**ECS Fargate Microservices CI/CD Project**

A fully containerized microservices architecture built on AWS using **ECS Fargate**, integrated with **AWS CodePipeline** and **CodeBuild** for CI/CD, storing Docker images in **Amazon ECR**, and load balanced by an **Application Load Balancer (ALB)**.

**1. Project Overview**

This project demonstrates:

* How to containerize multiple Spring Boot microservices.
* How to automate build, push to ECR, and deploy to ECS using AWS CodePipeline.
* Using an ALB to distribute traffic and perform health checks.
* Managing builds and deployments efficiently with Docker and AWS managed services.

It is designed to be **cost-effective**, **scalable**, and **easy to extend**.

**2. Architecture**

**Components**

| **Layer** | **Services/Tools** |
| --- | --- |
| Source | GitHub (repository for microservices + Dockerfiles) |
| CI | AWS CodePipeline + AWS CodeBuild |
| Container Reg. | Amazon ECR |
| Orchestration | ECS Fargate |
| Networking | ALB, Target Groups, VPC, Subnets |

**Deployment Flow**

1. Code pushed to GitHub (main branch).
2. Webhook triggers CodePipeline.
3. CodeBuild:
   * Builds Docker image.
   * Pushes image to ECR.
   * Creates imagedefinitions.json.
4. ECS service is updated to pull the new image.
5. ALB routes traffic to healthy tasks.

**3. Microservices**

Each microservice is a **Spring Boot** application exposing REST APIs.

| **Service** | **Purpose** | **Example Endpoint** |
| --- | --- | --- |
| User Service | Manages users | /user/api/users |
| Product Service | Manages products | /product/api/products |
| Auth Service | Handles authentication (future enhancement) | /auth/api/login |

All are containerized using Docker.

**4. CI/CD Pipeline Explained**

| **Stage** | **Tool** | **What it does** |
| --- | --- | --- |
| Source | CodePipeline | Monitors GitHub repo; triggers on push to main |
| Build | CodeBuild | Builds Docker image, tags, pushes to ECR, generates imagedefinitions.json |
| Deploy | CodePipeline | ECS deploy action uses image definitions to update ECS service |

**📝 buildspec.yml (summary)**

* Pre-build: Login to ECR.
* Build: Docker build & tag.
* Post-build: Docker push + create imagedefinitions.json.

**5. Deployment & Networking**

* **ECS Fargate** tasks run in private subnets.
* **ALB** in public subnets routes external traffic to target groups.
* Health check configured at /product/api/health or /user/api/health.
* Security groups control access (only ALB can reach ECS).

**6. Health Checks and Load Balancing**

* **ALB** directs traffic to ECS tasks.
* **Health checks** ensure only healthy containers get traffic.
  + Example: /product/api/health
* If the health check path returns 200 OK, container is marked healthy.
* If failing, ECS automatically restarts the task.

**7. Monitoring & Logging**

* **CloudWatch Logs:** ECS tasks stream logs automatically.
* **Metrics:** CPU/memory usage.
* **Alarms:** Can auto-scale or send alerts.
* **Troubleshooting:** Use ALB logs to debug 404s or failed health checks.

**8. Troubleshooting**

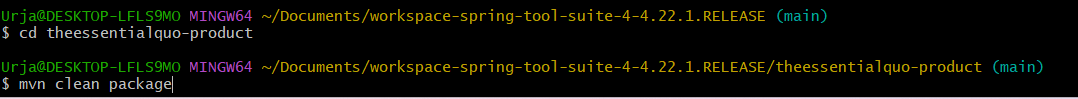
| **Issue** | **Solution** |
| --- | --- |
| Webhook not triggering | Check GitHub App connection in AWS Developer Tools. |
| imagedefinitions.json missing | Verify buildspec.yml syntax. |
| Health checks failing | Confirm correct path & port. |
| 429 Docker pull limit | Use ECR private base image or authenticated Docker Hub account. |
| Permission denied | Ensure IAM role for CodePipeline & CodeBuild has correct policies. |

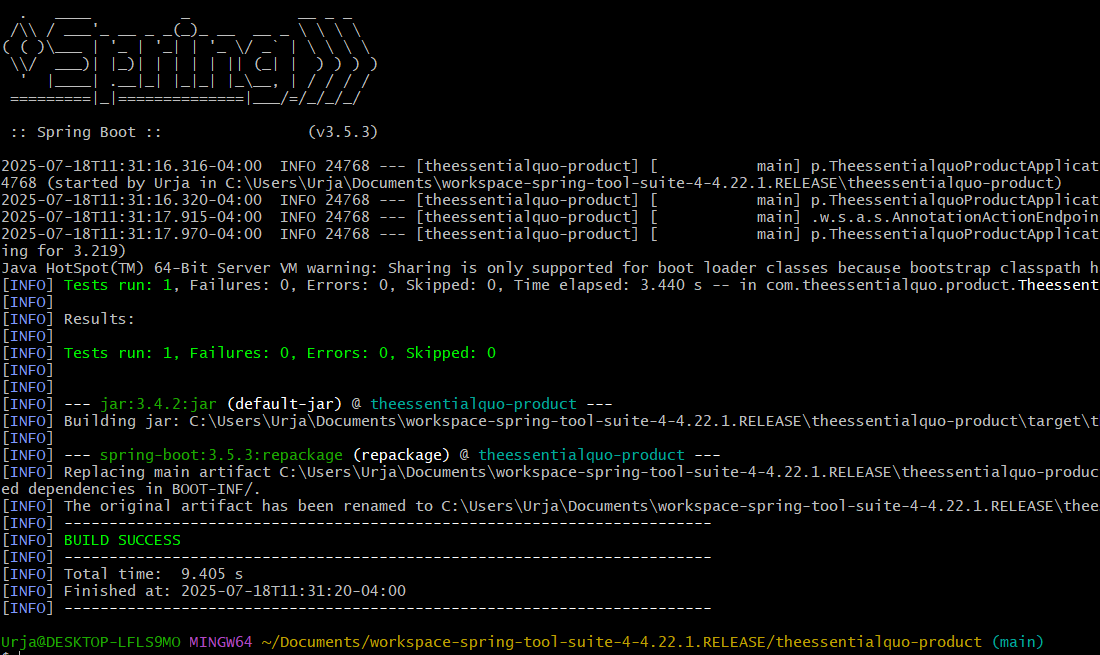
**9. Steps to implement the project in phases**

