System Design Document

CollabCloud

Table of Contents

Introduction

During the initial architecture planning of CollabCloud, it was clear to us that the platform had many moving parts within the front-end, back-end and the database. As a social network platform, we are expected to serve multiple concurrent users and have a system set in place to make real time updates based on likes and messages. Because of these reasons and more, we decided that a three-tier architecture works best for CollabCloud. The following link from IBM explains a bit more in depth: Three Tier Architecture

Three Tier Architecture

The three-tier architecture is made up of (as the name suggests), three layers of components each responsible for its own part of the application. The following section shows our Architecture Diagram and describes each of these layers, explaining the trade-offs the technologies and the uses of each layer

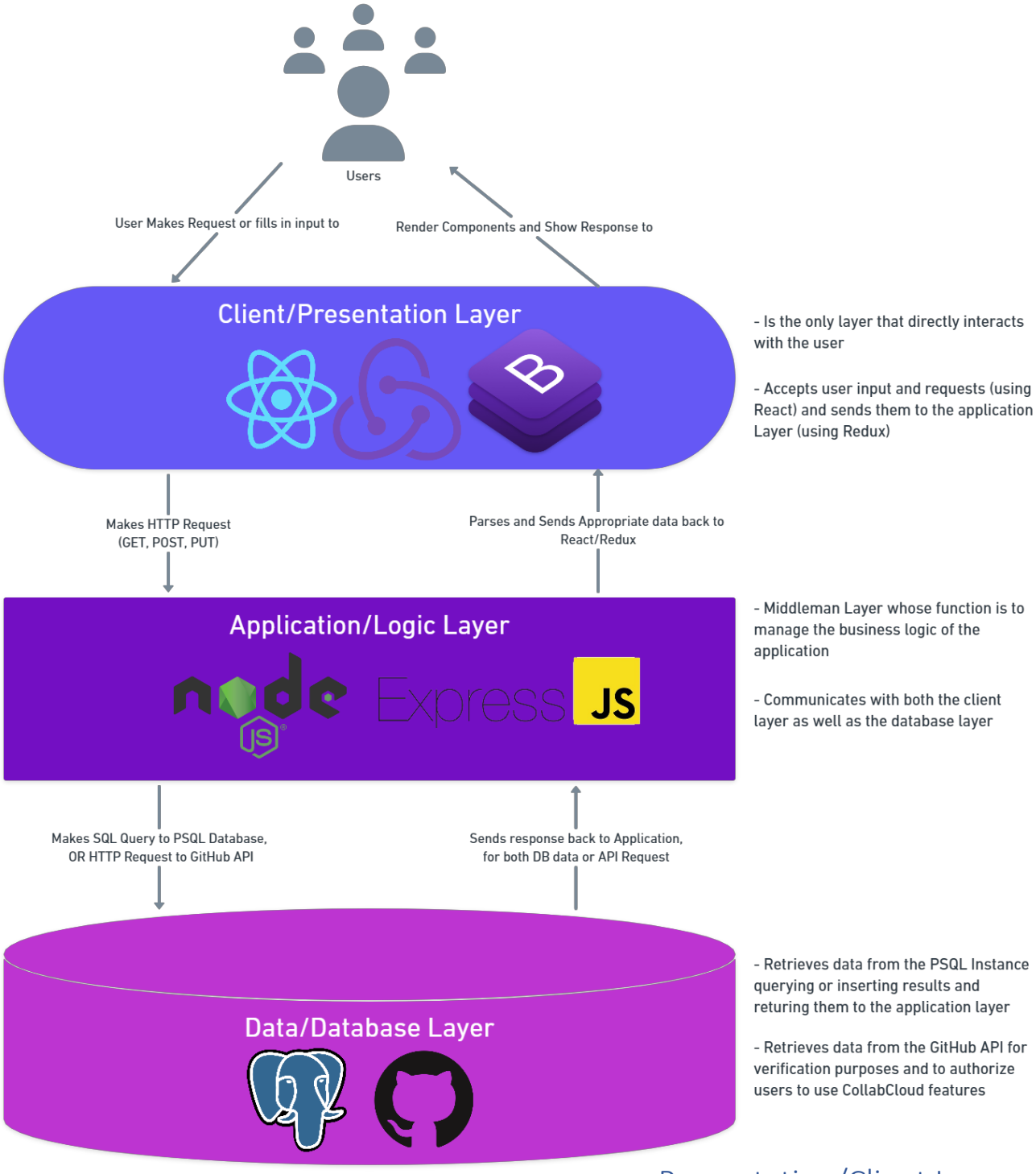
Environment Interaction

The foundation of CollabCloud resides in 2 folders: the *client & server* folder. These 2 folders are all that are required to deploy a local instance of collab cloud. Any OS with Node.js installed is able to run the CollabCloud instance, along w/ the Node.js server a PostgreSQL database is also required to act as a database. CollabCloud is set up to host on 2 ports of a local server; the server will run on **localhost:5000** and the client (React) instance will run on **localhost:3000** where the user will be able to interact with.

CRC Cards

The CRC cards associated with this System Design Document are located in a file called:

CRC Cards.pdf



Presentation/Client Layer

This is the topmost layer of the application and it is the only layer that the user is able to view and interact with. The main function of this layer is to accept user input and display output based on the request

- In our system, this client tier can be represented by our front-end framework **React.js** which will be written in JavaScript. Our React instance will accept user inputs and will display outputs using controlled state components

- Our React instance will not communicate directly with our database; it is an independent component and instead will only be able to send requests to our application layer (which we will see next). As such, the functions of our client components are: display data to the user, accept input from the user, and make requests to the application layer

- A few benefits of abstracting the client tier includes: allowing the development of the front-end independent of the data layer, being able to simply swap out updates in the front end without affecting or changing anything in the back-end, and providing ease of maintenance of the overall code base. In the case of system failure of the database or server, the client will still be able to run on its own since it runs in its own instance

The Application/Logic Layer

