Software Production Engineering

Course Project

MT2023006 Sreeparna Das



**Introduction**

This is a Blog Management System designed in React for adding blogs, editing blogs, and more. We used the MERN stack (MongoDB, Express, React, and NodeJS) to build this application.

**Github Repo Link: https://github.com/sree04/blog-application.git**

#### DevOps tools:

**Source Control Management:** Git and GitHub

**Continuous Integration Pipeline:** Jenkins **Containerization:** Docker

**Container Orchestration:** Docker compose



**Front End:** React, Tailwind CSS, Vite

**Logger:** Winston **Monitoring:** ELK Stack **Database:** MongoDB **Testing:** Chai, Jest

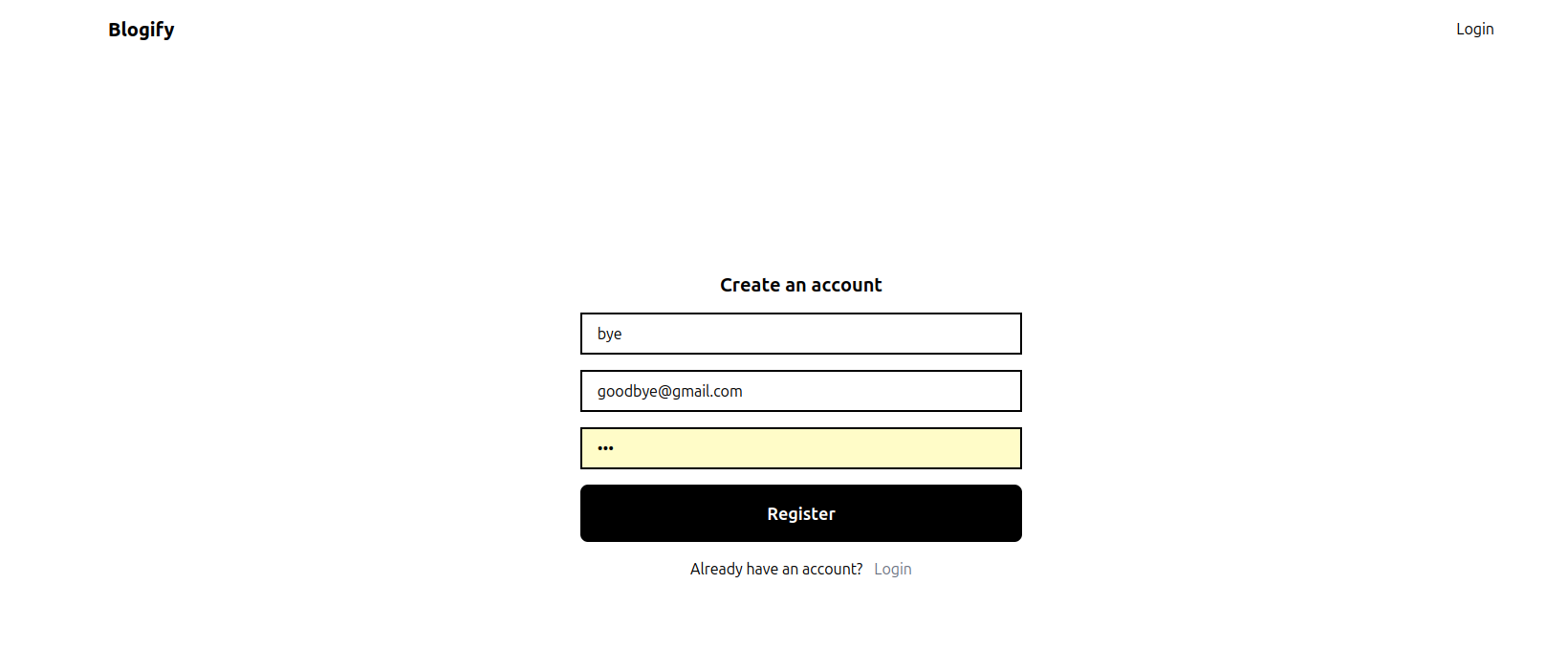
#### Other dependencies

* **bcryptjs -** For password hashing
* **jsonwebtoken** - For JWT auth
* **express -** For the backend server
* **mongoose** - For MongoDB operations

#### Features

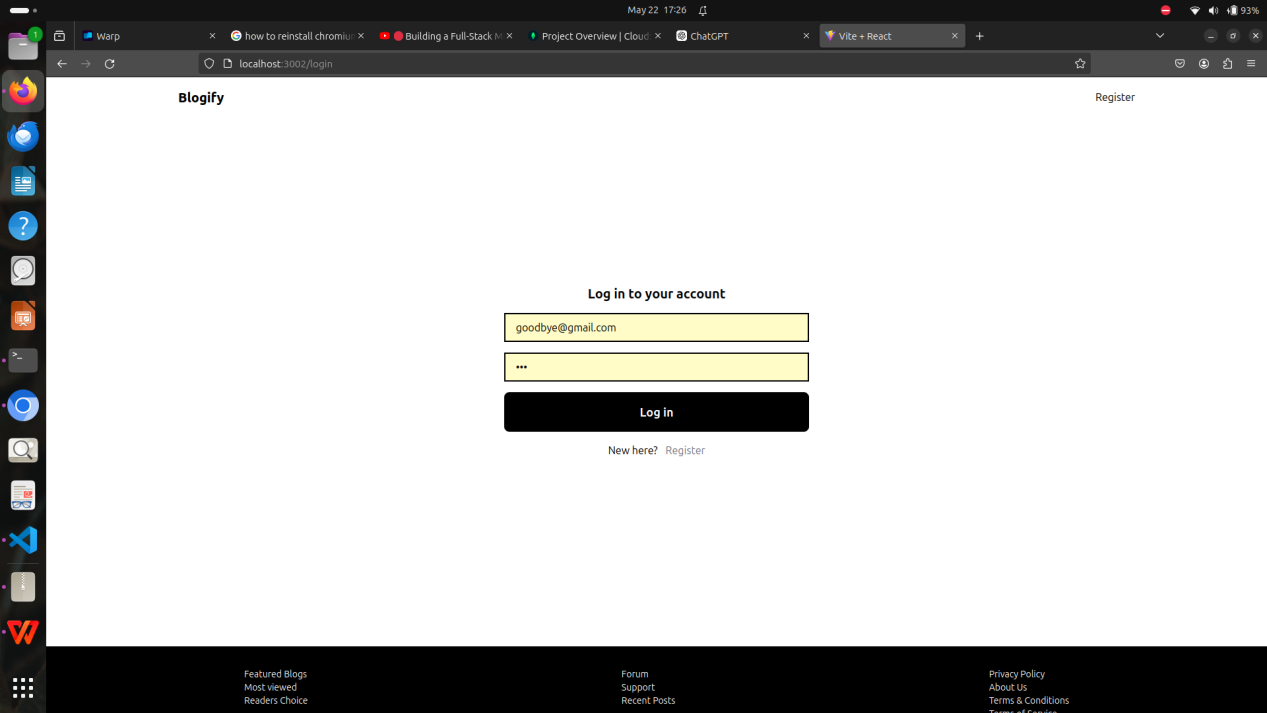
* **Register**

New Users can register with the system.



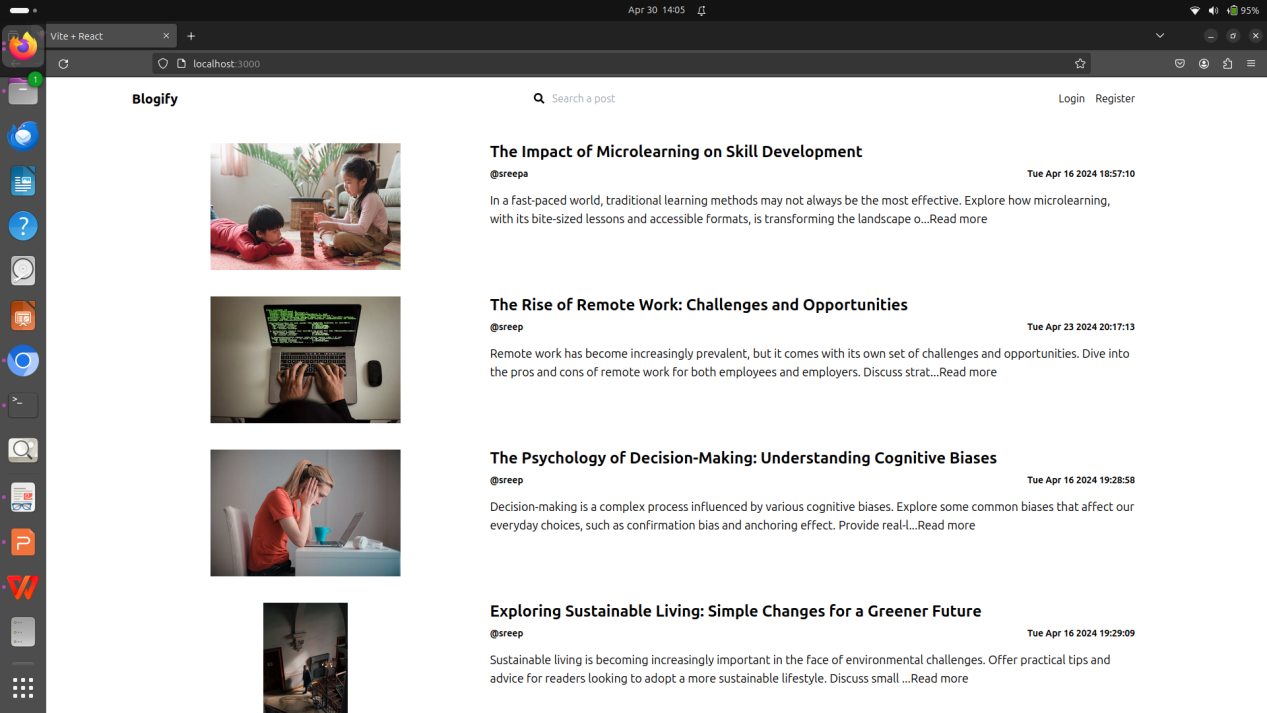
#### Login

Registered users can log in to the system with their credentials.



