# Project Report: Scalable Static Website

Project: Project: Scalable Static Website with S3 + Cloudflare + GitHub Actions

Author: Mazin

**1. Introduction**

The objective of this project was to implement a modern, serverless, and automated hosting solution for a static website. Traditional web hosting is often manual, requires server management, and is difficult to scale. This project's goal was to build a hands-off CI/CD pipeline using cloud services. This pipeline automatically deploys website updates from a code repository, creating a highly reliable and scalable solution.

**2. Abstract**

This project successfully implemented a Continuous Integration and Continuous Deployment (CI/CD) pipeline for a static website using AWS S3 and GitHub Actions. A static website (HTML/CSS) was stored in a GitHub repository, and an AWS S3 bucket was configured for public website hosting. A GitHub Actions workflow was then developed to automatically detect a git push to the main branch, sync the files from the correct project subfolder to the S3 bucket, and publish the changes live to the internet.

**3. Tools Used**

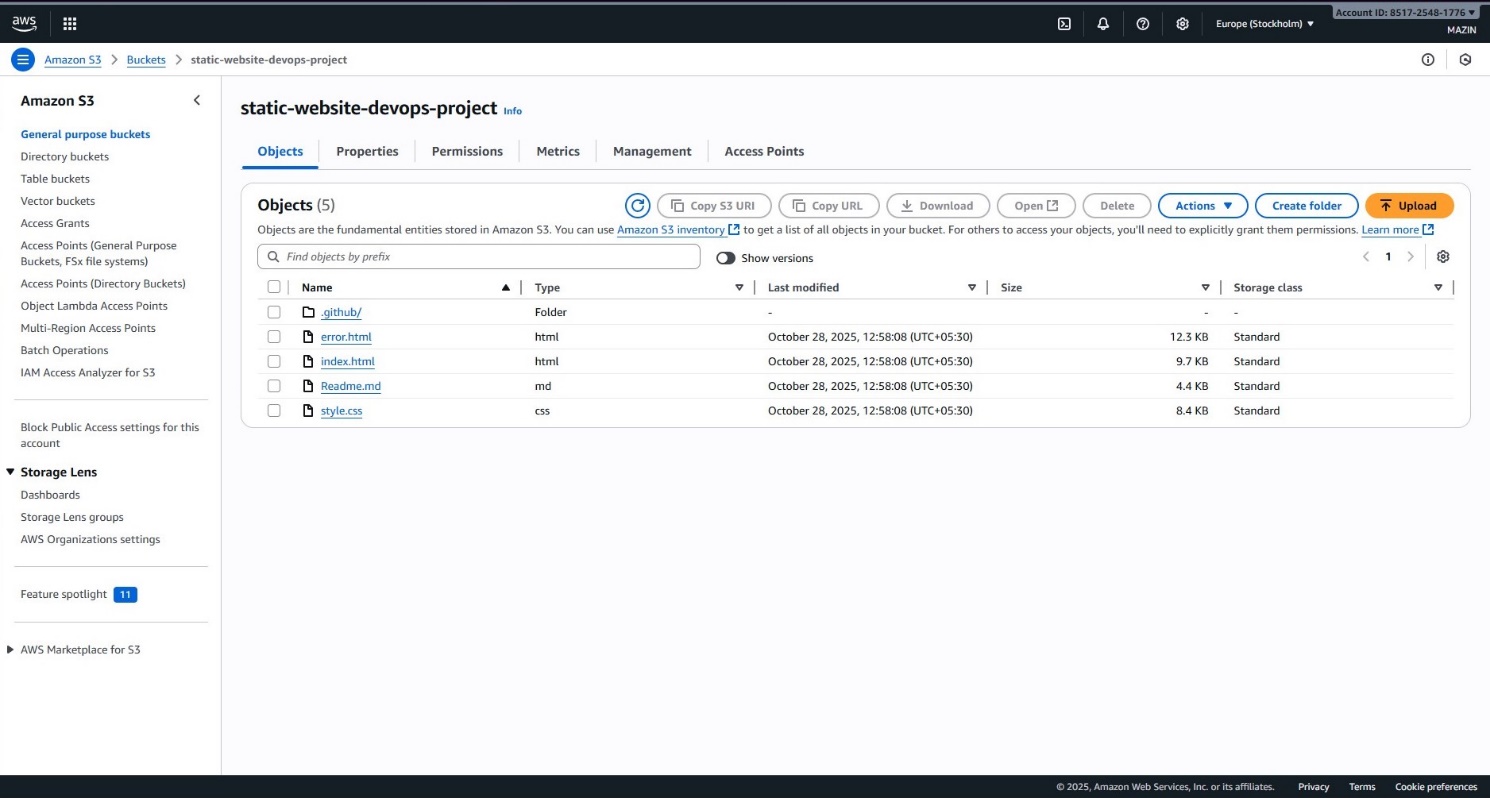
* **AWS S3 (Simple Storage Service):** Used as the serverless hosting service for the static website files.
* **GitHub Actions:** Used as the CI/CD platform to automate the build and deployment process.
* **AWS IAM (Identity and Access Management):** Used to create a secure user with programmatic access, allowing GitHub Actions to communicate with the AWS API.
* **HTML/CSS:** Used for the content of the static website.