**Part 1**

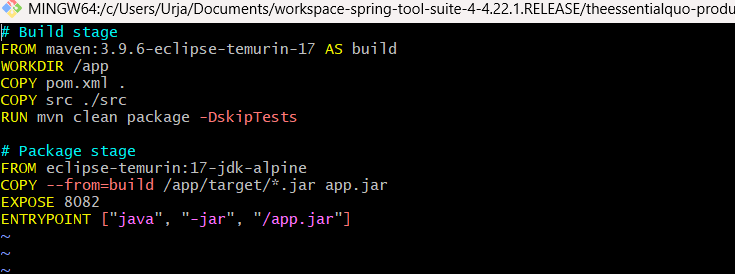
Create 2 spring boot microservices.each having its own pom.xml , so that it can create a .jar file when application needs to run.

Microservices: theessentialquo-user, theessentialquo-product  
  
then to run the application locally and create a .jar file we will run mvn clean package which mill save the .jar in the target folder, which will be later used in dockerfile.





-Now we need to build a docker image of the application so we will create a Dockerfile and keep it in the same location where the pom.xml is so that it can access the pom.xml and .jar ,as it needs only these two to run the application.



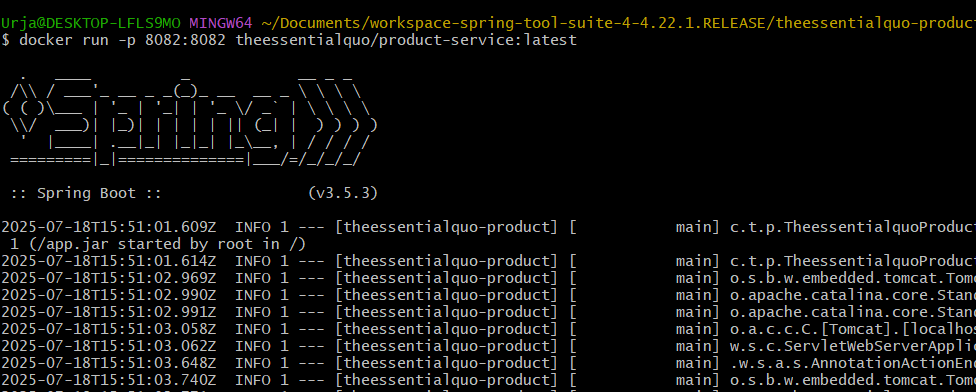
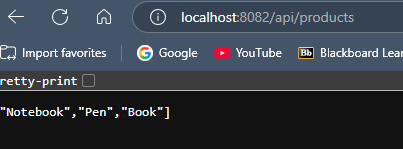
We will do a multi stage build, where in the build stage we will build the application from a base image ‘maven:3.9.6-eclipse-temurin-17’ and name it as build. Then we will set the /app as the working directory where the application will be stored. Since we only need pom.xml and the src code we will copy these to the /app directory. And then to build the project we will use maven clean package command.

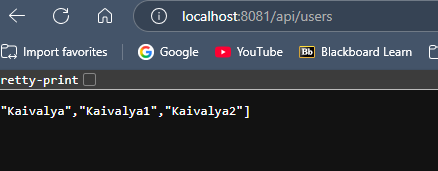
In the package stage, we will again take an base image which is light and has only jdk to run the .jar files. Then we will copy the .jar files from the build stage present in the target folder, to the app.jar. Then we will expose the microservice to a port 8082, and give a entrypoint command to run the app.jar file.

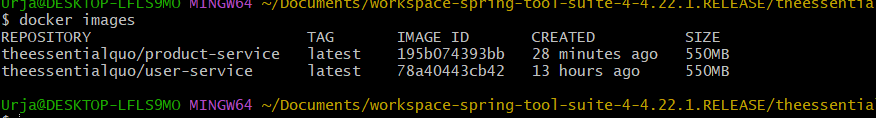
-Now build the docker image with the Dockerfile created.  

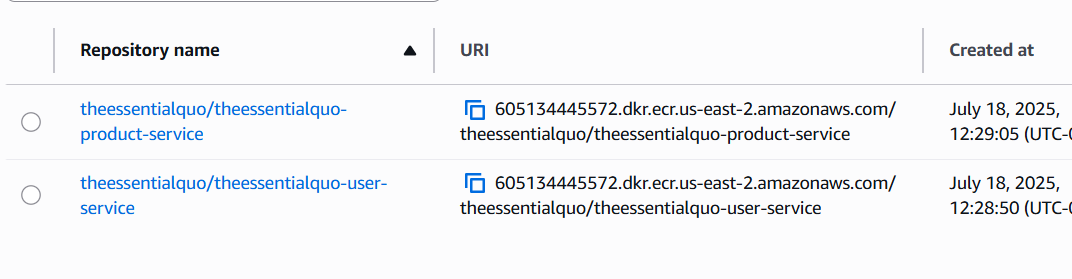

-Once the image is built we can run the image as a container.  

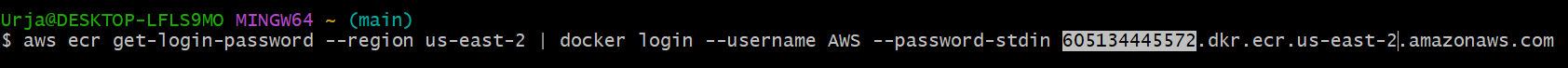

-The the spring boot application will be running , and we can test it by going to local host and the port given and the address provided in the controller.

