International Institute of Information Technology, Bangalore

(IIIT Bangalore)

# Software Production Engineering Project Report

**FinTrack - Shareable Investment Watchlists!**

Under the Guidance of Prof. B. Thangaraju Teaching Assistant: Neha Kothari

Jasvin James Manjaly Thakur Devendrasingh MT2021059 MT2021146

# TABLE OF CONTENTS

**Table of Contents**

1. [Abstract 4](#_TOC_250008)
2. [Introduction 5](#_TOC_250007)

Overview 5

Features 6

1. [System Configuration 7](#_TOC_250006)

Host System Configuration 7

Project Technology Stack 7

DevOps Tools 7

1. [Software Development Life Cycle 8](#_TOC_250005)

Installations 8

[Testing 11](#_TOC_250004)

Source Control Management 15

Containerization with Docker 16

Docker Compose 18

Ansible 20

Continuous Integration: Jenkins 24

Ansible Playbook 29

PaaS Deployment with Netlify & Heroku & Github Actions 32

Monitoring - ELK Stack 38

1. Experimental Setup 41

Functional Requirements 41

Non-Functional Requirements 41

Architectural Diagram 42

Code Walkthrough 43

1. [Result and Discussion 59](#_TOC_250003)
2. [Scope for Future Work 65](#_TOC_250002)
3. [Conclusion 66](#_TOC_250001)
4. [References 67](#_TOC_250000)

# Abstract



***“Share your Investment Watchlists now!”***

The youth of our country are getting more and more interested in the securities market. For a newbie, it would be more helpful if they can get an overview of the correct investments by looking at the watchlists of experienced players in the ring.

There is a need for a platform where users can share their investments easily with their friends or relatives.

Typically any investment or investment tracker platform only lets you create watchlists for your own account and not let them be shareable.

**Fintrack** is the exact solution for this.

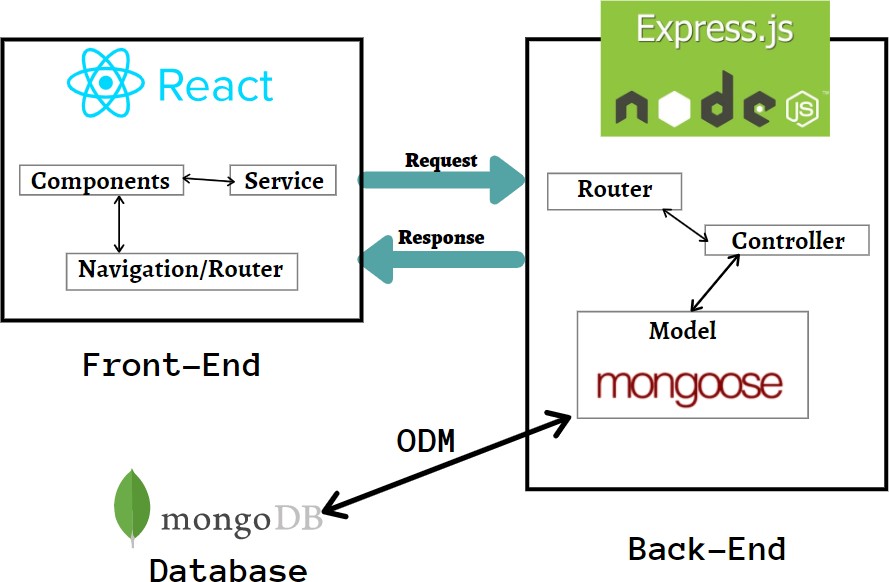
Here users can create their watchlists for stocks and mutual funds (MFs) as per their investments and share it with anyone they would want to - with ease!

# Introduction

## Overview

**FinTrack** is a website that allows users to create their watchlists for stocks and mutual funds (MFs) and share it anytime, anywhere and with anyone they would want to.

TechStack used → MERN (MongoDB, ExpressJS, ReactJS, NodeJS)



## Features

### Login/Signup

Account creation and login for users.

### Home Page

Shows all the watchlists of the user.

### Search Instruments

Search for stocks and mutual funds (MFs) on a search bar and add them to watchlists. A user can create multiple watchlists so there will be an option displayed to the user to determine which watchlist to add the current security to.

### Create Watchlist

Users can create any number of watchlists they want with no duplicate entries. A watchlist can be a stock or MF watchlist but not both.

### Share Watchlist

Each watchlist will have a share button that will send a publicly shareable link to anyone that visits that URL.

### View Watchlist (Single Watchlist View)

This will show the specified watchlist’s instruments. The view displayed when a user opens the shareable watchlist URL.

### Delete Watchlist

Users can delete their watchlists anytime they want.

### Delete Instrument

Delete a stock/MF from a Watchlist.

1. **View live stock/mutual fund price** → Each stock/MF has a URL to the TickerTape page for the corresponding stock/MF which will show the dynamic rate of change of stock/MF prices.

# System Configuration

## Host System Configuration

**Operating System:** 20.04.4 LTS (Focal Fossa)

**CPU and RAM:** 8 core processor with 16 GB RAM

**Kernel Version:** Linux version 5.13.0-40-generic

## Project Technology Stack

**Frontend:** React (HTML, JSX, JavaScript, Bootstrap)

**Backend:** Express (NodeJS, JavaScript)

**Database:** MongoDB (No-SQL)

**Cloud:** MongoDB Atlas, Netlify, Heroku

**Build Tool:** npm

## DevOps Tools

**Source Control Management:** Git/Github

**Continuous Integration:** Jenkins/Github Actions

**Containerization:** Docker/Docker Hub

**Container Orchestration:** Docker Compose

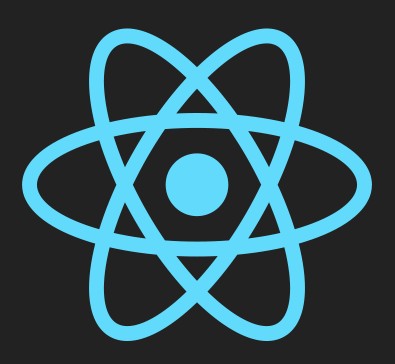
**Testing:** Supertest (Backend) + React Testing Library (Frontend) **Continuous Deployment:** Ansible and/or Heroku & Netlify **Logger:** Morgan

**Monitoring:** ELK Stack (Elastic Search, Logstash, Kibana)

**Secrets Management:** Ansible Vault

# Software Development Life Cycle

## Installations



**React**

React is a free and open-source front-end JavaScript library for building user interfaces based on UI components. It is maintained by Meta and a community of individual developers and companies.

### Update local before installing:

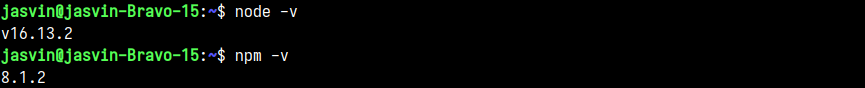
Keep the local packages and softwares updated.

sudo apt-get update

### Install NodeJS and NPM:

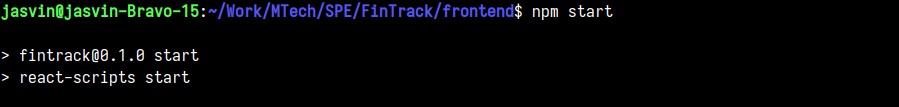
Inorder to run React, Node environment shall be installed before starting,

sudo apt-get install nodejs sudo apt-get install npm



**React App**

npx create-react-app fintrack





## Express

Express.js, or simply Express, is a back end web application framework for Node.js, released as free and open-source software under the MIT License. It is designed for building web applications and APIs. It has been called the de facto standard server framework for Node.js.

### Express App

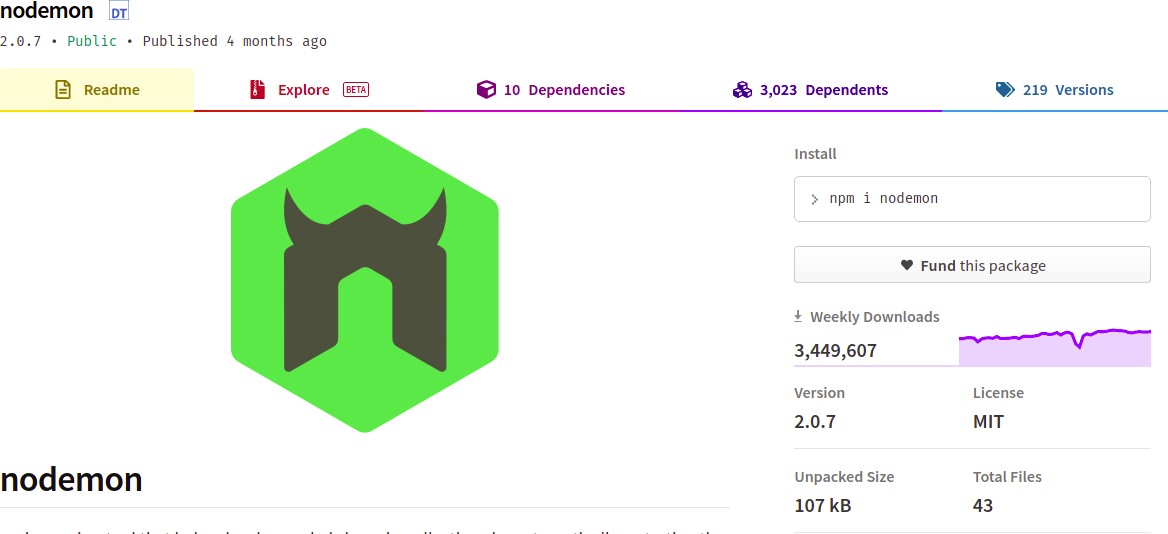
Initializing node project with the default config:

npm init -y

*-y* automates the config with node default values i.e without going through the interactive process.

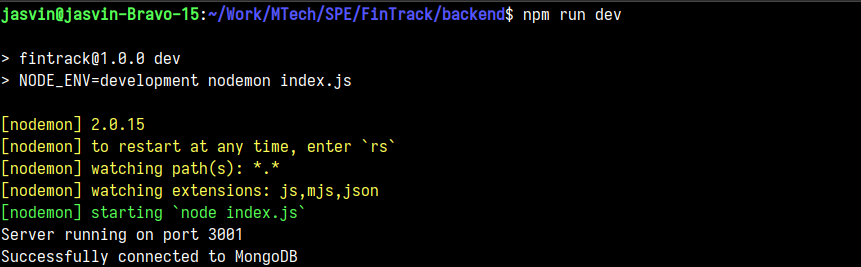
### Running the Node Application locally:

node server.js

**Running the Node Application locally with nodemon:**

Nodemon automatically restarts the node application whenever a file change in the directory is detected.

npm install nodemon nodemon server.js

The server is setup to run on localhost:3001

## Testing

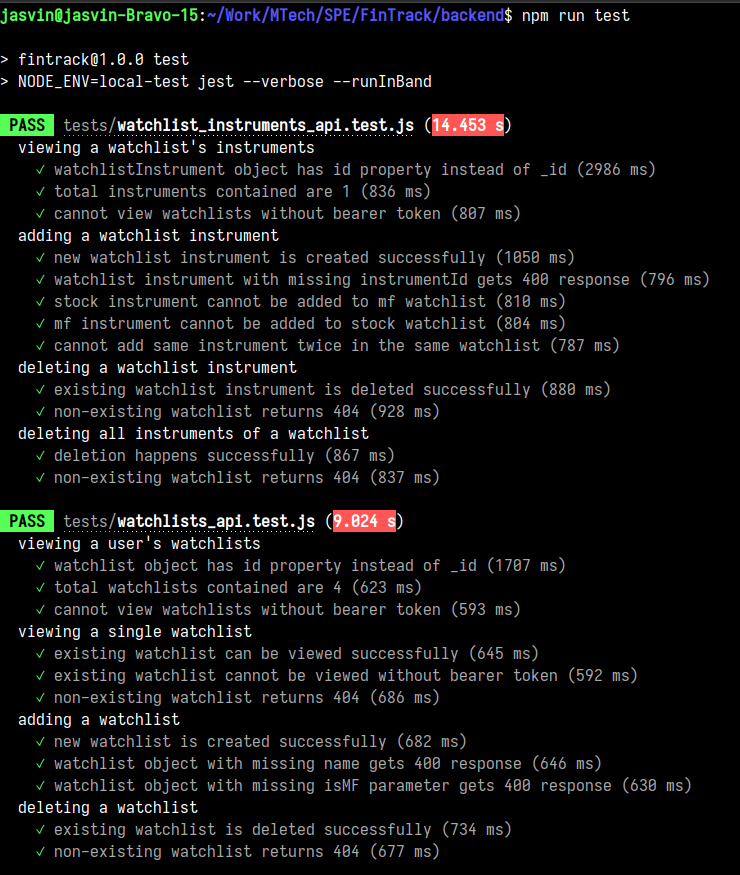
* + 1. **Backend Testing (Supertest)**

Supertest provides a high-level abstraction for testing HTTP, while still allowing you to drop down to the [lower-level API](https://visionmedia.github.io/superagent/) provided by superagent.

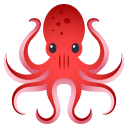
Here is a snippet of how to write tests with Supertest, in this case, we are testing the

*/users/* API. Similarly we have tested the */watchlists* and */watchlistInstruments* APIs.

Note that these files must have the extension *.test.js* in order for Jest to pick up these files as testing files. After this you can run the tests by running the command *npm run test*



For backend testing we have created a testing database on MongoDB Atlas which has its own URI separate from the production database. This ensures that testing doesn’t interfere with the production database at any point in time and we can do operations like deleting entire entries from collections in the testing database in order to maintain consistency between tests.



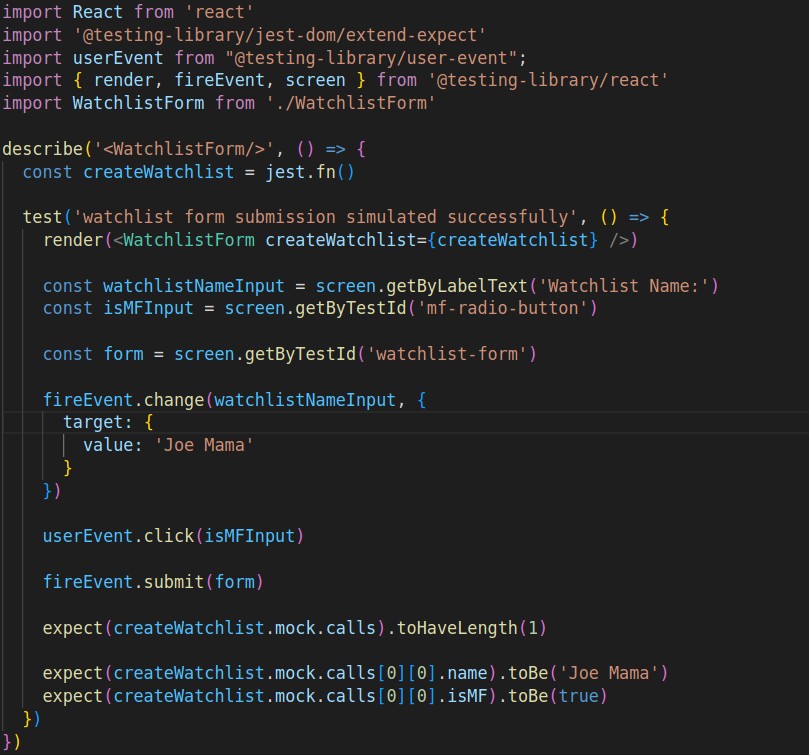
## Frontend Testing (React Testing Library)

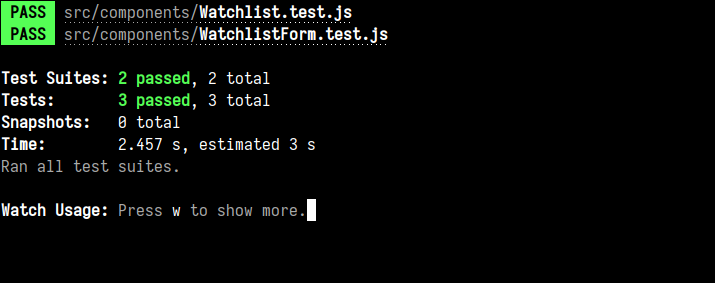
The React Testing Library is a very light-weight solution for testing React components. It provides light utility functions on top of react-dom and react-dom/test-utils, in a way that encourages better testing practices. Its primary guiding principle is:

“The more your tests resemble the way your software is used, the more confidence they can give you.”

So rather than dealing with instances of rendered React components, your tests will work with actual DOM nodes. The utilities this library provides facilitate querying the DOM in the same way the user would. Finding form elements by their label text (just like a user would), finding links and buttons from their text (like a user would). It also exposes a recommended way to find elements by a data-testid as an "escape hatch" for elements where the text content and label do not make sense or are not practical.

Here is how we have tested the WatchlistForm component with just the frontend, no backend involved.



We run these tests using the *npm run test* command as well,



## Source Control Management (SCM)

Source Control Management is used for tracking the file change history, source code, etc. It helps us in many ways in keeping the running project in a structured and organized way.

