CS 816 - Software Production Engineering

Major Project - Integrating Image Captioning with DevOps

Report by,
MT2023024 - Himarishitha Reddy Kakunuri,
Himarishitha.Kakunuri@iiitb.ac.in

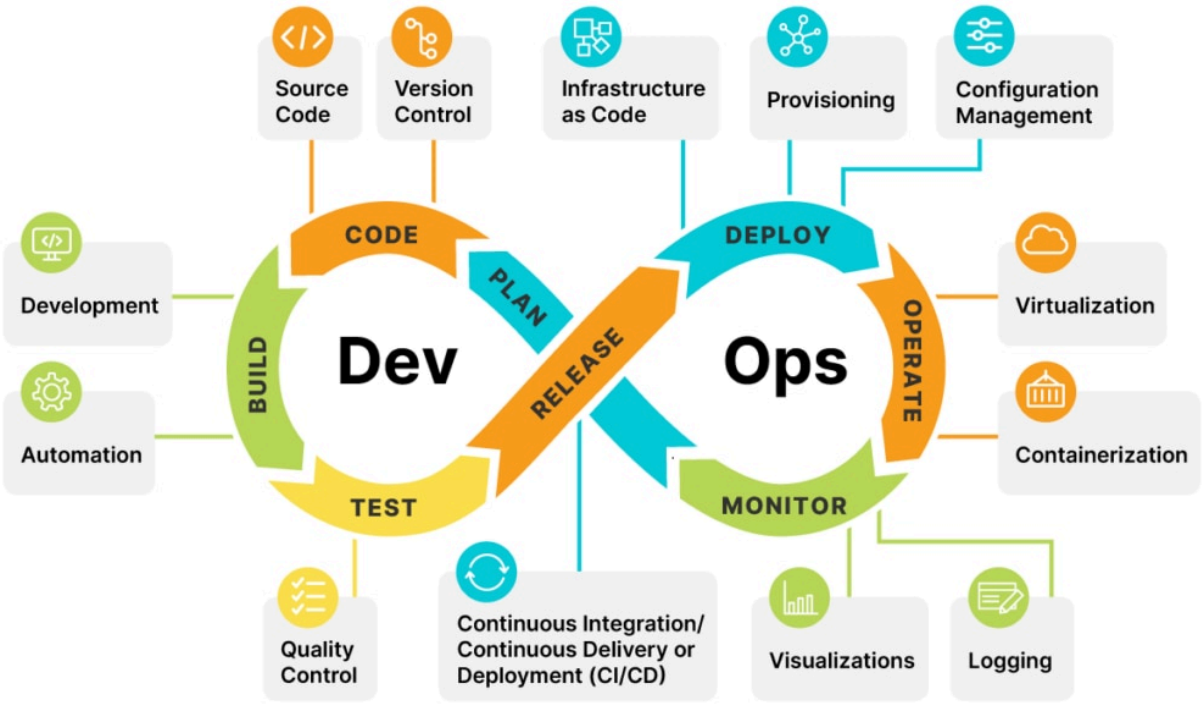
MT2023017 - Abhipsa Panda
Abhipsa.Panda@iiitb.ac.in

Introduction:

In this project, we aim to create and deploy an image captioning system by integrating a machine learning model into our application and using DevOps practices. Image captioning involves generating descriptive text for images, which can help with tasks like providing accessibility for visually impaired individuals, automating content creation, and improving search engine results.

We use pretrained model weights to make the image captioning process efficient. By applying DevOps practices, we ensure that the integration, deployment, and monitoring processes are smooth and effective. This approach allows us to build a system that is reliable and can handle the demands of modern applications, providing a great user experience.

Devops:



What is DevOps?

DevOps is a set of practices, principles that aim to enhance collaboration and communication between software development (Dev) and IT operations (Ops) teams. The primary goal of DevOps is to improve and automate the process of software delivery, making it more efficient, reliable, and scalable.

DevOps is a collaborative approach that spans the entire software development process, starting from planning and coding, all the way to testing, deployment, and monitoring. It aims to break down the traditional barriers between development and operations teams, promoting a culture of teamwork and shared responsibility. By incorporating automation, continuous integration, continuous delivery (CI/CD), and other best practices, DevOps enables organizations to deliver high-quality software more efficiently and at an accelerated pace.

Why DevOps?

- Faster Time-to-Market: DevOps enables rapid and continuous delivery of software, reducing the time it takes to bring new features or updates to end-users.
- Improved Collaboration: By breaking down the barriers between development and operations teams, DevOps fosters better communication and collaboration, leading to more effective problem-solving and innovation.
- Increased Efficiency: Automation of repetitive tasks streamlines the development and deployment processes, reducing errors and freeing up resources for more strategic tasks.
- Enhanced Quality: Continuous integration and continuous testing practices ensure that code changes are thoroughly tested, leading to higher-quality software with fewer defects.
- Scalability: DevOps practices make it easier to scale infrastructure and applications to meet growing demands, ensuring that systems can handle increased workloads.
- Cost Efficiency: Through automation and efficient use of resources, DevOps can contribute to cost savings in terms of both time and infrastructure.

In summary, DevOps is a response to the need for faster, more reliable, and collaborative software development and delivery. It brings together people, processes, and tools to create a culture of continuous improvement and innovation in the world of IT and software development.

Devops Practices:

1. **Continuous Integration (CI):**

   Continuous Integration (CI) is a development practice that automates the process of integrating and testing code changes, allowing teams to catch bugs early in the development cycle. Automated tests are executed as part of the CI process, ensuring the quality of the code. The key goal is promoting frequent and reliable deployments, and reducing the cost of fixing issues that may arise later in the development process.

2. **Continuous Delivery (CD):**

   Continuous Delivery (CD) extends the principles of CI by automating the code delivery process to various environments, facilitating consistent and repeatable releases. It involves building, testing, and deploying code changes to multiple environments, creating deployable artifacts.

3. **Continuous Deployment (CD):**

   Continuous Deployment takes the automation one step further by releasing every code change directly to the production environment. This approach aims for rapid release cycles, minimal downtime, and an entirely automated release process. Continuous Deployment is suitable for environments where frequent and automated releases are a priority, promoting agility and responsiveness to user needs.

4. **Version Control:**

   Version Control is a fundamental DevOps practice that manages code revisions, tracks changes, and facilitates collaboration among developers. Utilizing version control systems like Git, it provides a structured approach to merging code changes, handling conflicts, and rolling back changes to earlier states.

5. **Agile Software Development:**

   Agile Software Development is an approach emphasizing collaboration, customer feedback, and adaptability through short release cycles. Agile frameworks like

Kanban and Scrum facilitate continuous changes based on user feedback. It contrasts with traditional models like waterfall, offering flexibility and responsiveness to evolving customer needs throughout the development process.

6. **Infrastructure as Code (IaC):**

   Infrastructure as Code involves managing and provisioning infrastructure using machine-readable scripts. Tools like Terraform and Ansible enable the automated and consistent creation of infrastructure, reducing manual errors, and promoting scalability.

7. **Configuration Management:**

   Configuration Management focuses on managing the state of resources, including servers and databases, in a systematic way. Through configuration management tools, teams can roll out changes systematically, track system states, and avoid configuration drift, where a system's configuration deviates over time from the desired state.

8. **Continuous Monitoring:**

   Continuous Monitoring is like keeping a close eye on your entire application to make sure it's working well. It checks things in real-time, collecting detailed information about how the application is doing. If any issue arises, it sends out alerts so the team can take quick action. This helps the team fix issues right away and plan for making the application even better in the future.

9. **Planning:**

   In the planning phase, DevOps teams identify,define, and describe features and capabilities for applications and systems. They track task progress at different levels, from single products to multiple product portfolios. Planning is crucial for setting the direction of development efforts, aligning with business goals, and ensuring a systematic approach to project execution.

10. **Development:**

    The development phase encompasses all the aspects of the software code development, including coding, testing, and integration. DevOps teams select

development environments, collaborate through version control like Git, and automate steps.

11. **Delivery:**

    Delivery involves deploying applications consistently into production environments, ideally through continuous delivery (CD). Technologies like infrastructure as code, containers, and microservices contribute to delivering fully governed infrastructure environments.

12. **Operations:**

    The operations phase focuses on maintaining, monitoring, and troubleshooting applications in production environments. DevOps teams prioritize system reliability, high availability, strong security, and zero downtime. Automated delivery and secure deployment practices help swiftly identify and resolve issues, ensuring a constant focus on the operational aspects of the application.

Devops Tools:

DevOps involves a wide array of tools that facilitate collaboration, automation, and efficiency across the software development and IT operations lifecycle. Here are some DevOps tools used:

1. **Maven:**

   Maven is an essential build automation and project management tool primarily used in Java development. It simplifies the building process by managing project dependencies, compiling source code, running tests, and packaging artifacts. Maven employs a Project Object Model (POM) file, which acts as a configuration file specifying project details.

2. **Git:**

   Git is a distributed version control system in collaborative software development. It enables multiple developers to work on a project simultaneously, tracking changes, and maintaining version history. Git operates locally, allowing

developers to commit changes, create branches, and merge code seamlessly. It's branching model supports parallel development, enabling teams to work on features independently.

3. **Jenkins:**

   Jenkins is a open-source automation server crucial for implementing continuous integration and continuous delivery (CI/CD) pipelines. It automates repetitive tasks associated with building, testing, and deploying code, allowing developers to focus on coding. Jenkins integrates seamlessly with version control systems like Git, triggering builds and tests upon code changes. With an extensive ecosystem of plugins, Jenkins supports integration with various tools and services.

4. **Docker:**

   Docker has revolutionized containerization, offering a platform for developing, shipping, and running applications consistently across different environments. Containers, encapsulating applications and dependencies, enhance scalability and resource efficiency. Docker's simplicity lies in Dockerfiles, enabling developers to define application environments.

5. **Docker Compose:**

   Docker Compose is a tool for defining and running multi-container Docker applications. It allows developers to configure application services in a YAML file, enabling them to create and start all the services with a single command. Docker Compose simplifies the management of multiple containers by providing a unified way to orchestrate, scale, and interconnect containers in a development or testing environment. It ensures consistency and reproducibility, making it easier to set up and maintain complex application stacks.

6. **Ansible:**

   Ansible is a open-source automation tool used for configuration management, application deployment, and task automation. It employs a declarative language, YAML, for defining playbooks that describe desired configurations and tasks on target systems. Ansible operates over SSH, ensuring secure communication without requiring agents on managed nodes.

7. **Kubernetes:**

   Kubernetes is an open-source container orchestration platform designed to automate deploying, scaling, and managing containerized applications. It groups containers into logical units for easy management and discovery. Kubernetes ensures high availability by distributing application instances across a cluster of servers and automatically restarting failed containers. It also supports rolling updates and rollbacks, allowing for seamless application updates.

8. **ELK Stack (Elasticsearch, Logstash, Kibana):**

   The ELK Stack is a powerful suite of tools for searching, analyzing, and visualizing log data. Elasticsearch is a distributed search and analytics engine, Logstash is a data processing pipeline that ingests, transforms, and ships data, and Kibana is a visualization tool for exploring and presenting data stored in Elasticsearch. Together, they provide a comprehensive solution for centralized logging and real-time analysis.

9. **Webhooks:**

   Webhooks play a crucial role in facilitating real-time communication and automation in DevOps workflows. A webhook is an HTTP callback that sends data from one application to another as soon as a specific event occurs. In the context of DevOps, webhooks are commonly used to trigger automated actions in response to events like code commits, issue updates, or build completions. For example, when code is pushed to a version control repository, a webhook can notify a CI/CD system to initiate an automated build and deployment process.

10. **Ngrok:**

    Ngrok is a valuable tool for creating secure tunnels to local servers, making them accessible over the internet. In DevOps scenarios, Ngrok is often used to expose locally hosted services or web applications during development and testing. By providing a public URL for a locally running server, Ngrok enables external systems or services to interact with the local environment. This is particularly useful when testing webhooks or integrating with third-party services that require external access.

Image Captioning ML Model:

The model that we are using for training has an encoder-decoder architecture which takes an image as input and generates text caption as output.

The basic model architecture can be described below:

1. The encoder used is Vision Transformer .
2. The Decoder used is GPT2.
3. The model trained on Flickr 8k dataset.
4. The hugging face Seq2SeqTrainer from transformers module is used for finetuning the model.
5. The PyTorch module is used for data-processing.

We are saving the trained configurations and weights of the model in HuggingFace for further use.As these files are very large, we can't directly push it into HuggingFace. So, we have used GitLFS for pushing them to Hugging Face.

Use the following commands can be used to setup hugging face

1. pip install huggingface_hub
2. huggingface-cli login

Enter your hugging face credentials to login.

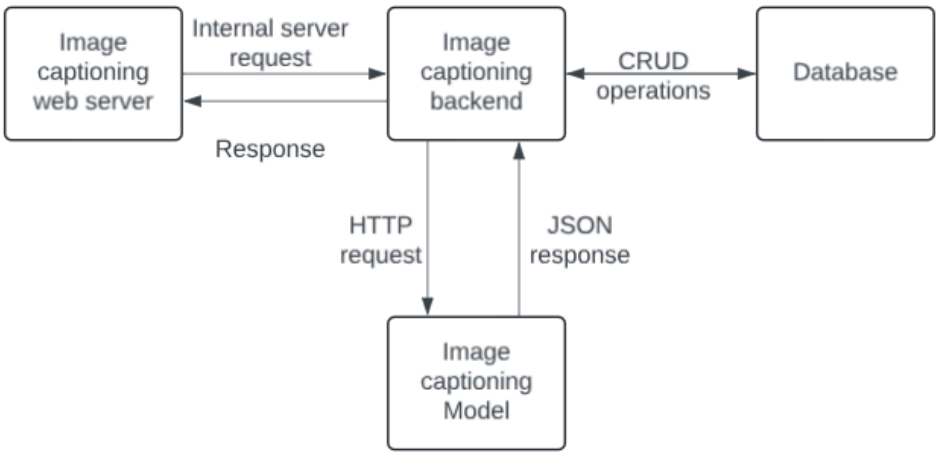
Use the following commands to set up git lfs

1. sudo apt update
2. sudo apt install git-lfs
3. git lfs install
4. git lfs track "*.pt"

Use the following steps to push the model into hugging face hub :-

1. git clone https://huggingface.co/username/repo-name
2. cd repo-name
3. cp /path/to/your/model/files/* .
4. git add .
5. git commit -m "Add model files"
6. git push origin main

Architecture:



This is the high level overview of our application architecture. Our application has 4 services which are image captioning backend (using Java Spring Boot), image captioning frontend (using React JS), image captioning Model (using pytorch,transformers and hosted using Django framework) , database(MySQL).

Application workflow:

1. If not registered, the user registers into the system by entering the required details.
2. After a successful registration, the user logs in.
3. Then encounters a brief dashboard which has the option to view the past requests and another option to make a new request.
4. Once clicked on to generate a new caption,a new page will appear to enable uploading the image and once clicked on the predict button, the caption will be generated.
5. Once clicked on past captions, users can view the past image requests made along with their generated captions.
6. After the user completes his/her work, they can log out.

Workflow:

1. **IntelliJ IDEA with Maven:**

   The project commenced with the utilization of IntelliJ IDEA for development and Maven for project management. IntelliJ served as the integrated development environment, while Maven facilitated the handling of project dependencies and build processes through the configuration specified in the pom.xml file.

2. **Model Integration:**

   The machine learning model was integrated into the backend of the application, using pretrained weights obtained from trained model which is deployed in Hugging Face.This involved ensuring that the model could be accessed and utilized by the application, facilitating image captioning task.

3. **GitHub Version Control:**

   For version control, the project was hosted on GitHub. This involved creating a repository to manage source code, enabling collaborative work. Git workflows were established to effectively manage branches, commits, and code collaboration within the GitHub environment.

4. **Jenkins Continuous Integration:**

   Continuous integration was implemented through Jenkins, an automation server. Jenkins was configured to retrieve source code from the GitHub repository. Maven, integrated into Jenkins, managed the build process, automating tasks such as compilation, testing, and packaging. The CI pipeline was optimized with automatic triggers for efficient code integration.

5. **Docker Containerization:**

   Containerization was achieved using Docker. A Dockerfile was crafted within the project, defining the specifications for the Docker image. Jenkins was then configured to automatically build the Docker image and push it to a container registry, such as Docker Hub. Docker Compose was used to define and manage multi-container Docker applications, including the model, backend, and frontend, ensuring consistent and reproducible environments for development.

6. **Ansible Automation:**

   Automation of deployment processes was implemented using Ansible. Ansible playbooks were created to define the steps for deployment and configuration. Integration with Jenkins allowed for the automation of end-to-end deployment processes, ensuring consistency across various deployment environments.

Building Code and Testing:

Code Development:

Backend:

1. Install IntelliJ IDEA:
   a. Download and install the IntelliJ IDEA IDE from the official website (https://www.jetbrains.com/idea/download/).
   b. Follow the installation instructions based on your operating system.
2. Open IntelliJ IDEA:
   a. Launch IntelliJ IDEA after the installation is complete.
3. Create a New Project:
   a. Click on "Create New Project" on the welcome screen.
   b. Select "Maven" on the left side and click "Next."
4. Configure Project:
   a. Enter the GroupId and ArtifactId. These are identifiers for your project, similar to a namespace and project name.
   b. Choose a project location and click "Next."
5. Set up Maven:
   a. If IntelliJ IDEA detects that Maven is not configured, it will prompt you to configure it. Confirm the Maven configuration.
   b. Alternatively, you can configure Maven manually by going to "File" > "Project Structure" > "Project" > "Project SDK" (select your Java SDK) and "Project" > "Project" > "Project Language Level" (set to the appropriate Java version).
   c. Click "Finish" to create the project. IntelliJ IDEA will generate the project structure and download the required Maven dependencies.
6. Explore Project Structure:

    a. Once the project is created, you'll see the project structure in the Project Explorer on the left side.
    b. The src folder contains the main and test Java source files.
    c. The pom.xml file is the Maven Project Object Model, where you can define project configurations and dependencies.
7. Write Code,Test Cases:
    a. Open the Java file under `src/main/java` and start writing your code.
    b. Open the Java file under `src/main/test` and start writing testcases.
8. Run Maven Build:
    a. To build your project, right-click on the `pom.xml` file and select "Run Maven" > "Clean" and then "Run Maven" > "Install."
    b. This will compile your code, run tests, and package your application.

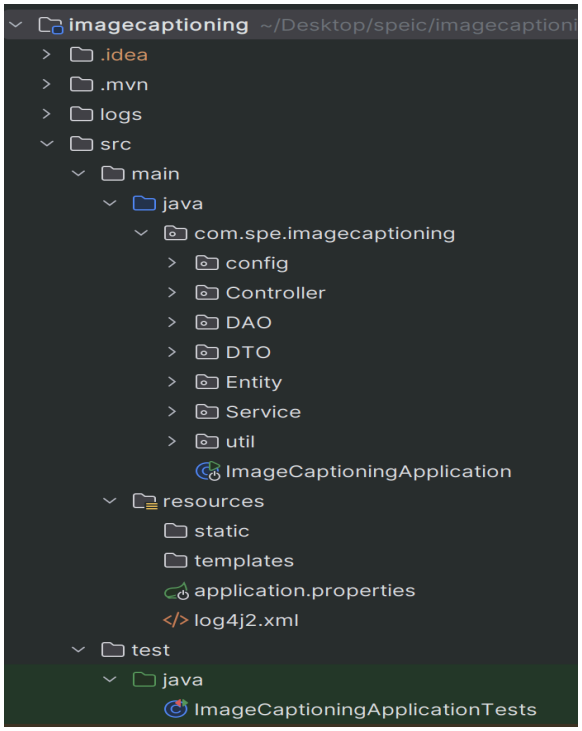
The project layout, as displayed in the image, follows a standard convention. You can find the primary source code in the src/main/java folder. This is where the main logic of your application resides, organized within a package structure that aligns with your project's GroupId and ArtifactId.

Additionally,src/test/java folder is designated for test cases. In this directory, you can organize your test classes, ensuring a clear separation between production code and testing code.

And pom.xml (Project Object Model) file in a Maven project serves as a configuration file where you define project settings, dependencies, plugins, and various build parameters. Here we need to mention all our dependencies in the following format:

```
<dependency>

   <groupId>.....</groupId>

   <artifactId>.....</artifactId>

   <version>....</version>

</dependency>
```

Project Structure:



pom.xml:

We have included several key dependencies and configurations in the pom.xml file to ensure the project is fully functional and efficient. Here are the details:

- JUnit: Included for unit test cases, ensuring thorough testing of the application.
- Apache Log4j: Utilized for logging the results, providing robust logging capabilities.
- Spring Boot Starters: Added for data JPA, web development, security, and testing purposes.
- MySQL Connector: Integrated for runtime database connectivity.
- JSON Library: Included for handling JSON data.

These dependencies are explicitly mentioned in the pom.xml file. Additionally, the Java compiler configuration is set to version 17. Exclusions for default logging dependencies are specified to avoid conflicts and ensure proper logging setup with Log4j.

This setup ensures that the project can effectively utilize Log4j for logging purposes, JUnit for testing, and other Spring Boot functionalities for development and deployment. The specific versions mentioned in the pom.xml indicate the precise releases of these libraries on which the project relies.

Model Integration with backend:

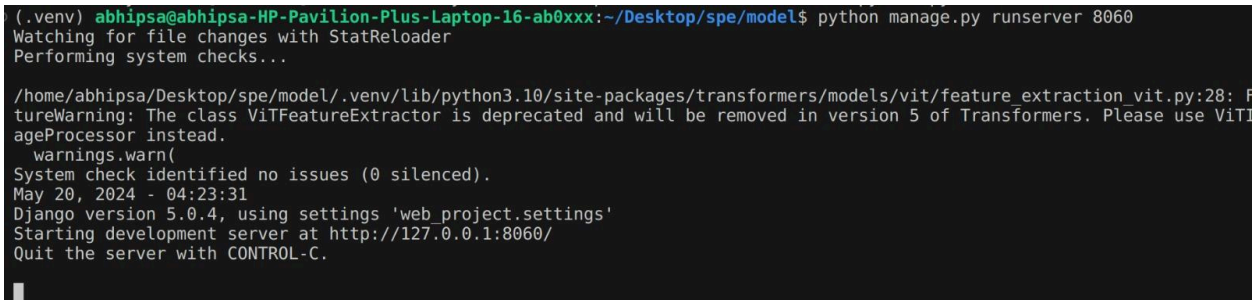
We are using Django to integrate our model to backend because it seamlessly integrates machine learning models into web applications, offering scalability, security, and easy request handling, making it an ideal framework for hosting ML models.

Setup Django in Visual Studio Code using the reference :
https://code.visualstudio.com/docs/python/tutorial-django

We are using Django view function to generate captions for images using a pre-trained model. The steps followed are:

1. Importing necessary libraries including PIL (Python Imaging Library) for image processing, transformers for natural language processing tasks, multiprocessing for parallel processing, and Django for web framework functionalities.
2. Defining a configuration class `Config` to hold various parameters like model names, batch sizes, learning rate, etc.
3. Initializing a ViT (Vision Transformer) feature extractor and tokenizer for text processing.
4. Defining a function `load_model()` to load a pre-trained image captioning model from the Hugging Face model hub using the `VisionEncoderDecoderModel`.
5. Defining a Django view function `generate_caption(request)` which accepts POST requests containing image data in base64 format. It decodes the image, processes it, generates a caption using the loaded model, and returns the caption as JSON response.
6. The `process_image(image)` function converts the image to a PyTorch tensor, generates a caption using the loaded model, splits the generated caption into sentences, trims it to contain only the first three sentences, and returns the trimmed caption.

We can also check our model by running command python manage.py runserver 8060(port no):



We are using Java configuration class to establish a standardized method for configuring and managing RestTemplate instances in a Spring application. By annotating a method with @Bean, it instructs Spring to create and manage a RestTemplate bean. This bean is a reusable object for making HTTP requests, facilitating communication between different parts of the application and external services. The RestTemplate bean can be injected into other components, allowing for consistent and centralized HTTP client configuration. Overall, this class promotes modularity, maintainability, and abstraction by encapsulating HTTP client configuration within the Spring IoC container.

We can call the model from the backend service by sending a HTTP POST request to a model service URL, passing image data as a JSON payload and receiving a JSON response containing the generated caption.

Test Cases:

The JUnit test case is the set of code that ensures whether our program code works as expected or not. In Java, there are two types of unit testing possible, Manual testing and Automated testing. Manual testing is a special type of testing in which the test cases are executed without using any tool. This is how you can write test cases:

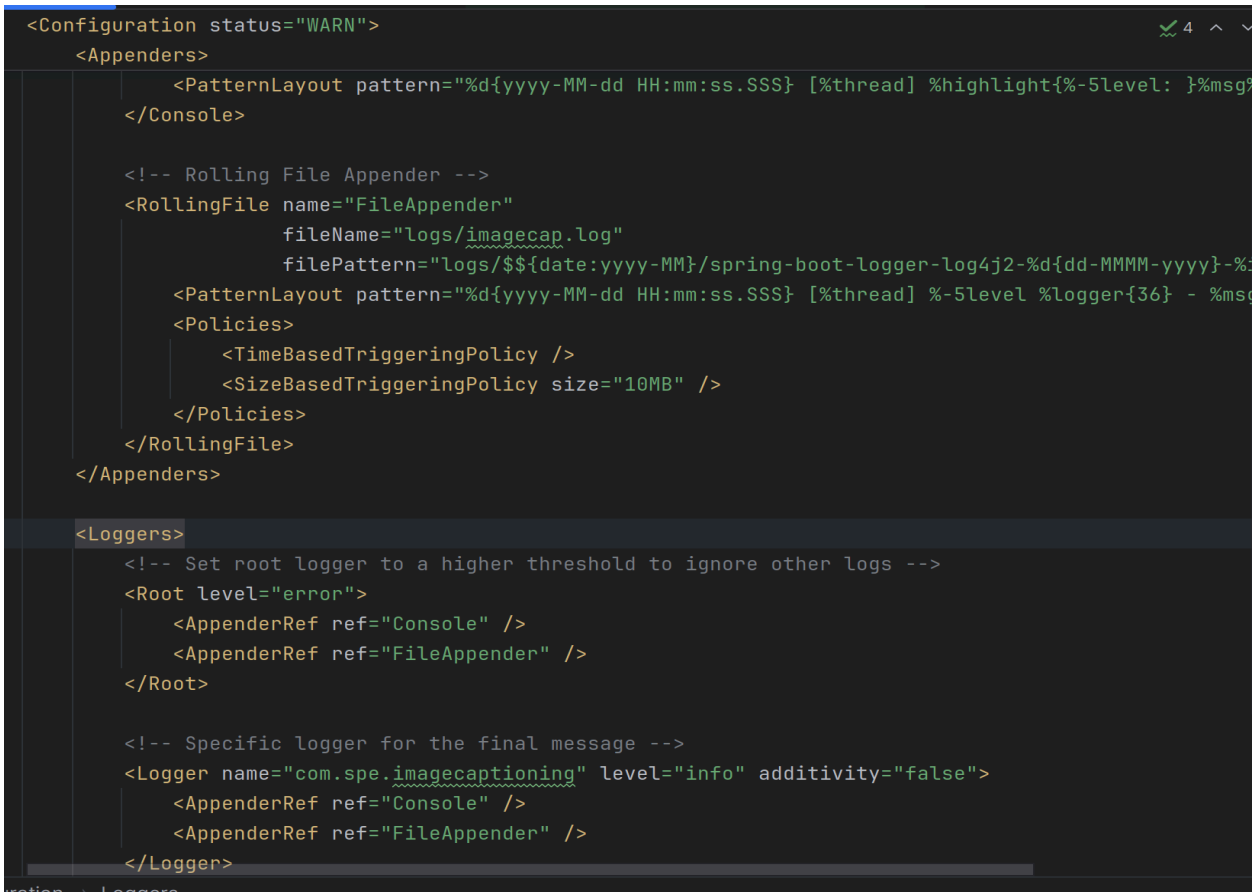
```java
import ...

@SpringBootTest(classes = ImageCaptioningApplication.class)  rishithaiiitb
public class ImageCaptioningApplicationTests {

    @Autowired
    UserService userService;

    @Test  rishithaiiitb
    void contextLoads() {
    }
    @BeforeAll  rishithaiiitb
    static void testInit() { System.out.println(" TESTING STARTED"); }

    @Test  rishithaiiitb
    public void testNoUser() {
        LoginDTO loginDTO = new LoginDTO();
        loginDTO.setEmail("user1@gmail.com");
        loginDTO.setPassword("Test123");
        assertFalse(userService.verifyUser(loginDTO),  message: "Expected no user found");
    }

    @Test  rishithaiiitb
    public void testNegativeLoginDetails() {
        LoginDTO loginDTO = new LoginDTO();
        loginDTO.setEmail("rishitha1201@gmail.com");
        loginDTO.setPassword("123");
        assertFalse(userService.verifyUser(loginDTO),  message: "Expected login to fail");
    }
```

Logs:

To configure logging in a Spring Boot application using Log4j2, follow these steps:

- Navigate to the root directory of your project and then to the src/main/resources directory.
    - Create a new file named log4j2.xml in this directory.
- Add the necessary configuration to this file to define the loggers and appenders.
    - The configuration includes a console appender to output log messages to the system console and a rolling file appender to write log messages to a file named imagecap.log in the logs directory.

