Pet Clinic

Automated Deployment Pipeline

Deirdre Lee

devops T00126583

Contents

[Stage 1: Code and Tools 3](#_Toc164973962)

[Stage 1.1: Summary of Application 3](#_Toc164973963)

[Stage 1.2: Tool Chain 3](#_Toc164973964)

[Stage 1.3: Flow Diagram 4](#_Toc164973965)

[Stage 2: Continuous Integration 4](#_Toc164973966)

[Stage 2.1: Branching Strategy 4](#_Toc164973968)

[Stage 2.2: Install & Configure Continuous Integration 5](#_Toc164973969)

[Stage 2.3: Automate Build Process 6](#_Toc164973970)

[Stage 2.4: Configure Git Authentication 6](#_Toc164973971)

[Since Git authentication had been previously configured on the project laptop for GitHub, there was no need to set up this step again. 6](#_Toc164973972)

[Stage 2.5: Configure GitHub Actions to send notifications 6](#_Toc164973973)

[Stage 2.6: Unit Test Execution 7](#_Toc164973974)

[Stage 2.7: Code Quality Tools 8](#_Toc164973975)

[Stage 3: Building the Code & Configuring the Pipeline 9](#_Toc164973976)

[Stage 3.1: Task Pipelines 9](#_Toc164973977)

[Stage 3.2: Deploy Application 9](#_Toc164973978)

[Stage 3.3: Build Pipeline for CI Lifecycle 11](#_Toc164973979)

[Stage 4: End-to-End Automation Of The Application Delivery Lifecycle 11](#_Toc164973980)

[Stage 4.1: Docker Installation and Configuration 11](#_Toc164973981)

[Stage 5: Cloud Provisioning and Configuration Management 12](#_Toc164973982)

[Stage 5.1: Resource Provisioning in a Cloud Environment 12](#_Toc164973983)

[Stage 6: Deploying Application 12](#_Toc164973984)

[Stage 6.1: Environment Selection 12](#_Toc164973985)

[Stage 6.3: Container-Based Deployment 13](#_Toc164973986)

[Stage 6.4: Deployment Configuration 13](#_Toc164973987)

[Stage 6.5: Deployment Execution 14](#_Toc164973988)

[In this project, the deployment execution is fully automated through a series of GitHub Actions workflows tailored for seamless integration with Docker. Upon each code push to the main branch or pull request, the workflow initiates a multi-step process. Firstly, the application is built and tested using Maven, with unit test results uploaded for analysis. SonarCloud analyses the codebase. Following successful analysis, the Docker image is built, tagged, and pushed to Docker Hub for distribution. Finally, the Docker container is deployed, with the existing container stopped and removed to ensure a clean environment. The latest image is pulled from Docker Hub, a new container instance is started, and the workflow waits until the container is up and running before completion. This workflow enables efficient and reliable deployment of the PetClinic application, seamlessly incorporating Docker into the CI/CD pipeline. Fig 6.5.1 shows a summary of the pipeline. 14](#_Toc164973989)

[Stage 6.5: Post-Deployment Testing 14](#_Toc164973990)

[References 15](#_Toc164973991)

# Stage 1: Code and Tools

## Stage 1.1: Summary of Application

Pet clinic is an application developed using the Spring Boot framework, which can be built using either Maven or Gradle, with the help of included wrapper scripts. (spring-projects, n.d.) Its web interface consists of a homepage and a navigation bar with three options: the ‘owners/find’ endpoint, where you can add an owner using a form or search for one using the last name; A ‘/vets’ endpoint, listing all vets along with options to view the results as XML or JSON; and an ‘/oups’ endpoint demonstrating the page shown when an exception occurs.