Repository Link: <https://github.com/james-jasvin/FinTrack>

The frontend is created in the *frontend/* directory and the backend is created in the

*backend/* directory. Initializing the project with git:

git init

git remote add origin https://github.com/james-jasvin/FinTrack.git

Workflow:

git add <files>

git commit -m "commit message" git pull origin master

git push origin master

The code is first pulled before making a push to make sure that our project is in the latest stage and to avoid merge conflicts. For the above, the pull and push is done on the “master” branch.

Working on a feature/issue:

git checkout master

git checkout -b "<your\_branch\_name>"

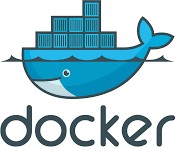
After creating a branch, required changes are done and a pull request to the main is created.

git add <files>

git commit -m "commit message" git pull origin master

git push origin master

Then merge the pull request from Github.



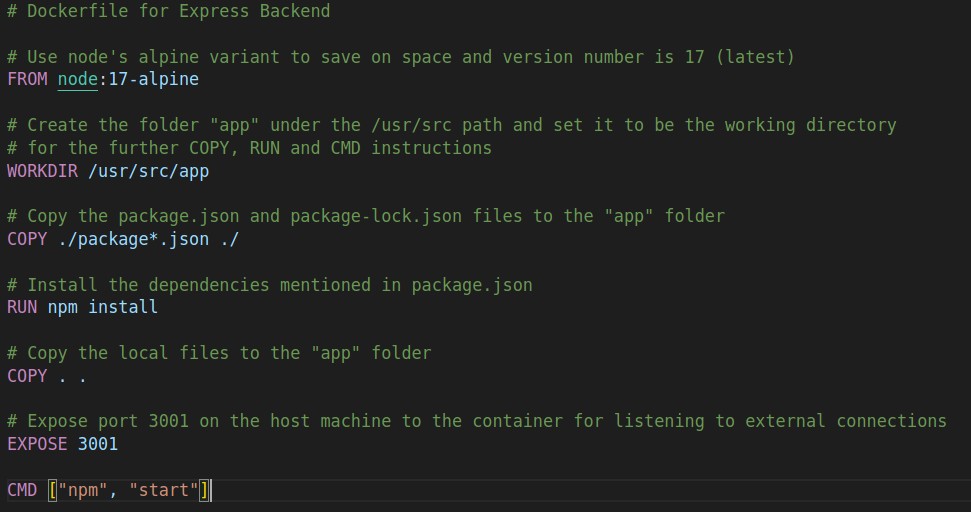
## Containerization with Docker

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.

With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker’s methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

Docker builds images automatically by reading the instructions from a Dockerfile -- a text file that contains all commands, in order, needed to build a given image. A Dockerfile adheres to a specific format and set of instructions which you can find at Dockerfile reference.

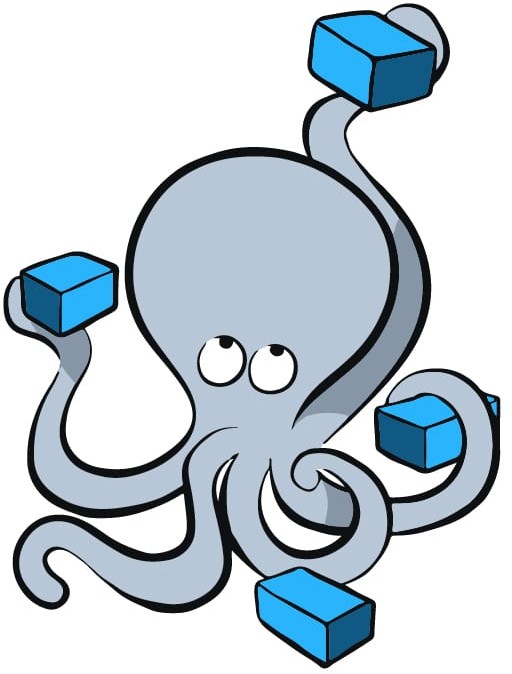
### Frontend Dockerfile

**Backend Dockerfile**

Running *docker build* using this Dockerfile as the source creates the required Docker image that is ready to run our application.

We need to build the Docker image and push it to Docker Hub which requires logging in to the DockerHub account as well.

This will be covered in the Continuous Integration section.



## Docker Compose

Docker Compose is a tool that was developed to help define and share multi-container applications. With Compose, we can create a YAML file to define the services and with a single command, can spin everything up or tear it all down.

The big advantage of using Compose is you can define your application stack in a file, keep it at the root of your project repo (it’s now version controlled), and easily enable someone else to contribute to your project. Someone would only need to clone your repo and start the Compose app.

On the host machine, follow the instruction of this link <https://docs.docker.com/compose/install/>

After installation, you should be able to run the following and see version information.

docker-compose version

### Compose File

At the root of the app project, create a file named *docker-compose.yml*

You can look at the Compose file reference for Compose file syntax and features. <https://docs.docker.com/compose/compose-file/>

Note: *“version”* parameter is not required in Docker Compose YAML files since version 1.27.

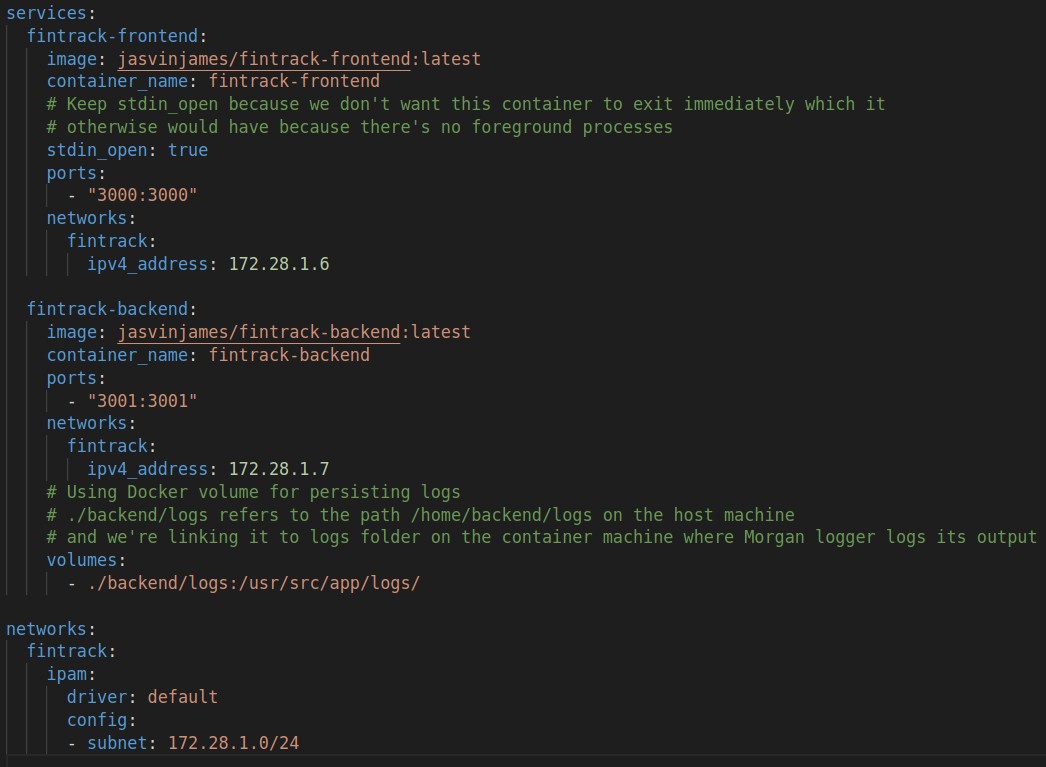
Next, we’ll define the list of services (or containers) we want to run as part of our application.

Create two containers for running the app, one for the frontend and one for the backend.

Map port 3000 on the host machine to port 3000 on the frontend container because that’s where React runs and do the same with the backend but with port 3001 because we’ve configured the Express app to listen on port 3001.

Attach both containers to the same network which is named *fintrack*, this enables the frontend container to communicate with the backend container and vice-versa. We fix

the subnet of the *fintrack* network and then apply static IP addresses to both the frontend and backend containers. Because of the static IP addresses, the frontend can fire queries to the backend using a fixed backend URL.

The reason for using a Docker volume is already specified in the code itself.



## Ansible

Ansible is an open-source automation tool, or platform, used for IT tasks such as configuration management, application deployment, intraservice orchestration and provisioning.

Ansible is mainly used to perform a lot of tasks that otherwise are time-consuming, complex, repetitive, and can make a lot of errors or issues.

**Note:** We are going to be pulling the Docker Hub image to the host system for Ansible deployment.

### Creating Inventory file

The inventory file is used to specify the list of managed hosts/server machines. The inventory file looks as,

<host-machine-IP-address> ansible\_user=jasvin

*jasvin* is the host machine’s user on which Ansible shall execute the specified commands.

Create this file in the root directory of your project repository with the name *inventory*.

### Configuring OpenSSH Server

Install openssh-server on the host machine and because it is the Jenkins user that is going to be doing the pulling of Docker image, we need to SSH from the Jenkins user to the user that is specified in the inventory file, i.e. *jasvin*.

apt-get install openssh-server service ssh restart

su jenkins

ssh-keygen -t rsa

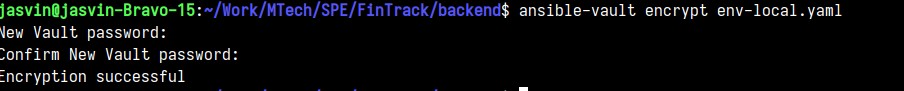
ssh-copy-id <host-username>@<host-ip-address> sudo chmod 666 /var/run/docker.sock

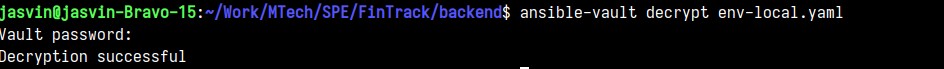
The *chmod* command is required so that the Jenkins user has access to the Docker socket for performing the docker build and push operations.

Because we use MongoDB Atlas for storing our database, we require the MongoDB URI in order to access the database and this URI should be kept secret. However we have to send this URI in a *.env* type of file to the backend container, how to do this and not leak the URI in any way? This is where Ansible Vault comes in the picture.

### Ansible Vault

Ansible Vault is a feature of ansible that allows you to keep sensitive data such as passwords or keys in encrypted files, rather than as plaintext in playbooks or roles. These vault files can then be distributed or placed in source control.

First we have to encrypt the environment file using *ansible-vault*

Command for decrypting the environment file using *ansible-vault*

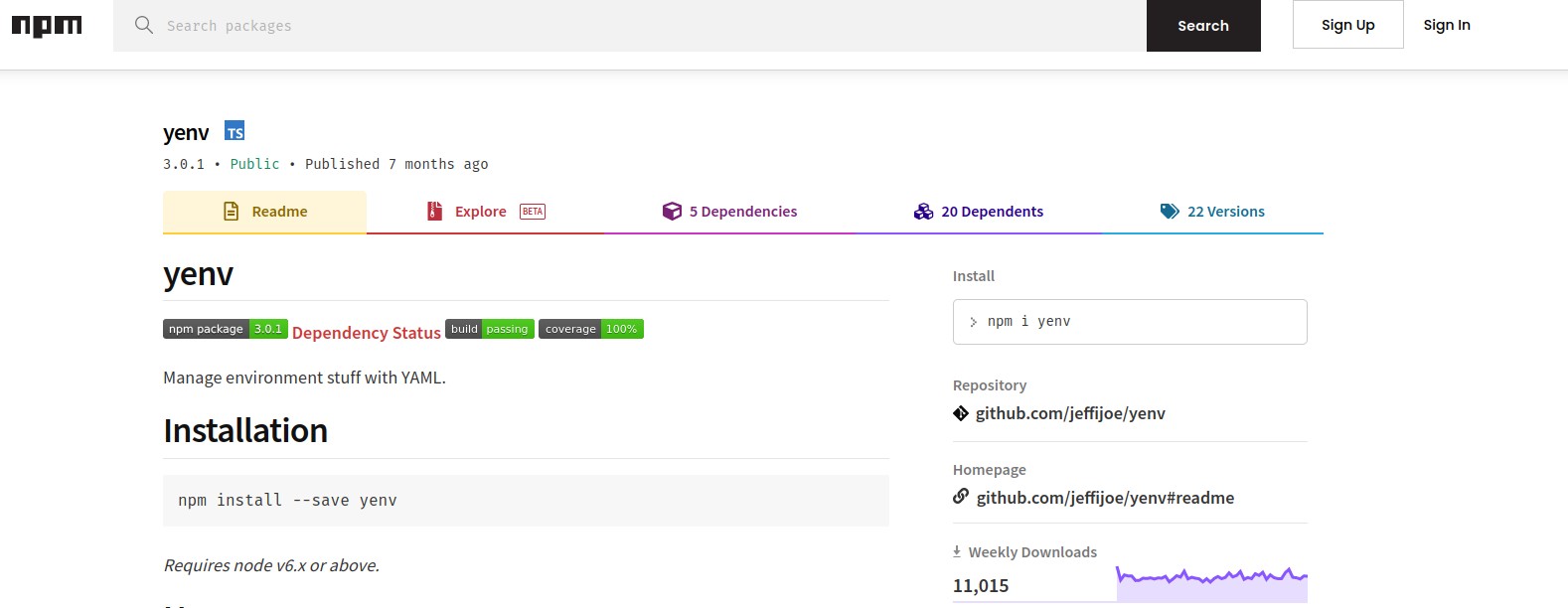
### Encrypted YAML file

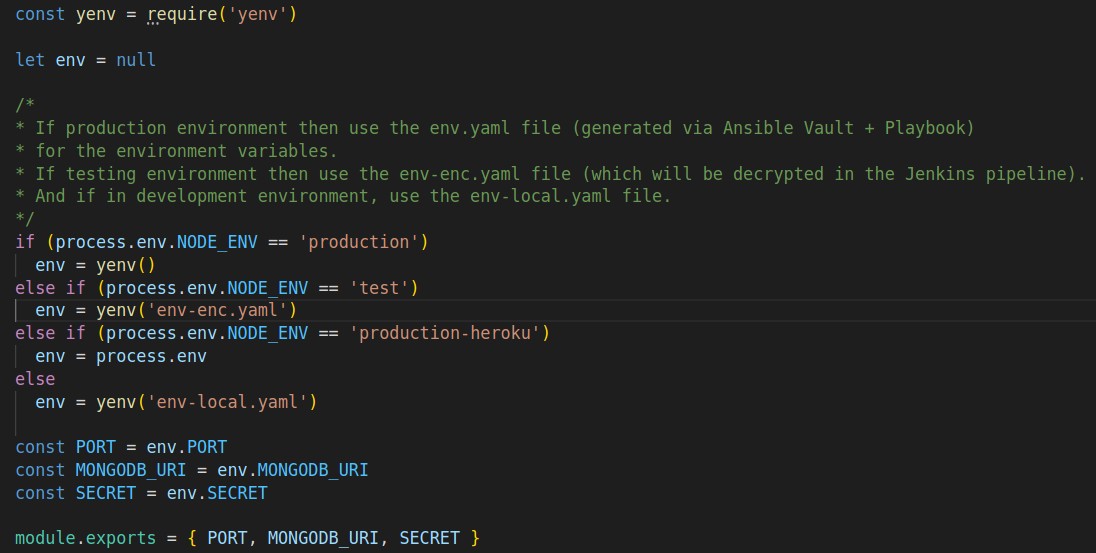
We are using yaml files for storing environment configuration instead of a *.env* file because Ansible Vault requires a yaml or JSON type file for encrypting and decrypting. This will become more clear when the playbook is explained.

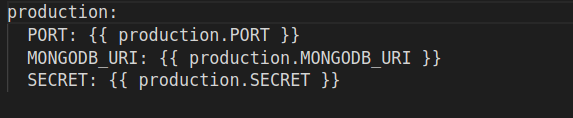
### Environment YAML decrypted

The environment tags like production, local-test and development are what will be used to separate the environment variables from each other, because some values like URI will be different for production and testing environments and so on.

