AUTOMATION OF DEVELOPING APPLICATION(API) USING DEVOPS(CICD)

A PROJECT REPORT

Submitted by

DHANUSH G [REGISTER NO:211422104100]

EZHIL VELAN S [REGISTER NO:211422104120]

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

PANIMALAR ENGINEERING COLLEGE,

CHENNAI- 600123.

(An Autonomous Institution Affiliated to Anna University, Chennai)

OCTOBER 2024

BONAFIDE CERTIFICATE

Certified that this project report **"AUTOMATION OF DEVELOPING APPLICATION (API) USING DEVOPS (CICD)"** is the bonafide work of "**DHANUSH G (211422104100), EZHIL VELAN S (211422104120)"** who carried out the project work under my supervision.

SIGNATURE

DR.L. JABASHEELA, M.E.,Ph.D.,
HEAD OF THE DEPARTMENT

DEPARTMENT OF CSE,
PANIMALAR ENGINEERING
COLLEGE,
NASARATHPETTAI,
POONAMALLE,
CHENNAI – 600123

SIGNATURE

Mr. M. MAHENDRAN, M. Tech,
ASSISTANT PROFESSOR

DEPARTMENT OF CSE,
PANIMALAR ENGINEERING
COLLEGE,
NASARATHPETTAI,
POONAMALLE,
CHENNAI – 600123

Certified that the above candidates were examined in the End Semester Project Viva-Voce Examination held on.....

INTERNAL EXAMINER

EXTERNAL EXAMINER

DECLARATION BY THE STUDENT

We **DHANUSH G (211422104100), EZHIL VELAN S (211422104120)** hereby declare that this project report titled **"AUTOMATION OF DEVELOPING APPLICATION (API) USING DEVOPS (CICD)"**, under the guidance of **Mr. M. MAHENDRAN, M. Tech**, is the original work done by us and we have not plagiarized or submitted to any other degree in any university by us.

1. **DHANUSH G**

2. **EZHIL VELAN S**

ABSTRACT

The project demonstrates the automation of API application development and deployment using a CI/CD pipeline implemented on Google Cloud Platform (GCP). A Python Flask application, fetching data from the "http://api.open-notify.org/astros.json" open source endpoint, is containerized using Docker and deployed to Google Cloud Run. The entire process, from code commit to deployment, is automated using Google Cloud Build. This approach ensures efficient and reliable deployment of the API, minimizing manual intervention and improving the overall development workflow. The project leverages Docker for containerization, a `*cloudbuild.yaml*` file for CI/CD orchestration, and Cloud Run for serverless deployment.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW

This project demonstrates a modern approach to developing and deploying a web application programming interface (API). The API, written in Python and utilizing the Flask framework, retrieveFs and presents real-time data. The key innovation lies in the fully automated deployment process, leveraging a continuous integration and continuous deployment (CI/CD) pipeline built on Google Cloud Platform (GCP). This automation significantly streamlines development, leading to faster releases, reduced errors, and improved reliability. The project highlights the benefits of DevOps practices for efficient software delivery

1.2 PROBLEM DEFINITION

Manual deployment of web applications is often slow, error-prone, and inconsistent across different environments. This can lead to delays in releasing new features and updates, increased risk of deployment failures, and difficulties in maintaining a consistent production environment. Furthermore, scaling a manually deployed application to handle increased traffic is complex and time-consuming. This project directly addresses these problems by implementing a CI/CD pipeline that automates the entire deployment process, from code commit to cloud deployment. This automation ensures faster, more reliable deployments, and simplifies the process of scaling the application to meet changing demands.

CHAPTER 2

SYSTEM ANALYSIS AND ARCHITECTURE

2.1 EXISTING SYSTEM:

In traditional software development models, application development and deployment, including APIs, follow a manual process that can often be slow and error-prone. Here are the key aspects of an existing system:

1. **Manual Code Integration:**

- Developers work on code in isolation, often leading to large-scale integration challenges at the end of a development cycle.
- Lack of continuous code review and integration leads to "integration hell."

2. **Manual Testing:**

- Testing is usually done manually at the end of the development cycle, leading to delayed feedback and the detection of bugs late in the process.
- Limited automated test cases.

