



PRIFYSGOL CYMRU
Y Drindod Dewi Sant
UNIVERSITY OF WALES
Trinity Saint David
SWANSEA - ABERTAWE

Applied Computing

Wales Institute of Science and Art (WISA)

Network Programmability (ACCB5029)

Lab 7: Application Deployment

ICONS Used in this document:



Do research and complete the task. Need referencing/ in-text citations. Shall be included as a part of your coursework.



Record your Reflection.

 **Networking**
CISCO **Academy**

Task 1: (Build a sample Web App in a Docker Container)**Objectives**

1. Create and work with bash file.
2. Create, and configure Web App
3. Understand Dockers
 - 3.1. What is Docker?
 - 3.2. Build, Run and Verify the Docker container.

Task 2: (Research and build a new application in a Docker Container)**Objectives**

4. Discuss with your group (the group you are working for Coursework 1) and propose a new application, build it in a docker container, run it and document with adequate annotations and timestamped screenshots.
5. Individually, write a 250 words personal reflection on your understanding about API's and its deployment.

Required Resources

Host computer with DEVASC VM.

Instructions

Launch the DEVASC VM

1. Create a Simple Bash Script

Bash knowledge is crucial for working with continuous integration, continuous deployment, containers, and with your development environment. Bash scripts help programmers automate a variety of tasks in one script file. In this part, you will briefly review how to create a bash script. Later in the lab, you will use a bash script to automate the creation of a web app inside of a Docker container.

- 1.1. Create an empty bash script file and open using nano text editor.

Change your working directory to `~/labs/devnet-src/sample-app` and add a new file called `mybashfile.sh`.

```
devasc@labvm:~$ cd labs/devnet-src/sample-app/  
devasc@labvm:~/labs/devnet-src/sample-app$ touch mybashfile.sh  
devasc@labvm:~/labs/devnet-src/sample-app$ nano mybashfile.sh
```

- 1.2. Add the hash 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 (not shown here) 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.

Add the hash bang which tells the system that this file includes commands that need to be run in the bash shell.

#!/bin/bash

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.

- 1.3. Do research to refresh your skill to understand what a hash bang is and how to work with command-line text editors. This will be very useful during your next level of the course. Earlier in our lecture, we referred this command as “She-Bang”. Is there any difference between She-bang and Hash bang?



Document your Understanding and discuss with your group to reflect your understanding.

- 1.4. 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 `name`, and display a string of text with the name entered.

```
GNU nano 4.8 mybashfile.sh
#!/bin/bash
echo -n "Your Name Please!"
read name
echo "Your name is $name."
```

- 1.4.1. Exit nano and save your script.

Press CTRL+X, then Y, then ENTER to exit nano and save your script.

- 1.5. Run your script from the command line.

You can run it directly from the command line using the following command.

```
devasc@labvm:~/labs/devnet-src/sample-app$ bash mybashfile.sh
Your Name Please!Nitheesh
Your name is Nitheesh.
```

- 1.6. Change the mode of the script to an executable file for all users.

Change the mode of the script to an executable using the chmod command. (Refer to our Lab activities done in L4 – ACCA4024 module, 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).

```
devasc@labvm:~/labs/devnet-src/sample-app$ ls -l mybashfile.sh
-rw-rw-r-- 1 devasc devasc 78 Mar 10 05:35 mybashfile.sh
devasc@labvm:~/labs/devnet-src/sample-app$ chmod a+x mybashfile.sh
devasc@labvm:~/labs/devnet-src/sample-app$ ls -l mybashfile.sh
-rwxrwxr-x 1 devasc devasc 78 Mar 10 05:35 mybashfile.sh
```

- 1.7. Rename and execute the file.

- 1.7.1. 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.

- 1.7.2. Execute the script from the command line.

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 "./".

```
devasc@labvm:~/labs/devnet-src/sample-app$ mv mybashfile.sh userdetails
devasc@labvm:~/labs/devnet-src/sample-app$ ./userdetails
Your Name Please!Murugan
Your name is Murugan.
```

- 1.8. Investigate other bash scripts.



Do research to refresh your skill to understand what bash scripts are? Work with your group to complete simple tasks using bash scripts.



Document your Understanding and the discussions with your group to reflect your understanding on Bash scripts.

