Fair Rent Split-wise

Darpan Singh (IMT2020133)
Nandula Satya Prasanna Koushik (IMT2020096)

December 14, 2023

Abstract

This report provides a detailed overview of the *Fair Rent Split-wise* web application, which is built using Flask, HTML/CSS, and SQLAlchemy. Ansible is used for deployment, Kubernetes is used for orchestration, Jenkins is used for automation, and Docker is used for containerization. The application's features, architecture, and the tools and technologies used in its development and deployment are all described in the report.

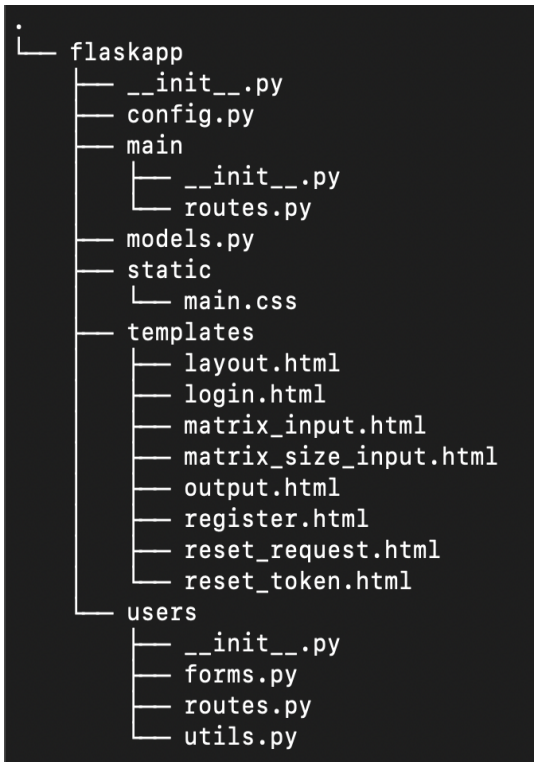
1 Web Application

1.1 Introduction

Since each room in the property is unique, renting a property and allocating the rent to the users can be a laborious task. Additionally, it is frequently unfair to divide the rent equally among the users. In order to address this type of issue, we created the *Fair Rent Split* web application, which not only keeps track of past rent split information but also assists users in dividing the cost of house rent equally with their roommates. The front-end design of the application is done with HTML and CSS, and it is constructed with the Flask web framework. The Object-Relational Mapping (ORM) tool for communicating with the database is SQLAlchemy.
We followed industry best practices for coding by using blueprints and a modular framework. The Flask program has been separated into folders, and each folder contains files that are identical to one another. These folders are all packages with their own blueprints. Better error detection and debugging are made possible by this. Additionally, it helps in the production and deployment of the application. The exact algorithm for splitting the rent fairly is beyond the scope of this project so we decided to exclude it from the report. We have attached the directory tree of the Flask app below.
The following provides a more detailed overview of implementation:

- **Forms:** We have used flask_wtf from the Flaskform library to take input for all the necessary user information

    1. *RegistrationForm:*
        - Allows users to register by providing their username, email, and password, and confirming the password.
        - Includes validation rules for each field, such as required data, length restrictions, and a regular expression pattern for the username.



  - Performs additional validation to check if the username and email are already taken by querying the database.
  - If any validation fails, appropriate error messages are displayed.

2. *LoginForm:*
  - Provides a login form where users can enter their email and password.
  - Includes validation rules for the email and password fields, ensuring they are not empty.
  - Includes a ”Remember Me” checkbox to allow users to stay logged in after closing the browser.
  - Displays an error message if the email or password is incorrect.
3. *RequestResetForm:*
  - Allows users to request a password reset by providing their registered email.
  - Includes validation to check if the provided email exists in the database.
  - If the email is not found, an error message is displayed.
4. *ResetPasswordForm:*
  - Provides a form to reset the user’s password.
  - Requires the user to enter a new password and confirm it.
  - Includes validation to ensure the password fields are not empty and that the confirmation matches the new password.
  - Displays an error message if the password fields do not match.

- **Models:** The User model represents a user in the application, storing basic information about users, detailed description:
  - User model: represents a user in the app and inherits from db.Model and UserMixin.

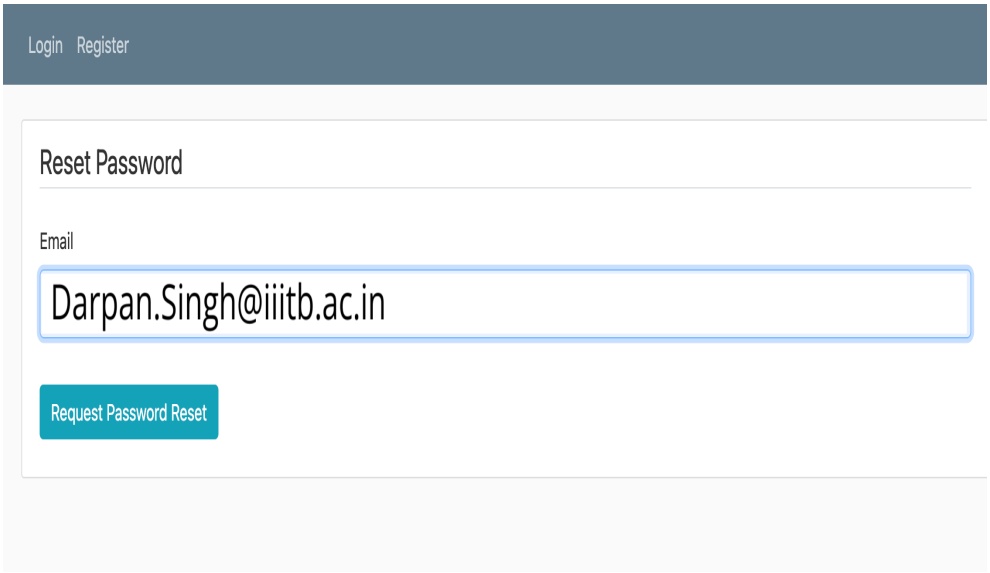
- Attributes: id (primary key integer), the username (string, unique, max length 20), email (string, unique, max length 120), password (string, required), history (string, max length 8000).
- get_reset_token: generates password reset token using itsdangerous library's URLSafeTimedSerializer.
- verify_reset_token: validates password reset token, returns corresponding user object or None.
- __repr__: defines string representation of User object for debugging purposes.
- load_user: callback function for Flask-Login to reload user object from user ID stored in session.

- **Routes:** Routes in a Flask application specify URL endpoints for accessing access to various application features. These routes normally consist of a route path and an associated function for processing the request, and they can be prefixed with the keyword "api" to represent an API endpoint. Detailed information on every route:

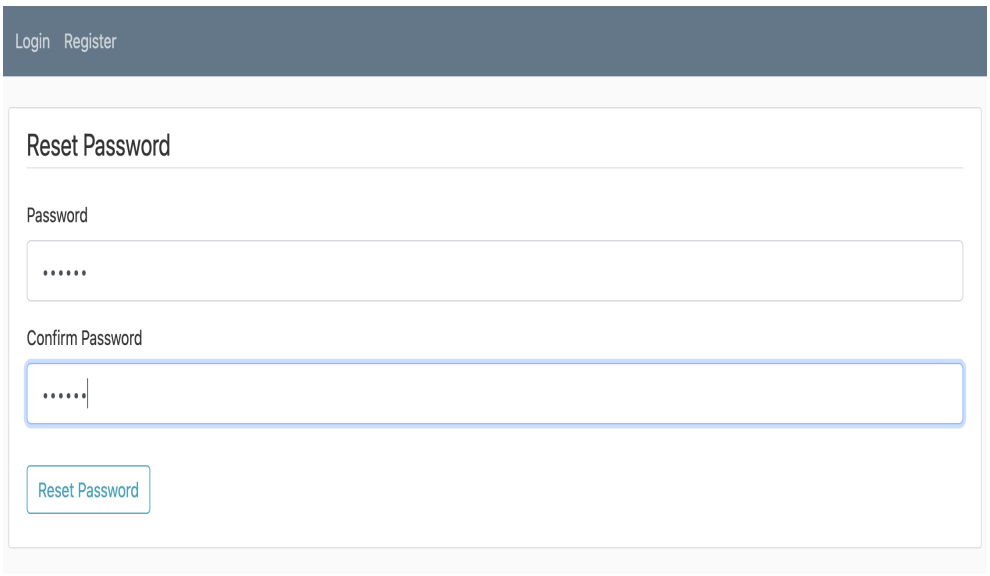
1. *register:* It is the page where the users can register for our application.
   - Renders the registration form template (register.html) for users to create a new account.
   - Validates and processes the submitted form data.
   - Hashes the password using bcrypt before storing it in the database.
   - Creates a new user record in the database.
   - Redirects to the login page after a successful registration.

