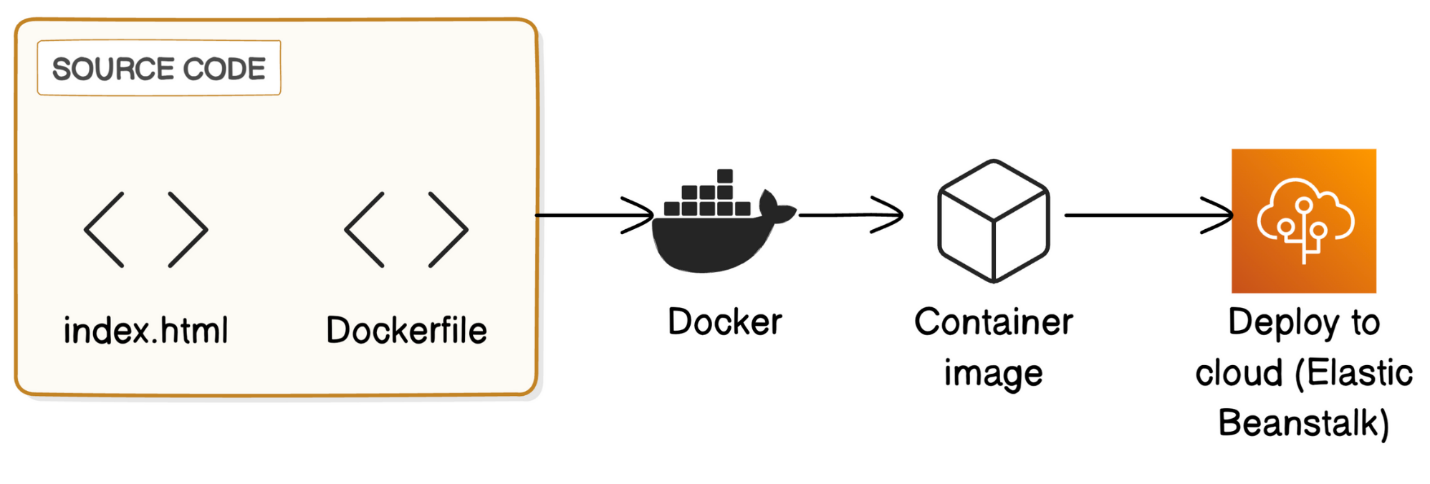
Deploy an App with Docker

## Introduction



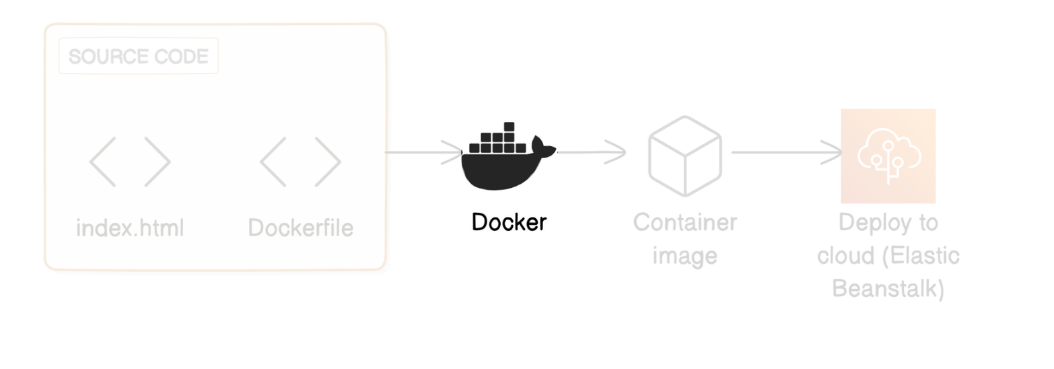
**Why care about containers?**

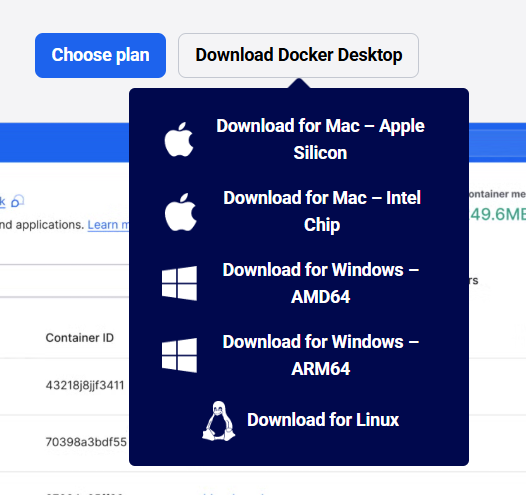
* When developers create applications, they often use containers to package up their work and share it in a way that’s easy for other people to run. Using containers has absolutely changed the game for making software fast and efficient, and it’s also the reason why deploying software has become much simpler.

In this project, I will demonstrate how to build and run a Docker container, and deploy it to AWS Elastic Beanstalk. I’m doing this project to learn how to containerize an application and host it in the cloud so I can later deploy my own websites and applications.

## Step #1 - Installing Docker

Docker is the foundation tool for today’s project. The first element of the above diagram, the source code, represent an app that I am about to build. Technically, I could work on the source code first, but it will actually be a lot easier to understand the source code if I install and use Docker first.





Downloaded for Windows – AMD64 for my PC.

**What is Docker Desktop?**

* It is a program that makes it easy to work with Docker, a tool for creating and managing containers. Engineers use Docker Desktop to build, test, and deploy applications right from their computers and use Docker in a user-friendly way.

**What are containers? Why do they exist?**

* Containers solve a common problem called the “it works on my machine” problem. Have you ever seen someone run code perfectly on their computer, but somehow get errors when you run the exact same thing on your computer?
* Software is not guaranteed to run and behave the way it’s supposed to in every computer, because each machine comes with its own operating system, software versions, local resources and more.

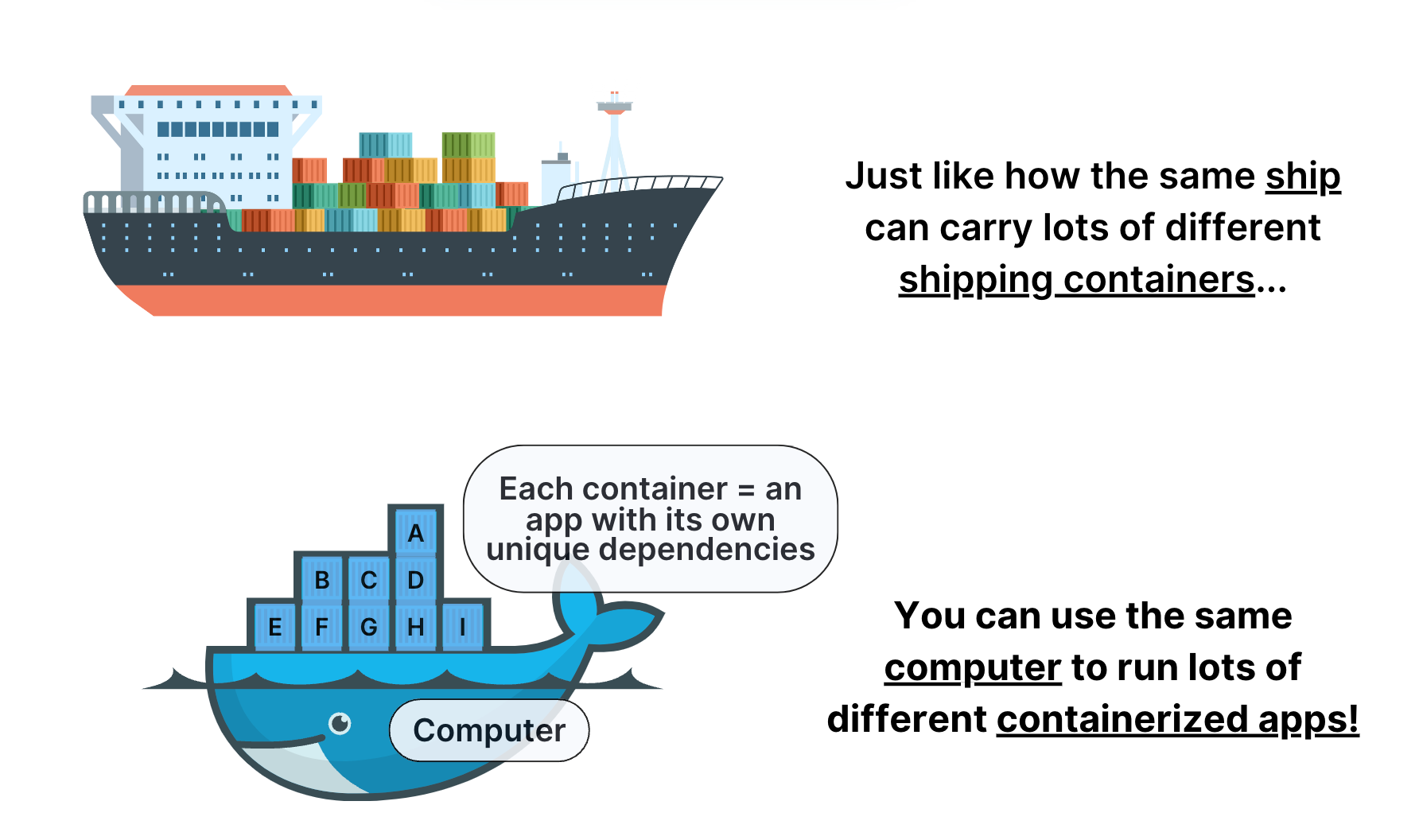
Containers tackle this problem in two ways:

1. Containers package up **my application** and everything it needs to run (dependencies etc.) in one file. Now other developers can run this package instead of the application itself, which gets the application working much faster.
2. Containers also let you run other dev’s applications and software much faster, since a container comes with everything you need to get it working ASAP.