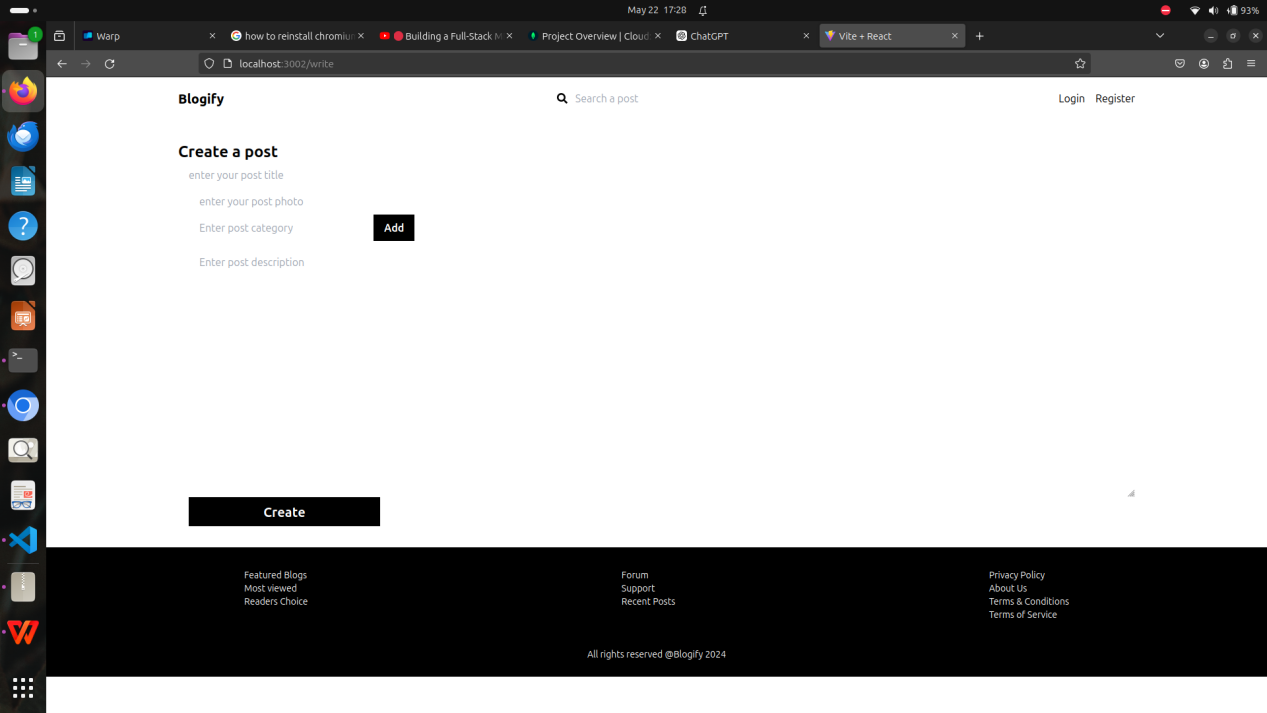
#### List of blog posts

Upon logging in, the user will be displayed a list of blogs and each blog shows when it was posted and by whom it was posted



#### Write posts

Upon logging in, the user will be displayed a list of blogs and each blog shows when it was posted and by whom it was posted



## Introduction

## What is DevOps ?

## DevOps is a development paradigm that is an evolution of earlier principles like Agile or

## Waterfall model. We have developed paradigms that have improved the efficiency and efficacy

## of software development.

## While the waterfall model is all but obsolete, agile methodologies bring faster development

## cycles to finish the programming phase as fast as possible.

## With DevOps, we acknowledge that programming and in extension any form of software

## development is not isolated to any steps and instead is a holistic development that requires you

## to have multiple phases all intertwined and also continuously working to upgrade each of those

## phases as well. Hence it does not make sense to have hard demarcation and teams and also to

## have strict segregation into phases.

## Agile connected and removed the walls between the customer and the development team. With

## DevOps methodology we are further breaking down the wall between the development team

## and the operations and testing team.

## The major breakthrough of such an approach is that we will get the opportunity to have

## continuous development by the development team, continuous deployment of features by the

## operations team guarantee stability while the customer and end user can give feedback.

## Parallely, the testing team finds and reports bugs with the release as well.

## This closely integrated working of multiple teams that have good understanding of not only their

## core tasks but also respect the tasks of other teams will allow all players involved to work in a

## harmonious fashion and give much better productivity and overall lesser finger pointing and

## blame game.

## How to implement DevOps ?

## With such advantages it is not unlikely to have a proper set of tools that allows such possibility.

## DevOps requires a set of tools that allows all of the stakeholders of the project to collaborate

## and work on the project. Then we also need tools to automatically run test cases that can test if

## the feature is working properly and also if the addition of a new feature has broken any of the

## old features as well. Once we have verified that we need a tool to run the entire project in a

## replicable environment that allows the stakeholders to check and see if the product meets their

## expectations. Finally we also have multiple instances of the product running across multiple

## servers for testing, monitoring and redundancy purposes. This full repertoire of features and

## technology stack allows the end user to use the project in peace knowing that the quality check

## is of the highest level and quality assurance and backup is top notch.

## On the technical side of things, to implement the above mentioned scenarios, we need the

## following tools:

## ● IDE:

## We need a suitable IDE that supports the integration of Maven which is a build

## tool that allows us to automate the build and testing process of locally generated and

## modified code. We also need an IDE that supports integration with a Git. This is

## necessary for version control as well as collaboration of multiple developers to parallelly

## work on a project without interfering with each other's feature development.

## ● Git (VCS):

## This is needed by developers to allow them to roll back any changes or also to

## work on multiple features or versions without disturbing a known working configuration of

## the project. This is not to be confused with GitHub.

## ● GitHub (Remote VCS & Collaboration):

## It is not enough that one developer is able to work on multiple versions on the

## project. More likely scenario is that multiple developers are remotely all working on a project that all are assigned different features. Then every developer will have their own

## local copy of code and hence we need a global working repository of the project that can

## be used by others further down the pipeline. Hence the need for Remote Version

## Control. Some examples are GItHub, as used in this mini project or GItLab as an

## alternative. For projects that have sensitive information we can even set up our own

## dedicated Git server which will keep everything in our own cloud storage.

## ● Jenkins:

## Once we have tools to track changes and allow multiple developers to work on

## the same project, we also need to add the operations team into the DevOps continuous

## workflow as well. The first order of priority is to be able to largely automate and

## streamline the process of building the project especially when there are changes made

## to the repository. This can be done using Jenkins along with the GitHub WebHook. This

## means that builds are started whenever changes are pushed to the remote repository

## and the operations team now only needs to monitor Jenkins rather than manually having

## to start jobs.

## ● Jenkins Pipeline:

## Another useful Jenkins feature is the CI/CD Pipeline that you can create. This is

## even more liberating for the operations team as they can chain operations one after the

## other based on the success or failure of each step. Further it also offers the convenience

## that each step of building, testing and deploying can all be monitored from a single point.

## ● Docker:

## With the project now readily available, we also need an easy way to make it

## deployable. This is where we use containers. Docker provides us the ability to wrap up

## the entire project or product that we are developing into an executable container that can

## be exported and imported as a single file with minimal setup or configuration needed at

## the deployment sites. Containers are like virtual machines but are much lighter and

## hence very useful to deploy compared to setting up a virtual machine at each client.

## ● Docker Hub:

## This is an extension of Docker where the same way we have a remote repository

## for Git, we also have a remote repository to store all the containers that we create. This

## means the developers can build and create a container and then anywhere else the

## container can be pulled and tested. Meanwhile, the mini project uses Docker Hub as the

## repository to store the docker image that we create along with the latest tag that allows

## us to mention which is the most recently created image.

## ● Docker Compose:

## Compose is a tool for defining and running multi-container Dokcer applications.

## With Compose, YAML file can used to configure application’s services. Then, with a

## single command, services can be created and started from the configuration. Compose

## works in all environments including production, staging, development, testing and CL

## workflos. It is used to run multiple containers on a single service. It helps overcome

## problem and easily handle multiple containers.

## ● Kubernetes:

## This tool is used to automate the deployment, scaling, and management of

## containerized applications. It allows us to automate having to pull the docker images

## from DockerHub by writing scripts. It also goes one step further to automate the

## deployment specifications as well with backups like replica and auto reloading of

## crashed services to further make our application stable and more customer friendly.

## ● Ansible:

## Ansible is a deployment tool that forms the final step of this mini project. Now that

## the development and operations team are in sync, we also need to eliminate the blame

## game and finger pointing that occurs when a bug or crash happens. To ensure that all

## releases and deployment are in repeatable and consistent environments we will use

## Ansible which offers Infrastructure as a Service (IaaS) which allows us to create identical

## environments for the project. In this mini project we are using Ansible as the final step to

## pull the docker container from Docker Hub and then execute the container which in

## essence contains the project itself.

## ● ELK Stack:

## Throughout the whole development, testing and deployment process we are also

## utilising ELK Stack which stands for Elasticsearch, Logstash and Kibana. We are using

## this to monitor and generate appropriate log files which are really required and helpful

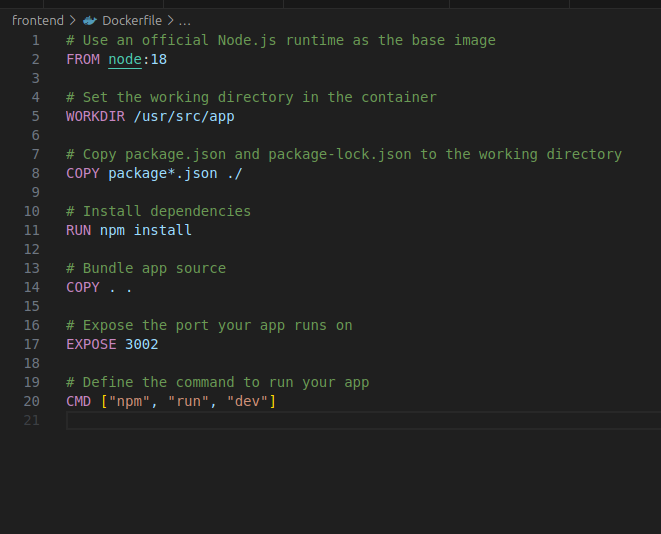
## when things go wrong. To diagnose any crashes or to even monitor the performance of

## certain functions and features, ELK allows us to monitor, collect, summarise and

## visualise the logs.

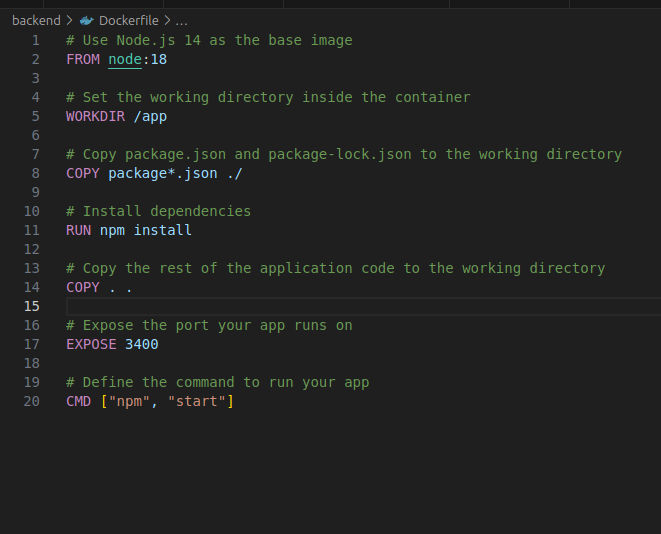
## Docker

#### Front-end docker ﬁle



**Backend docker ﬁle**

#### 

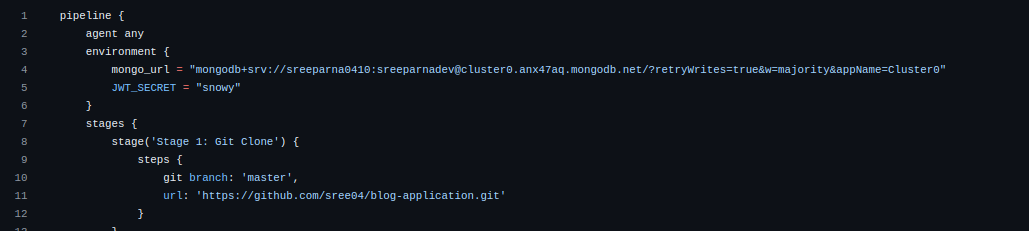


This Dockerﬁle is for a Node.js application. It starts with the Node.js 18 as the base image, sets the working directory to /app, installs the nodemon tool globally for automatic server restarts, and then copies the package.json and package-lock.json ﬁles to the container before running npm ci to install dependencies. The application code is then copied into the container, and it exposes port 3002. The ﬁnal command in the Dockerﬁle utilizes nodemon to run the backend server and npm to run the frontend server.

