CLI

Contents

[Using --help Flag: 2](#_Toc187948912)

[1. Creating a Docker Container 2](#_Toc187948913)

[Steps to Create a Docker Container (Long Way): 2](#_Toc187948914)

[Create Docker Container: The Short Way 4](#_Toc187948915)

[2. Creating Docker Containers from Dockerfiles 5](#_Toc187948916)

[3. Image Layering in Docker 8](#_Toc187948917)

[How Image Layers Work 8](#_Toc187948918)

[Caching in Docker 9](#_Toc187948919)

[Tips for Leveraging Caching 10](#_Toc187948920)

[4. Interacting with Running Containers 11](#_Toc187948921)

[5. Removing Containers 13](#_Toc187948922)

[Removing images 14](#_Toc187948923)

[6. Docker Hub 15](#_Toc187948924)

[Introduction 15](#_Toc187948925)

[Pushing Images to the Docker Registry 15](#_Toc187948926)

[Checking Your Images in Docker Hub 16](#_Toc187948927)

[7. Container Lifecycle Commands Summary**:** 17](#_Toc187948928)

## Using --help Flag:

* Running docker --help displays a list of top-level commands and general usage information.
* Example: Running docker network --help provides details about the docker network command and its sub-commands.
  + Sub-commands: connect, create, disconnect.
* To explore a sub-command, use the --help flag again:
  + Example: docker network create --help.
  + This shows:
    - **Usage:** A one-line description of the command.
    - **Options:** All supported flags for the command.

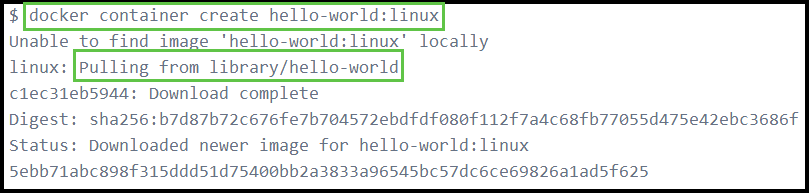
# Creating a Docker Container

### Overview of Docker Containers:

* Containers are created from container images.
* Images are pre-packaged file systems containing the app, its environment, and a start instruction (entry point).

## Steps to Create a Docker Container (Long Way):

* Use docker container create to create a container from an image.
  + If the image doesn’t exist locally, Docker **pulls** it from Docker Hub by default.
  + Example: docker container create hello-world creates a container from the hello-world image.
  + Adding a tag: Use : followed by the tag (e.g., hello-world:linux).
* Verify that the image was pulled successfully (look for pull complete messages).
* Docker assigns a unique ID to each container.



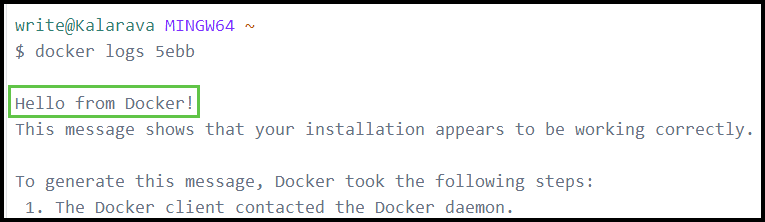
### Starting the Container:

* docker container create does not start the container.
* Run docker ps to list running containers.
* To see all containers (including stopped ones), use docker ps --all.
* Start the container with docker container start <container-ID>.
* Example: Copy the container ID (or use the first few characters) and run docker container start 5ebb.



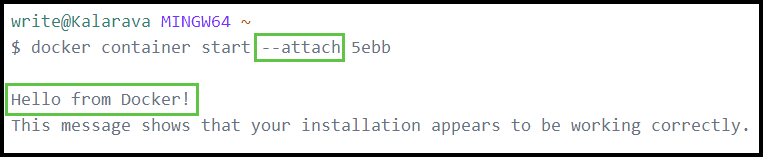
### Viewing Logs:

* Check logs with docker logs <container-ID>.
  + Example: docker logs 5ebb displays the friendly message from the hello-world container.
* Use logs for troubleshooting containers that fail.



