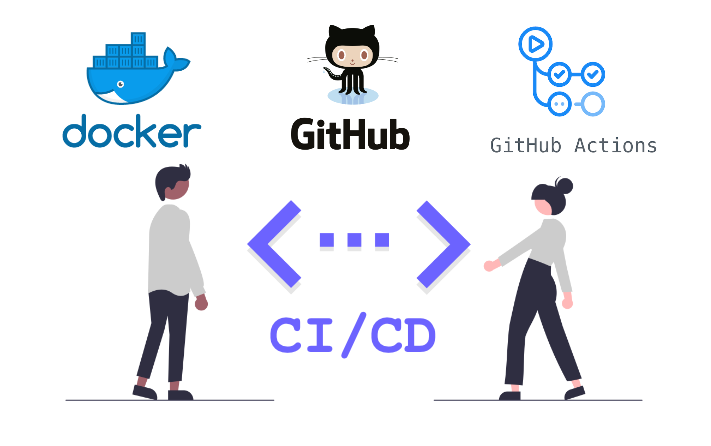
**Building a CI/CD Pipeline with GitHub Actions and Docker (Part 1)**

**A guide to develop a workflow engine to build and conduct automated testing on your microservices**



**Introduction**

Continuous Integration is the first step to CD where it runs automated tests in the main code branch to ensure that bugs are detected early and prevent the system from breaking when new codes are integrated.

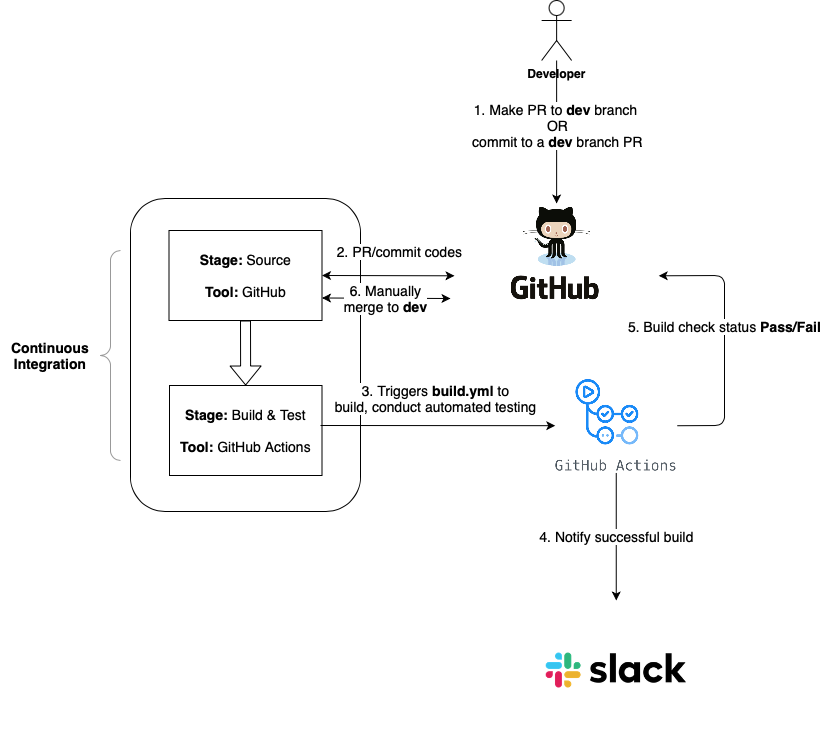
In this article, I will provide a guide on how you can adopt continuous integration and automated testing with microservices and GitHub Actions CI-docker workflow.

**How it works**

Microservices are containerized using Docker thus, we develop a CI-docker workflow to help us build and conduct automated testing.

The CI solution contains 2 stages:

1. **Source**   
   In this stage, developers commit and push changes to GitHub
2. **Build & Test**   
   In this stage, a workflow engine is triggered when a pull request (PR) to **develop** a branch is made OR code is pushed to a **develop** branch PR.   
   The workflow engine would build and spin up the docker containers and automated testing will be conducted by running through the test cases that were created.   
   Once the workflow has been completed, a notification will be sent to slack indicating the status of the build.



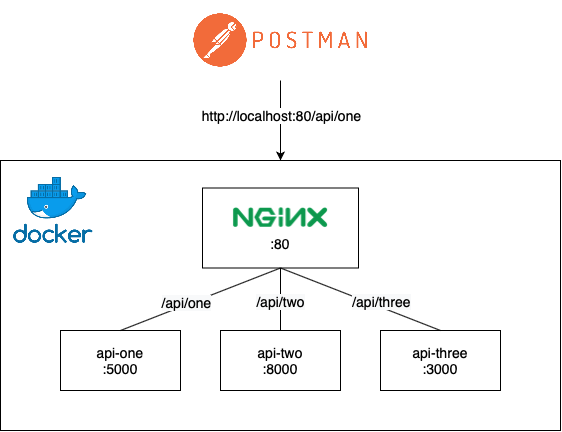
CI Workflow

**Overview of Testing System and Architecture**

Before we move on to how the workflow is set up, I’ll provide a brief overview of the testing system and architecture to help you better understand the rationale behind the following steps.