2. Create a Sample Web App

Before we can launch an application in a Docker container, we first need to have the app. We will create a very simple Python script that will display the IP address of the client when the client visits the web page.

2.1. Install Flask and open a port on the DEVASC VM firewall.

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, do research for more information and tutorials on the Flask framework. It is highly appreciated if you could discuss with your group and document your reflection.

```
devasc@labvm:~/labs/devnet-src/sample-app$ pip3 install flask
```

(Troubleshoot if required)

2.2. Open an application “sample_app.py file”

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.

2.2.1. Import required methods from flask library.

```
from flask import Flask
from flask import request
```

2.2.2. Create an instance of the Flask class and name it sample. *Be sure to use two underscores before and after the "name".*

```
sample = Flask(__name__)
```

- 2.2.3. Define a method to display the client IP address: Configure Flask so that when a user visits the default page (root directory), it displays a message with the IP address of the client.

```
@sample.route("/")
def main():
    return "You are calling me from " + request.remote_addr + "\n"
```

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.

- 2.2.4. Configure the app to run locally at `http://0.0.0.0:8080`, which is also `http://localhost:8080`. *Be sure to use two underscores before and after "name", and before and after "main".*

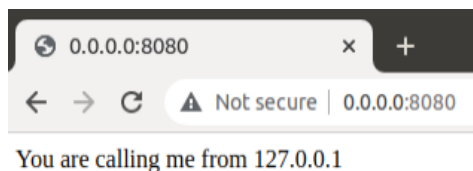
```
if __name__ == "__main__":
    sample.run(host="0.0.0.0", port=8080)
```

- 2.2.5. Save your script and run it from the command line. 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, troubleshoot.

```
* 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/ (Press CTRL+C to quit)
```

- 2.2.6. 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 following output:



If you receive an "HTTP 400 Bad Request" response, check your `sample_app.py` script carefully.

(or)

Open another terminal window and use the command-line URL tool (cURL) to verify the server's response.

```
devasc@labvm:~$ curl http://0.0.0.0:8080
You are calling me from 127.0.0.1
devasc@labvm:~$
```

- 2.2.7. Stop the server: Return to the terminal window where the server is running and press CTRL+C to stop the server.

2.3. Configure the Web App to use Website Files

In this part, build out the sample web app to include an index.html page and style.css specification. The index.html is typically the first page loaded in a client's web browser when visiting your website. The style.css is a style sheet used to customize the look of the web page.

- 2.3.1. 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. 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>My First Sample app</title>
  <link rel="stylesheet" href="/static/style.css" />
</head>
<body>
  <h1>You are calling me from {{request.remote_addr}}</h1>
  <h2> Created by - Nitheesh for accb5029 </h2>
</body>
</html>
devasc@labvm:~/labs/devnet-src/sample-app$ cat static/style.css
body {background: rgb(242, 6, 22);}
```

- 2.3.2. Update the Python code for the sample web app.

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.

```
# A sample web application created using python.
# Importing Flask library
from flask import Flask
from flask import request

# Importing render_template method from the flask library
# HTML file can be rendered in Flask automatically using the render_template function
from flask import render_template

sample = Flask(__name__)
```

```
@sample.route("/")
def main():
    #request.remote_addr is a property in the Flask web framework
    # It asks the IP address of the client that sent the current request.
    # This property returns a string representing the IP address of the client machine or the last proxy that sent the request.

    #return "You are calling me from " + request.remote_addr + "\n"
    # editing to the return function to use render_Template
    return render_template("index.html")

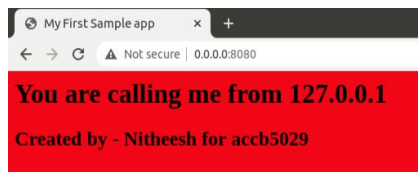
#Use the function run in the main
if __name__ == "__main__":
    sample.run(host="0.0.0.0", port=8080)
```

2.3.3. Save and run your script.

```
* 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/ (Press CTRL+C to quit)
```

Note: If you got Traceback output and an error with the message with something like OSErrno: [Errno 98] Address already in use, then you did not shut down your previous server. Return to the terminal window where that server is running and press CTRL+C to end the server process. Re-run your script.

2.3.4. Verify your program is running using Chromium web browser or terminal window.



