**Name: Vignesh Baiju  
Roll no: A003**

### SAP ID: 70612300051 Subject: Devops Final Exam Q1.A:

### Ans:

### Q1.B:

### 

### Ans:

### Q1.C:

### 

### Ans:

### Q1.D:

### 

### Ans:

### Q2:

### Ans:

Sure! As a Python software developer, here’s a detailed guide tailored to your workflow using GitHub, Visual Studio Code (VSCode), Git extensions, and Python-specific tools for debugging and fixing bugs in your app's code.

### Step 1: ****Clone the Repository and Set Up Your Development Environment****

Before you can resolve any bugs, you need to clone the repository and set up your local environment with the necessary dependencies.

#### **Detailed Steps:**

1. **Clone the repository using GitHub or Git extension in VSCode**:
   * Open **VSCode**.
   * Press Ctrl+Shift+P to open the Command Palette.
   * Type Git: Clone and press Enter.
   * Paste the repository URL from GitHub (e.g., https://github.com/user/repository.git).
   * Choose the folder where you want to clone the repository.
2. **Set up Python environment**:
   * Open the terminal in VSCode by pressing `Ctrl+``.
   * **Create a virtual environment** for the project:
     + Run: python -m venv venv to create a virtual environment named venv.
   * **Activate the virtual environment**:
     + On Windows, run: .\venv\Scripts\activate.
     + On macOS/Linux, run: source venv/bin/activate.
   * **Install dependencies** (if there’s a requirements.txt file):
     + Run: pip install -r requirements.txt.
3. **Install VSCode extensions for Python**:
   * Ensure that the **Python extension** for VSCode is installed. You can find it in the Extensions Marketplace within VSCode.
   * Install any other relevant extensions like **Pylance** (for IntelliSense), **Python Docstring Generator**, or **flake8** (for linting).

#### **Explanation:**

* Cloning the repository to your local machine allows you to work on the code offline.
* Setting up a **virtual environment** ensures that all dependencies for the project are isolated, avoiding potential version conflicts with other Python projects on your machine.

### Step 2: ****Create a New Branch for Bug Fixing****

It’s a good practice to create a new branch for each bug fix, so your changes don’t affect the main branch until they’re tested and ready.

#### **Detailed Steps:**

1. Open the **Source Control** panel in **VSCode** by pressing Ctrl+Shift+G.
2. Click on the **branch name** (e.g., main or master) at the top of the panel.
3. In the pop-up menu, select **Create New Branch**.
4. Name your branch something descriptive, like bugfix/module-name.
5. Ensure you're now working in the new branch.

#### **Explanation:**

Creating a new branch for each fix allows you to work on the bug without disrupting the main codebase. You can test and review changes independently, ensuring stability.

### Step 3: ****Reproduce the Bug and Identify the Cause****

Before applying any fixes, it’s essential to reproduce the bug and understand its root cause.

#### **Detailed Steps:**

1. **Find the bug report**:
   * Visit the **Issues** tab of the repository on GitHub.
   * Look for the issue related to the bug you need to fix. Read through the description, any error logs, and reproduction steps provided in the comments.
2. **Reproduce the issue**:
   * To run the Python app, check the README or any relevant documentation for instructions on how to start the app. This could involve running python app.py or using a flask or django command depending on your framework.
   * Follow the steps to reproduce the bug. This might involve running a specific function or endpoint that’s causing the error.
3. **Check for error logs**:
   * If the bug is related to an error message or stack trace, open the terminal or **VSCode Output Panel**.
   * Check for **Python exceptions** or any logs that might point to a specific line in the code where the bug occurs.

#### **Explanation:**

Reproducing the bug locally is critical to understanding how it manifests. Error logs, stack traces, and debug outputs will give you clues on where the issue lies in the code.

### Step 4: ****Debug the Code and Apply the Fix****

Now that you’ve identified the bug, it’s time to debug the code and apply the necessary fix.

