DevOps Intern Exercise #3 | Deploy a Basic React App using GitHub Actions, Docker, and AWS ECS

Written by Danzig James A. Orcales on March 24, 2025

In this exercise, you'll orchestrate an automated deployment of a Dockerized React application to an AWS EC2 instance using a GitHub Actions pipeline. This hands-on guide will walk you through launching and configuring a virtual machine, uploading your Docker image to ECR, and running your application on the cloud. By completing this task, you'll build valuable skills in cloud infrastructure, container management, and secure deployment practices.

Table of Contents

Dockerize a React App

Vite Project

On github, create a new empty repo named react-app. On your local machine, execute these commands. I'm on Debian so I'll be using the apt package manager. During Vite initialization, name the project as react-app. Afterwards, you can access the app on http://localhost:5173

```
sudo apt update && \
sudo apt install docker-ce nodejs npm -y && \

npm create vite@latest -y && \
cd react-app && \
npm install && \

git init && \
git commit -am “rada rada” && \
git branch -M main
git remote add origin git@github.com:<USERNAME>/react-app.git && \
git push -u origin main && \


cat << EOF > Dockerfile
# Alpine is a lightweight Linux distro that reduces the size of the image.
FROM node:18-alpine

# Arbitrarily set the working directory inside the image to /app.
WORKDIR /app

# package-lock.json ensures consistent installation of dependencies by locking versions
# and preventing unexpected behavior during builds.
COPY package.json .
COPY package-lock.json .

# Install the project dependencies listed in package.json.
RUN npm install

# Copy all project files from your local working dir into the image's /app dir.
COPY . .

# Document that the application will use port 5173.
# EXPOSE doesn’t actually open the port but serves as metadata,
# informing users and tools which port the container listens on.
EXPOSE 5173

# Define the default command to run the Vite development server.
ENTRYPOINT ["npm","run","dev","--","--host","--strictPort"]
EOF


sudo docker build -t exercise/react-app . && \
sudo docker run --rm -p 5173:5173 --name react-app exercise/react-app
```

