**CS673 Software Engineering** 

**Team X - Project Name**

**Software Design Document**

| Team Member | Role(s) | Signature | Date |
| --- | --- | --- | --- |
| Bharat Gogineni | Team leader |  |  |
| Ruiqi Chang | Requirement Leader | *Ruiqi Chang* | 5/28/2022 |
| Chinmay Bhelke | Design and Implementation Leader | *Chinmay Bhelke* | 5/15/2022 |
| Benoît Clemenceau | QA Leader | *Benoît Clemenceau* | 5/15/2022 |
| Taina Conde | Configuration Leader | *Taina Conde* | 5/25/2022 |
| Chinmay Bhelke | Security Leader | *Chinmay Bhelke* | 5/27/2022 |

**Revision history**

| **Version** | **Author** | **Date** | **Change** |
| --- | --- | --- | --- |
| **1.0** | **Group 2 All Members** | **05/29/2022** | **Initial Version** |
| **2.0** | **Group 2 All Members** | **6/12/2022** | Software Architecture, class diagram, and Database Design |

[Introduction](#_heading=h.gjdgxs)

[Software Architecture](#_heading=h.30j0zll)

[Database Design (if applied)](#_heading=h.1fob9te)

[Security Design](#_heading=h.tyjcwt)

[Design Patterns](#_heading=h.3rdcrjn)

[Key Algorithms](#_heading=h.3dy6vkm)

[UI Design](#_heading=h.3znysh7)

[Classes and Methods](#_heading=h.26in1rg)

[References](#_heading=h.4d34og8)

[Glossary](#_heading=h.17dp8vu)

# Introduction

In this section, give an overview of this document, and also address the design goals of your software system.

This document describes the high-level design goals and overall design portrayal of the FollowUp web application. Our top design goals are **usability**, **reliability, compatibility**, **responsiveness, and security.** In this document, we describe the ***Client-Server*** architecture and the ***MVC*** architecture, which are the software architecture chosen to help accomplish these goals**.**

Additionally, this document includes the class diagrams, detailing each component and their relationship to each other. Then, we also address the Business Logic used in this application.

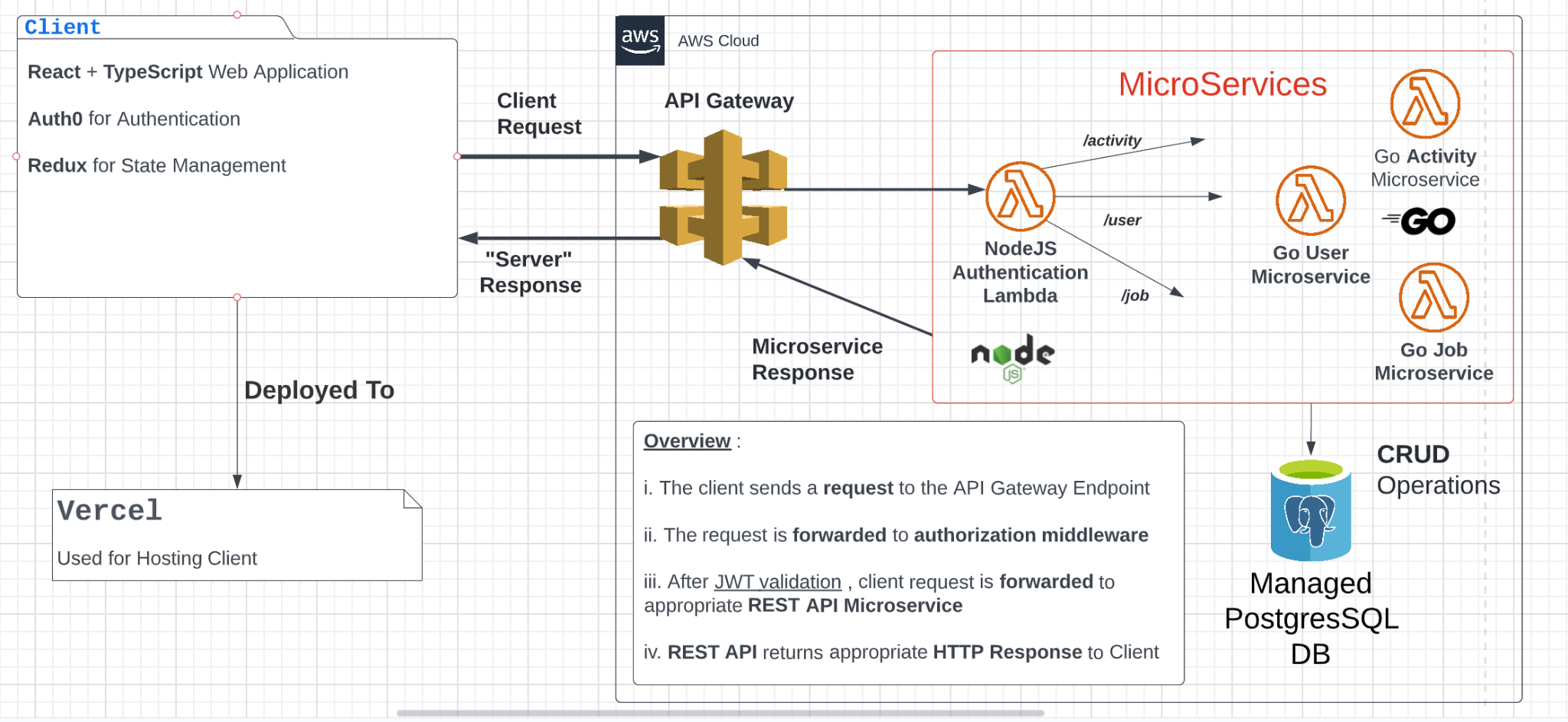
We are using ReactJs as the front-end framework and distributed Go microservices as our backend. In this document, we describe both the UI and the database designs of the application. Furthermore, the design patterns and the security design are also detailed in their respective sections.