#### **Detailed Steps:**

1. **Set up debugging in VSCode**:
   * Open the file where the bug occurs.
   * Set breakpoints by clicking on the left margin next to the line numbers in VSCode where you suspect the issue lies.
   * In the terminal, run the app using python -m pdb app.py or use VSCode's integrated debugger by pressing F5 to start the debugging session. This allows you to step through the code, inspect variables, and trace the issue.
2. **Apply the fix**:
   * Once you identify the root cause, make the necessary changes to fix the bug. For example, if it’s a variable initialization issue, change the variable type or value. If it’s a missing import, add the appropriate import statement.
   * Ensure the fix addresses the problem without introducing new issues.
3. **Test the fix**:
   * After implementing the fix, rerun the application to ensure the bug is resolved. For example, if you’re using pytest for testing, run pytest to verify that your change passes the tests.
   * If applicable, write new test cases to cover the bug scenario and ensure it’s properly handled in the future.

#### **Explanation:**

VSCode's **debugger** allows you to inspect your code at runtime, helping you pinpoint the exact location of the bug. The fix must be validated through testing to ensure that it works as expected and doesn’t break other parts of the code.

### Step 5: ****Commit the Fix and Push Changes to GitHub****

Once you’ve confirmed that the bug is fixed, commit your changes and push them to GitHub.

#### **Detailed Steps:**

1. **Stage and commit the changes**:
   * Go to the **Source Control** panel in VSCode.
   * Review the files that have been modified.
   * Stage the changes by clicking the plus icon next to each file or + next to the **Changes** header.
   * Write a meaningful commit message such as "Fix issue with module X where Y caused Z error".
   * Commit the changes.
2. **Push the changes to GitHub**:
   * Open the terminal in VSCode and run git push origin bugfix/module-name to push the changes to your remote GitHub repository.
   * Alternatively, use the **Push** button in VSCode's Source Control panel.
3. **Create a Pull Request (PR)**:
   * Go to **GitHub** and navigate to the **Pull Requests** tab.
   * Click on **New Pull Request**.
   * Choose the base branch (e.g., main) and compare it with your feature branch (bugfix/module-name).
   * Add a description to your PR outlining the changes made and the bug that was fixed. Tag any team members who need to review it.

#### **Explanation:**

* **Committing** records your changes in version control. **Pushing** sends those changes to GitHub.
* Creating a **Pull Request (PR)** allows other developers to review your changes before they’re merged into the main branch, ensuring code quality and collaboration.

### Summary of the Steps:

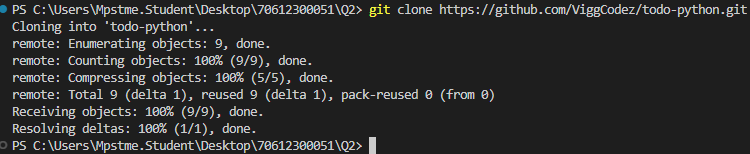
1. **Clone the repository** and set up a Python development environment with a virtual environment.
2. **Create a new branch** for your bug fix.
3. **Reproduce the bug** and identify its cause by reading through the issue description and analyzing error logs.
4. **Debug the code**, apply the fix, and test the changes to ensure they work.
5. **Commit and push** the changes to GitHub, and create a Pull Request for review.

These steps ensure that you follow a structured approach to debugging and fixing bugs in Python while using GitHub and VSCode for efficient code management and collaboration.

Commands explanation:  
Certainly! Below is an explanation of each command used in the process of debugging and fixing bugs in Python, with a focus on Git, VSCode, and Python-specific tools.

