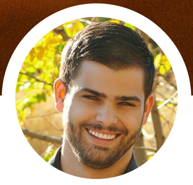


ZACHARY FLOWER
FREELANCE WEB DEVELOPER AND WRITER

Deploying Docker Apps to AWS

Share this Codeship Guide

About the Author.

Zachary Flower is a freelance web developer, writer, and polymath. He has an eye for simplicity and usability, and strives to build products with both the end user and business goals in mind.

Deploying Docker Apps to AWS

Amazon Web Services (AWS) is one of the most ubiquitous cloud computing providers available today, delivering a seemingly endless suite of tools for cloud computing and development.

From servers to storage and databases to deployment, if you need it, AWS probably has it. While it is used by companies like Netflix, Slack, and Pinterest to run their services, store data, and develop and deploy new site features, the full scope of AWS's entire product set can be overwhelming at best.

Despite AWS's seemingly endless supply of products, there are really only a handful that are dedicated to delivering and serving application code. These are the Amazon EC2 Container Service, AWS Elastic Beanstalk, and AWS CodeDeploy. Each of these services can be deployed using Codeship.

ABOUT THIS EBOOK

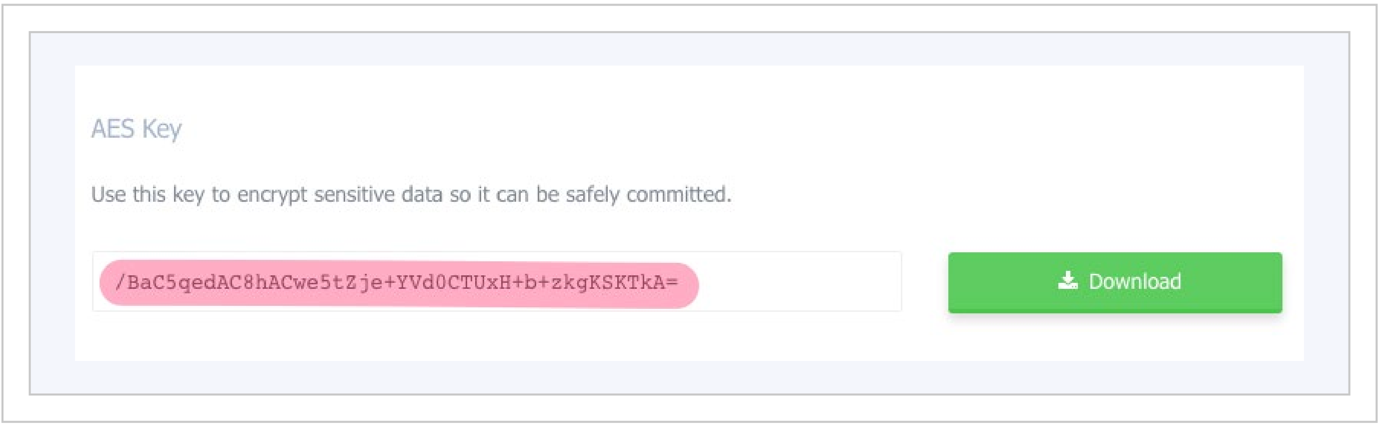
In this book you will learn how to set up Continuous Deployment to Amazon Web Services (AWS) for your Docker apps using Codeship's CI and CD service.

Setting up AWS Authentication

Before you can get started with AWS deployments, you must first add your AWS access keys to your project repository in order for Codeship to be able to access your AWS account. Because these access keys are secret (and therefore a horrible idea to commit to version control in plain text), Codeship's CLI tool called "Jet" offers some awesome functionality for encrypting them. If you want to learn more about how to work with the Codeship Jet CLI you can watch a re-run of Brendan Fosberry's webinar here. Codeship's existing tutorial is far more thorough than this article warrants, but if you want to skip the link, here are the abridged steps:

Step 1 – Find your AES key

Sign into your codeship.com account, go to the General page of your Codeship project's settings, scroll down to the AES Key section, and copy the key:


AES Key

Use this key to encrypt sensitive data so it can be safely committed.

```
/BaC5qedAC8hACwe5tZje+YVd0CTUxH+b+zkgKSKTkA=
```

Step 2 – Save the key

Paste the key into a file called `codeship.aes` in your project's root folder, and add the filename to your `.gitignore` file.

```
echo "KEY_COPIED_FROM_CODESHIP.COM" > codeship.aes
echo "codeship.aes" >> .gitignore
```

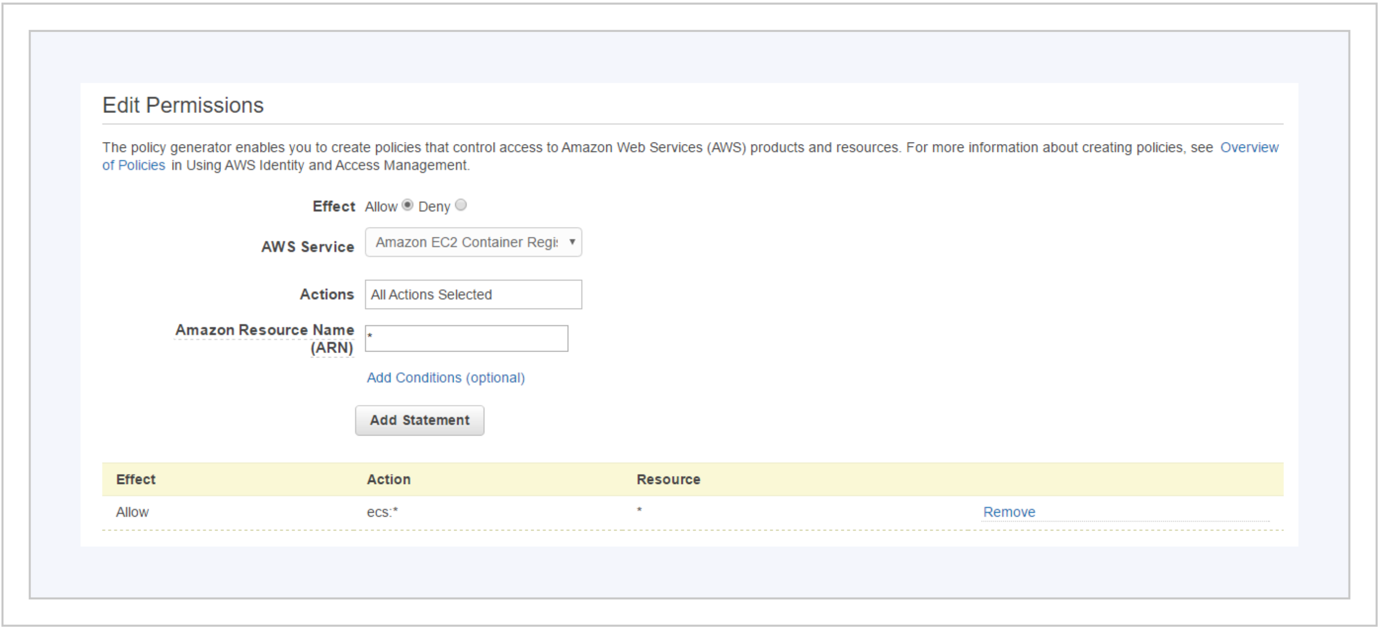
Step 3 – Encrypt AWS credentials

Add your AWS credentials to a file called `aws.env`, encrypt said file, and then add the unencrypted credentials to your `.gitignore` file.

```
# AWS
AWS_ACCESS_KEY_ID=XXXXXXXXXXX
AWS_SECRET_ACCESS_KEY=XXXXXXXXXX
AWS_DEFAULT_REGION=us-east-1
```

```
# jet encrypt [--key-path=codeship.aes] plain_file encrypted_file
jet encrypt aws.env aws.env.encrypted
# also add the plain text version to .gitignore
echo "aws.env" >> .gitignore
```

If you're not sure where to find your AWS credentials, you can generate a new set of "Access Keys" in the IAM Management Console. It's important to note that you can only have two active pairs of root keys at any given time, and once you generate them, you will not be able to retrieve the secret key again, so it is recommended to take advantage of multiple users and managed policies. While the permissions you enable for an IAM policy will vary depending on your use case, the Policy Generator is an excellent tool for clearly defining which services and actions to allow or deny within a policy.