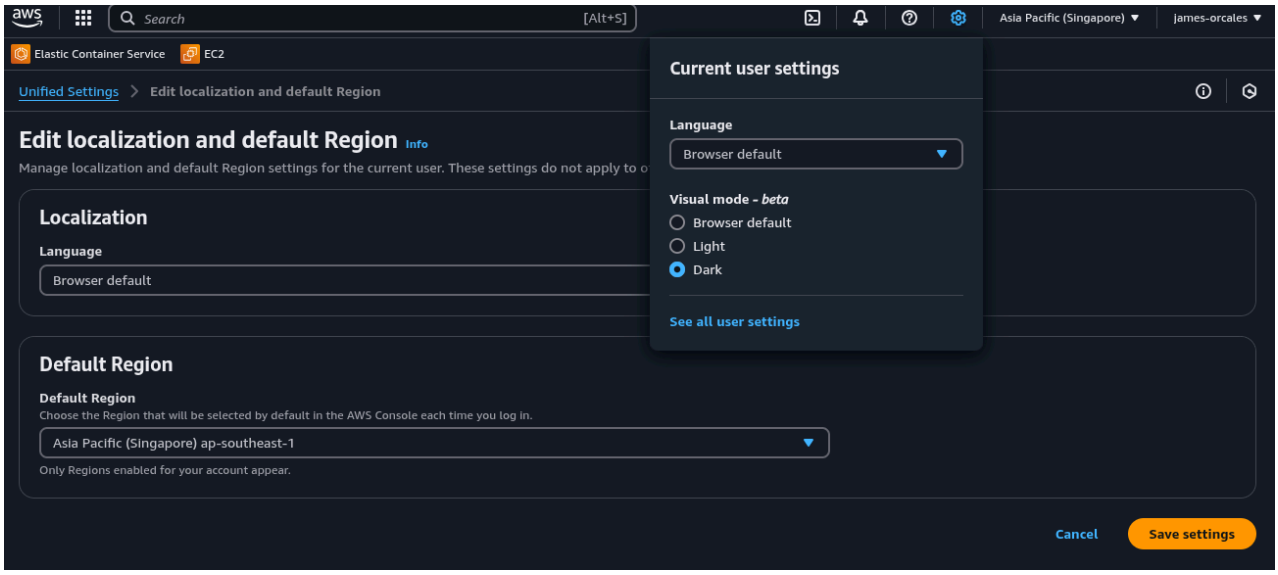
Set Up an EC2 Instance

Sign Up for an AWS Account

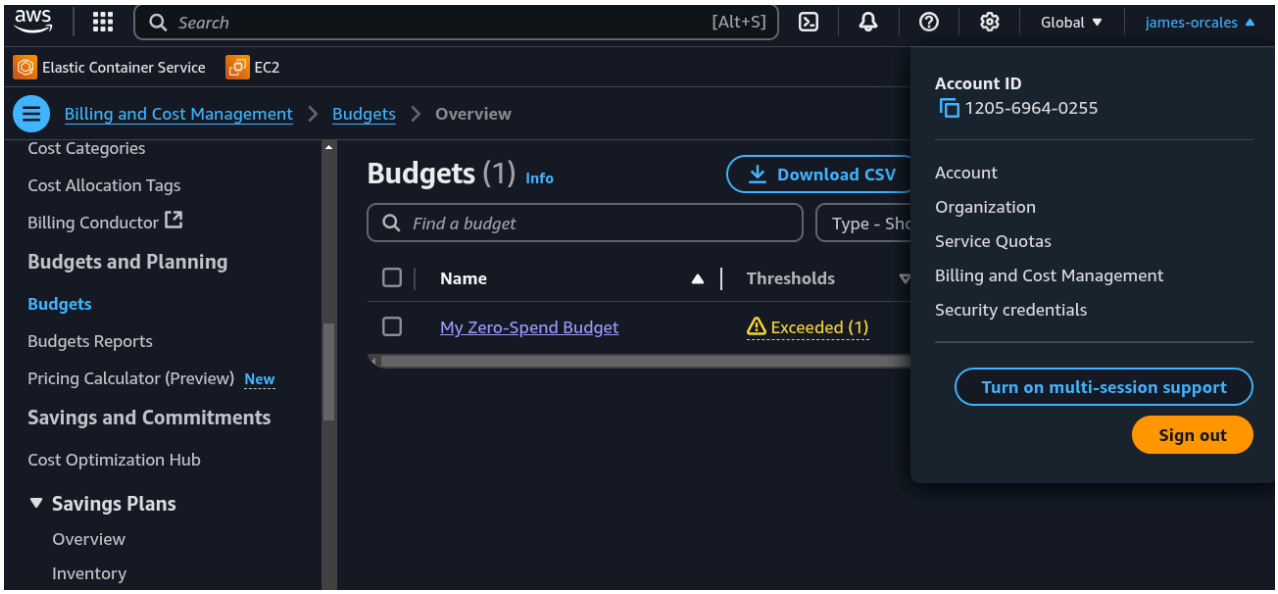
Sign up for a free-tier AWS account. If you need a credit card, you can use the Maya Virtual Card, which is easy to access. AWS requires at least $1 in your card for verification purposes. To fund your Maya card, transfer money from GCash using the phone number linked to your Maya account. When entering billing details on AWS, be sure to use the 16-digit number of your Maya Virtual Card.

Account Region and Zero-spend Budget

- Under user settings, set the account region to Singapore (ap-southeast-1).



- Under Billing and Cost Management > Budgets > Overview, create a new budget. Select the default “Zero Spend Budget” template. You will receive an email if any of your resources expenses exceed $0.01. **Note that services in AWS will not automatically shutdown when the budget is exceeded.** Though during this exercise, you may see fees incurred by the EC2 instance that we will launch, this is covered by the free-tier plan. You can verify that no funds are deducted by checking your bank account balance.

