Penman, Jack

Devops coursework 2

Contents

[Important links 2](#_Toc27085790)

[Set up 3](#_Toc27085791)

[Jenkins set up 4](#_Toc27085792)

[Testing 7](#_Toc27085793)

[Docker & Docker hub 9](#_Toc27085794)

[Production VM setup with Ansible 11](#_Toc27085795)

[Kuberneties 13](#_Toc27085796)

# Important links

* Github repository: <https://github.com/jackPenman/coursework_2>

Contains all files that were referred to in this report

* Dockerhub link: <https://hub.docker.com/repository/docker/jpenma200/coursework2>

# Set up

The build vm was set up through the azure interface. When it was deployed I Installed docker onto it and from there started both the Jenkins blue ocean container and the soanrQube container. Certain ports had to be opened by adding an inbound security rule to allow me to use the ssh command to access the vm and access the Jenkins and sonar interfaces.

I also forked the supplied repository through the github interface and created a branch to work off of so that I could have one working master version of the code that I don’t break when I am working on the coursework.

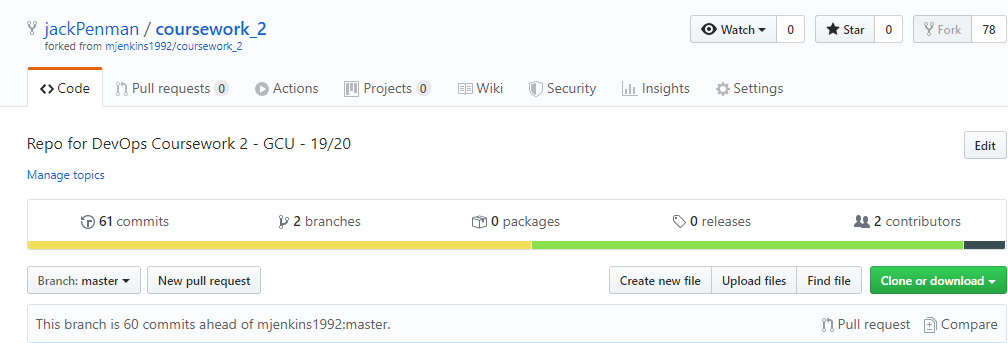


Figure 1Shows the forked github repository

# Jenkins set up

In order to ensure that the build and deployment process was as automated as possible, I decided to use a Jenkins pipeline. A Jenkins pipeline is a process which enables the implementation of continuous delivery to a Jenkins project. This means that implementing a Jenkins pipeline can automate the process of building, testing and deploying the product in a repeatable and reliable manner. In order to facilitate the development of Jenkins pipeline a Jenkins file needs to be created, this is essentially a text file which defines a set of steps that a build needs to follow in order for it to be completed.

In the case of this coursework the pipeline is being used to automatically build deploy etc whenever a push is made to the repository in order to ensure that the deployed code is the most up to date version. In order for this to work. I firstly set up a webhook between the git repository and the Jenkins server. This was done by setting up a ‘GitHub hook trigger for GITScm polling’. This simply sends a json data package to the Jenkins server on a push instructing it to run a build.

In order to implement a Jenkins pipeline a Jenkins File was created and then pushed to the repository. I decided to write this in a declarative format as it is the most recent version which is the most user friendly. A declarative pipeline offers a way to define a Jenkins pipeline by declaring a hierarchy of steps which must be run one after the other in a build.