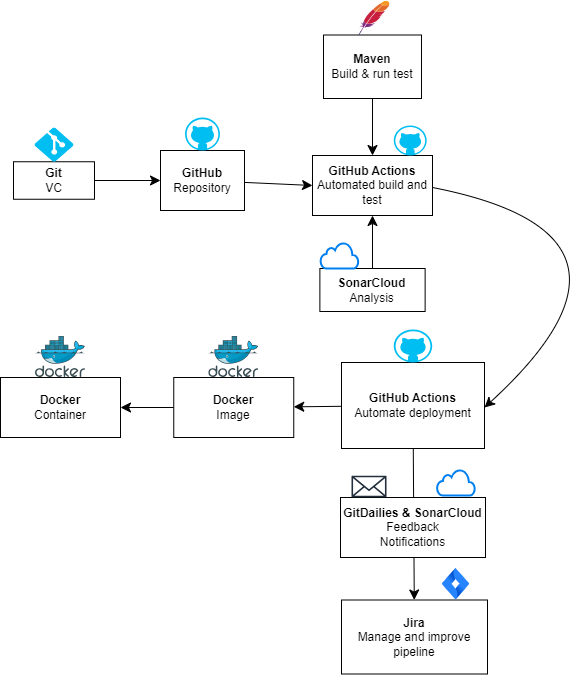
Many components from the Spring stack were used in the construction of this project, including Spring MCV, which provides a model-view-controller architecture, and Spring Data JPA, facilitating interaction with the associated database. (Singh Raina & Giraldo, 2022). Thymeleaf serves as the template engine, used for server-side rendering. Additionally, the in-memory H2 database is used by default to store data during runtime. The Junit framework is used for the unit testing.

The application was sourced from the Spring Projects repository. (spring-projects, n.d.)

## Stage 1.2: Tool Chain

|  |  |
| --- | --- |
|  |  |
| Source Code Repositories | GitHub |
| Build Tools | Maven |
| Continuous Integration | GitHub Actions |
| Resource Provisioning Tools  Containers | Docker |
| Continuous Delivery | GitHub Actions |
| Continuous Deployment | GitHub Actions |
| Continuous Monitoring | SonarCloud |
| Continuous Feedback | Email |
| Continuous Improvement | Jira |

## Stage 1.3: Flow Diagram



# Stage 2: Continuous Integration



## Stage 2.1: Branching Strategy

Before proceeding I decided to set up a branching strategy based on Git Flow. (Thummala, 2023) The main branch will serve as the production-ready codebase, where only stable and thoroughly tested code will be merged. Direct commits will not be permitted to maintain code integrity and stability.

The development branch, on the other hand, will act as the integration point for all project changes, including bug fixes and the addition of new features. A workflow will be implemented on this branch to automate the build and testing process, ensuring that changes are thoroughly validated before integration. A list of branches are shown in Fig 2.1.1.



Fig 2.1.1 Branches list

The introduction of these branches required the setup of a branch protection rule on GitHub to guarantee smooth integration and adherence to the established workflow. The rule entails that a pull request be approved before merging, as well as passing status checks.

## Stage 2.2: Install & Configure Continuous Integration

As I am using GitHub for version control, I decided to use GitHub Actions to implement continuous integration in the project.

GitHub Actions is a CI/CD platform for automating tasks and workflows in GitHub repositories. It uses YAML files to define workflows triggered by events like pushes or pull requests. Actions are reusable applications, covering repetitive or complex tasks. (GitHub, 2024)

It involved creating a workflow file, which automates the testing and deployment of the project whenever changes are pushed to the repository. I first had to create a directory in the project called **.github/workflows** and within this directory created a yaml file called **flow.yml**, as shown in Fig 2.2.1. This file will contain all the steps to be performed as part of a build.



Fig 2.2.1 Setting up CI in GitHub Actions

## Stage 2.3: Automate Build Process

The **maven-build.yaml** file sets up a workflow for automating the build process of the **PetClinic** Java project using Maven. This starter workflow, along with many others, is provided on GitHub. It is triggered by pushes or pull requests to the main branch of the project. The job is called ‘build’ and performs basic build steps, including running the latest version of Ubuntu, checking out the code, and setting up the JDK with Java 17. It caches the Maven packages to speed up future builds using **actions/cache** to cache the **.m2/repository** directory, which is populated based on the **pom.xml** file. The final step runs the Maven wrapper with the package option. This compiles the source code, runs tests, and packages the application into a JAR file. Slight updates to the default file were made to suit the project's needs.

## Stage 2.4: Configure Git Authentication

Since Git authentication had been previously configured on the project laptop for GitHub, there was no need to set up this step again.

## Stage 2.5: Configure GitHub Actions to send notifications

I installed the GitDailies application on GitHub, enabling daily alerts for project status updates. Additionally, I configured two extra alerts to send emails regarding the success or failure of the GitHub Actions workflow. This setup is illustrated in both Fig 2.5.1 and Fig 2.5.2.