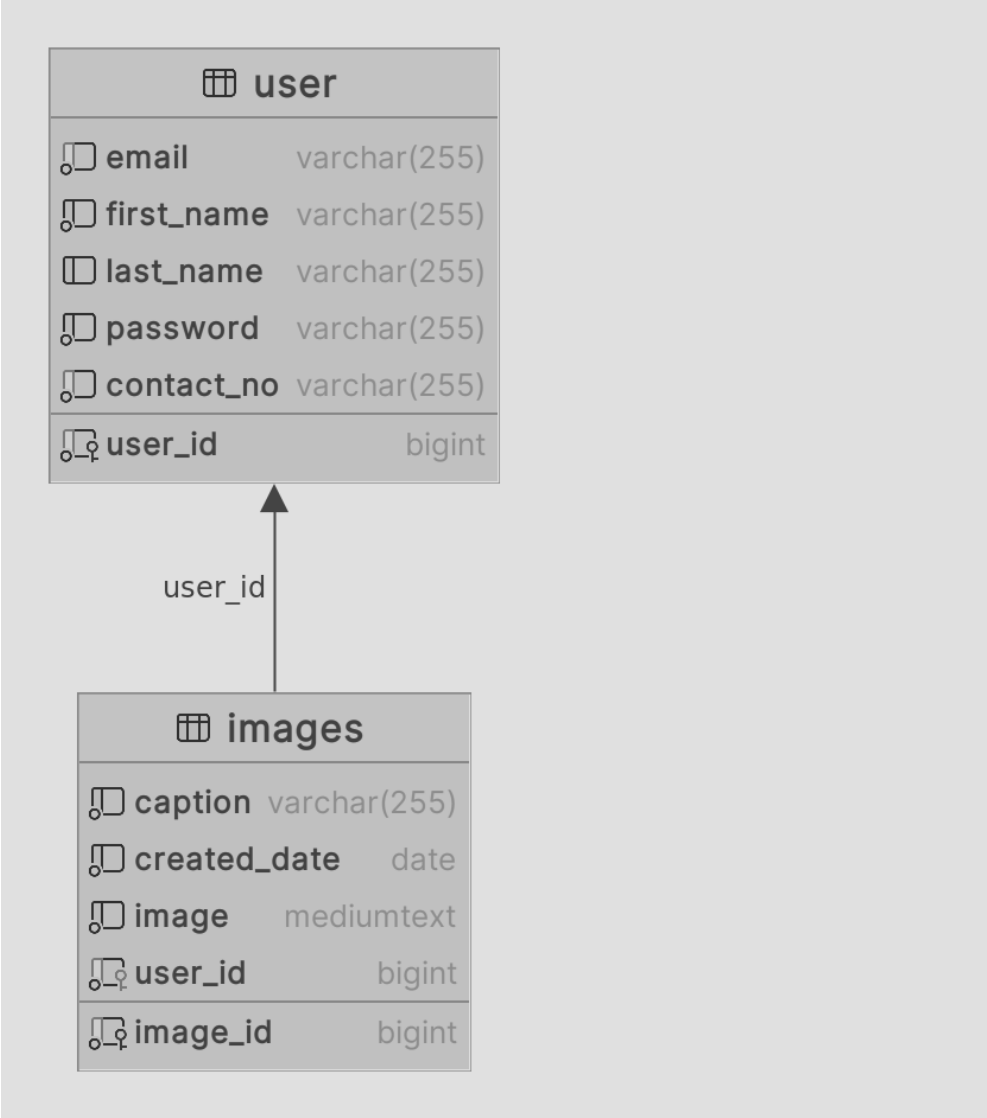
The file appender uses policies to roll over logs based on time and size constraints, specifically creating a new log file daily or when the file size exceeds 10MB. The root logger is set to capture all log messages at the "error" level or higher, while a specific logger for the com.spe.imagecaptioning package is set to capture "info" level messages or higher. The log4j2.xml file looks like this:



```
<Configuration status="WARN">
    <Appenders>
            <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %highlight{%-5level: }%msg
        </Console>

        <!-- Rolling File Appender -->
        <RollingFile name="FileAppender"
                     fileName="logs/imagecap.log"
                     filePattern="logs/$${date:yyyy-MM}/spring-boot-logger-log4j2-%d{dd-MMMM-yyyy}-%
            <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg
            <Policies>
                <TimeBasedTriggeringPolicy />
                <SizeBasedTriggeringPolicy size="10MB" />
            </Policies>
        </RollingFile>
    </Appenders>

    <Loggers>
        <!-- Set root logger to a higher threshold to ignore other logs -->
        <Root level="error">
            <AppenderRef ref="Console" />
            <AppenderRef ref="FileAppender" />
        </Root>

        <!-- Specific logger for the final message -->
        <Logger name="com.spe.imagecaptioning" level="info" additivity="false">
            <AppenderRef ref="Console" />
            <AppenderRef ref="FileAppender" />
        </Logger>
```

Database Design:

In our project, we have developed two key entities: User and Image. The User entity is responsible for storing detailed information about each user, including their credentials and profile data. The Image entity is designed to store information related to images, specifically the images uploaded by users and the captions generated for these images. This structure allows us to efficiently manage and link user data with their corresponding images and captions, ensuring a streamlined and organized data flow within the application.



APIs:

1. User Login

Endpoint: POST /user/login

Description: Authenticates a user based on the provided login credentials.

Input:

- Request Body: An object with the following fields:
    - email: The user's email address.
    - password: The user's password.

Output: A boolean value (true if the login is successful, false otherwise).

2. User Registration

Endpoint: POST /user/register

Description: Registers a new user with the provided details.

Input:

- Request Body: An object representing a User with the following fields:
    - email: The user's email address.
    - password: The user's password.
    - name: The user's name.
    - contact: The user’s contact.

Output: An object representing the newly registered user.

3. Get Past Image Captions

Endpoint: GET /user/getPastImageCaptions/{email}

Description: Retrieves the past images and their captions for a given user email.

Input:

- Path Variable: email - The user's email address.

Output: An array of objects representing images with their captions.

4. Generate Caption

Endpoint: POST /user/generateCaption/{email}

Description: Generates a caption for a given image and associates it with the specified user email.

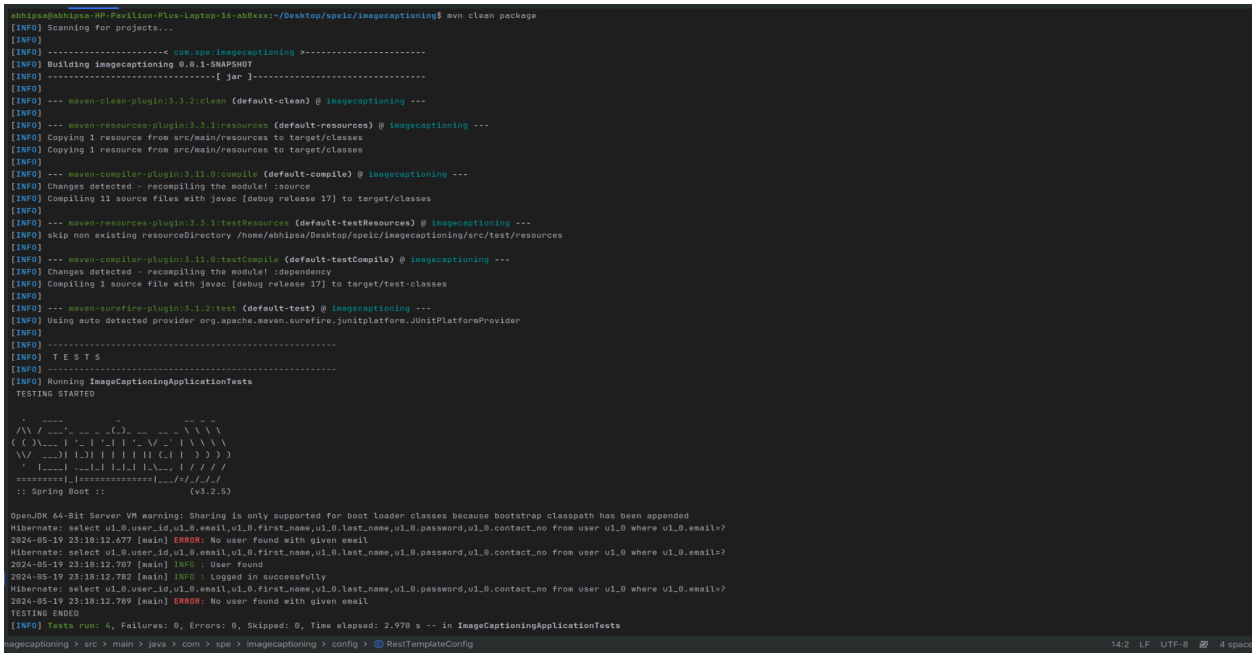
Input:

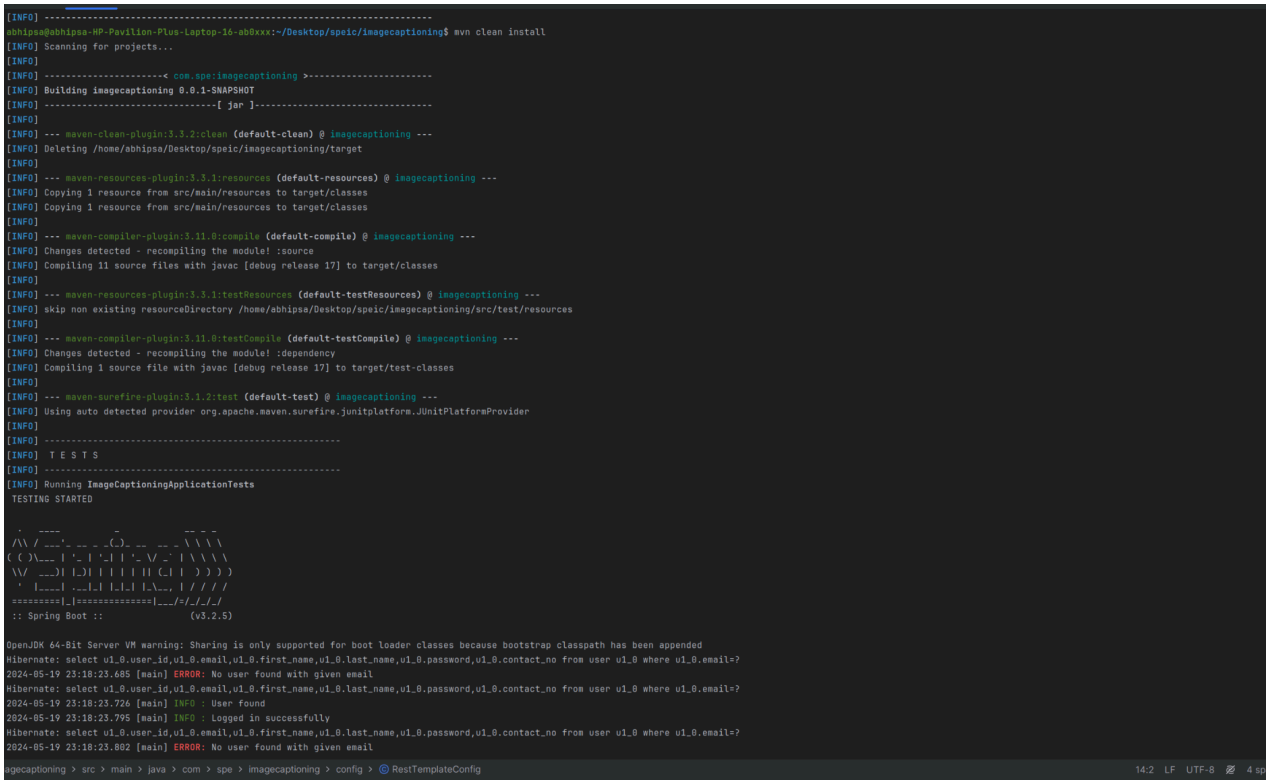
- Path Variable: email - The user's email address.
- Request Body: An object representing an Image with the following field:
    - image: Blob data of the image.

Output: A string representing the generated caption for the image.

Creating JAR File:

- Navigate to Project Directory:
    - Open the command prompt.
    - Change the current directory to your Maven Java project's root directory.
- Clean the Project:
    - Run mvn clean package. This cleans the project by deleting the target directory, which contains the output of previous builds.
- Package the Application:
    - Execute mvn install. This command compiles your source code, runs tests, and packages the application into a JAR file. The resulting JAR file is placed in the target directory.

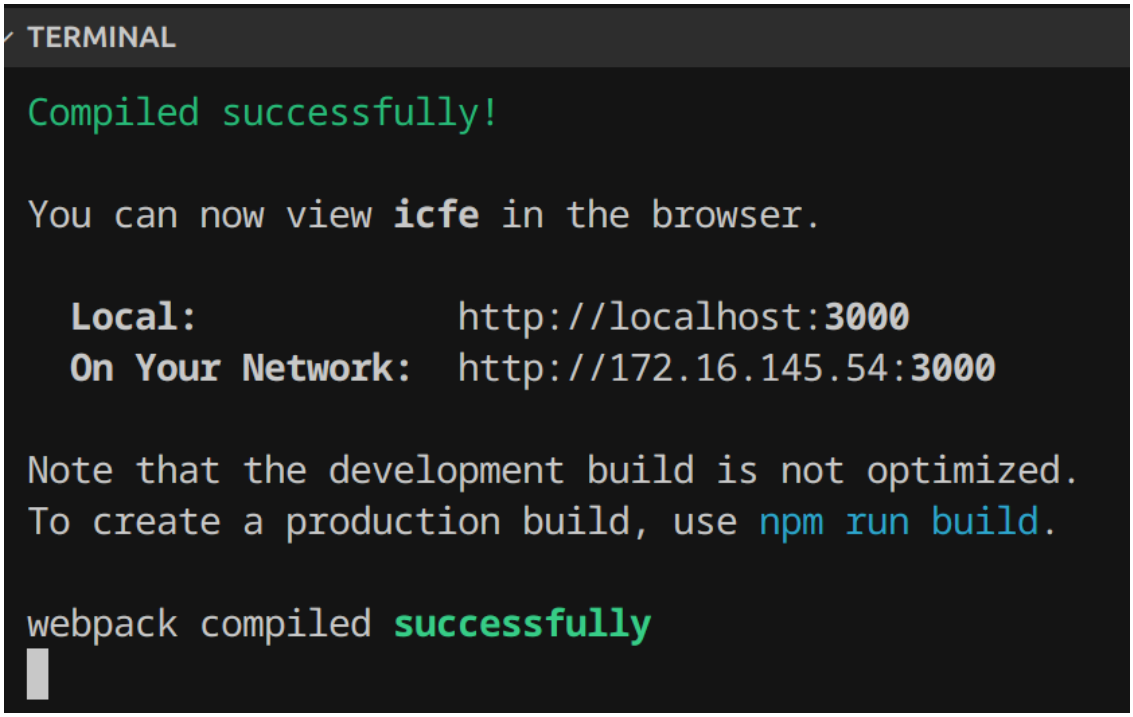




Frontend:

1. Install Node.js and npm:
    a. Download and install the latest version of Node.js from the official Node.js website(https://nodejs.org/en).
    b. Follow the installation instructions for your operating system.
    c. This will also install npm (Node Package Manager), which is required for managing dependencies.
2. Create a New React Application:
    a. Open your command prompt or terminal.
    b. Navigate to the directory where you want to create your React project using the cd command.
    c. Run npx create-react-app my-app to create a new React application (replace my-app with your desired project name).
    d. Change to the newly created project directory with cd my-app.
3. Install Necessary Dependencies:
    a. Install additional libraries using npm. For example, to install React Router for routing, run npm install react-router-dom.
4. Create and Organize Components:

    a. Inside the src folder, create a components directory to organize your React components.
    b. Create individual component files (e.g., Header.js, Footer.js, ImageUpload.js) within the components directory.
5. Set Up Routing:
    a. Set up routing to navigate between different pages in your application.
    b. Update your main application file to include routes.
6. Connect to Backend API:
    a. Use axios or the Fetch API to communicate with your backend.
    b. Install axios using npm: npm install axios.
7. Run the Application:
    a. Start the React development server by running npm start in your project directory.
    b. Open your web browser and navigate to http://localhost:3000 to see your React application in action.



Project Structure

src Folder:

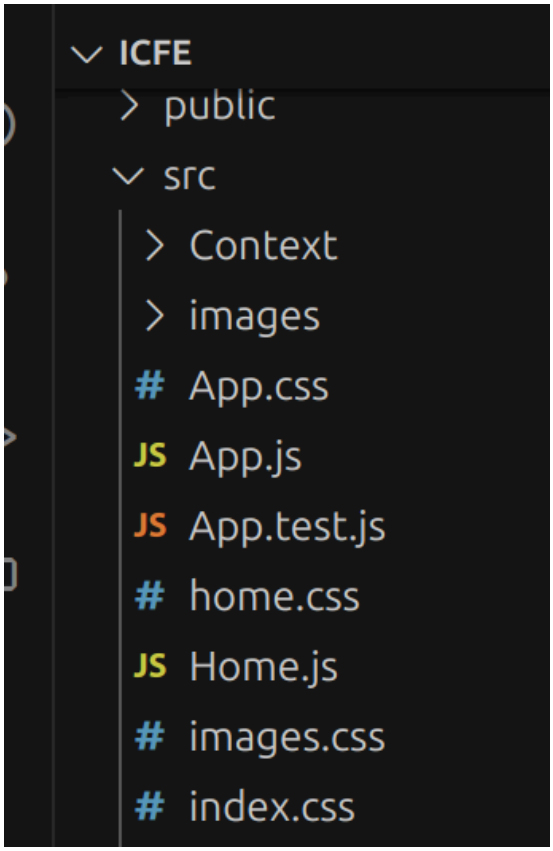
- Contains the main source files for your React application.
- Organized into subdirectories like components for reusable UI components.

public Folder:

- Contains static files such as index.html.
- Assets placed here are accessible directly via URL paths.

package.json:

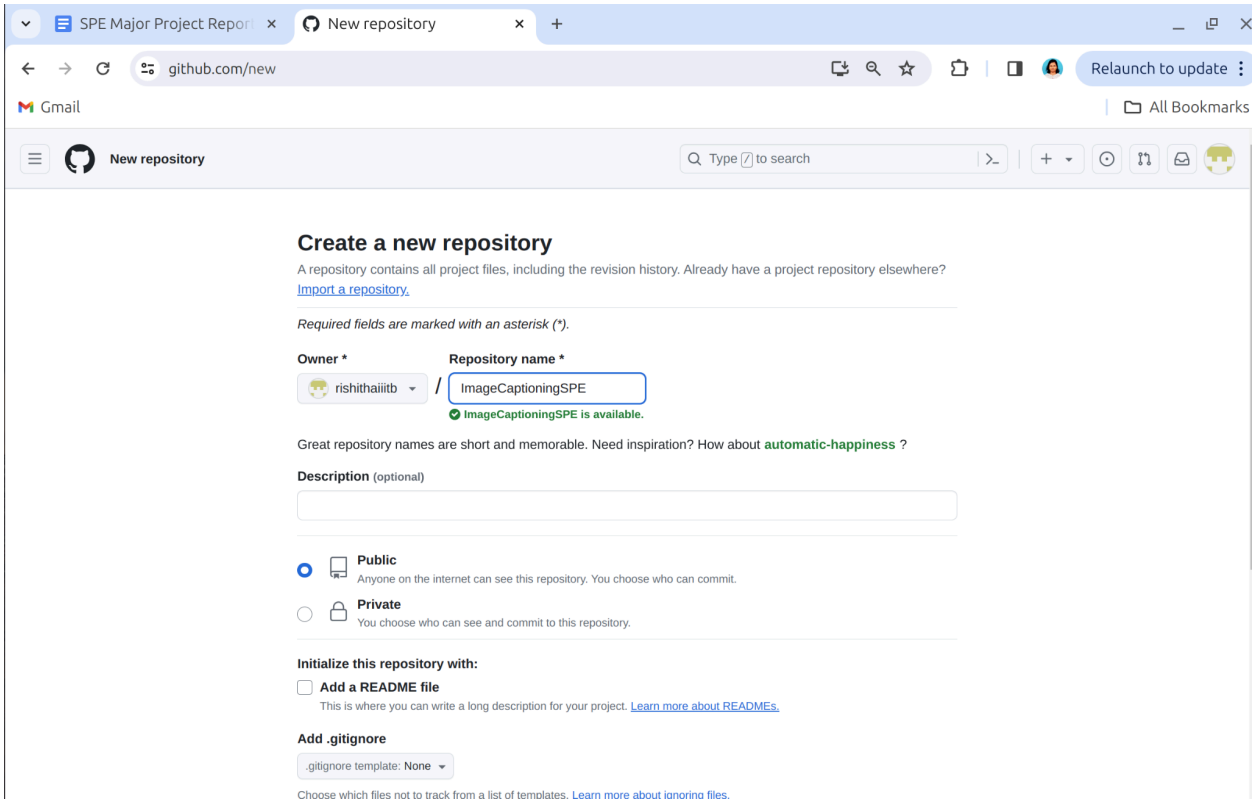
- Lists project dependencies and scripts.
- Defines configurations for building and running the application.



Github Version Control:

We should follow the steps mentioned below to set up version control for your project using GitHub:

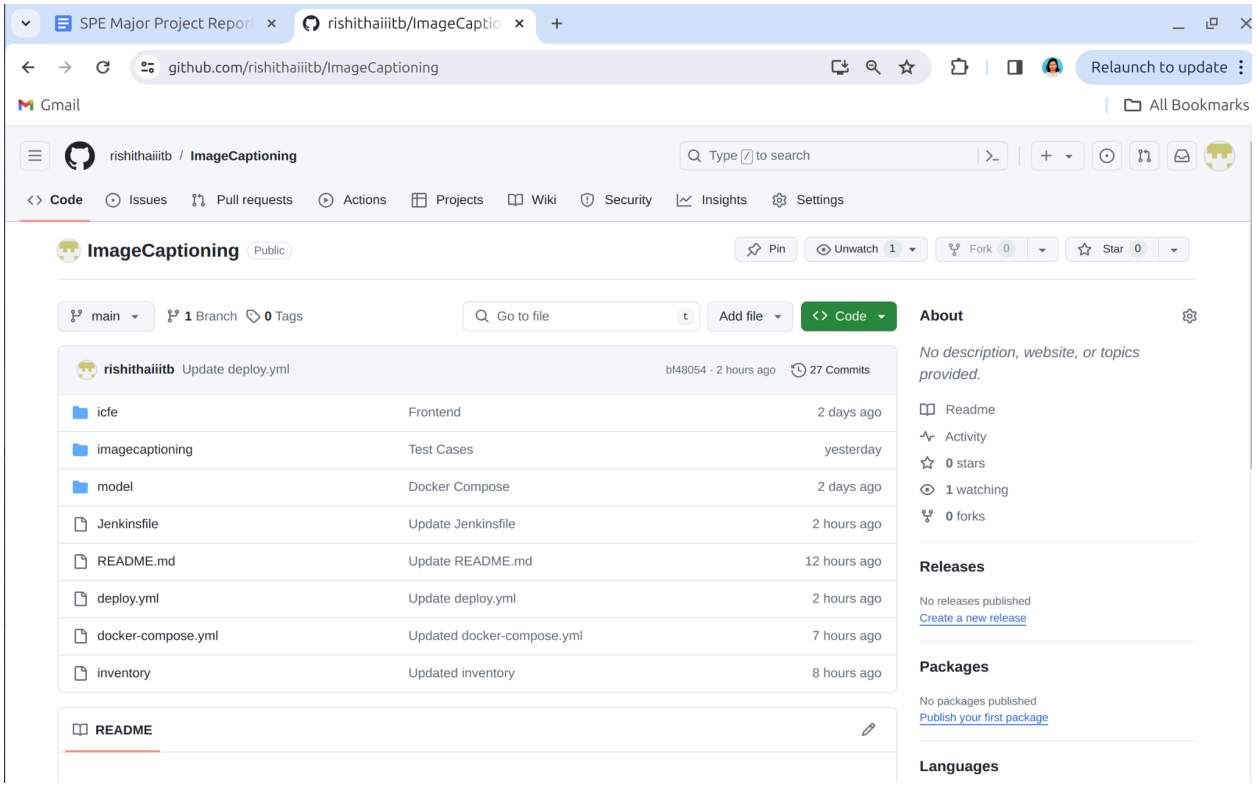
1. Create a New Repository:
    a. Once logged in, click on the "+" sign in the top right corner and select "New repository."
    b. Provide a name for your repository, a short description, choose public or private visibility, and initialize with a README if needed.
    c. Click "Create repository."