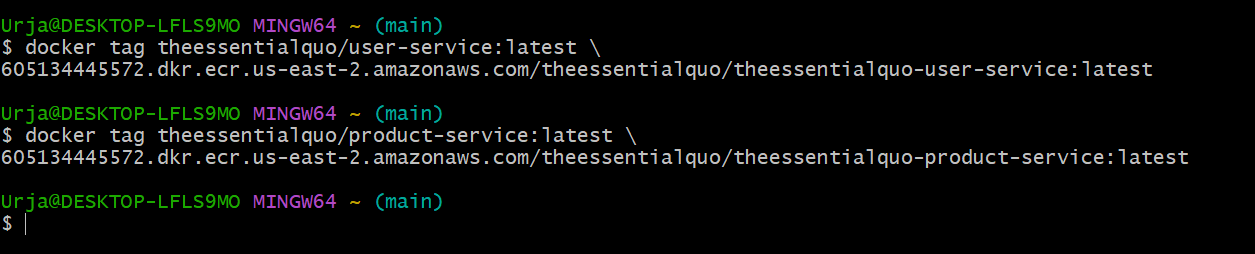
  
  


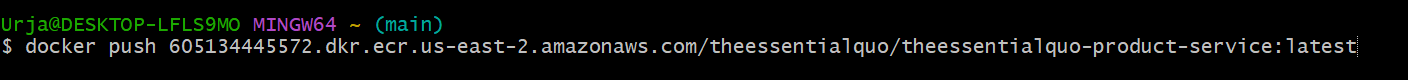
-Similarly we will create a spring boot micrservice for users and reate a docker image and run it.  


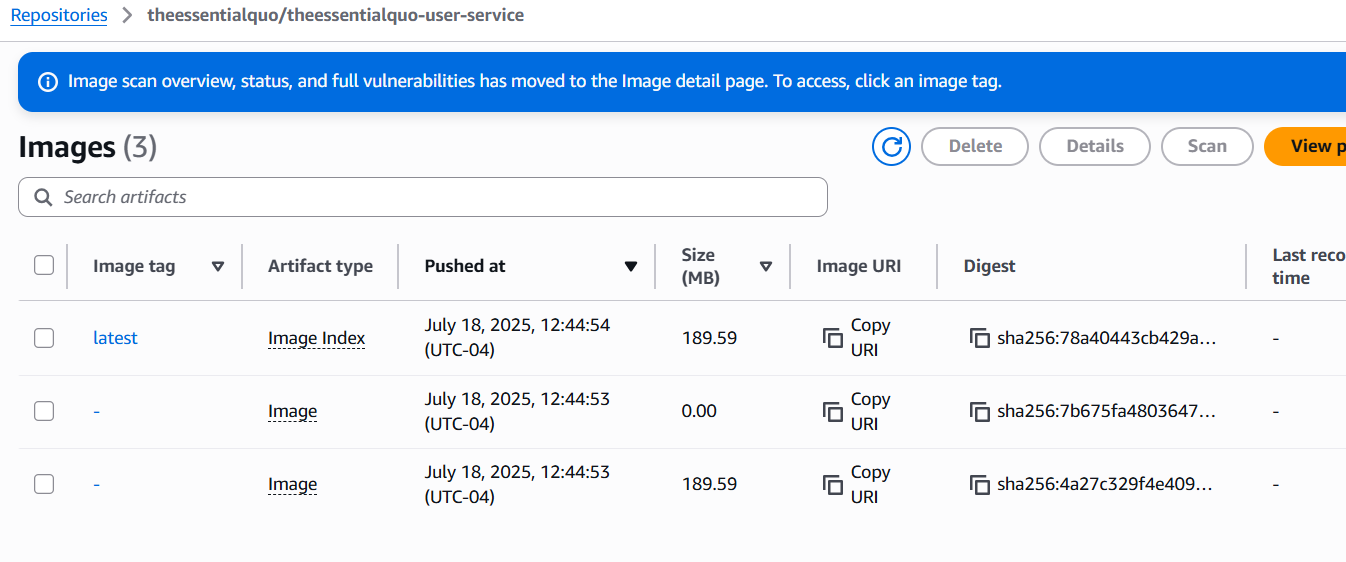


-Now we need to push these images to amazon ECR (elastic container registry). So we will create 2 repositories in the ecr.  


-Then we will got to bash and write below command to authenticate the docker to aws ECR. Where the highlighted number is aws account id.  


-Now tag the docker images and place a path where it will be placed.  


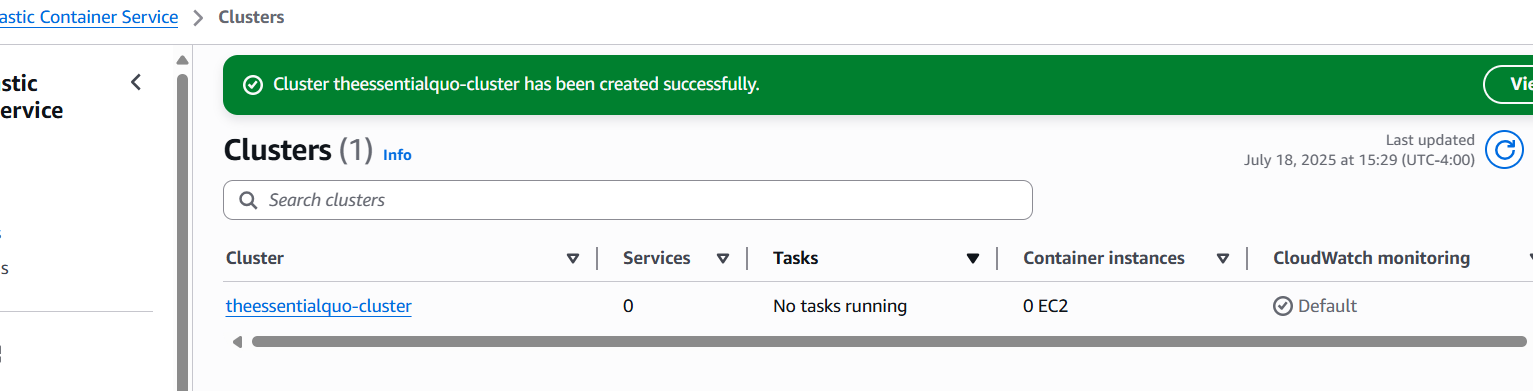
-Then push these images to ECR.  


