

**Integrating Docker with Devops Automated Workflows**

How Docker and Dockerized apps can be integrated with existing development and operational, dare I say, DevOps-like processes, like continuous integration and continuous delivery?

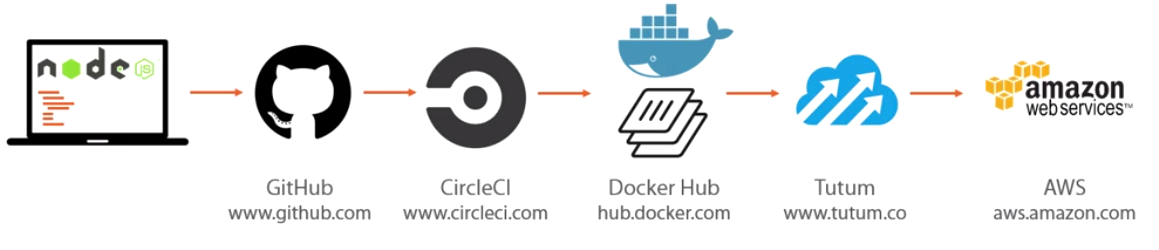
1\_Course Introduction

We've got this super simple Node.js web app, which if you want to follow along on your own laptop or whatever, you can totally fork, clone..

the course is all about taking a Dockerized app and -->running it from source code on our laptop through code management with Git and GitHub,--> automated CI-->triggering image builds on Docker Hub, all the way through to deploying and production.

And it's that workflow that the course is about, and how we integrate Docker into it.

Using: **CircleCI hosted CI platform for our CI portion.**



**Amazon Web Services as our production platform**, and we're going to push to Amazon Web Services using the **Tutum container platform.**

Workflow

Create accounts on: GitHub, CircleCI, Docker Hub, Tutum, and AWS..

All the way from source code through GitHub, CI tests on CircleCI, automated builds on Docker Hub, and pushing to AWS via Tutum, an entirely automated workflow.

we're going to take some source code that we're tracking with Git, push it to a public repo on GitHub, perform a test build, push a successful test build to Docker Hub, and eventually throw our app onto the internet via Amazon Web Services. That's our workflow



Dokerize

It’s just configuring an app to run in a Docker container, or multiple containers.

I don't know, maybe you've got a scalable micro service based app, but the app we're using in this course is super simple, it's going to run in a single Docker container.

Now the way that we actually go about Dockerizing an app, so making it so it can run in an container, is using a Dockerfile.

Dockerfile: simple kind of script or config file that outlines and automates the steps required to install and configure your app in a container, well an image actually, but the end goal is a container.

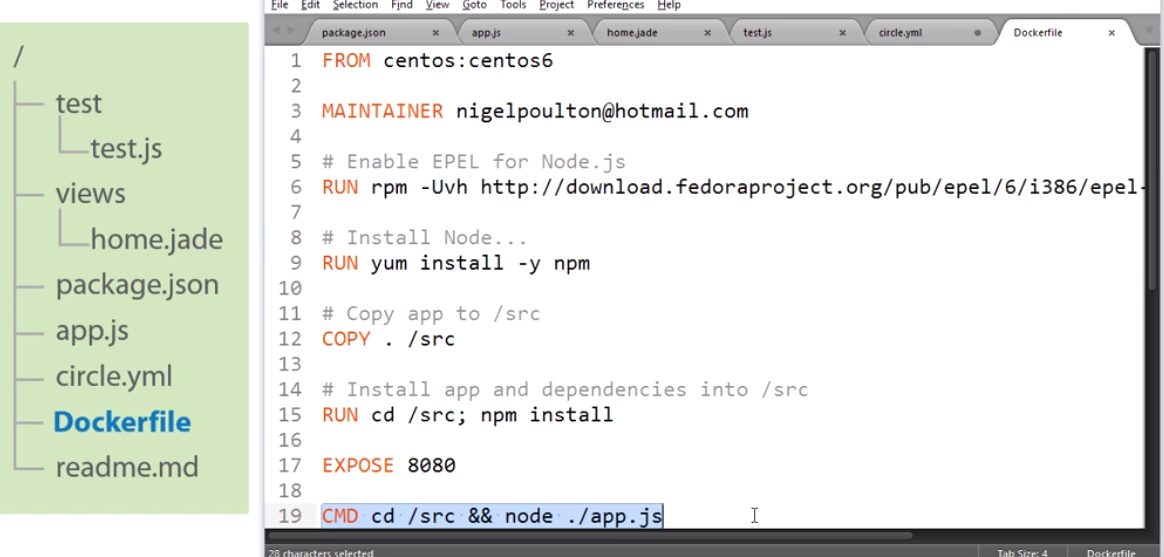
Dockerfile

We're going to build a CentOS Linux-based container, run a couple of commands to install our node components, copy in our source code, install our app dependencies, expose port 8080 in our container, and then the command.

Introducing the App

it's a node app that uses the Express web framework, and this is what it looks like when we point a web browser at it





Dockerfile. almost like script file, its job to tell Docker how to build a Docker image out of our app, to Dockerize it.

we can see we're going to run a CentOS container,

we install a bunch of stuff,

copy in our app,

npm install, but from within our app's directory, so that means it's going to read our package.json, and install our program and its dependencies. we expose port 8080 that our app's listening on, and then

we run our app.

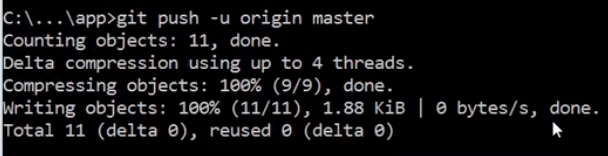
Fork this repo:

2\_Pushing the App to GitHub

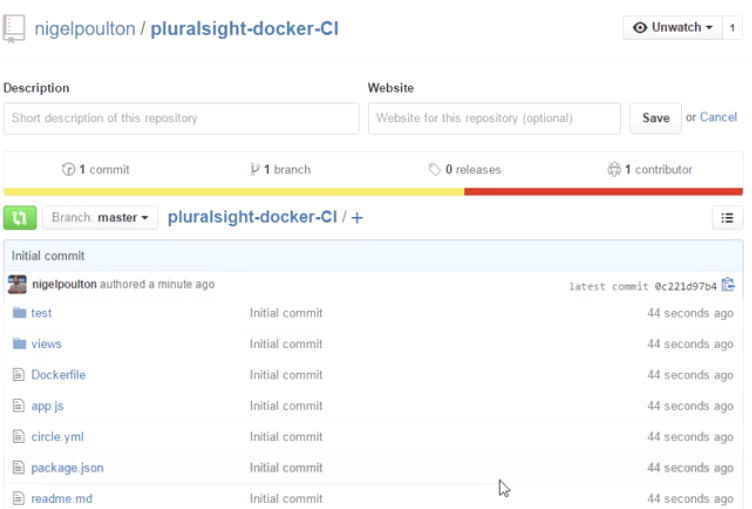
<https://github.com/SteveDuitsman/pluralsight-docker-CI-master>

1. Create repository on github
2. On local windows,



3.)