2. *login:* It is the page where new users can log in to our application
   - Renders the login form template (login.html) for users to authenticate and log in.
   - Validates and processes the submitted form data.
   - Checks if the user exists and verifies the password using bcrypt.
   - Logs in the user and sets the session cookie.
   - Redirects to the previous page or the size input page after successful login.

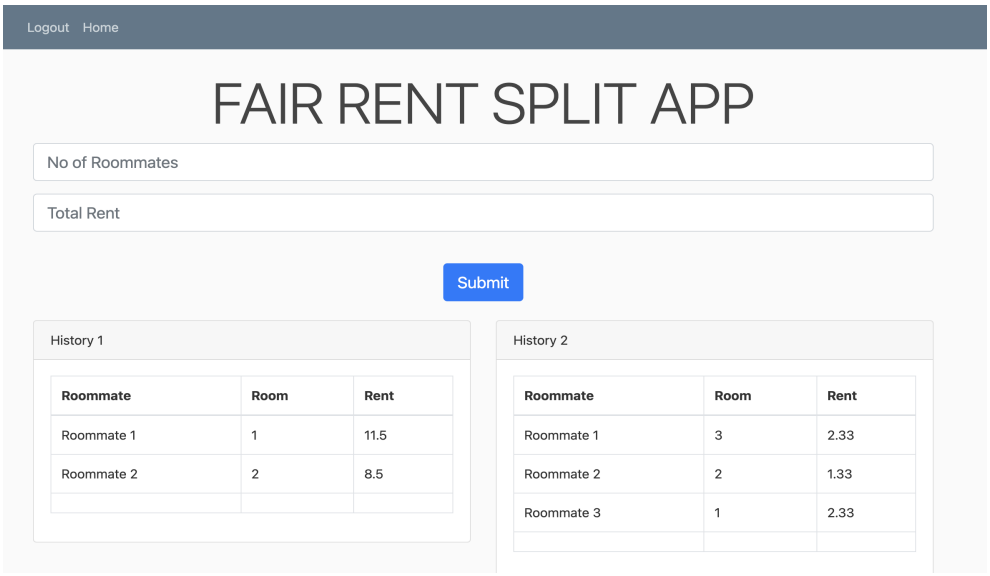
3. *logout:* It helps the user to log out from the application.
   - Logs out the currently authenticated user and clears the session.
   - Redirects to the login page after successful logout.
4. *reset_password:* It can be used by a user to reset their password.
   - Renders the password reset request form template (reset_request.html).
   - Validates and processes the submitted form data.
   - Sends a password reset email to the user's registered email address.
   - Displays a flash message to inform the user that an email has been sent with instructions.



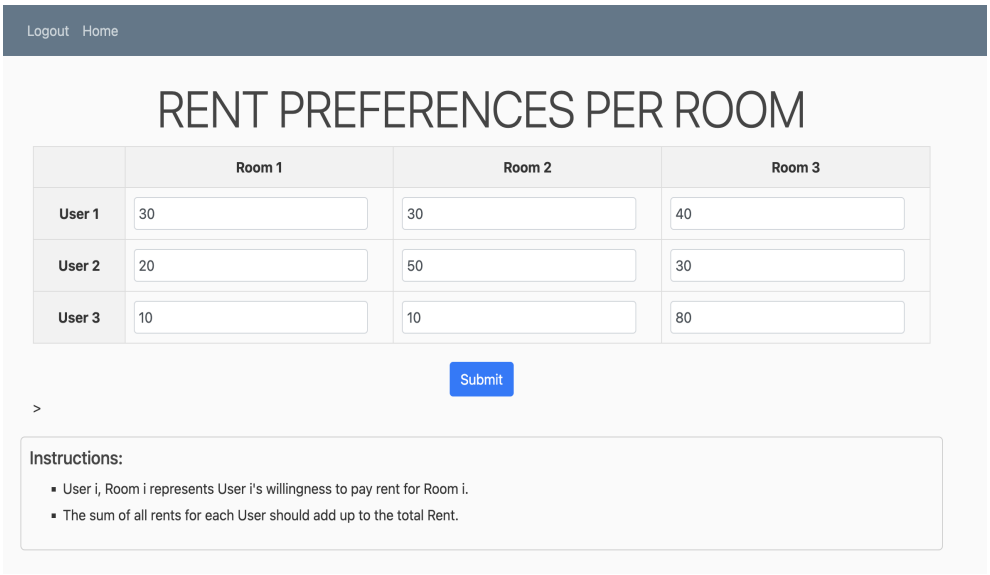
5. *reset_token:* It is used to send password reset tokens in the mail.
   - Verifies the password reset token from the URL.
   - Renders the password reset form template (reset_token.html) if the token is valid.
   - Validates and processes the submitted form data.
   - Updates the user's password with the new hashed password.
   - Displays a flash message to inform the user that the password has been reset.



6. *index:* It is the basic default route
   - Redirects the user to the login page.
   - Logs the access to the index page using the current app's logger.
7. *size_input:* It is home route for every user
   - Renders the matrix_size_input.html template, which allows the user to input the number of rooms and total rent.
   - Retrieves the user object from the current session and queries the user's history from the database.
   - Parses the history and prepares it for display by converting it into a list of unique entries.
   - Passes the prepared history as a parameter to the template for rendering.
   - Requires the user to be logged in to access the route.

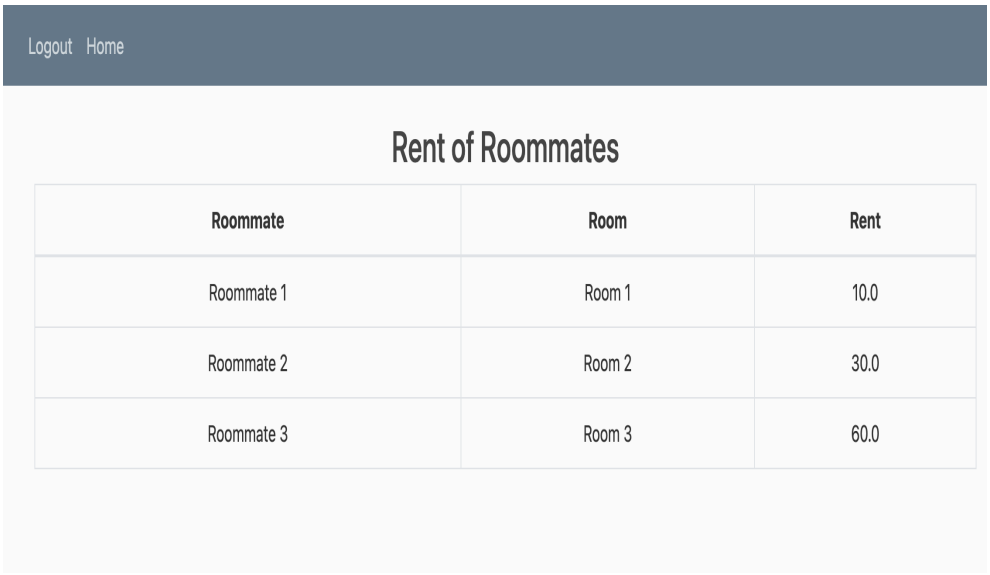


8. *matrix_input:* It is the page where we take user input
   - Retrieves the number of rooms and total rent values from the form data submitted by the user.
   - Renders the matrix_input.html template, which displays a form for the user to input the matrix values.
   - Passes the matrix size and rents as parameters to the template for rendering.
   - Requires the user to be logged in to access the route.