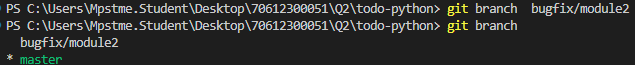
### ****Git Commands**** (Version Control)

#### **1. git clone <repository-url>**

* **Purpose**: Clones a remote repository to your local machine.
* **Explanation**: This command downloads a complete copy of the repository, including all the branches and history, from GitHub or any other Git-hosted source.
* **Example**:
* git clone https://github.com/ViggCodez/todo-python.git  
    
  

#### **2. git branch <branch-name>**

* **Purpose**: Creates a new branch.
* **Explanation**: This command is used to create a new branch in the local Git repository. Branches allow you to isolate your work from the main branch.
* **Example**:
* git branch bugfix/module2



#### **3. git checkout <branch-name>**

* **Purpose**: Switches to an existing branch.
* **Explanation**: This command is used to switch your working directory to the specified branch. If you want to work on a branch like bugfix/module-name, this command will switch you to that branch.
* **Example**:
* git checkout bugfix/module2

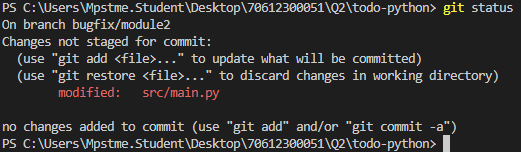


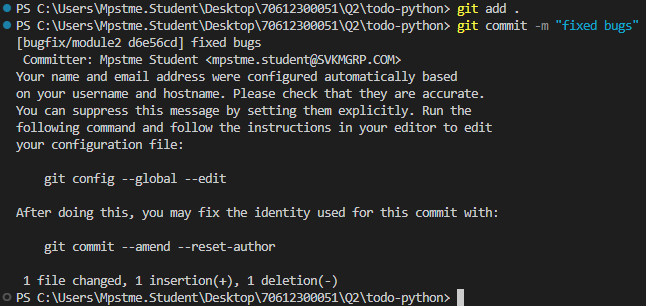
#### **4. git push origin <branch-name>**

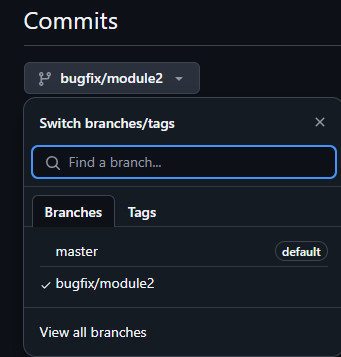
* **Purpose**: Pushes your local branch and its commits to the remote GitHub repository.
* **Explanation**: After committing changes locally, you use git push to upload those changes to GitHub or any other remote Git server. The origin refers to the default remote repository.
* **Example**:
* git push origin bugfix/module-name
* 

#### **5. git commit -m "Commit message"**

* **Purpose**: Commits changes to the local repository.
* **Explanation**: This command stages the current changes (which you've added using git add) and then commits them with a message that describes the changes. The -m flag is used to add the commit message inline.
* **Example**:
* git commit -m "Fix issue with module X causing Y error"





  
 and we can see commits in bugfix branch

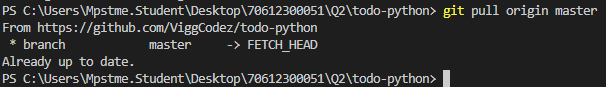
#### **6. git status**

* **Purpose**: Shows the current status of the repository, including which files are staged, modified, or untracked.
* **Explanation**: This command is useful to check which files you have changed, which are staged for commit, and which are untracked.
* **Example**:
* git status



#### **7. git pull**

* **Purpose**: Fetches changes from the remote repository and merges them with your local branch.
* **Explanation**: This command updates your local repository with any changes made by others on the remote repository (e.g., GitHub). This is useful for staying in sync with the project.
* **Example**:
* git pull origin master



### ****Python Commands**** (For Development and Testing)

#### **1. python -m venv <env-name>**

* **Purpose**: Creates a virtual environment.
* **Explanation**: This command creates an isolated Python environment for your project. This is useful for managing dependencies specific to a project and avoiding conflicts with other Python projects on your system.
* **Example**:
* python -m venv venv