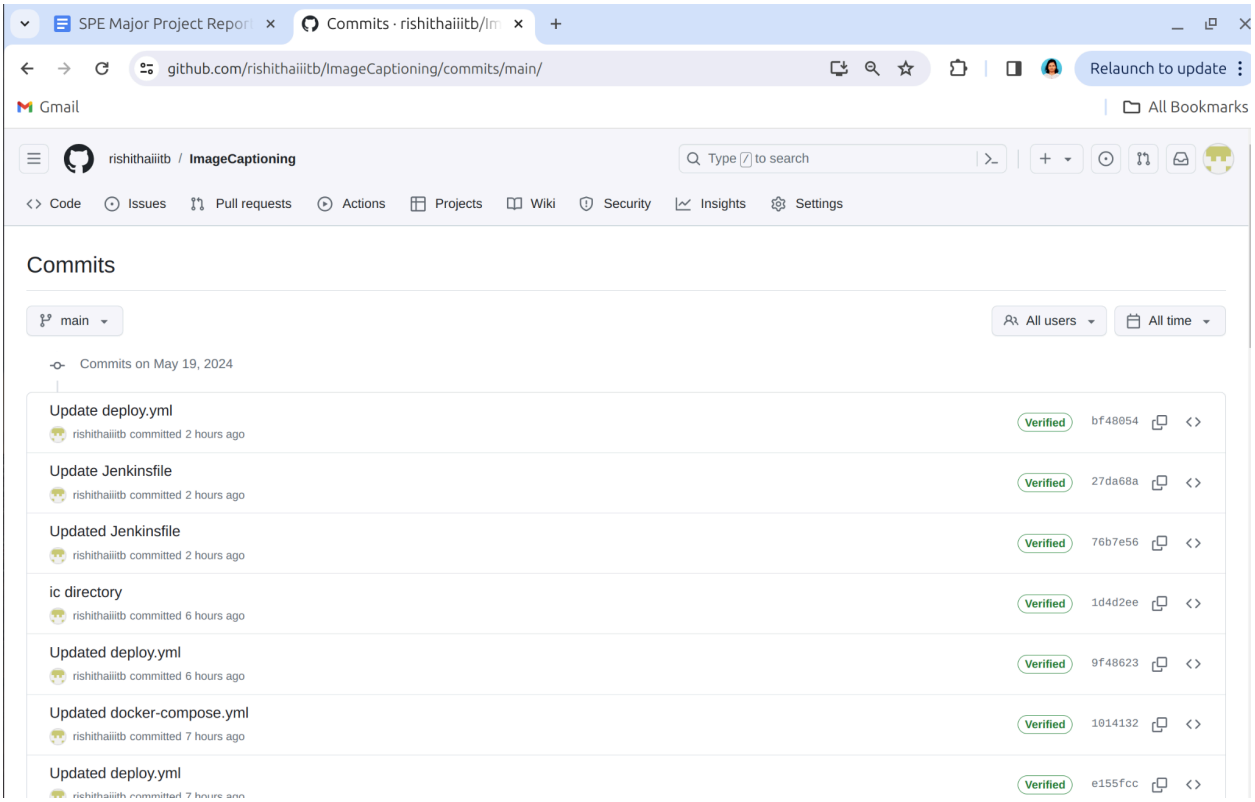


2. Clone the Repository:
    a. Copy the repository URL from the "Code" dropdown.
    b. Open a terminal on your local machine.
    c. Navigate to the directory where you want to store your project.
    d. Run the necessary command to clone the repository.
3. Add and Commit Changes:
    a. Navigate into the cloned repository.

b. Create or copy your project files into this directory.
c. Stage and commit your changes using the version control system.

```
git init
git add .
git commit -m "Initial Commit"
git branch -M main
git remote add origin https://github.com/rishithaiiitb/ImageCaptioning.git
git push -u origin main
```





Jenkins Pipeline:

Installing Jenkins:

Install Java :
sudo apt-get update
sudo apt install -y openjdk-11-jdk

To check: java - - version

Install Jenkins :
Add the Jenkins repository key:
wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add
Add the Jenkins repository to your system:
sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'

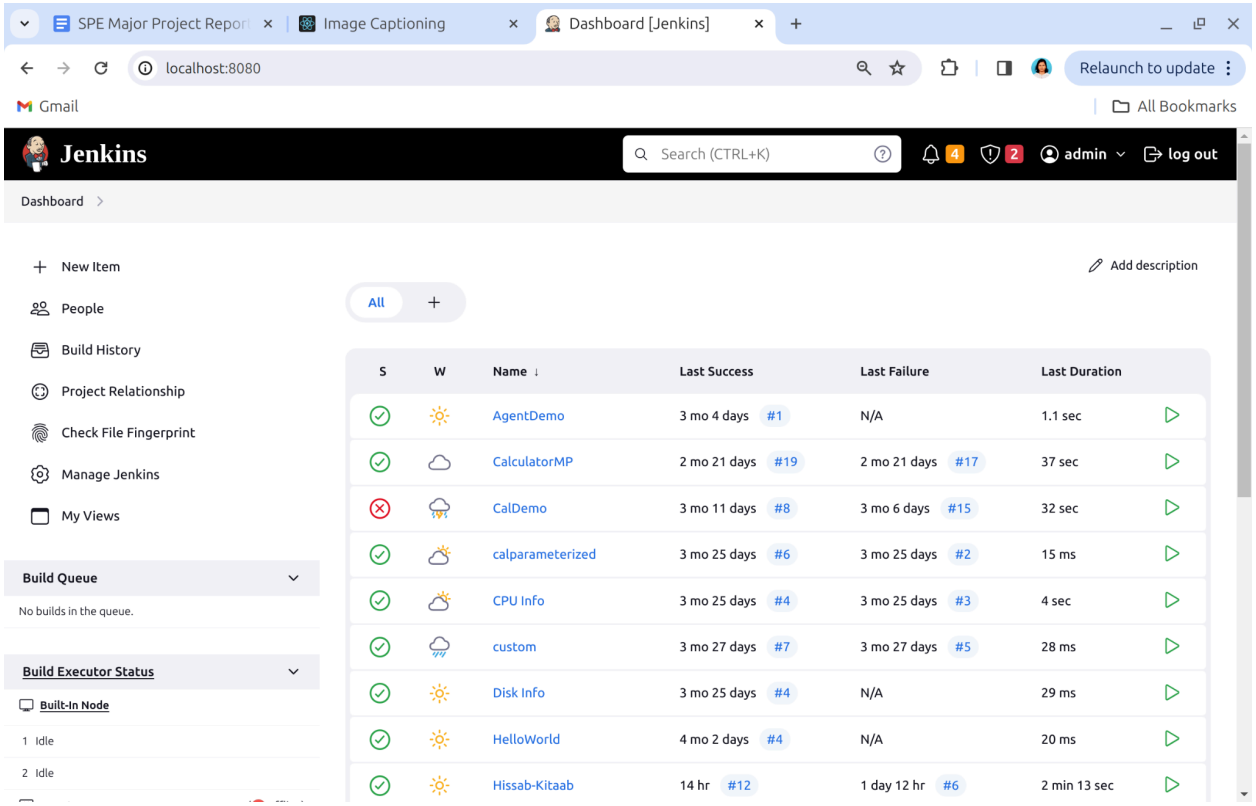
Install Jenkins :
sudo apt-get update
sudo apt-get install jenkins

To check Jenkins version : vim /var/lib/jenkins/config.xml
To copy admin password : sudo cat /var/lib/jenkins/secrets/initialAdminPassword

Start Jenkins:
sudo service jenkins start

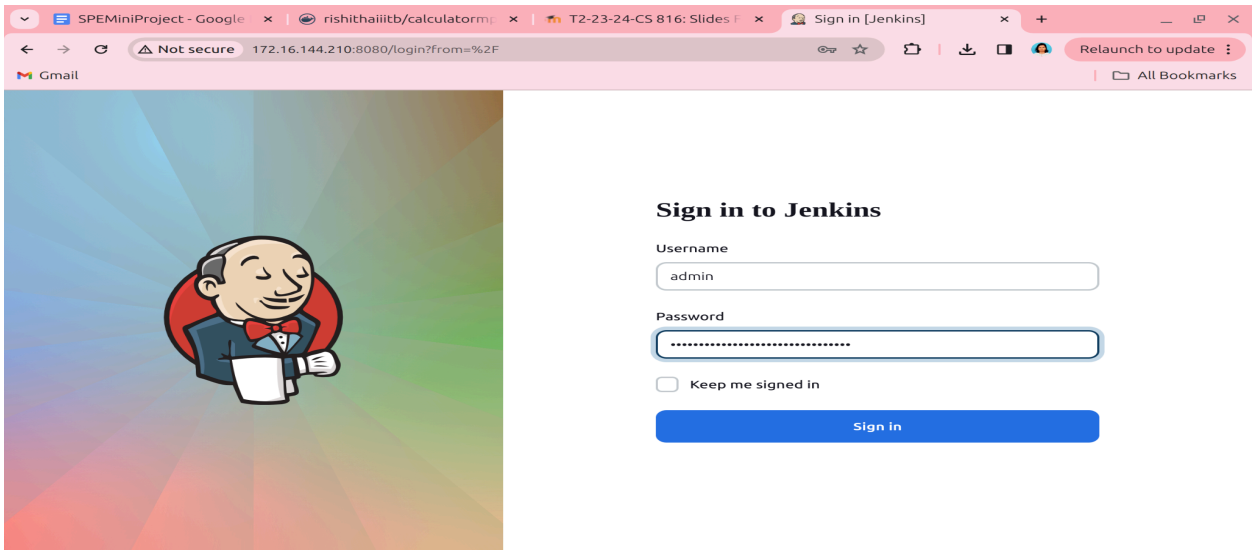
Check Status:
sudo service jenkins status

It should be running then you can open http://localhost:8080/ were we can see the jenkins login page and in that we can give username as admin and password can be copied by following the steps mentioned above. After logging in you can see the jenkins dashboard as mentioned below:

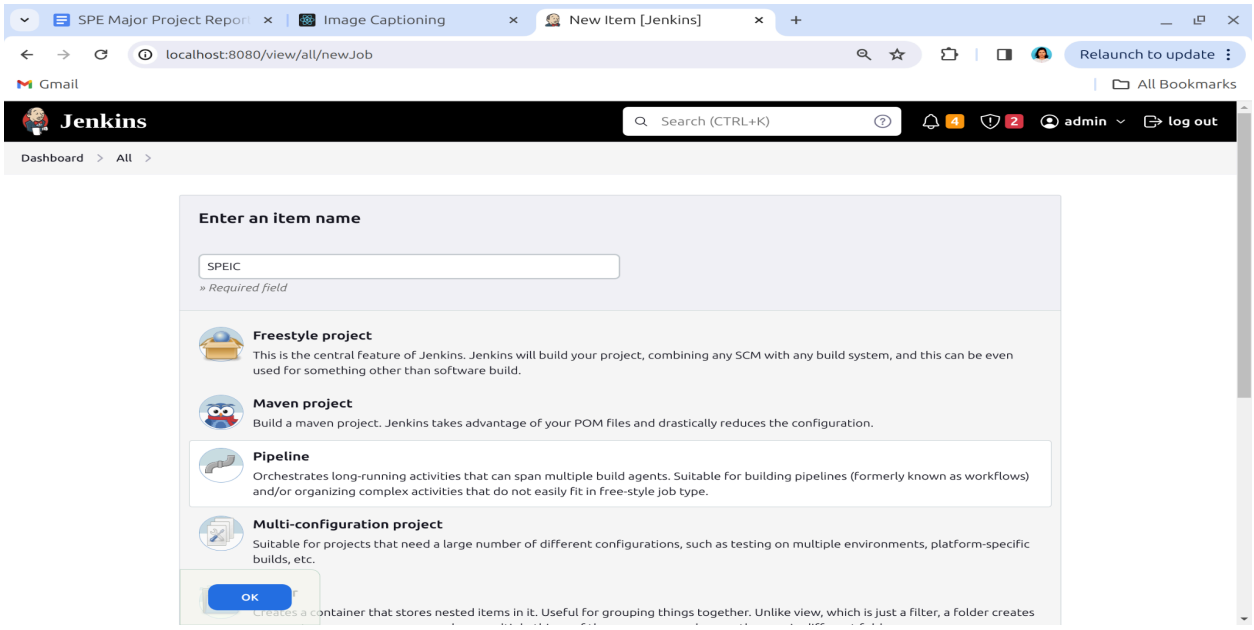


Creating Pipeline in Jenkins:

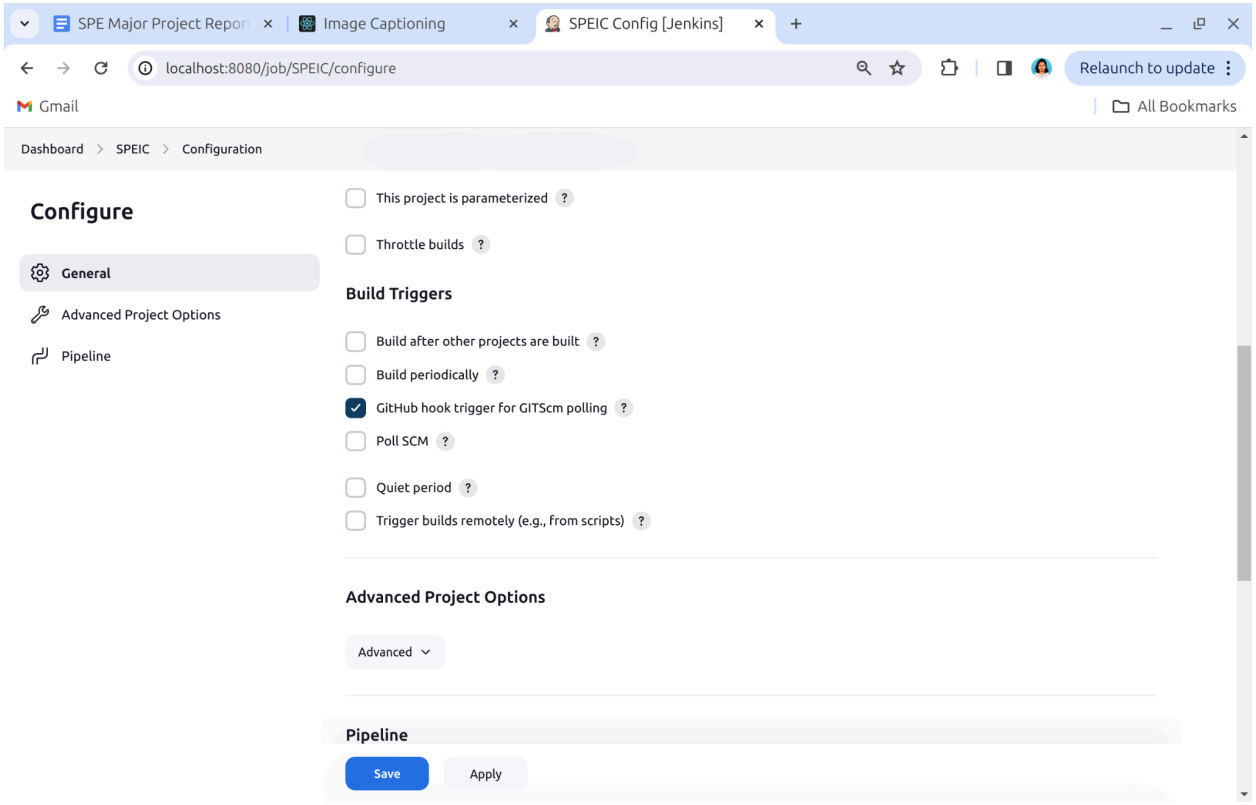
1. Navigate to Jenkins Dashboard:
    a. Log in to your Jenkins dashboard.



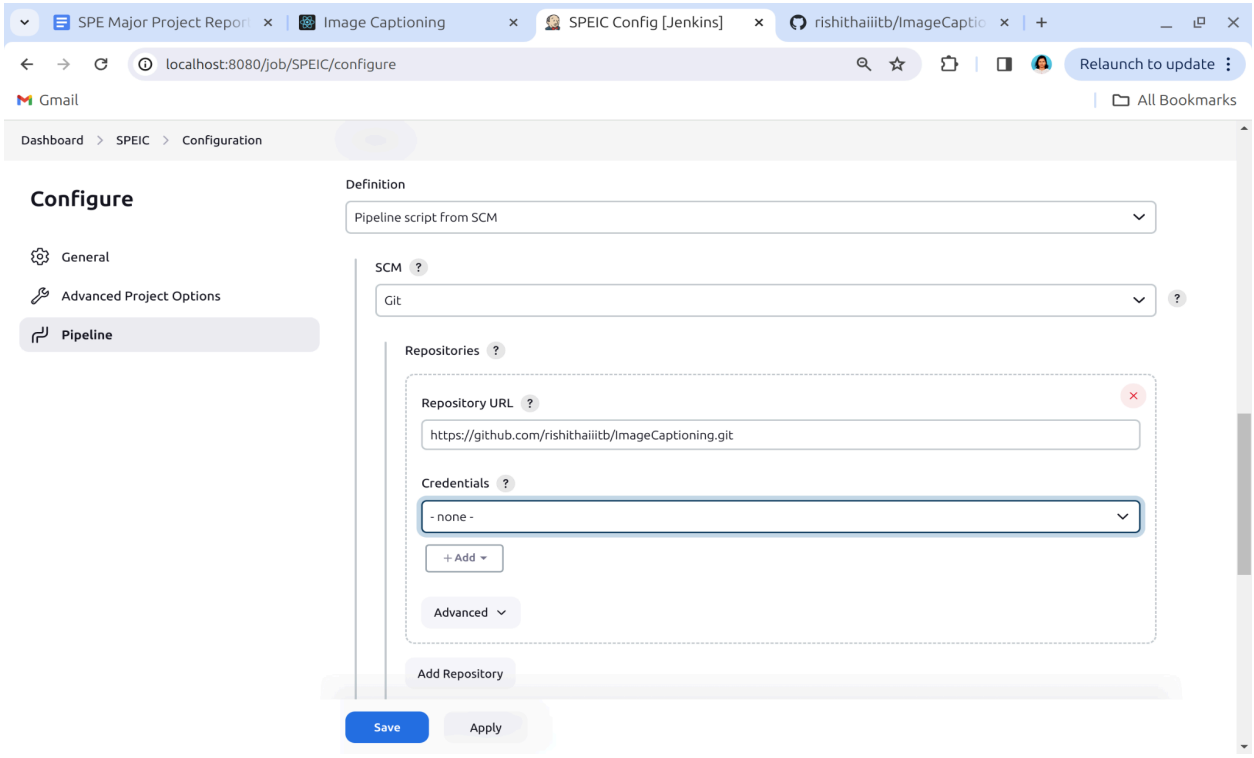
2. Create a New Pipeline Project:
    a. Click on "New Item" to create a new Jenkins item.
    b. Choose "Pipeline" as the project type.
    c. Provide a name for your pipeline project (e.g., "SPEIC").
    d. Click "OK" to proceed.

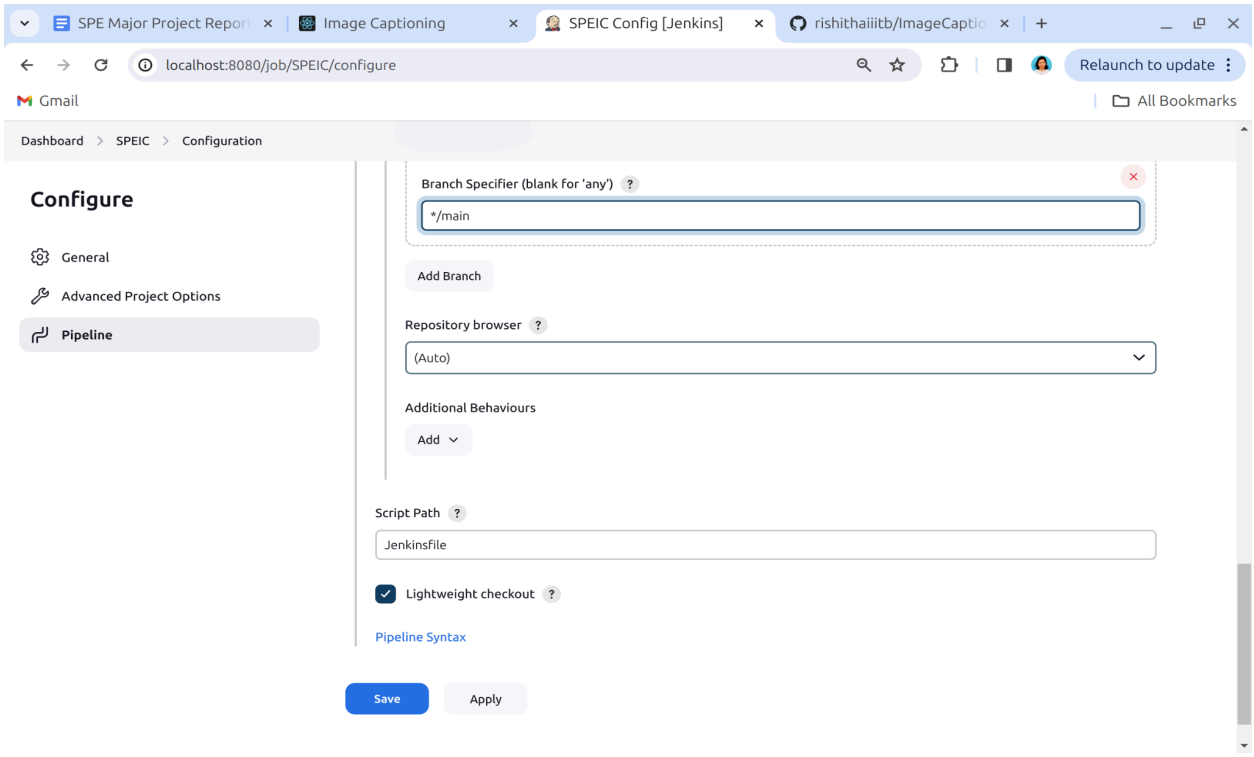


3. Configure GitHub Hook Trigger:
    a. After creating the pipeline, you'll be taken to the job configuration page.
    b. Scroll down to the "Build Triggers" section.
    c. Check the option "GitHub hook trigger for GITScm polling." This enables Jenkins to automatically trigger a build when changes are pushed to your GitHub repository.



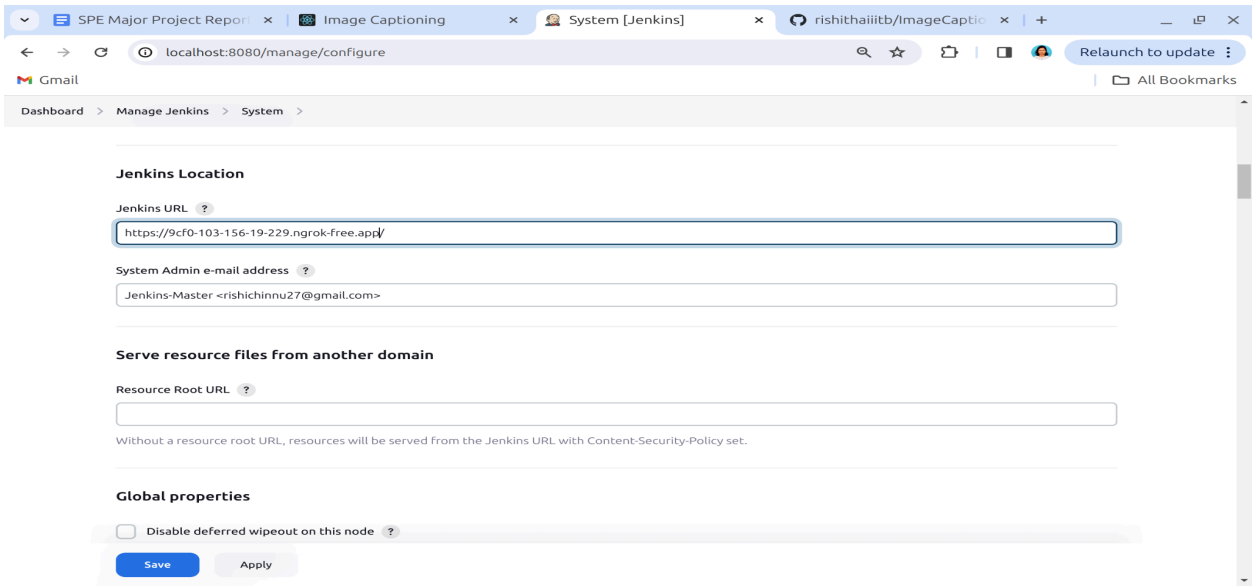
4. Configure Pipeline Script:
    a. In the "Pipeline" section, you need to define the script that Jenkins will execute during the pipeline.
    b. You can write your pipeline script directly in the "Script" text area or reference a script from your version control system.
    c. Scroll down to the bottom of the page and click "Save" to save your pipeline configuration.


SPE Major Project Report
Image Captioning
SPEIC Config [Jenkins]
rishithaiiitb/ImageCaptio
localhost:8080/job/SPEIC/configure
Relaunch to update
Gmail
All Bookmarks
Dashboard > SPEIC > Configuration
Configure
General
Advanced Project Options
Pipeline
Definition
Pipeline script from SCM
SCM
Git
Repositories
Repository URL
https://github.com/rishithaiiitb/ImageCaptioning.git
Credentials
- none -
+ Add
Advanced
Add Repository
Save
Apply

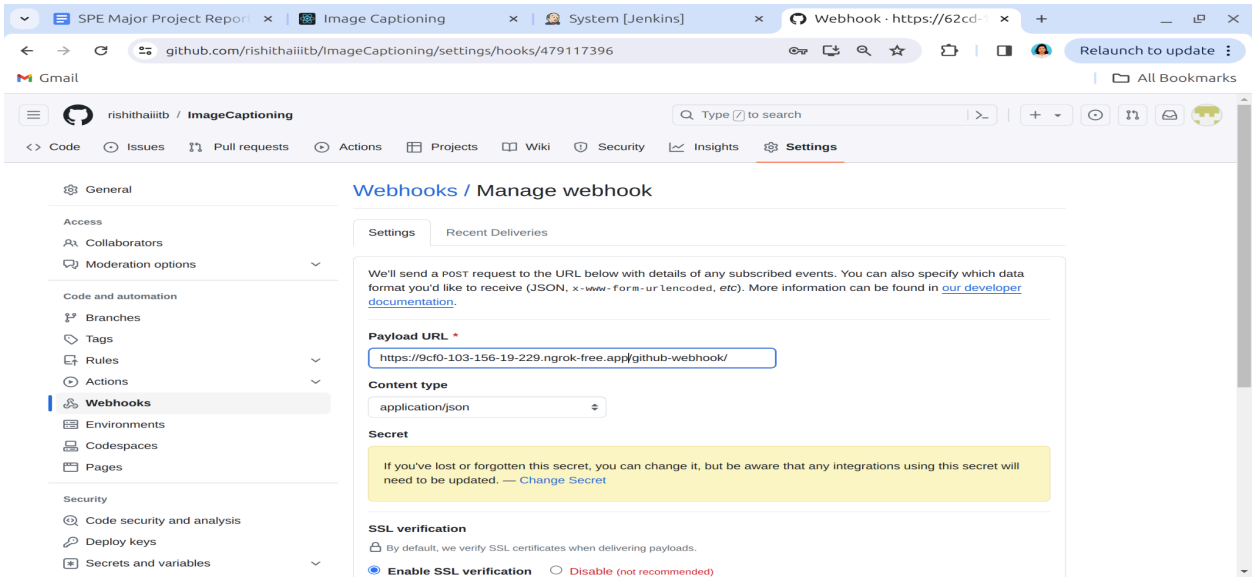

SPE Major Project Report
Image Captioning
SPEIC Config [Jenkins]
rishithaiiitb/ImageCaptio
localhost:8080/job/SPEIC/configure
Relaunch to update
Gmail
All Bookmarks
Dashboard > SPEIC > Configuration
Configure
General
Advanced Project Options
Pipeline
Branch Specifier (blank for 'any')
*/main
Add Branch
Repository browser
(Auto)
Additional Behaviours
Add
Script Path
Jenkinsfile
Lightweight checkout
Pipeline Syntax
Save
Apply

5. Set Up GitHub Webhook:
    a. Open your GitHub repository in a new tab.
    b. Go to the "Settings" tab in your GitHub repository.
    c. Navigate to "Webhooks & Services" or "Webhooks" section.
    d. Click "Add webhook."
    e. In the Payload URL field, enter the URL of your Jenkins server followed by /github-webhook/ (e.g., http://your-jenkins-server/github-webhook/).
    f. Set the Content type to "application/json."
    g. Select "Just the push event" or configure according to your needs.
    h. Click "Add webhook" to save.

