TASK

Containers – Docker

Visit our website

Introduction

At this point, you have had exposure to developing programs using different frameworks and libraries. You may have wondered how people keep their heads wrapped around all the different requirements of making a program work when running it somewhere that isn't your PC. There are many different ways of doing this. In this task, we'll be covering containerisation.

WHAT IS DEPLOYMENT?

Deployment is the technical name for getting your program out to the world and running on your users' devices – be it in a web browser on a laptop, as a smartphone app, or even on a smart fridge. Back in the early days of software development, this usually only happened once for a software product, since it was very difficult to update an app after its deployment. Today, with more devices connected to the internet than people are walking the Earth, it's easier than ever to update apps (O'Neill, 2024; Vailshery, 2024). Nowadays, it's common for an app to be deployed multiple times a month – just look at how frequently your smartphone's apps update!

From the developer's point of view, there are many ways to deploy an app. You've already seen one such way earlier in this course – GitHub Pages. It's wonderful for simple web pages, but it's not intended for anything more complex. If you want your app to store and retrieve data on a database, handle user authentication, or almost anything else, you'll need a more comprehensive environment.

TRADITIONAL DEPLOYMENT

Different kinds of applications deploy differently. For instance, mobile apps deploy with systems managed by Google Play Store or then iOS App Store; Windows apps deploy with Microsoft Store. The traditional way of deploying web-based applications is by using remote virtual machines (VMs). These are special kinds of operating systems that are built to work with data and interact with other programs (not people) – they are light on system resources and come without any graphics. All your code and its dependencies would be installed in this virtual environment. Doing this requires a background in system administration and is beyond the scope of this course.

CONTAINERISATION

In 2013, a company called Docker developed a much simpler way of deploying applications using containers. A container is a virtual environment you can configure any way you like with the purpose of being a fully encompassing package of your software. Ideally, you can give the container to anyone with a PC and they would be able to run your program with no hassle. All your application's dependencies are included within the container.

GETTING DOCKER RUNNING

Docker Desktop is an app that helps you manage Docker containers on your machine. You can install it for your operating system by going to this Docker webpage: **https://www.docker.com/products/docker-desktop**.

If your machine does not meet the system requirements, you'll have to do this entire task using Docker Playground. This is a service that gives you an online virtual space in which you can spin up and test Docker containers to your heart's content. It works by giving you access to a VM with Docker already installed. Go to the **Play with Docker** website and log in with your Docker Hub account. If you don't have a Docker Hub account, create one now by visiting the **Docker Hub**. You'll need an account later in this lesson.

After logging in with Docker Playground, click "Start". The playground has a **four-hour time limit**, after which everything you did there will be deleted. It should be enough time to do this entire task, but if you feel you need more time, we recommend doing this task in two sessions: for the first session follow along with the tutorial, and for the second session do the tasks at the end of the lesson. In between these sessions, you should click on "Close Session" and start again.

After clicking "Start" you'll be able to create instances. These instances are lightweight, standalone, and executable software packages that include everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings, where you can execute any commands you like. This is possible because of a technology called **secure shell** (SSH). SSH is the de-facto way of executing remote commands on VMs. It's the reason you can type commands into a terminal in your web browser while they execute on a machine elsewhere in the world.

If you have Docker Desktop working, you'll notice it provides a neat graphical user interface (GUI), but we'll only be using the command-line version (which comes installed with Docker Desktop) in this lesson. This is because the command line gives us much more flexibility and declarability when using Docker. Its syntax is also platform-independent (it works the same on any operating system), and it better helps you understand how Docker works.

HELLO WORLD

To verify that Docker is installed and works, run the following command from your terminal. Note that Docker recommends always issuing commands in elevated mode (run as admin on Windows or root/sudo on macOS and Linux). Docker Playground automatically uses root to issue commands.

```
docker run hello-world
```

It will do some downloads and, finally, you'll see output starting with "Hello from Docker!". The output explains what it did, but in a nutshell, Docker downloaded an image (which is like a container specification), from which it created a container, within which it ran a program, the output of which was sent to your terminal. Without you needing to install any extra software, it just did all that. And it can do much more.

CREATING A SIMPLE CONTAINER

We will now create a simple webpage and containerise it so that it can be served from any machine. The first step is to make the webpage. Using your IDE, create a file called **index.html** and put anything you want in it. We recommend a few styled lines of text and perhaps an image. Feel free to use Bootstrap to style your page.

