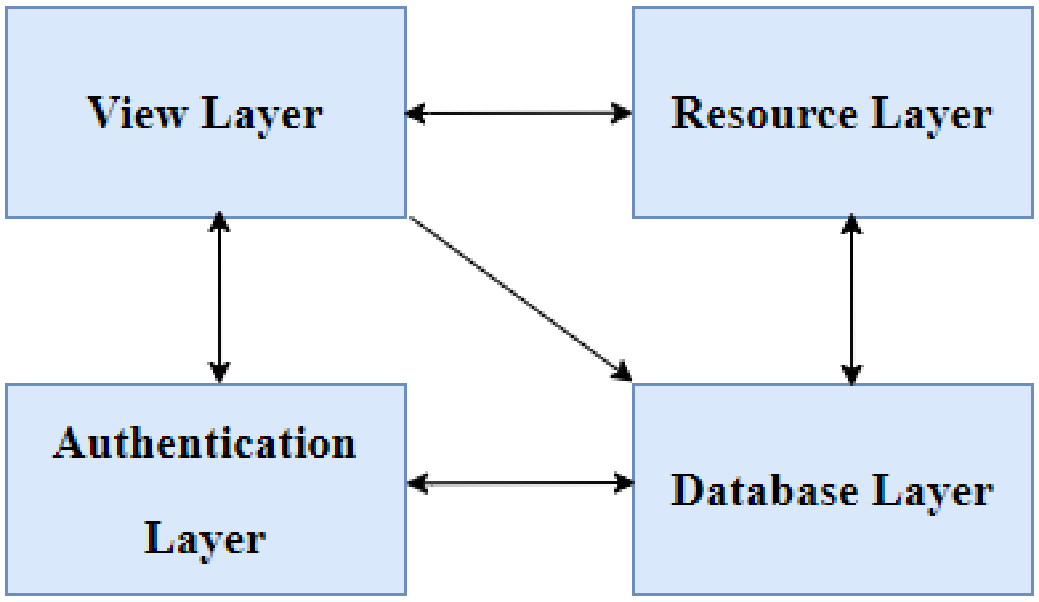


Figure 1: System architecture

3 View Layer Subsystems

The View Layer is responsible for all rendering of the web pages, including browser-side rendering (done on the front end) and server-side rendering (done on the back end). The React library is used to route pages through HTTP requests to the UI Express server, generate HTML pages on the back end for server-side rendering, and create React Components which control how the user interacts with the displayed web pages. The UI Express server serves all of the web pages and gives responses to any requests from the browser. The other main library used for rendering is React-Bootstrap, which gives us an easy way to design the interface components and support mobile devices.

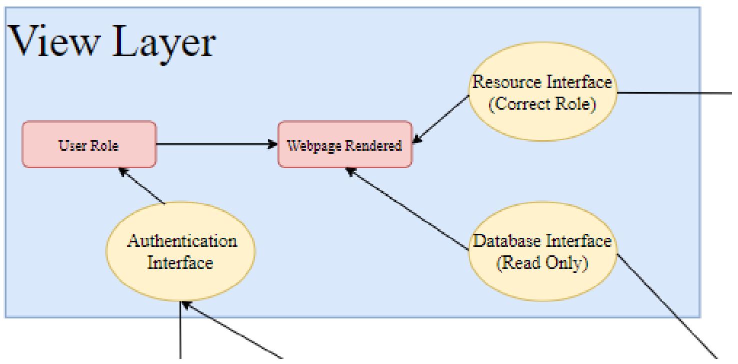

View Layer
Resource Interface (Correct Role)
User Role
Webpage Rendered
Authentication Interface
Database Interface (Read Only)

Figure 2: View Layer Diagram

3.1 Layer Browser Support

Different browsers have differing support for JavaScript constructs. This makes it difficult to use the latest JavaScript and React versions and have multiple browsers supported. As a solution, this project uses the Babel library, which is a JavaScript compiler that transforms the code to support any browser specified. This library also supports React. Along with Babel, we will use Babel-polyfill which fills in JavaScript functions that may not be defined on older browsers.

3.2 Layer Software Dependencies

This project uses NPM and NodeJS for development. Here are some of the necessary libraries for the front end.
"babel-polyfill": "6.26.0",
"bootstrap": "3.4.1",
"express": "4.17.1",
"react": "16.14.0",
"react-bootstrap": "0.33.1",
"react-dom": "16.14.0",
"react-router-bootstrap": "0.25.0",
"react-router-dom": "4.3.1",
"react-select": "2.4.4",

3.3 User Role Subsystem

The User Role Subsystem determines the level of interaction the user can have with the web page. There are four main user roles. By default, the user that is not signed in is considered a visitor. Visitors can view most of the web pages on the website, including blogs, news, updates, and sign up for email notifications. They cannot update or create and upload things to the website. Whenever a user signs in, the user is authenticated by the Authentication Layer and is set to the correct role.