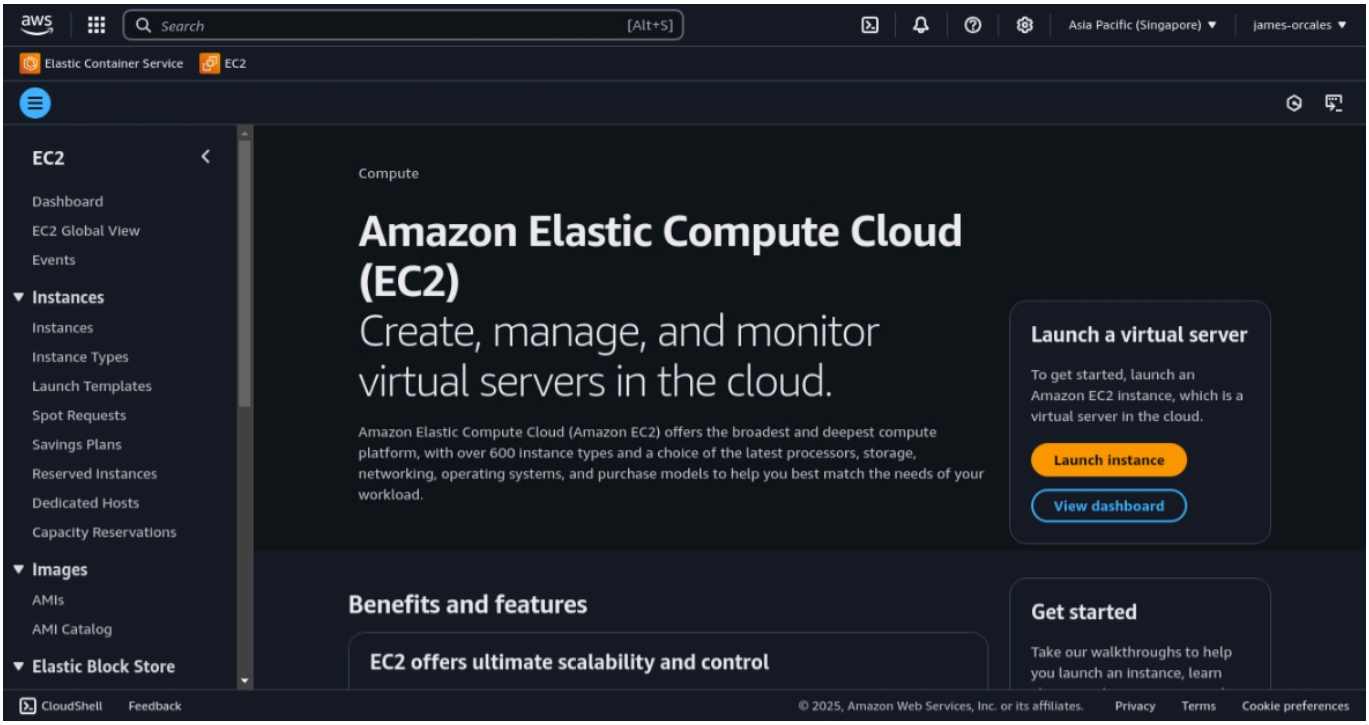


Launch an EC2 Instance

An Amazon Elastic Compute Cloud (EC2) instance is a virtual machine running on AWS infrastructure, acting like a computer you can configure and control remotely. Similar to leaving a physical laptop running continuously in your closet, an EC2 instance operates 24/7 unless stopped, consuming resources and incurring charges. However, unlike a physical device, EC2 offers flexible configuration for CPU, memory, and storage, and it's accessible globally via the internet, making it highly scalable and efficient for hosting applications or performing computations on demand.

In the AWS Console home, search for the EC2 service and launch an instance. Configure the instance as follows:

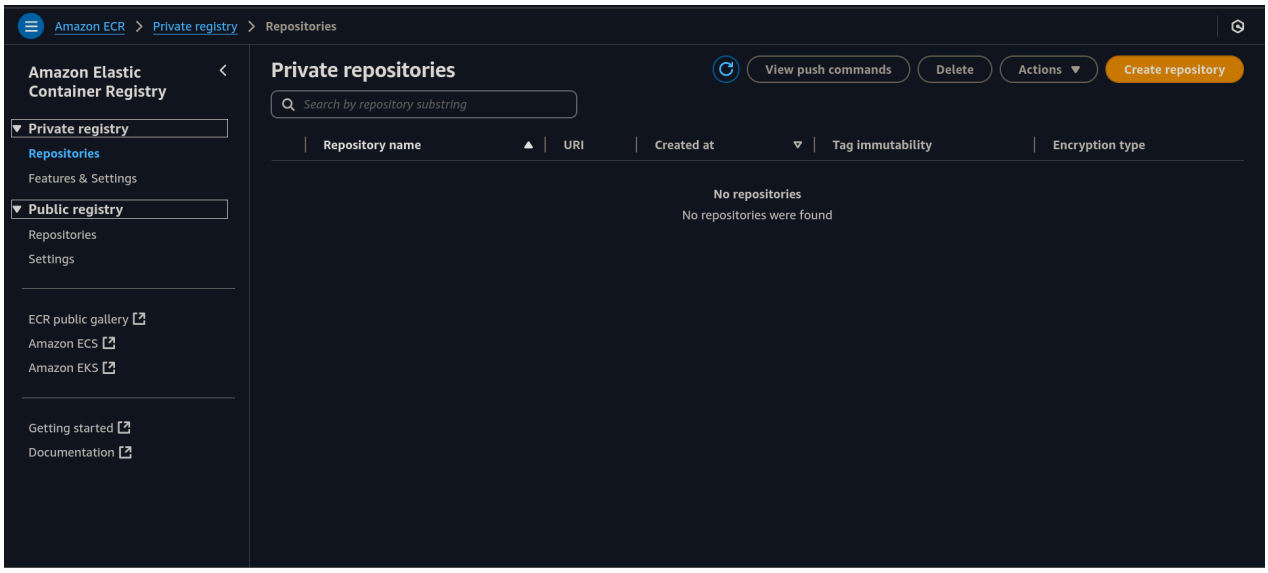
- **Amazon Machine Image**: Amazon Linux 2023 AMI 64-bit (x86)
- **Instance Type**: t2.micro
- **Create a new key pair** (if you don't have one already). This is what you use to log into the instance through SSH. Remember to save the file:
    - **Key pair type:** RSA
    - **Private key file format**: .pem
- **Create a security group.** This sets up the firewall for your instance.
    - **Allow SSH traffic from:** My IP
      Your IP would be the only one allowed to SSH into the instance.
    - **Allow HTTP traffic from the Internet**: true
      This is set to allow traffic from anywhere. We'll manually update the security group later to only allow HTTP connections from your IP.
- **Storage:** 1x8 GiB gp3
- Advanced configuration
    - **IAM Instance Profile:** ecsInstanceRole



ECR Registry and Cluster Setup

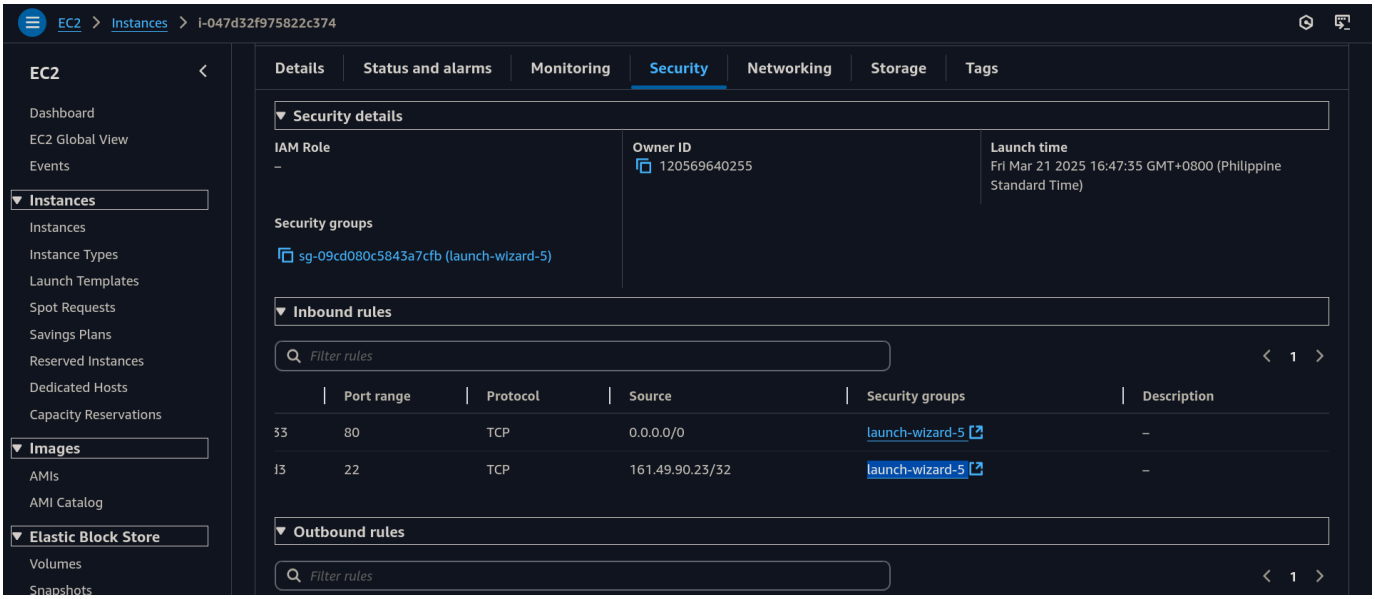
Create Repository

To deploy a Docker image to an EC2 instance, it needs to be accessible over the network. Uploading the image to a registry like Amazon Elastic Container Registry (ECR) allows EC2 and other AWS services to pull and use the image. Under ECR > Private Registry > Repositories, create a new repository with the name `exercise/react-app`.