```
devasc@labvm:~$ curl http://0.0.0.0:8080
<html>
<head>
  <title>My First Sample app</title>
  <link rel="stylesheet" href="/static/style.css" />
</head>
<body>
  <h1>You are calling me from 127.0.0.1</h1>
  <h2> Created by - Nitheesh for accb5029 </h2>
</body>
```

3. Understanding Dockers

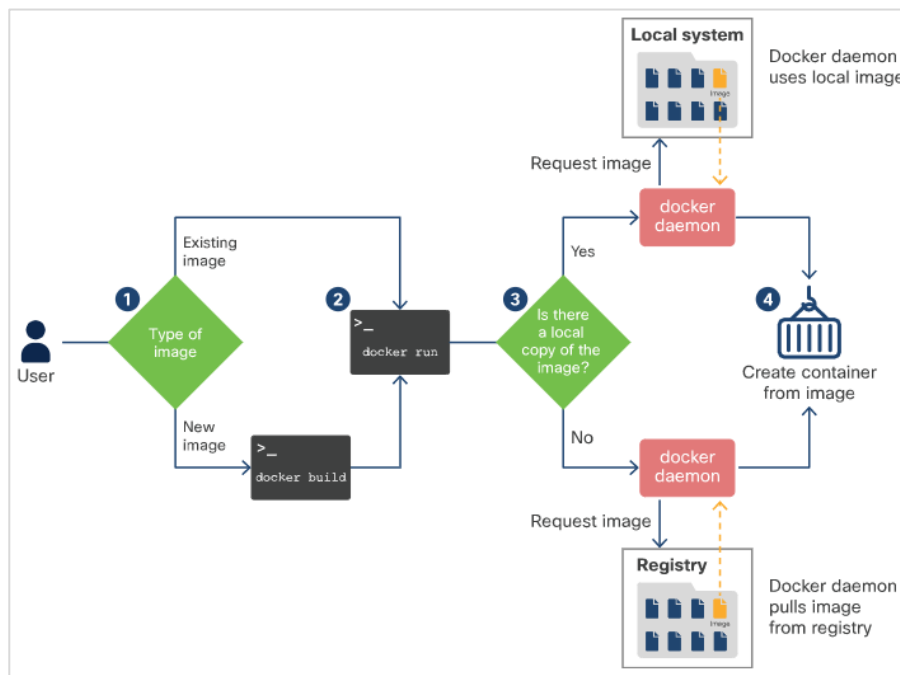
3.1. What is Docker?

The most popular way to containerize an application is to deploy it as a Docker container. A container is a way of encapsulating everything you need to run your application, so that it can easily be deployed in a variety of environments. **Docker is a way of creating and running that container.**

Docker is a format that wraps several different technologies to create containers including:

- + **Namespaces** - These isolate different parts of the running container.
- + **Control groups** - These **cgroups** are a standard Linux concept that enables the system to limit the resources, used by an application.
- + **Union File Systems** - These **UnionFS** are file systems that are built layer by layer, combining resources.

A Docker image is a **set of read-only files** that have no state and contains source code, libraries, and other dependencies needed to run an application. **A Docker container is the run-time instance of a Docker image.** Creating a container involves pulling an image or a template from a repository, then using it to create a container.



Dockerfile is a simple text-file which is required to compile the code. It defines the steps that the **docker build** command takes to create an image that can be used to create the target container.

Eg:

Consider the following **Dockerfile** that containerizes a Python app and the explanation of the commands are as follows:

```
FROM python
WORKDIR /home/ubuntu
COPY ./sample-app.py /home/ubuntu/.
RUN pip install flask
CMD python /home/ubuntu/sample-app.py
EXPOSE 8080
```

- The FROM command installs Python in the Docker image.
- The WORKDIR command tells Docker to use /home/ubuntu as the working directory.
- The COPY command tells Docker to copy the file from Dockerfile's current directory into /home/ubuntu.
- The RUN command allows to directly run commands on the container.
- The CMD command will start the server when the user run the actual container.
- The EXPOSE command tells Docker that the user want to expose port 8080.

Docker takes advantage of what is stored in cache to speed up the process.

The docker build command is used to build the image.

The Docker goes through each step in the Dockerfile, starting with the base image. If this image does not exist on the system, Docker pulls it from the registry. The default registry is Docker Hub.

Between steps such as executing a command, Docker creates a new container and builds an intermediate image, a new layer, by saving that container.

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

An application can be deployed on a bare metal server (physical server dedicated to a single-tenant environment) or in a virtual machine, like you just did in the previous part. It can also be deployed in a containerized solution like Docker.

In this part, you will create a bash script and add commands to it that complete the following tasks to build and run a Docker container:

- + **Create temporary directories to store the website files.**
- + **Copy the website directories and sample_app.py to the temporary directory.**
- + **Build a Dockerfile.**
- + **Build the Docker container.**
- + **Start the container and verify it is running.**

3.2.1. 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 hash bang and the commands to create a directory structure with tempdir as the parent folder.
- + Copy the website directories and sample_app.py to the temporary directory.

```
#!/bin/bash
# command to create a directory structure with tempdir as the parent folder
mkdir tempdir
mkdir tempdir/templates
mkdir tempdir/static
```

3.2.2. In the sample-app.sh file, Copy the website directory and script to tempdir.

```
#Copy the website directories and sample_app.py to the temporary directory
cp sample_app.py tempdir/.
#Recursive copy the template files
cp -r templates/* tempdir/templates/.
cp -r static/* tempdir/static/.
```

3.2.3. Create a Dockerfile.

In this step, you 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.

- 3.2.3.1. You need Python running in the container, so add the Docker FROM command to install Python in the container.

```
echo "FROM python" > tempdir/Dockerfile
```

- 3.2.3.2. Your sample_app.py script needs Flask, so add the Docker RUN command to install Flask in the container.

```
echo "RUN pip install flask" >> tempdir/Dockerfile
```

3.2.3.3. Your container will need the website folders and the sample_app.py script to run the app, so add the Docker COPY commands to add them to a directory in the Docker container.

- + In this example, 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 "COPY ./static /home/myapp/static/" >> tempdir/Dockerfile
echo "COPY ./templates /home/myapp/templates/" >> tempdir/Dockerfile
echo "COPY sample_app.py /home/myapp/" >> tempdir/Dockerfile
```

3.2.3.4. Use the Docker EXPOSE command to expose port 8080 for use by the webserver.

```
echo "EXPOSE 8080" >> tempdir/Dockerfile
```

3.2.3.5. Finally, add the Docker CMD command to execute the Python script.

```
echo "CMD python3 /home/myapp/sample_app.py" >> tempdir/Dockerfile
```

3.2.4. 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 .
```

3.2.5. Start the container and verify it is running.

3.2.5.1. 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
```

The docker run options indicate the following:

- + -t specifies that you want a terminal created for the container so the you can access it at the command line.
- + -d indicates that you want the container to run in the background and print the container ID when executing the docker ps -a command.
- + -p specifies that you want to publish the container's internal port to the host. The first "8080" references the port for the app running in the docker container (our sampleapp). the second "8080" tells docker

to use this port on the host. These values do not have to be the same. For example, an internal port 80 to external 800 (80:800).

- + `--name` specifies first what you want to call the instance of the container (samplerunning) and then the container image that the instance will be based on (sampleapp). The instance name can be anything you want. However, the image name needs to match the container name you specified in the docker build command (sampleapp).

3.2.5.2. 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.

```
docker ps -a
```

3.2.6. Save your bash script.

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.

3.2.6.1. Execute the bash script from the command line.

```
devasc@labvm:~/labs/devnet-src/sample-app$ ./sample-app.sh
```

Not having permission? Make necessary changes!!!

