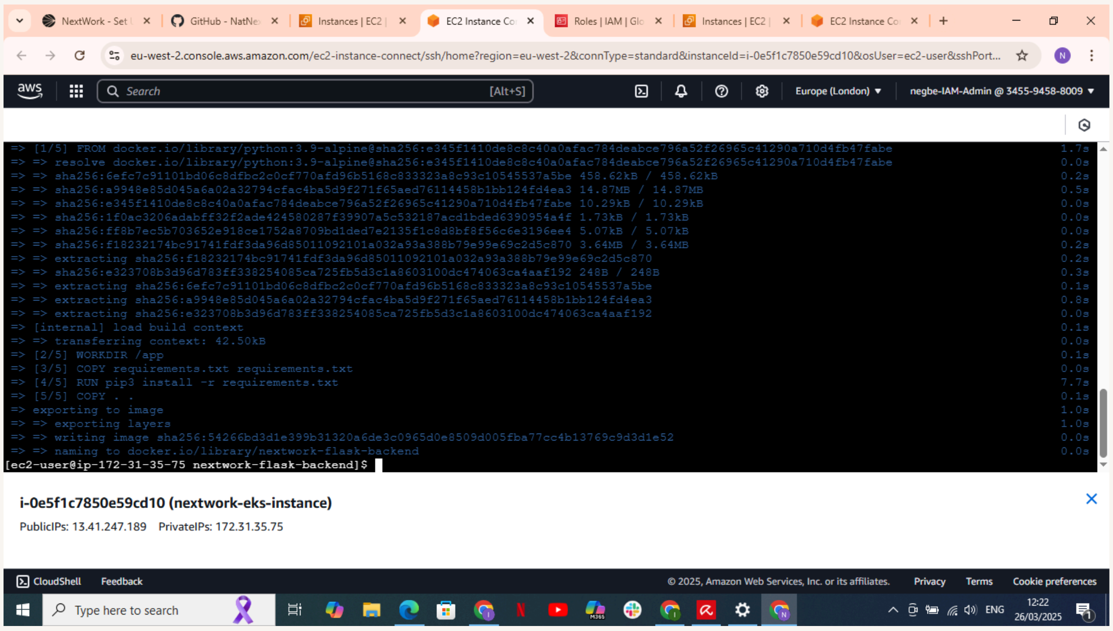
nextwork.org

Set Up Kubernetes Deployment

N Negbe Pierre



```
=> [1/5] FROM docker.io/library/python:3.9-alpine@sha256:e345f1410de8c8c40a0afac784deabce796a52f26965c41290a710d4fb47fabe                1.7s
=> => resolve docker.io/library/python:3.9-alpine@sha256:e345f1410de8c8c40a0afac784deabce796a52f26965c41290a710d4fb47fabe                0.0s
=> => sha256:6efc7c91101bd06c8dfbc2c0cf770afd96b5168c833323a8c93c10545537a5be 458.62kB / 458.62kB                                       0.2s
=> => sha256:a9948e85d045a6a02a32794cfac4ba5d9f271f65aed76114458b1bb124fd4ea3 14.87MB / 14.87MB                                         0.5s
=> => sha256:e345f1410de8c8c40a0afac784deabce796a52f26965c41290a710d4fb47fabe 10.29kB / 10.29kB                                         0.0s
=> => sha256:1f0ac3206adabff32f2ade424580287f39907a5c532187acd1bded6390954a4f 1.73kB / 1.73kB                                           0.0s
=> => sha256:ff8b7ec5b703652e918ce1752a8709bd1ded7e2135f1c8d8bf8f56c6e3196ee4 5.07kB / 5.07kB                                           0.0s
=> => sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5c870 3.64MB / 3.64MB                                           0.2s
=> => extracting sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5c870                                                0.2s
=> => sha256:e323708b3d96d783ff338254085ca725fb5d3c1a8603100dc474063ca4aaf192 248B / 248B                                               0.3s
=> => extracting sha256:6efc7c91101bd06c8dfbc2c0cf770afd96b5168c833323a8c93c10545537a5be                                                0.1s
=> => extracting sha256:a9948e85d045a6a02a32794cfac4ba5d9f271f65aed76114458b1bb124fd4ea3                                                0.8s
=> => extracting sha256:e323708b3d96d783ff338254085ca725fb5d3c1a8603100dc474063ca4aaf192                                                0.0s
=> [internal] load build context                                                                                                         0.1s
=> => transferring context: 42.50kB                                                                                                      0.0s
=> [2/5] WORKDIR /app                                                                                                                    0.1s
=> [3/5] COPY requirements.txt requirements.txt                                                                                          0.0s
=> [4/5] RUN pip3 install -r requirements.txt                                                                                            7.7s
=> [5/5] COPY . .                                                                                                                        0.1s
=> exporting to image                                                                                                                    1.0s
=> => exporting layers                                                                                                                   1.0s
=> => writing image sha256:54266bd3d1e399b31320a6de3c0965d0e8509d005fba77cc4b13769c9d3d1e52                                              0.0s
=> => naming to docker.io/library/nextwork-flask-backend                                                                                 0.0s
[ec2-user@ip-172-31-35-75 nextwork-flask-backend]$
```