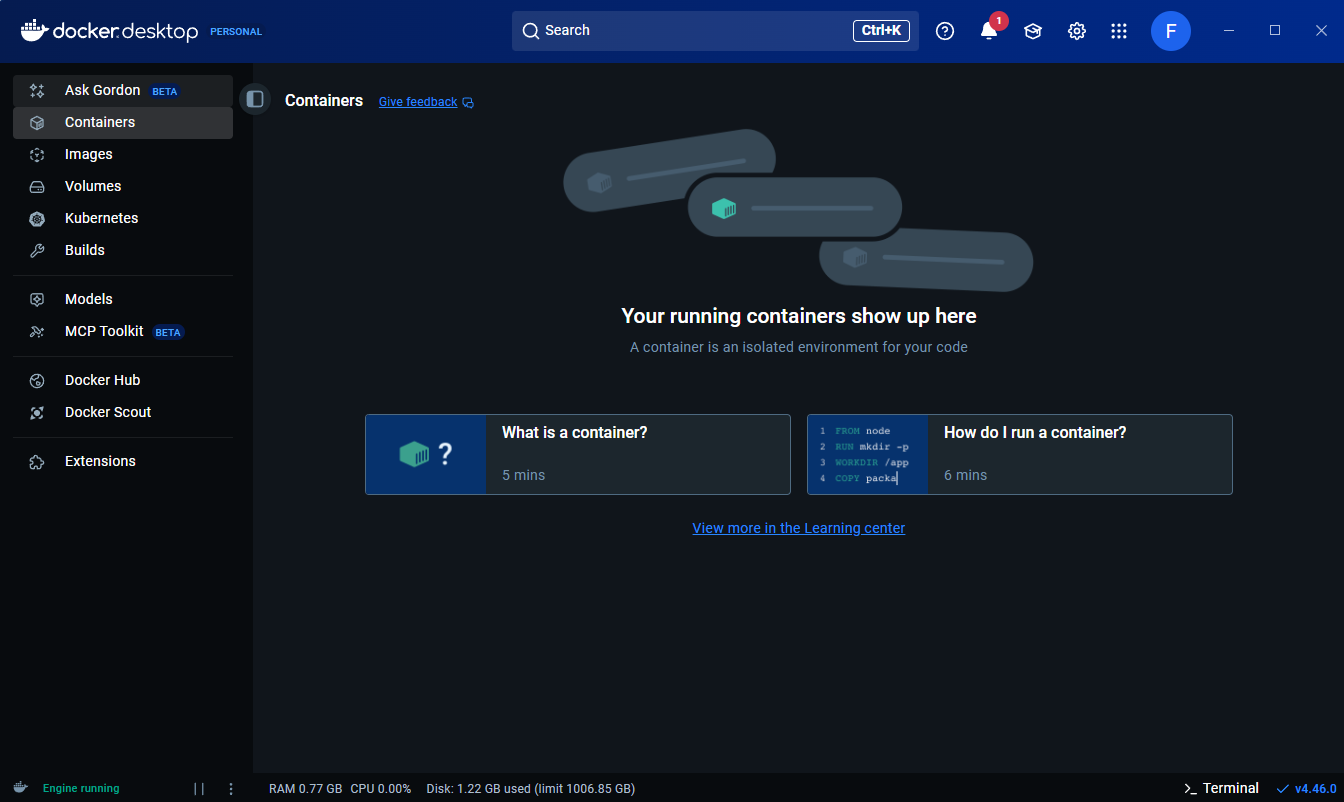
Containers are mostly used during the development phase on an app. For instance, when a team is developing a complex web app, they might use containers to make sure everyone in the team is working in exactly the same environment, even though they are on different machines. That’s why containers are such a popular DevOps tool, a tool for making software development and releases more efficient.

**So what does Docker do?**

Docker helps you create, manage, and deploy these containers efficiently. You can think of Docker as a tool that helps you create the cargo and load them onto ships.



Think of containers like those big shipping containers you see stacked on cargo ships. Each container has everything an app needs to run - its own code, libraries, and settings. You can load a bunch of these containers onto your computer, and they’ll all work smoothly together. That’s because containers package things up so neatly, the app just works wherever you put it. Moving apps around becomes a breeze!

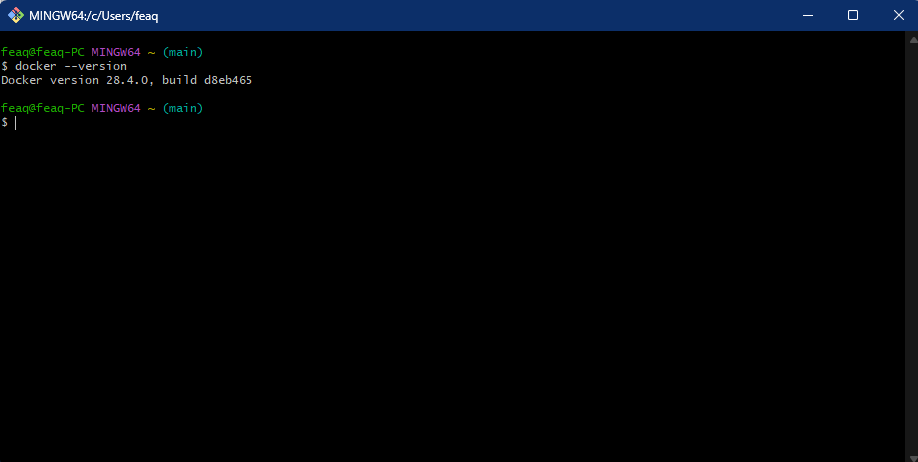


Docker Desktop Dashboard

Docker Desktop lets me manage everything about my containers. I can create new containers, adjust their settings, or monitor how they run.

**Extra for Experts: What are the tabs on the left-hand side of this dashboard?**

* **Ask Gordon** is an AI assistant for Docker questions - handy if you need quick help or tips!
* **Containers** is where you can start, stop, and manage your containers.
* **Images** is where you manage your container templates. We'll cover this in the next step, don't worry if you don't get it now!
* **Volumes** is where you can access data saved in your containers, even after you stop/restart them.
* **Builds** is where you manage images you custom created or loaded into Docker.
* **Models** is where you explore AI models that you can run and manage within Docker.
* **MCP Toolkit** is like an app store for running AI tools (called MCP servers) using containers.
* **Docker Hub** is Docker’s online repository where you can find and share Docker images (i.e. those container templates).
* **Docker Scout** is a feature for analyzing your Docker images for security issues.
* **Extensions** is where you can integrate other tools into Docker, like container monitoring and security.

  
Verified Docker via the command *“docker –version”* in my terminal

  
Verified that Docker Daemon is running in my system

**What is Docker daemon?**

It is a background process that manages the Docker containers on my computer. It takes commands from the Docker client (for e.g., commands I type into the terminal, or clicks I make through the Docker Desktop) and does the heavy lifting of building, running, and distributing my containers.

## Step #2 - Run a Pre-Built Container Image

In this step, I am going to create a container using a container image, a template/blueprint for containers.

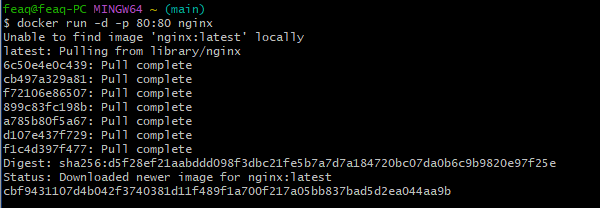
**What is a container image?**

Container image is a blueprint or template for making containers. It tells Docker what exactly to put inside each container, things like my app’s code, the libraries it needs, and any other required files.