3. **Slow and Manual Deployments:**

- Application (API) deployments are performed manually or with minimal automation, leading to errors, inconsistencies, and slower deployment cycles.
- No version control of deployments or rollback strategies in place, often resulting in downtime during production issues.

4. **Lack of Collaboration:**

- Development, operations, and testing teams work in silos, which increases communication gaps.
- Coordination between different teams happens too late in the lifecycle, causing delays in releases.

5. **Inconsistent Environments:**

- Different environments (development, testing, production) may not be consistently configured, leading to “it works on my machine” scenarios.

2.2 PROPOSED SYSTEM

In a DevOps environment, automating the entire process from development to production ensures faster, reliable, and repeatable deployments. The proposed system focuses on using Continuous Integration (CI) and Continuous Deployment (CD) to streamline API development and deployment

1. **Automated Code Integration (CI):**

- Developers frequently commit code to a shared repository.
- Automated builds and integration testing happen with every commit, ensuring that code is continuously integrated without conflicts.
- Tools: Git

2. **Automated Testing Pipelines:**

- Continuous Testing ensures every piece of code is tested automatically.
- Unit, integration, and functional tests are automated using CI pipelines, with feedback provided immediately.
- API tests are written to check for correctness, performance, and security at each stage.
- Tools: Postman (for API tests)

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

- Environments (dev, test, production) are automatically provisioned using IaC scripts, ensuring consistency.
- No manual configuration is required for infrastructure, reducing human error.

- Tools: Google Cloud

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

- APIs are automatically deployed to production environments after successful integration and testing.
- Canary or blue-green deployments ensure that changes can be tested in production with minimal risk.
- Version control, rollback, and monitoring are built into the pipeline.
- Tools: Docker , Cloud Build.

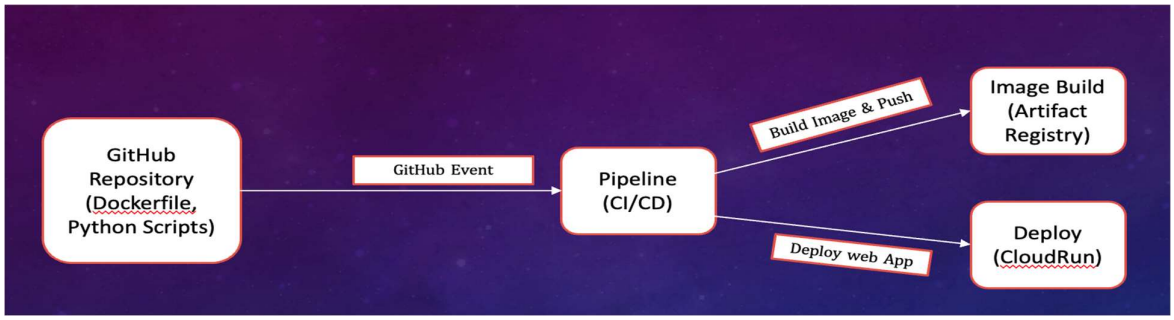
5. **Monitoring and Logging:**

- Automated monitoring and logging ensure real-time tracking of API performance and behavior post-deployment.
- Alerts and automated incident responses can be triggered for any issue in the production environment.

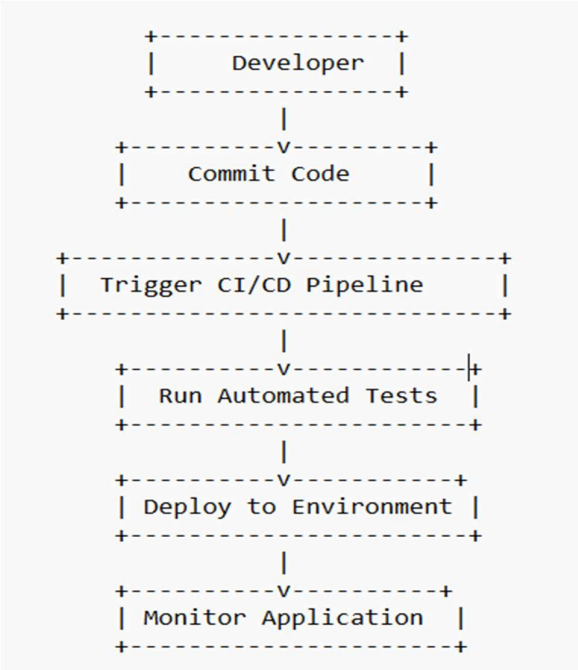
6. **Shorter Release Cycles:**

- Frequent and automated releases lead to faster time to market, allowing features or bug fixes to be deployed in a matter of hours rather than weeks.
- Features can be released in small increments, reducing the risk associated with large releases.