## Jenkins

We used Jenkins pipeline scm from GitHub. The pipeline script was cloned from the GitHub repository And the code was also cloned from the same repository.

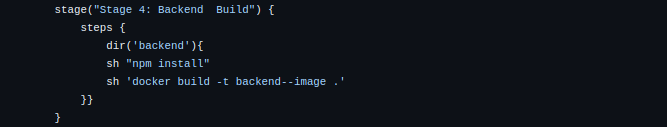
The ﬁrst step was to clone the repo.



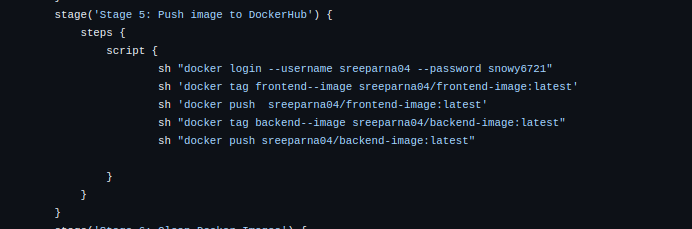
The second step is to build the front-end image. All the contents are required to build the front-end image

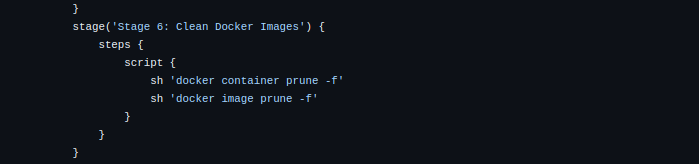


Similarly, the next step is to build the back-end image.



The next step is to push the docker images to the DockerHub.



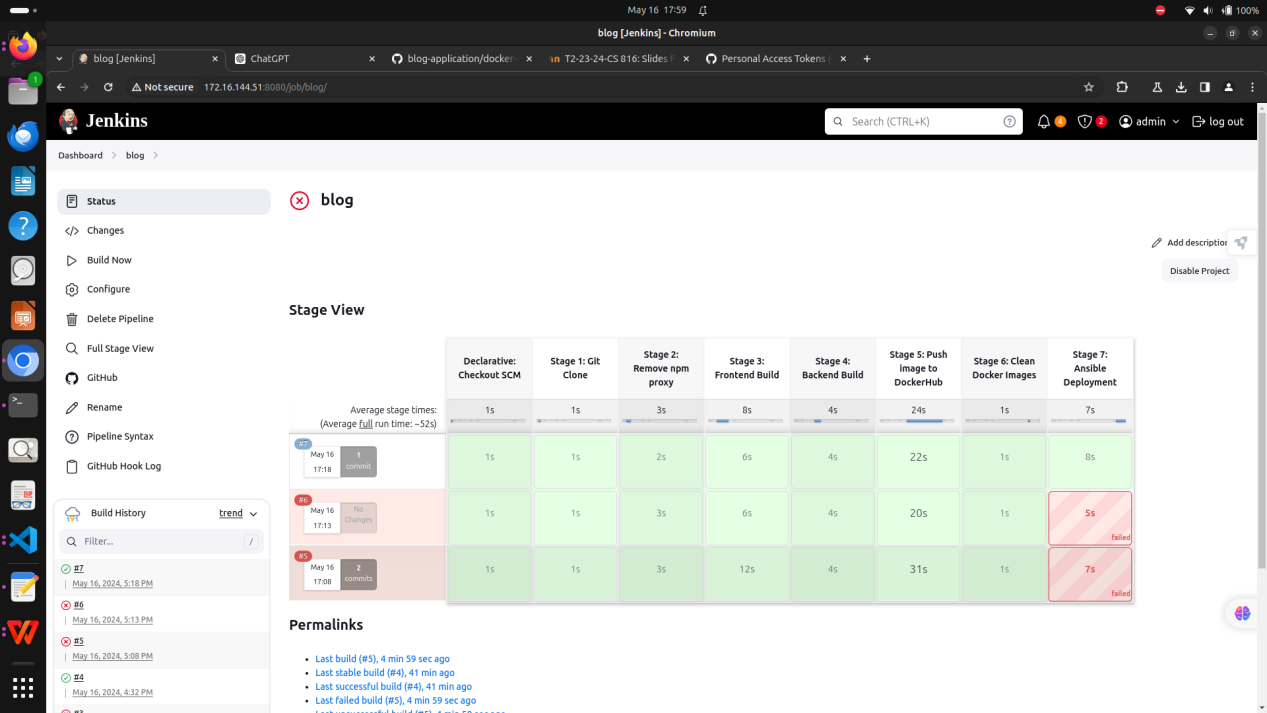


We can use the docker [image/container] prune command to delete old docker and dangling docker images/containers.

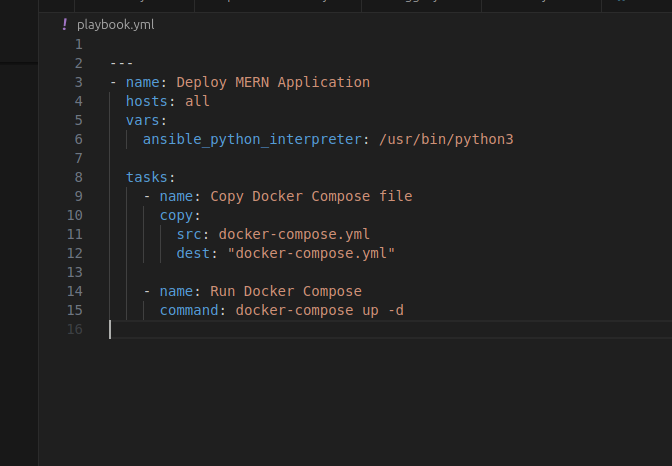
Using Ansible, we can start the docker images by assigning the port and declaring the environment variables.



##### Jenkins Pipeline Result



Playbook.yml has the permissions to the docker-compose and start docker images.



# Deployments

### Using localhost

**npm run dev:** start the front-end Environment variables for the front-end.

**REACT\_APP\_BASE\_URL=http://localhost:3002 nodemon index.js:** Start the back end.

Environment variables for back-end.

##### jwt\_secret=snowy

MONGO\_URL=mongodb+srv://sreeparna0410:sreeparnadev@cluster0.anx47aq.mongodb.net/?retryWrites=true&w=majority&appName=Cluster0

**PORT=3400**

**NODE\_ENV**=production

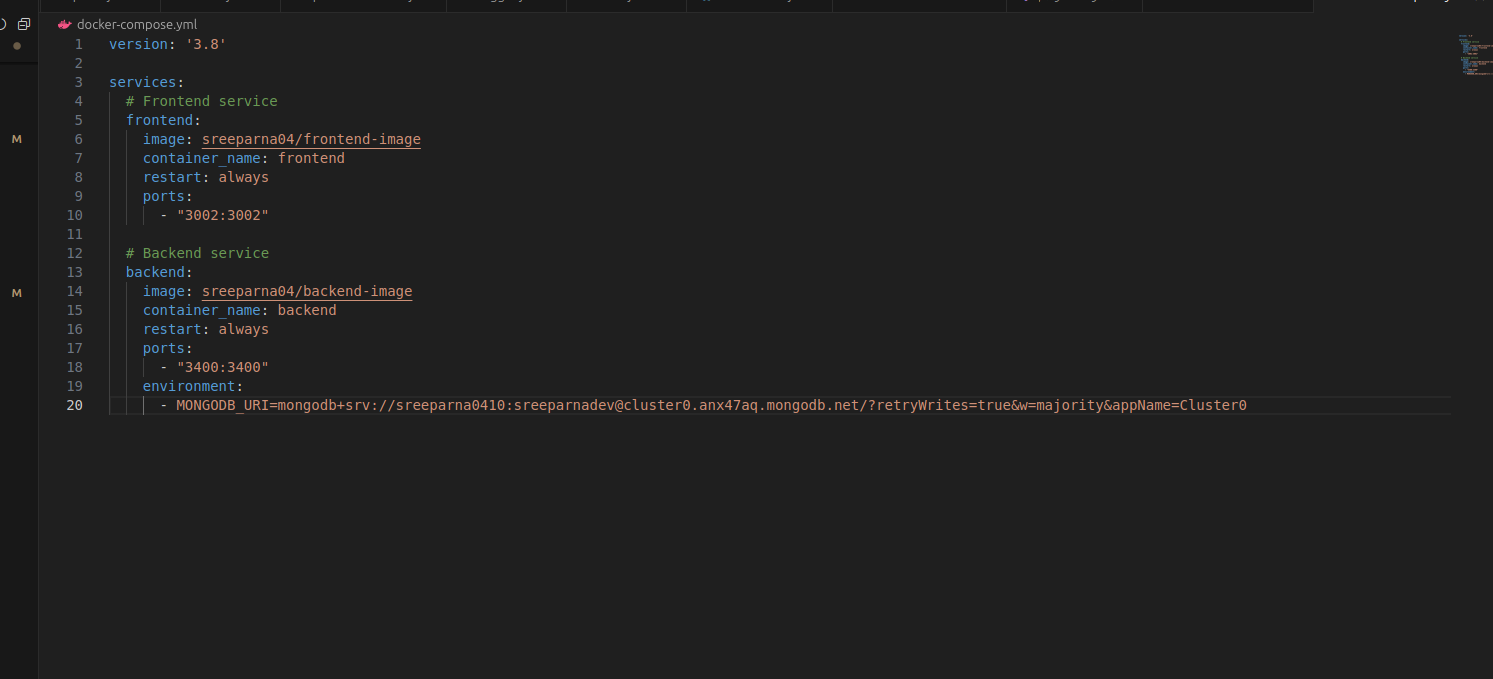
Here we used a cloud database.

### Using docker-compose

* While using docker images, we used the Mongo DB docker image to connect to the database.