i-0e5f1c7850e59cd10 (nextwork-eks-instance)

PublicIPs: 13.41.247.189 PrivateIPs: 172.31.35.75

eu-west-2.console.aws.amazon.com/ec2-instance-connect/ssh/home?region=eu-west-2&connType=standard&instanceId=i-0e5f1c7850e59cd10&osUser=ec2-user&sshPort...
Europe (London)
negbe-IAM-Admin @ 3455-9458-8009
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

Introducing Today's Project!

In this project, I will pull a backend application from GitHub, containerize it using Docker, and push the image to Amazon ECR because this simulates a real-world DevOps workflow. It helps me understand how developers prepare apps for Kubernetes deployment and test my ability to troubleshoot configuration issues effectively—all skills highly relevant to modern cloud-native development.

Tools and concepts

I used Amazon EKS, Git, Docker, and Amazon ECR to containerize a backend application and deploy it using Kubernetes. These tools worked together to simulate a real-world DevOps workflow where code is developed, containerized, pushed to a container registry, and then deployed to a managed Kubernetes service. Key steps include: Cloning the backend code from a GitHub repository that already had a Dockerfile and backend logic written using Flask. Creating a Kubernetes cluster using eksctl, which provisioned the EKS infrastructure on AWS. Installing Docker on the EC2 instance and resolving permission errors by adding the ec2-user to the Docker group, then starting the Docker daemon. Building a Docker image of the backend application using the Dockerfile, which defined the environment and instructions for the image. Pushing the Docker image to Amazon ECR, which acts as a container registry for storing images that Kubernetes can later pull. Exploring the backend files like requirements