And we can verify the images in ECR.  


**Part 2: Deploying Microservices on ECS Fargate**

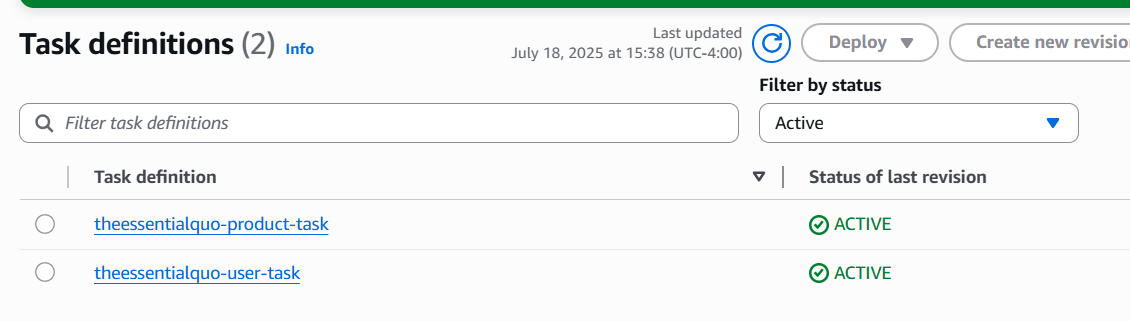
1. Create ECS Fargate Cluster

* Go to the ecs in aws console and create cluster



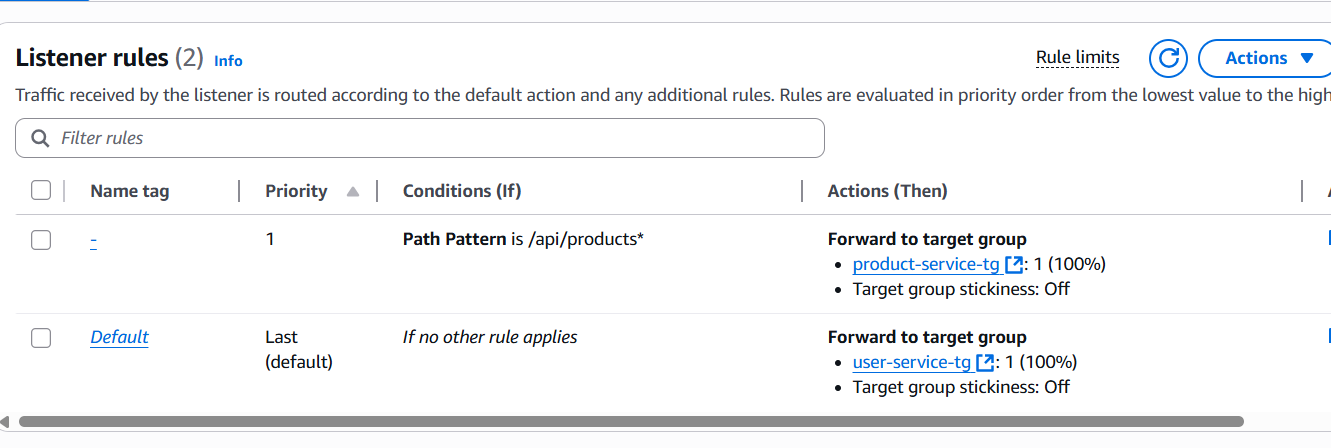
1. Create ECS Task Definitions (one per service)

* Create two different task definition one for each microservice. Name them and select fargate. Add the container and name it user-service, and add the uri of the container.Also add the port mappings as per the Dockerfile.

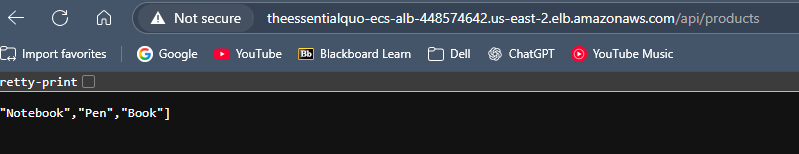


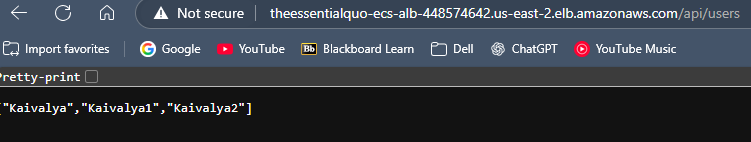
1. Create ECS Fargate Services

* Now go to the cluster theessentialquo-cluster and create a service for each task which can run the containers.
* Give the name to the service and select the task family for which we want to create the service. Then select the fargate option and the capacity and how many base tasks should be running.
* Then select the deployment strategy as rolling update or blue/green strategy. Where the rolling update will be replacing old tasks one by one. While the blue/green will create two environments and blue(current) and green(new) version. So once the green environment is tested we can configure our load balancer to new environment and if we want to roll back we can again go to blue. We will be using rolling update since it’s a simple application.
* Then in the networking section select the vpc which we want to use and the public subnets and the related internet gateways. The security group for the task service should allow access from the load balancer via the port on which its running.
* Then to use load balancer we need to create a load balancer which listens on port 80 for http and create the target group with path based routing. Here the controller in the java microservices has a path of ‘/api/users’ so use the same path in the routing of load balancer to avoid conflicts. The security group should allow all traffic from the internet on port 80.