# Software Architecture

In this section, you will describe the decomposition of your software system, which includes each component (which may be in terms of package or folder) and the relationship between components. You shall have at least one diagram to show the whole architecture of . The interface of each component and dependency between components should also be described. If any framework is used, it shall be defined here too.

| **Software Architecture**  **FollowUp** implements a ***Client-Server*** Software Architecture. The **Client** is a ***React-TypeScript*** static ***Single Page Web Application*** deployed to **Github Pages**. The **Server** is an ***Event Driven Microservices Architecture*** hosted on **AWS**.  The clients do not share any resources between other clients and the clients send requests to the individual backend microservices which live in the cloud. We use individual microservices to handle various application tasks - ex: we have a separate microservice to handle User Job Requests and a separate microservice to handle User Authentication. All of the serverless CRUD REST API Microservices are Go Lambda functions and the Authorization Middleware API is a NodeJS Lambda function.  The microservices live behind an **API Gateway** which routes client requests to the backend. Between the **API Gateway** and the **CRUD Microservices** , we have an **Authorization microservice** which validates users and only routes requests to the backend if user credentials are valid.  For user validation , we use **JWT Tokens** and a **RSA-256 Encryption Algorithm** to generate the public and private keys to encrypt and decrypt the JWT’s generated by valid users. We use a federated social identity provider - **Auth0** - to handle a lot of this authorization logic and token generation.  User data for the application is stored on a **PostgreSQL Database** hosted on the cloud. CRUD Operations on the DB are only possible through authenticated users who have a valid account and credentials through the various Microservices.  For the frontend client side application , **Redux** is used for **state management** and **optimistic caching** is implemented to avoid redundant network requests. Specifically , **Redux Toolkit** and **RTK Query** is used to achieve this. Redux uses an architecture similar to **MVC**.  The MVC pattern decomposes the software system into three components: the Model, the View, and the Controller. The Model contains the app's data and logic. The View does the presentation of the model to the user. The Controller connects the Model and the View through actions, which minimizes the dependency between them. Redux works in a similar way. The Store contains the state (data), the Reducer handles the logic to change the data, and the Components just receive the data and render it.    The client is deployed on a public URL hosted on **Github Pages** and can only be accessed once the user is logged in with valid credentials. **Authorization** and **user management** is handled by **Auth0** and integrated using **JSON Web Tokens** for the client-server connections.  Our application implements a **CI/CD** pipeline *(Continuous IntegratIon Continuous Deployment)* to deploy backend changes using **Vercel** for deployments and we use a YAML configuration file to maintain and deploy our microservices architecture. |
| --- |

