CircleCI

Monday, September 3, 2018 9:02 PM

Introduction - this is how HiDeoo got me started for Bot Land

HiDeoo gave me some notes here on how to get started:
- https://circleci.com/
- Log In.
- Login with BitBucket.
- Grant access.
- Add Projects.
- Find "botland" in the list.
- Click on the "Set Up Project" button associated to Bot Land on the right.
- Operating system: Linux
- Language: Node

Then, below, they show you 5 steps that you have to follow. DO NOT FOLLOW these steps. Don't add anything or any file to the repository. (I should do that later myself in the associated PR).
- The last step should have on the right a "Start building" button. Click it.
- It should start building and immediately fail as there is no config.

From here, I think we should be fine. I'm using "think" as I've used CircleCI a lot, for most of my projects, at work, etc. but only for Github & never for BitBucket. I also can't be made an admin on CirclCI as permissions are based on the Github Organization / BitBucket team, etc. so I would need an admin permission on BitBucket to change settings on CircleCI (and that's a big no from both parts).
If there is any issue, I hope we should be able to figure it out, me telling you what to change or in the worst case scenario, a call / screensharing session.

From <https://bpaste.net/raw/fde3eb354fef>

- Create a new Sinopia user as CircleCI will need to pull down @botland/* packages.
- If you have not moved from the previous page, you should be in the "botland" project page with the first build failing.
- On the top right, there is a gear to go to the project settings.
- Environment Variables
- Add Variable
  - BL_REGISTRY_URL -> http://104.42.195.223:4873
  - BL_REGISTRY_USER -> newly created user name
  - BL_REGISTRY_PASSWORD -> newly created user password

Only you (and any other admin from BitBucket) will see these variables content, they'll be usable from our integration scripts but always hidden to users.

From <https://bpaste.net/raw/ed05d35f15fb>

- Settings → Advanced Settings → ● Build forked pull requests (On) - this way, PRs will trigger CI.
- Settings → Advanced Settings → ● Auto-cancel redundant builds (On)
- Settings → Advanced Settings → ● Pass secrets to builds from forked pull requests (On) - only do this if you trust people who can submit pull requests

Basics

- Jobs/workflows (reference)
  - Workflows can be set for manual approval, that way they don't run without user intervention in the UI (reference)

    - All workflows are run that aren't set for manual approval (reference) or time-based scheduling (reference)
    - In GitHub, if you don't want to have to wait for manual jobs in order for the status to be green, then you can update settings on GitHub (reference)
        - Summary: go to Settings → Branches → Add a new branch protection rule that applies to "*" or "something-*"
- Building Docker images (reference)
- Using YAML anchors and aliases so that you don't have to repeat yourself (reference)
- Configuration reference (reference) - this includes all YAML commands like "run"
- Built-in environment variables like CIRCLE_SHA1 (reference)
- CircleCI Dockerfiles can be found here
- As of October 2018, they gave 1,000 free minutes per month.
- You *can* specify multiple Docker images like this:

  ```
  docker:
    - image: circleci/node:8.12
    - image: circleci/mysql:5.7.17
  ```

  However, what this is doing is creating two separate containers. So the Node image can access MySQL because MySQL is a service exposed to localhost (in much the same way that the official Docker tutorial lets you use Redis from its own container), but something like this may not work how you expect:

  ```
  docker:
    - image: circleci/node:8.12
    - image: circleci/python:3.6.1
  ```

  Here, there will still be two separate containers, but since Python is an executable and not a service, you won't be able to run python3 from the Node container. If you want to run Python on a Node image specifically, read this section of the notes. If you just want to have multiple arbitrary images combined in one with whatever executables you want, then make your own Docker image (see this section).
- When making a configuration, run whatever commands you want to make "debugging" easier, e.g.:

  ```
  - run:
      name: Print environment variable
      command: echo Env var is $some_env_var
  ```

An executor is what kind of back-end to run on: Docker, machine, or macos (useful for iOS apps).

Running Python on a Node image

The easy way of running Python on a Node image

There are two solutions here:

- SUPER EASY: find an image that has Python and Node on Docker Hub, e.g. this one.
- EASY: make an image yourself - you may need to include parts of the CircleCI images (e.g. their extension to Python installs "jq" for parsing JSON data for the AWS CLI and installs Docker for building images). See below for an example Dockerfile.
    - You don't even need to push this to DH if you're only going to use it locally