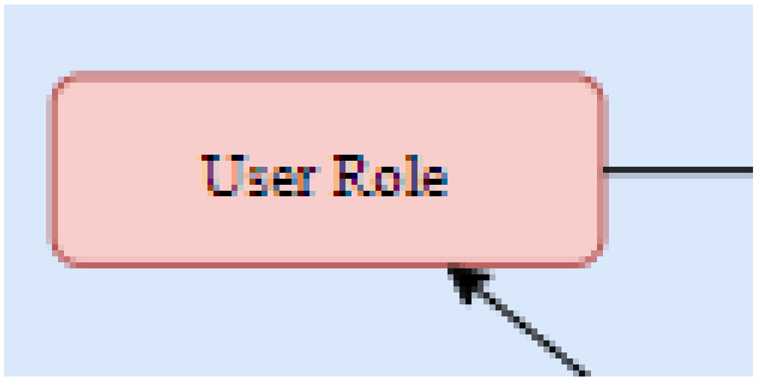


Figure 3: User Role Subsystem

3.3.1 Subsystem Software Dependencies

React is the main software dependency for setting the User Role. React can store Contexts, which act as global variables for React Components. This will allow each React Component an easy way to get the user's role.

3.3.2 Subsystem Programming Languages

JavaScript.

3.3.3 Subsystem Data Structures

The data structure used here is the React.createContext function. It takes in a JavaScript object and returns a data structure that can be input into any React Component. The user's name, ID, and authorization level is stored in this data structure and any web page can retrieve this information to display the correct information and allow the correct level of control.

3.4 Webpage Rendered Subsystem

The HTML page contains the JavaScript file of the rendered React Component along with Bootstrap CSS. The web page itself can be rendered on the browser where the JavaScript is executed on the client's computer. It can also be rendered on the server, where the JavaScript is compiled on the back end and placed into the served HTML file. The main tools to control the interaction and display of the web page is React and Bootstrap.

3.4.1 Subsystem Software Dependencies

React and Bootstrap.

3.4.2 Subsystem Programming Languages

JavaScript.

3.4.3 Subsystem Data Structures

The main data structure relevant for rendering web pages are React Components. A React Component is a class that acts as an element inside HTML. It determines the way the web page will be rendered, stores its own state, and has methods to handle almost any interaction with the user, such as what happens when the component mounts or its state updates.

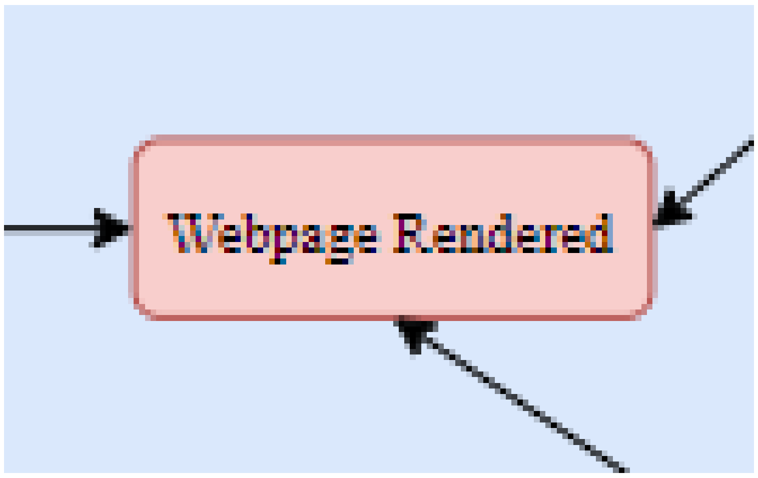


Figure 4: Webpage Rendered Subsystem

3.4.4 Subsystem Data Processing

The React Components will be routed according to the HTTP request sent to the server. These requests come from the URL bar by the user typing in the request for navigating to that page from another part of the website. The way React processes these requests are through routing. Routing chooses the appropriate React Component to display based on the request. There can also be more inputs to the URL, such as matches or searches, where variables are set through the request to have more control over what is displayed.

3.5 Authentication Interface

The authentication interface is how the front end will access the Authentication Layer. The Webpage Rendered Subsystem uses this interface to set the user role whenever the user signs in. This interface depends on which service the user uses to sign in. Whatever the sign-in data will be, it will be sent to the Authentication Layer as a JSON document though a POST request. The user will then be authenticated on the back end, and a response will be sent back to the View Layer with a cookie attached that stores the user's information. This way, the user does not need to sign back in on browser refreshes or when navigating to other parts of the website, as the session will be stored.

