# SPE Final Project Report

## Project Application Features & Functionalities

### Team Members:

1. Vishal Cheeti - IMT2020513
2. Chakradhar Vanarasi - IMT2020021

**GitHub repo**: https://github.com/chakradhar63/Ride_Ready

1. **master** branch - Contains everything for the web app and handles deployment using Docker images of frontend, backend, and MongoDB.
2. **test3** branch - Contains everything for the web app and also includes files for testing and logging.

## Project Idea & Innovation

Introducing "**Ride Ready**," our user-friendly car rental app designed to simplify and elevate your travel experience. In today's fast-paced world, reliable transportation is a necessity, and our app goes beyond the constraints of car ownership and traditional rentals. With "Ride Ready," you have the freedom to embark on your journeys at your own pace, choosing the perfect vehicle effortlessly. The app's simplicity allows hassle-free bookings with just a few taps on your desktop. Our goal is to bring flexibility and convenience to your travel plans, seamlessly integrating transportation into your adventures.

Innovation is at the heart of 'Ride Ready.' In this fast-moving world, we're redefining car rentals, focusing on your needs with an easy-to-use app that combines convenience, flexibility, and cutting-edge technology. Choose from a variety of vehicles and book your ride effortlessly. We're breaking free from old-fashioned car ownership and rentals, offering more than just an app - '**Ride Ready**' is our commitment to making travel exciting and better through smart solutions that adapt to today's world.

## Tech-stack used

- **React.js:** We have used React.js for front-end. It is a JavaScript library which is used for building user interfaces.
- **Node.js:** We have used Node.js for the back end of the project. It is a JavaScript runtime that allows developers to execute JavaScript code on the server side.
- **Express.js:** Express has been important in constructing the server-side logic and managing the routing and middleware for our application.
- **MongoDB:** We have used MongoDB as our database. It is an open-source NoSQL database management system that is designed to handle large volumes of data with a flexible and scalable approach.

## Key Features & Functionalities

"CRUD operations," stands for Create, Read, Update, and Delete. CRUD represents the basic operations that can be performed on data. Here's a brief explanation of each:

1. **Create (C):** This operation involves creating new data or records in a database. It typically corresponds to the addition of a new entry.
2. **Read (R):** This operation is about retrieving or reading existing data from a database. It corresponds to viewing or fetching records.
3. **Update (U):** This operation involves modifying existing data in a database. It corresponds to editing or updating the values of a record.
4. **Delete (D):** This operation is about removing data or records from a database. It corresponds to deleting or removing entries.

## User features:

**Booking operations:**

- Users can meticulously choose a car of their preference and define specific booking slots. This can be considered as a “create” operation.

- The flexibility to adjust booking dates is granted to users, contingent upon the new slot being unassigned to another user. This can be considered an “update” operation.
- Cancellation of bookings is possible for users, with adherence to a stipulated cancellation fee. This can be considered as a “Delete” operation.

**Booking retrieval:**

- The 'Bookings' section provides users with a consolidated view of both current and past bookings, enhancing accessibility and user experience. This can be considered as a “Read” operation.

## Admin features:

**Car Management:**

- Administrators are vested with the authority to seamlessly introduce new cars to the platform, thereby expanding the fleet available for user rentals. This can be considered as a “Create” operation.
- The capability to modify details for cars uploaded by administrators is facilitated, ensuring accurate and up-to-date information. This can be considered as an “Update” operation.
- Removal of cars from the rental portal is within the administrative purview, providing control over the platform's offerings. This can be considered as a “Delete” operation.

**Administrative Oversight:**

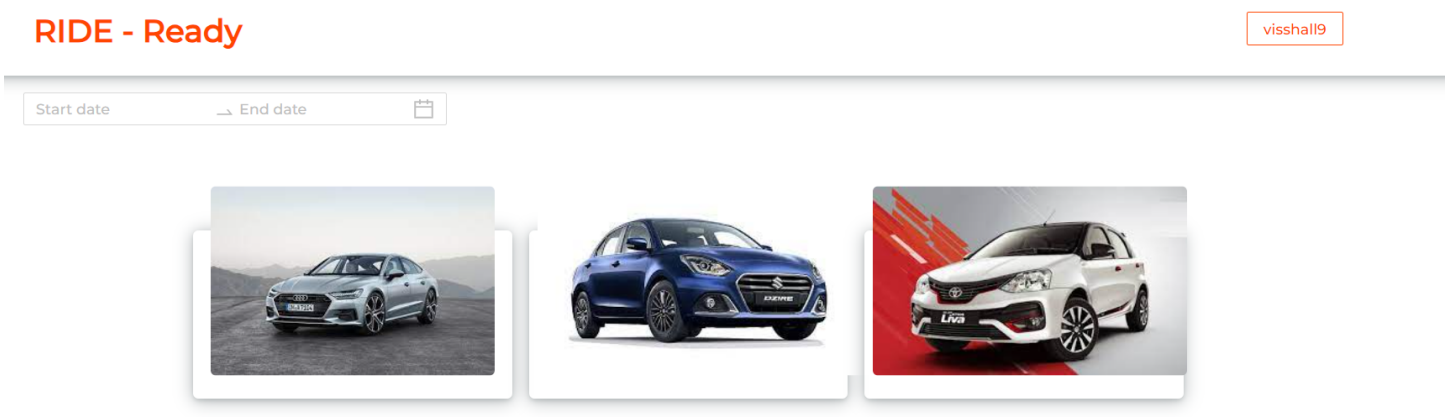
- The 'Admin' section serves as a centralized hub for administrators to retrieve a comprehensive list of cars uploaded, streamlining administrative oversight. This can be considered as a “Read” operation.

## Complete overview and UI of the project

The user will first be introduced to the login page of the project.

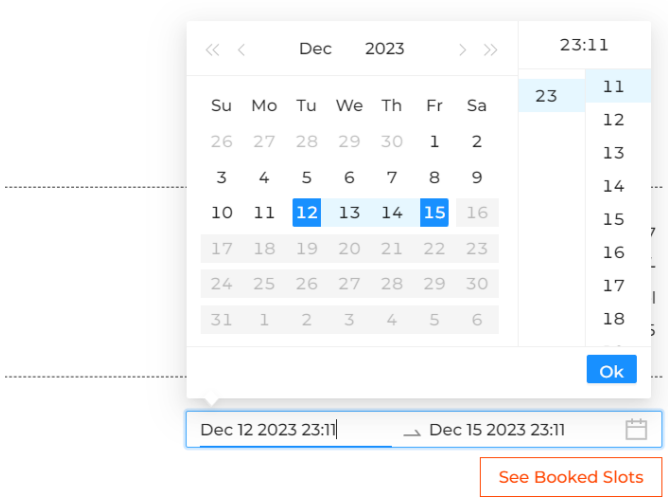
Then the user will be able to register and log into the website seamlessly.


The user can pick a car from a wide variety of available options.

