Deploying A Web Application Using Docker and AWS Elastic Beanstalk

Table of Contents:

1. Project Overview

In the modern software development landscape, containerization has become an essential practice for ensuring seamless application deployment across different environments. Docker, a leading containerization platform, simplifies packaging applications and their dependencies into isolated containers. In this project, we explore how to deploy a web application using Docker and AWS Elastic Beanstalk. This step-by-step guide provides a clear and professional approach to containerization and cloud deployment, making it easier for developers to implement similar workflows in their projects.

2. Overview of Docker and Containers

What is Docker?

Docker is a platform that enables developers to build, ship, and run applications inside lightweight, portable containers. It eliminates environment inconsistencies by encapsulating application dependencies within a single unit.

What are Containers?

Containers provide isolated environments where applications run along with their dependencies, ensuring consistent performance across different systems. Unlike traditional virtual machines, containers share the host OS kernel, making them more efficient in terms of resource utilization.

3. Docker Desktop Overview

Docker Desktop is a user-friendly application that allows developers to build, test, and run containerized applications on their local machine. It provides an easy way to manage Docker containers, images, networks, and volumes through a graphical interface while also supporting command-line interactions.

Why Use Docker Desktop?

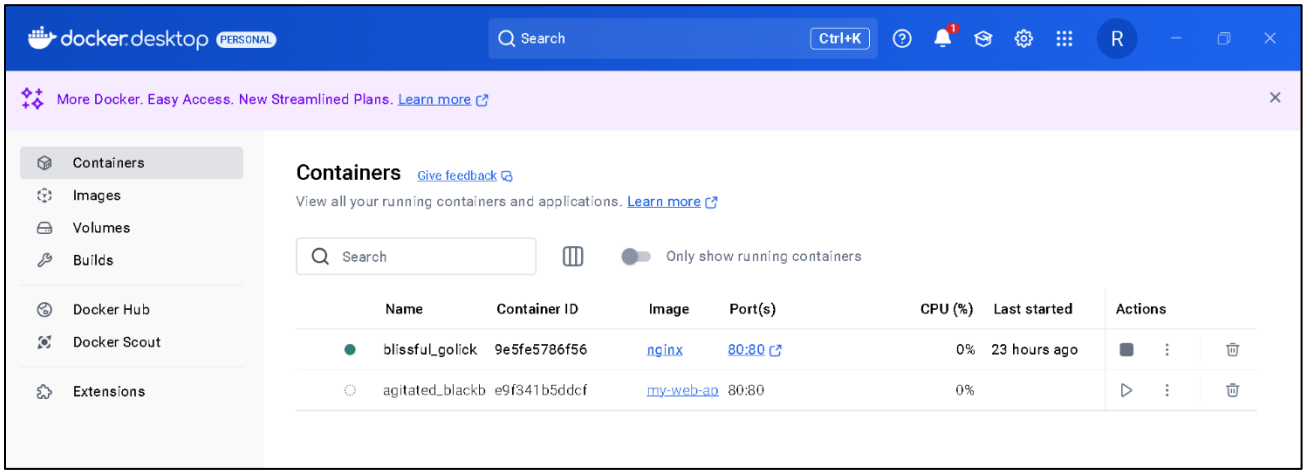
- Provides a lightweight virtualized environment to run containers.
- Simplifies local development and testing before deploying to production.
- Includes an intuitive dashboard to monitor running containers and manage images.
- Ensures consistency across different environments (local, staging, production).

How Docker Desktop Fits into This Project

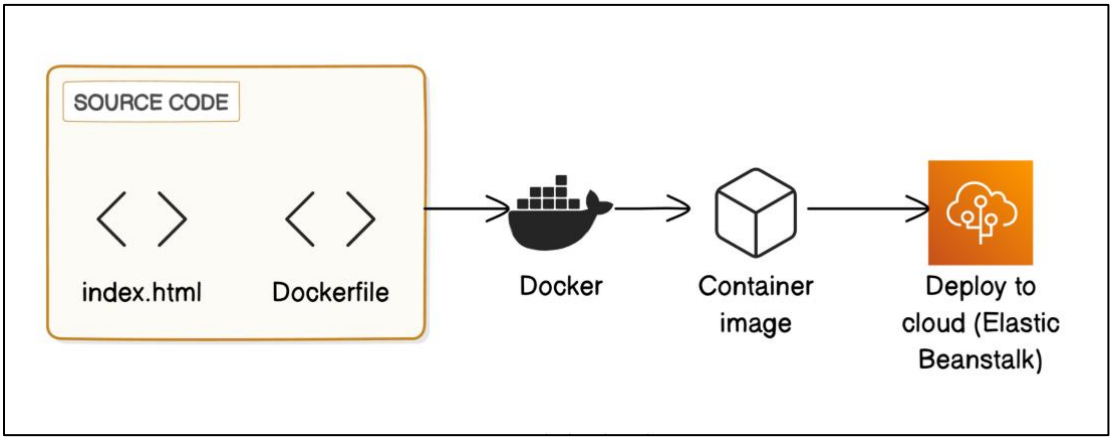
In this project, **Docker Desktop** is used to:

1. Build and test containerized applications locally before deploying them to AWS Elastic Beanstalk.
2. Run an **Nginx container** and verify its functionality on the local machine.
3. Create and manage custom Docker images that will be deployed to AWS.