This is the middle layer of the application and its function is to manage the business logic of the application. This layer has access to both the client and the data layer therefore it acts as the middleman of the application, taking requests, fetching the data, and returning it to the client. This layer is able to manage requests such as the following: Sending user data to the database to register a new user, reference email and password in the database to allow a user to login to the platform, accept messages from users and send them to other users etc.

- In our system, the application layer will be handled by **Express.js** an API middleware to allows the application to receive requests and send responses, Express will be sitting on top of **Node.js,** a JavaScript run time environment powering the backend server-side scripts

- Using Express and Node, the application layer will be able to take requests from our React instance, process them for queries or API requests, and then send them to our data layer, from there we process the response and send it back to the front end

- The value of a single application middle-man is that it allows us to abstract out our front-end and database layer allowing our developers to work on those components independently and then later, connecting them together using our application layer. In the event of a system failure (e.g. DB crash), the application

layer is able to process the results to the front end and let the user know that the DB is no longer in service

The Data/Database Layer

This layer is “bottom” most layer of our system. The primary function of this layer is to house all of the data required for our application; this could be anything from User data information to Repository information. This layer accepts requests from the application which takes that request queries the information and sends it back to the application layer

- In our system, we have 2 primary sources of data storage. The first is our **PostgreSQL** database where we store data that is directly relevant to our CollabCloud platform this includes but is not limited to: user information, posts, messages, repository meta data etc. The second source of data we will utilize is the GitHub API

- The primary purpose of utilizing the GitHub API is for verification purposes. Before a user is able to register an account on CollabCloud, they must authorize their account to give CollabCloud access to their repositories (Using the GitHub OAuth feature). After authorizing their accounts, they are able to utilize the full set of features available on CollabCloud

- Abstracting the data layer allows us to enforce security policies independent from the front end or back end. When dealing with user data we want to make sure we are following the best security practices. In the case of a server or client crash, CollabCloud’s will be well preserved as the DB is running its own instance away from the client or server