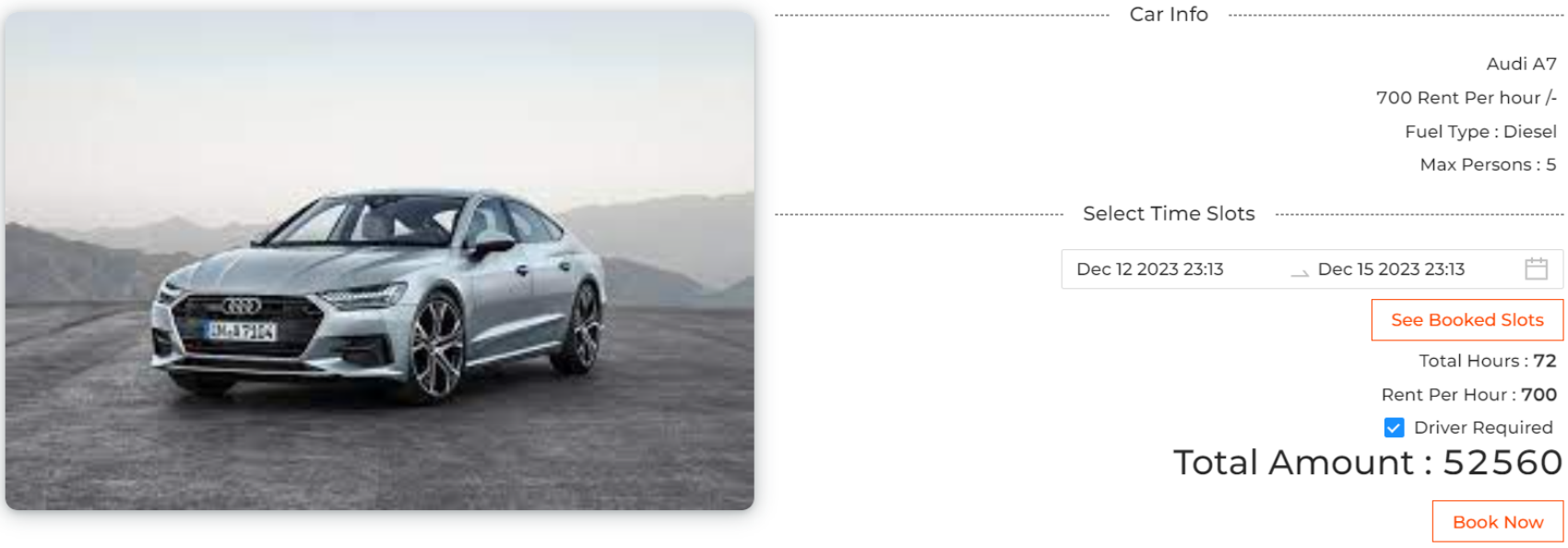



All the necessary information about the car can be seen on the dashboard.

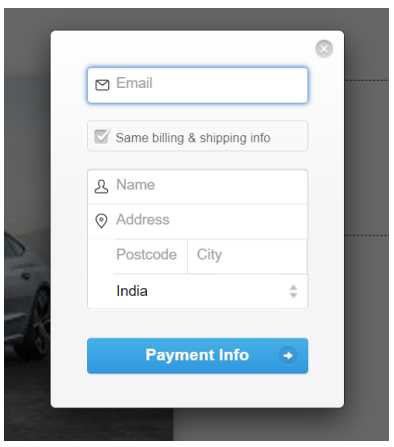

After choosing a car, the user can select the time slot of the booking



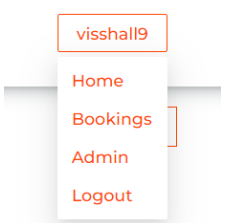

User can then choose to proceed with/without a driver and get ready to pay



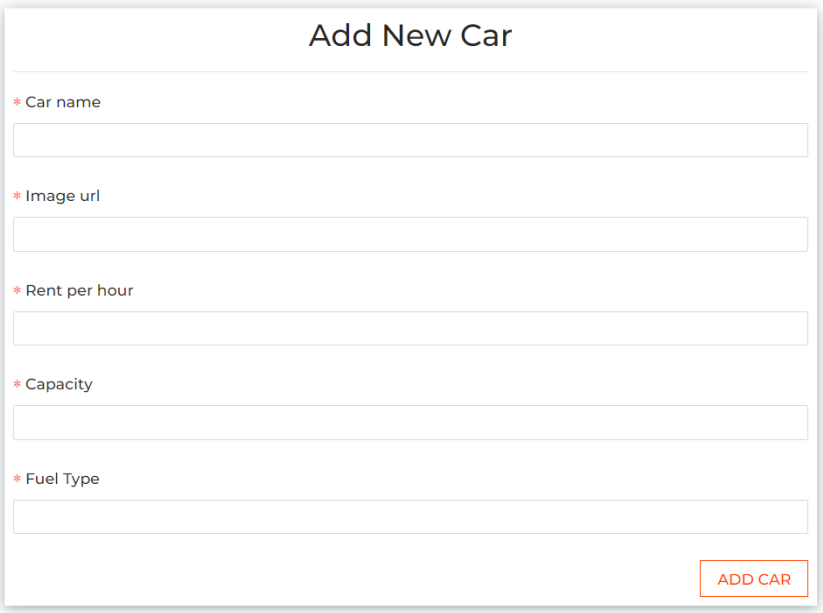

The user then fills out the payment info and the booking gets confirmed after the payment



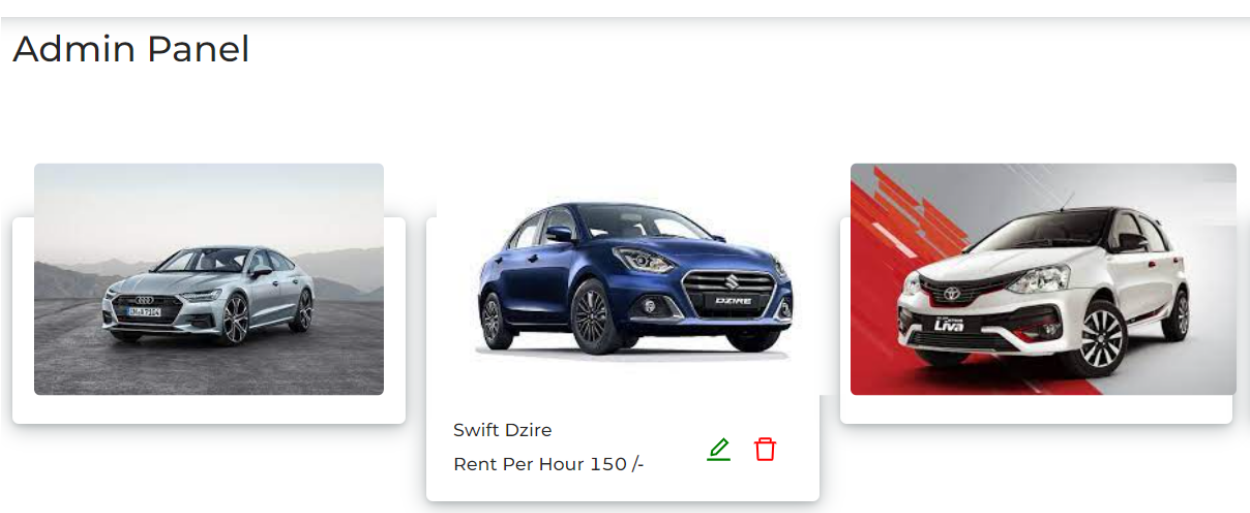

Users can check all the previous bookings in the “Bookings” section.




If the user wants to rent his car to others, he can do it by choosing “Add car”




In the admin panel, the user can choose to edit and delete the car that he has registered for renting




These multiple features and seamless onboarding experience make it very easy and convenient for the users to book any car as per their requirement and also allow them to register their cars for renting whenever they want to do it. They can even remove their cars from the portal whenever they want provided it's not already booked by others.

## DevOps Introduction

DevOps blends software development and IT operations to automate and enhance software delivery and infrastructure management. It aims to speed up development, increase release frequency, and enhance software quality. DevOps promotes teamwork, leveraging automation, continuous integration, delivery, and monitoring. The goal is faster, efficient software delivery with top-notch quality and reliability, particularly crucial in today's software-driven world.

Our project aims to build and deploy our “**Ride-Ready**” application using automation tools like Docker, GitHub, Maven, JUnit, Jenkins, Ansible, and ELK stack. The primary objective is to better understand key DevOps concepts such as Continuous Integration, Continuous Deployment, and Configuration Management by building a Jenkins pipeline. Through this project, we'll illustrate how applying DevOps practices can enhance productivity and foster collaboration between development and operations teams.

## Advantages of DevOps

- **Faster Fixes**: DevOps enables rapid identification and resolution of issues, reducing the time it takes to fix problems in software. This leads to more stable and reliable applications.
- **Quicker Market Entry**: DevOps streamlines the development and deployment process, allowing new features and updates to be released faster. This speed-to-market advantage is essential for staying competitive.
- **More Frequent Deployments**: With DevOps, software can be deployed more frequently. This continuous deployment approach keeps applications up-to-date and responsive to changing user needs.
- **Reduced Release Failures**: DevOps practices emphasize automation, testing, and quality control. This results in fewer failures and errors during software releases, enhancing overall reliability.
- **Quicker Recovery from Crashes**: In case of system crashes or failures, DevOps provides efficient recovery mechanisms. Automated backups and redundancy reduce downtime, ensuring a faster return to normal operation.