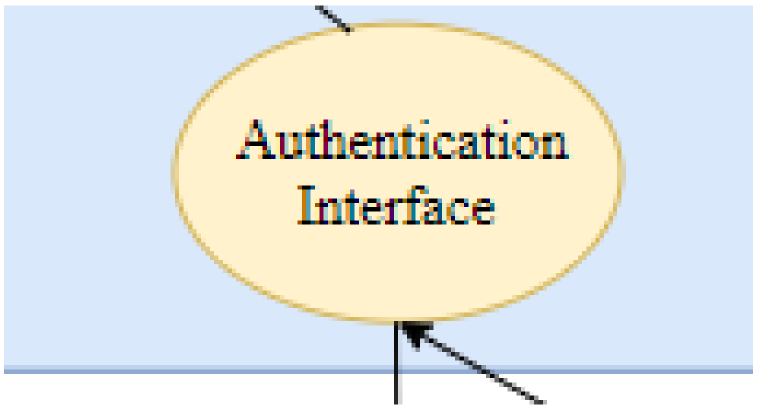


Figure 5: Authentication Interface

3.5.1 Subsystem Software Dependencies

React and Bootstrap. Cookies are attached to the HTTP requests and responses sent to the back end.

3.5.2 Subsystem Programming Languages

JavaScript.

3.5.3 Subsystem Data Structures

The user's sign-in request will be sent through a request to the back end in the from of a POST. The request body will be in JSON format which will be handled by libraries on the back end.

3.6 Resource Interface

The Resource Interface connects the View Layer to the Resource Layer. The View Layer can access APIs that the Resource Layer provides to create, read, update, and delete information on the back end. These APIs are given through the back end server running on another port which the View Layer can send HTTP requests to. The server sends back JavaScript objects which the View Layer can use to send to the Webpage Rendered Subsystem. These APIs can be accessed by any React Component or file in the View Layer. The User Role determines which APIs can be called. APIs with side-effects such as uploading information would be denied if the user does not have the correct role.

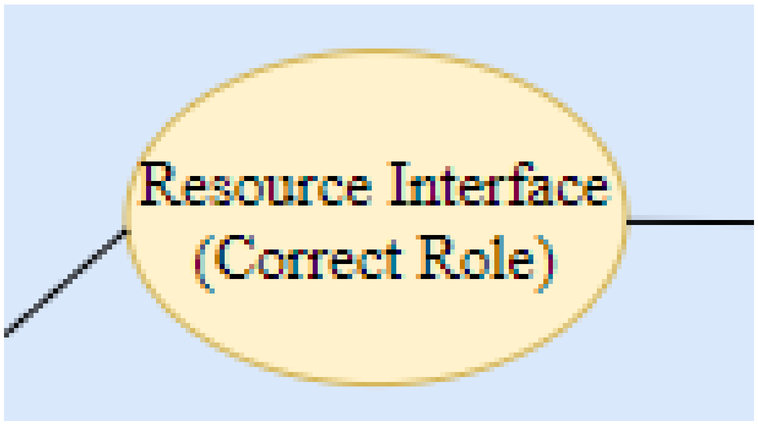


Figure 6: Resource Interface

3.6.1 Subsystem Software Dependencies

There are no software dependencies for this interface. All of the dependencies are in the Resource Layer and the View Layer can retrieve information only through an HTTP request.

3.6.2 Subsystem Programming Languages

JavaScript.

3.6.3 Subsystem Data Processing

The requests are sent in a form of GraphQL query. The Resource Interface needs to know the specification of these APIs through documentation. The documentation will include things such as which input are necessary for the request and what type of information is sent back.

3.7 Database Interface

For server-side rendering, the web pages will be initially displayed with the correct React Component. Instead of loading the Component and retrieving the necessary information to be displayed through an

HTTP request, the information is initially available to the Component. This is how the View Layer uses the Database Interface for server-side rendering.

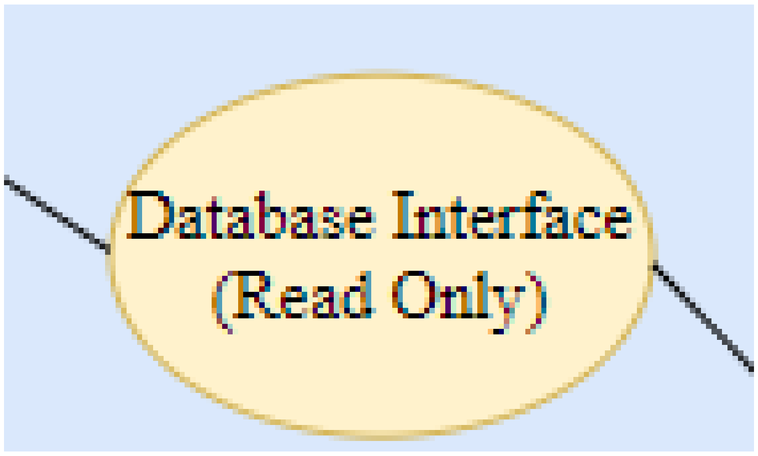


Figure 7: Database Interface

3.7.1 Subsystem Software Dependencies

MongoDB.

3.7.2 Subsystem Programming Languages

JavaScript.

3.7.3 Subsystem Data Processing

The APIs used are the same as the Resource Layer's APIs.