9. *output:* It is the page where we output the distribution of rent among the users.
   - Retrieves the matrix values from the form data submitted by the user and constructs the matrix accordingly.
   - Validates the matrix values to ensure they add up to the total rent, displaying an error message if they don't.
   - Solves the matrix by calling the solve function with the matrix size, matrix, and rent as arguments.
   - Retrieves the user object from the current session and queries the user's history from the database.

- Updates the user's history by appending the new room assignments to the existing history.
- Commits the changes to the database.
- Renders the output.html template, which displays the final room assignments and size to the user.
- Requires the user to be logged in to access the route.



- **Utility functions:** This function consists of the main logic and the algorithm of splitting rent fairly. Brief description of every function,

1. *MinCostMatching:* It is used to allot rooms to every user.
   - Performs the minimum cost matching algorithm using the Hungarian algorithm.
   - Takes a cost matrix, Lmate list, and Rmate list as input.
   - Returns the room allotment pairs based on the minimum cost matching.
2. *lp:* It is used to perform linear optimization to fit rent constraints.
   - Performs linear programming optimization to maximize utility.
   - Takes room allotment information, input matrix transpose, total rent, and input matrix as input.
   - Solves the linear program to find the maximum utility and returns the utility value.
3. *solve:* For the final computation of rent, it is the most crucial function that calls other helper functions.
   - Solves the room assignment problem.
   - Takes the number of rooms (n), input matrix (inp), and total rent (rent) as input.
   - Calls MinCostMatching to find the room allotment pairs.
   - Calls lp to calculate the utility and average utility.
   - Creates a final answer list with room assignments and utility values.
   - Returns the final answer list.
4. *send_reset_email:* It is the helper function to send mail.
   - Sends a password reset email to the user.
   - Takes a user object as input.

- Generates a password reset token using the get_reset_token() method of the user object.
- Constructs and sends an email message with the reset token included.

1.2 Future Scope:

There were a few features that we were unable to implement due to resources and time constraints but can be done in the near future. The most appealing of these features is grouping, which enables a property owner to invite users to the platform and manage information about rent splits for each and every property while also sending copies of each rental agreement to each user for their own reference. The user can pay his portion of the rent directly through the portal, and our app will collect rent from each user and send the money to the owner. Other features are user-specific, If the rent is not paid, a feature to remind users to make regular rent payments is also under consideration. To see expenses for rentals and owner revenue, more statistical characteristics can be included. At last, the security of the app for payments and emails can be strengthened.

2 Architecture

The application is containerized using Docker, which allows for easy deployment and scalability. Kubernetes is used for container orchestration, providing automated deployment, scaling, and management of the application. Ansible is used for continuous deployment management, allowing for efficient and automated deployment of new releases.

Jenkins is used for the creation of the CI/CD pipeline, which automates the build, test, and deployment process. The pipeline includes steps for building and pushing Docker images, deploying the application to Kubernetes, and running automated tests to ensure the application is functioning properly.

3 Tools Used

1. **Github**: Source Code Management tool
2. **Jenkins**: Automation server that helps in building, testing, and deploying the web application, providing Continuous Integration and Continuous Deployment of the web application
3. **WebHooks**: To help Jenkins trigger the job to build when the user commits new changes in the Github repository
4. **Ngrok**: To expose the Jenkins server running on the local machine to the Internet to perform WebHook
5. **Pytest**: Write and run tests for the web application
6. **Docker**: Containerize the web application using Docker. The created Docker image is then pushed to the public Docker hub repository. It also contains the environment required for the application to run
7. **Kubernetes**: It automates the task of container management. We use it for pulling the docker image from the Docker hub repository, deploying the application and making it available outside the cluster

8. **Ansible**: It enables Infrastructure as Code. We use it to deploy the application to kubernetes cluster and run the service to expose the application running on the pod. It automates the tasks and easily does the job for a large number of servers

9. **ELK Stack**: Helps in monitoring the log file generated by the application. The log file for this application contains the logs for each action taken by the user

4 Steps Involved

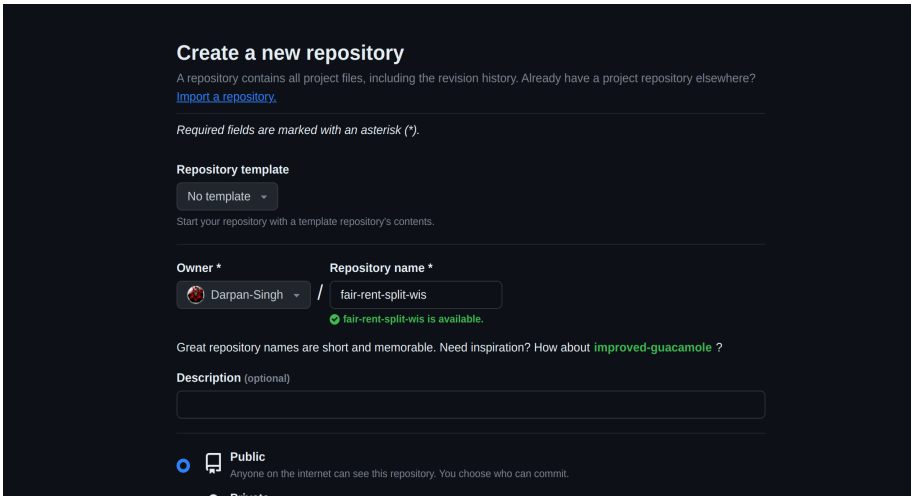
1. Install Java 11

    - sudo apt-get update
    - sudo apt install -y openjdk-11-jdk

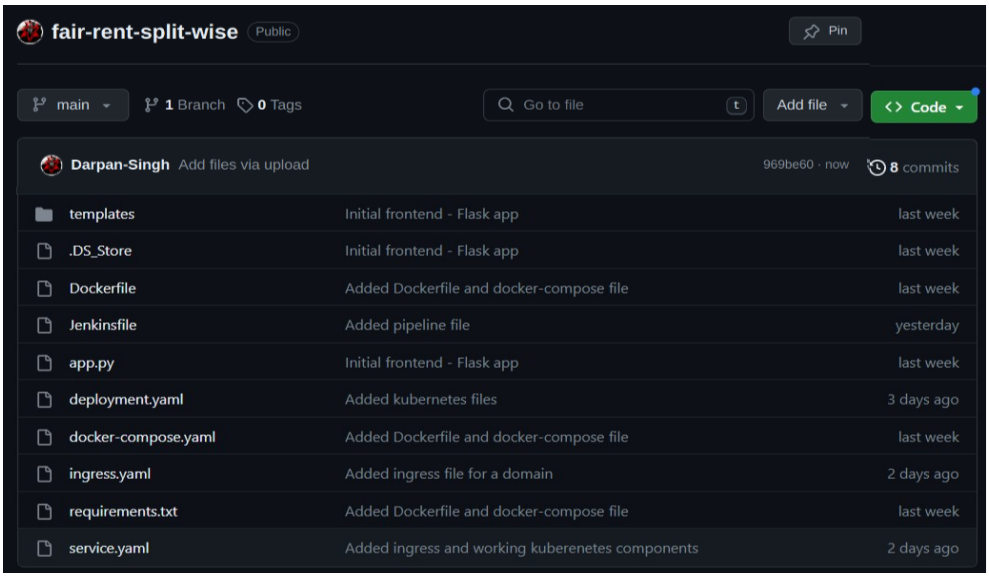
```
nidhish@ubuntu:~$ java --version
openjdk 11.0.18 2023-01-17
OpenJDK Runtime Environment (build 11.0.18+10-post-Ubuntu-0ubuntu120.04.1)
OpenJDK 64-Bit Server VM (build 11.0.18+10-post-Ubuntu-0ubuntu120.04.1, mixed mode, sharing)
```

