



Get Started With CI/CD Using GitHub Actions

Part 1 of “Comparing CI/CD Tools”. In this introduction to GitHub Actions, we’ll explore how easy it is to build and deploy a simple Python Flask application to AWS Elastic Beanstalk.

Welcome to the first part of a series of articles **comparing CI/CD platforms**. To help evaluate, compare and contrast the tools currently dominating the market, the goal will be to automate the deployment of a Flask application onto AWS Elastic Beanstalk. A new deployment will need to occur after every push to main branch, and during the series this same requirement will be implemented across a multitude of CI/CD tools.

First up, part 1 will focus on **GitHub Actions**, where the [support for CI/CD was first announced in August 2019](#). I’ll document the steps taken along the way to achieve the goal with GitHub Actions, and there’s a write up of the positives and negatives in a conclusion at the end of this article.

Our application

The Flask application

Our [Flask application](#) is a simple “Hello World” example, and for the purposes of demonstrating running unit tests within the pipeline, we have included a test case as well.

AWS Elastic Beanstalk

Our Flask application is going to be deployed to AWS Elastic Beanstalk which is a service that automates the deployment and scaling of a web application. It takes in our source code and takes care of all the infrastructure configuration.

There’s a handy CLI for Elastic Beanstalk, and ultimately we’re going to automate the following commands into our pipeline. However, these are just for reference and you do not need to run these now.

```
$ eb init -p python-3.6 hello-world --region us-east-1
```

```
$ eb create hello-world-env
```

Introduction to GitHub Actions

- An **event** will trigger a **workflow**.
- A **workflow** contains one or more **jobs**.
- A **job** is executed on a **runner** and will contain one or more **steps**.
- A **step** will contain a single **action** or a series of inline commands.
- The run can generate **artifacts** and use a **cache**.

Runner

A job *runner* is any machine with the GitHub Actions Runner Application installed — these machines can be GitHub-hosted or you can run your own.

GitHub-hosted runners are virtual machines — Linux (Ubuntu), Windows, and macOS — and are fully managed by GitHub and come with a lot of installed software. For example, the [image for Ubuntu](#) comes armed with multiple libraries, including the AWS CLI

and Google SDK. As part of the free-tier, GitHub provides 2,000 minutes per month of its hosted runners.

The runner is defined in the workflow yaml with the `runs-on` property.

Action

An *action* is the smallest unit of work in the GitHub Actions ecosystem. These are the individual tasks that can be combined as *steps* to create a *job*. You can create your own actions, use community-shared actions from the [Marketplace](#), and customize public actions. Actions can also be packaged up as docker containers that are executed on the *runner* machine — these shared actions in the Marketplace may provide additional software or scripts in a container that encapsulate more complex or repeat scenarios.

Step

A *step* is an individual task that can run a series of commands or a single *action*. By default all steps will run directly on the job runner virtual machine unless the step refers to an action that is configured to run in a container.

Job

A *job* is a group of one or more *steps*, and all steps within the same job will share the filesystem.

Jobs can run at the same time in parallel (default) or run sequentially — to define that run jobs are to sequentially, specify the dependencies in the job specification. For example, a workflow can have two sequential jobs that *build* and *deploy* code, where the deploy job is dependent on the status of the build job. If the build job fails, the deploy job will not run.

Workflow

A *workflow* is the overall automated process that is made up of one or more *jobs*. You define a workflow using YAML format files and these are saved alongside your application code in the `.github/workflows` directory.

Dependency Caching

Caching third party dependencies is a common technique when building out a CI/CD pipeline as it can save minutes from repeatedly downloading the same dependencies. It's useful to save off the contents of `node_modules` for npm; the `.m2/repository` for Apache Maven; and in our case the pip cache from `~/.cache/pip`.

GitHub Actions provides a simple solution that allows us to save off the path and all its contents to the cache at the end of a *successful completion* of the job. The cache will be restored to the path at the start of the next job execution. Pay attention however, because if the job fails the cache will not be saved.

Steps

These are the four steps we will follow to create our CI/CD pipeline.

