CS 130: Software Engineering

Professor: Miryung Kim

TA: Sneha Shankar

ZOW

Part B Report

Team:

Naim Ayat, Christopher Aziz, Austin Berke,
Dmitri Brereton, Reinaldo Daniswara, Aidan Wolk

November 15, 2018

Project URL: https://github.com/NaimAyat/Zow

Project Brief

Zow is now equipped with the following features:

- End-to-end authentication
- An email sending system
- Applicant summary view
- Login functionality
- Landing page
- Form creator (frontend)
- Form viewer (frontend)
- Interview scheduler (frontend)

The backend is being designed around dependency-injection principles; we've employed the GraphQL API and enabled session-based authentication and authorization. At this point, the most intricate aspects of our application are complete. The tasks that remain for Part C are relatively straightforward and primarily cosmetic.

Design

To run the current version of Zow, first download Docker and Docker Compose. Then, enter the Zow project directory and run the script `run.sh`.

Once this process is complete, you will be ready to access Zow at `localhost:3000`. To enable the emailing functionality, of course, you'll need to provide your own API key for SendGrid. To do this, read the instructions in `server/config/template.json` and create the file `server/config/private.json` accordingly.

Frontend

The frontend of Zow was scaffolded with Create React App by Facebook. We decided that React was an obvious choice, as our vision for Zow was that of a highly dynamic and responsive web application. Furthermore, React enables modular frontend design, making collaboration more efficient. Upon launching Zow, the user lands on our homepage (Fig. 1), written in TypeScript. This temporary design has a couple buttons that will link into other pages, such as facts about Zow and recruiters' login page.

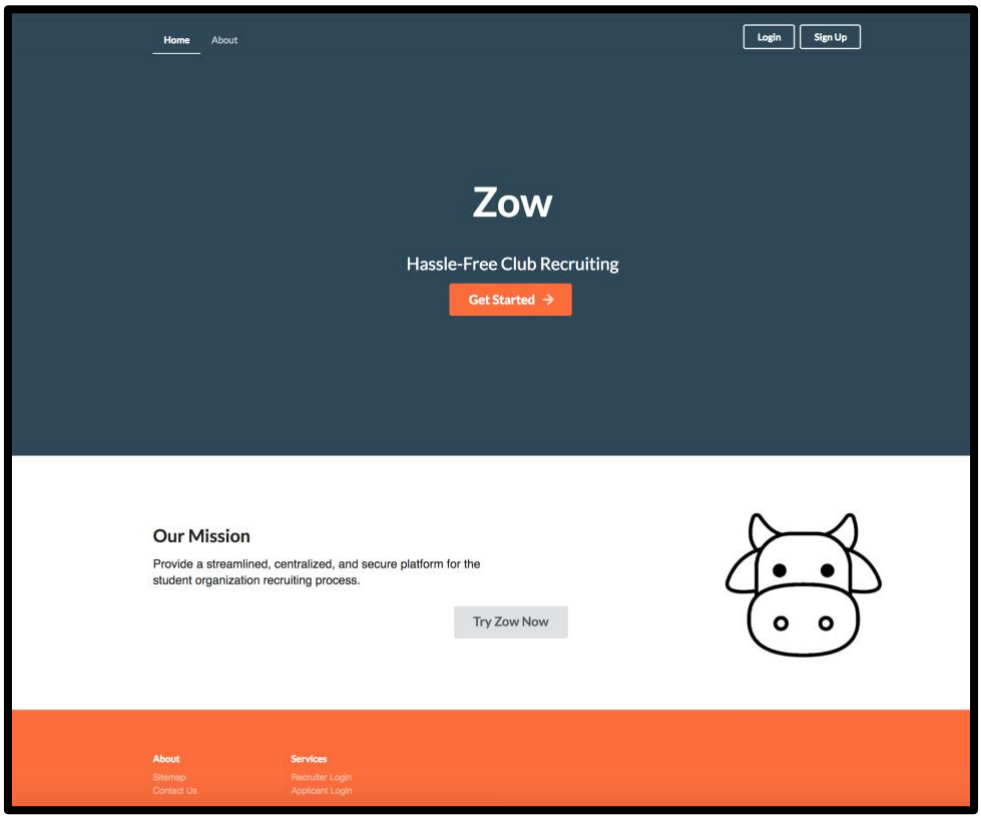


Figure 1: Zow landing page.

Via the menu bar, a user may choose to be routed to any of the following: Form Creation, Form Viewing, Applicant Summary, Applicant Scoring, Interview Selection, or Login. In the complete version of Zow, users will only see a selection of these routing options if they are logged in; the current layout is only for demonstrative purposes.

