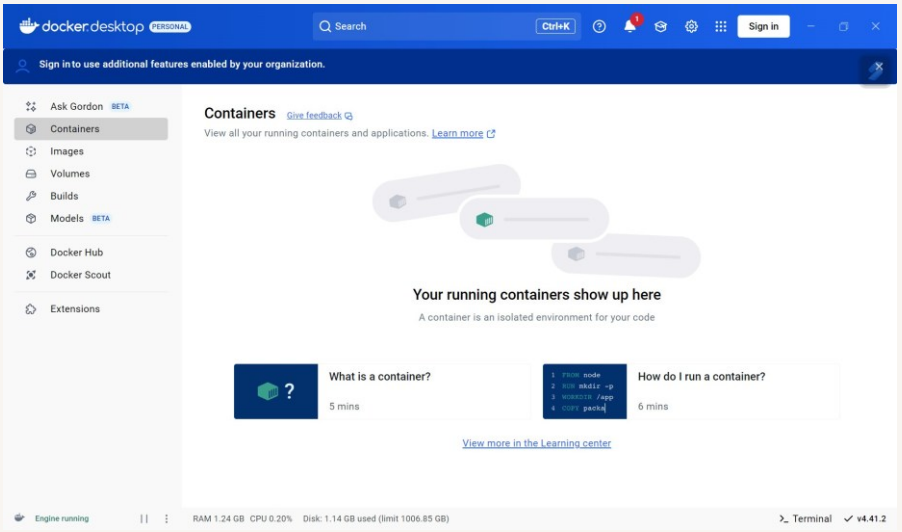
Darpan Patel

Deploy an App with Docker



Darpan Patel

Introducing Today's Project!

What is Docker?

Docker is an open-source platform that enables developers to package applications and their dependencies into containers—lightweight, portable, and consistent environments that run the same regardless of the underlying system. Containers make it easy to develop, ship, and run applications reliably across different computing environments. In today's project, I used Docker to: Create a Dockerfile, which described how to build a custom image for my web app. Package my HTML app into a Docker container, ensuring that it runs consistently across environments. Build and test the Docker image locally using Docker Desktop. Deploy the image to the cloud using AWS Elastic Beanstalk, which automatically handled provisioning the infrastructure and running the container. This project demonstrated how Docker simplifies deployment, and how powerful it can be when combined with AWS for scalable cloud hosting.

One thing I didn't expect...

One thing I didn't expect in this project was how easy it is to update a deployed app using Elastic Beanstalk. Initially, I assumed that every change would require reconfiguring the entire environment or redeploying everything from scratch. But once the environment was set up, it was surprisingly simple to just zip the updated files, upload the new version, and deploy it with just a few clicks. This made the development cycle feel smooth and efficient—great for quick iterations and testing changes live!

This project took me...

This project took me approximately 2 hours to complete from start to finish. The time included: Setting up the Elastic Beanstalk environment Writing and testing the initial index.html and Dockerfile Packaging and deploying the first version Updating the HTML with a new image Creating a new ZIP file and deploying the updated version Some extra time went into troubleshooting initial errors (like missing Dockerfile or

incorrect packaging), but overall, it was a hands-on way to understand how containerized apps are deployed with Elastic Beanstalk.

Understanding Containers and Docker

Containers

In this step, I will download and install Docker Desktop because it provides a userfriendly interface and environment to create, manage, and run containers on my local machine. Installing Docker Desktop is essential to work with containers, which package applications and their dependencies into a portable, consistent unit that can run anywhere. This solves the “it works on my machine” problem by ensuring the app runs the same way regardless of the underlying system. Docker acts like a tool to build and ship these containers efficiently, making it a fundamental piece in modern software development and deployment workflows.

A container image is a lightweight, standalone, and executable package that includes everything needed to run a piece of software—such as the application code, runtime, system tools, libraries, and dependencies. It serves as a blueprint for creating containers, ensuring that applications run consistently across different environments. When you launch a container from an image, you're essentially creating an isolated environment that behaves exactly the same no matter where it's deployed—on a developer's laptop, a server, or in the cloud. This makes container images essential for reliable software deployment and development collaboration.