## Tools Used

- **Source Code Management:** Git and GitHub
  Git is a code-tracking system that helps teams work together and keep track of code changes. GitHub, based on Git, is a web platform that stores code repositories, making collaboration, issue tracking, and project management easier. These tools are vital for managing code in software projects, fostering teamwork, version control, and smoother development.
- **Continuous Integration**: Jenkins
  Jenkins is a popular open-source tool for Continuous Integration and Continuous Delivery in software development. It automates tasks like building, testing, and deploying code changes smoothly and efficiently. With Jenkins, developers can integrate their code frequently, catch and fix issues early, and automate the software delivery to different environments.
- **Containerization**: Docker
  Docker is a widely used platform for containerization, a way to package and run applications with all their requirements in isolated containers. These containers are easy to move, reliable, and efficient, simplifying application development, testing, and deployment across various settings. Docker ensures that a project's components and setup stay the same, making it beneficial for smooth deployment and scalability, particularly in complex software projects.
- **Configuration Management and Continuous Deployment**: Ansible
  Ansible is an open-source tool that automates configuration management, application deployment, and task automation. It simplifies server and application management, ensuring consistent configurations and automating routine tasks. Ansible is essential for maintaining Configuration Management and supports faster, reliable software releases through automated deployment in Continuous Deployment practices.
- **Streamlining Build Automation**: GitHub Webhooks
  GitHub Webhooks are a feature that allows developers to automate actions based on events in a repository. They send HTTP POST requests to chosen endpoints, like a Continuous Integration (CI) server. Webhooks streamline build automation by notifying the CI server when actions like code push or pull requests happen. This automatic process ensures continuous integration and testing, making software development more efficient and reliable.
- **Setting Up GitHub Webhook for Communication with Jenkins**: Ngrok
  Ngrok is a service that creates secure tunnels for making local services accessible online. When setting up GitHub Webhooks to talk to Jenkins, Ngrok helps by providing an external URL for your local Jenkins server. This way, GitHub's Webhooks can reach your Jenkins for automatic triggers. Ngrok makes connecting GitHub Webhooks to Jenkins easy, ensuring smooth communication, even if your Jenkins is running locally.
- **Log Management:** ELK Stack (Elasticsearch, Logstash, and Kibana)
  The ELK Stack is a powerful combination of three open-source tools: Elasticsearch, Logstash, and Kibana, designed for log management and analytics.
  - **Elasticsearch:** A search and analytics engine that stores and indexes data, making it easy to search, analyze, and visualize large volumes of log data.

- **Logstash:** An ETL (Extract, Transform, Load) tool that collects, processes, and forwards logs and events from various sources to Elasticsearch. It ensures data normalization and enrichment.
- **Kibana:** A data visualization and exploration tool that allows users to create interactive dashboards and reports based on the data stored in Elasticsearch.

## Configuring the tools

- **Git and GitHub**

```
sudo apt-get update
sudo apt install git
git --version
```

- **Jenkins**

```
wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -
sudo sh -c 'echo deb https://pkg.jenkins.io/debian binary/ >/etc/apt/sources.list.d/jenkins.list
sudo apt install ca-certificates
sudo apt-get update
sudo apt-get install jenkins
sudo service jenkins start
sudo service jenkins status

# For jenkins password
sudo cat /var/lib/jenkins/secrets/initialAdminPassword

# Go to http://localhost:8080/ to use jenkins
# Paste the password there and start jenkins
```

- **Docker**

```
sudo apt-get install docker.io
sudo apt install apt-transport-https ca-certificates curl
sudo apt-get install software-properties-common
curl -fsSL https://get.docker.com -o get-socker.sh
bash get-docker.sh
docker -v
```

- **Ansible**

```
sudo apt-get update
sudo apt install openssh-server
sudo apt-add-repository ppa:ansible/ansible
sudo apt update
sudo apt install ansible
```

- **Ngrok**

```
# sign up in https://ngrok.com/
# Download ngrok from: https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.tgz

# Extract ngrok from the terminal:
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
```

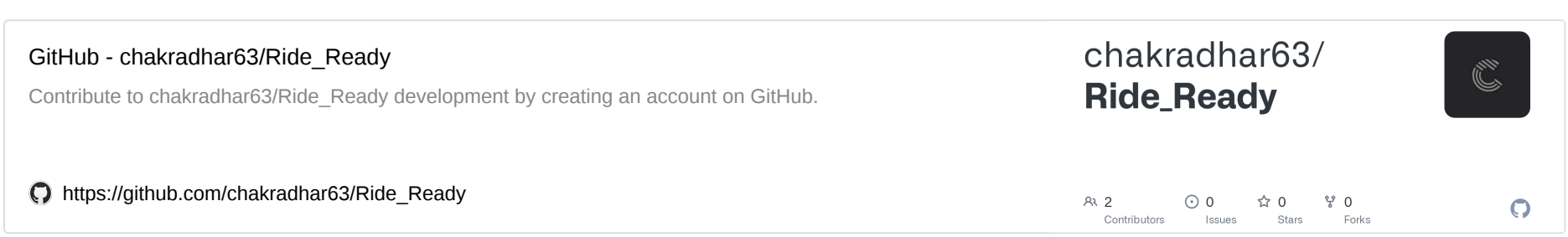
```
ngrok authtoken <token>

# Copy the public ip address for your local host
ngrok http 8080
```

# Project Implementation

## Integrating Version Control with Git and GitHub

With our functional project in place, the next step is introducing version control using Git. Git enables us to work on different branches, make commits to track changes and revert if needed. To make our code accessible from anywhere, we'll use GitHub to host our Git repository remotely. This setup allows us to separate development from deployment and opens the door for tools like Jenkins to fetch updates remotely.

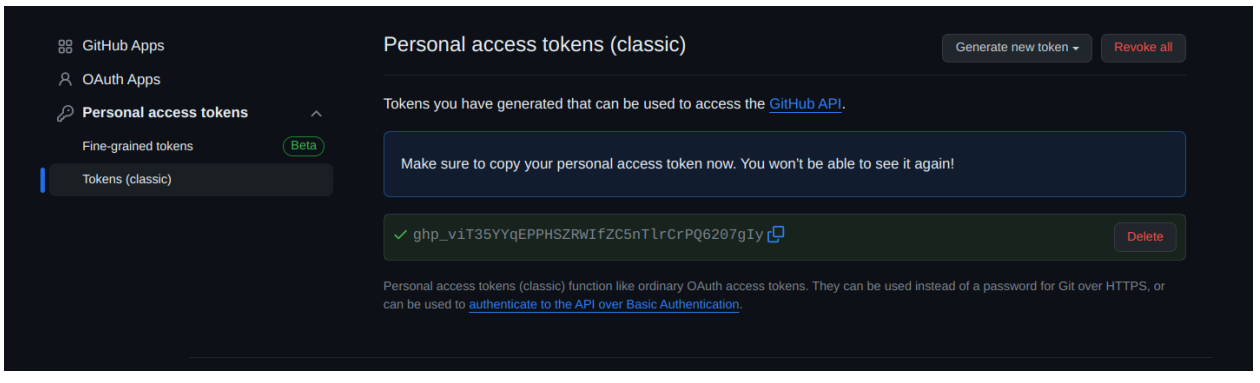
To get started, we first install Git on our local system. Once that's done, we initialize Git for our project by running a specific command. Afterward, we set up a remote repository, which allows us to send our local project changes to this remote location. It's essential to create this repository without a readme or gitignore file. This sets the foundation for managing our project with Git.




```
# Initializes a new Git repository in the current directory
git init

# Configures the global Git user name as 'chakradhar06'
git config --global user.name chakradhar06

# CConfigures the global Git user email as 'chakrivanarasi@gmail.com'
git config --global user.email chakrivanarasi@gmail.com

# Stages all changes in the current directory for the next commit
git add .

# Checks the status of the Git repository, displaying untracked, modified, and committed files
git status

# Commits the staged changes with the provided commit message
git commit -m "commit message"

# Adds a remote repository named 'origin' with the URL
git remote add origin https://github.com/chakradhar63/SimpleCalculator
```

To push our project changes to the remote GitHub repository, we start by creating personal access tokens. To do this, we access our GitHub profile, go to "Developer settings," select "Personal access token," and create a classic token. It's important to save the secret phrase that this token generates. This token will be used to authenticate and connect our local project to the remote repository.

```
# Pushes local changes to the 'master' branch of the 'origin' remote repository
git push -u origin master
```




Once we push the branch to the remote repository, you'll find your changes on GitHub under the "master" branch. This sets up a working system with both local and remote repositories, allowing us to sync and collaborate with multiple developers on the project. It's a foundation for team collaboration and version control.

