RAMAIAH
Institute of Technology

# Containerized Applications with Docker and Kubernetes

RAMAIAH
Institute of Technology

# Containerizing Your Application with Docker

# Introduction to Dockers

- **Docker** is a containerization tool, which became open source in 2013.
- It allows you to isolate an application from its host system so that the application becomes portable.
- And the code tested on a developer's workstation can be deployed to production with fewer concerns about execution runtime dependencies.
- A **container** is a system that embeds an application and its dependencies.
- Unlike a VM, a container contains only a light operating system with only the elements required for the OS, such as system libraries, binaries, and code dependencies.

# Introduction to Dockers

- The principal difference between VMs and containers is that each VM that is hosted on a hypervisor contains a complete OS.
- It is therefore completely independent of the guest OS that is on the hypervisor.
- Containers don't contain a complete OS – only a few binaries—but they are dependent on the guest OS, using its resources (CPU, RAM, and network).

# Introduction to Dockers

- Containers Vs. Virtual Machine

| Containers | Virtual Machine |
| --- | --- |
| Integration in a container is faster and cheap. | Integration in virtual is slow and costly. |
| No wastage of memory. | Wastage of memory. |
| It uses the same kernel, but different distribution. | It uses multiple independent operating systems. |
| | |

# Introduction to Dockers

- Why use Dockers:
  - Easy to install and run software without worrying about setup or dependencies.
  - Developers use Docker to eliminate machine problems, i.e. "but code is worked on my laptop." when working on code together with co-workers.
  - Operators use Docker to run and manage apps in isolated containers for better compute density.
  - Enterprises use Docker to securely built agile software delivery pipelines to ship new application features faster and more securely.
  - Since docker is not only used for the deployment, but it is also a great platform for development, and helps in increasing customer's satisfaction.

# Introduction to Dockers

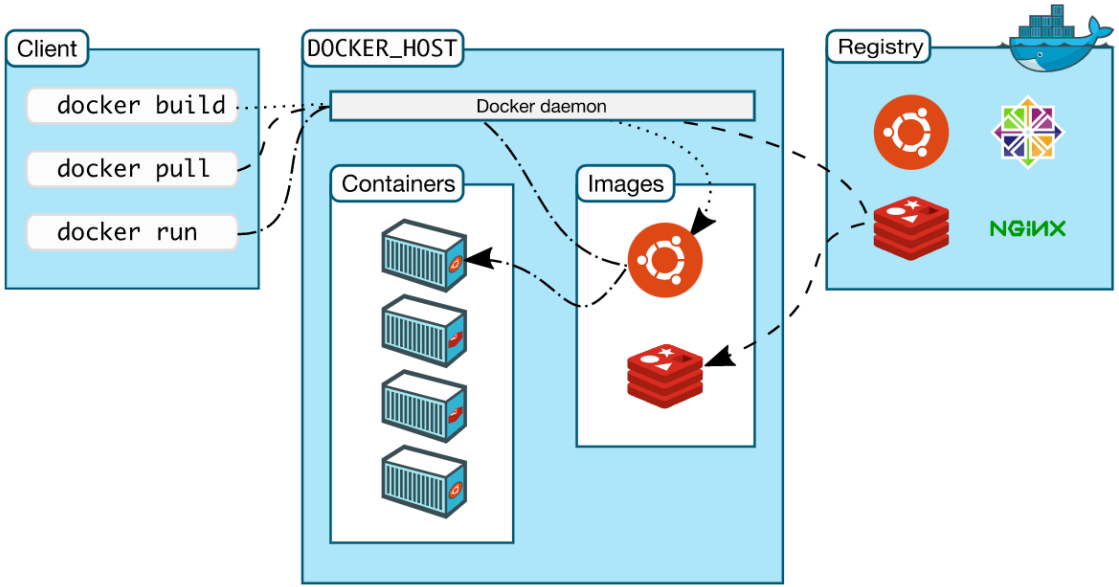
- Advantages of Dockers:
  - It runs the container in seconds instead of minutes.
  - It uses less memory.
  - It provides lightweight virtualization.
  - It does not a require full operating system to run applications.
  - It uses application dependencies to reduce the risk.
  - Docker allows you to use a remote repository to share your container with others.
  - It provides continuous deployment and testing environment.

# Introduction to Dockers

- Disadvantages of Dockers:
  - It increases complexity due to an additional layer.
  - In Docker, it is difficult to manage large amount of containers.
  - Some features such as container self -registration, containers self-inspects, copying files form host to the container, and more are missing in the Docker.
  - Docker is not a good solution for applications that require rich graphical interface.
  - Docker provides cross-platform compatibility means if an application is designed to run in a Docker container on Windows, then it can't run on Linux or vice versa.

# Components of Docker

- There are four components of docker:
- Docker client and server
- Docker image
- Docker registry
- Docker container



# Components of Docker

- **Docker Client and Server:**
- This is a command-line-instructed solution by using the terminal to issue commands from the Docker client to the Docker daemon.
- The communication between the Docker client and the Docker host is via a REST API.
- Ex: A Docker Pull command would send an instruction to the daemon and perform the operation by interacting with other components (image, container, registry).
- The Docker daemon itself is actually a server that interacts with the operating system and performs services.

# Components of Docker

- **Docker Client and Server:**
- Docker daemon constantly listens across the REST API to see if it needs to perform any specific requests.
- To trigger and start the whole process, use the Dockered command within the Docker daemon. And it will start all of the performances.
- Then you have a Docker host, which lets you run the Docker daemon and registry.

# Components of Docker

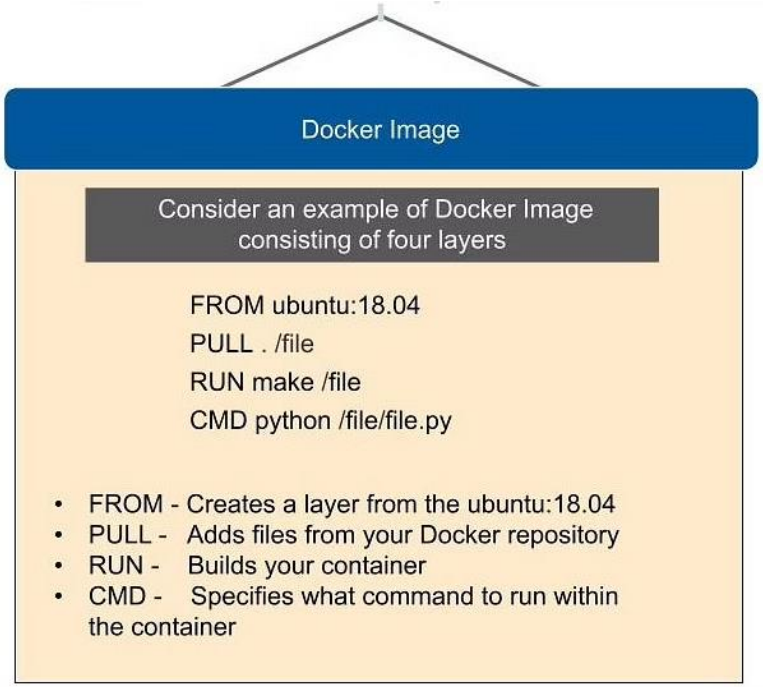
- **Docker Image:**
- A Docker image is a template that contains instructions for the Docker container.
- That template is written in a YAML, which stands for Yet Another Markup Language.
- The Docker image is hosted as a file in the Docker registry.
- The image has several key layers, and each layer depends on the layer below it.

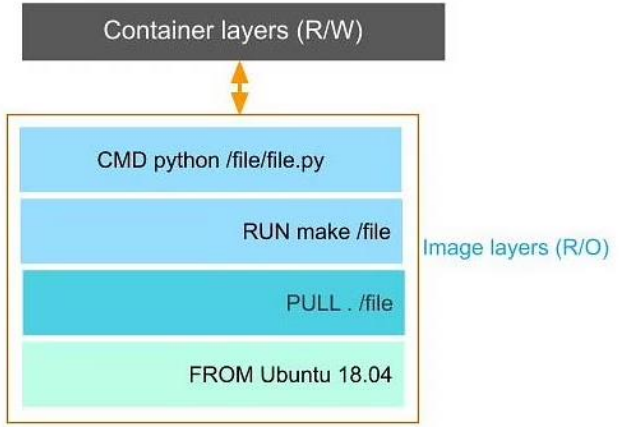
# Components of Docker

- **Docker Image:**
- Image layers are created by executing each command in the Dockerfile and are in the read-only format.
- Start with base layer, which will typically have base image and base operating system.
- And then have a layer of dependencies above that.
- These then comprise the instructions in a read-only file that would become your Dockerfile.

# Components of Docker

- Docker Image:



# Components of Docker

- **Docker Image:**
- In the previous image, there are four layers of instructions: From, Pull, Run and CMD.
- The From command creates a layer based on Ubuntu, and then add files from the Docker repository to the base command of that base layer.
- **Pull**: Adds files from your Docker repository.
- **Run**: Builds your container.
- **CMD**: Specifies which command to run within the container.

# Components of Docker

- **Docker Registry:**
- The Docker registry is used to host various types of images and distribute the images from.
- The repository itself is just a collection of Docker images, which are built on instructions written in YAML and are very easily stored and shared.
- Give name tags to the Docker images so that it’s easy to find and share them within the Docker registry.
- One way to start managing a registry is to use the publicly accessible Docker hub registry, which is available to anybody.

# Components of Docker

- **Docker Registry:**
- You can also create your own registry for your own use internally.
- The registry that you create internally can have both public and private images that you create.
- The commands you would use to connect the registry are Push and Pull.
- Use the Push command to push a new container environment you've created from your local manager node to the Docker registry.
- Use a Pull command to retrieve new clients (Docker image) created from the Docker registry.

# Components of Docker

- **Docker Registry:**
- A Pull command pulls and retrieves a Docker image from the Docker registry.
- A Push command allows you to take a new command that you've created and push it to the registry, whether it's Docker hub or your own private registry.

# Components of Docker

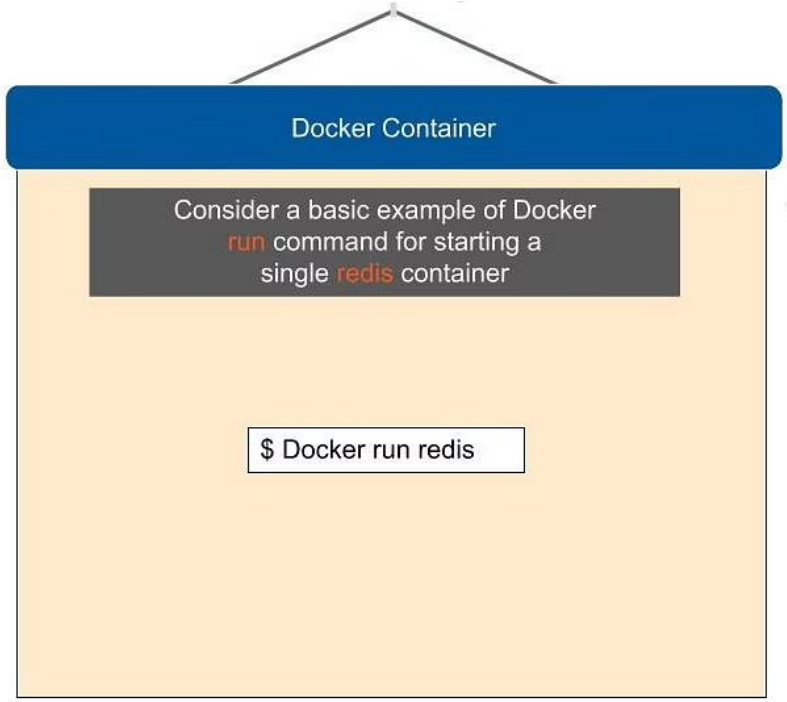
- **Docker Container:**
- The Docker container is an executable package of applications and its dependencies bundled together.
- It gives all the instructions for the solution you're looking to run.
- It is lightweight due to the built-in structural redundancy.
- The container is also portable.
- Another benefit is that it runs completely in isolation.

# Components of Docker

- **Docker Container:**
- Even if you are running a container, it's guaranteed not to be impacted by any host OS securities or unique setups, unlike with a virtual machine or a non containerized environment.
- The memory for a Docker environment can be shared across multiple containers.
- This is useful, especially when you have a virtual machine that has a defined amount of memory for each environment.

# Components of Docker

- Docker Container:




Suppose a user runs *$ Docker run redis* command, the following happens:

- In case you don't have a Docker Image locally, the Docker pulls the image from your Registry
- Now, Docker creates a new container redis from the existing Docker Image
- Docker creates a container layer of read-write filesystem

Read-only

# Components of Docker

- **Docker Container:**
- **$ Docker run redis**
- If Redis image is not locally installed, it will be pulled from the registry.
- After this, the new Docker container Redis will be available within your environment so you can start using it.
- Containers are lightweight because they do not have some of the additional layers that virtual machines do.
- The biggest layer Docker doesn't have is the hypervisor, and it doesn't need to run on a host operating system.

# Advanced Components of Docker

1 Docker Compose:

- It is designed for running multiple containers as a single service.
- It does so by running each container in isolation but allowing the containers to interact with one another.
- Write the compose environments using YAML.
- Use Docker Compose if you are running an Apache server with a single database and you need to create additional containers to run additional services without having to start each one separately.
- You would write a set of files using Docker compose to do that.

# Advanced Components of Docker

2 Docker Swamp:

- It is a service for containers that allows IT administrators and developers to create and manage a cluster of swarm nodes within the Docker platform.
- Each node of Docker swarm is a Docker daemon, and all Docker daemons interact using the Docker API.
- A swarm consists of two types of nodes: a manager node and a worker node.
- A manager node maintains cluster management tasks.
- Worker nodes receive and execute tasks from the manager node.

# Installing Docker

- Docker's community edition (CE) is free and is very well suited to developers and small teams.
- If Docker is to be used throughout a company, it is better to use Docker Enterprise, which is not free.
- Docker is a cross-platform tool that can be installed on Windows, Linux, or macOS.
- Also it is natively present on some cloud providers, such as AWS and Azure.

# Installing Docker

- To operate, Docker needs the following elements:

1. **The Docker client**: This allows you to perform various operations on the command line.
2. **The Docker daemon**: This is Docker's engine.
3. **Docker Hub**: This is a public (with a free option available) registry of Docker images.

- Before installing Docker, we will first create an account on Docker Hub.

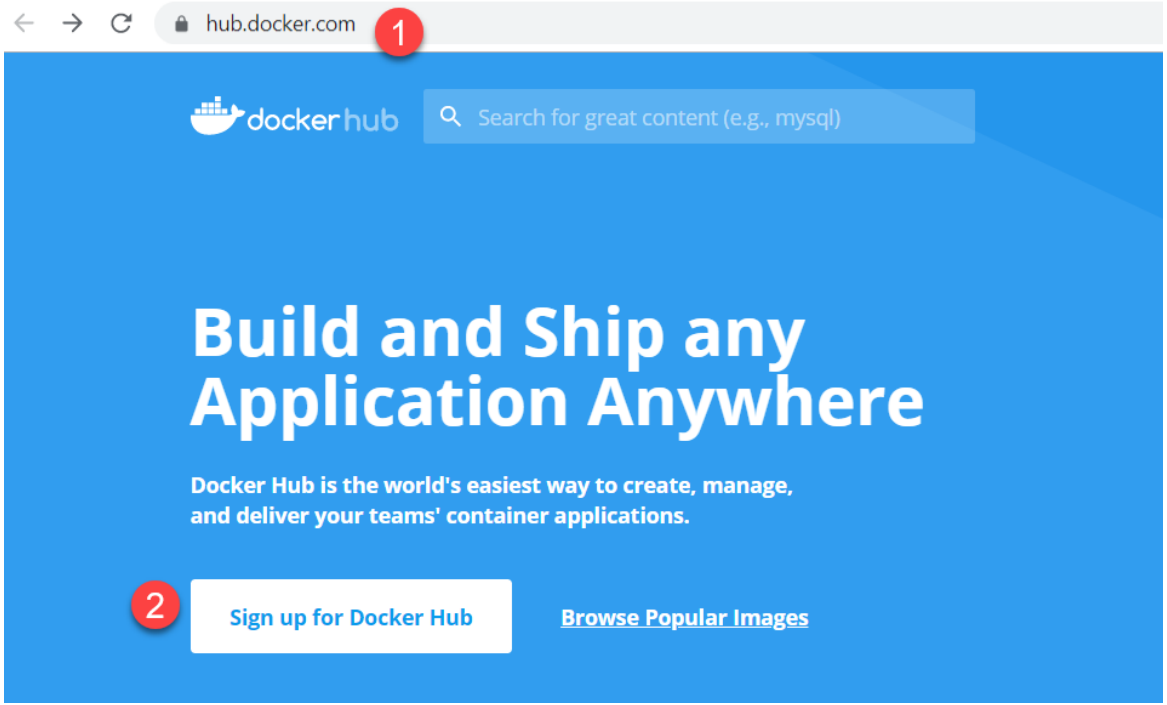
# Installing Docker

- **Registering on Docker Hub**
- Docker Hub is a public space called a registry, containing more than 2 million public Docker images that have been deposited by companies, communities, and even individual users.
- To register on Docker Hub and list Docker images, perform the following steps:

1. Go to https:/ / hub. docker. com/ and click on the Sign up for Docker Hub button:

# Installing Docker

- Registering on Docker Hub



# Installing Docker

- **Registering on Docker Hub**

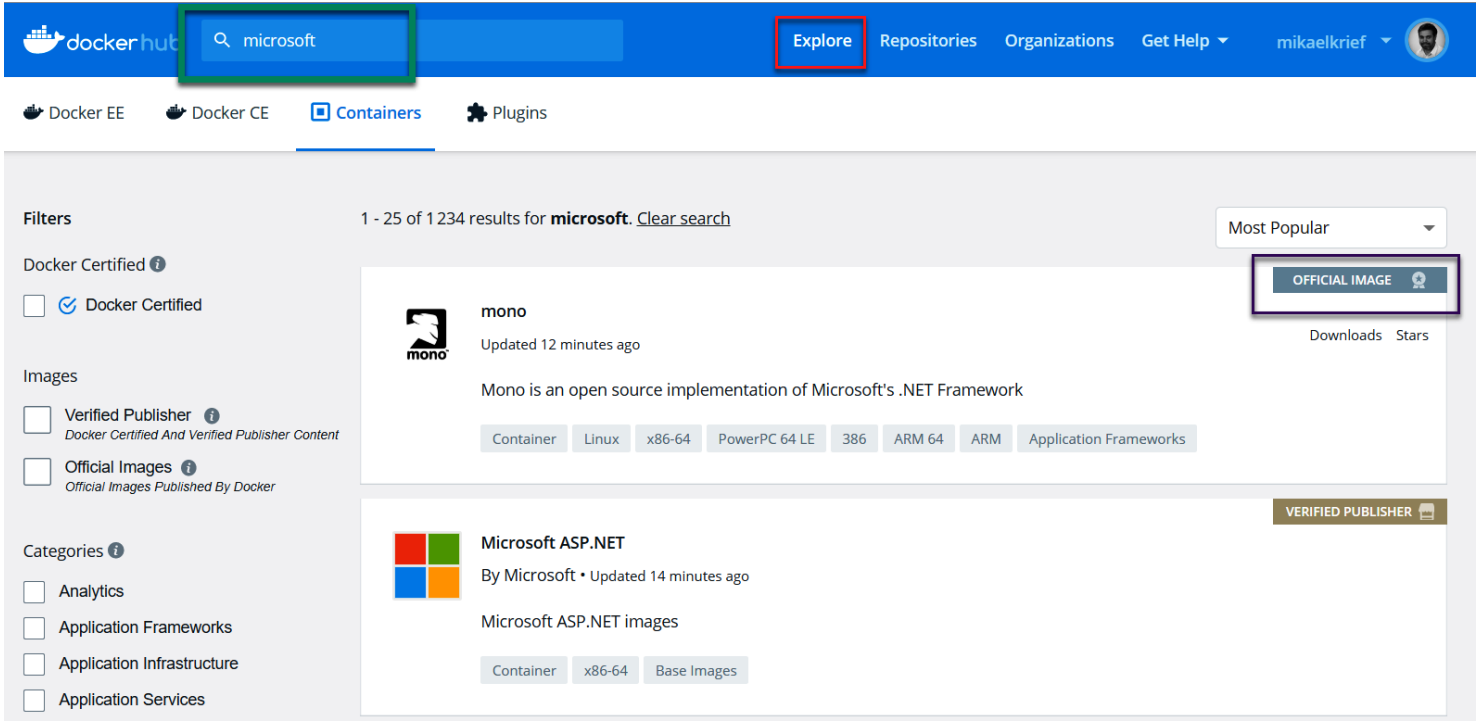
2. Fill in the form with a unique ID, an email, and a password.

