**BULILDING CONTINOUES INTEGRATION USING DOCKERHUB**

**A MINI PROJECT REPORT**

**18CSC316J – ESSENTIALS IN CLOUD DEVOPS**

***Submitted by***

**P NIRANJAN REDDY [RA2111030010177]**

**J SRUJAN CHOWDARY [RA2111030010178]**

**E SAI THORAN [RA2111030010179]**

**P VENKATA RITHWIK [RA2111030010180]**

***Under the Guidance of***

**Dr. R. Radhika**

**Assistant Professor, Department of Networking and Communications**

***In partial satisfaction of the requirements for the degree of***

**BACHELOR OF TECHNOLOGY**

**in**

**COMPUTER SCIENCE ENGINEERING**

**with**

**specialization in CYBER SECURITY**



**SCHOOL OF COMPUTING**

**COLLEGE OF ENGINEERING AND TECHNOLOGY**

**SRM INSTITUTE OF SCIENCE AND TECHNOLOGY**

**KATTANKULATHUR – 603203**

# APRIL 2024

COLLEGE OF ENGINEERING & TECHNOLOGY

SRM INSTITUTE OF SCIENCE & TECHNOLOGY

S.R.M. NAGAR, KATTANKULATHUR – 603203

(Under Section 3 of UGC Act, 1956)

**BONAFIDE CERTIFICATE**

Certified that Mini project report titled **“BUILDING CONTINOUES INTEGRATION USING DOCKERHUB”** is the bonanfide work of **P NIRANJAN [RA2111030010177], J SRUJAN CHOWDARY [RA2111030010178], E SAI THORAN [RA2111030010179], P VENKATA RITHWIK [RA2111030010180].** who carried out the minor project under my supervision. Certified further, that to the best of my knowledge, the work reported herein does not form any other project report or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

SIGNATURE

Dr. R . Radhika

Assistant Professor

Department of Networking

and Communications

INDEX

Introduction \_\_\_\_\_\_\_\_\_\_\_\_2

How does Continuous Integration Works\_\_\_\_\_\_\_\_\_\_\_\_3

Benefits and challenges of continuous integration\_\_\_\_\_5

Key concepts in Contineous Integration\_\_\_\_\_\_\_\_\_\_\_\_\_\_6

Benefits of using GitHub Actions \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_12

Building the CI Pipeline with GitHub Actions\_\_\_\_\_\_\_\_\_15

Testing and Monitoring the CI Pipeline\_\_\_\_\_\_\_\_\_\_\_\_\_21

Demo of the project\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_25

Result & Conclusion\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_31

INTRODUCTION:

Continuous Integration Explained:

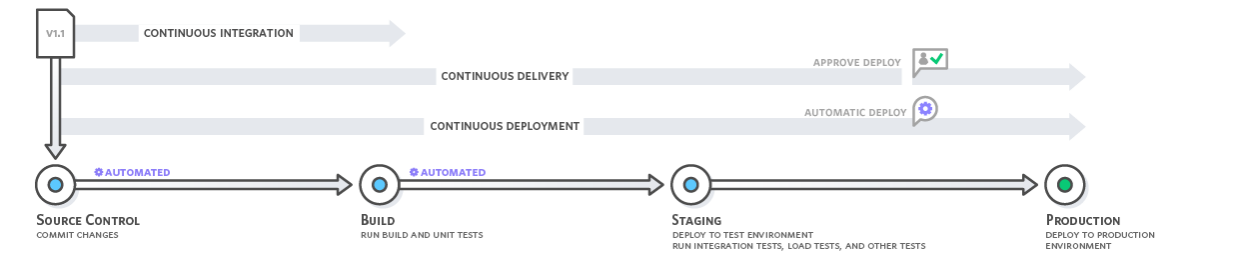
Continuous integration is a Devops software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run. Continuous integration most often refers to the build or integration stage of the software release process and entails both an automation component (e.g. a CI or build service) and a cultural component (e.g. learning to integrate frequently). The key goals of continuous integration are to find and address bugs quicker, improve software quality, and reduce the time it takes to validate and release new software updates.

Why is Continuous Integration Needed?:

In the past, developers on a team might work in isolation for an extended period of time and only merge their changes to the master branch once their work was completed. This made merging code changes difficult and time-consuming, and also resulted in bugs accumulating for a long time without correction. These factors made it harder to deliver updates to customers quickly.