*yenv* library for reading environment variables,



Using the *yenv* library in the *config.js* file for setting up environment variables

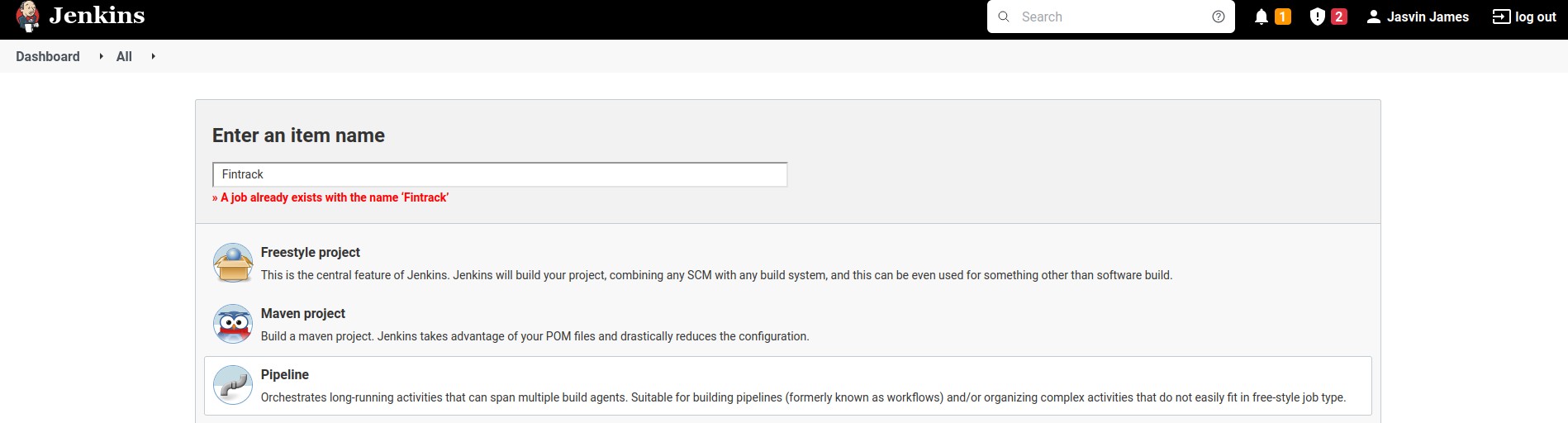
Jinja2 Template that will be required for decrypting environment variables via Ansible Vault,

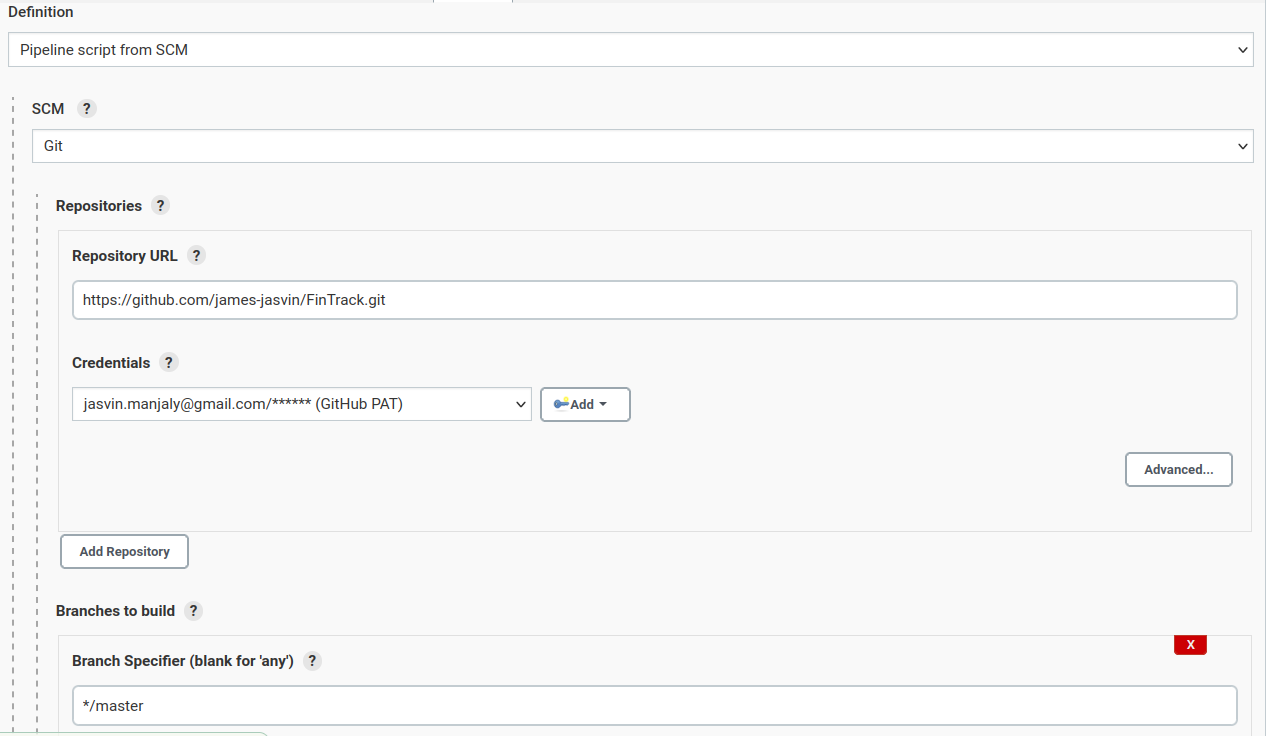
The Ansible playbook will be explained after the next section.



## Continuous Integration: Jenkins

Jenkins is an open source automation server. It helps automate the parts of software development related to building, testing, and deploying, facilitating continuous integration and continuous delivery. It is a server-based system that runs in servlet containers such as Apache Tomcat.

Create a Jenkins Pipeline project,

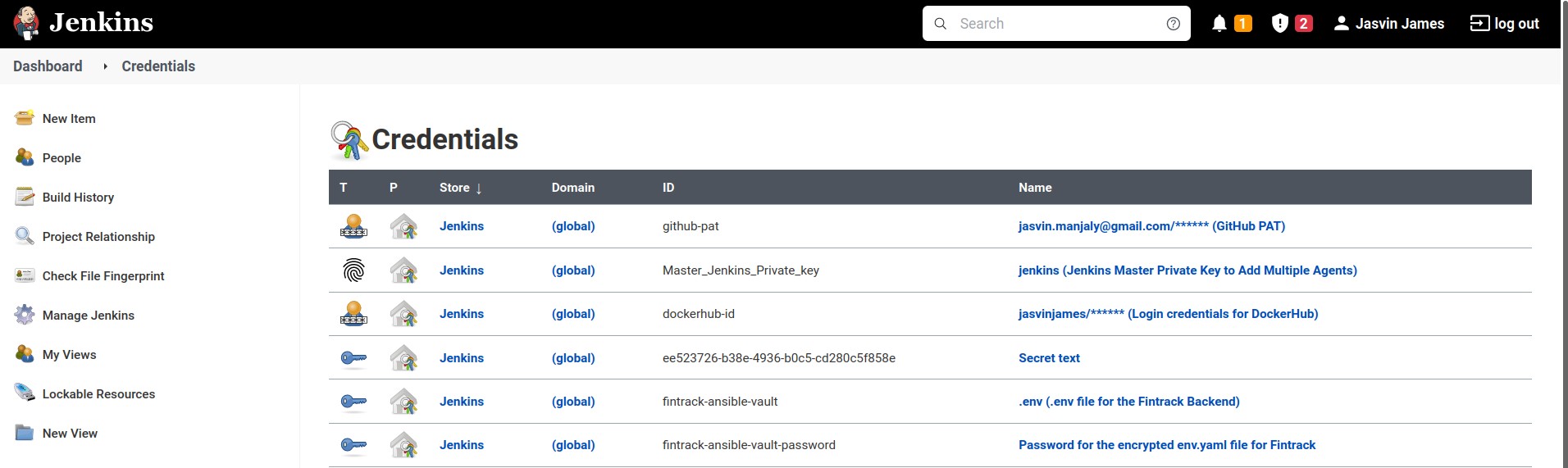
Reading Jenkins pipeline script, *Jenkinsfile* from the SCM repository

### Credentials & Jenkins

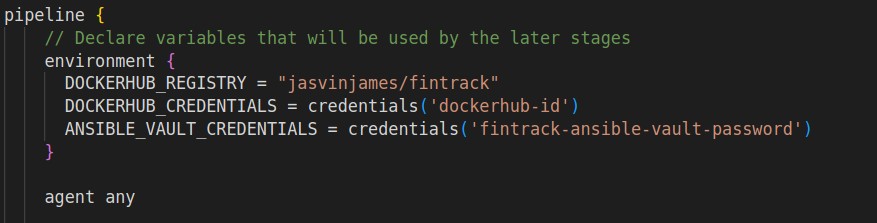
**Github PAT:** For accessing private Github repositories.

**Docker Hub Credentials:** For logging into Docker Hub account and pushing Docker images (it is preferred to use Access Token instead of your actual Docker Hub password).

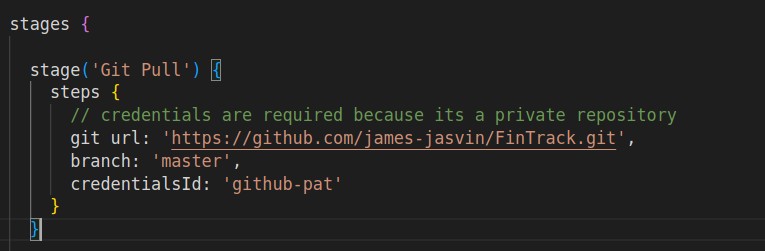
**Ansible Vault Password:** Save the password that was used to encrypt the

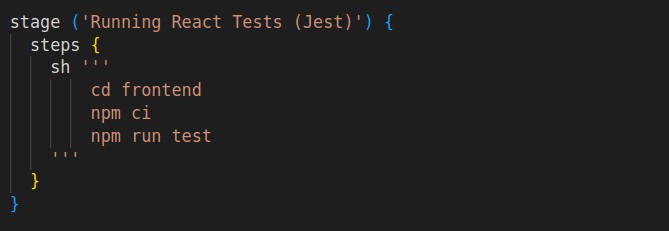
*env-enc.yaml* file as a *Secret Text* credential on Jenkins.

### Pipeline Script

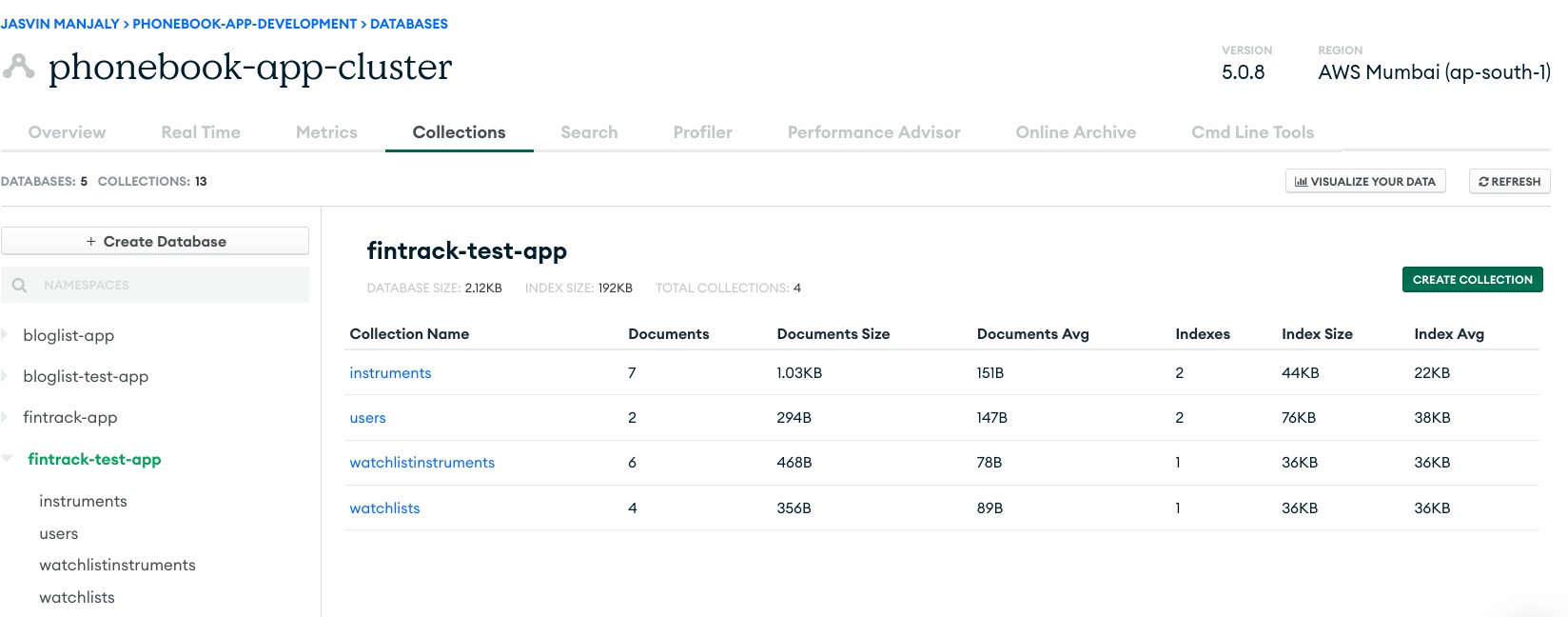
First we set up the environment variables which includes importing the Docker Hub and Ansible Vault passwords as credentials.

Then we perform a Git pull on the repository’s master branch with the Github PAT as the

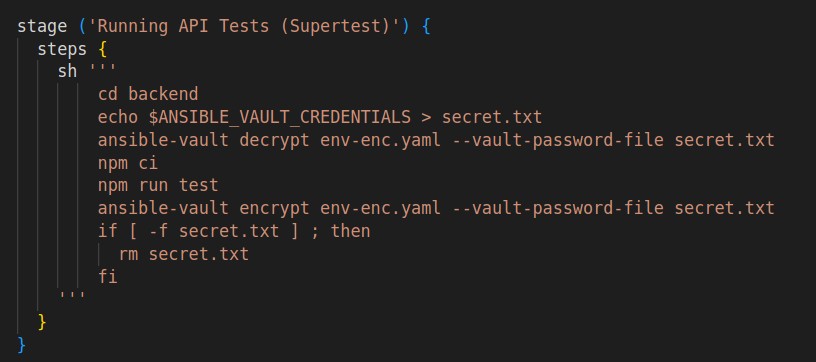
*credentialsId*.

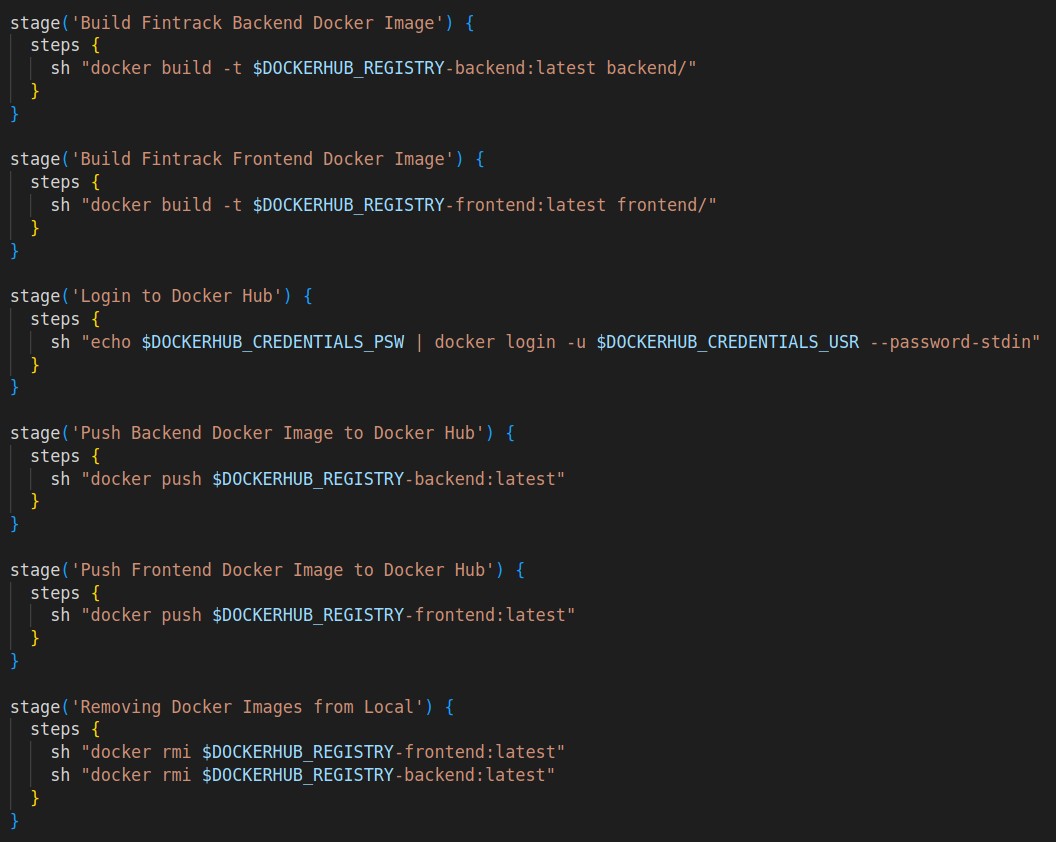
After this we run the frontend testing which is done via the React Testing Library and for this we use a multi-line shell script (triple quoted string) in which we *cd* to the *frontend* directory, install all the dependencies and run the tests.

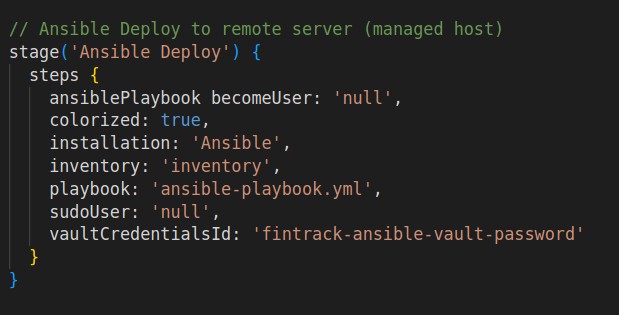
Now we run backend testing via the Supertest library. An important point to note here is that we need to connect to the MongoDB Atlas cluster in order to access the database while testing. So we have set up a separate database in the same collection for testing.



But a big problem now is that we have encrypted the yaml file which contains this testing database’s URI, so what to do?

We decrypt the *env-enc.yaml* file using the Ansible Vault password which we have saved as a credential in Jenkins, store it in a file called *secret.txt*, run the backend tests, encrypt the file once again using the *secret.txt* file and now delete the *secret.txt* file if it exists.