3. Once your account is created, you can then log in to the site, and this account will allow you to upload custom images and download Docker Desktop.

4. To view and explore the images available from Docker Hub, go to the Explore section.

# Installing Docker

- Registering on Docker Hub

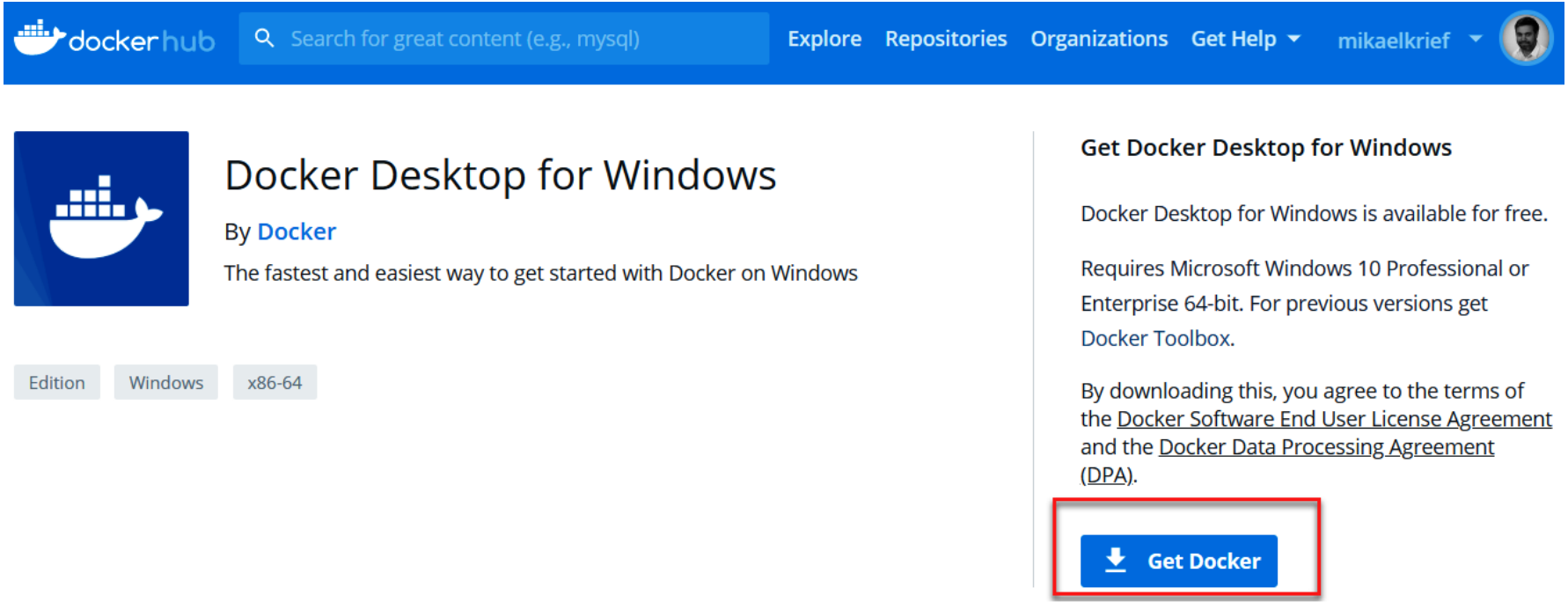


# Installing Docker

- **Docker Installation:**
- To install Docker on a Windows machine, it is necessary to first check the hardware requirements, which are as follows:
  - Windows 10 64 bit with at least 4 GB of RAM
  - A virtualization system (such as Hyper-V) enabled.
- To install Docker Desktop follow these steps:
- 1. First, download Docker Desktop by clicking on the Get Docker button from Docker Hub at **https://hub.docker.com/editions/community/docker-ce-desktop-windows** and log in if you are not already connected to Docker Hub.

# Installing Docker

- Docker Installation:

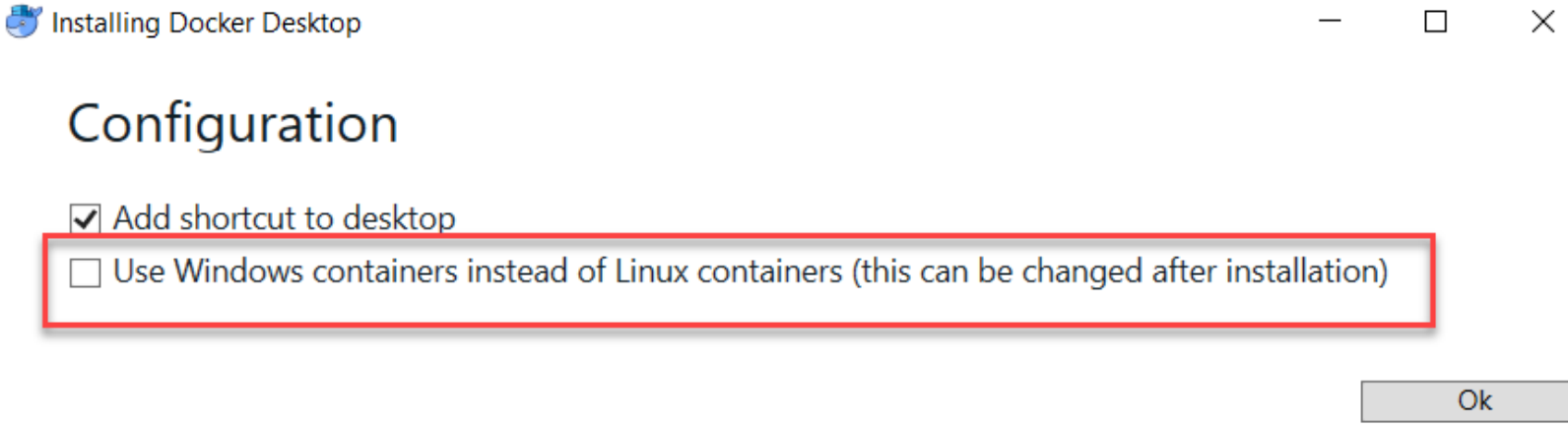


# Installing Docker

- **Docker Installation:**

2. Once that's downloaded, click on the downloaded EXE file.

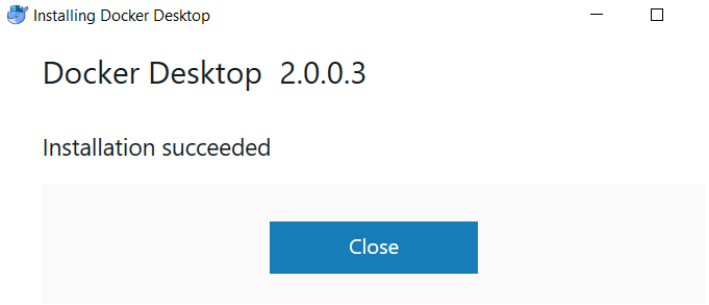
3. Then, take the single configuration step, which is a choice between using Windows or Linux containers:



# Installing Docker

- **Docker Installation:**

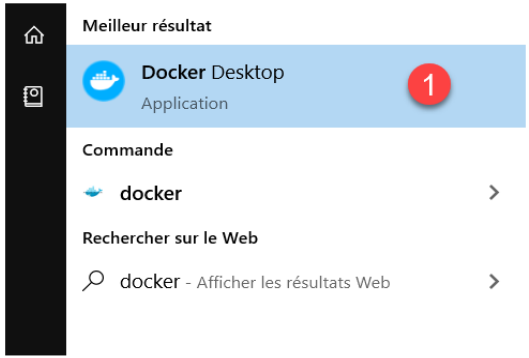
4. Once the installation is complete, we'll get a confirmation message and a button to close the installation:




5. Finally, to start Docker, launch the Docker Desktop program. An icon will appear in the notification bar indicating that Docker is starting. It will then ask you to log in to Docker Hub via a small window. The startup steps of Docker Desktop are shown in the following screenshot:

# Installing Docker

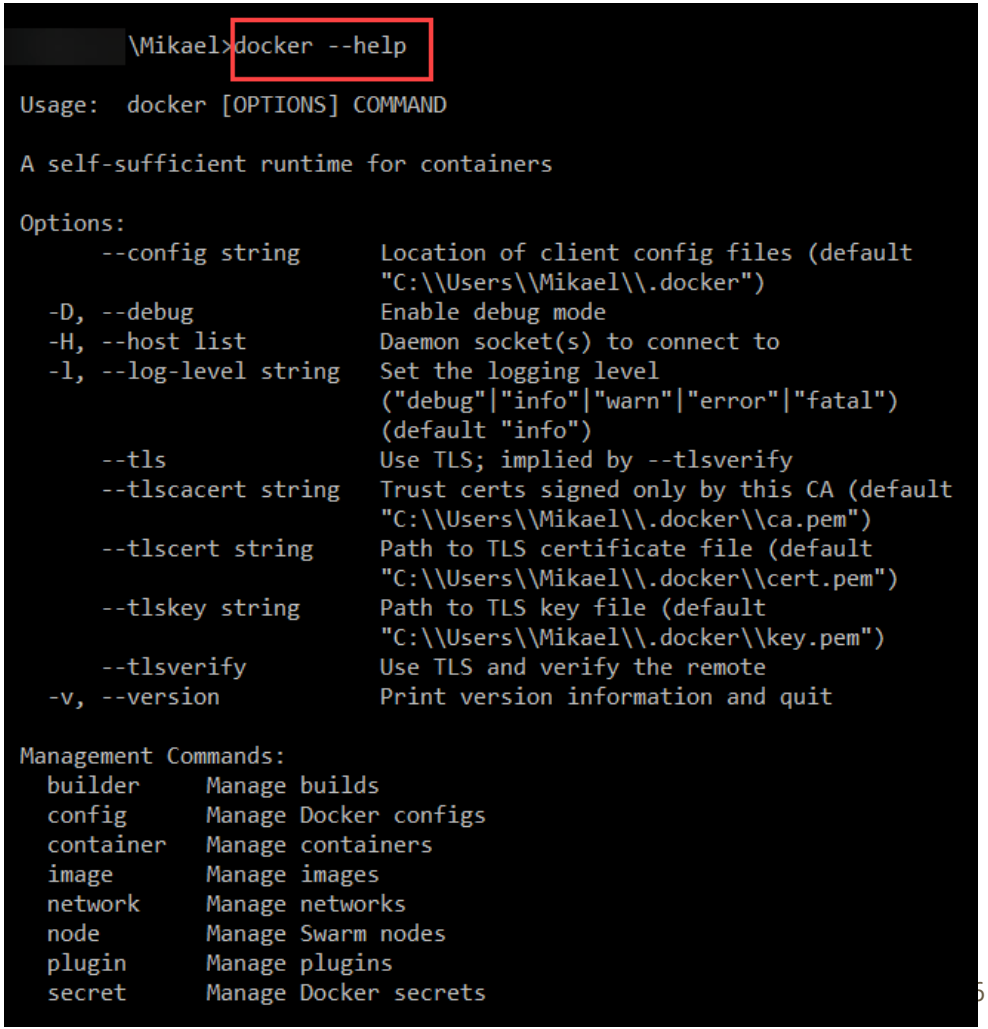
- Docker Installation:





# Installing Docker

- Docker Installation:
- To check your Docker installation, open the Terminal window (it will also work on a Windows PowerShell Terminal), then execute the following command:
- docker --help



# Installing Docker

- **An overview of Docker's elements:**
- Docker's fundamental elements are **Dockerfiles, containers, and volumes**.
- A Docker image is a basic element of Docker and consists of a text document called a Dockerfile.
- Dockerfile contains the binaries and application files to containerize.
- A container is an instance that is executed from a Docker image.
- It is possible to have several instances of the same image within a container that the application will run.

# Installing Docker

- **An overview of Docker's elements:**
- Finally, a volume is storage space that is physically located on the host OS (that is, outside the container).
- It can be shared across multiple containers if required.
- This space will allow the storage of persistent elements (files or databases).
- To manipulate these elements, use command lines.

# Creating a Dockerfile

- A basic Docker element is a file called a Dockerfile, which contains step-by-step instructions for building a Docker image.
- To understand how to create a Dockerfile, look at an example that build a Docker image that contains an Apache web server and a web application.
- **Writing a Dockerfile:**
- First create an HTML page that will be the web application.
- Create a new appdocker directory and an index.html page in it, which includes the example code that displays welcome text on a web page:

# Creating a Dockerfile

- Writing a Dockerfile:

```
<html>
  <body>
    <h1>Welcome to my new app</h1>
    This page is test for my demo Dockerfile.<br />
    Enjoy ...
  </body>
</html>
```

- Then, in the same directory, create a Dockerfile (without an extension) with the following content:.

```
FROM httpd:latest
COPY index.html /usr/local/apache2/htdocs/
```

# Creating a Dockerfile

- **Writing a Dockerfile:**
- To create a Dockerfile, start with the FROM statement.
- The required FROM statement defines the base image, which will be used for Docker image.
- Any Docker image is built from another Docker image.
- This base image can be saved either in Docker Hub or in another registry (Ex: Artifactory, Nexus Repository, or Azure Container Registry).
- In this code example, the Apache httpd image is used and tagged the latest version, https://hub.docker.com/_/httpd/.

# Creating a Dockerfile

- **Writing a Dockerfile:**
- And use the FROM httpd:latest Dockerfile instruction.
- Then, use the COPY instruction to execute the image construction process.
- Docker copies the local index.html file into the /usr/local/apache2/htdocs/ directory of the image.

# Creating a Dockerfile

- **Dockerfile Instructions Overview:**
- A Dockerfile file is comprised of many instructions.
- There are other instructions that will allow to build a Docker image.
- Here is an overview of the principal instructions that can be used:
- **FROM**: This instruction is used to define the base image for our image, as shown in the example detailed in the Writing a Dockerfile section.
- **COPY and ADD**: These are used to copy one or more local files into an image. The Add instruction supports an extra two functionalities, to refer to a URL and to extract compressed files.

# Creating a Dockerfile

- Dockerfile Instructions Overview:
- RUN and CMD: This instruction takes a command as a parameter that will be executed during the construction of the image.
- The Run instruction creates a layer so that it can be cached and versioned.
- The CMD instruction defines a default command to be executed during the call to run the image.
- The CMD instruction can be overwritten at runtime with an extra parameter provided.

# Creating a Dockerfile

- **Dockerfile Instructions Overview:**
- **RUN and CMD**: Write the following example of the RUN instruction in a Dockerfile to execute the apt-get command:
- **RUN apt-get update**
- This instruction updates the apt packages that are already present in the image and create a layer.
- Use the CMD instruction in the following example will display a docker message:
- **CMD "echo docker"**

# Creating a Dockerfile

- Dockerfile Instructions Overview:
- ENV: Allows to instantiate environment variables that can be used to build an image.
- These environment variables will persist throughout the life of the container, as follows:
- ENV myvar=mykey
- WORKDIR: This instruction gives the execution directory of the container, as follows:
- WORKDIR usr/local/apache2

# Creating a Dockerfile

- **Dockerfile Instructions Overview:**
- **ENTRYPOINT**: If container needs something more complex, then use the ENTRYPOINT command. Used in conjunction with CMD for parameters, ENTRYPOINT sets the main command for the image, to run an image as if it were that command.
- **EXPOSE**: This command exposes the ports that the software uses, ready for you to map to the host when running a container with the -p argument.
- **VOLUME**: The docker VOLUMEN command is used to create mount point in the image. This mount point can be used to mount volumes from the Docker host or form the other containers.

# Building and Running a Container on a Local Machine

- The execution of Docker is performed by these different operations:

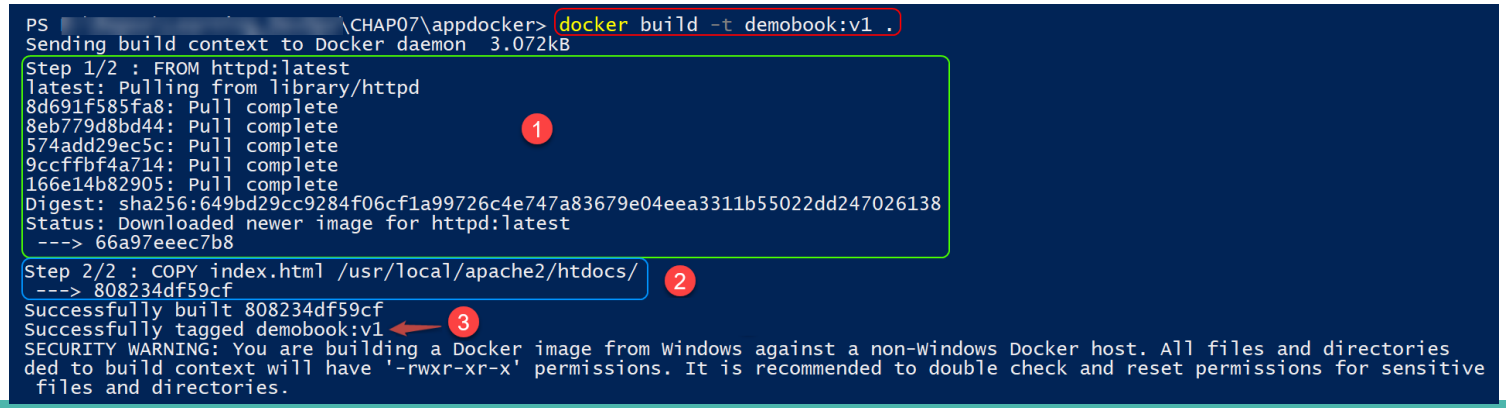
1. Building a Docker image from a Dockerfile
2. Instantiating a new container locally from this image
3. Testing our locally containerized application

# Building and Running a Container on a Local Machine

- **Building a Docker image:**
- To build a Docker image from our previously created Dockerfile that contains the following instructions:
  - *FROM httpd:latest*
  - *COPY index.html /usr/local/apache2/htdocs/*
- Go to a Terminal to head into the directory that contains the Dockerfile, and then execute the docker build command with the following syntax:
  - *docker build -t demobook:v1*

# Building and Running a Container on a Local Machine

- **Building a Docker image:**
- The -t argument indicates the name of the image and its tag. In this example, demobook is image and v1 is the tag.
- The . (dot) at the end of the command specifies to use the files in the current directory.



# Building and Running a Container on a Local Machine

- **Building a Docker image:**
- Executing the docker build command downloads the base image indicated in the Dockerfile from Docker Hub, and then Docker executes the various instructions that are mentioned in the Dockerfile.
- At the end of the execution, obtain a locally stored Docker demobook image.
- Check if the image is successfully created by executing the following Docker command:
- **docker images**

```
PS [redacted]\Learning_DevOps\CHAP07\appdocker> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
demobook            v1                  a121d88f6e18        23 minutes ago      140MB
httpd               latest              e77c77f17b46        6 days ago          140MB
```

# Building and Running a Container on a Local Machine

- **Instantiating a new Container of an Image:**
- To instantiate a container of Docker image created, execute the docker run command in the Terminal with the following syntax:
- **docker run -d --name demoapp -p 8080:80 demobook:v1**
- The -d parameter indicates that the container will run in the background.
- In the -name parameter, we indicate the name of the container we want.
- In the -p parameter, we indicate the desired port translation; that is, in our example, port 80 of the container will be translated to port 8080 on our local machine.