## Creating and integrating with the Jenkins pipeline

With our repositories in place, we're now going to establish a CI/CD pipeline in Jenkins. This pipeline serves as the core of our continuous integration and continuous deployment process. It outlines the sequence of steps and their execution order. We'll create this pipeline as a new item in Jenkins, and it will initiate cloning the remote repository we've set up, enabling us to deploy it using Ansible. This pipeline streamlines our development and deployment workflow. We've structured the pipeline into six stages, which are outlined below:

1. **Git Clone**
2. **Testing**
3. **Building Front-end Image**
4. **Building Back-end Image**
5. **Pushing the Docker Image to the DockerHub**
6. **Cleaning Docker Images**
7. **Ansible Deployment**

### Stage 1 - Git Clone

```
stage('Stage 1: Git Clone'){
steps{
        git branch: 'master',
                        url:'https://github.com/chakradhar63/Ride_Ready.git'
            }
}
```

In the above script, the git branch must match the branch in GitHub and the URL must be the git URL to the project from GitHub.

### Stage 2 - Testing

In the second stage, we're aiming to run the local repository and check if the project compiles successfully and tests the front-end and back-end files of the application.

In the Testing stage of our pipeline, we're doing two important things. First, we're checking the front part of our website (front-end) to make sure it's built correctly and works as expected. We're also testing the back part (back-end) to ensure the server and database are functioning properly.

```
stage('Testing'){
        steps{
                dir('frontend'){
                        sh "npm install"
                        sh "npm test"
                }
                dir('backend'){
                        sh "npm install"
                        sh "npm test"
                }
        }
}
```

```
  User API Tests
Mongo DB Connection Successfull
[2023-12-15 17:46:16.591] [error] [Failure] Login Failure: Incorrect username or password.
[2023-12-15 17:46:16.595] [error] [Failure] Login Failure. Please try again later.
    ✔ Signin user - Failure (2715ms)
[2023-12-15 17:46:16.678] [info] [Success] Login Success: User 'IMT2020021' has successfully logged in.
    ✔ Signin user - Success (76ms)


  2 passing (3s)
```

In this **server testing** file, we've set up a Mocha test suite utilizing the **Chai** assertion library and **Chai HTTP** for making HTTP requests to evaluate the user authentication functionality of our server-side API. The test suite comprises two key scenarios for user **signin**: one focusing on failure and the other on success.

In the first scenario, we attempt to signin a user with invalid credentials. The test queries the User model to find a user with the provided credentials, then sends a POST request to the '/api/users/login' endpoint with the same credentials. The expectation is that the response status should be 400 (Bad Request), signaling an unsuccessful login attempt. Additionally, we verify that the response body is an object.

In the second scenario, we aim to signin a user with valid credentials. Similar to the first case, we query the User model to find a user with the specified credentials and send a POST request to the '/api/users/login' endpoint. This time, we expect the response status to be 200 (OK), indicating a successful login. Similar to the previous scenario, we verify that the response body is an object.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

(node:84011) [DEP0040] DeprecationWarning: The `punycode` module is deprecated. Please use a userland alternative instead.
(Use `node --trace-deprecation ...` to show where the warning was created)
 PASS  src/components/Login.component.test.js (30.909 s)
  ✓ renders login form (285 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        42.941 s
Ran all test suites.

Watch Usage: Press w to show more.
```

Frontend testing checks if a web application's visible components work as expected. The provided code is a Jest test for a React login form. It ensures the form renders correctly by checking the presence of email and password inputs, and a submit button option. This helps catch potential issues early, ensuring the front end behaves as intended for a better user experience.

The provided code is a frontend test for a React login form using Jest and testing-library/react. It checks if the form renders correctly by verifying the presence of email and password input fields, a submit button. The test, wrapped in MemoryRouter for React Router support, uses expect statements to ensure specific elements are present in the rendered output. This proactive testing helps detect issues early, ensuring the frontend components work as intended and instilling confidence in the user interface's reliability.

## Building Docker Image

Now that we can clone and build the project, the next step is to containerize it with Docker. This involves packaging all the project's required dependencies into a single container, which can be used for remote deployment in later stages. To begin, we need to add a **Dockerfile** to our project, which will outline the necessary steps and scripts for converting our project into a Docker container. This file should be created in the root directory of the project.

### Stage 3 - Building Front-end Image

```
FROM node:21.4.0
WORKDIR /
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm", "start"]
```

This Dockerfile sets up an environment for a Node.js application. It starts with Node.js version 21.4.0, copies the package files for dependency installation, installs dependencies, copies the application code, exposes port 3000, and sets the default command to start the Node.js app using `npm start` when the container runs. In essence, it creates a container ready to run a Node.js application with its dependencies.

## Stage 4 - Building Back-end Image

```
FROM node:18.0.0-alpine
WORKDIR /
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 5000
CMD ["npm", "start"]
```

This Dockerfile sets up an environment to run a Node.js application. It begins by using Node.js version 18.0.0 on an Alpine Linux base image. It then sets the working directory to the root, copies package files for dependency management, installs the required dependencies with `npm install`, and copies the entire application code. The Dockerfile exposes port 5000 and, when the container starts, it runs the command `npm start` to launch the Node.js application. In simple terms, it's a recipe for creating a Docker container that runs a Node.js app on port 5000.

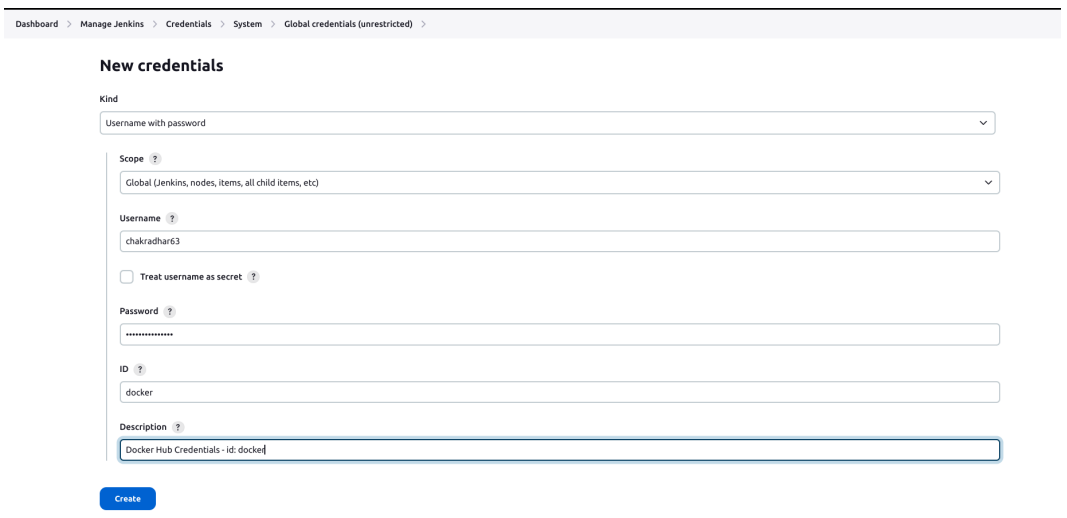
```
stage('Build Frontend Image') {
            steps {
                        sh 'docker build -t frontend-image ./frontend'
            }
}
stage('Build Backend Image') {
            steps {
                        sh 'docker build -t backend-image ./backend'
            }
}
```

In the pipeline, we have two stages dedicated to building Docker images for the front end and backend of our application. In the "Build Frontend Image" stage, we execute a command (`docker build -t frontend-image ./frontend`) to create a Docker image for the frontend, tagging it as "frontend-image." Similarly, in the "Build Backend Image" stage, we use `docker build -t backend-image ./backend` to construct a Docker image for the backend, labeled as "backend-image." These stages essentially package our application components into Docker images, making them ready for deployment and ensuring consistency across different environments.

## Stage 5 - Push Docker Image to DockerHub

Docker Hub serves as a cloud storage for Docker images. It enables us to create a Docker image on one system, upload it to Docker Hub, and then download and run the same image on another system. This capability is crucial for remote builds and deployments, as it allows us to easily share and deploy Docker images.

After registering on Docker Hub, we'll need to provide our username and password to Jenkins. This information will be saved as credentials in Jenkins under the name "docker". These credentials are essential for pushing the Docker image we created earlier to Docker Hub, allowing us to store and share our Docker images on the platform.




Once the credentials have been saved, we can execute the next stage of the pipeline with the following script:

```
stage('Push Images to DockerHub') {
        steps {
```

```
                withCredentials([usernamePassword(credentialsId: 'DockerHubCred',
                usernameVariable: 'DOCKER_USERNAME', passwordVariable: 'DOCKER_PASSWORD')]) {
                        sh 'docker login -u $DOCKER_USERNAME -p $DOCKER_PASSWORD'
            sh 'docker tag frontend-image chakradhar63/frontend-image:latest'
            sh 'docker push chakradhar63/frontend-image:latest'
            sh 'docker tag backend-image chakradhar63/backend-image:latest'
            sh 'docker push chakradhar63/backend-image:latest'
                }
        }
}
```

In the "Push Images to DockerHub" stage, we securely log in to DockerHub using credentials. Once logged in, we tag our previously built frontend and backend Docker images with a specific version ("latest"). This tagging helps organize different versions of our images. Finally, we push these tagged images to DockerHub under the repository of "chakradhar63," making them accessible for deployment and sharing with others. Essentially, this stage ensures our Docker images are securely stored on DockerHub for easy retrieval and deployment in various environments.

The "docker build" command follows the pattern: `<DockerHub username>/<container name>:<tag>`. To check if the image is created successfully, you can use a command to view the available containers. This helps ensure that the image is being generated as intended.