Below is a simple diagram illustrating how Docker Desktop is used in the development process:



4. Architecture Diagram



5. Setting Up the Environment

Before proceeding with deployment, ensure the following tools are installed on your system:

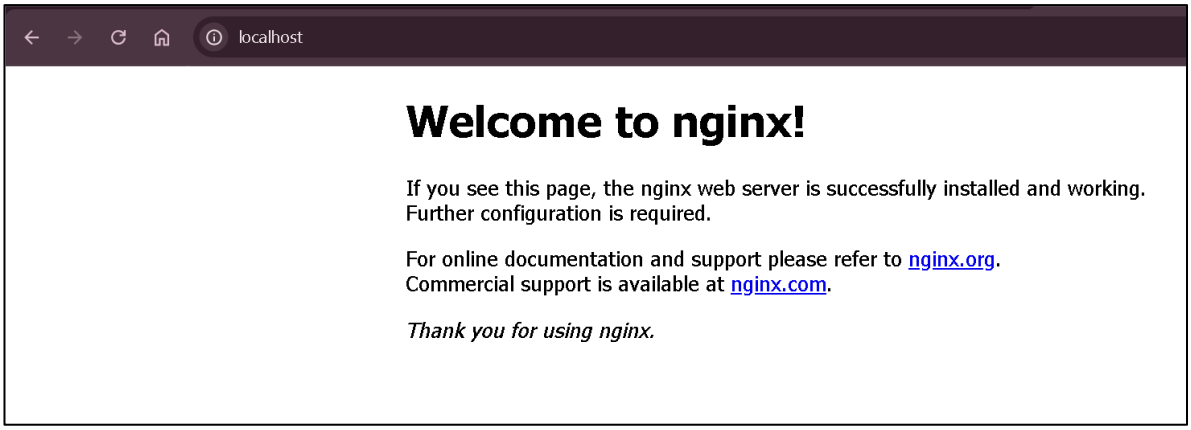
- **Docker Desktop** – To build and run containers locally.
- **AWS CLI** – For interacting with AWS services.
- **Elastic Beanstalk CLI** – To deploy applications to AWS Elastic Beanstalk.

6. Running an Nginx Image

Nginx is a high-performance, open-source web server that also functions as areverse proxy, load balancer, and caching server. It is designed to handle a largenumber of simultaneous connections efficiently, making it ideal for modern webapplication

The command I ran to start a new container was:

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



7. Creating a Custom Docker Image

A **Dockerfile** is a script containing instructions for building a Docker image. Docker reads the Dockerfile to understand how to configure your application's environment and which software packages to install.

Steps to Create a Custom Nginx Image

1. Use the latest Nginx image:
    - Start with the latest version of Nginx as the base image:

```
FROM nginx:latest
```

2. Replace the default HTML file
    - Copy your custom index.html file to replace the default Nginx web page:
      (The index.html file contains a simple webpage that will be displayed when accessing the Nginx server.)

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

3. Expose port 80
    - Open port 80 to allow web traffic:

```
EXPOSE 80
```

File Edit View

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

8. Building the Custom Docker Image

- To build the image using your Dockerfile, run the following command in the terminal:

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

- The -t my-web-app flag assigns a name (my-web-app) to the image.
- The . at the end tells Docker to look for the **Dockerfile** in the current directory.

Now, the custom Nginx image is ready to be used!!!

9. Running the Docker Container

Once the image is built, launch a container:

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

To verify the container is running, navigate to **http://localhost in a web browser.**

*When I first tried to run my custom Docker image, I encountered an error because **port 80 was already in use by another running container**. This is a common issue when running multiple containers that use the same port.*