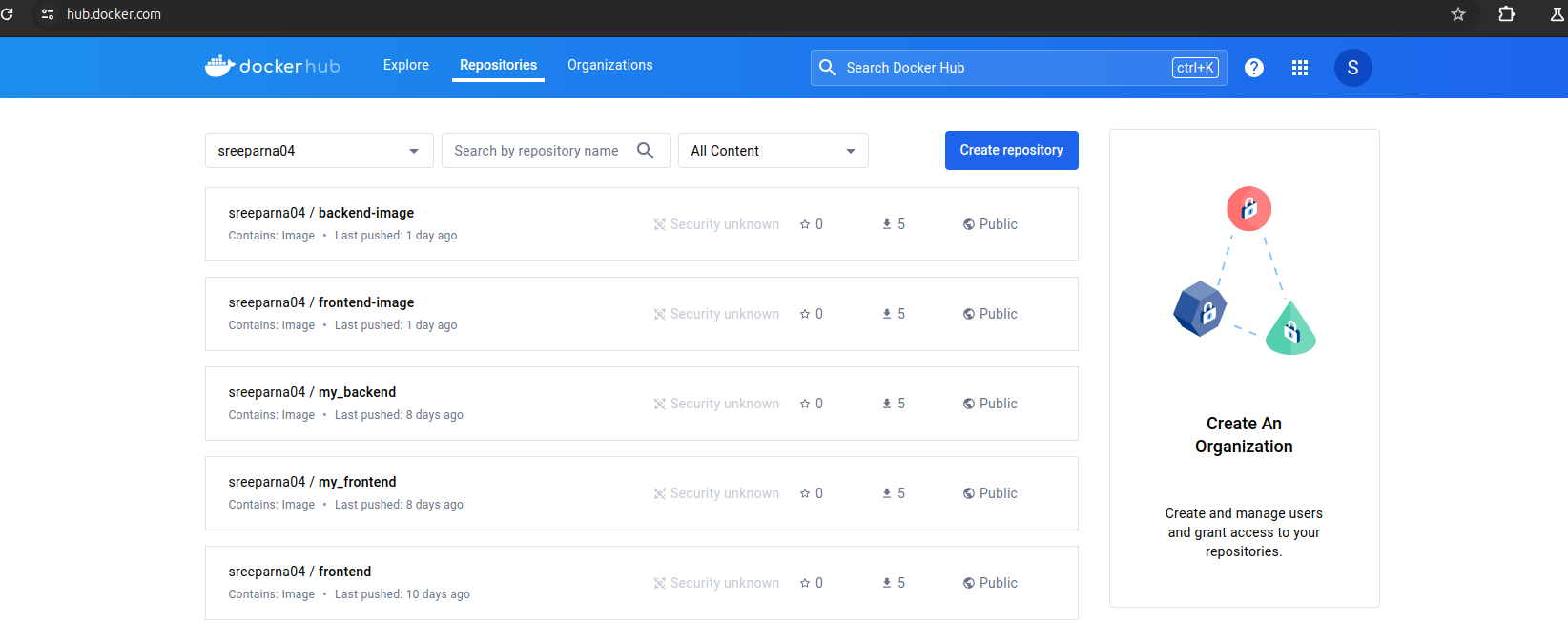
. Communication happens using the local host.

* The front-end image exposes the 3002 port and it connects to localhost:3002
* The backend image exposes to 3400 port and it connects to localhost:3400

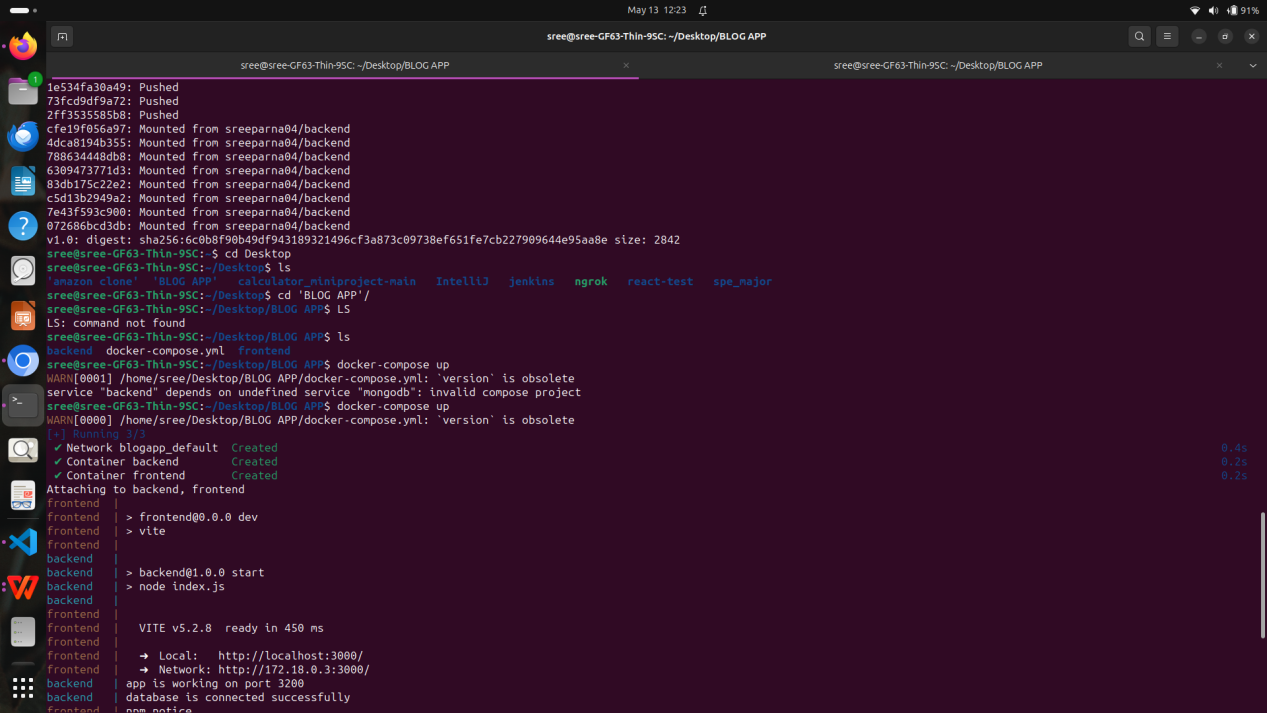


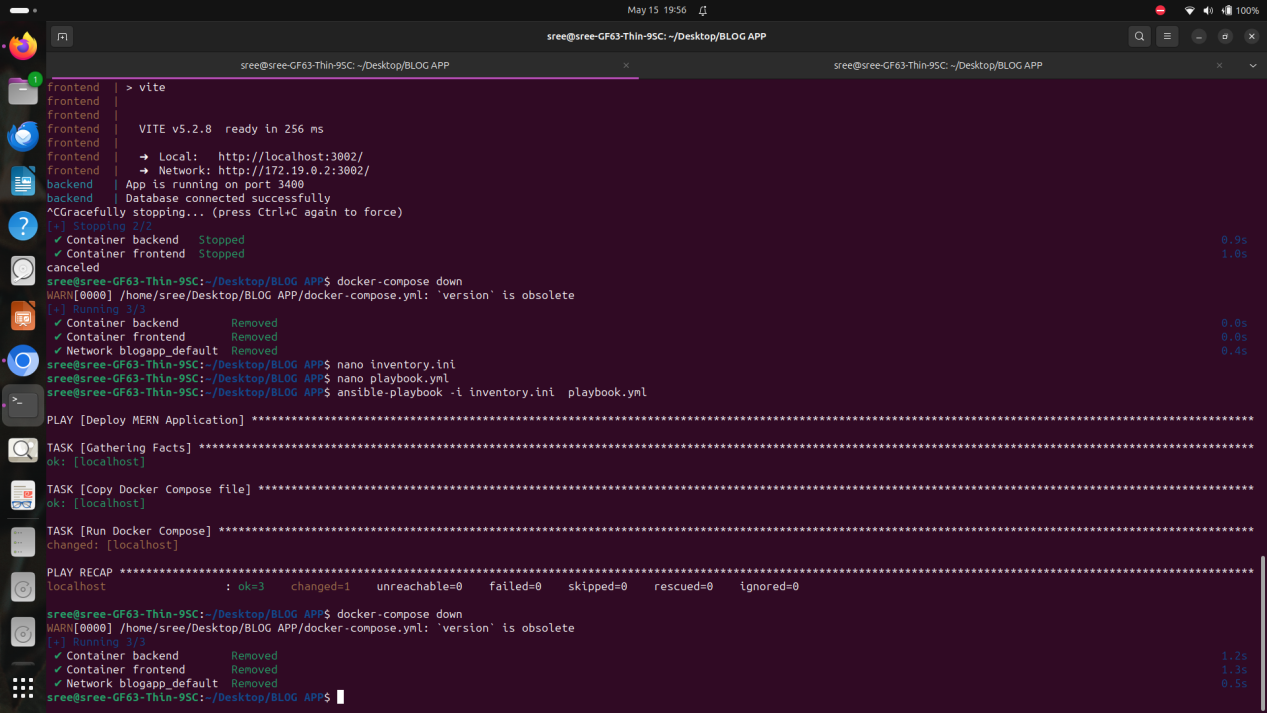
This Docker Compose ﬁle deﬁnes a multi-container environment for a full-stack application with a frontend, backend, and MongoDB database. It utilizes three services: "container\_front" for the frontend, "container\_back" for the backend, and "mongodb\_db" for the MongoDB database. The frontend and backend containers are connected to the "mynetwork" network, enabling communication. The "container\_front" service exposes its application on port 3002, forwarding requests to the backend hosted on port 3400. The backend relies on a MongoDB database with connection details speciﬁed in the environment variables.

This is the image of my DockerHub which is displaying all the images:



Deployment from docker-compose.





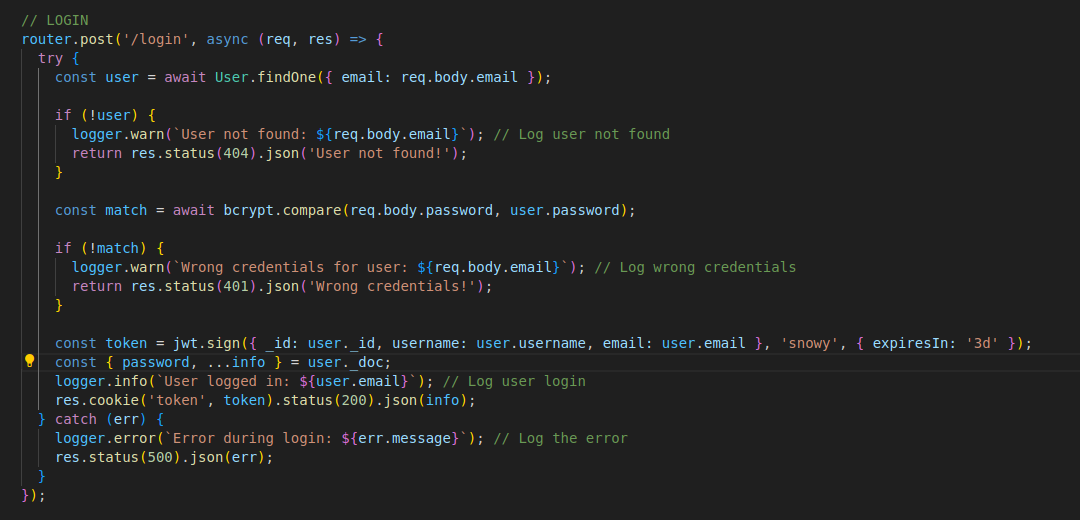
# API Documentation

The **"/api/auth"** route handles user-related operations, allowing for the management of user data. The **"/api/posts"** route facilitates interactions with post-related data, oﬀering endpoints for post-related operations. The **"/api/user"** route is responsible for handling tje modifications or deletion of the users .

