## Lab 12:  Build a Sample Web App in a Docker

### Case Study

NetDev Solutions, a DevOps-focused consultancy, was facing challenges with inconsistent development environments and manual deployment processes. With a distributed team and increasing project load, the need for standardization and automation became critical. To modernize its workflow, the company launched an internal initiative to train developers in containerization using Docker and automation with bash scripting. The goal was to streamline application setup, reduce errors, and ensure consistent deployments across environments.

### Business Challenge

Developers at NetDev Solutions often ran into issues due to mismatched system dependencies and manual setup steps. Each new environment required custom configurations, slowing down testing and deployment. There was no automated process to build or run application containers, and new team members found it difficult to get up to speed. The lack of standardized, reusable scripts made scaling projects inefficient and error-prone.

### Solution

To address this, NetDev implemented a lab-based approach to teach developers how to containerize a Python web application using Docker and automate the process with bash scripting. The lab included:

1. Create a Simple Bash Script

2. Create a Sample Web App

3. Configure the Web App to Use Website Files

4. Create a Bash Script to Build and Run a Docker Container

5. Build, Run, and Verify the Docker Container

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **// Create a Simple Bash Script**  1. Power on the **DEVASC** virtual machine. Open a **Terminal** window to begin scripting and development tasks.  2. Create an empty bash script file. Change your working directory to **~/labs/devnet-src/sample-app** and add a new file called **user-input.sh**.    3. Use the **nano** command to open the nano text editor.    4. Add the ‘she-bang’ to the top of the script. From here you can enter commands for your bash script. Use the arrow keys to navigate in **nano**. Notice the commands at the bottom for managing the file. The carat symbol **(^)** indicates that you use the **CTRL** or **Command** key on your keyboard. For example, to exit **nano**, type **CTRL+X**.    **Note:** You can use a graphical text editor or open the file with VS Code. However, you should be familiar with command-line text editors like **nano** and **vim**. Search the internet for tutorials to refresh your skill or learn more about them.  5. To add simple bash commands to the script, enter some simple **bash** commands for your script. The following commands will ask the user for a name, set the name to a variable called **userName**, and display a string of text with the user’s name.   |  | | --- | | echo -n "Enter Your Name: "  read userName  echo "Your name is $userName." |   6. Exit **nano** and save your script. Press **CTRL+X**, then **Y**, then **ENTER** to exit nano and save your script.  7. You can run it directly from the command line using the command **bash user-input.sh.**    8. To change the mode of the script to an executable file for all users. Use the **chmod** command. Set the options to **a+x** to make the script executable **(x)** by all users **(a)**. After using **chmod**, notice permissions have been modified for users, groups, and others to include the **x** (executable).    9. You can rename the file to remove the extension so that users do not have to add .sh to the command to execute the script.    10. Now the script can be run from the command line without the **source** command or an extension. To run a bash script without the source command, you must preface the script with **./**.    // **Create a Sample Web App**  1. Before we can launch an application in a Docker container, we first need to have the app. In this part, you will create a very simple Python script that will display the IP address of the client when the client visits the web page.  2. Web application developers using Python typically leverage a framework. A framework is a code library to make it easier for developers to create reliable, scalable and maintainable web applications. **Flask** is a web application framework written in Python. Other frameworks include Tornado and Pyramid.  You will use this framework to create the sample web app. Flask receives requests and then provides a response to the user in the web app. This is useful for dynamic web applications because it allows user interaction and dynamic content. What makes your sample web app dynamic is that it will be displaying the IP address of the client.  **Note:** Understanding Flask functions, methods, and libraries are beyond the scope of this course. It is used in this lab to show how quickly you can get a web application up and running. If you want to learn more, search the internet for more information and tutorials on the Flask framework.  3. Open a terminal window and import flask using **pip install flask**.    4. Open the **sample\_app.py** file located in the **/sample-app** directory. You can do this inside VS Code or you can use a command-line text editor like nano or vim. Add the commands to import methods from flask. Add the commands **from flask import Flask** and **from flask import request** to import the required methods from the flask library.  5. Create an instance of the Flask class and name it **sample**. Next, configure **Flask** so that when a user visits the default page (root directory), it displays a message with the IP address of the client.  6. Notice the **@sample.route(“/”)** Flask statement. Frameworks such as Flask use a routing technique (.route) to refer to an application URL (this not to be confused with network routing). Here the “/” (root directory) is bound to the **main()** function. So, when the user goes to **http://localhost:8080/** (root directory) URL, the output of the return statement will be displayed in the browser.  8. Finally, configure Flask to run the app locally at **http://0.0.0.0:8080**, which is also **http://localhost:8080.** Be sure to use two underscores before and after “main”.  9. Save your script and run it from the command line using command **python3 /home/devasc/sample\_app.py**. You should see the following output which indicates that your **sample-app** server is running. If you do not see the following output or if you receive an error message, check your **sample\_app.py** script carefully.    10. You can verify the server is running. Open the **Chromium** web browser and enter **0.0.0.0:8080** in the URL field. You should get the output: **You are calling me from 127.0.0.1.**    **Note:** If you receive an “HTTP 400 Bad Request” response, check your sample\_app.py script carefully.  11. Return to the terminal window where the server is running and press **CTRL+C** to stop the server.  **// Configure the Web App to use Website Files**  1. Explore the directories that will be used by the web app. The directories templates and static are already in the sample-app directory. Open the **index.html** and **style.css** to view their contents. If you are familiar with HTML and CSS, feel free to customize these directories and files as much as you like. However, be sure you keep the embedded {{**request.remote\_addr**}} Python code in the index.html file as this is the dynamic aspect of the sample web app.   |  | | --- | | devasc@labvm:~/labs/devnet-src/sample-app$ cat templates/index.html  <html>  <head>  <title>Sample app</title>  <link rel="stylesheet" href="/static/style.css" />  </head>  <body>  <h1>You are calling me from {{request.remote\_addr}}</h1>  </body>  </html>  devasc@labvm:~/labs/devnet-src/sample-app$ cat static/style.css  body {background: lightsteelblue;}  devasc@labvm:~/labs/devnet-src/sample-app$ |     2. Now that you have explored the basic website files, you need to update the **sample\_app.py** file so that it renders the **index.html** file instead of just returning data. Generating HTML content using Python code can be cumbersome, especially when using conditional statements or repeating structures. The HTML file can be rendered in Flask automatically using the **render\_template** function. This requires importing the render\_template method from the flask library and editing to the **return** function. Make the highlighted edits to your script.   |  | | --- | | from flask import Flask  from flask import request  from flask import render\_template  sample = Flask(\_\_name\_\_)  @sample.route("/")  def main():  return render\_template("index.html")  if \_\_name\_\_ == "\_\_main\_\_":  sample.run(host="0.0.0.0", port=8080) |   3. Save and run your **sampe-app.py** script using command **python3 sample\_app.py**. You should get output like the following:   |  | | --- | | devasc@labvm:~/labs/devnet-src/sample-app$ python3 sample\_app.py  \* Serving Flask app "sample-app" (lazy loading)  \* Environment: production  WARNING: This is a development server. Do not use it in a production deployment.  Use a production WSGI server instead.  \* Debug mode: off  \* Running on http://0.0.0.0:8080/ |     4. To verify your program is running. Open the Chromium web browser and enter **http://0.0.0.0:8080/** in the URL field. You should get the same output as before. However, your background will be light steel blue and the text will be formatted as H1.    