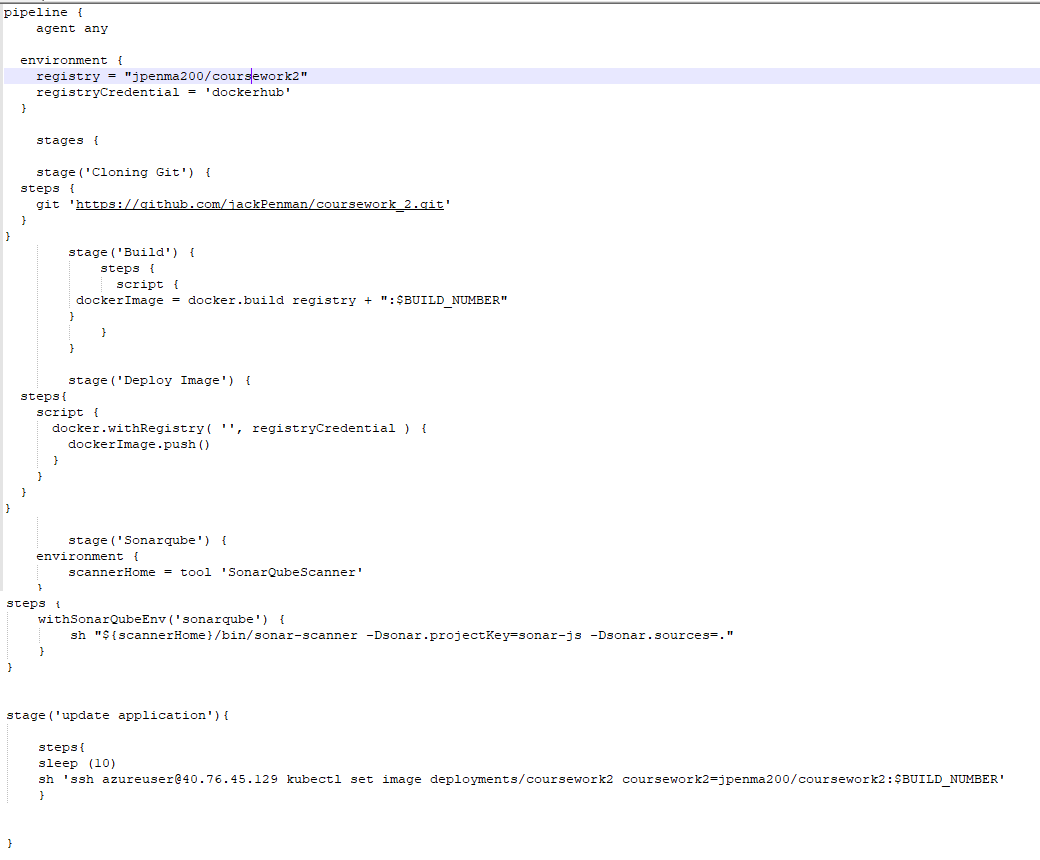


Figure 2 This is my Jenkins file showing all stages of a build



Figure 3 Shows all stages of the pipeline passing

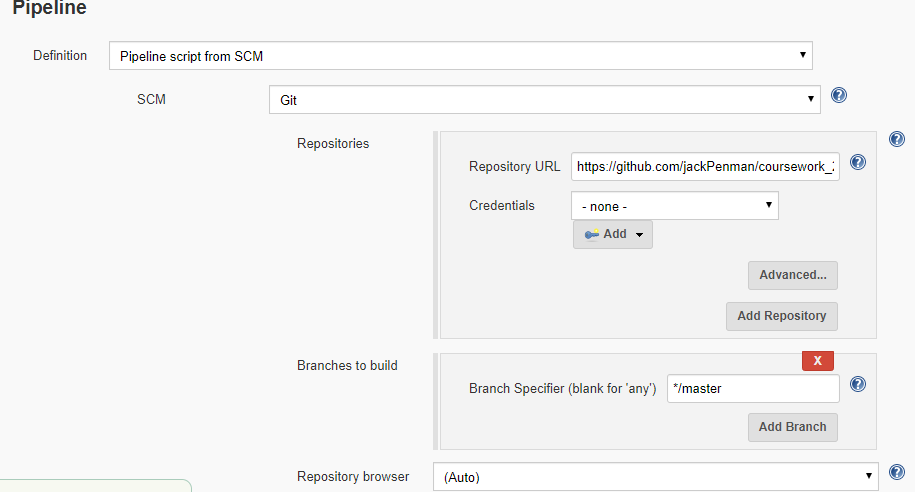


Figure 4 Shows how the jenkin file is being pointed to in the jenkins settings

# Testing

To test the application, I chose to use SonarQube. This is a code analysis tool that is commonly used in the quality assurance stage of a Devops process. It enables continuous code inspection which means that technical debt and potential issues with the code are picked up during a Jenkins build and can be resolved before the application is sent to the liver server.

In order to set up the automated testing of this project I had to first set up and configure SonarQube onto docker. SonarQube is already available on the blue ocean container I decided to use this as my static testing method. I set this up by starting the container and applying the required docker set up of it by installing the needed plugins onto the container. After this I had to install the sonarScaner plugin onto the Jenkins application which would allow me to hook up Jenkins to sonar.

This was done by creating a webhook between the two containers and setting up the authentication so that the webhook wouldn’t be rejected.

After this a step was added to the Jenkins file which firstly defines the sonar server and sonar credentials as variables and uses them to run a sonar command. This part of the Jenkins pipeline defines the parameters needed for a sonar analysis as well, when defining these I made sure to tell it to examine the code in JavaScript otherwise there would’ve been issues in this stage of the pipeline as sonar defaults to a java analysis.