2. Add the code to Github

    - Create a new Github repository

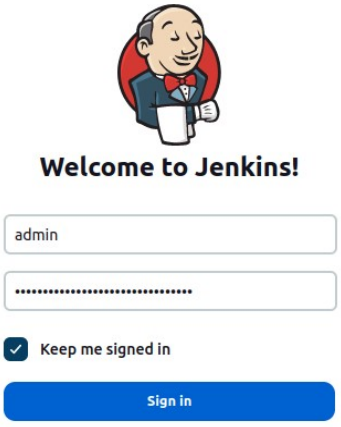


    - Follow the below steps to add the local web application code to the newly created Github repository
        (a) git init
        (b) git remote add origin https://github.com/Darpan-Singh/fair-rent-split-wise.git
        (c) git branch -M main
        (d) git add .
        (e) git commit -m ”Add Web Application code”
        (f) git push -u origin main

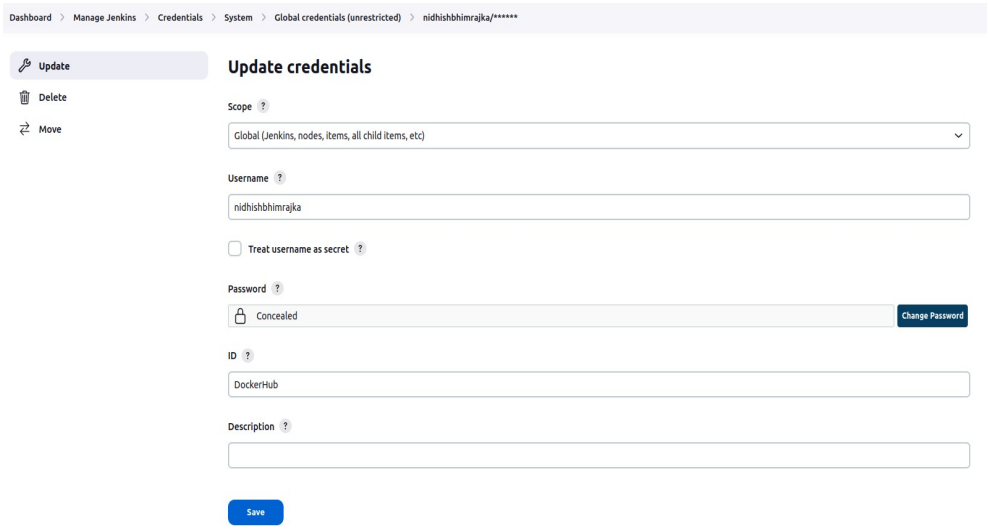


3. Installing and setting-up Jenkins

- Follow the below step to install Jenkins
  (a) wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -
  (b) sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
  (c) sudo apt install ca-certificates
  (d) sudo apt-get update
  (e) sudo apt-get install jenkins
- Open `http://localhost:8080/` in localhost to login to the Jenkins server (The admin password of Jenkins is found in: sudo cat /var/lib/jenkins/secrets/initialAdminPassword)

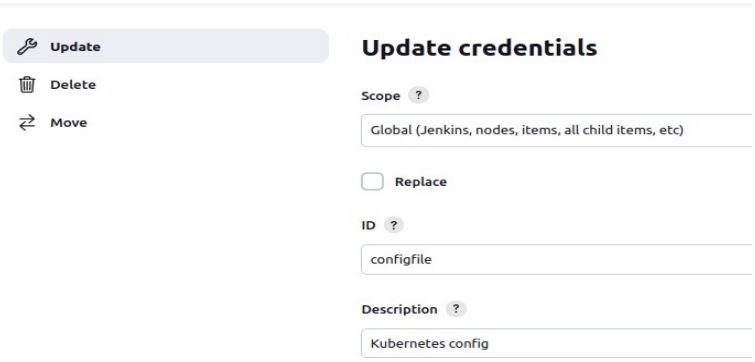


- Configuring Jenkins (DockerHub and Kubernetes)
  (a) Adding **DockerHub** credentials in Credentials
  (b) Go to Jenkins Dashboard → Manage Jenkins → Manage Credentials → global → Add new credential
  (c) Now add the DockerHub credentials as below:



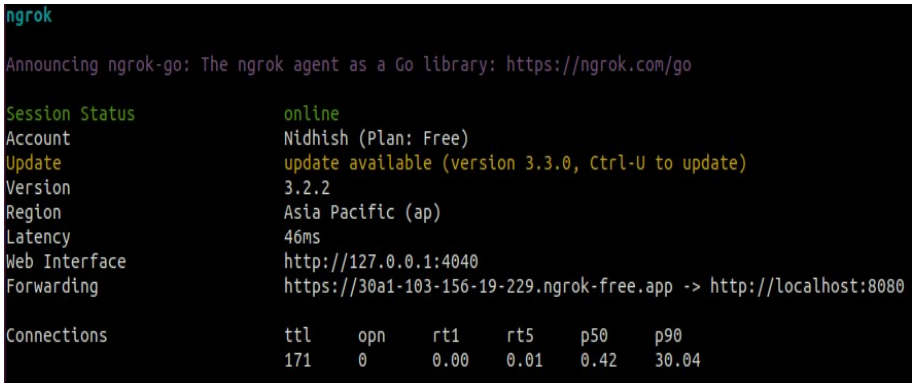
(d) Adding **Kubernetes** credentials in Credentials

(e) Follow the same steps as above and instead of password, choose secret file and add the kubernetes config file as below:



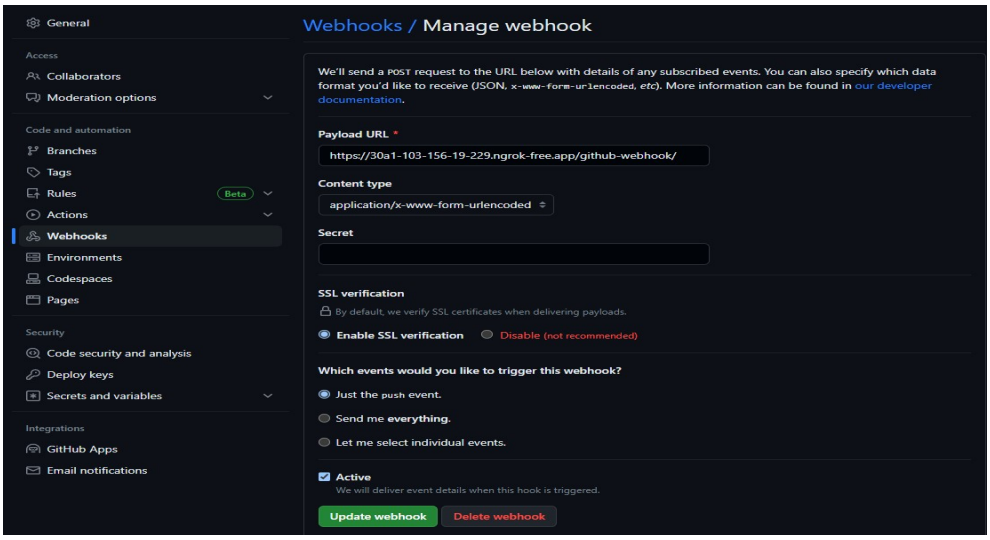
4. Installing ngrok and getting public IP for Jenkins server running on localhost

- Sign up in https://ngrok.com/
- Download ngrok from here
- Extract ngrok: sudo tar xvzf /Downloads/ngrok-stable-linux-amd64.tgz -C /usr/local/bin
- Copy Authtoken from: https://dashboard.ngrok.com/get-started/your-authtoken
- Add Authtoken: ngrok authtoken <token>
- Execute: ngrok http http://localhost:8080; copy the public ip address for the Jenkins server running on localhost

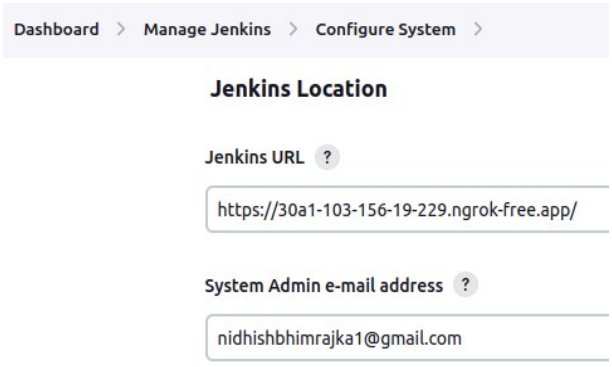


5. Setting up WebHook for the created Github repository

- Go to the Settings page of the repository and add a webhook (Put the public IP from ngrok followed by /github-webook/)