5. Return to the terminal window where the server is running and press CTRL+C to stop the server.  **// Create a Bash Script to Build and Run a Docker Container**  1. In this part, you will create a bash script and add commands to build and run a Docker container.  2. First create temporary directories to store the website files. Open the **sample-app.sh** bash script file in the **~/labs/devnet-src/sample-app** directory. Add the **‘she-bang’** and the commands to create a directory structure with **tempdir** as the parent folder.   |  | | --- | | #!/bin/bash  mkdir tempdir  mkdir tempdir/templates  mkdir tempdir/static |   3. Secondly to Copy the website directories and **sample\_app.py** to the temporary directory. In the sample-app.sh file, add the commands to copy the website directory and script to **tempdir**.   |  | | --- | | cp sample\_app.py tempdir/.  cp -r templates/\* tempdir/templates/.  cp -r static/\* tempdir/static/. |   4. Now to create a **Dockerfile** enter the necessary bash **echo** commands to the **sample-app.sh** file to create a Dockerfile in the **tempdir**. This Dockerfile will be used to build the container. You need Python running in the container, so add the Docker **FROM** command to install Python in the container.  5. Your **sample\_app.py** script needs Flask, so add the Docker **RUN** command to install Flask in the container. Your container will need the website folders and the **py** script to run the app, so add the Docker **COPY** commands to add them to a directory in the Docker container. Here, you will create **/home/myapp** as the parent directory inside the Docker container. Besides copying the sample\_app.py file to the Dockerfile, you will also be copying the index.html file from the templates directory and the style.css file from the static directory.   |  | | --- | | echo "FROM python" >> tempdir/Dockerfile  echo "RUN pip install flask" >> tempdir/Dockerfile  echo "COPY ./static /home/myapp/static/" >> tempdir/Dockerfile  echo "COPY ./templates /home/myapp/templates/" >> tempdir/Dockerfile  echo "COPY sample\_app.py /home/myapp/" >> tempdir/Dockerfile |   6. Use the Docker **EXPOSE** command to expose port 8080 for use by the webserver. Also add the Docker **CMD** command to execute the Python script.   |  | | --- | | echo "EXPOSE 8080" >> tempdir/Dockerfile  echo "CMD python3 /home/myapp/sample\_app.py" >> tempdir/Dockerfile |   7. Now to build the Docker container, add the commands to the **sample-app.sh** file to switch to the **tempdir** directory and build the Docker container. The **docker build** command **-t** option allows you to specify the name of the container and the trailing period (.) indicates that you want the container built in the current directory.   |  | | --- | | cd tempdir  docker build -t sampleapp . |   8. To start the container and verify it is running. Add the **docker run** command to the **sample-app.sh** file to start the container.   |  | | --- | | docker run -t -d -p 8080:8080 --name samplerunning sampleapp |   9. Add the **docker ps -a** command to display all currently running Docker containers. This command will be the last one executed by the bash script.    10. Lastly, Save your bash script.  **//Build, Run, and Verify the Docker Container**  1. In this part, you will execute bash script which will make the directories, copy over the files, create a Dockerfile, build the Docker container, run an instance of the Docker container, and display output from the **docker ps -a** command showing details of the container currently running. Then you will investigate the Docker container, stop the container from running, and remove the container.  **Note:** Be sure you stopped any other web server processes you may still have running from the previous parts of this lab.  **2.** Execute the bash script from the command line using **./sample-app.sh** . After creating the **tempdir** directories, the script executes the commands to build the Docker container.      3. The creation of the **tempdir** directories is not shown in the output for the script. You could add **echo** commands to print out messages when they are successfully created. You can also verify they are there with the **ls** Remember, this directory has the files and folders used to build the container and launch the web app. It is not the container that was built. Notice the Dockerfile created by your bash script. Open this file to see how it looks in its final form without the **echo.