Identifying the Issue

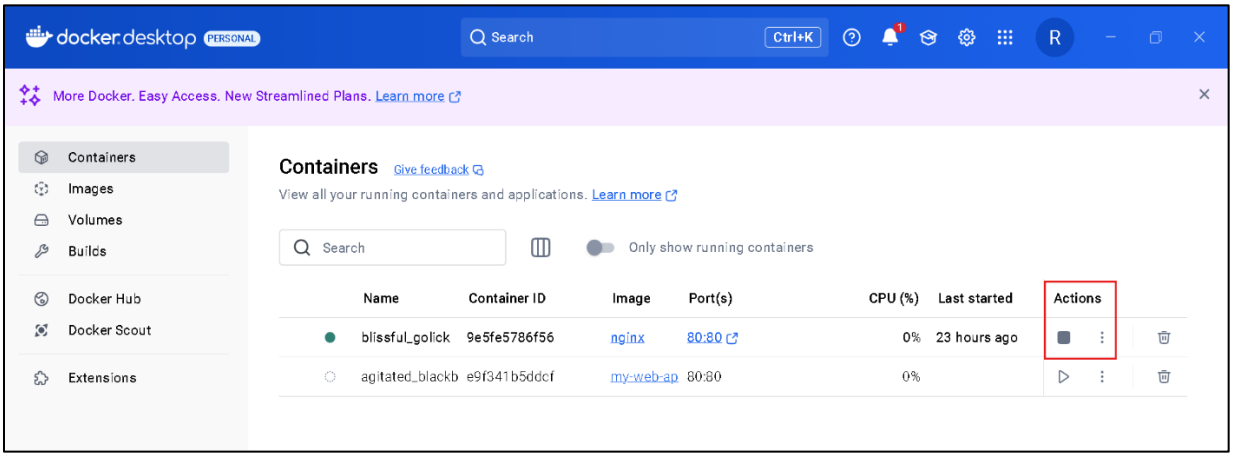
The error message indicated that the port was already allocated, preventing my container from starting. To diagnose this, I checked the running containers in **Docker Desktop**, where I found an existing container already using port 80.

```
 => => exporting config sha256:63263c2fc4b824f66c0608b3317e665945b1573134d3779a2d019ced86e45a2d      0.0s
 => => exporting attestation manifest sha256:6c7a45cd6877c5802feb5a9be836390c4a8e8a1310ed4b0201e21d  0.0s
 => => exporting manifest list sha256:a9307b73de2de6ff4d1a153884bb01cfedfac1cd96825a162fe9aff0cc159  0.0s
 => => naming to docker.io/library/my-web-app:latest                                               0.0s
 => => unpacking to docker.io/library/my-web-app:latest                                            0.1s
PS C:\Users\reesa\OneDrive\Desktop\Compute> docker run -d -p 80:80 my-web-app
e9f341b5ddcfd960da4aae2b4237d2241222884c57535d88ea32321491fc5e08
docker: Error response from daemon: driver failed programming external connectivity on endpoint agitated_b
lackburn (bb5a12d9cc1a7446c563641b116e34fcfad0ab5498b52fadd79376f9debbe385): Bind for 0.0.0.0:80 failed: p
ort is already allocated.
PS C:\Users\reesa\OneDrive\Desktop\Compute>
```

Resolving the Issue

To free up port 80 and allow my new container to run, I did the following:

1. Opened **Docker Desktop** and identified the container using port 80.
2. Stopped the container manually in the UI as shown in the screenshot below



OR

Used the following Docker command to stop the running container:

```
PS C:\Users\reesa\OneDrive\Desktop\Compute> docker ps --filter "publish=80"
CONTAINER ID   IMAGE     COMMAND                  CREATED         STATUS          PORTS                NAME
S
9e5fe5786f56   nginx     "/docker-entrypoint.…"   23 hours ago    Up 15 seconds   0.0.0.0:80->80/tcp   blis
sful_golick
PS C:\Users\reesa\OneDrive\Desktop\Compute> docker stop 9e5fe5786f56
9e5fe5786f56
PS C:\Users\reesa\OneDrive\Desktop\Compute>
RAM 1.00 GB  CPU 0.00%   Disk: 1.39 GB used (limit 1006.85 GB)
```

```
docker stop <container_id>
```

(Replace <container_id> with the actual ID of the container using port 80.)

3. After stopping the conflicting container, I re-ran my custom Docker image, and it started successfully.

Understanding Containers and Images

- A **Docker image** is like a **blueprint**—it includes the application code, dependencies, and configuration.
- A **Docker container** is a **running instance** of this image.
- In this case, my container was hosting a **web server** that served an index.html file.

Now that the issue was resolved, my custom container ran successfully, and I could access my web application without any problems.

10. Deploying Custom Image with AWS Elastic Beanstalk

Introduction to Elastic Beanstalk

AWS Elastic Beanstalk is a fully managed service that allows you to deploy cloud applications without having to worry about the underlying infrastructure. With Elastic Beanstalk, you simply upload your application code, and Elastic Beanstalk handles everything else: provisioning servers, managing scaling, and balancing loads.

Here, I'll walk you through how I deployed my custom Docker image using Elastic Beanstalk, which was quick and easy. The platform automatically handled deployment, scaling, and management, making my application accessible to users around the world.

