Docker Essentials: A Beginner's Guide to Modern Application Deployment

Shohruh MIRYUSUPOPV

February 20, 2024

Abstract

This tutorial offers a comprehensive introduction to Docker, a platform that has revolutionized software deployment by encapsulating applications into lightweight, portable containers. It demystifies the Docker ecosystem, explaining its core components—the Docker Client, Docker Daemon, and Docker Registry—and their interplay. Participants will learn how to set up Docker, create and manage containers, and understand the stark efficiencies Docker offers over traditional virtual machines. Aimed at beginners, this guide ensures that even those with no prior experience with Docker will gain the necessary skills to utilize container technology effectively, marking their first step into the world of modern, container-based application deployment.

Contents

1 Introduction **3**
- 1.1 Understanding Docker: A Tour to History . . . 3
- 1.2 Definitions and outline of tutorial . . . 3

2 Docker Pipeline and Architecture **5**
- 2.1 Main Pipeline . . . 5
- 2.2 Comparison to Virtual Machines . . . 6
- 2.3 Docker's Architecture . . . 7

3 Building the Docker Image **7**

4 Working with Storage in Docker **8**
- 4.1 Storage types . . . 8
- 4.2 Create container with mounted volume . . . 9
- 4.3 Creating a Container with a Bind Mount . . . 10
- 4.4 Volumes vs. Bind Mounts: Pros and Cons . . . 11
  - 4.4.1 Advantages of Volumes . . . 11
  - 4.4.2 Advantages of Bind Mounts . . . 12
  - 4.4.3 Choosing Between Volumes and Bind Mounts in Windows 12

5 Running and Managing the Container **12**
5.1 Running the Container 12
5.2 Managing Containers and Images 13

6 Summary **14**

A Dockerfile manifest **14**

B Example of `requirements.txt` File **16**

C Pulling Windows Images **17**

1 Introduction

1.1 Understanding Docker: A Tour to History

To grasp the fundamental concept of Docker, let's take a brief journey back in time. Imagine the era when goods were transported across the globe in barrels, crates, and various other containers. Each of these containers had its own unique size and shape. This diversity presented a significant challenge: unloading ships was a cumbersome and time-consuming task, which, in turn, increased the cost of shipping.

This scenario changed dramatically with the introduction of standardized shipping containers. These containers, uniform in size and shape, could be easily loaded, unloaded, and stacked on ships, trains, and trucks. This standardization not only streamlined the entire shipping process but also significantly reduced costs and improved efficiency.

Docker does something very similar in the world of software development and deployment. Just as standardized containers revolutionized the shipping industry by making it easier to transport goods globally, Docker containers revolutionize software development by making it easier to create, deploy, and run applications.

With Docker, applications are packaged along with their dependencies into containers, which are standardized units for software development. These containers ensure that applications run smoothly and consistently across any environment, from a developer's personal laptop to a high-capacity cloud server. This eliminates the common issue of encountering bugs or inconsistencies when moving software from one computing environment to another, a problem often summarized as "it works on my machine."

1.2 Definitions and outline of tutorial

This tutorial provides a brief introduction to Docker containerization with a focus on Windows containers. Docker is a platform that allows you to package your application and its dependencies into a container, which can then be run on any system that supports Docker. This ensures that your application works uniformly and consistently across any environment. First we give definitions used in the context of Docker and containerization [2].

Docker Image: A Docker image is a lightweight, standalone, executable package that includes everything needed to run a piece of software, including the code, runtime, libraries, environment variables, and config files. The image file is created from `.Dockerfile`.

`.Dockerfile`: The starting point in the Docker containerization process. It contains a set of instructions to build a Docker image, specifying the base image, software installations, configurations, and the application to run.

Docker Container (Ephemeral): A container is a runtime instance of a Docker image. It runs completely isolated from the host system by default,

only accessing files and ports if configured to do so. It is described as "ephemeral" to underline that it is temporary and designed to perform specific tasks or run specific applications and then be destroyed. This ensures a clean, predictable environment for each execution.

Volume Mounting: This refers to the process of attaching a volume (a designated directory on the host or in the cloud) to a container, allowing for data persistence and sharing between containers.

Container Logs: Logs provide insights into the operations and events happening within a container. They are crucial for debugging and monitoring the behavior of applications running inside containers.