1. Create an empty repository in GitHub and set up an initial workflow and run our first “build” job. This workflow will run after every `push` event on the `main` branch.
2. Add our Python code for a “Hello World” application, and update the steps in our “build” job definition to test the Python code after every change.
3. Create a “deploy-to-test” job definition which is configured to run after the “build” job has successfully completed. We will install the Elastic Beanstalk CLI, configure our AWS credentials using GitHub Secrets, and create the application and environment.
4. With an existing environment, the “deploy-to-test” job will fail on the second run — so we need to fix that issue so that the workflow will run after every push event. We'll make a change to the application to test its repeatability.

Step 1. Create GitHub repository and initial workflow

Create your GitHub repository and upload your initial application code. My demonstration code is a simple “Hello World” Flask

application, a test case, and a requirements.txt for defining the dependencies.

If you are following along your initial structure should contain three files:

1. [application.py](#)
2. [test_application.py](#)
3. [requirements.txt](#)

With your code now in place, head over to the GitHub console, go into “Actions”, and create a “New Workflow”.

GitHub will detect you have a Python application and will give you some starter pipelines. There’s many to choose from — some from GitHub, some from third parties — for the purpose of this demo, select “set up a workflow yourself” so that we can start with a blank canvas and learn as we go.

Get started with GitHub Actions

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow template to get started.

Skip this and [set up a workflow yourself](#) →

The default name is `main.yaml` — paste the following to get started.

```
name: CI
```

```
on:
```

```
  push:
```

```
    branches: [main]
```

```
jobs:
```

```
  build:
```

```
    runs-on: ubuntu-latest
```

```
  steps:
```

```
# Checks-out your repository under $GITHUB_WORKSPACE
```

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

```
# Set up Python 3.6 environment
```

```
- name: Set up Python 3.6
```

```
  uses: actions/setup-python@v1
```

```
  with:
```

```
    python-version: "3.6"
```

```
# Install dependencies
```

```
- name: Install dependencies
```

```
  run: |
```

```
    python -m pip install --upgrade pip
```

```
    pip install -r requirements.txt
```

```
# Run our unit tests
```

```
- name: Run unit tests
```

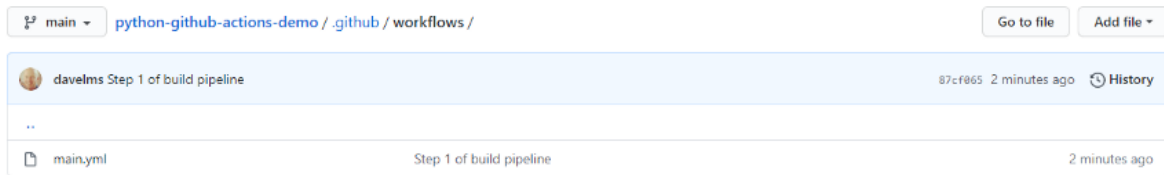
```
  run: |
```

```
    python test_application.py
```

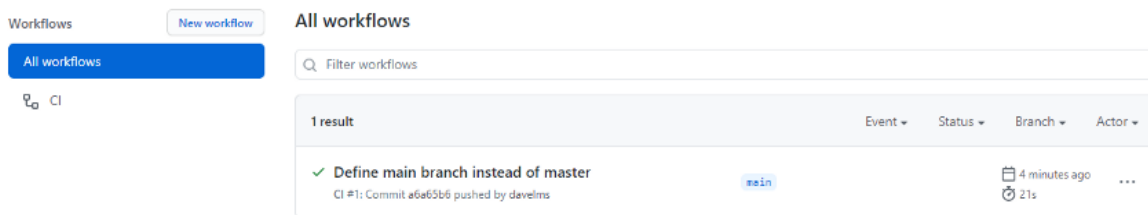
Our initial workflow defines *1 job* (called “build”) consisting of *4 steps*:

1. Checkout the code
2. Set up Python 3.6 environment
3. Install dependencies
4. Run & test the application code

When you save it will create a file at the path `.github/workflows/main.yaml` and from here on you make, add, commit, and push updates just as you would with any other file in a git repository.



Your workflow should run immediately. Check back into “Actions” to see the output. At this stage, you should have run your first GitHub Actions workflow.



Well done!

Step 2. Add a cache to speed up runtime.