If you're using Docker Playground, we recommend you use the command-line text editor Vim to create and edit files. Use this **beginner's guide to Vim** to get started. To get files you have on GitHub on your machine, use `git clone [repo_url]` to get it on the VM.

A **Docker container** is an environment in which your code will run. A **Docker image** is the bundle of your code and its dependencies that can be created into said container. We need to create a Docker image and, to do so, we must tell Docker exactly what we need in it. We do this by specifying a Docker file. Create a file called **Dockerfile** (no extension) and place the following lines in it:

```
FROM nginx

COPY . /usr/share/nginx/html
```

Docker has a great open-source community of developers publishing their images for everyone to use for free, called **Docker Hub**. Later in this lesson, you'll be publishing your own Docker image and becoming part of that community too. One of these developer groups created an image called **nginx**, which is a popular web server. The first line in our Docker file tells Docker to fetch this image from the Docker Hub because we'll be building on it and using it for our own image. This is common practice in containerisation.

The second line tells Docker to copy everything in our directory (that's what the dot means) to a specific directory inside the container. This directory is where nginx serves its web content. So we definitely want our **index.html** and other web files copied in there.

We will now create a Docker image. Make sure all your web files and the Docker file are in the same directory and then use `cd` to navigate to that directory in your terminal. Run the following command:

```
docker build -t my-website ./
```

This will first download the nginx image and then copy your web files into a new image-based thereon. The `-t` flag specifies the tag (name) of your image. The `./` argument indicates that the image should be built from the current directory. The image will now have been created and exists somewhere on your machine. You don't have to worry about where it is or what it looks like – all you need to know is how to create it from your Docker file. This is the standard way of using and exporting Docker images.

RUNNING THE IMAGE

To create the container, you 'run' the image. To do so, issue the following command:

```
docker run -d -p 80:80 my-website
```

The `-d` flag tells Docker to run the image in detached mode (in the background). The `-p` flag specifies that your machine's network port 80 should link to the container's port 80. This creates a network tunnel of sorts between your computer and the container. You'll see that the command outputs a long alphanumeric string. This is the ID of the running container. You'll be needing it later, but don't worry about copying it – there are easy ways of getting that ID again from Docker.

If you're running Docker on your local machine, open your browser and go to **http://localhost**. If you're using Docker Playground, you should see a button with the number "80" next to "OPEN PORT" at the top of the webpage. Click on it to visit that VM's exposed port 80.

You should now see your website!

By default, nginx will keep serving indefinitely. Shut down your container by first getting its ID from this command:

```
docker ps
```

You'll notice that it only prints the first 12 characters of the container's ID. This is because Docker is smart enough to understand which container you're referring to without listing the entire ID. Go ahead and shut down the container using only the first three characters of its ID:

```
docker stop [ID]
```

DEPLOYING YOUR DOCKER IMAGE

Now that your Docker image is ready to run, we'll look at how we can deploy it so that anyone on any machine can run it. The first step is to publish it on Docker Hub. Go create a free account now if you haven't yet on **Docker Hub**.

Click the "Create Repository" button to create your own Docker repository. It's much like a GitHub repository, except it's just for Docker images. Also, every Docker program is automatically configured to interact with Docker Hub easily. Call it something simple, like **my-website**, and make sure it's public.

Before uploading it, you need to rename your image so that it matches your repository. The format is `user/repo`, where `user` is your Docker Hub username and `repo` is the repository's name.

```
docker tag my-website [user]/[repo]
```

Before you can upload your image, you need to login from the command line. Use the command below. It will prompt you for your Docker Hub username and password.

```
docker login
```

You can now upload your image.

```
docker push [user]/[repo]
```

RUNNING YOUR IMAGE ELSEWHERE

To prove that your Docker image can run anywhere, we'll be running it on a VM hosted in the USA. If you haven't been using Docker Playground thus far, go to the **Play with Docker** website and login with your Docker Hub account:. Then click "Start", and on the next screen click "add new instance".

In the terminal, issue the following command to automatically download and run your image:

```
docker run -d -p 80:80 [user]/[repo]
```

Docker on the VM will download your image from Docker Hub and do all the relevant preparations and installation to make your app work. Docker is pretty smart when it comes to images. It doesn't just make a sealed package with all the necessary code and dependencies. Instead, it uses a modular approach and splits an image up into layers – each having only what is necessary to build upon the previous layer. For example, the image you just built has an nginx layer, and then your layer (your code). Nginx will have its own set of layers, too. You can see each layer being downloaded and set up individually in the docker command's output.