4 Resource Layer Subsystems

The Resource Layer is run on a back end Apollo / Express server that is different than the front end UI Express server. This server is responsible for defining all of the possible APIs that the front end can use to access the database. When any API is called, the Resource Layer executes the appropriate API. These APIs include creating entries into the database, reading entries, updating entries, and deleting entries. The APIs are created through GraphQL, which creates the definitions of the queries, and the implementations are created through JavaScript on the back end. Most of the implementations access the MongoDB database and update its information.

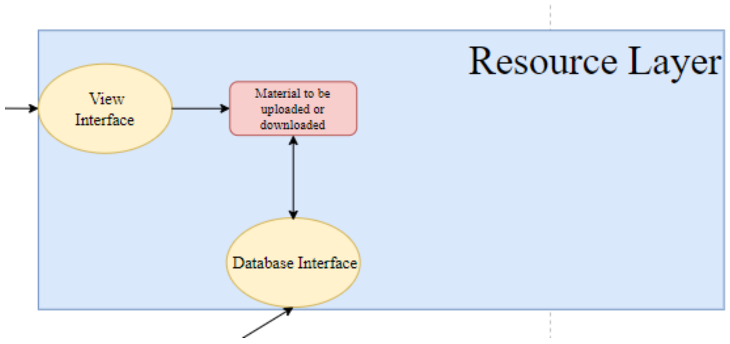


Figure 8: Resource Layer Diagram

4.1 Authentication Awareness

The APIs must know the authorization level of the user to know whether or not to deny access to the API. GraphQL has an optional context value in its APIs that can be used to store outside information. We can put the user role in this context and choose to deny service.

4.2 Layer Software Dependencies

The dependencies include the Express and MongoDB libraries to run the server and database, but we need the additional libraries Apollo-Server-Express and GraphQL libraries. The Apollo-Server-Express acts as middleware that can support GraphQL schemas and will be installed on the Express server.
"apollo-server-express": "2.19.2",
"express": "4.17.1",
"graphql": "0.13.2",
"mongodb": "3.6.3",

4.3 Material Subsystem

The Material Subsystem is what is used to upload information the user provides to the website or download information. These will include the blogs, course lecture / notes, and resumes. Each of these will have their own GraphQL definitions and implementations that will be provided as APIs to the View Layer. When an HTTP request is made to the server, the appropriate API is called and the database can be read or written to.

4.3.1 Subsystem Software Dependencies

GraphQL and MongoDB.

4.3.2 Subsystem Programming Languages

JavaScript.

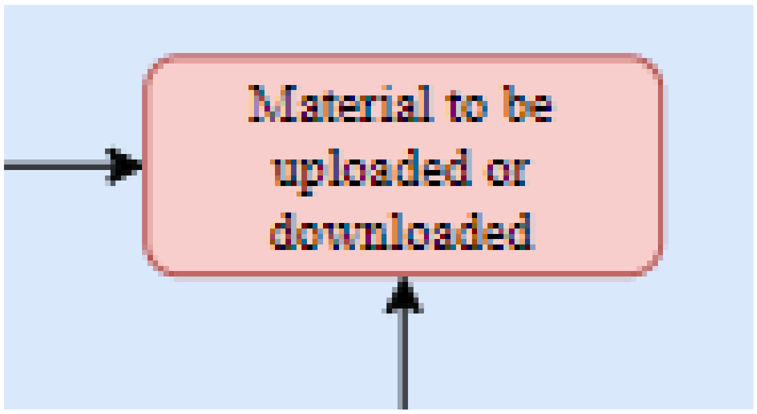


Figure 9: Material Subsystem

4.3.3 Subsystem Data Structures

GraphQL is strongly typed, which means each API must explicitly define what its inputs are and what its outputs are. For example, we will need to create a Blog type, which can be the return type of a read operation. This would contain fields for its ID, contents, owner, date of creation, and so on. We would need to define similar types for every operation we want to support. That means we would need to define a Blog type, User type, types for lectures, notes, and resumes, and so on.

4.4 View Interface

The View Interface connects the View Layer to the Resource Layer. The way the application exposes the API is through the Apollo Server middleware that can be attached to the Express server. The Apollo server takes in the definitions of all the APIs, the resolvers which are the implementations of those APIs, any contexts to use such as the current user, and a way to output error. Then, this server is installed on the Express applications on a specific path. Whenever an HTTP request is sent to this path, the Apollo server responds with the correct API.

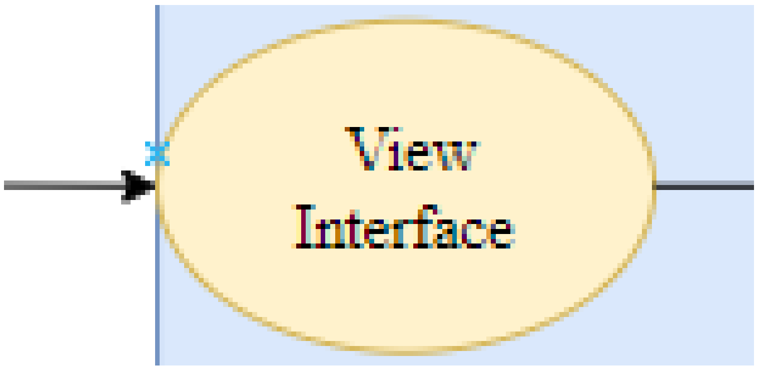


Figure 10: View Interface

4.4.1 Subsystem Software Dependencies

Apollo, Express, NodeJS.

