Team 5 Design Document

Prototype Description Recap

Our software's purpose is to allow an admin user (such as doctors, managers, professors, etc.) to receive videos from users. From this we can enable asynchronized communication which allows for more efficient communication, enhancing the experience for both the user and admin. The intention of our software is to provide an easy to use, secure solution to asynchronous video sharing.

List of Features

- Cypress Testing Framework (**Josh**)
    - This feature involved setting up and retroactively writing regression, e2e and component tests for previously manually tested front-end components. This framework was introduced as a way to have tests that will interact with the components similar to a user, to test workflows that may have been hard to test with individual components and to simulate manual testing.
- Amplify Prod/Local Drift Bug (**Josh**)
    - This bug is considered a feature as the time taken to resolve it was immense and lots of thought and research was put into this bug. This bug had our production environment on AWS and our local environments drifting in terms of content. Our authentication system was different on each and created a significant amount of blockage for the rest of our tasks. This was resolved by a simple line fix in the environment setup of our production environment but nonetheless set up a lot of hassle in our progress.
- Automated Documentation System (**Beck**)
    - This documentation system created easy, hassle free documentation for our code using JSDoc. This program allows in-line and function headers to be transformed into automatically generated documentation that allows for easy reading of the code. This is pivotal as it creates an easy system to allow us to read each other's code and understand with documents how to easily approach access and changes to that code.

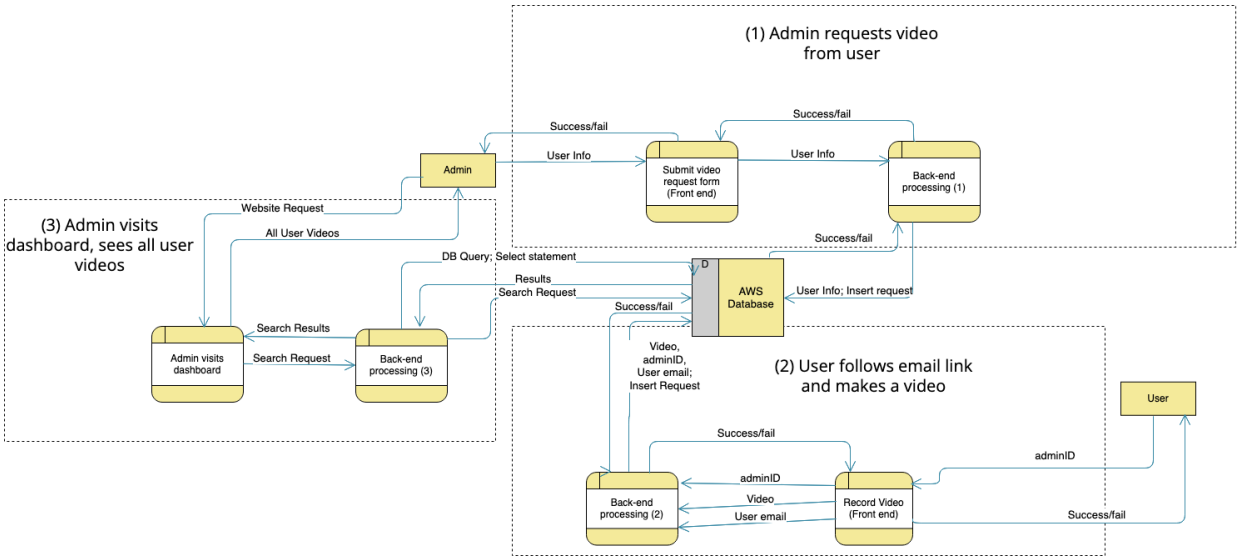
- Sent/Received Table (with Data Querying) (**Beck**)
    - This table is our primary focus of this application. This table can be found within our “doctor” dashboard where the doctor user will have stored their list of patients they have sent links to along with any received video responses, patient information and video link status. This is the central focus of our application on the doctor user group side and will allow easy access to proper patient information. This also includes querying the database with graphql.
- Webpage Routing/ Protected Routes (Different Pages, header) (**Abhinav**)
    - This feature is an important yet strangely difficult one to implement .React is often used for single-page apps - where most of the code is on a single .js file. We found some difficulties in creating multiple pages that would interact and route with each other. Specifically, we found issues with having our code on multiple pages and confirming authentication and how pages could be accessed without proper authorization or login. In the end this was resolved; we needed to use React Router DOM to create and protect routes while treating different webpages as Components that would be rendered using the Router.
- Tested and Developed UI components (React) (**Abhinav**)
    - Our aim is to have a library of reusable UI components for our webpage. We chose to build on top of the existing Amplify UI kit for building our UI components, relying on its well-tested primitive components to avoid unnecessary work and benefit from an established design philosophy. We also took a mobile-driven approach to development so that the website can render in an accessible way across multiple devices. In regards to testing, we are now using Cypress to write component tests before we start writing the code for it to ensure it renders as expected - this might need to become E2E testing to see how components interact with one another.
- Recording Video Functionality (**Kael**)
    - This functionality is the other primary focus of our application, this creates the workflow of the patient user group to send videos back to their doctor. Our recording functionality is currently implemented as a recording on the webpage which is then downloaded and further uploaded to the platform again. This is intended to be changed to not involve the users file system but we have