**4. Steps Involved in Building the Project**

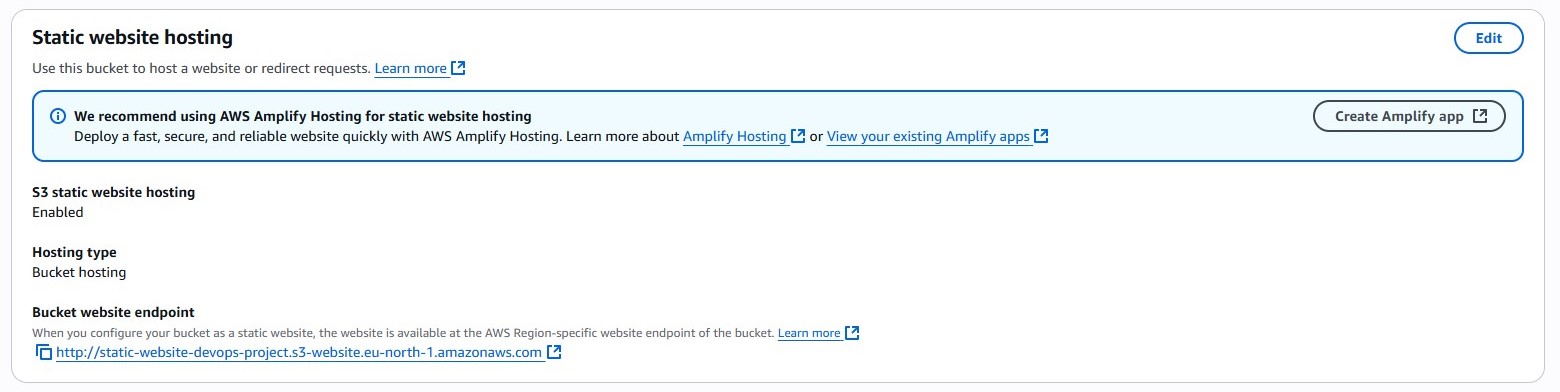
The project was built and validated in the following steps:

**Step 1: S3 Bucket and Hosting Configuration**

An S3 bucket (static-website-devops-project) was created. Static website hosting was enabled, with index.html as the index document.