4.) On github(F5)



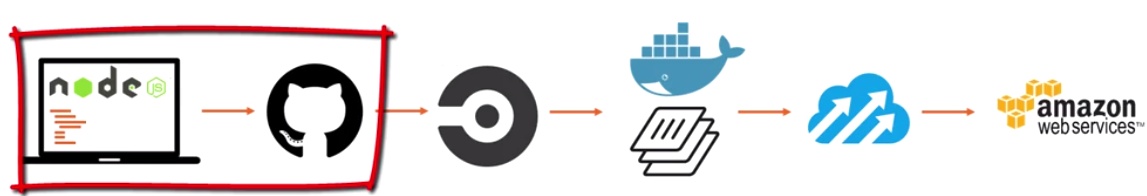
Summary

We’ve got our Node.js web app, built with

Express and Jade and simple HTTP assertion test, which we built with Mocha and SuperTest.



It’s all uploaded to GitHub ready to rock and roll. So that's pretty much this bit of our overall workflow in place…

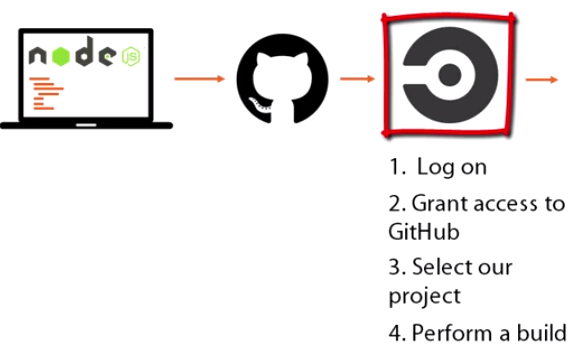


3\_Configuring Test Builds

All right then, we've got our project uploaded to GitHub.

Now, the CI tool using is->CircleCI, a hosted platform for automated testing and continuous integration.

CircleCI, so we're going to log on, give it access to our GitHub repo, select our project, our app, and perform a build, let's go.



CI tools: CircleCI, Shippable, Jenkins, drone.io, Travis CI

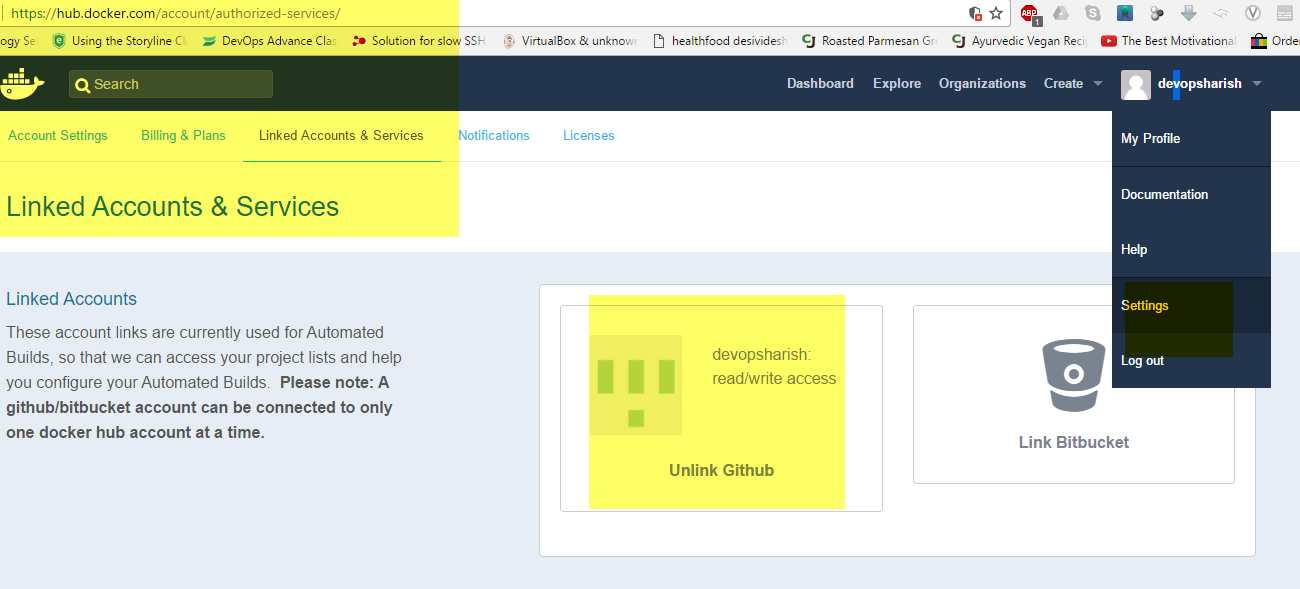
Performing the Test Build

So, here at circleci.com, if you're a first timer you're going to want to go sign up for free, then hit Authorize With GitHub over here, this swings us over to GitHub, we need to hit Authorize application.

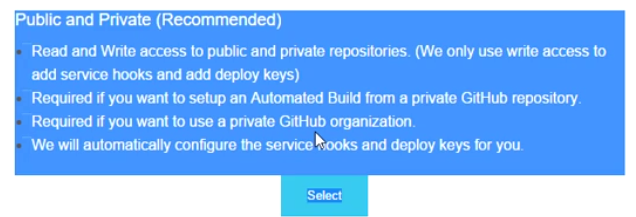
04. Adding Docker Hub Automated Builds to the Mix

Docker Hub, the default public repo for Docker images.

Once you are signed up and you're logged in into dockerhub, you need to link your GitHub account. So that's Settings up here, then Linked Accounts and Services