Recruiter-Exclusive Features

If we visit the Form Creation page (Fig. 2), we are presented with two mandatory input fields: Form Name and Email Question (which solicits the applicant for their email address). Note that the Email Question is required because Zow maintains communication between recruiters and applicants entirely via email. The form creator also allows us to add any combination of six other question types:

- Phone Question
- Short Text
- Long Text
- Radio Button
- Checkbox
- Dropdown

File uploads are a planned addition.

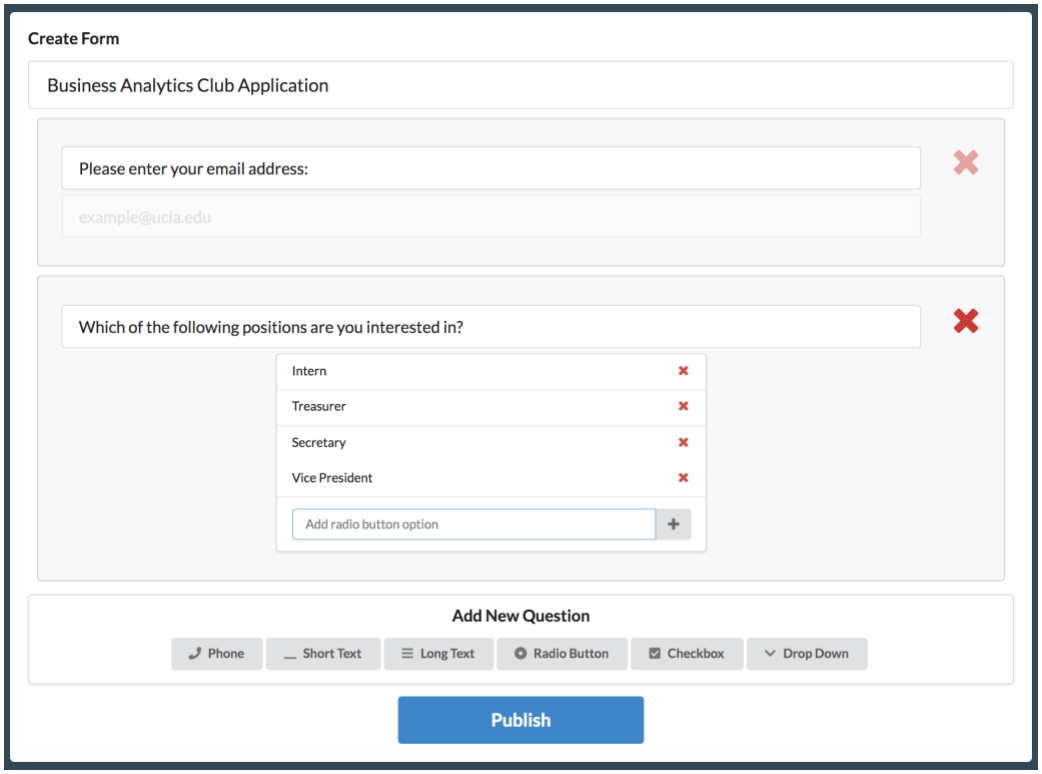


Figure 2: Form Creation page. Exclusively available to recruiters.

The Form Viewer page (Fig. 3) logically follows the Form Creation page; it displays an interactive version of the form that was previously created so that recruiters can test and review their work.