### Attaching to the Container’s Output:

* Use docker container start --attach <container-ID> to start the container and immediately view its output.
* For long-running containers, use docker container attach after starting the container.



### Key Points:

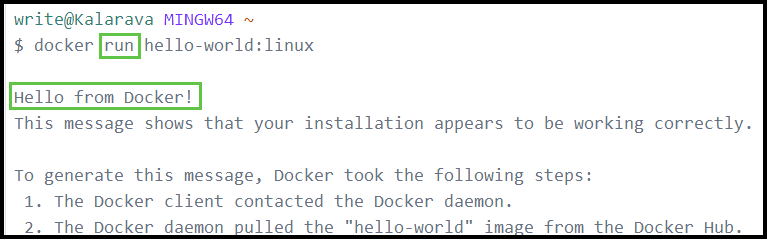
* Containers are not deleted automatically after creation or execution.
* The same container can be started multiple times without recreating it.

## Create Docker Container: The Short Way

### **Run a container using the docker run command:**

docker run hello-world:linux

* This command:
  + Creates a container from the hello-world:linux image.
  + Starts the container.
  + Attaches to the container to display its output.



* So,
* docker run =

# Creating Docker Containers from Dockerfiles

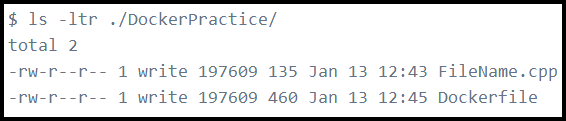
### Introduction

* So far, we have been creating containers from pre-existing Docker images available on Docker Hub.
* To create a container for our own application, we need to build a custom Docker image.
* Docker provides tools to:
  + Build custom images using Dockerfiles.
  + Start containers from those images.
  + Optionally, push the images to Docker Hub to share with others.

### Building a Custom Docker Image

#### Files Required:

* **Dockerfile**: A file containing instructions to build the Docker image.
* **Example App**: A C++ program which prints ‘Hello, World!’ on the screen.



### Dockerfile Syntax Overview

* A Dockerfile is a plain text file with a specific syntax and set of instructions to define how to build an image.

#### Key Dockerfile Instructions:

* FROM:
  + Specifies the base image for the custom image.
  + The base image can be:
    - A local image.
    - A remote image from Docker Hub (default behavior if not local).
* LABEL:
  + Adds metadata to the image (e.g., maintainer information).
* COPY:
  + Copies files from the host machine (build context) to the container image.
  + The "context" is typically the directory passed to the docker build command (default: current working directory).
* RUN:
  + Executes commands during the image build process.
  + Used to install additional software or configure the environment for the application.
  + Example: Installing g++ for the program to build and to run.
* ENTRYPOINT:
  + Defines the default command or script executed by the container when it starts.
  + Alternative: CMD, which also defines a default command, but with different behavior.

#### Example Explanation:

* In the provided Dockerfile:
  + The FROM instruction specifies the base image.
  + The RUN instructions g++ to compile C++ program.
* The CMD instruction specifies the hello object file as the default executable when the container runs.
* Build the custom image using the docker build command.

docker build -t my-first-image ./DockerPractice

#### Command Breakdown

docker build:

* This is the base command used to build a Docker image from a Dockerfile.

-t my-first-image:

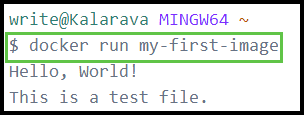
* The -t flag stands for "tag". It assigns a name (or tag) to the resulting image.
* In this case, the image will be named my-first-image.

**./DockerPractice**:

* This is the path to the directory containing the Dockerfile.
* The . (dot) at the beginning signifies the current directory. If your Dockerfile is located in a subdirectory named DockerPractice, this tells Docker to use that directory as the build context.

### Run the container

* Run the created image file using docker run command.



# Image Layering in Docker

* Docker images are built in a **layered file system**. Each layer represents an intermediate stage of the image build process, and together, these layers create the final image.