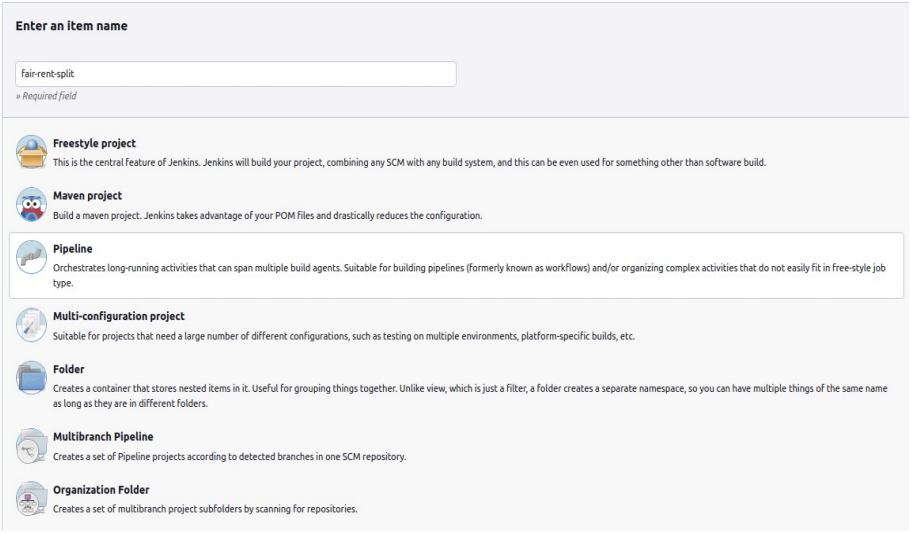


- Add this public IP to Jenkins configuration (Go to Jenkins Dashboard → Manage Jenkins → Configure System, and add the URL in Jenkins location)

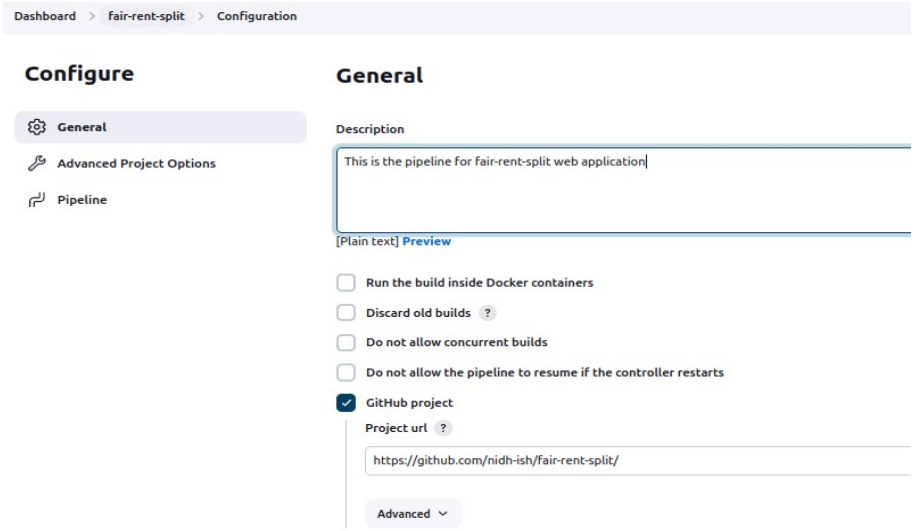


6. Creating a Pipeline project in Jenkins for the CI/CD

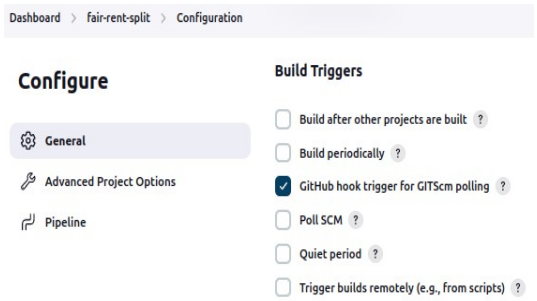
- Go to Jenkins Dashboard, then click on "New Item"
- Enter the project name (fair-rent-split) and select Pipeline type



- Check the Github project box in the “General” configuration and add the link to the project’s github repository



- In the “Build Triggers” section check the “Github hook trigger for GITScm polling (This is trigger the build job when a change is committed to the github repository)



- Write the pipeline script to be executed once a new change is committed. Following are the pipeline stages:
  (a) Clone git: Pulls the code from the remote Github repository

```
stage('Clone git'){
    steps{
        git url: 'https://github.com/Darpan-Singh/fair-rent-split-wise.git', branch: 'test'
    }
}
```

  (b) Build Docker Image: This creates a docker image on the local system. It will then be pushed to the public repository in Docker hub. From there other servers can pull the image to run the application. The details of the Docker hub are specified in the environment

```
environment {
    DOCKER_HUB_REGISTRY = 'https://registry.hub.docker.com'
    DOCKER_HUB_CREDENTIALS = 'DockerHub'
    IMAGE_NAME = "Darpan-Singh/fair-rent-split-wise"
    IMAGE_TAG = "latest"
    KUBECONFIG = credentials('configfile')
}

stage('Build Docker Image') {
    steps {
        script {
            dockerImage = docker.build("${IMAGE_NAME}:${IMAGE_TAG}", "-f Dockerfile .")
        }
    }
}
```

(c) Push Docker Image: The docker image built on the local system from the previous stage is pushed to our public repository in Docker hub from where other servers can pull the image. We run the command: sudo chmod 666 /var/run/docker.sock to give the permission for performing the above task

```
stage('Push Docker Image') {
    steps {
        script {
            docker.withRegistry('', DOCKER_HUB_CREDENTIALS) {
                dockerImage.push()
            }
        }
    }
}
```

(d) Remove old deployment: Old deployment of the web application, if any, is removed from the cluster

```
stage('Remove old deployment') {
    steps {
        withKubeConfig([credentialsId: 'configfile']) {
            sh 'kubectl delete deployment my-app --ignore-not-found=true'
            sh 'kubectl delete service my-app --ignore-not-found=true'
        }
    }
}
```

(e) Deploy using ansible: Run the kubernetes *deployment* file to deploy the docker image on the remote server. Make the application available outside the cluster by running the *service* file. Assign a domain name to the application by running the *ingress* file. (All the file use the kubernetes config file set in the environment) Overall, it enables Infrastructure as Code and automates the deployment part.

```
stage('Deploy using ansible'){
    steps{
        ansiblePlaybook colorized: true, disableHostKeyChecking: true, installation: 'Ansible', inventory: 'inventory.yaml', playbook: 'playbook.yaml', extras: "-e 'kubeconfig_file=${KUBECONFIG}'"
    }
}
```

(f) Test the Flask Application: Run the *pytest* test-cases to test the deployed application.

```
stage('Test the Flask Application'){
    steps{
        sh 'python3 -m pytest'
    }
}
```

Stage View:

5 Containerize - Docker

1. Installing Docker
    - sudo apt-get install docker.io
2. Writing a Dockerfile to generate a docker image of the web application. This Dockerfile sets up a Python 3.9 environment, installs the required Python dependencies, copies the application code into the container, sets environment variables for the application, exposes the required port, and runs the command to start the Flask application.

```
# Use an official Python runtime as a parent image
FROM python:3.9-slim-buster

# Set the working directory to /app
WORKDIR /app

# Copy the requirements file into the container
COPY requirements.txt .

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Copy the rest of the application code into the container
COPY . .

# Set environment variables for the application
ENV FLASK_APP=app.py
ENV FLASK_ENV=development
ENV SQLALCHEMY_DATABASE_URI=sqlite:///site.db
ENV SQLALCHEMY_TRACK_MODIFICATIONS=False

# Expose the port that the application runs on
EXPOSE 5000

# Run the command to start the application
CMD ["flask", "run", "--host=0.0.0.0"]
```