After this we build the Docker images, push them to Docker Hub and remove the local Docker images.

Now we finally call the Ansible playbook, note that we also supply the Ansible Vault password to it.

## Ansible Playbook

The main purpose of creating playbooks is that it encapsulates all the tasks under one playbook file.

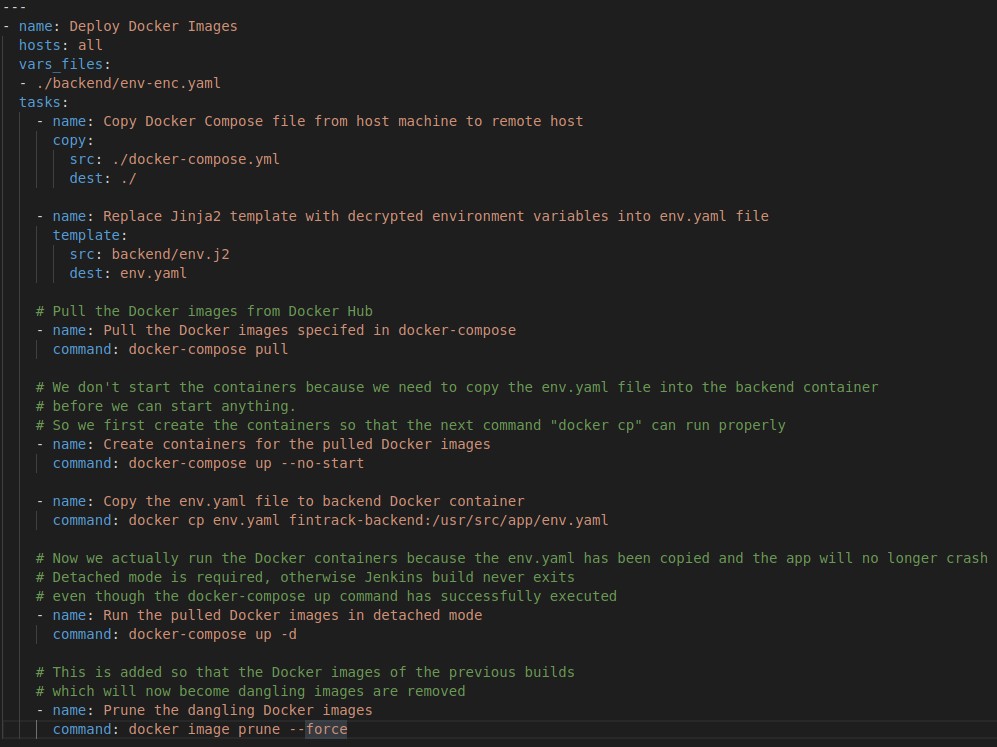
In this playbook, the toughest task is to use the encrypted *env-enc.yaml* file and decrypt it somehow to send to the managed nodes and somehow place it inside the running container.

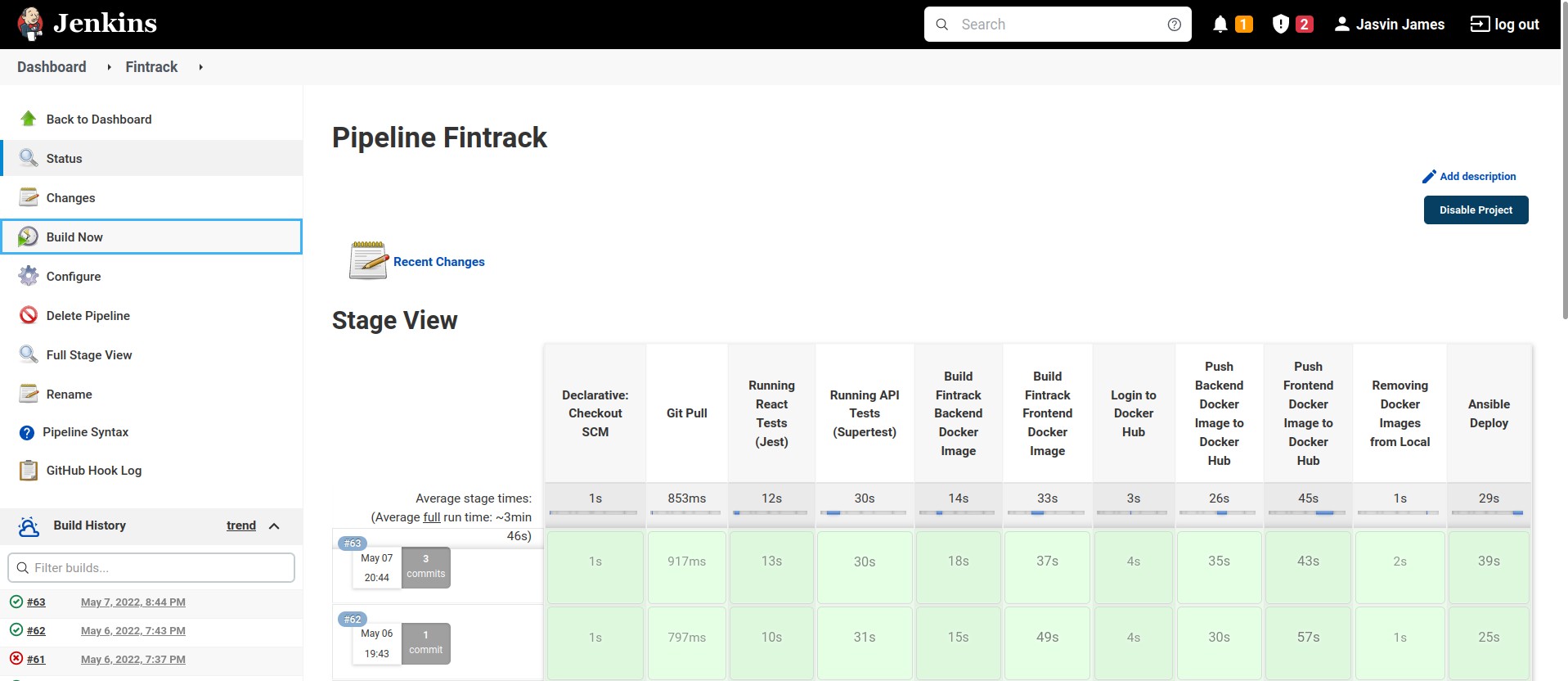
That is why we use the *vars\_files* module with the Ansible playbook. We specify the *env-enc.yaml* as the file to look for and because Jenkins invoked the Ansible playbook with the Vault credentials, the file will now be decrypted for direct use but we cannot copy this file into the Docker container, which is why we use templating and this is where that *env.j2* template comes into the picture.

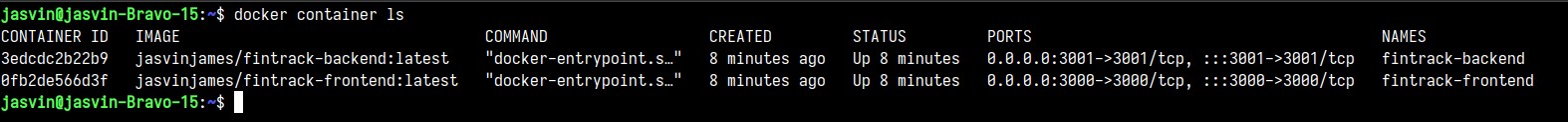
The *backend/env.j2* file which contains Jinja2 template variables will now be replaced by the decrypted environment variables in *env-enc.yaml* and we use the template module to store the created file under the name, *env.yaml* under the root directory of the managed nodes.

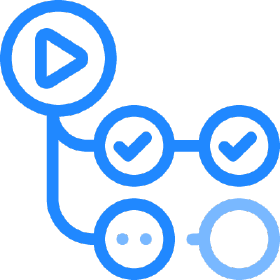
Now all we have to do is copy this *env.yaml* file into the backend container.

Everything else is explained in the comments of the playbook whose image is attached below.



Running Jenkins build,

Running Docker containers after build,

## PaaS Deployment with Netlify & Heroku & Github Actions

We can deploy a MERN stack based application directly to Heroku and serve the React frontend as a static production build, however this is only applicable when the React app is a SPA (Single Page Application), whereas we have used React Router in order to create a Multi Page Application. So that is where Netlify helps with its ease of use for deployment as well as handling React Router based applications.

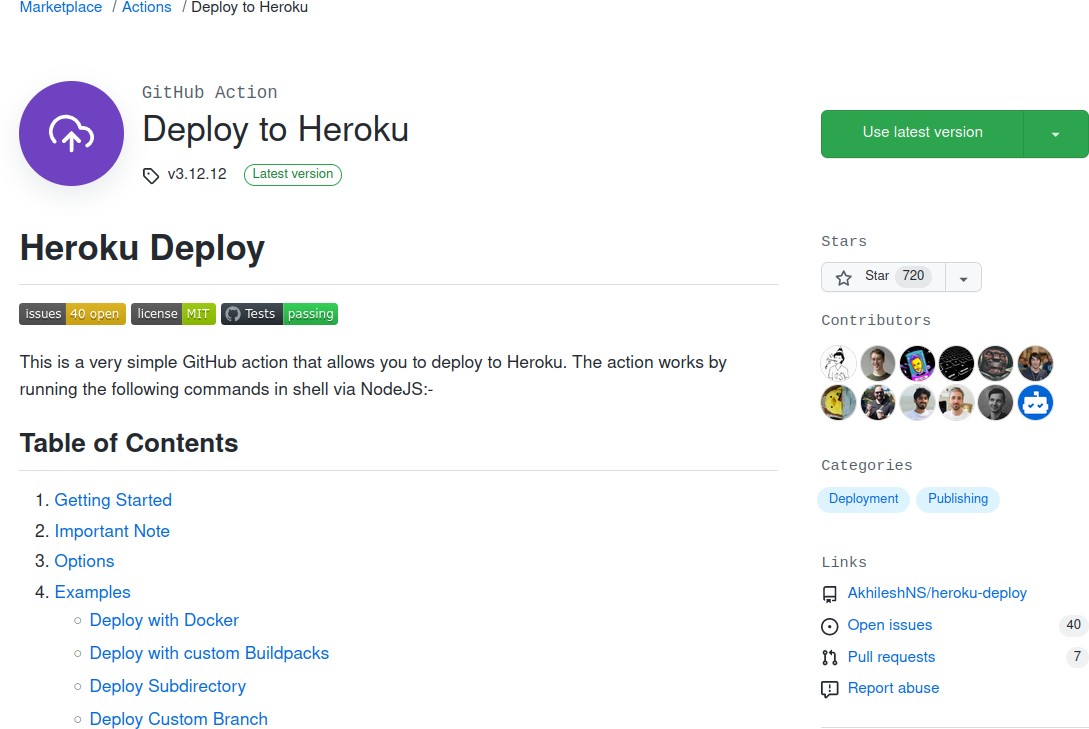
Following describes the steps that have to be followed for deploying the Express backend to Heroku and the React frontend to Netlify.

### Heroku Configuration

Create a Heroku account and create a new app.

We can deploy to Heroku using several methods but we have used the Github Actions route. Before we could directly connect a Github repository to a Heroku app but a recent vulnerability with Heroku has forced them to disable this option.

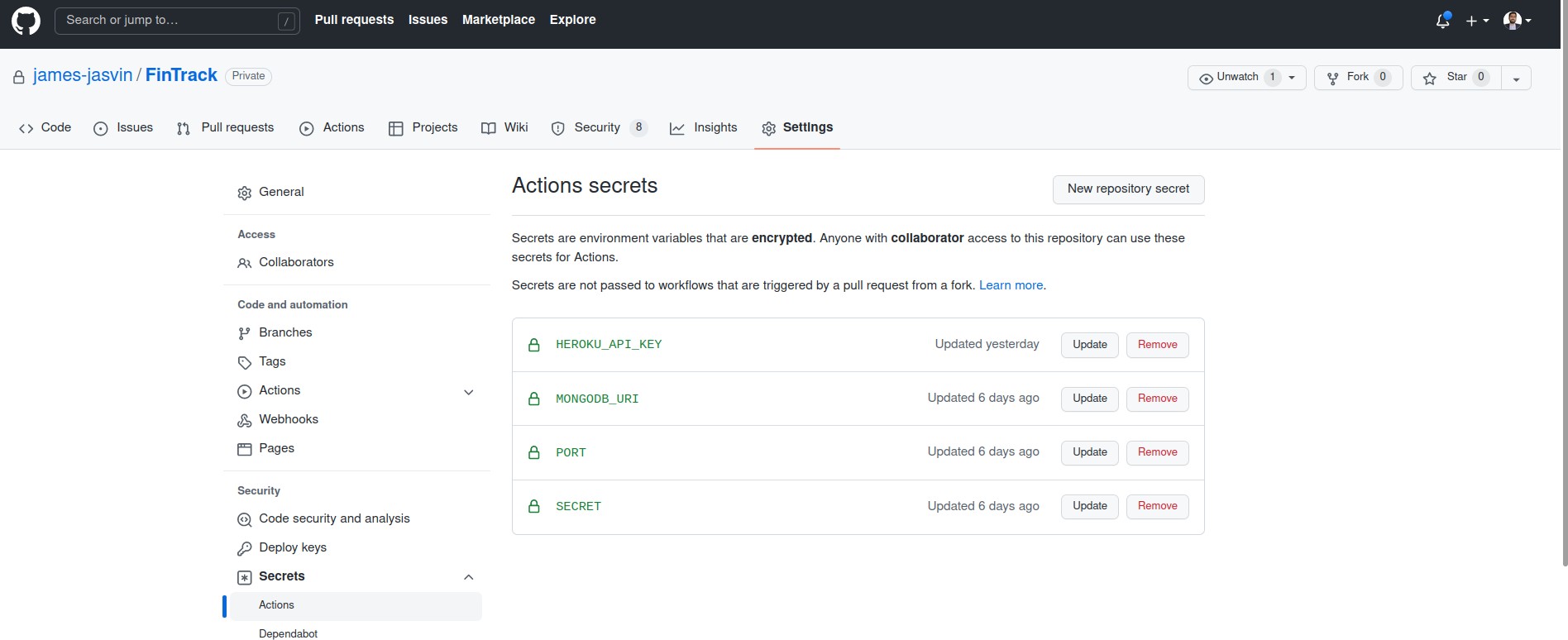
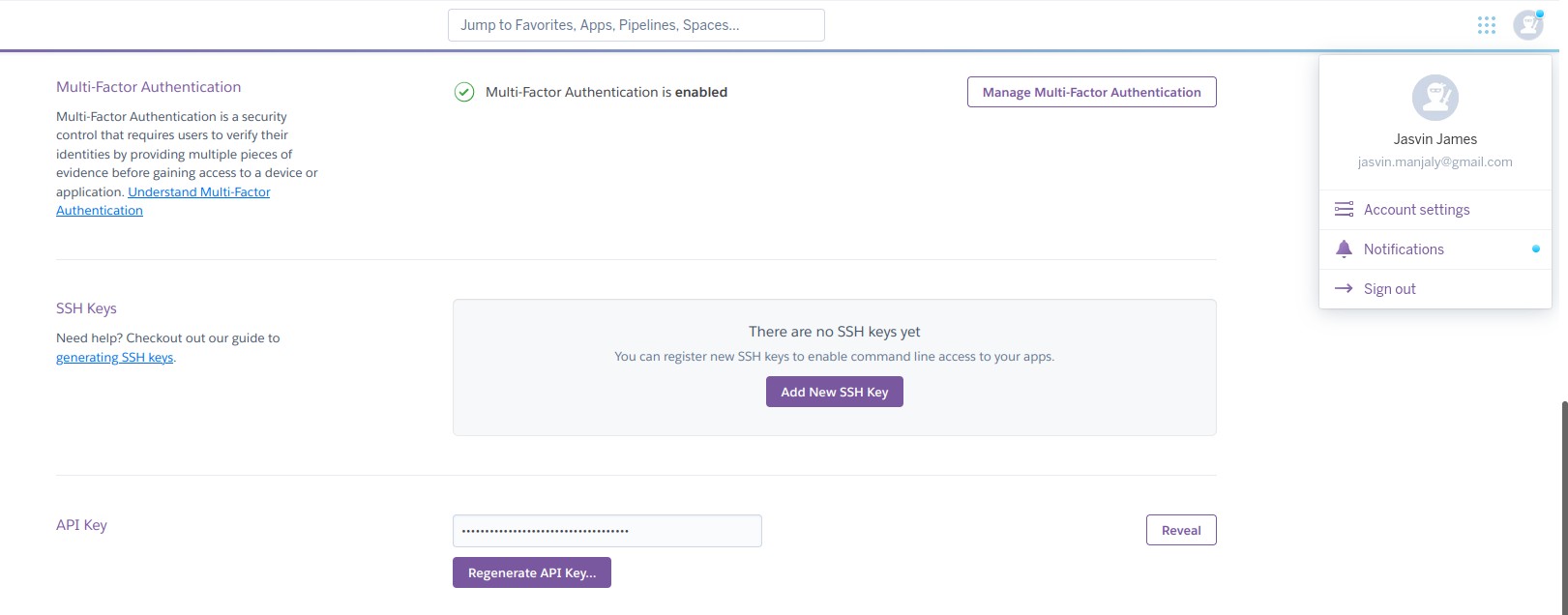
So we use a third-party Github Action for deploying to Heroku