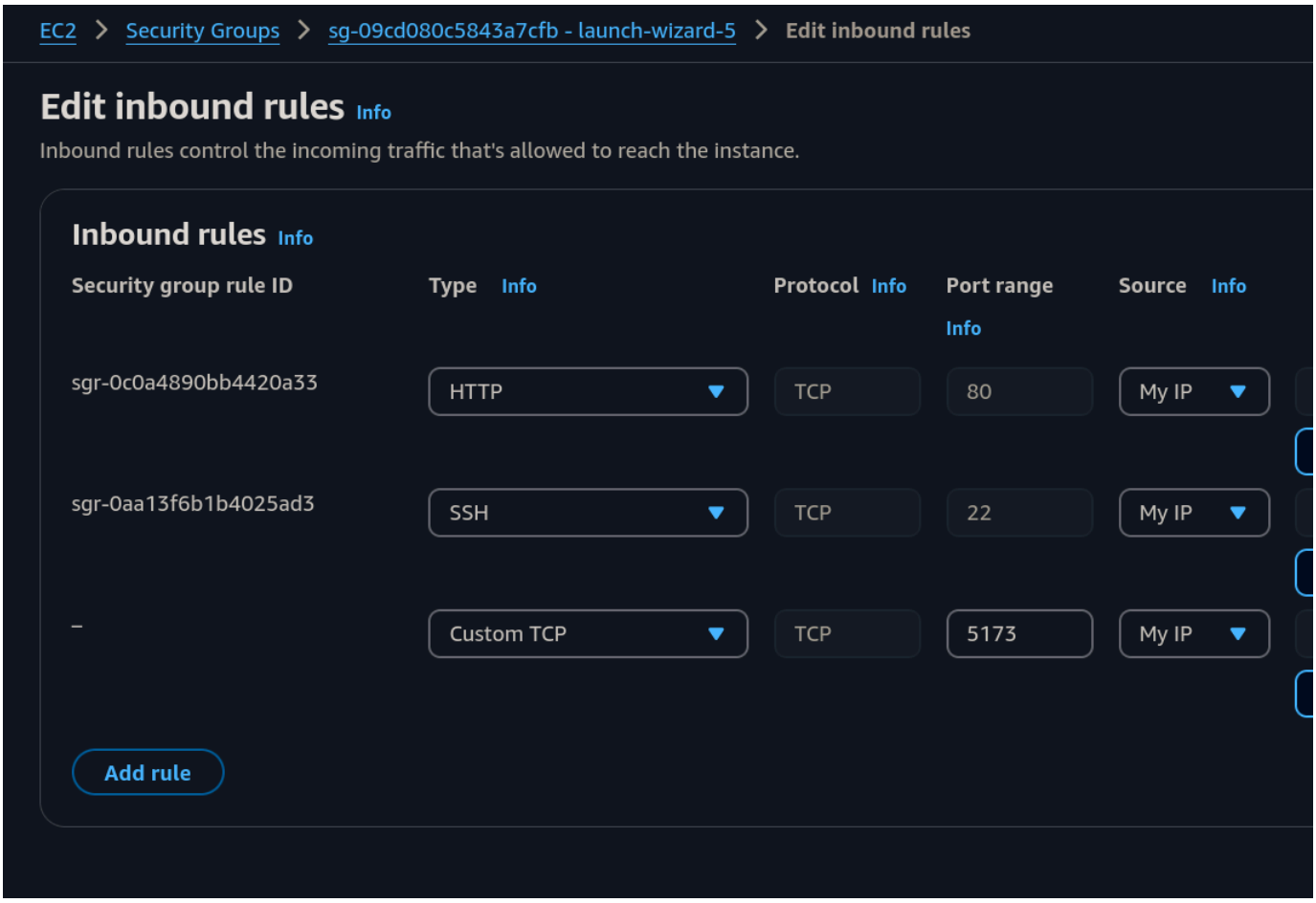


Modify Security Group

Under EC2 > Instances > [INSTANCE_ID] > Security > Inbound rules, click on the instance's associated security group.