Docker Commands: Throughout the tutorial, we will use various Docker commands to manage images, containers, and volumes. These commands allow you to build, run, stop, and remove containers and images, as well as inspect logs and manage volumes.

Docker volume: A volume is a persistent data storage mechanism that allows data to be kept in a designated area outside the container's filesystem. This means that the data in the volume can survive container restarts and can be shared between multiple containers. Volumes are managed by Docker and are stored on the host filesystem, but they are managed in a way that is more flexible and secure than simply mounting a host directory into a container. Volumes can be used for various purposes, such as storing database files, unique configurations, or any data that should persist or be accessible across container instances.

Docker Daemon It is often referred to as `dockerd`, is a background service that manages the entire Docker lifecycle of containers and images. It listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes.

- When building an image, the Daemon assembles the image based on the instructions in the Dockerfile.
- For pulling an image, the Daemon fetches the specified image from the Docker registry.
- To run a container, the Daemon instantiates an instance of an image, providing an isolated environment for its execution.

Docker Host it is the machine on which the Docker Daemon runs. It's responsible for running containers and managing Docker services.

Docker Client The Docker Client is the user interface to Docker. It provides the means for users to interact with Docker via the command line using commands such as docker build, docker pull, and docker run. When a command is issued from the client:

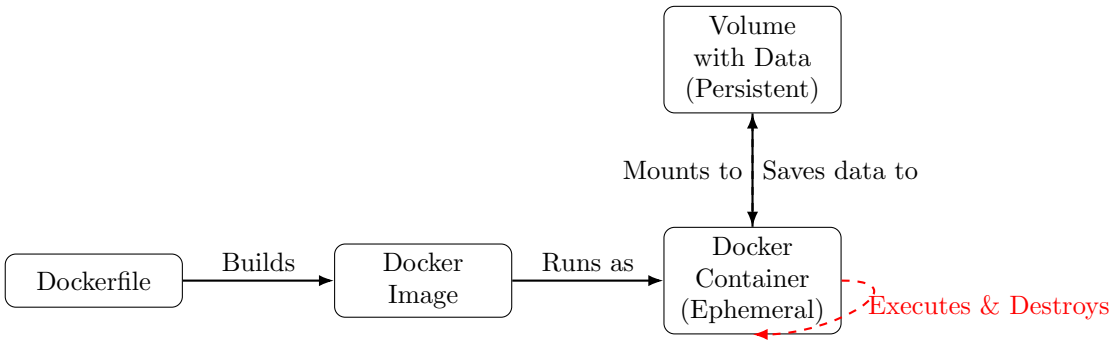

Figure 1: Docker's main pipeline.

- `docker build` sends a request to the Docker Daemon to build an image from a Dockerfile.
- `docker pull` asks the Docker Daemon to pull an image from a registry.
- `docker run` requests the Docker Daemon to run a container from an image.

Registry The Registry is a storage and content delivery system, holding Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. Private registries can also be used.

We will go through the process of building a Docker image, running a container with volume mounting, and inspecting logs and volumes. This tutorial is designed for beginners and aims to provide a practical understanding of Docker containerization on Windows.

2 Docker Pipeline and Architecture

2.1 Main Pipeline

This diagram illustrates the process of creating and running a Docker container, emphasizing the ephemeral nature of containers and the persistent storage provided by Docker volumes.

Builds : The Dockerfile is used to build the Docker Image, encapsulating the application and its environment. Runs as: The Docker Image then serves as the basis for creating an ephemeral Docker Container, which is executed to run the application.

Mounts to : A Volume with Data is mounted to the Docker Container, providing a persistent storage solution that allows the container to read from and write data to a persistent storage area.

Saves data to : Data generated or modified by the application running in the container is saved back to the Volume. This ensures that important data is not lost when the container is destroyed.

Executes & Destroys : Highlighting the lifecycle of the container, this loop indicates that after the container has executed its task, it can be destroyed. The destruction of the container does not affect the persistence of data stored in the volume, which can be reused or accessed by future containers.