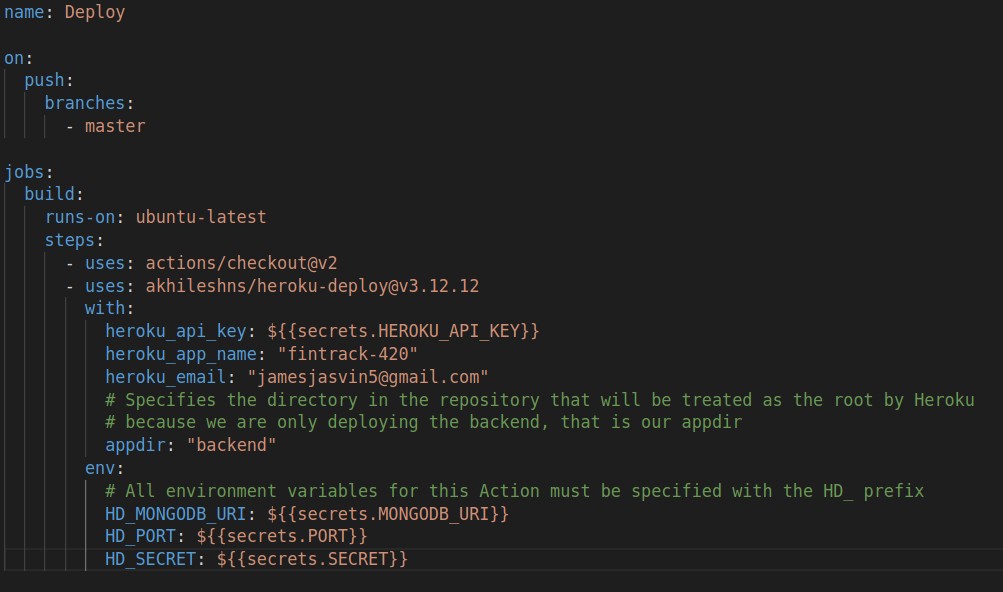


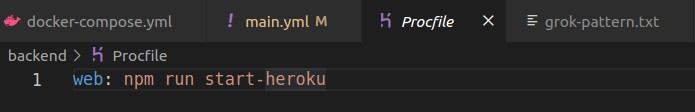
For the Action to work, it should be able to login to your Heroku account and we also need a new way to manage secret environment variables because Ansible will not be

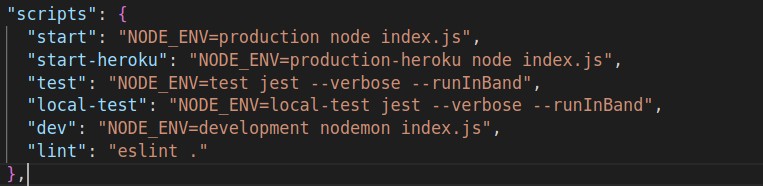
involved with this deployment. We configure these secret environment variables in the repository’s secrets which can be set by going to the *Settings* menu of your repository.

Also to login to Heroku we save the Heroku API key (available in the *Account Settings*

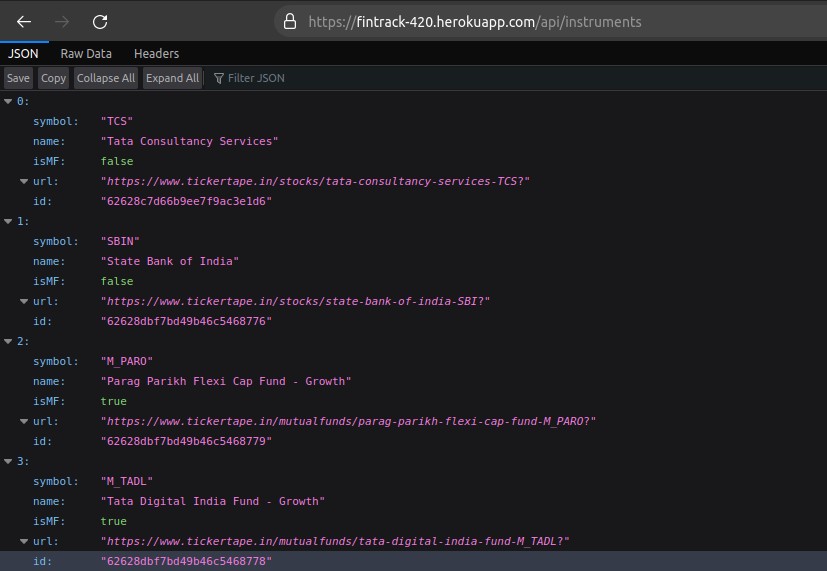
section of your Heroku account) in the Github secrets as well.

And this is what the yml file looks like for the Workflow,

Finally in order to get a Heroku app running we need to add a *Procfile* which in this case will be in the *backend* directory.

*start-heroku* is a *NODE\_ENV* value that is set in *package.json* and it was also used in the *config.js* file to specify the fact that we’ll be reading the environment variables from the *process.env* JSON.

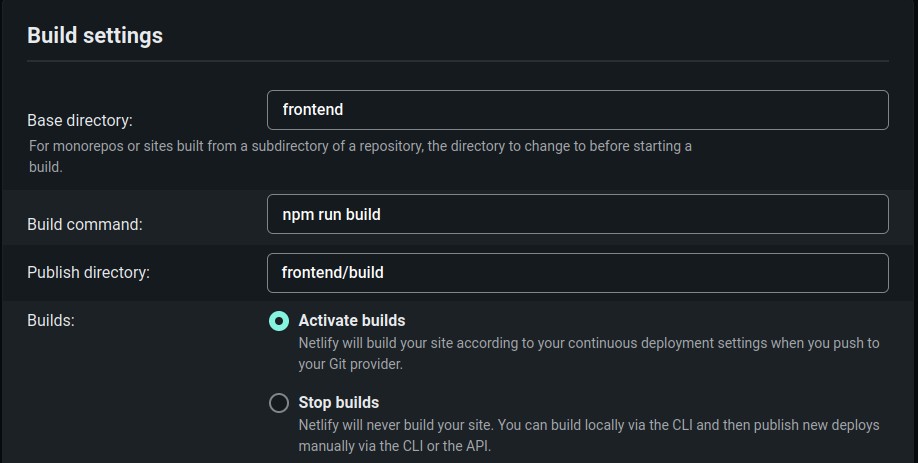
Now pushing these changes to your Github repo will automatically trigger the Action and build the Heroku app.

Verifying that the Heroku app is indeed serving the backend.

### Netlify Configuration

Create a Netlify account and link your Github repository in order to deploy the frontend app.

Update the Deploy build settings as follows,



The *Publish directory* is *frontend/build* otherwise Netlify will try to read the *build* directory in the root directory itself which is not where *npm run build* will save the *build* directory and crash the app.

Now the frontend will also automatically deploy but there will be a problem with React Router. You cannot directly access any routed paths such as

<https://fintrack-420.netlify.app/search> without going through the home page first.

For this you have to add a file called *\_redirects* under the *public* folder of your React app which also contains the *index.html* file.

This basically tells Netlify that if the URL to serve has anything after the ‘/’ then redirect to rendering *index.html* with a 200 status code.

And now React Router will be able to pick up the specified URL and appropriately route the user to the correct page without having to go through the home page, nor result in an error.

We will also have to add the same environment variables that we have used in the

*.env.development* file in the frontend in order for this deployment to work.

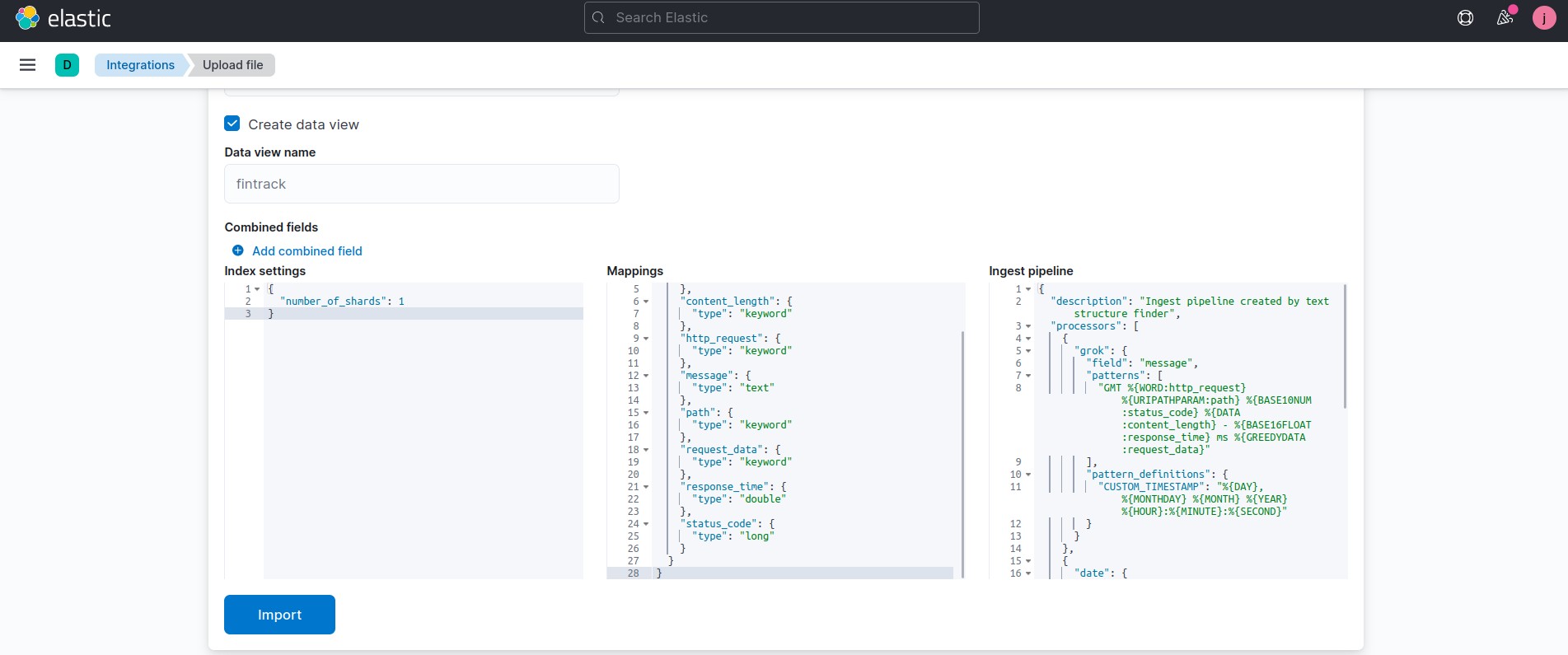
And that does it for the entire Heroku + Netlify configuration!



## Monitoring - ELK Stack

"ELK" is the acronym for three open source projects: Elasticsearch, Logstash, and Kibana. ELK stack gives us the ability to aggregate logs from all the systems and applications, analyze these logs, and create visualizations for application and infrastructure monitoring, faster troubleshooting, security analytics, and more.

To use the ELK stack we first require the logs that are generated from the application. Because we have used Docker volumes for persisting the logs as specified in the Docker Compose file, we can get the logs at the path */home/backend/logs/access.log.*

Now we can upload this log file into our Elastic Cloud cluster.

To infer the proper fields from the log file we have to use the following Grok pattern

"CUSTOM\_TIMESTAMP": "%{DAY}, %{MONTHDAY} %{MONTH} %{YEAR} %{TIME}"

"%{CUSTOM\_TIMESTAMP:timestamp} GMT %{WORD:http\_request}

%{URIPATHPARAM:path} %{BASE10NUM:status\_code} %{DATA:content\_length}

- %{BASE16FLOAT:response\_time} ms %{GREEDYDATA:request\_data}"

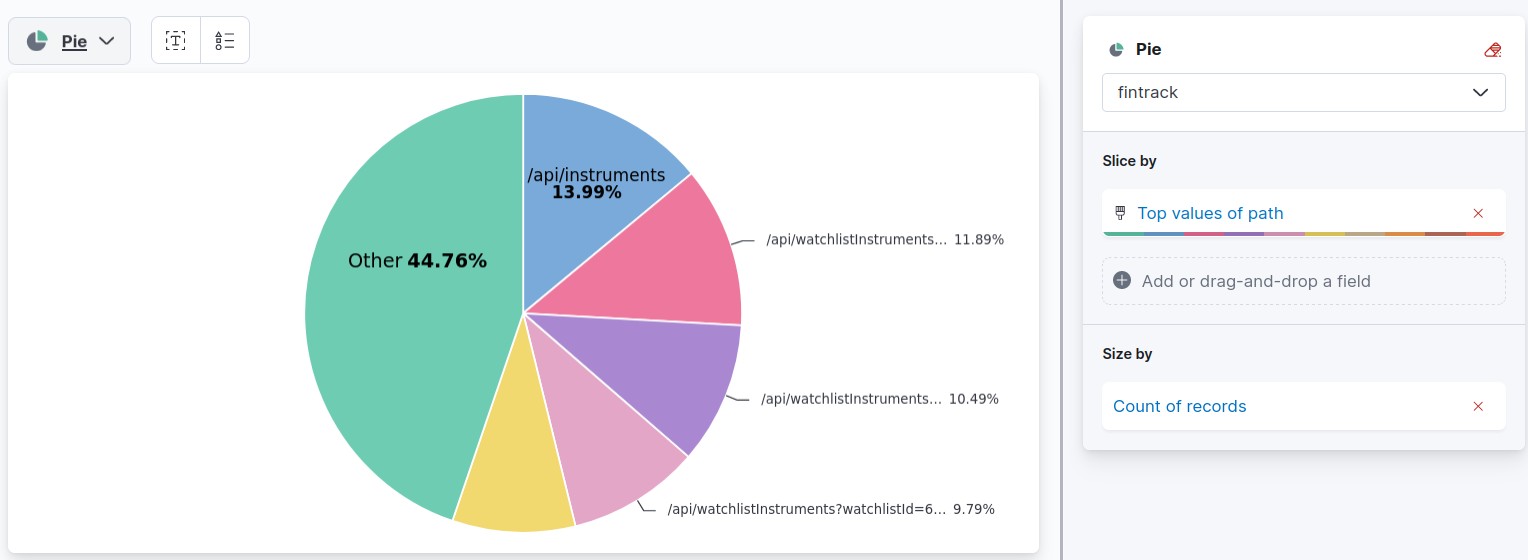
## Visualizations

### Time Series View

1. **Visualizing http\_request field**

### Visualizing status\_code field

1. **Visualizing path field**



We also created a Kibana Dashboard containing these three visualizations.

# Experimental Setup

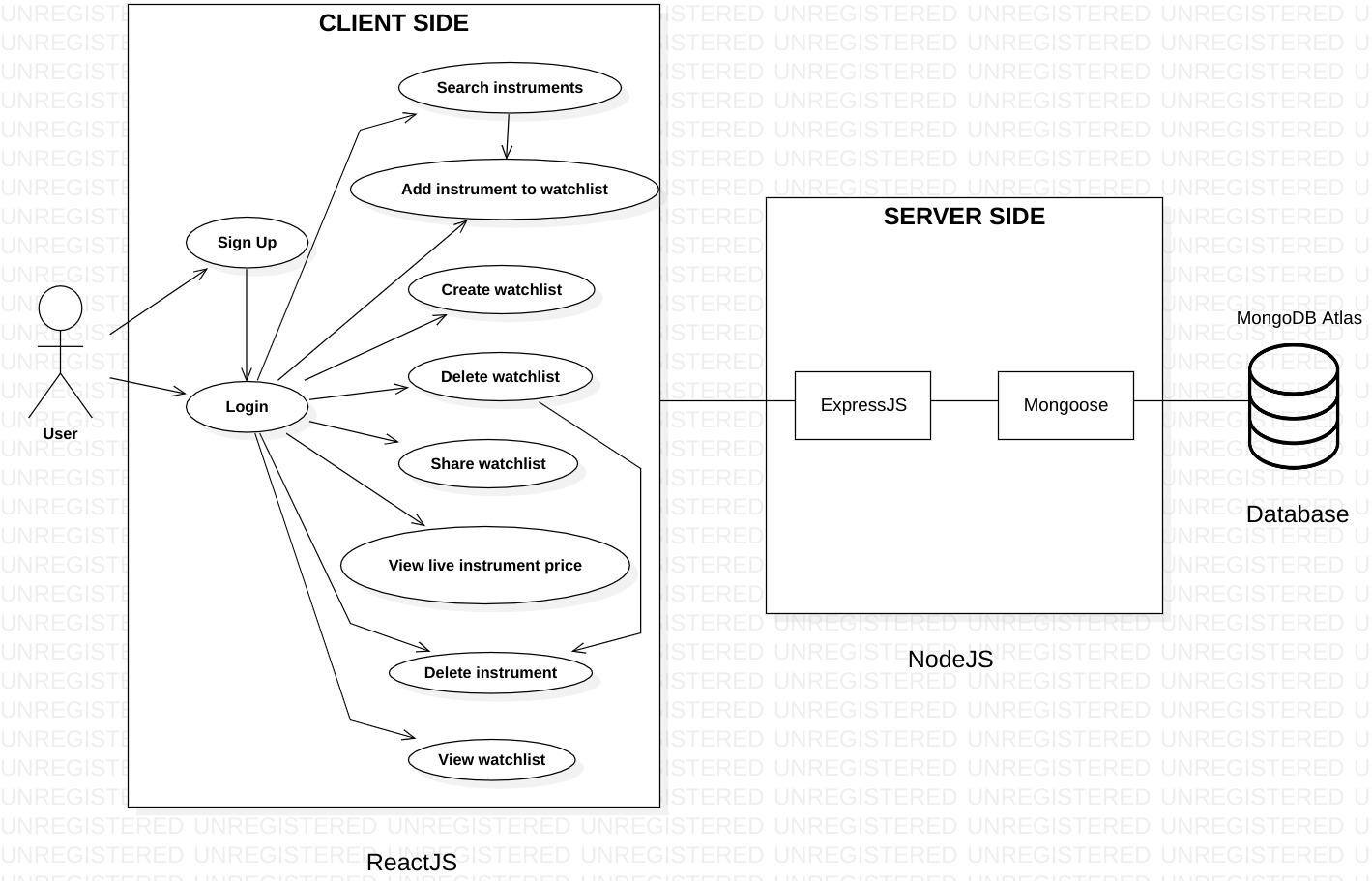
## Functional Requirements:

* + 1. Users can register and create their account.
    2. Users can create watchlists without creating duplicate entries.
    3. Users can view their watchlists and its contained instruments.
    4. Users can edit the desired watchlist (delete the instruments or add the instruments) as per their wish.
    5. Users can search for the instruments for adding them to their watchlist.
    6. Each watchlist will have a share button that will send a publicly shareable link to anyone that visits that URL.
    7. To view a shared watchlist, a user must be logged in to a Fintrack account, which can be any account but a login is required.
    8. Watchlists cannot be amalgamated. ‘Mutual Funds’ watchlists must only contain Mutual Funds, whereas ‘Stocks’ watchlists must contain only Stocks.