* So when we try to access our api through load balancer we can see the data being rendred.

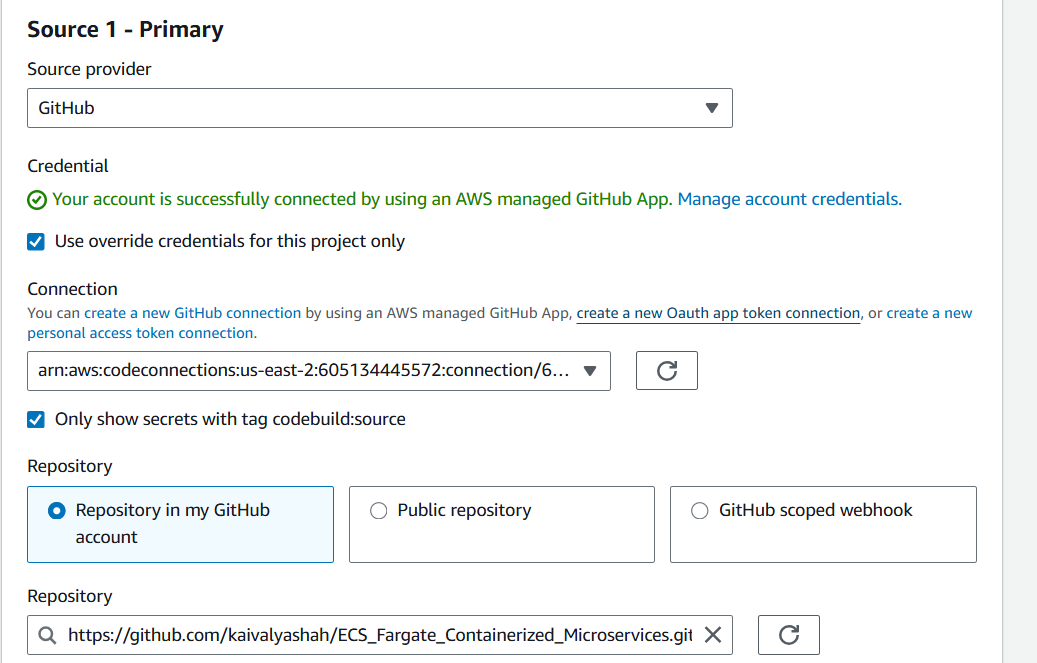




**Part 3: CI/CD Pipeline for Microservices**

1. Set up AWS CodeBuild

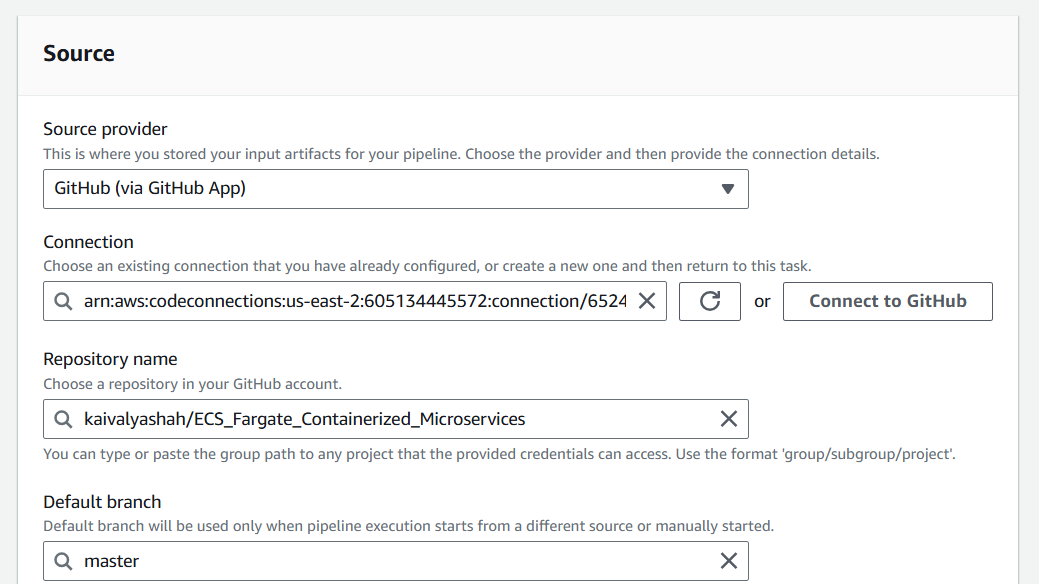
* First push you project to the git repository.
* Go to codebuild and select the create or build new project.
* Then give the name of the project and connect it to github in the source section. Select he repository it is in.