Luckily, all this is managed automatically by Docker and you need only to concern yourself with your layer.

When the run command is done, you should see a button with the number "80" next to "OPEN PORT" at the top of the webpage. Click on it to visit that VM's exposed port 80. You should see your website now running completely on its own on the world wide web.

CONSOLE-BASED CONTAINERS

Containers are usually used for web applications, but sometimes you may want to use them for console-based apps too. In a new directory, create a file called **hello.py** and make it print something simple to output. Ensure that it works normally. If you're using Docker Playground, test the code on your machine first before putting it on the VM. The VM comes with Python installed. However, we'll specifically be using a different Python interpreter, **PyPy**, that doesn't come installed with the VM.

Create a Docker file in the same directory with the following contents (if you're using Docker Playground, use one of the described methods to get it on the VM):

```
FROM pypy:latest

WORKDIR /app

COPY . /app

CMD python hello.py
```

- The first line bases the image on PyPy, which is an alternative implementation of Python. We specified the latest version. The Python version you've been using is commonly referred to as CPython. However, there are different implementations of Python's specification that are better optimised for a range of scenarios. PyPy is one of them.
- The `WORKDIR` is the working directory where your app will be built and executed.
- Finally, `CMD` specifies the main command to be run when the image boots.
- Note that `WORKDIR` and `CMD` aren't needed for our web app, because nginx takes care of its own execution.

Build and run the image:

```
docker build -t python-app ./

docker run python-app
```

Note that we do not have to specify any ports or create a network tunnel. Docker automatically pipes the output of the `CMD` command to your console.

You should now see the output of your application printed directly to your console. If your Python script takes command-line input, you should specify the `-i` flag, which passes your command-line input directly to the containerised app. For example:

```
docker run -i python-app
```

Afterwards, the container automatically shuts down because it has nothing further to do (unlike our web app that keeps serving at all times).

Try pushing your new image to Docker Hub and running it in Docker Playground to check that it works anywhere.

CONTAINERISING A DJANGO WEB APPLICATION

Now that you've seen how to containerise simple Python scripts, let's apply these same Docker principles to a more complex application - a Django web app. This will show you how Docker handles multi-file projects, dependencies, and web services.

Step 1: Start Docker Desktop

IMPORTANT: Before running any Docker commands, you MUST start Docker Desktop:

1.1 Open Docker Desktop application on your computer
1.2 Wait for Docker Desktop to fully start up before proceeding to creating your Django app.
1.3 The Docker icon in your system tray should show it's running (not grayed out)

Step 2: Create the Django Project

2.1. Create the project folder:

```
mkdir django_docker_demo
```

2.2. Navigate into the directory:

```
cd django_docker_demo
```

2.3. Install Django:

```
pip install django
```

2.4. Start Django project:

```
django-admin startproject mysite .
```

2.5. Create a simple app:

```
python manage.py startapp hello
```

Step 3: Add a Simple View

3.1. Add the following code in **`hello/views.py`** to define a basic view that returns a plain-text response:

```
from django.http import HttpResponse

def home(request):
    return HttpResponse("Hello, Docker!")
```

3.2. Map the URL in **`hello/urls.py`**. Create the file if it doesn't exist:

```
from django.urls import path
from .views import home

urlpatterns = [
    path('', home, name='home'),
]
```

3.3. Update the main URL configuration in **`mysite/urls.py`** to include the hello app's URLs:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('hello.urls')),
]
```

3.4. In **`mysite/settings.py`,** make sure the hello app is listed under INSTALLED_APPS so Django knows to include it:

```
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
```

```
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "hello", # Add this line
]
```

Step 4: Set Up Docker Configuration Files

4.1. Create a file called "`Dockerfile`" in the root directory of your project. This file should not have an extension). This file defines the environment your app will run in. It tells Docker which base image to use, sets environment variables, installs dependencies, and copies your project files into the image.

```
# Use official Python image
FROM python:3.11-slim

# Set environment variables
ENV PYTHONDONTWRITEBYTECODE 1
ENV PYTHONUNBUFFERED 1

# Set work directory
WORKDIR /code

# Install dependencies
COPY requirements.txt /code/
RUN pip install --upgrade pip && pip install -r requirements.txt

# Copy project
COPY . /code/
```

4.2. Create a `requirements.txt`. This file lists all the Python packages your project depends on. Docker will use this to install the necessary packages inside the container.