2.2 Comparison to Virtual Machines

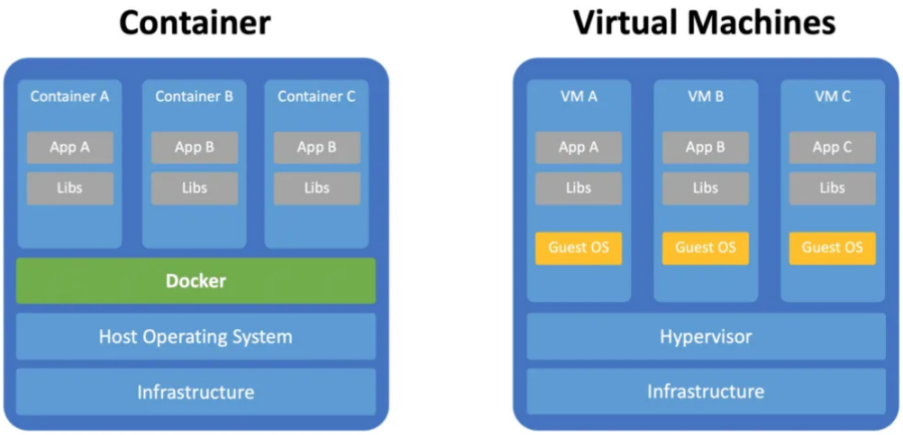

Figure 2: Dockers architecture

On the left of Figure 2, the "Container" section illustrates how containers operate. Multiple containers, labeled as Container A, B, and C, are shown each containing an application (App A, B, and B, respectively) and its libraries (Libs). These containers all share the same host operating system, which is indicated directly below the containers, and in turn, the host OS runs directly on the infrastructure. On the right, the "Virtual Machines" section shows a setup with three separate VMs (VM A, B, and C). Each VM runs its own application and libraries, similar to the containers. However, each VM also includes a guest operating system. These guest OS instances are individual for each VM, which adds additional overhead. Containers share the same OS kernel and isolate the applications at a higher level, which can result in better resource utilization and faster startup times compared to VMs.

2.3 Docker's Architecture

Figure 3 provides a high-level overview of the Docker architecture, illustrating the process from image creation or acquisition to the running of containers, all coordinated by the Docker Daemon in response to commands issued by the Docker Client.

The client is where Docker users execute commands. When a user issues a docker build command, the client sends this instruction to the Docker Daemon, which resides on the Docker host. The Docker Daemon processes this command and builds an image, which it can then push to a registry—a service that hosts and distributes Docker images. Similarly, when the client issues docker pull, it instructs the Docker Daemon to fetch an image from the registry and store it locally on the Docker host. Finally, the docker run command tells the Docker Daemon to create a new container from a specified image. The container is a runtime instance of an image and executes on the Docker host.

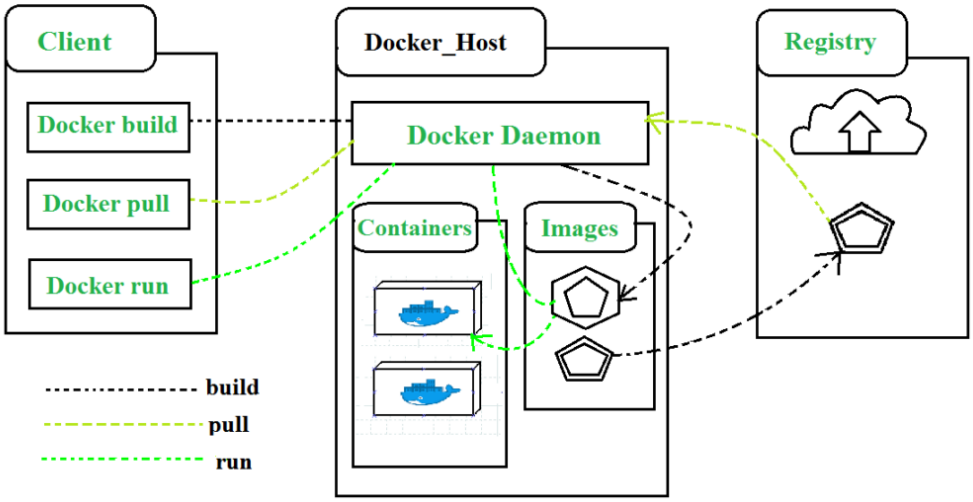

Figure 3: Dockers architecture [5]

3 Building the Docker Image

All the instructions are passed through Docker Client. Once Docker is installed, they can open PowerShell (on Windows), Terminal (on macOS), or any other command-line interface to interact with Docker using the Docker Client. It is through this interface that users will send commands to the Docker Daemon to manage the lifecycle of containers and images.