**NGINX Proxy Server**

A proxy server acts as an API gateway that allows the outside world to reach the microservices via the proxy server by upstreaming the requests to the respective microservice. It can be set up using NGINX docker image in your docker compose file.



NGINX as API Gateway

**MongoDB as database**

For testing database, mongo is set up in docker compose file as well using mongo docker image. This ensures that the development or production database will not be diluted with testing data.

**JEST**

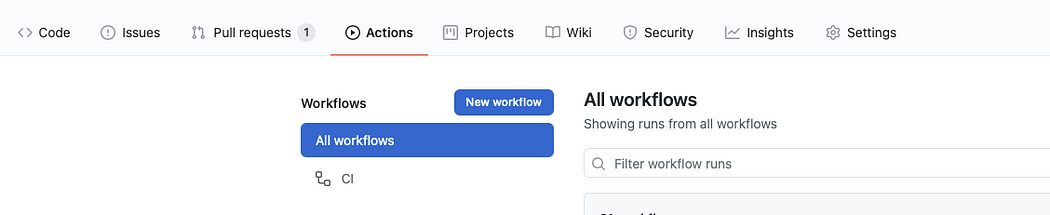
JEST is a Javascript testing tool used to develop API test suites together with Frisby.

**Setting up a workflow with GitHub Actions**

Now that you have gotten a gist of how the CI pipeline works, let’s delve deeper into the set up!

**Creating a workflow**

Head over to GitHub Actions to create a new workflow where you will be asked to create a workflow template and select **Simple workflow (actions/starter-workflows)**. Follow the steps given by GitHub and you should see a .github/workflows folder created in your repository. I have named my workflow file as build.yml .



Creating a workflow

**Set up runner**

Your build.yml file contains the steps required for the workflow. As we would be utilising docker commands to spin up the containers, we need to run the environment using an image lucasalt/act\_base created by [LucasC](https://github.com/LucasCtrl/act_base)trl. Without this, we will not be able to run docker-compose which is required to spin up the containers. Next, include path as the name of your repository.

name: CI-dev-pipeline  
on:  
 # Triggers the workflow on push or pull request events but only for the develop branch  
 push:  
 branches: [ develop ]  
 pull\_request:  
 branches: [ develop ] # to change to develop  
jobs:  
 build:  
 # The type of runner that the job will run on  
 runs-on: ubuntu-latest  
 container:  
 image: lucasalt/act\_base:latest  
 steps:  
 # Checks-out your repository under $GITHUB\_WORKSPACE, so your job can access it  
 - uses: actions/checkout@v2  
 with:  
 path: "<INSERT\_NAME\_OF\_REPOSITORY>" - name: RunOne  
 run: |  
 echo Hello, world!

**Set up GitHub Actions Locally**

**Step 1: Installing Act**

Due to the limited build minutes we have on GitHub actions, it’s always good to conduct a test run of the workflow locally. We can do so by installing [act](https://github.com/nektos/act) to run GitHub actions locally.

**Step 2: Obtaining GitHub Token**

Before we run act, we need to obtain a github token for authentication purposes. Head over to ***Settings > Developer Settings*** ***> Personal access tokens*** to generate a github token if you do not have one.

**Step 3: Running act**

After installing, use the GitHub token that you have obtained previously and run the following command in your terminal

act -s GITHUB\_TOKEN=<YOUR\_GITHUB\_TOKEN>

You will be given a choice to select the type of docker image. For testing purposes, we will select micro. Once selected, it will run your workflow accordingly. By default, it triggers push event workflow. If successful, you should be able to see **"Hello world"** as we have indicated in the previous section.

**Setting up Build Stage**

Now that you’ve gotten your GitHub Actions to work locally, use it as much as you can for testing before pushing to GitHub.

**Build docker containers**

In your build.yml, run your docker compose file to start building the containers in GitHub Actions environment which allows us to run our automated tests against. I have used docker-compose.test.yml to build containers in a test environment locally, you can replace it with the docker compose file that you have set up for your containers.

- name: Build docker  
 run: |   
 echo ---Building and starting up docker---  
 docker-compose -f ./<REPO\_NAME>/docker-compose.test.yml -p testing up -d  
 echo ---Containers up---

**Setting up Test stage**

As Github Actions runs the workflow in a user-defined network, we need to containerize the testing folder to be in the same network as other microservices for communication to happen. Containers in separate networks are unable to communicate thus, API test cases will not be able to get a response from the microservices. Read more about docker networking [here](https://docs.docker.com/network/network-tutorial-standalone/).