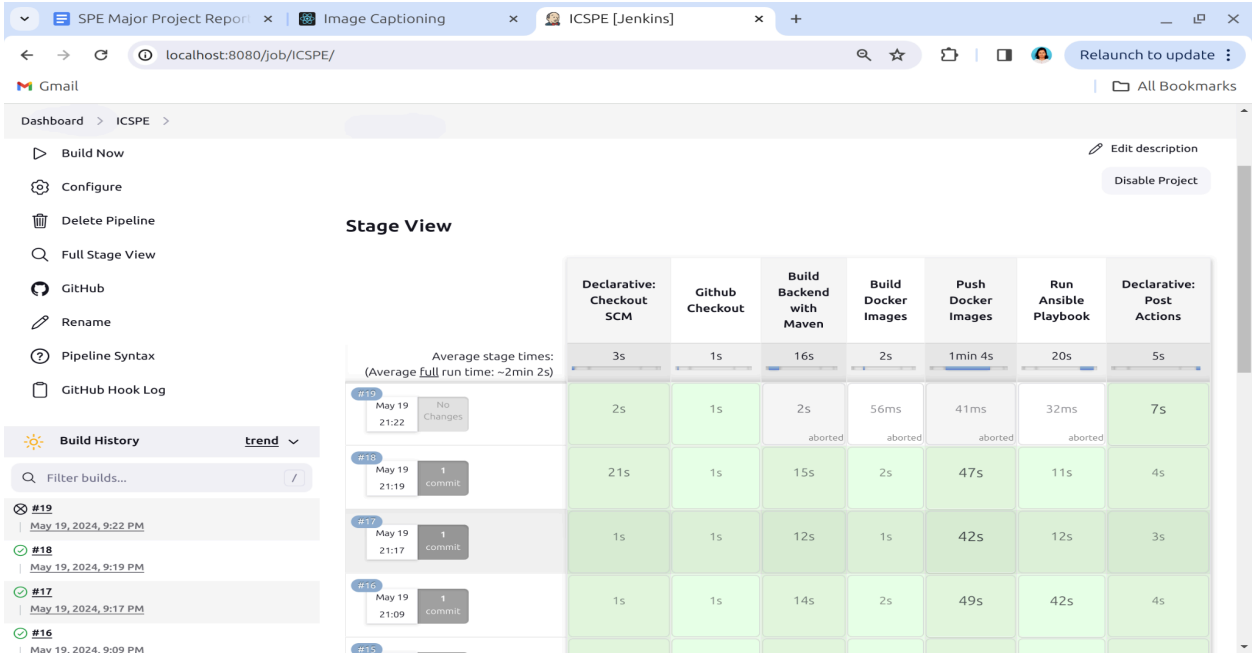
For jenkins url we can use ngrok to obtain public url by giving this command in the terminal:ngrok http 8080 then we need to configure this url in jenkins also as mentioned below:



Github Repository:

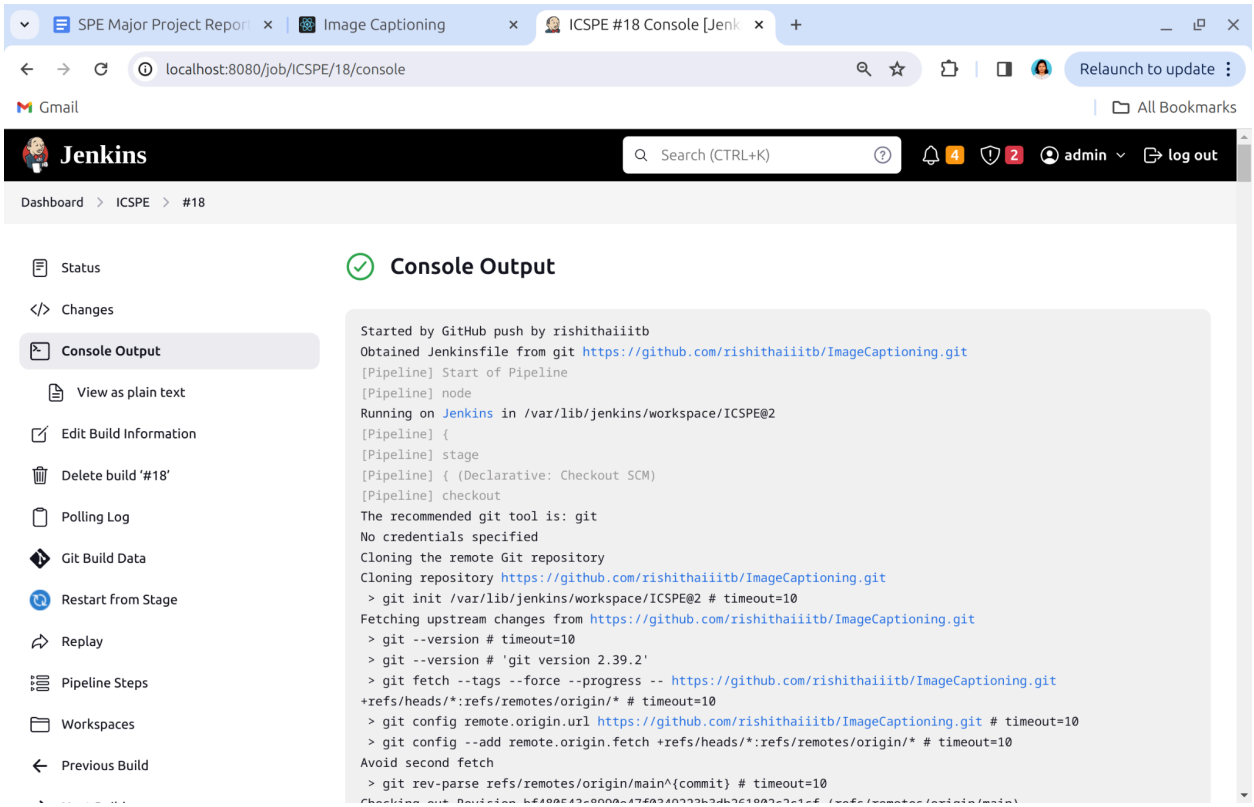


6. Trigger the Pipeline:
    a. Make a change in your GitHub repository (e.g., push a new commit).
    b. This change should automatically trigger the Jenkins pipeline due to the GitHub webhook.



7. View Pipeline Execution:
    a. Go back to your Jenkins dashboard.
    b. Navigate to your pipeline project.
    c. You should see the pipeline execution triggered by the GitHub webhook.

In this we can see that pipeline is started by github.So whenever our github repository has new changes then this pipeline is automatically triggered in the and all the stages mentioned in the pipeline script are executed. We need to keep ngrok active to achieve this automatic triggering.



Pipeline Script:

```
pipeline {
    agent any

    environment {
        DOCKERHUB_CREDENTIALS = 'DockerHubCred'
        DOCKER_COMPOSE_FILE = 'docker-compose.yml'
        GITHUB_REPO_URL = 'https://github.com/rishithaiiitb/ImageCaptioning.git'
    }
```

```
stages {
    stage('Github Checkout') {
        steps {
            script {
                git branch: 'main', url: "${GITHUB_REPO_URL}"
            }
        }
    }

    stage('Build Backend with Maven') {
        steps {
            script {
                dir('imagecaptioning') {
                        sh 'mvn clean package'
                    sh 'mvn clean install'
                }
            }
        }
    }


    stage('Build Docker Images') {
        steps {
            script {
                docker.build('rishithaiiitb/backend', './imagecaptioning')
                docker.build('rishithaiiitb/frontend', './icfe')
                docker.build('rishithaiiitb/bemodel', './model')
            }
        }
    }

    stage('Push Docker Images') {
        steps {
            script {
                docker.withRegistry('', "${DOCKERHUB_CREDENTIALS}") {
                    // Push Backend Docker Image
                    sh 'docker tag rishithaiiitb/backend rishithaiiitb/backend:latest'
                    sh 'docker push rishithaiiitb/backend:latest'

                    // Push Frontend Docker Image
                    sh 'docker tag rishithaiiitb/frontend rishithaiiitb/frontend:latest'
                    sh 'docker push rishithaiiitb/frontend:latest'

                    // Push Model Docker Image
                    sh 'docker tag rishithaiiitb/bemodel rishithaiiitb/bemodel:latest'
                    sh 'docker push rishithaiiitb/bemodel:latest'
                }
```

```
            }
        }
    }

    stage('Run Ansible Playbook') {
        steps {
            script {
                ansiblePlaybook(
                    playbook: 'deploy.yml',
                    inventory: 'inventory'
                )
            }
        }
    }
}

post {
    always {
        script {
            sh 'docker logout'
        }
        script {
            emailext(
                subject: "Pipeline Status: ${currentBuild.result}",
                body: '''Build Status: ${currentBuild.result}
                        <br><br>
                        Build URL: ${BUILD_URL}
                        <br><br>
                        Check the Jenkins console for details.''',
                to: "rishichinnu27@gmail.com",
                from: "jenkins@yourdomain.com",
                mimeType: 'text/html'
            )
        }
    }
}
}
```

Explanation:

Environment Variables:

- `DOCKERHUB_CREDENTIALS`: Credentials for DockerHub login.
- `DOCKER_COMPOSE_FILE`: The Docker Compose file to be used.

- `GITHUB_REPO_URL`: URL of the GitHub repository to be checked out.

Stage 1: GitHub Checkout

Checks out the source code from the specified GitHub repository and branch (`main`). This ensures the pipeline works with the latest version of the code.

Stage 2: Build Backend with Maven

Builds the Maven project by cleaning and packaging the backend code located in the `imagecaptioning` directory. This compiles the code, runs tests, and generates the necessary JAR files.

Stage 3: Build Docker Images

Constructs Docker images for the backend, frontend, and model using their respective Dockerfiles located in the `imagecaptioning`, `icfe`, and `model` directories.

Stage 4: Push Docker Images

Tags and pushes the newly built Docker images to DockerHub using the provided credentials. This makes the Docker images available for deployment.

Stage 5: Run Ansible Playbook

Executes an Ansible playbook defined in the `deploy.yml` file to automate the deployment tasks. This configures infrastructure and applications according to the defined specifications in the playbook and inventory file.

Post-build Actions:

Always:

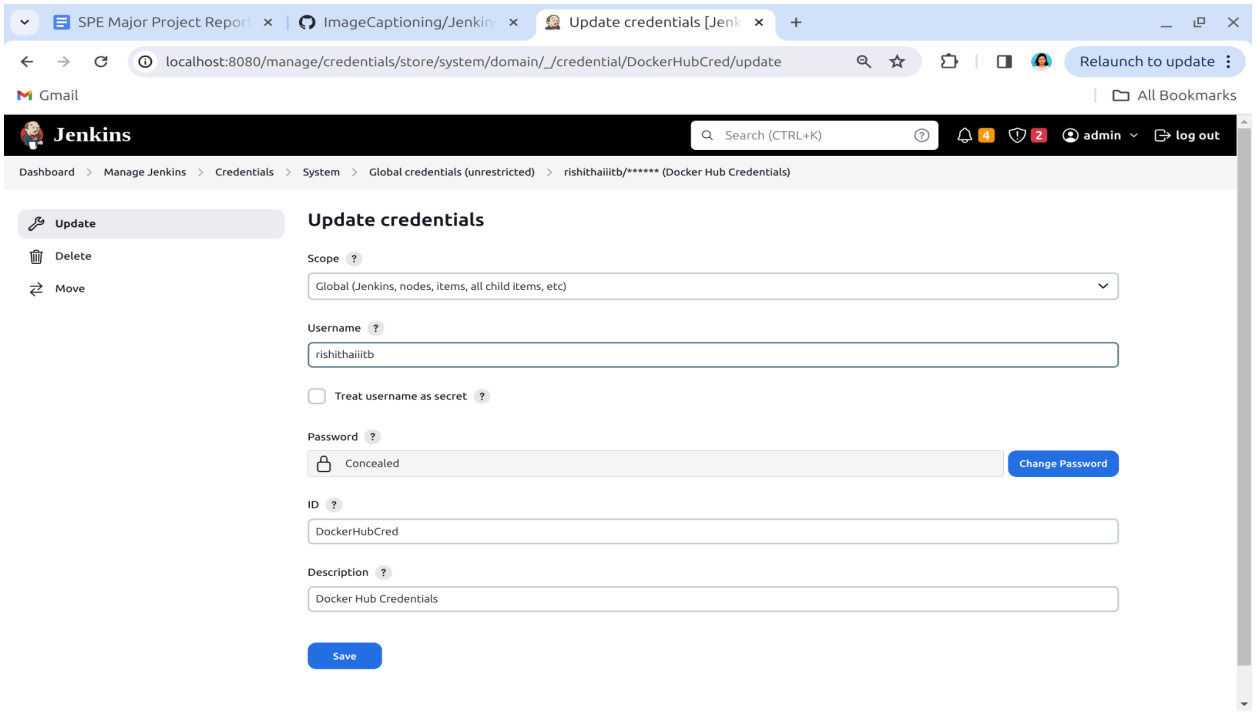
Docker Logout: Logs out from Docker to ensure credentials are not left exposed.

Email Notification: Sends an email notification to the specified email address (`rishichinnu27@gmail.com`) with the pipeline's status and a link to the build for quick access and reference.

Configurations:

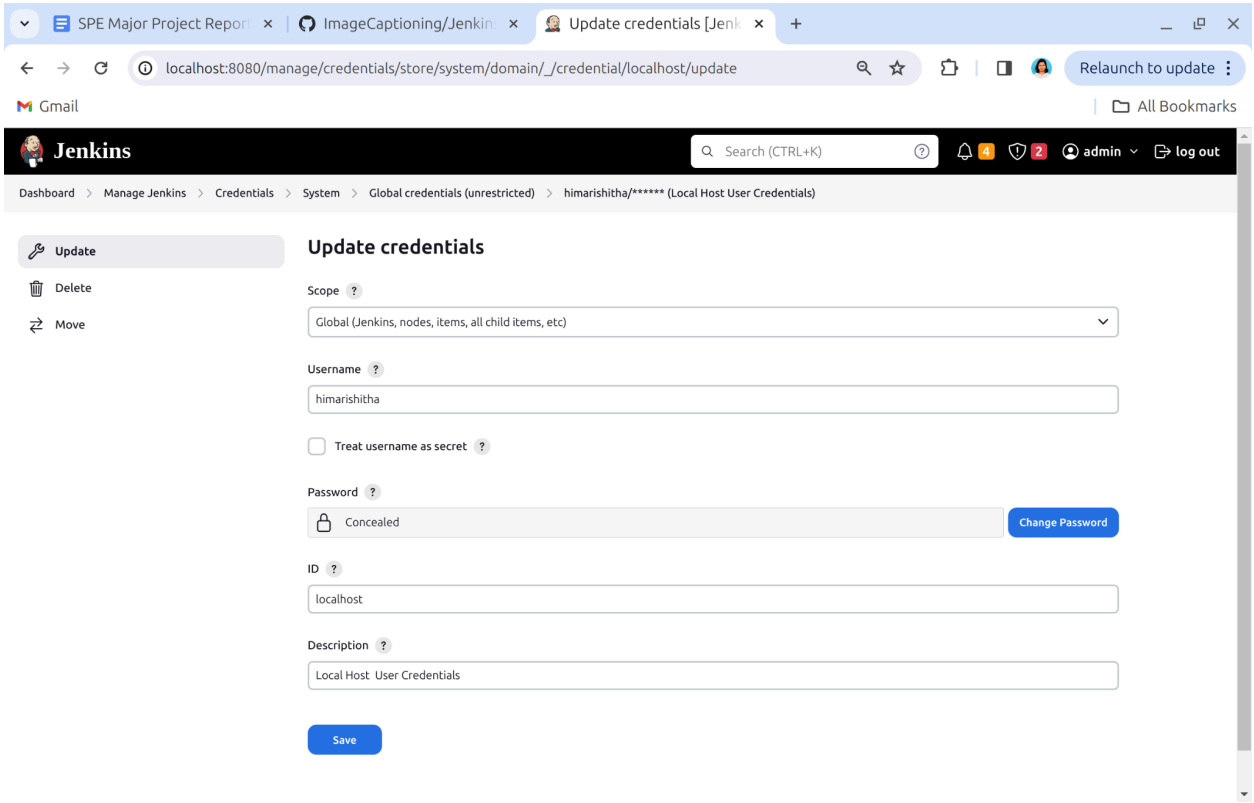
Docker Credentials:

- Navigate to Jenkins:
  - Open your Jenkins dashboard in a web browser.
- Access Credentials Management:
  - Click on "Manage Jenkins" in the left sidebar.
- Manage Credentials:
  - Click on "Manage Credentials" from the dropdown menu.
- Add Docker Hub Credentials:
  - Click on "Global credentials (unrestricted)."
  - Click "(Add Credentials)."
  - Choose "Username with password" as the kind.
  - Enter Docker Hub username and password.
  - Optionally, provide an ID (e.g., "DockerHubCred").
  - Click "OK" to save.



Ansible Credentials:

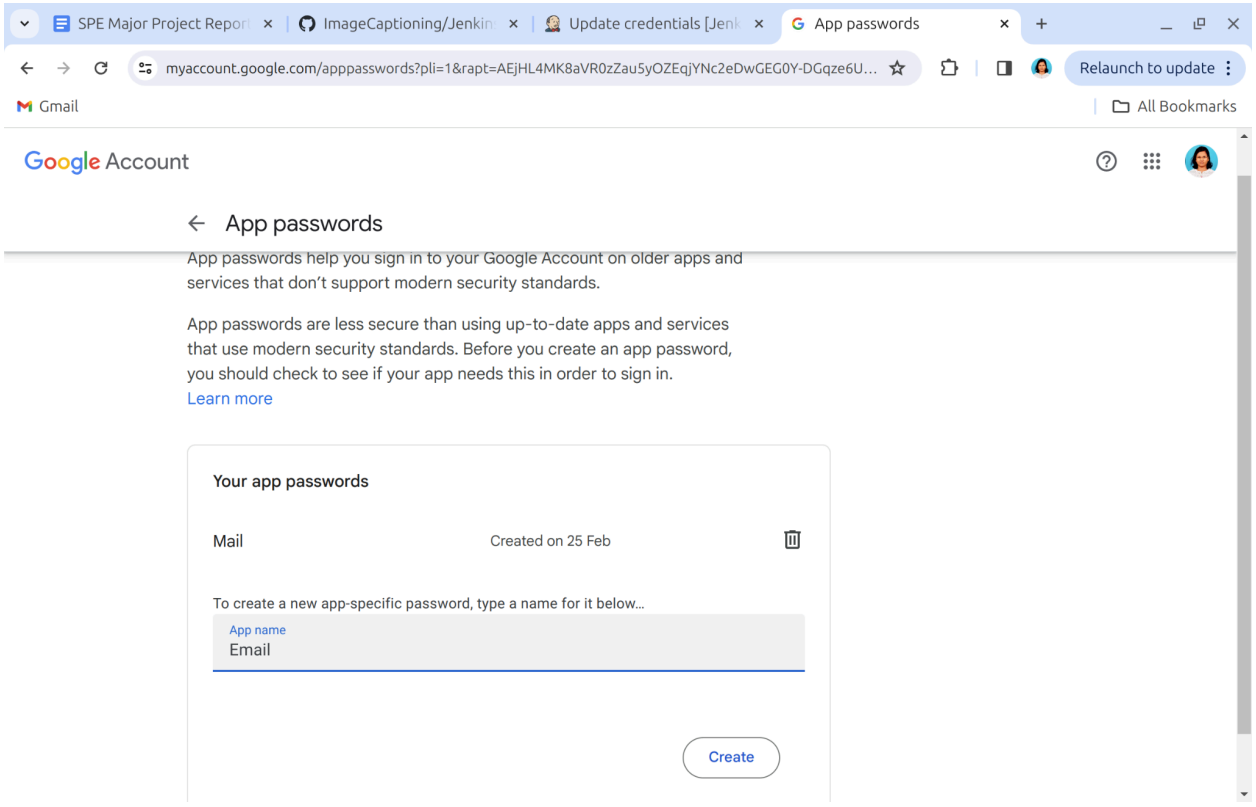
- Generate SSH Key Pair:
    - On the machine where Ansible will be executed, generate an SSH key pair using the ssh-keygen command.
- Navigate to Jenkins Dashboard:
    - Open your Jenkins dashboard in a web browser.
- Access Credentials Management:
    - Click on "Manage Jenkins" in the left sidebar.
- Manage Credentials:
    - Click on "Manage Credentials" from the dropdown menu.
- Add SSH Key Credentials:
    - Click on "(Add Credentials)" on the left.
    - Choose the credential type as "Secret text" or "SSH Username with private key."
    - Enter the SSH private key or secret in the provided fields(Local Host Credentials).
    - Optionally, provide an ID for these credentials (e.g., "AnsibleCred").
    - Click "OK" to save the credentials.



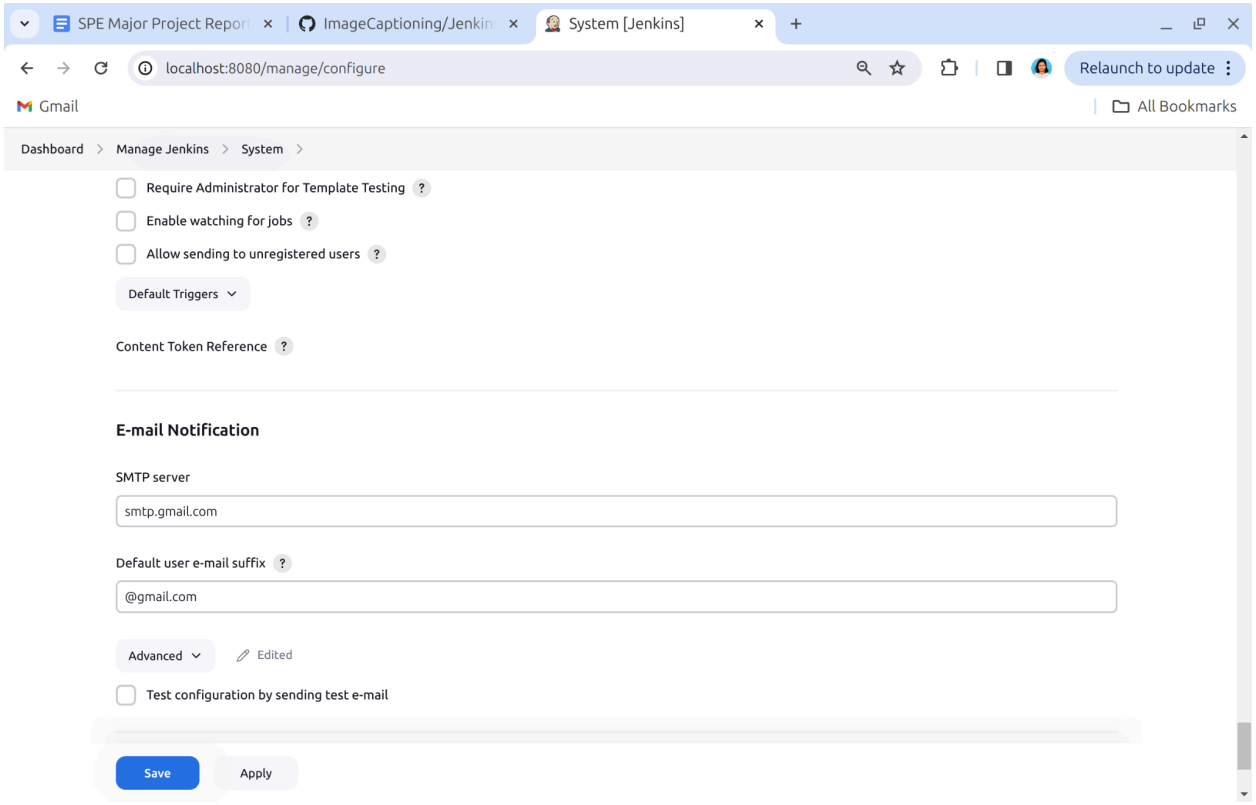
Email Notification:

To automate email notifications after each Jenkins build and include a link to the corresponding build, proceed by configuring email settings in Jenkins using the following steps:

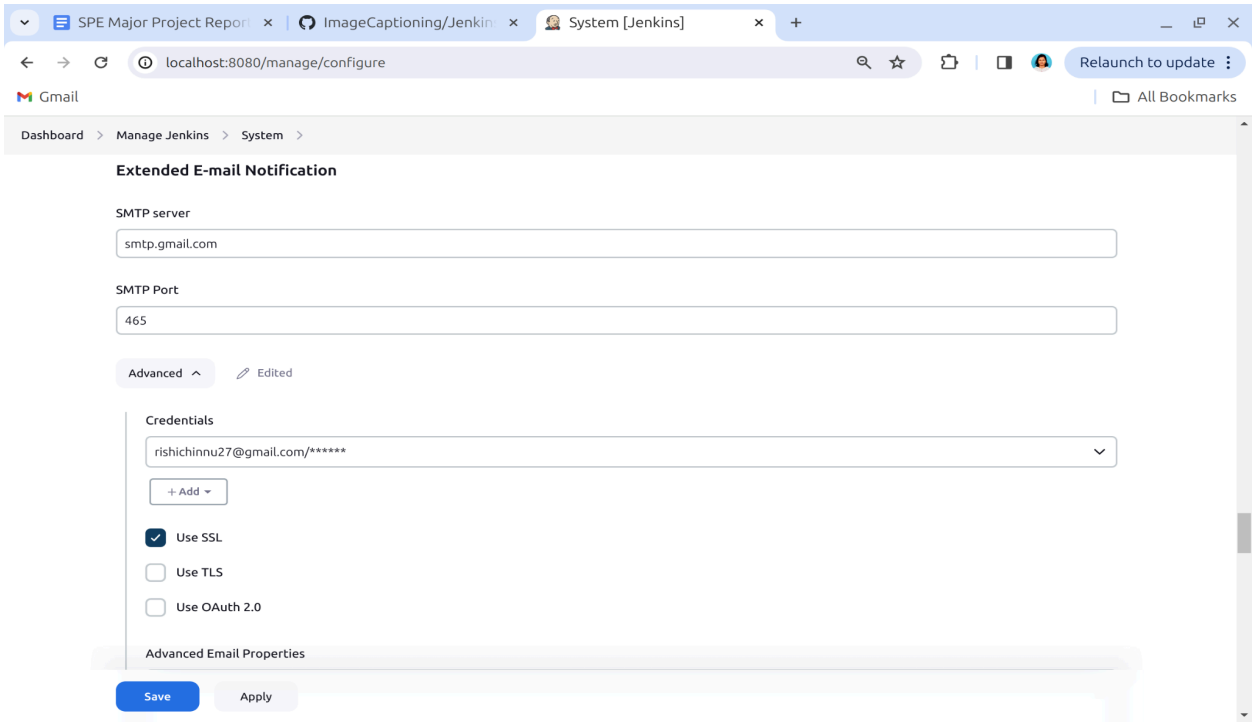
1. Generate App Password:
    a. Go to your Google Account's Security page.
    b. Go to 2-step verification tab and click on "App passwords."
    c. Select "Mail" and the device/application for which you're generating the password.
    d. Click "Generate."
    e. Once generated, copy the app password provided by Google. This is a one-time password specifically for the chosen application.