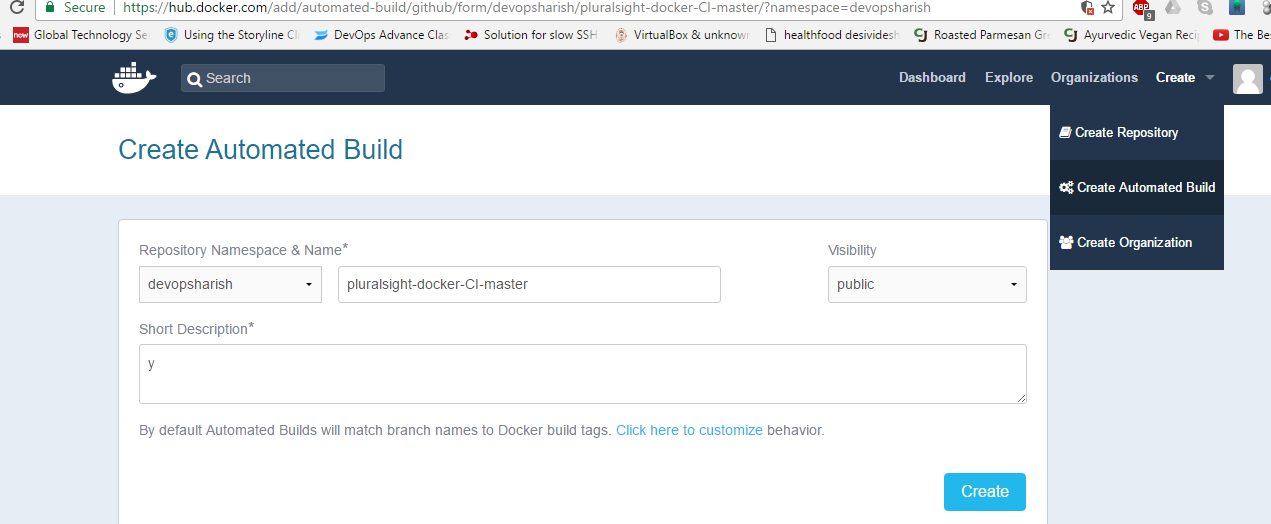


04\_01-Creating a Docker Hub Automated Build Repo



Going to work with private and public repos too..

Create automated build



Create Automated Build is uber important because an automated build is a special kind of Docker Hub repo.

The idea with automated build repos is that we can link them with a GitHub repo, or a Bitbucket if that's your thing, and then :

Anytime we push changes to GitHub it pings Docker Hub via a service hook basically telling Docker Hub about the updates. Docker Hub then goes and automatically builds a new Docker image based on the updated repo.

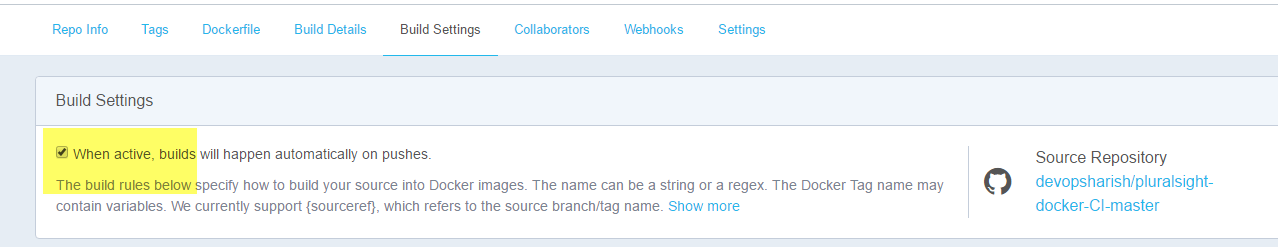
automated build repos trigger a new Docker image build every time we push a code update to GitHub, and that's cool and all, but it's not actually what we want here, is it?

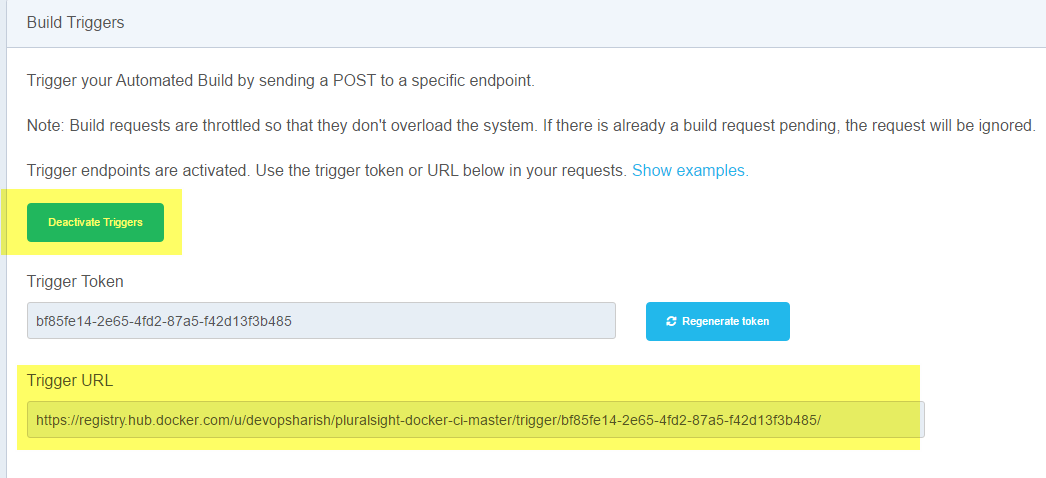
**We want to trigger automated builds programmatically, or only after a green build in CircleCI. So, only after Circle's tested that we haven't broken anything.**

Build Settings --> uncheck that option again

Build Trigger, here, current status set to Inactive---> Activate

Trigger URL here---> give this back to CircleCI so that we can tell Circle to hit this trigger after every successful or green build



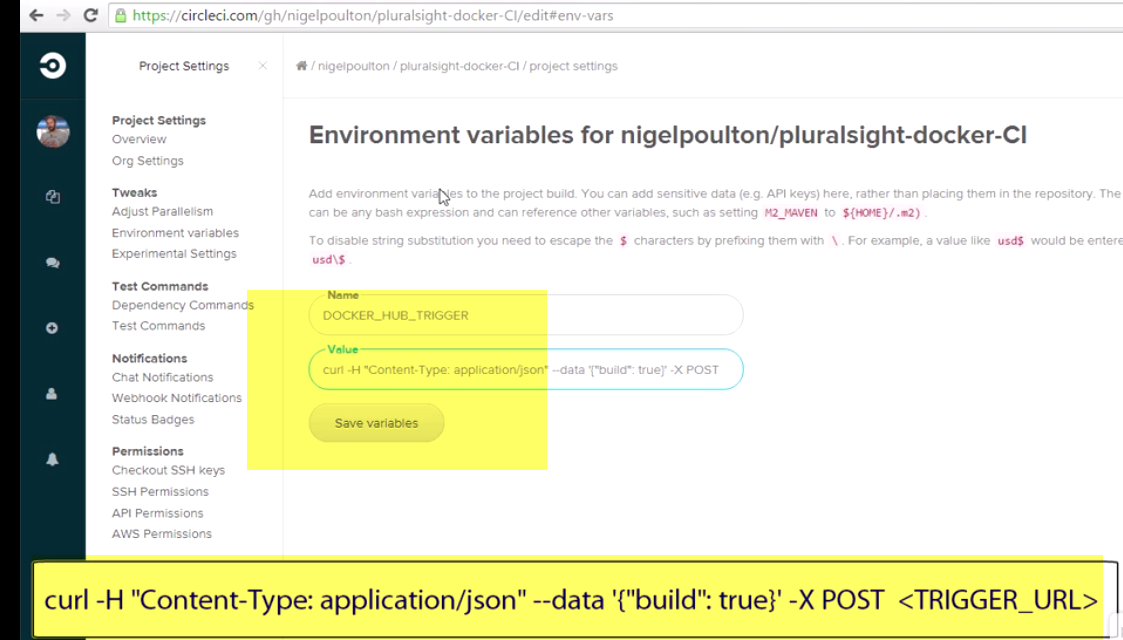


Creating a Successful Build Trigger from CircleCI to Docker Hub

CircleCI-->Project Settings-->create new environment variable.