Edit Permissions

The policy generator enables you to create policies that control access to Amazon Web Services (AWS) products and resources. For more information about creating policies, see Overview of Policies in Using AWS Identity and Access Management.

Effect	[✓ Allow] Deny
AWS Service	Amazon EC2 Container Regi:
Actions	All Actions Selected
Amazon Resource Name (ARN)	*

Effect	Action	Resource	
Allow	ecs:*	*	Remove

Now, all that is necessary to enable authenticated AWS commands is to add the `encrypted_env_file` directive to any Codeship service that needs it:

CODE

```
aws_dockercfg:
  image: codeship/aws-ecr-dockercfg-generator
  add_docker: true
  encrypted_env_file: aws.env.encrypted
```

This line essentially tells Codeship to decrypt the specified file using your account's AES key and add the contents to the Docker container's environment variables. As explained by Amazon's documentation, the AWS CLI looks for credentials directly in the environment variables, which means that all you need to do is install the AWS CLI tools to a configured Docker container, and you are good to go (while we will be using a pre-built Docker container provided by Codeship to accomplish this, you can find the steps to accomplish this manually here).

Amazon EC2 Container Service

Over the past few years, the use of container technology like Docker has grown dramatically within organizations of all sizes, which has led popular cloud providers like Google and Amazon to build in tools to better support this. Like I said, if there is a feature you need, AWS probably has it, and Docker support is no exception.

But just because AWS supports it doesn't mean getting it all up and running is a simple and intuitive process. Having a large number of products means that important information inevitably gets lost in the noise. For Docker support, the three most important products to be aware of are the Amazon Elastic Compute Cloud (EC2), the Amazon EC2 Container Service (ECS), and the Amazon EC2 Container Registry (ECR).

EC2 is one of the most popular product offerings on AWS, providing affordable and scalable virtual servers in the cloud. Because of EC2's simple web interface, developers can easily increase or decrease computing resources in minutes while only paying for the resources they use.

In the same vein as Kubernetes and Docker Swarm, ECS is Amazon's solution for running and managing Docker containers across a cluster of Amazon EC2 instances. Should you be interested in learning more about Kubernetes and Docker Swarm I would recommend checking out this and this eBook. ECS allows you to easily manage and scale your application without any additional complexity in how you design, build, and run your product. The greatest thing about Amazon ECS (in my opinion) is that there is no additional charge to use it, which means that you only pay for EC2 usage and nothing else.

Similar to Docker Hub and the Google Cloud Container Registry, Amazon ECR is a Docker image registry that makes it "easy for developers to store, manage, and deploy Docker container images." Generally, the best Docker

registry to use will be the one that integrates best with the rest of your infrastructure, so when paired with Amazon ECS, ECR simplifies the Docker container deployment workflow by providing highly available hosting of Docker images from directly within the AWS ecosystem. If you want to learn more about Docker hosting you should check out this eBook: „The Shortlist of Docker Hosting“.

While the details of actually getting started with Amazon's EC2 Container technology and Codeship are both out of the scope of this article (you can find a great ECS tutorial here and Codeship tutorial here), deploying new Docker images to it with Codeship is a straightforward process.

Configuration

The most complicated piece of the Codeship-ECR integration puzzle is authentication. While we could see above that the AWS access keys can be encrypted and added to the repository for interaction with AWS services, the Docker registry authorization is only temporarily valid. Because of this, we need to define a custom service in the `codeship-services.yml` file to generate one-off `dockercfg` files for each deploy.