Every response has a tag with success. If the response is true, it means the operation is passed, and if the response is false it means the operation is failed, along with an error message.

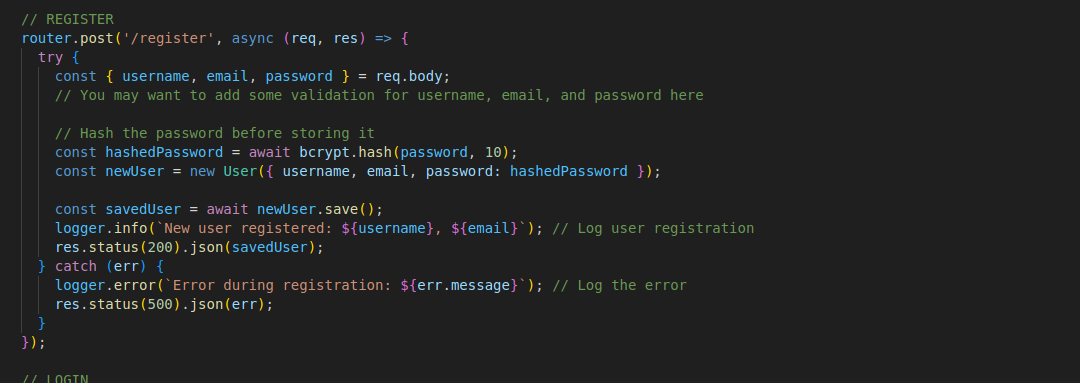
### /api/auth/login

Helps to log in to the user with email and password.



### /api/auth/register

Helps to register the user with username,email and password.



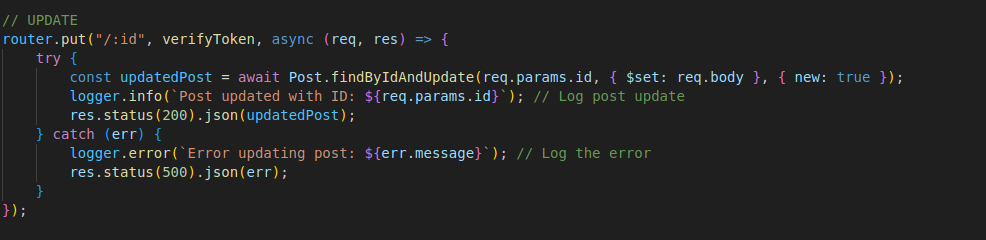
### /api/auth/logout



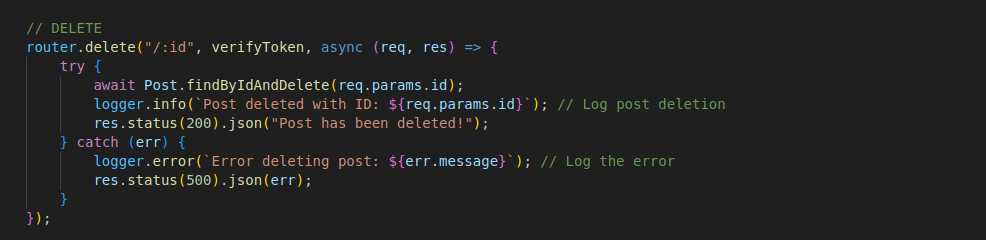
### /api/posts/create



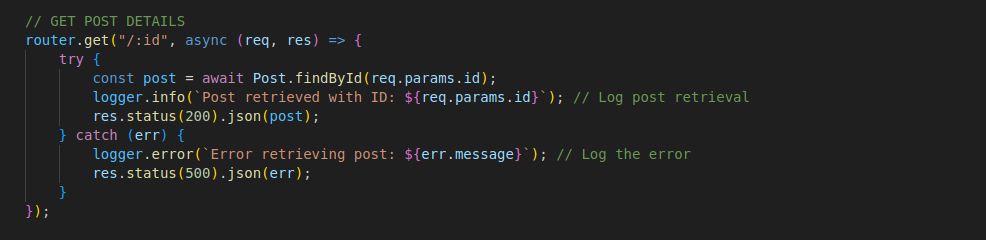
### /api/posts/update



### /api/posts/delete



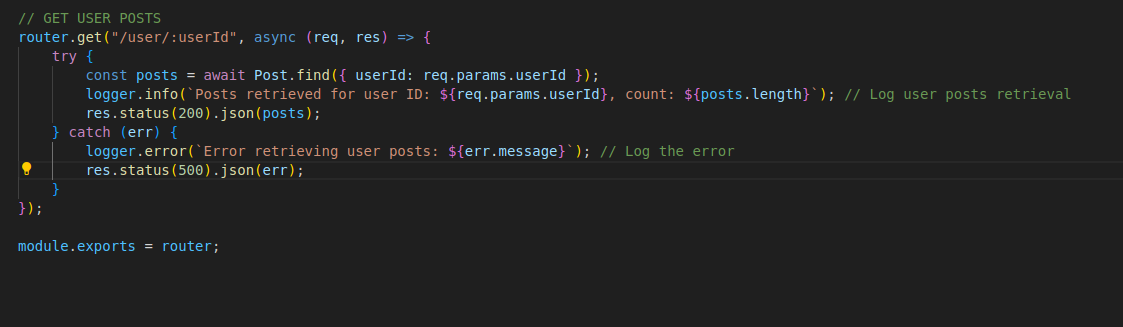
### /api/posts/post details



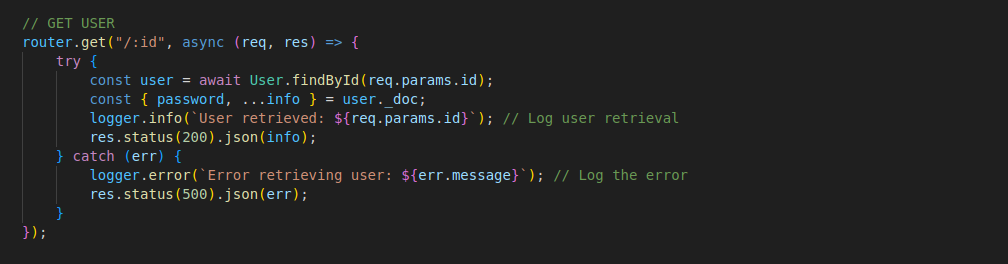
### /api/posts/all posts



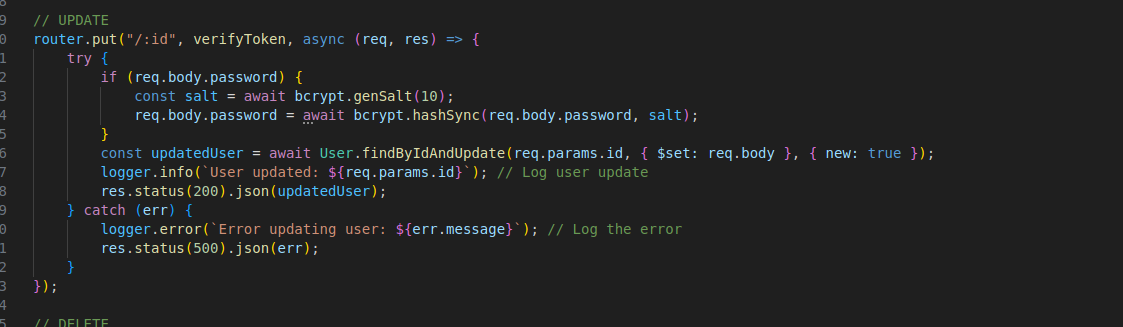
### /api/posts/user post



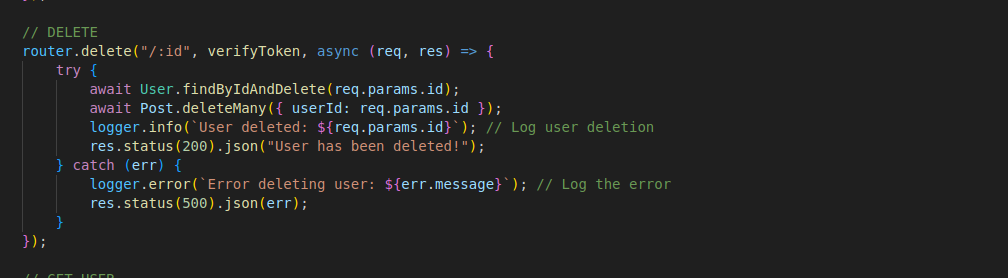
### /api/users/get user



### /api/users/update



### /api/users/delete



# API testing Documentation

The **"/api/auth"** route handles user-related operations, allowing for the management of user data. The **"/api/posts"** route facilitates interactions with post-related data, oﬀering endpoints for post-related operations. The **"/api/user"** route is responsible for handling tje modifications or deletion of the users .

Every response has a tag with success. If the response is true, it means the operation is passed, and if the response is false it means the operation is failed, along with an error message.

**Post API Routes Tests**

**Create a New Post**

Purpose: Verify that a new post can be created.

Test: Sends a POST request to /api/posts/create with post data and authorization token.

Expectation: The response should have a status of 200, and the response body should include the post properties such as \_id, title, desc, username, and userId.

**Update an Existing Post**

Purpose: Verify that an existing post can be updated.

Test: Sends a PUT request to /api/posts/:id with update data and authorization token.

Expectation: The response should have a status of 200, and the response body should reflect the updated post title.

**Get Post Details**

Purpose: Verify that the details of a specific post can be retrieved.

Test: Sends a GET request to /api/posts/:id.

Expectation: The response should have a status of 200, and the response body should include the post properties such as \_id and title.

**Get All Posts**

Purpose: Verify that all posts can be retrieved.

Test: Sends a GET request to /api/posts.

Expectation: The response should have a status of 200, and the response body should be an array containing all posts, with a length of 2.

**Get Posts by User**

Purpose: Verify that all posts by a specific user can be retrieved.

Test: Sends a GET request to /api/posts/user/:userId.

Expectation: The response should have a status of 200, and the response body should be an array containing the posts made by the specified user, with a length of 2.

**Delete a Post**

Purpose: Verify that a post can be deleted.

Test: Sends a DELETE request to /api/posts/:id with an authorization token.

Expectation: The response should have a status of 200, and the response body should indicate that the post has been deleted.

**Return 404 if Post Does Not Exist**

Purpose: Verify that a proper error message is returned when trying to delete a non-existent post.

Test: Sends a DELETE request to /api/posts/:id with a non-existent post ID and an authorization token.