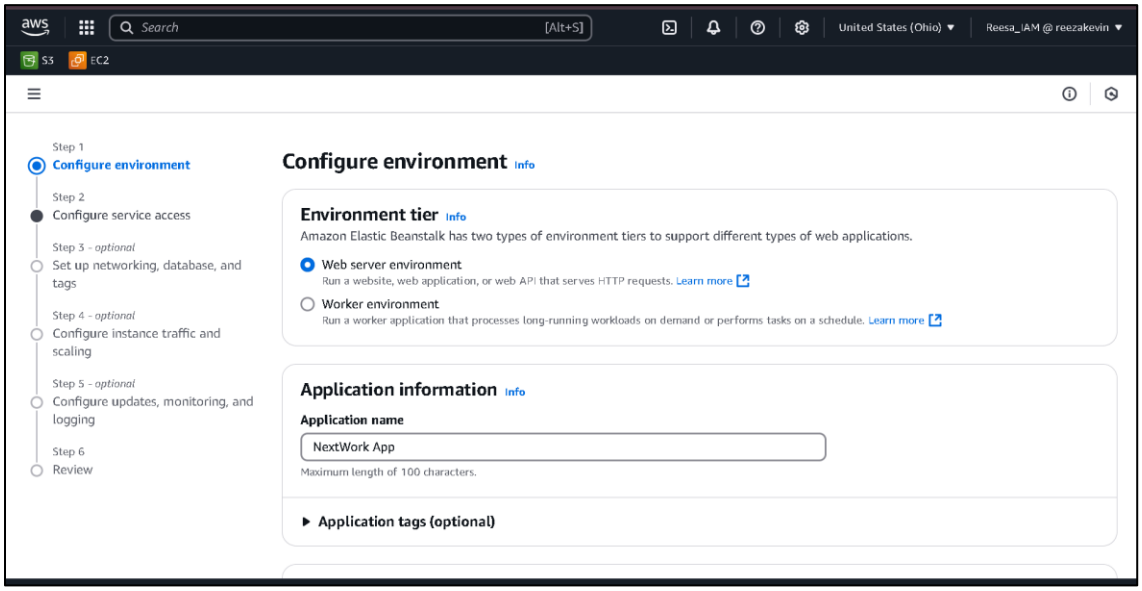
Steps to Deploy Your Application on Elastic Beanstalk

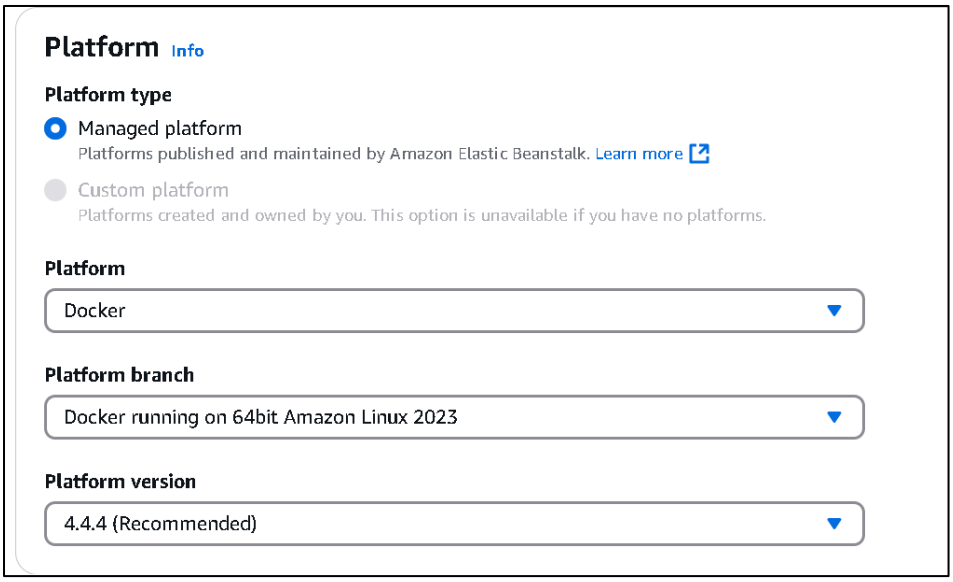
10.1. Create an Elastic Beanstalk Application

- **Log in to AWS Management Console:** Log in as your IAM user.
- **Search for Elastic Beanstalk:** Go to the AWS Management Console, search for Elastic Beanstalk, and click on the service.

10.2. Create a New Application

- Click on **Create Application** on the Elastic Beanstalk homepage.
- **Configure Environment:**
    - Leave the **Environment Tier** as the default.
    - Enter a name for your application (e.g., NextWork App).
    - Under the **Platform** section, select **Docker**.
    - The **Platform branch** and **Version** will be selected automatically.