```
chakradhar@chakradhar:~/Documents/Test_Project_2 (image_comm)/Ride_Ready$ sudo docker images
[sudo] password for chakradhar:
REPOSITORY                    TAG       IMAGE ID       CREATED       SIZE
chakradhar63/backend-image    latest    ea11c7997625   3 days ago    202MB
backend-image                 latest    ea11c7997625   3 days ago    202MB
chakradhar63/frontend-image   latest    1eb178420b86   3 days ago    1.81GB
frontend-image                latest    1eb178420b86   3 days ago    1.81GB
mongo                         latest    5acb2131d51f   11 days ago   757MB
```

## Stage 6 - Cleaning the Docker Images

Whenever we run the pipeline, a new image is generated. Only the one with the "latest" tag is pushed to Docker Hub. To prevent potential issues, we need to remove older images, ensuring that when we pull images later in the pipeline, there are no conflicts with existing file names.

To clean up and remove all images except the most recently generated one, we'll incorporate the following commands into the pipeline script.

```
stage('Clean docker images'){
            steps{
                        script{
                                    sh 'docker container prune -f'
                  sh 'docker image prune -f'
                        }
            }
}
```

These commands will eliminate any dangling images and containers in the pipeline. Once this stage is executed, running `sudo docker images` will reveal only a single remaining image, ensuring a clean and uncluttered environment.

```
chakradhar@chakradhar:~/Documents/Test_Project_2 (image_comm)/Ride_Ready$ sudo docker images
[sudo] password for chakradhar:
REPOSITORY                    TAG       IMAGE ID       CREATED       SIZE
chakradhar63/backend-image    latest    ea11c7997625   3 days ago    202MB
backend-image                 latest    ea11c7997625   3 days ago    202MB
chakradhar63/frontend-image   latest    1eb178420b86   3 days ago    1.81GB
frontend-image                latest    1eb178420b86   3 days ago    1.81GB
mongo                         latest    5acb2131d51f   11 days ago   757MB
```

## Stage 7 - Ansible Deployment

With our Docker image on Docker Hub, the next step is to pull and create a container using Ansible for deployment. To facilitate deployment, we'll store deployment details within the project directory under a **Deployment** sub-directory. This sub-directory will house two files: **inventory**, listing the clients for deployment, and **deploy.yml**, specifying the image to be pulled and deployed.

Docker Compose serves as a valuable tool designed to simplify the definition and distribution of multi-container applications. By crafting a YAML file, we can articulate various services and effortlessly launch multiple containers with a single command. The elegance of utilizing a compose file extends beyond personal convenience—it enables seamless collaboration as others can

effortlessly run the application. With the repository cloned, initiating the entire application becomes a straightforward task, streamlining the onboarding process for team members or collaborators who can execute a single command to get the application up and running.

```
version: '3'
services:
  frontend:
    image: frontend-image:latest
    ports:
      - '3000:3000'
    environment:
      - REACT_APP_BASE_URL=http://localhost:5000
  backend:
    image: backend-image:latest
    ports:
      - '5000:5000'
```

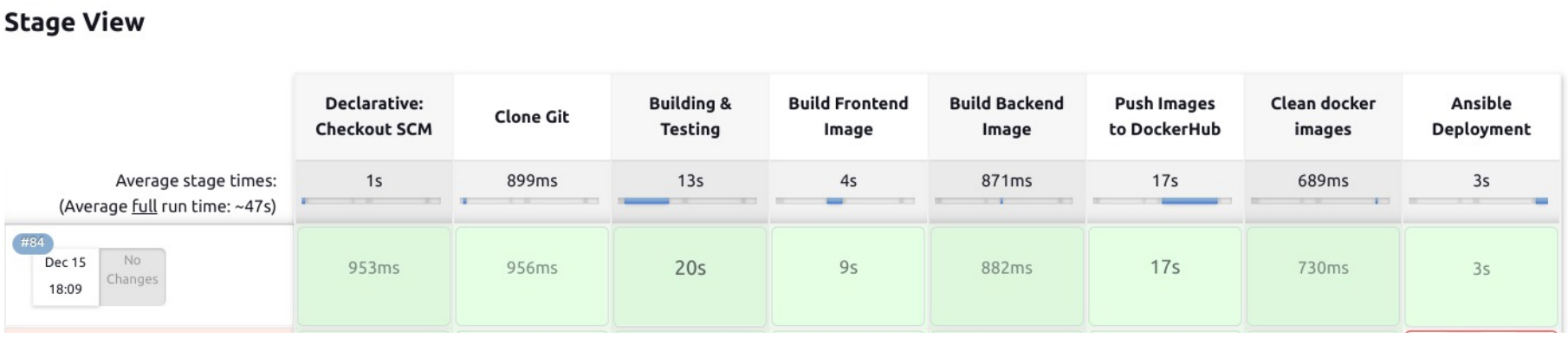
The Docker Compose file orchestrates the deployment of our RideReady application, comprising frontend and backend services. For the frontend, we use the latest version of the "frontend-image," exposing it on port 3000. The backend, represented by the "backend-image," is set to run on port 5000. This configuration establishes a seamless connection between the two, enabling communication with our MongoDB cloud database. After building and pushing these images to Docker Hub, pulling them from there ensures consistent deployment. When accessing https://localhost:3000, the web application seamlessly operates in the local environment, showcasing the successful integration of frontend and backend components.

To automate the deployment process, we use Ansible with a straightforward playbook. This playbook does two crucial tasks. First, it copies the Docker Compose file into the target environment. Second, it executes the command `docker compose up -d` to launch the services in detached mode, ensuring the application runs persistently. This automation simplifies the deployment workflow, making it easy to maintain and replicate across various environments. Together, Docker Compose and Ansible streamline the deployment of our RideReady application, enhancing efficiency and consistency.

```
version: '3'
services:
  container_front:
    image: frontend-image
    ports:
      - "3000:3000"
    networks:
      - mynetwork
    depends_on:
      - container_back
    environment:
      - REACT_APP_BASE_URL=http://localhost:5000

  container_back:
    image: backend-image
    ports:
      - "5000:5000"
    networks:
      - mynetwork
    extra_hosts:
      - "host.docker.internal:host-gateway"
    environment:
      - MONGO_URL=mongodb://mongodb_db:27017/speDB
      - jwt_secret=eef5f9245c142460c20d70063583558d30f02e88455ee91a9e9d19bd49fb9baf49787e5bd250
      - PORT=5000

  mongodb_db:
    networks:
```

```
      - mynetwork
    image: mongo:latest
    ports:
      - "27017:27017"