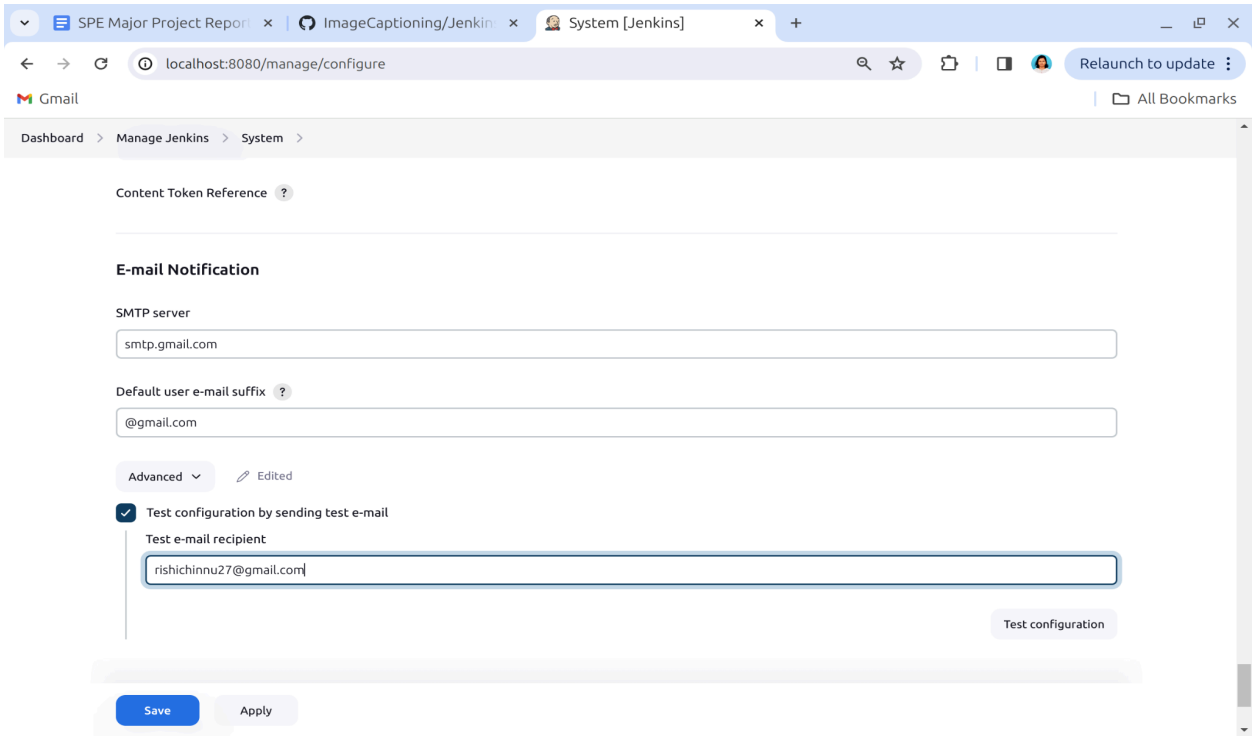
2. Configure Jenkins Email Notification:
    a. Open your Jenkins instance and navigate to "Manage Jenkins" > "Configure System."
    b. In the "E-mail Notification" section, enter the following:
        i. SMTP server: smtp.gmail.com
        ii. SMTP port: 465
        iii. Use SSL: Checked
        iv. SMTP Authentication: Checked
        v. Username: Your Gmail email address
        vi. Password: Paste the copied App Password
        vii. Default user email suffix: (Optional) If applicable



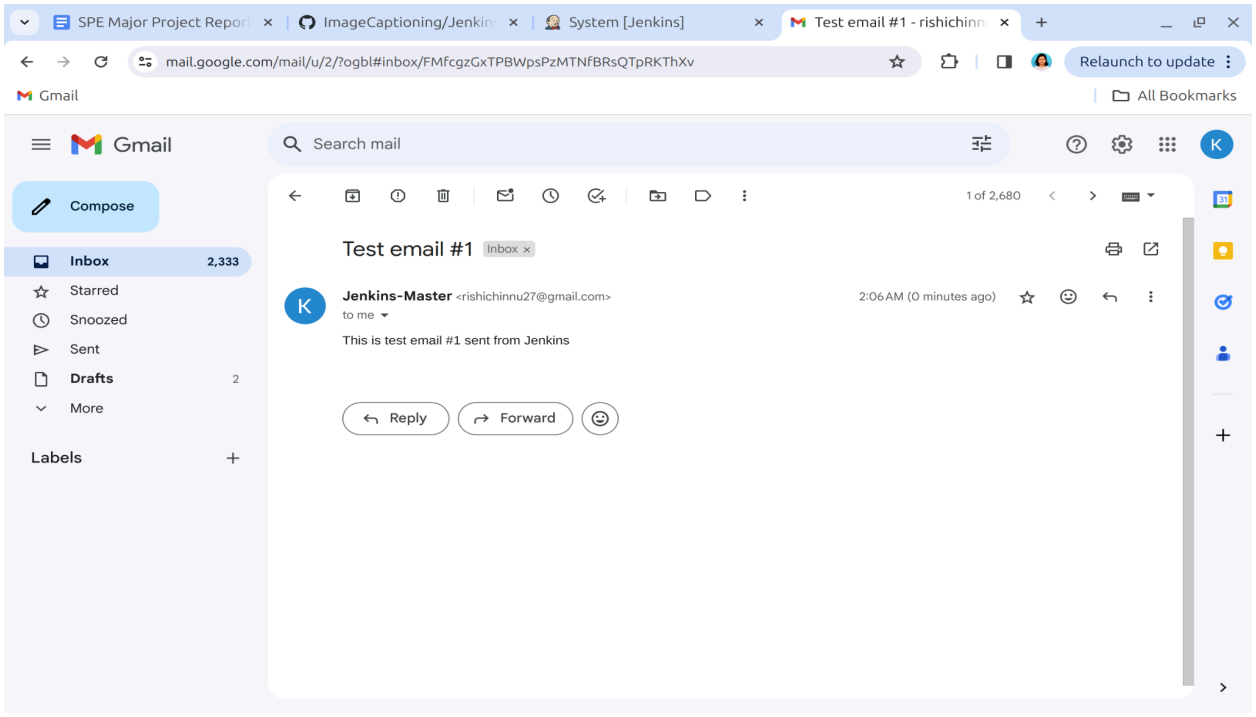
3. Advanced Settings:
    a. Click on "Advanced" to reveal additional settings:
        i. SMTP Port: Set the port for your SMTP server.
        ii. Use SSL: Checked



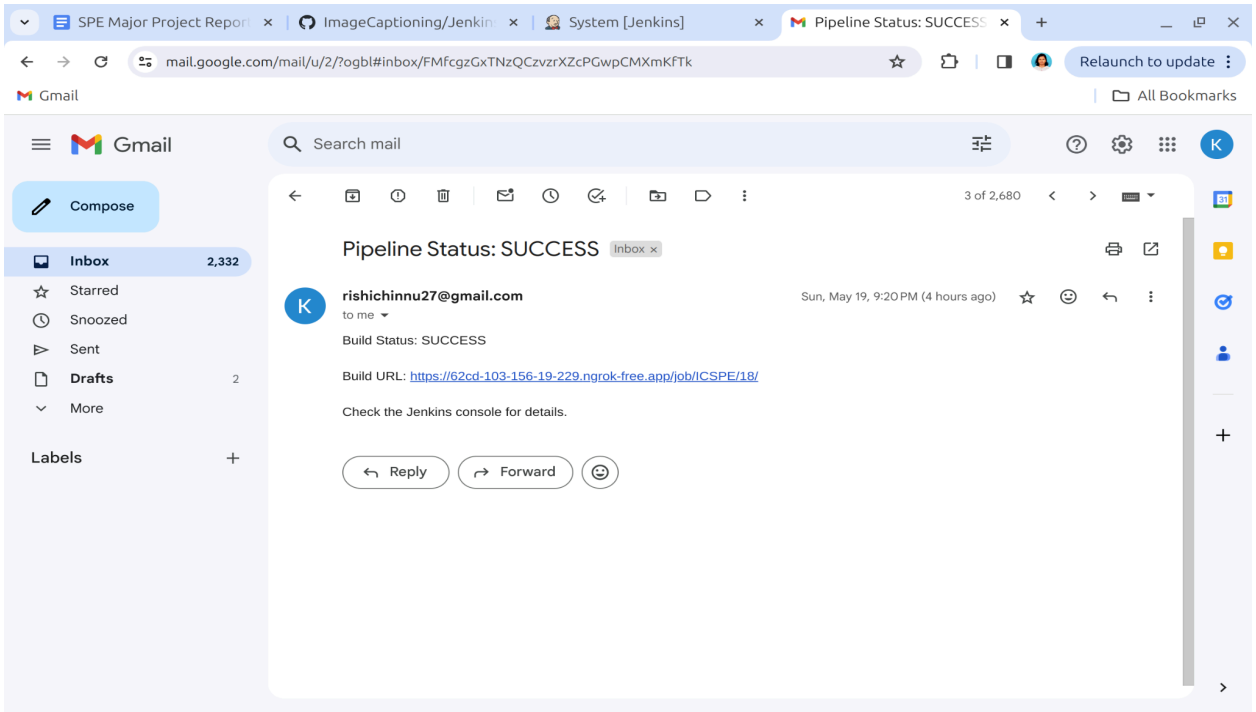
4. Test Configuration:
    a. Use the "Test configuration" button to send a test email. This helps ensure that your configuration is correct.



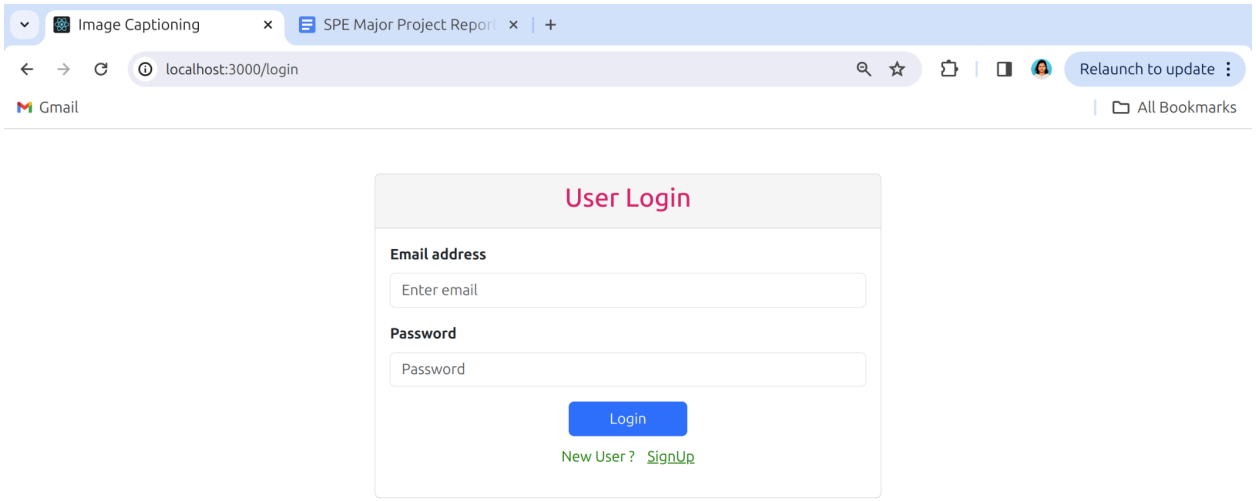
Test Email:



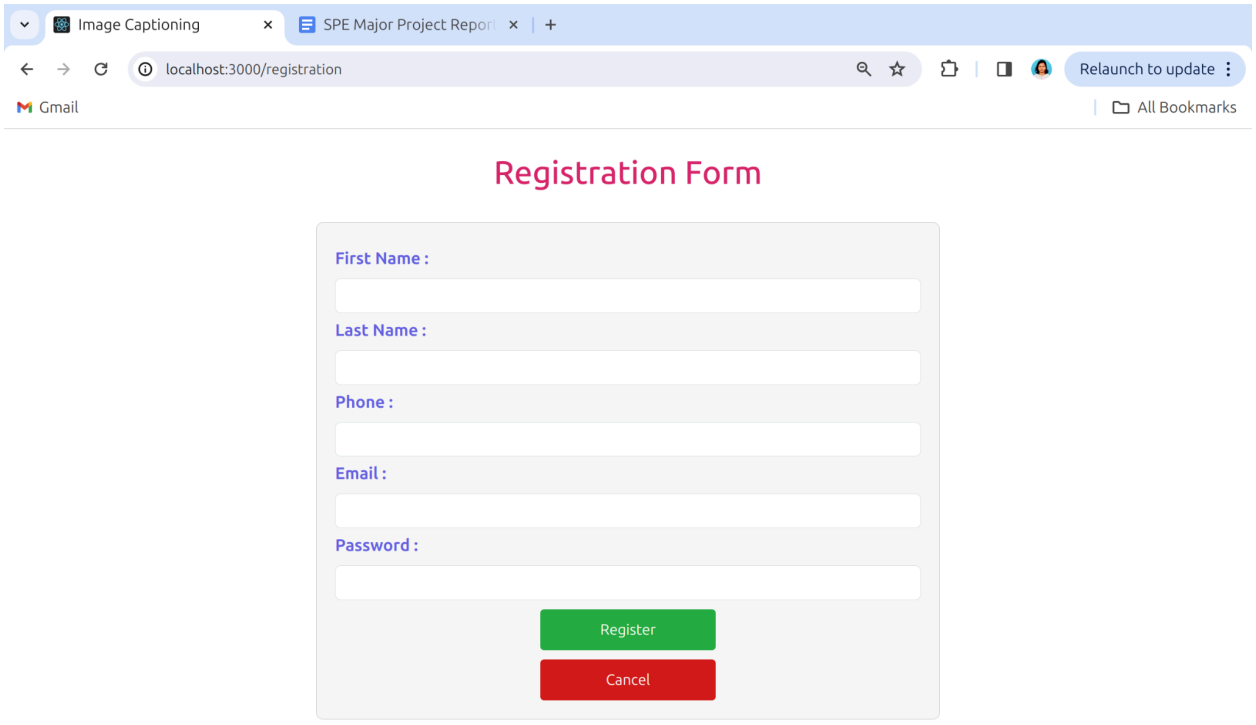
Mail received after build:



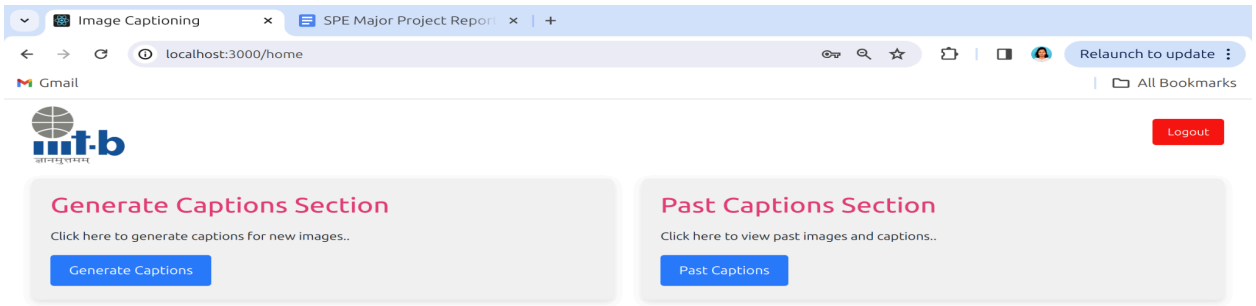
UI:



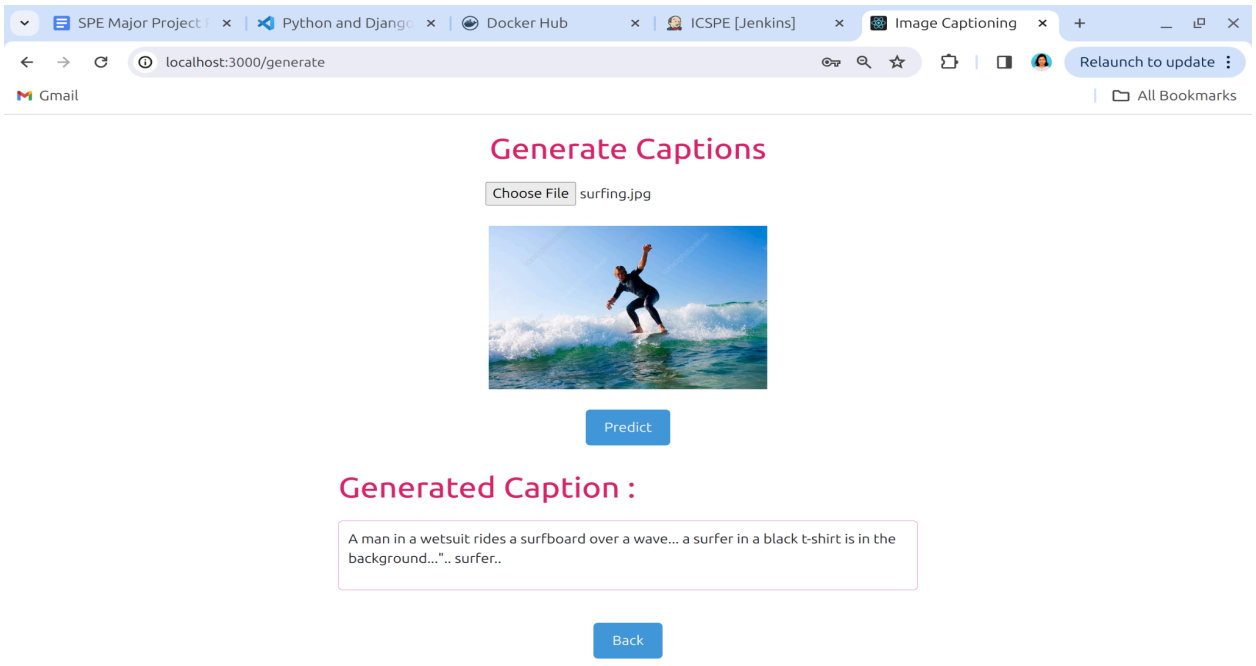
This is the login page where users can log in using email address and password.



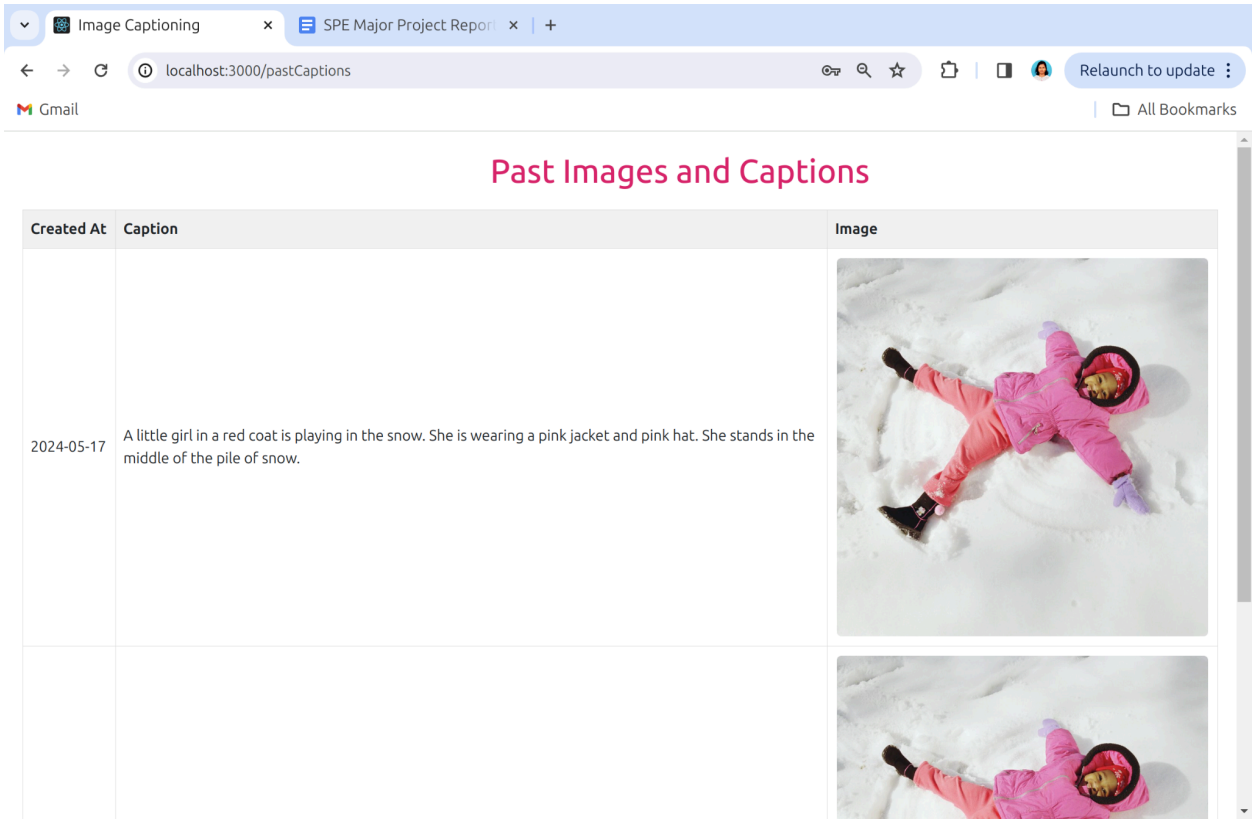
This is the registration page where users can register by entering the required credentials



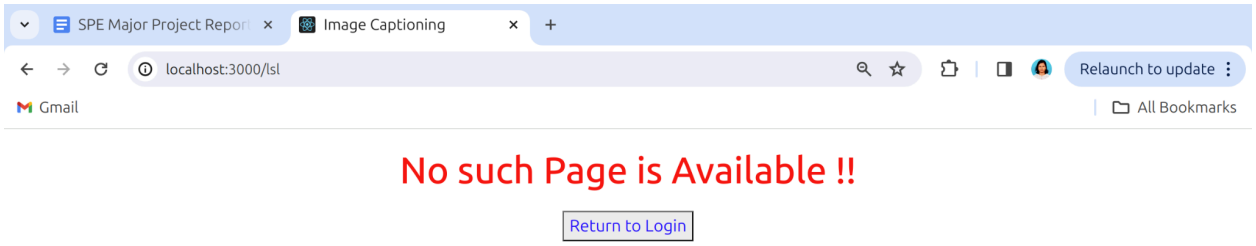
This is the dashboard where user will be redirected to after logging in successfully.



Once clicked on “generate caption section” in the previous page, the user will be redirected to this page where the user can upload an image to generate the caption.



Once clicked on “past captions section” in the previous page, the user will be redirected to this page where the user can view the previously uploaded images along with their generated caption.



This is the default page that the user will get if the user enters the URL of some page that doesn’t exist.

Containerization:

Containerization involves encapsulating an application and its dependencies into a container, which can then be run consistently across various environments. Docker is a popular tool for containerization, and Dockerfiles are scripts used to define the steps to create Docker images.

We are creating 3 containers:

1. Frontend

```
FROM node:alpine

WORKDIR /frontend

COPY . /frontend

RUN npm install --legacy-peer-deps
RUN npm install axios

CMD ["npm","start"]
```

---

1. `FROM node:alpine`
   - This specifies the base image for the Docker image. `node:alpine` is a lightweight Node.js image based on Alpine Linux, which is known for its small size.
2. `WORKDIR /frontend`
   - This sets the working directory inside the Docker container to `/frontend`. All subsequent commands will be run in this directory.
3. `COPY . /frontend`
   - This copies all files from the current directory on your host machine to the `/frontend` directory in the Docker container. This includes your application's source code and other necessary files.
4. `RUN npm install --legacy-peer-deps`
   - This command installs the dependencies listed in your `package.json` file. The `--legacy-peer-deps` flag tells npm to ignore peer dependencies that

might be incompatible, which can help resolve some dependency issues This line specifies the default command to be executed when a container is started from the image. It runs the Java application by executing the JAR file with the `java -jar` command.

5. `RUN npm install axios`

   - This command installs the dependencies listed in your `package.json` file. The `--legacy-peer-deps` flag tells npm to ignore peer dependencies that might be incompatible, which can help resolve some dependency issues

6. `CMD ["npm","start"]`
   - This sets the default command to be run when the container starts. `npm start` typically runs the `start` script defined in your `package.json`. This script usually starts your application, such as launching a development server for a frontend application.

2. Backend

```
# Base image
FROM openjdk:17

# Expose the desired port
EXPOSE 8020

#Copying Jar File
ADD ./target/imagecaptioning-0.0.1-SNAPSHOT.jar ./

#Setting Working Directory in container
WORKDIR ./

#Run Jar File
CMD ["java", "-jar", "imagecaptioning-0.0.1-SNAPSHOT.jar"]
```

---

1. `FROM openjdk:17`
   - This specifies the base image for the Docker image. `openjdk:17` is an official OpenJDK image that includes the Java Development Kit (JDK)

version 17. This image provides the runtime environment needed to run Java applications.

2. `EXPOSE 8020`
    - This informs Docker that the container will listen on port 8020 at runtime. This does not actually publish the port; it just serves as documentation and a hint for users running the container that the application inside expects to be accessed on this port.
3. `ADD ./target/imagecaptioning-0.0.1-SNAPSHOT.jar ./`
    - This copies the JAR file from the `target` directory on your host machine to the current directory (`./`) in the Docker container. `ADD` can handle local files and URLs, but here it functions similarly to `COPY` for local files.
4. `WORKDIR ./`
    - This sets the working directory inside the Docker container to the current directory (`./`). All subsequent commands will be run in this directory. However, since this is already the default directory, you can omit this line if you prefer.
5. `CMD ["java", "-jar", "imagecaptioning-0.0.1-SNAPSHOT.jar"]`
    - This sets the default command to be run when the container starts. In this case, it runs the JAR file using `java -jar`, which starts the Java application.