#### **2. source venv/bin/activate (Linux/macOS) or .\venv\Scripts\activate (Windows)**

* **Purpose**: Activates the virtual environment.
* **Explanation**: After creating the virtual environment, you activate it to begin using it. Once activated, any Python commands will use the environment’s specific interpreter and installed libraries.
* **Example (Linux/macOS)**:
* source venv/bin/activate
* **Example (Windows)**:
* .\venv\Scripts\activate

#### **3. pip install -r requirements.txt**

* **Purpose**: Installs dependencies listed in a requirements.txt file.
* **Explanation**: This command installs all Python packages and libraries listed in the requirements.txt file, which is often used to specify the necessary packages for a project.
* **Example**:
* pip install -r requirements.txt

#### **4. python app.py**

* **Purpose**: Runs the Python application.
* **Explanation**: This command is used to start a Python application. The app.py file is typically the entry point for the application. For Flask, Django, or other web applications, this command starts the server.
* **Example**:
* python app.py

#### **5. python -m pdb app.py**

* **Purpose**: Runs the Python debugger (pdb) to step through the code.
* **Explanation**: This command is used to invoke Python’s built-in debugger. It allows you to step through the code, inspect variables, and find the root cause of issues. This is a key command for debugging in Python.
* **Example**:
* python -m pdb app.py

#### **6. pytest**

* **Purpose**: Runs the test suite.
* **Explanation**: pytest is a popular testing framework in Python. It runs your unit tests and provides a summary of the results. It's commonly used to ensure that your code changes don’t break existing functionality.
* **Example**:
* pytest

### ****VSCode Specific Commands****

#### **1. Ctrl+Shift+P**

* **Purpose**: Opens the Command Palette in VSCode.
* **Explanation**: The Command Palette allows you to execute commands in VSCode without needing to navigate menus. For example, you can use it to clone a repository, open files, or run tasks.

#### **2. Ctrl+Shift+G**

* **Purpose**: Opens the Source Control panel.
* **Explanation**: This command opens the Git interface within VSCode, allowing you to view the current status of your repository, commit changes, and push them to a remote repository.

#### **3. F5**

* **Purpose**: Starts the debugging session.
* **Explanation**: Pressing F5 in VSCode begins a debugging session, allowing you to run your Python code with breakpoints, inspect variables, and step through the code.

### Q5:

### 

### 

### Ans:

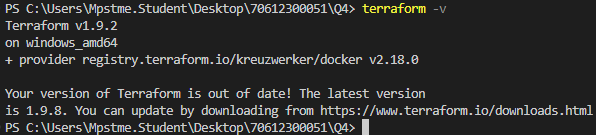
### Q4: Ans: Step 1: Install Prerequisites

Before proceeding, we make sure we have the necessary tools installed:

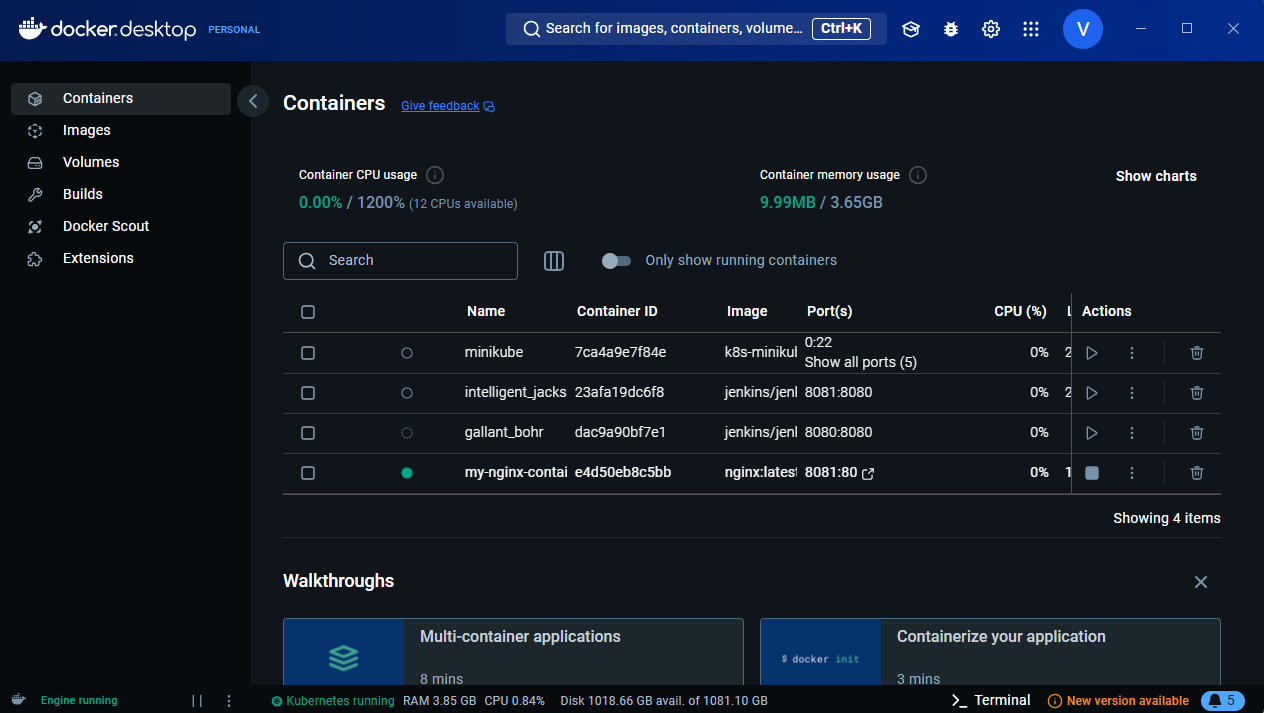
1. **Install Docker from website and check version to verify:**



1. **Install Terraform**:



1. **Ensure Docker is Running**:
   * If Docker is not running, start Docker Desktop and let it run on background.



### Step 2: ****Create a Working Directory for Terraform Configuration****

Create a directory on your local machine where you will store your Terraform configuration files. (in my case it is Q4)

### Step 3: ****Create a Terraform Configuration File****

Inside your working directory, create a file named main.tf which will contain the Terraform configuration for Docker.

***Code:***  
**Main.tf**

terraform {

required\_providers {

docker = {

source = "kreuzwerker/docker"

version = "2.18.0"

}

}

}

provider "docker" {

host = "npipe:////./pipe/docker\_engine" # Use named pipe for Docker on Windows

}

resource "docker\_image" "nginx" {

name = "nginx:latest" # Correct way to specify the image with the tag

}

resource "docker\_container" "nginx" {

name = "my-nginx-container"

image = docker\_image.nginx.name

ports {

internal = 80

external = 8081

}

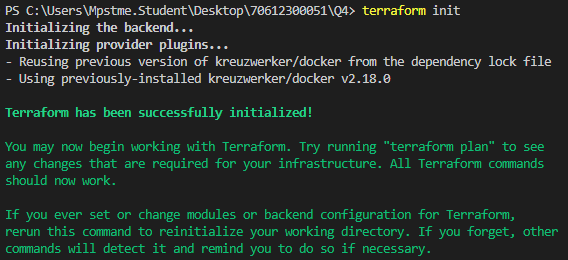
}

#### **Explanation of Configuration:**

* **Docker Provider Configuration**: Specifies how Terraform should interact with Docker on your machine using the named pipe for Windows (npipe:////./pipe/docker\_engine).
* **Docker Image**: Pulls the nginx:latest image from Docker Hub to be used for the container.
* **Docker Container**: Creates a Docker container named my-nginx-container from the Nginx image and maps port 80 in the container to port 8081 on your local machine.
* **Outputs**: Displays the container’s name, IP address, ports, and the image used.

### Step 4: ****Initialize Terraform****

Initialize your Terraform working directory. This will download the necessary provider plugins (like docker).