You can create as many containers as you want from the same image, and they will all work the same way, no matter where you run them (as long as you have Docker or a similar tool). This consistency is a huge benefit for teams, since everyone can run the same app setup on their own computer without weird, unique issues popping up. We’re less likely to see situations where an app only works for some members in the team and not for others. It also makes it much quicker for new team members to get started since they can spin up the app in just a few steps.

Back in my terminal, I ran the following command:

* *docker run -d -p 80:80 nginx*



***docker run*** starts a new container. I am using a pre-existing container called **nginx** and I am starting this container in detached mode ***(-d)*** so it runs in the background. Then, this ***-p 80:80*** maps port 80 on my host machine to port 80 in the container, which mean I will be able to access the webpage that Nginx is running through my computer’s web browser.

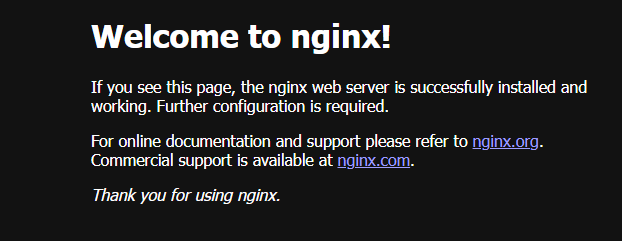
**What is Nginx?**

Nginx (pronounced “engine-x”) is a web server, which is a program I use to run website and web apps. If I want people to visit my site, I am going to need a web server to deliver my website’s files to their browser.

Engineers and developers pick Nginx when they need to manage lots of visitors at once, making sure the site does not slow down when traffic spikes. Nginx is also the web server of choice for containerized apps because it is lighter and uses less memory, compared to other options like Apache.

**Where did we get the Nginx image?**  
If you're wondering where the Nginx image is from, you can find nginx in **Docker Hub** (i.e. Docker's library of public container images) by [clicking here.](https://hub.docker.com/_/nginx) Because Nginx is a pre-existing image, Docker knew where to find it in Docker Hub and downloaded it automatically.

**What is Docker Hub?**  
**Docker Hub** is a **container registry,** which means it's an online library where people can share and find Docker images. AWS also has its own registry called Amazon ECR (Elastic Container Registry).

Further confirmed that I have installed nginx via going into <http://localhost>

How did I end up seeing this Nginx welcome page?

When I type <http://localhost> in my browser, the browser sends a tiny message (called a request) to my own computer asking, “Please show me a webpage.”

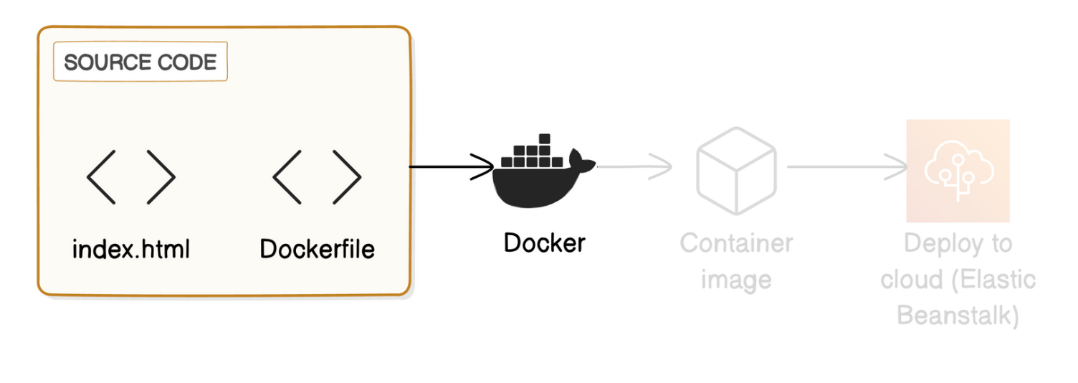
But how does my computer know which webpage to show? That is where ports come in. My PC has tens of thousands of ports, which I can think of as numbered channels that help direct requests to the correct server. Web browsers and web servers use port 80 by default.

1. When I ran *-p 80:80 nginx*, this command connects port 80 on my PC to port 80 in my container (which is running Nginx).
2. Then, when I type localhost into my browser, it sends a request to port 80 of my computer.
3. Because of the connection I have set up, that traffic flows straight into port 80 in my container.
4. My container knows that port 80 requests should be automatically forwarded to its web server (i.e., Nginx), so Nginx receives the request and responds with its “Welcome” webpage.

Notice how I could run an Nginx server and open a webpage in seconds using containers. Doing the same with a virtual machine like Amazon EC2 would take much longer as creating EC2 instance takes more time because I would need to install a full OS and other software to create the web server. In general, containers let me load and deploy applications faster than virtual machines.

## Step #3 - Build my custom image

In this step, I am going to write an instruction sheet or how-to-guide for building my own custom container image. Then I will get Docker to read that file and build a container image with it.



Created a file called Dockerfile via terminal on my project’s directory.

**What is a Dockerfile?**

* It is a document with all the instructions for building my Docker image. Docker would read a Dockerfile to understand how to set up my application’s environment and which software packages it should install.

In the Dockerfile, I added the following lines:

*FROM nginx:latest  
COPY index.html /usr/share/nginx/html/  
EXPOSE 80*

**What do these lines mean?**

The first line, *FROM nginx:latest,* means my image starts as a copy of the latest Nginx image, but I will make a few modifications/additions to it to customize it for what’s needed. Think of it like you copying an existing cheesecake recipe (the Nginx image) and adding your own twist to it (your modifications, specified in the Dockerfile). The modified recipe creates a new dish, but luckily, I did not have to write the entire recipe from scratch.

The next two lines in the Dockerfile customizes the base Nginx image:

1. *COPY index.html /usr/share/nginx/html/* replaces the default HTML file provided by Nginx with my own custom index.html file, so I am customizing the Nginx server to serve my own web content.
2. *EXPOSE 80* means I want the container to receive web traffic through port 80, which makes it easier for users and other services to reach my web app.

Next, I am going to create the web page for my container.

Created an index.html in via my terminal with *vim index.html.*

**What is an HTML file?**

The HTML file determines what my webpage will look like. It’s a document that lays out my webpage’s layout, text, images and links. When Nginx opens my HTML file, it will read the code inside and display it on the user’s browser.

In the vim editor, I added the following HTML content:

<html>

<head>

<title>My Web App</title>

</head>

<body>

<h1>Hello from Syafiq's custom Docker image!</h1>

</body>

</html>

This index.html will show visitors a heading that says “Hello from Syafiq’s custom Docker image”

Next, to build a new container image, run:

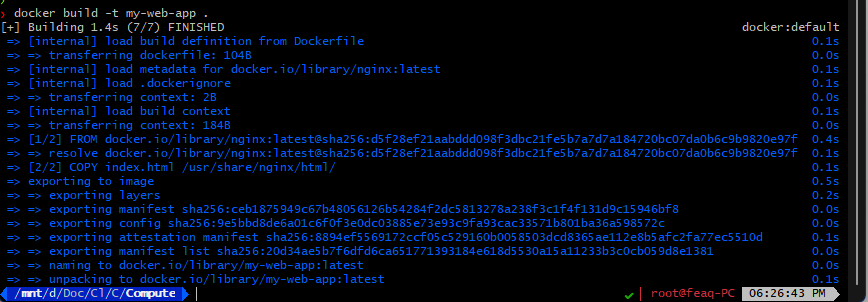
* *docker build -t my-web-app .*

**What does building an image mean?**

Building an image means I am creating a Docker image using the instructions in a Dockerfile. Docker will then use the built image as the blueprint to create containers that run my application anywhere.

**What do these commands do?**

*-t my-web-app* names my image my-web-app, and the . tells Docker to find the Dockerfile in the current directory i.e., the Compute folder.