## **How Image Layers Work**

1. **Base Layer**:
   * The first layer is typically a base image (e.g., ubuntu, alpine) that provides the foundational operating system or runtime.
2. **Intermediate Layers**:
   * Each subsequent command in the Dockerfile creates a new layer. For example:
   * RUN apt-get update creates one layer.
   * RUN apt-get install -y curl creates another.
3. **Read-Only Layers**:
   * Layers in a Docker image is immutable (read-only). When you run a container, Docker adds a **read-write layer** on top of the image layers, allowing temporary changes during runtime.
4. **Shared Layers**:
   * If multiple images share the same base or intermediate layers, Docker reuses those layers instead of duplicating them, reducing disk space usage.

### **Benefits of Layering**

* **Modularity**: Each layer can be independently built, reused, or modified.
* **Caching**: Layers that don’t change are cached, making subsequent builds faster.
* **Efficiency**: Shared layers across images reduce storage requirements.

## **Caching in Docker**

* Docker uses **layer caching** to optimize image builds by reusing existing layers when possible. This feature speeds up the build process and reduces resource usage.

### **How Caching Works**

1. **Dockerfile Instructions**:

* Docker processes the Dockerfile sequentially, from top to bottom.
* For each instruction, it checks if a cached layer can be reused. If a match is found, the cached layer is used.

1. **Matching Criteria**:

* The instruction and its context (e.g., input files) must be identical to a previously built layer for it to be reused.
* If an instruction changes, Docker invalidates that layer and all subsequent layers, forcing them to rebuild.

### **Example of Caching**

* **Dockerfile**:

FROM ubuntu:20.04

RUN apt-get update

RUN apt-get install -y curl

COPY . /app

CMD ["bash"]

* **Build Process**:
  + FROMubuntu:20.04: Cached if the base image is unchanged.
  + RUNapt-get update: Cached unless the command or image state changes.
  + RUNapt-get install -y curl: Cached unless the previous step changes.
  + COPY. /app: Not cached if the content of the . (current directory) changes.
  + CMD["bash"]: Always executed at runtime, not cached.

## **Tips for Leveraging Caching**

* **Order Instructions Strategically**:
  + Place frequently changing instructions (e.g., COPY, ADD) near the bottom of the Dockerfile to avoid invalidating earlier layers.
* **Minimize Changes**:
  + Avoid unnecessary changes to files in the build context (.), as it invalidates caching for COPY or ADD.
* **Use Multistage Builds**:
  + For complex builds, split the Dockerfile into multiple stages to optimize caching and reduce image size.

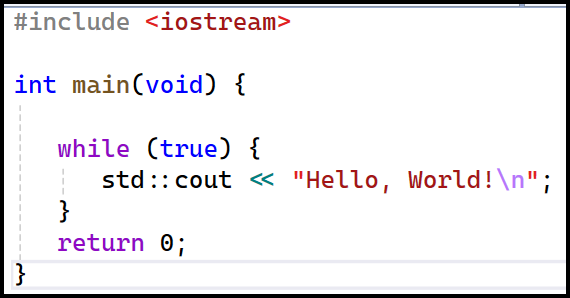
# Interacting with Running Containers

* The docker run <image-name> starts the container and attaches to the terminal by default.

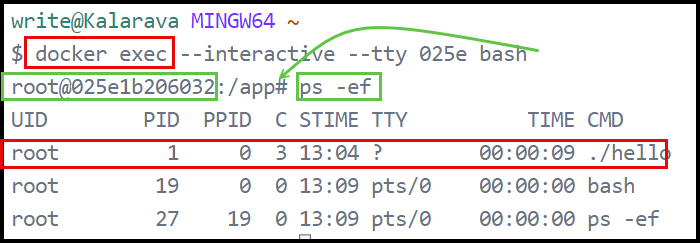
### Handling attached containers:

* If the **terminal hangs**:
  + Open a new terminal.
  + Use docker ps to find the container ID.
  + Stop the container with docker kill <container-id>.