**Containerizing testing folder**

Create adocker-compose.test-github.yml file to spin up api-test container. API\_URL is set to the container name of your proxy server, in this case, I’m using proxy. Similarly for MONGO\_URL.

version: '3'  
services:  
 api-test:  
 container\_name: api-test  
 build:  
 dockerfile: Dockerfile  
 context: ./testing  
 command:  
 bash -c "npm run test; npm run test-failure"  
 networks:  
 - <YOUR\_USER\_DEFINED\_NETWORK>  
 environment:  
 - API\_URL=http://proxy:80  
 - MONGO\_URL=mongodb://mongo:27017/<DB\_NAME>  
 ports:  
 - '4000:4000'

Modify the API URL and database URL in your config file accordingly to allow test cases to run locally and in a docker container.

// testing/config.js  
exports.config = {  
 "url": process.env.API\_URL || "http://localhost:80",  
 "mongoClient": process.env.MONGO\_URL || "mongodb://localhost:27017/<DB\_NAME>",  
};

A Dockerfile is also created in testing folder to build api-test container, install relevant packages, and exposing them as port 4000.

FROM node:10  
  
RUN mkdir -p /usr/src/app  
WORKDIR /usr/src/appCOPY package.json /usr/src/appRUN npm install  
RUN npm install jest --global   
RUN npm audit; exit 0 && npm audit fixCOPY . /usr/src/appEXPOSE 4000

**Running test suites with docker**

We will now modify the build.yml to spin up the api-test container using docker-compose.test-github.yml that we have created previously. Test cases will run upon starting up the container.

- name: Run test  
 run: |  
 echo --- Running test cases ---  
 docker-compose -f ./<REPO\_NAME>/docker-compose.test-github.yml -p testing up --build --exit-code-from api-test  
 echo --- Completed test cases ---

**Feedback using Slack Notifications**

**Step 1: Create a Slack app**

Firstly, we would need to generate a SLACK\_WEBHOOK\_URL to send a notification to your Slack channel. Slack provides a seamless experience for you to create a Slack app, you can start creating one [here](https://api.slack.com/apps).

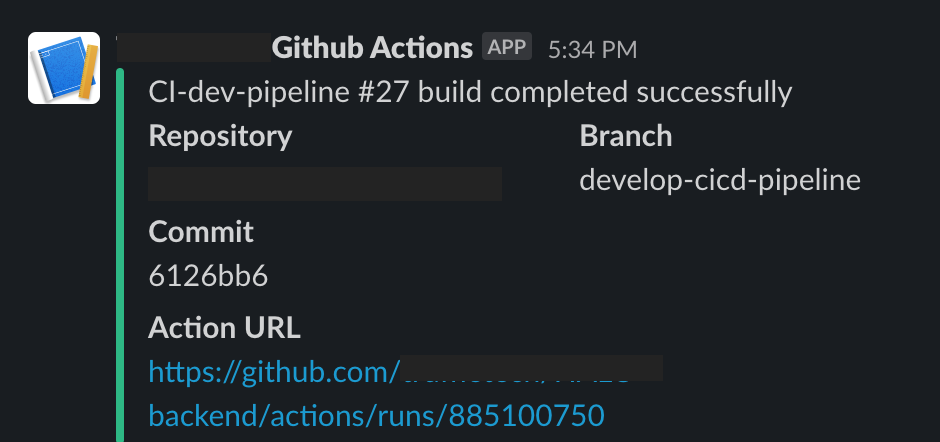
**Step 2: Store webhook URL in GitHub Secrets**

Once you have generated a SLACK\_WEBHOOK\_URL , add them into your **GitHub secrets** under settings.

**Step 3: Modify build.yml**

Using [simple-slack-notify@master](https://github.com/edge/simple-slack-notify), we can send a success/failed notification to Slack while including additional details such as branch name, repository name etc. with [GitHub’s environment variables.](https://docs.github.com/en/actions/reference/environment-variables)

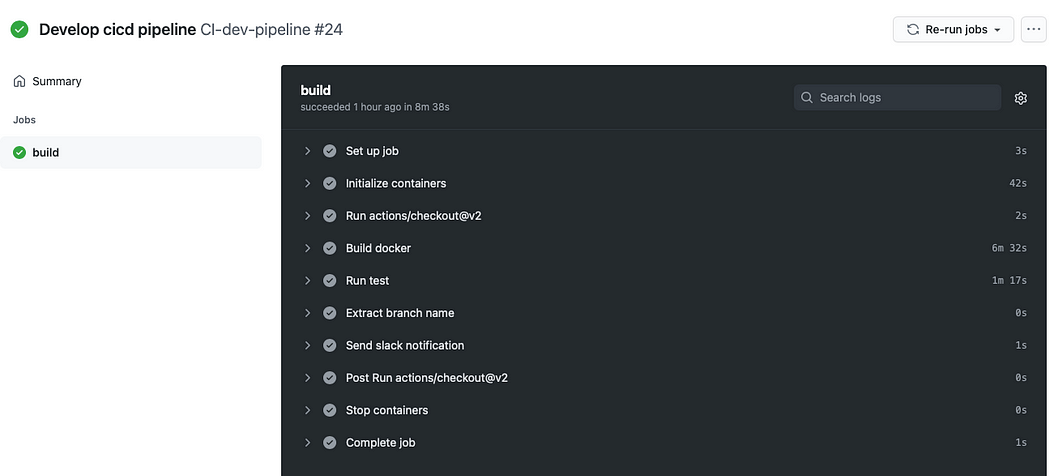
- name: Send slack notification  
 if: always()  
 uses: edge/simple-slack-notify@master   
 env:  
 SLACK\_WEBHOOK\_URL: ${{ secrets.SLACK\_WEBHOOK\_URL }}  
 with:  
 channel: '#deployment'  
 status: ${{ job.status }}  
 success\_text: Build completed successfully  
 failure\_text: 'Build failed'  
 cancelled\_text: 'Build was cancelled' fields: |  
 [{ "title": "Repository", "value": "${env.GITHUB\_REPOSITORY}", "short": true },  
 { "title": "Branch", "value": "${env.BRANCH}", "short": true },  
 { "title": "Action URL", "value": "${env.GITHUB\_SERVER\_URL}/${env.GITHUB\_REPOSITORY}/actions/runs/${env.GITHUB\_RUN\_ID}"}]