# Building and Running a Container on a Local Machine

- **Instantiating a new Container of an Image:**
- And finally, the last parameter of the command is the name of the image and its tag.
- The execution of this command is shown in the following screenshot:

```
PS [redacted]\Learning_DevOps\CHAP0/\appdocker> docker run -d --name demoapp -p 8080:80 demobook:v1
381b476d62e568f382f251e0834fd8c69f713eb14ea41c95e5cd7004afdbb879
```

- This command displays the ID of the container, and the container runs in the background.
- It is also possible to display the list of containers running on the local machine, by executing the following command:

# Building and Running a Container on a Local Machine

- Instantiating a new Container of an Image:
- docker ps
- The following screenshot shows the execution with our container:

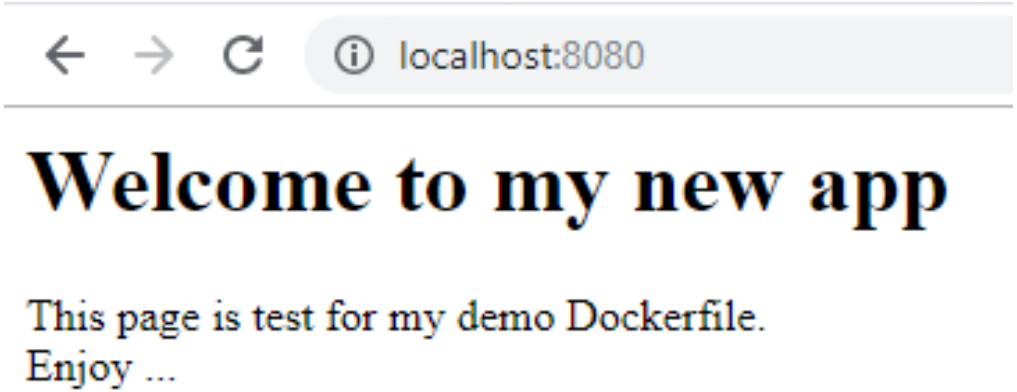
```
PS [redacted]\Learning_DevOps\CHAP07\appdocker> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
381b476d62e5        demobook:v1         "httpd-foreground"       3 minutes ago       Up 3 minutes        0.0.0.0:8080->80/tcp     demoapp
```

# Building and Running a Container on a Local Machine

- **Testing a Container locally :**
- Everything that runs in a container remains inside it.
- This is the principle of container isolation.
- However, with the port translation and with the run command, you can test your container on your local machine.
- To do this, open a web browser and enter http://localhost:8080 with 8080, which represents the translation port indicated in the command, and here is the result:

# Building and Running a Container on a Local Machine

- Testing a Container locally :



# Pushing an image to Docker Hub

- The goal of creating a Docker image that contains an application is to be able to use it on servers that contain Docker and host the company's applications.
- In order for an image to be downloaded to another computer, it must be saved in a Docker image registry.
- There are several Docker registries that can be installed on-premise.
- If you want to create a public image, you can push it (or upload it) to Docker Hub, which is Docker's public (and free) registry.

# Pushing an image to Docker Hub

- To push a Docker image to Docker Hub, perform the following steps:

1. Sign in to Docker Hub: Log in to Docker Hub using the following command:
   *docker login -u <your dockerhub login>*

```
PS [redacted]\Learning_DevOps\CHAP07\appdocker> docker login -u mikaelkr[redacted]
Password:
Login Succeeded
```

2. Retrieving the image ID: The next step consists of retrieving the ID of the image that has been created. Execute the docker images command to display the list of images with their ID.

```
PS [redacted]\Learning_DevOps\CHAP07\appdocker> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
demobook            v1                  a121d88f6e18        6 hours ago         140MB
httpd               latest              e77c77f17b46        6 days ago          140MB
```

# Pushing an image to Docker Hub

- To push a Docker image to Docker Hub, perform the following steps:

3. Tag the image for Docker Hub: With the ID of the image we retrieved, we will now tag the image for Docker Hub. To do so, the following command is executed: *docker tag <image ID> <dockerhub login>/demobook:v1*

```
PS [redacted]\Learning_DevOps\CHAP07\appdocker> docker tag a121d88f6e18 mikaelkrief/demobook:v1
```

# Pushing an image to Docker Hub

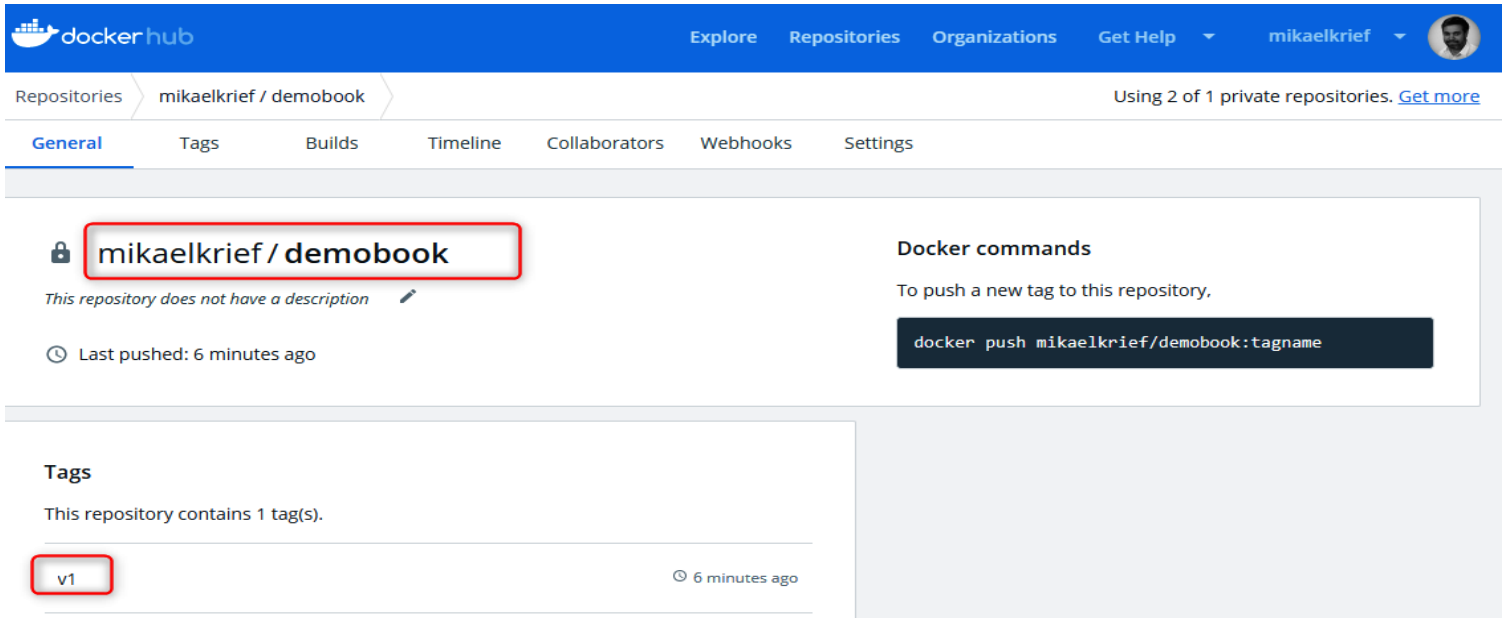
4. Push the image Docker in the Docker Hub: After tagging the image, the last step is to push the tagged image to Docker Hub.

- Execute the following command:
- *docker push docker.io/<dockerhub login>/demobook:v1*

```
PS [redacted]\Learning_DevOps\CHAP07\appdocker> docker push docker.io/mikaelkrief/demobook:v1
The push refers to repository [docker.io/mikaelkrief/demobook]
e5df7a05d9b7: Pushed
6c4a74a82dc9: Mounted from library/httpd
9cba8b480e83: Mounted from library/httpd
25797e1a8e3f: Mounted from library/httpd
d2583584487e: Mounted from library/httpd
cf5b3c6798f7: Mounted from library/httpd
v1: digest: sha256:ffe4e6e67b8bf200a1c86d42a00730491ded2e63279ddbaeb7e7ffdf3b56cd89 size: 1574
```

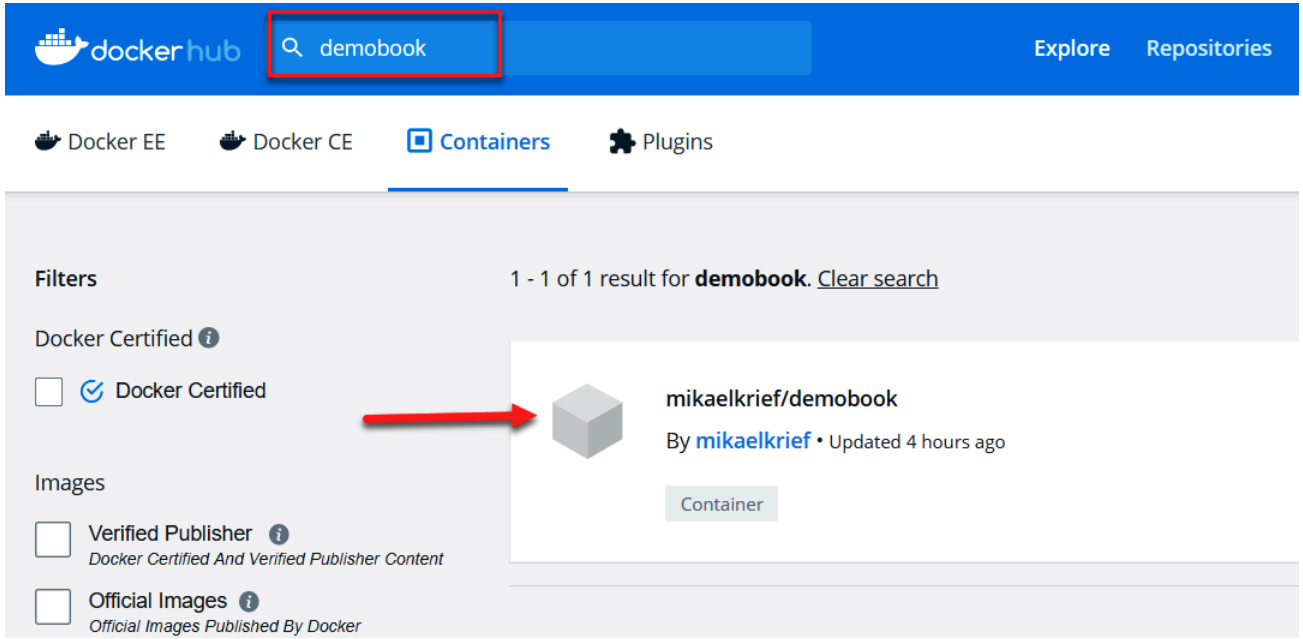
# Pushing an image to Docker Hub

- To view the pushed image in Docker Hub, connect to the Docker Hub web portal at https://hub.docker.com/ and see that the image is present.



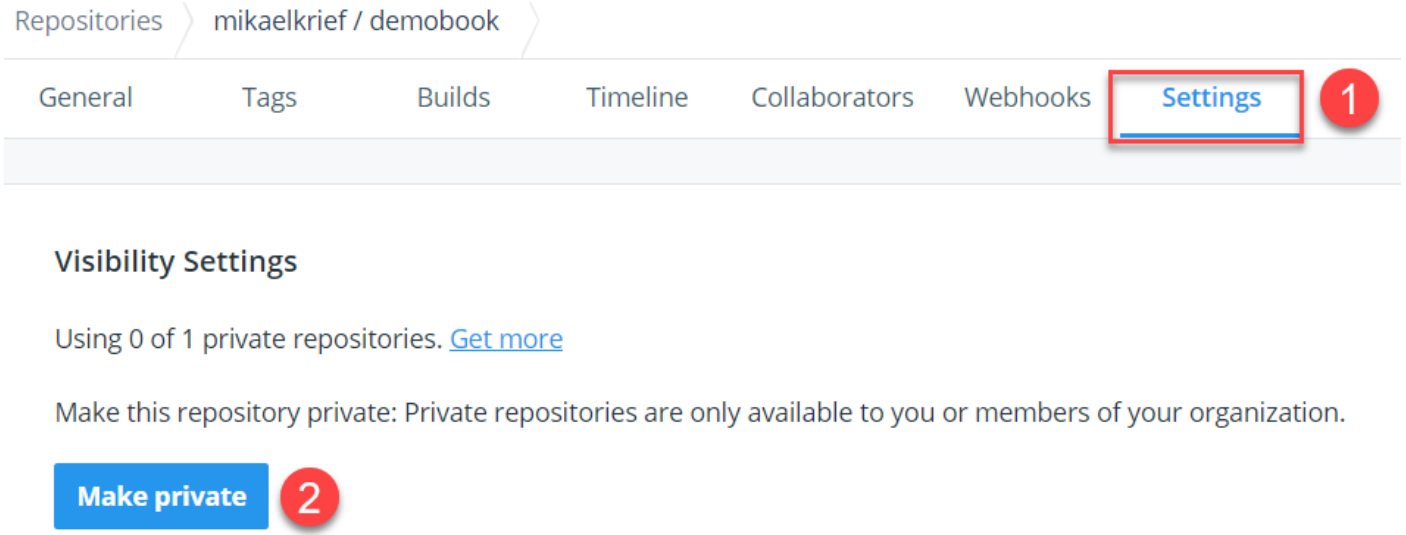
# Pushing an image to Docker Hub

- By default, the image pushed to Docker Hub is in public mode – everybody can view it in the explorer and use it.



# Pushing an image to Docker Hub

- To make this image private – that is, you must be authenticated to be able to use it – you must go to the Settings of the image and click on the Make private button:



# Deploying a container to ACI with a CI/CD pipeline

- One of the reasons Docker has quickly become attractive to developers and operations teams is that the deployment of Docker images and containers has made CI and CD pipelines for enterprise applications easier.
- To automate the deployment of our application, we will create a CI/CD pipeline that deploys the Docker image that contains our application in ACI.
- ACI is a managed service from Azure that allows you to deploy containers very easily, without having to worry about the hardware architecture.

# Deploying a container to ACI with a CI/CD pipeline

- In this section:
  - The Terraform code of the Azure ACI and its integration with our Docker image.
  - An example of a CI/CD pipeline in Azure Pipelines, which allows you to execute the Terraform code.

# Deploying a container to ACI with a CI/CD pipeline

- **The Terraform code for ACI:**
- To provision an ACI resource with Terraform, navigate to a new terraform-aci directory and create a Terraform file, main.tf.
- In this code, provide Terraform code for a resource group and ACI resource using the azurerm-container-group Terraform object.
- This main.tf file contains the following Terraform code:

```
resource "azurerm_resource_group" "acidemobook" {
 name = "demoBook"
 location = "westus2"
}
```

# Deploying a container to ACI with a CI/CD pipeline

- The Terraform code for ACI:
- Add the Terraform code for the variable declarations:

```
variable "imageversion" {
 description ="Tag of the image to deploy"
}
variable "dockerhub-username" {
 description ="Tag of the image to deploy"
}
```

# Deploying a container to ACI with a CI/CD pipeline

- The Terraform code for ACI:
- Add the Terraform code for the ACI with the azurerm-container-group resource block:

```
resource "azurerm_container_group" "aci-myapp" {
    name = "aci-agent"
    location = "West Europe"
    resource_group_name = azurerm_resource_group.acidemobook.name
    os_type = "linux"
    container {
        name = "myappdemo"
        image = "docker.io/mikaelkrief/${var.dockerhub-
username}:${var.imageversion}"
        cpu = "0.5" memory = "1.5"
        ports {
         port = 80
         protocol = "TCP"
          }
      }
}
```

# Deploying a container to ACI with a CI/CD pipeline

- **The Terraform code for ACI:** In this code, we do the following:
- Declare imageversion and dockerhub-username variables, which will be instantiated during the CI/CD pipeline and include the username and the tag of the image to be deployed.
- Use the azurerm-container-group resource from Terraform to manage the ACI. In its image property, we indicate the information of the image to be deployed; that is, its full name in Docker Hub as well as its tag, which in our example is deported in the imageversion variable.
- Finally, in order to protect the tfstate file, use the Terraform remote backend by using an Azure blob storage.

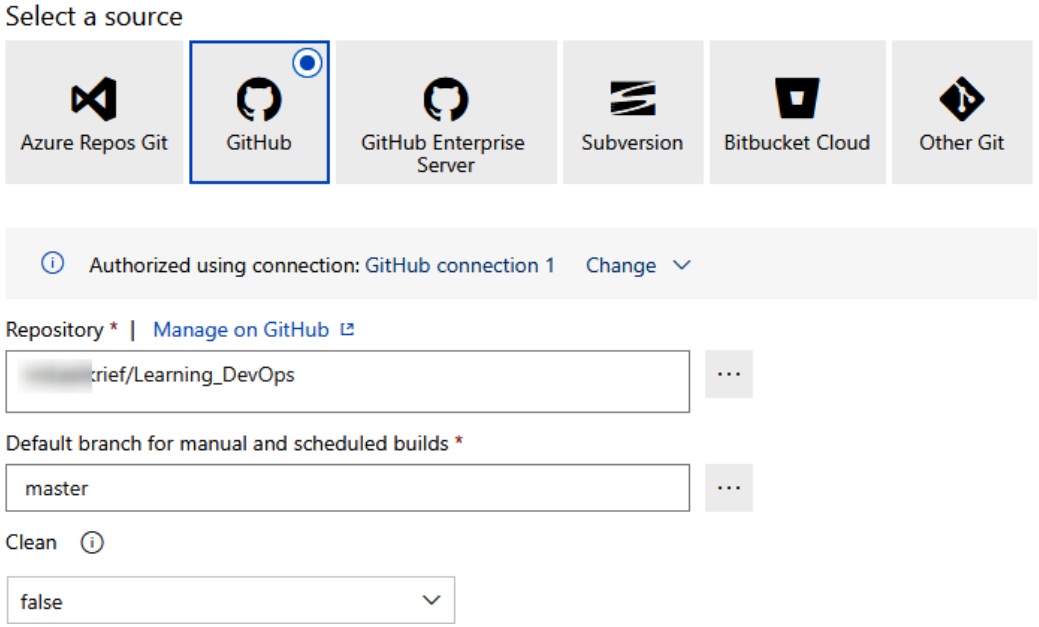
# Deploying a container to ACI with a CI/CD pipeline

- **Creating a CI/CD pipeline for the container:**
- To create a CI/CD pipeline that will build image and execute the Terraform code, use all the tools in Continuous Integration and Continuous Delivery stage.
- To visualize the pipeline, use Azure Pipelines, which is one of the detailed tools.
- To implement the CI/CD pipeline in Azure Pipelines, we will proceed with these steps:

# Deploying a container to ACI with a CI/CD pipeline

- **Creating a CI/CD pipeline for the container:**

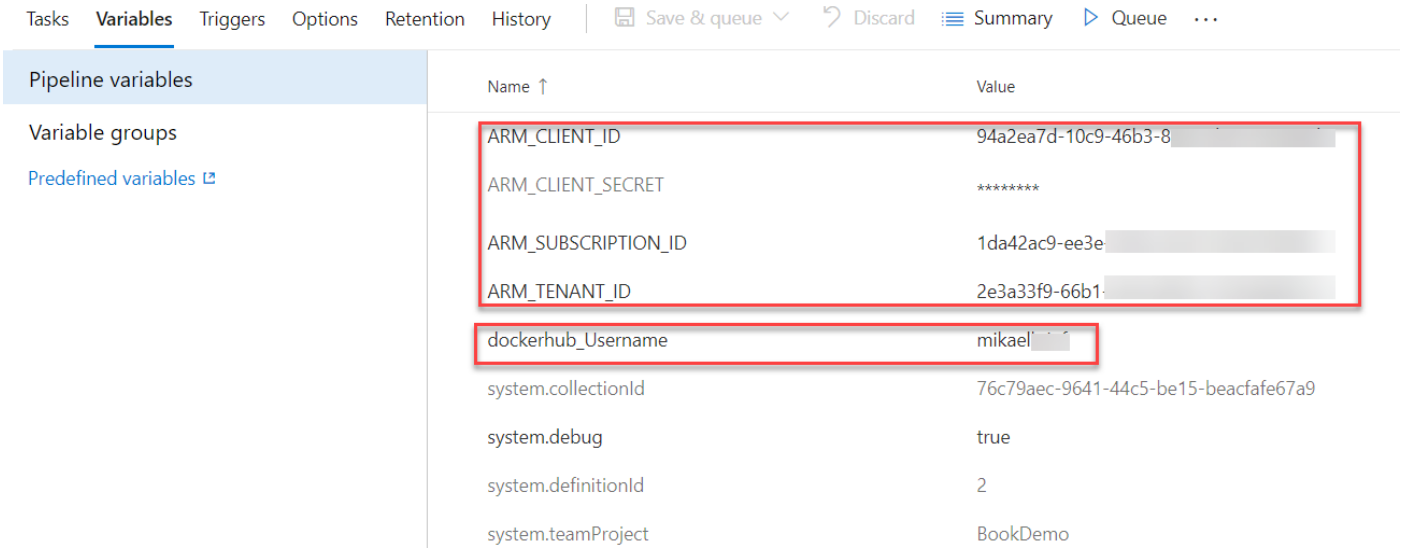
1. Create a new build definition whose Source code will point to the fork of the GitHub repository (https://github.com/PacktPublishing/Learning-DevOps), and select the root folder of this repository:



# Deploying a container to ACI with a CI/CD pipeline

- **Creating a CI/CD pipeline for the container:**

2. Then, on the Variables tab, define the variables that will be used in the pipeline. The following screenshot shows the information on the Variables tab:.