**   |  | | --- | | devasc@labvm:~/labs/devnet-src/sample-app$ ls tempdir/  Dockerfile sample\_app.py static templates  devasc@labvm:~/labs/devnet-src/sample-app$ cat tempdir/Dockerfile  FROM python  RUN pip install flask  COPY ./static /home/myapp/static/  COPY ./templates /home/myapp/templates/  COPY sample\_app.py /home/myapp/  EXPOSE 8080  CMD python3 /home/myapp/sample\_app.py |     4. The output for the **docker ps -a** command may be hard to read depending on the width of your terminal display. You can redirect it to a text file where you can view it better without word wrapping.   |  | | --- | | devasc@labvm:~/labs/devnet-src/sample-app$ docker ps -a >> running.txt |     5. The Docker container creates its own IP address from a private network address space. Verify the web app is running and reporting the IP address. In a web browser at **http://localhost:8080**, you should see the message **You are calling me from 172.17.0.1** formatted as H1 on a light steel blue background. You can also use the **curl** command.   |  | | --- | | devasc@labvm:~/labs/devnet-src/sample-app$ curl http://172.17.0.1:8080  <html>  <head>  <title>Sample app</title>  <link rel="stylesheet" href="/static/style.css" />  </head>  <body>  <h1>You are calling me from 172.17.0.1</h1>  </body>  devasc@labvm:~/labs/devnet-src/sample-app$ |     6. By default, Docker uses the **IPv4 172.17.0.0/16** subnet for container networking. (This address can be changed if necessary.) Enter the command ip address to display all the **ip** **addresses** used by your instance of the DEVASC VM. You should see the loopback address 127.0.0.1 that the web app used earlier in the lab and the new Docker interface with the IP address 172.17.0.1.    7. To access the running container, enter the **docker exec -it** command specifying the name of the running container (**samplerunning**) and that you want a bash shell (/bin/bash). The **-i** option specifies that you want it to be interactive and the **-t** option specifies that you want terminal access. The prompt changes to **root@containerID**. Your container ID will be different than the one shown below. Notice the container ID matches the ID shown in the output from **docker ps -a**.   |  | | --- | | devasc@labvm:~/labs/devnet-src/sample-app$ docker exec -it samplerunning /bin/bash  root@8953a95374ff:/# |   8. You are now in root access for the **samplerunning** Docker container. From here, you can use Linux commands to explore the Docker container. Enter ls to see the directory structure at the root level.   |  | | --- | | root@8953a95374ff:/# ls  bin dev home lib64 mnt proc run srv tmp var  boot etc lib media opt root sbin sys usr  root@8953a95374ff:/# |     9. Recall that in your bash script, you added commands in the Dockerfile that copied your app directories and files to the **home/myapp**. Enter the **ls** command again for that folder to see your **sample\_app.py** script and directories. To get a better understanding of what is included in your Docker container, you may wish to use the **ls** command to examine other directories such as /etc and /bin.   |  | | --- | | root@8953a95374ff:/# ls home/myapp/  sample\_app.py static templates  root@8953a95374ff:/# |     10. Exit the Docker container to return to the DEVASC VM command line.   |  | | --- | | root@8953a95374ff:/# exit  exit  devasc@labvm:~/labs/devnet-src/sample-app$ |   11. You can stop the Docker container with the docker stop command specifying the name of the running container. It will take a few seconds to clean up and cache the container. You can see that it still exists by entering the docker ps -a However, if you refresh the web page for **http://localhost:8080**, you will see the web app is no longer running.   |  | | --- | | devasc@labvm:~/labs/devnet-src/sample-app$ docker stop samplerunning  samplerunning  devasc@labvm:~/labs/devnet-src/sample-app$ docker ps -a  CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES |     12. You can restart a stopped container with the **docker start**. The container will immediately spin up.   |  | | --- | | devasc@labvm:~/labs/devnet-src/sample-app$ docker start samplerunning  samplerunning |     13. To permanently remove the container, first stop it and then remove it with the **docker rm** You can always rebuild it again executing the **sample-app** program. Use the **docker ps -a** command to verify the container has been removed.   |  | | --- | | devasc@labvm:~/labs/devnet-src/sample-app$ docker stop samplerunning  samplerunning  devasc@labvm:~/labs/devnet-src/sample-app$ docker rm samplerunning  samplerunning  devasc@labvm:~/labs/devnet-src/sample-app$ docker ps -a  CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  devasc@labvm:~/labs/devnet-src/sample-app$ | |

