DEC 23, 2015

Git プッシュから Amazon ECS に自動デプロイする仕組みを構築する

この記事は Container with AWS Advent Calendar 2015 (http://qiita.com/advent-calendar/2015/container-with-aws) の 23 日目です.

昨日は inokappa (http://qiita.com/inokappa) さんの （ショロカレ 21 日目）ずっと待ってた Amazon ECR を一瞬、使ってみた (http://inokara.hateblo.jp/entry/2015/12/22/092353) でした. 僕自身も GA のニュースを聞いて早速さわってみたんです (https://toris.io/2015/12/happy-general-availability-of-amazon-ec2-container-registry/)が、自前運用の Docker Registry から脱却できそうでうれしい限りです. はやいとこ東京リージョンにきてくれないかしら.

さて、本日は Git push を契機とした Amazon ECS への自動デプロイの仕組みを構築しよう、という企画です.

意外とめんどくさい Amazon ECS への手動デプロイ

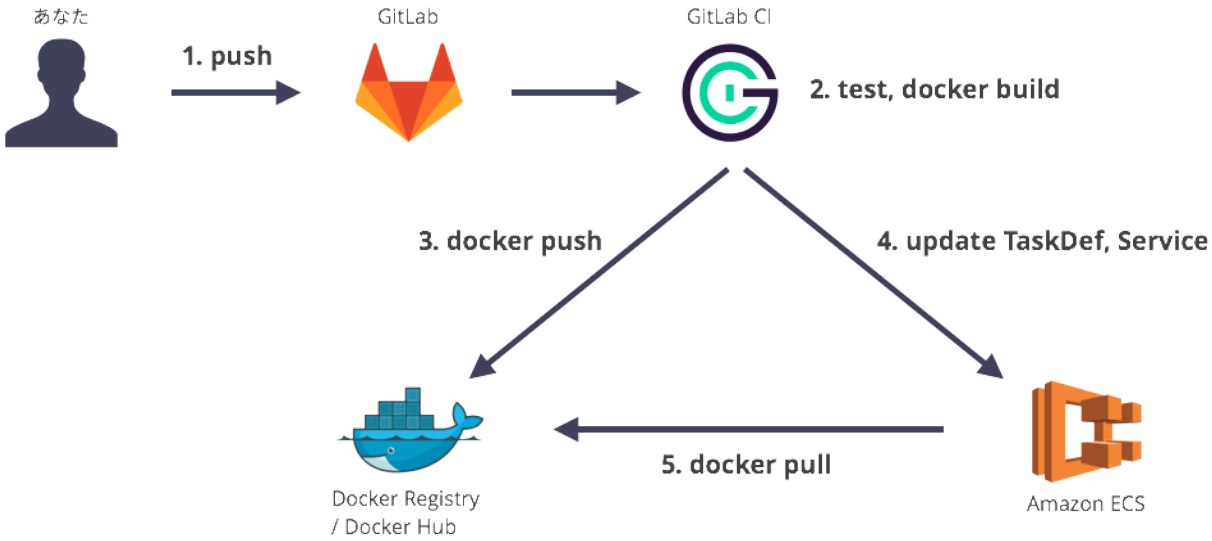
Amaozon ECS に新バージョンのコンテナを手動デプロイしようとすると、何も考えないと以下のようになります.

- Task Definition を更新
- Service の参照する Task Definition を上記のものに変更する
- 待つ

こうやって書くとそんなに面倒ではなさそうに見えますが、常に最新の master ブランチが動いている環境で手作業デプロイをやると思うと… 恐ろしくて夜も眠れません.

こんな感じのデプロイの仕組みはどうでしょうか

基本的な流れを図にしてみました.



僕の所属するチームでは GitLab + GitLab CI を利用しているのでそれを例にしてますが、いわゆる Git リモートリポジトリ + CI システムが動いていれば同じ流れが実現できます. (関係ないけど僕は GitLab CI のことが大好きです)

図について簡単に並べると、以下のようになります.

1. git push
2. CI システムでテスト + Docker イメージのビルド
3. Docker イメージをプッシュ
4. Amazon ECS の Task Definition の中身と Service の設定を更新
5. Amazon ECS が最新の Docker イメージを pull してきて走らせてくれる

いいですね. 人間がやることは git push だけです. 怠惰こそ美徳です.

CI システムから Slack とかに通知する仕組みがすでにある場合は、新しいコンテナが走ったらお知らせするようにしてもいいかもしれません.

こんな感じです

こういう仕組みを構築しようとしたことがある人はすぐに気づかれたと思いますが、上記の図の中でちょっと手間がかかりそうなのは「4. Task Definition の中身と Service の設定を更新」 ですね.