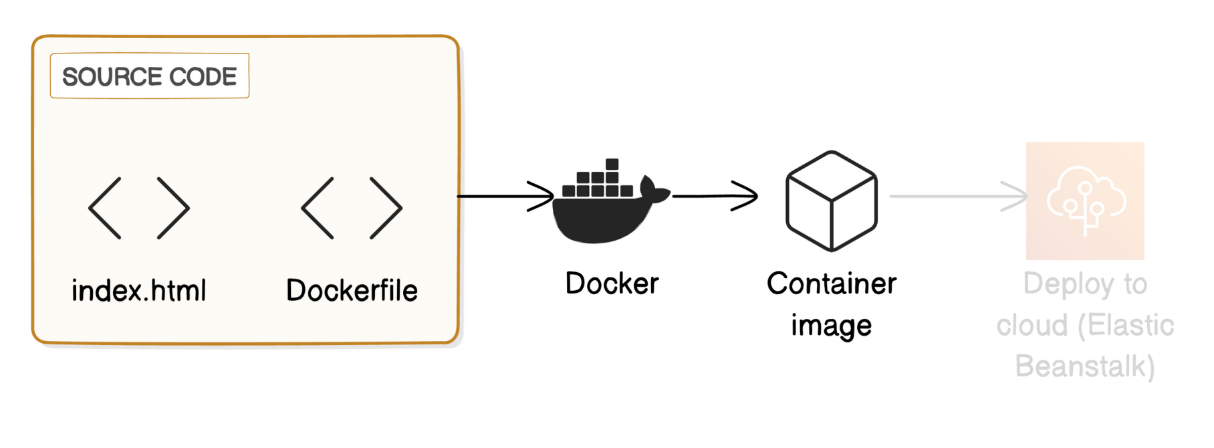
  
The result of the command.

The terminal response shows the steps Docker took to build your image from the Dockerfile:

* Loading build definition: Docker is loading the instructions in your Dockerfile to understand how to build the image.
* [1/2] FROM docker.io...: Docker is processing the first line in your Dockerfile.
* [2/2] COPY index.html...: Docker is processing the second line in your Dockerfile, which tells it to copy your index.html file into the container's file system.
* Exporting to image: Docker is combining all the instructions applied to the image together, making it ready to run containers.

## Step #4 - Run a container with my custom image.

In this step, I am going to use the custom container image that I have built to create containers based on that image.



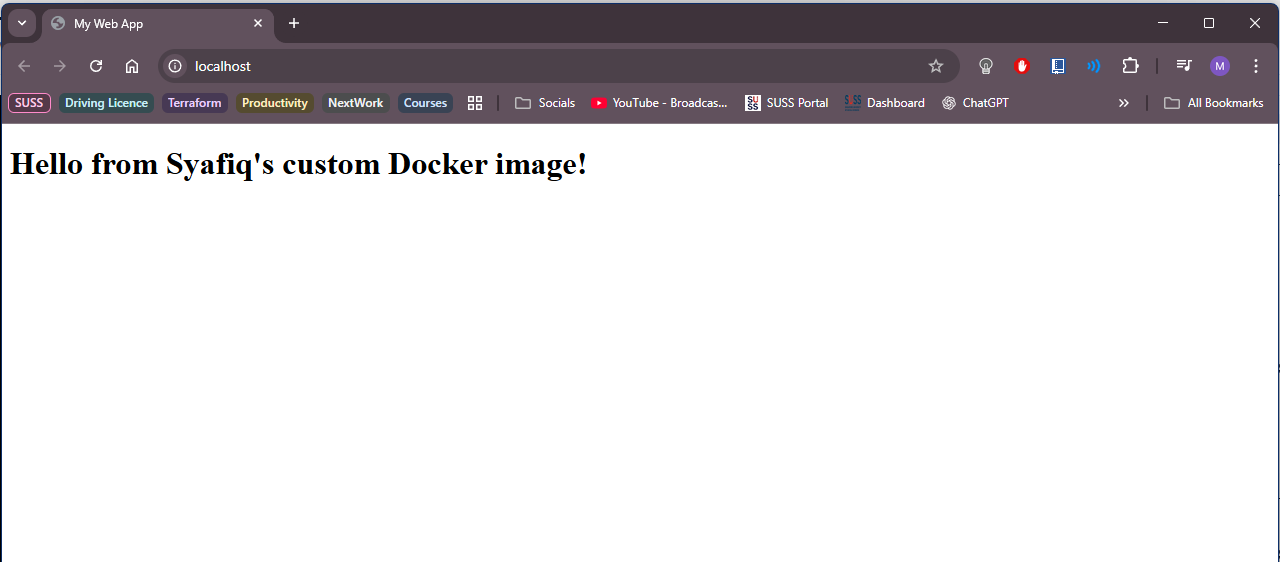
To run the image, I ran this command:

* *docker run -d -p 80:80 my-web-app*

**What does this command do?**

Similar to the command I used before, I am asking Docker to run my-web-app image as a container in the background (-d), making it accessible through port 80 on my local machine (-p 80:80).

There was an error where my localhost page is still showing Nginx web page despite port 80 being my-web-app. All I had to was a hard refresh in the browser via Ctrl – Shift – R to clear the cache that was stored in my browser.



**Difference between containers and container images.**

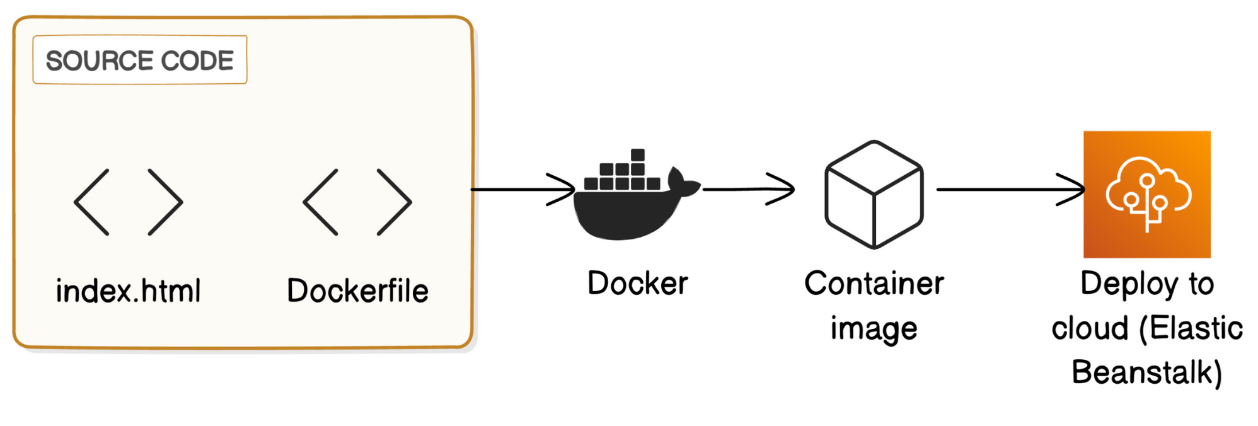
* in this example, the container image is the **blueprint** that tells Docker the application code, dependencies, libraries, etc., that should go into a container. The container is the actual **software** that’s created from this image and running the web server displaying my index.html.

## Step #5 – Deploy my custom image to Elastic Beanstalk

Next, I will deploy my container app to AWS cloud. I will need to log in to my AWS with my IAM user account.

In this step, I will deploy my custom Docker image to AWS Elastic Beanstalk, a service that makes it easy for developers to deploy applications in the cloud.

This is the grand finale for the project – complete the application development process by making my app public on the internet.



Seach and click on Elastic Beanstalk service via my AWS Console once logged in.