    - Set the working directory inside the container to **/app**.
    - Copy *requirements.txt* (contains application dependencies) from host machine to /app inside the container. Install application dependencies listed in *requirements.txt* using *pip*
    - Copy all the files and directories from the host machine to the **/app** directory inside the container
    - Sets the `FLASK_APP` environment variable to *app.py*, which is the name of the Flask application file
    - Set the `FLASK_ENV` environment variable to *development*, which enables the development environment for the Flask application
    - Set the `SQLALCHEMY_DATABASE_URI` environment variable to specify the URI for the SQLite database that the application uses

- Set the `SQLALCHEMY_TRACK_MODIFICATIONS` environment variable to *False*, which disables tracking modifications to the database
- Expose port 5000 inside the container to the host machine
- Specify the command to run when the container starts, that is to run the Flask application using the flask run command with the –host=0.0.0.0 option to make the application accessible from outside the container

6 Container Orchestration - Kubernetes

1. *minikube*

   - It allows to run a single-node Kubernetes cluster on the local machine
   - Installing minikube
     - sudo apt-get update
     - curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
     - sudo install minikube-linux-amd64 /usr/local/bin/minikube
   - Start minikube
     - minikube start

     It will create a new virtual machine and start a single-node Kubernetes cluster on the local machine

2. *kubectl*

   - It is a command line tool for interacting with the kubernetes cluster, that is, send requests to create Pods, Deployments, Services, manage them, etc.
   - Installing kubectl
     - sudo apt-get update
     - sudo apt-get install -y kubectl

     Now using kubectl, we can interact with the single-node kubernetes cluster to deploy our web application

3. Writing the *deployment.yaml* file that contains a set of specifications of the desired state of deployment.

   - Persistent Volumne Claim (PVC)
     - For the application to get access to a Persistent Volume

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-app-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
```

- We define a PVC named *my-app-pvc*, which requests 1GB of storage with ReadWriteOnce access mode. This means that the PVC can be mounted as a read-write volume by a single node at a time.

- Deployment
  - To create a deployment that runs the containerized web application

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-container
        image: darpansingh/fair-rent-split:latest
        ports:
        - containerPort: 5000
        env:
        - name: FLASK_APP
          value: app.py
        volumeMounts:
        - name: instance-volume
          mountPath: /app/instance
      volumes:
      - name: instance-volume
        persistentVolumeClaim:
          claimName: my-app-pvc
```

  - We define a **Deployment** named *my-app*, which creates 1 replica of the containerized web application. The selector specifies that the Deployment should manage the ReplicaSet whose Pods have a label app: my-app. In our case, only one pod will be created.
  - The *template* section specifies the Pod template for the Deployment.
  - The *containers* section defines a single container named my-container, which uses the Docker image darpansingh/fair-rent-split:latest. The image is pulled from the public Docker hub repository.
  - The container exposes port 5000 and sets an environment variable `FLASK_APP` to *app.py*.
  - The *volumeMounts* section specifies that the container should mount a volume named *instance-volume* at the path /app/instance.
  - The *volumes* section specifies that the *instance-volume* should be created from the PVC named *my-app-pvc*.

4. Writing the *service.yaml* file to expose the deployment to the network.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app
spec:
  selector:
    app: my-app
  type: NodePort
  ports:
    - name: http
      port: 80
      targetPort: 5000
      nodePort: 30022
      protocol: TCP
```

- We create a **Service** object with the name *my-app*. The metadata section is used to provide additional information about the object, such as labels, annotations, and so on.
- The spec section defines the behavior of the **Service**. In this case, the selector field specifies that the **Service** should select all pods with the label app: *my-app*. This means that the **Service** will forward traffic to all pods that match the label selector.
- The *type* field specifies the type of **Service**. We set it to *NodePort* to expose the **Service** on a static port on each node in the cluster.
- The ports *section* maps port 80 of the **Service** to port 5000 of the target Pod(s). The *name* field is an arbitrary name given to the port. The *nodePort* field specifies the static port number that will be used on each node to access the **Service**. The *protocol* field specifies the protocol used by the port, which is set to TCP in this case.
- So, when this **Service** makes the web application accessible externally using the node's IP address and the static NodePort.

5. Writing the *ingress.yaml* file to map a domain name to the node's IP address.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - host: fair-rent-split.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: my-app
            port:
              name: http
```

- The *spec* section is used to define the behavior of the Ingress.
- *rules* is an array of IngressRule objects that define how traffic should be routed.
- *host* specifies the hostname that the Ingress should listen on which in this case is fair-rent-split.com.
- *http* specifies that the Ingress should handle HTTP traffic.

- *path* specifies the path that incoming traffic should match which in this case is /.
- *pathType* specifies the type of matching that should be used. In this case, it's set to *Prefix* to match any path that starts with /.
- *backend* specifies the service that incoming traffic should be forwarded to.
- *name* specifies the name of the service that incoming traffic should be forwarded to. In this case, it's set to my-app.
- *port* specifies the port of the service that incoming traffic should be forwarded to. In this case, it's set to http to match the name of the Port object in the Service resource.
- When this **Ingress** is created, it will route incoming HTTP traffic for the host *fair-rent-split.com* and the path prefix / to the *my-app* Kubernetes service on port http. This allows users to access the *my-app* service using the URL http://fair-rent-split.com.
- Since, we don't own the domain, we have to first map fair-rent-split.com to the IP address of the service locally. For that add this mapping in the file: /etc/hosts.

7 Deployment - Ansible

1. Installing ansible
    - sudo apt install ansible
2. For automating the deployment part so that the web application can be deployed and managed on one or more systems.
    - Writing *inventory.yaml* file to define the lists of hosts that Ansible will manage. In our case, the localhost is the only host that Ansible will manage. The *vars* section defines variables that contains the name of the deployment, service and ingress files (resp. web application) which can be used in the *playbook.yaml* file.

```yaml
all:
  hosts:
    localhost:
      ansible_connection: local
  vars:
    deployment_file: deployment.yaml
    service_file: service.yaml
    ingress_file: ingress.yaml
    app_name: my-app
```

    - Writing *playbook.yaml* file to define a set of tasks that Ansible will perform on the target hosts defined in the *inventory.yaml* file.

```yaml
- name: Deploy my-app to Kubernetes
  hosts: localhost
  gather_facts: no
  tasks:
  - name: Apply deployment file
    command: kubectl apply -f {{ deployment_file }} --kubeconfig={{ kubeconfig_file }}
    args:
      chdir: k8
  - name: Apply service file
    command: kubectl apply -f {{ service_file }} --kubeconfig={{ kubeconfig_file }}
    args:
      chdir: k8
  - name: Wait for deployment to complete
    command: kubectl rollout status deployment/{{ app_name }} --kubeconfig={{ kubeconfig_file }}
    args:
      chdir: k8
  - name: Apply Ingress configuration
    command: kubectl apply -f {{ ingress_file }} --kubeconfig={{ kubeconfig_file }}
    args:
      chdir: k8
```

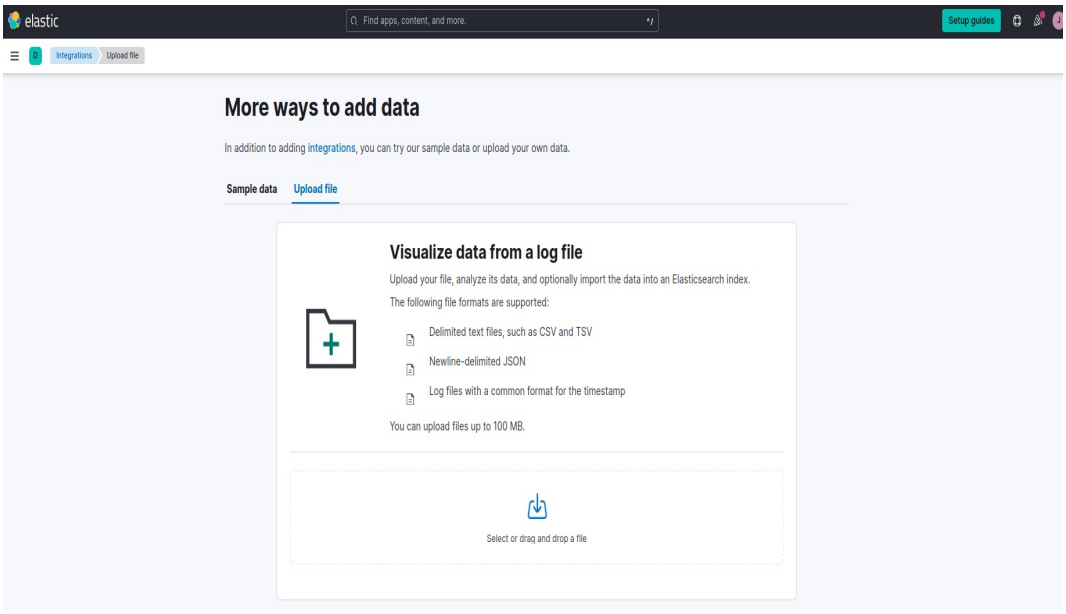
- The name of the playbook is Deploy my-app to Kubernetes.
- The hosts section specifies the target host(s) for the playbook, which in this case is the localhost host defined in the inventory file.
- The `gather_facts` option is set to no, which means that Ansible will not gather facts about the target system before running the tasks in the playbook.
- The deployment, service, and ingress file defined are run using the *kubectl* command on the target host. For running the commands, kubernetes config file is required which is specified in each of the commands (Passed as parameter with the ansible-playbook command).