To build a Docker image, especially one that requires Python libraries, we need to include a requirements.txt file in our build context. This file lists all the Python dependencies that need to be pre-installed in the image. When building the image, we can specify a GitHub token as a build argument to allow access to private repositories if necessary. The image can then be tagged as `my-app`. Including the `requirements.txt` ensures that the resulting Docker image is pre-equipped with the necessary Python libraries for the application to run properly.

```
# Build a Docker image from a Dockerfile
# --build-arg: Passes the build-time variable GITHUB_TOKEN
# with a value of 'sometoken'
# -t: Tags the resulting image as 'my-app'
# .: Indicates that the build context is the current directory,
# containing the Dockerfile
docker login
docker build --build-arg GITHUB_TOKEN=sometoken -t my-app .
```

- Hardcoding the GitHub token, or any sensitive information, directly in your build commands or Dockerfiles is not recommended due to security risks. Anyone who has access to your Dockerfile or command history could potentially extract this sensitive token.
- The docker build command should be executed in the directory where the `.Dockerfile` is located. This directory is referred to as the "build context." By specifying `.` at the end of the command, you are telling Docker that the build context is the current directory. This means all the files and directories in the current directory are available to the Docker daemon to build the image. An example of `.Dockerfile` can be found in Appendix A.

4 Working with Storage in Docker

4.1 Storage types

There are three main types of mounts shown that connect the container to storage resources [6]:

- Bind Mount: This connects a specific directory or file on the host machine's filesystem to a path within the Docker container, allowing the container to read from and write to the host's filesystem directly.
- Volume: This is a Docker-managed portion of the filesystem that is abstracted from the host, providing a persistent and container-agnostic way to store data. Volumes are not tied to the specific container that creates them, so they can be easily shared among multiple containers.
- tmpfs Mount: This storage option mounts a temporary file storage facility directly into the container's memory space. It is backed by the host

machine's memory, but unlike bind mounts and volumes, a tmpfs mount is temporary and is deleted when the container stops.

Figure 4 showcases the different storage options available for a Docker container.

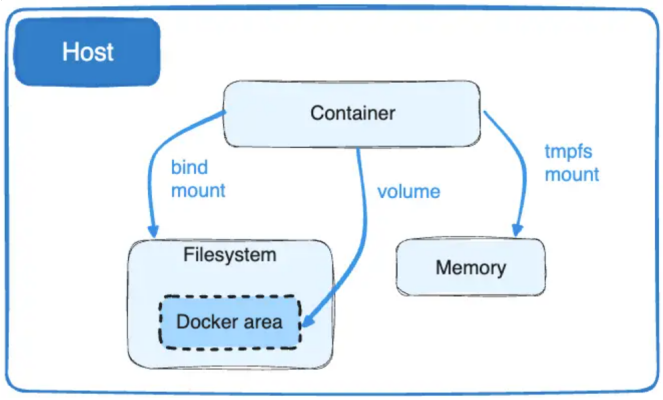

Figure 4: Docker Container Storage Options [1]

The in the following we usename and path variables:

```
$containerName = "mycontainer"
$hostDataPath = "C:\path_to_data\data"
$containerDataPath = "C:\data"
```

- `$containerName`: Specifies the name of the container (iis-container) for easy identification and management.
- `$hostDataPath`: Specifies the path on the host machine (`"C:\path_to_data\data"`) that you want to share with the container. This is the source directory for the bind mount.
- `$containerDataPath`: Specifies the path inside the container (`C:\data`) where the host directory will be mounted. This makes the host directory accessible inside the container at this location.

4.2 Create container with mounted volume

To create a Docker volume, you use the docker volume create command followed by the name of the volume. This command creates a new volume that can be mounted into containers to persist data.

```
docker volume create test_vol
```

To see a list of all volumes on your Docker host, use

```
docker volume rm test_vol
```

We can inspect a volume to get detailed information about it

```
docker volume inspect test_vol
```

To remove a Docker volume, which is useful for cleaning up unused or temporary data, you use the docker volume rm command followed by the name of the volume. It's important to ensure that no containers are currently using the volume you wish to remove.

```
docker volume rm test_vol
```

