System Design Document

FitHub

Last update: 10/21/2022

Environment

The entire team runs on a Mac OS based laptop however our application is runnable on all different OS systems as long as all of the requirements highlighted on the readme file are met and installed properly. To set up our web application we use react version 18.2.0 to design our front end and to build our interfaces based on UI components. We furthermore utilize tailwind to assist in our styling and to expedite the process of adding desired stylistic features wherever desired. We also use Express.js to build our RESTful APIs in combination with Node.js version 16.17.0 to help manage servers and routes. We utilize MongoDB to store our data as our document database to build flexible schemas and to best match our agile development methodology. We utilize npm version 8.15.0 to serve as our package manager to place modules in place to allow node to find them and manage dependency conflicts. Furthermore we utilize axios version 0.27.2 to make HTTP requests from the browser and handle the transformation of req and res data. Lastly there is no constraints on the ip address or network configuration of the user as we have included ip address 0.0.0.0 to allow our service to bind to all network interfaces.

Architecture

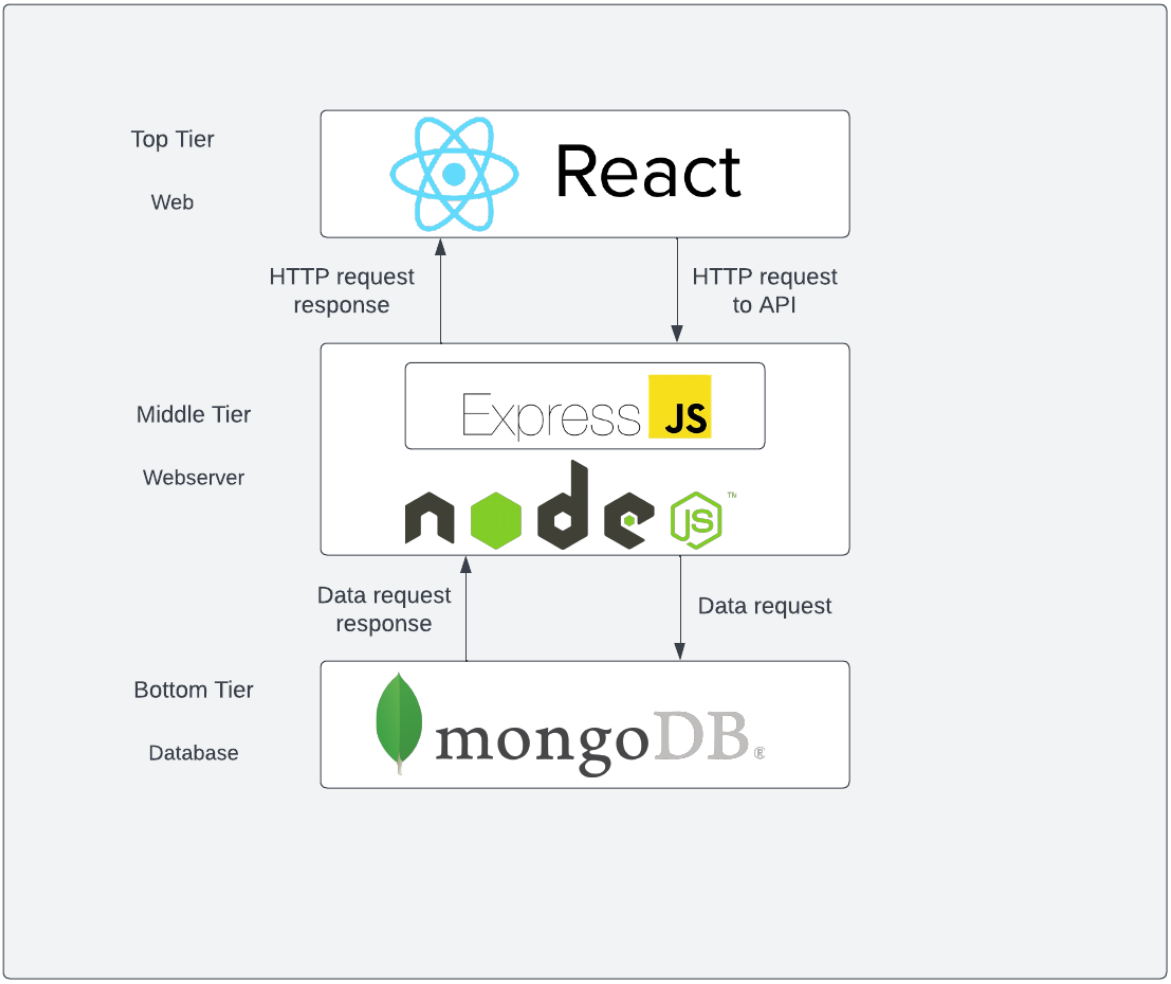
FitHub's architecture is the typical MERN stack application architecture, a three-tiered approach. We use MongoDB as our document database, a NoSQL database hosted by MongoDB. We use the Express.js web framework to handle routing of requests to the API. We use React as the client-side JavaScript framework, to render the website on the client. We use NodeJS as the language for the webserver.