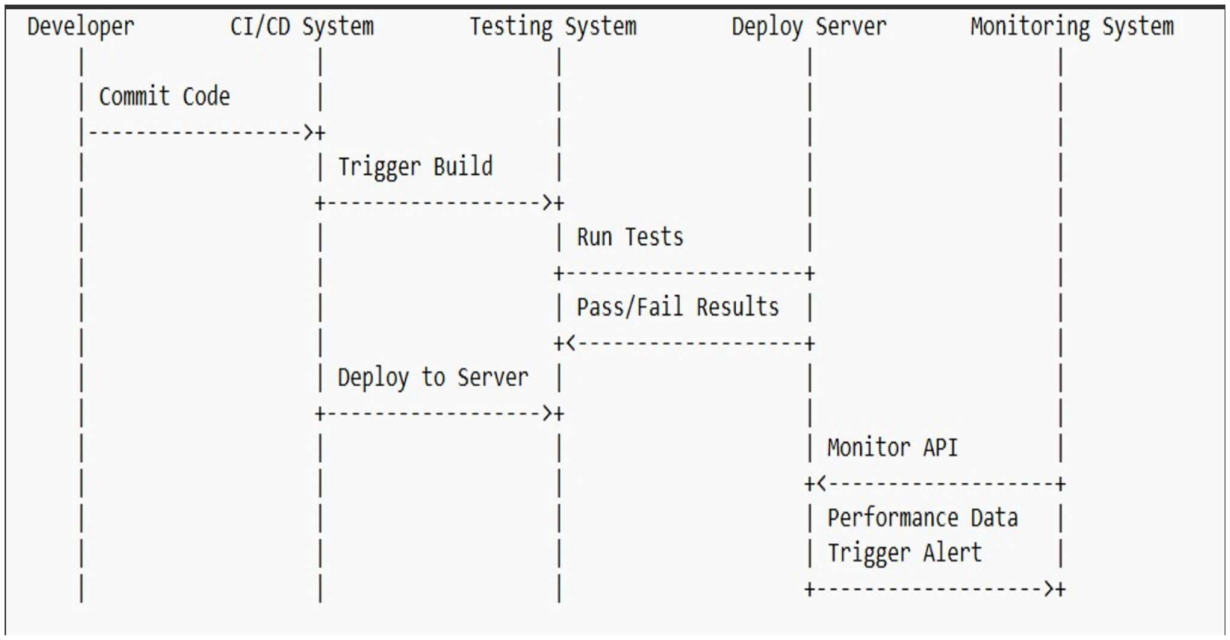
2.3 ARCHITECTURE FLOW DIAGRAM



2.4 USE CASE DIAGRAM



2.5 SEQUENCE DIAGRAM



CHAPTER 3

3.1 WHAT IS CI/CD

CI/CD are essential practices in modern "*Software development*". These practices aim to streamline and automate the processes involved in delivering software, from initial development to final deployment

3.2 CONTINUOUS INTEGRATION (CI)

Continuous integration refers to the practice of regularly integrating code changes from multiple contributors into a shared repository. The primary goal is to detect and address integration issues early in development. Developers commit their code changes to a version control system (such as Git), and an automated build system then compiles the code, performs unit tests, and ensures that the changes do not break the existing codebase**.**

3.3 CONTINUOUS DEPLOYMENT (CD)

Continuous deployment is an extension of continuous integration (CI) that enhances the automation process by automatically deploying every code change that passes the CI process to production. The ultimate goal is to make new features or fixes available to end-users immediately. Continuous deployment requires a robust CI foundation to ensure that only thoroughly tested and validated code updates are automatically deployed.

This approach significantly reduces the time between writing code and making it available to users, enabling organizations to deliver software updates faster and more efficiently.