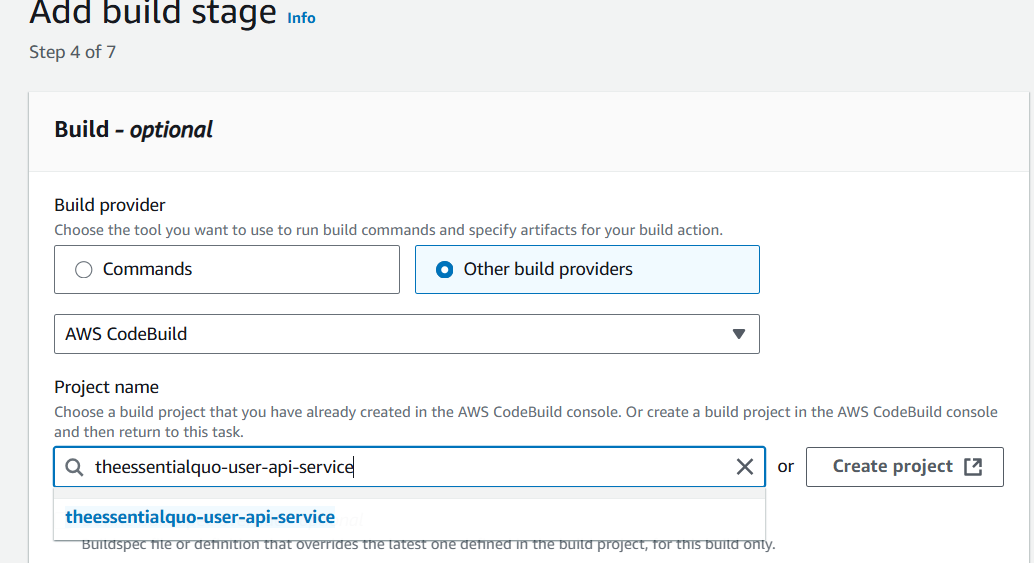
* In the Buildspec section insert build commands and provide the buildspec details.
* Select or create a service role and build the project, add the ecrpoweruser policy for the project to connect to ecr.
* Verify the docker tags and the names of the image in the ECR.
* Once the build is successful, verify the image created in the ECR.

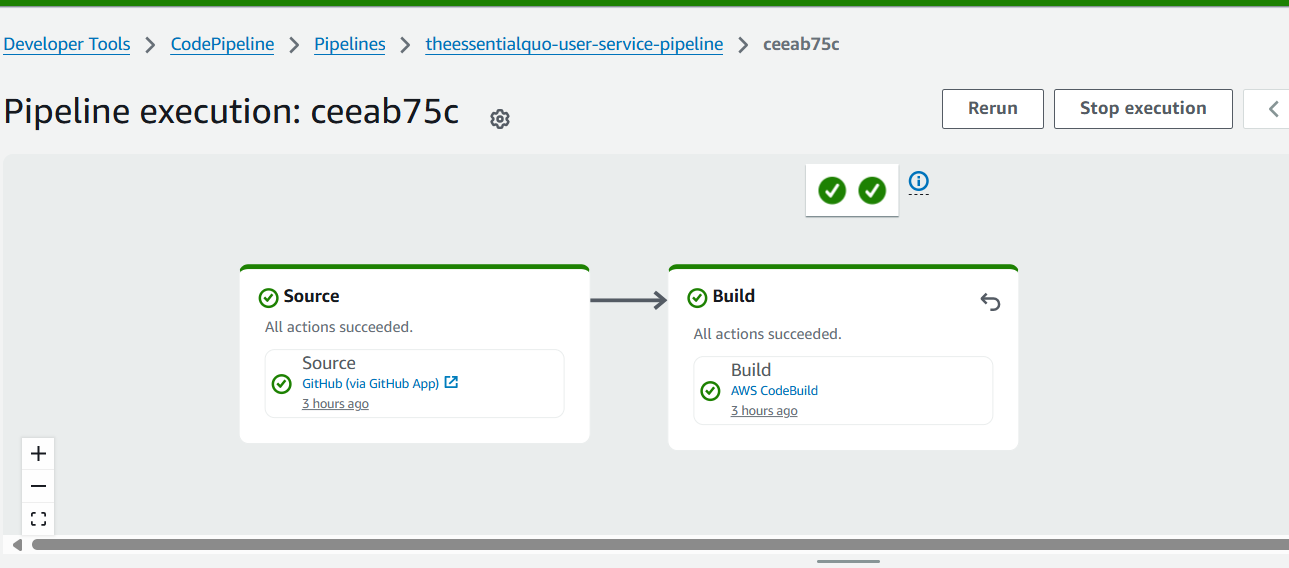
1. Set up AWS Codepipeline for CI only integrating codebuild.

* Codepipeline is the service provided by the aws to manage ci/cd. We will be using it to connect the stages:  
  Source: pull code from Github/CodeCommit.  
  Build: build docker image in codebuild.  
  Deploy: deploy the image into ECS fargate.
* Now create a custom pipeline and give a suitable name to it.
* Give the source provider github and select the connection and the repo name.

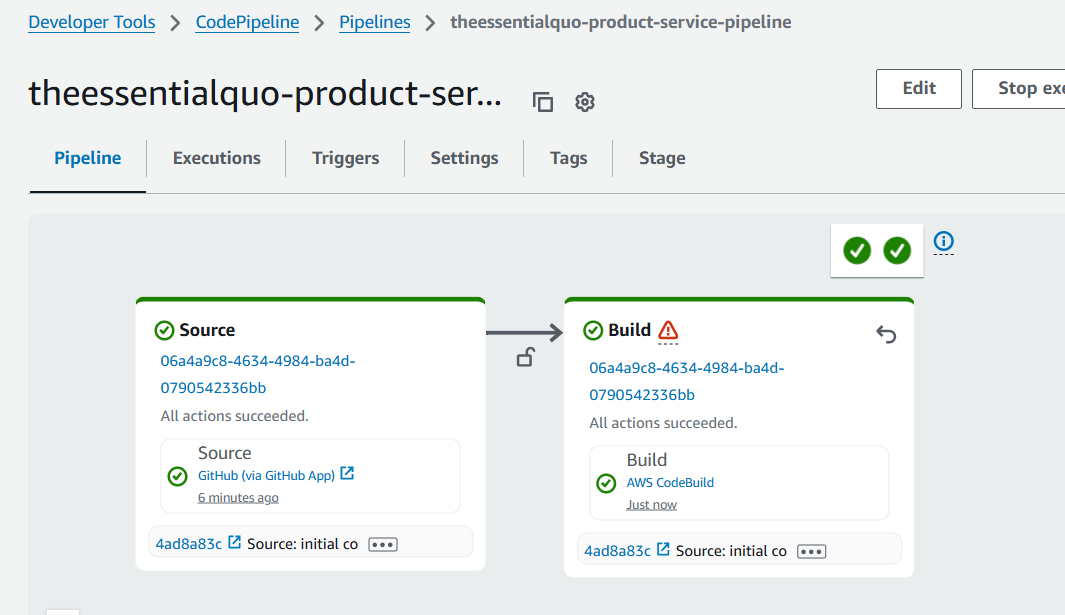


* Now choose the build provider, i.e. codebuild where we created our project.