10.3. Prepare Your Application Code

Before you upload your application code, you need to modify the index.html file:

- **Edit index.html:**
  - Open the index.html file on your local machine.
  - Find the line that says <h1>Hello from Reesa's custom Docker image!</h1>.
  - Add a new line beneath it to personalize the message:

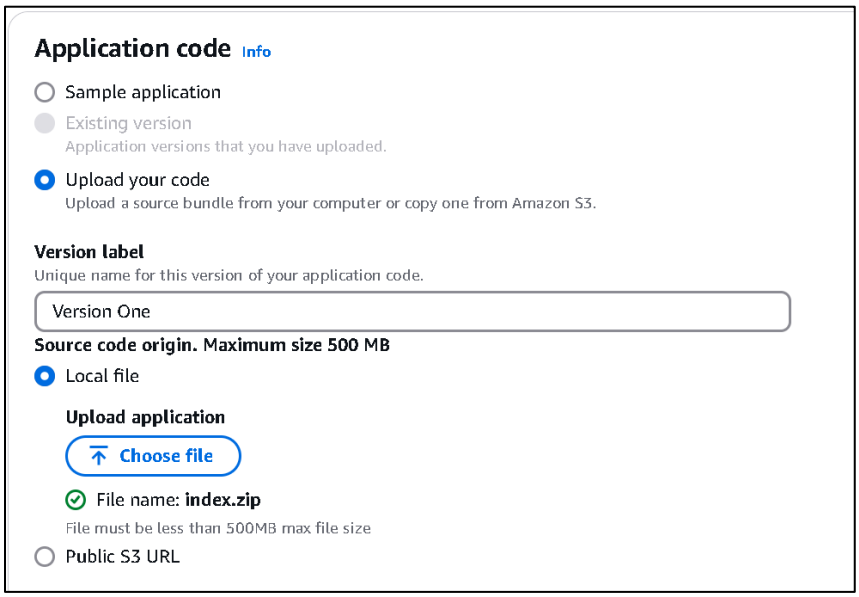
```
<h1>Hello from YOURNAME's custom Docker image!</h1>

<h1>If I can see this, it means Elastic Beanstalk has deployed an image with my work.</h1>
```

- **Prepare the ZIP File:**
  - Create a ZIP file containing Dockerfile and index.html at the root level (not inside any subfolder).
  - To create the ZIP file, select both files and compress them.

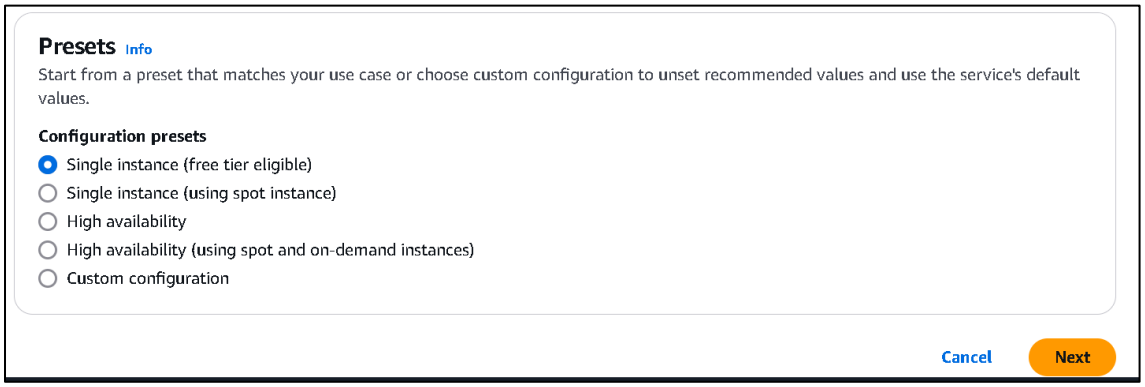
10.4. Upload Application Code to Elastic Beanstalk

- Back on the Elastic Beanstalk setup page, in the Application Code section, select Upload your code.
- Enter Version One as the Version Label.
- Select Local file and click Choose file to upload the ZIP file you created.