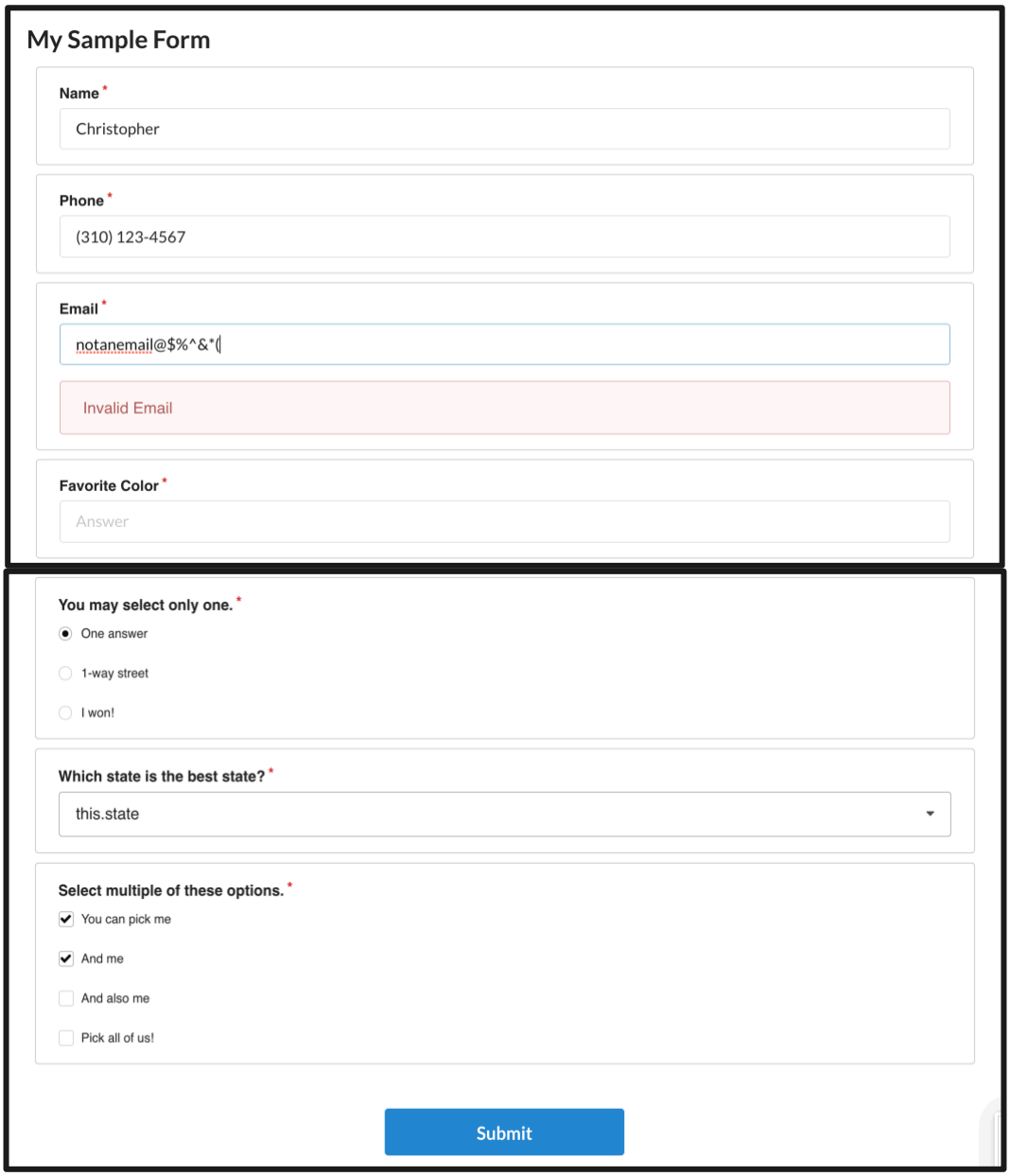
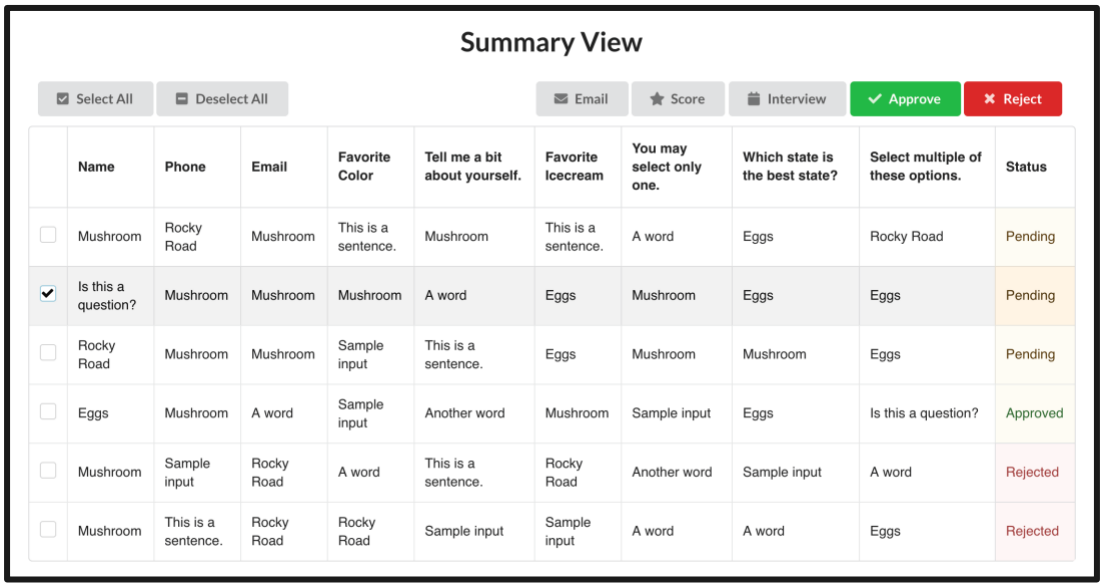


Figure 3: Form Viewer page. Exclusively available to recruiters.

The Applicant Summary page (Fig. 4) is designed as an easy-to read recruiter hub for regular visits throughout the recruiting process. The page presents a table of values populated with data from the forms submitted by applicants. Each applicant has a checkbox to the left of his or her name, enabling the recruiter to interact with entire groups of people with a single click. The recruiter may choose to email, score, interview, approve, or reject the applicants who are

checked. Barring the “score” option, each of these functionalities will automatically send an email alert to update the affected applicants on their candidacy status.