8 Log using ELK

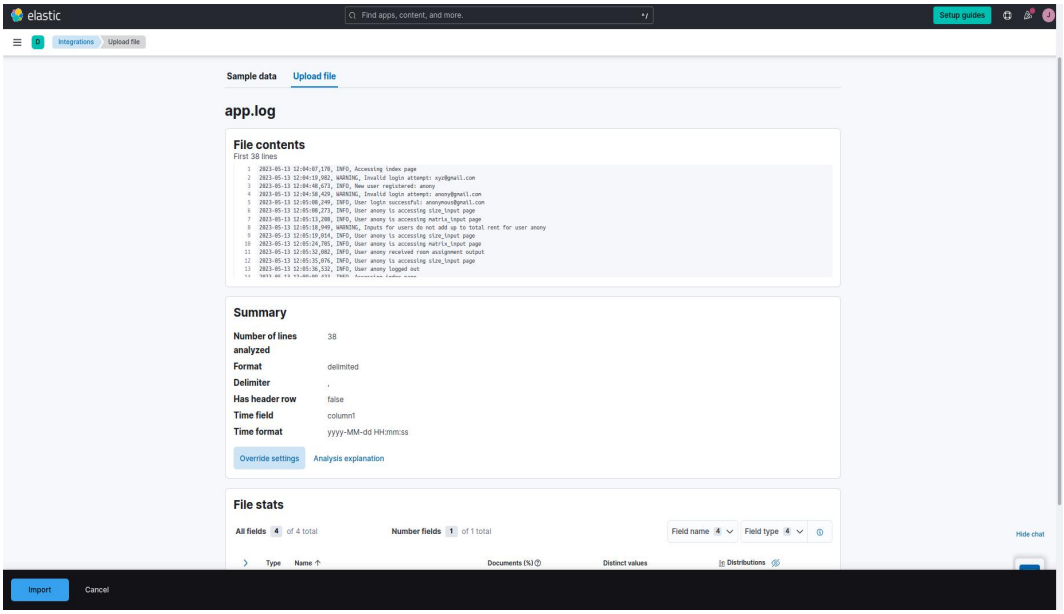
1. The log file named *app.log* is created inside the *instance* folder of the pod running the web application.
2. We extract the log file from the pod using the following command:

   kubectl cp pod-name:/app/instance/app.log app.log

3. The extracted *app.log* file is uploaded to ELK cloud for monitoring and visualisation as follows:
   - Run ELK cloud and upload the log file through Kibana

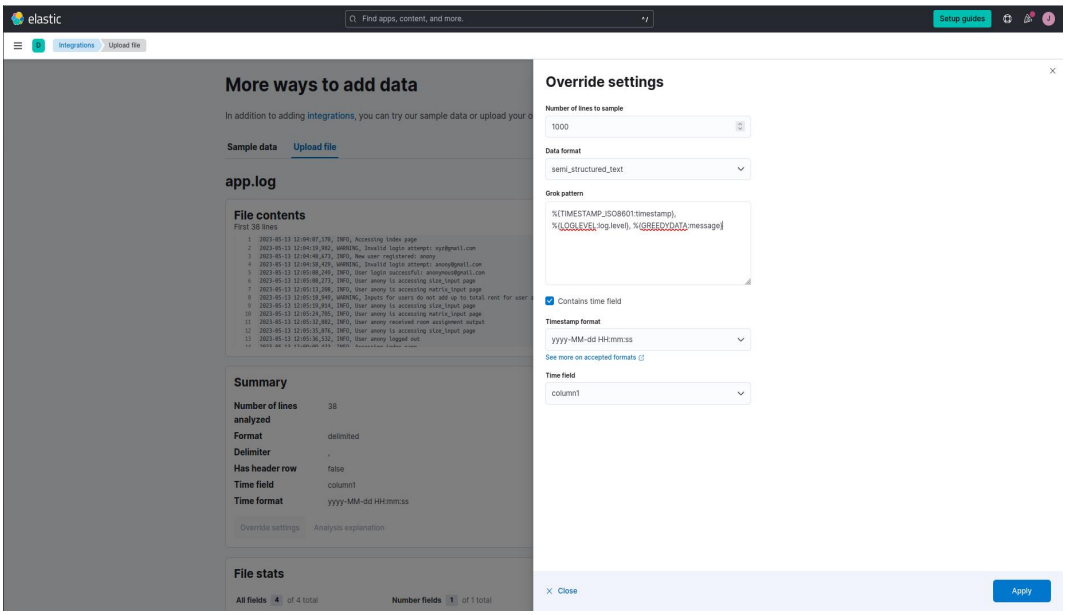


- After uploading the file, click on ”Override settings” to apply Grok Filter

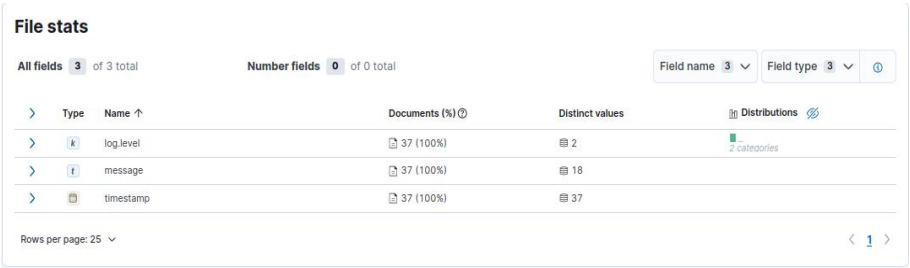


- Add the following Grok Filter:

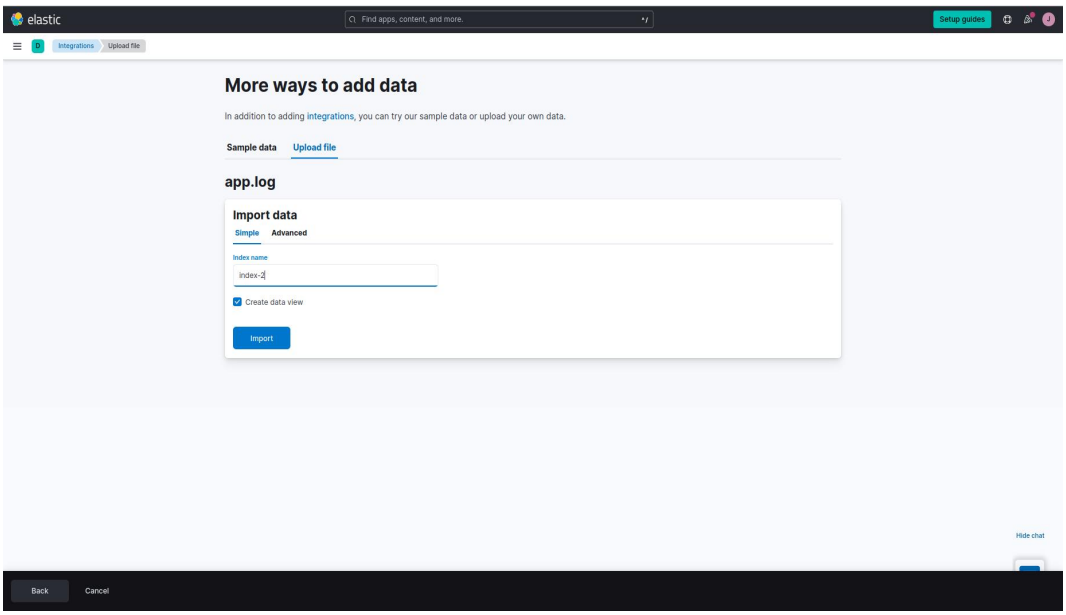
%{TIMESTAMP_ISO8601:timestamp}, %{LOGLEVEL:log.level}, %{GREEDYDATA:message}