How does Continuous Integration Work?:

With continuous integration, developers frequently commit to a shared repository using a version control system such as Git. Prior to each commit, developers may choose to run local unit tests on their code as an extra verification layer before integrating. A continuous integration service automatically builds and runs unit tests on the new code changes to immediately surface any errors.

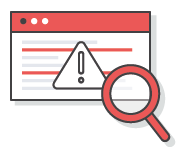


Continuous Integration Benefits



Improve Developer Productivity

Continuous integration helps your team be more productive by freeing developers from manual tasks and encouraging behaviors that help reduce the number of errors and bugs released to customers.



Find and Address Bugs Quicker

With more frequent testing, your team can discover and address bugs earlier before they grow into larger problems later.



Deliver Updates Faster

Continuous integration helps your team deliver updates to their customers faster and more frequently.

Benefits and challenges of continuous integration:

Continuous integration is an essential aspect of DevOps and high-performing software teams. Yet CI benefits are not limited to the engineering team but greatly benefit the overall organization. CI enables better transparency and insight into the process of software development and delivery. These benefits enable the rest of the organization to better plan and execute go to market strategies. The following are some of the overall organizational benefits of CI.

Enable scaling

CI enables organizations to scale in engineering team size, codebase size, and infrastructure. By minimizing code integration bureaucracy and communication overhead, CI helps build DevOps and agile workflows. It allows each team member to own a new code change through to release. CI enables scaling by removing any organizational dependencies between development of individual features. Developers can now work on features in an isolated silo and have assurances that their code will seamlessly integrate with the rest of the codebase, which is a core DevOps process.

Improve the feedback loop

Faster feedback on business decisions is another powerful side effect of CI. Product teams can test ideas and iterate product designs faster with an optimized CI platform. Changes can be rapidly pushed and measured for success. Bugs or other issues can be quickly addressed and repaired.

Enhance communication

CI improves overall engineering communication and accountability, which enables greater collaboration between development and operations in a DevOps team. By introducing pull request workflows tied to CI, developers gain passive knowledge share. Pull requests allow developers to observe and comment on code from other team members. Developers can now view and collaborate on feature branches with other developers as the features progress through the CI Pipeline. CI can also be used to help QA resource expenses. An efficient CI pipeline with high-confidence automated test coverage will safeguard from regressions and ensure that new features match a specification. Before new code is merged it must pass the CI test assertion suite which will prevent any new regressions.

The benefits of CI far outweigh any challenges in adoption. That said, it is important to be aware of the challenges of CI. The real challenges of CI arise when transitioning a project from no CI to CI. Most modern software projects will adopt CI from early inception stages and alleviate the challenges of later adoption.

Here's a are the key concepts in Continuous Integration (CI):

Version Control Integration:

As mentioned earlier, CI relies heavily on Version Control Systems (VCS) like Git. Here's how they work together:

Developers commit their code changes to the VCS repository.

The CI system is configured to monitor the VCS repository for specific events, typically a push or merge of new code.

Upon detecting a triggering event, the CI system retrieves the latest code from the VCS and initiates the build and test pipeline.

Build Automation:

The build stage aims to transform the source code into a working application. This can involve:

Compiling code written in languages like C++ or Java.

Interpreting and packaging code written in scripting languages like Python or JavaScript.

Running build tools like Maven or Gradle to manage dependencies and automate complex build processes.

Testing Automation:

A crucial aspect of CI is automated testing. Different types of tests are commonly used:

Unit Tests: These tests focus on verifying the functionality of individual units of code (e.g., functions or classes).

Integration Tests: These tests ensure different parts of the application work together seamlessly.

End-to-End Tests: These tests simulate real-world user interactions and verify the overall functionality of the application.

Static Code Analysis: This process analyzes the code without executing it, identifying potential issues like coding style violations or security vulnerabilities.

Continuous Feedback Loop:

CI emphasizes a continuous feedback loop for developers:

Code changes are committed to the VCS.

The CI pipeline automatically builds, tests, and analyzes the code.

The CI system provides results (success/failure) and detailed reports to developers.

Developers address any identified issues and commit fixes.