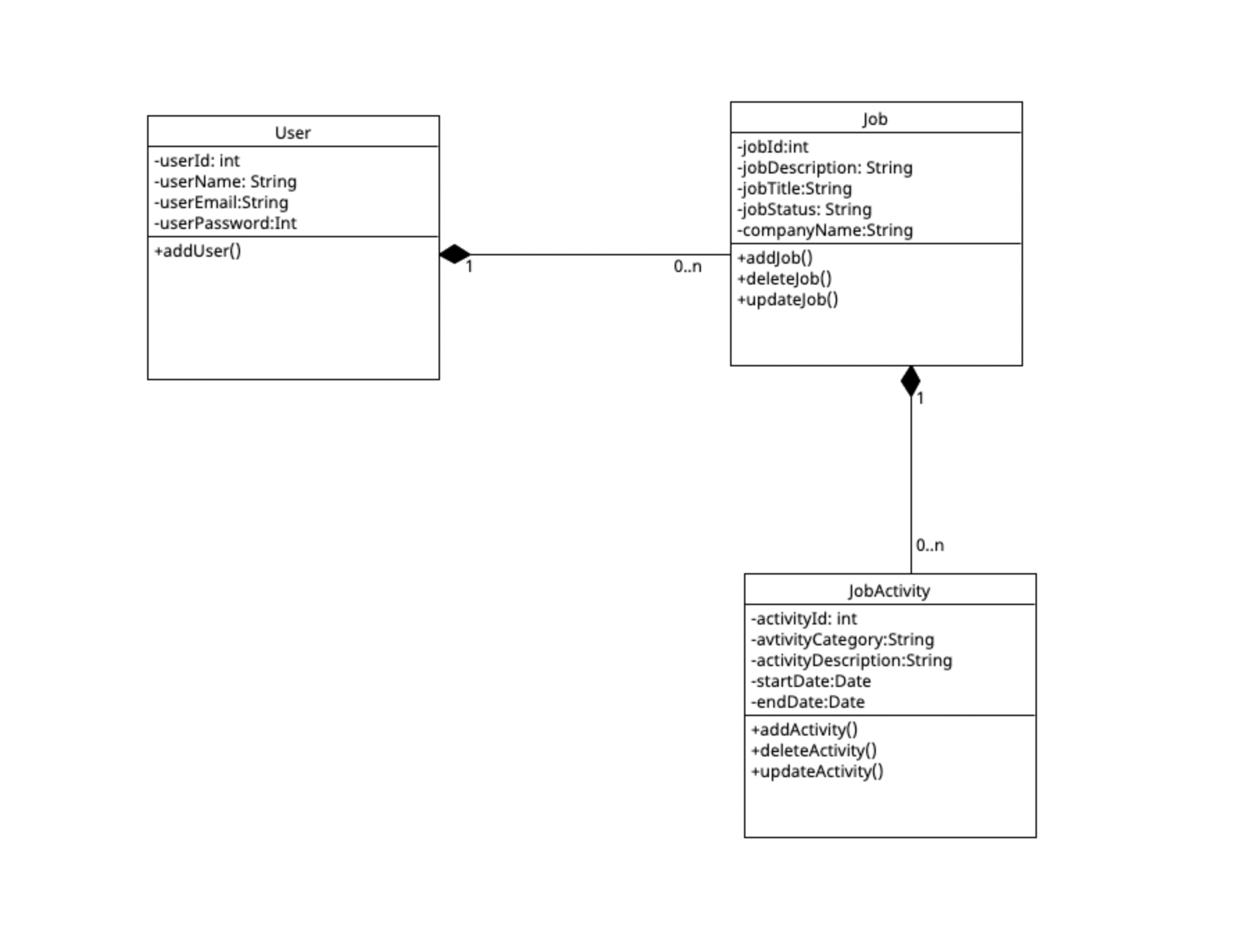
[**Software Architecture Diagram**](https://lucid.app/lucidchart/1d8baa49-51c5-4722-bd8f-7f44a54a6c46/edit?invitationId=inv_04306066-2dab-49a9-b5d7-850b56af1d5c&page=0_0#) **:**



# Class Diagram

In this section, you will provide a detailed description of each component (or package) and use one or multiple class diagrams to show the main classes and their relationships in each component.