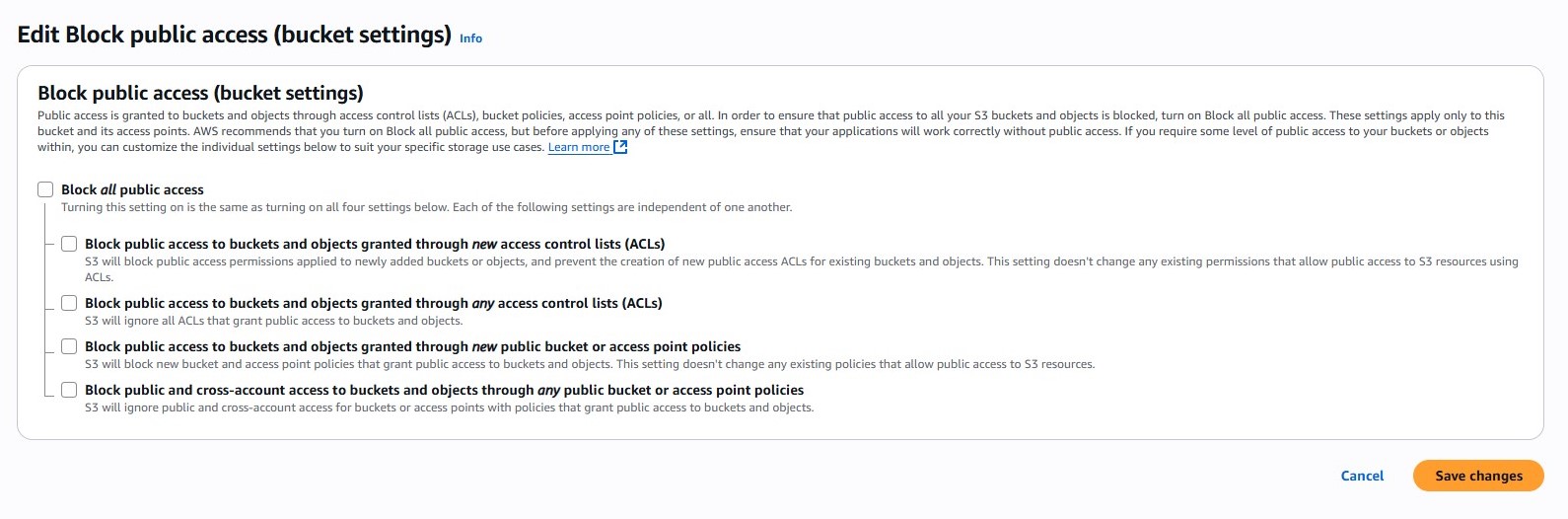
****

**S3 Bucket Files**



**Static Website Hosting Enabled**

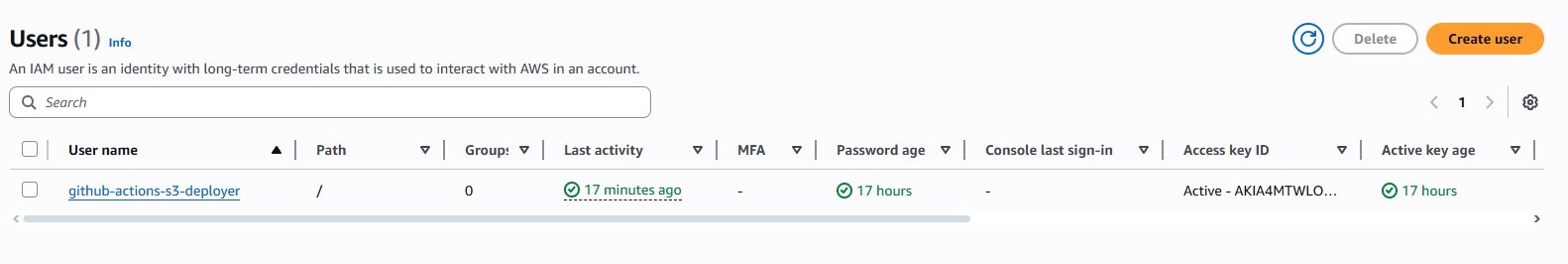
To make the site public, "Block all public access" was disabled, and a specific bucket policy was applied to grant s3:GetObject permissions to all users.