implemented the basic functionalities of an easy to use record and stop button for our user group where user-friendly design is a priority.

- Database Querying and Setup (**Kael**)
    - Database querying and setup took a lot of time and resources at the start of our development. Initially we focus on using AWS RDS as we wanted to focus on having a relational database but later found that with AWS Amplify that GraphQL would have the best implementation. This was then implemented and as none of us had much experience with NoSQL databases it took lots of time to figure out querying this database and manipulating it to make use of it in our program.
- PR Previews and Automated Testing on PRs (**Abby**)
    - Automated testing is a requirement for our project and took some time to implement it the way we thought best. This automated testing was first configured with Amplify in which our unit tests were run automatically on AWS and additionally we generated a build preview for each PR in which we performed manually testing. This was later involved with the cypress testing framework in which specs are also run alongside these previews using github actions to complete our full testing suite for each PR made into the develop branch.
- Homepage & Doctor Dashboard & Profile Page (**Abby**)
    - The pages of our program were essential to build according to our figma design, while it seemed like a simple enough task it proved to be difficult alongside the routing issues and creating our workflow as designed in figma. This was developed as smaller issues that turned into a feature itself and required setting up the layout, basic data querying and components for our main access pages.

System Architecture

Diagram:



Component Explanations:

A major group of components of our system architecture is our backend “model” which consists of our AWS database, and back-end processing nodes on the diagram. This component refers to the AWS Amplify backend, AWS cognito authentication, and DynamoDB systems that all process our applications systems and create a simple place to retrieve information for our admins and store the videos of for our users. These components belong in the back layer as they require a degree of abstraction to handle the privacy and security of the data added by users and admins into our system.

The other group of components of this system is our frontend “controller” which refers to our client side code that is also hosted on AWS Amplify systems, within this system we have our React framework that allows for a quick, responsive web application. In our diagram these are found as the “record video”, “admin video dashboard” and “submit video request form.” These systems are in the layer shown as they create the proper interface between the view and model to properly organize and maintain our data to keep our user experience consistent.

Architecture Choice Explanation:

The chosen pattern we based our diagrams off of is a MVC (Model View Controller) architecture. We chose this architecture as it best matched the proper workflow we had planned for this system. The pros of this setup is being able to do most of our work/processing on the server (AWS) side to free up the processing on client-side for quick loading webapps. We can effectively trust AWS with all this processing due to the scale of our application compared to the vast load AWS is able to handle. Along with this since parts are segmented among three levels it is easy to build upon and add more to the project, allowing easy testing and rapid development of new builds of our application.