To enable a cache for the pip dependencies, we can use the built-in `cache` Action to save off the pip cache located at `~/.cache/pip`. We set the cache `path` based on the location of the pip cache, and we use our `requirements.txt` file as a hash to detect when we need to rebuild the cache.

Insert the following into your `main.yml` immediately after the `setup-python` step. Save and commit.

```
- name: Get pip cache dir
  id: pip-cache
  run: |
    echo "::set-output name=dir::$(pip cache dir)"
- name: Cache pip
  uses: actions/cache@v1
  with:
    path: ${{ steps.pip-cache.outputs.dir }}
    key: ${{ runner.os }}-pip-${{ hashFiles('**/requirements.txt') }}
    restore-keys: |
```

```
${{ runner.os }}-pip-
```

Run it once and you will see that there is no cache hit.

Re-run your workflow and you will see a hit. This will save time for larger applications with lots of dependencies.

```
Cache pip 1s
1 ▶ Run actions/cache@v1
9 Cache Size: ~2 MB (2161709 B)
10 /bin/tar -xz -f /home/runner/work/_temp/7348321b-4e6c-4804-835b-a03ad5004128/cache.tgz -C /home/runner/.cache/pip
11 Cache restored from key: Linux-pip-79cc741f37eb92e3563feb4414429935bda1788c43a7db93d394b5e0744d78c4
```

Step completed. Well done!

Step 3a. Create our initial “deploy-to-test” job

Now that we have a successful “build”, we want to deploy the application for testing purposes. We could continue in a sequential manner within the “build” job, but let’s demonstrate how we would create a multi-job workflow instead.

By default, jobs run in **parallel**. To run jobs **sequentially**, as we require, we have to specify we have a dependency between “deploy-to-test” upon “build”.

To achieve this we state our “deploy-to-test” job `needs` “build” to be successful before the job will run — so if “build” fails, GitHub Actions will not invoke the “deploy-to-test” job.

deploy-to-test:

```
# Only run this job if "build" has ended successfully
```

```
needs:
```

```
- build
```

We’ll run on Ubuntu as we did before, and we’ll need to ensure that we checkout the git repository and set up Python 3.6 again — it’s not shared.

Although, AWS CLI is installed we default, the Elastic Beanstalk CLI is not so we need to install it using the `pip install awsebcli` command.

```
# Elastic Beanstalk CLI version
```

```
- name: Get EB CLI version
```

```
run: |
```



```
python -m pip install --upgrade pip setuptools wheel
```

```
pip install awsebcli --upgrade
```

```
eb --version
```

Here's the full definition to append to your `main.yaml` file.

deploy-to-test:

```
# Only run this job if "build" has ended successfully
```

```
needs:
```

```
- build
```

```
runs-on: ubuntu-latest
```

```
steps:
```

```
# Checks-out your repository under $GITHUB_WORKSPACE
```

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

```
# Set up Python 3.6 environment
```

```
- name: Set up Python 3.6
```

```
  uses: actions/setup-python@v1
```

```
  with:
```

```
    python-version: "3.6"
```

```
# Set up cache for pip
```

```
- name: Get pip cache dir
```

```
  id: pip-cache
```

```
  run: |
```

```
    echo "::set-output name=dir::$(pip cache dir)"
```

```
- name: Cache pip
```

```
  uses: actions/cache@v1
```

```
  with:
```

```
    path: ${ steps.pip-cache.outputs.dir }
```

```
    key: ${ runner.os }-pip-${ hashFiles('**/requirements.txt') }
```

```
restore-keys: |
  ${{ runner.os }}-pip-
```

```
# Elastic Beanstalk CLI version
```

```
- name: Get EB CLI version
```