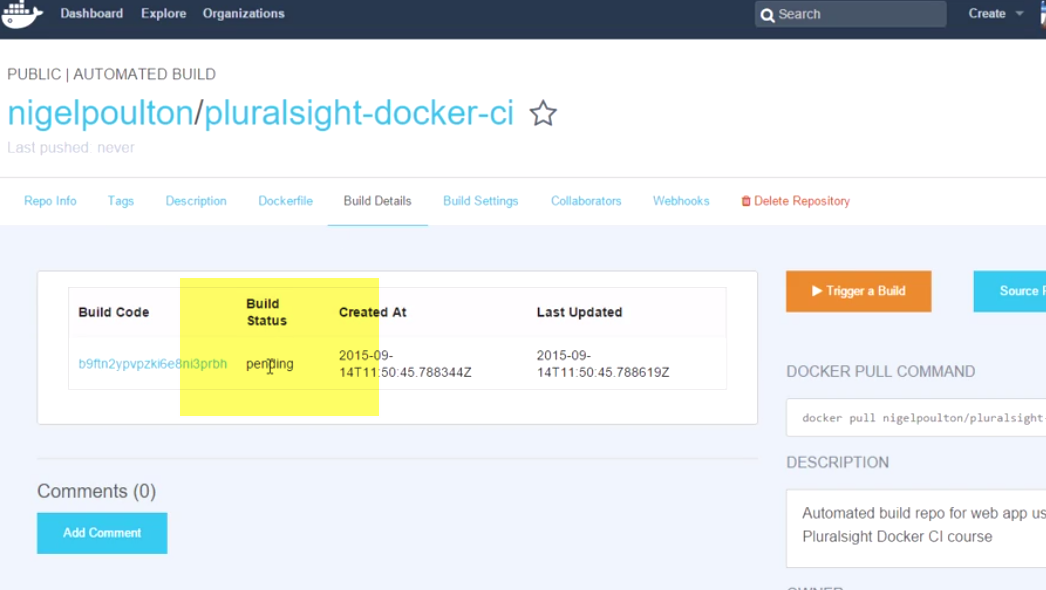
DOCKER\_HUB\_TRIGGER, the name doesn't really matter as long as we can remember it later on,

value we give it this curl command here,



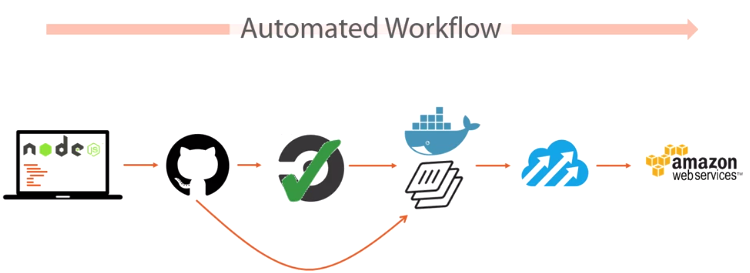
Make changes to file in github->Push changes.

We can see:



A new Docker Hub automated build is running.

So basically, following our green CircleCI build and test run, Circle made that HTTPS post to the trigger URL that we got from Docker Hub, which in turn kicked off a clone of the project repo from GitHub, Dockerfile and all, and started a new automated build of a Docker image

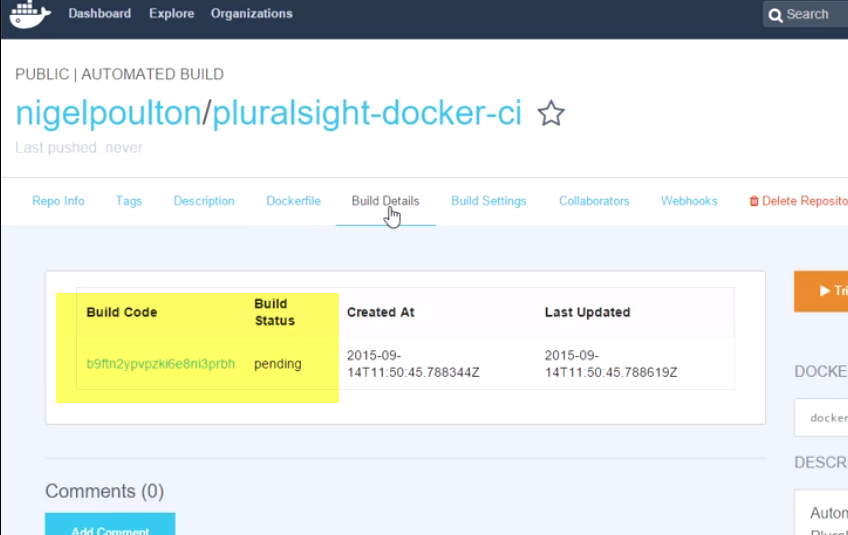


Verifying the Automated Build

Automated build process includes putting your Docker file in the repo, this ends up making your Dockerfile available to anyone who's got access to the repo, and that'll be the whole world.

**Also, committing and pushing changes to GitHub, or Bitbucket if that's what you're using, is the only way, I repeat, the only way to update an automated build repo. So you cannot do a normal Docker push to an automated build repo, I know this because I've tried, and failed, and tried, and failed, then I read the documentation, and well, you live and you learn.**

**We caN CHECK PROGRESS OVER HERE…build details**



Summary

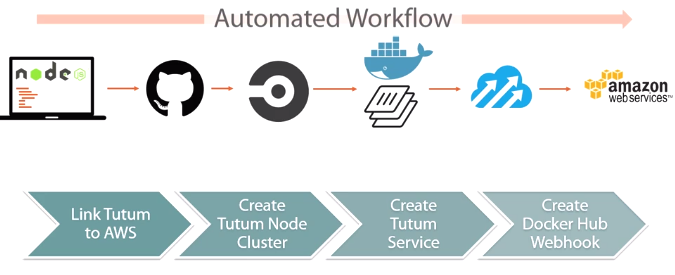
Created our GitHub repo and pushed our code, we've linked Circle with our GitHub repo and told it to track it,

so anytime we push an update it's going to trigger a new Circle test build

we've just hooked up Circle with Docker Hub so that anytime one of those test builds completes successfully, like a green build per say, well, we trigger Docker Hub to create a new Docker image of it.

Next up, we're going to see how to push that Docker automated build image into production.