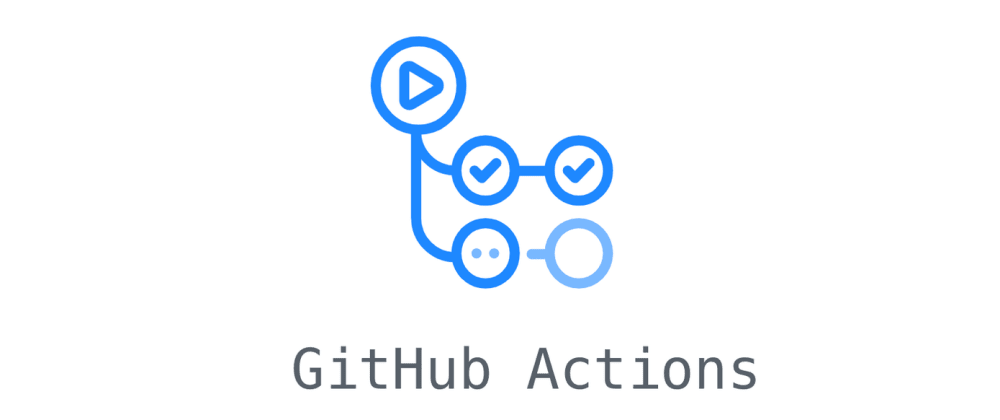
Configuration Management:

To ensure consistency across different environments (development, testing, production), CI often leverages configuration management tools like Ansible or Puppet. These tools automate the configuration of servers and infrastructure, ensuring a consistent environment for building, testing, and deploying applications.

Branching Strategies:

CI can be effectively used with various branching strategies in Git. For instance, feature branches can be used to develop new features in isolation. Merging these branches back to the main branch triggers the CI pipeline, ensuring the new features integrate seamlessly with the existing codebase.

By understanding these key concepts, you can appreciate the power of CI in automating the software development lifecycle. CI promotes early detection of issues, improves code quality, and allows developers to focus on building new features with confidence.



GitHub Actions is a built-in CI/CD (Continuous Integration and Continuous Delivery) platform offered by GitHub. It allows you to automate workflows that run in response to specific events within your GitHub repository. Here's a detailed explanation of its core aspects:

Key Concepts:

Workflows: These are the fundamental building blocks in GitHub Actions. They are defined in YAML files (.yml) stored within the .github/workflows directory of your repository. A workflow typically consists of one or more jobs that execute specific tasks in a defined sequence.

Jobs: These represent individual steps within a workflow. Each job runs in a virtual environment (like Ubuntu or Windows) and can contain multiple steps to be executed sequentially or in parallel.

Steps: These are the atomic actions performed within a job. They can involve:

Using built-in actions provided by GitHub (e.g., checking out code from the repository, running commands, uploading artifacts).

Utilizing custom actions created by the community or yourself for specific functionalities.

Actions: Reusable units of code that encapsulate specific tasks. They can be built-in by GitHub or custom-made. The GitHub Marketplace offers a vast library of actions for various functionalities like building, testing, deploying applications to different platforms, and interacting with external services.

Benefits of using GitHub Actions:

Simple and Integrated: Since it's built directly into GitHub, GitHub Actions offers a user-friendly experience for developers already familiar with the platform. Workflows are defined in code alongside your project, promoting easy collaboration and version control.

Flexibility and Customization: The use of YAML files allows for flexible and customizable workflows. You can tailor them to fit the specific needs of your project, defining complex pipelines with conditional logic, parallel execution, and error handling.

Extensive Action Library:

The GitHub Marketplace provides a rich ecosystem of actions, eliminating the need to reinvent the wheel for common tasks. You can find actions for building various languages, running tests, deploying to different platforms, and interacting with third-party services.

Event-Driven Automation:

Workflows can be triggered by various events within your repository, including pushes to specific branches, pull requests being opened or merged, scheduled executions, or even external events from other services.

Common Use Cases:

Here are some typical scenarios where GitHub Actions shines:

Automated Builds and Tests:

Upon a code push, a workflow can automatically build your application, run unit and integration tests, and report results. This ensures code quality and catches issues early.

Continuous Delivery:

Workflows can be configured to deploy your application to different environments (staging, production) upon successful builds and tests. This enables faster and more reliable deployments.