4.4.2 Subsystem Programming Languages

JavaScript.

4.4.3 Subsystem Data Processing

Once a request is made to the server, it executes the correct resolver. The data is usually stored inside of the database, so the resolver will make a connection to the database and retrieve or update information inside of the database. The return values of all of the resolvers are JavaScript objects that were extracted from MongoDB.

4.5 Database Interface

We need a way to connect to the database and retrieve information from it for most of the resolvers. The database interface depends if we need to connect to a locally installed database or if another service is hosting the database for us. For this application, we are using a locally installed database. The methods given to us are all of the MongoDB methods that can be run on a collection of documents. We can have multiple databases and multiple collections inside of each database, which acts as tables with schemas, and inside of collections there are the documents that fill the table. There are operations to create, read, update, delete, and aggregate entries inside of the database.

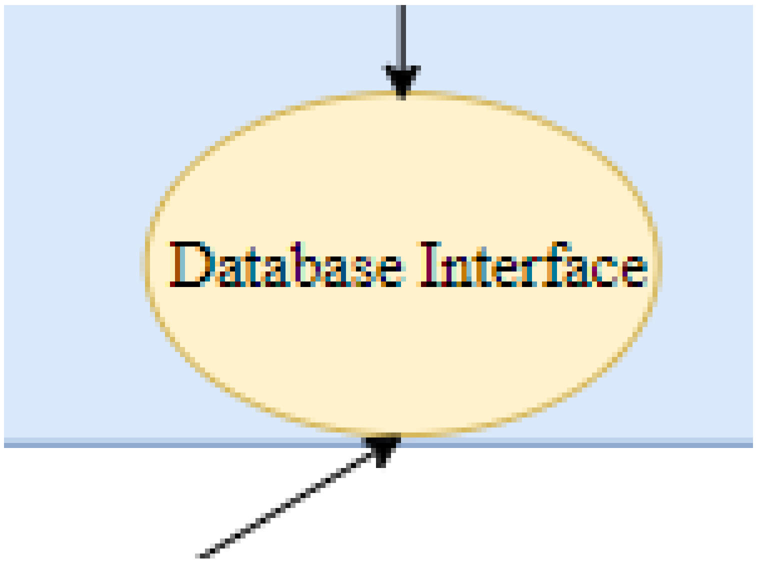


Figure 11: Database Interface

4.5.1 Subsystem Software Dependencies

MongoDB and a instance of MongoD running on the server.

4.5.2 Subsystem Programming Languages

JavaScript.

4.5.3 Subsystem Data Structures

MongoDB does not enforce a schema for its documents. This means that documents inside of each collection can have differing fields or be completely different objects all together. This may make it difficult to use the database consistently, so schemas are mostly defined by the GraphQL types mentioned earlier. For example, a collection containing all of the blogs would have its documents follow the type defined by Blog, and that way working with GraphQL and MongoDB would be simple.

4.5.4 Subsystem Data Processing

In MongoDB, multiple people could be writing to the database at the same time which creates issues for ID numbers. Using MongoDB's findOneAndUpdate operation, which is atomic, can help us avoid this. For this algorithm, we create a new collection storing metadata about all the other collections inside of the database including the counter, which is the number of documents inside that collection. Then, when we have a write operation to the collection, we can use that atomic operation to update the counter one at a time, giving us a new ID.

5 Authentication Layer Subsystems

The Authentication Layer will run on the back end and is responsible for signing up users for the website and logging users into their account. There are multiple technologies that we can use to make this process easier, such as using Google's sign in API for google accounts or Github's version. Currently, we are planning on making our own sign in system that takes a token from the front end. This token is then sent to the API server and is authenticated on the back end. After the authentication, the sign in session is persisted through a cookie which keeps a JSON Web Token. The benefits of using a JSON Web Token is that any sign in API's information can be stored in it and it is encrypted on the back end. This cookie will be attached to HTTP requests and responses to and from the server.