ですが、その話に進む前にまずは試しに GitLab CI の作業手順書である `.gitlab-ci.yml` のサンプルを貼ってみます. (travis.yml とか circle.yml のようなものです)

```
stages:
  - test
  - build
  - deploy
  - clean

# テスト
test:
  stage: test
  script:
    - export APP_ENV=testing
    - ./scripts/test.sh

# ビルド
build:
  stage: build
  script:
    - ./scripts/compile_assets.sh
    - docker build -t my.registy/xxxxxx/image:$CI_BUILD_REF
    - docker push my.registy/xxxxxx/image:$CI_BUILD_REF

# デプロイ
deploy:
  stage: deploy
  script:
    - ./scripts/ecs-deploy -c my-cluster -n my-service -i m
y.registy/xxxxxx/image:$CI_BUILD_REF -t 300
  when: on_success
  only:
    - master

# お片づけ
clean:
  stage: clean
  script:
    - docker rmi my.registy/xxxxxx/image:$CI_BUILD_REF
  when: always
```

どこかの CI サービスのそれとちょっと似ている気がしますが気にしないことにしましょう.

Git プッシュを契機にこの定義が実行され、

- test, build, deploy, clean が順番に流れる
- deploy については 前のステージが全て成功(when: on_success)かつ master ブランチへのプッシュのときのみ

という簡単な内容です.

(僕がこの仕組みを運用しているのはプロダクトの検証環境のため、 master ブランチのみデプロイが走るようにしています. どのブランチへのプッシュでもデプロイが走って欲しいのであれば、こういった制約をかけなくていいと思います.)

まずは Docker イメージを作る

test ステージは置いておいて、まずはビルドのところを見てみましょう.

```
build:
  stage: build
  script:
    - ./scripts/compile_assets.sh
    - docker build -t my.registy/xxxxxx/image:$CI_BUILD_REF
    - docker push my.registy/xxxxxx/image:$CI_BUILD_REF
```

build の時のタグに `$CI_BUILD_REF` という環境変数を渡しています. GitLab CI においては Git のコミットハッシュです.

Amazon ECS は Task Definition と Service を更新しても、Docker イメージ名とタグに変更が入っていない場合は docker pull してくれないため、毎回違うタグでイメージを作成する必要があります. (ただ、このタグが変わってないとダメっていうのは僕の勘違いかもしれないのであんまり自信はないです. ドキュメントにも特にそういう記述はない感じ (http://docs.aws.amazon.com/AmazonECS/latest/developerguide/update-service.html). ECS GA 当初はタグにブランチ名を使う運用もしていたので、同じ名前+タグでも明示的に pull してくれていた気もする…)

Docker イメージがどの時点のリビジョンから作成されたかを知りたい場合のインジケーターにもなるので、ここの値は変えておいて損はないです.

デメリットがあるとしたら CI サーバー で docker images コマンドを実行したときに同じイメージ名で違うタグのものが大量に並ぶことくらいでしょうか. 大したデメリットではありませんね. (上の方に挙げた .gitlab-ci.yml の例で言うと、ジョブの最後に docker rmi でイメージを消しておくことで CI サーバーのディスク容量がヒーヒー言うことをなるべく回避しようとしています.)

Docker イメージができあがったら Amazon ECS へ

```
deploy:
  stage: deploy
  script:
    - ./scripts/ecs-deploy -c my-cluster -n my-service -i m
y.registy/xxxxxx/image:$CI_BUILD_REF -t 300
  when: on_success
  only:
    - master
```

4 行目の `./scripts/ecs-deploy` が本日の主役となるコマンドで、GitHub の silinternational/ecs-deploy (https://github.com/silinternational/ecs-deploy) で公開されているシェルスクリプトです.

```
./scripts/ecs-deploy -c my-cluster -n my-service -i my.regis
ty/xxxxxx/image:$CI_BUILD_REF -t 300
```

という記述で、

- my-cluster というクラスターで動いている
- my-service というサービスが参照する Task Def の image パラメーターを
- my.registy/xxxxxx/image: $CI_BUILD_REF という値に更新して
- サービスが最新の Task Definition で動いているかどうかを 300 秒を限度に待つ

という動きを実現できます.

本日の主役 ecs-deploy

ここからは ecs-deploy の中身を俯瞰していきます.

このスクリプトは内部的に awscli (https://aws.amazon.com/cli/) を使っており、以下の API が利用されています.