In this architecture we have set the “model” as our backend DynamoDB system, described as “AWS Database”, along with our backend implementation on AWS Amplify, and other AWS systems described as “back-end processing” . This describes the data and model that will be accessed and manipulated from the system to send back to the user. The “controller” describes our client-side code that handles the logic of the webpages and connects to our model to manipulate and send submissions based on user actions. In this case the “view” would refer to the displayed data and resulting web pages that the model sends back to the view to update it based on user actions on the webpages.

Design Changes:

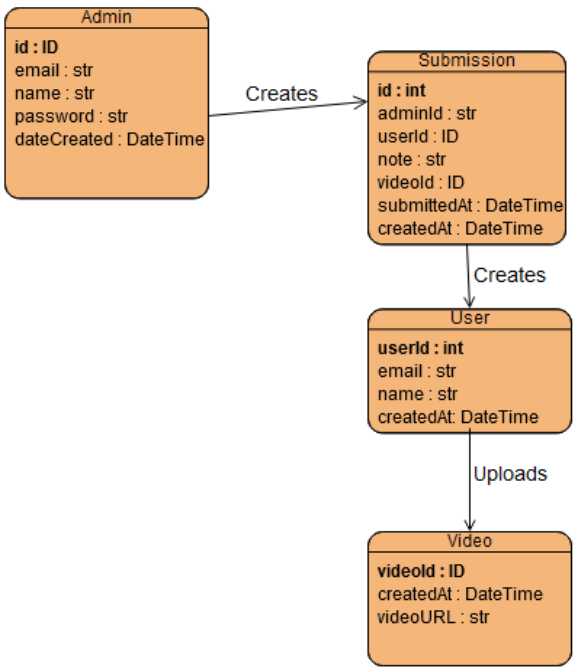
The first design choice made to our system’s architecture is found within the component choice of our database. We chose an SQL based backend originally due to prior knowledge with the technology and could easily implement it into our system. This design was quickly questioned once we saw the ease of the integrated systems within AWS Amplify. This database was based on DynamoDB, a fast NoSQL database system that none of our group members had any experience in. Yet due to the accessibility of DynamoDB we decided to switch to this system and found nothing but success learning the NoSQL system. We have found its speed and ease of integration let us focus less on setting up the database’s connections and more on our actual entities

inside of it. This also allowed us to keep our backend processing enclosed within the AWS Amplify system and not have to deal with querying many components within AWS.

The second design choice that was made relating to the structure of our architecture was adding another AWS layer that greatly assisted the usability and development of our application. This choice was to use AWS Cognito instead of creating a full authorization suite from our database and react components. This saved what could have been weeks of development to creating a full, secure auth suite into a few hours worth of work for a product that vastly surpassed anything we created. Cognito included sign-up code confirmation, 'forgot my password', easy, secure password storage among many other features. It greatly benefitted us to see what other AWS components we could use within our system due to the nature of the project which highly encouraged AWS usage, resulting in a better and more polished product.

Database Design

Diagram:



Explanation:

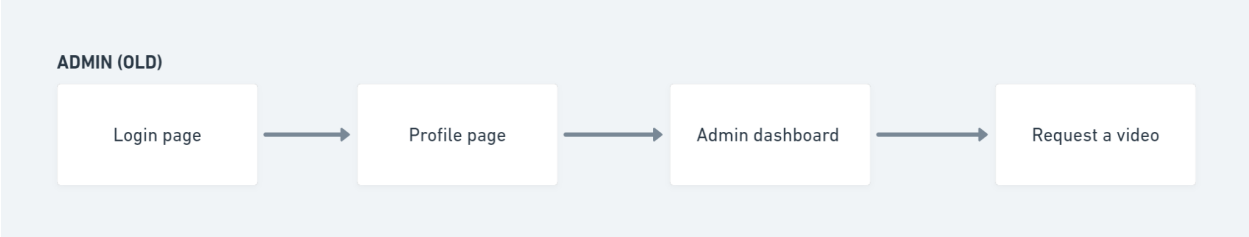
This diagram is our ER diagram for the current system implemented and consists of the tables “Admin”, “Submission”, “User” and “Video.” This workflow of tables provides a very linear path that matches the workflow of the application. Admin users will always have a table of submissions on their admin dashboard which will contain users who will upload videos to the system. This provides a clear and simplistic way to query the data in a way that allows full access for admins to their users' interactions with the software.