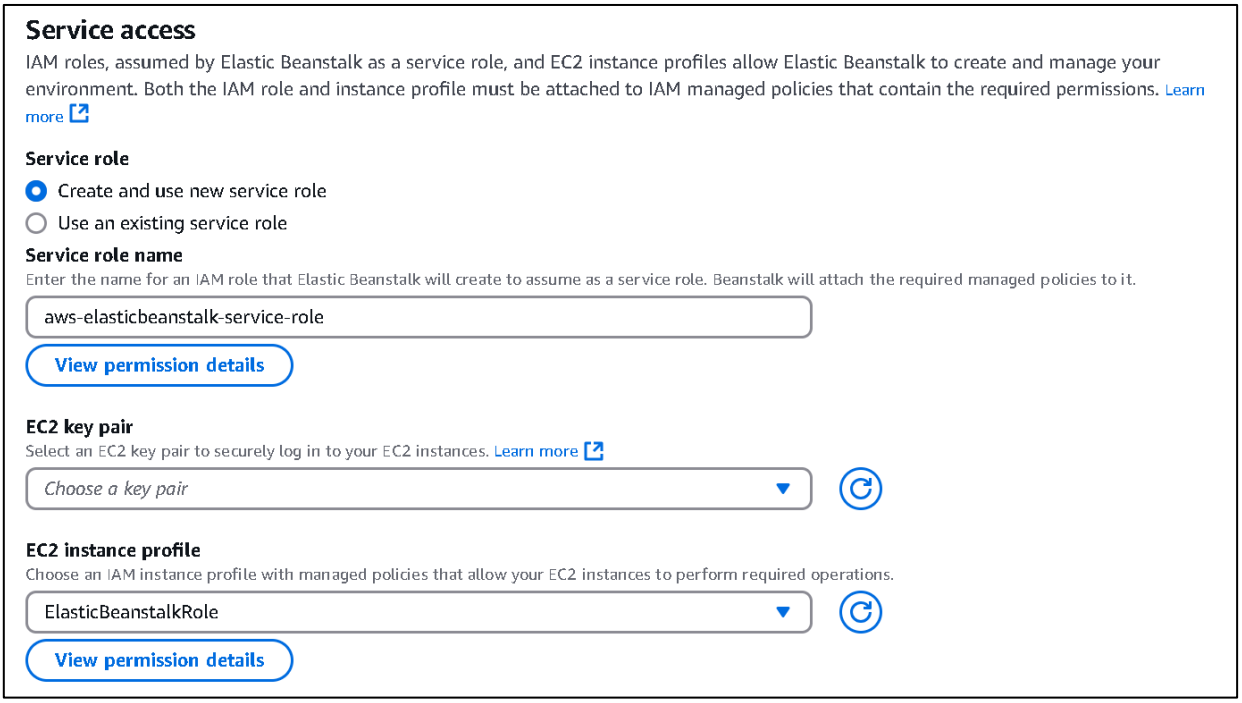
10.5. Configure the Environment

- Under Presets, select Single instance (Free tier eligible).
- Select Next.



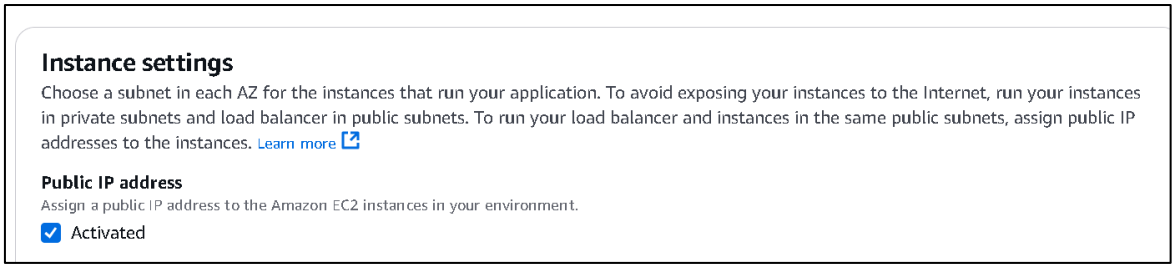
10.6. Service Access Configuration

- On the Configure service access page:
    - Select Create and use new service role. This will create a new IAM role for Elastic Beanstalk to use.
    - Keep the default service role name (aws-elasticbeanstalk-service-role).
    - Ignore the EC2 Key Pair dropdown (we don't need to access the EC2 instance directly).
    - Under EC2 instance profile, choose ecsInstanceRole.
- Select Next.