3.4 WHAT ARE CI/CD PIPELINES?

CI/CD pipelines bridge the gap between continuous integration and continuous deployment. While CI focuses on integrating code changes and validating them through automated testing, CD extends this process by automating the deployment of successful changes to production. Together, they create a seamless and efficient software delivery pipeline.

CI/CD pipelines are the backbone of the CI/CD process. These pipelines are automated workflows that orchestrate the steps involved in building, testing, and deploying software. A CI/CD pipeline ensures that updated code is consistently and efficiently moved from development through testing and staging to production. This automation significantly reduces the likelihood of errors and accelerates the overall delivery process.

3.5 KEY CI/CD COMPONENTS AND STAGES

CI/CD pipelines consist of several key components and stages. These typically include source code repositories, build systems, automated testing frameworks, and deployment mechanisms. The stages of a typical CI/CD pipeline include

Source code management: This stage involves version control systems such as Git, where developers collaborate and manage changes to the source code

Build: The build stage compiles the source code, creating executable artifacts that can be deployed.

Test: Automated testing ensures that the code meets predefined quality standards. This includes unit tests, integration tests, and other forms of testing depending on the project requirements.

Deployment: Once the code has passed all tests, it is deployed to different environments, from development and staging to production.

CI/CD pipelines automate development and deployment processes, reducing manual intervention and the likelihood of human errors. Automation ensures that every change made to the codebase is automatically built, tested, and deployed, fostering a continuous and reliable delivery cycle.

CHAPTER 4

API (Advance Programming Interface)

4.1 WHAT IS API

APIs are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols.

4.2 WHAT DOES API STAND FOR

API stands for Application Programming Interface. In the context of APIs, the word Application refers to any software with a distinct function

4.3 TOOLS AND FRAMEWORK FOR API DEVELOPMENT

* Programming Languages: Python

* API Frameworks: Flask

* API Testing Tools: Postman, Curl and Chrome Browser

CHAPTER 5

PYTHON API APPLICATION

This section details the Python-based API application responsible for fetching and serving data from the Open Notify API (*http://api.open-notify.org/astros.json*). This application is built using the Flask framework, a lightweight and flexible microframework for Python web development. The application's simplicity allows for easy understanding and maintenance, while its functionality provides a practical example of API interaction and data handling

5.1IMPORTING NECESSARY LIBRARIES

The code begins by importing the necessary libraries:

"***from flask import Flask, jsonify***": Imports the Flask class for creating the web application and the ***jsonify*** function for converting Python dictionaries into JSON responses. Flask is a crucial component, providing the structure for handling HTTP requests and generating responses.

import requests: Imports the ***requests*** library, used for making HTTP requests to external APIs. This library simplifies the process of fetching data from the Open Notify API.

import os: Imports the ***os*** module, which is used to access environment variables. This is essential for making the application adaptable to different deployment environments.

5.2 APPLICATION INITIALIZATION

app = Flask(__name__) creates a Flask application instance. ***__name__*** provides the application's name, which is used for various internal purposes by Flask.

5.3 API ENDPOINT CONFIGURATION:

'url = 'http://api.open-notify.org/astros.json' defines the URL of the external API from which data will be fetched. This line demonstrates a key aspect of API design: clearly specifying the data source. This URL points to the Open Notify API, which provides real-time information about astronauts currently in space.

5.4 DEFINING THE API ROUTE (/) AND HANDLING REQUESTS

@app.route('/') is a Flask decorator, associating the **get_astros** function with the root URL (/). This function is executed when a GET request is made to the application's root URL.

The **get_astros** function performs the following actions:

- **response = requests.get(url):** Makes a GET request to the Open Notify API using the **requests** library. This fetches the JSON data from the specified URL.
- **if response.status_code == 200**:: Checks the HTTP status code of the response. A status code of 200 indicates a successful request.
- **data = response.json():** If the request was successful, this line parses the JSON response into a Python dictionary.
- **return jsonify(data)**: Returns the parsed data as a JSON response to the client. The **jsonify** function ensures that the data is correctly formatted as JSON.
- **else: return jsonify({'error': f'Failed to retrieve data: {response.status_code}'}), response.status_code:** If the request failed (status code other than 200), it returns a JSON error message containing the HTTP status code, providing valuable feedback to the client about the failure.