Here's the Dockerfile that I ended up creating:

```
FROM circleci/python:3.6.1

COPY --from=circleci/node:8.12 /opt/yarn-* /opt/yarn
```

```
COPY --from=circleci/node:8.12 /usr/local/bin/node /usr/local/bin/
COPY --from=circleci/node:8.12 /usr/local/lib/node_modules
/usr/local/lib/node_modules
RUN sudo ln -s /usr/local/lib/node_modules/npm/bin/npm-cli.js
/usr/local/bin/npm && \
    sudo ln -s /usr/local/lib/node_modules/npm/bin/npx-cli.js
/usr/local/bin/npx && \
    sudo ln -s /opt/yarn/bin/yarn /usr/local/bin/yarn && \
    sudo ln -s /opt/yarn/bin/yarnpkg /usr/local/bin/yarnpkg

CMD ["/bin/sh"]
```

Explanations for the above:

- It's easier/faster to copy Node than it is to install Python (see installing Python the hard way), which is why I start with the Python image and add Node in.
- I copy /opt/yarn-* so that I don't have to specify the Yarn version. This means that it gets saved to a folder named "yarn" rather than "yarn-v1.9.4" or whatever it is in circleci/node:8.12.
- The symlinks are to mirror how circleci/node:8.12 operates.
- The reason I sourced a circleci Docker image to begin with is so that I didn't have to worry about installing JQ or Docker. Originally, I had this configuration, but then Docker wasn't installed:

```
FROM nikolaik/python-nodejs:latest

RUN apt update && apt install -y jq

CMD ["/bin/sh"]
```

The hard way of running Python on a Node image

OH BOY. This took *so long* to figure out. The overall summary:

- You probably don't want this; read the easy way to do this.
- I needed Python 3.6, not 3.4.
- The steps to copy into config.yml are below.
- This ends up adding a lot of time to the build process. On my relatively beefy machine, it took 50% of my physical CPU for about 70 seconds just for the Python step. This is only needed when running locally since you can just use the circleci/python:3.6.1 Docker image online and have all of this in an instant (you'd be able to do that locally too if you could cache previous steps; see below for full explanation).
- Eventually, I should be able to just install Python3 (and maybe python3-pip if needed) using "apt"

Here was the scenario I was going for:

- I wanted to deploy a container that I built to AWS
- I wanted to test this locally

Testing locally doesn't let you run more than one job at a time, so I needed to combine the "build" and "deploy" jobs into a single, mega-job. That was problematic because building requires Node, and deploying requires Python, and those don't operate nicely across two containers. I figured I would just use the Node Docker image and install Python onto it, but I kept getting a problem where "ensurepip" wasn't installed:

```
# python3 -V
Python 3.4.2

# pip3 -V
pip 1.5.6 from /usr/lib/python3/dist-packages (python 3.4)
```

python3 -m venv venv
Error: Command '['/venv/bin/python3', '-Im', 'ensurepip', '--upgrade', '--default-pip']' returned non-zero exit status 1

python3 -Im ensurepip
/usr/bin/python3: No module named ensurepip

# python3
>>> help('modules')
(it doesn't list ensurepip - no idea why though)

I installed the circleci/python:3.6.1 Docker image and everything was set up correctly there, so I hunted down the closest possible Dockerfiles that I could find:

- Python 3.6 for [Debian] jessie
- CircleCI's extension of Python 3.6.3 (note: this doesn't actually extend the one above, but it's close enough)

In the Python 3.6 image, we can see that they wget get-pip.py from some pypa.io. I tried this out myself in Python 3.4, but it didn't help. I concluded that I needed Python 3.6, which unfortunately isn't in the main Debian repositories. I found instructions on how to install Python 3.6 from a tar file here. The instructions include running "./configure --enable-optimizations", which I believe produces an optimized Python build. That's nice, but the tests took >5 minutes to run, which IMO was unacceptable. I found instructions here on how to skip running the tests. This gave me a final set of steps that I could add to the config.yml:

```
- run:
    name: Install Python3
    command: |
      wget https://www.python.org/ftp/python/3.6.3/Python-3.6.3.tgz
      tar xvf Python-3.6.3.tgz
      cd Python-3.6.3
      ./configure
      make -j8
      sudo make altinstall
      sudo ln -s python3.6 /usr/local/bin/python3
```

To test this, I just made a container from a Node image, connected to it with " docker exec -it -u 0 {container ID} /bin/sh", and ran the individual steps starting with wget and ending in "sudo make altinstall". It produces /usr/local/bin/python3.6. Running that and typing help('modules') shows that "ensurepip" is installed correctly, which meant that "python3 -m venv venv" will work from this line of their config.yml.

Skipping CI (reference)

If you don't want CI to run for any of the commits in an entire push, then add "[skip ci]" or "[ci skip]" anywhere in any commit.

Deploying to AWS ECR (Elastic Container Registry) (reference)

These notes are helpful if you want to build a Docker image and push it to ECR.

> HiDeoo: It builds & dockerize an app and then push it to ECR and deploy it to ECS Fargate which is your exact use case Adam13531 so you'll be covered

The final config.yaml that *they* produce from these instructions is here. That repo also contains other mandatory files like requirements.txt (so that the AWS CLI can be installed) and deploy.sh (which is responsible for putting the container on Fargate).