### Running commands inside a running container:

* docker exec allows additional commands to be run:
  + Example: docker exec <container-id> date (runs date command).
  + Example: docker exec --interactive --tty <container-id> bash (starts an interactive bash shell inside the container).
* Example:
  + We have a C++ program the prints ‘Hello, World!’ on the terminal infinitely…
  + If we run the container, the program continuously prints the string on the screen.
  + To open an interactive terminal, use…

docker exec --interactive --tty 025e bash

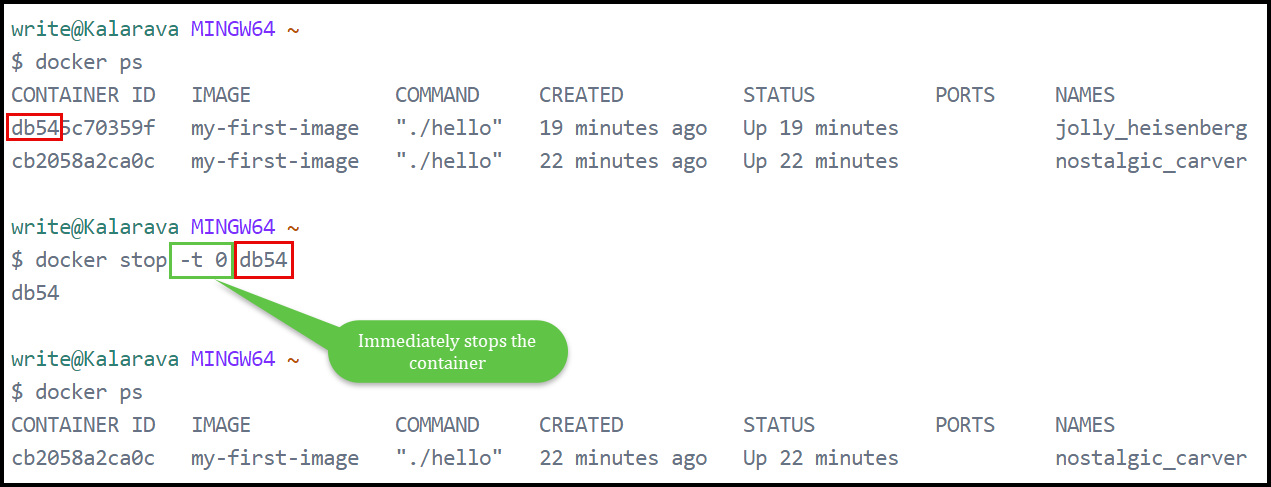


### Stop the Container

* To stop this running container,
  + Get the container id using docker ps -all command…
  + Use the docker stop <container-id> command to stop the container.



* The docker stop <container-id> command takes almost 10-15 seconds to stop the container. If we want to stop it immediately, we can give -t option.



# Removing Containers

### Remove a Stopped Container:

* Use docker rm <container\_id> to delete a stopped container.
* docker rm will not remove running containers; they must be stopped first.

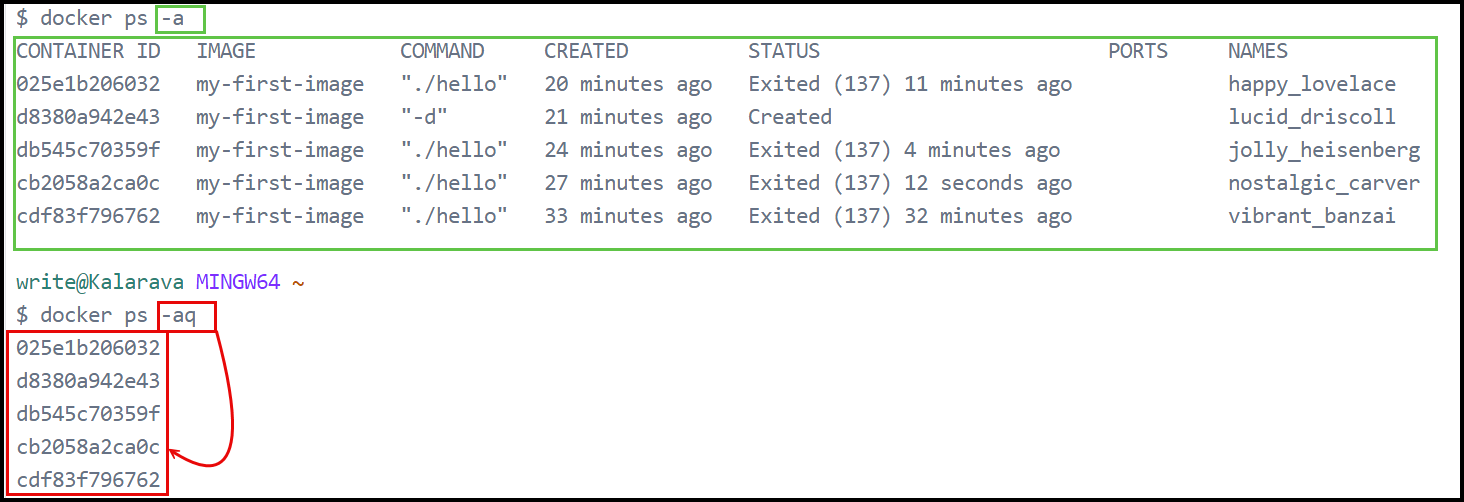
### Force Removing Running Containers:

* Use docker rm -f <container\_id> to forcefully stop and remove a running container.

### **Remove Multiple Containers:**

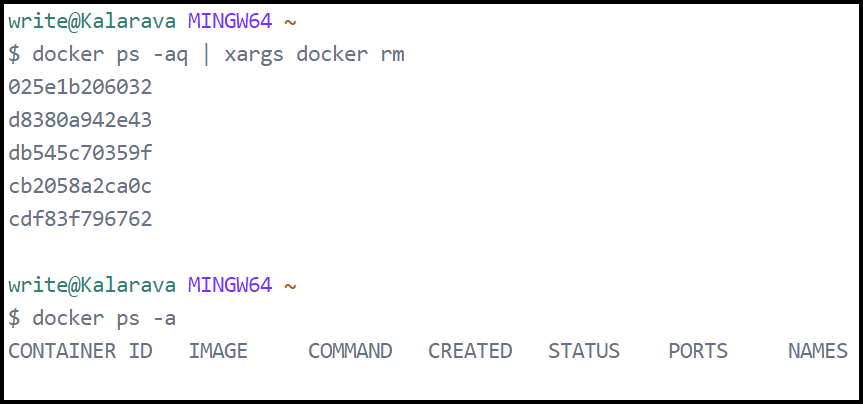
* To list only the container IDs, we can use…

docker ps -aq



* Use xargs to remove all containers in one command...

docker ps -a -q | xargs docker rm



## Removing images

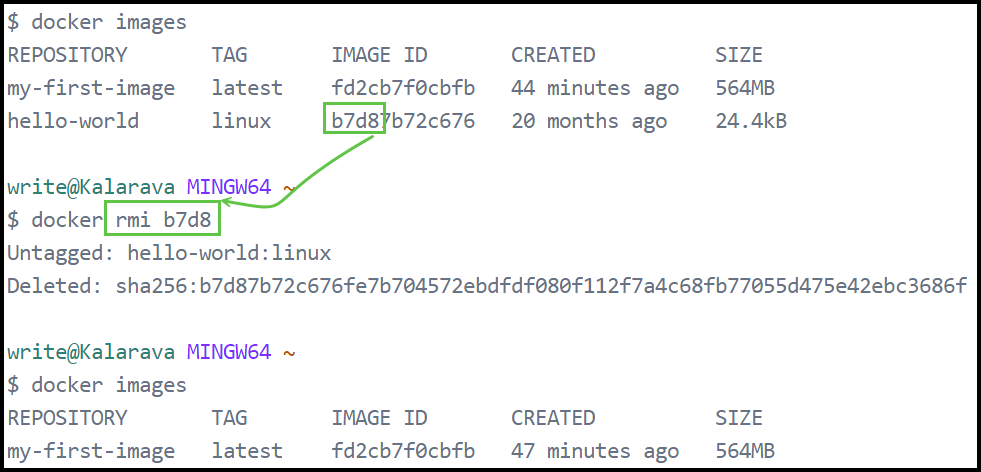
### List Docker Images:

* Use docker images to list all images.



### Remove a Specific Image:

* Use docker rmi <image\_name> to remove an image.