5.5 APPLICATION EXECUTION:

if __name__ == "__main__": ensures that the following code is executed only when the script is run directly (not when imported as a module).

port = int(os.environ.get("PORT", 8080)) retrieves the port number from the environment variable PORT. If the variable is not set, it defaults to 8080. This is crucial for deploying the application to platforms like Cloud Run, where the port number might be assigned dynamically.

app.run(debug=True, host='0.0.0.0', port=port) starts the Flask development server. debug=True enables debugging mode, which is helpful during development but should be disabled in production. host='0.0.0.0' makes the application accessible from all network interfaces, which is necessary for cloud deployments. port=port uses the port number obtained from the environment variable.

CHAPTER 6

DOCKERFILE AND CONTAINERIZATION

6.1 UNDERSTANDING DOCKER AND CONTAINERIZATION:

Docker is a platform that uses containerization to package applications and their dependencies into isolated units called containers. Containers share the host operating system's kernel but have their own isolated filesystem, network, and process space. This ensures consistency and portability, as the application will run the same way regardless of the underlying infrastructure (development, testing, production).

6.2 ANALYZING THE DOCKERFILE:

The Dockerfile provides instructions for building a Docker image. Let's break down each instruction:

FROM python:3.9-slim-buster: This line specifies the base image for the container. python:3.9-slim-buster indicates that the image is based on a slimmed-down version of the official Python 3.9 runtime for Debian Buster. Using a slim image reduces the image size, improving download and deployment times.

WORKDIR /app: Sets the working directory inside the container to /app. All subsequent commands will be executed within this directory. This is a best practice for organizing the container's filesystem.

COPY . /app: Copies all files and directories from the current directory (where the Dockerfile is located) into the /app directory within the container. This includes the app.py, requirements.txt, and any other necessary files.

RUN pip install --no-cache-dir -r requirements.txt: This command installs the Python packages listed in the requirements.txt file. --no-cache-dir prevents pip from using a cache, ensuring that the latest versions of packages are installed.

CMD ["python", "app.py"]: Specifies the command to run when the container starts. This starts the Flask application by executing app.py.

6.3 BENEFITS OF CONTAINERIZATION:

Portability: The containerized application can run consistently across different environments (development, testing, production) without requiring modifications.

Reproducibility: The Dockerfile ensures that the application is built identically each time, eliminating inconsistencies.

Scalability: Containers can be easily scaled up or down based on demand.

Isolation: Containers isolate the application from the underlying system, reducing conflicts and improving security.

Efficiency: Containers share the host OS kernel, making them more lightweight and efficient than virtual machines.

CHAPTER 7

CLOUD BUILD CI/CD PIPELINE (`cloudbuild.yaml`)

This section details the CI/CD pipeline implemented using Google Cloud Build, configured via the cloudbuild.yaml file. This pipeline automates the process of building, testing (implicitly through successful build), and deploying the Python API application to Google Cloud Run. The automation significantly reduces manual effort, increases deployment speed, and minimizes human error.

7.1 PIPELINE STRUCTURE (*cloudbuild.yaml*)

The *cloudbuild.yaml* file defines the steps of the CI/CD pipeline. It's a YAML configuration file that Cloud Build uses to orchestrate the build process. The file is structured into sections:

steps: This section defines a sequence of build steps. Each step is executed sequentially.

options: This section configures options for the entire build process.

timeout: This sets a timeout for the entire build.

7.2 BUILD, PUSH AND DEPLOY STEPS

The steps section comprises three main stages:

- **image-build**: This step builds the Docker image.
- **image-push**: This step pushes the built Docker image to Google Artifact Registry.
- **deploy-cloudrun**: This step deploys the image to Google Cloud Run.
- **options**: logging: **CLOUD_LOGGING_ONLY:** Configures Cloud Build to only log to Cloud Logging.