# Deploying a container to ACI with a CI/CD pipeline

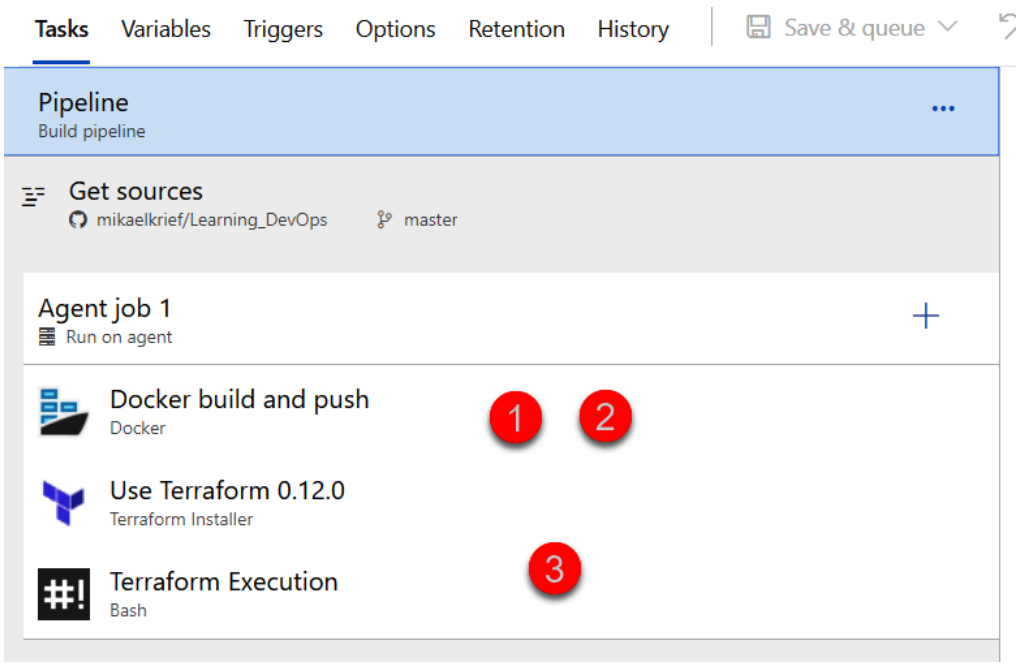
- **Creating a CI/CD pipeline for the container:**

3. Then, on the Tasks tab, take the following steps:

    1. Run the docker build command on the Dockerfile.

    2. Push the image to Docker Hub.

    3. Run the Terraform code to update the ACI with the new version of the updated image.

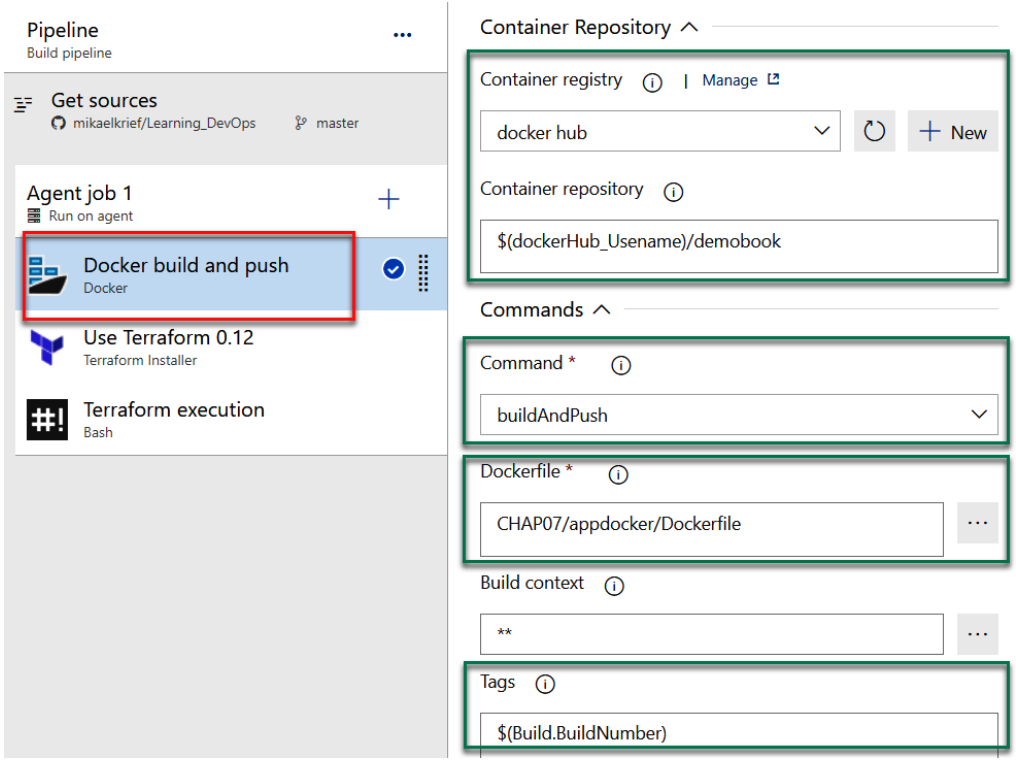


# Deploying a container to ACI with a CI/CD pipeline

- **Creating a CI/CD pipeline for the container:**

4. The first task, Docker build and push, allows you to build the Docker image and push it to Docker Hub. Its configuration is quite simple:

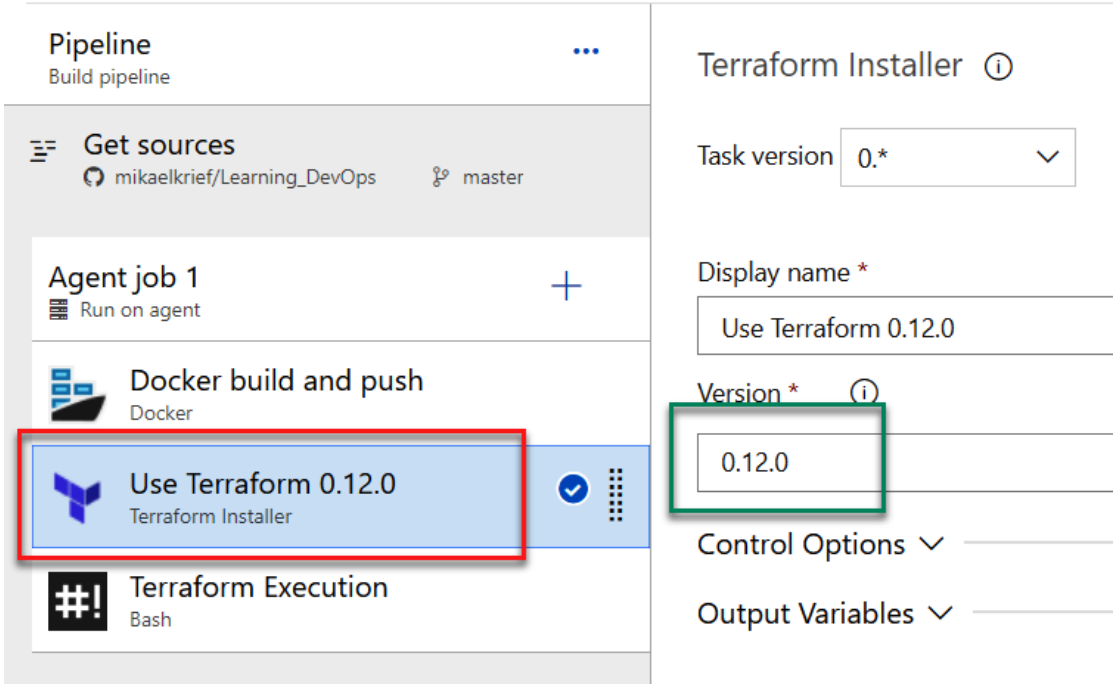
- Its required parameters are:
  - The connection to Docker Hub
  - The tag of the image that will be pushed to Docker Hub



# Deploying a container to ACI with a CI/CD pipeline

- **Creating a CI/CD pipeline for the container:**

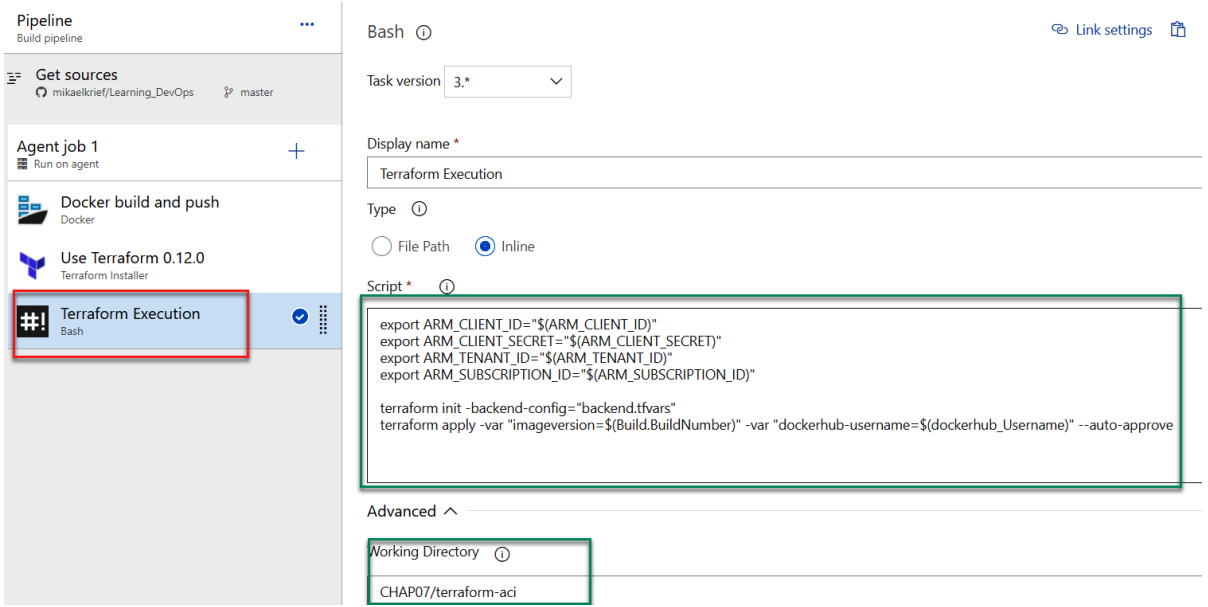
5. The second task, Terraform Installer, allows you to download Terraform on the pipeline agent by specifying the version of Terraform that you want:



# Deploying a container to ACI with a CI/CD pipeline

- **Creating a CI/CD pipeline for the container:**

6. The last task, Bash, allows you to execute a Bash script, and this screenshot shows its configuration:



# Deploying a container to ACI with a CI/CD pipeline

- Creating a CI/CD pipeline for the container:
- The configured script is as follows:

```
export ARM_CLIENT_ID="$(ARM_CLIENT_ID)"
export ARM_CLIENT_SECRET="$(ARM_CLIENT_SECRET)"
export ARM_TENANT_ID="$(ARM_TENANT_ID)"
export ARM_SUBSCRIPTION_ID="$(ARM_SUBSCRIPTION_ID)"
terraform init -backend-config="backend.tfvars"
terraform apply -var "imageversion=$(Build.BuildNumber)" -var
"dockerhub-username=$(dockerhub_Username)" --auto-approve
```

# Deploying a container to ACI with a CI/CD pipeline

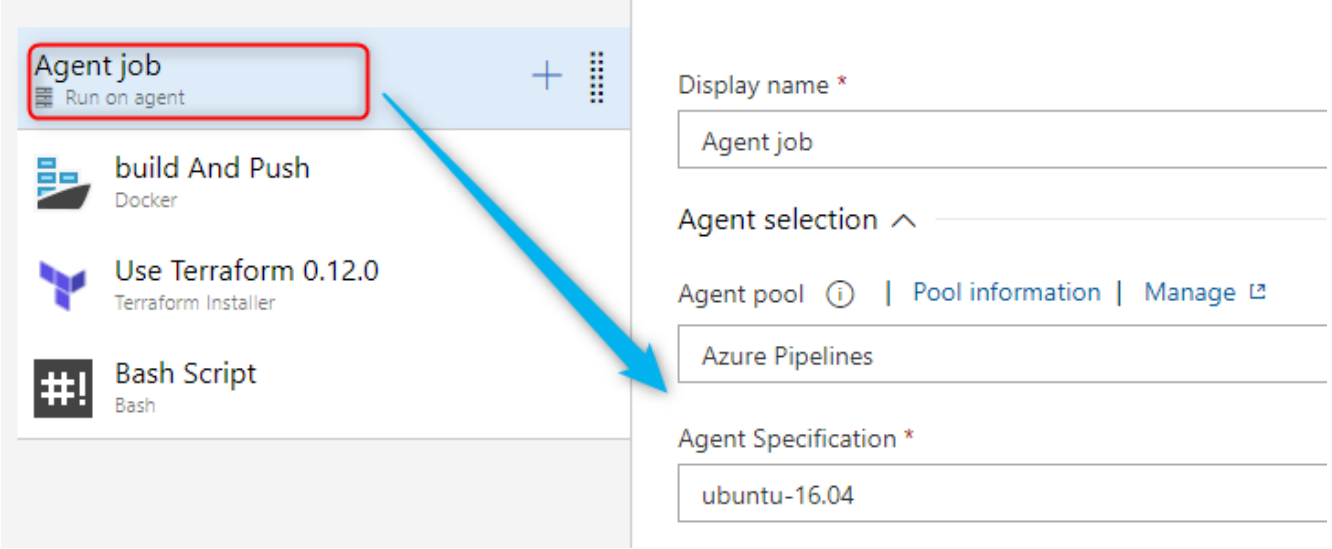
- **Creating a CI/CD pipeline for the container:**
- This script performs three actions, which are done in order:

1. Exports the environment variables required for Terraform.
2. Executes the terraform init command.
3. Executes terraform apply to apply the changes, with the two -var parameters, which are our Docker Hub username as well as the tag to apply. These parameters allow the execution of a container with the new image that has just been pushed to Docker Hub.

# Deploying a container to ACI with a CI/CD pipeline

- **Creating a CI/CD pipeline for the container:**

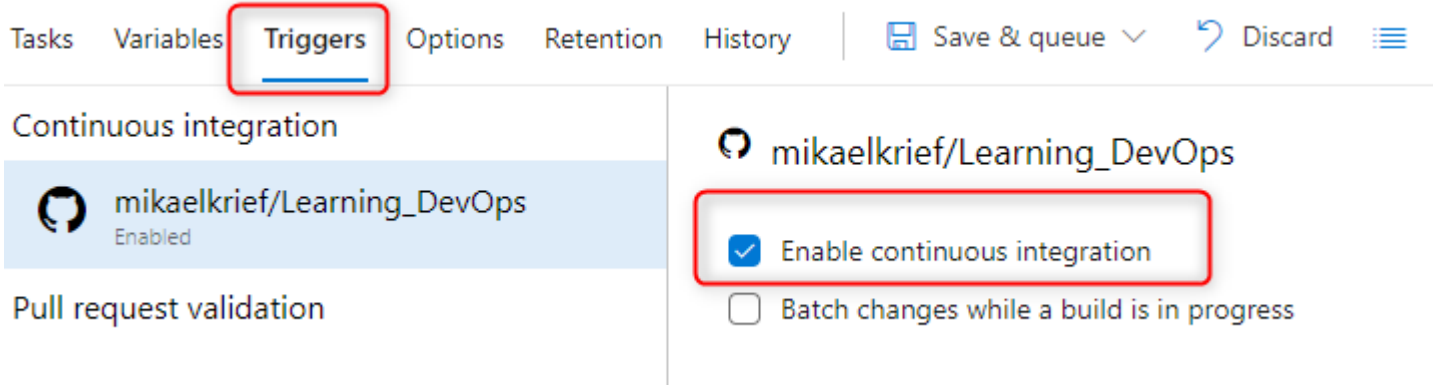
7. Then, to configure the build agent to use in the Agent job options, use the Azure Pipelines agent hosted Ubuntu 16.04, shown in the following screenshot:



# Deploying a container to ACI with a CI/CD pipeline

- **Creating a CI/CD pipeline for the container:**

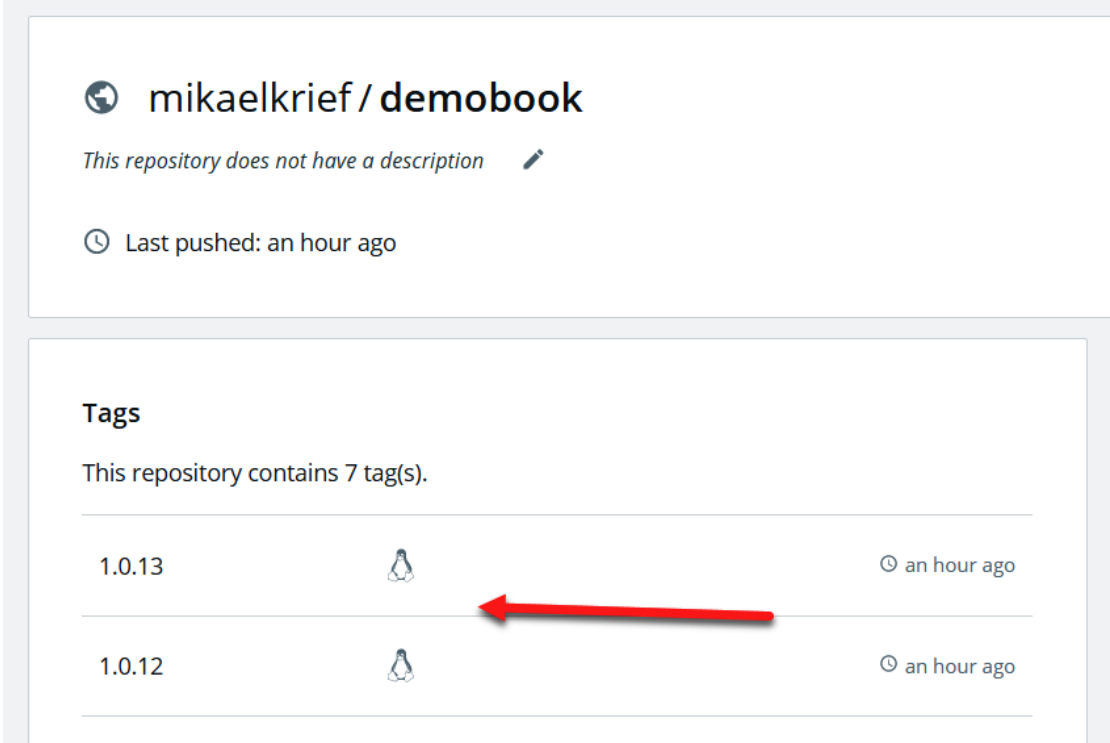
8. Finally, the last configuration is the trigger configuration on the Triggers tab, to enable the continuous integration with the trigger of this build at each commit :




That is the configuration of the CI/CD pipeline in Azure Pipelines.