Static Code Analysis:

Actions can be used to integrate static code analysis tools to identify potential coding style violations, security vulnerabilities, or other issues in your codebase.

Code Review Automation:

Workflows can be triggered by pull requests and run automated checks like linting or code coverage analysis, providing feedback to developers before code is merged.

Project Management Automation:

Actions can be used to automate tasks like creating project boards, assigning issues, or sending notifications based on specific events.

By leveraging GitHub Actions, developers can significantly streamline their software development workflows. From automating builds and tests to facilitating continuous delivery and code reviews, GitHub Actions empowers teams to deliver high-quality software faster and more efficiently.

Building the CI Pipeline with GitHub Actions:

Continuous Integration (CI) is an essential software development practice that automates tasks like building, testing, and potentially deploying code after every change. GitHub Actions is a powerful built-in CI/CD platform within GitHub that allows you to streamline this process. Here's a detailed explanation of how to create a CI pipeline using GitHub Actions:

1. Creating a Workflow YAML File (.yml):

What is a workflow YAML file?

This YAML file is the heart of your CI pipeline in GitHub Actions. It defines the workflow itself, specifying the sequence of steps to be executed, the triggers that initiate the workflow, and the environment where these steps run. YAML (YAML Ain't Markup Language) is a human-readable data serialization format, making it easier to write and maintain your CI configuration.

Where to create the workflow YAML file?

Create a directory named .github/workflows within your project's root directory on GitHub. This is the standard location where GitHub Actions looks for workflow files. Inside this directory, create a new YAML file and name it descriptively (e.g., ci.yml or build\_and\_test.yml).

Structure of the workflow YAML file:

A basic workflow YAML file consists of three main sections:

name: A human-readable name for your workflow (e.g., "CI Pipeline").

on: This section defines the events that trigger the workflow execution. Common triggers include:

push: When there's a push to a specific branch (e.g., main).

pull\_request: When a pull request is opened or merged.

schedule: At specified intervals (e.g., daily).

jobs: This section defines one or more jobs within your workflow. A job is a group of sequential or parallel steps that perform specific tasks. Each job can be further customized with its own execution environment and steps.

2. Workflow Stages: Push, Pull Request:

Defining Stages:

Within your workflow YAML file, you can optionally define stages. Stages help visually organize the different phases of your CI pipeline, improving readability and maintainability. Examples of common stages include:

Checkout:

This stage retrieves the latest code from the version control system (e.g., Git) into the runner environment.

Build:

This stage builds your application, typically involving compiling code or assembling components.

Test:

This stage executes automated tests to ensure your code functions correctly.

Deploy :

This stage can deploy the built and tested application to a staging or production environment.

Push vs. Pull Request Triggers:

The on section of your workflow specifies the events that trigger the pipeline. Here's how Push and Pull Request triggers work:

Push trigger:

When you push code changes to a specific branch (e.g., main), the CI pipeline is automatically triggered. This is ideal for catching regressions (introducing bugs) early in the development cycle.

Pull Request trigger:

When a pull request is opened or merged, the CI pipeline can be triggered. This allows developers to receive automated feedback on their code changes before they are merged into the main codebase. You can choose to run different stages or actions depending on whether the pull request is opened or merged.

3. Common Actions: Checkout code, Run tests, Build application:

GitHub Actions offers a vast library of pre-built actions that simplify common tasks in your CI pipeline. Here's a breakdown of some frequently used actions:

Checkout code (actions/checkout@v3):

This built-in action retrieves the latest code from your GitHub repository and downloads it to the runner environment where your workflow executes.

Run tests (e.g., actions/runner/run@v2):

These actions allow you to execute various types of tests within your pipeline. Popular options include:

Unit testing frameworks like Jest (JavaScript) or JUnit (Java) can be integrated via custom runners.

End-to-end (e2e) testing frameworks like Cypress or Selenium can be used to simulate user interactions and test overall application functionality.

Build application (e.g., actions/build@v2): Actions can be used to automate the build process for various programming languages. Examples include:

Building a Node.js application might involve installing dependencies using npm install and then building the project using npm run build.

Building a Java application might involve compiling the code using tools like Maven or Gradle.