**What is Elastic Beanstalk?**

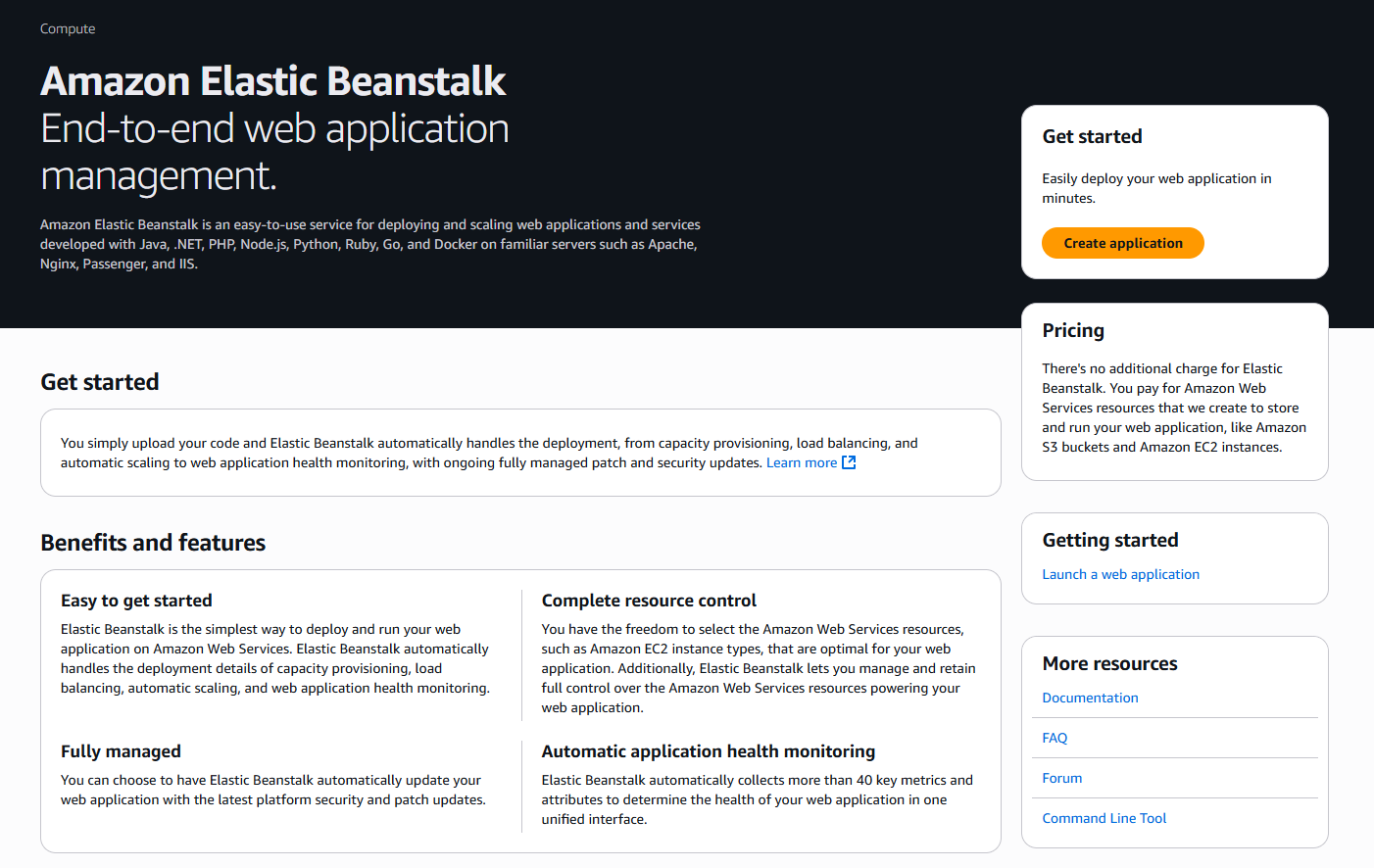
**AWS Elastic Beanstalk** is a service that makes it easy to deploy cloud application without worrying about the underlying infrastructure, basically a Platform-as-a-Service (PaaS). I will simply need to upload my code and Elastic Beanstalk handles everything needed to get it running, like setting up servers and managing scaling. This lets me focus on my application code without having to spend too much time on managing cloud infra.

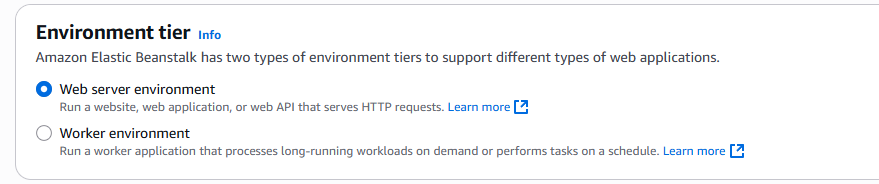
In more technical terms, Elastic Beanstalk handle things like:

* Capacity provisioning
* Load balancing
* Auto Scaling
* Application health monitoring

**How is Elastic Beanstalk related to containers and Docker?**

Elastic Beanstalk is AWS’s way of running my Docker containers without the setup headaches. Build and test my app locally, wrap it into a Docker image, then hand that to Elastic Beanstalk. It spins up the servers, launches my container, and keeps everything running for me – no extra config needed.

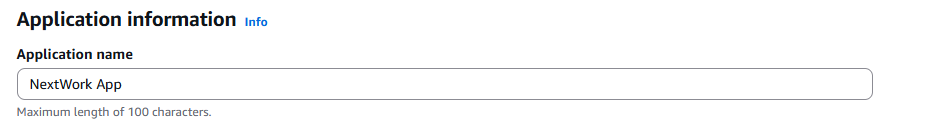
  
Clicked on Create Application

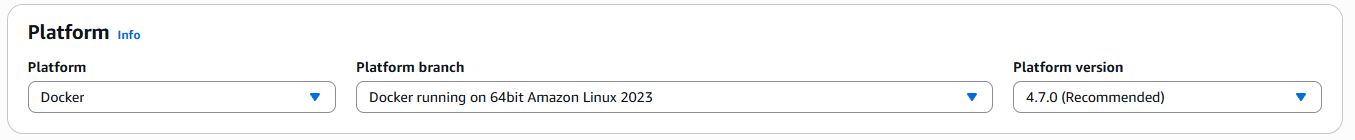
  
Left it as default.

An environment in EB is like the settings for the virtual machines that will run my application.

When I configure my environment, I am deciding the instance type (the type of EC2 instance that will be running the container), scaling options (how many instances to run), network settings, and more.

Elastic Beanstalk (EB) uses EC2 instance (virtual servers) under the hood to run my app. Even though I am deploying a Docker container, it ultimately runs on an EC2 instance managed by EB.

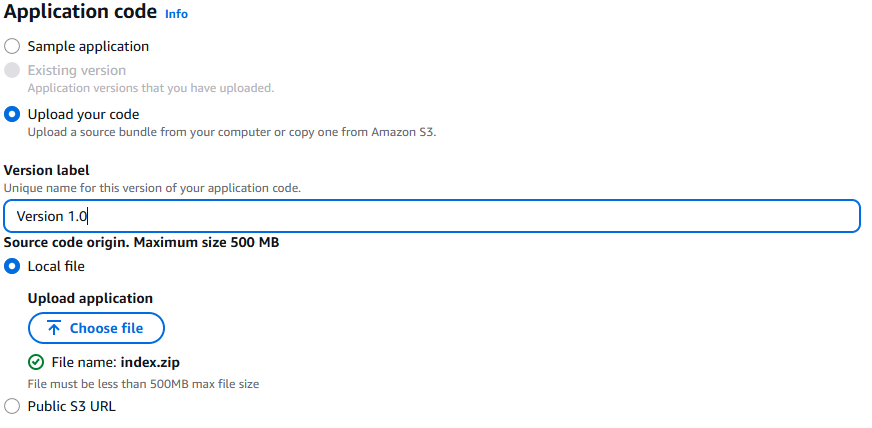
  
Named my application ‘NextWork App’

  
Platform will be Docker since I am working with Docker from the beginning.

**What does platform mean?**

The platform is the technology that will open and run my application. I chose Docker because it’s the best platform for deploying container images built with Docker.

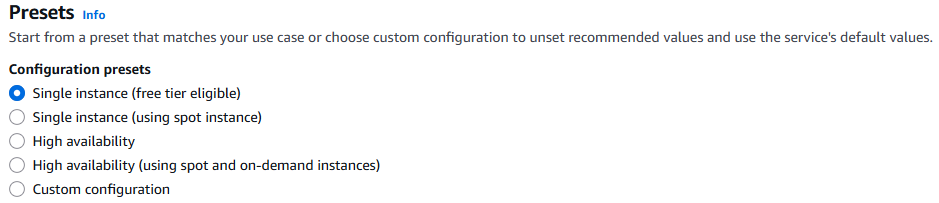
* **Platform branch** is the type of Docker I am using e.g., the default **Docker running on 64bit Amazon Linux 2023** means I am using a type of Docker that is optimized to be used on an Amazon Linux 2023 operating system.
* **Platform version** is the specific version of Docker I’m using within that branch.