Sample Screenshot of Build Success Status Notification

**Putting it all together**

Once you have made a pull request or a push to dev branch, you can view the workflow in action by heading to **Actions** tab in your GitHub Repo.



Workflow in action!

Here is the gist of how we have set up build.yml so far!

|  |
| --- |
| name: CI-dev-pipeline |
|  |

|  |
| --- |
| on: |
|  |

|  |
| --- |
| # Triggers the workflow on push or pull request events but only for the develop branch |
|  |

|  |
| --- |
| push: |
|  |

|  |
| --- |
| branches: [ develop ] |
|  |

|  |
| --- |
| pull\_request: |
|  |

|  |
| --- |
| branches: [ develop ] |
|  |

|  |
| --- |
| jobs: |
|  |

|  |
| --- |
| build: |
|  |

|  |
| --- |
| # The type of runner that the job will run on |
|  |

|  |
| --- |
| runs-on: ubuntu-latest |
|  |

|  |
| --- |
| container: |
|  |

|  |
| --- |
| image: lucasalt/act\_base:latest |
|  |

|  |
| --- |
| steps: |
|  |

|  |
| --- |
| # Checks-out your repository under $GITHUB\_WORKSPACE, so your job can access it |
|  |

|  |
| --- |
| - uses: actions/checkout@v2 |
|  |

|  |
| --- |
| with: |
|  |

|  |
| --- |
| path: "<INSERT\_NAME\_OF\_REPOSITORY>" |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| - name: Build docker |
|  |

|  |
| --- |
| run: | |
|  |

|  |
| --- |
| echo ---Building and starting up docker--- |
|  |

|  |
| --- |
| docker-compose -f ./<REPO\_NAME>/docker-compose.test.yml -p testing up -d |
|  |

|  |
| --- |
| echo ---Containers up--- |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| - name: Run test |
|  |

|  |
| --- |
| run: | |
|  |

|  |
| --- |
| echo --- Running test cases --- |
|  |

|  |
| --- |
| docker-compose -f ./<REPO\_NAME>/docker-compose.test-github.yml -p testing up --build --exit-code-from api-test |
|  |

|  |
| --- |
| echo --- Completed test cases --- |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| - name: Extract commit |
|  |

|  |
| --- |
| shell: bash |
|  |

|  |
| --- |
| run: | |
|  |

|  |
| --- |
| echo "GITHUB\_SHA\_SHORT=$(echo $GITHUB\_SHA | cut -c1-7)" >> $GITHUB\_ENV |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| - name: Get branch name (merge) |
|  |

|  |
| --- |
| if: github.event\_name != 'pull\_request' |
|  |

|  |
| --- |
| shell: bash |
|  |

|  |
| --- |
| run: echo "BRANCH=$(echo ${GITHUB\_REF#refs/heads/} | tr / -)" >> $GITHUB\_ENV |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| - name: Get branch name (pull request) |
|  |

|  |
| --- |
| if: github.event\_name == 'pull\_request' |
|  |

|  |
| --- |
| shell: bash |
|  |

|  |
| --- |
| run: echo "BRANCH=$(echo ${GITHUB\_HEAD\_REF} | tr / -)" >> $GITHUB\_ENV |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| - name: Send slack notification |
|  |

|  |
| --- |
| if: always() |
|  |

|  |
| --- |
| uses: edge/simple-slack-notify@master |
|  |

|  |
| --- |
| env: |
|  |

|  |
| --- |
| SLACK\_WEBHOOK\_URL: ${{ secrets.SLACK\_WEBHOOK\_URL }} |
|  |

|  |
| --- |
| with: |
|  |

|  |
| --- |
| channel: '#deployment' |
|  |

|  |
| --- |
| status: ${{ job.status }} |
|  |

|  |
| --- |
| success\_text: '${env.GITHUB\_WORKFLOW} #${env.GITHUB\_RUN\_NUMBER} build completed successfully' |
|  |