Testing and Monitoring the CI Pipeline:

A robust CI pipeline emphasizes not just building your application, but also rigorously testing it to ensure functionality and quality. Here's a detailed explanation of testing and monitoring within CI pipelines using GitHub Actions:

1. Types of Tests:

Unit Tests: These tests focus on verifying the functionality of individual units of code, typically functions, classes, or modules. They are designed to be small, fast, and highly focused, isolating specific pieces of code for testing. Unit tests help catch bugs early in the development process, improving code quality and maintainability.

Integration Tests:

These tests ensure different parts of your application work together seamlessly. They simulate how components interact with each other, verifying data flow and overall system behavior. Integration tests are typically larger and more complex than unit tests, but they offer a broader view of how your application functions as a whole.

2. Integrating Testing Frameworks with GitHub Actions:

GitHub Actions doesn't directly provide built-in functionality for executing tests. However, it excels at integrating with various testing frameworks through custom runners or dedicated actions available in the GitHub Marketplace. Here's how to integrate popular testing frameworks:

Unit Testing Frameworks:

Jest (JavaScript): The actions/jest@v2 action allows you to easily run Jest tests within your CI pipeline. This action installs Jest dependencies and executes your test suite based on the configuration in your project's package.json file.

JUnit (Java): You can leverage the actions/runner/run@v2 action and a custom script to run JUnit tests. The script would download and install the JUnit runner and then execute your test suite located in the project directory.

End-to-End (e2e) Testing Frameworks:

Cypress: The cypress-io/action@v1 action from the marketplace streamlines running Cypress tests. It sets up the necessary dependencies and executes your tests based on the configuration in your cypress.json file.

Selenium: Integrating Selenium requires more effort as it involves setting up a WebDriver and configuring the environment. However, actions like selenium/selenium-hub@v4 can help with managing the WebDriver container during test execution.

3. Monitoring Pipeline Builds and Results:

Monitoring your CI pipeline is crucial to ensure its smooth operation and identify potential issues. Here's how to stay informed about your pipeline builds:

GitHub Actions Interface:

The "Actions" tab in your GitHub repository provides a clear overview of your workflow executions. You can see the status of each workflow run (success, failure, in progress), view detailed logs for each job and step, and identify any errors that might have occurred.

Notifications:

Configure your workflow to send notifications based on build status. GitHub Actions allows sending notifications on failures, successful builds, or any other state change through email, Slack, or other integrations.

CI/CD Dashboard Tools:

Consider using dedicated CI/CD dashboard tools that integrate with GitHub Actions. These tools provide a centralized view of your CI pipelines across different repositories, offering comprehensive visualizations of build history, success rates, and potential bottlenecks. Popular options include Jenkins X, CircleCI Dashboard, or Azure DevOps Pipelines.

Now we will start the live demo of the project :

Introduction:

In this project, we implemented a Continuous Integration/Continuous Deployment (CI/CD) pipeline for a Python application using GitHub Actions. CI/CD automation facilitates efficient building, testing, and deployment of software applications, enhancing development workflows and overall project management.

Objective:

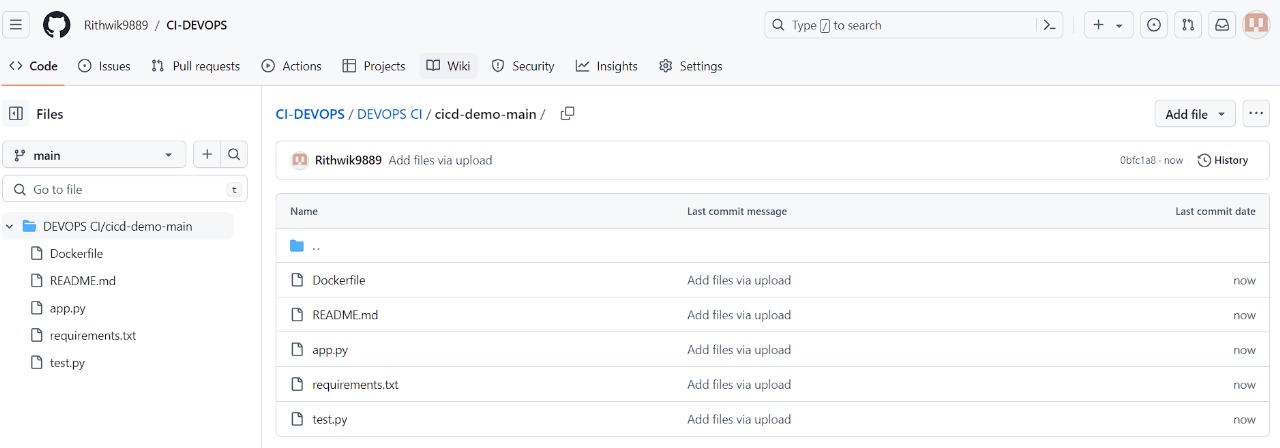
The primary objective of this project was to demonstrate the automation of building, testing, and deploying a Python application using GitHub Actions. The application, a simple Flask-based web service displaying a "Hello, World!" message, was containerized using Docker for streamlined deployment.