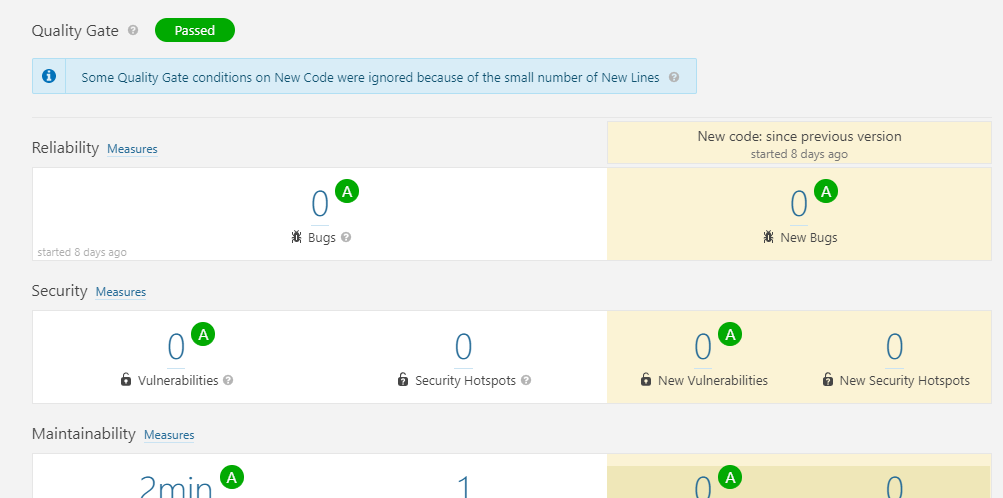


Figure 5 Shows the successful analysis of the code

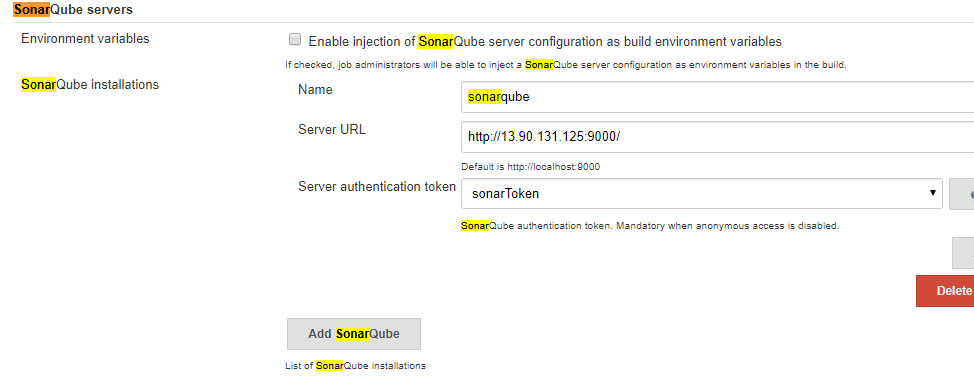


Figure 6 Shows the sonar config

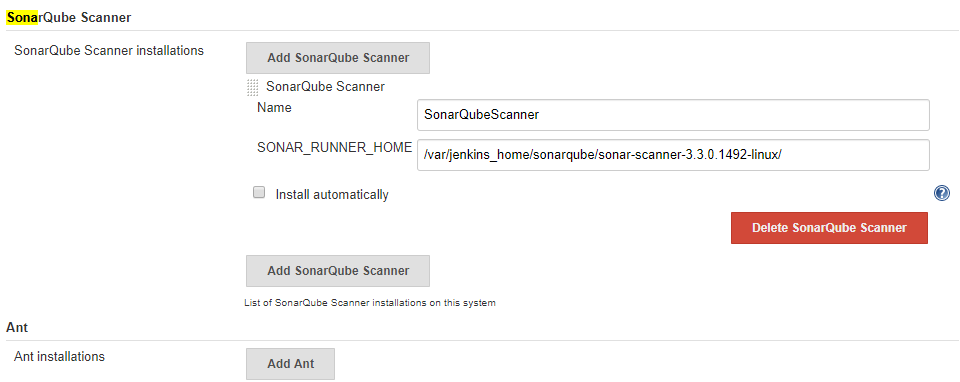


Figure 7 Shows the sonar plugin set up

# Docker & Docker hub

Docker is a tool which can be used to package and deploy application using “Containers”. A container is a fully self-contained runtime environment for an application which contains all the dependencies that an application needs to run. This means that applications can be ran on a virtual machine without having to set up the environment specifically to run this application.