There are three main class: User, Job, and Job activity. The user class is for login on the landing page and show the right job cards of the user on the landing page.Job class is for add, update and delete the job information. The Job activity class is for the activities of the each job. User also can add, delete and update job activities for the job. About page introduce the basic concept of FollowUp project. The app structure was so that the jobs live in a Database and do not exist without a userID field , so every job belongs to a user and every user can have 0 to multiple jobs.

Class Diagram

# UI Design

In this section, you can describe your UI design

The following UI mockups are the Desktop and Mobile versions of FollowUp UI designs, respectively. All pages have a navigation bar with the app's logo. The navigation items depend on which page the user is on.

When the user first visits the app's website, the page the user sees is represented on the Desktop1 frame and we call it the landing page. On the landing page, the user can see the value proposition, some information about FollowUp, and the user can also log in.

Next, the user will be redirected to the auth0 authentication page. Desktop2 and Desktop 3 frames only simulate that process.

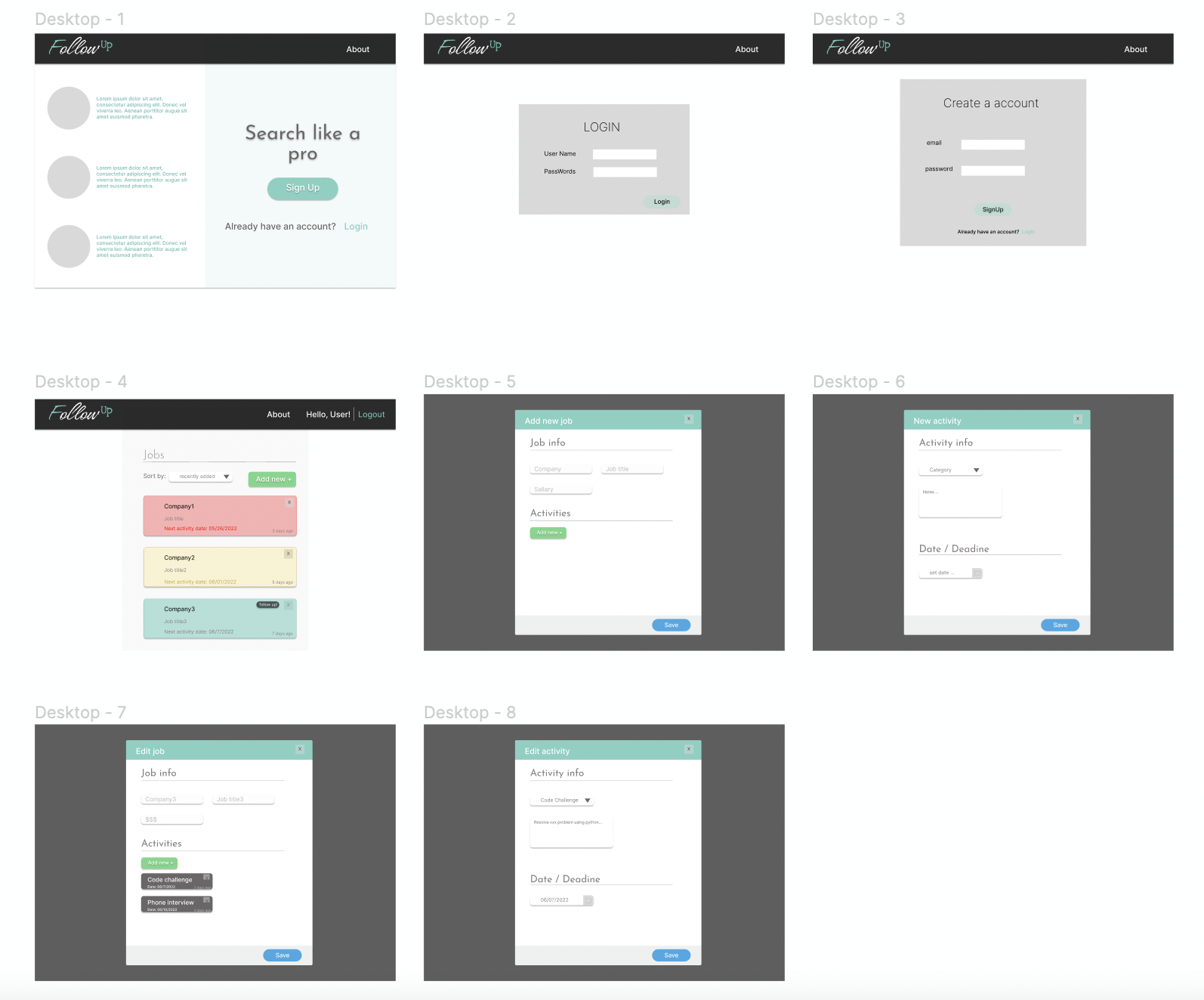
After the user logs in, the navigation bar will display the logout button, the user picture, and the message "Hello, User!", where "User" is replaced by the user given name retrieved from auth0. On this page, there is an "add new" button that lets the user add job cards. Additionally, the user can view his/her job cards (if the user had added them before) or the user sees no cards. This page is represented by Desktop 4.