	Name	Phone	Email	Favorite Color	Tell me a bit about yourself.	Favorite Icecream	You may select only one.	Which state is the best state?	Select multiple of these options.	Status
☐	Mushroom	Rocky Road	Mushroom	This is a sentence.	Mushroom	This is a sentence.	A word	Eggs	Rocky Road	Pending
☑	Is this a question?	Mushroom	Mushroom	Mushroom	A word	Eggs	Mushroom	Eggs	Eggs	Pending
☐	Rocky Road	Mushroom	Mushroom	Sample input	This is a sentence.	Eggs	Mushroom	Mushroom	Eggs	Pending
☐	Eggs	Mushroom	A word	Sample input	Another word	Mushroom	Sample input	Eggs	Is this a question?	Approved
☐	Mushroom	Sample input	Rocky Road	A word	This is a sentence.	Rocky Road	Another word	Sample input	A word	Rejected
☐	Mushroom	This is a sentence.	Rocky Road	Rocky Road	Sample input	Sample input	A word	A word	Eggs	Rejected

Figure 4: Applicant Summary page. Exclusively available to recruiters.

Applicant-Exclusive Features

Once the recruiter decides to interview a candidate, the candidate will receive an email containing a unique link to register for an interview time slot. Clicking this link will bring the applicant to the Interview Scheduling page (Fig. 5), wherein he or she may view all the available and booked timeslots. When applicants select and submit a time slot, they receive an email confirming their appointment.

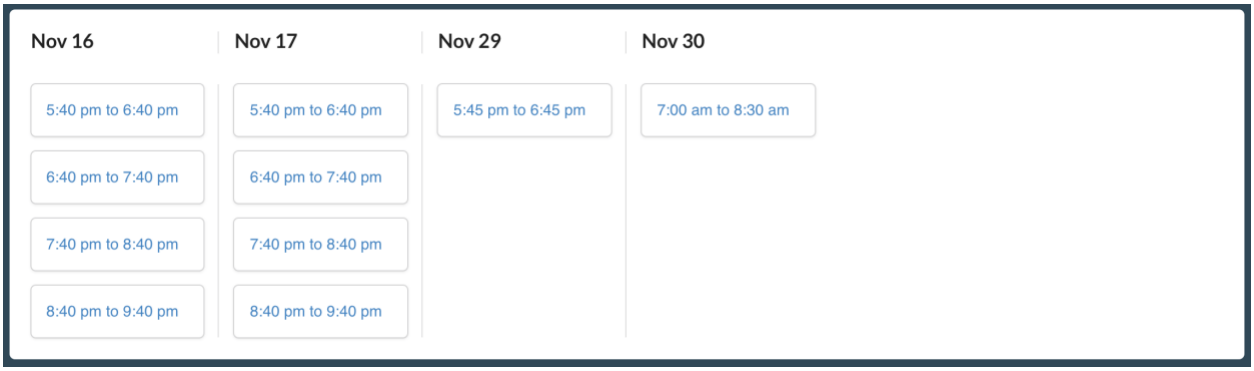


Figure 5: Interview Scheduler page. Exclusively available to applicants.

Backend

The backend is designed according to the Model-View-Control-Service (MVCS) architecture. We have separated the backend primarily into three layers: a database layer, a service layer, and a handler layer. The handler layer is "user"-facing, meaning that it is exposed to the frontend. It receives requests through its GraphQL endpoint and delivers these requests through very shallow methods into the service layer. The service layer is responsible for business logic, such as authorizing requests and fetching data from the database layer. This layer contains the most code. Finally, the database layer is responsible for the connection with Zow's database and organizing the database schema.

The handlers are interfaces using GraphQL. This technology allows us to define a strict schema of allowable interactions between the frontend and backend, and the types of data passed between. Unlike REST endpoints, GraphQL enforces proper use of the protocol through the schema. This allows us to work independently on frontend and backend components, as long as they fulfil this strictly specified interface. The backend handles requests from the GraphQL API with "resolvers," which implement the methods described in the GraphQL interface.

Services are responsible for managing a single component of the application. We currently have services responsible for authentication, password encryption, email sending, form management, and user management. These services contain the primary backend logic, and allow the handlers and database components to be loosely coupled to the rest of the backend. These are primarily designed in a dependency-injection manner, so that we provide simple interfaces for all components to implement. When using the components elsewhere, we use the interface type to refer to them, and pass in dependencies on construction. This makes components much more testable, as it is easier to mock all dependencies.

To persist data we use MongoDB. The backend maintains a connection to the database, and we have a "model" layer which interfaces with the database directly. The Model layer is written using the Mongoose JavaScript library, which allows defining Object-Oriented schemas for your data. As MongoDB is document-based, it is trivial to store JSON-like objects we produce in our TypeScript source.