|  |
| --- |
| failure\_text: '${env.GITHUB\_WORKFLOW} #${env.GITHUB\_RUN\_NUMBER} build failed' |
|  |

|  |
| --- |
| cancelled\_text: '${env.GITHUB\_WORKFLOW} #${env.GITHUB\_RUN\_NUMBER} build was cancelled' |
|  |

|  |
| --- |
| fields: | |
|  |

|  |
| --- |
| [{ "title": "Repository", "value": "${env.GITHUB\_REPOSITORY}", "short": true }, |
|  |

|  |
| --- |
| { "title": "Branch", "value": "${env.BRANCH}", "short": true }, |
|  |

|  |
| --- |
| { "title": "Commit", "value": "${env.GITHUB\_SHA\_SHORT}", "short": true }, |
|  |

{ "title": "Action URL", "value": "${env.GITHUB\_SERVER\_URL}/${env.GITHUB\_REPOSITORY}/actions/runs/${env.GITHUB\_RUN\_ID}"}]

# Building a CI/CD Pipeline with GitHub Actions and Docker (Part 2)

## A guide to deploying Microservices to AWS Elastic Container Registry (ECR) and AWS Elastic Container Service (ECS) Fargate with GitHub Actions

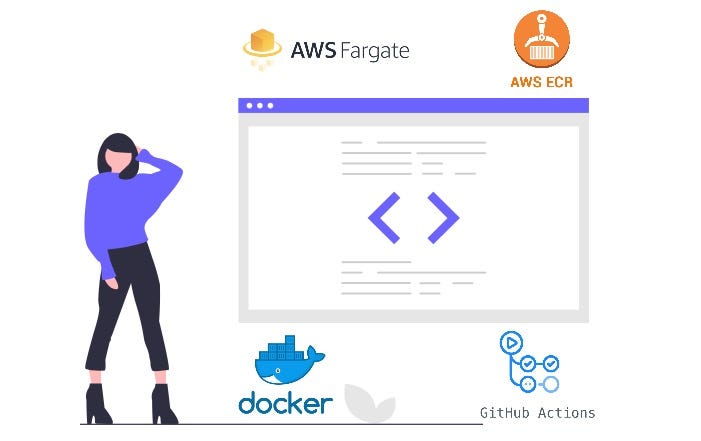


Photo By Author

# Introduction

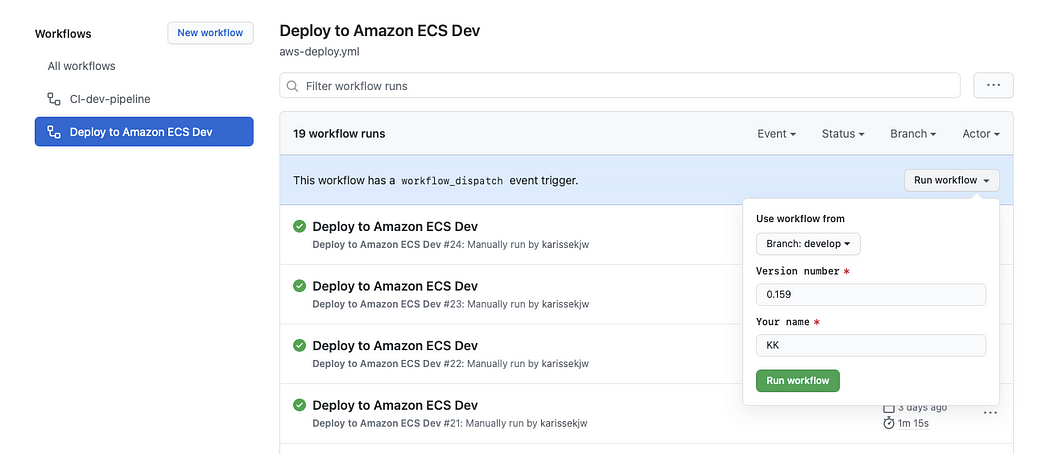
Continuous Integration and Continuous Delivery are essential to achieving a more efficient process of rolling out features to the consumers of your application. We may encounter situations whereby there are multiple pull requests undergoing automated testing. A fully automated deployment process would deploy each pull request (PR) once it has passed the automated testing. However, if you are deploying docker images, it could be a time consuming process which would be wasteful on our free build minutes that GitHub has given us especially if you are on a tight budget (like me!). In situations like this, how can we deploy the features to the development environment once all the pull requests have been merged safely?

GitHub Action’s workflow dispatch does exactly the trick for us. In this article, I will share how workflow dispatch could help push Docker images to AWS Elastic Container Registry (ECR) and deploying them to AWS Elastic Container Service (ECS) Fargate seamlessly, while managing the version of the application. If you have not read how we can adopt continuous integration and automated testing with GitHub Actions, you can read all about it in Part 1 of this [article](https://faun.pub/building-a-ci-cd-pipeline-with-github-actions-and-docker-part-1-a9d8709c31fb).