**Public Access Policy**

**Step 2: IAM Security Configuration**

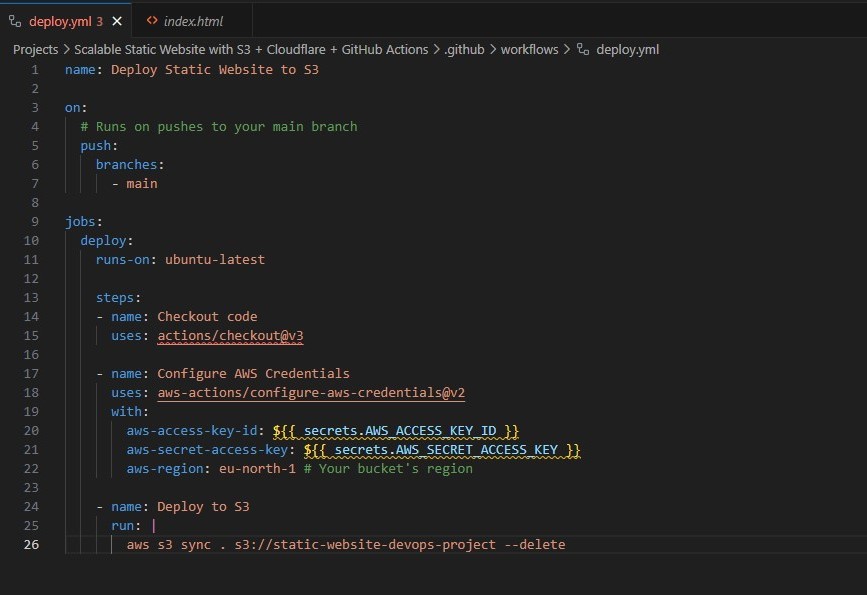
A dedicated IAM user (github-actions-s3-deployer) was created for the GitHub Actions workflow. This user was granted programmatic access and the permissions necessary to sync files to the S3 bucket.

****

**IAM User for GitHub Actions**

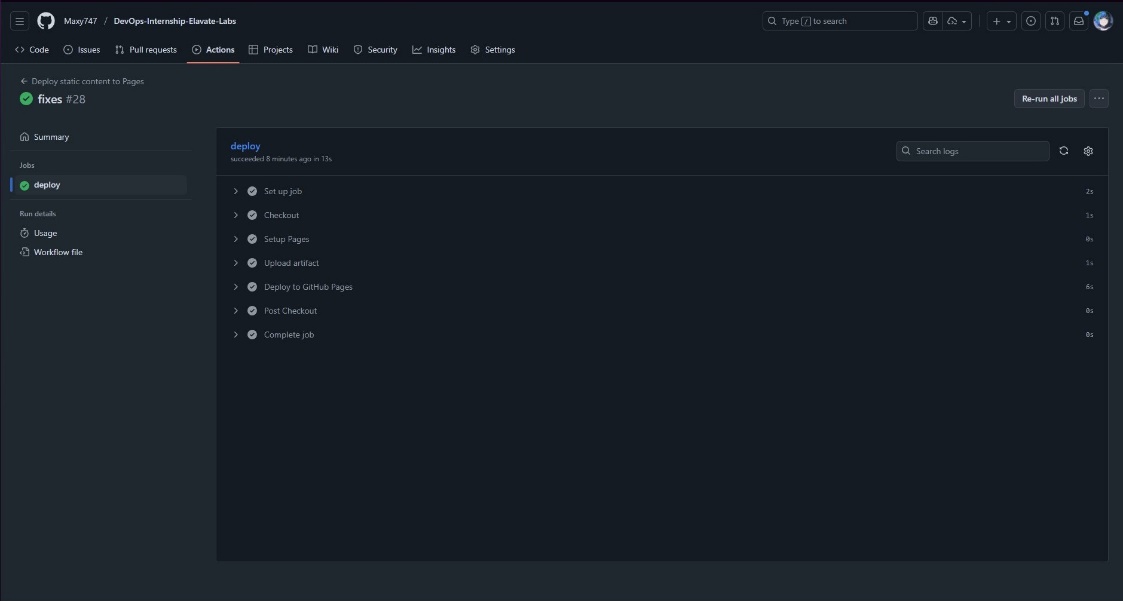
**Step 3: GitHub Actions CI/CD Workflow**

A workflow file (.github/workflows/deployS3.yml) was created in the repository. This file defines the CI/CD pipeline.



**Workflow .yml File**

The workflow was configured to trigger on a push to the main branch. It checks out the code, configures AWS credentials (via stored secrets), and runs the aws s3 sync command. This command was precisely configured to sync *only* the project subfolder ("Projects/Scalable Static Website...") to the root of the S3 bucket.