5\_Pushing Our App to Production



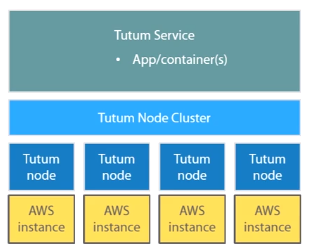
We’ve configured Docker Hub to automatically build a new image every time CircleCI gets a notification from GitHub and runs a successful test build. All that remains now is to push this baby to production..

And according to our workflow diagram here, we want to push it to Amazon Web Services via the Tutum Docker platform.

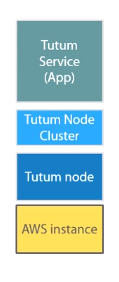
Configure Tutum, first up, so that it can actually deploy to AWS.

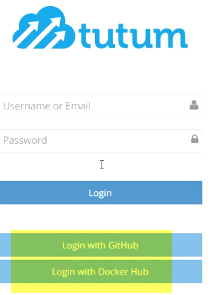
It's going to involve a small amount of Tutum and AWS config.

Configure a webhook on Docker Hub, just a way for Docker Hub to ping Tutum telling it when it's got a new image and that it's time to deploy it to production.



at the very bottom of our picture we've got our AWS instances, these map to Tutum nodes and they're deployed via Tutum, then, several of them can be grouped together into Tutum node clusters, and it's on top of these Tutum node clusters that we deploy Tutum services, and the Tutum service, it runs an app or a container, and it can scale across multiple nodes in a node cluster.

For us though, we're going to deploy a single Tutum node, so a single AWS instance, and we'll configure it as a one-node cluster, not much of a cluster, I get it, then on top of that we'll configure a service to deploy our app into a single container.

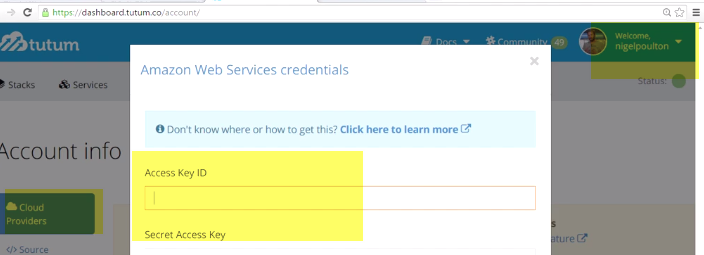


05\_02-Configuring Tutum with AWS

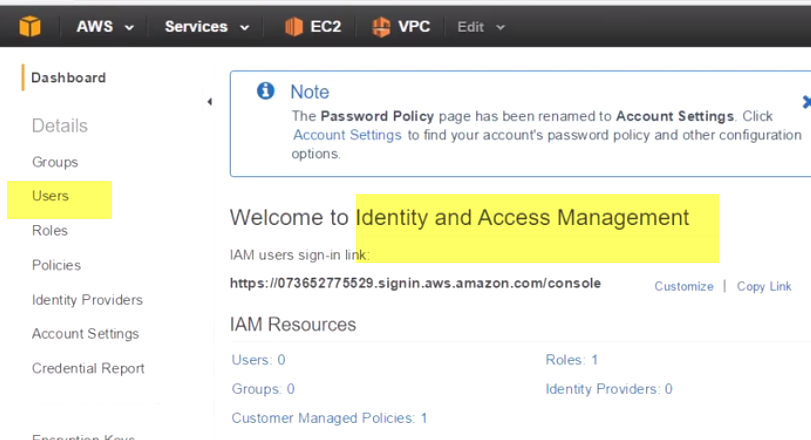
first thing we want to do is link Tutum here to our AWS account, though other cloud providers are totally supported.

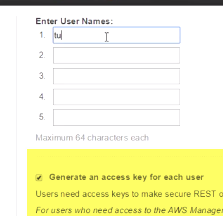
account details -->Cloud Providers over here--->just pick your provider,

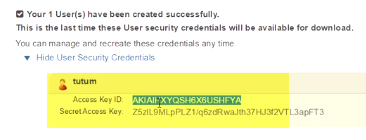
For access key.., I'll go AWS.



**Identity and Access Management, Users, and we want to create a new one,**

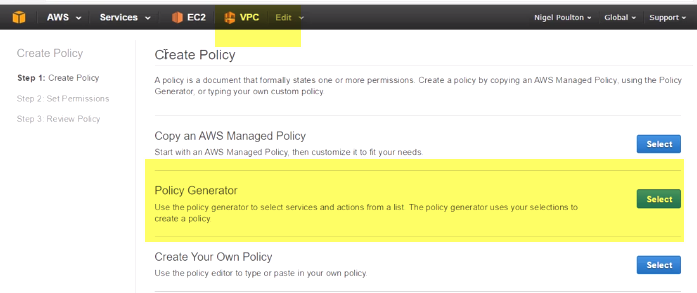




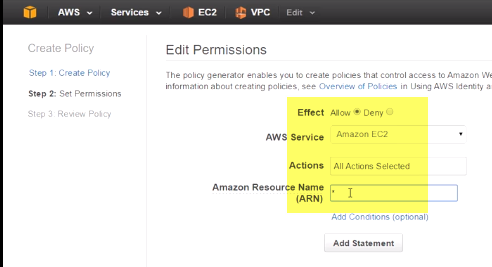


Jump over to Policies. Now then, what we're going to do here is create a policy that allows the Tutum user that we've just created to work with EC2 instances,because remember, we're going to need Tutum to be allowed spin up and tear down instances and the likes, because instances are what Tutum nodes and node clusters are made of, so Tutum needs to be able to manage EC2 instances.

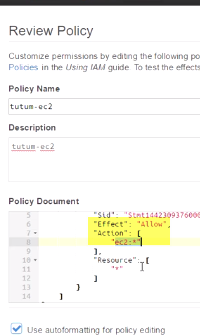
So we go Create Policy,



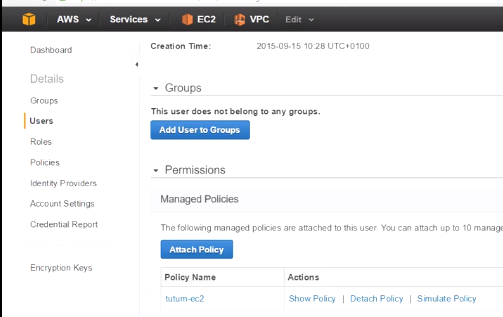
allow EC2



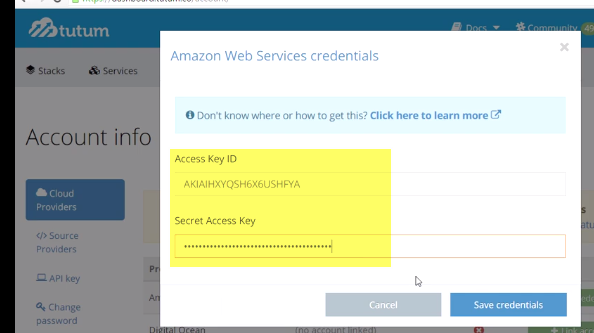
allow EC2-> allow all actions on all resources, Next, give it a name, and a description.