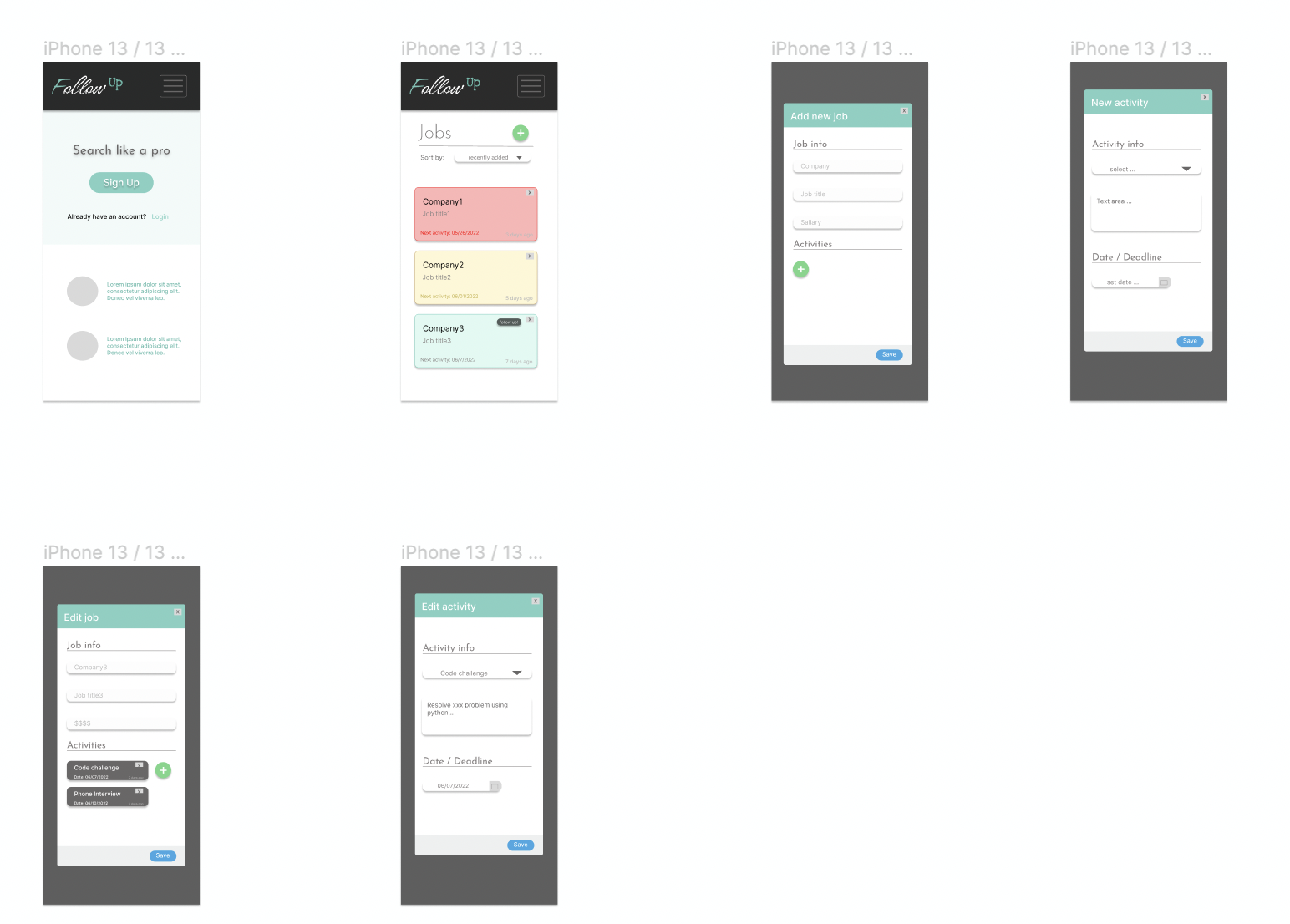
When the user clicks the "add new" job button, the popup on Desktop5 opens up. The user can enter the company name, job title, and add some activities. Regarding the salary, we are still deciding if we are going to use an API to give a salary range to the user according to the company and job title or if we are not going to put a salary field at all.