Implementation:

1. Application Overview:

- Created a basic Python application utilizing the Flask module to serve a "Hello, World!" message.

- Containerized the application using Docker, ensuring portability and consistency across environments.



3. CI/CD Pipeline Configuration:

- Created a YAML file named "ci-cd-pipeline.yml" to define the CI/CD workflow.

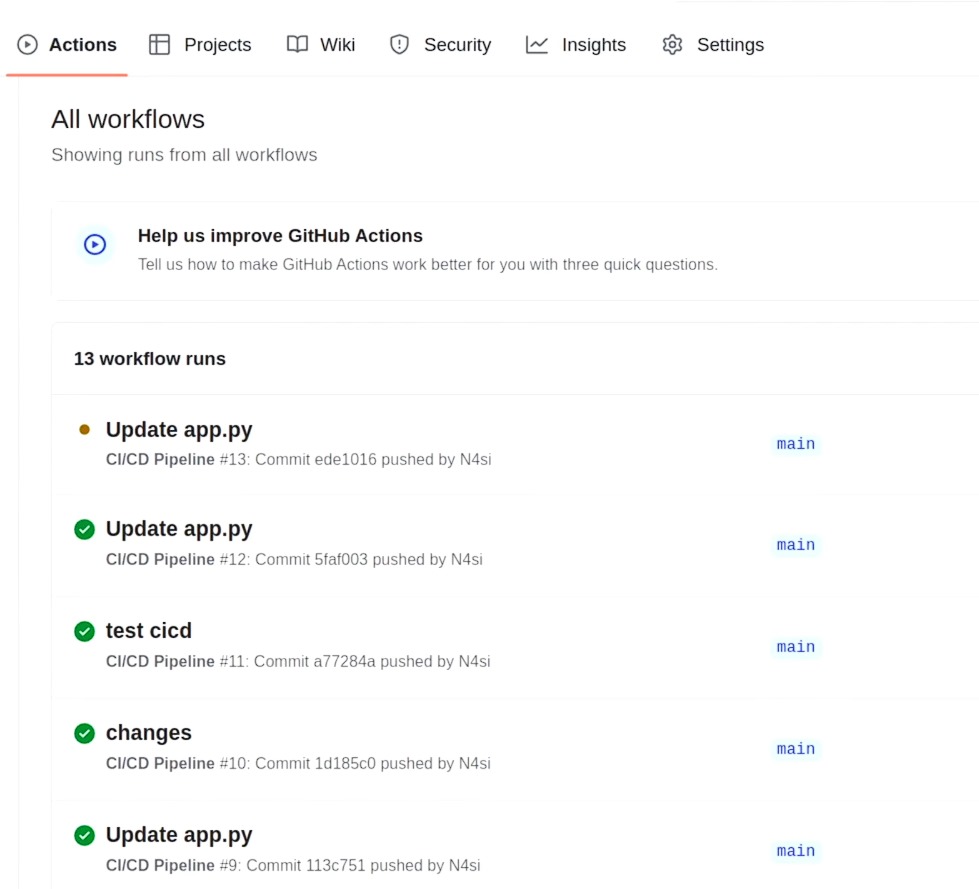
- Configured two jobs within the pipeline: build and test.

- Specified steps for each job, including checking out the code, logging in to Docker Hub, building, tagging, and pushing the Docker image.

4. Testing:

- Implemented a test suite using pytest to ensure the integrity of the application.