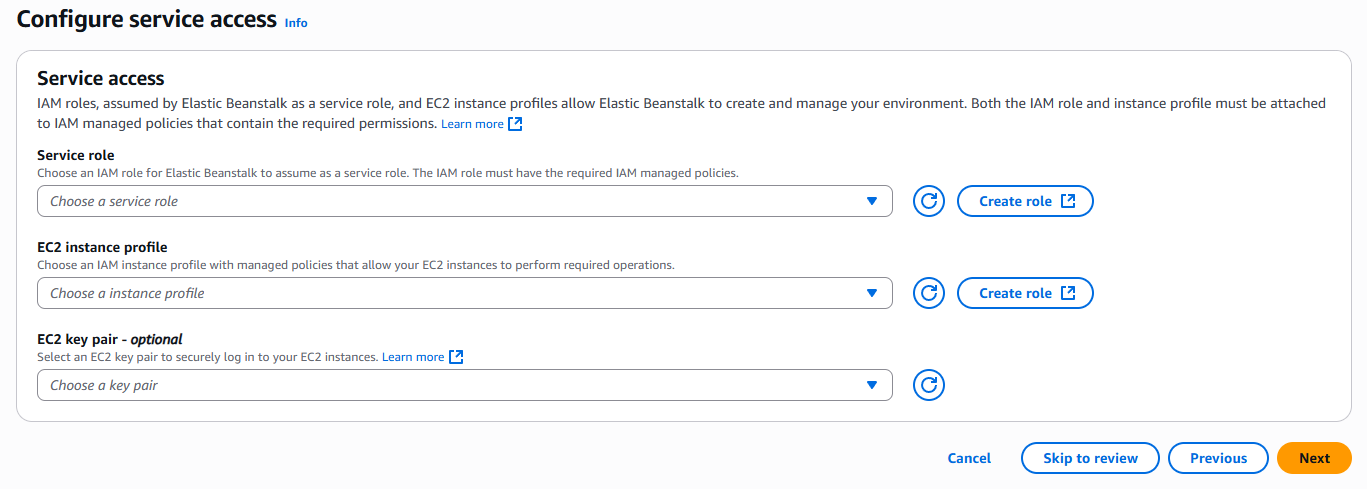
  
In the **Application code** section, selected **Upload your code.**

To deploy my code, Elastic Beanstalk needs my source code to be in a ZIP file. So in my folder, I selected both Dockerfile and index.html and proceed to compress them together, hence the file name “index.zip” in the above image.

Before proceeding to upload, I have edited my index.html with a line that says “This is the updated version in EBS”, to ensure that it is working as intended. Also labelled the **Version label** as ‘Version 1.0’.

**Make sure not to select and compress the whole folder/directory. EB would get confused if it opens the zip file and finds a subfolder inside, instead of the files themselves. Make sure to select the two files and compress them at the file level.**

  
Selected Single instance as this is great for testing and is free tier eligible.



In the **Configure service access,** I selected **Create role** under the **Service role** heading.

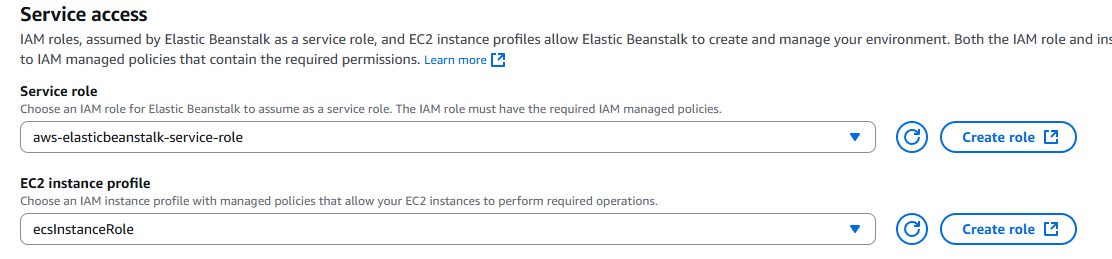
This opens up a new tab with AWS IAM. Once I create a new role on this tab, I can head back to my EB setup to use it.

**What is a service role?**

It is an AWS IAM role that lets Elastic Beanstalk permission to call other AWS services on my behalf.

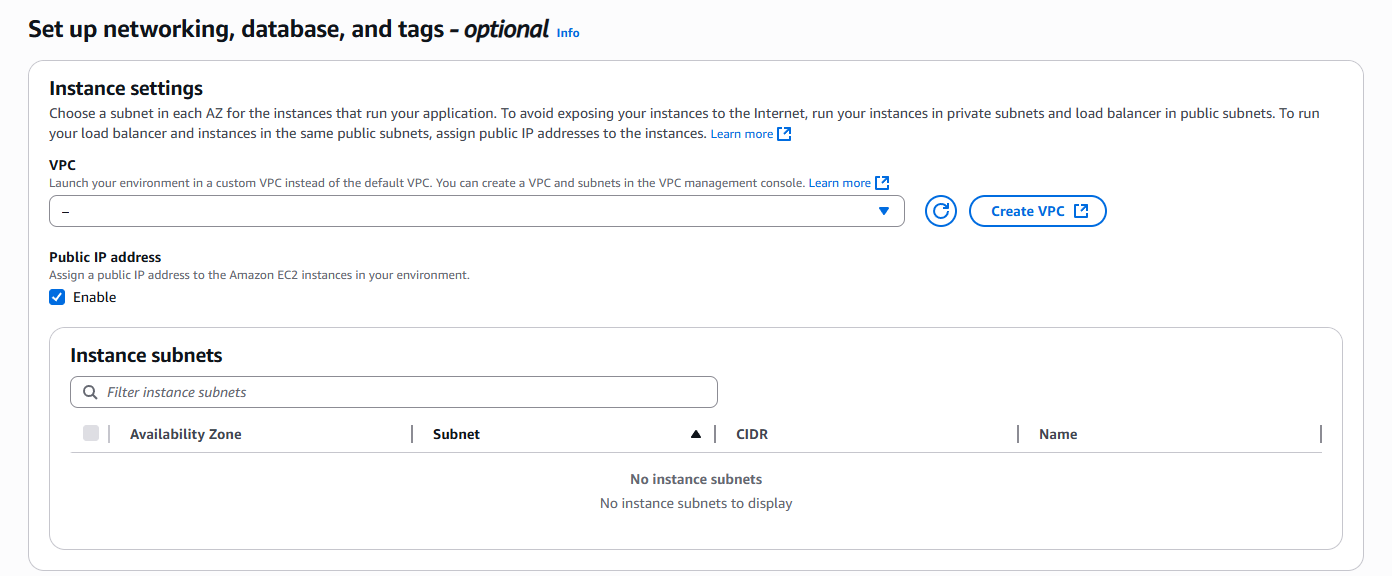
**Why does Elastic Beanstalk need a service role?**

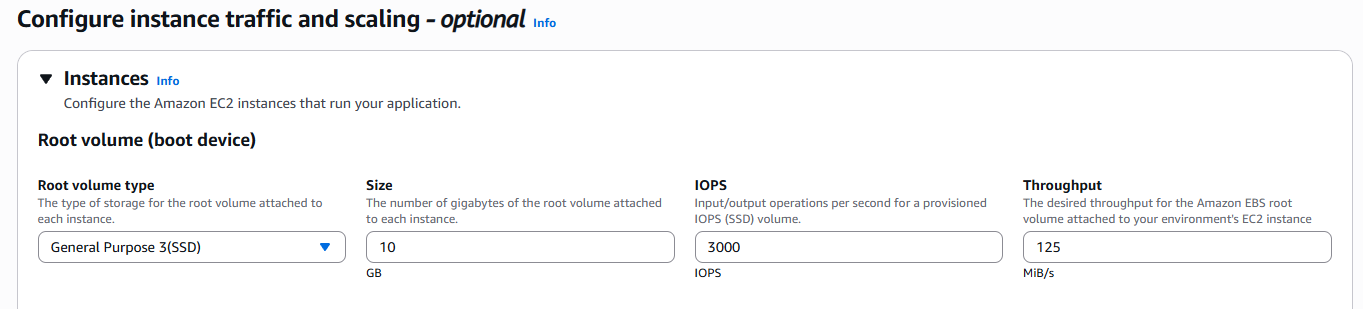
One of the benefits of Elastic Beanstalk is that it can handle your application's cloud infrastructure for you, and it can only do that if it has the permission to use my AWS resources automatically e.g. creating a new EC2 instance. It also helps with automating tasks like scaling and health monitoring!



In the IAM console, I used all the default settings that AWS has set for me to create the role.

And the same goes for EC2 instance profile role.

  
In the setup networking page, I enabled **Public IP address**. This makes my EC2 instance and application accessible from the internet.

  
For the **Configure instance traffic and scaling** page, I selected **General Purpose 3 (SSD)** for the root vol type.

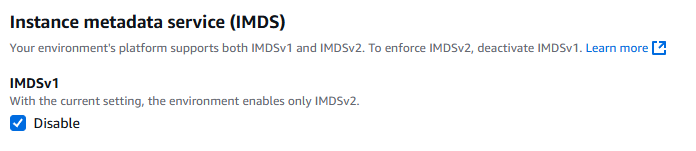
**What does root volume type mean?**

The root volume is my EC2 instance’s storage space, just like how my local computer would have its own storage space for the operating system and files I keep locally.