Project reflection

2 hours

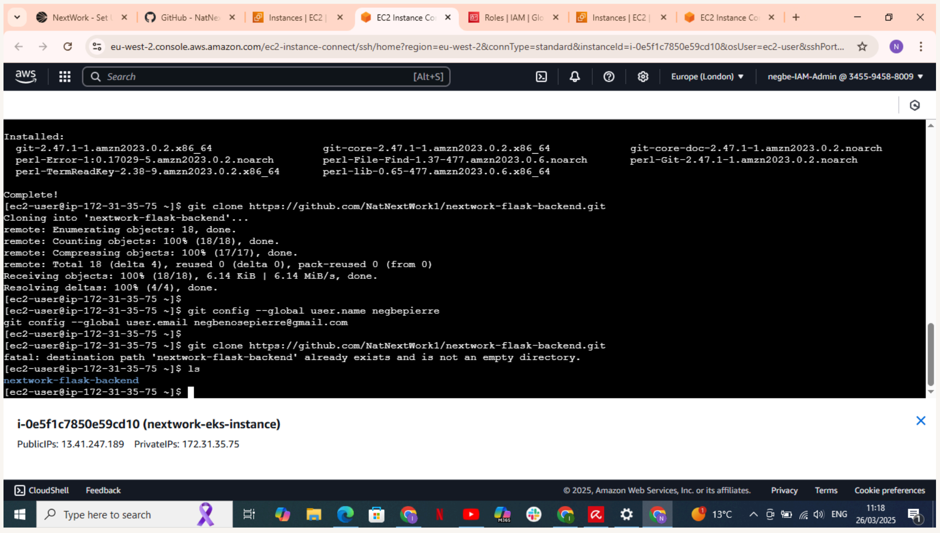
Something new that I learnt from this experience was how containerization and Kubernetes work together in a real-world deployment pipeline. While I had a basic understanding of Docker and containers before, this project showed me how a Docker image is built using a Dockerfile, pushed to Amazon ECR, and then pulled by Amazon EKS for deployment. I also learned how to troubleshoot common issues like Docker permission errors and how to grant the right access to the ec2-user. More importantly, I gained insight into how APIs are created and consumed within a Flask app, and how backend logic connects to external APIs to fetch and process data. Seeing the entire process from source code to a running app inside a Kubernetes cluster really brought the DevOps workflow to life for me.

What I'm deploying

To set up today's project, I launched a Kubernetes cluster. Steps I took to do this included: launching an EC2 instance, installing eksctl, and configuring the instance's AWS credentials. I then used eksctl to define the cluster's name, region, version, node type, and scaling parameters (min, max, desired nodes). I ran the create cluster command to deploy the cluster in the EU (London) region, making it ready to run containerized apps using Kubernetes efficiently.

I'm deploying an app's backend

Next, I retrieved the backend that I plan to deploy. An app's backend means the part of the application that handles things users don't see—like storing data, handling user logins, or running logic. I retrieved backend code by cloning a GitHub repository called nextwork-flask-backend. This copied all the backend files (like app.py, Dockerfile, and requirements.txt) into my EC2 instance so I can build and deploy them later in the project.


NextWork - Set
GitHub - NatNe
Instances | EC2
EC2 Instance Co
Roles | IAM | Glo
Instances | EC2
EC2 Instance Co
eu-west-2.console.aws.amazon.com/ec2-instance-connect/ssh/home?region=eu-west-2&connType=standard&instanceId=i-0e5f1c7850e59cd10&osUser=ec2-user&sshPort...
Search [Alt+S]
Europe (London)
negbe-IAM-Admin @ 3455-9458-8009

```
Installed:
  git-2.47.1-1.amzn2023.0.2.x86_64              git-core-2.47.1-1.amzn2023.0.2.x86_64         git-core-doc-2.47.1-1.amzn2023.0.2.noarch
  perl-Error-1:0.17029-5.amzn2023.0.2.noarch    perl-File-Find-1.37-477.amzn2023.0.6.noarch   perl-Git-2.47.1-1.amzn2023.0.2.noarch
  perl-TermReadKey-2.38-9.amzn2023.0.2.x86_64   perl-lib-0.65-477.amzn2023.0.6.x86_64

Complete!
[ec2-user@ip-172-31-35-75 ~]$ git clone https://github.com/NatNextWork1/nextwork-flask-backend.git
Cloning into 'nextwork-flask-backend'...
remote: Enumerating objects: 18, done.
remote: Counting objects: 100% (18/18), done.
remote: Compressing objects: 100% (17/17), done.
remote: Total 18 (delta 4), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (18/18), 6.14 KiB | 6.14 MiB/s, done.
Resolving deltas: 100% (4/4), done.
[ec2-user@ip-172-31-35-75 ~]$
[ec2-user@ip-172-31-35-75 ~]$ git config --global user.name negbepierre
git config --global user.email negbenosepierre@gmail.com
[ec2-user@ip-172-31-35-75 ~]$
[ec2-user@ip-172-31-35-75 ~]$ git clone https://github.com/NatNextWork1/nextwork-flask-backend.git
fatal: destination path 'nextwork-flask-backend' already exists and is not an empty directory.
[ec2-user@ip-172-31-35-75 ~]$ ls
nextwork-flask-backend
[ec2-user@ip-172-31-35-75 ~]$
```

i-0e5f1c7850e59cd10 (nextwork-eks-instance)