Docker hub is an open source cloud based repository much like Github that allows users to store and copy docker images instead of code. This allows for the easy distribution of container images allowing for the easy setup of stable docker containers for a user who pulls them. Dockerhub integration can also be an important step in a continuous integration pipeline as It can be used as a place for a fully tested and built container to be stored so that it can be deployed into a run time environment by a tool such as Jenkins.

In order to implement a push to dockerhub as part of my pipeline I firstly had to add a step to my pipeline which builds and packages my image ready for a push. To accomplish this, I had to set two environment variables first of which simply being the registry address and the other being the credentials which will be declared in Jenkins. Secondly a file called the Dockerfile had to be created in the project, this file defines what the system should do in order to build a docker image. Thirdly I had to set up the docker hub credentials which were set up in the Jenkins UI and referred to in this Jenkins file. After these variables were defined the build stage of the pipeline uses a docker build command and appends the current build number onto the end of the image name so that the different versions of this image can be differentiated between. The “coursework 2” docker hub repository had to be created manually through the web UI for dockerhub so that this stage could be created.

For the push process of uploading this built image to dockerhub an additional pipeline step was added that uses the previously created environment variables to send the image to the repository with the supplied credentials.

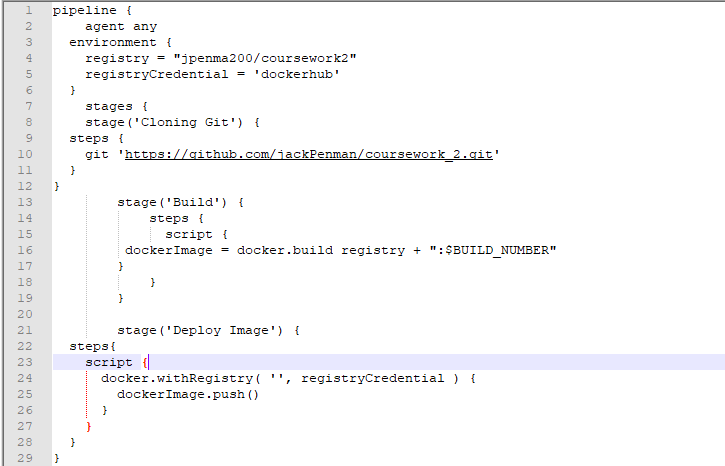


Figure 8 Jenkins file showing build and deploy stages needed for dockerhub integration

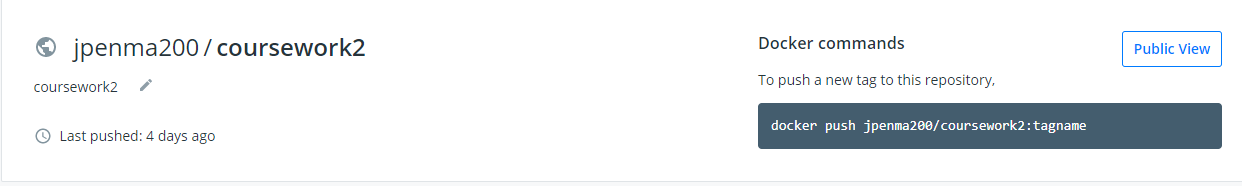


Figure 9 Shows dockerhub with a copy of the coursework 2 docker image

# 

# Production VM setup with Ansible

Ansible is an open source configuration management tool used for the automation of configuration of certain systems or used to automate the deployment of applications through scripts called “playbooks”.

In order to automate the deployment of my second VM I firstly had to install Ansible onto the first VM manually, this was done by using an apt get command and installing it using pip.

I then had to set up Ansible so that it had the credentials it needs to access azure so that it can create new VM’s, as part of this I also had to generate an ssh key which will be used to allow communication between the master VM and the VM generated by the playbook. This key would be used in the Ansible playbook to allow file transfer between the two VMs, important for transferring bash scripts which are used to set up some tools and attributes.

After these configuration steps were complete I created the playbook vm\_create.yml. This is the script which deploys the new VM. I made sure to use the vm size of Standard\_D2s\_v3 as this is the version which supports virtualizations which Kuberneties needs to run.

After this vm\_config.yml was made, which takes two bash scripts that I created that install node and k8s respectively and passes them into the new vm via ssh so that they can be ran to configure the vm.