3. Model

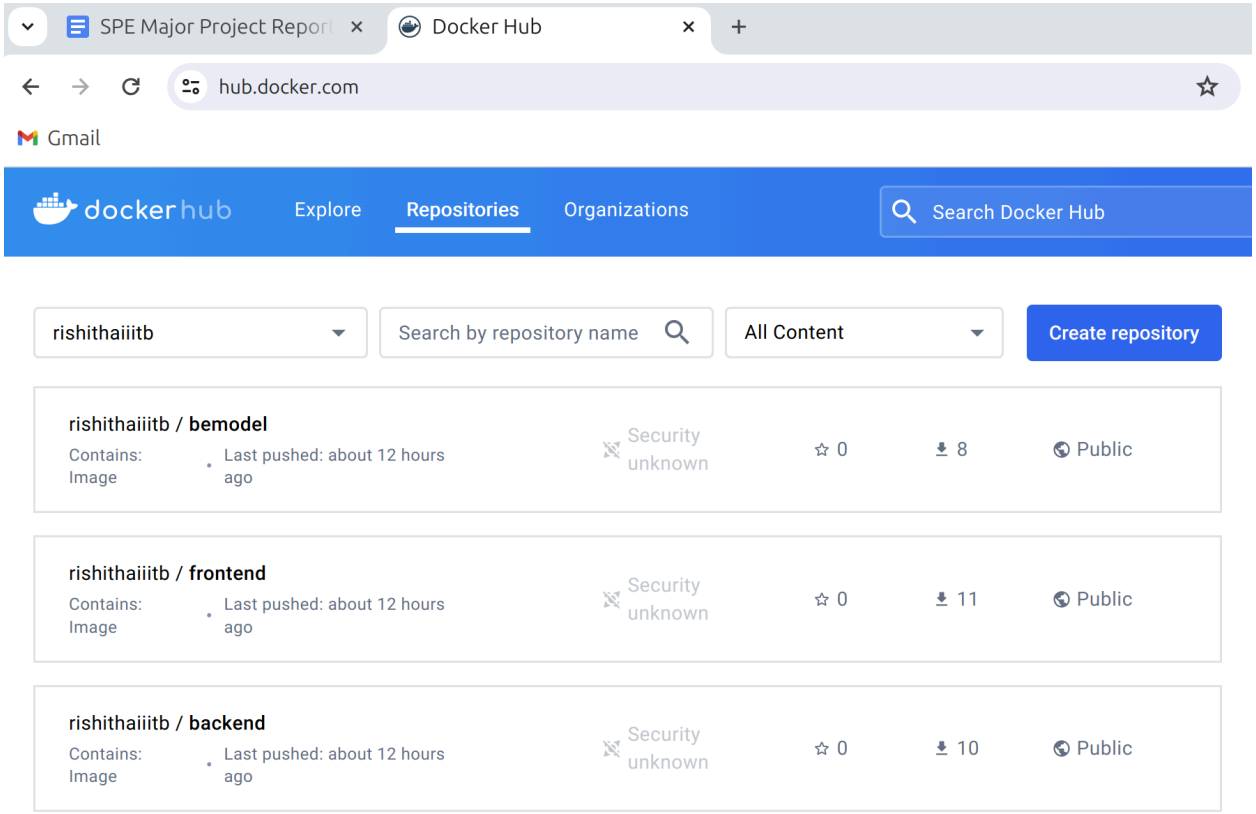
```
# Base image
FROM python:3.10


# Set environment variables
ENV PYTHONUNBUFFERED 1
ENV PYTHONDONTWRITEBYTECODE 1

# Set the working directory
WORKDIR /myapp

# Install dependencies
COPY requirements.txt .
RUN pip install -r ./requirements.txt

# Copy the Django project
COPY . .

EXPOSE 8060

# Run the Django server on localhost and port 8094
CMD ["python", "manage.py", "runserver", "0.0.0.0:8060"]
```

1. `FROM python:3.10`
    - This specifies the base image for the Docker image. `python:3.10` is an official Python image that includes Python version 3.10. This image provides the runtime environment needed to run Python applications.
2. `ENV PYTHONUNBUFFERED 1`
    - This environment variable ensures that the Python output is sent straight to the terminal (without being buffered). This can be useful for logging purposes.
3. `ENV PYTHONDONTWRITEBYTECODE 1`
    - This environment variable tells Python not to write `.pyc` files (Python bytecode) to the disk. This can help reduce the number of files generated and avoid potential issues with bytecode compatibility.
4. `WORKDIR /myapp`
    - This sets the working directory inside the Docker container to `/myapp`. All subsequent commands will be run in this directory.
5. `COPY requirements.txt .`
    - This copies the `requirements.txt` file from the current directory on your host machine to the `/myapp` directory in the Docker container. This file typically contains a list of the Python dependencies for your project.
6. `RUN pip install -r ./requirements.txt`
    - This command installs the Python dependencies listed in the `requirements.txt` file using `pip`.
7. `COPY . .`
    - This copies all the files from the current directory on your host machine to the `/myapp` directory in the Docker container. This includes your Django project's source code and other necessary files.
8. `EXPOSE 8060`
    - This informs Docker that the container will listen on port 8060 at runtime. This does not actually publish the port; it just serves as documentation and a hint for users running the container that the application inside expects to be accessed on this port.
9. `CMD ["python", "manage.py", "runserver", "0.0.0.0:8060"]`
    - This sets the default command to be run when the container starts. In this case, it runs the Django development server, making the application accessible on all network interfaces (`0.0.0.0`) at port 8060.

After the pipeline execution is complete, it becomes evident that our Docker images are efficiently pushed to docker hub and also retrieved from the specified registry. This process involves the seamless integration of the Jenkins pipeline with Docker, where the necessary image is pulled from the designated repository.

We can run these docker images using command “docker run -it –name (any name) <image name>:latest”.

```
himarishitha@himarishitha-Inspiron-13-5310:~$ docker images
REPOSITORY                    TAG       IMAGE ID       CREATED          SIZE
rishithaiiitb/backend         latest    a22b76423693   12 hours ago     523MB
rishithaiiitb/bemodel         latest    1165b161875f   22 hours ago     9.18GB
rishithaiiitb/frontend        latest    55ce9161999e   22 hours ago     383MB
```



Docker Compose:

Docker Compose is a tool designed to simplify the management of multi-container Docker applications. It allows developers to define and orchestrate multiple interconnected services using a single YAML configuration file. This configuration file specifies all the necessary services, networks, and volumes required for the application, enabling developers to manage complex setups with ease. With commands like `docker-compose up` and `docker-compose down`, Docker Compose can start and stop all defined services simultaneously, streamlining the development and deployment processes.

The main advantages of Docker Compose include simplifying multi-container management, ensuring reproducibility across different environments, and facilitating easier configuration and scaling of services. By encapsulating the application's infrastructure in a versioned configuration file, Docker Compose makes it straightforward to share and maintain consistent environments between development, testing, and production. This tool is especially valuable for applications that require multiple services, such as a web server, database, and caching layer, as it allows for seamless service orchestration and management.

Service

In Docker Compose, a service represents a single container configuration within a multi-container application. Each service is defined in the docker-compose.yml file and specifies how the container should be built, run, and interact with other services. Services can be thought of as the building blocks of your application, each responsible for a specific function, such as running a web server, database, or any other component needed for the application to function.

Network

In Docker Compose, networks facilitate communication between services. Each service in a Compose file can be connected to one or more networks, allowing them to communicate seamlessly and securely. Docker Compose automatically creates a default network for the application, but custom networks can be defined to provide more control over the connectivity and isolation of services.

Volume

In Docker Compose Volumes provide a way to store data outside of the container's filesystem, ensuring that the data remains intact even if the container is stopped, removed, or recreated. Volumes are particularly useful for maintaining database data, configuration files, logs, and other persistent data that should not be lost when containers are updated or restarted, a volume is a mechanism to persist data generated or used by Docker containers.

Port Mapping in service

Port mapping in Docker Compose is a mechanism that links a port on the host machine to a port on a container, enabling access to services running inside containers from outside the Docker network. Defined in the `docker-compose.yml` file using the `ports` directive, port mapping is essential for making containerized applications accessible, such as accessing a web server or database running inside a container. This process ensures that services are reachable from the host or external networks, enhances security by controlling which ports are exposed, and helps avoid port conflicts, allowing multiple services to run concurrently on different ports of the same host.

```yaml
version: '3.8'

services:
  imagecapdb:
    container_name: database
    image: mysql
    ports:
      - "3307:3306"
    environment:
      MYSQL_ROOT_PASSWORD: Rishi@2001
      MYSQL_DATABASE: imagecaptioning
    volumes:
      - imagecapdb_data:/var/lib/mysql
    networks:
      - ic-network
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      retries: 5
      timeout: 60s

  model-service:
    container_name: model
    image: rishithaiiitb/bemodel
    ports:
      - "8060:8060"
    networks:
      - ic-network
    restart: on-failure:1

  imagecap-service:
    container_name: backend
    image: rishithaiiitb/backend
    ports:
      - "8020:8020"
    environment:
      SPRING_DATASOURCE_URL: jdbc:mysql://imagecapdb:3306/imagecaptioning?createDatabaseIfNotExist=true
      SPRING_DATASOURCE_USERNAME: root
      SPRING_DATASOURCE_PASSWORD: Rishi@2001
      SPRING_DATASOURCE_DRIVER_CLASS_NAME: com.mysql.cj.jdbc.Driver
      SPRING_JPA_HIBERNATE_DDL_AUTO: update
      SPRING_JPA_SHOW_SQL: true
      MODEL_SERVICE_URL: http://model-service:8060/predict/
```

```
    container_name: backend
    image: rishithaiiitb/backend
    ports:
      - "8020:8020"
    environment:
      SPRING_DATASOURCE_URL: jdbc:mysql://imagecapdb:3306/imagecaptioning?createDatabaseIfNotExist=true
      SPRING_DATASOURCE_USERNAME: root
      SPRING_DATASOURCE_PASSWORD: Rishi@2001
      SPRING_DATASOURCE_DRIVER_CLASS_NAME: com.mysql.cj.jdbc.Driver
      SPRING_JPA_HIBERNATE_DDL_AUTO: update
      SPRING_JPA_SHOW_SQL: true
      MODEL_SERVICE_URL: http://model-service:8060/predict/
    volumes:
      - logs:/logs
    depends_on:
      imagecapdb:
        condition: service_healthy
      model-service:
        condition: service_started
    networks:
      - ic-network
    restart: on-failure:1

  imagecapapp:
    container_name: frontend
    image: rishithaiiitb/frontend
    ports:
      - "3000:3000"
    depends_on:
      imagecap-service:
        condition: service_started
    networks:
      - ic-network

networks:
  ic-network:

volumes:
  imagecapdb_data:
  logs:
```

Services

1. `imagecapdb` (MySQL Database):
    - Container Name: `database`
    - Image: `mysql`
    - Ports: Maps host port `3307` to container port `3306`.
    - Environment Variables: Sets the root password and database name.
    - Volumes: Persists data using a named volume `imagecapdb_data`.
    - Networks: Part of the custom network `ic-network`.
    - Healthcheck: Checks the database's health with retries and timeout settings to ensure it is ready before dependent services start.
2. `model-service` (Model Service):
    - Container Name: `model`

- Image: `rishithaiiitb/bemodel`
- Ports: Exposes port `8060`.
- Networks: Part of the `ic-network`.
- Restart Policy: Restarts on failure up to one time.

3. `imagecap-service` (Backend Service):
    - Container Name: `backend`
    - Image: `rishithaiiitb/backend`
    - Ports: Exposes port `8020`.
    - Environment Variables: Configures database connection and other settings.
    - Volumes: Uses the `logs` volume for storing logs.
    - Depends On: Waits for the `imagecapdb` service to be healthy and the `model-service` to start before starting.
    - Networks: Part of the `ic-network`.
    - Restart Policy: Restarts on failure up to one time.
4. `imagecapapp` (Frontend Application):
    - Container Name: `frontend`
    - Image: `rishithaiiitb/frontend`
    - Ports: Exposes port `3000`.
    - Depends On: Waits for the `imagecap-service` to start before starting.
    - Networks: Part of the `ic-network`.

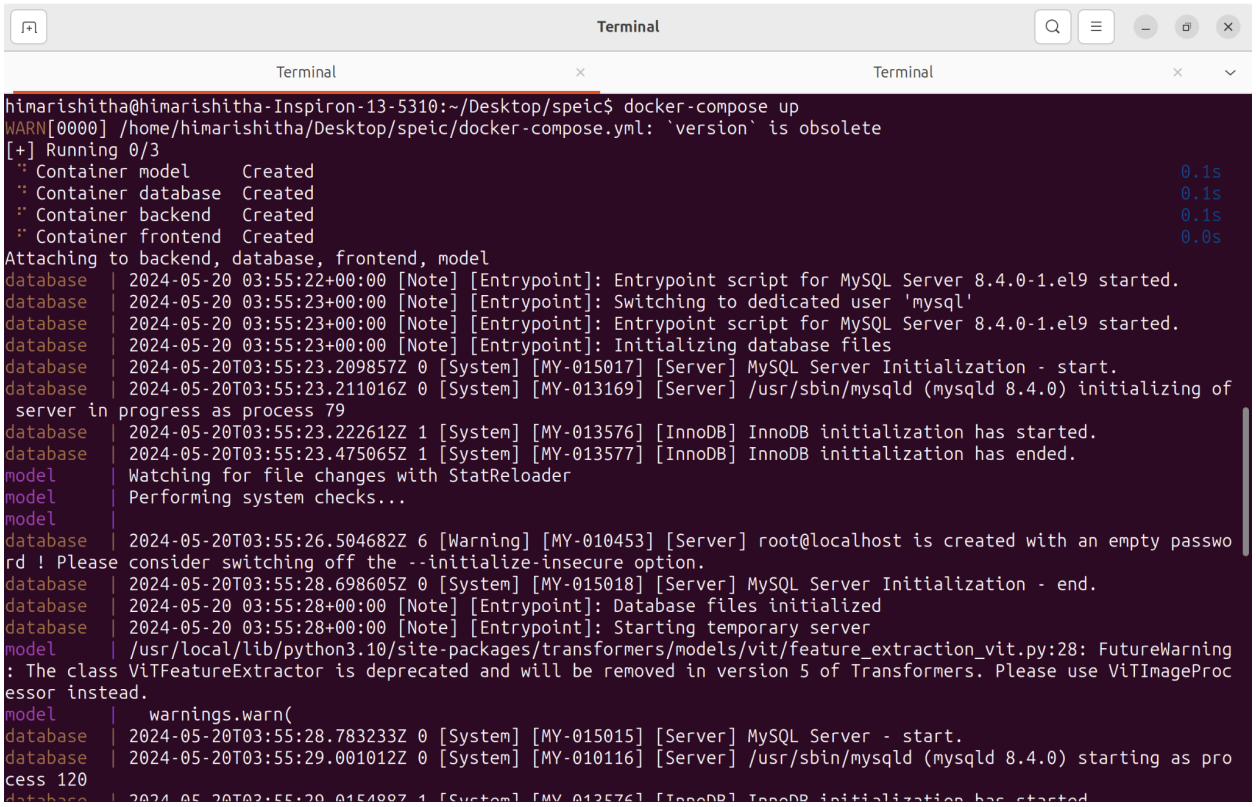
Networks and Volumes

- Networks: A custom network `ic-network` is defined to enable communication between the services.
- Volumes: Two named volumes are defined:
    - imagecapdb_data for persisting database data.
    - `logs` for storing backend logs.

This configuration ensures that all services can communicate effectively, dependencies are respected, and data is persisted as needed. The health check and restart policies add robustness, helping to maintain the application's stability.

We can start our docker-compose by by setting the present working directory same as the directory containing the docker-compose.yml.
We can run the following command to start docker compose:

$ docker-compose up



$ docker-compose down

The `docker-compose down` command is used to stop and remove the containers, networks, volumes, and images created by `docker-compose up`. This command essentially tears down the Docker environment specified in your `docker-compose.yml` file.

```
himarishitha@himarishitha-Inspiron-13-5310:~/Desktop/speic$ docker-compose down
WARN[0000] /home/himarishitha/Desktop/speic/docker-compose.yml: `version` is obsolete
[+] Running 5/5
 ✔ Container frontend          Removed
 ✔ Container backend           Removed
 ✔ Container database          Removed
 ✔ Container model             Removed
 ✔ Network speic_ic-network    Removed
himarishitha@himarishitha-Inspiron-13-5310:~/Desktop/speic$
```

The volumes attached to the containers can be accessed even after stopping them.
We can follow the below steps/commands to access the volumes:

1. docker volume ls
2. docker inspect <volume_name> (find out the mount point)
3. sudo ls /var/lib/docker/volumes/<volume_name>/_data (get the list of files inside volume)
4. sudo cat /var/lib/docker/volumes/<volume_name>/*_data*/<file_name> (access the files)

Logs created in backend will be stored in volume.

```
himarishitha@himarishitha-Inspiron-13-5310:~$ docker volume ls
DRIVER    VOLUME NAME
local     ic_imagecapdb_data
local     ic_logs
local     speic_imagecapdb_data
local     speic_logs
himarishitha@himarishitha-Inspiron-13-5310:~$ docker inspect volume ic_logs
[
    {
        "CreatedAt": "2024-05-19T17:38:41+05:30",
        "Driver": "local",
        "Labels": {
            "com.docker.compose.project": "ic",
            "com.docker.compose.version": "2.26.0",
            "com.docker.compose.volume": "logs"
        },
        "Mountpoint": "/var/lib/docker/volumes/ic_logs/_data",
        "Name": "ic_logs",
        "Options": null,
        "Scope": "local"
    }
]
```

```
himarishitha@himarishitha-Inspiron-13-5310:~$ sudo ls /var/lib/docker/volumes/ic_logs/_data
imagecap.log
himarishitha@himarishitha-Inspiron-13-5310:~$ sudo cat  /var/lib/docker/volumes/ic_logs/_data/imagecap.log
2024-05-19 12:09:14.198 [main] INFO  com.spe.imagecaptioning.ImageCaptioningApplication - Starting ImageCaptioningAppli
cation v0.0.1-SNAPSHOT using Java 17.0.2 with PID 1 (/imagecaptioning-0.0.1-SNAPSHOT.jar started by root in /)
2024-05-19 12:09:14.217 [main] INFO  com.spe.imagecaptioning.ImageCaptioningApplication - No active profile set, fallin
g back to 1 default profile: "default"
2024-05-19 12:09:19.751 [main] INFO  com.spe.imagecaptioning.ImageCaptioningApplication - Started ImageCaptioningApplic
ation in 6.077 seconds (process running for 7.303)
2024-05-19 12:09:19.755 [main] INFO  com.spe.imagecaptioning.Controller.UserController - Image captioning application s
tarted
2024-05-19 12:18:30.078 [http-nio-8020-exec-2] INFO  com.spe.imagecaptioning.Controller.UserController - Register end-p
oint called
2024-05-19 12:18:30.079 [http-nio-8020-exec-2] INFO  com.spe.imagecaptioning.Service.UserServiceImpl - New user created
2024-05-19 12:18:30.122 [http-nio-8020-exec-2] INFO  com.spe.imagecaptioning.Controller.UserController - Register respo
nse sent successfully
2024-05-19 12:18:36.643 [http-nio-8020-exec-4] INFO  com.spe.imagecaptioning.Controller.UserController - Register end-p
oint called
2024-05-19 12:18:36.643 [http-nio-8020-exec-4] ERROR com.spe.imagecaptioning.Service.UserServiceImpl - User Details can
not be null
2024-05-19 12:18:36.643 [http-nio-8020-exec-4] INFO  com.spe.imagecaptioning.Controller.UserController - Register respo
nse sent successfully
2024-05-19 12:18:49.175 [http-nio-8020-exec-5] INFO  com.spe.imagecaptioning.Controller.UserController - Login end-poin
t called
2024-05-19 12:18:49.262 [http-nio-8020-exec-5] INFO  com.spe.imagecaptioning.Service.UserServiceImpl - User found
2024-05-19 12:18:49.336 [http-nio-8020-exec-5] INFO  com.spe.imagecaptioning.Service.UserServiceImpl - Logged in succes
sfully
2024-05-19 12:18:49.336 [http-nio-8020-exec-5] INFO  com.spe.imagecaptioning.Controller.UserController - Login response
 sent successfully
2024-05-19 12:18:55.205 [http-nio-8020-exec-6] INFO  com.spe.imagecaptioning.Controller.UserController - Login end-poin
t called
2024-05-19 12:18:55.211 [http-nio-8020-exec-6] ERROR com.spe.imagecaptioning.Service.UserServiceImpl - No user found wi
```

Continuous Deployment:

Ansible is an open-source automation tool used for configuration management, application deployment, task automation, and IT orchesctration. It allows you to define the desired state of your systems using simple, human-readable YAML files called playbooks. Ansible operates without requiring a dedicated server or agent software on the managed nodes, using SSH for communication, which simplifies setup and maintenance. This agentless nature makes Ansible lightweight and easy to use, facilitating seamless integration into existing environments.

Ansible is crucial for modern IT operations because it enables consistent and repeatable configurations across various environments, reducing the risk of human error and ensuring uniformity. By automating routine tasks and deployments, Ansible saves time and effort, allowing IT teams to focus on more strategic initiatives. Its modular design and vast ecosystem of built-in modules support a wide range of platforms and applications, making it a versatile tool for managing complex infrastructure and streamlining DevOps workflows.

1. Ansible inventory

An Ansible inventory file is a configuration file that specifies the hosts and groups of hosts that Ansible will manage. It provides crucial details such as IP addresses, domain names, and connection information for target systems. By organizing hosts into groups, the inventory file allows for efficient and targeted execution of Ansible commands, modules, and playbooks. This structured approach simplifies the management of multiple hosts, enhances scalability, and ensures that automation tasks can be directed at specific subsets of the infrastructure as needed. The inventory file supports various formats, with INI being the most common, enabling easy organization and variable management for different host groups

1. Group Name `[host1]`:
    - This line indicates the start of a group named `host1`. You can define multiple hosts under this group. Groups are useful for organizing hosts into categories, allowing you to run tasks on all hosts within a group simultaneously.
2. Host Entry `localhost`:

- `localhost` is the alias for the host being configured. This can be any name you choose to reference this host within your playbooks.

3. Host Variables:
    - `ansible_host=127.0.0.1`: Specifies the IP address or hostname of the target machine. Here, `127.0.0.1` refers to the local machine.
    - `ansible_user=himarishitha`: Specifies the username to log into the host.
    - `ansible_ssh_pass=Rishi@2001`: Specifies the SSH password for the user. Note: Using plaintext passwords is not recommended for security reasons; consider using Ansible Vault or SSH keys.
    - `ansible_become_pass=Rishi@2001`: Specifies the password for privilege escalation (sudo). Again, consider using more secure methods for handling passwords.
    - `ansible_ssh_common_args='-o StrictHostKeyChecking=no'`: Provides additional SSH arguments. Here, it disables strict host key checking, which can be useful in environments where host keys change frequently or are not pre-configured.

2. Ansible playbook

An Ansible playbook is a YAML-based configuration file that defines a series of tasks to be executed on managed hosts, specifying the desired state and actions to be taken on those hosts. Playbooks enable automation of complex IT tasks, such as software installation, configuration management, and orchestration across multiple systems. They are essential for ensuring consistency and repeatability in IT operations, reducing the risk of human error, and saving time by automating routine processes. By using playbooks, IT teams can efficiently manage and deploy applications and infrastructure, streamlining DevOps workflows and improving overall operational efficiency.

```yaml
---
- name: Deploy Docker Compose Stack
  hosts: all
  become: true

  tasks:
    - name: Install Docker
      apt:
        name: docker.io
        state: present
        update_cache: yes

    - name: Install Docker Compose
      apt:
        name: docker-compose
        state: present

    - name: Cleanup existing Docker containers
      shell: |
        docker rm -f database backend model frontend || true

    - name: Cleanup specific Docker images
      shell: |
        docker images -q rishithaiiitb/bemodel rishithaiiitb/backend rishithaiiitb/frontend | xargs -r docker rmi -f || true

    - name: Prune unused Docker images
      shell: docker image prune -f

    - name: Check if /ic directory exists
      stat:
        path: /ic
      register: ic_directory

    - name: Create Docker Compose working directory if it does not exist
      file:
        path: /ic
        state: directory
        owner: root
        group: root
        mode: '0755'
      when: ic_directory.stat.exists == False



    - name: Copy Docker Compose file to host
      copy:
        src: /var/lib/jenkins/workspace/ImageCaptioning/docker-compose.yml
        dest: /ic/docker-compose.yml

    - name: Pull Model Service Image
      docker_image:
        name: rishithaiiitb/bemodel
        source: pull

    - name: Pull Backend Image
      docker_image:
        name: rishithaiiitb/backend
        source: pull

    - name: Pull Frontend Image
      docker_image:
        name: rishithaiiitb/frontend
        source: pull

    - name: Start Docker Compose Stack
      command: docker-compose up -d
      args:
        chdir: /ic/
```

Steps followed in playbook:

1. Install Docker:
    - Ensures that Docker is installed on the target machine. It uses the `apt` module to install `docker.io` and updates the package cache.
2. Install Docker Compose:

- Installs Docker Compose using the `apt` module to ensure the required version is present.

3. Cleanup Existing Docker Containers:
    - Removes any existing Docker containers named `database`, `backend`, `model`, and `frontend` to avoid conflicts. The `|| true` part ensures the playbook continues even if the containers don't exist.
    - This part is done in ansible playbook instead of jenkins file to ensure that this step is executed in the managed hosts not in the host running jenkins.
4. Cleanup Specific Docker Images:
    - Removes specific Docker images (`rishithaiiitb/bemodel`, `rishithaiiitb/backend`, and `rishithaiiitb/frontend`) to ensure fresh copies are pulled. The `|| true` part allows the task to continue if no images match the criteria.
    - This part is done in ansible playbook instead of jenkins file to ensure that this step is executed in the managed hosts not in the host running jenkins.
5. Prune Unused Docker Images:
    - Runs the Docker command to prune unused images, freeing up space and ensuring the system is clean.
    - If we just use docker rmi to remove the image it is just untagging the images, not removing them completely from the machine. Therefore, to remove the untagged images we are using this step.
6. Check if /ic Directory Exists:
    - Checks if the `/ic` directory exists on the host machine and registers the result.
7. Create Docker Compose Working Directory if it Does Not Exist:
    - Creates the `/ic` directory if it doesn't already exist, ensuring the required directory structure is in place for the Docker Compose setup.
8. Copy Docker Compose File to Host:
    - Copies the `docker-compose.yml` file from the Jenkins workspace to the `/ic` directory on the host machine. This ensures that the correct configuration is used for the Docker Compose setup.
9. Pull Model Service Image:
    - Pulls the latest version of the model service Docker image (`rishithaiiitb/bemodel`) to ensure the container runs with the most recent updates.
10. Pull Backend Image:
    - Pulls the latest version of the backend service Docker image (`rishithaiiitb/backend`) to ensure the container runs with the most recent updates.

11. Pull Frontend Image:
    - Pulls the latest version of the frontend service Docker image (`rishithaiiitb/frontend`) to ensure the container runs with the most recent updates.
12. Start Docker Compose Stack:
    - Executes the `docker-compose up -d` command in the `/ic` directory, starting the entire stack of services in detached mode, ensuring the application runs in the background.

```
himarishitha@himarishitha-Inspiron-13-5310:~$ docker ps
CONTAINER ID   IMAGE                    COMMAND                  CREATED         STATUS                   PORTS
                                          NAMES
c351fd921392   rishithaiiitb/frontend   "docker-entrypoint.s…"   3 minutes ago   Up 3 minutes             0.0.0.0:3000-
>3000/tcp, :::3000->3000/tcp               frontend
759daa7aae8f   rishithaiiitb/backend    "java -jar imagecapt…"   3 minutes ago   Up 3 minutes             0.0.0.0:8020-
>8020/tcp, :::8020->8020/tcp               backend
b20f67bbe58b   mysql                    "docker-entrypoint.s…"   3 minutes ago   Up 3 minutes (healthy)   33060/tcp, 0.
0.0.0:3307->3306/tcp, :::3307->3306/tcp    database
c843e6ea60bb   rishithaiiitb/bemodel    "python manage.py ru…"   3 minutes ago   Up 3 minutes             0.0.0.0:8060-
>8060/tcp, :::8060->8060/tcp               model
himarishitha@himarishitha-Inspiron-13-5310:~$
```

Monitoring and Visualizing Log file using ELK Stack:

In our application, which integrates a Django server, React frontend, Java Spring Boot backend, and Docker Compose, effective monitoring and visualization of log files are crucial. The ELK Stack (Elasticsearch, Logstash, and Kibana) serves this purpose efficiently.

Elasticsearch indexes and stores logs generated by the Spring Boot backend, and Dockerized services. Logstash processes these logs, extracting relevant information such as error messages, performance metrics, and user activities, and sends them to Elasticsearch. Kibana provides a user-friendly interface to visualize this data, allowing us to create dashboards that display real-time insights into application performance, user behavior, and system health.