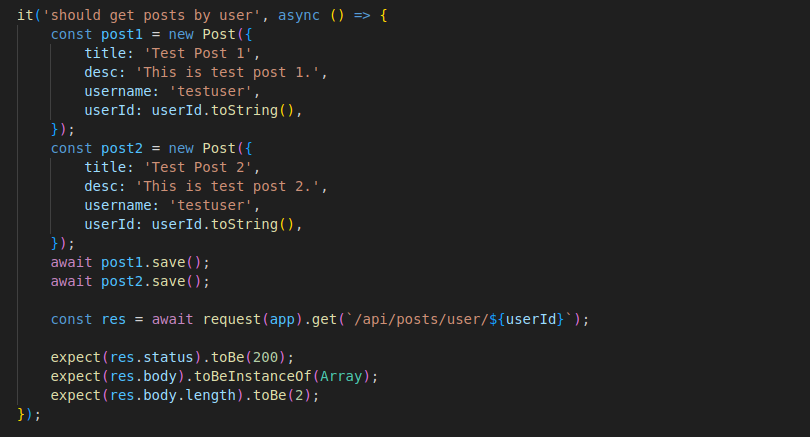
Expectation: The response should have a status of 404, and the response body should indicate that the post was not found.

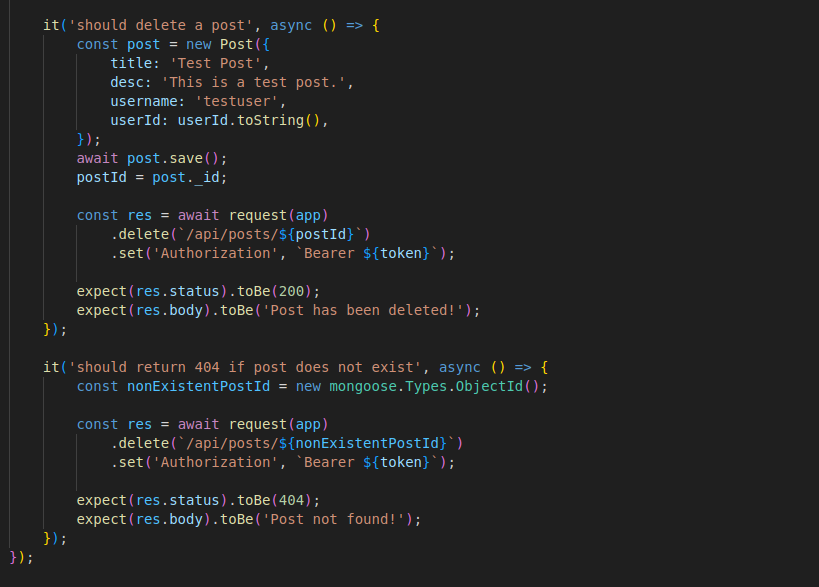












**User API Routes Tests**

**Setup and Teardown**

beforeAll: Sets up an in-memory MongoDB server and connects to it before any tests run. It also starts the Express server.

afterAll: Cleans up by dropping the database, closing the MongoDB connection, stopping the in-memory MongoDB server, and closing the Express server after all tests have run.

beforeEach: Clears the User collection to ensure each test runs with a clean state.

User Registration Tests

should register a new user

Purpose: Verify that a new user can be registered successfully.

Test: Sends a POST request to /api/auth/register with user details.

Expectation: The response should have a status of 200 and should contain the username and email properties.

should return 500 if user registration fails

Purpose: Verify the server returns an error status when registration fails.

Test: Sends a POST request to /api/auth/register with an empty request body.

Expectation: The response should have a status of 500.

**User Login Tests**

**should login a user and return a token**

Purpose: Verify that a user can log in successfully and receive a token.

Test: Creates a new user, then sends a POST request to /api/auth/login with the correct credentials.

Expectation: The response should have a status of 200, should contain the username and email properties, and should set a token in the cookies.

should return 404 if user is not found

Purpose: Verify the server returns a 404 status when a user is not found.

Test: Sends a POST request to /api/auth/login with credentials that do not match any user in the database.

Expectation: The response should have a status of 404 and should indicate that the user was not found.

should return 401 if password is incorrect

Purpose: Verify the server returns a 401 status when the password is incorrect.

Test: Creates a new user, then sends a POST request to /api/auth/login with the correct email but an incorrect password.

Expectation: The response should have a status of 401 and should indicate that the credentials are wrong.

User Logout Test

should logout a user

Purpose: Verify that a user can log out successfully.

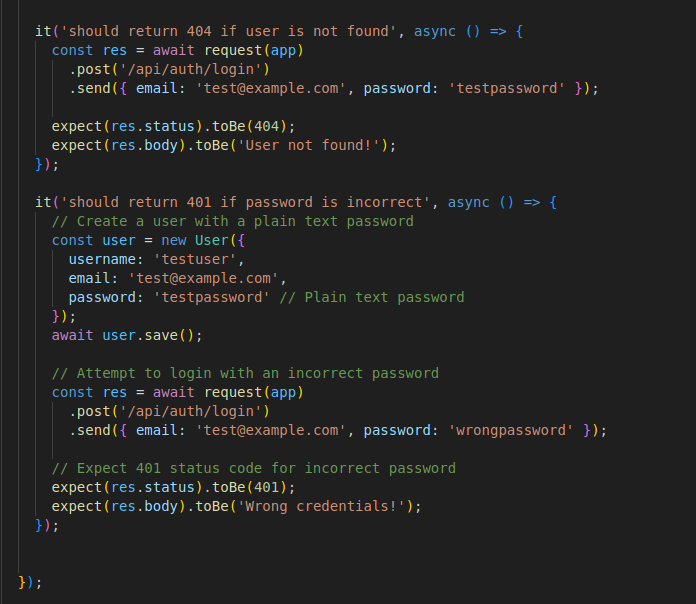
Test: Sends a GET request to /api/auth/logout.

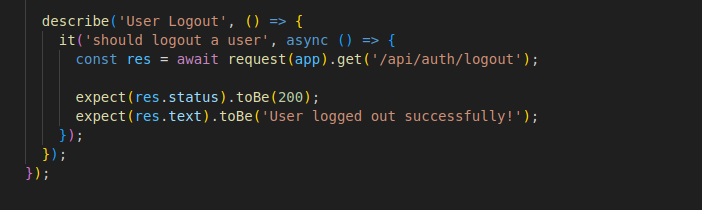
Expectation: The response should have a status of 200 and should indicate that the user has been logged out successfully.

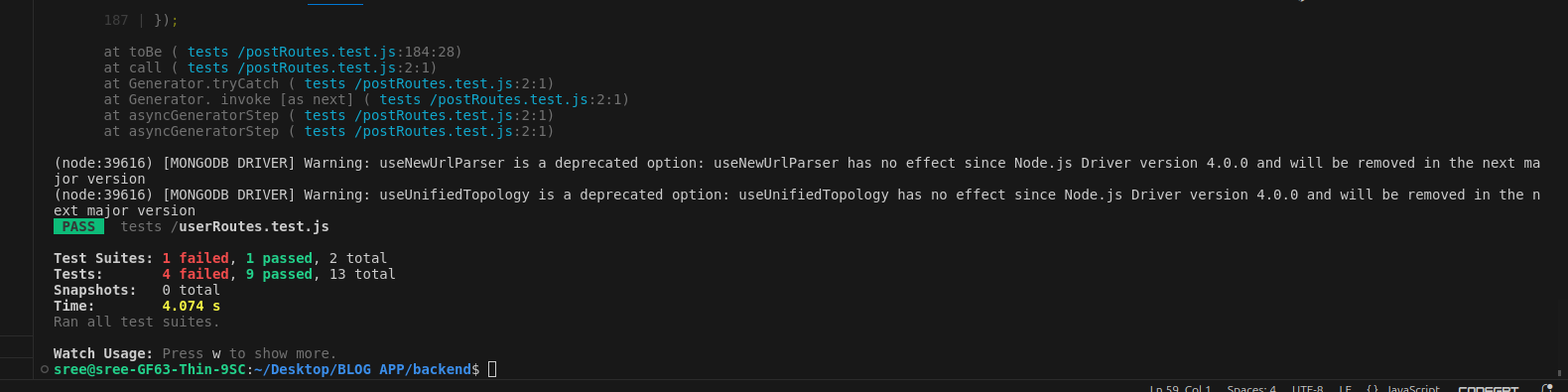




The password is usually saved in db after encrypting the user password.