To create and work with Docker volumes, we use the following commands:

```
# Create a new container with a volume mounted
docker run \
  # Assign the name $containerName to the new container
  --name $containerName \
  # Mount the volume my_app_volume to C:\data in the container
  -v my_app_volume:C:\data \
  -it \ # Interactive mode with a tty
  # Image to use: Windows Server Core LTSC 2019
  mcr.microsoft.com/windows/servercore:ltsc2019 \
  cmd # Start the container with a command prompt session
```

- –name `$containerName` assigns the name `$containerName` to the new container for easier reference. `-v my_app_volume:C:\data` mounts the `my_app_volume` volume to the container's `C:\data` directory, allowing for data persistence between container restarts or sharing data among multiple containers.
- `-it` indicates that the container should be run in interactive mode with a terminal session (tty), allowing you to interact with the container's command prompt directly.
- `mcr.microsoft.com/windows/servercore:ltsc2019` specifies the base image for the container. In this case, it's the Windows Server Core LTSC 2019 image from Microsoft's Container Registry.
- `cmd` is the command to start a command prompt session inside the container, allowing you to execute further commands within the container's environment. Following the docker run command, the docker volume inspect command is used to retrieve detailed information about the specified volume, `my_app_volume`.

4.3 Creating a Container with a Bind Mount

In this part of the tutorial, we'll explore how to run a Docker container with specific configurations, including setting memory limits, using bind mounts for persistent data storage, and specifying the container's base image and command. This is particularly useful for creating a development environment that closely mirrors production settings or for any scenario requiring specific container configurations.

We run the container specifying RAM limits.

```
# Run a Docker container with custom settings
docker run \
  --name $containerName \ # Assigns a custom name to the container
  --mount type=bind,source=$hostDataPath,target=$containerDataPath \ # Creates a
     ↪ bind mount
  --memory 45g \ # Limits the container's memory usage to 45 GB
  -it \ # Interactive mode with a TTY; useful for attaching later if needed
  -d \ # Detached mode; the container runs in the background
  my-app \ # Specifies the Docker image to use for the container
  powershell # Sets PowerShell as the command to execute on container start
```

In this command:

- `--name $containerName` assigns the name 'dev-container' to the new container for easier reference.
- `--mount type=bind,source=$hostDataPath` creates a bind mount from the host's specific directory to a directory inside the container. This allows for direct editing and storage of files on the host that the container can also access.
- `--memory 45g` restricts the container to use a maximum of 45 GB of memory, which is useful for managing system resources.
- `-it` indicates that the container should run in interactive mode with a terminal session (tty), enabling direct interaction with the container's command prompt.
- `powershell` initiates a command prompt session within the container, allowing further command execution within the container's environment.

4.4 Volumes vs. Bind Mounts: Pros and Cons

While Docker volumes offer several benefits for data storage and management in containers, the choice between using volumes and bind mounts often depends on the specific requirements of your environment and application. Below, we explore the advantages and disadvantages of both, with a focus on their performance and adaptability in Windows environments.

4.4.1 Advantages of Volumes

- **Portability**: Volumes facilitate easier backup and migration of data across systems compared to bind mounts.
- **Manageability**: Managed through Docker CLI or API, volumes provide a more straightforward approach to storage management.
- **Compatibility**: Universally compatible with both Linux and Windows containers, offering a flexible storage solution.
- **Security**: Safer to share among multiple containers, with less risk compared to bind mounts.

- **Extensibility**: Support for volume drivers allows expansion of storage functionalities, such as remote storage and encryption.
- **Initialization**: Capable of being pre-populated with container data, which streamlines the setup process.
- **Performance**: Volumes exhibit superior performance on Docker Desktop for Mac and Windows, outperforming bind mounts in these environments.

4.4.2 Advantages of Bind Mounts

- **Direct Access**: Bind mounts provide direct access to the host file system, enabling real-time editing and monitoring of files from the host.
- **Simplicity**: Easy to set up, especially for developers looking to quickly mount local development environments into a container.
- **Performance in Windows**: On Windows systems, bind mounts can sometimes offer better performance for certain workloads, particularly when dealing with large files or directories.

4.4.3 Choosing Between Volumes and Bind Mounts in Windows

For Windows users, the choice between volumes and bind mounts may lean towards bind mounts for several reasons:

- **File System Integration**: Bind mounts in Windows allow for seamless integration with the host's file system, making it easier for developers to work with files stored on Windows drives.
- **Development Workflows**: Given the direct access to the host file system, bind mounts are often preferred for development workflows on Windows, where real-time code changes and testing are common.
- **Performance Considerations**: Although volumes offer performance benefits on Docker Desktop, specific use cases or legacy applications running on Windows containers might benefit more from the direct file system access provided by bind mounts.