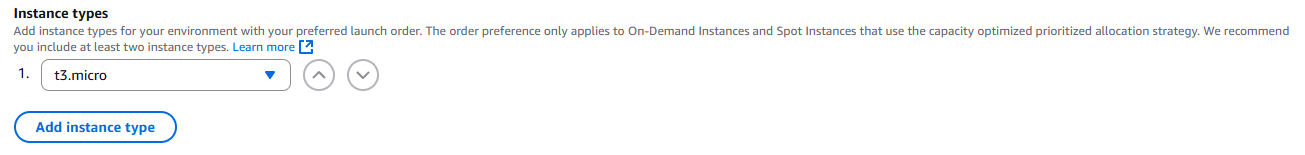
**General Purpose 3 (SSD)** is Free Tier eligible and is fit for most application.

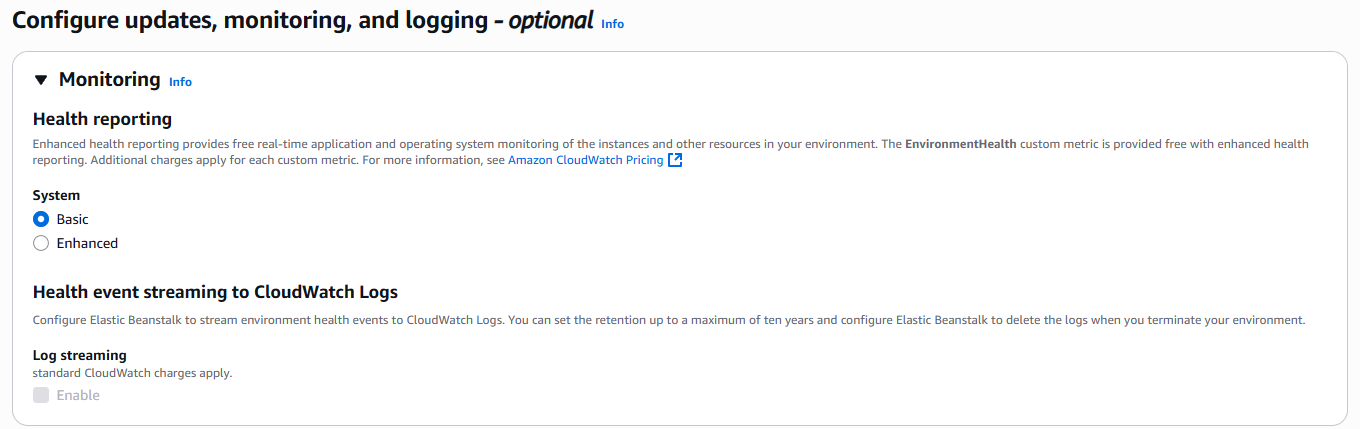
Other root volume types are fit for special use cases, like **Provisioned IOPS SSD** for applications with high-performance databases, or **Magnetic** for data in long-term storage.

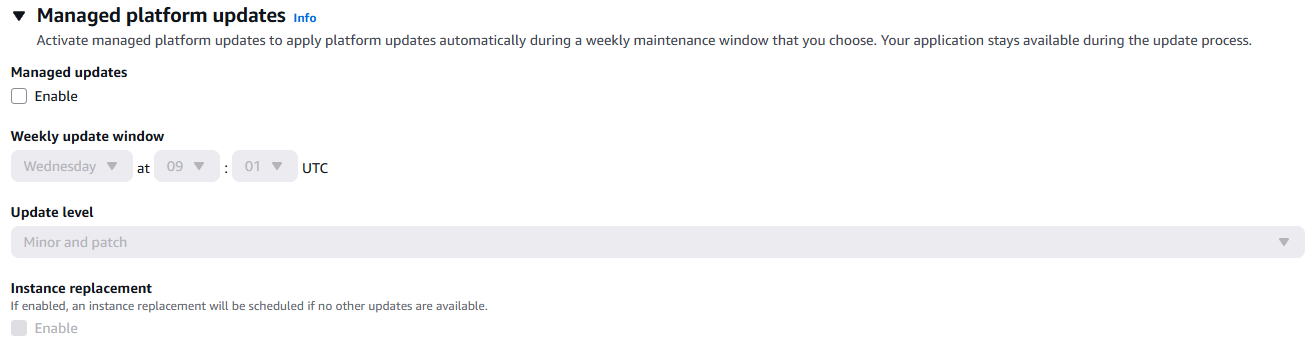
Everything else is defaulted for the root vol.

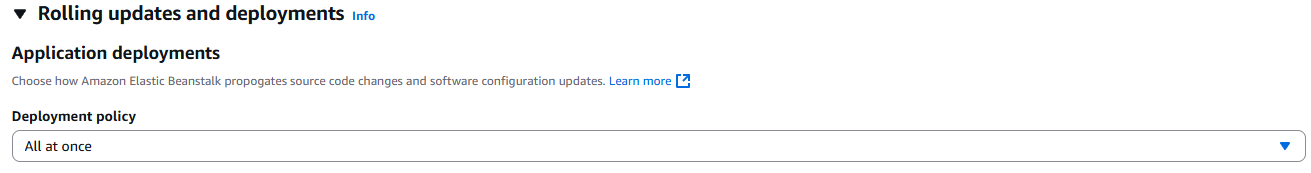
  
Ensured this is disabled.

IMDS is a service that gives application temporary credentials to my EC2 instances. For example, I can use IMDS to give my application the permission to access S3 to get data. Since my app would not need any access to other AWS services, I won’t need IMDS.

  
Under **Capacity,** removed t3.small from the **Instance types** to ensure this stays free. Everything else is default.

  
In the **Configure updates, monitoring, and logging** page, I switched to Basic to keep my project free.

  
Unticked **Enable** in the **Managed updates section**.

  
Left this as default.

**What are application deployments?**

Application deployments move a new version of my software into production on Elastic Beanstalk. This includes uploading my code, setting configuration options, and finally, replacing the old version with the new one on the live environment that users see.

There are two main types of deployments, also called deployment policies:

* **All at once** updates all instances at the same time, resulting in a brief period of downtime.
* **Rolling** updates instances gradually, which minimizes downtime.

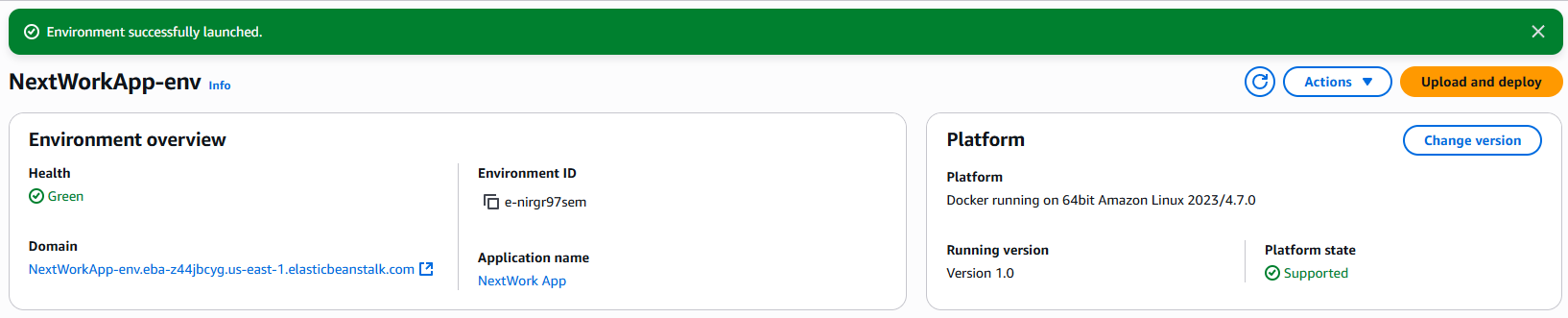
Left all other options, including the **Platform software** section as default.

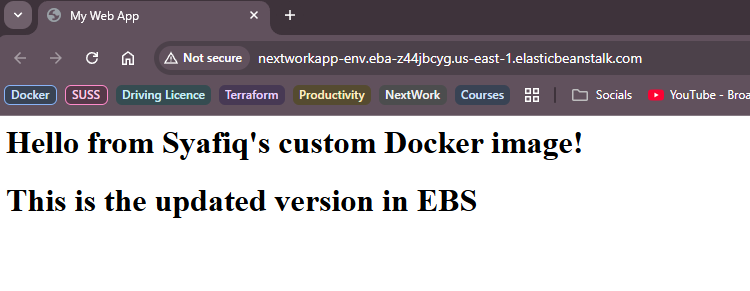
For the final step, I will review the Elastic Beanstalk setup. This rounds up my selections across the last five pages.

Quick checklist and scavenger hunt. Is...