This article provides more details on the Three Tier architecture (note that naming conventions might differ slightly to what we use, i.e. Presentation vs. Web and Application vs. Webserver, these are terms used generally vs. terms specific to MERN stack usually):
https://www.ibm.com/cloud/learn/three-tier-architecture

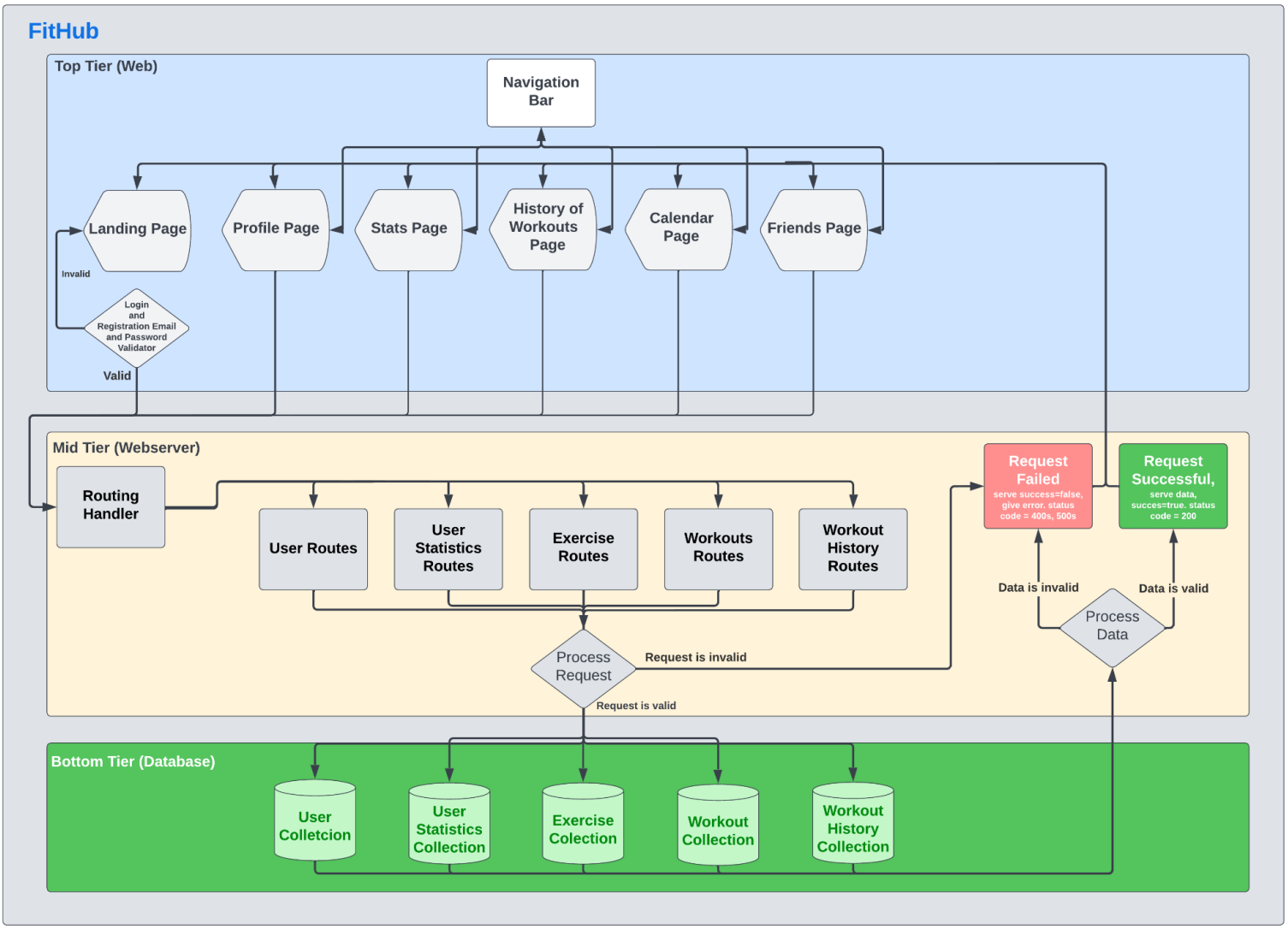
The bottom of the three tiers of the architecture is the database layer, which is using MongoDB. This will store all the user and app data on the Mongo server's, where our database is hosted. We will access this data via the middle tier, which is comprised of ExpressJS and NodeJS. NodeJS is the language which we are using on the webserver which handles all the backend processing. Inside the webserver, we are using the NodeJS library called ExpressJS, to handle routing of requests to the API. In this tier, we process requests to the backend, get/set data from the MongoDB via requests, and honor and serve these requests to the sender. The top tier of the architecture is React, which is run on the client. React is the JavaScript library chosen for this project, which will be used to render the components of our website on the client side. This