Our original ER diagram was made without regard to any specific system so a lot of guesswork had to be made. Once our requirements became more clear we started to refine the ER diagram into what it is now. Our major design change that we implemented stemmed from learning that all tables made with Amazon DynamoDB automatically generate “createdAt” and “id” fields, thus those had to be included in our diagram. This fortunately assisted our database as the unique id fields created an easy primary key for each entity. Otherwise our data is modeled to provide a consistent and simple flow of data with our admin user controlling the system’s functions. At one point, it was thought to have thought admin and user comments, but that decision was decided against due to redundancy and lack of use as we thought that users would be able to comment within the video and doctors may not want to provide any sensitive information in a comment field.

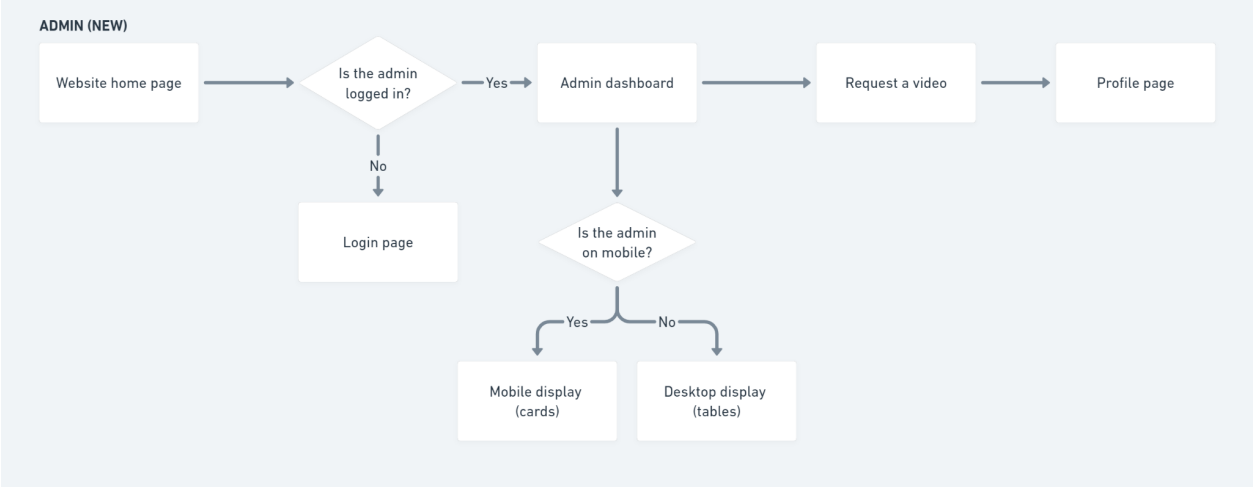
A second change we implemented to our database’s design we added the submission table which helped to simplify the flow of information and greatly helped querying with being able to directly track on the submission dashboard which user was connected to which video and admin. The submission or doctor dashboard is the key component of our admin’s ability to track the video submissions they have sent out, thus this needs to be a very thoroughly vetted part of our application. Having the submission table as a table in our database allows for querying to be from a singular table instead of having to query through multiple database tables.