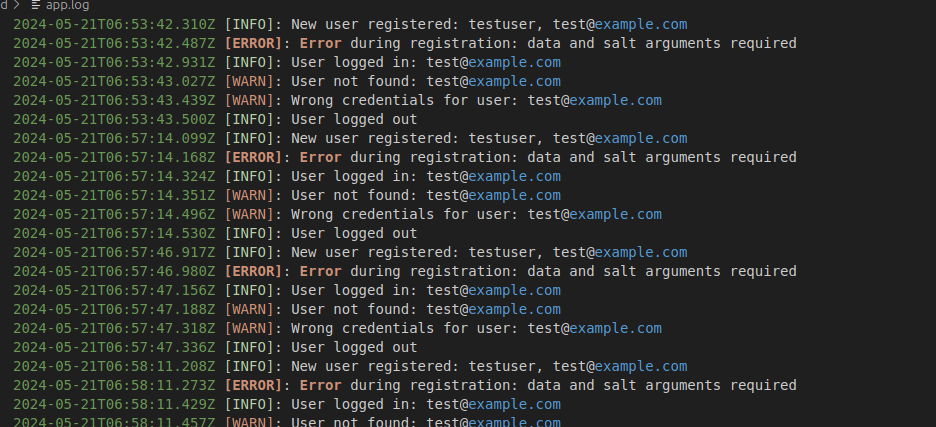






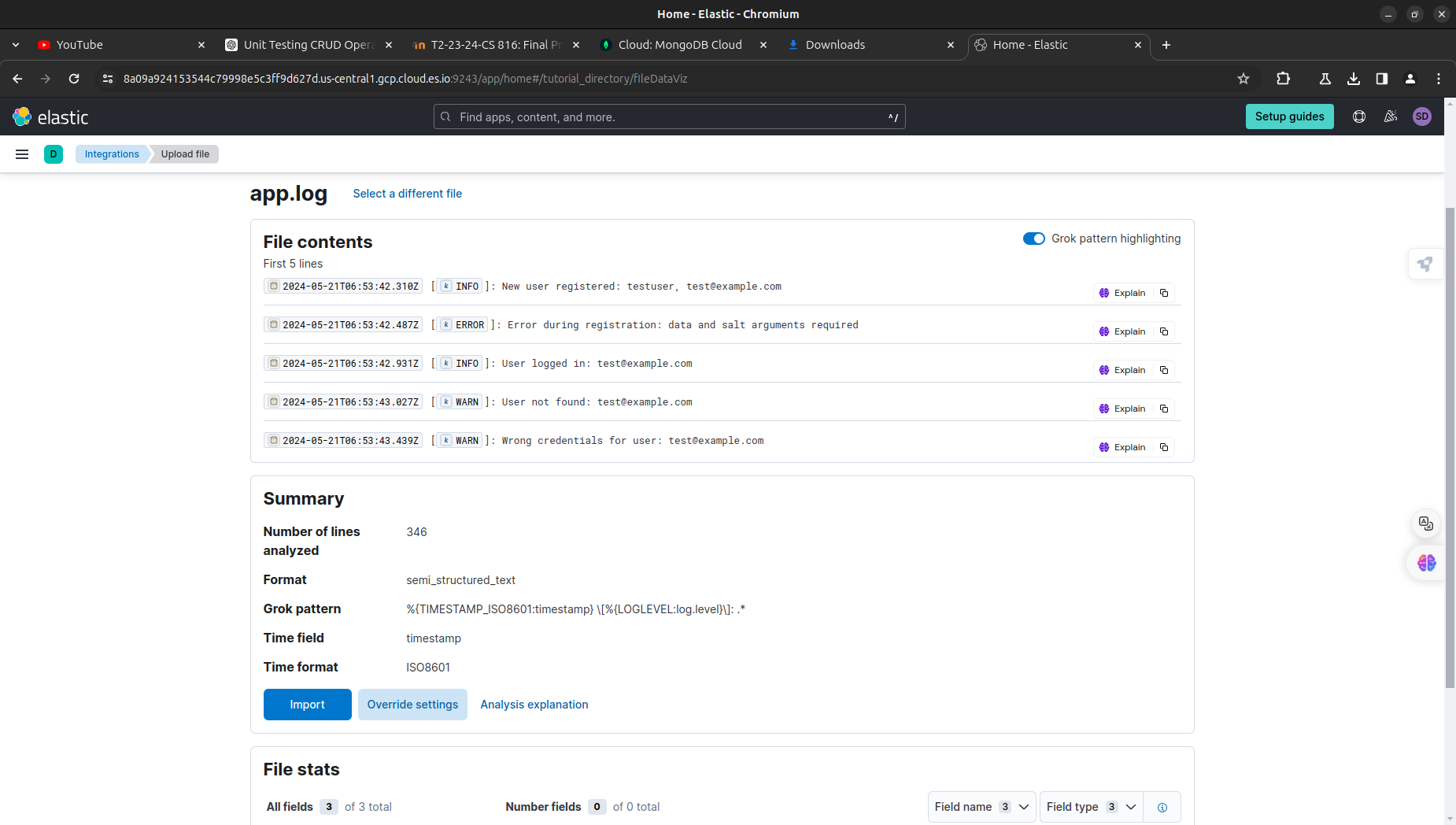
### Logging

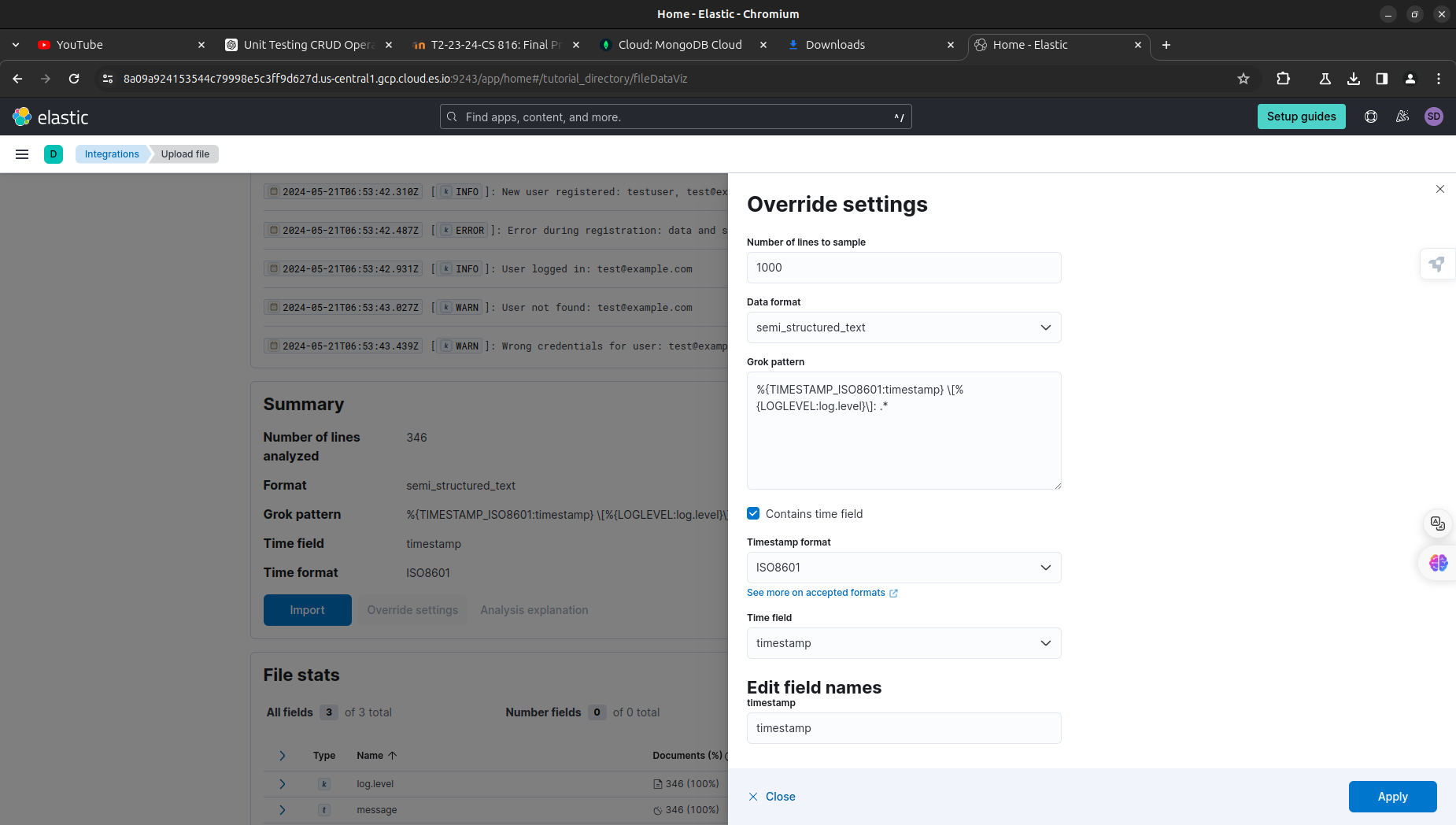
We used the Winston package for logging. We logged every operation and calls that were made in the backend along with the result, time, and type of deployment (test, info, error).



## ELK stack

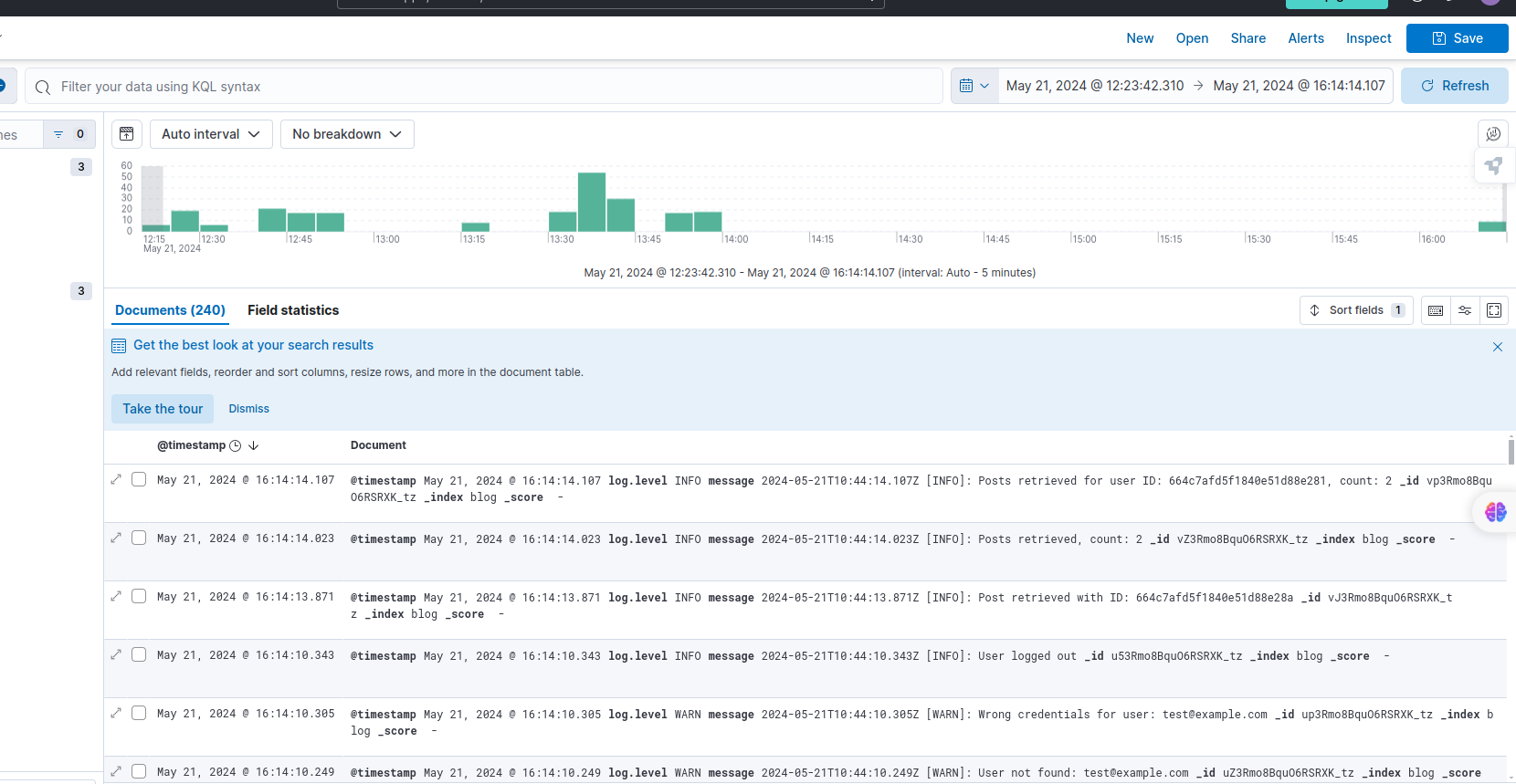
Grok pattern( timestamp ("YYYY-MM-DD HH:mm:ss.SSS") + info/error + message))