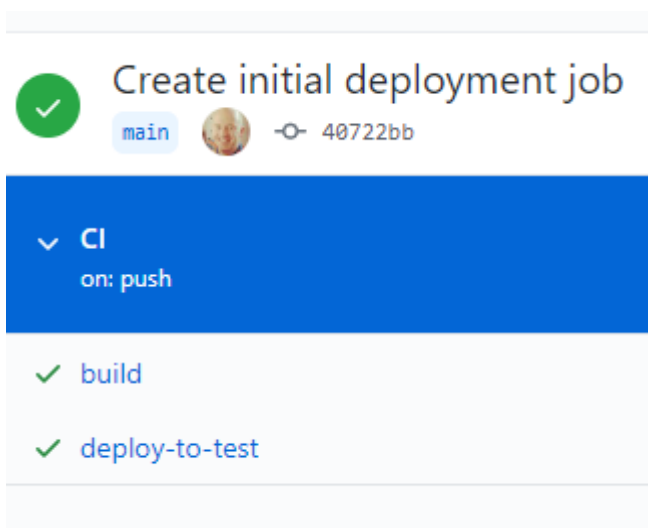
```
run: |
```

```
python -m pip install --upgrade pip
```

```
pip install awsebcli --upgrade
```

```
eb --version
```

At this stage, you should now have the Elastic Beanstalk CLI deployed and available to your “deploy-to-test” job. In the UI you will see the additional job listed in the left hand column.



Check the output to confirm the Elastic Beanstalk CLI version was displayed.

```
122 EB CLI 3.19.1 (Python 3.6.1)
```

Well done, another milestone achieved.

Step 3b. Configure our AWS credentials using GitHub Secrets

Configure two secrets in GitHub to contain your User access key and client secret. Secrets are made available to your job as `${{ secrets.XXXX }}`.

- **AWS_ACCESS_KEY_ID**

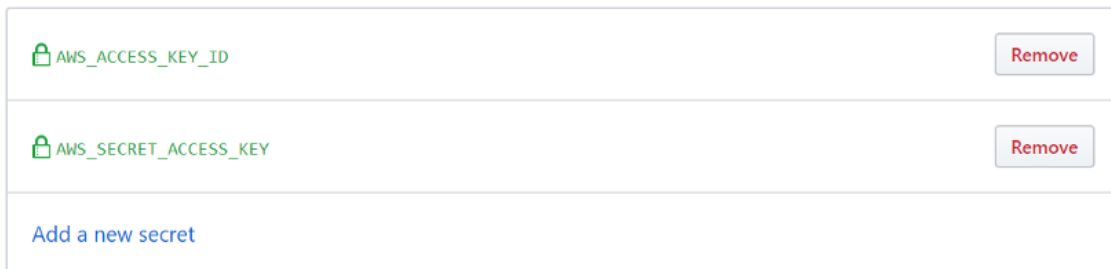
- **AWS_SECRET_ACCESS_KEY**

To set these up, head into the GitHub console and *Secrets* is under *Settings*.

Secrets

Secrets are environment variables that are **encrypted** and only exposed to selected actions. Anyone with **collaborator** access to this repository can use these secrets in a workflow.

Secrets are not passed to workflows that are triggered by a pull request from a fork. [Learn more](#).



(remember, AWS security best practice would be to create a dedicate User with programmatic access and having permissions just to deploy to AWS Elastic Beanstalk).

In a new step, we can add the Action `configure-aws-credentials` which is a docker container provided via the AWS-team managed project `aws-actions`.

- name: Configure AWS Credentials

uses: aws-actions/configure-aws-credentials@v1

with:

aws-access-key-id: \${{ secrets.AWS_ACCESS_KEY_ID }}

aws-secret-access-key: \${{ secrets.AWS_SECRET_ACCESS_KEY }}

aws-region: us-east-1

There's no visual clue that this worked, so commit and jump on to Step 3c.

Step 3c. Deploy Application to Elastic Beanstalk

Elastic Beanstalk is so easy to use from the command prompt using the CLI. Remember those two CLI commands I shared at the very start and said not to run? We are going to include these commands

in our pipeline and build our first Elastic Beanstalk application and environment.

```
# Create the Elastic Beanstalk application
```

```
- name: Create EBS application
```

```
run: |
```

```
  eb init -p python-3.6 hello-world --region us-east-1
```

```
# Create the Elastic Beanstalk environment
```

```
- name: Create test environment
```