Users--> Attach Policy, and if we scroll all the way down, this is in alphabetical order, there we are, and attach it

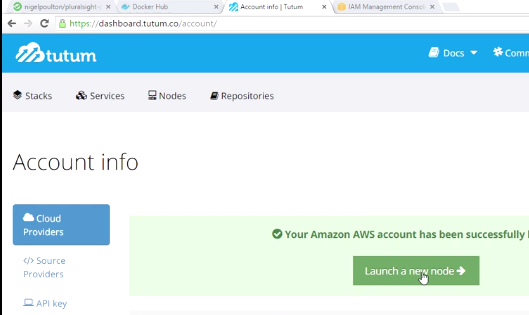


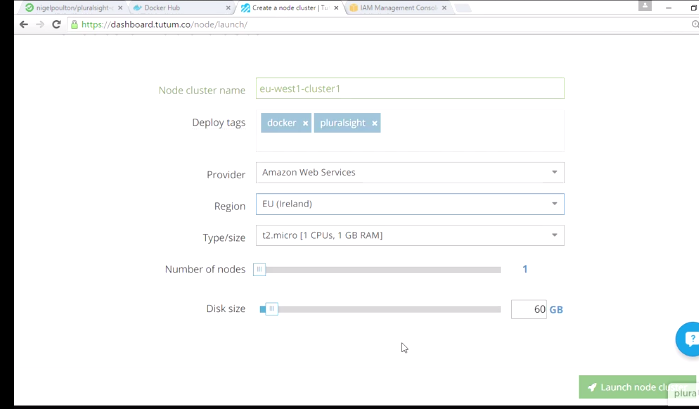
Back to Tutum. Now let's paste those creds in.



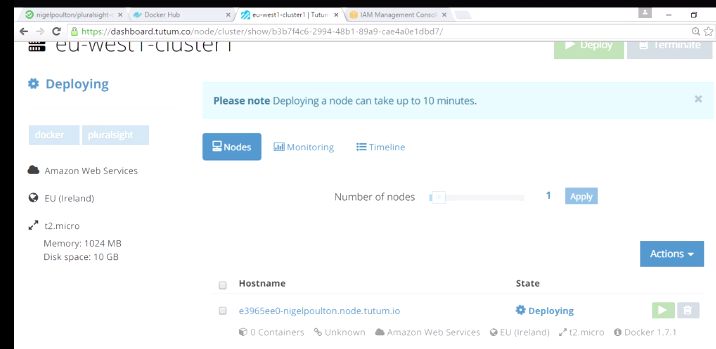
So, now that we're all linked up with AWS, we need to create a Tutum node cluster.

Create Node--> Launch a new node cluster.



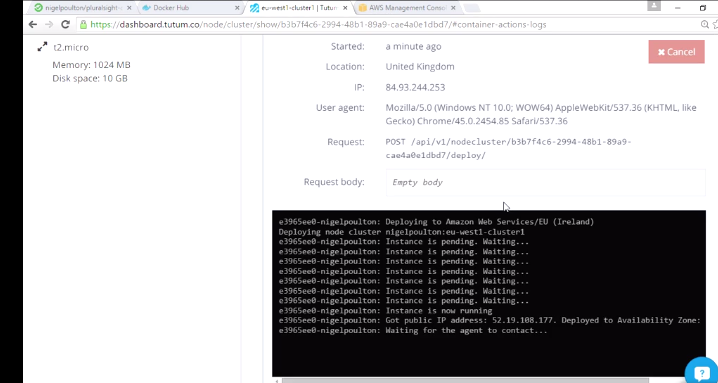


Deploy tags, well I'll go with docker and Pluralsight. Yes we're AWS, and EU Ireland, also known as the EU West1 Region



now we can see the status here, deploying, and we can see its config, so to Amazon Web Services in EU Ireland on a t2.micro running version 1.7.1 of the Docker Engine.

Check Timeline for status

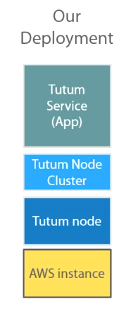


we just configured Tutum to spin up an AWS instance for us, which Tutum maps back to a node within a node cluster.

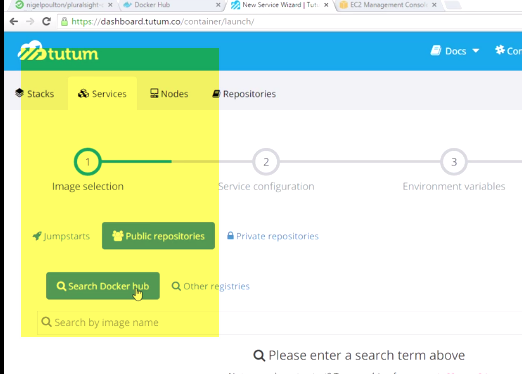
05\_04-Creating a Tutum Service

We’ve got our node cluster, that's our AWS instance, or instances

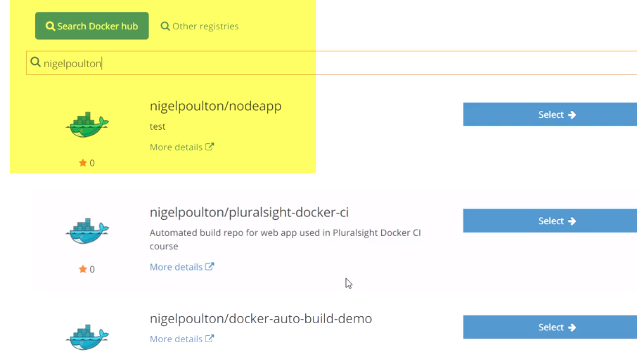
Then remember, on top of this node cluster we overlay Tutum services, and then it's on top of these services that we deploy containers, so our app.

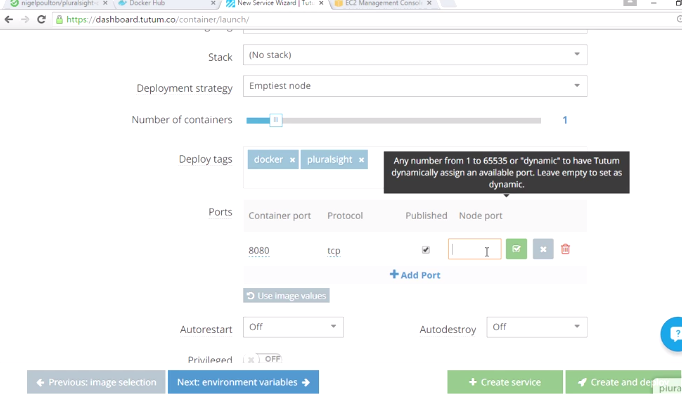


Now then, within a service we can deploy multiple instances of our containerized app, so we could have, I don't know, 10, 50, 100 containers if that's what we needed, they're all just based off of the same image and they run on the same Tutum service.