### Step 5: ****Plan the Terraform Deployment****

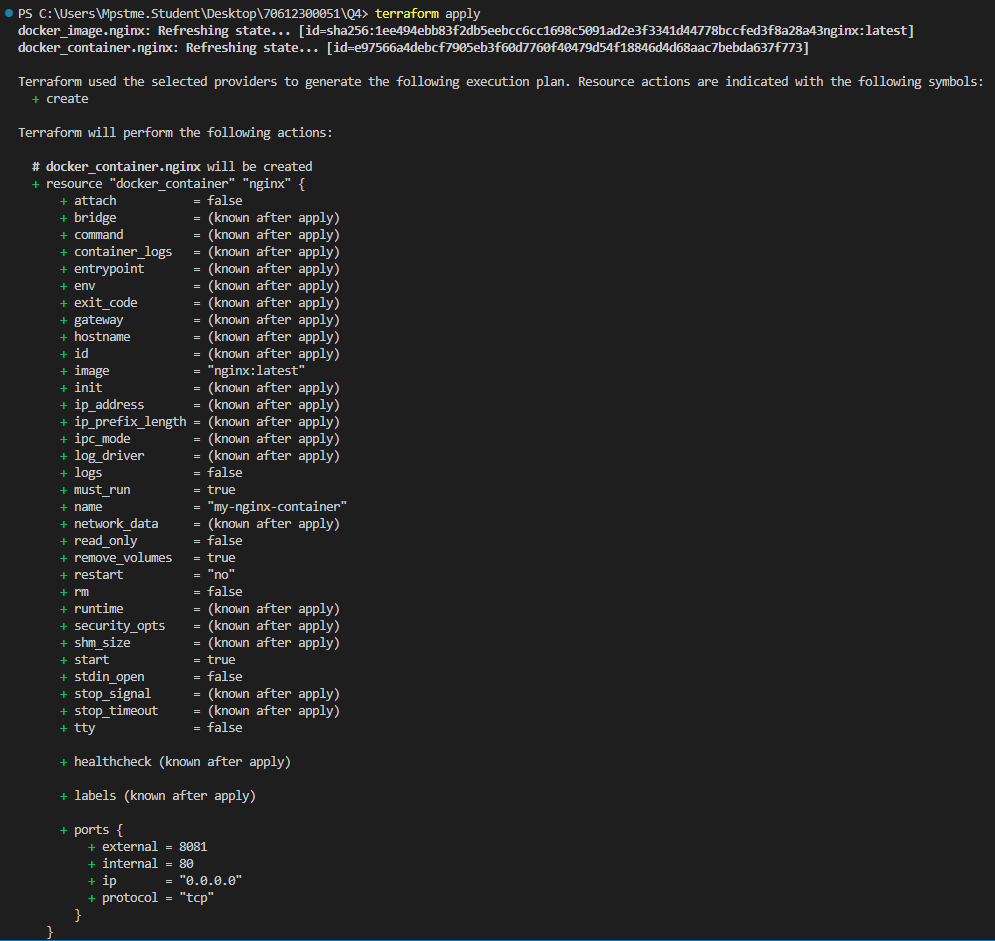
Run the following command to preview the changes that Terraform will make. It will show you what Docker containers and images will be created.

terraform plan

Terraform will show a summary of what resources it plans to create, modify, or delete. If everything looks good, proceed to the next step.

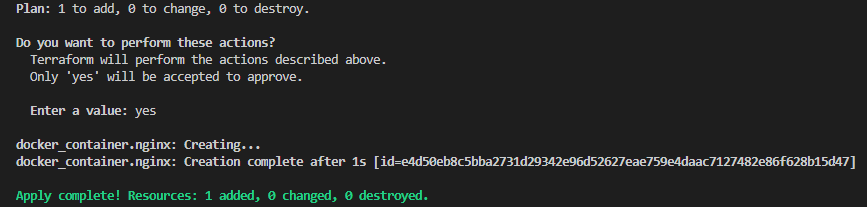
### Step 6: ****Apply the Terraform Configuration****

Now that you've reviewed the plan, apply the configuration to actually create the Docker container and image.