networks:
  mynetwork:
```

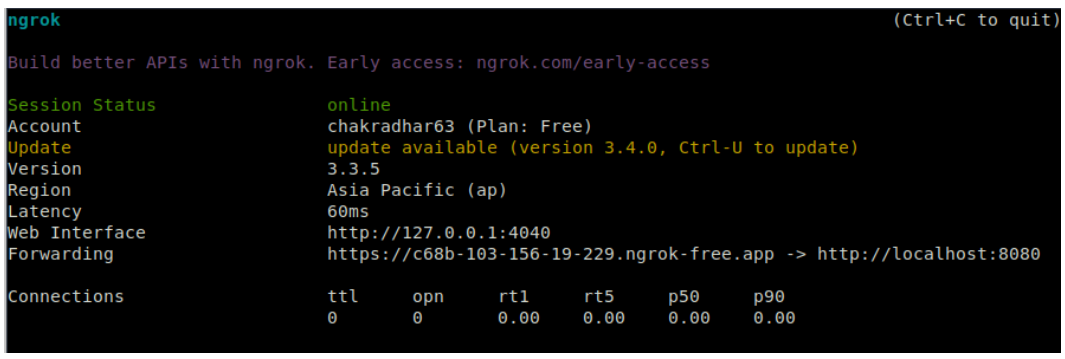
This Docker Compose file orchestrates the deployment of a web application consisting of three services: `container_front` for the front, `container_back` for the backend, and `mongodb_db` for the database using the latest Mongo image. The front end is exposed on port 3000, the back end on port 5000, and MongoDB on port 27017. They all share a common network named `mynetwork` for communication. The frontend relies on the backend (`container_back`) and specifies its base URL as "http://localhost:5000". The backend uses MongoDB with a connection string, and both the frontend and backend share a secret for JSON Web Token (JWT) authentication. The `depends_on` ensures the frontend starts only after the backend is up. Overall, this Docker Compose file streamlines the deployment of a web application with clear communication channels between the frontend, backend, and database.

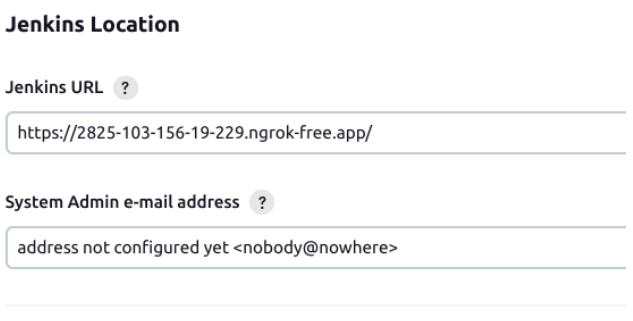



## Configuring GitSCM Polling in Jenkins using ngrok

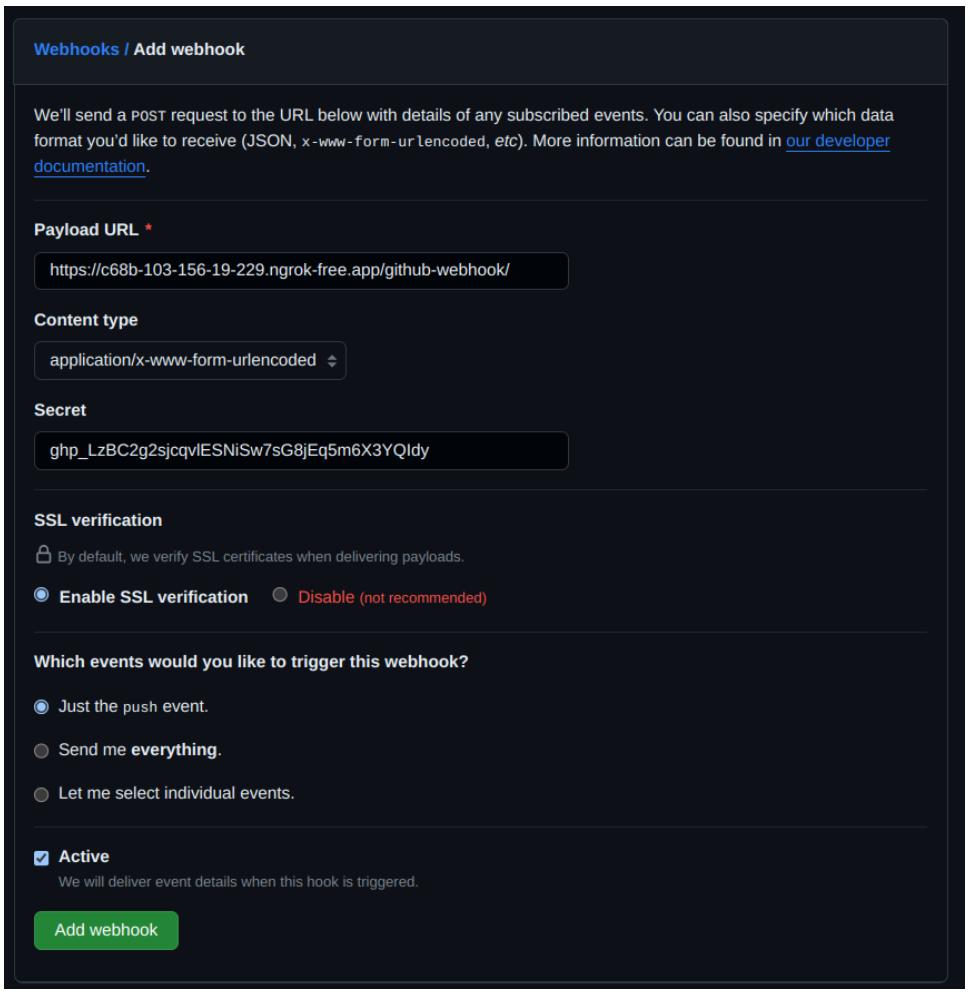
Webhooks are automated messages that kick in when changes happen, such as updates to your GitHub repository. In our case, this webhook setup ensures that the Jenkins pipeline is automatically initiated when changes are pushed to the repository. To enable webhooks, you need to convert your local machine's private IP address into a public one.

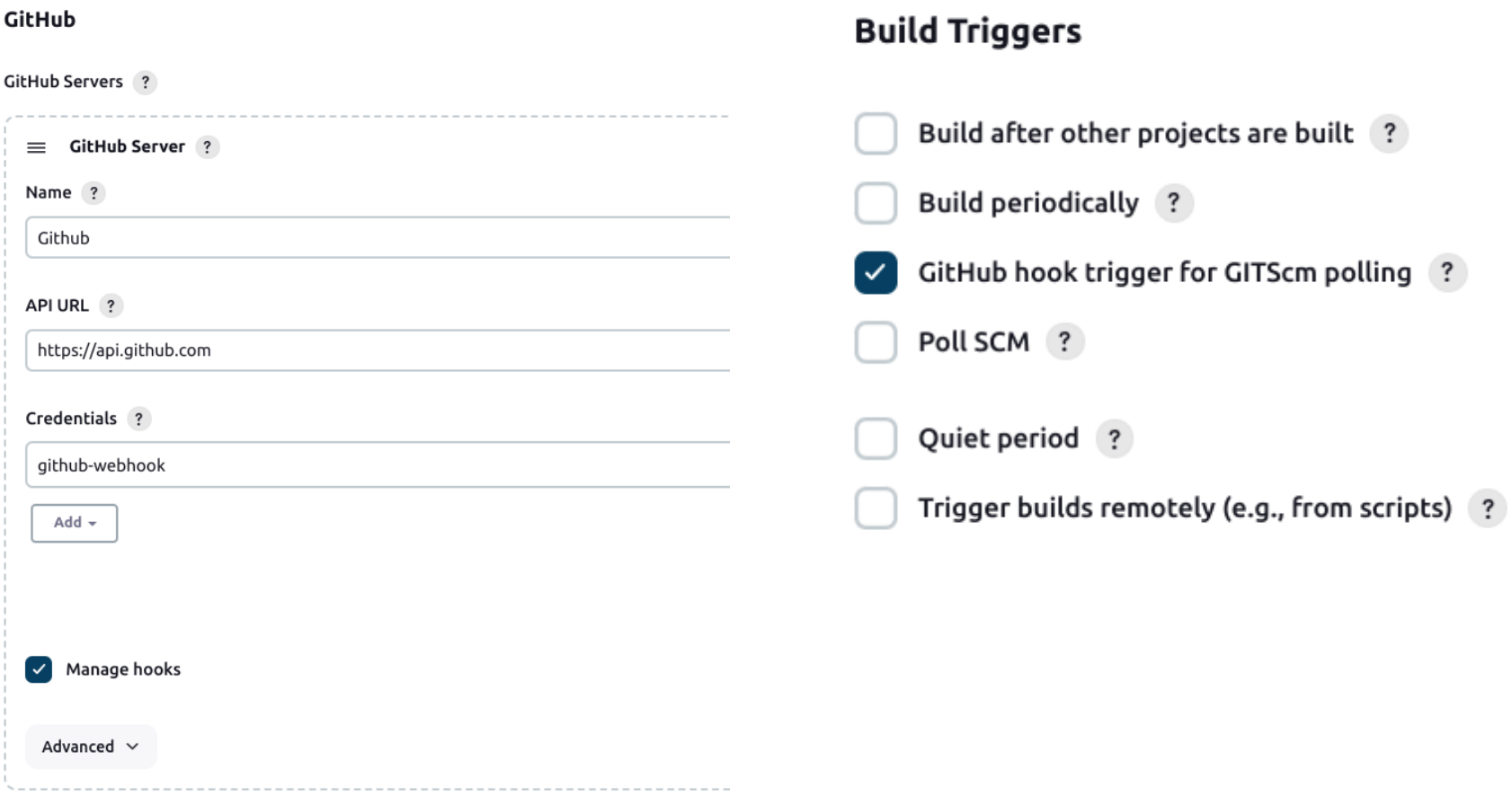
**ngrok** is a tool that establishes secure tunnels from a public endpoint to a web service running on your local machine. This is handy for Jenkins automation with webhooks. We'll use a GitHub personal access token in combination with ngrok to facilitate receiving POST requests from GitHub to Jenkins through the tunnel created by ngrok. To start ngrok, use the command: `ngrok http 8080`.




To set up Webhooks, visit the GitHub remote repository, go to "Settings," then "Webhooks," and select "Add Webhook." Use the 32-character alphanumeric personal access token as the secret. Paste it in this field. The payload URL should be `<ngrok forwarding URL>/github-webhook/`. This configuration establishes the connection between GitHub and your local Jenkins via ngrok.




In Jenkins, navigate to your pipeline settings, and ensure the **GitScm polling build trigger** is enabled. Then, go to the Dashboard, access "Manage Jenkins," and select "Configure System." In the "Jenkins URL" field, insert the ngrok forwarding IP to help Jenkins identify itself publicly. On the same screen, include the GitHub API URL and your credentials. After this setup, when you push changes to GitHub, Jenkins will automatically initiate the build and proceed through the pipeline stages without manual intervention. This automates your workflow.

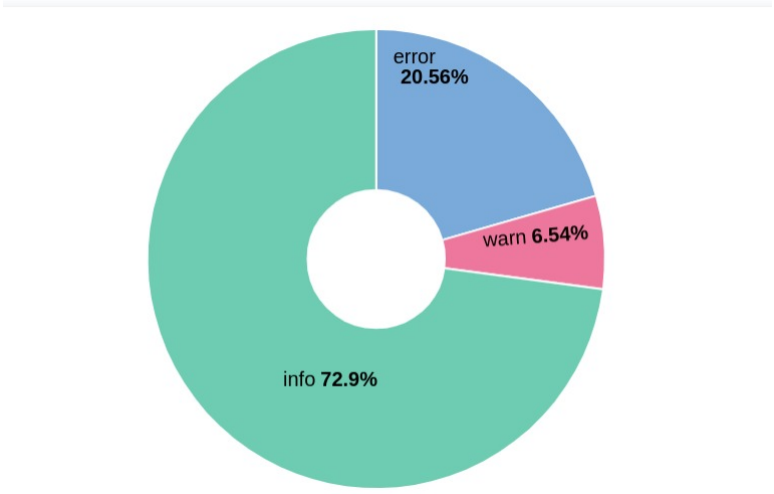
## Log Management:

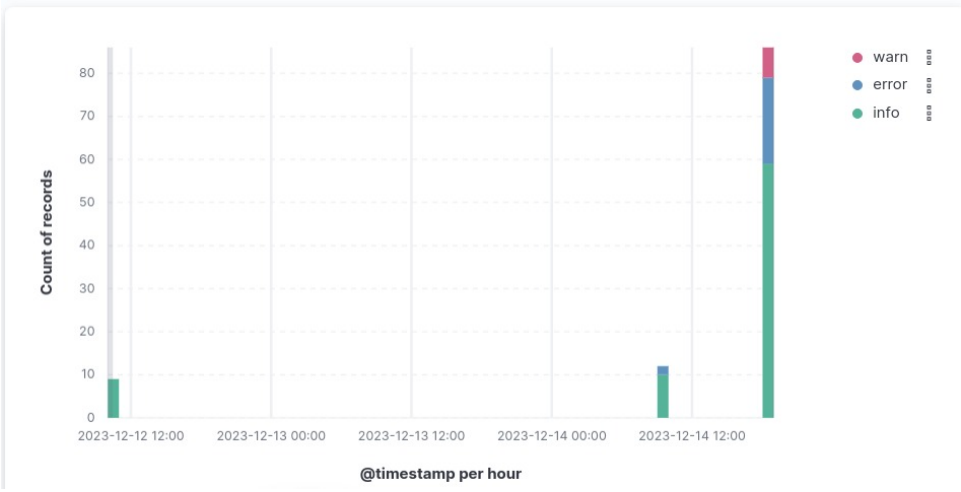
In the final step of our project, we leverage the ELK Stack—composed of Elasticsearch, Logstash, and Kibana—for efficient log management and analysis. Elasticsearch acts as a robust search and analytics engine, storing and indexing vast amounts of log data for easy retrieval. Logstash serves as an Extract, Transform, Load (ETL) tool, collecting, processing, and forwarding logs from diverse sources to Elasticsearch, ensuring data normalization and enrichment. Kibana, our data visualization tool, empowers users to create interactive dashboards and reports based on the stored data in Elasticsearch. This integrated stack allows us to visualize log files effectively, extracting valuable insights, and facilitating comprehensive analysis to enhance our understanding of the project's performance and identify useful patterns or trends.

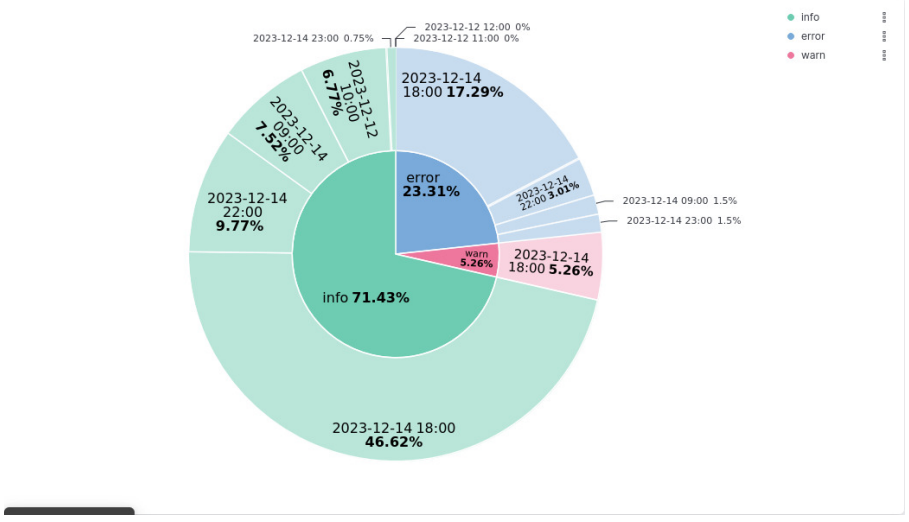
In our server-side code, we utilize **Winston**, a logging library, to generate log files that capture essential information about our application's behavior. Winston allows us to set up a flexible logging system with customizable log formats and levels. The configuration includes options for console logging during development, colorization for readability, and storing logs in a file named `server.log`. This logging setup provides valuable insights into the execution of our server, helping us track events, troubleshoot issues, and monitor the overall health of our application.