By implementing the ELK Stack, we can monitor our application more effectively, quickly identify and resolve issues, and ensure optimal performance. This integration enhances our DevOps practices, enabling continuous monitoring and proactive maintenance of our system

Steps to visualize logs:

1. Install elastic search and kibana
    a. First, we are going to copy the public GPG key of elastic search using the following command:

    curl -fsSL https://artifacts.elastic.co/GPG-KEY-elasticsearch |sudo gpg --dearmor -o /usr/share/keyrings/elastic.gpg

    b. Then we add the source of elastic search so that apt will use it to install it:

    echo "deb [signed-by=/usr/share/keyrings/elastic.gpg] https://artifacts.elastic.co/packages/7.x/apt stable main" | sudo tee -a /etc/apt/sources.list.d/elastic-7.x.list

    c. Then we perform apt update and install elastic search using the following command:

    sudo apt update

    sudo apt install elasticsearch

    d. Once you have installed elastic search, you can run it with the following command:

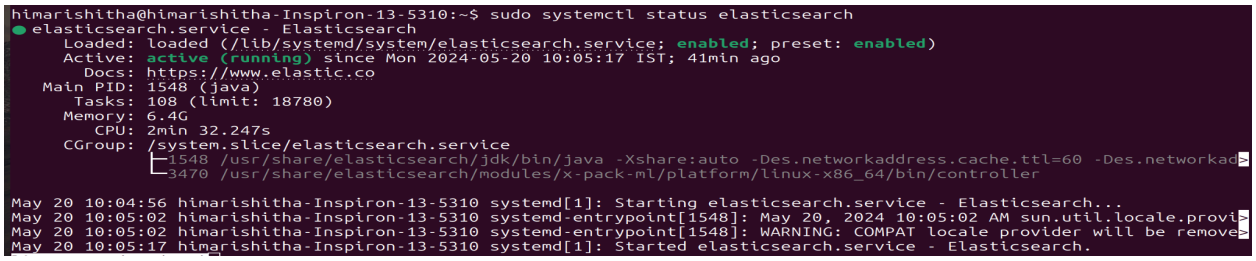
    sudo systemctl start elasticsearch

    e. In case you want elastic search to run on system start up itself, then run the following command:

    sudo systemctl enable elasticsearch

    f. We can also check the status of elastic search to see if it is running or not using the following command:

    sudo systemctl status elasticsearch



g. Now that we have install elastic search, we can install kibana by running:

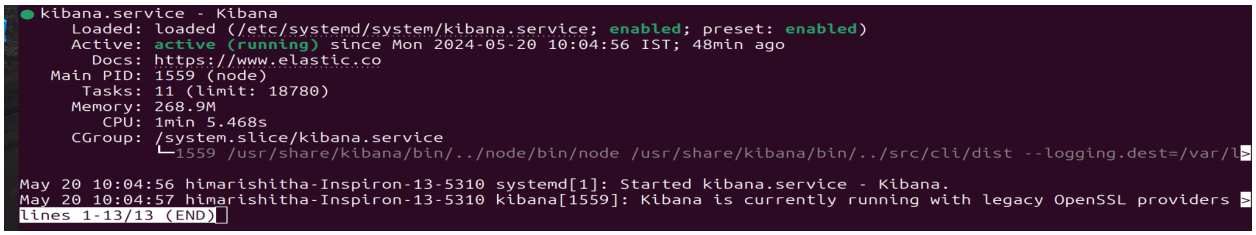
   sudo apt install kibana

h. Again, to start kibana we can use the following command:

   sudo systemctl start kibana

i. And to check if it is running or not, we will use the following command

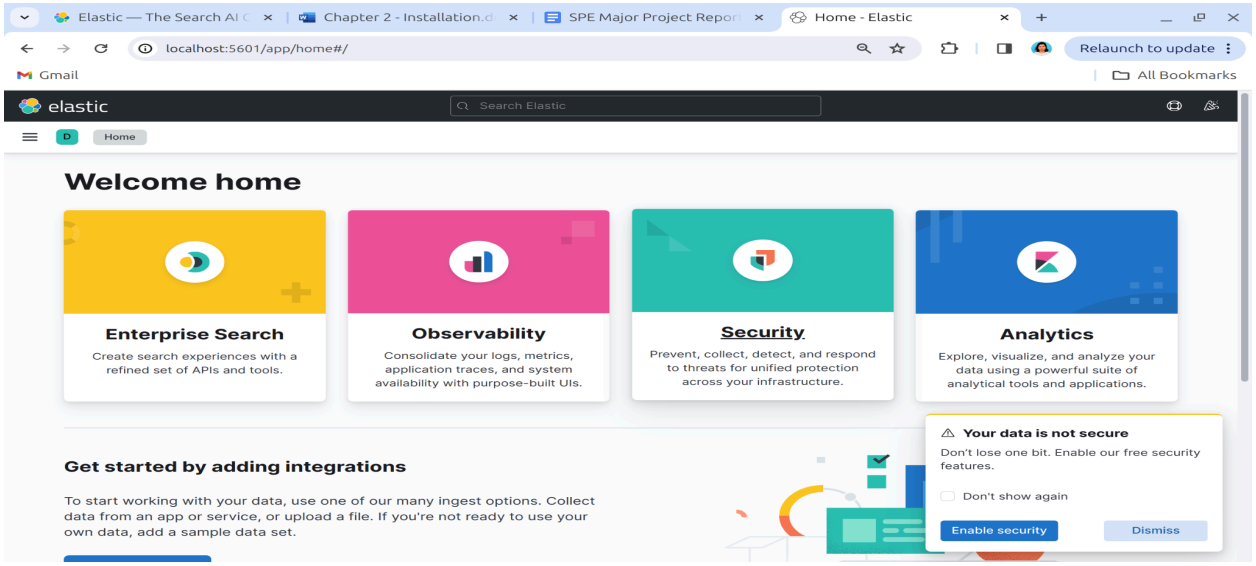
   sudo systemctl status kibana



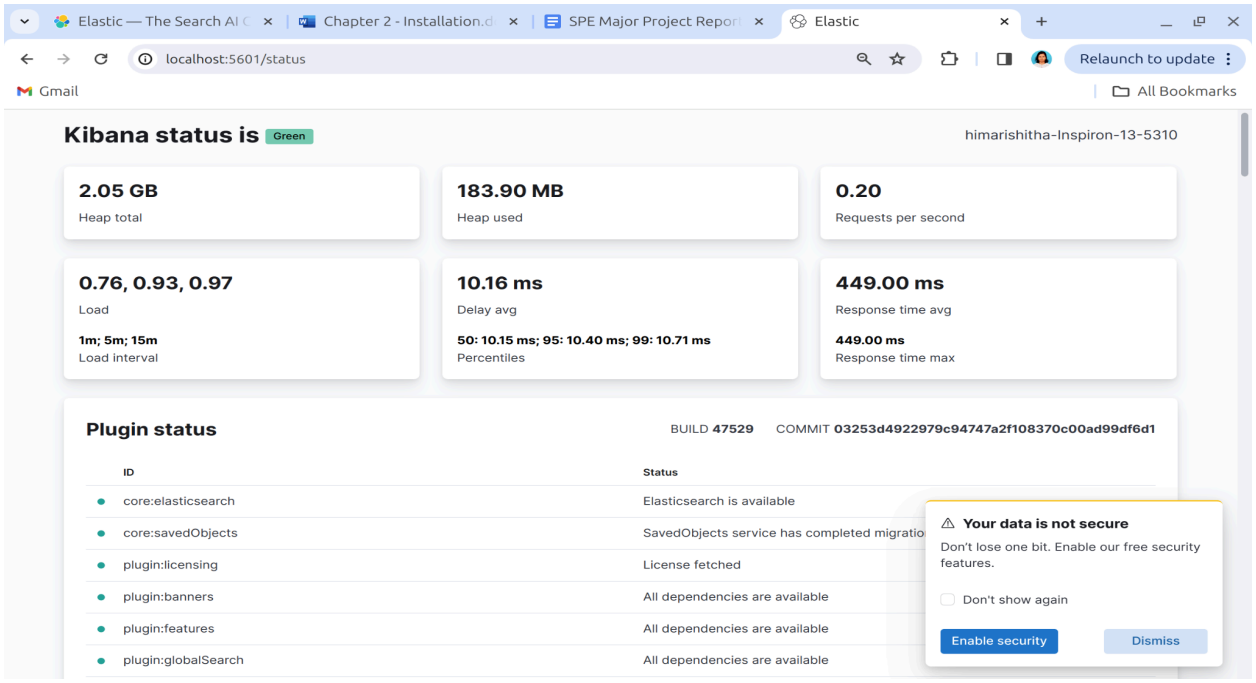
j. Once Kibana is running, we are going to setup a username and password for kibana. The user is going to be kadmin. Once you run the following command, it will prompt you for a password which you will enter as well:

   echo "kadmin:`openssl passwd -apr1`" | sudo tee -a /etc/nginx/htpasswd.users

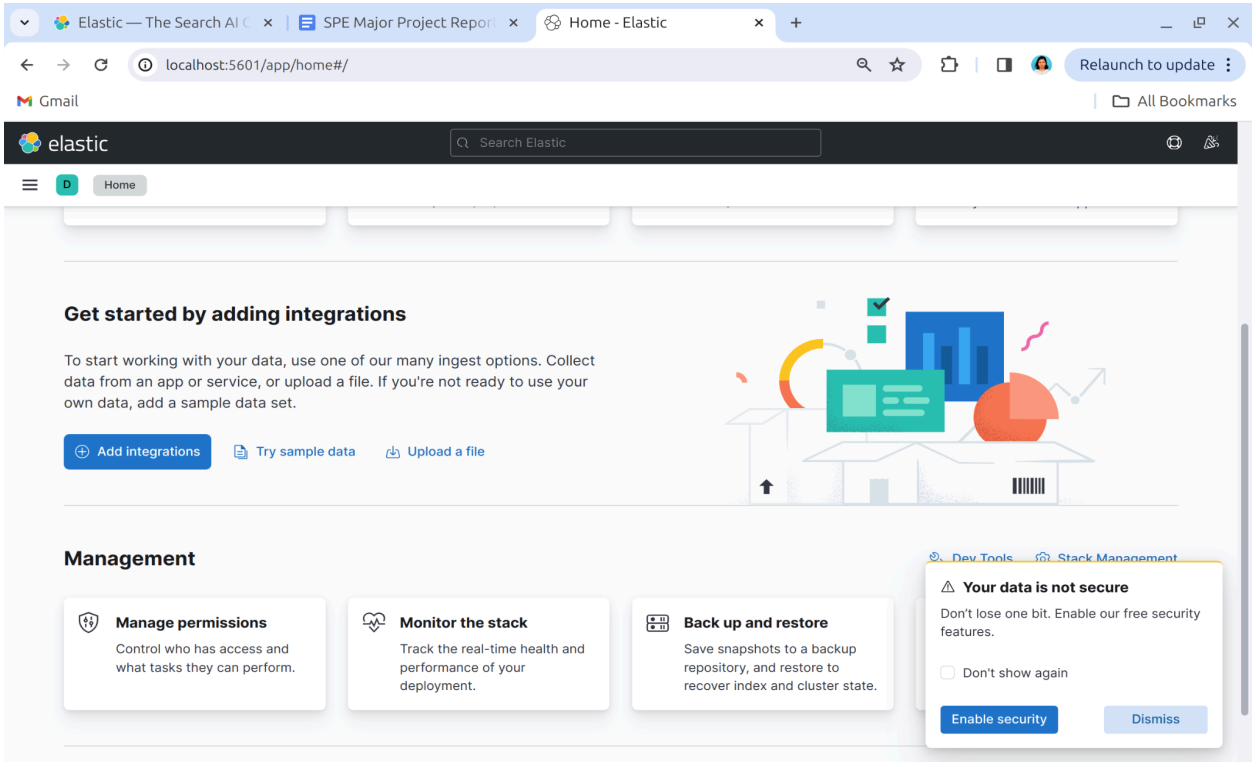
Visit http://localhost:5601/ to access the Kibana dashboard:



Visit http://localhost:5601/status to see the status of your Kibana service:

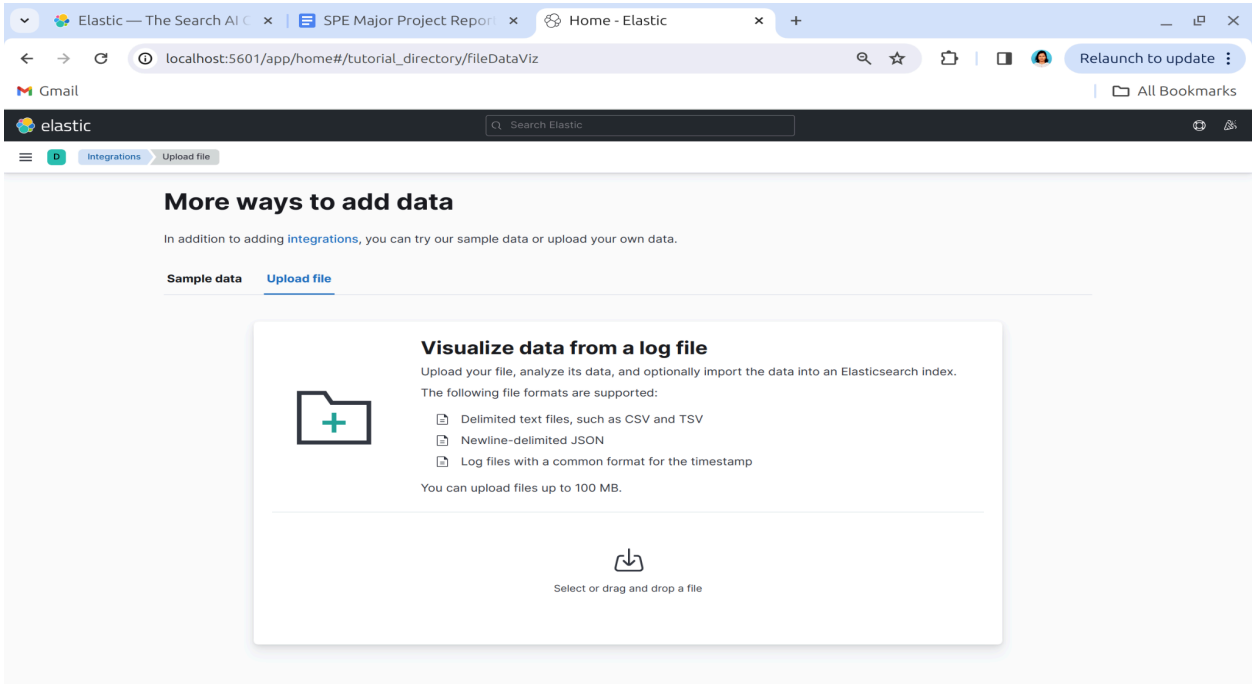


Then we can upload our log file to visualize:

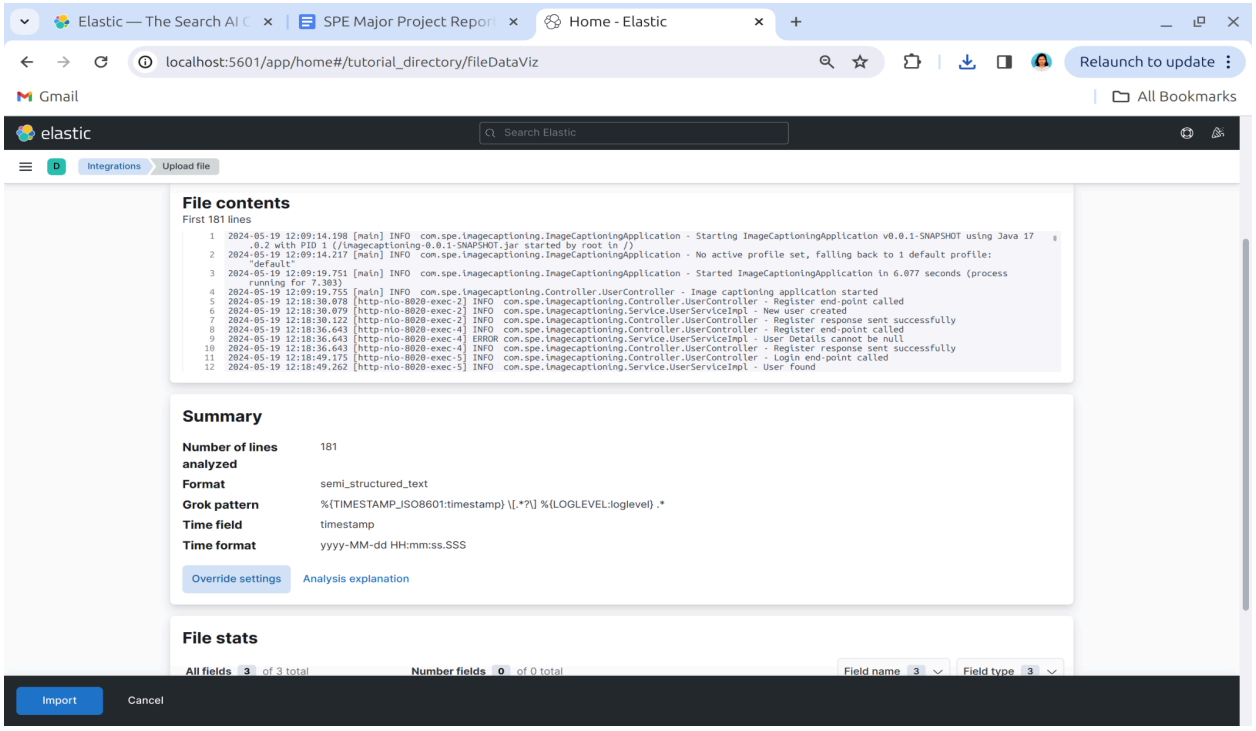


Visualization:

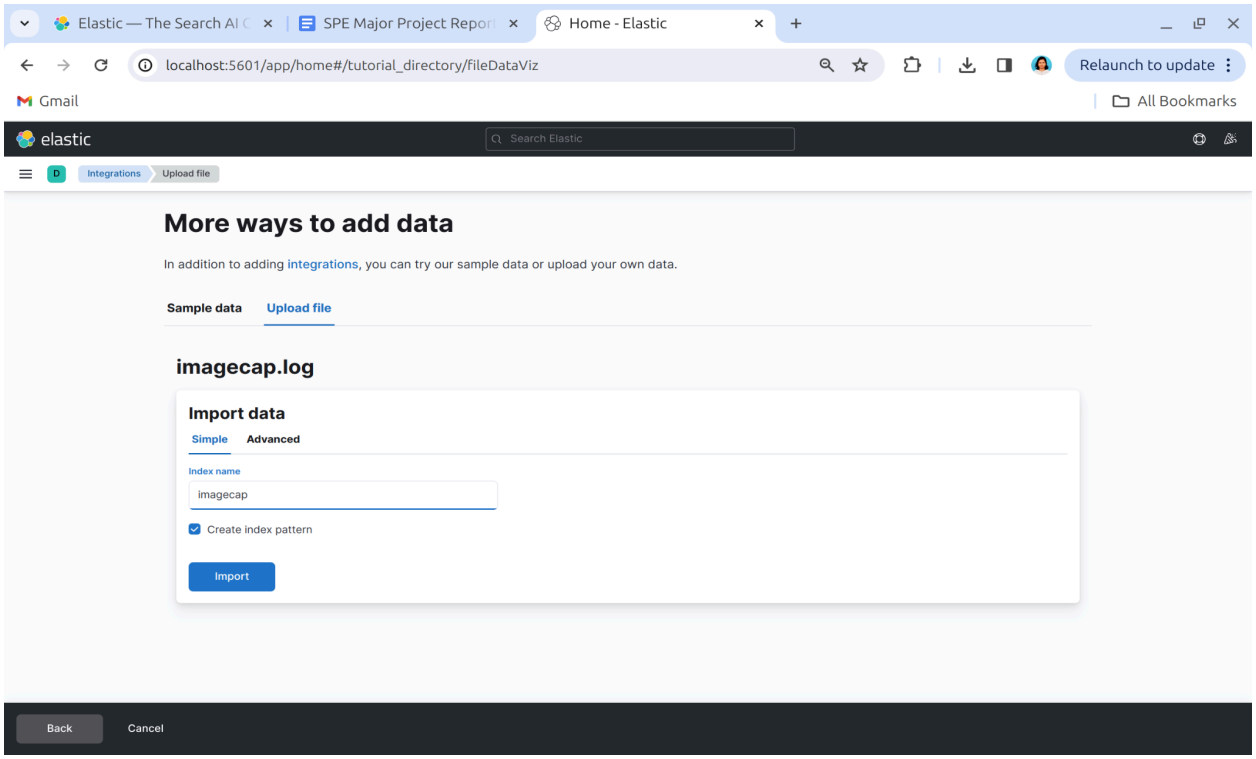
Upload your log file here:



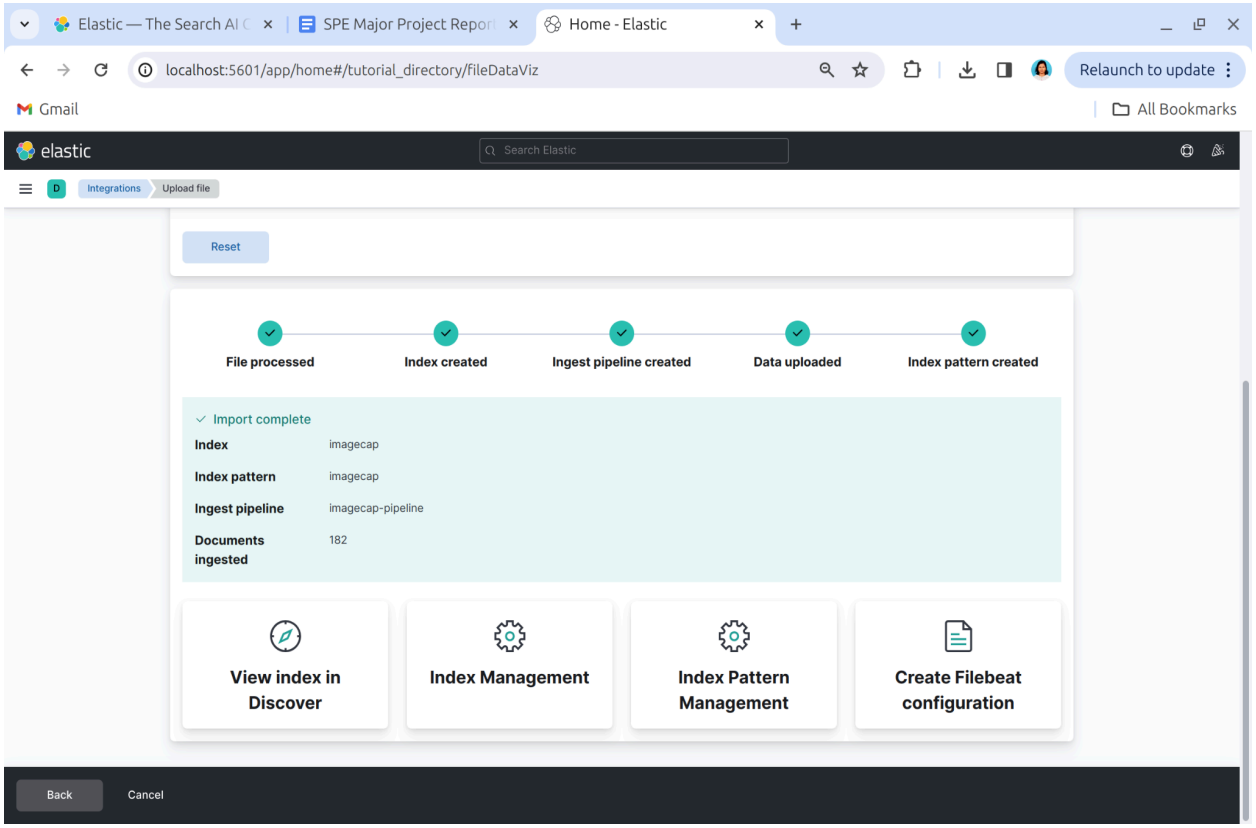
After uploading click on import to extract logs into kibana:



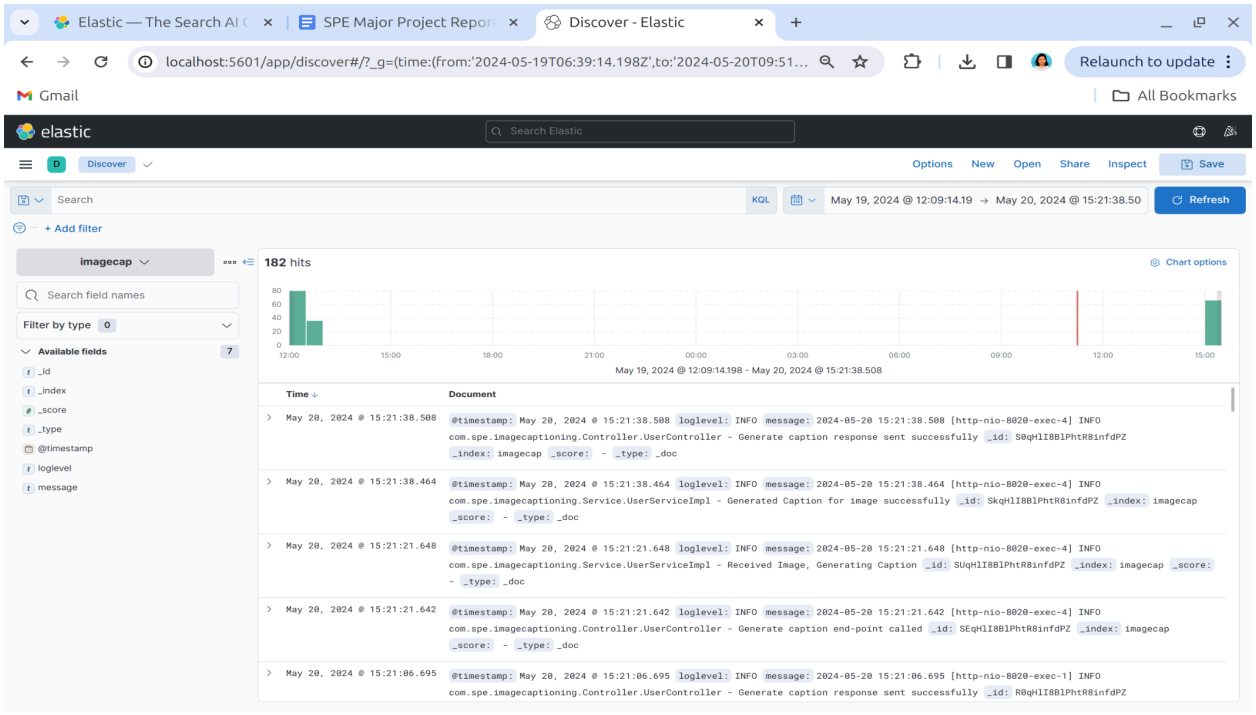
Create an index as shown below:



Then we can see our log file is processed:

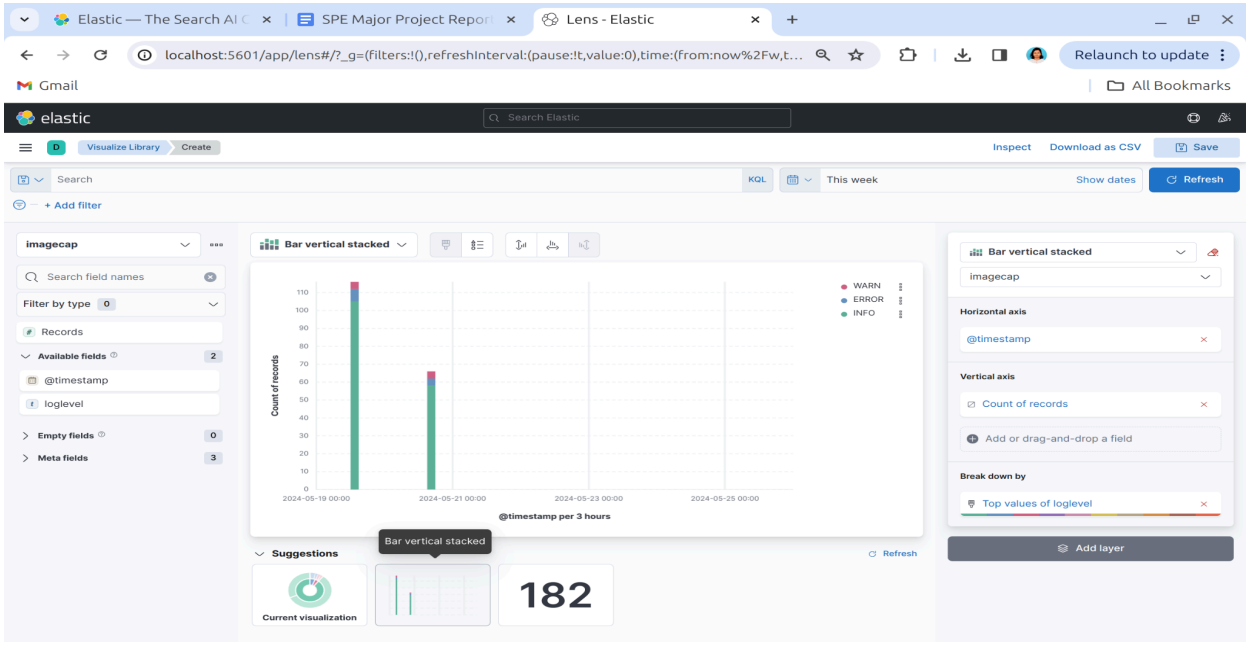


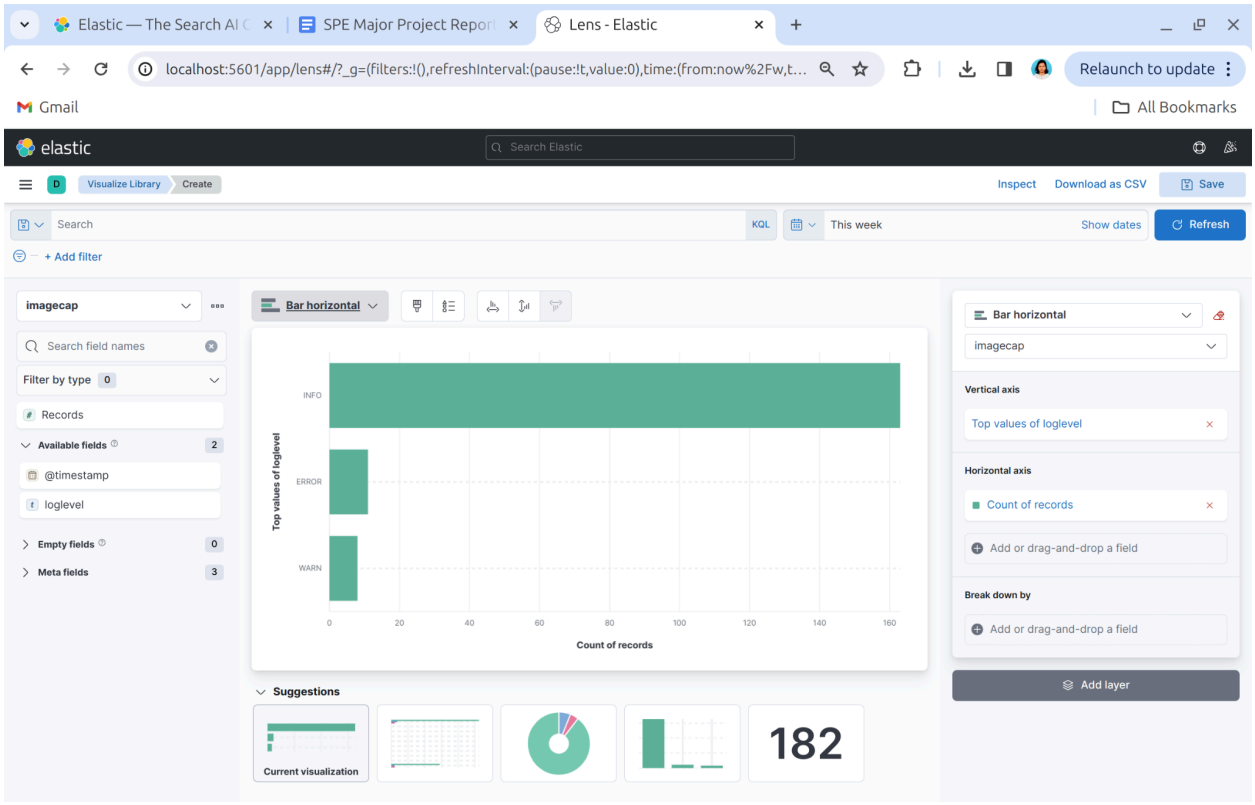
Then when we click on the view index in discover to see the count of hits and the logs as shown below.

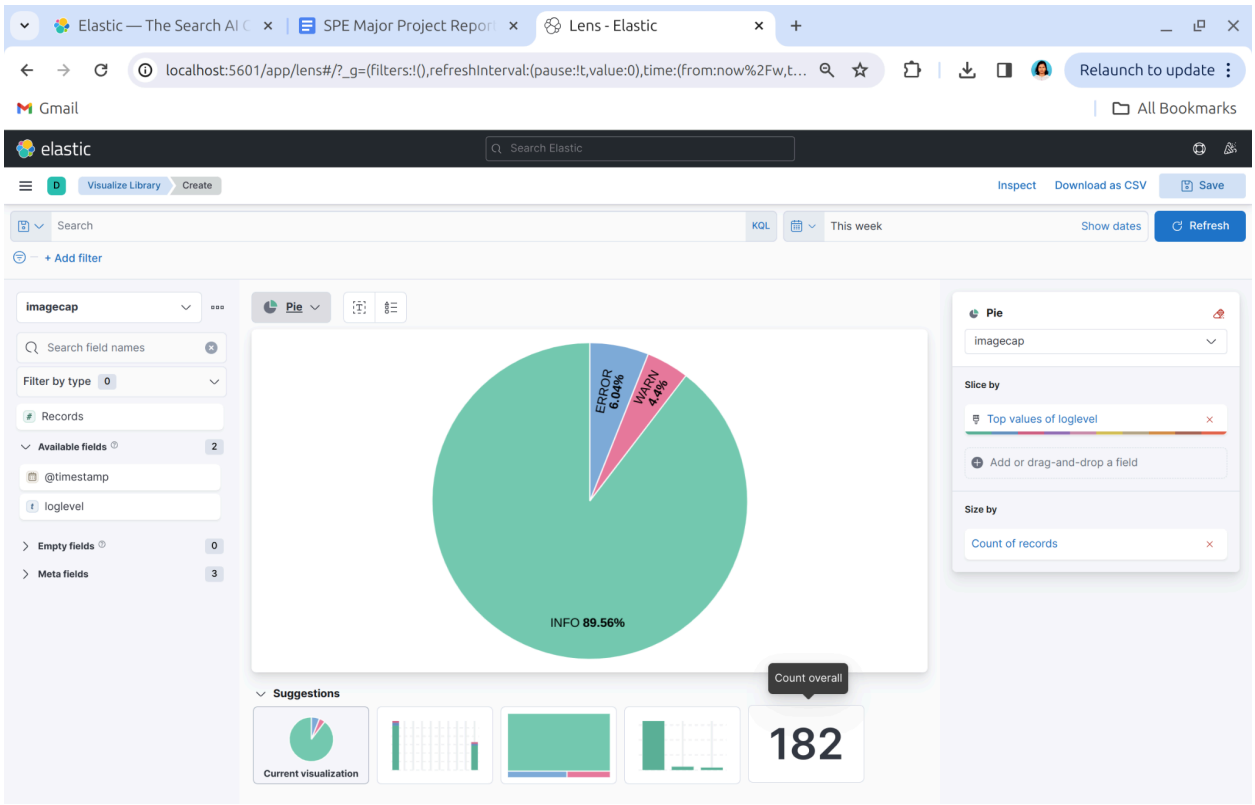


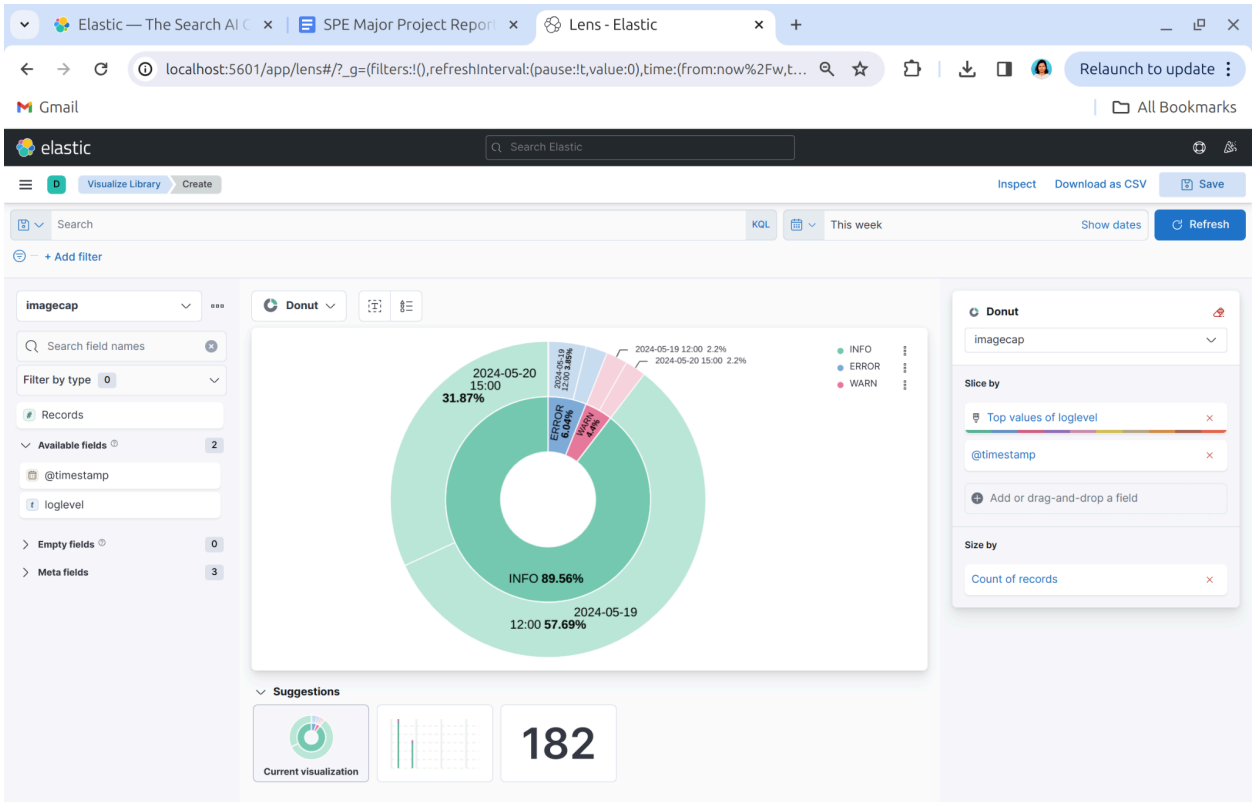
Some Visualizations:

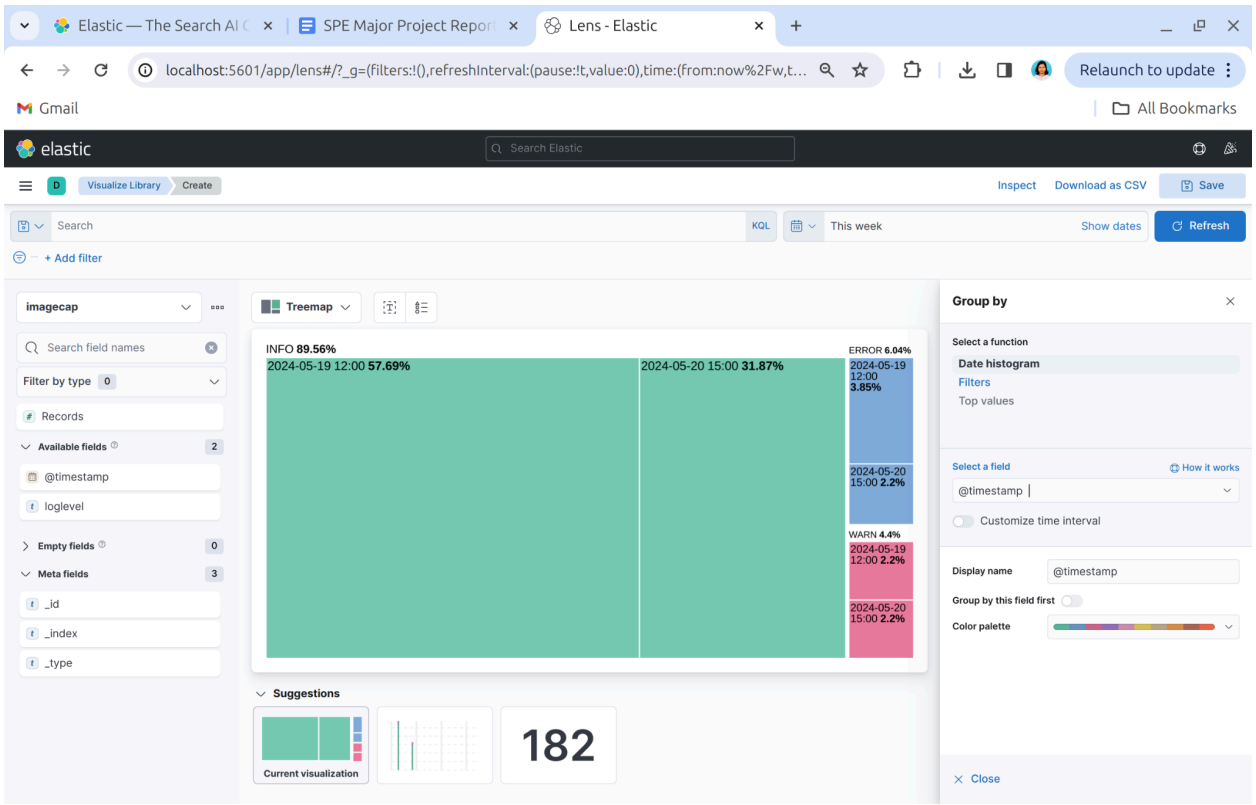
We have attached some visualizations which were created based on log levels and time stamps in our log file.

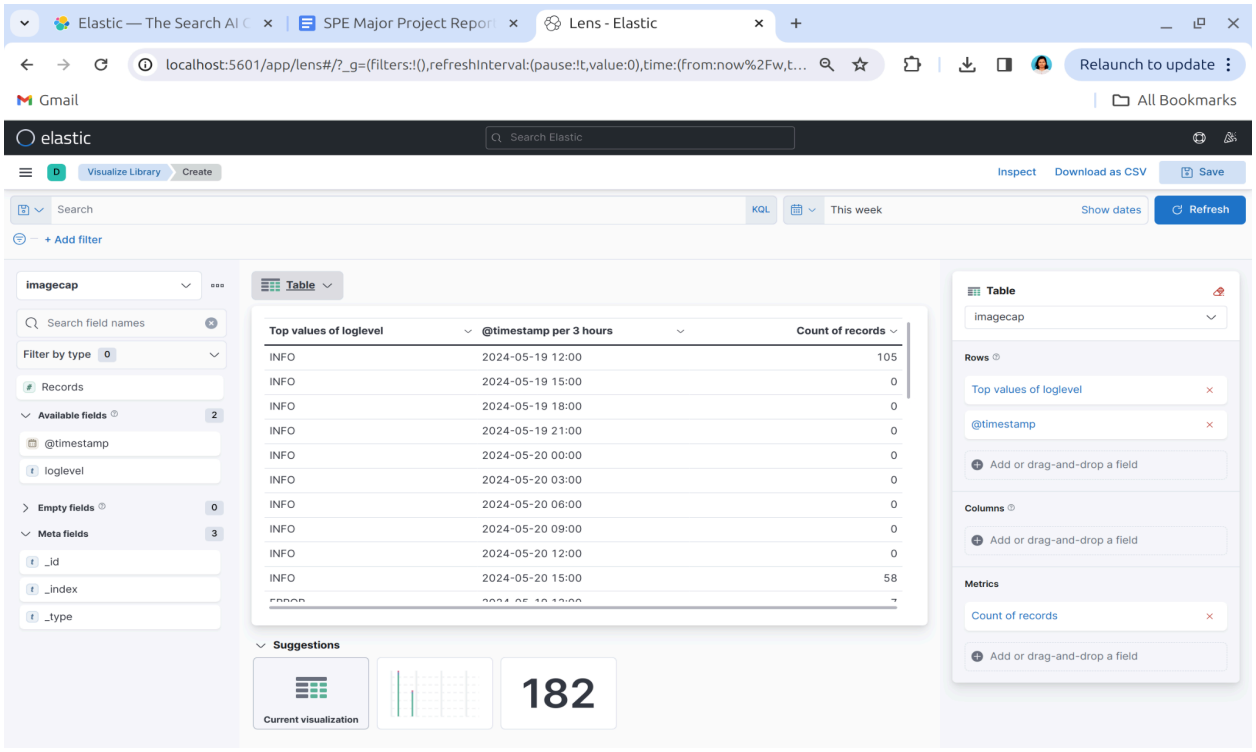


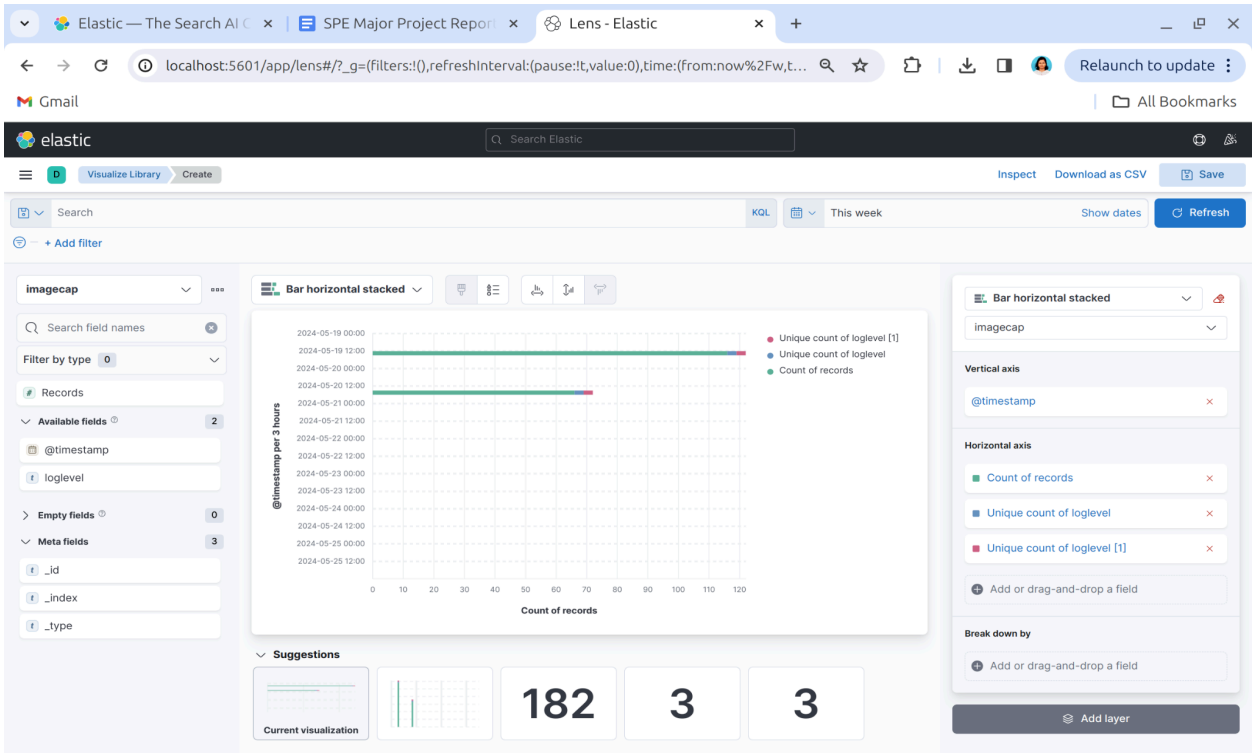

Elastic — The Search AI
SPE Major Project Report
Lens - Elastic
localhost:5601/app/lens#/?_g=(filters:!(),refreshInterval:(pause:!t,value:0),time:(from:now%2Fw,t...
Relaunch to update
Gmail
All Bookmarks
elastic
Search Elastic
Visualize Library
Create
Inspect
Download as CSV
Save
Search
KQL
This week
Show dates
Refresh
+ Add filter
imagecap
Search field names
Filter by type 0
Records
Available fields 2
@timestamp
loglevel
Empty fields 0
Meta fields 3
Bar horizontal
INFO
ERROR
WARN
Top values of loglevel
0
20
40
60
80
100
120
140
160
Count of records
Suggestions
Current visualization
182
Bar horizontal
imagecap
Vertical axis
Top values of loglevel
Horizontal axis
Count of records
Add or drag-and-drop a field
Break down by
Add or drag-and-drop a field
Add layer


Elastic — The Search AI
SPE Major Project Report
Lens - Elastic
localhost:5601/app/lens#/?_g=(filters:!(),refreshInterval:(pause:!t,value:0),time:(from:now%2Fw,t...
Relaunch to update
Gmail
All Bookmarks
elastic
Search Elastic
Visualize Library
Create
Inspect
Download as CSV
Save
Search
KQL
This week
Show dates
Refresh
+ Add filter
imagecap
Search field names
Filter by type 0
Records
Available fields 2
@timestamp
loglevel
Empty fields 0
Meta fields 3
Pie
ERROR 6.04%
WARN 4.4%
INFO 89.56%
Suggestions
Current visualization
Count overall
182
Pie
imagecap
Slice by
Top values of loglevel
Add or drag-and-drop a field
Size by
Count of records


Elastic — The Search AI
SPE Major Project Report
Lens - Elastic
localhost:5601/app/lens#/?_g=(filters:!(),refreshInterval:(pause:!t,value:0),time:(from:now%2Fw,t...
Relaunch to update
Gmail
All Bookmarks
elastic
Search Elastic
Visualize Library
Create
Inspect
Download as CSV
Save
Search
KQL
This week
Show dates
Refresh
+ Add filter
imagecap
Search field names
Filter by type 0
Records
Available fields 2
@timestamp
loglevel
Empty fields 0
Meta fields 3
Donut
2024-05-19 12:00 2.2%
2024-05-20 15:00 2.2%
2024-05-19 12:00 3.85%
2024-05-20 15:00 31.87%
ERROR 6.04%
WARN 4.4%
INFO 89.56%
2024-05-19 12:00 57.69%
INFO
ERROR
WARN
Slice by
Top values of loglevel
@timestamp
Add or drag-and-drop a field
Size by
Count of records
Suggestions
Current visualization
182


Elastic — The Search AI
SPE Major Project Report
Lens - Elastic
localhost:5601/app/lens#/?_g=(filters:!(),refreshInterval:(pause:!t,value:0),time:(from:now%2Fw,t...
Relaunch to update
Gmail
All Bookmarks
elastic
Search Elastic
Visualize Library
Create
Inspect
Download as CSV
Save
Search
KQL
This week
Show dates
Refresh
+ Add filter
imagecap
Search field names
Filter by type 0
Records
Available fields 2
@timestamp
loglevel
Empty fields 0
Meta fields 3
_id
_index
_type
Treemap
INFO 89.56%
2024-05-19 12:00 57.69%
2024-05-20 15:00 31.87%
ERROR 6.04%
2024-05-19 12:00 3.85%
2024-05-20 15:00 2.2%
WARN 4.4%
2024-05-19 12:00 2.2%
2024-05-20 15:00 2.2%
Group by
Select a function
Date histogram
Filters
Top values
Select a field
How it works
@timestamp
Customize time interval
Display name
@timestamp
Group by this field first
Color palette
Close
Suggestions
Current visualization
182


Elastic — The Search AI
SPE Major Project Report
Lens - Elastic
localhost:5601/app/lens#/?_g=(filters:!(),refreshInterval:(pause:!t,value:0),time:(from:now%2Fw,t...
Relaunch to update
Gmail
All Bookmarks
elastic
Search Elastic
Visualize Library
Create
Inspect
Download as CSV
Save
Search
KQL
This week
Show dates
Refresh
+ Add filter
imagecap
Search field names
Filter by type 0
Records
Available fields 2
@timestamp
loglevel
Empty fields 0
Meta fields 3
_id
_index
_type
Table
Top values of loglevel
@timestamp per 3 hours
Count of records
INFO 2024-05-19 12:00 105
INFO 2024-05-19 15:00 0
INFO 2024-05-19 18:00 0
INFO 2024-05-19 21:00 0
INFO 2024-05-20 00:00 0
INFO 2024-05-20 03:00 0
INFO 2024-05-20 06:00 0
INFO 2024-05-20 09:00 0
INFO 2024-05-20 12:00 0
INFO 2024-05-20 15:00 58
Suggestions
Current visualization
182
Table
imagecap
Rows
Top values of loglevel
@timestamp
Add or drag-and-drop a field
Columns
Add or drag-and-drop a field
Metrics
Count of records
Add or drag-and-drop a field


Elastic — The Search AI
SPE Major Project Report
Lens - Elastic
localhost:5601/app/lens#/?_g=(filters:!(),refreshInterval:(pause:!t,value:0),time:(from:now%2Fw,t...
Relaunch to update
Gmail
All Bookmarks
elastic
Search Elastic
Visualize Library
Create
Inspect
Download as CSV
Save
Search
KQL
This week
Show dates
Refresh
+ Add filter
imagecap
Search field names
Filter by type 0
Records
Available fields 2
@timestamp
loglevel
Empty fields 0
Meta fields 3
_id
_index
_type
Bar horizontal stacked
@timestamp per 3 hours
Count of records
Unique count of loglevel [1]
Unique count of loglevel
Count of records
Suggestions
Current visualization
182
3
3
Bar horizontal stacked
imagecap
Vertical axis
@timestamp
Horizontal axis
Count of records
Unique count of loglevel
Unique count of loglevel [1]
Add or drag-and-drop a field
Break down by
Add or drag-and-drop a field
Add layer

Conclusion:

As we conclude this image captioning project, it marks a significant achievement in integrating advanced machine learning and DevOps practices to optimize development and deployment processes. The project's foundation lies in the strategic use of a pretrained image captioning model, combined with key DevOps tools such as Git, Jenkins, Maven, Docker, and Ansible. Each of these tools has played a crucial role in creating a more efficient and collaborative workflow.

The implementation of Continuous Integration (CI) and Continuous Delivery (CD) practices, supported by Jenkins, has automated the integration, testing, and delivery of our application code. This automation not only reduces the likelihood of bugs but also ensures a streamlined and consistent release process. Git's version control system has provided a structured approach to code management, fostering collaboration among team members and maintaining a reliable version history.

By leveraging Docker, we have created containerized applications that ensure seamless deployment across various environments. Docker Compose has facilitated the management of multi-container setups, including the backend, frontend, and the pretrained image captioning model, ensuring consistent and reproducible environments for development and testing.

Ansible has enhanced the project's Infrastructure as Code (IaC) capabilities, automating the provisioning and configuration of infrastructure. This automation improves scalability and reduces the risk of manual errors, ensuring that the deployment environment is consistent and reliable.

Overall, the integration of a pretrained image captioning model within a robust DevOps pipeline has enabled us to deliver a high-quality, scalable, and efficient application. The combination of advanced machine learning and DevOps practices has paved the way for future enhancements and innovations in our software development lifecycle.

Github Repository:
https://github.com/rishithaiiitb/ImageCaptioning.git

Docker Hub:
Model:
https://hub.docker.com/repository/docker/rishithaiiitb/bemodel/general
Backend:
https://hub.docker.com/repository/docker/rishithaiiitb/backend/general
Frontend:
https://hub.docker.com/repository/docker/rishithaiiitb/frontend/general