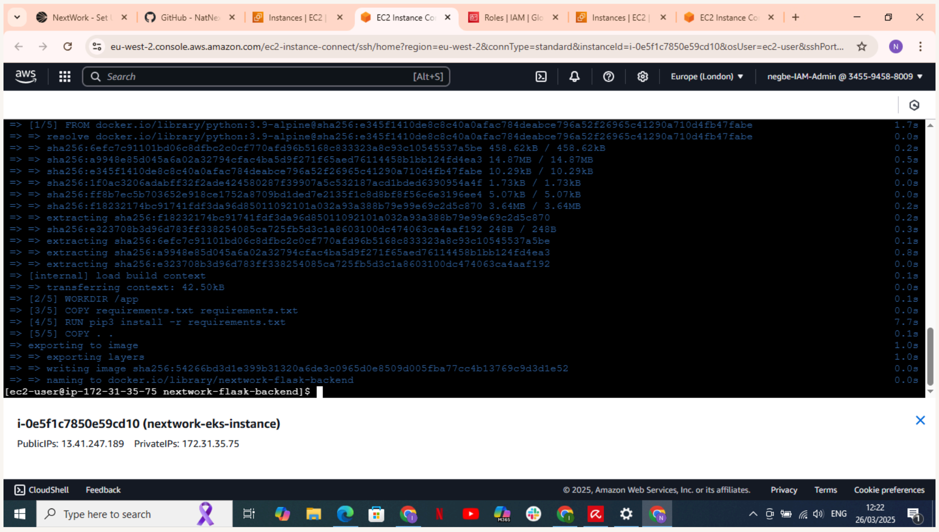
PublicIPs: 13.41.247.189 PrivateIPs: 172.31.35.75

Building a container image

Once I cloned the backend code, my next step is to build a container image of the backend. This is because Kubernetes deploys applications using container images. A container image includes everything the backend app needs—its code, libraries, and settings—so it can run consistently on any system. By building this image, I'm making the app ready for Kubernetes to deploy and manage smoothly across different environments

When I tried to build a Docker image of the backend, I ran into a permissions error because the ec2-user (the default user for the EC2 instance) didn't have permission to run Docker commands. The Docker engine was set up for the root user, and I wasn't using sudo, so Docker couldn't connect to its daemon. To fix this, I either needed to prefix the command with sudo or give ec2-user permission to use Docker without needing elevated rights.