I specifically do *not* want to have to commit any Docker-related resources (e.g. Dockerfile or the ECS task definition) from CI. So, for example, the Docker image that we create should be tagged with something like "latest" so that I don't have to update the task definition.

The steps start with using Terraform to set up a VPC, a role, a cluster, a service, and an ECR repository. I did all of that manually, so I skipped those steps. I'm also not planning on using Terraform. Because of that, I needed to manually add these environment variables in CircleCI (Workflows → Click the ⚙ of the project you want → Build settings → Environment variables):

- AWS_ACCESS_KEY_ID
- AWS_SECRET_ACCESS_KEY
- AWS_DEFAULT_REGION
- AWS_ACCOUNT_ID
- AWS_RESOURCE_NAME_PREFIX

I didn't use their $FULL_IMAGE_NAME stuff because

I learned quite a bit here for how to choose the executor for something like this. In my case, I need Node/Yarn to be able to even call my "docker:build" script from my package.json, so I would want a base image of something like "circleci/node:8.12" so that I have Node and Yarn. I also need "setup_remote_docker " since it's for building Docker images in a secure environment. Note that "setup_remote_docker " is not compatible with "machine" or "macos" executors, so you *have* to use Docker.

> Because I'm using circleci/node as my Docker image, I don't need to install the Docker client since they've already done that in the image.

CLI and testing locally (reference)

Basics

- If you want to test a configuration without burning through all of your monthly minutes, you can clone your repository and test locally. Keep in mind that this does not export environment variables as those are encrypted (reference). You'll need to do "-e key=value" to test that.
    - To obscure this while streaming, you can make a shell script that exports the environment variables (i.e. just a bunch of "export access_key=secret" lines), then "$ source" that script so that your current shell gets the variables, then you can specify "-e access_key= $access_key". This is not *actual* security, it's just one extra layer to make it a little more difficult to leak secrets.
        - Keep in mind that in Bash, there are lots of characters that you have to escape, so it's best to just surround your variables in single quotes, e.g. "export secret_key='\{}bunch of garbage[]:={'"
- You cannot use the CLI to run a workflow; you have to run individual jobs. That should be fine this since workflows are just a combination of jobs anyway; you can make a giant job or separate jobs just for testing locally.
- Installation (reference)
    - Get the circleci executable (reference)
    - You have to create a token for this
    - The token gets saved into ~/.circleci/cli.yml
    - Note: they tell you to validate your configuration without necessarily telling you to clone a repo, so "circleci config validate" is going to fail since it can't read the configuration from your home directory by default (although you *can* specify it manually as an argument).

When they say "clone a repo", they just mean your Git repo that happens to have a .circleci/config.yml. This is why you should be storing your config.yml in your repository rather than in their back-end.

- Running is only done via "circleci local execute", of which "circleci build" happens to be an alias for. By default, the job name is "build", but you can override it with "--job JOB_NAME". Note that this *always* runs in a container, so you won't see any changes to your local filesystem.
- Locally, you can't explore the whole container after CircleCI is done. If you want to be able to do that, I think you may be able to explore the "--volume" option so that the host can store what's in the container.
    - The way you specify volumes follows Docker's format:
        - [14:07] v1cus: It is -v [PATH_ON_YOUR_MACHINE]:[PATH_IN_THE_DOCKER_CONTAINER]
        - I never got this to work. I think the issue may be this one, but I'm honestly not sure. The command I tried was circleci local execute --job verdaccio_buil d --branch adam-verdaccio -v "/home/adam/docker_volumes:/home/circleci/botland" and the error I got was "Error: Unable to create directory '/home/circleci/botland/packages/verdaccio'". I just gave up and switched to using "ls" from inside the container.
    - Remotely, you can choose "rerun job with SSH" in the "rerun workflow" dropdown, but this costs monthly minutes.
    - As a workaround, you can just run "ls" in the container from the config.yml. That's for chumps though.
        - **As a BETTER workaround**, you can add a "sleep 10000" to the config.yml and then connect to the container:
            - Add this step

```
- run:
    name: Testeroni
    command: sleep 10000
```

            - When that step gets hit, do a "$ docker ps" and look for the container that does not have "circleci build" in it.
            - Connect to it with docker exec -it -u 0 {container ID} /bin/sh
            - "top" or any other built-in command that doesn't exit would also be fine.
            - Keep in mind that you only get 10 minutes no matter what you do, then you'll get an error saying "Error: Too long with no output (exceeded 10m0s)".
- You can specify which branch you want to build with via "--branch BRANCH_NAME". This is helpful if your files are not in the master branch, which is what's assumed by default.
- Your code is likely checked out via the "- checkout" command, so any changes you make to versioned assets needs to be pushed to the branch that you're specifying via "--branch BRANCH_NAME" when running CircleCI.