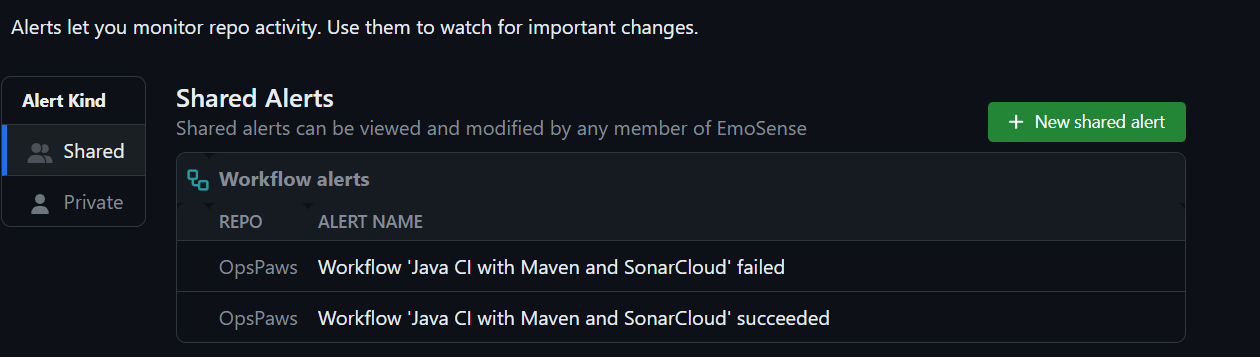


Fig 2.5.1 GitDailies Email Alerts

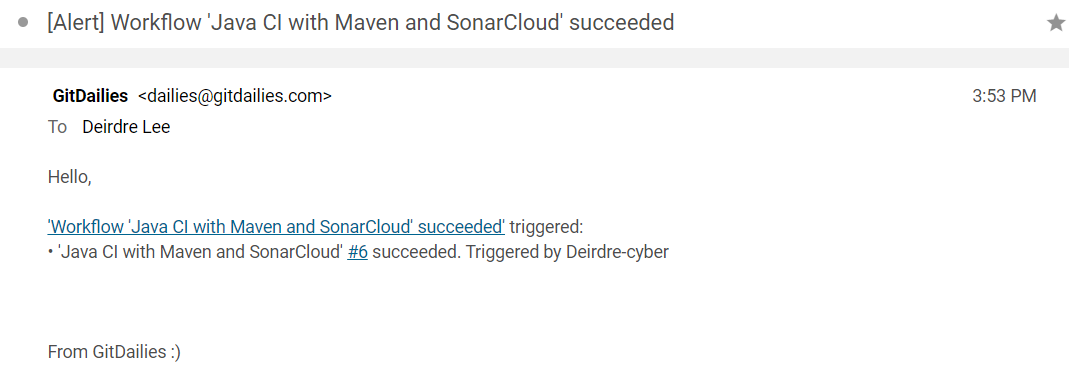


Fig 2 Email alert

## Stage 2.6: Unit Test Execution

Using the Maven wrapper command **./mvn test** I ran the tests, of which 45 were run. Two tests were skipped and the rest passed as can be seen in Fig 2.6.1.



Fig 2.6.1 ‘mvn test’ result output

Following this, I proceeded to enhance the workflow file by incorporating steps to execute unit tests as part of the test job, the outcome of which is displayed in Figure 2.6.2. Additionally, I ensured the generation of a Surefire report, which users can access in the 'Artifacts' section of GitHub Actions, as depicted in Figure 2.6.3. This report is downloadable as a zip file and contains comprehensive test reports, as outlined in Casperson (Casperson, 2023)