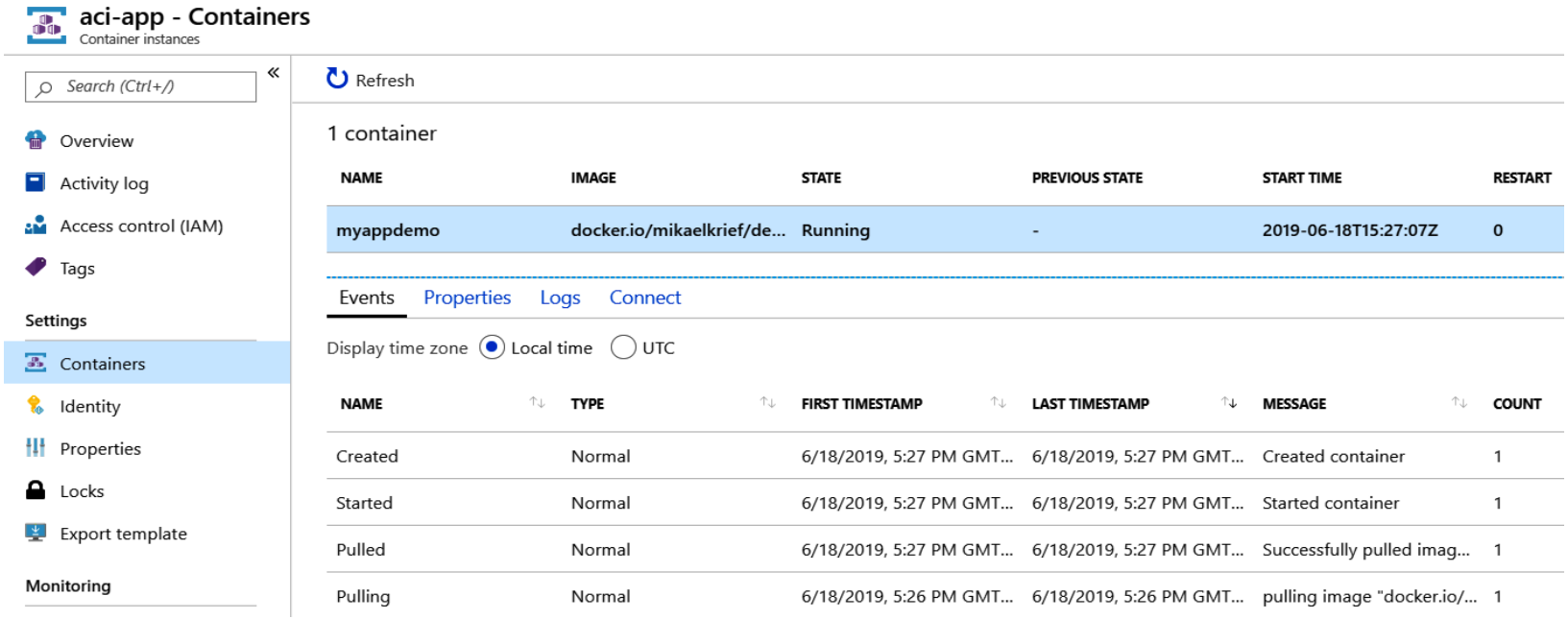
# Deploying a container to ACI with a CI/CD pipeline

- Trigger this build and at the end of its execution, notice a new version of the Docker image which corresponds to the number of the build that pushed the Docker image into the Docker Hub:



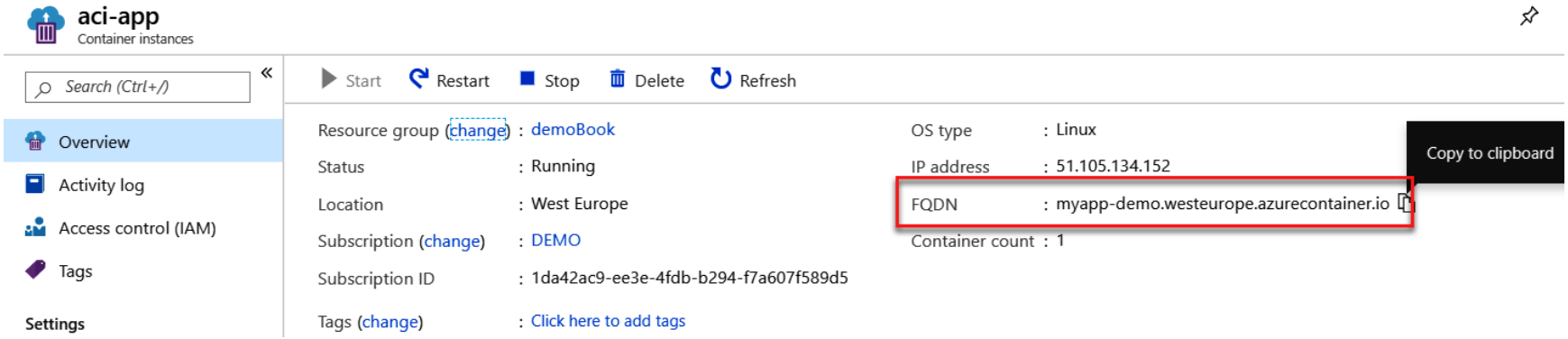
# Deploying a container to ACI with a CI/CD pipeline

- In the Azure portal, we have our ACI, aci-app, with our container, mydemoapp :



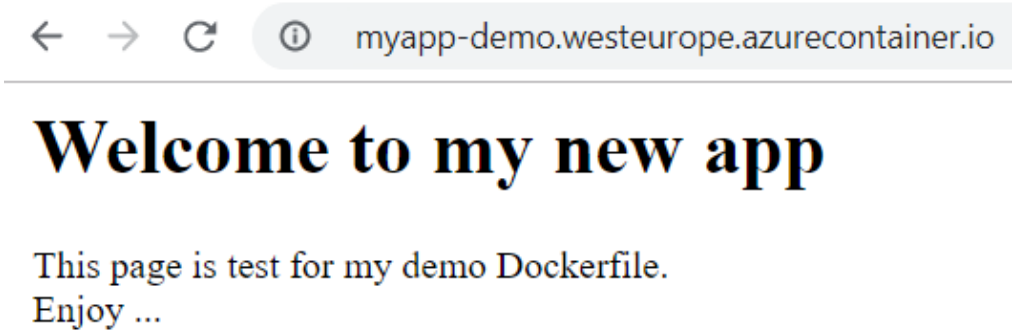
# Deploying a container to ACI with a CI/CD pipeline

- Notice that the container is running well.
- Now, to access our application, we need to retrieve the public FQDN URL of the container provided in the Azure portal:



# Deploying a container to ACI with a CI/CD pipeline

- Open a web browser with this URL Our web application is displayed correctly:




- The next time the application is updated, the CI/CD build is triggered, a new version of the image will be pushed into Docker Hub, and a new container will be loaded with this new version of the image.

RAMAIAH
Institute of Technology

# Managing Containers Effectively with Kubernetes

# Introduction to Kubernetes

- There are two major container orchestration tools on the market:
  - Docker Swarm
  - Kubernetes

- The major difference between the platforms is based on complexity. Kubernetes is well suited for complex applications.
- On the other hand, Docker Swarm is designed for ease of use, making it a preferable choice for simple applications.

# Introduction to Kubernetes

- Features of Kubernetes
  - Automated Scheduling
  - Self-Healing Capabilities
  - Automated rollouts & rollback
  - Horizontal Scaling & Load Balancing
  - Offers environment consistency for development, testing, and production

# Introduction to Kubernetes

- Features of Kubernetes
  - Infrastructure is loosely coupled to each component can act as a separate unit.
  - Provides a higher density of resource utilization
  - Offers enterprise-ready features
  - Application-centric management
  - Auto-scalable infrastructure
  - You can create predictable infrastructure

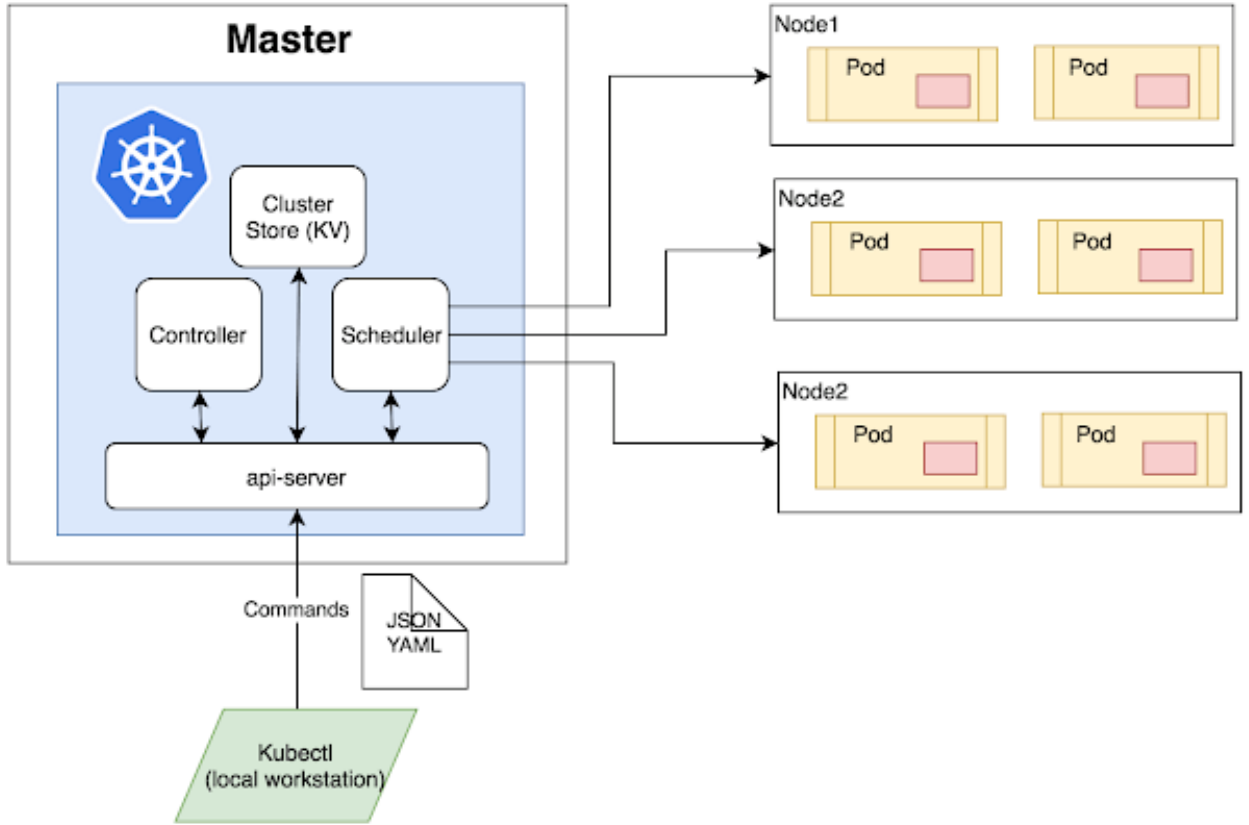
# Introduction to Kubernetes

- Kubernetes Basics
- Cluster: It is a collection of hosts(servers) that helps you to aggregate their available resources. That includes ram, CPU, ram, disk, and their devices into a usable pool.
- Master: The master is a collection of components which make up the control panel of Kubernetes. These components are used for all cluster decisions. It includes both scheduling and responding to cluster events.

# Introduction to Kubernetes

- Kubernetes Basics
- **Node:** It is a single host which is capable of running on a physical or virtual machine. A node should run both kube-proxy, minikube, and kubelet which are considered as a part of the cluster.
- **Namespace:** It is a logical cluster or environment. It is a widely used method which is used for scoping access or dividing a cluster.

# Kubernetes Architecture



# Kubernetes Architecture

- Master Node:
  - The master node is the first and most vital component which is responsible for the management of Kubernetes cluster.
  - It is the entry point for all kind of administrative tasks.
  - There might be more than one master node in the cluster to check for fault tolerance.
  - The master node has various components like API Server, Controller Manager, Scheduler, and ETCD.

# Kubernetes Architecture

- API Server:
  - The API server acts as an entry point for all the REST commands used for controlling the cluster.
- Scheduler:
  - The scheduler schedules the tasks to the slave node.
  - It stores the resource usage information for every slave node.
  - It is responsible for distributing the workload.

# Kubernetes Architecture

- Scheduler:
  - It also helps you to track how the working load is used on cluster nodes.
  - It helps you to place the workload on resources which are available and accept the workload.
- Etcd:
  - etcd components store configuration detail and wright values.
  - It communicates with the most component to receive commands and work.
  - It also manages network rules and port forwarding activity.

# Kubernetes Architecture

- **Worker/Slave nodes**:
  - Worker nodes are another essential component which contains all the required services to manage the networking between the containers, communicate with the master node, which allows you to assign resources to the scheduled containers.
- **Kubelet:**
  - This gets the configuration of a Pod from the API server and ensures that the described containers are up and running.

# Kubernetes Architecture

- **Docker Container**:
  - Docker container runs on each of the worker nodes, which runs the configured pods.
- **Kube-proxy:**
  - Kube-proxy acts as a load balancer and network proxy to perform service on a single worker node.
- **Pods:**
  - A pod is a combination of single or multiple containers that logically run together on nodes.

# Kubernetes - Other Key Terminologies

- **Replication Controllers**
  - A replication controller is an object which defines a pod template.
  - It also controls parameters to scale identical replicas of Pod horizontally by increasing or decreasing the number of running copies.
- **Replication Sets**
  - Replication sets are an interaction on the replication controller design with flexibility in how the controller recognizes the pods it is meant to manage.
  - It replaces replication controllers because of their higher replicate selection capability.

# Kubernetes - Other Key Terminologies

- **Deployments**
  - Deployment is a common workload which can be directly created and manage.
  - Deployment use replication set as a building block which adds the feature of life cycle management.
- **Stateful Sets**
  - It is a specialized pod control which offers ordering and uniqueness.
  - It is mainly used to have fine-grained control, which you have a particular need regarding deployment order, stable networking, and persistent data.

# Kubernetes - Other Key Terminologies

- Daemon Sets
  - Daemon sets are another specialized form of pod controller that runs a copy of a pod on every node in the cluster.
  - This type of pod controller is an effective method for deploying pods that allows you to perform maintenance and offers services for the nodes themselves.

# Kubernetes vs. Docker Swarm

| Parameter | Docker Swarm | Kubernetes |
|---|---|---|
| Scaling | No Autoscaling | Auto-scaling |
| Load balancing | Does auto load balancing | Manually configure your load balancing settings |
| Storage volume sharing | Shares storage volumes with any other container | Shares storage volumes between multiple containers inside the same Pod |
| Use of logining and monitoring tool | Use 3rd party tool like ELK | Provide an in-built tool for logging and monitoring. |
| Installation | Easy & fast | Complicated & time-consuming |
| GUI | GUI not available | GUI is available |
| Scalability | Scaling up is faster than K8S, but cluster strength not as robust | Scaling up is slow compared to Swarm, but guarantees stronger cluster state Load balancing requires manual service configuration |
| Load Balancing | Provides a built-in load balancing technique | Process scheduling to maintain services while updating |
| Updates & Rollbacks Data Volumes Logging & Monitoring | Progressive updates and service health monitoring. | Only shared with containers in same Pod Inbuilt logging & monitoring tools. |

# Introduction to Kubernetes

- Advantages of Kubernetes:
  - Easy organization of service with pods
  - It is developed by Google, who bring years of valuable industry experience to the table
  - Largest community among container orchestration tools
  - Offers a variety of storage options, including on-premises, SANs and public clouds
  - Adheres to the principals of immutable infrastructure
  - Kubernetes can run on-premises bare metal, OpenStack, public clouds Google, Azure, AWS, etc.

# Introduction to Kubernetes

- Advantages of Kubernetes:
  - Helps you to avoid vendor lock issues as it can use any vendor-specific APIs or services except where Kubernetes provides an abstraction, e.g., load balancer and storage.
  - Containerization using kubernetes allows package software to serve these goals. It will enable applications that need to be released and updated without any downtime.
  - Kubernetes allows you to assure those containerized applications run where and when you want and helps you to find resources and tools which you want to work.

# Introduction to Kubernetes

- **Disadvantages of Kubernetes:**
  - Kubenetes dashboard not as useful as it should be
  - Kubernetes is a little bit complicated and unnecessary in environments where all development is done locally.
  - Security is not very effective.

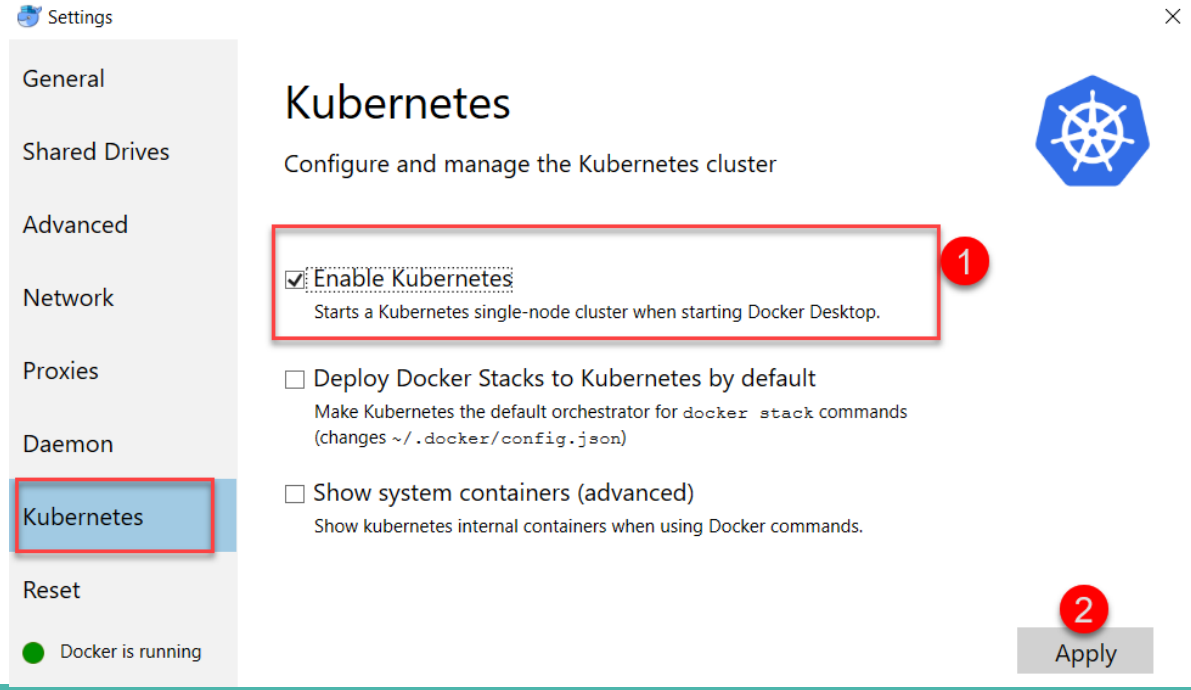
# Installing Kubernetes on a local machine

- When developing a containerized application that is to be hosted on Kubernetes, it is important to be able to run the application (with its containers) on your local machine, before deploying it on remote Kubernetes production clusters.
- In order to install a Kubernetes cluster locally, there are several solutions, which are as follows:
- The **first solution** is to use Docker Desktop.