When the user wants to add an activity to the job and clicks the "add new" button on Desktop5, the popup on Desktop6 opens up. We are still deciding on the input fields, but for the moment, the user can enter the activity category, make some notes and set a start date and an end date for the activity. Not all of those fields are represented on Desktop6. We plan to work on that on the next iteration.

For the moment, when the user clicks on the job card, an popup opens up. It is populated with that card information and the user can edit any information he/she wants. That is represented by Desktop7. However, we are planning to change that logic. The user would click on the card and it would expand like an accordion. The expanded part would show the other information about that job and show the activities related to that job. The team plans to change the UI designs on the next iteration.

Lastly, Desktop8 shows the popup that opens up when the user clicks on a specific activity on the job card. It is the same popup on Desktop6 populated with that activity information.





# Database Design (if applicable)

We are using a **managed Postgres Database** hosted in the cloud to store our user and application data. The database will scale according to usage and has disaster recovery and fault tolerance mechanisms in place.

For the **application logic** , there is a table to store **Jobs** a user is tracking as well as **Activities** that are associated with jobs. We are decoding the ***JWT Access token*** in the backend to extract the user email ID for the user items and using this as a composite key to track unique items.

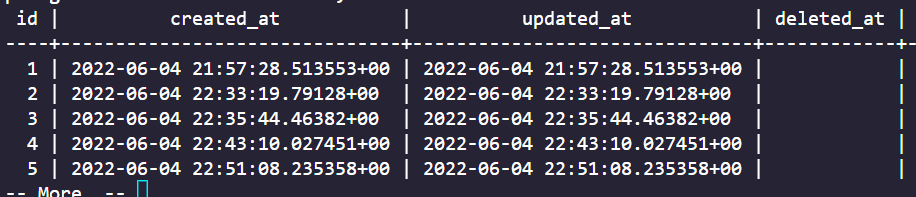
Job Table Schema



Activity Table Schema



We are using an **ORM** called **Gorm** for **Go** to abstract some of the logic and for every record the following fields are automatically generated at time of record insertion :



# Security Design -

In this section, you shall describe any security design in your software system.’