CHAPTER 8

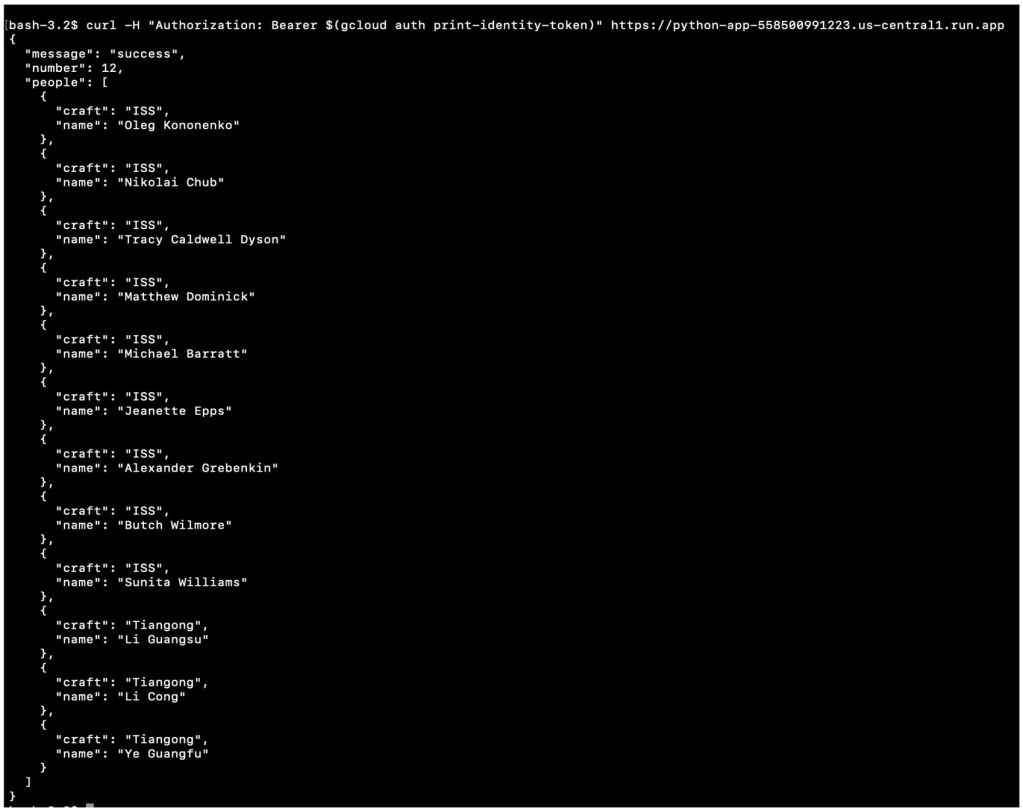
TESTING AND VALIDATION

This section details the methods used to test and validate the deployed API endpoint on Google Cloud Run, ensuring its functionality and accessibility. Three approaches were used: using the curl command-line tool, the Postman API client, and a standard web browser (Chrome). Each method provides a different perspective on testing and validation, ensuring comprehensive coverage.

8.1 USING curl

The curl command-line tool offers a simple and effective way to test the API endpoint. The following command was used

curl -i -H "Authorization: Bearer $(gcloud auth print-identity-token)" -X GET https://python-app-558500991223.us-central1.run.app



Let's break down the command

curl: The curl command-line tool is used to transfer data with URLs.

-i: This flag instructs curl to include the HTTP headers in the output, providing detailed information about the response.

-H "Authorization: Bearer $(gcloud auth print-identity-token)": This adds an Authorization header to the request. **gcloud auth print-identity-token** retrieves an access token from the Google Cloud command-line tool, which is necessary if your Cloud Run service requires authentication. The token is included in the **Authorization** header using the Bearer token scheme.