Anyway, so we go Services up here, and create our first, we want Public repos here, and Search Docker hub.





I guess, publish to the same port or to a different port number. Let's go with 80. Cool. ..Create and deploy down here..

Timeline:



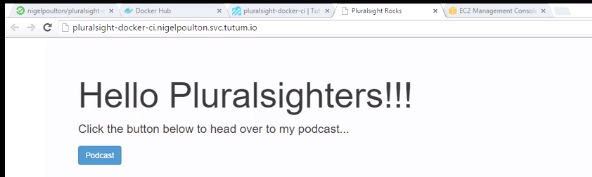
So, it's checked Docker Hub, it's located the newest image, and it's started deploying from it, choosing the best node, well, yeah, okay, not a hard choice, we've only got one, and then it's deploying.

Now if you're familiar with Docker, which I assume most of you probably are, we've got a Docker pull going on here,

downloading the image layers, and when that's done..

final bit of deployment stuff, and we should be in business.





So if we look at our endpoints here, we've got one for the service itself, which if we'd scaled our app over a bunch of nodes and containers, then each container endpoint down here would be load balanced behind the service endpoint. And of course we can do all kinds of DNS magic here with CNAMES and stuff to give it nice, friendly DNS names. Anyway, moment of truth, if we hit this pop out here, yeah there it is, oh and fully working too, brilliant. Now, one last thing for us to do. We need to go and configure a webhook in Docker Hub.

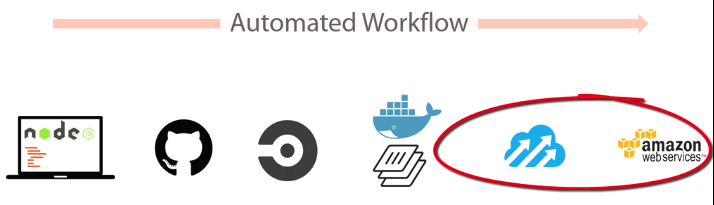
05\_05-Creating a Docker Hub Webhook

Docker Hub supports webhooks..

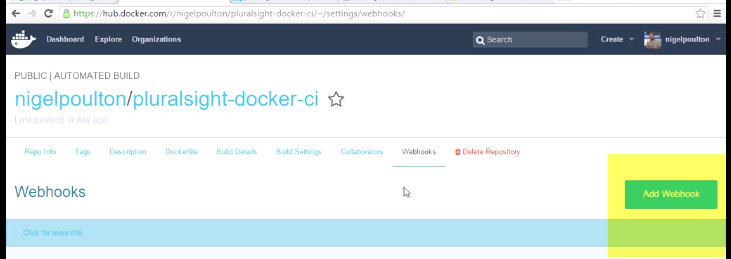
it's a way to make HTTPS post to a remote endpoint, the point of them being, once a successful automated build completes, we can ping the next item in the chain.

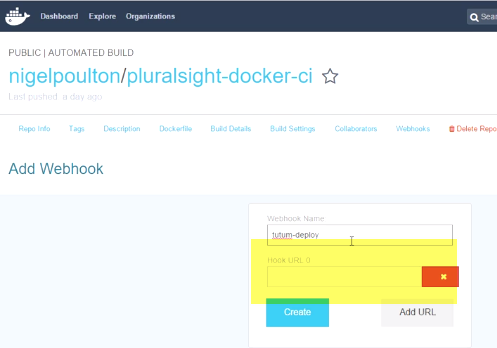
We are at the business end of the workflow right now.

So, we've pushed our code to GitHub, GitHub pings CircleCI via what it calls a service hook, think webhook, Circle does its stuff and assuming a green build, Circle then pings Docker Hub, again think webhook, I'm sure you're getting the picture.

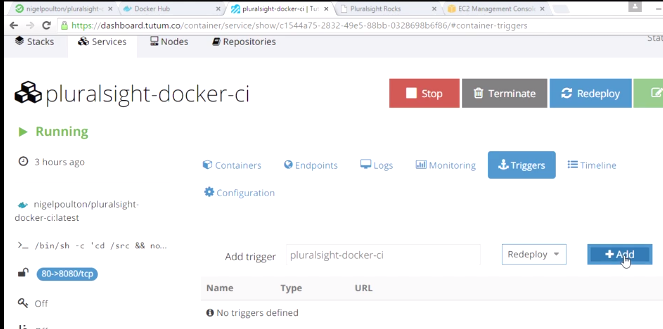


Well, we're all good and automated up to this point, and we've just seeing how Docker Hub automated builds get triggered after the green CircleCI build, but after that, at least as things stand right now, that is where our automation ends. We've not configured any trigger from Docker Hub to Tutum, that is, yet. So, that's where webhooks come into play.