1. In Docker Desktop, activate the Enable Kubernetes option in Settings in Kubernetes tab

# Installing Kubernetes on a local machine

1. In Docker Desktop, activate the Enable Kubernetes option in Settings in Kubernetes tab



# Installing Kubernetes on a local machine

2. After clicking on the Apply button, Docker Desktop will install a mini Kubernetes cluster, and the kubectl client tool, on the local machine.

- The second solution is to install Minikube, which also installs a simplified Kubernetes cluster locally.
- Following the local installation of Kubernetes, check its installation by executing the following command in a Terminal:
- kubectl version --short

```
C:\Users\Mikael>kubectl version --short
Client Version: v1.14.6
Server Version: v1.13.10
```

# Installing the Kubernetes dashboard

- After installing our Kubernetes cluster, there is a need for another element, which is the Kubernetes dashboard.
- In order to install the Kubernetes dashboard, which is a pre-packaged containerized web application that will be deployed in our cluster, we will run the following command in a Terminal:

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/master/aio/deploy/re
commended.yaml
```

# Installing the Kubernetes dashboard

- Its execution is shown in the following screenshot:

```
                    >kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-beta1/aio/deploy/recommended.yaml
namespace "kubernetes-dashboard" created
serviceaccount "kubernetes-dashboard" created
service "kubernetes-dashboard" created
secret "kubernetes-dashboard-certs" created
secret "kubernetes-dashboard-csrf" created
secret "kubernetes-dashboard-key-holder" created
configmap "kubernetes-dashboard-settings" created
role.rbac.authorization.k8s.io "kubernetes-dashboard" created
clusterrole.rbac.authorization.k8s.io "kubernetes-dashboard" created
rolebinding.rbac.authorization.k8s.io "kubernetes-dashboard" created
clusterrolebinding.rbac.authorization.k8s.io "kubernetes-dashboard" created
deployment.apps "kubernetes-dashboard" created
service "dashboard-metrics-scraper" created
deployment.apps "kubernetes-metrics-scraper" created
```

# Installing the Kubernetes dashboard

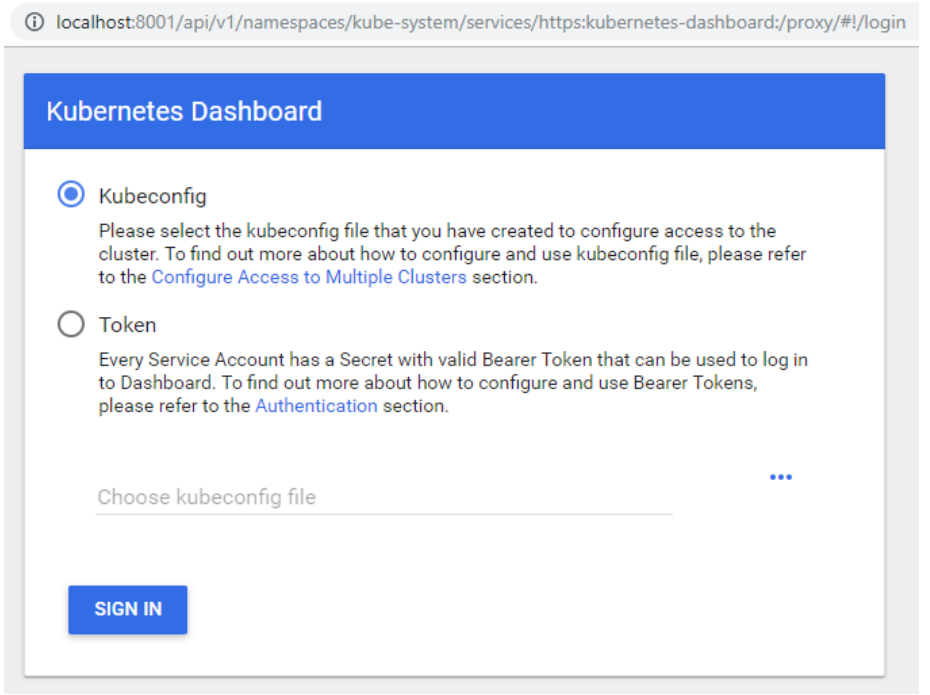
- To open the dashboard and connect to it from our local machine, first create a proxy between the Kubernetes cluster and our machine by performing the following steps:
- 1. To create the proxy, we execute the kubectl proxy command in a Terminal, and the detail of the execution is shown in the following screenshot:

```
>kubectl proxy
Starting to serve on 127.0.0.1:8001
```

- The proxy is open on the localhost address (127.0.0.1) with the 8001 port.

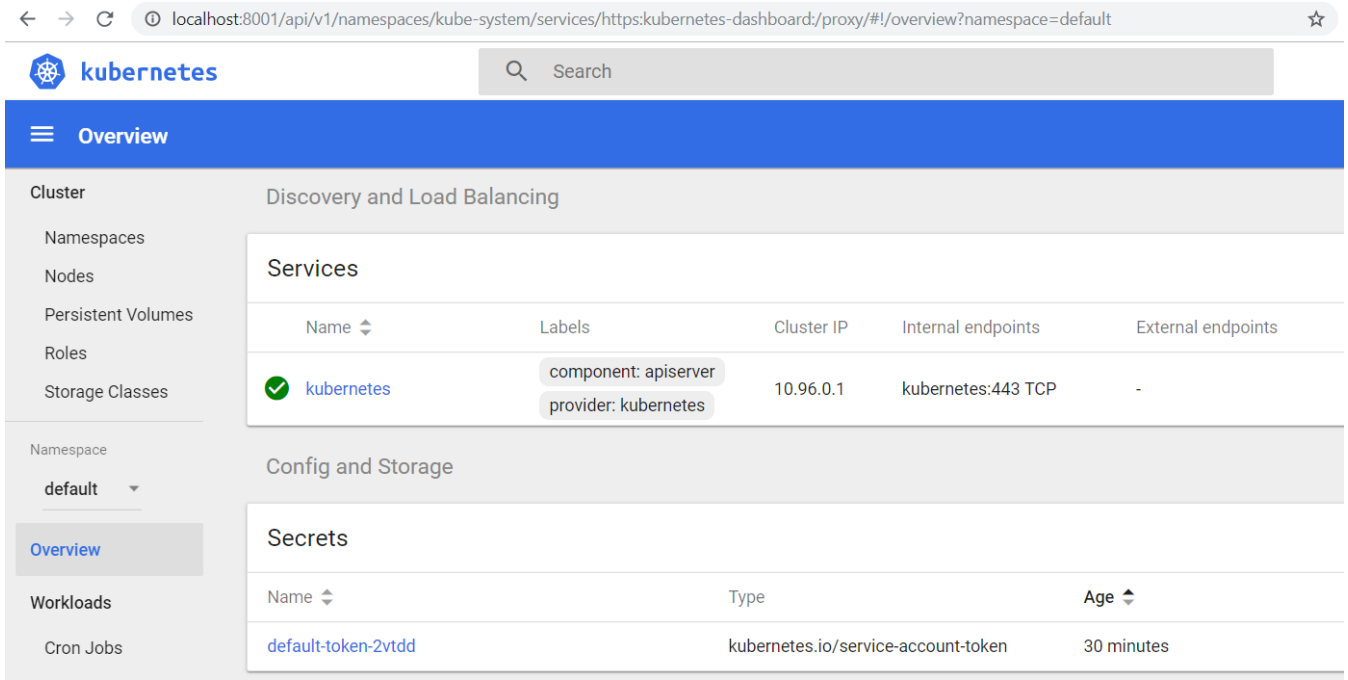
# Installing the Kubernetes dashboard

- Then, in a web browser, open the URL
- http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/#/login
- This is a local URL (localhost and 8001) that is created by the proxy, and that points to the Kubernetes dashboard application that we have installed.



# Installing the Kubernetes dashboard

- After clicking on the SIGN IN button, the dashboard is displayed as follows:



# First example of Kubernetes application deployment

- After installing our Kubernetes cluster, deploy an application in it.
- First of all, it is important to know that when deploying an application in Kubernetes, create a new instance of the Docker image in a cluster pod, and need to have a Docker image that contains the application.
- To deploy a instance of the Docker image, create a new k8sdeploy folder, and, inside it, create a Kubernetes deployment YAML specification file (myappdeployment.yml) with the following content:

# First example of Kubernetes application deployment

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
 name: webapp
spec:
 selector:
    matchLabels:
      app: webapp
 replicas: 2
 template:
    metadata:
     labels:
       app: webapp
   spec:
      containers:
      - name: demobookk8s
        image: mikaelkrief/demobook:latest
        ports:
        - containerPort: 80
```

# First example of Kubernetes application deployment

- In this code, description of deployment is as follows:
  - The apiVersion property is the version of api that should be used.
  - In the Kind property, we indicate that the specification type is deployment.
  - The replicas property indicates the number of pods that Kubernetes will create in the cluster; here, we choose two instances.

# First example of Kubernetes application deployment

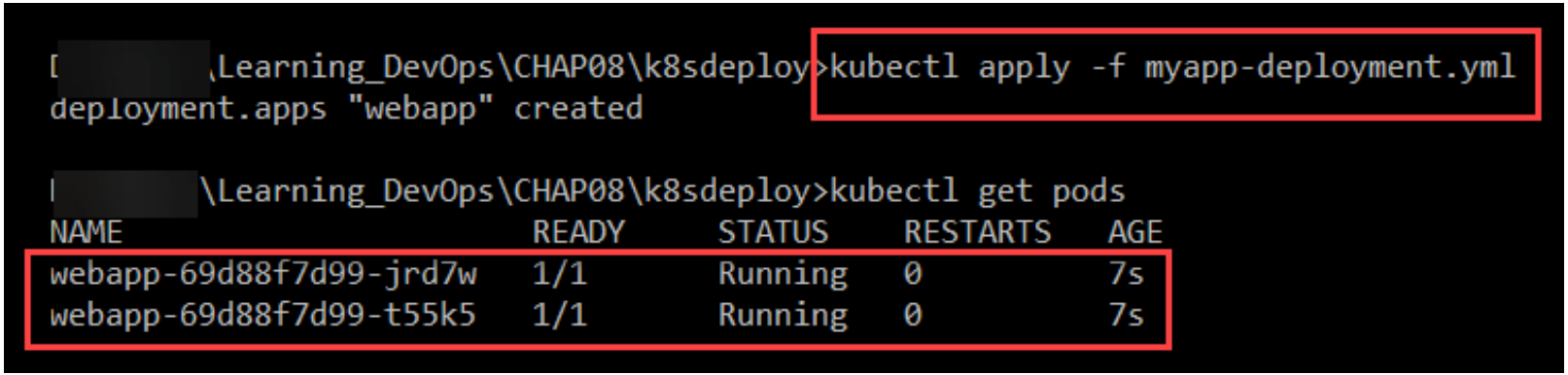
- In this example, chose two replicas, which can, at the very least, distribute the traffic charge of the application (put in more replicas if there is a high volume of load).
- And also ensure the proper functioning of the application.
- Therefore, if one of the two pods has a problem, the other, which is an identical replica, will ensure the proper functioning of the application.
- Then, in the containers section, we indicate the image (from the Docker Hub) with name and tag.
- Finally, the ports property indicates the port that the container will use within the cluster.

# First example of Kubernetes application deployment

- To deploy our application, we go to our Terminal, and execute one of the essential kubectl commands (kubectl apply) as follows:
  - kubectl apply -f myapp-deployment.yml
- The -f parameter corresponds to the YAML specification file.
- This command applies the deployment that is described in the YAML specification file on the Kubernetes cluster.
- Following the execution of this command, check the status of this deployment, by displaying the list of pods in the cluster.

# First example of Kubernetes application deployment

- To do this in the Terminal, we execute the kubectl get pods command, which returns the list of cluster pods.
- The following screenshot shows the execution of the deployment and displays the information in the pods, which we use to check the deployment:



# First example of Kubernetes application deployment

- In the preceding screenshot, the second command displays two pods, with the name (webapp) specified in the YAML file, followed by a unique ID, and Running status.
- Also visualize the status of cluster on the Kubernetes web dashboard, the webapp deployment with the Docker image that has been used, and the two pods that have been created.
- The application has been successfully deployed in Kubernetes cluster.
- But, for the moment, it is only accessible inside the cluster only.
- And for it to be usable, we need to expose it outside the cluster.

# First example of Kubernetes application deployment

- In order to access the web application outside the cluster, add a service type and a NodePort category element to the cluster.
- To add this service type and NodePort, in the same way as for deployment, create a second YAML file (myapp-service.yml) of the service specification in the same k8sdeploy directory, which has the following code:
- In this code, we specify the kind, Service, as well as the type of service, NodePort.

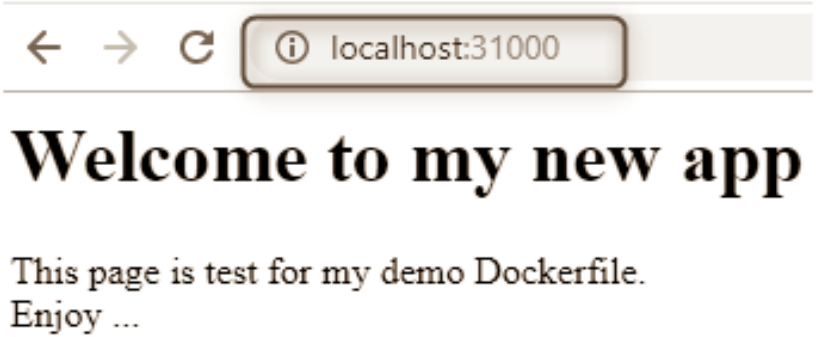
```
---
apiVersion: v1
kind: Service
metadata:
 name: webapp
 labels:
 app: webapp
spec:
 type: NodePort
 ports:
 - port: 80
   targetPort: 80
   nodePort: 31000
selector:
 app: webapp
```

# First example of Kubernetes application deployment

- Then, in the ports section, we specify the port translation: the 80 port, which is exposed internally, and the 31000 port, which is exposed externally to the cluster.
- To create this service on the cluster, we execute the kubectl apply command, but this time with our myapp-service.yaml file as a parameter, as follows:
- **kubectl apply -f myapp-service.yml**

# First example of Kubernetes application deployment

- The execution of the command creates the service within the cluster, and, to test the application, open a web browser with the http://localhost:31000 URL, and the page is displayed as follows:




- The application is now deployed on a Kubernetes cluster, and it can be accessed from outside the cluster.

# Using HELM as a package manager

- As previously discussed, all the actions that are carried out on the Kubernetes cluster are done via the kubectl tool and the YAML specification files.
- In a company that deploys several microservice applications on a K8S cluster, often notice a large number of these YAML specification files, and this poses a maintenance problem.
- In order to solve this maintenance problem, use HELM, which is the package manager for Kubernetes.

# Using HELM as a package manager

- HELM is, therefore, a repository that will allow the sharing of packages called charts, and that contain ready-to-use Kubernetes specification file templates.
- HELM is composed of two parts:
- A **client tool**, which allows us to list the packages of a repository, and to indicate the package(s) to be installed.
- A **server tool** called Tiller, which is in the Kubernetes cluster, and receives information from the client tool and installs the package charts.

# Using HELM as a package manager

- Installing Helm, and how to use it to deploy an application:

1. **Install the Helm client:**

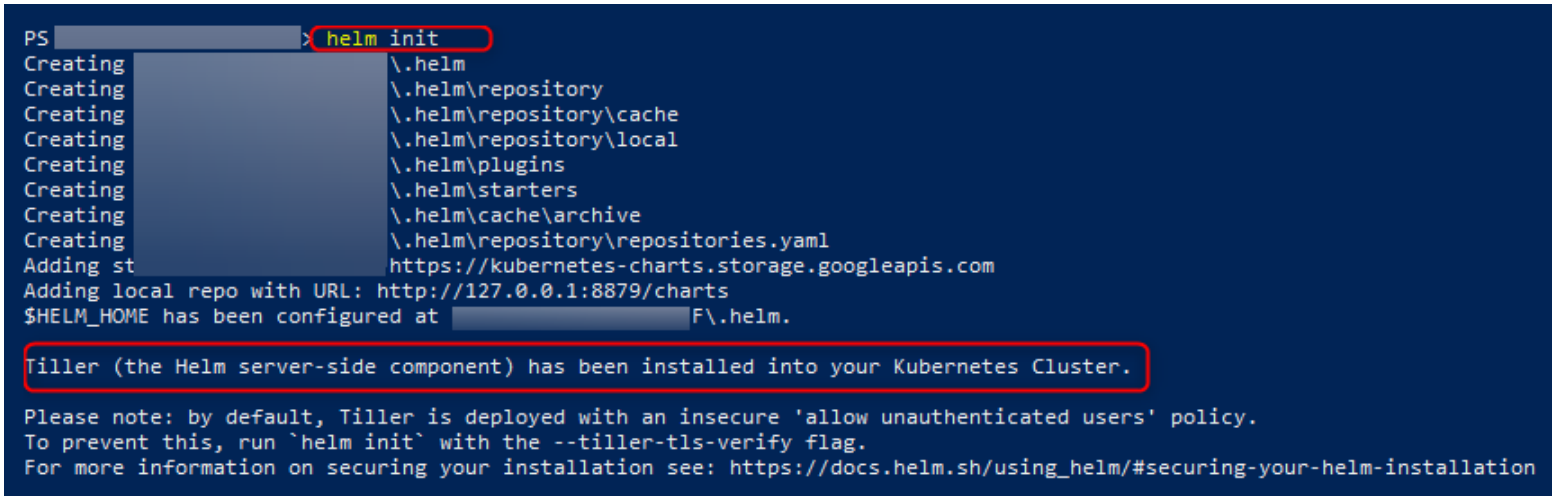
- In Windows
  - **choco install kubernetes-helm –y**
- To check its installation, execute the helm --help command

```
PS C:\Users\Mikael> helm --help
The Kubernetes package manager

To begin working with Helm, run the 'helm init' command:

        $ helm init

This will install Tiller to your running Kubernetes cluster.
It will also set up any necessary local configuration.

Common actions from this point include:

- helm search:    search for charts
- helm fetch:     download a chart to your local directory to view
- helm install:   upload the chart to Kubernetes
- helm list:      list releases of charts
```

# Using HELM as a package manager

2. **Install the Tiller**: To install the Helm server component on our Kubernetes cluster, execute the following command:

- **helm init**



# Using HELM as a package manager

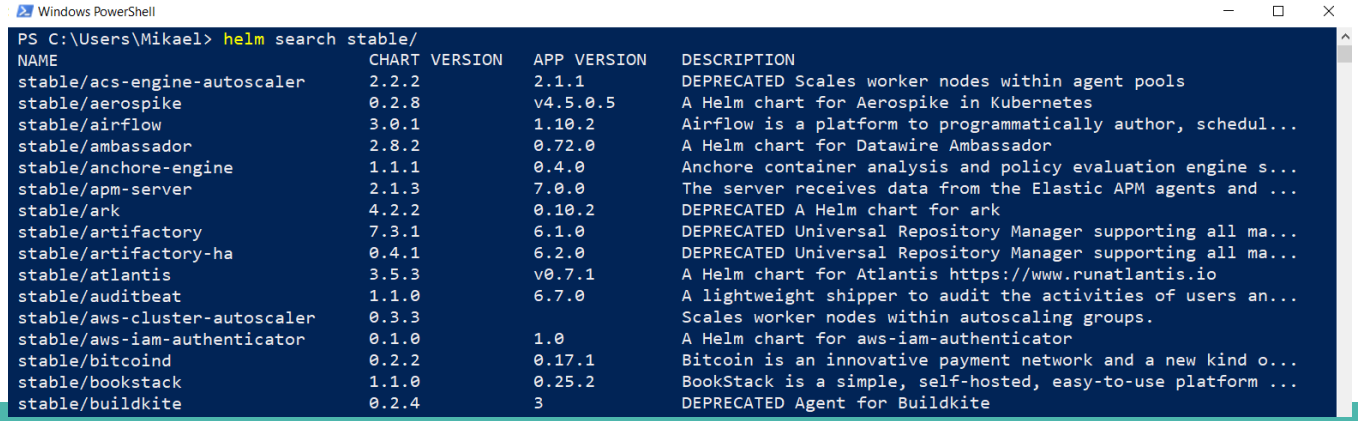
3. **Search charts**: The packages that are contained in a HELM repository are called charts.