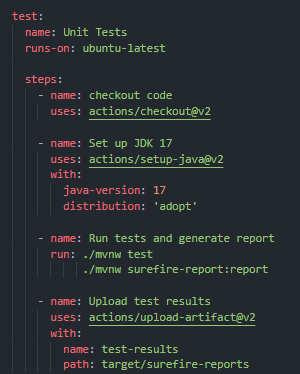


Fig 2.6.2 maven-build.yaml

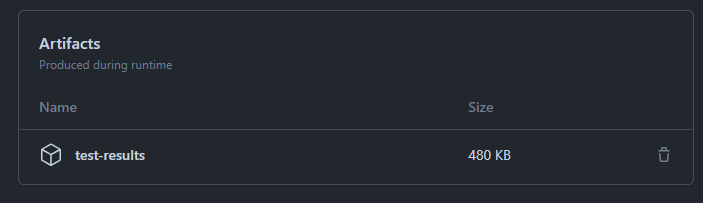
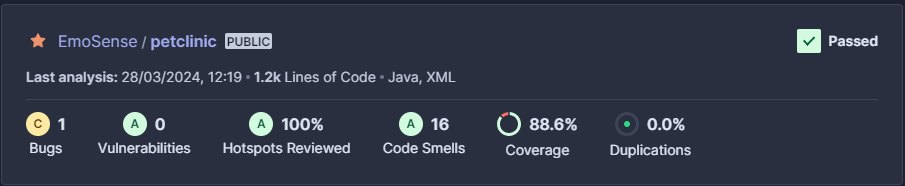


Fig 2.6.3 Artifacts, including test results

## Stage 2.7: Code Quality Tools



I decided to use SonarCloud for code quality analysis and reporting because I have some experience working with it and wanted to further my skills using it. I added the project to my SonarCloud account and updated various settings, including creating a new quality gate for the project. Then, I added a SonarCloud Analysis job to the workflow file, as shown in Figure 2.7.1. This task builds the project, analyses the source code and sends results to SonarCloud.

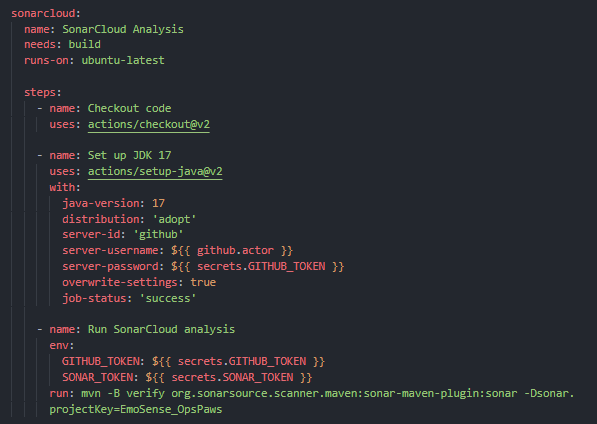


Fig 2.7.1 SonarCloud job

The link to the project in SonarCloud is as follows:

<https://sonarcloud.io/summary/new_code?id=EmoSense_OpsPaws>

An example of a notification email of quality gate pass can be seen in Fig 2.7.2.

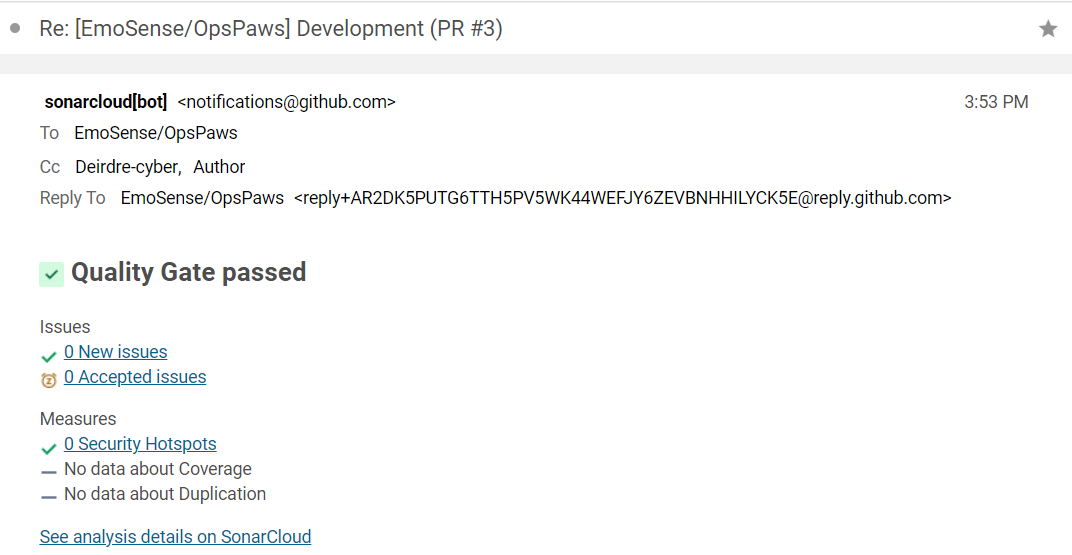


Fig 2.7.2 Email notification

# Stage 3: Building the Code & Configuring the Pipeline

## Stage 3.1: Task Pipelines

Up until now, stages of the Java pipeline have been defined, including the build stage and quality check, using GitHub actions. All that remains for this part of the project is to deploy the application to a test environment.