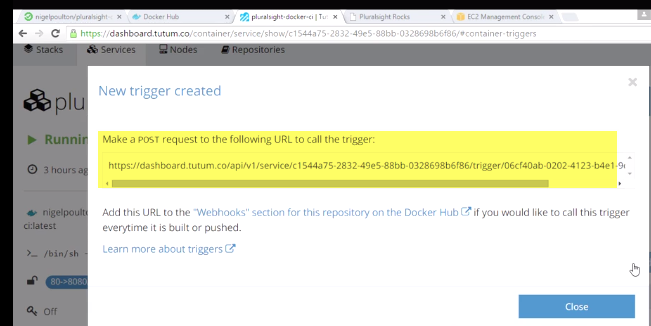


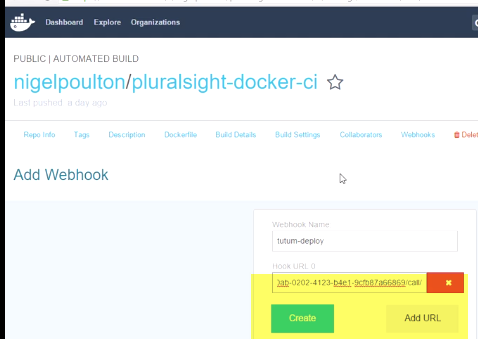


For URL;Got go

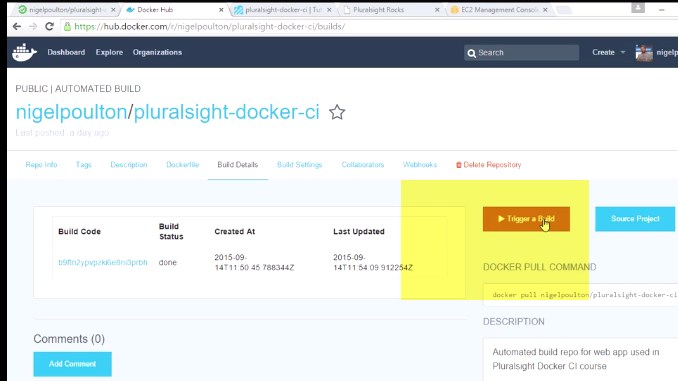


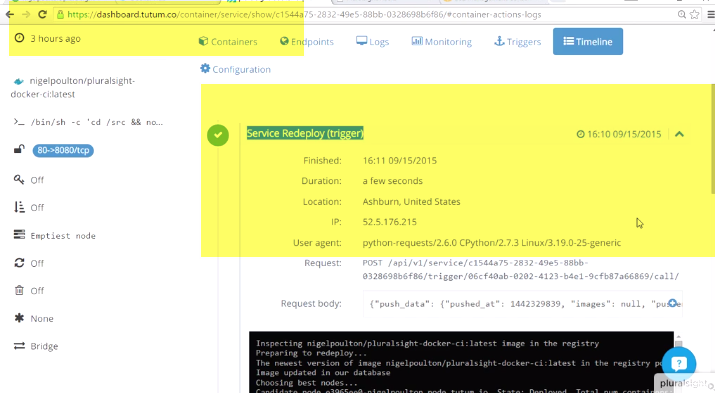
Redeploy





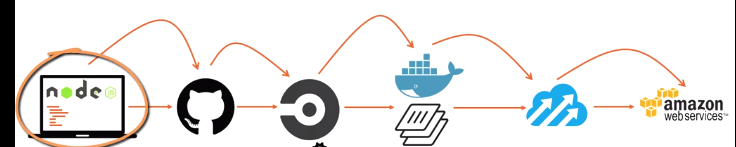
On Dockerhub🡪manuall build





Summary

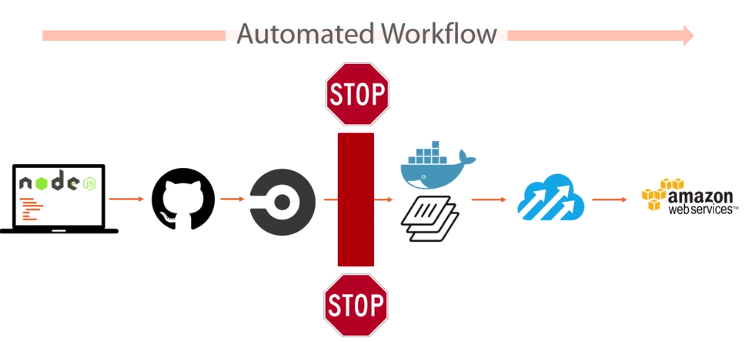
We've gone all the way from the code on our lappy, which if you're a Pluralsight Plus subscriber you've got in the course materials, but we've gone from that code through---> pushing to GitHub-->linking GitHub to CircleCI--> configuring and running test builds--> pushing green test builds to Docker Hub---> having Docker Hub create new images--> and having Tutum pull those images from Docker Hub --> push them to the world via Amazon Web Services, and the beauty of it, is automation.



Any changes we make to our app code over here, as soon as we push to GitHub, it's going to automagically go through the entire automated workflow.

So Circle's going to run our tests, if they pass it's going to trigger an automated build on Docker Hub, when that build is done Docker Hub's going to ping Tutum, and Tutum's going to Docker pull and launch a new container based on the updated image with our updated app code in it. I love it. So what's left? Well, next up we're going to break our app and see what that does to our workflow.

06. Breaking and Fixing the App



Vital part of the flow is the build test we've got configured in Circle, and if that test fails, then the broken build absolutely must not leak through to production. For us, production is AWS. It’s actually worthless if it can't stop an error leaking through and breaking production.

We're going to introduce an error into our code and we'll push it to GitHub, this will trigger a test build on Circle, test fail and our workflow had better grind to a halt.

If workflow automated, the fix is just going to sail right through to production without any effort from us.

Breaking the App

To simulate the break, we edit our app.js file here and if we change the location of our views to which doesn’t exists

git add app.js, git commit, push to GitHub.

Now then, because all this hocus-pocus is automated---------> Circle has queued it.

We’ve got red.

A decent feature of Circle is the ability to SSH into a broken build. So, if we scroll up here and we hit the With SSH button.

Git Bash--Login-->inside build container that's running our test.

Circle uses containers to run build tests, but this container that we're in is not built per our Dockerfile.

Anyway, we're in and we can poke around. So let's find our app.

And I think it's, yep, this is it. So, let's see if there's anything listening on port 8080. Nope. Okay, I'm not seeing anything with node in it running there.



2nd way to ssh:



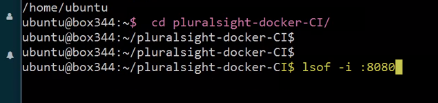
Adhoc Fixing

A decent feature of Circle is the ability to SSH into a broken build. So, if we scroll up here and we hit the With SSH button.

Git Bash--Login-->inside build container that's running our test.

Circle uses containers to run build tests, but this container that we're in is not built per our Dockerfile.

Anyway, we're in and we can poke around. So let's find our app.

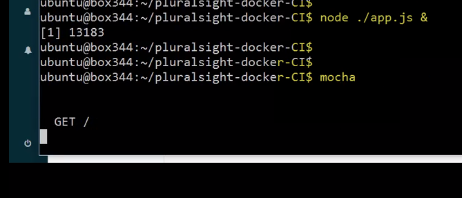


--nothing

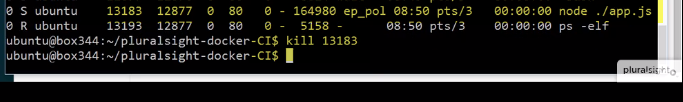
Ps –elf

And I think it's, yep, this is it. So, let's see if there's anything listening on port 8080. Nope. Okay, I'm not seeing anything with node in it running there.

Let's try running our app manually, and run our test again, okay, **same error.**

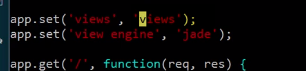


Let's kill our app,

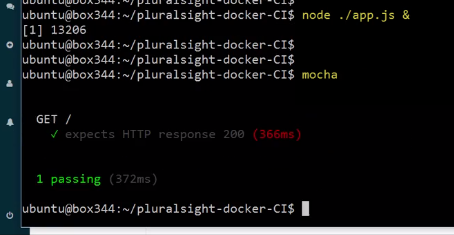


let's see if we can fix the problem while we're SSH'd in. So let's come on down and we'll get rid of that.





run our app again, let's try Mocha



all fixed.

We totally knew what the issue was before we started. But you get the picture, we can SSH in, mess around, try out various fixes, all of that good stuff. Cool, well now let's go and fix it in our Git repo and see it all sail through to production.

Fixing the App

With our fix in hand, let's jump back to our code and make the fix in the real world.