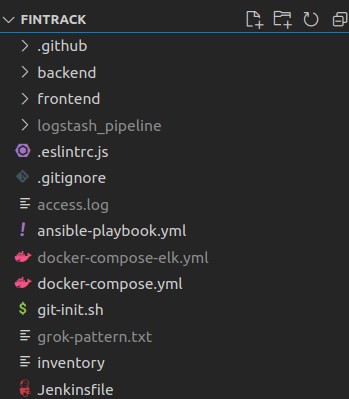
## Non-Functional Requirements:

* + 1. **Portability**: To ensure portability, Docker images have been built for the frontend and backend.
    2. **Scalability**: The database is created on MongoDB Atlas and in case of high traffic, Atlas will automatically handle this via scaling and thus scalability is achieved.
    3. **Security**: To improve security, JWT tokens are being used for setting up a session and checking the authorization.
    4. **User Friendly**: The website has to be user friendly and error messages should pop up when relevant.
    5. **Performance**: The performance of the website should not degrade in case of multiple users.

## Architectural Diagram:



* 1. **Code Walkthrough**

The folder structure that we have followed includes the client-side *frontend/* and the server-side *backend/* with subfolders within them to ensure proper flow of data in the website.

**Directory structure for the project**

* + 1. **Backend**

### Directory structure for the Backend

We have incorporated MVC architecture (Model View Controller) in the project.

### Model:

* + - * It is known as the lowest level which means it is responsible for maintaining data.
      * The Model is actually connected to the database so anything you do with data - adding or retrieving data is done in the Model component.
      * It responds to the controller requests because the controller never talks to the database by itself. The Model talks to the database back and forth and then it gives the needed data to the controller.

Here are the Models that we have used in our project.

### User Model

const mongoose = require('mongoose')

const uniqueValidator = require('mongoose-unique-validator')

const userSchema = new mongoose.Schema({

username: { type: String, required: true, minLength: 3, unique: true }, name: { type: String, required: true },

passwordHash: { type: String, required: true },

})

1. **Instrument Model**

const instrumentSchema = new mongoose.Schema({

symbol: { type: String, required: true, unique: true }, name: String,

isMF: { type: Boolean, required: true }, url: { type: String, required: true }

})

### Watchlist Model

const watchlistSchema = new mongoose.Schema({ name: { type: String, required: true },

user: {

type: mongoose.Schema.Types.ObjectId, ref: 'User'

},

isMF: { type: Boolean, required: true }

})

1. **WatchlistInstrument Model**

const mongoose = require('mongoose')

const watchlistInstrumentSchema = new mongoose.Schema({ watchlist: {

type: mongoose.Schema.Types.ObjectId, ref: 'Watchlist',

required: true

},

instrument: {

type: mongoose.Schema.Types.ObjectId, ref: 'Instrument',

required: true

}

})

**View:**

* + Data representation is done by the View component.
  + It actually generates UI or user interface for the user.
  + So in web applications when you think of the View component just think of it as the HTML/CSS part.
  + Views are created by the data which is collected by the model component but this data isn’t taken directly from the Model and instead obtained through the Controller.
  + View only speaks to the Controller.

**Controller:**

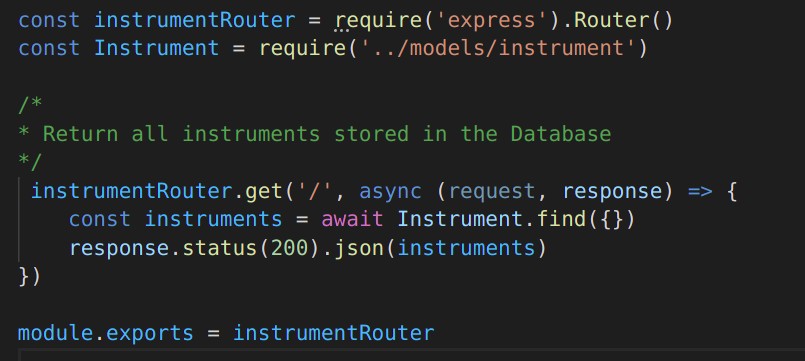
* + It’s known as the main man because the controller is the component that enables the interconnection between the View and the Model, so it acts as an intermediary.
  + The Controller doesn’t have to worry about handling data logic, it just tells the Model what to do.
  + After receiving data from the Model, the Controller processes it and sends the information to the View.
  + **Note**: Views and Models cannot talk directly.

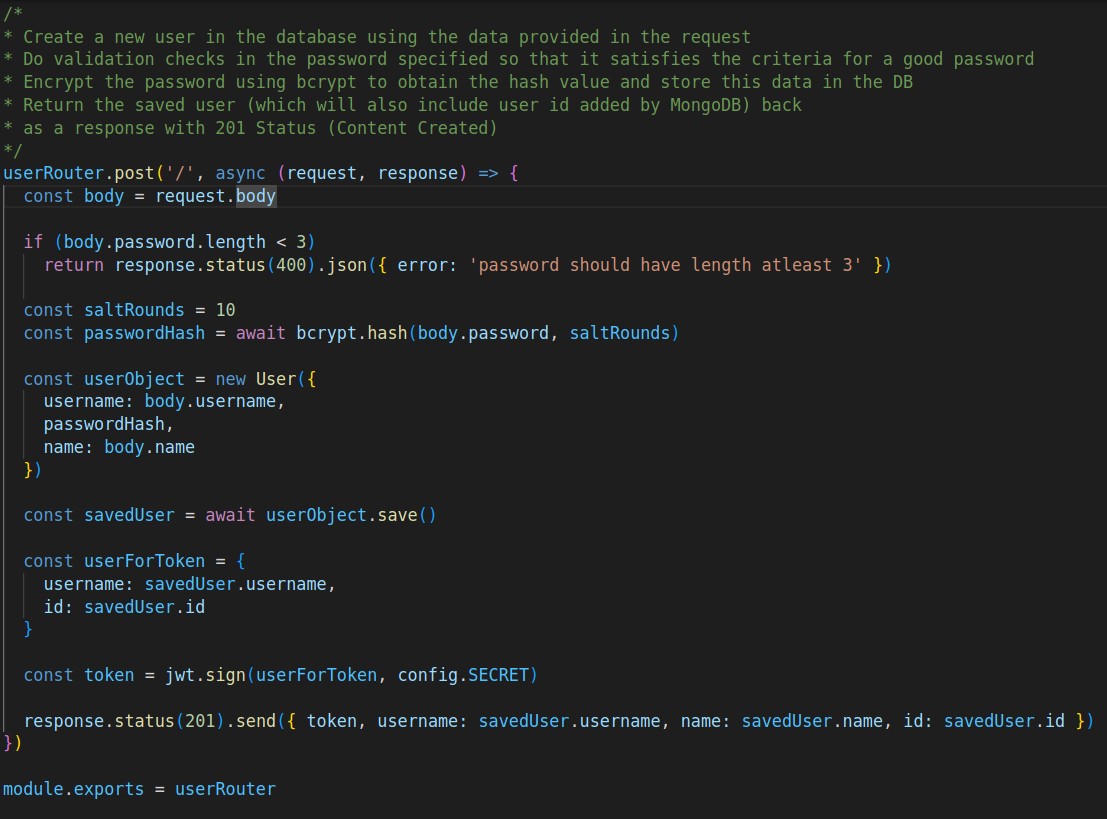
**Express & NodeJS** do all the functional programming and will be used to write the Business Tier.

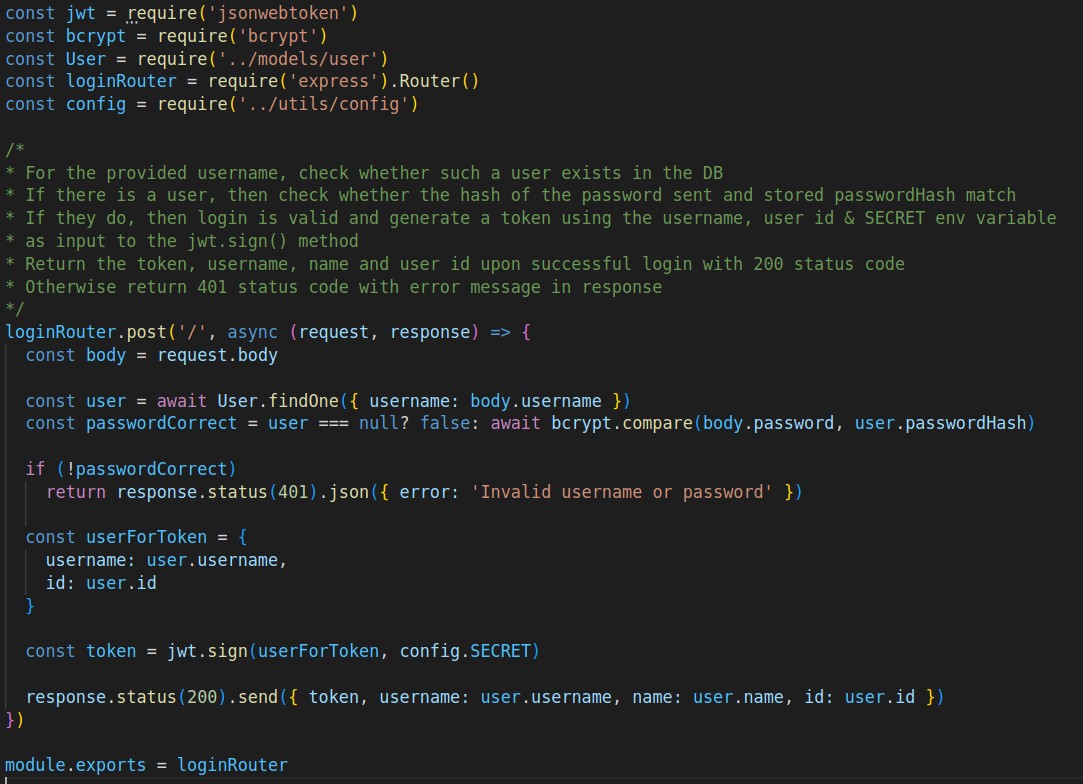
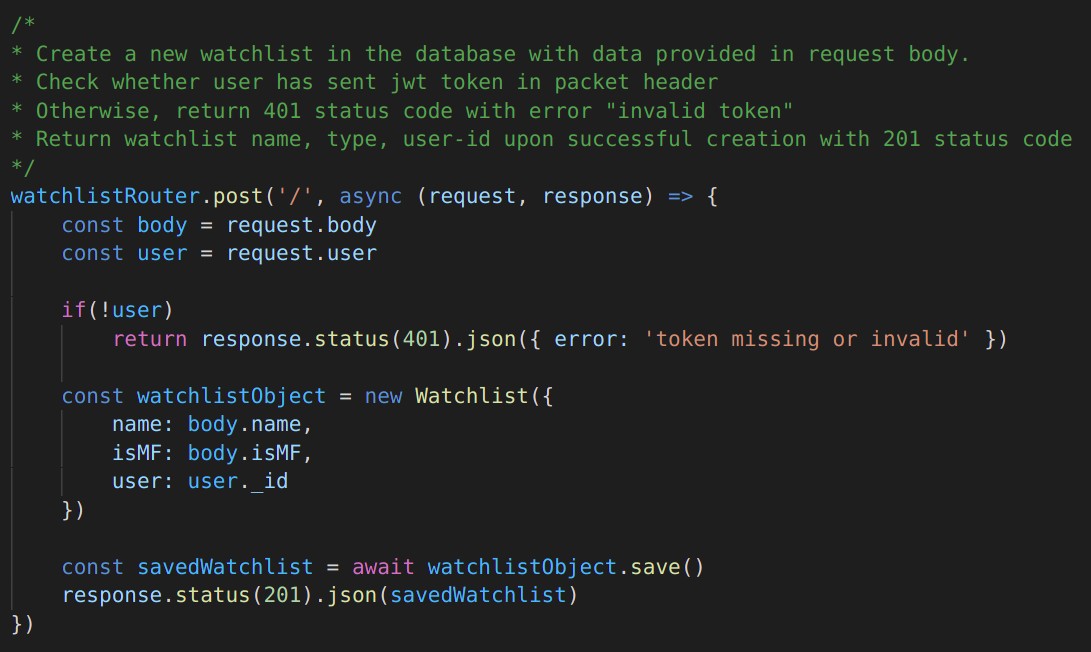
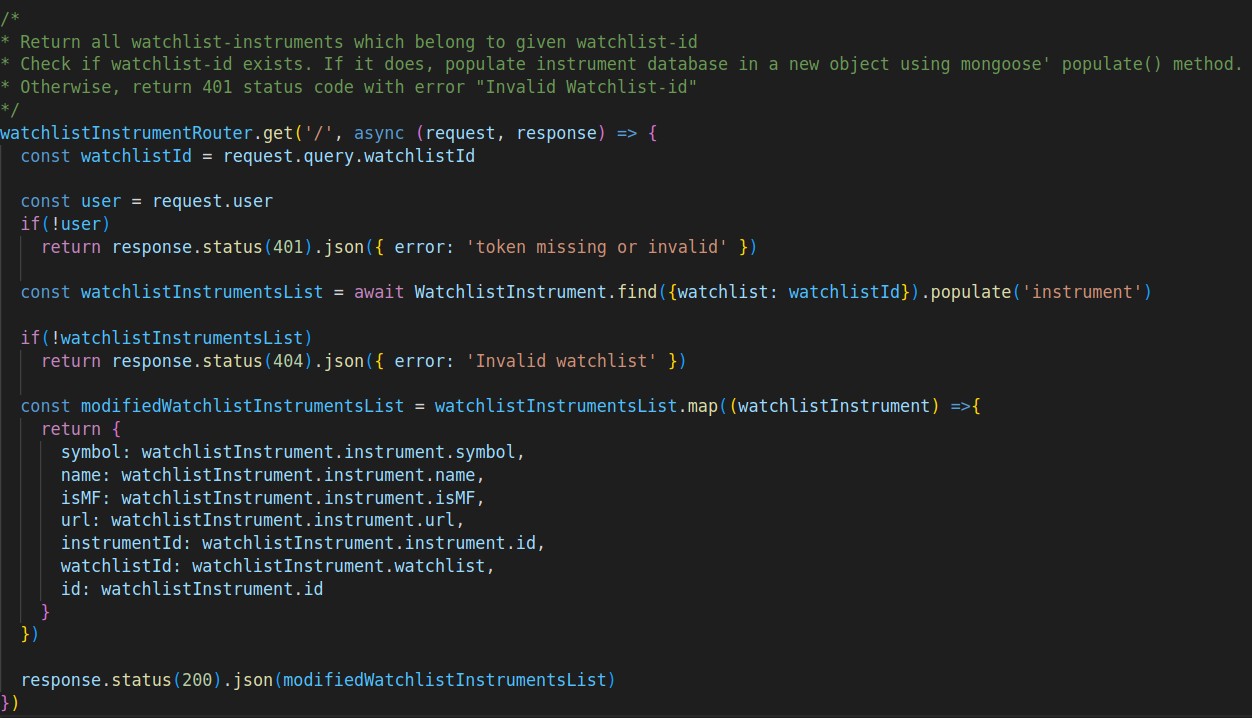
This tier represents the Application Server that acts as the bridge of communication between the Client and Database. This tier will serve the React components to the

user’s device and accept HTTP requests from the user and follow with the appropriate response.

Here is an overview of the Controllers in our project.

1. **Instruments Controller**
2. **Users Controller**

Handles the signup of users into our web-app.