- Charts are composed of files that are templates of Kubernetes specification files for an application.
- With the charts, it's possible to deploy an application in Kubernetes without having to write any YAML specification files.
- So, to deploy an application, we will use its corresponding chart, and we will pass some configuration variables of this application.

# Using HELM as a package manager

3. **Search charts**: Once HELM is installed, install a chart that is in the HELM public repository, but first, to display the list of public charts, run the following command:

- **helm search stable/**
- The stable/ parameter is the name of Helm's public repository.



# Using HELM as a package manager

4. **Deploy an application with Helm**: To illustrate the use of Helm, we will deploy a WordPress application in Kubernetes cluster by using a Helm chart.

- In order to do this, execute the helm install command as follows:
- **helm install stable/wordpress --name mywp**
- Helm installs a WordPress instance called mywp, and all of the Kubernetes components, on the local Kubernetes cluster.
- Also display the list of Helm packages that are installed on the cluster by executing the following command:
- **helm ls**

# Using HELM as a package manager

- And, to remove a package and all of its components, for example, to remove the application installed with this package, execute the helm delete command:
- **helm delete mywp –purge**
- The purge parameter indicates that everything has been deleted from this application.

# Using Azure Kubernetes Service (AKS)

- A production Kubernetes cluster can often be complex to install and configure.
- This type of installation requires the availability of servers, human resources with skills regarding the installation and management of a K8S cluster, and the implementation of an enhanced security policy to protect the applications.
- To overcome these problems, cloud providers offer managed Kubernetes cluster services.

# Using Azure Kubernetes Service (AKS)

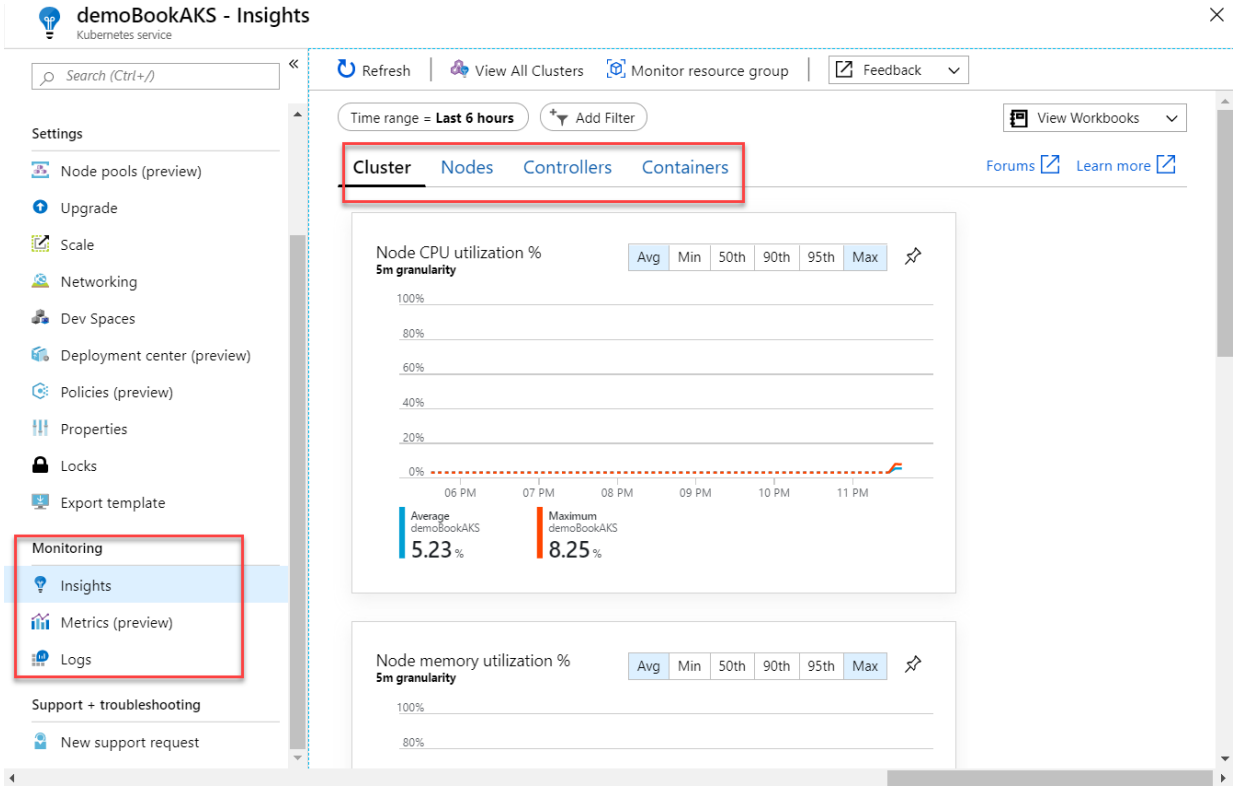
- AKS is an Azure service that allows us to create and manage a real Kubernetes cluster as a managed service.
- The advantage of this managed Kubernetes cluster is that we don't have to worry about its hardware installation, and that the management of the master part is done entirely by Azure when the nodes are installed on VMs.
- The use of this service is free; what is charged is the cost of the VMs on which the nodes are installed.

# Using Azure Kubernetes Service (AKS)

- Advantages of AKS
- AKS is a Kubernetes service that is managed in Azure.
- This has the advantage of being integrated with Azure.
- **Ready to use**: In AKS, the Kubernetes web dashboard is natively installed.
- **Integrated monitoring services**: AKS also has all of Azure's integrated monitoring services, including container monitoring, cluster performance management, and log management.

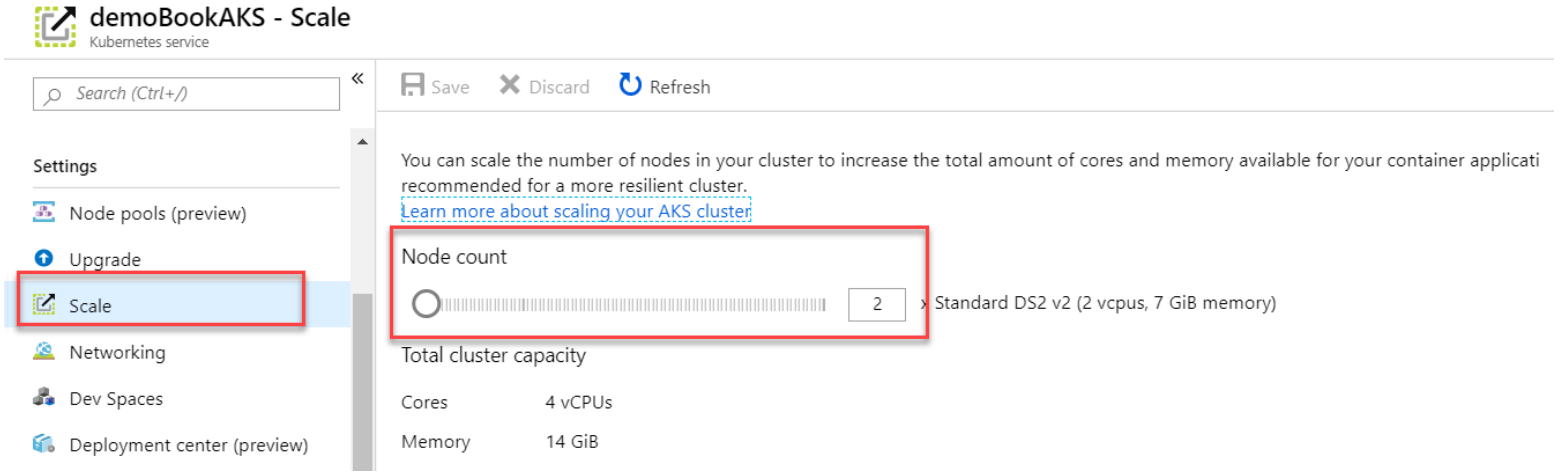
# Using Azure Kubernetes Service (AKS)

- Advantages of AKS
- Integrated monitoring services:



# Using Azure Kubernetes Service (AKS)

- Advantages of AKS
- **Very easy to scale**: AKS allows the quick and direct scaling of the number of nodes of a cluster via the portal, or via scripts.



# Using Azure Kubernetes Service (AKS)

- Advantages of AKS
- If we have an Azure subscription and we want to use Kubernetes, it's intuitive and quick to install.
- AKS has a number of advantages, such as integrated monitoring and scaling in the Azure portal.
- Using the kubectl tool does not require any changes compared to a local Kubernetes.

# Using Azure Kubernetes Service (AKS)

- If we have an Azure subscription and we want to use Kubernetes, it's intuitive and quick to install.
- AKS has a number of advantages, such as integrated monitoring and scaling in the Azure portal.
- Using the kubectl tool does not require any changes compared to a local Kubernetes.

# Creating a CI/CD pipeline for Kubernetes with Azure Pipelines

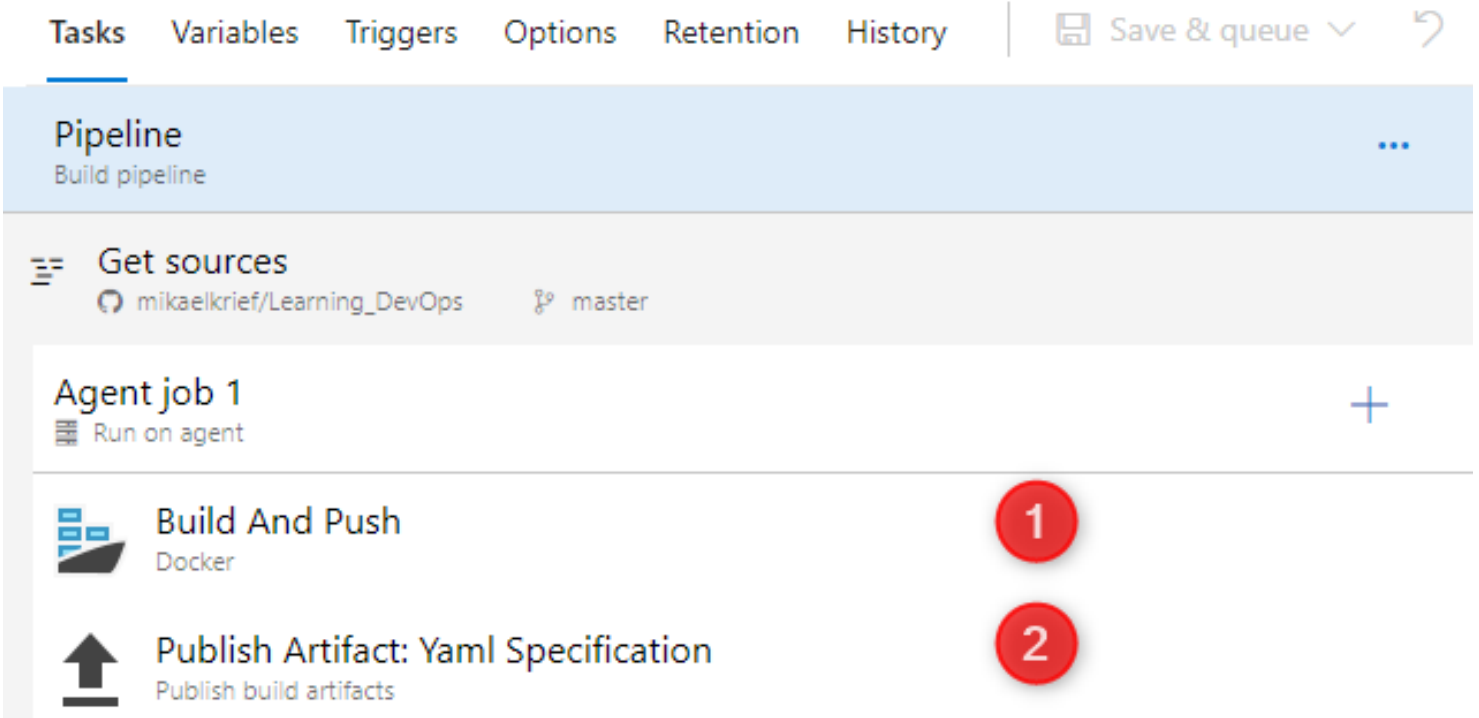
- Creating a complete CI/CD pipeline for Kubernetes, from the creation of a new Docker image pushed in the Docker Hub, to its deployment in an AKS cluster.
- To build this pipeline, we'll use the Azure Pipelines service that is in Azure DevOps.
- This continuous integration pipeline will be composed of the following:
  - A build that will be in charge of building and promoting a new Docker image in the Docker Hub.
  - A release that will use our YAML deployment specification file to deploy the latest version of the image in an AKS cluster.

# The build and push of the image in the Docker Hub

- In Azure DevOps, create a new build definition that will be in Classic design editor mode, and that will point to the source code that contains the Docker file.
- In this build definition, configure the Tasks tab with two steps, in this order:
  - The build and push of the Docker image.
  - The publication of the build artifacts, which are the K8S YAML specification files that will be deployed during the release.

# The build and push of the image in the Docker Hub

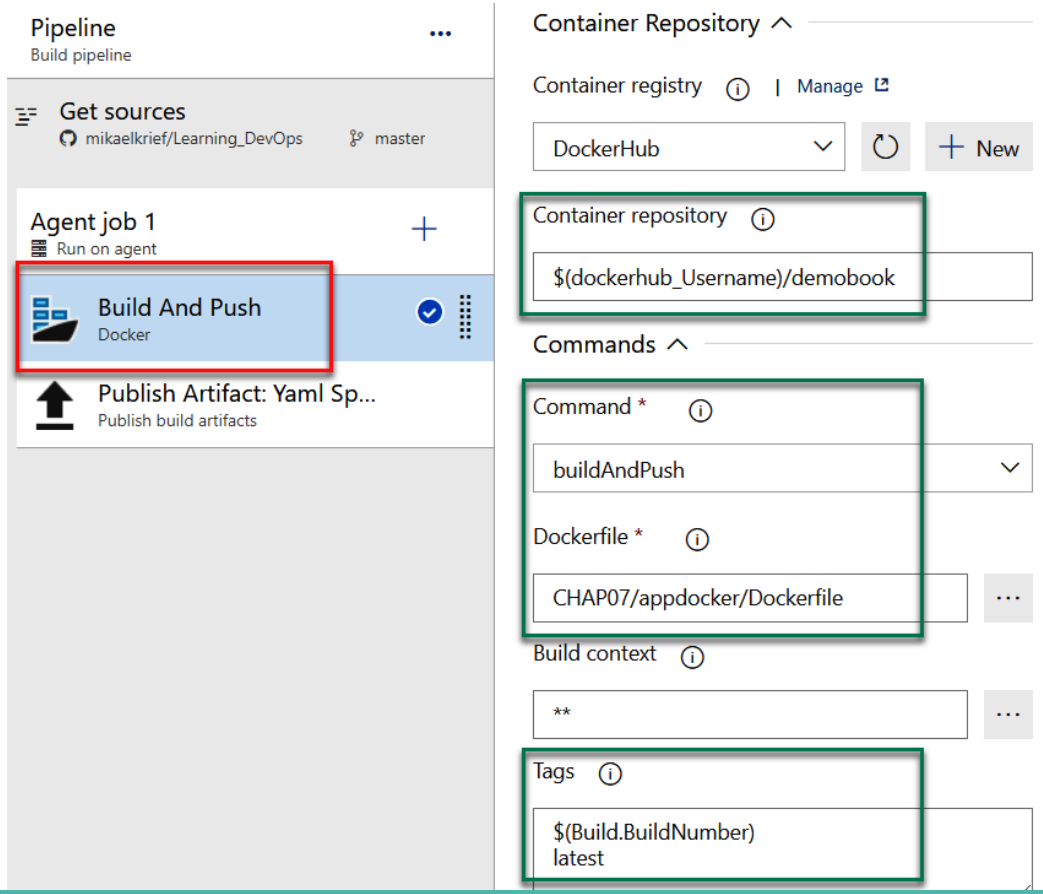
- The sequences of the tasks that configure the build pipeline are demonstrated in the following screenshot:



# The build and push of the image in the Docker Hub

- Detailed configuration steps of this build pipeline:

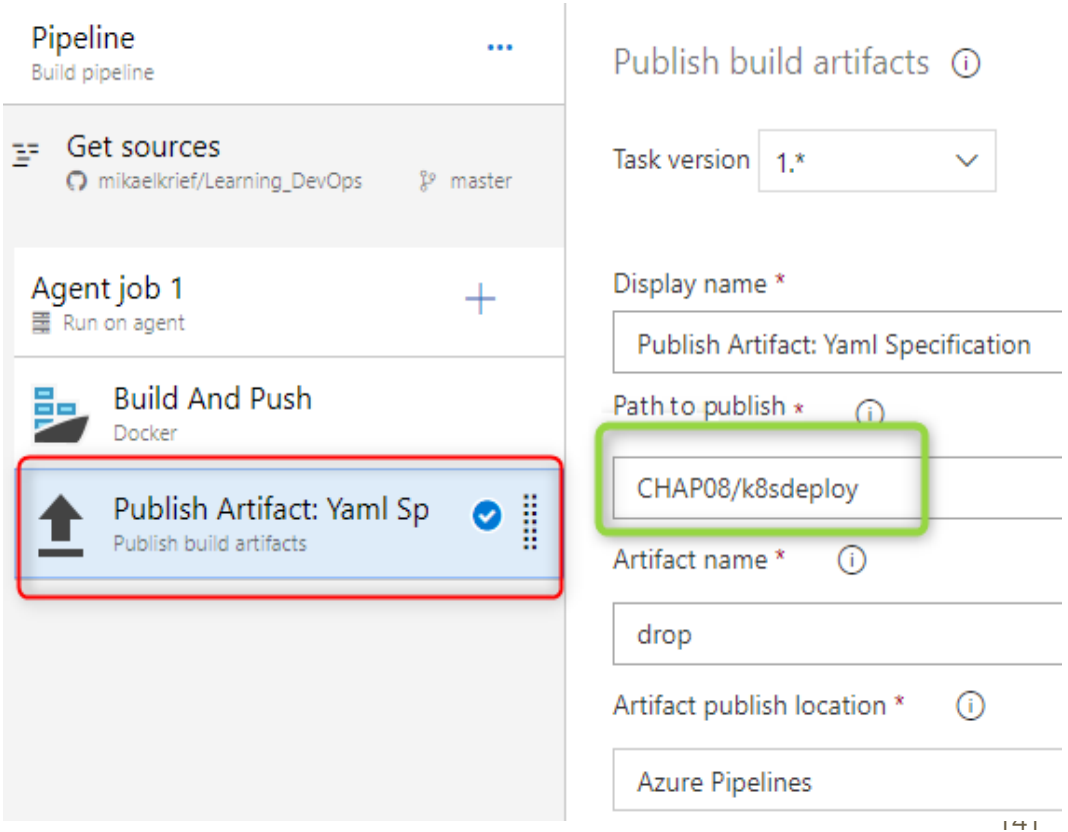
1. The configuration of the task that builds and pushes the Docker image:



# The build and push of the image in the Docker Hub

- Detailed configuration steps of this build pipeline:

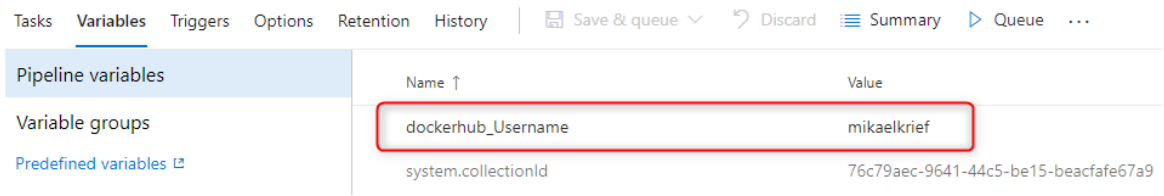
2. The configuration of the task that publishes artifacts of the Kubernetes YAML files as release artifacts, as follows