Edit the inbound rules by adding a Custom TCP rule for port 5173. Set the rule's source to `My IP` to ensure access is restricted to the device you're currently using to access the AWS console. Keep in mind that this IP address is specific to your current network. Connecting to a different network will result in a different IP address. To find your current IP address, run `curl ifconfig.me` on your local machine.



Create a Cluster

Under ECR > Clusters > Create a Cluster, follow these configuration settings:

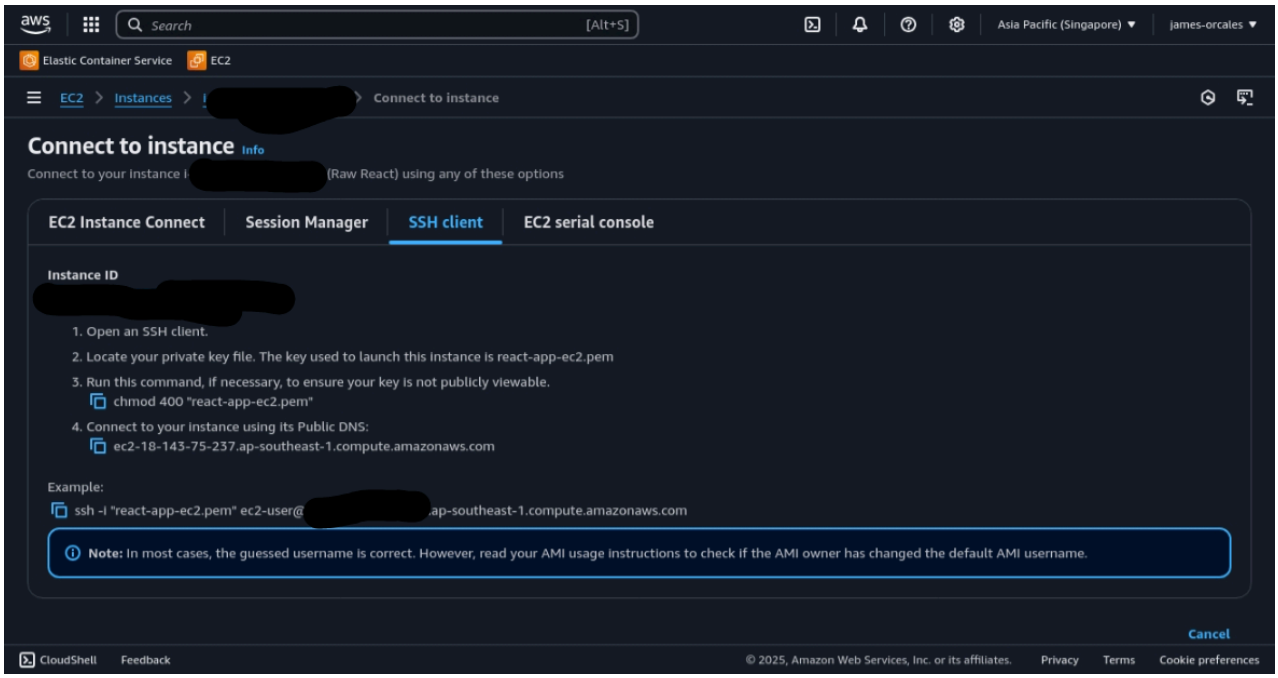
- **Name**: react-app-cluster
- Infrastructure
    - Amazon EC2 Instances only
    - **Provisioning Model:** Spot
    - **Allocation strategy:** Lowest Price
    - **Container Instance Amazon Machine Image (AMI):** Amazon Linux 2023
    - **EC2 Instance Type:** t2.micro
    - **EC2 Instance Role:** ecsInstanceRole
    - **Desired Capacity**
        - **Minimum:** 0
        - **Maximum:** 1
    - **SSH Key Pair:** Select the key pair we created when we launched an EC2 instance.
    - **Security group name:** Select the security group we created when we launched an EC2 instance.
    - **Auto-assign public IP:** Turn on

Register EC2 instance to the Cluster

In our setup, we opted for the standard Amazon Linux 2023 AMI, which does not come pre-optimized for ECS. This means that it lacks the pre-installed configurations and components specifically tailored for seamless integration with ECS clusters, such as the Amazon ECS container agent and Docker runtime. For scenarios requiring ECS-optimized features, AWS provides dedicated ECS-optimized AMIs, which can be sourced directly from the AMI Marketplace.