CODE

```
aws_dockercfg:
  image: codeship/aws-ecr-dockercfg-generator
  add_docker: true
  encrypted_env_file: aws.env.encrypted
```

This section sets up a Docker container based on Codeship's `aws-ecr-dockercfg-generator` Docker image (you can check out a great guide to using this image from their blog here) using your AWS credentials. This allows you to run any necessary AWS commands in an authenticated and pre-configured environment.

Deployment

Now that we have an authenticated AWS environment with auto-generated `dockercfg` files, we need to set up our deployment steps to use it in the `codeship-steps.yml` file.

Let's say, for example, that we have a Docker image (defined as app in the `codeship-services.yml` file) that we want to push to ECR. To accomplish this, add the following to your `codeship-steps.yml` file:

CODE

```
- service: app
  type: push
  tag: master
  image_name: 874084658176.dkr.ecr.us-east-1.amazonaws.com/myorg/myapp
  registry: https://874084658176.dkr.ecr.us-east-1.amazonaws.com
  dockercfg_service: aws_dockercfg
```

This step tells Codeship to push the `app` service image up to our Docker registry using the defined `aws_dockercfg` service. Behind the scenes, this Codeship image is doing a few things to facilitate the push to ECR.

The first thing that happens in the `aws_dockercfg` service is that it is generating a login command that can be used to authenticate the Docker command line tool with AWS:

```
$ aws ecr get-login
docker login -u AWS -p password -e none https://aws_account_id.dkr.ecr.us-east-1.amazonaws.com
```

Once the returned docker login command is run, the `aws_dockercfg` service has 12 hours to use it before it expires. Next, the image is pushed up to ECR using the standard docker push command (which is what the push step does regardless of the registry):

```
$ docker push aws_account_id.dkr.ecr.region.amazonaws.com/my-web-app
The push refers to a repository [aws_account_id.dkr.ecr.region.amazonaws.com/my-web-app] (len: 1)
e9ae3c220b23: Pushed
a6785352b25c: Pushed
0998bf8fb9e9: Pushed
0a85502c06c9: Pushed
latest: digest: sha256:01f58d96d1fa90e3eb0dd0ac3d893bcaf00d736f2bc82539d3531170e707648c size: 6778
```

It is important to note that, at this point, these configuration changes only push up the new Docker image to the registry. They do not automatically update any currently running services that are utilizing previous versions of the Docker image. While I won't get into the specifics of that in this article (there are a multitude of ways ECR could be set up to facilitate it), the AWS CLI tool does provide functionality for programmatic management of ECS services and tasks.

AWS CodeDeploy + AWS Elastic Beanstalk

So, we've seen how to deploy Docker images to ECR, but what about Amazon's other non-Docker code delivery methods? Elastic Beanstalk and CodeDeploy are both very similar deployment products with one major difference: **resource management.**

AWS CodeDeploy is a service that automates code deployments to currently running EC2 instances. This is a similar solution to many atomic deployment services, as it does not provision resources or provide application-level monitoring. Elastic Beanstalk, on the other hand, is a web application deployment service that can launch additional AWS resources (like load balancers and EC2 instances) and deploy code changes.

Thanks to some hard work already done by Codeship, getting set up with both Elastic Beanstalk and CodeDeploy deployments on Codeship is incredibly straightforward. Because we already did the work of getting authentication working above, the first real step of enabling CodeDeploy and Elastic Beanstalk deploys is adding the following service to your `codeship-services.yml` file:

CODE
```yaml
awsdeployment:
  image: codeship/aws-deployment
  encrypted_env_file: aws.env.encrypted
  volumes:
    - ./:/deploy
```

This section defines a new Docker container using Codeship's `aws-deployment` image using our previously defined environment file, and mounts the project directory to a directory called `/deploy` within the container. Mounting the current project directory within the container is important because the deployment service needs to be able to actually access the code it is to deploy.