5 Running and Managing the Container

5.1 Running the Container

Running a container with a mounted Volume. With our image built, we can now run a container from it. We will create a volume named `test_vol` and mount it to the container's `C:/data` directory. We will also allocate 45 GB of memory to our container and start a PowerShell session. Details on the parameters and variables can be found in section 4.2.

```
# Create a new Docker volume named test_vol
docker volume create test_vol

# Run a container using the test-app image
docker run \
  -v test_vol:C:/data \ # Mount the volume test_vol
  # to C:/data in the container
  --name $containerName \ # Name the container $containerName
  --memory 45g \ # Allocate 45 GB of memory to the container
  -it \ # Interactive mode with a tty
  test-app \ # Image name to use
  powershell # Start a PowerShell session inside the container
```

Running a container with Bind Mount. As in the previous part, we assume the the image is built, we can now run a container from it. We mount a Bind Mount it to the container's `C:/data` directory and follow the same instructions.

```
# Run a Docker container with custom settings
docker run \
  --name $containerName \ # Assigns a custom name to the container
  --mount type=bind,source=$hostDataPath,target=$containerDataPath \ # Creates a
    ↪ bind mount
  --memory 45g \ # Limits the container's memory usage to 45 GB
  -it \ # Interactive mode with a TTY; useful for attaching later if needed
  -d \ # Detached mode; the container runs in the background
  my-app \ # Specifies the Docker image to use for the container
  powershell # Sets PowerShell as the command to execute on container start
```

Details on the parameters and variables can be found in section 4.3.

5.2 Managing Containers and Images

To manage our Docker containers and images, we can use various Docker commands:

```
# List all running containers
docker ps

# Stop a running container
docker stop $containerName

# View logs of a container
docker logs $containerName
docker logs -f $containerName

# Remove a stopped container
docker rm $containerName

# List all Docker images
docker images

# Remove an image
docker rmi my-app1
docker rmi -f c6cc01e60919

# Start a PowerShell session in a new container
docker run -it --entrypoint powershell my-app
```

```

# Execute PowerShell in a running container
docker exec -it $containerName powershell
```

6 Summary

This tutorial covered the basics of Docker containerization on Windows, including image creation, container management, and volume handling.

A Dockerfile manifest

This `.Dockerfile` is designed for creating a Windows-based Docker image that sets up a Python development environment with Git, and is tailored for deploying a Python application.

```
# Use a Windows base image
ARG GITHUB_TOKEN

FROM mcr.microsoft.com/windows/servercore:ltsc2022

# Set the working directory in the container
WORKDIR C:\\myapp

# Use PowerShell for subsequent commands
SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop
    ↪ ';"]

# Install Python and Git
# Note: Replace the URLs with the actual URLs for Python and Git
    ↪ installers
RUN Invoke-WebRequest -Uri 'https://www.python.org/ftp/python
    ↪ /3.9.0/python-3.9.0-amd64.exe' -OutFile 'python_installer.
    ↪ exe' -UseBasicParsing; \
    Start-Process python_installer.exe -ArgumentList '/quiet
    ↪ InstallAllUsers=1 PrependPath=1' -Wait; \
    Remove-Item python_installer.exe -Force; \
    Invoke-WebRequest -Uri 'https://github.com/git-for-windows/
    ↪ git/releases/download/v2.28.0.windows.1/Git-2.28.0-64-bit.
    ↪ exe' -OutFile 'git_installer.exe' -UseBasicParsing; \
    Start-Process git_installer.exe -ArgumentList '/VERYSILENT /
    ↪ NORESTART /NOCANCEL /SP-' -Wait; \
    Remove-Item git_installer.exe -Force
    # Invoke-WebRequest -Uri 'https://curl.se/ca/cacert.pem' -
    ↪ OutFile 'cacert.pem'; \
```

```
    # [System.Environment]::SetEnvironmentVariable('
    ↪ GIT_SSL_CAINFO', 'C:\codpy\cacert.pem', [System.
    ↪ EnvironmentVariableTarget]::Machine); \
    # Remove-Item cacert.pem -Force

# Add Python and Git to PATH (replace with the actual
    ↪ installation paths if different)
RUN $env:Path += ';C:\\Program Files\\Python39;C:\\Program Files
    ↪ \\Python39\\Scripts;C:\\Program Files\\Git\\cmd'; \
    [Environment]::SetEnvironmentVariable('Path', $env:Path, [
    ↪ EnvironmentVariableTarget]::Machine)

# Set the GitHub token as an environment variable
ENV GITHUB_TOKEN=$GITHUB_TOKEN

# Use the token to install the codpy package from the private
    ↪ repository
RUN $Env:GITHUB_TOKEN='YOURGITHUBTOKEN'; \
    git clone https://yourrepo:$Env:GITHUB_TOKEN@github.com/
    ↪ yourrepo/myapp.git; \
    cd myapp; \
    pip install .

# Reset the working directory to C:\app
WORKDIR C:\\app

# Copy the content of the local src directory to the working
    ↪ directory
COPY . C:\\app

# Install any dependencies from requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# If necessary,  copy the specific .pyd file to the site-packages
    ↪  directory
COPY ["yourapp.cp39-win_amd64.pyd", "C:/Program Files/Python39/
    ↪ Lib/site-packages/"]

# Command to run on container start
#CMD ["powershell", "-Command", "while ($true) { Start-Sleep -
    ↪ Seconds 3600 }"]

CMD ["python", "C:\\app\\main.py"]
```