These AMIs are designed to simplify cluster registration and streamline container orchestration. **It is paramount that you follow this section sequentially.**

- Under EC2 > Instances, select your newly created instance and click connect. Execute the sample command as is to SSH into your instance.



- Execute these commands in the instance, **not** on your local machine.

```
sudo dnf update && \
sudo dnf install docker -y && \
sudo mkdir -p /etc/ecs && \
echo ECS_CLUSTER=react-app-cluster | sudo tee -a /etc/ecs/ecs.config > /dev/null &&
\
sudo curl -O
https://s3.ap-southeast-1.amazonaws.com/amazon-ecs-agent-ap-southeast-1/amazon-ecs-i
nit-latest.x86_64.rpm && \
sudo dnf localinstall -y amazon-ecs-init-latest.x86_64.rpm
```

Edit the `/lib/systemd/system/ecs.service` file and add the following line at the end of the `[Unit]` section. Refer to Installing the Amazon ECS container agent for more info.

```
After=cloud-final.service
```

Lastly, start the service.

```
sudo systemctl enable --now ecs
```

GitHub Actions CI/CD Pipeline

Task Definition Template

In the root directory of your project, create a file named `aws_task_definition.json`. This file acts as the base configuration for the `aws-actions/amazon-ecs-render-task-definition@v1` GitHub Action.

When your workflow includes optional parameters, they will be dynamically appended to this JSON configuration during execution. However, note that these changes will not modify the original `aws_task_definition.json` file; the updated configuration exists temporarily for the runtime.:

```
{
        "family": "react-app-task",
        "networkMode": "bridge",
        "containerDefinitions": [
                {
                        "name": "react-app-container”,
                        "portMappings": [
                                {
                                        "containerPort": 5173,
                                        "hostPort": 5173,
                                        "protocol": "tcp",
                                        "name": "vite-port",
                                        "appProtocol": "http"
                                }
                        ],
                        "essential": true
                }
        ],
        "cpu": "300",
        "memory": "400"
}
```

Workflow File

In your project root dir, create `./.github/workflows/deploys.yml` with the following contents below. This GitHub Actions workflow automates deploying a React app to Amazon ECS. It triggers on changes to the `main` branch, builds a Docker image, pushes it to a private ECR repository, and creates a new ECS task definition revision. The workflow then deploys the updated task definition to ECS.

Read more about Workflow syntax for GitHub Actions.
Read more about Accessing contextual information about workflow runs
To see available actions, refer to github.com/actions and github.com/aws-actions. Read the `action.yml` in the respective action repos to see their inputs and outputs:

```
name: Deploy React App to ECS
on:
  push:
    branches:
```

```
      - main
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - name: Git Checkout
        uses: actions/checkout@v4
      - name: AWS CLI Authentication
        uses: aws-actions/configure-aws-credentials@v4
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ap-southeast-1
      - name: ECR Login
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@v2
      - name: Build and Deploy image to ECR Private
        env:
          REGISTRY: ${{ steps.login-ecr.outputs.registry }}
          REPOSITORY: exercise/react-app
          IMAGE_TAG: ${{ github.sha }}
        run: |
          docker build -t $REGISTRY/$REPOSITORY:$IMAGE_TAG .
          docker push $REGISTRY/$REPOSITORY:$IMAGE_TAG
      - name: Create ECS Task Definition Revision
        id: task-revision
        uses: aws-actions/amazon-ecs-render-task-definition@v1
        with:
          task-definition: aws_task_definition.json
          task-definition-family: react-app-task
          container-name: react-app-container
          image: ${{ steps.login-ecr.outputs.registry }}/exercise/react-app:${{ github.sha
}}
      - name: Deploy ECS Task Definiton
        uses: aws-actions/amazon-ecs-deploy-task-definition@v2
        with:
          task-definition: ${{ steps.task-revision.outputs.task-definition }}
          service: react-app-service
          cluster: react-app-cluster
          force-new-deployment: true
```

Repo Secrets

In the Repository Settings, go to Secrets and Variables > Actions and add two repository secrets named `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`. These access tokens can be obtained by creating root access keys in your AWS account. To do this, log in to the AWS Management Console, navigate to My Security Credentials, and generate new access keys under the Access Keys section. Keep these keys secure and only use them in trusted workflows, as root access provides unrestricted permissions.