Deployment

To run the Zow website, we utilize the tool Docker, a containerization platform. This allows us to sandbox our builds and running applications in a deterministic manner, so that nobody's build and execution depends on local dependencies, for instance. Docker also enables efficient builds by caching and reusing various build stages in the application. For instance, early in the server's build process third party dependencies are downloaded and installed. If the server code changes while these dependencies remain consistent, then Docker simply uses the cached container "image" and does not have to repeat the dependency installation step.

We make use of several separate Docker containers: the database, the backend, the frontend, and a proxy server. To manage all of them, we use Docker Compose. This allows us to specify several Docker containers to run together and configure independently. The proxy server is the only container exposed

externally, or outside of the Docker Compose cluster. It forwards HTTP requests to either the frontend or backend, determined by the URL of the request. The topology of the containers can be seen below:

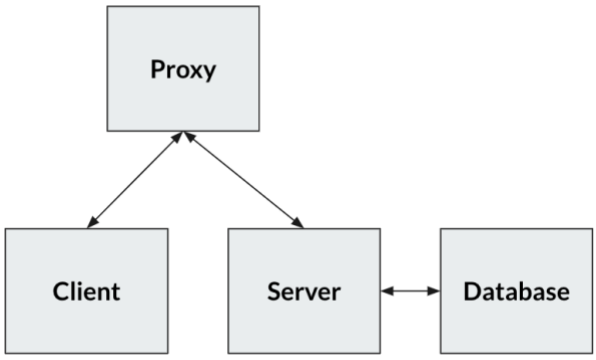


Figure 6. Container topology. The "Proxy" container routes frontend requests to the "Client" container and API requests to the "Server" container. The "Server" container communicates directly with the "Database" container.

API Documentation

As discussed in Backend Design, our backend is implemented atop a MongoDB database, with a service layer handling the business logic for processing requests, and a resolver layer handling the incoming GraphQL queries. Because of this design, there are two interfaces we have defined – the service interface which is used internally within the backend, and the GraphQL interface which is utilized by the frontend (Figure 7).

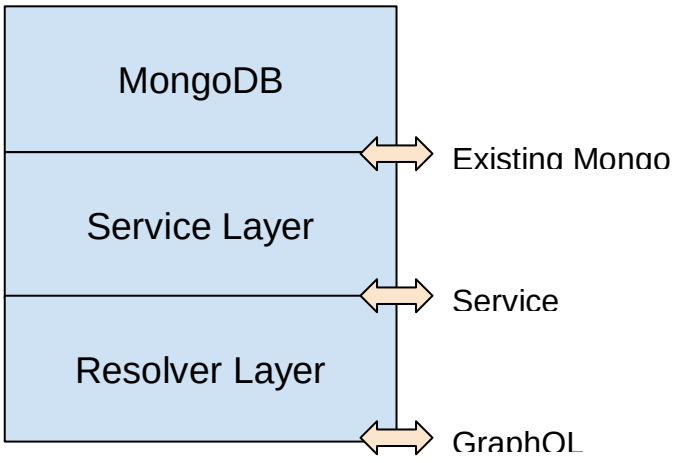


Figure 7. Backend layers and their respective interfaces.

In order to better define data types as they are passed around our application, we have defined several entities in TypeScript for use in our APIs. The entity interfaces, as well as their constituent fields and associated data types are detailed below.

Type Interfaces

Entity	Constituents
IAnswer	id?: any; question: IQuestion; value: string;
IForm	id?: any; owners: IUser[]; questions: IQuestion[]; responses: IResponse[]; interviewSlots: IInterviewSlot[];
IInterviewSlot	id?: any; start: Date; end: Date; available: boolean; intervieweeEmail: string;
IQuestion	id?: any; prompt: string; options: string[]; type: string;
IScore	score: number; notes: string; user: IUser;
IResponse	id?: any; respondent: string; answers: IAnswer[]; scores: IScore[];
IUser	id?: any; name: string; email: string; password: string;

The Service-Layer and GraphQL interfaces we have defined are documented below. We are using Typedoc[1] to generate documentation from our Javadoc-style comments, and the definitions included below are the same Javadoc-style comments used in our actual implementation. An example of the generated documentation can be seen in Figure 8..