1. **Login Controller**
2. **Watchlists Controller**
3. **WatchlistInstruments Controller**

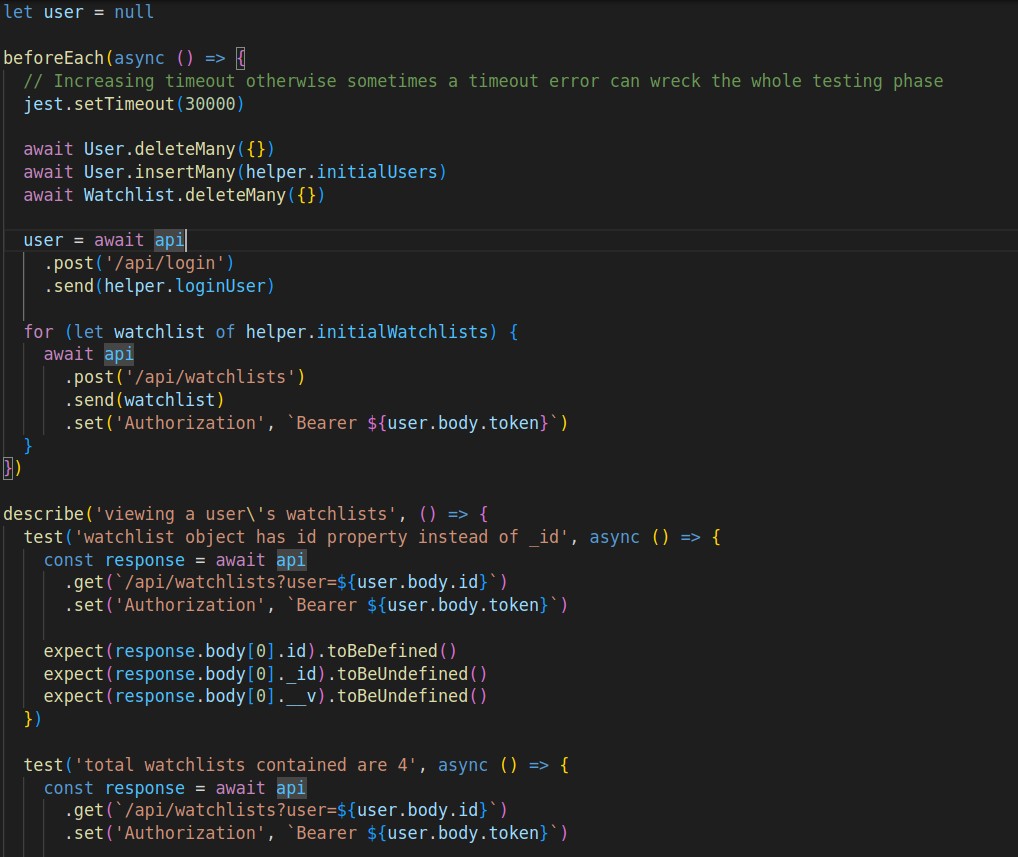
***utils* directory**

Contains middleware, config and logger files to support controller’s functionalities.

***tests* directory**

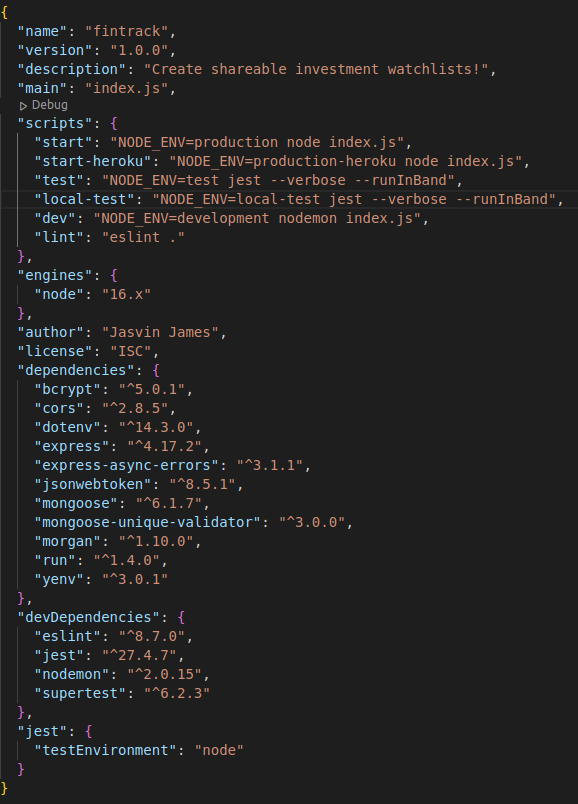
Contains the supertest test files as described in the SDLC section.

Here is a snippet of the *watchlists\_api.test.js* file which shows how we use the beforeEach() method to login as an existing user and perform authorized operations in the tests with the JWT token.



#### package.json

This file is the heart of our Node project. It records important metadata about our project which is required before publishing to npm and also defines functional attributes of our project that npm uses to install dependencies, run scripts, and identify the entry point to our package.



## API Documentation

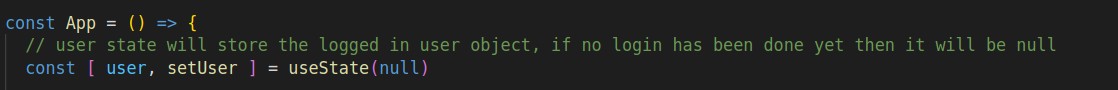
|  |  |  |  |
| --- | --- | --- | --- |
| **Endpoint** | **HTTP**  **method** | **Input** | **Description** |
| /api/users | GET | – | Get all the users from the database |
| /api/users | POST | username, name, password | User signup → Store user details in database  →Do validation checks in the password specified → Store the encrypted password in the database → Return token (JWT specification), name, username, id |
| /api/login | POST | username, password | User login → Check username → do password validation →Return token, name, username, id |
| /api/watchlists/:watc hlistid | GET | watchlist-id | Return watchlist data with given watchlist ID |
| /api/watchlists?user  =user.id | GET | user-id | Return all the watchlists of the given user |
| /api/watchlists | POST | watchlist-name  , type, user-id | Check JWT token → Create watchlist → Return watchlist name, type, user-id |
| /api/watchlists/:watc hlistid | DELETE | watchlist-id | Delete watchlist with given watchlist-id |
| /api/watchlistInstrum ents | POST | Watchlist-id, instrument-id | Check if watchlist-instrument already exists in the given watchlist → Check if added instrument is of same type as as watchlist → Return symbol, name, type, url, id , watchlist-id, instrument-id |
| /api/watchlistInstrum ents?watchlistId=wa tchlist.id | GET | watchlist-id | Return all watchlist-instruments (symbol, name, type, url, id , watchlist-id, instrument-id, watchlistInstrument-id) which belong to given watchlist-id |
| /api/watchlistInstrum ents/:watchlistInstru ment.id | DELETE | watchlistInstru ment-id | Delete the watchlist-instrument with given watchlistInstrument-id |
| /api/watchlistInstrum ents?watchlistId=wa tchlist.id | DELETE | watchlist-id | Delete all instruments which belong to given watchlist-id |
| /api/instruments | GET | – | Return all the instruments stored in database |

* + 1. **Frontend**

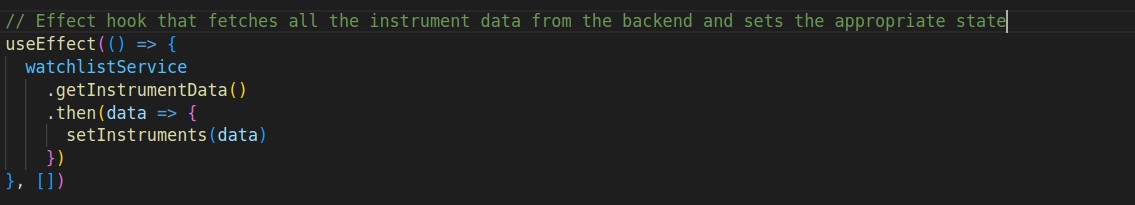
### Directory structure for frontend

We use functional components, React state and React hooks in order to build the frontend. In order to enable routing we have used React Router (BrowserRouter).

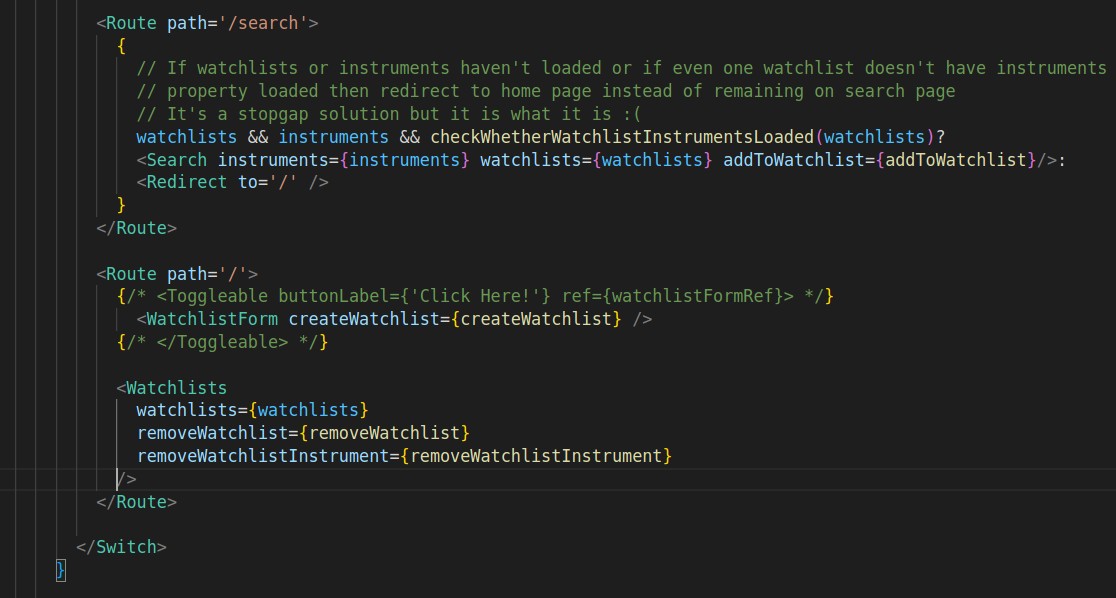
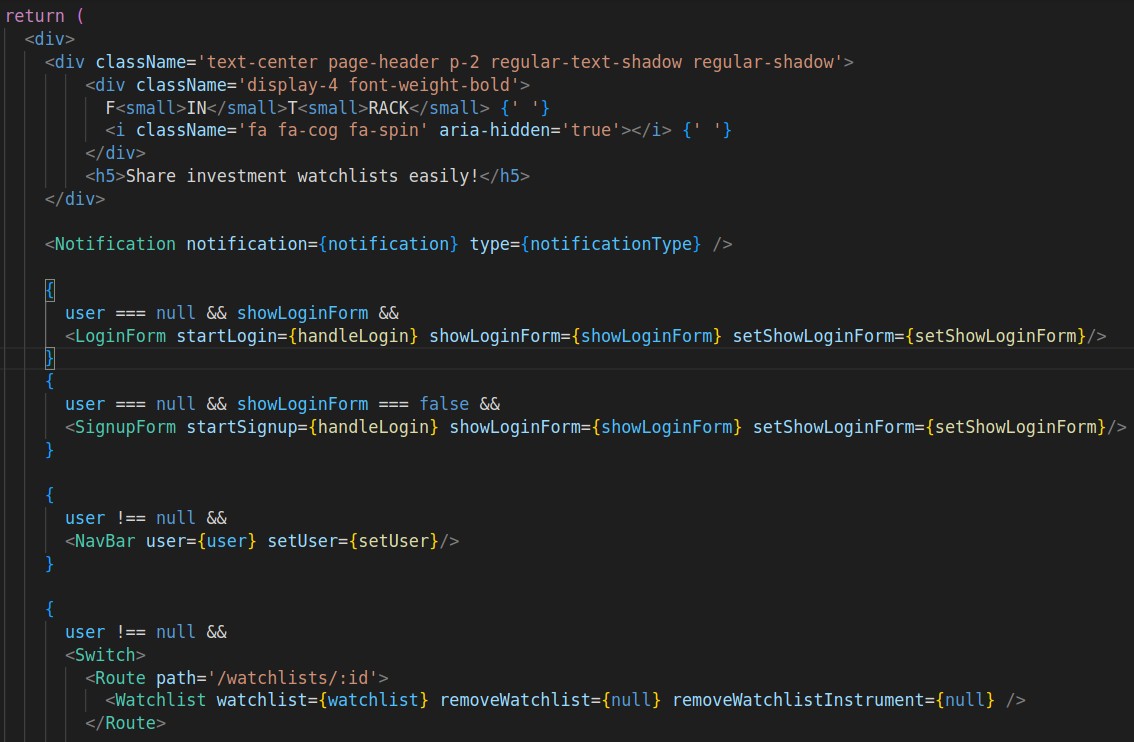
**Components**

**Start of the App *component (App.js)***

### How React Router is used to match routes appropriately

**Effect Hook usage**

### Effect Hook + React Router to fetch correct watchlist data

**JSX that the *App* component returns**

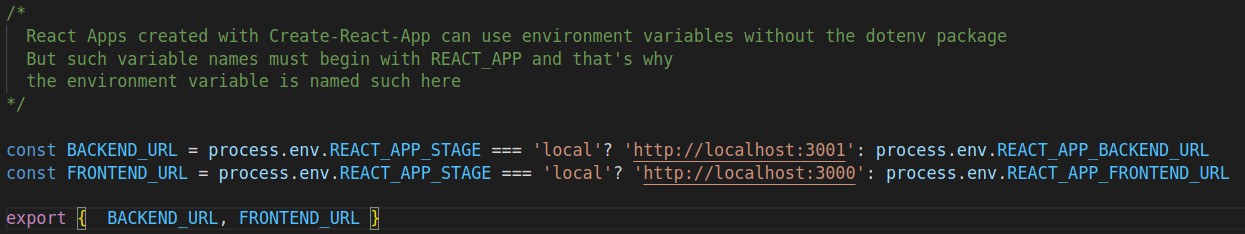
### Services

**Login & Signup Service**

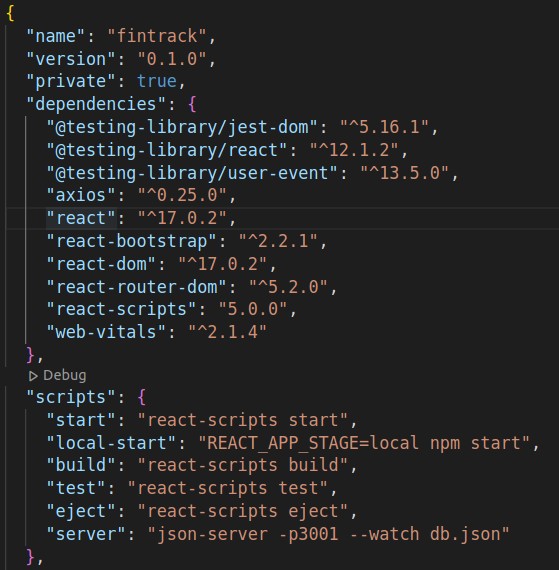
### Create & Delete Watchlist Services with Token Authentication

Similar structure is then followed for all other required services.

**URL Management with Environment variables: config.js**

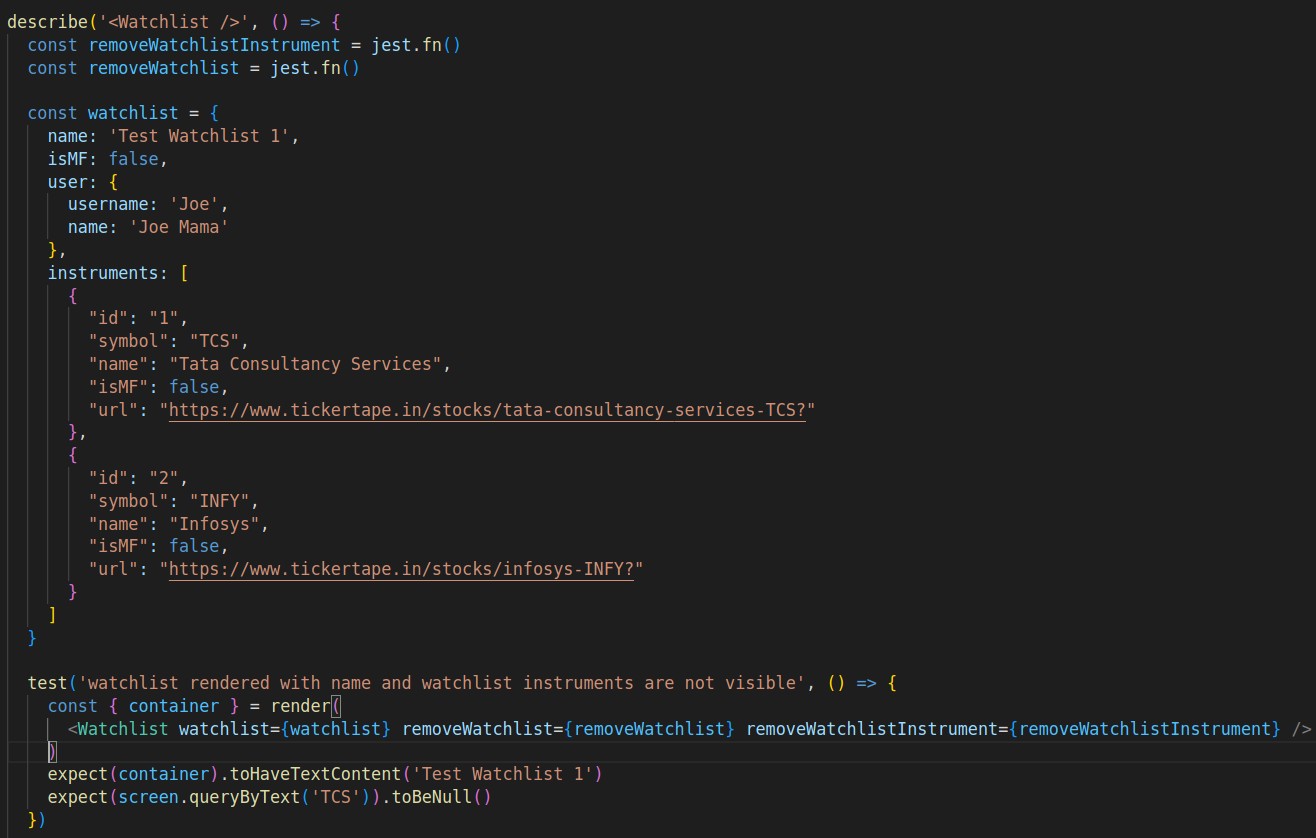


#### .env.development file

How the REACT\_APP\_STAGE variable is used in the *package.json* scripts,

We know the IP addresses of the backend and frontend because these are made static in the Docker-Compose of the backend and frontend containers respectively.