Initialization

Execute the following commands. This will immediately trigger the action:

```
git commit -am "added workflows" && \
git push
```

The GitHub action **will fail** because we haven't created a service yet. We triggered the github action here to upload our first image and create our first task definition. We could have also done everything once manually to avoid the error.

Create a Service

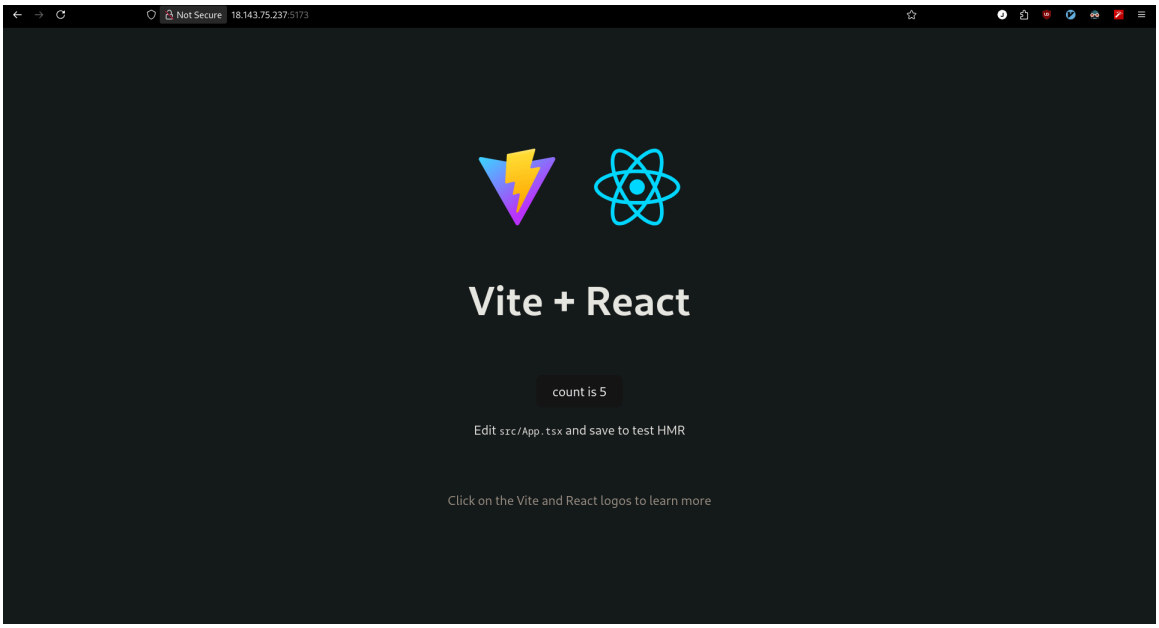
Under ECS > Clusters > react-app-cluster > Services > Create service, use the following configuration:

- **Launch Type:** EC2
- **Deployment configuration:** Service
- **Task definition family:** react-app-container
- **Service name:** react-app-service
- **Min running tasks %:** 0
- **Max running tasks %:** 100
- **Turn on Availability Zone rebalancing:** false
- Networking
    - **Security group name:** Use the security group we created when we launched an EC2 instance.

Conclusion

Congratulations! You can now access your web app. Try editing `./src/App.jsx` and pushing a new commit. Your website should update after a few seconds.

This exercise demonstrates the process of deploying a Dockerized React application to an AWS EC2 instance. By following the outlined steps, you gain hands-on experience with AWS account setup, EC2 configuration, Containerization, and application deployment. These skills are valuable for building scalable and accessible web applications, preparing you for real-world DevOps challenges. Successfully completing this task showcases your ability to combine technical proficiency with cloud computing to deliver effective solutions.



Troubleshooting

Service Deployment Failure

Under ECR > Clusters > [CLUSTER_NAME] > Services > [SERVICE_NAME] > Deployments > Events, you can check the sequential steps of the deployment to see what was wrong. You can see error messages here such as the EC2 instance not meeting the hardware requirements of its tasks or the ECS agent being disconnected which involves revisioning the task or connecting via SSH to the EC2 instance and configuring it.

ECS Agent

If the EC2 instance is failing to register with your cluster or you're experiencing errors related to the ECS Agent, connect to the EC2 instance via SSH and check the agent service logs under `/var/log/ecs`