## Stage 3.2: Deploy Application

I decided to create a Docker image for the web application, push it to Docker Hub, and then deploy the container to Azure as it seemed more efficient, flexible, and cost-friendly. (Hanselman, 2018)

The first step was to create a Dockerfile within the application, configuring the Dockerfile to copy and run the application once the container starts, as shown in Fig 3.2.1. I then built the Docker image, as shown in Fig 3.2.2, to test that everything was working as it should be.

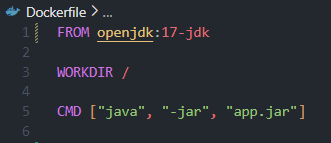


Fig 3.2.1 Dockerfile

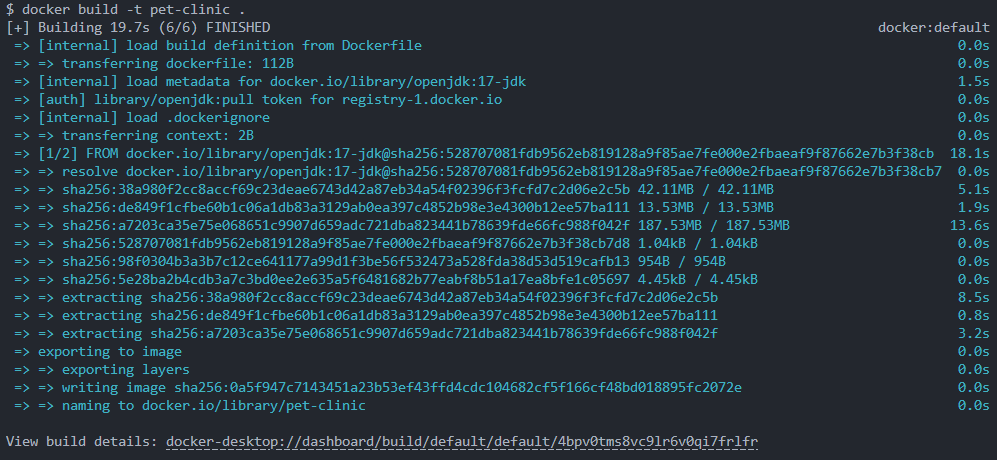


Fig 3.2.2 docker build -t pet-clinic:latest .

I then updated the workflow file to include the docker job. This job includes the following steps: log in to Docker, build the image, tag the image, and then push it to Docker Hub. The job can be seen in Fig 3.2.3. Before completion of this job, it was also necessary to add the relevant secret to the GitHub repository.

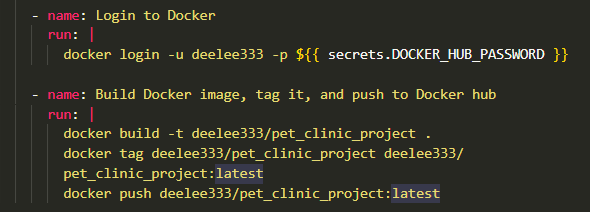


Fig 3.2.3 Docker job

Once I was happy with the results I saved and pushed the changes to Git Hub, which triggered a pull request that began the workflow. Once it passed, it was time to move on to the deployment of the image to a container instance. Job results are shown in Fig 3.2.4.

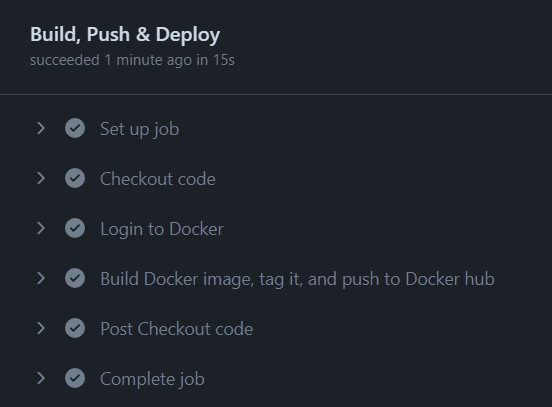


Fig 3.2.4 Successful Docker job

## Stage 3.3: Build Pipeline for CI Lifecycle

In the final workflow file the pipeline is structured as follows:

**Build**

The build job handles building the project with Maven.