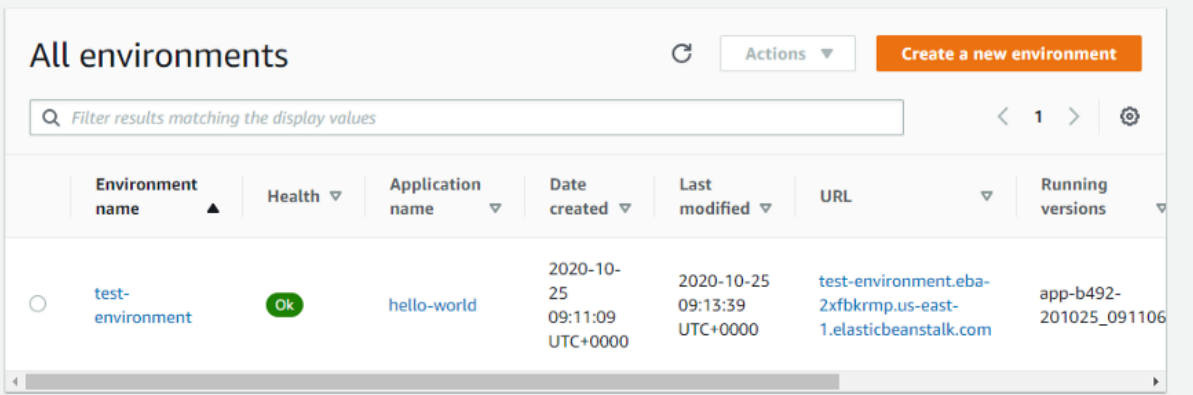
```
run: |
```

```
  eb create test-environment
```

Commit your changes, and you should see your application successfully running on Elastic Beanstalk. Check the command line for confirmation of the endpoint:

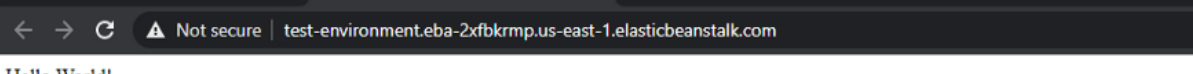
```
35 2020-10-25 09:13:38 INFO Application available at test-environment.eba-2xfbkrmp.us-east-1.elasticbeanstalk.com.
```

And verify the same in the AWS Console.



Environment name	Health	Application name	Date created	Last modified	URL	Running versions
test-environment	Ok	hello-world	2020-10-25 09:11:09 UTC+0000	2020-10-25 09:13:39 UTC+0000	test-environment.eba-2xfbkrmp.us-east-1.elasticbeanstalk.com	app-b492-201025_091106

And here's the application successfully running on AWS...



Not secure | test-environment.eba-2xfbkrmp.us-east-1.elasticbeanstalk.com

Hello World!

Well done, you've successfully run a build, test, deploy to Elastic Beanstalk.

Step 4. Avoiding failure on subsequent job executions

Our workflow is working and has successfully created the initial Elastic Beanstalk application and environment. The **first version** has been installed.

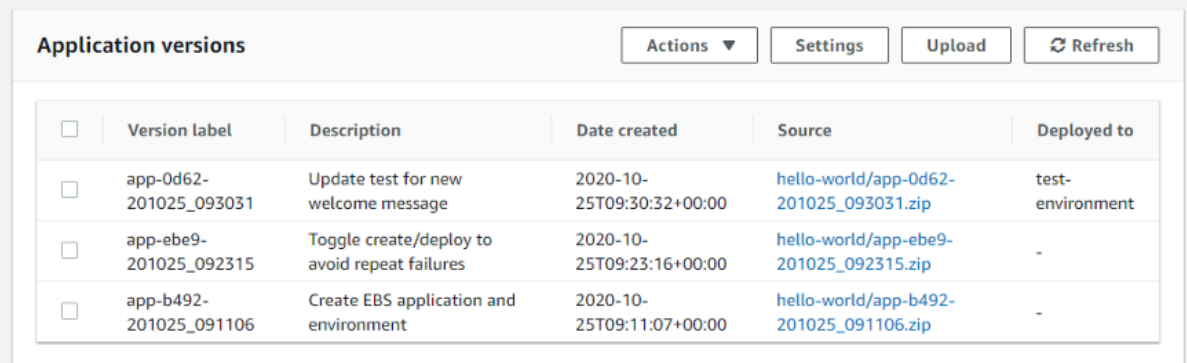
But what happens when we make a change and push that to master branch. Unfortunately, the `eb create test-environment` will **fail** when it is run for a second time — we need to detect that scenario, and run a `eb deploy` instead.