**Client Side**

* We use Auth0 for Authorization for Users on the client Side
* Created an Auth0 Provider and Wrapped our Frontend Web App Around it and integrated Authorization - users can only login if they have a valid account and credentials
* The Web App is only accessible for validated users and the components are hidden using conditional logic unless the user has authentication credentials

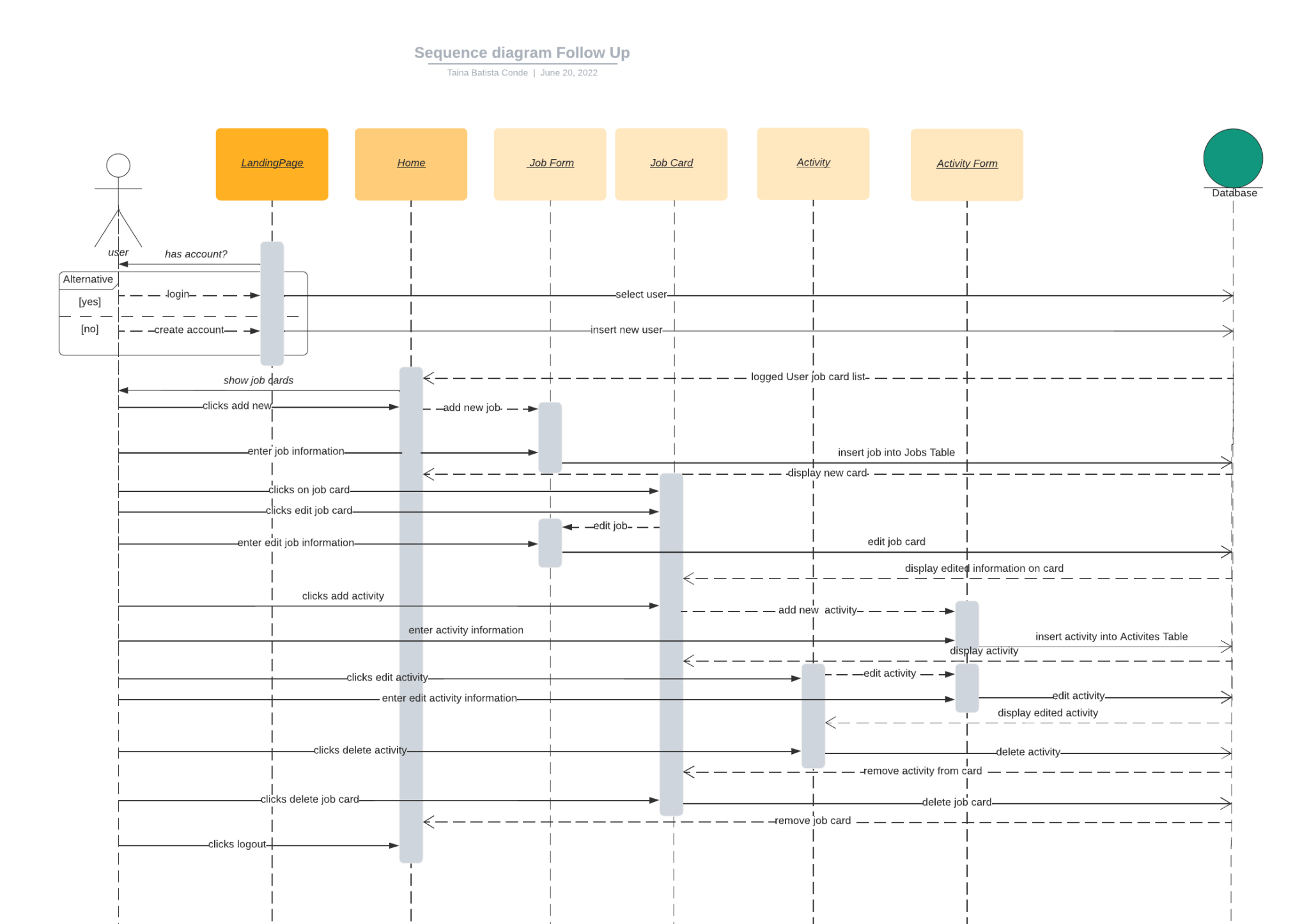
**Backend - Cloud Microservices architecture**

* We have independent Lambda functions that act as Microservices for our Application logic
* All of the Microservices live behind a **NodeJS** Authorization middleware serverless function.
* Authenticated users pass the encoded JWT and request body to the Middleware, and then our middleware will route the request body to the backend once the JWT is validated
* We have Microservices to handle CRUD Operations for the following entities - Users , Jobs , Activities

# Business Logic

In this section, you shall describe any key algorithms used in your software system, either in terms of pseudocode or flowchart, or sequence diagrams.