After all of the above, here's a sample build command that I use:

```
circleci build --job verdaccio_build --branch adam-verdaccio
```

How I run this locally from Windows (well, a VM in Windows)

- I set up a Linux VM and SSH into it
- I use FileZilla to edit config.yml over SFTP
    - **MAKE SURE TO UPLOAD FILES EVERY TIME YOU SAVE THEM.** FileZilla has had requests for 10+ years to allow auto-uploading files after modifications, but they keep turning them down.
    - **MAKE SURE NOT TO HAVE CRLF LINE ENDINGS OR YOUR EXECUTABLE SCRIPTS WILL BREAK WITH CRYPTIC ERRORS**
- I make .circleci/secret_env.sh (which I edit over FileZilla as well). The contents need to roughly

mirror what's in the environment variables in production.
    - export foo=bar
    - export baz=qux
- I source that secret_env.sh so that the "-e" parameters below will work
- I use the localcircleci Docker image that I made. For more information, look at this section.
- Anything with "attach_workspace" that later attempts to use "workspace/" should just drop the "workspace/" part. E.g. below, you would delete the bolded part:

```
- attach_workspace:
    at: workspace
- run:
    name: Load image
    command: |
      sudo docker load --input workspace/docker-image/image.tar
```

You also have to make sure the working directory is correct, otherwise ./docker-image may not exist.

- I put everything into a single "mega" job for testing since workflows don't work locally.
- I run the command to build: $ circleci build --job verdaccio_build_and_deploy --branch adam-verdaccio -e AWS_ACCESS_KEY_ID=$AWS_ACCESS_KEY_ID -e AWS_ACCOUNT_ID=$AWS_ACCOUNT_ID -e AWS_DEFAULT_REGION=$AWS_DEFAULT_REGION -e AWS_RESOURCE_NAME_PREFIX=$AWS_RESOURCE_NAME_PREFIX -e AWS_SECRET_ACCESS_KEY=$AWS_SECRET_ACCESS_KEY -e htpasswd=$htpasswd
- When I'm done, I undo anything from above, e.g. I have to split jobs back into separate jobs/workflows.

localcircleci

This is the name of the local Docker image that I built specifically for myself to be able to run CircleCI. There are two major things that this does:

- Set up all of the required programs (Node, Python/PIP, JQ, Docker) from a single image since we have to make a single, "mega" job rather than being able to use workflows
- Set "USER root" so that I don't have to specify "sudo" all over config.yml, even in some non-obvious places like "yarn docker:*" or "aws ecr get-login". Being root one way or another is needed to avoid this problem.

To build this Docker image, navigate to packages/verdaccio and run "yarn docker:build:localforcircleci".

Troubleshooting

Catch-all "some command didn't work"

- Is your working directory correct? E.g. I was trying to run "yarn docker:save", but I was in the wrong directory, so it said "command not found".

: No such file or directory

This happened to me due to line-ending issues (of all reasons).

I was editing on Windows through FileZilla (so Git wasn't involved to be able to normalize endings just messed up line endings this entire time, so the whole file ended in CRLF instead of LF, so the #! at the top (#!/usr/bin/env bash) tried finding a path ending in a CR which did not exist.

We encountered a problem adding your identity to CircleCI…

"The identity provided by your VCS service is already associated with a different account. Please try

logging in using that account or contact us for further help."

I don't know *why* this shows up exactly, but you can't just click "Go to app" to fix it since that will be a different session. An incognito window works, but so does clearing site data via F12 → Application → Clear storage → Clear site data. I believe I had to do this on both GitHub and Circleci, but it said circleci was storing 0 bytes as it is.

Your repo is called "project"

(…and potentially resides at ~/project)

This is the result of not having a working_directory at the job level (reference).

Permission denied to Docker daemon socket

Note: I was trying to run "circleci" locally with a config.yml that tried doing "docker build".

The error is here:

> Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Post http:///Fvar/Frun/Fdocker.sock/v1.38/build?buildargs=%7B%7D&cachefrom=%5B%5D&cgroupparent=&cpuperiod=0&cpuquota=0&cpusetcpus=&cpusetmems=&cpushares=0&dockerfile=Dockerfile&labels=%7B%7D&memory=0&memswap=0&networkmode=default&rm=1&session=7si3jiwks4jkgtw0ibnbij2x9&shmsize=0&t=botland/Fverdaccio%3A1.0.0&target=&ulimits=null&version=1: dial unix /var/run/docker.sock: connect: permission denied

The resource that finally helped me figure out what to do is this discussion.

Overall conclusion: any "Docker *" commands in the config.yml should have "sudo" before them when testing locally. **Make sure to remove "sudo" before pushing.**