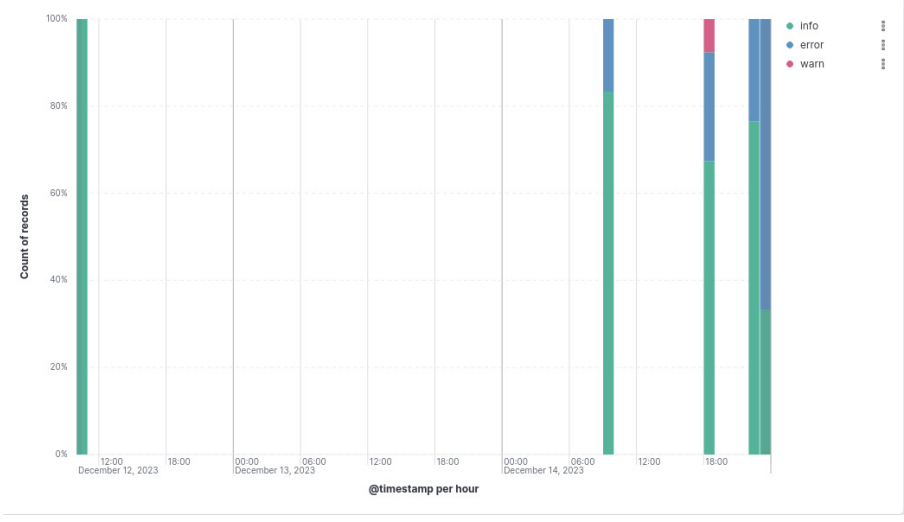
```
[2023-12-14 18:37:19.501] [info] [Success] There are 9 cars available for booking.
[2023-12-14 18:41:17.510] [info] [Success] Car Details Updated Successfully: The information for
[2023-12-14 18:41:17.514] [warn] [Warning] Check the information correctly once again.
[2023-12-14 18:41:18.201] [info] [Success] List Cars Success: All available cars for rent displa
[2023-12-14 18:41:18.295] [info] [Success] There are 9 cars available for booking.
[2023-12-14 18:42:10.665] [info] [Success] Car Details Updated Successfully: The information for
[2023-12-14 18:42:10.665] [warn] [Warning] Check the information correctly once again.
[2023-12-14 18:42:10.666] [warn] [Warning] Check the information correctly once again.
[2023-12-14 18:42:11.378] [info] [Success] List Cars Success: All available cars for rent displa
[2023-12-14 18:42:11.465] [info] [Success] There are 9 cars available for booking.
[2023-12-14 18:42:20.576] [info] [Success] Car Deleted Successfully: The selected car has been s
[2023-12-14 18:42:21.305] [info] [Success] List Cars Success: All available cars for rent displa
[2023-12-14 18:42:21.381] [info] [Success] There are 8 cars available for booking.
[2023-12-14 18:43:52.173] [error] [Failure] Car Details Update Failed: Unable to update the info
[2023-12-14 18:43:54.391] [error] [Failure] Car Details Update Failed: Unable to update the info
```

The application generates logs, which are sent to the ELK stack for analysis. Using this stack, we've created visualizations that provide a clear and organized representation of the log data. These visualizations help us quickly understand and identify patterns, issues, or trends within the application's log information.




## API Documentation

### 1. Endpoint: `/api/users/login`

This endpoint is used for user login, allowing users to authenticate and access secured resources.

**Method:**

- **POST**: To submit user credentials for authentication.

**Request:**

- **Content Type:** `application/json`
- **Body:**
  - **username** (string, required): The unique username of the user.
  - **password** (string, required): The user's password.

**Response:**

- **Status Codes:**
  - **200 OK:** Successful login.
  - **400 Bad Request:** Invalid username or password.
- **Content Type:** `application/json`
- **Body (for successful login):**
  - **token** (string): Authentication token for accessing secured resources.
  - **user** (object): User details.

---

## 2. Endpoint: `/api/users/register`

This endpoint is used for user registration, allowing new users to create an account.

**Method:**

- **POST**: To submit user details for registration.

**Request:**

- **Content Type:** `application/json`
- **Body:**
  - **username** (string, required): The desired username for the new user.
  - **email** (string, required): The email address of the new user.
  - **password** (string, required): The password for the new user.

**Response:**

- **Status Codes:**
  - **201 Created:** Successful user registration.
  - **400 Bad Request:** Invalid request payload.
- **Content Type:** `application/json`
- **Body (for successful registration):**
  - **user** (object): User details.

---

## 3. Endpoint: `/api/cars/getallcars`

This endpoint retrieves information about all available cars.

**Method:**

- **GET**: To retrieve a list of all cars.

**Response:**

- **Status Codes:**
  - **200 OK:** Successful retrieval of car information.
  - **404 Not Found:** No cars found.
- **Content Type:** `application/json`

- **Body (for successful retrieval):**
  - **cars** (array): List of car objects, each containing details like name, capacity, fuel type, and rental information.

---

## 4. Endpoint: `/api/cars/addcar`

This endpoint is used to add a new car to the system.

**Method:**

- **POST**: To submit details for adding a new car.

**Request:**

- **Content Type:** `application/json`
- **Body:**
  - **name** (string, required): The name of the new car.
  - **capacity** (integer, required): The seating capacity of the new car.
  - **fuelType** (string, required): The fuel type of the new car (e.g., Petrol, Diesel).
  - **rentPerDay** (number, required): The rental cost per day for the new car.

**Response:**

- **Status Codes:**
  - **201 Created:** Successful addition of the new car.