**Workflow Run in Progress:**

**Step 4: Deployment and Validation**

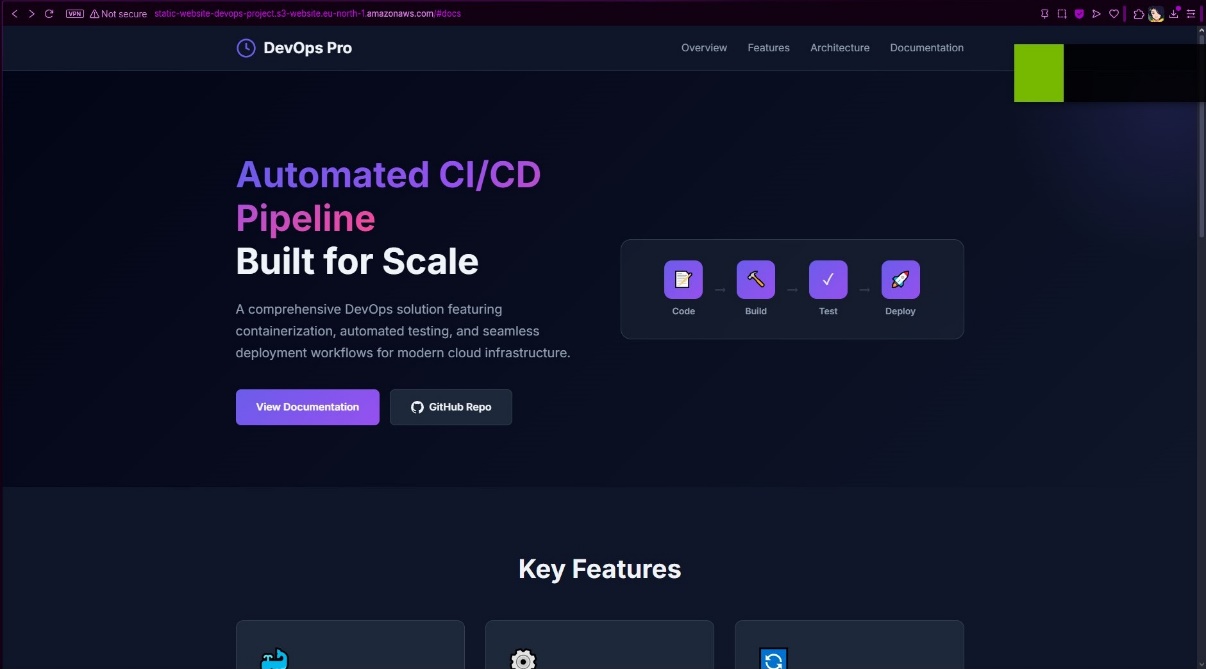
A commit was pushed to the main branch, triggering the workflow. The workflow ran successfully, as shown by the green checkmark.



**Workflow Succeeded**

After the workflow completed, the S3 bucket was correctly populated with the website files.

The successful deployment was validated by accessing the public S3 website endpoint, which correctly displayed the live website.



**Website successfully deployed and live at the S3 endpoint**

**Live Website:** <http://static-website-devops-project.s3-website.eu-north-1.amazonaws.com>

**5. Conclusion**

This project successfully achieved its core objective of creating an automated deployment pipeline for a static website. By integrating GitHub Actions with AWS S3, the manual process of uploading files is eliminated, improving efficiency and reliability. The infrastructure is now fully automated: any git push to the main branch results in a live-staged deployment.

The original project scope also included Cloudflare integration for a global CDN and HTTPS. This step was skipped to focus on the foundational CI/CD pipeline. The next logical step would be to register a free domain and add it to Cloudflare, completing the project's scalability and security goals.

# Project Report: CI/CD Pipeline with GitHub Actions & Docker

**Project:** Project 4: CI/CD Pipeline with GitHub Actions & Docker

**Author:** Mazin

**1. Introduction**

The objective of this project was to establish a fundamental Continuous Integration and Continuous Deployment (CI/CD) pipeline for a containerized application [cite: devops projects.pdf]. Modern software development relies heavily on automation to ensure code quality, consistent builds, and faster deployments. This project demonstrates the core principles of CI/CD using industry-standard tools like Docker for containerization, GitHub Actions for automation, and Minikube for simulating a deployment environment.