Docker

Docker is a platform that allows you to package applications and all their dependencies into a lightweight, portable unit called a **container**. These containers ensure your app runs the same regardless of where it’s deployed—eliminating the classic "it works on my machine" problem.

Darpan Patel

Docker helps developers build, ship, and run applications efficiently across different environments. Docker Desktop is a **user-friendly application** that runs Docker on your local machine. It provides a graphical interface to manage containers, images, volumes, and networks without needing to rely entirely on the command line. It's the main way developers interact with Docker when working on Mac, Windows, or Linux, and is especially useful for local development and testing.

The Docker daemon is a **background process** that runs on your computer and is responsible for **building, running, and managing Docker containers**. It listens for Docker commands (like `docker run` or `docker build`) from the Docker client—whether you enter those commands in the terminal or click buttons in Docker Desktop —and then executes them behind the scenes. More technically, the Docker daemon: **Creates and manages containers and images**. **Handles container networking and storage**. Communicates with Docker registries (like Docker Hub) to download or push images. Ensures the containers behave consistently, even across different systems. Without the Docker daemon running, Docker simply won't work—it's the engine that powers everything.

Running an Nginx Image

Nginx is a high-performance, open-source web server that can also be used as a reverse proxy, load balancer, and HTTP cache. It is known for its ability to handle a large number of simultaneous connections efficiently, making it a popular choice for serving websites, APIs, and other web-based applications. Originally designed to solve the C10K problem (handling 10,000 concurrent connections), Nginx is often used by developers and companies to: Serve static web content (like HTML, CSS, and JavaScript files), Act as a **reverse proxy** to route client requests to backend servers, Load balance traffic across multiple servers for improved performance and reliability, Handle HTTPS and SSL/TLS termination. Because of its lightweight design and scalability, Nginx is widely used in modern web infrastructure and often deployed in containerized environments like Docker.

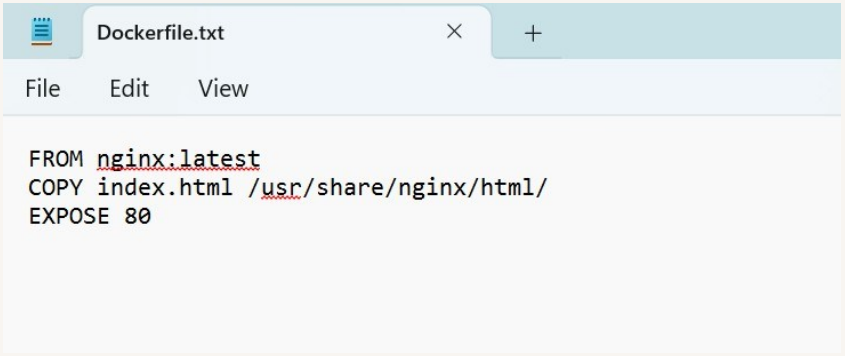
The command I ran to start a new container was: docker run -d -p 80:80 nginx This command tells Docker to run a new container using the pre-built nginx image, in detached mode (-d, so it runs in the background), and maps port 80 on the host machine to port 80 inside the container (-p 80:80), allowing you to access the Nginx web server through your browser.

Creating a Custom Image

The Dockerfile is a text file that contains a list of instructions for Docker to follow in order to build a custom container image. It defines the base image to use, any additional files to include (like `index.html`), configurations such as exposed ports, and other setup steps needed to prepare a containerized environment for an application.

My Dockerfile tells Docker three things: 1. `FROM nginx:latest` – Start by using the latest official Nginx image as the base for the new image. This provides a preconfigured web server environment. 2. `COPY index.html /usr/share/nginx/html/` – Copy my custom `index.html` file from my local folder into the Nginx web server's default HTML directory, so Nginx will serve my custom webpage. 3. `EXPOSE 80` – Indicate that the container will listen for web traffic on port 80, which is the standard port for HTTP, making it accessible for web browsers and users.

The command I used to build a custom image with my Dockerfile was: docker build -t my-web-app . The . at the end of the command means Docker should look for the Dockerfile and other necessary files in the current directory (in this case, the Compute folder). Docker reads the instructions in the Dockerfile to create a custom image, which includes the base Nginx image and adds the custom index.html file that I provided.