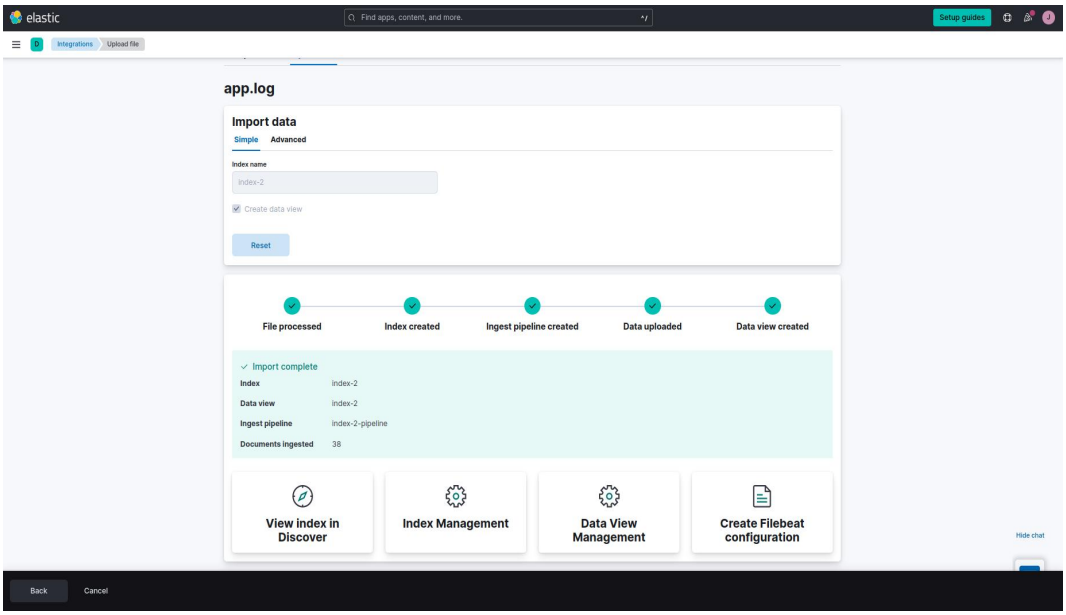
- Confirm that the data is properly ingested by logstash into elasticsearch



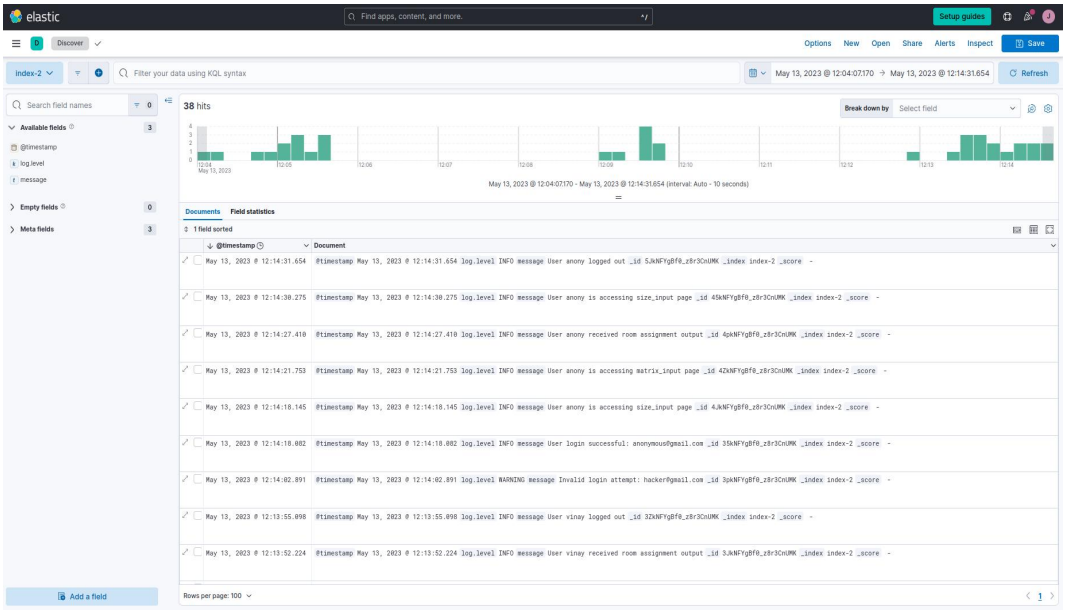
- Save the index



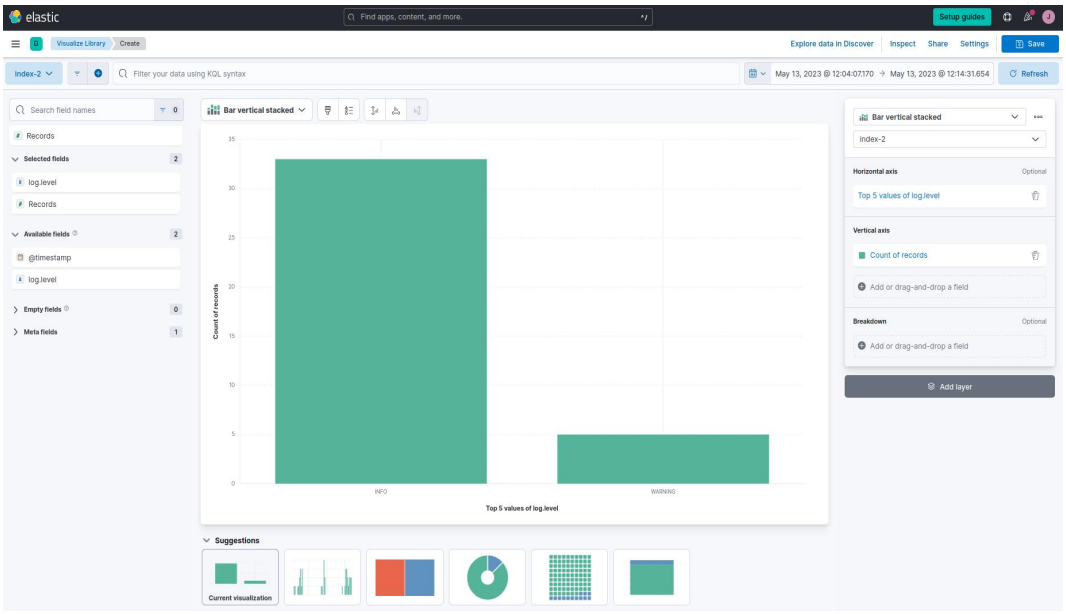
- Once the index is created, go to ”View index in Discover”



- Once the logs are properly shipped to Elasticsearch, we can create Visualisations



Below is an example of visualization created as per the *log-level* field


elastic
Find apps, content, and more.
Setup guides
Visualize Library
Create
Explore data in Discover
Inspect
Share
Settings
Save
Index-2
Filter your data using KQL syntax
May 13, 2023 @ 12:04:07.170
May 13, 2023 @ 12:14:31.654
Refresh
Search field names
Records
Selected fields
log.level
Records
Available fields
@timestamp
log.level
Empty fields
Meta fields
Bar vertical stacked
Count of records
INFO
WARNING
Top 5 values of log.level
Suggestions
Current visualization
Bar vertical stacked
Index-2
Horizontal axis
Optional
Top 5 values of log.level
Vertical axis
Count of records
Add or drag-and-drop a field
Breakdown
Optional
Add or drag-and-drop a field
Add layer