  - **400 Bad Request:** Invalid request payload.
- **Content Type:** `application/json`
- **Body (for successful addition):**
  - **car** (object): Details of the newly added car.

---

## 5. Endpoint: `/api/cars/editcar`

This endpoint allows editing the details of an existing car in the system.

**Method:**

- **PUT**: To update details for an existing car.

**Request:**

- **Content Type:** `application/json`
- **Body:**
  - **carId** (string, required): The unique identifier of the car to be edited.
  - **name** (string, optional): The updated name of the car.
  - **capacity** (integer, optional): The updated seating capacity of the car.
  - **fuelType** (string, optional): The updated fuel type of the car.
  - **rentPerDay** (number, optional): The updated rental cost per day for the car.

**Response:**

- **Status Codes:**
  - **200 OK:** Successful update of the car details.
  - **400 Bad Request:** Invalid request payload.
- **Content Type:** `application/json`
- **Body (for successful update):**
  - **car** (object): Updated details of the car.

---

## 6. Endpoint: `/api/cars/deletecar`

This endpoint allows deleting a car from the system.

**Method:**

- **DELETE**: To remove an existing car.

**Request:**

- **Content Type:** `application/json`
- **Body:**
  - **carId** (string, required): The unique identifier of the car to be deleted.

**Response:**

- **Status Codes:**
  - **204 No Content:** Successful deletion of the car.
  - **400 Bad Request:** Invalid request payload.

---

## 7. Endpoint: `/api/bookings/bookcar`

This endpoint allows users to book a car for a specified time slot.

**Method:**

- **POST**: To submit details for booking a car.

**Request:**

- **Content Type:** `application/json`
- **Body:**
  - **carId** (string, required): The unique identifier of the car to be booked.
  - **startTime** (string, required): The start time of the booking in ISO format.
  - **endTime** (string, required): The end time of the booking in ISO format.
- **Status Codes:**
  - **201 Created:** Successful booking of the car.
  - **400 Bad Request:** Invalid request payload or overlapping booking times.
- **Content Type:** `application/json`
- **Body (for successful booking):**
  - **booking** (object): Details of the newly created booking.

---

## 8. Endpoint: `/api/bookings/getallbookings`

This endpoint retrieves a list of all car bookings in the system.

**Method:**

- **GET**: To fetch the list of all bookings.

**Response:**

- **Status Codes:**
  - **200 OK:** Successful retrieval of the list of bookings.
  - **404 Not Found:** No bookings found.
- **Content Type:** `application/json`
- **Body (for successful retrieval):**
  - **bookings** (array): List of booking objects.

---

# Challenges faced during the project

- We encountered numerous challenges when working with Kubernetes, whereas the process was comparatively smoother using Docker Compose.
- We encountered challenges when attempting to utilize Ansible for deploying the application across multiple systems.
- We faced issues when we tried to establish the communication between the front-end image and the back-end image.
- We encountered errors when attempting to use the MongoDB database image instead of utilizing the cloud-based MongoDB database.
- While the back-end testing has been smooth, we came across many errors when we tried to do the front-end testing.
- While the errors mentioned above are some specific ones, we also encountered many smaller issues during the installation and execution of certain software.

## Future scope of the project

- Provide options such as bikes and bicycles to rent.
- Verification of user details such as Aadhar card and driver's license in the app itself.
- Put a filter option so that users can filter the options based on fuel type, company, and price.
- Add a “reviews and ratings” section for every car so that users can get an idea about the car before choosing it.
- Provide push notifications for booking confirmations, reminders, and updates.
- Provide support for multiple languages to cater to a diverse user base.
- Provide a FAQ page and customer support through chat, email, or phone so that it becomes easier for users to resolve their queries.

## References


- **https://docs.docker.com/get-started/02_our_app/**
- **http://stackoverflow.com**
- Class Slides
- **https://react.dev/**