3.2.6.2. You should see output like the following.

- + After creating the tempdir directories, the script executes the commands to build the Docker container.
- + Notice that Step 7/7 in the output executes the `sample_app.py` that creates the web server.
- + Also, notice the container ID. You will see this in the Docker command prompt later in the lab.

```

Sending build context to Docker daemon  7.168kB
Step 1/7 : FROM python
--> 33ced9363487
Step 2/7 : RUN pip install flask
--> Using cache
--> d57b17f1d990
Step 3/7 : COPY ./static /home/myapp/static/
--> Using cache
--> 920f359868ee
Step 4/7 : COPY ./templates /home/myapp/templates/
--> Using cache
--> 34b177e7be85
Step 5/7 : COPY sample_app.py /home/myapp/
--> Using cache
--> d6d5f9d8cdec
Step 6/7 : EXPOSE 8080
--> Using cache
--> d12065730248
Step 7/7 : CMD python3 /home/myapp/sample_app.py
--> Using cache
--> f8a55a3458f9
Successfully built f8a55a3458f9
Successfully tagged sampleapp:latest
71903c82d3017bf6b7d8c0de491608411354eb0eb6ef1a256e42a4ad216864ea
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
71903c82d301        sampleapp           "/bin/sh -c 'python3..."   1 second ago        Up Less than a second    0.0.0.0:8080->8080/tcp    samplerunning

```

3.2.7. Investigate the running Docker container and the web app.

- + You could add echo commands to print out messages when they are successfully created.
- + You can also verify they are there with the ls command.
- + 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.

3.2.7.1. Use appropriate commands to investigate the Docker container and the



web app. You shall use ls commands, echo commands etc. to investigate the docker container. What are the commands used and write your observations.

3.2.7.2. Verify the web app is running and reporting the IP address.

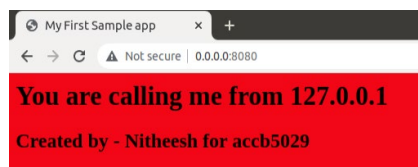


3.2.7.3. You can also use the curl command and check:

```
devasc@labvm:~$ curl http://172.17.0.1:8080
<html>
<head>
  <title>My First Sample app</title>
  <link rel="stylesheet" href="/static/style.css" />
</head>
<body>
  <h1>You are calling me from 172.17.0.1</h1>
  <h2> Created by - Nitheesh for accb5029 </h2>
</body>
</html>devasc@labvm:~$
```

If you didn't get the above output, troubleshoot. Check if the web app is running? Record your troubleshooting steps.

"Calling me from 172.17.0.1"? Where is this IP Address from? Did you understand what it is? Earlier, before creating a docker container, when we executed this web app, we got.



And now it is 172.17.0.1. Why? Reflect on this.

3.2.7.4. Access and explore the running container.

Remember that a Docker container is a way of encapsulating everything you need to run your application so that it can easily be deployed in a variety of environments--not just in your DEVASC VM.

- + 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`. Notice the container ID matches the ID shown in the output from `docker ps -a`.

```
devasc@labvm:~$ docker exec -it samplerunning /bin/bash
root@71903c82d301:/#
```

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@71903c82d301:/# ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
```

Recall that in your bash script, you added commands in the Dockerfile that copied your app directories and files to the `home/myapp` directory. 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` and finally, exit the Docker container to return to the DEVASC VM command line.

```
root@71903c82d301:/# ls home/myapp/
sample_app.py static templates
root@71903c82d301:/# exit
exit
```

3.2.8. Stop and remove the Docker container.

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` command. However, if you refresh the web page for `http://localhost:8080`, you will see the web app is no longer running.

```
devasc@labvm:~$ docker stop samplerunning
samplerunning
devasc@labvm:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
71903c82d301	sampleapp	"/bin/sh -c 'python3..."	33 minutes ago	Exited (137) 5 seconds ago		samplerunning

You can restart a stopped container with the docker start command. The container will immediately spin up.

3.2.9. To permanently remove the container,

- + first stop it and then remove it with the docker rm command.
- + 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:~$ docker stop samplerunning
samplerunning
devasc@labvm:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
71903c82d301	sampleapp	"/bin/sh -c 'python3..."	33 minutes ago	Exited (137) 5 seconds ago		samplerunning

```
devasc@labvm:~$ docker rm samplerunning
samplerunning
devasc@labvm:~$ docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

Task 2: (Research and build a new application in a Docker Container)

Objectives

1. Discuss with your group (the group you are working for Coursework 1) and propose a new application, build it in a docker container, run it and document with adequate annotations and timestamped screenshots.
2. Individually, write a 250 words personal reflection on your understanding about API's and its deployment.
3. Submit the work in Moodle.

/* End of document.