To solve the permissions error, I added the ec2-user to the Docker group using the command sudo usermod -a -G docker ec2-user. The Docker group is a special user group in Linux that allows users to run Docker commands without needing to type sudo every time. By adding ec2-user to this group, I gave it the right permissions to build Docker images and run Docker commands freely.



```
=> [1/5] FROM docker.io/library/python:3.9-alpine@sha256:e345f1410de8c8c40a0afac784deabce796a52f26965c41290a710d4fb47fabe      1.7s
=> => resolve docker.io/library/python:3.9-alpine@sha256:e345f1410de8c8c40a0afac784deabce796a52f26965c41290a710d4fb47fabe      0.0s
=> => sha256:6efc7c91101bd06c8dfbc2c0cf770afd96b5168c833323a8c93c10545537a5be 458.62kB / 458.62kB                              0.2s
=> => sha256:a9948e85d045a6a02a32794cfac4ba5d9f271f65aed76114458b1bb124fd4ea3 14.87MB / 14.87MB                                0.5s
=> => sha256:e345f1410de8c8c40a0afac784deabce796a52f26965c41290a710d4fb47fabe 10.29kB / 10.29kB                                0.0s
=> => sha256:1f0ac3206adabff32f2ade424580287f39907a5c532187acd1bded6390954a4f 1.73kB / 1.73kB                                  0.0s
=> => sha256:ff8b7ec5b703652e918ce1752a8709bd1ded7e2135f1c8d8bf8f56c6e3196ee4 5.07kB / 5.07kB                                  0.0s
=> => sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5c870 3.64MB / 3.64MB                                  0.2s
=> => extracting sha256:f18232174bc91741fdf3da96d85011092101a032a93a388b79e99e69c2d5c870                                       0.2s
=> => sha256:e323708b3d96d783ff338254085ca725fb5d3c1a8603100dc474063ca4aaf192 248B / 248B                                      0.3s
=> => extracting sha256:6efc7c91101bd06c8dfbc2c0cf770afd96b5168c833323a8c93c10545537a5be                                       0.1s
=> => extracting sha256:a9948e85d045a6a02a32794cfac4ba5d9f271f65aed76114458b1bb124fd4ea3                                       0.8s
=> => extracting sha256:e323708b3d96d783ff338254085ca725fb5d3c1a8603100dc474063ca4aaf192                                       0.0s
=> [internal] load build context                                                                                               0.1s
=> => transferring context: 42.50kB                                                                                            0.0s
=> [2/5] WORKDIR /app                                                                                                          0.1s
=> [3/5] COPY requirements.txt requirements.txt                                                                                0.0s
=> [4/5] RUN pip3 install -r requirements.txt                                                                                  7.7s
=> [5/5] COPY . .                                                                                                              0.1s
=> exporting to image                                                                                                          1.0s
=> => exporting layers                                                                                                         1.0s
=> => writing image sha256:54266bd3d1e399b31320a6de3c0965d0e8509d005fba77cc4b13769c9d3d1e52                                    0.0s
=> => naming to docker.io/library/nextwork-flask-backend                                                                       0.0s
[ec2-user@ip-172-31-35-75 nextwork-flask-backend]$
```

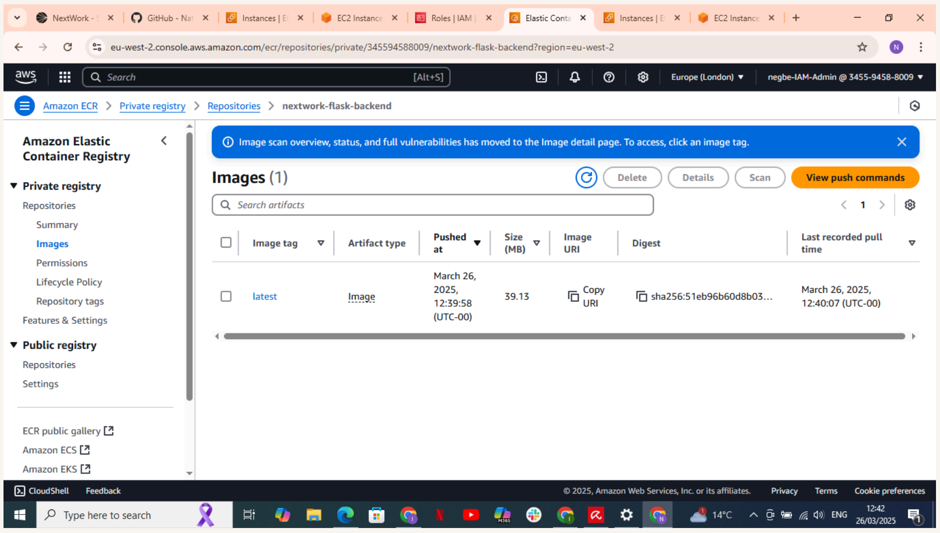
i-0e5f1c7850e59cd10 (nextwork-eks-instance)

PublicIPs: 13.41.247.189 PrivateIPs: 172.31.35.75

Container Registry

I'm using Amazon ECR in this project to store and manage my Docker container image. ECR is a good choice for the job because it's an AWS-managed container registry that integrates smoothly with EKS, allowing Kubernetes to deploy apps directly from it.