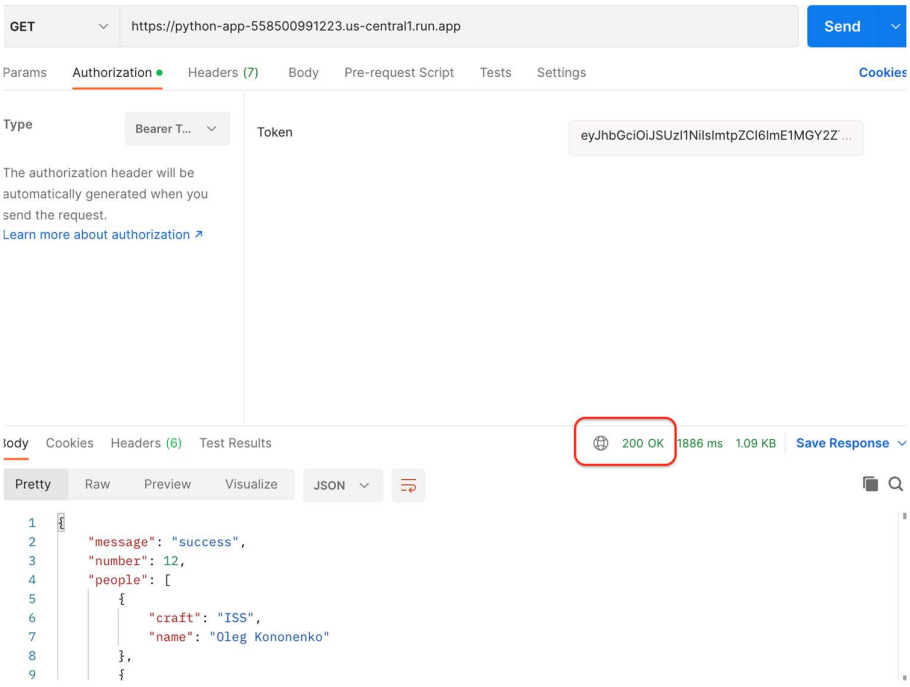
-X GET: Specifies that this is a GET request.

https://python-app-558500991223.us-central1.run.app: This is the URL of the deployed Cloud Run service.

The output of this command should be the JSON response from the API, along with HTTP headers indicating the status code (200 for success) and other relevant information.

8.2 USING POSTMAN

- Postman is a popular API testing tool providing a user-friendly interface. The following steps were performed:
- A new request was created in Postman using the GET method.
- The URL ***https://python-app-558500991223.us-central1.run.app*** was entered.
- An Authorization header was added, specifying the Bearer token obtained using the command ***gcloud auth print-access-token***. This is the same authentication approach as with curl.



A successful response should show an HTTP status code of 200 and the JSON data returned by the API.

8.3 USING A WEB BROWSER (CHROME)

A web browser offers a simple way to test the API's basic functionality, but it's less suited for detailed testing. The URL https://python-app-558500991223.us-central1.run.app was accessed directly in Chrome.

The browser displays the JSON data returned by the API. A screenshot showing the JSON response in the browser included here. This test primarily verifies that the application is accessible and returns data. It doesn't offer the same level of detail as curl or Postman for verifying headers and status codes.


python-app-558500991223.us-central1.run.app
{
"message": "success",
"number": 12,
"people": [
{
"craft": "ISS",
"name": "Oleg Kononenko"
},
{
"craft": "ISS",
"name": "Nikolai Chub"
},
{
"craft": "ISS",
"name": "Tracy Caldwell Dyson"
},
{
"craft": "ISS",
"name": "Matthew Dominick"
},
{
"craft": "ISS",
"name": "Michael Barratt"
},
{
"craft": "ISS",
"name": "Jeanette Epps"
},
{
"craft": "ISS",
"name": "Alexander Grebenkin"
},
{
"craft": "ISS",
"name": "Butch Wilmore"
},
{
"craft": "ISS",
"name": "Sunita Williams"
},
{
"craft": "Tiangong",
"name": "Li Guangsu"
},
{
"craft": "Tiangong",
"name": "Li Cong"
},
{
"craft": "Tiangong",
"name": "Ye Guangfu"
}
]
}

CHAPTER 9

CONCLUSION

The project successfully demonstrated the implementation of a CI/CD pipeline for a Python API application on Google Cloud Run. While the automated deployment process worked effectively, we encountered challenges in deploying in clourun with port. After added port information in app.py, it was deploying successfully. These challenges highlighted the importance of careful planning and configuration when working with cloud-based infrastructure. The project also underscored the value of comprehensive testing, emphasizing the need for robust automated testing in future iterations. This experience provided valuable lessons in cloud deployment best practices and reinforced the importance of thorough planning and testing in a CI/CD environment.

REFERENCES

[1] Google Cloud Platform Documentation: Cloud Run.
[https://cloud.google.com/run] (https://cloud.google.com/run).

[2] Open Notify API Documentation: https://open-notify.org/Open-Notify-API/

[3] Cloud Build: https://cloud.google.com/build/docs/overview

[4] Cloud Artifact Registry: https://cloud.google.com/artifact-registry/docs/overview

[5] Docker Overview: https://docs.docker.com/get-started/docker-overview/