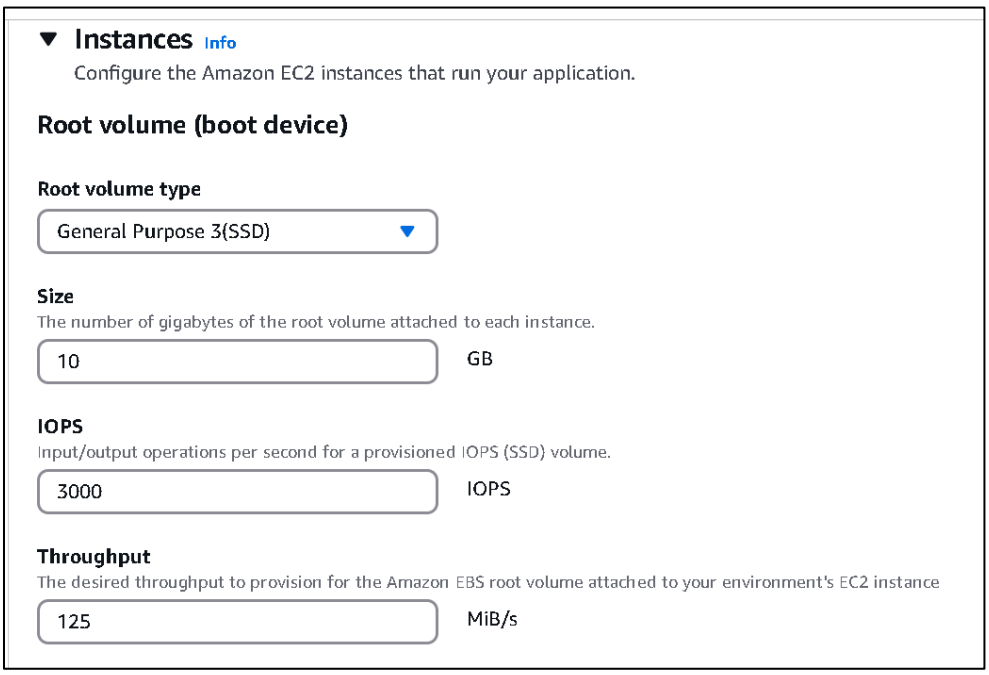
10.7. Set up Networking and Tags

- Networking: Ignore the Virtual Private Cloud (VPC) section (the default VPC is fine).
- Under Instance Settings, check Activated for the Public IP address option.
- Skip setting up a database for this project.
- Select Next.



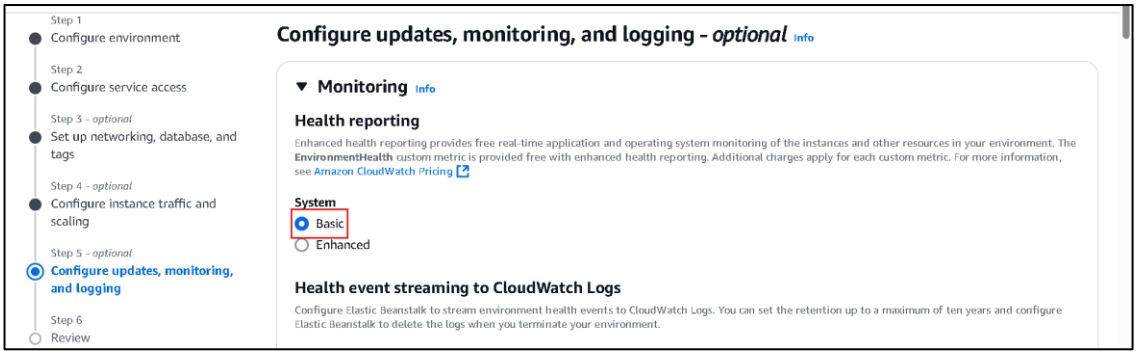
10.8. Configure Instance Traffic and Scaling

- Under **Instances**, select:
  - **Root volume type**: General Purpose 3 (SSD).
  - **Size**: 10 GB.
- Under **Instance metadata service (IMDS)**, select **Deactivated** for IMDSv1.
- Select **Next**.



10.9. Configure Updates, Monitoring, and Logging

- In the **Monitoring** section, select **Basic** for your system.
- **Managed Platform Updates**: Uncheck the **Activated** checkbox since we won't need managed updates.
- For **Rolling updates and deployments**, accept the default **All at once** deployment policy.
- Leave all other settings as default.
- Select **Next**.