## Lab 13:  Build a CI-CD Pipeline Using Jenkins

### Case Study

CloudOps Innovators, a software solutions provider specializing in agile development and DevOps consulting, is expanding its cloud-native product line to meet increasing client demand for rapid and reliable software delivery. To maintain velocity and ensure code quality, the engineering team needed a robust automation strategy that aligned with DevOps best practices. The leadership decided to implement a Continuous Integration and Continuous Deployment (CI/CD) pipeline using Jenkins to automate code testing and deployment across development teams.

To prepare developers for this transition, CloudOps Innovators introduced a lab-based exercise where participants would use GitHub, Docker, and Jenkins to create a fully functioning CI/CD pipeline. The lab simulated real-world conditions, guiding learners through code versioning, build automation, test integration, and pipeline creation.

### Business Challenge

Before adopting CI/CD, CloudOps Innovators faced several operational hurdles. Code changes were manually tested and deployed, leading to delayed feedback cycles and occasional production errors. Developers spent significant time coordinating builds and resolving merge conflicts due to inconsistent code integration practices. Additionally, with multiple teams contributing to the same codebase, it became increasingly difficult to ensure application stability after each update.

The company lacked an automated system to run unit tests and trigger builds with every code change. This slowed down the release process and introduced a higher risk of undetected bugs reaching production. Leadership recognized the need for automation to improve development efficiency, ensure consistent test coverage, and maintain high deployment reliability.

### Solution

To solve these challenges, CloudOps Innovators designed a hands-on lab to teach developers how to build and manage a CI/CD pipeline using Jenkins. The solution was implemented through the following structured steps:

1. Commit the Sample App to Git

2. Modify the Sample App and Push Changes to Git

3. Download and Run the Jenkins Docker Image

4. Configure Jenkins

5. Use Jenkins to Run a Build of Your App

6. Use Jenkins to Test a Build

7. Create a Pipeline in Jenkins

|  |  |  |  |
| --- | --- | --- | --- |
| **// Commit the Sample App to Git**  1. Login at **https://github.com/** with your credentials. Select the “**New repository**” button or click on the “**+**” icon in the upper right corner and select “**New repository** “. In the Repository name write **Sample-app, in** Description write **Explore CI/CD with GitHub and Jenkins** and then select **Private** and thenselect **Create repository.**    2. To configure your Git credentials locally in the VM. Open a terminal window **with VS Code** in the DEVASC VM. Use your name in place of “Sample User” for the name in quotes ” “. Use @example.com for your email address.    3. To Initialize a directory as the Git repository. You will use the sample-app files you created in a previous lab. However, those files are also stored for your convenience in the **/labs/devnet-src/jenkins/sample-app** directory. Navigate to the **jenkins/sample-app** directory and initialize it as a Git repository.    4.Use the **git remote add** command to add a Git URL with a remote alias of “origin” and point to the newly created repository on GitHub. Using the URL of the Git repository you created in Step 1, you should only need to replace the **github-username**in the following command with your GitHub username.   |  | | --- | | devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ git remote add origin https://github.com/github-username/sample-app.git |     **Note**: Your GitHub username is case-sensitive.  5. Use the **git add** command to stage the files in the **jenkins/sample-app** Use the asterisk (\*) argument to stage all files in the current directory.    6. Use the **git status** command to see the files and directories that are staged and ready to be committed to your GitHub repository.    7. Use the **git commit** command to commit the staged files and start tracking changes. Add a message of your choice as used in command git **commit -m "Committing sample-app files."**    8. Use the **git push** **origin master** command to push your local sample-app files to your GitHub repository.      **Note:** If VS Code prompts you with “The extension ‘Git’ wants to sign in using GitHub” instead of requesting your username, it means there was a misconfiguration in either your GitHub credentials or the repository URL. Ensure the URL includes the exact case-sensitive username and repository name. If you encounter a "repository not found" error after entering your credentials, it's likely due to an incorrect URL. In both cases, run **git remote rm origin** to remove the previous configuration, then repeat Steps with the correct details.  **// Modify the Sample App and Push Changes to Git**  1. Since both the Jenkins Docker image and the sample-app are configured to use port 8080, you need to change the port number in the sample-app files to avoid a conflict. After modifying the port, you will run the sample-app to verify that it functions correctly on the new port. Once confirmed, you will push the updated files to your GitHub repository.  2. Ensure that you remain in the **~/labs/devnet-src/jenkins/sample-app** directory, as this is where the files linked to your GitHub repository are located. Open both **sample\_app.py** and **sample-app.sh** for editing. In **sample\_app.py**, modify the single occurrence of port **8080** and replace it with **5050** as required.     3. In **sample-app.sh**, change the three instances of port **8080** to **5050** as shown below.    4. Nowenter thebashcommand to **build your app using the new port 5050.**    5. Open a browser tab and navigate to **localhost:5050**. You should see the message **You are calling me from 172.17.0.1.**    6. Shut down the server when you have verified that it is operating on port 5050. Return to the terminal window where the server is running and press CTRL+C to stop the server.  7. Now you are ready to push your changes to your GitHub repository. Enter the commands **git add \*** and then **git status**.    8. Next, use command **git commit -m "Changed port from 8080 to 5050."** And then **git push origin master.**    9. You can verify that your GitHub repository is updated by visiting **https://github.com/github-user/sample-app**. You should see your new message (Changed port from 8080 to 5050.) and that the latest commit timestamp has been updated.    **// Download and Run the Jenkins Docker Image**  1. In this part, you will download the Jenkins Docker image. You will then start an instance of the image and verify that the Jenkins server is running. The Jenkins Docker image is stored here: **https://hub.docker.com/r/jenkins/jenkins**. At the time of the writing of this lab, that site specifies that you use the **docker pull jenkins/jenkins**command to download the latest Jenkins container.    2. Enter the following command on **one line**. You may need to copy it to a text editor if you are viewing a PDF version of this lab to avoid line breaks. This command will start the Jenkins Docker container and then allow Docker commands to be executed inside your Jenkins server.   |  | | --- | | devasc@labvm:~/labs/devnet-src/jenkins/sample-app$ docker run --rm -u root -p 8080:8080 -v jenkins-data:/var/jenkins\_home -v $(which docker):/usr/bin/docker -v /var/run/docker.sock:/var/run/docker.sock -v "$HOME":/home --name jenkins\_server jenkins/jenkins:lts |   3. The Jenkins server should now be running. Copy the admin password that displays in the output, as shown in the following.  **Note**: Do not enter any commands in this server window. If you accidentally stop the Jenkins server, you will need to re-enter the **docker run** command from Step 2. After the initial install, the admin password is displayed as shown below.    **// Configure Jenkins**  1. In this Part, you will complete the initial configuration of the Jenkins server. Navigate to **http://localhost:8080/** and login in with your copied admin password.  2. Click **Install suggested plugins** and wait for Jenkins to download and install the plugins. In the terminal window, you will see log messages as the installation proceeds. Be sure that you do not close this terminal window. You can open another terminal window for access to the command line.    **3.** After the installation finishes, you are presented with the **Create First Admin User** window. For now, click **Skip and continue as admin** at the bottom.    **4.** In the **Instance Configuration** window, do not change anything. Click **Save and Finish**at the bottom.    **5.** In the next window, click **Start using Jenkins**. You should now be on the main dashboard with a **Welcome to Jenkins!