[1] https://typedoc.org/

Service Layer Interface

*** AuthorizationService**

Function	Definition
newUser()	/** * Creates new user in database. * @param name * username * @param email * e-mail address for new user * @param password * password for new user * @return true if successful creation, false * otherwise */
login()	/** * Logs in user and creates a session. * @param email * e-mail address for user * @param password * password for user * @return true if successful login, false otherwise */
logout()	/** * Logs out user if a session is currently active. * * @return void */

*** EncryptService**

Function	Definition
testPassword()	/** * Tests plaintext password against hashed password. * * @param password * plaintext password to test * @param passwordHash * hashed password to compare against * * @return true if passwords are equivalent, false * otherwise */
hashPassword()	/** * Salts and hashes plaintext password. * * @param password * plaintext password to hash

	* * @return hashed and salted password */

*** FormService**

Function	Definition
getFormByID()	/** * Retrieves an IForm given a form ID. * @param formID * ID of form to retrieve * @return IForm for given ID */
getFormsByUser()	/** * Retrieves all IForms for a given user. * @param user * IUser for which forms are retrieved * @return IForms for given user */
getQuestions()	/** * Retrieves IQuestions for given form. * @param formID * form ID for which to retrieve questions * @return all associated IQuestions */
getResponses()	/** * Retrieves IResponses for given form. * @param formID * form ID for which to retrieve responses * @return all associated IResponses */
saveForm()	/** * Creates given form in database. * * @param form * IForm to create in database * * @return void */
createNewForm()	/** * Creates a new IForm for the given user. * * @param author * user to associate form with * * @return the newly created IForm */

addResponse()	/** * Associates a response with a given form. * * @param formID * form ID associate response with * @param answers * array of answers to include in response * * @return void */
addOwner()	/** * Associates an owner with a given form. * * @param formID * form ID associate owner with * @param newOwner * user to add as owner to form * * @return void */

*** UserService**

Function	Definition
findUserByUsernameAnd Password()	/** * Retrieves an IUser matching provided * username/password. * Compares provided password with hashed password in * database. * * @param username * name of user to retrieve * @param password * password of user to retrieve * * @return IUser matching credentials * @throws "User not found" if no matching user */
findUserById()	/** * Retrieves an IUser matching provided ID. * * @param id * id of user to retrieve * * @return IUser matching ID * @throws "User not found" if no matching user */
newUser()	/** * Creates a new user with the given username and * password.

	* Hashes password with Encryption Service * before entering in database. * * @param username * username of new user * @param password * password of new user * * @return void */

GraphQL Interface

Function	Definition
async currentUser()	/** * Retrieves the currently logged in user. * * @return currently logged in user */
async form()	/** * Retrieves a form given a form ID. * @param formID * ID of form to retrieve * @return IForm for given ID */
async forms()	/** * Retrieves all forms for a given user. * @param user * user for which forms are retrieved * @return IForms for given user */
async user()	/** * Retrieves a user given a user ID. * @param userID * ID of user to retrieve * @return IUser for given ID */
async login()	/** * Authenticates and logs in a user. * @param email * email address of user * @param password * password of user * @return true if successful login, false * otherwise */

async newUser()	/** * Creates a new user. * @param name * name of user * @param email * email address of user * @param password * password of user * @return true if successful creation, false * otherwise */
async logout()	/** * Logs out the current user. * * @return true if successful logout, false * otherwise */
async createForm()	/** * Creates a form for the currently logged in * user. * * @return the newly created IForm */
async addResponse()	/** * Adds a response for the provided form. * @param formID * ID of form * @param answers * IAnswers to questions for response * @return void */
async addOwner()	/** * Adds an owner to the provided form. * @param formID * ID of form * @param userID * owner to add to form * @return void */
async questions()	/** * Retrieves questions for given form. * @param form * form as provided by top level query * @return all associated IQuestions */
async responses()	/** * Retrieves responses for given form.

	* @param form * form as provided by top level query * @return all associated IResponses */

Figure 8. An excerpt from the Typedoc-generated documentation.
This is documenting addOwner from the Form Service.

Changeability Analysis

Due to the layered design of our APIs, we make strong use of the information hiding principle. For example, the internal structure of our MongoDB implementation is handled by the existing interface – any internal design change to Mongo will be irrelevant to our application's functionality. Similarly, our design decisions that may change at the service level are protected by the service-layer API, and our design decisions that may change at the resolver level are protected by our GraphQL API.

An example that illustrates this at the service level is a scenario in which we decide to change the login process to utilize two-factor authentication (a scenario that is becoming increasingly common in response to modern security concerns). In this case, we must modify the Authorization Service's internal code, but the functions to login(), logout() will remain effective in their signatures. The only change that would be necessary would be to add an intermediate step to check the second factor before approving the login, and no changes would be required at the resolver level.