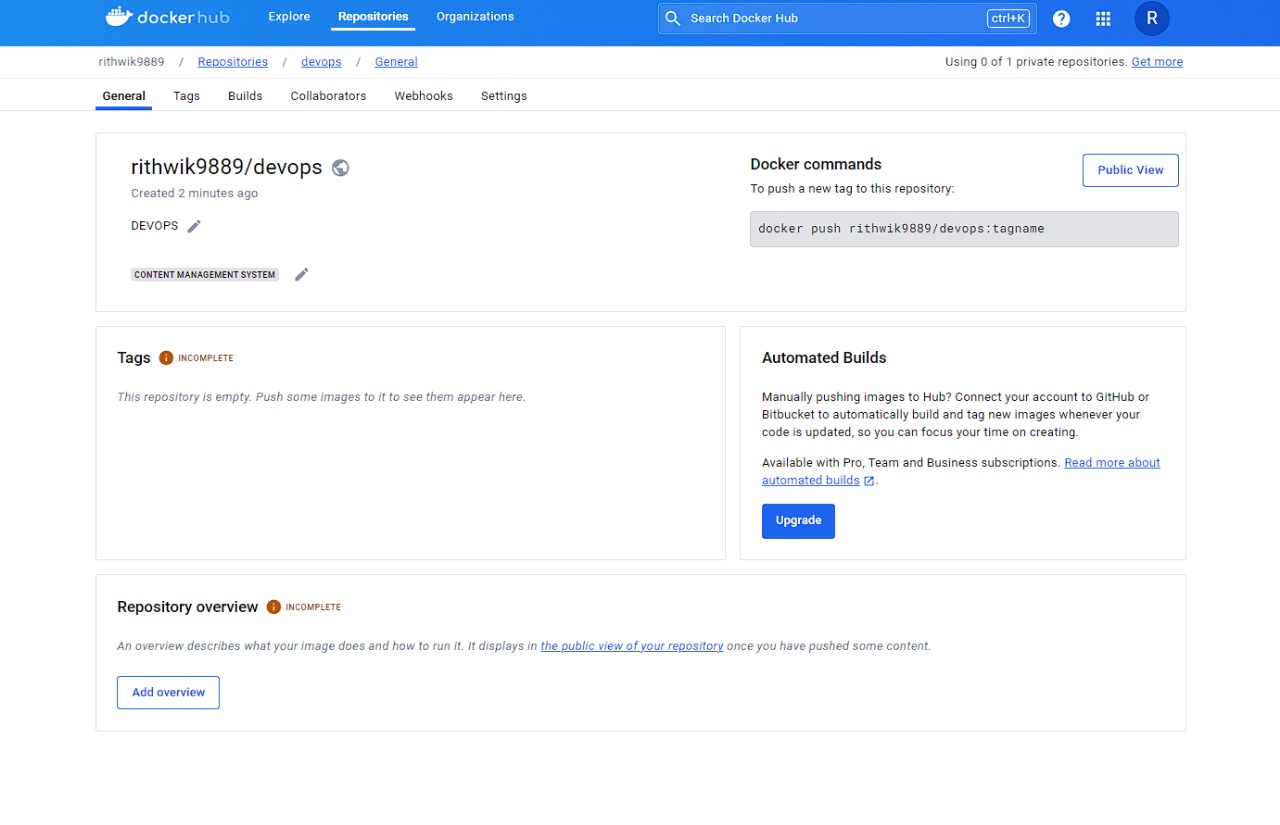
- Created a test file ("test.py") to validate that the message returned by the application is always "Hello, World!" and verifies the HTTP status code.



5. Deployment:

- Automated deployment of the Docker image to Docker Hub upon successful build completion.

- Demonstrated how the CI/CD pipeline reacts to changes in the application codebase, triggering automated builds and tests.



Results:

- Successfully demonstrated the CI/CD pipeline in action, showcasing the automation of building, testing, and deploying the Python application.

- Detected and reported failures in the test stage when the application message was modified, ensuring code quality and consistency.

- Received email notifications for pipeline failures, enabling prompt resolution of issues by developers.



Conclusion:

- This project illustrated the practical implementation of CI/CD principles using GitHub Actions for automating software development workflows.

- By integrating CI/CD pipelines into the development process, teams can achieve faster and more reliable software delivery while maintaining code quality and consistency.

- Future iterations of the project could explore advanced CI/CD practices, such as integration with infrastructure automation tools like Terraform and Ansible, to further streamline the development and deployment process.