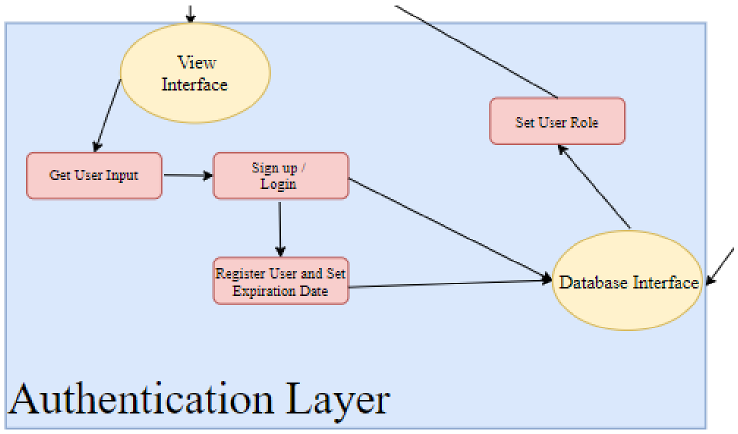


Figure 12: Authentication Layer Diagram

5.1 Layer Software Dependencies

Here are the libraries installed through NPM that our NodeJS server will use for authentication.
"body-parser": "1.19.0",
"cookie-parser": "1.4.5",
"express": "4.17.1",
"jsonwebtoken": 8.5.1",

5.2 Get User Input Subsystem

All functionalities that relate to signing the user in and out, registering the user to the website, and authentication will be available by sending HTTP requests to the server. Similar to how we request resources by accessing the GraphQL path, we will install a new path on the Express server for authorization.

5.2.1 Subsystem Software Dependencies

We need to send HTTP requests to the appropriate path so that the server responds with the correct response. We can use a JavaScript Fetch to send this request, but the structure of the request needs to be different to support cookies. We will use a POST method to send JSON data and specify to include credentials.

5.2.2 Subsystem Programming Languages

JavaScript.

Figure 13: Get User Input Subsystem

5.2.3 Subsystem Data Structures

All data sent through the HTTP request will be in JSON format. This is the structure of all tokens sent over no matter what library or API we use. This will be supported by the JSON Web Token library we are using.

5.3 Sign Up / Login Subsystem

The Sign Up / Login Subsystem is dependent on the different forms of signing in we will support. Like mentioned in the introduction, we could use Google's or any other API that supports authorization and authentication. If the user is signing up for the website, we will go to the Register User Subsystem. If it is a login request, we access the Database Interface and store the user's profile information inside the JSON Web Token. We will access this database through MongoDB and not GraphQL because we do not want to expose the users on the Apollo Server.

Figure 14: Sign Up / Login Subsystem

5.3.1 Subsystem Software Dependencies

MongoDB and any libraries we choose to support signing in.

5.3.2 Subsystem Programming Languages

JavaScript.

5.4 Register User Subsystem

To register a user, we need to make sure the user is a legitimate ACM Chapter member or officer. The only way an officer can be registered is through a system administrator. Members, on the other hand, can be registered by officers. Officers will get notifications to register these users. Along with registration, most users need an expiration date set. For yearly members, their account cannot be signed into unless their membership is updated. For lifetime members, the officers are in control of when their account expires. We can set expiration dates inside MongoDB, and when the date is reached an update to the database will occur.

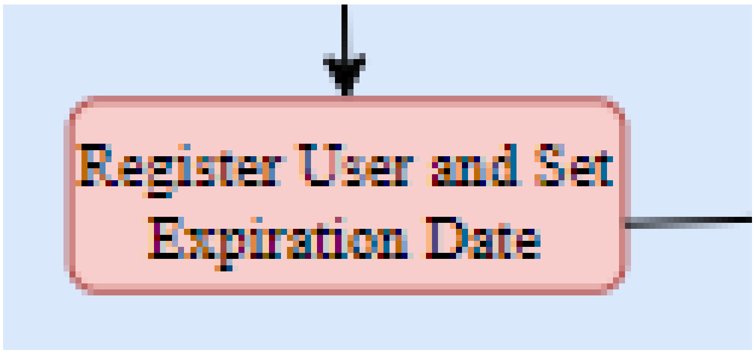


Figure 15: Register User Subsystem

5.4.1 Subsystem Software Dependencies

MongoDB and Express.

5.4.2 Subsystem Programming Languages

JavaScript.

5.5 Set User Role Subsystem

The user profile obtained from MongoDB needs to be sent back to the front end server. We need the cookie containing the JSON Web Token to be sent in the Express HTTP response. The Web Token contains all of the user's credentials and necessary information, such as user role, user ID, name and email, and so on.

5.5.1 Subsystem Software Dependencies

Apollo and Express servers, MongoDB, JSON Web Tokens, and cookies.

5.5.2 Subsystem Programming Languages

JavaScript.

5.6 View Interface

The View Interface is responsible for sending responses back to the HTML page. The created response is different than the origin of the request. Because the request is made on the UI server and the response comes from another back end server running on a different origin, we could run into CORS issues. We will use some different methods to solve these issues and maintain security by altering the HTTP requests.

5.6.1 Subsystem Software Dependencies

Apollo, Express, NodeJS.