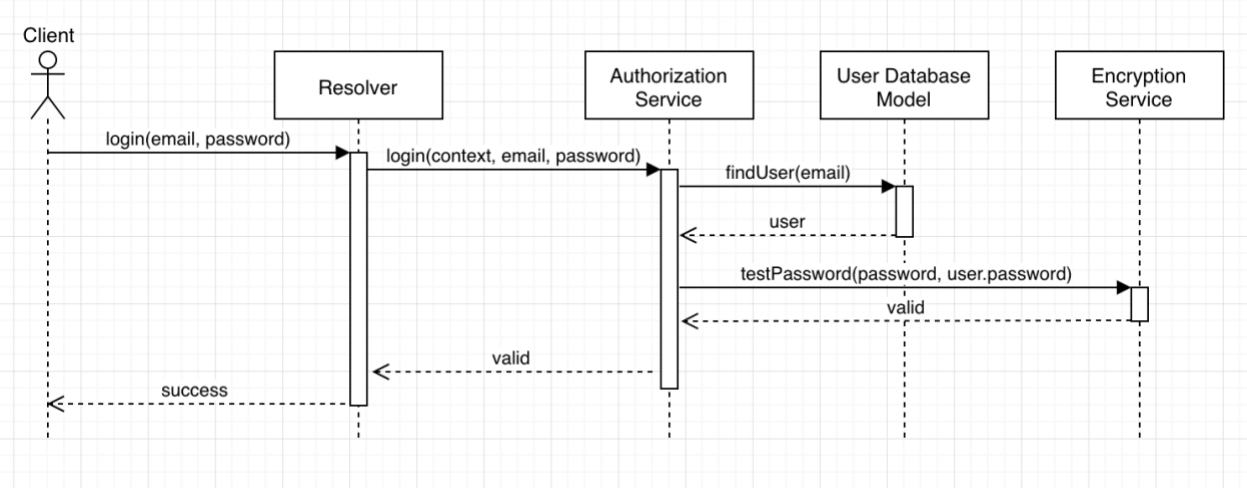
With the GraphQL API, we can address a similar scenario. If we decided to make an internal design decision to remove the service layer completely and directly access the MongoDB database within the

resolvers, we would be able to maintain the outward-facing GraphQL query structure without any change in functionality.

Similarly within the backend, we utilized the information hiding principle to mask design decisions for password encryption. We use the Bcrypt hashing algorithm to securely store user passwords in our database. We made this choice because Bcrypt is designed specifically for safely encrypting passwords. However, it is a changeable design decision. Thus, it was hidden as a concrete implementation of a generic encryption interface.

Example Sequence Diagram

Below is a sequence diagram outlining a user login process. We can identify the API calls that are happening from the client's perspective (front-end), as well as those at the resolver and service levels.



Testing

Overview

Testing is a very important part of software development, ensuring a codebase is robust and easily maintainable. For our project, we used a test driven development approach, where we wrote tests alongside the code instead of waiting until later.

We used Jest, which is a new and popular JavaScript testing framework, to handle all of our testing for this project for both frontend and backend.

Our current test coverage is as follows:

- Statement coverage: 66.67%
- Branch coverage: 85.71%
- Function coverage: 81.25%
- Line coverage: 66.67%

Testing scenarios

Backend

Testing Scenario	Description	Expected outcome	Github URL
Reading the server configuration file when it exists	Our server should be able to read its configuration file successfully. This involves loading the file, as well as accessing the keys within the file.	A successful test should load the configuration file successfully and access keys. If the server configuration values match the test file values then the test is successful. If the server has no configuration values, or has the wrong ones, then the test has failed.	https://github.com/NaimAyat/Zow/blob/master/server/src/config/index.test.ts
Reading the server configuration file when it doesn't exist	If there is no configuration file available, there should be no configuration values in the server.	A successful test should result in the server configuration values being undefined.	https://github.com/NaimAyat/Zow/blob/master/server/src/config/index.test.ts
Saving an Answer to the database without required fields	Saving an Answer to the database without the required fields should not be possible. This is part of our MongoDB schema design. The rest of our app depends on required fields always being there, so we need to make sure we aren't storing any documents without them.	A successful test should result in a ValidationError when attempting to save an Answer that doesn't have its required fields of question and value.	https://github.com/NaimAyat/Zow/blob/master/server/src/db/models/Answer.test.ts
Saving a Form to the database without required fields	Saving a Form to the database without the required fields should not be possible. This is part of our MongoDB schema design. The rest of our app depends on required fields always being there, so we need to make sure we aren't	A successful test should result in a ValidationError when attempting to save an Form that doesn't have its required fields of owners and questions.	https://github.com/NaimAyat/Zow/blob/master/server/src/db/models/Form.test.ts