library handles DOM routing, which is swapping components to display different portions of the website, and also handles state changes. State changes can happen when the user interacts with the website, where a request might be fired to the backend. The top tier will interact with the backend solely through making HTTP requests to the API endpoints to get, post, put, delete, etc. The result from these requests will then be utilized by React to display it on the client, so that the user is able to see it.

A diagram of the architecture can be seen below:



System Decomposition Design Diagram



Starting from the top tier our web the navigation bar is able to direct/redirect to each of our pages in the top tier. All pages are able to make post and get requests to the mid server tier. This post and get request will be handed off to the routing handler and that will then distribute the requests to whatever route we have. These routes will check if what the user is giving you is valid, if it is valid it will give request successful else it will give request failed. If valid it will send a request to the database and that data will be processed by process data. The data will be processed if only the data returned from the data base makes sense. If you are searching to make a new friend, you will check if there actually is a friend that is valid for you to search. give the valid/invalid status to the frontend.

Process data will check if the data returned aligns with the request.

CRC Cards

Class Name: `user`	
Parent Class N/A Subclasses friend	
Responsibilities: ● Search for a friend ● Update profile ● Send friend requests ● Accept friend requests ● Register ● Create workout ● Schedule workout ● View workout Streak ● Login/Logout ● Track Progress ● Check workout history, weight, height	Collaborators: ● friend ● calendar ● workout ● exercise ● history

Class Name: `friend`	
Parent Class User Subclasses N/A	
Responsibilities: ● Schedule workout with other users	Collaborators: ● calendar ● user

Class Name: `workout`	
Parent Class N/A Subclasses N/A	
Responsibilities: ● Create personalized workout ● Add/Delete exercise	Collaborators: ● user ● calendar

• Start/End workout • Log workout	• exercise • history

Class Name: `exercise`	
Parent Class N/A Subclasses N/A	
Responsibilities: • Add/delete sets • Add/delete repetitions • Set weight	Collaborators: • user • workout

Class Name: `calendar`	
Parent Class N/A Subclasses N/A	
Responsibilities: • See upcoming workout • Schedule workout with/without friends	Collaborators: • user • friend • workout

Class Name: `history`	
Parent Class N/A Subclasses N/A	
Responsibilities: • See weight history • See height history • See BMI History • See previous workout	Collaborators: • user • workout