### Step 7: ****Verify the Deployment****

Once terraform apply finishes, Terraform will output the results, which might look something like this:



Terraform will ask you to confirm the action. Type yes to proceed.

Terraform will:

* Pull the nginx:latest image if it isn't already present.
* Create the Docker container named my-nginx-container with port 8081 (because 8080 was not available) mapped to port 80 inside the container.
* Display the outputs you defined (container name, IP address, etc.).

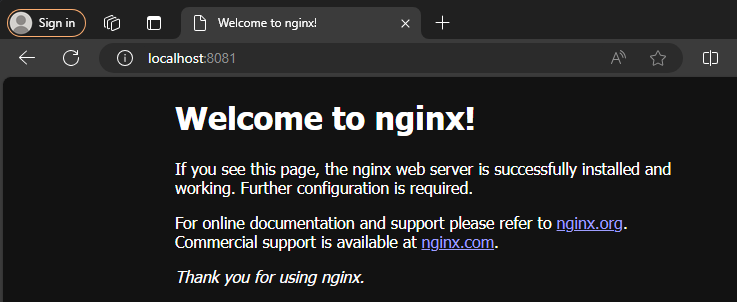
Now, you can:

1. **Check the running Docker containers**:



1. **Access the Nginx container** by opening a browser and navigating to:

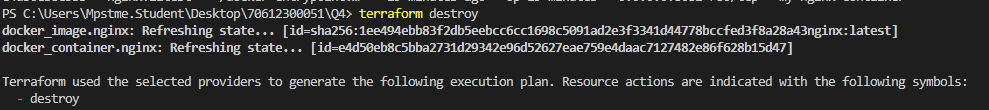
<http://localhost:8081>

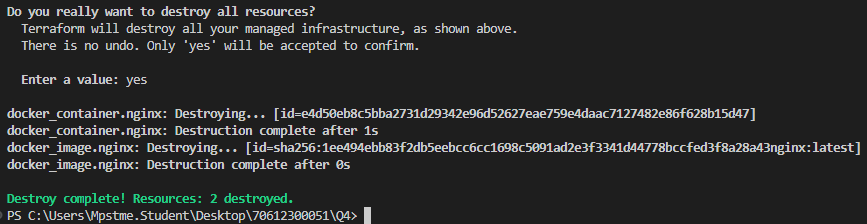


You should see the default Nginx welcome page.

### Step 9: ****Destroy Resources (Optional)****

If you want to clean up and remove all resources created by Terraform, run:





Terraform will ask you to confirm. Type yes to remove the resources.

### Q7:

### 

### ****Step 1: Setting Up Docker on Windows****

### 

### ****Step 2: Prepare Your Application with Docker****

#### **2.1. Create a Dockerfile**

The **Dockerfile** is a script that defines the environment and steps required to build a Docker image for your application. Here's an example of a simple Dockerfile for a Python-based web app using Flask.

# Use the official Python image as the base image

FROM python:3.9-slim

# Set the working directory in the container

WORKDIR /app

# Copy the application files into the container

COPY . /app

# Install the required dependencies

RUN pip install -r requirements.txt

# Expose the port the app will run on

EXPOSE 5000

# Command to run the application

CMD ["python", "app.py"]

1. Place the Dockerfile in the root of your web app project.
2. The requirements.txt file should list all necessary Python dependencies (e.g., Flask, Gunicorn).

#### 2.2. **Build the Docker Image**

To build the Docker image for your app, navigate to the directory containing your Dockerfile and run the following command:

docker build -t my-web-app .

This will create a Docker image called my-web-app. You can verify the image was created using:

docker images

### ****Step 3: Running Docker Containers****

After building the Docker image, you can run it as a container:

docker run -d -p 5000:5000 --name web-app my-web-app

* -d runs the container in detached mode.
* -p 5000:5000 maps port 5000 from the container to port 5000 on your host.
* --name web-app assigns a name to the running container.