	storing any documents without them.		
Saving an InterviewSlot to the database without required fields	Saving an InterviewSlot to the database without the required fields should not be possible. This is part of our MongoDB schema design. The rest of our app depends on required fields always being there, so we need to make sure we aren't storing any documents without them.	A successful test should result in a ValidationError when attempting to save an InterviewSlot that doesn't have its required fields of start and end.	https://github.com/NaimAyat/Zow/blob/master/server/src/db/models/InterviewSlot.test.ts
Saving a Question to the database without required fields	Saving a Question to the database without the required fields should not be possible. This is part of our MongoDB schema design. The rest of our app depends on required fields always being there, so we need to make sure we aren't storing any documents without them.	A successful test should result in a ValidationError when attempting to save an Question that doesn't have its required fields of prompt and type. Furthermore, to be valid, the question type must come from the following enum: const questionTypes = ["SHORT_TEXT", "LONG_TEXT", "PHONE", "EMAIL", "DROPDOWN", "CHECKBOX", "RADIO", "FILE"]; So an attempt to save a Question that has a type which is not part of the enum should also result in a ValidationError	https://github.com/NaimAyat/Zow/blob/master/server/src/db/models/Question.test.ts
Saving a Response to the database without required fields	Saving an Response to the database without the required fields should	A successful test should result in a ValidationError when	https://github.com/NaimAyat/Zow/blob/master/server/src/db/models/

	not be possible. This is part of our MongoDB schema design. The rest of our app depends on required fields always being there, so we need to make sure we aren't storing any documents without them.	attempting to save a Response that doesn't have its required fields of respondent, answers. Furthermore, a Score which is a sub-object within a Response, should have scoring.score and scoring.user. Responses that have a Score without the required Score fields should also lead to a ValidationError	Response.test.ts
Saving a User to the database without required fields	Saving a User to the database without the required fields should not be possible. This is part of our MongoDB schema design. The rest of our app depends on required fields always being there, so we need to make sure we aren't storing any documents without them.	A successful test should result in a ValidationError when attempting to save a User that doesn't have its required fields of email and password.	https://github.com/NaimAyat/Zow/blob/master/server/src/db/models/User.test.ts
Sending emails via sendgrid with the email module	Our app sends emails using the sendgrid API. We send emails for a variety of things including interview scheduling. Our email module should activate the sendgrid API then use it to send emails.	A successful test should do the following: 1. Return a valid emailer object when using the getEmailer(config) function 2. Activate the sendgrid API using this configuration 3. Make sendgrid API calls when sending emails	https://github.com/NaimAyat/Zow/blob/master/server/src/email/index.test.ts
Failing gracefully when sendgrid API key is unset	In the unlikely scenario that our sendgrid API key is not set, we don't want our entire app to crash. The email module should fail	A successful test should return a valid emailer object when using getMailer(config). It should also not throw an error when	https://github.com/NaimAyat/Zow/blob/master/server/src/email/index.test.ts

	gracefully when such an exceptional case occurs.	email.send() is used.	
GraphQL server should be created properly	GraphQL is a key part of our project that's used for data access. We specifically use the ApolloServer, which should be created without any errors.	A successful test should return a valid ApolloServer object when calling getGQLApolloServer().	https://github.com/NaimAyat/Zow/blob/master/server/src/handlers/graphql-server.test.ts
Query resolvers should be exported properly	Resolvers are a key component when using GraphQL. Our resolver module should export a Query resolver which will be used to handle all queries from the frontend.	A successful test should return a resolver with the property "Query" when calling getResolvers().	https://github.com/NaimAyat/Zow/blob/master/server/src/handlers/resolvers.test.ts
Passwords should be securely encrypted	We're storing passwords in our database, which means we have to take extra security measures. Our encryptService module handles all of our app's encryption needs. Passwords are hashed with 10 salt rounds to be very secure. This test is very important to ensuring our app's security.	A successful test should ensure the following conditions are met: 1. The hash is different from the original password 2. The hashes for same passwords are different 3. The hashes for different passwords are different 4. Wrong passwords are not accepted 5. Correct passwords are accepted	https://github.com/NaimAyat/Zow/blob/master/server/src/services/encrypt.test.ts

Frontend

Testing Scenario	**Description**	**Expected outcome**	**Github URL**
Main App component rendering properly	The App component is the parent of all other	A successful test should render the <App />	https://github.com/NaimAyat/Zow/blob/maste

	components, so we need to make sure it is rendering properly for everything else to work.	component without any errors.	r/client/src/tests/App.test.tsx
LoginForm component rendering properly	The LoginForm component is used for making user accounts and dealing with authentication. We need to ensure it renders properly as it is an integral part of the app.	A successful test should render the <LoginForm /> component without any errors.	https://github.com/NaimAyat/Zow/blob/master/client/src/tests/LoginForm.test.tsx
SummaryPage component rendering properly	The SummaryPage component is used to display a summary of all statistics from applications. It is the main dashboard for our users, so we need to ensure it renders properly	A successful test should render the <SummaryPage /> component without any errors.	https://github.com/NaimAyat/Zow/blob/master/client/src/tests/SummaryPage.test.tsx

Test directories

For the backend, tests are included in the same directory as the file to be tested. Github link: https://github.com/NaimAyat/Zow/tree/master/server/src (Tests are in each subfolder/module)

For the frontend, there is a testing folder. Github link: https://github.com/NaimAyat/Zow/tree/master/client/src/tests