# How it works?

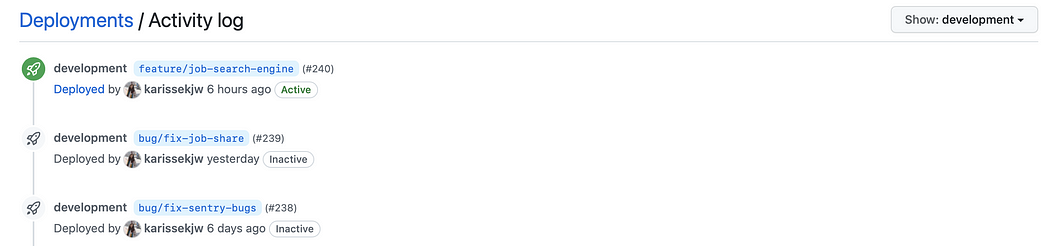
## Overview

We will be leveraging GitHub Action’s workflow dispatch whereby we can trigger the deployment process manually once all the PRs have been merged. Another advantage it has is it allows us to customise input information which in this case, could help us manage the versioning of the application with ease. Once we run the workflow, it will kickstart the deployment process and all we need to do next is to wait for it to be done!



Screenshot of GitHub Actions Workflow Log

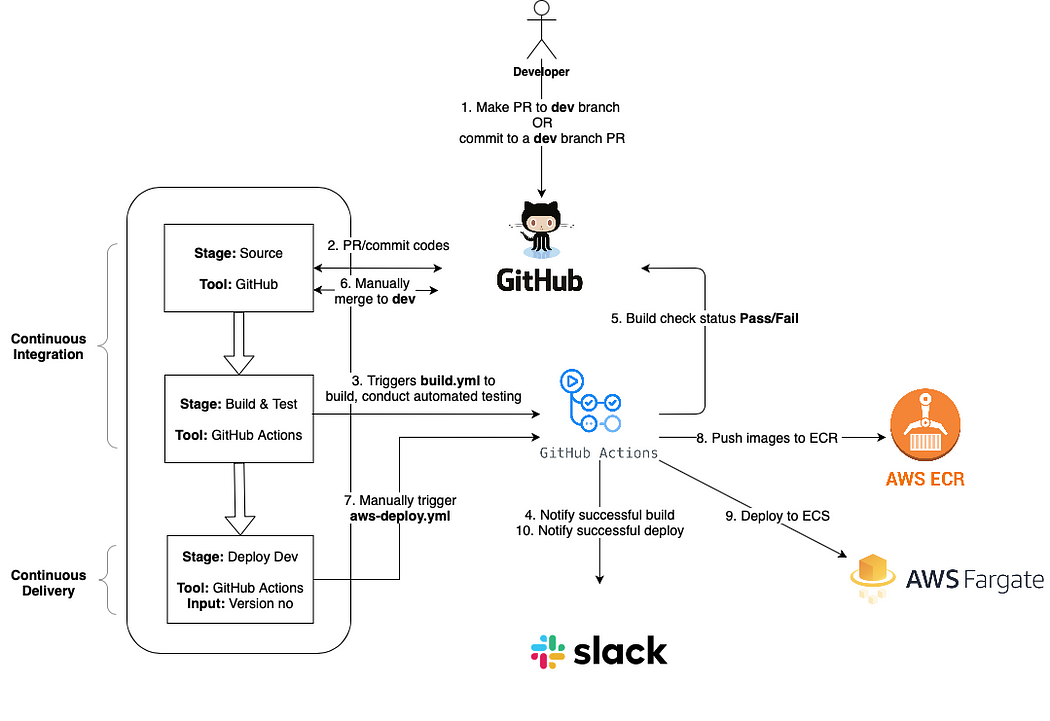
You can track your deployment activity on GitHub as well where any failed or successful deployment will be reflected here.



Screenshot of GitHub Deployment Activity Log

## Workflow

In addition to what was mentioned in Part 1, this would be the overall workflow of our CI/CD pipeline that we will be achieving! It looks like a challenge but I hope it’s something that would get you excited to work on it.



CI/CD Workflow Diagram

One important point that is worth noting is people often get confused between continuous delivery and continuous deployment. This article [here](https://semaphoreci.com/blog/2017/07/27/what-is-the-difference-between-continuous-integration-continuous-deployment-and-continuous-delivery.html) provides a very good summary of their differences which I encourage you to check out if you would like to gain a better understanding of CI/CD.

# Resources

Here is a summary of the files and environment variables that we will be using for this set-up.