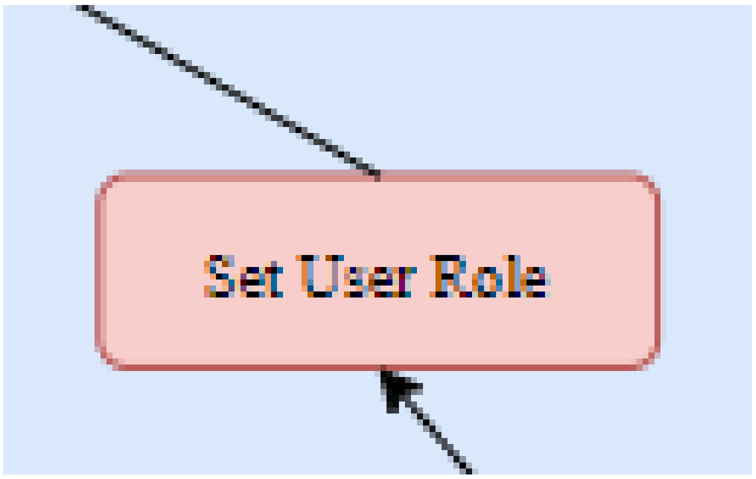


Figure 16: Set User Role Subsystem

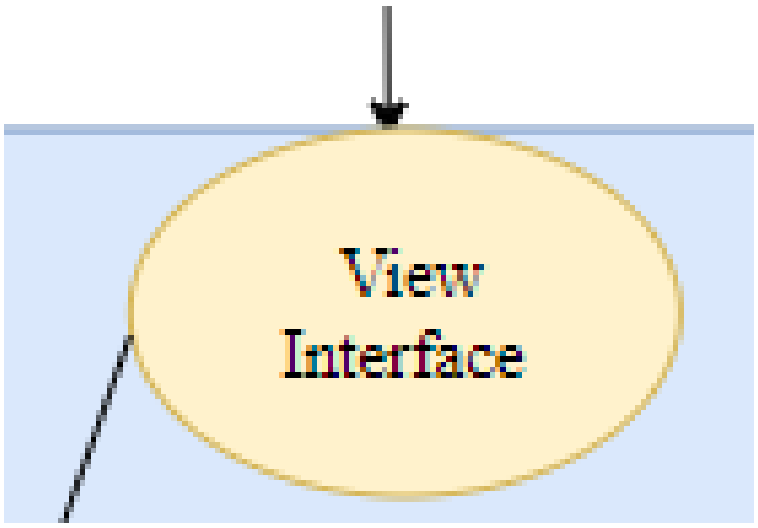


Figure 17: View Interface

5.6.2 Subsystem Programming Languages

JavaScript

5.6.3 Subsystem Data Processing

To enable CORS, the Apollo Server can set a CORS object that will allow different origins to communicate the Web Token. To do this, we set an origin, which would be the UI server, the methods allowed, which would be only POST, and we specify that the response's cookie is HttpOnly. We set these last two for security reasons.

5.7 Database Interface

The Database Interface used in the Authentication Layer is the same as the Resource Layer. The layer is exposed to the same GraphQL APIs are exposed to this layer to interact with the database.

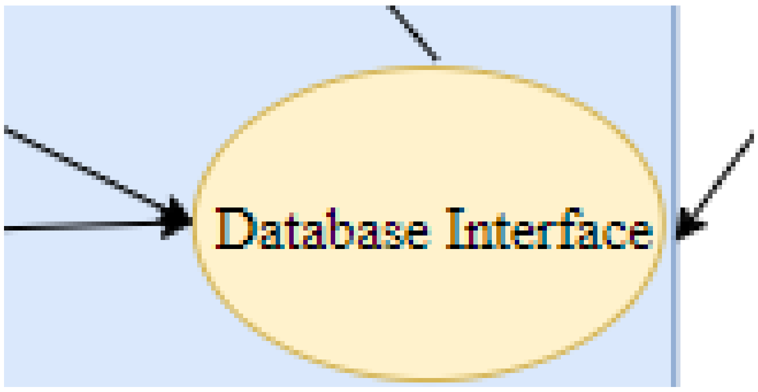


Figure 18: Database Interface

5.7.1 Subsystem Software Dependencies

MongoDB and an instance of MongoD running on the server.

5.7.2 Subsystem Programming Languages

JavaScript.

6 Database Layer Subsystems

The Database Layer controls the database and controls the way in which servers can interact with the database. There are some high level choices we can make about how to run the database, such as whether or not to run a local database or use an online hosting service. In this project, we are choosing to run a local database. Either way, however, it does not change how the database itself is accessed or used.

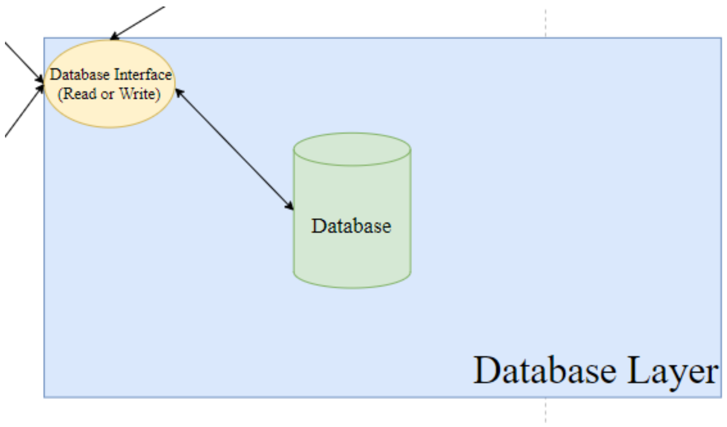


Figure 19: Database Layer Diagram

6.1 Layer Software Dependencies

MongoDB and Express.