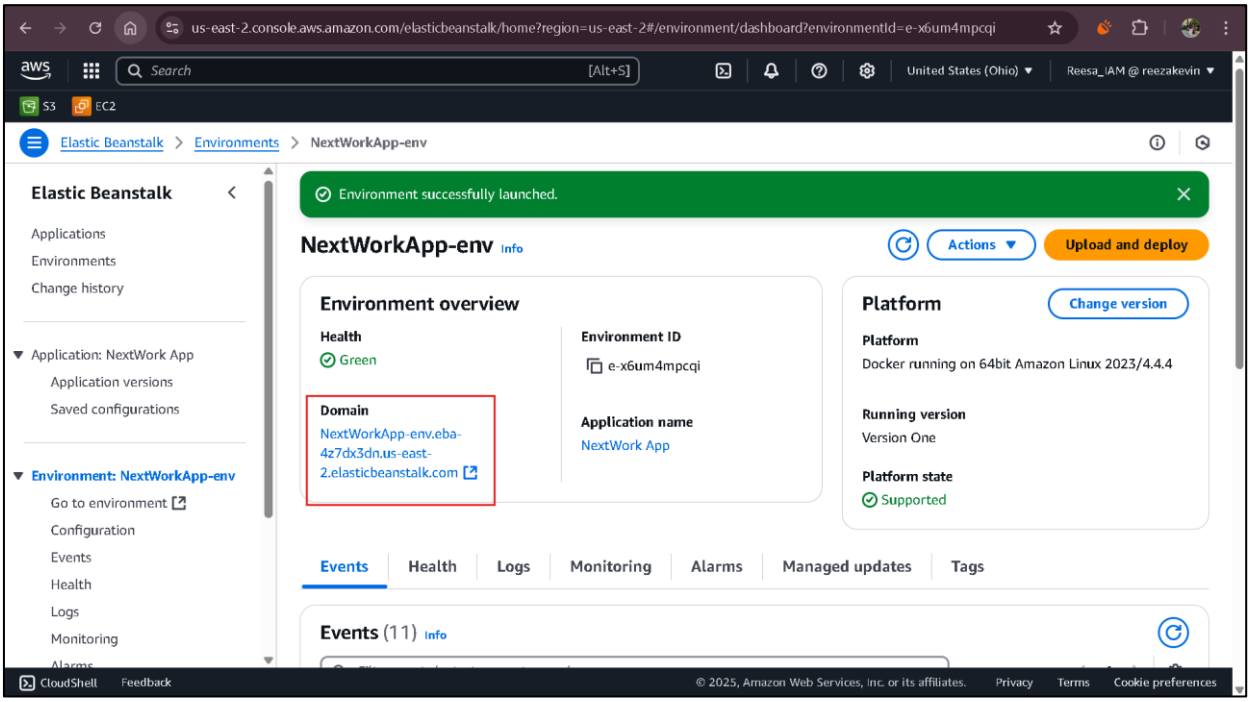


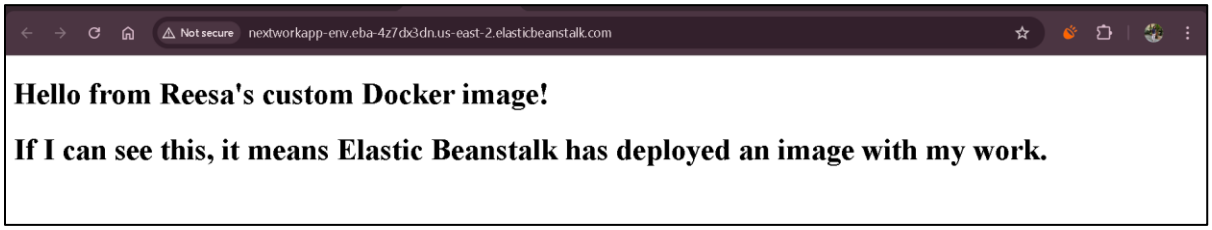
10.10. Review and Submit

- Review the configuration settings for your Elastic Beanstalk application.
    - Ensure the following:
        - Application name is NextWork App.
        - Application code is a ZIP file.
        - EC2 instance profile is set to ecsInstanceRole.
        - Public IP address is activated.
        - Root volume type is gp3.
        - Environment type is set to **Single instance**.
        - System is set to **Basic**.
        - Managed updates are deactivated.
        - Deployment policy is **AllAtOnce**.
- Click **Submit** to start the environment creation and application deployment process. This can take several minutes.

10.11. View Your Application

- Once the environment is successfully launched, click on the **Domain** link on the environment dashboard to see your application in action.
- You should see your updated HTML file, confirming that Elastic Beanstalk has successfully deployed your ZIP file's contents as a container image.





11. Cleaning Up Resources to Avoid Charges

To avoid any unwanted charges on your AWS account, it's essential to delete the resources you created once you're done with your project. Here's what you need to delete:

1. **Elastic Beanstalk Environment:**
   - Terminate the Elastic Beanstalk environment.
   - Delete the application from the Elastic Beanstalk console.
2. **S3 Bucket:**
   - Go to the S3 console and delete the automatically created bucket that starts with "elasticbeanstalk."
   - Empty and then delete the bucket.
3. **Containers and Container Images:**
   - Stop and remove any running Docker containers.

```
docker stop <container_id>
```

```
docker rm <container_id>
```

   - Delete the Docker container images from your local machine.

```
docker rmi <image_id>
```

By following these steps, you'll ensure that no unnecessary resources are left running and avoid any additional charges.

12. Conclusion

In this project, we successfully demonstrated how to containerize a web application using Docker and deploy it to AWS Elastic Beanstalk. By following the steps outlined, you learned how to build a custom Docker image, test it locally, and deploy it effortlessly using Elastic Beanstalk's fully managed services. With Docker's containerization capabilities and Elastic Beanstalk's automated infrastructure management, developers can streamline application deployment and ensure scalability. Moreover, we highlighted the importance of cleaning up resources to avoid unnecessary charges, ensuring the best practices for cost optimization. By mastering this process, you are equipped to deploy applications efficiently and manage your cloud resources effectively.