## Files used

* aws-deploy.yml — contains the workflow that is triggered manually to deploy to AWS dev environment
* deploy.json — contains the list of container names that should be deployed
* deploy.sh — deployment script that contains functions that will be used to push images to AWS ECR and deploying to AWS ECS

## GitHub Secrets used

* AWS\_ACCESS\_KEY\_ID
* AWS\_SECRET\_ACCESS\_KEY
* AWS\_ACCOUNT\_ID
* SLACK\_WEBHOOK\_URL

# Version Management

Healthcheck is an important tool that helps ensure that our application is up and running. To extend its purpose, it can also be a way to track the version of the application by including the version number as the response of your healthcheck API output. The API\_VERSION\_NO can be obtained from the version that you have specified when triggering the workflow. In the next section, I will explain how you can actualise it.

healthCheck(req, res) {  
 return res.status(200).send({  
 versionNo: process.env.API\_VERSION\_NO  
 })  
}

# Setting Up GitHub Workflow Dispatch

Before we create our deployment script, we will first create a workflow dispatch that allows us to trigger the deployment workflow manually. In your .github/workflows folder, create a file named aws-deploy.yml .

## 1. Deployment Inputs

This is where you get to set up inputs when triggering the deployment workflow. In our case here, we would like to include the version number.

on:  
 workflow\_dispatch:  
 inputs:  
 VERSION:  
 description: 'Version number'  
 default: '0.154'  
 required: true  
name: Deploy to Amazon ECS Dev

## 2. Environment Variables

Next include all relevant environment variables such as inputs, GitHub secrets and GitHub environment variables e.g. BUILD\_ID to keep track of the deployment ID.

jobs:  
 deploy:  
 name: Deploy  
 runs-on: ubuntu-latest  
 container:  
 image: lucasalt/act\_base:latest  
 environment: development  
 env:   
 AWS\_REGION: ap-southeast-1  
 APP\_NAME: "app"  
 AWS\_ECR\_ADDRESS: ${{ secrets.AWS\_ECR\_ADDRESS }}  
 AWS\_ACCOUNT\_ID: ${{ secrets.AWS\_ACCOUNT\_ID }}  
 VERSION: ${{ github.event.inputs.VERSION }}  
 BUILD\_ID: ${{ github.run\_number }}

## 3. Installing dependencies

As we will be running awscli and jq commands for our deployment later on, we would need to install these dependencies in this step. Note that we will be using awscliv2 for this.

steps:  
 - name: Install sudo package  
 run: apt update && apt install sudo - name: Installing packages  
 run: |  
 sudo apt-get update -y  
 sudo apt-get install jq curl -y  
 curl "<https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip>" -o "awscliv2.zip"  
 unzip awscliv2.zip  
 sudo ./aws/install

## 4. Configure AWS Credentials

Authentication is our next important step as we need to be authenticated with AWS before we can run our deployment script. Using awscliv2 that we have installed in our previous step, we will [configure](https://docs.aws.amazon.com/cli/latest/reference/configure/set.html) our credentials with AWS\_ACCESS\_KEY\_ID and AWS\_SECRET\_ACCESS\_KEY obtained from GitHub Secrets.

- name: Configure AWS credentials  
 run: |  
 aws configure set aws\_access\_key\_id "${{ secrets.AWS\_ACCESS\_KEY\_ID }}"  
 aws configure set aws\_secret\_access\_key "${{ secrets.AWS\_SECRET\_ACCESS\_KEY}}  
 aws configure set region "$AWS\_REGION"  
 aws configure set output json

## 5. Deployment

Lastly, there are 2 steps to this deployment process:

* Building and pushing images to ECR
* Deploying images from ECR to AWS ECS

We would need to pass along ECR and version information into our script for authentication and version tracking.

- name: Push to ECR  
 run: |  
 cd app  
 sh deploy.sh build-to-ecr $AWS\_REGION $APP\_NAME $AWS\_ECR\_ADDRESS $AWS\_ACCOUNT\_ID $VERSION $ACCOUNT\_TAG $BUILD\_ID- name: Deploy to Fargate  
 run: |  
 cd app  
 sh deploy.sh deploy

# Setting Up Deployment Script

We’re done with getting our workflow set up! 🎉 . Next, we would need to create a deployment script called deploy.sh in your root folder to create functions that can aid our deployment workflow. This deployment script contains functions that will be called by our aws-deploy.yml workflow.

## 1. Authenticating ECR

Firstly, we would need to authenticate with AWS ECR using our AWS credentials which was passed from our aws-deploy.yml workflow.

# ENV variables from Github Workflow  
AWS\_REGION=${AWS\_REGION}   
AWS\_ECR\_ADDRESS=${AWS\_ECR\_ADDRESS}  
AWS\_ACCOUNT\_ID=${AWS\_ACCOUNT\_ID}  
APP\_NAME=${APP\_NAME}  
VERSION\_NO=${VERSION\_NO}  
ECR\_SERVICE\_URL="${AWS\_ACCOUNT\_ID}.${AWS\_ECR\_ADDRESS}"  
VERSION\_TAG="latest"  
BUILD\_ID=${BUILD\_ID}set -e # exit early if issues ariseaws ecr get-login-password --region ap-southeast-1 | docker login --username AWS --password-stdin ${ECR\_SERVICE\_URL}