Container registries like Amazon ECR are great for Kubernetes deployment because they provide a secure and centralized place to store container images. This makes it easy for Kubernetes to pull the exact version of an app it needs, ensuring consistent and reliable deployments. It also simplifies collaboration between teams and supports scaling by making images easily accessible across different environments.



eu-west-2.console.aws.amazon.com/ecr/repositories/private/345594588009/nextwork-flask-backend?region=eu-west-2

Europe (London) · negbe-IAM-Admin @ 3455-9458-8009

Amazon ECR > Private registry > Repositories > nextwork-flask-backend

Image scan overview, status, and full vulnerabilities has moved to the Image detail page. To access, click an image tag.

Images (1)

Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest	Last recorded pull time
latest	Image	March 26, 2025, 12:39:58 (UTC-00)	39.13	Copy URI	sha256:51eb96b60d8b03...	March 26, 2025, 12:40:07 (UTC-00)

EXTRA: Backend Explained

After reviewing the app's backend code, I've learnt that the backend is responsible for handling user input, retrieving external data, and formatting it for output in a way that the frontend—or the user—can easily understand. It uses Flask to create a simple API that connects to the Hacker News Search API. When a user enters a topic, this input is passed into the backend, which then fetches relevant data from the external API. The code processes this raw data by filtering and organizing it into a structured format using JSON. This processed information is then returned to the user, providing them with clean and relevant search results. The logic behind this is all housed in the app.py file, which orchestrates how data is requested, handled, and delivered. In essence, I now understand that the backend plays the role of a smart middleman. It listens for requests, communicates with external services to gather information, and ensures the response is readable and usable. This exploration

Unpacking three key backend files

The requirements.txt file is a crucial component of the backend project because it defines all the Python libraries that the application depends on to function properly. When building the backend, developers include this file to ensure that anyone who works on or deploys the app can install the exact versions of the tools it needs. In this project, the file specifies the backend uses Flask as its web framework, providing the structure and functionality to handle user requests and serve responses. To help the app expose an API, it uses an extension called Flask-RESTX, which simplifies the process of building RESTful interfaces. The app also needs the Requests library, which allows it to communicate with external services and retrieve data from the web. Additionally, Werkzeug is included as it handles the routing system used by Flask. This lets the application know which function to run when it receives a specific request. By including these dependencies in requirements.txt, the build pr

The Dockerfile gives Docker instructions on how to build a container image for the backend of an application. It acts like a recipe, telling Docker exactly what steps to follow in order to create a working environment where the application can run smoothly and consistently across any system. This is what makes containerization so powerful—everything the app needs to function is bundled into one lightweight, portable image that behaves the same way no matter where it's deployed. Key commands in this Dockerfile include FROM python:3.9-alpine, which specifies the base image to use—here, a slim version of Python 3.9 that helps keep the final image small and efficient. The LABEL command adds metadata, identifying the image's author. With WORKDIR /app, Docker sets the working directory to /app, meaning all subsequent commands will be run from that location in the container. The Dockerfile also copies files from your local project into the image. It uses COPY requirements.txt requirements.t

The app.py file contains the core logic of the backend application. It defines how the application should behave, how it should respond to incoming requests, and how it communicates with external services. In this case, the file is built using Flask, a lightweight Python web framework, and Flask-RESTx, which helps to easily build RESTful APIs. At the beginning of the file, several essential libraries are imported, including Flask, Flask-RESTx, Requests, and JSON. These libraries enable the app to handle HTTP requests, build endpoints, and interact with data in JSON format. The Flask app is initialized, and a new API is attached to it using Api(app). A single route is defined using the @api.route('/contents/<string:topic>') decorator, which allows users to search for content by passing a topic through the URL. Inside the route, there is a class called SearchContentsResource, and it contains a get method. When a user sends a GET request to this route, the backend takes the topic provid

Everyone should be in a job they love.

Check out nextwork.org for more projects