### Handling Dependencies:

* If a container is using an image, stop and remove the container before removing the image.
* Use docker rmi -f <image\_name> to force image removal, but this may cause issues.

# Docker Hub

## Introduction

### Container Image Registry

* A container image registry is a platform for storing and tracking container images.
* Images are tracked using tags, which combine the image name and version (e.g., image\_name:version).
* If no version is specified, the default tag is latest.

### Docker Hub

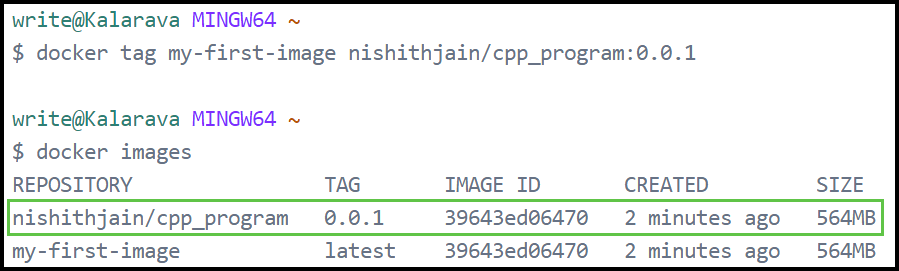
* Docker Hub is the default registry used by the Docker client.
* Features:
  + Publicly accessible for anyone to push and pull images.
  + Automatically used by Docker when an image is referenced without a specific registry.
* Example:

FROM ubuntu:latest

* + Fetches the latest version of the ubuntu image from Docker Hub.

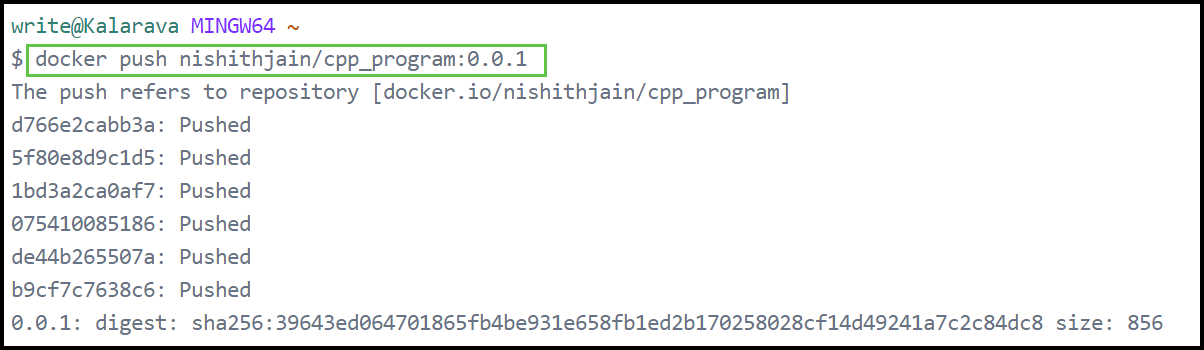
## Pushing Images to the Docker Registry

* Create Docker Hub Account.
* Use the docker login command to log into Docker Hub via the Docker CLI.
* Renaming (Tagging) an Image for the Docker Registry.
  + docker tag <local\_image> <username>/<repository>:<tag>
    - <local\_image> is the name of your existing Docker image.
    - <username>/<repository> is the desired name on Docker Hub.
    - <tag> is an optional version identifier (e.g., latest, 0.0.1).
  + Example:



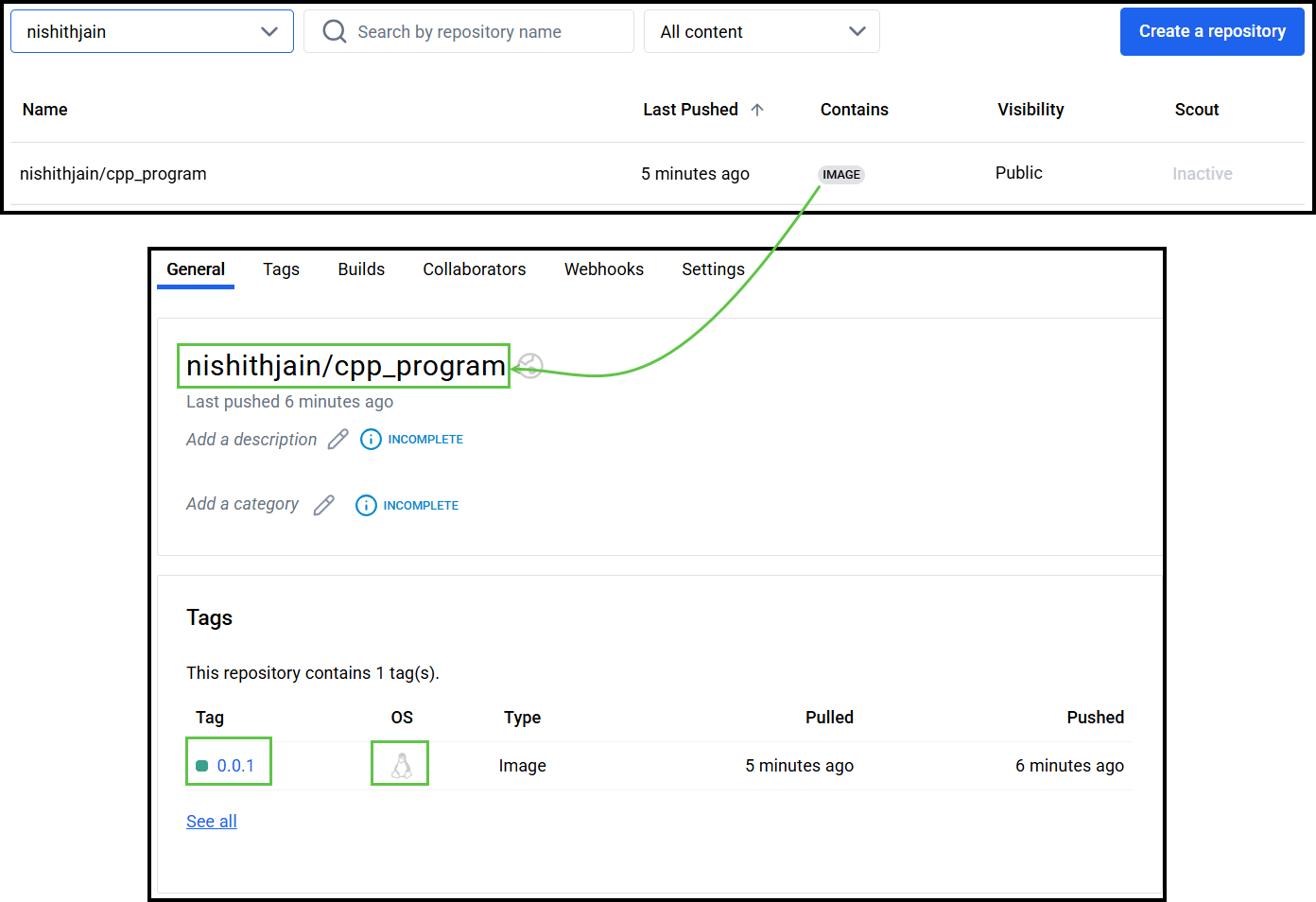
* Push the tagged image to Docker Hub using the docker push command:

docker push <username>/<repository>:<tag>



## **Checking Your Images in Docker Hub**

* Log into Docker Hub via the browser.
* Navigate to the **Images** section.
* Locate the image you just pushed.
  + View details like tags, OS compatibility, and last pushed timestamp.



# **Container Lifecycle Commands Summary:**

|  |  |  |
| --- | --- | --- |
| **State** | **Command** | **Description** |
| **Created** | docker create <image> | Initializes a container without starting it. |
| **Running** | docker run <image> or docker start | Starts the container. |
| **Paused** | docker pause <container\_id> | Freezes all processes in the container. |
| **Unpaused** | docker unpause <container\_id> | Resumes all processes in the container. |
| **Stopped** | docker stop <container\_id> | Stops the container but retains it. |
| **Exited** | Automatically when processes end or stopped | Container process terminates. |
| **Removed** | docker rm <container\_id> | Deletes the container permanently. |