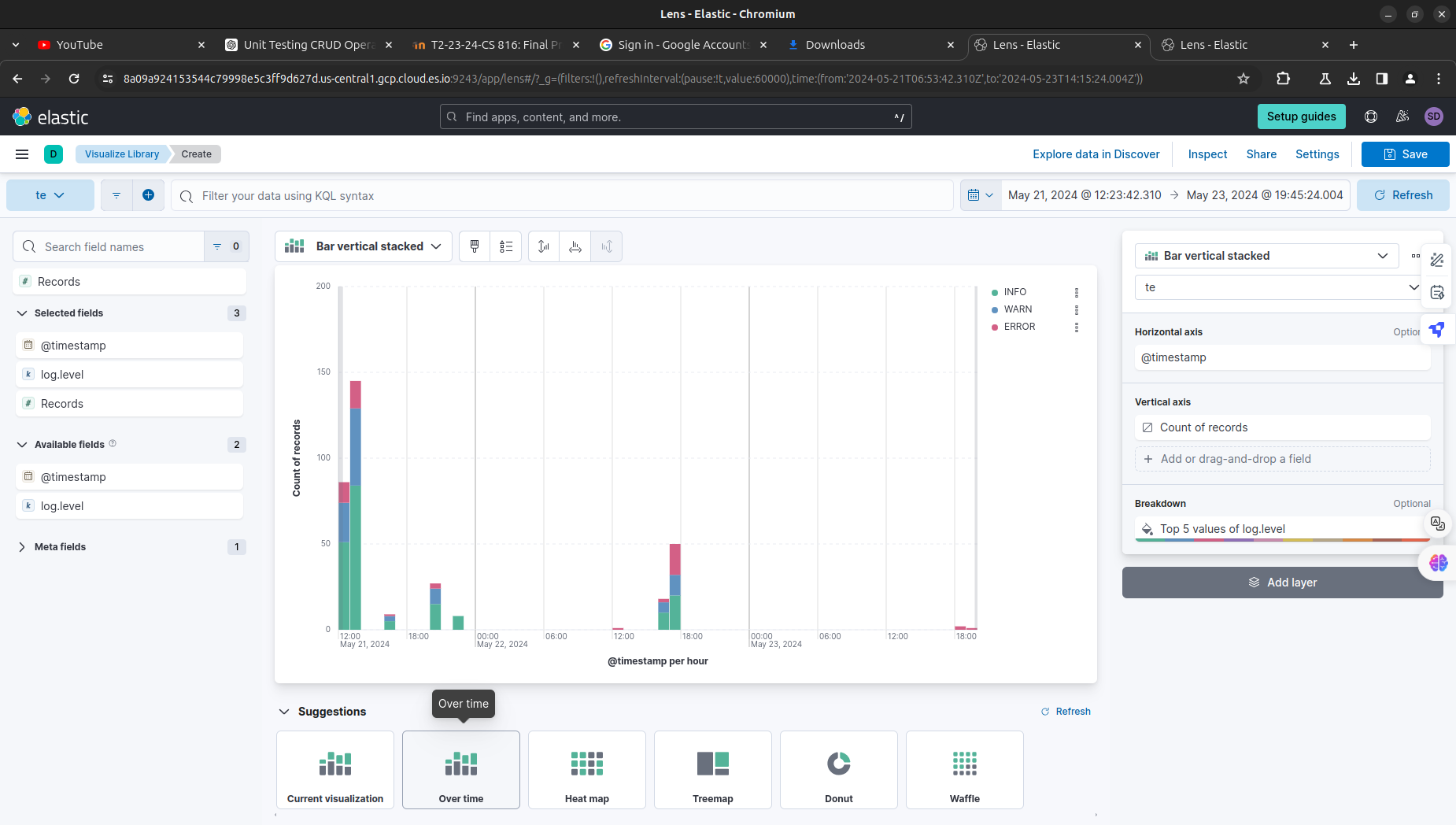


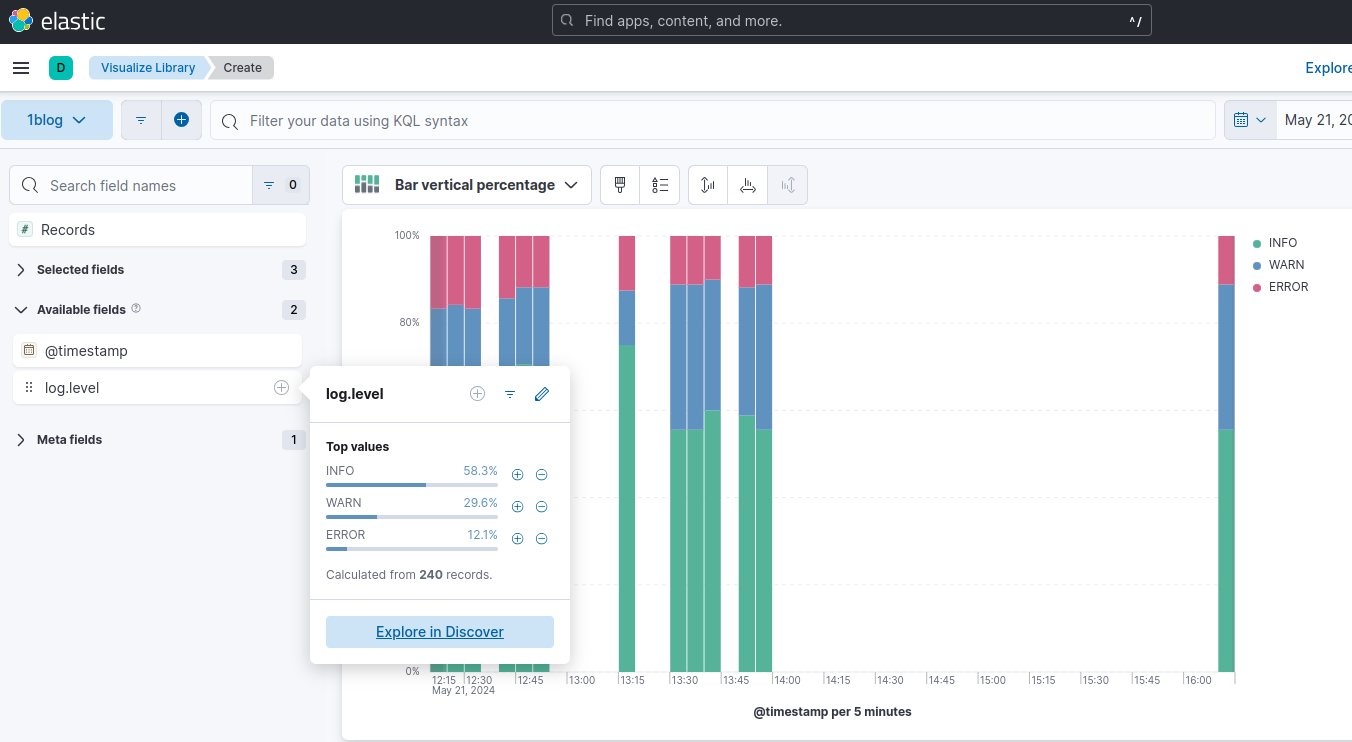


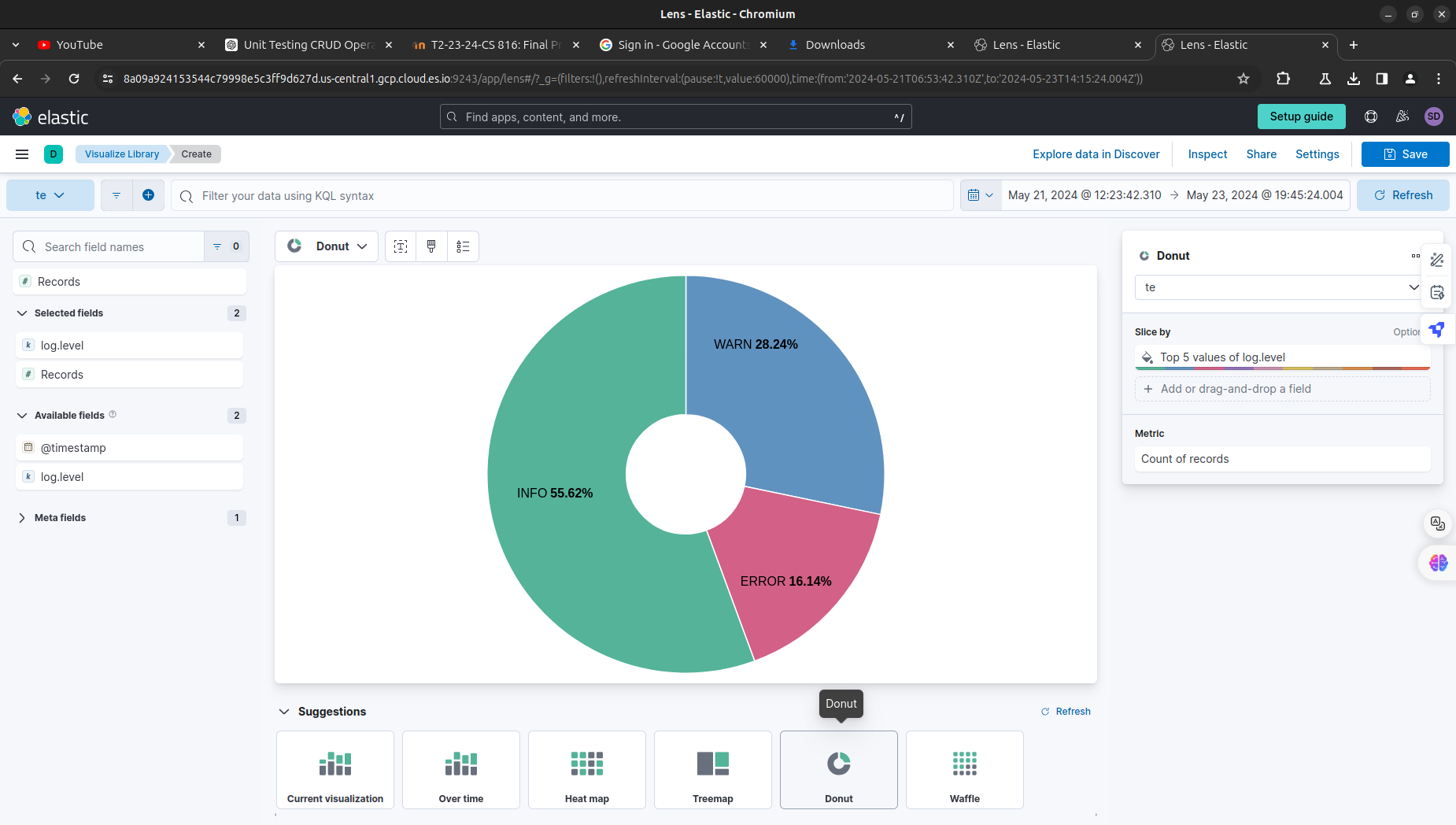
ELK stack consists of 3 applications: Elasticsearch, Logstash, and Kibana. ELK stack is useful for collecting logs from all applications, analyzing these logs, and creating visualizations.

Each log has a Timestamp, deployment type (info/ error), and message.









##### References

* Class Slides
* [stackoverﬂow.com](http://stackoverflow.com/) for debugging various errors and other warnings
* <https://tailwindcss.com/docs/>for utility classes
* [https://stackoverﬂow.com/a/11104156/15069364](https://stackoverflow.com/a/11104156/15069364) For process.env error
* Jacob Bhaiya’s “Chapter 3 Simple Calculator Program.pdf” for Jenkinsﬁle
* <https://docs.docker.com/get-started/02_our_app/>
* <https://docs.docker.com/get-started/docker_cheatsheet.pdf>

## Errors

1. Initially, i had no idea about proxy, later we added an environment variable to api calls. This is not an error but we changed the api calls on every page.
2. Communication between front-end and back-end images after creating docker images. Later i got to know about the network and communication between them.

##### Future Scope

* Replicate the same for renting any other objects like cars or any other things.
* Add functionality for the user to edit and delete the posts.
* Instead of giving an image URL for the post , an image upload button can be added.