### Testing

Snippet of the *Watchlist.test.js* file which tests the *Watchlist* component along with mock data.

Similar testing format is used for other component tests with *react-testing-library.*

# Result and Discussion

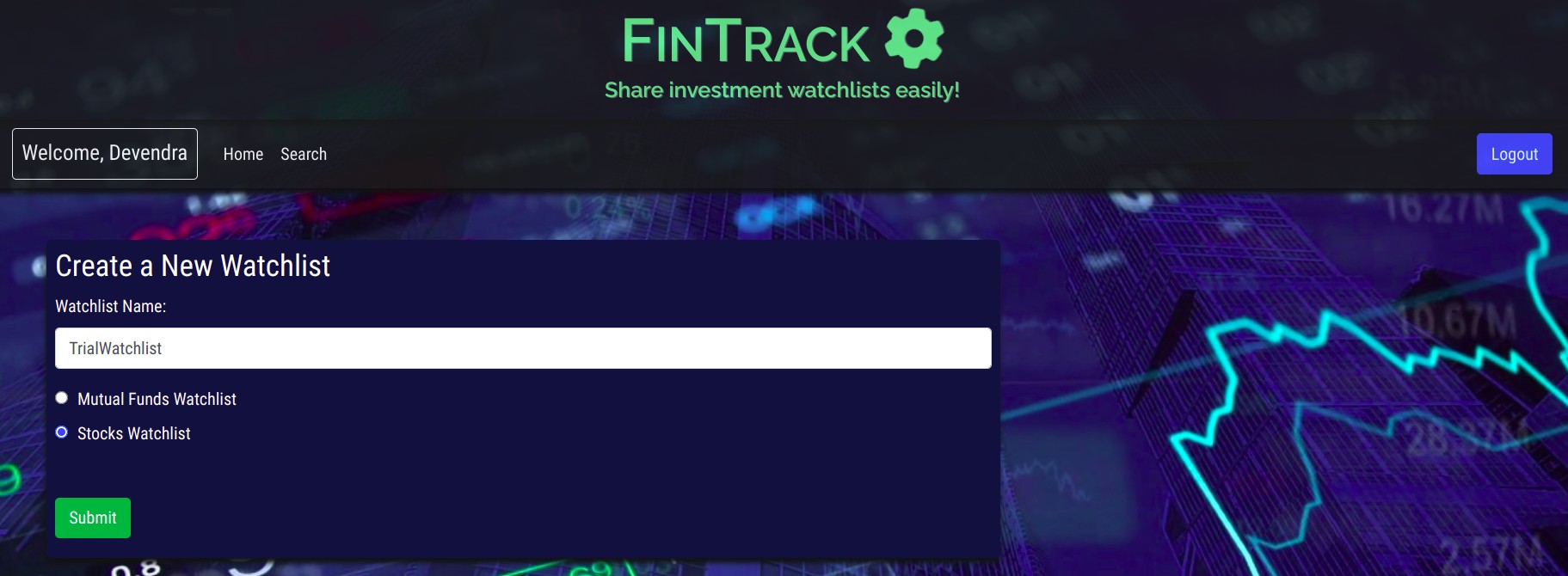
## Login Page

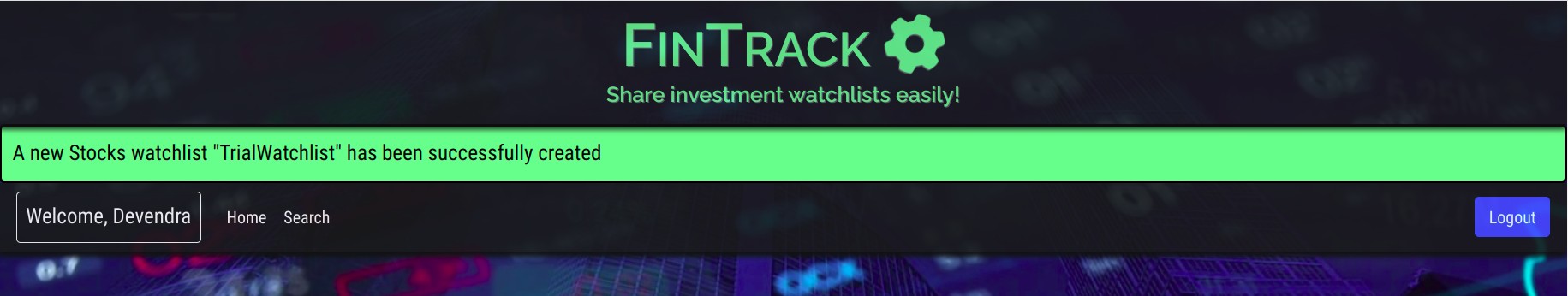
* 1. **Signup Page**



## Home Page

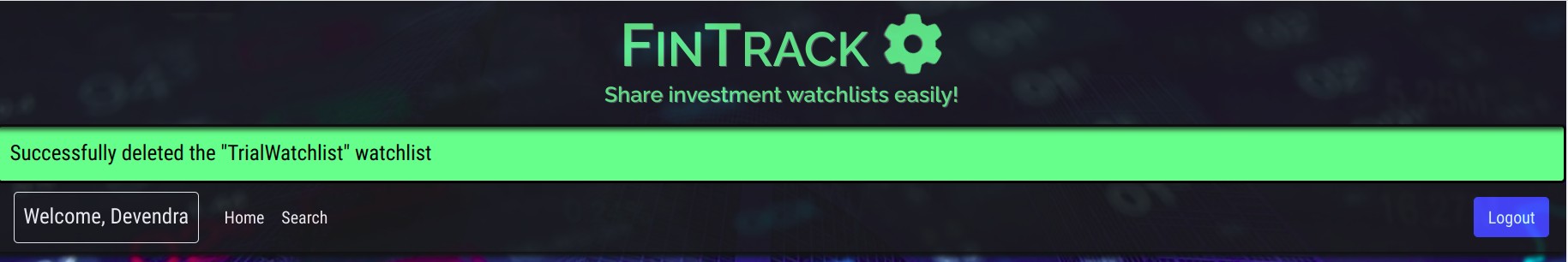
* 1. **Watchlist Form**



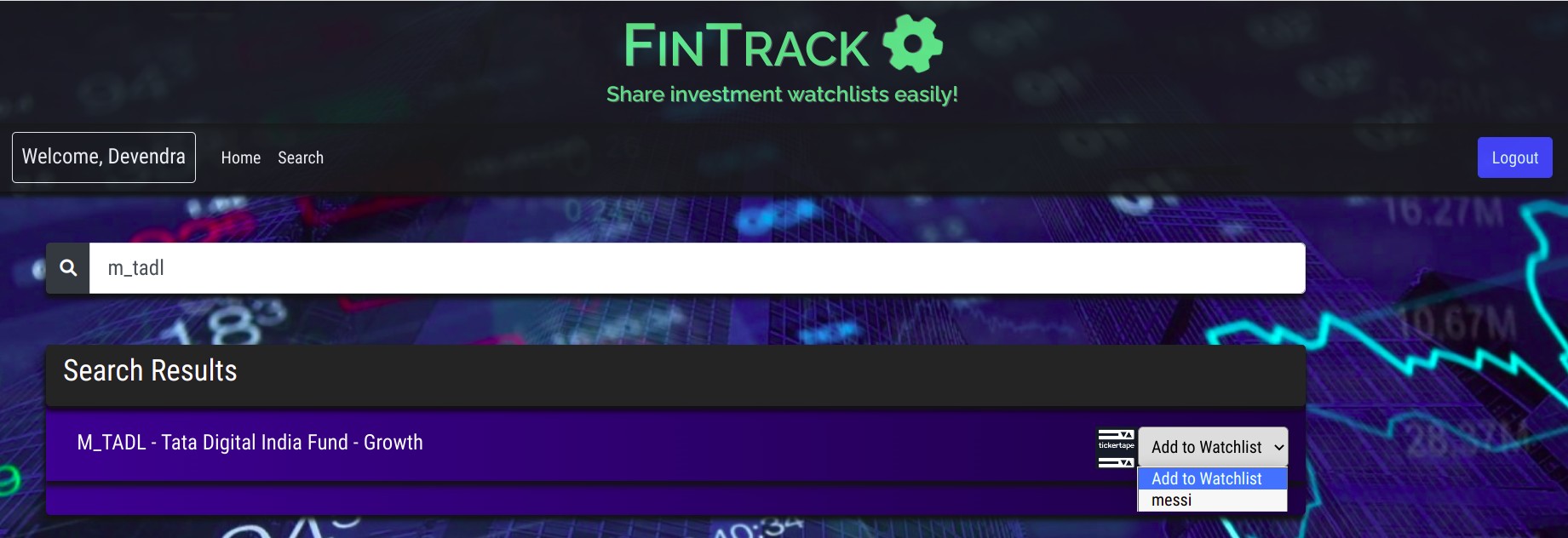
**Watchlist created successfully**

## Delete a Watchlist

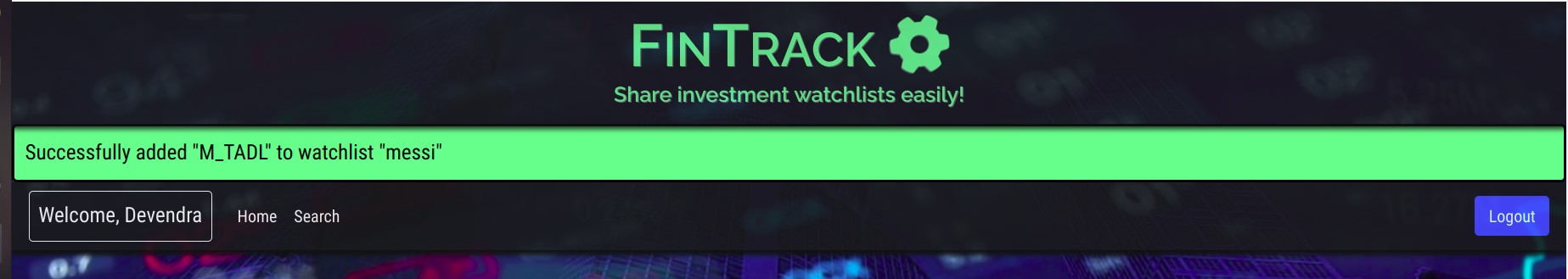
**Watchlist deleted successfully**



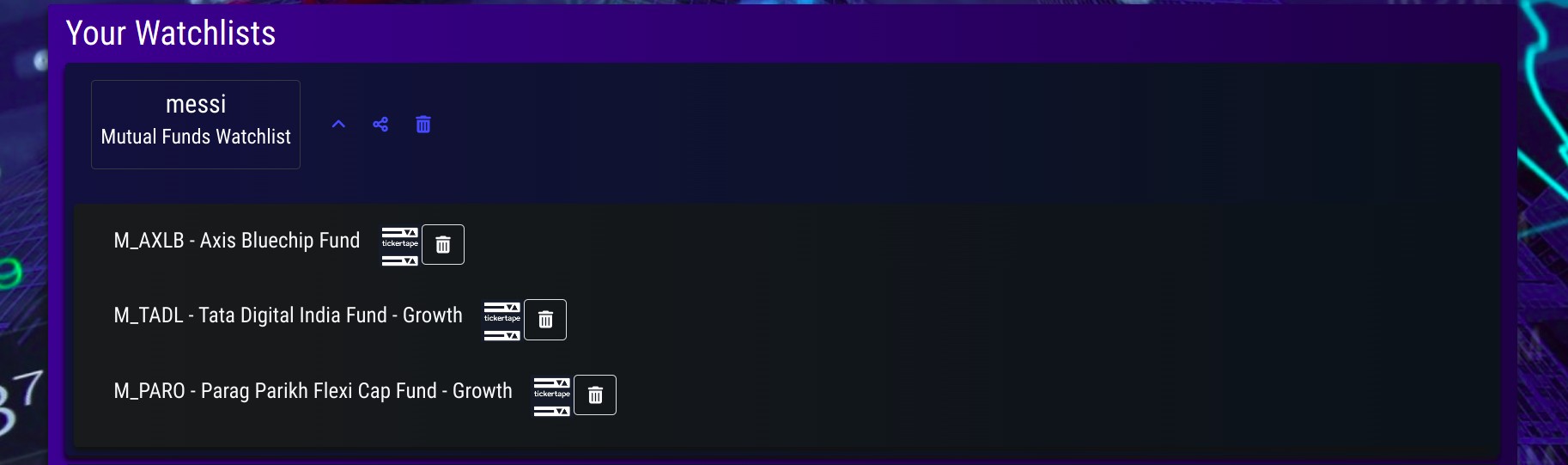
## Search Instruments

* 1. **Adding an Instrument in a Watchlist**

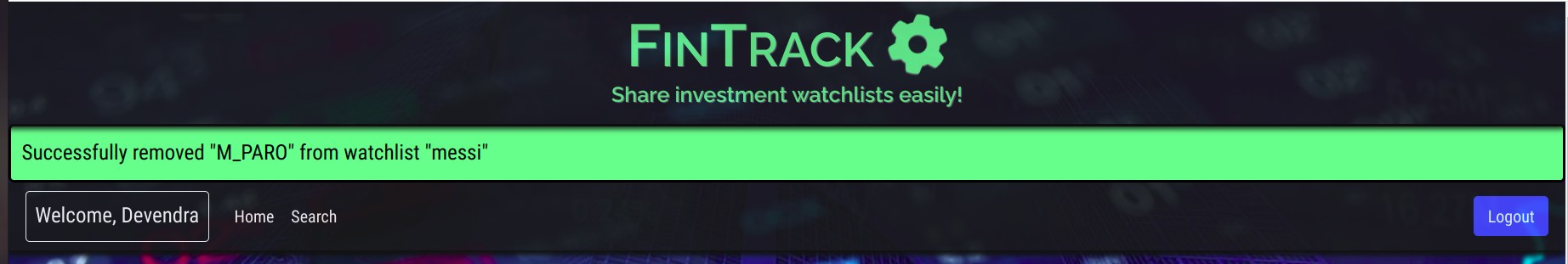
**Successfully added instrument to watchlist**



## View a Watchlist

* 1. **Delete an Instrument from a Watchlist**

**Successfully deleted the instrument from watchlist**



## Share Watchlist

**Viewing shared watchlist from another account**



# Scope for Future Work

### Rename a watchlist

Users can rename their already existing watchlists.

### Rearrangement of instruments in watchlist

Users can rearrange the instruments in any of their watchlists as per their priority.

### Dynamic presentation of actual price of instrument

Users can view the dynamic rate of change of prices of stock/MFs.

### Add end-to-end testing

Test the entire application from frontend to backend via automation tools like Selenium, Cypress, etc.

### Deployment on EC2 instance

Deploy the project on Amazon EC2 instance or a proper IaaS deployment basically.

# Conclusion

We have successfully created the web-app *Fintrack* that will be helpful for users to share their watchlists with anyone they would want to.

We have integrated the entire DevOps CI/CD pipeline with the help of Jenkins.

Used Docker containers for increasing portability and adopting a microservice architecture development.

Used Docker Compose to orchestrate container deployment and manage them.

Used supertest for backend API testing and the React Testing Library for frontend testing.

Used Ansible for deployment and Ansible Vault with Jenkins for secrets management with templating via Jinja2.

We also visualized the logs on the ELK stack in an Elastic Cloud cluster.

And finally we also provided a PaaS deployment with the Heroku + Netlify + Github Actions alternative.

# References

1. <https://fullstackopen.com/>
2. <https://github.com/john-smilga/node-express-course>
3. <https://reactjs.org/docs/getting-started.html>
4. <https://expressjs.com/>
5. <https://mongoosejs.com/docs/api.html>
6. <https://www.npmjs.com/>
7. <https://docs.ansible.com/>
8. <https://blog.ktz.me/secret-management-with-docker-compose-and-ansible/>
9. <https://www.freecodecamp.org/news/how-to-deploy-react-router-based-app-to-netlify/>
10. <https://github.com/marketplace/actions/deploy-to-heroku>