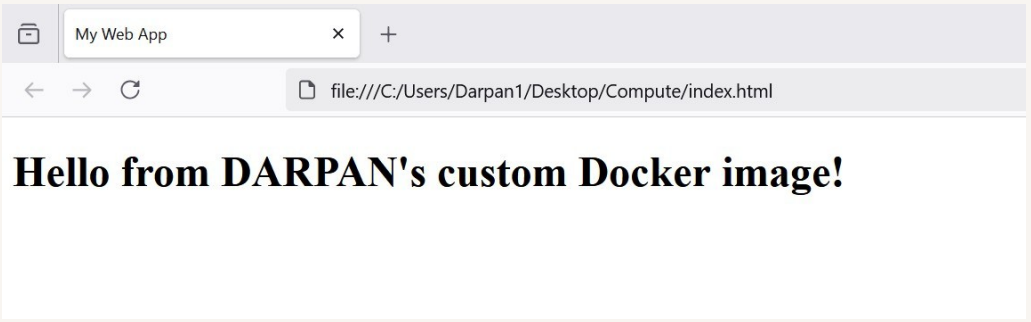


Running My Custom Image

There was an error when I ran my custom image because port 80 was already being used by another container — likely the default Nginx container I ran earlier. I resolved this by identifying the running container using `docker ps --filter "publish=80"` to find the container ID, and then stopping it with `docker stop <container_id>`. After that, I was able to successfully run my custom image without any errors.

In this example, the container image is the blueprint that contains all the instructions and files needed to create the web server environment—including the Nginx base image and your custom `index.html` file. The container is the actual running instance created from that image, which is the live web server serving your webpage at `http://localhost`.

Darpan Patel



Elastic Beanstalk

Elastic Beanstalk is an AWS service that simplifies the process of deploying and managing applications in the cloud. It automatically handles the infrastructure setup, such as provisioning servers, load balancing, auto-scaling, and monitoring the health of your application. You just upload your code—or in this case, your Docker container—and Elastic Beanstalk takes care of the rest, allowing you to focus on development instead of cloud operations.

Deploying my custom image with Elastic Beanstalk took me about **15 to 20 minutes** from start to finish. This included: Creating the `Dockerfile` and `index.html` Packaging everything into a ZIP file Uploading the ZIP to Elastic Beanstalk Configuring the environment (instance settings, IAM roles, and network) Waiting for the environment to launch and deploy the app Most of the time was spent waiting for AWS to provision resources and complete the deployment. Once deployed, accessing the application via the provided public URL worked immediately.

Darpan Patel

Deploying App Updates

To learn how to deploy app updates with Elastic Beanstalk, I updated my app by modifying the `index.html` file to include a new subheading and an image below the existing `<h1>` tag. I added an `<h2>` element with the text “And here's a special image chosen by me:” followed by an `<img>` tag that displays a custom image hosted online. This enhancement was intended to make the app more personalized and visually engaging. I verified those changes by first opening the updated HTML file in my local browser to ensure everything rendered properly. Once confirmed, I rebuilt the Docker image, created a new ZIP bundle with the updated files, and deployed it through the Elastic Beanstalk dashboard. After deployment, I refreshed the web page at the Elastic Beanstalk environment URL and confirmed that the subheading and image appeared live, confirming a successful update and deployment process. This helped me understand how Elastic Beanstalk handles updated Docker containers.

My app updates didn't show up in my live environment immediately because
Elastic Beanstalk was still running the old version of my app using the original ZIP file I had uploaded. The changes I made to `index.html` were only saved locally and not yet deployed to the cloud. **To deploy my changes, I only had to** delete the previous ZIP file, compress the updated `index.html` and `Dockerfile` into a new ZIP archive, and then use the **Upload and Deploy** option in the Elastic Beanstalk console. I gave this new version a label (e.g., "Version Two") and clicked **Deploy**. After waiting a few minutes, my changes—including the new subheading and image—were live and visible on my deployed web app.

Darpan Patel