* **Application name** NextWork App?
* **Application code** a zip file?
* **EC2 instance profile** set to **ecsInstanceRole**?
* **Public IP address** set to true?
* **Root volume type** set to gp3?
* **Environment type** set to Single instance?
* **System** set to basic?
* **Managed updates** set to Disabled?
* **Deployment policy** set to AllAtOnce?

Proceed to **Create.**

  
Environment successfully launched. Took about 5mins.

****This is the result as you can see in the domain name.

## Step #6 – Deploy an App update

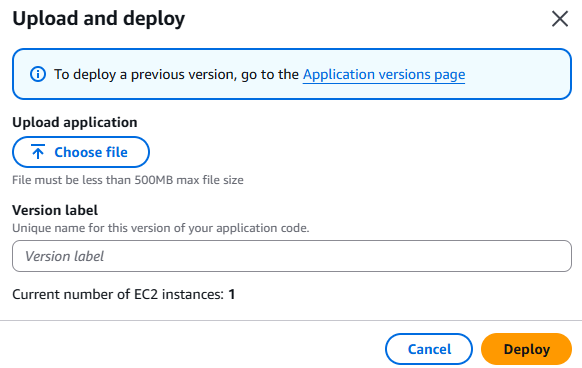
I accidentally spelled the acronym of Elastic Beanstalk as EBS instead of EB since EBS stands for Elastic Block Store in AWS. Hence, I will need to do a quick app update via AWS CloudShell.

Went to my index.html file to change the typo via my PC’s terminal. (nano index.html).

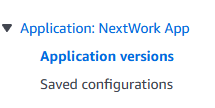
Once done, in my local host terminal, used the command:

* *‘zip my-eb-app.zip Dockerfile index.html’* to compress both the files into a compressed folder called ‘my-eb-app.zip’
* Had to install zip in my terminal with the command: ‘sudo apt install -y zip’, since the initial command did not work.

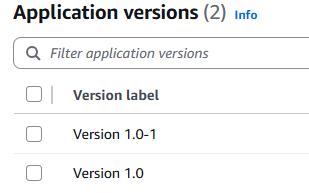
On the Elastic Beanstalk console page, I uploaded the new version.

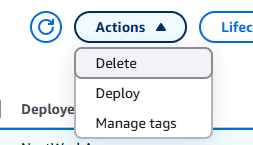


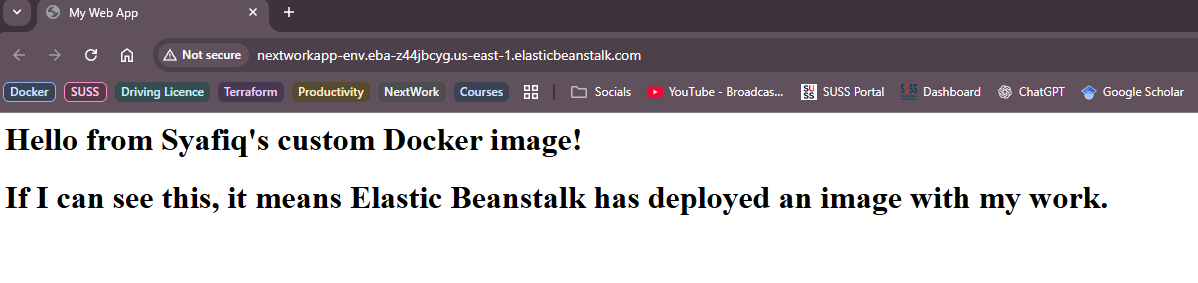
Once uploaded, I have to change the version to the latest one which I named it Version 1.0-1 (I forgot to rename it to 1.1 so it auto added -1).



On the left sidebar, I clicked on **Application versions**. Here, I can view all the versions that are currently available.



  
Select the new version and deploy.

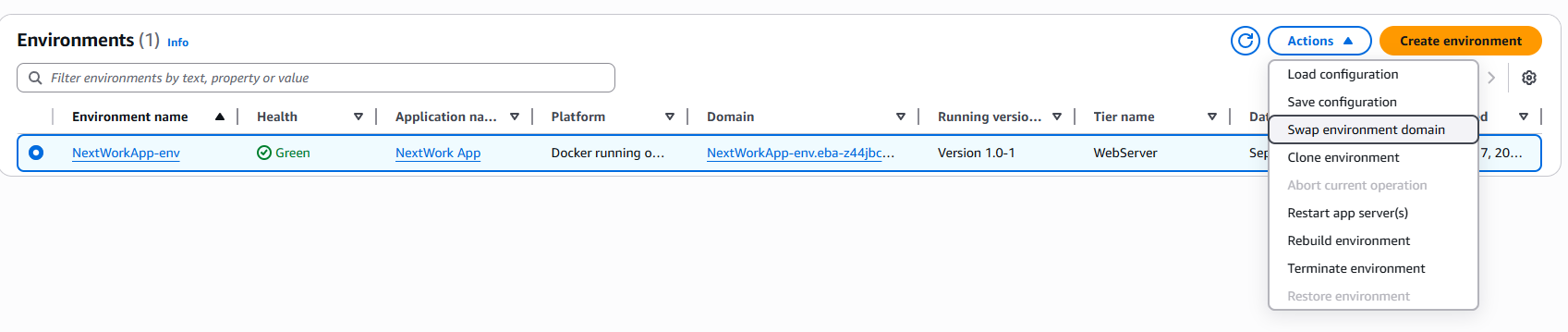
  
Done, hard refresh my web browser to view the updated web app.

## Delete Resources!

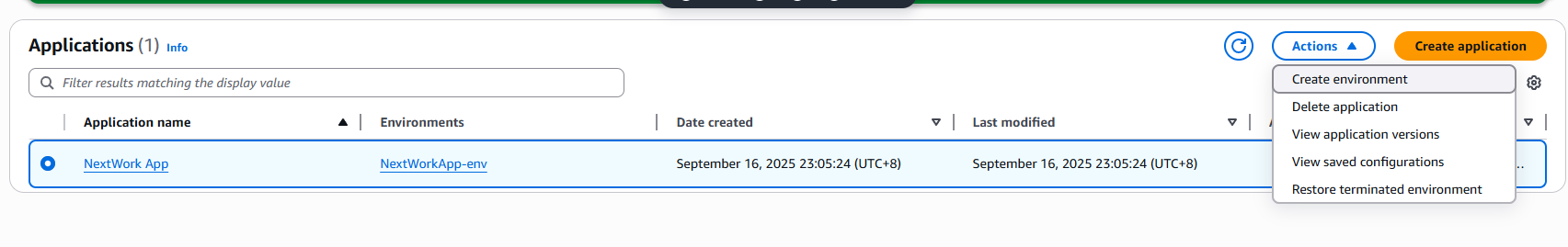
IMPORTANT STEP to make sure I delete all my resources when done to avoid unexpected charges!

**Services to delete/terminate:**

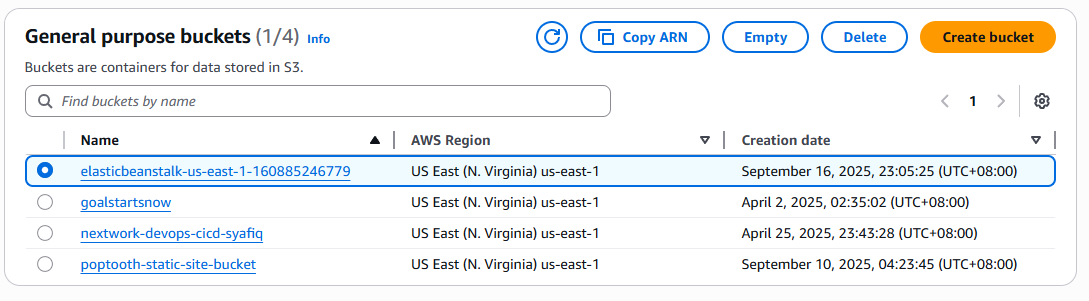
**Elastic Beanstalk Environment: (THIS HAS TO BE THE FIRST STEP BEFORE APPLICATION!)**



**Application**

****

**Stored files in S3 bucket**

****

**Containers and container images** (don’t cost any money to store containers in Docker, but it’s best practice to remove resources that I won’t need anymore).

## Summary

Today you've learned how to:

* 🐳 **Install and run Docker:** Installed Docker on my local machine, which lets me create, manage, and run containers for my applications.
* 🔨 **Build a custom Docker image:** Used a Dockerfile to define a Docker image that packages up my index.html file.
* ☁️ **Deploy your image to AWS Elastic Beanstalk:** Uploaded my custom Docker image to AWS Elastic Beanstalk, which served my application in the cloud so I can access it from the internet!