Once we've defined the deployment service, we can then utilize it to deploy to both of these services with very little configuration. To enable Codeship deployments using CodeDeploy, add the following step to your `codeship-steps.yml` file:

CODE
```yaml
- service: awsdeployment
  command: codeship_aws_codedeploy_deploy /deploy APPLICATION_NAME DEPLOYMENT_
  GROUP_NAME S3_BUCKET
```

Because Codeship already did all the work to actually facilitate CodeDeploy deployments, all it takes to push your code up to AWS is to plug a few parameters into the `codeship_aws_codedeploy_deploy` command: the CodeDeploy application name, the deployment group name, and the S3 bucket to upload the artifact to. On the flip side, enabling Elastic Beanstalk deployments is just as simple. It also utilizes the `awsdeployment` service, but the command that is run in the `codeship-steps.yml` file changes a bit:

CODE

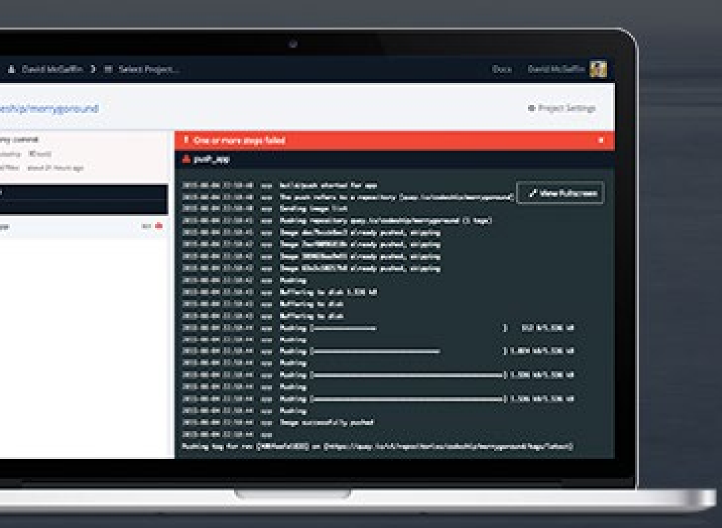
```
- service: awsdeployment
  command: codeship_aws_eb_deploy /deploy APPLICATION_NAME
  ENVIRONMENT_NAME S3_BUCKET
```

Just like the CodeDeploy example, all it takes to deploy to Elastic Beanstalk using Codeship are a few parameters: the Elastic Beanstalk application name, the Elastic Beanstalk environment name, and the S3 bucket to upload the artifact to. While this process is incredibly straightforward, if you'd like to see a practical example of deploying an existing project to both of these services using Codeship, I've mirrored the code shown above onto a fork of the Lumen project. Just take a look at the codeship-services.yml and codeship-steps.yml files and you'll see what I mean.

Conclusion

The above are all relatively basic examples of what can easily become a very complicated process. However, by using Codeship, even the most complex deployment process can be handled with only a little effort.

AWS isn't an easy thing to learn; there are a lot of nuances that can take time to take in and understand. But thanks to tools like Codeship, integrating your CI/CD process with AWS is surprisingly easy.



CODESHIP

Share this Codeship Guide

More Codeship Resources.

CODESHIP

Share this Codeship Guide

About Codeship.

Codeship is a hosted Continuous Integration service that fits all your needs.
Codeship Basic provides pre-installed dependencies and a simple setup UI that let you incorporate CI and CD in only minutes. Codeship Pro has native Docker support and gives you full control of your CI and CD setup while providing the convenience of a hosted solution.

Codeship Basic

A simple out-of-the-box Continuous Integration service that just works.

Starting at $0/month.

- Works out of the box
- Preinstalled CI dependencies
- Optimized hosted infrastructure
- Quick & simple setup

LEARN MORE

Codeship Pro

A fully customizable hosted Continuous Integration service.

Starting at $0/month.

- Customizability & Full Autonomy
- Local CLI tool
- Dedicated single-tenant instances
- Deploy anywhere

LEARN MORE