```
Django>=4.2
```

4.3. Create a file called `docker-compose.yml` in the root directory of your project. This file defines how to run your Docker containers. It specifies the services, build instructions, commands to start your app, and how ports and volumes should be mapped.

```
services:
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - "8000:8000"
```

Step 5: Run the Django App with Docker Desktop

5.1. Run this command to build your Docker image based on the Dockerfile and dependencies:

```
docker-compose build
```

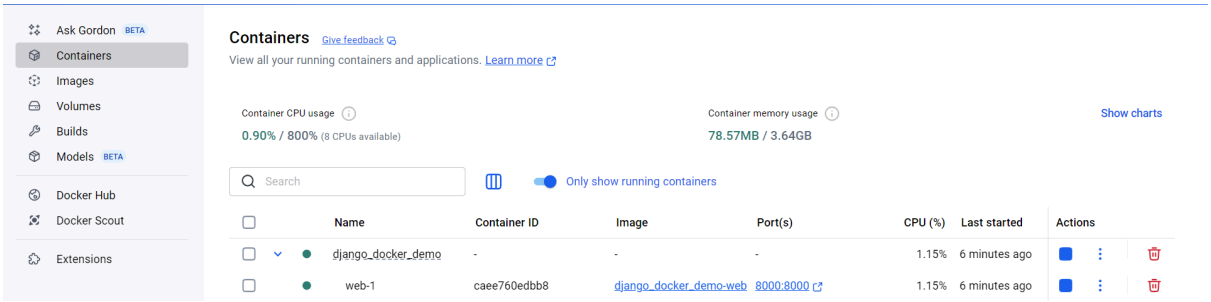
5.2. Run your container and launch the Django development server:

```
docker-compose up
```

Step 6: Verify Containerization Success

In Docker Desktop - After completing the previous steps, open the Docker Desktop application and check the following:

1. Click on “Containers” in the left sidebar.
2. You should see your container listed.
3. The container should show a green “Running” status, indicating it’s active.



In Your Web Browser - To verify that your Django app is running correctly:

1. Open your browser and go to "`http://localhost:8000`"
2. You should see the message: "Hello, Docker"

Additional Steps (Command Line)

1. Check Running Containers:

Use the following command to list active containers. You should see your Django container running and mapped to port `8000:8000`:

```
docker ps
```

2. View Container Logs:

This command will display the Django server logs so you can confirm the app started successfully:

```
docker-compose logs
```

3. Stop the Container:

To stop and remove the running container, use:

```
docker-compose down
```

Publishing Your Django Image

You can publish this to Docker Hub just like our previous examples. Remember to login to Docker Hub (if not already logged in).

1. Check what image Docker Compose created:

```
docker images
```

2. Tag the image using the actual name from the previous step:

```
docker tag [actual-image-name] [your-username]/django-app
```

3. Thereafter, push to Docker Hub:

```
docker push [your-username]/django-app
```

CAVEATS

Containers are cool, but they can't do everything. Most notably, containers are very bad at running GUI programs. This is because GUI programs expect special libraries, temporary directories, and internal network ports in order to interact with your desktop environment properly. While it is possible, it's very difficult to set up and quite prone to bugs. For this course, you'll only ever be expected to manage containers for console-based and network-based apps. This is generally true in the industry too.

Practical task

Containerise any console-based Python project that you have previously completed in this bootcamp.

Do all of the following instructions:

- Create an appropriate `Dockerfile` for your code.
- Create an image from this `Dockerfile`.
- Upload your image to Docker Hub.
- Ensure that your image can run on a machine that isn't your computer by using Docker Playground.

Submit the following:

- The `Dockerfile` you created.

- A link to your Docker Hub repository in a text file called **docker1.txt**.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Optional extension

For all past and future capstone projects, if you decide to publish your code on GitHub, include a working Dockerfile. This shows future recruiters that you understand containerisation, which is a big bonus in the software developer market.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

REFERENCE LIST

Vailshery, L. S. (2024). Number of IoT connections worldwide from 2022 to 2033, with forecasts from 2024 to 2033. *Statista*.
https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/

O'Neill, A. (2024). Global population by continent as of July 1, 2024. *Statista*.
https://www.statista.com/statistics/262881/global-population-by-continent/