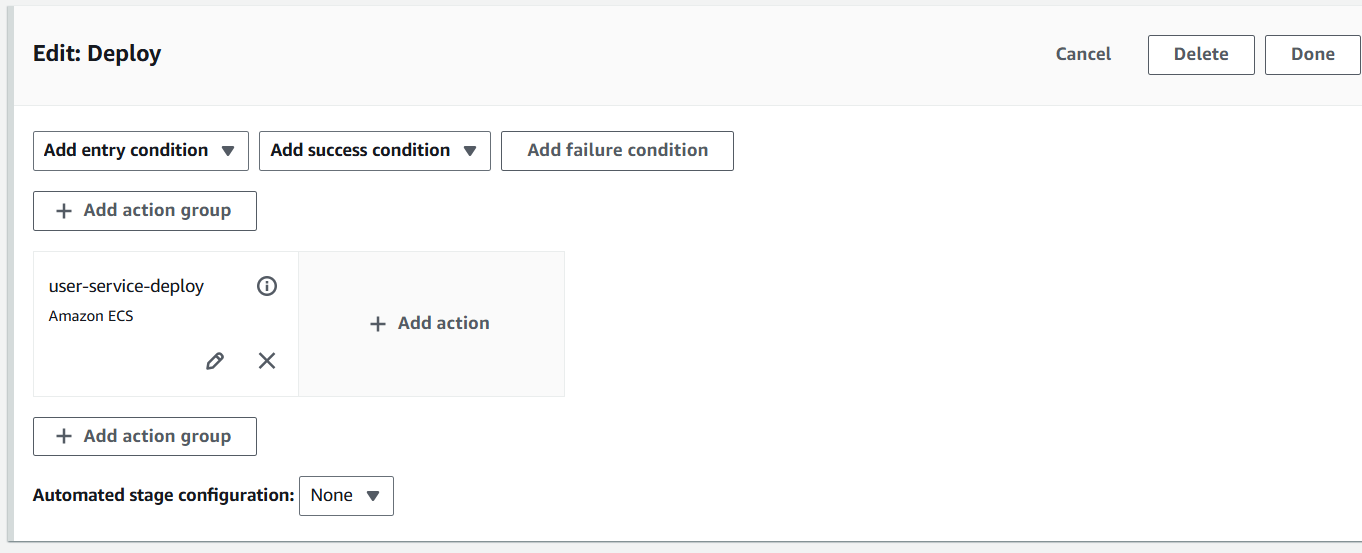
* Now skip to the final create pipeline because first we are only doing ci with code build, so we will add the codebuild only to the pipeline.  
  then run the pipeline.  
  

1. Similarly create a code build project for the theessentialquo-product-api-service, and create a pipeline for the same with codepipeline.

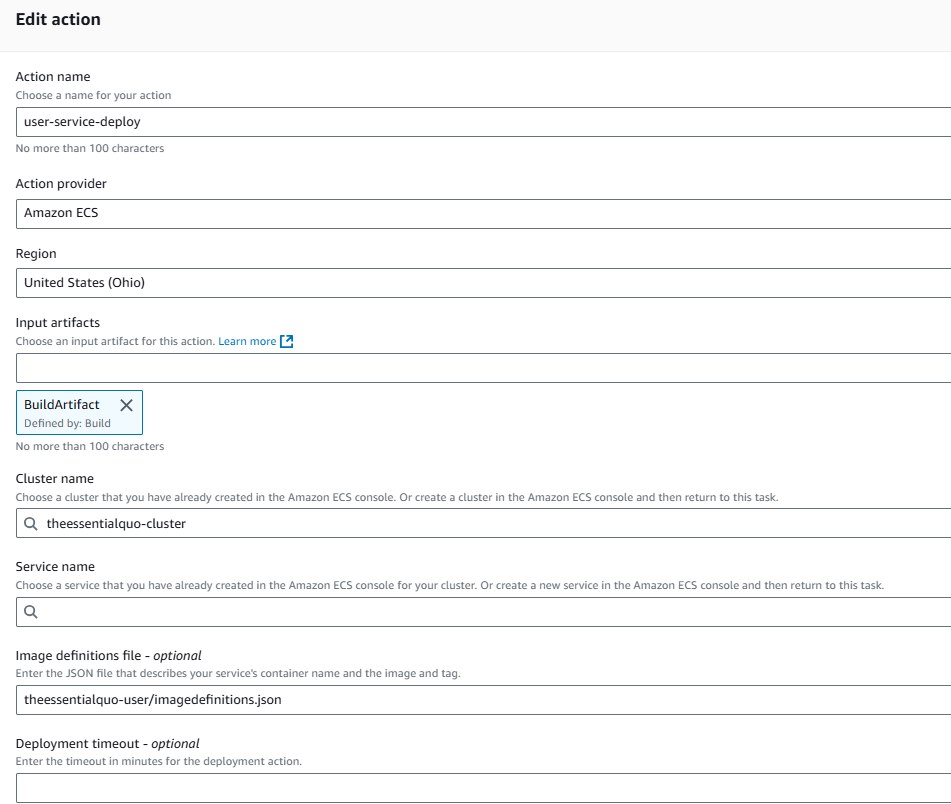


1. Implement AWS codedeploy

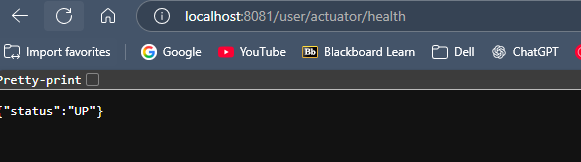
* Check the user-service created is running on ecs using the task definations.
* Now add a Deploy stage to the user-service pipeline.