**Test**

The test job runs unit tests and generates reports. It depends on the build job.

**SonarCloud Analysis**

The SonarCloud job performs the SonarCloud analysis. It depends on the build job to ensure that the project is built before analysis.

**Docker Image Build and Push**

This job logs in to Docker Hub and then builds, tags and pushes an image to Docker Hub.

**Deploy**

The deploy job pulls the latest Docker image from Docker Hub and then deploys it to a newly created Docker container. It depends on both the test and SonarCloud jobs to ensure that the Docker image being deployed has passed all tests and analysis.

**Post Deploy**

The post\_deploy\_test job performs post-deployment tests, such as pulling the deployed Docker image and running the integration tests. It depends on the deploy job.

# Stage 4: End-to-End Automation Of The Application Delivery Lifecycle

## Stage 4.1: Docker Installation and Configuration

I was unable to deploy the image to an Azure Container as planned as the option is not possible with a student developer account as access to certain services is restricted, including access to credentials. As I was also unsuccessful in gaining credits for Heroku I decided to deploy the image to a local container using Docker Desktop. This was done from the Docker CLI, as shown in Fig 4.1.1.



Fig 4 Create Docker Container

# Stage 5: Cloud Provisioning and Configuration Management

## Stage 5.1: Resource Provisioning in a Cloud Environment

In attempting to deploy the application image to Azure, I encountered a restriction related to permissions as a student developer account, shown in Fig 5.1.1. Despite efforts to push the image directly to Azure, this was hindered by the absence of the necessary permissions. However, I managed to create a resource group locally through the Azure CL. Within this resource group, I successfully instantiated a container instance containing the application image, as shown in Fig 5.1.2 and 5.1.3. Regrettably, I encountered authentication issues when attempting to perform these actions through the workflow file so had to abandon this avenue.

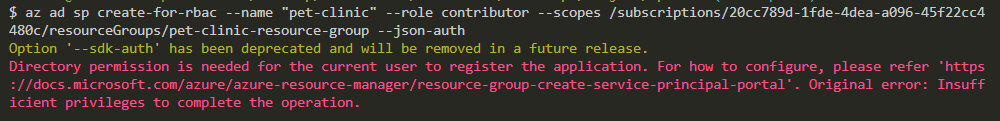


Fig 5.1.1 Credential error

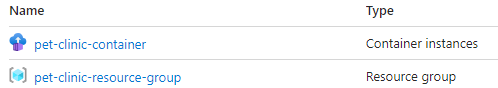


Fig 5.1.2 Resource Group and Container Instance

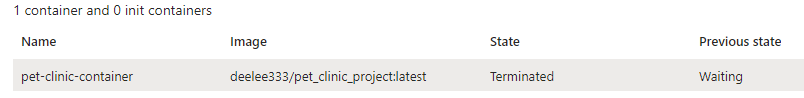


Fig 5.1.3 Container instance including image from docker hub

# Stage 6: Deploying Application

## Stage 6.1: Environment Selection

After researching various container platforms including AWS, Azure, and Google Cloud, I opted to use Docker for its containerisation capabilities and seamless integration with GitHub Actions. Docker's pricing structure offers cost-effectiveness, while its compatibility with GitHub Actions ensures streamlined development and deployment workflows.

## Stage 6.3: Container-Based Deployment

In this container-based deployment process, after building and pushing the Docker image to Docker Hub, a container is created and the image is pulled from Docker Hub. Following the image push, the workflow uses Docker CLI commands to create a new container instance. Subsequently, the latest version of the application is pulled from Docker Hub and deployed into the container. This ensures that the application is readily available for deployment in the containerised environment.

## Stage 6.4: Deployment Configuration

In deploying the project using Docker and GitHub Actions, attention to deployment configurations is an important factor contributing to the success and security of the application. Secret management ensures the confidentiality of sensitive information like authentication tokens and login details, both of which were employed for this project, as shown in Fig 6.4.1.