6.2 Database Subsystem

The Database Subsystem is a running instance of MongoDB. By running a MongoDB process, we can use NPM libraries to access the database on our system and call all of the functions that MongoDB supports. For the NodeJS server files to access the database, we will define a function that makes a connection to the running process and export a variable that is the database itself.

6.2.1 Subsystem Operating System

The database should be supported on Windows, Mac, and Linux.

6.2.2 Subsystem Software Dependencies

MongoDB, MongoD, Express, NodeJS, and NPM.

6.2.3 Subsystem Programming Languages

JavaScript.

6.2.4 Subsystem Data Processing

MongoDB has a very useful function called createIndex that can be run on a collection. By default, whenever we query a collection, such as finding all documents whose field matches a number or owner, the database will need to scan every document in that collection. By making an index on a field, MongoDB creates an efficient, ordered data structure so that the query on that field will be very fast. This is also useful for searches on text documents to find certain keywords very fast. In addition, indexes allow us to make fields unique, so that no two documents in a collection can have the same value for that field. We are planning to make the user's IDs unique and have indexes on Blog titles and descriptions for fast searches.

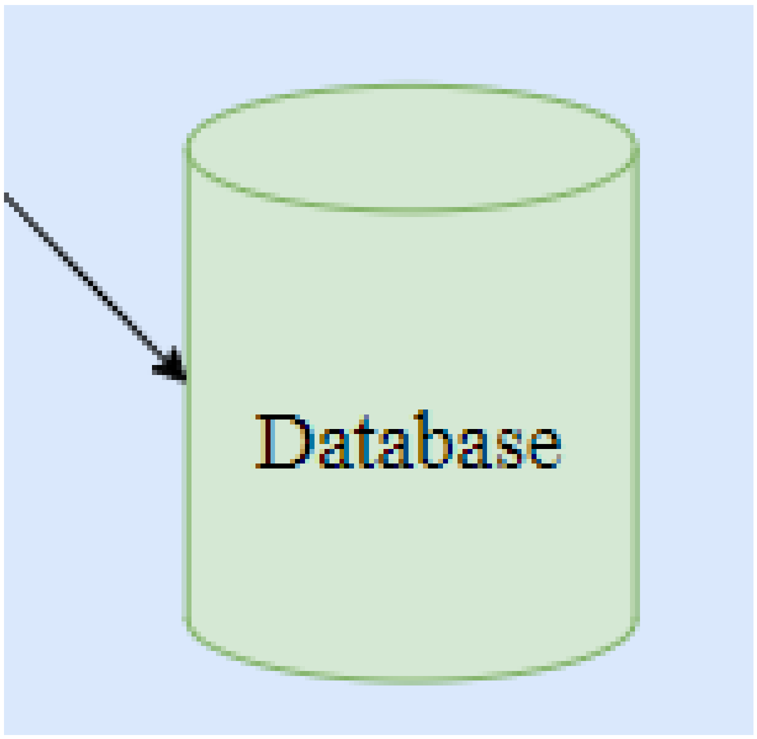


Figure 20: Database Subsystem

6.3 Database Interface

The database interface exposes the way the server can connect to the database. As described in the introduction, any file that needs access to the database will need to import a global variable. then, all database queries and mutations can be run on that local variable.

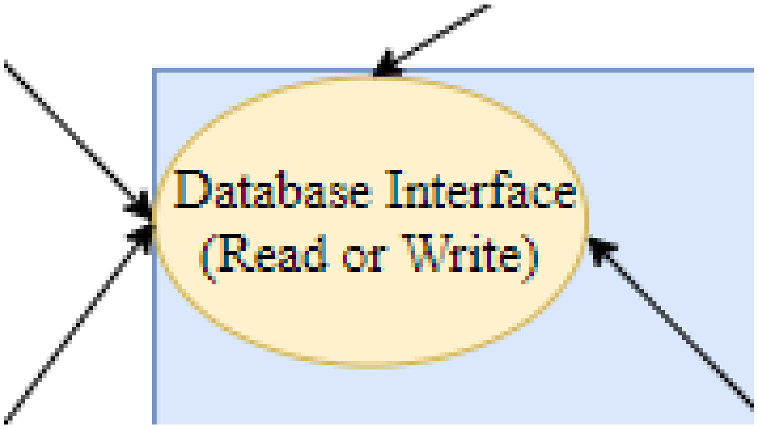


Figure 21: Database Interface

6.3.1 Subsystem Software Dependencies

MongoDB, MongoClient, MongoD, Express.

6.3.2 Subsystem Programming Languages

JavaScript.

7 Appendix A

REFERENCES