## 2. Building Images and pushing to ECR

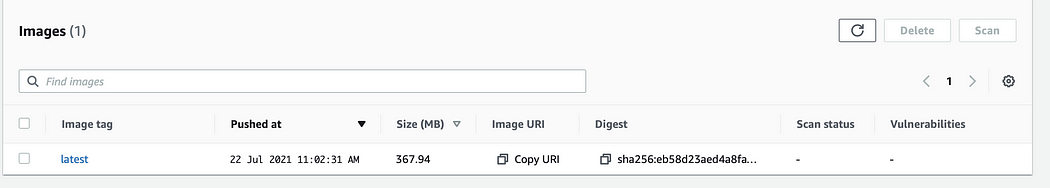
As we will be deploying multiple containers, jq makes our life easier by allowing us to loop through the list of containers in deploy.json that we would need to build in our script.

These are the key steps that this function will make:

* Build our images and set api\_version\_no as the build argument. By doing so, you are able to set the version number as an environment variable to be used for the healthcheck API.
* We will tag the image version as latest as we would want our ECS to deploy the latest image uploaded.
* Push the images to the respective repository. Depending on the task definition that you have set in ECS, your ECR\_REPOSITORY\_NAME may differ from this. **Do modify them accordingly to ensure that your ECS is able to obtain the correct image to deploy**.

# build images and push to ECR  
if [ “$1” = “build-to-ecr” ];then  
 echo "AWS Account: "${ACCOUNT\_TAG}  
 echo "building the docker images..." for row in $(jq -c '.[]' deploy.json); do  
 \_jq() {  
 echo ${row} | jq -r ${1}  
 }  
   
 CONTAINER\_NAME=$(\_jq '.containerName')  
 # create all docker images and push all to ECR  
 ECR\_REPOSITORY\_NAME=${APP\_NAME}\_${CONTAINER\_NAME}  
 ECR\_REPO\_URL="${ECR\_SERVICE\_URL}/${ECR\_REPOSITORY\_NAME}" aws ecr create-repository --repository-name ${ECR\_REPOSITORY\_NAME:?}  
   
 docker build --tag ${CONTAINER\_NAME}:${VERSION\_TAG} \\  
 --build-arg api\_version\_no=${VERSION\_NO}-${BUILD\_ID} \\  
 . docker tag ${CONTAINER\_NAME}:${VERSION\_TAG} ${ECR\_SERVICE\_URL}/${ECR\_REPOSITORY\_NAME}:${VERSION\_TAG}  
   
 # Push the image to ECR  
 docker push ${ECR\_SERVICE\_URL}/${ECR\_REPOSITORY\_NAME}:${VERSION\_TAG} done  
fi

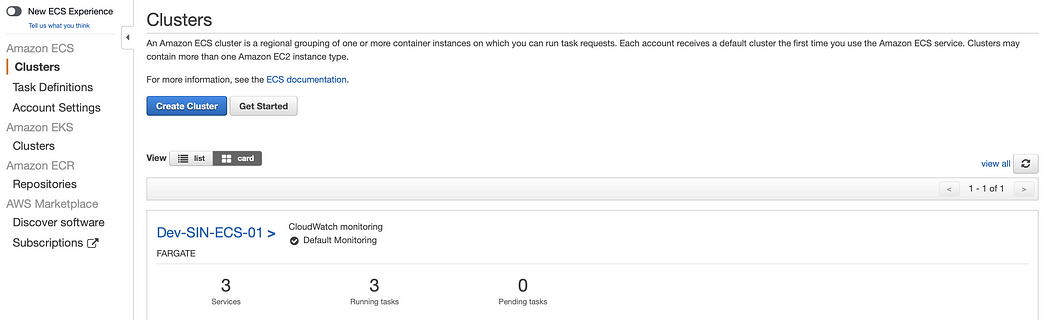
Once the images have been pushed, you should be able to view them in AWS ECR.



Screenshot of AWS ECR

## 3. Deploying to AWS ECS

Finally, take note of the name of your ECS cluster that you have created in AWS, you would need to specify the cluster and service that you would like to update.



Screenshot of AWS ECS Cluster

By running the following function in your deploy.sh script, AWS ECS will deploy the images to Dev-service-01-users service in the Dev-SIN-ECS-01 cluster.

if [ "$1" = "deploy" ];then  
 aws ecs update-service --force-new-deployment --cluster Dev-SIN-ECS-01 --service Dev-service-01-users  
  
fi

# Deployment In Action

Now that we are done with the set up, we can start our deployment!

1. Head over to **GitHub > Actions > Deploy to Amazon ECS Dev (Workflow) > Run workflow**
2. Ensure that you are using workflow from develop branch as that is our main branch which contains the most updated source codes that we would want to deploy
3. Include **version number** in the input
4. Click on **“Run workflow”** and the deployment process should begin.
5. Once the deployment is successful, a slack notification will be sent. It takes time for the ECS tasks to provision and run ~7mins
6. Once the new tasks are up and running, run the healthcheck API on postman (or your browser) to determine if the newest version has been deployed successfully.