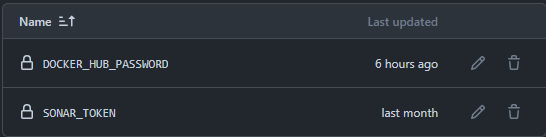


Fig 6 GitHub Secrets

Harnessing the capabilities of GitHub Actions and Docker for containerisation provides a streamlined approach to configuring deployments, enhancing both security and flexibility in the deployment process.

## Stage 6.5: Deployment Execution

In this project, the deployment execution is fully automated through a series of GitHub Actions workflows tailored for seamless integration with Docker. Upon each code push to the main branch or pull request, the workflow initiates a multi-step process. Firstly, the application is built and tested using Maven, with unit test results uploaded for analysis. SonarCloud analyses the codebase. Following successful analysis, the Docker image is built, tagged, and pushed to Docker Hub for distribution. Finally, the Docker container is deployed, with the existing container stopped and removed to ensure a clean environment. The latest image is pulled from Docker Hub, a new container instance is started, and the workflow waits until the container is up and running before completion. This workflow enables efficient and reliable deployment of the PetClinic application, seamlessly incorporating Docker into the CI/CD pipeline. Fig 6.5.1 shows a summary of the pipeline.

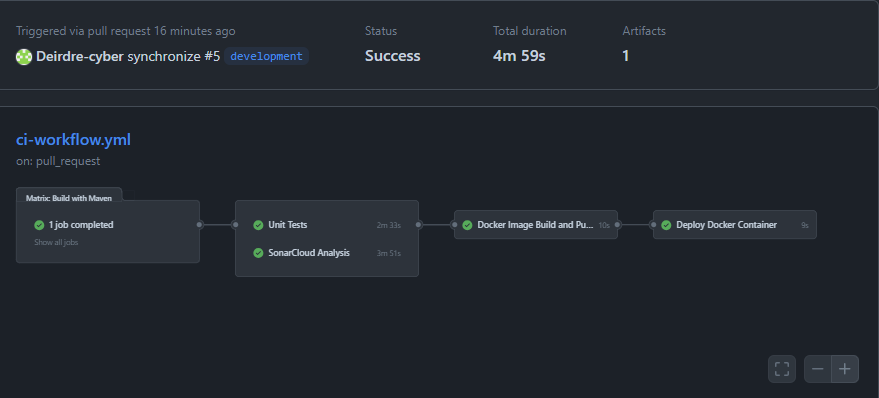


Fig 7 CI/CD Pipeline

## Stage 6.5: Post-Deployment Testing

The post-deployment process ensures the deployed application's functionality in the GitHub environment. In this GitHub Actions workflow, after deploying the Docker container, we pull the latest image and run integration tests locally. These tests, executed using the mvn verify command with the dev profile activated, validate the application's behaviour. The tests confirm that the application is operational and responsive in the deployment environment. If the integration tests pass, signifying the application's correct functioning, the post-deployment test is considered successful.

# References

Casperson, M., 2023. *Running unit tests in GitHub Actions.* [Online]   
Available at: https://octopus.com/blog/githubactions-running-unit-tests  
[Accessed 28 March 2024].

GitHub, 2024. *Understanding GitHub Actions.* [Online]   
Available at: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions  
[Accessed 28 March 2024].

Hanselman, S., 2018. *Penny Pinching in the Cloud: Deploying Containers cheaply to Azure.* [Online]   
Available at: https://www.hanselman.com/blog/penny-pinching-in-the-cloud-deploying-containers-cheaply-to-azure  
[Accessed 15 April 2024].

Singh Raina, A. & Giraldo, J., 2022. *Containerizing a Legendary PetClinic App Built with Spring Boot.* [Online]   
Available at: https://www.docker.com/blog/containerizing-a-legendary-petclinic-app-built-with-spring-boot/  
[Accessed 7 March 2024].

spring-projects, n.d. *spring-petclinic.* [Online]   
Available at: https://github.com/spring-projects/spring-petclinic  
[Accessed 7 March 2024].

Thummala, S., 2023. *Choosing the Right Git Branching Strategy: A Comparative Analysis.* [Online]   
Available at: https://medium.com/@sreekanth.thummala/choosing-the-right-git-branching-strategy-a-comparative-analysis-f5e635443423  
[Accessed 28 March 2024].