**2. Abstract**

This project successfully created a full CI/CD pipeline for a simple Python Flask web application. The application code and associated tests were developed and stored in a GitHub repository. A Dockerfile was written to containerize the application, packaging it with all its dependencies. A GitHub Actions workflow was configured to automatically trigger on code pushes to the main branch. This workflow first runs automated tests (pytest); if they pass, it builds the Docker image and pushes the tagged image to Docker Hub. Finally, Kubernetes configuration files (deployment.yml, service.yml) were created to deploy and expose the containerized application locally using Minikube, simulating a real-world deployment process.

**3. Tools Used**

* **Python:** Programming language used for the web application.
* **Flask:** Micro web framework for Python used to build the simple web server.
* **pytest:** Framework used for writing and running automated tests for the Python application.
* **Docker:** Platform used to containerize the application, ensuring consistency across environments.
* **Docker Hub:** Cloud-based registry service used to store and distribute the built Docker images.
* **GitHub Actions:** CI/CD platform used to automate the testing, building, and pushing of the Docker image.
* **Minikube:** Tool used to run a local Kubernetes cluster for testing deployment configurations.
* **kubectl:** Command-line tool used to interact with the Kubernetes cluster (Minikube).
* **Git:** Version control system used to manage the source code.

**4. Steps Involved in Building the Project**

The pipeline was constructed and validated through the following sequence:

**Step 1: Application Development and Testing**

A simple Flask web application (app.py) was created to serve a "Hello World" message. Dependencies (Flask, pytest) were listed in requirements.txt. An automated test (app\_test.py) was written using pytest to verify the application's basic functionality.

**Step 2: Containerization with Docker**

A Dockerfile was created in the project directory. This file defined the steps to build a container image based on the official Python 3.10 slim image, install dependencies, copy the application code, expose the necessary port, and set the command to run the Flask application.

The image was built locally using docker build -t my-python-app . and tested successfully using docker run -p 5000:5000 my-python-app, verifying accessibility via http://localhost:5000.

**Step 3: CI Pipeline Setup with GitHub Actions**

A workflow file (.github/workflows/project-4-docker-ci-workflow.yml) was created. This workflow configured:

* **Trigger:** On push to the main/master branch, specifically for changes within the project's subfolder.
* **Testing:** Automated execution of pytest after installing dependencies.
* **Docker Hub Login:** Secure login using DOCKER\_USERNAME and DOCKER\_PASSWORD stored as GitHub secrets.
* **Build & Push:** Building the Docker image using the Dockerfile and pushing it to Docker Hub, tagged with the repository name (e.g., mazinmazy/devops-internship-elavate-labs:latest).

**Step 4: CI Pipeline Validation**

Code changes were pushed to the GitHub repository, triggering the workflow. The workflow executed successfully, passing the tests and pushing the image to Docker Hub.