User Interface (UI) Design: Navigation Flow Diagrams

First Design Iteration (Not In-use):

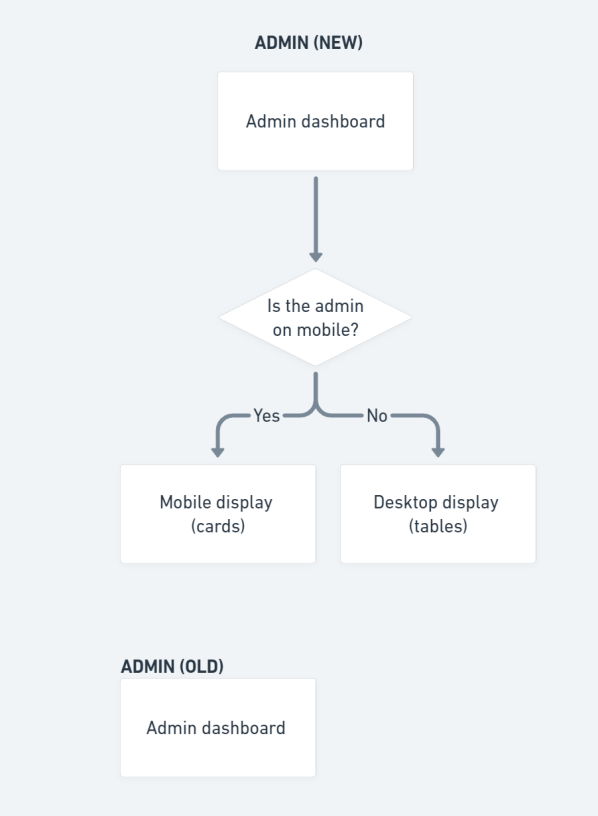


Final Design Iteration:



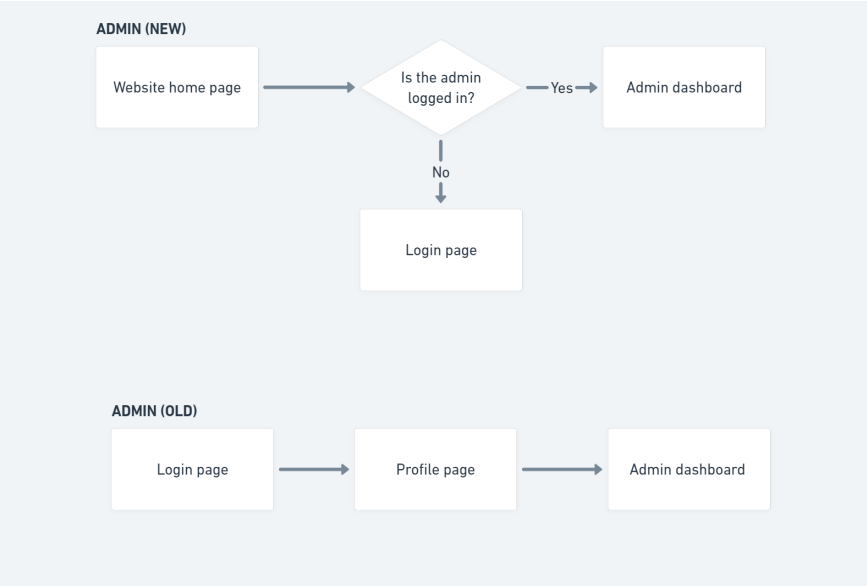
First Design Change:

We adjusted the display of the admin dashboard to better suit mobile users by making the original table responsive, making the display turn into cards on a smaller screen. This enables users on desktops to see more data in one glance, while users on mobile can see all of the information about an entry without scrolling. This navigation change improves data visualization for the admin and makes the mobile experience of our software much more user-friendly.



Second Design Change:

In our second navigation flow iteration we added a homepage and protected routes as a way for users to learn more about the software, and have a landing page to create an account or log in. The protected routes ensure that only those who have an account or are logged in can access the dashboard, request a video, and view the profile page. If they attempt to access any of these pages while not logged in they will be redirected to the login page. We also adjusted the order of pages in the navigation bar to match the most frequented pages (the dashboard will be frequented by the user much more than the profile page, and therefore will show up first upon logging in).



Choice of Diagram:

Almost all decisions in regard to UI have been made with the intention of prioritizing user accessibility and feasibility. To achieve this, we chose to build all components on top of an existing UI element developed by Amazon (Amplify UI kit) as it's been tested for accessibility and saves us time to follow an existing design philosophy. Our navigation flow was designed with security and convenience in mind.