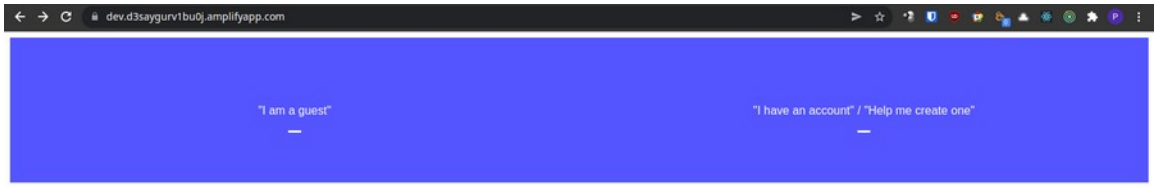
Gujar Parth Shailesh
A0235354H
CVWO Final Submission

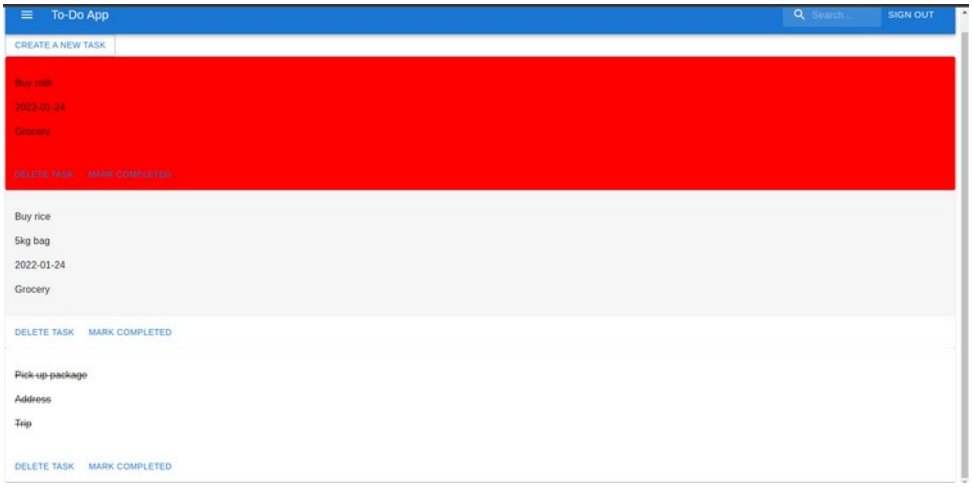
User Guide

1. At first, the start page is shown. Users can choose to continue as a guest or to quickly sign up. Guest users' data is only stored temporarily.

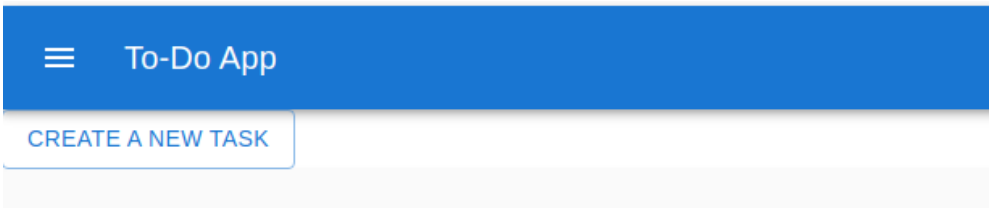


2. After the start page, the home page is shown. In the top app bar, users can navigate between the home page and the profile page. A search bar adds search functionality and will search for all fields of the task, including date (but the date format must be "YYYY-MM-DD"). Searches are case-insensitive.
3. Guest users can also choose to log in.

4. The home page looks like this:



5. Urgent tasks shown in red. Completed tasks are struck through. Tasks can be marked as completed by clicking on the 'Mark Completed' button, or deleted by clicking on the 'Delete Task' button.
6. To create a new task, click on the 'Create A New Task' button.



7. A modal will pop up. The name is a required field, but all other fields are optional. The default priority for a task is ‘Normal’, but can be set to ‘Urgent’ if required.
8. Tasks are tagged by the ‘Category’ field. Searching by category in the search bar works as well.
9. To update the task after creating it, click on the task from the home page. Another model will show up with the existing values for the user to update. Click on ‘Save’ to save the edits.
10. In the Profile page, user data can be deleted.

Accomplishments and Reflections

This project was trickier than I expected. It is the first complete web application that I have written, and the first time I have ever written frontend code. There were multiple issues involved, but on the whole, I enjoyed learning about new technologies such as React, AWS Amplify, Go, Redux, and GraphQL.

Initially, I wrote the frontend application in JavaScript, but I decided to re-write the components in TypeScript as I felt that the numerous types involved would help me make sure that I perform operations on the correct data. Indeed, there were numerous occasions when compilation errors due to incorrect types helped me debug my program. TypeScript was a bit confusing to learn initially, especially the section on generic types, but it was fun to learn.

My choice of AWS Amplify for deployment was definitely the wrong choice and the most costly in terms of time and energy. I chose AWS Amplify as I thought that it would present a superior alternative to a combination of AWS Beanstalk + EC2 + S3, where each AWS service would have to be managed separately. In fact, while AWS Amplify tried to make it easy for me to set up my application, there were multiple issues with Amplify. For example, the plethora of options combined with the arcane, scattered and sometimes outdated documentation meant that when I set up AWS DataStore on Amplify to help me manage my GraphQL API, I could not even set up a has-many relation due to an inherent quirk of the DataStore software. After

scrolling through the GitHub issues, I came to realise that DataStore was actually hindering my API deployment and thus decided to rollback to the purer choice of AWS AppSync + DynamoDB. This was one of the many weird and frustrating aspects of Amplify. While it helped me auto-generate some of my GraphQL queries, I think that the time wasted on debugging due to Amplify-specific issues and constraints would have been better spent on directly managing an EC2 instance.

One reason why I chose GraphQL (AWS AppSync) is that it appealed to me over the traditional REST API architecture. Rather than setting up multiple endpoints, GraphQL used a more intuitive approach to fetching data. A new technology, it was something I had wanted to learn and it will be useful in the future.

I also spent much more time than expected on user authentication and enabling the guest user. While enabling an authenticated login flow with AWS Cogito was decidedly straightforward, adding separate components to handle the unauthenticated guest user required some deft understanding of React hooks and components. After working with some ungainly props that made the code unextendable and unmaintainable, I decided to implement Redux to store the user authentication state. While the idea behind Redux is simple (a global state accessible anywhere in the React application), the implementation was anything but so. Nevertheless, after some tinkering, I finally understood how Redux worked and implemented it properly.

I decided to implement client-side routing with React Router. I was glad I did this early, as React Router helped me navigate through my application more easily and manage the unauthenticated user's flow. The last thing I did was to set up an AWS Lambda function in Go for backups, with a weekly schedule under Cloudwatch. Go was interesting to learn, being different from Python and JavaScript.

Overall, I enjoyed the process of web development. Designing an entire application from scratch was harder than I thought it would be, but I learnt about core principles such as functional components composition in React, refactoring React code, building a GraphQL API, and managing a database. However, there are certain improvements to be made. On the whole, the UI is rather rudimentary and I had issues choosing the correct components and styles due to my weak understanding of CSS and Material UI. Furthermore, the functionality of the application was much more limited than what I envisioned it to be. Features such as drag-and-drop, arranging the tasks in columns by their properties such as date due or category, error handling for when API calls fail, etc. are some features a proper task management application would have included. I also left the ability to persist Redux state on page refresh unimplemented as I was attempting to prioritise other features first. Nevertheless, this project has taught me crucial skills as resilience when debugging and made me more aware of the need for flexibility and maintainibility when writing code.