** message.    **// Use Jenkins to Run a Build of Your App**  1. Click the **Create a job** link directly below the **Welcome to Jenkins**! Alternatively, you can click **New Item** in the menu on the left.    2. In the **Enter an item name** field, fill in the name **BuildAppJob**. Click **Freestyle project** as the job type. In the description, the SCM abbreviation stands for software configuration management, which is a classification of software that is responsible for tracking and controlling changes in software. Scroll to the bottom and click **OK**.    3. Click the **General tab**, add a description for your job. For example, **“My first Jenkins job.”**    4. Click the **Source Code Management** tab and choose the **Git** radio button. In the Repository URL field, add your GitHub repository link for the sample-app taking care to enter your case-sensitive username. Be sure to add the .git extension at the end of your URL. For example: [h**ttps://github.com/github-username/sample-app.git**](https://github.com/github-username/sample-app.git)**.** For **Credentials**, click the **Add** button and choose **Jenkins**.    5. In the **Add Credentials** dialog box, fill in your GitHub username and password, and then click **Add**.    **Note**: You will receive an error message that the connection has failed. This is because you have not selected the credentials yet.  6. In the dropdown for **Credentials** where it currently says **None**, choose the credentials you just configured. After you have added the correct URL and credentials, Jenkins tests access to the repository. You should have no error messages. If you do, verify your URL and credentials. You will need to **Add** them again as there is no way at this point to delete the ones you previously entered.    7. At the top of the **BuildAppJob** configuration window, click the **Build**. For the **Add build step** dropdown, choose **Execute shell.** In the **Command** field, enter the command **bash ./sample-app.sh** to run the build for sample-app.sh script. Click the **Save** You are returned to the Jenkins dashboard with the **BuildAppJob** selected.    8. On the left side, click **Build Now** to start the job. Jenkins will download your Git repository and execute the build command **bash ./sample-app.sh**.    9. On the left, in the **Build History** section, click your build number which should be the **#1** unless you have built the app multiple times. On the left, click **Console Output**. You should see output similar to the following. Notice the success messages at the bottom as well as the output from the **docker ps** -a command. Two docker containers are running: one for your sample-app running on local port 5050 and one for Jenkins on local port 8080.      10. Open another web browser tab and verify sample app is running. Type in the local address, **localhost:5050**. You should see the content of your index.html displayed in light steel blue background color with **You are calling me from 172.17.0.1** displayed in as H1.    **// Use Jenkins to Test a Build**  1. In this part, you will create a second job that tests the build to ensure that it is working properly.  **Note:** You need to stop and remove the samplerunning docker container.    2. To start a new job for testing your sample-app. Return to the **Jenkins** web browser tab and click the Jenkins link in the top left corner to return to the main dashboard. Click the **New Item** link to create a new job. In the Enter an item name field, fill in the name **TestAppJob**.mClick **Freestyle project** as the job type. Scroll to the bottom and click **OK**.    3. Add a description for your job. For example, “**My first Jenkins test.”** Leave **Source Code Management** set to **None**. Click the **Build** **Triggers** tab and check the box, **Build after** **other projects are built.** For **Projects to watch**, fill in the name **BuildAppJob**.    4. Now, write the test script that should run after a stable build of the BuildAppJob. Click the **Build.** Click **Add build step** and choose **Execute shell**. Enter the following script. The **if** command should be all on one line including the ; **then**. This command will **grep** the output returned from the **curl** command to see if **You are calling me from 172.17.0.1** is returned. If true, the script exits with a code of **0**, which means that there are no errors in the **BuildAppJob** If false, the script exits with a code of 1 which means the **BuildAppJob** failed. Click **Save** and then the **Back to Dashboard** link on the left side.    5. Refresh the web page with the refresh button for your browser. You should now see your two jobs listed in a table. For the **BuildAppJob** job, click the build button on the far right (a clock with an arrow).    6. To verify, see the timestamp for the Last Success column update for both **BuildAppJob** and **TestAppJob**. This means your code for both jobs ran without error. But you can also verify this for yourself.  **Note:** If timestamps do not update, make sure enable auto refresh is turned on by clicking the link in the top right corner.  7. Click the Link for **TestAppJob**. Under **Permalinks**, click the link for your last build,    8. Click **Console Output**. You should see output similar to the following:    9. It is not necessary to verify your sample app is running because the **TestAppJob** already did this for you. However, you can open a browser tab for 17.0.1:5050 to see that it is indeed running.  **// Create a Pipeline in Jenkins**  1. Click the **Jenkins** link in the top left, and then **New Item**. In the **Enter an item name** field, type **SamplePipeline**. Select **Pipeline** as the job type. Scroll to the bottom and click **OK**.    2. Along the top, click the tabs and investigate each section of the configuration page. Notice that there are a number of different ways to trigger a build. For the **SamplePipeline** job, you will trigger it manually. In the **Pipeline** section, add the following script. Click **Save** and you will be returned to the Jenkins dashboard for the **SamplePipeline**   |  | | --- | | node {  stage('Preparation') {  catchError(buildResult: 'SUCCESS') {  sh 'docker stop samplerunning'  sh 'docker rm samplerunning'  }  }  stage('Build') {  build 'BuildAppJob'  }  stage('Results') {  build 'TestAppJob'  }  } |     3. On the left, click **Build Now** to run the **SamplePipeline** job. If you coded your Pipeline script without error, then the **Stage** **View** should show three green boxes with number of seconds each stage took to build. If not, click Configure on the left to return to the **SamplePipeline** configuration and check your Pipeline script.    4. Click the latest build link under **Permalinks**, and then click **Console Output**. You should see output similar to the following: |