Figure 10 This shows the Ansible playbook successfully deploying the VM automatically



Figure 11 Shows the successful automated set up of the VM

# Kuberneties

Kuberneties is a tool which allows an application to be deployed in a scalable and easy to manage way. It allows for the extremely useful feature of rolling out application updates while avoiding downtime for an end user. It can also be used to implement load balancing so that if an application is experiencing a particularly large amount of traffic, this traffic can distribute evenly across an application by Kuberneties ensuring build stability. It does these things by using “Clusters”. A is the highest level component of Kuberneties, it contains two major components, a node and a master. The master is what manages all the nodes which are present in a cluster and a node is where the container will actually be ran. When a deployment is running on Kuberneties, the process of taking the supplied repository and deploying it to each node is automatically managed by Kuberneties, because of this if a node were to fail during a deployment or during run time it can be automatically be brought back up. During a deployment of a container a “pod” is created within a node. A pod can contain one or more containers and is used to group these containers together if they share common attributes such as an ip address for example. If a node goes down with a pod running on it, this pod is then relaunched on different node.

One of the main features of Kuberneties is its ability to easily scale an application. This can be done by creating new pods on available nodes which all contain the running container. These pods are all made accessible through the same point of access by the use of a service, this changes it so that all pods will run on the same IP address. During the roll out of an update for an application Kuberneties will automatically bring nodes down so that they can be updated while balancing the traffic that is using the previous version of the application through the other nodes in the service. When the update on one node is finished it will re-launch it and move onto updating the next node and so on.

In order to implement Kuberneties as part of this continuous deployment pipeline, the tools Minikube and Kubectl were installed by the previous Ansible step.

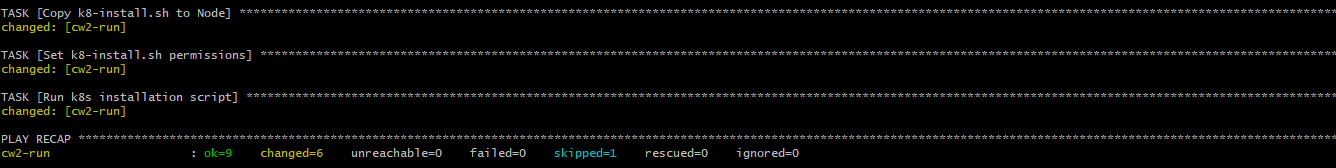


Figure 12Shows the install of these tools

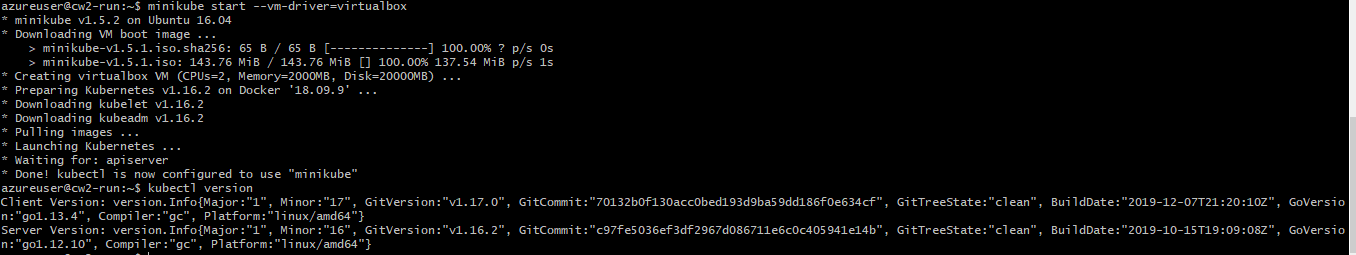


Figure 13 Shows Kuberneties started and running on this VM

After these tools were set up I manually deployed the container from dockerhub. I chose to do this part manually as it is a onetime set up step as the automated process will be updating this container as new code is added.



Figure 14 Shows the created deployment

I then created a service for the node that was running this container



Figure 15 Shows the running service

After this I manually scaled the application

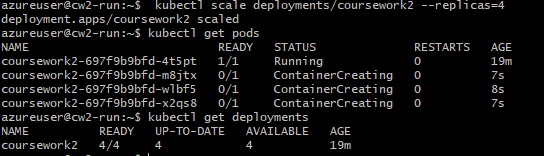


Figure 16 Shows the scaled application

In order to implement the rolling update for the pipeline I had to enter the bash console for the Jenkins container, from here I generated an ssh key ad added this to the authorized\_keys file in the Kuberneties vm, allowing me to ssh into the Kuberneties vm from the Jenkins container. Using this the final stage of the Jenkins file simply uses an sh command to input the update command.



Figure 17 Shows the application getting successfully updated