We can solve that problem with a bit of logic syntax and the `eb status test-environment` command. If our “test-environment” already exists, the command passes and we can deploy to it with `eb deploy`. Otherwise, if it fails, then we create “test-environment”.

Our updated command now looks like this.

```
(eb use test-environment && eb status test-environment && eb deploy) || eb create test-environment
```

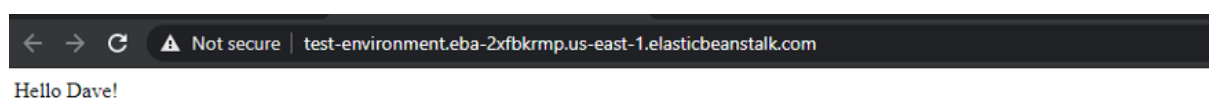
If we make a change (I edited “*Hello World!*” to “*Hello Dave!*”), we can see from the screenshot below that the GitHub Actions has deployed the new Application version to the “test-environment”.



The screenshot shows the 'Application versions' section of the Elastic Beanstalk console. It includes a table with columns for Version label, Description, Date created, Source, and Deployed to. The table lists three versions: 'app-0d62-201025_093031' (Update test for new welcome message), 'app-ebe9-201025_092315' (Toggle create/deploy to avoid repeat failures), and 'app-b492-201025_091106' (Create EBS application and environment). The first version is deployed to the 'test-environment'.

<input type="checkbox"/>	Version label	Description	Date created	Source	Deployed to
<input type="checkbox"/>	app-0d62-201025_093031	Update test for new welcome message	2020-10-25T09:30:32+00:00	hello-world/app-0d62-201025_093031.zip	test-environment
<input type="checkbox"/>	app-ebe9-201025_092315	Toggle create/deploy to avoid repeat failures	2020-10-25T09:23:16+00:00	hello-world/app-ebe9-201025_092315.zip	-
<input type="checkbox"/>	app-b492-201025_091106	Create EBS application and environment	2020-10-25T09:11:07+00:00	hello-world/app-b492-201025_091106.zip	-

Hitting the endpoint in the browser confirms the deployment was successful.



Well done, that’s the end of the demonstration. You have successfully built and deployed your application to Elastic Beanstalk and been through the cycle of a code change and seen that change

reflected in AWS almost immediately (my testing showed it took around 1 minute 30 seconds from commit to seeing the updated code running on AWS).

You can download the final [main.yml](#) from GitHub.

Conclusion

Article first written May 2020 and things change over time. At the time of writing, here were my findings...

+ Positives

- **Easy to get started** with a generous **2,000 minutes free tier**.
- The fact we're building upon GitHub where we are likely to already have our **git hosting, project pages, issues, team management**, etc. is an obvious bonus straight off the bat.
- GitHub Actions provides multiple **out-of-the-box workflow examples** for a wide range of coding languages, and these are augmented with ready-made third party Actions available via the **Marketplace**.
- **Extensible framework** — you can create your own Actions inline, or publish an Action and reference in your projects.
- **Jobs are started up quickly** and there is no observable lag.
- GitHub-hosted Runners come **pre-built with a lot of software**, including the CLI for AWS, Google Cloud Platform, and Azure cloud platforms.
- Fully feature **GitHub Actions API**, and support for **Web Hooks**.
- The **extensive documentation** is developer focussed and well presented.

- Negatives

- At the time of writing there isn't a built-in ability to chain workflows to create more complex pipelines, or to embed/re-use whole workflows.
- No support for manual triggers. While a user can re-run a failed job, that's the limit to the user-initiated actions. For example, a feature integrated directly into most other CI/CD platforms is the ability to implement a manual push-button stage for deployments.
- To address both of the above, we can use the **GitHub Actions API** and **Web Hooks**, triggering events from another tool or system (e.g, Slack, postman) or use custom branches and use the merge from one to another as a means to trigger the "next stage".

A note from the author

Thank you for reading this article — I hope you found it useful and please read the remainder of the series as I compare other CI/CD tools.