**Step 5: Local Deployment Configuration (Kubernetes)**

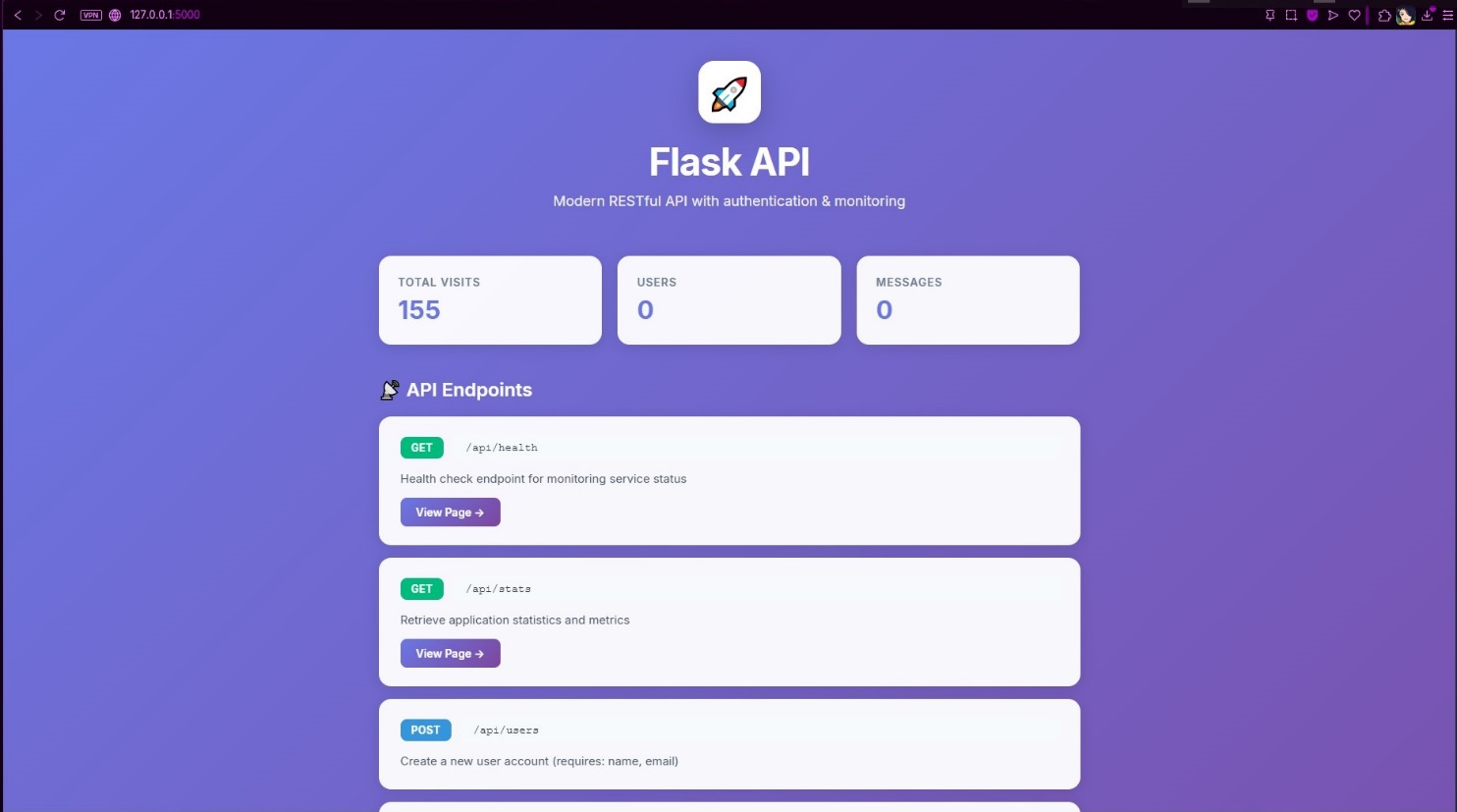
Two YAML files were created for Minikube:

* **deployment.yml:** Defined a Kubernetes Deployment to run two replicas of the application, pulling the correct image (mazinmazy/devops-internship-elavate-labs:latest) from Docker Hub.
* **service.yml:** Defined a Kubernetes Service of type NodePort to expose the Deployment externally, mapping traffic to the container's port 5000.

**Step 6: Deployment Validation with Minikube**

Minikube was started (minikube start). The Kubernetes configurations were applied using kubectl apply -f deployment.yml and kubectl apply -f service.yml. The status of the pods was checked using kubectl get pods, confirming they reached the Running state.

Finally, the application was accessed using minikube service python-app-service, which successfully opened the running application in the browser.

****

**5. Conclusion**

This project successfully demonstrated the creation of a complete CI/CD pipeline for a containerized application. It covered key DevOps practices including automated testing, containerization with Docker, automated image building and pushing with GitHub Actions, and deployment orchestration using Kubernetes (simulated locally with Minikube). The pipeline ensures that code changes are automatically tested and packaged, ready for deployment, significantly improving the reliability and speed of the development lifecycle. This foundational setup can be extended for more complex applications and deployment strategies.