Base Image : The image starts from a Windows Server Core base image (`ltsc2022`). This provides the underlying Windows operating system

environment.

Working Directory : It sets the working directory inside the container to `C:/ myapp`, which is where subsequent commands will operate.

Shell Setting : The shell is set to PowerShell with error preferences, ensuring that subsequent commands are run using PowerShell and that any errors stop the process.

Python and Git Installation : It installs Python and Git by downloading their installers from specified URLs and running them silently. After installation, the installers are removed to keep the image size down.

Path Environment Variable : The `Dockerfile` appends the Python and Git executable paths to the system PATH environment variable, allowing these programs to be run from any location within the container.

GitHub Token : Sets an environment variable `GITHUB_TOKEN` using the build argument `ARG GITHUB_TOKEN`, which is intended to be used for secure access to private repositories.

Clone and Install Python Application : It uses the provided GitHub token to clone a Python application from a private GitHub repository. Then it changes to the application directory and installs it using pip. This process assumes that the application has a setup.py file for installation.

Reset Working Directory : The working directory is reset to `C:/ app`.

Copy Application Files : Copies the application files from the local source directory to the container's working directory.

Install Dependencies : Installs any dependencies specified in requirements.txt using pip.

Copy `.pyd` File : If necessary, copies a specific `.pyd` file (a Python extension module) to the Python site-packages directory, which is where Python looks for installed packages.

Set Startup Command : Finally, it sets the default command to run the Python application (main.py) located at `C:/ app/ main.py` when the container starts.

B Example of `requirements.txt` File

```
scipy
torch
torchvision>=2.0.1
numpy
```

C Pulling Windows Images

Pulling Windows images from the Docker registry follows the same process as acquiring any Docker image, albeit with a specific focus on images tailored for the Windows ecosystem. It is often beneficial to first identify the desired Windows-specific images available on Docker Hub or another Docker registry. This can be accomplished through a web browser, directing searches towards keywords like "Windows", "Windows Server", or any applications of interest, for instance, "IIS", to pinpoint relevant images.

Once the appropriate Windows image is selected, it can be pulled using the `docker pull` command, appended with the name and tag of the image specifying its version. For example, executing

```
docker pull mcr.microsoft.com/windows/servercore/iis:windowsservercore-ltsc2019
```

retrieves the official Microsoft IIS image based on Windows Server Core 2019. The tag in the image name often reflects the Windows Server version, aiding in the precise selection of the required environment. Post-download, a verification step using `docker images` confirms the image's availability on the user's system, listing all Docker images present, including the newly acquired Windows image.

References

[1] Docker Docs: Volumes, Docker documentation

[2] Docker's glossary. https://docs.docker.com/glossary/

[3] Docker docs: docker build (docker image build)

[4] Use containers to Build, Share and Run your applications

[5] geeksforgeeks: Features of Docker

[6] Docker docs: Manage data in Docker