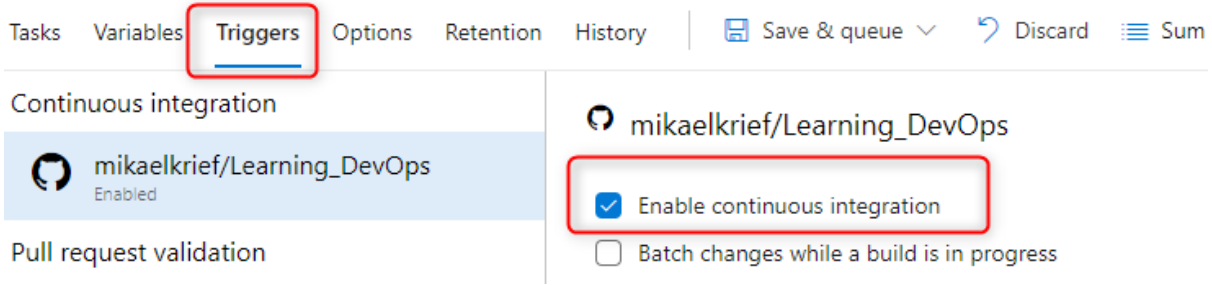
# The build and push of the image in the Docker Hub

- Detailed configuration steps of this build pipeline:

3. In the Variables tab, a variable is added that contains the Docker Hub username, as shown here:



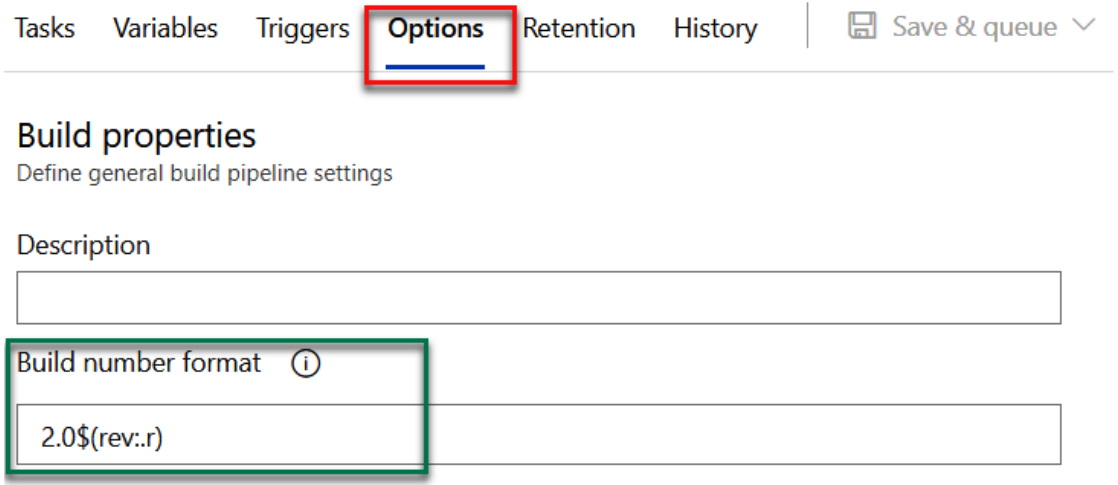

4. In the Triggers tab, continuous integration is enabled, as shown in the following screenshot:



# The build and push of the image in the Docker Hub

- Detailed configuration steps of this build pipeline:

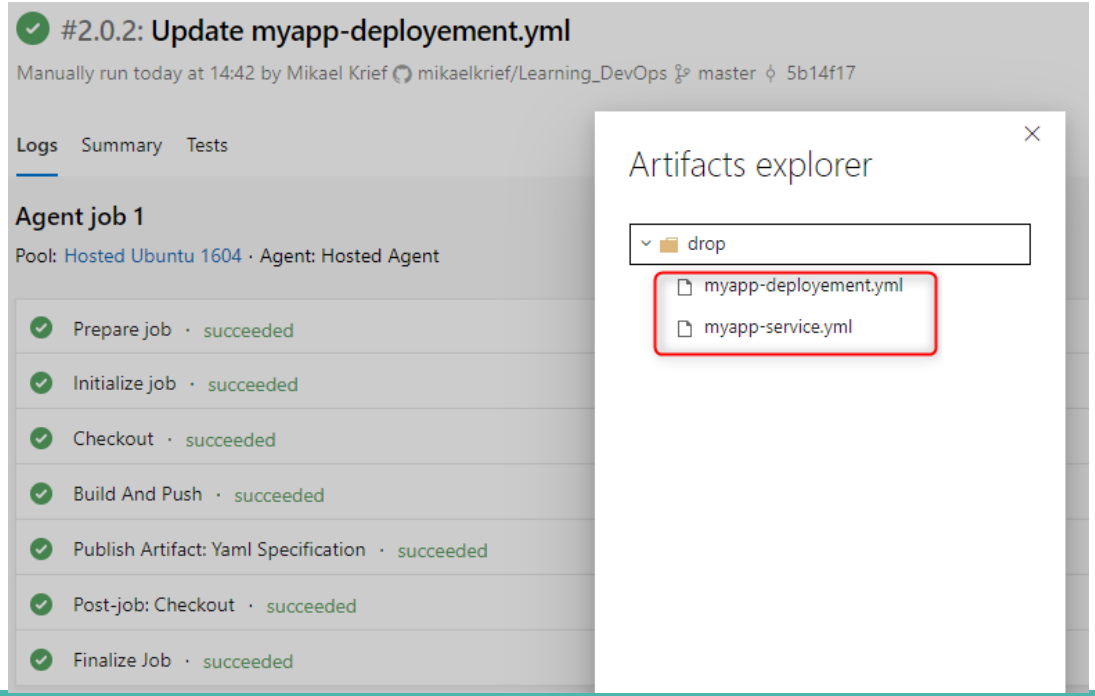
5. In the Options tab, we indicate the build number with the 2.0.patch pattern.




- This build number will be the tag of the Docker image that is uploaded into the Docker Hub. Once the configuration is finished, we save the build definition and execute it.

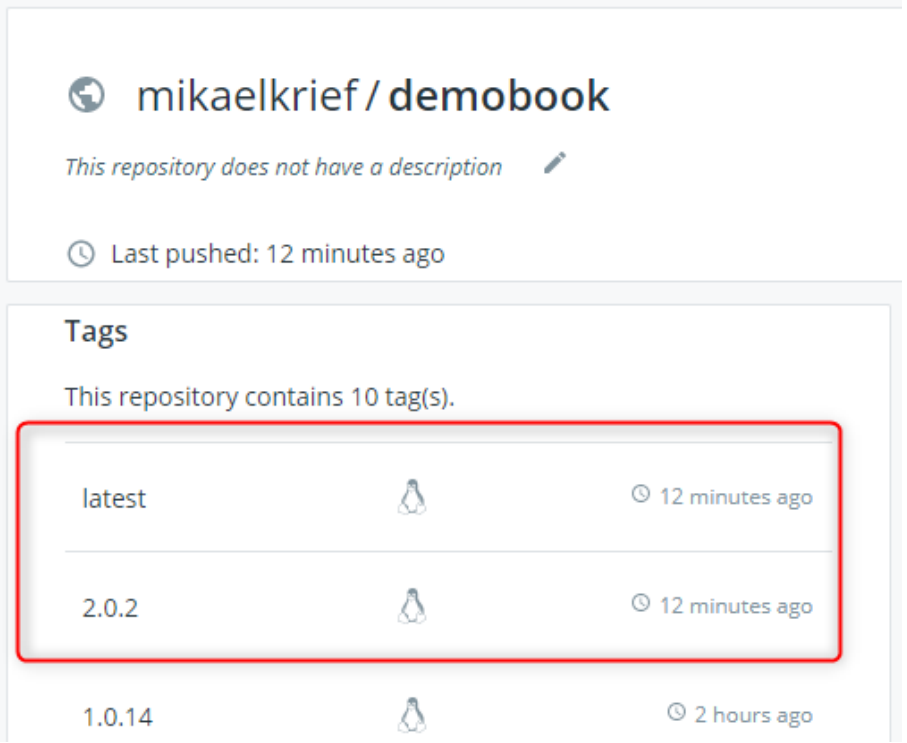
# The build and push of the image in the Docker Hub

- If the builds were successfully executed, notice the following:
- Build artifacts that contain the YAML specification for Kubernetes files:



# Creating a CI/CD pipeline for Kubernetes with Azure Pipelines

- In the Docker Hub, a new tag on the image that corresponds to the build number, as well as the latest tag, as shown in the following screenshot:.



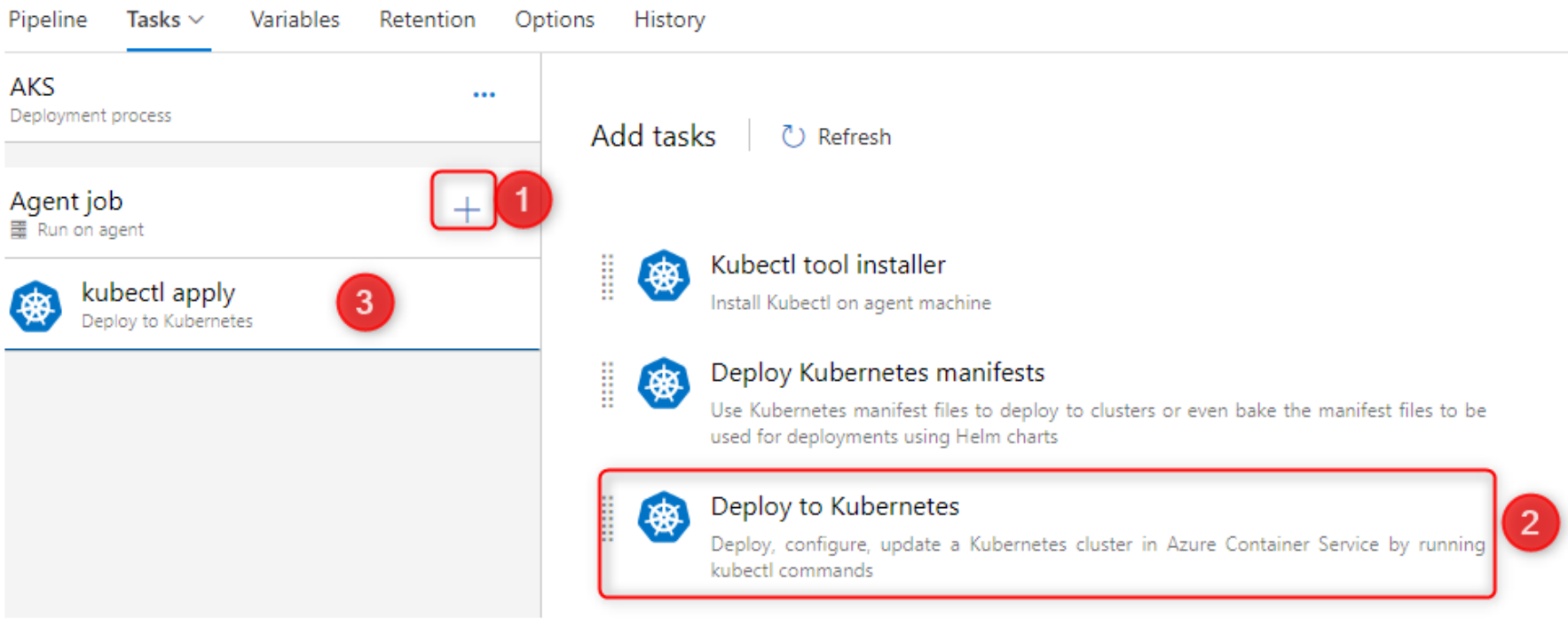
# Automatic deployment of the application in Kubernetes

- Create a new definition of release that automatically deploys our application in the AKS cluster that we created in the previous Using AKS section.
- For this deployment, in Azure Pipelines, create a new release by performing the following steps:

1. Regarding the choice of template for the release, select the Empty template.

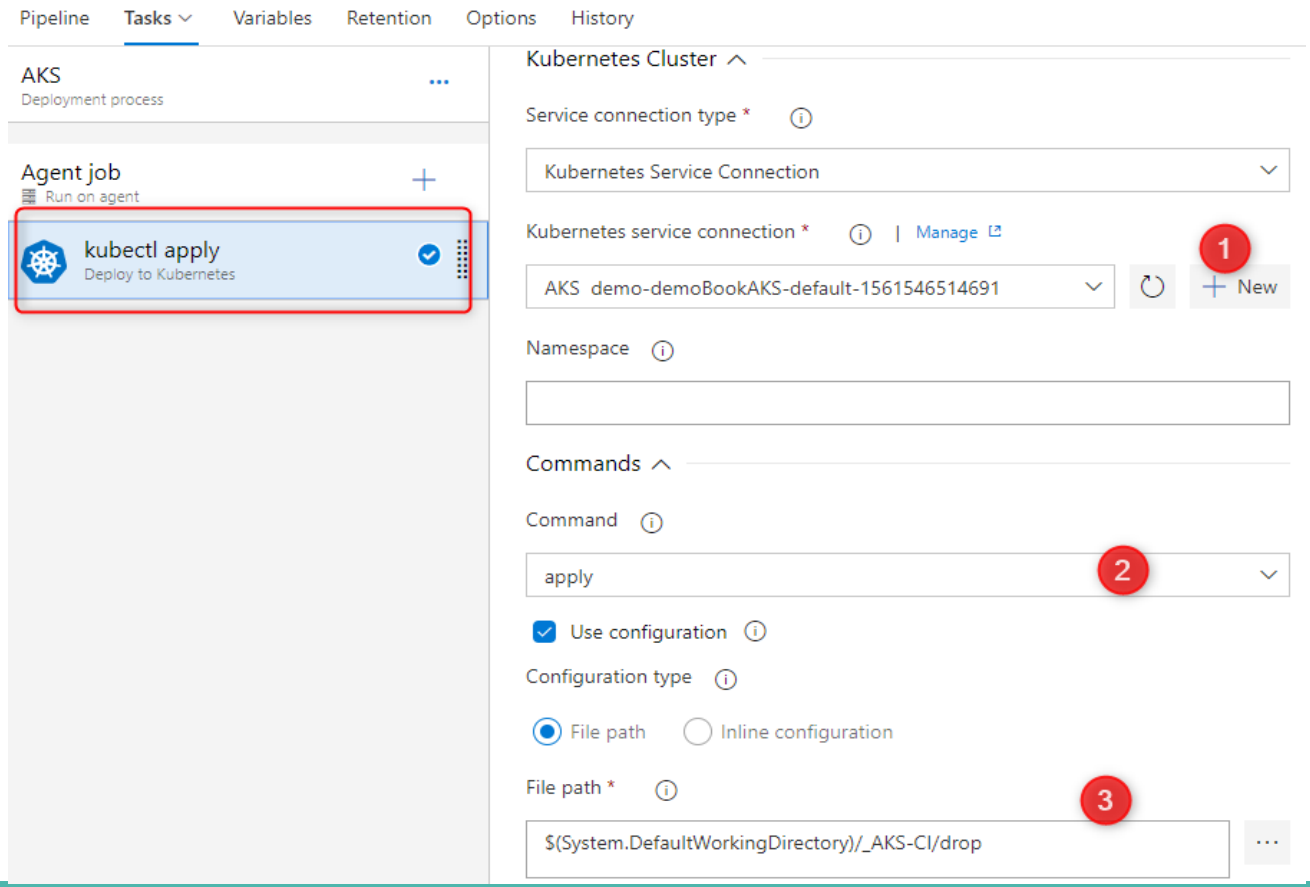
2. Create a stage called AKS, and inside add a task that allows the kubectl commands (this task is present by default in the Azure DevOps tasks catalog):

# Automatic deployment of the application in Kubernetes



# Automatic deployment of the application in Kubernetes

3. Add the Deploy to Kubernetes task to the Azure Pipelines tasks catalog with the following configuration.



# Automatic deployment of the application in Kubernetes

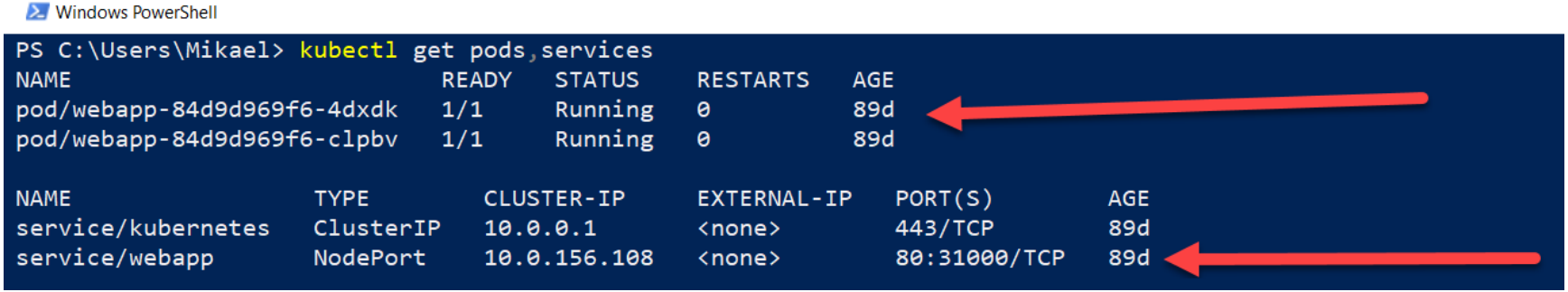
- The settings for the Deploy to Kubernetes task are as follows:
    - Choose the endpoint of the Kubernetes cluster—the New button allows us to add a new endpoint configuration of a cluster.
    - Then, choose the apply command to be executed by kubectl—here, we will execute an application.
    - Finally, choose the directory, coming from the artifacts, which contains the YAML specification files.

4. We save the release definition by clicking on the Save button.

5. Finally, we click on the Create a new release button, which triggers a deployment in our AKS cluster.

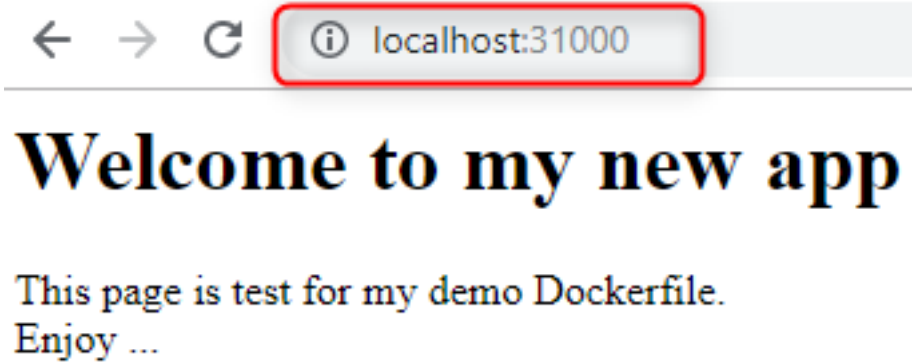
# Automatic deployment of the application in Kubernetes

- At the end of the release execution, it is possible to check that the application has been deployed by executing the command in a Terminal as follows:
  - **kubectl get pods,services**
- This command displays the list of pods and services that are present in our AKS Kubernetes cluster, and the result of this command is shown in the following screenshot:



# Automatic deployment of the application in Kubernetes

- We can see our two deployed web applications pods and the NodePort service that exposes our applications outside the cluster.
- Then, we open a web browser with the http://localhost:31000 URL, and our application is displayed correctly:



# Automatic deployment of the application in Kubernetes

- We have created a complete CI/CD pipeline that deploys an application in a Kubernetes cluster.
- If our application (HTML file) is modified, the build will create and push a new version of the image (in the latest tag), and then the release will apply the deployment on the Kubernetes cluster.
- Thus Created an end-to-end DevOps CI/CD pipeline in order to deploy an application in a Kubernetes cluster (AKS) with Azure Pipelines.

# Thank You