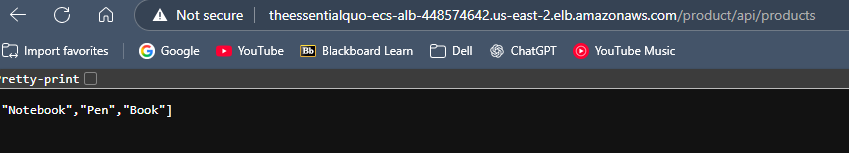
* In that add an action which will deploy the service running in the cluster. It needs the artifact from the build stage and the imagedefinitions.JSON file where the image id is stored. Make sure to give the correct path to the imagedination file.

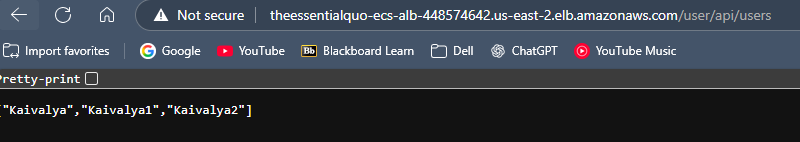


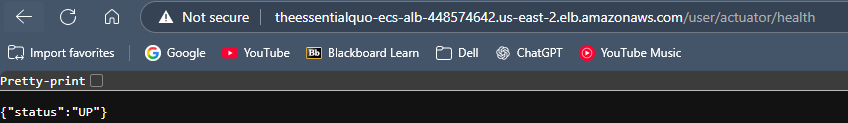
* After running the pipeline if the deploy stage is still in progress check for target group health checks. Here we are using spring boot app which needs the actuator dependency to reach to ‘/user/actuator/health’ path, to pass the health check.

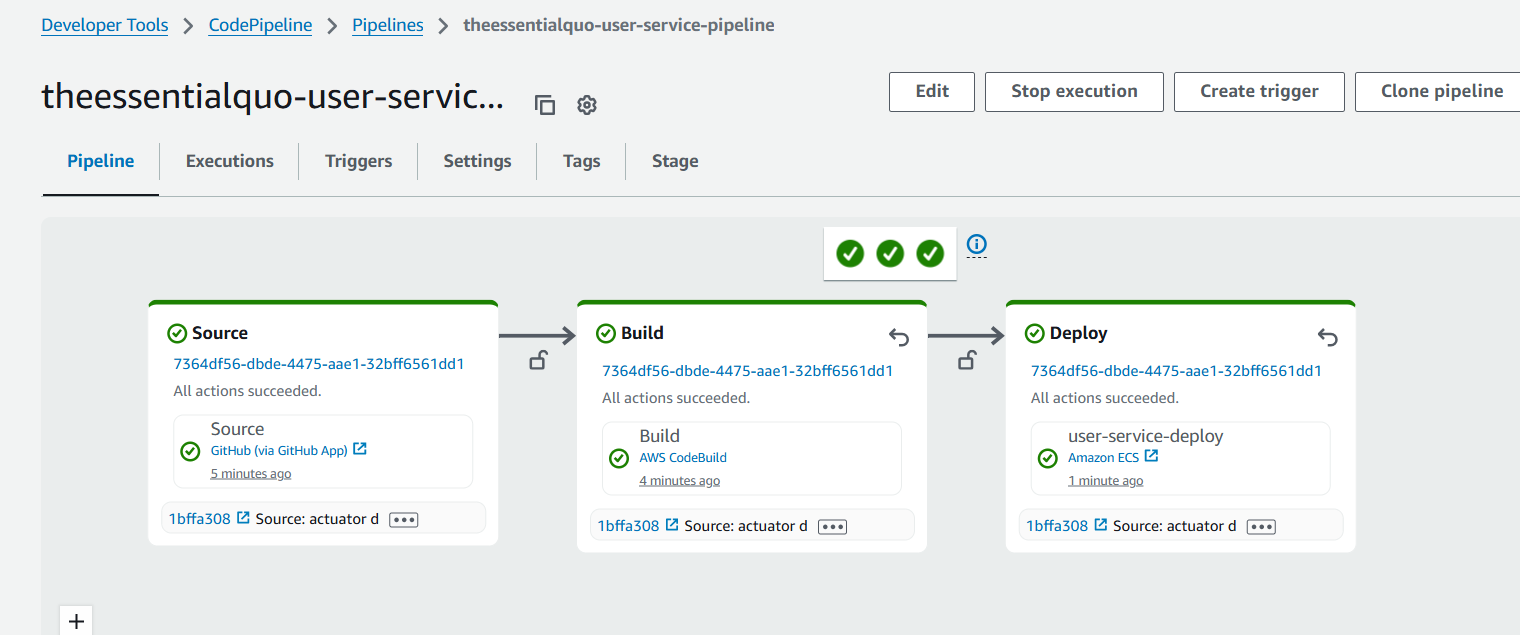


* After doing the necessary changes run the pipeline and check on the loadbalancer dns.









**10. Author**

**Kaivalya Shah**  
📧 kaivalyashah310@gmail.com  
🌐 LinkedIn  
💼 Passionate about cloud-native development, DevOps, and microservices architecture.

**Conclusion**

This project is a **realistic end-to-end pipeline** for deploying microservices with **high availability**, **zero downtime**, and **automated rollouts** using only **AWS managed services**.  
Feel free to clone, adapt, and improve!