CI/CD Strategy

documentation

Nedelchev,Tsanko T.N.

2023

Contents

[Introduction 2](#_Toc138003635)

[Version Control and Branching Strategy 3](#_Toc138003636)

[Testing 4](#_Toc138003637)

# Introduction

Continuous Integration and Continuous Delivery is a set of practices and processes that automate the building, testing and deployment of software applications. It is an essential component of modern day software development methodologies like DevOps and DevSecOps. Continuous integration focuses on the frequent and automated integration of code changes inside a repository. CI helps identify issues that arise sooner which leads to faster bug detection and improved code quality. Continuous Delivery extends the process of Continuous Integration by automating the deployment of software applications to various environments like a test environment or a production environment.

The purpose of this document is to explain my GitHub CI/CD strategy and stages of integration and delivery providing a comprehensive overview of the practices, tools, and processes I have implemented. It should serve as a guide to anyone exploring the GitHub repository of the “Workplace Organization Application”. That may include future development team members, teachers, stakeholders and future contributors.

The GitHub workflow of the “Workplace Organization Application” is separated into multiple stages – Version Control and Branching Strategy, Testing strategy, Building and Deployment strategy, Monitoring and Feedback, Security and Compliance.

# Version Control and Branching Strategy

A screenshot of a computer

Description automatically generated

The woa-app repository has 2 main branches with different purposes.

**Main branch**

The main branch contains the production level code. It only gets updated when the application is ready for a new deployment.

**Dev branch**

The dev branch gets updated when a new feature is completed. I contains the Building and testing pipelines along with security features which will be explained later in this document. The purpose of this branch is to test the code after a new feature in order to determine if the code is viable or not to be pushed into production.

**Feature branches**

Feature branches are not defined specifically. They are created when a developer is working on a new feature. After the developer is finished with the new feature they are working on they can push the changes from their **feature** branch to the **dev** branch where the new code is scanned and tested for vulnerabilities.

# Testing

The testing process is part of the continuous integration process and is mainly conducted in the **dev** branch.

**Sonar Lint**

A screenshot of a computer

Description automatically generated with medium confidence

The process begins during the development process. Using the Visual Studio Code extension for Sonar Lint which seeks code smells and vulnerabilities during the development process and alerts the developer so that they can fix the problem before they even push to the Git repository.

This already eliminates a big part of the code smells and bugs that the project might have or the developer might have introduced during the development process.

After pushing the changes from the feature branch to the dev branch there are 2 sets of GitHub actions that get triggered:

The **SonarQube Scan** **with unit test coverage metrics** action and the **Dockerize** action.

**SonarQube Scan**

A screenshot of a computer

Description automatically generated with medium confidence

This GitHub action contains only one Job called sonarqube.

A screen shot of a computer

Description automatically generated with medium confidence

The job begins with the copying of the repository files. This will serve a purpose during the next steps.

A screenshot of a computer

Description automatically generated with medium confidence

The next step installs the dependencies so that the pipeline can run the project’s unit tests.

A screenshot of a computer screen

Description automatically generated with low confidence

The unit tests are ran with the help of the Jest package and the **jest –coverage** script. This generates a unit test coverage report for SonarQube

A screenshot of a computer program

Description automatically generated with medium confidence

The unit tests are ran successfully and a coverage report id generated

A screenshot of a computer program

Description automatically generated with medium confidence

After the unit tests are completed, a SonarQube Scan is triggered. It uses the repository secrets of SONAR\_TOKEN and the SONAR\_HOST\_URL to connect to a SonarQube deployment that I have setup on an Azure Cloud VM. The rest of the ENV variables are part of the job and serve a purpose for the applications during the unit testing phase.

A picture containing text, font, screenshot

Description automatically generated  
After the scan is complete, a SonarQube report is generated on the SonarQube server instance.

The scan fails because the unit test percentage covered is too low.