To check if the container is running:

docker ps

### ****Step 4: Automate Deployment Using Ansible and Docker****

Now that your application is containerized, you can use **Ansible** to automate the deployment of the Docker container across multiple servers (whether on Windows or Linux). This can be done by leveraging **Docker-related Ansible modules**.

#### 4.1. **Install Ansible and Docker on Windows (WSL)**

1. **Install Ansible** on your WSL environment (as mentioned in previous steps).
2. Install **Docker** on your target machines (Linux or Windows-based systems) using Docker Desktop or Docker Engine, depending on the environment.

#### 4.2. **Ansible Docker Modules**

Ansible provides several Docker-related modules that can help you manage Docker containers, images, and volumes.

* **docker\_image**: Build or pull Docker images.
* **docker\_container**: Create, start, and manage Docker containers.

For example, to deploy a Docker container, you can use Ansible’s docker\_container module.

### ****Step 5: Create the Ansible Playbook for Docker Deployment****

You can now write an **Ansible playbook** to automate the Docker container deployment. Here’s an example playbook (deploy\_docker\_app.yml) that automates the deployment of a Docker container for your web application.

#### Example Playbook:

---

- name: Deploy web app using Docker

hosts: all

become: yes

tasks:

- name: Pull the Docker image

docker\_image:

name: my-web-app

source: build

path: /path/to/your/Dockerfile # Path where Dockerfile is located

force: yes

- name: Stop and remove any existing container

docker\_container:

name: web-app

state: absent

- name: Run the web app container

docker\_container:

name: web-app

image: my-web-app

state: started

restart\_policy: always

published\_ports:

- "5000:5000"

#### Explanation:

1. **docker\_image**: This task pulls the my-web-app image or builds it from a local Dockerfile located at /path/to/your/Dockerfile. You can use source: pull to fetch the image from a Docker registry instead of building it locally.
2. **docker\_container**: This task ensures that any previously running container with the name web-app is stopped and removed. Then it starts a new container with the my-web-app image, mapping the container’s port 5000 to the host’s port 5000.

### ****Step 6: Run the Ansible Playbook****

To run this playbook, you’ll need to specify your inventory file and run the following command:

ansible-playbook -i inventory/hosts.yml deploy\_docker\_app.yml

Where the inventory/hosts.yml file contains the list of your target hosts (either Linux or Windows servers) where Docker is installed.

#### Example inventory file (hosts.yml):

[web\_servers]

linux-server-1.example.com

windows-server-1.example.com

[web\_servers:vars]

ansible\_user=your\_user

ansible\_password=your\_password

ansible\_connection=winrm # for Windows

ansible\_winrm\_transport=ntlm # for Windows

### ****Step 7: Continuous Integration/Continuous Deployment (CI/CD)****

To integrate this deployment process into a CI/CD pipeline (e.g., using **GitLab CI**, **Jenkins**, or **GitHub Actions**), you can:

1. Trigger the CI/CD pipeline after a successful commit or merge.
2. Build the Docker image.
3. Run the Ansible playbook to deploy the updated container to your servers.

Here’s an example of a .gitlab-ci.yml file for GitLab CI:

stages:

- build

- deploy

build:

stage: build

script:

- docker build -t my-web-app .

deploy:

stage: deploy

script:

- ansible-playbook -i inventory/hosts.yml deploy\_docker\_app.yml

### ****Conclusion****

Using **Docker** in combination with **Ansible** provides a highly efficient way to deploy web applications across multiple servers, whether they are Windows or Linux-based. Docker allows you to containerize your application, making it portable and ensuring consistent environments across all deployment targets. Ansible, on the other hand, automates the setup, management, and orchestration of Docker containers, ensuring a smooth and repeatable deployment process. This approach is perfect for environments where you need flexibility, consistency, and ease of scaling.