The following sequence diagram illustrates interactions between objects in the project and the sequence in which those interactions occur..



**Microservices architecture** **Benefits** over a **Traditional Web Server**

* We have **independent Lambda functions** that act as **Microservices** for our Application logic
* All of the Microservices live behind an **Authorization middleware** serverless function.
* Authenticated users pass the **encoded JWT** and **request body** to the Middleware, and then our middleware will route the request body to the backend once the JWT is validated
* We have Microservices to handle CRUD Operations for the following entities - **Users , Jobs , Activities**
* The relationship between these entities can be referenced in the class diagram we have included
* Several **benefits** over a **traditional Backend Web Server Architecture** - our application is capable of **scaling to millions of users without any system or code refactoring**
* Individual microservices getting overloaded will not affect the performance of other API operations. For example if the Create Jobs functionality is under heavy stress and the Create User API is rarely used , we will only need to scale individual services effectively saving on cost and performance
* We will save significantly on costs because we do not need a server running 24/7 waiting for users to send requests

**Optimistic Caching :**

* User makes Network Requests and Fetches all Records from DB
* All User Records are Stored in Cache
* User makes PUT Network Request to add Record to DB
* Local Cache is Updated with New Item sent to DB instead of making redundant Network Request to Backend

| **Optimistic Caching PseudoCode :**  *// When user is on Home Page , they see all records from DB*  **let cache = [ ]**  *// empty cache array*  **const data = server.get()**  *// network request to pull records from DB*  **cache = data**  *// set cache to be the data we pulled*  *// Now User wants to add a record to DB*  **const record = server.put(record)** *// network request to add single record to DB*  *// If request to add record is successful , update local Cache with record*  **cache.push(record)**  // When User Goes back to Home Page , they will see all records from DB (including latest)  Thus , we save an extra network Request from the backend and improve the user experience |
| --- |

# Design Patterns

In this section, you shall describe any design patterns used in your software system.

1. **Higher Order Components**

The client side web application for FollowUp makes use of Higher Order Components. A Higher Order Component (in terms of our ReactJS+TS web app) is a function that takes a component and returns another component. We make use of Higher Order Components to share functionality between components and also to pass important props and data to components.

1. **Publisher - Subscriber**

Publisher and Subscriber or Pub-Sub is a design pattern where senders do not explicitly send the message to the receiver but rather send the messages to a publisher/topic. The subscribers or consumers of the message then subscribe to that topic and consume the message from the topic.

We use a similar Pub-Sub model in our application to send and consume dynamic data for a user in our various components. When a network request is made , we send the results from the request to a Global Store (Redux Store). The components in the application are logically checking for data to appear in the global store. Once the global store is populated , it triggers a code update of all components that were consuming data from the particular **‘topic’**.

# Any Additional Topics you would like to include.

# References

React Standard:

<https://reactjs.org/docs/design-principles.html>

AWS

<https://aws.amazon.com/>

Redux MVC:

https://rangle.io/blog/how-react-and-redux-brought-back-mvc-and-everyone-loved-it/

# 

# 

# Glossary

| **Acronym** | **Meaning** | **Description** |
| --- | --- | --- |
| **HOC** | Higher Order Component | A higher-order component (HOC) is an advanced technique in React for reusing component logic. HOCs are not part of the React API, per se. They are a pattern that emerges from React’s compositional nature. |
| **AWS** | Amazon Web Services | Amazon Web Services (AWS) is the world’s most comprehensive and broadly adopted cloud platform, offering over 200 fully featured services from data centers globally. |
| **EC2** | Elastic Cloud Compute | A virtual server that assists users to run numerous applications (called instances) on the AWS cloud infrastructure. |
| **OS** | Operating System | An operating system (OS) is system software that manages computer hardware, software resources, and provides common services for computer programs. |
| **MVC** | Model- View- Controller software architecture | Model–view–controller is a software design pattern commonly used for developing user interfaces that divide the related program logic into three interconnected elements. |
|  |  |  |