- describe-services (http://docs.aws.amazon.com/cli/latest/reference/ecs/describe-services.html)
- describe-task-definition (http://docs.aws.amazon.com/cli/latest/reference/ecs/describe-task-definition.html)
- register-task-definition (http://docs.aws.amazon.com/cli/latest/reference/ecs/register-task-definition.html)
- update-service (http://docs.aws.amazon.com/cli/latest/reference/ecs/update-service.html)
- list-tasks (http://docs.aws.amazon.com/cli/latest/reference/ecs/list-tasks.html)
- describe-tasks (http://docs.aws.amazon.com/cli/latest/reference/ecs/describe-tasks.html)

ですので、おなじみの `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_DEFAULT_REGION` といったパラメーターを、環境変数もしくは ecs-deploy コマンド実行時の引数で渡す必要があります. (もちろん –profile オプションも使えます)

ざっくりとまとめると、このスクリプトは以下のような順番で処理を進めていきます.

- パラメーターのバリデーション (AWS クレデンシャルとか、Docker イメージ名とか)

- 指定した Service が利用している Task Definition の中身 (json) を取得する
- json の image 定義を指定されたイメージ名で置き換える
- 新しい Task Definition を Amazon ECS に登録する
- 登録した Task Definition を実行時に指定された Service で利用するように更新する
- Service が新しい Task Definition を利用したコンテナを走らせているかをポーリングしながら待つ

最後までうまく進むと、新しい Docker イメージから走るコンテナが Service で走るところまで確認できます.

ここまで見てきてわかるとおり、上の方で面倒くさそうと書いた「4. Task Definition の中身と Service の設定を更新」を ecs-deploy コマンドが一手に引き受けてくれています.

非常にありがたい.

ecs-deploy の注意点

こんな便利そうな ecs-deploy スクリプトですが、いくつか利用する上でハマりそうなことがあります.

古いコンテナが生きているけど処理が完了になる

新しいコンテナが走っていることを ecs-deploy が検知すると “Service updated successfully, new task definition running.” というメッセージが出て処理が終了します. が、Amazon ECS はローリングデプロイを行うため、新しいコンテナが走った後もしばらくは古いコンテナが生きています. 更新されたはずなのにブラウザでアクセスしてみると古いコンテナでレンダリングされたデータが表示される、ということが起こりえます.

古いコンテナが終了するまでの時間は一概には言えないのですが、僕自身はブラウザをリロードして待つのは嫌なので ecs-deploy に手を加えて古いコンテナがいなくなるまでを監視対象とするようにして利用しています.

スクリプト内部で jq (https://stedolan.github.io/jq/) と awscli (https://aws.amazon.com/cli/) を利用している

先ほども出てきた awscli に加えて、jq も利用していますので、ecs-deploy を動かす CI サーバーなどに両者がインストールされている必要があります.

こういうのを見ると Go で再実装したくなりますね.

イメージ名の正規表現が Docker レジストリ v2 の仕様とちょっと違う

2015/12/22 現在、スクリプトの途中でイメージ名を正規表現でパースしている (`imageRegex` ってとこ)ところが、Docker の仕様で定められているもの (https://docs.docker.com/registry/spec/api/)とちょっと異なるようです.

v2 API の仕様的には、

- repository name has two or more path components, they must be separated by a forward slash (“/”)
- 各コンポーネントが `[a-z0-9]+(?:[._-][a-z0-9]+)*` にマッチする

とありますが、 ecs-deploy スクリプトはちょっと違う実装がされています.

具体的にはソースのこの部分 (https://github.com/toricls/ecs-deploy/blob/156b913fcf2bd8913fd9b548220b37d4dacbf430/ecs-deploy#L164)なんですが、

```
^([a-zA-Z0-9.\-]+):?([0-9]+)?/([a-zA-Z0-9\-]+)/?([a-zA-Z0-9
\-]+)?:?([a-zA-Z0-9\.\-]+)?$
```

という正規表現を利用して

```
domain:(port)/repository/img:tag
```

という形式にパースしようとしているようです.

また使える文字自体にも不足があり、 _ (アンダースコア)が入っているとパースしてくれない、という問題もあります.

僕自身はこの問題には特に引っかからなかったのですが、利用しているリポジトリ/イメージ名によっては多少のスクリプト修正が必要になるかもしれません.

(余談ですが、先日リリースされた Amazon ECR は v2 API の仕様に沿っているようで、domain/repos/repos/repos/img:tag みたいなリポジトリを定義できます)

まとめ

細かいハマリどころはありつつも、ecs-deploy スクリプト便利です.

ありがたいね〜という話でした.

Written by **toricls (https://toris.io/about/)**, opinions are my own.
AT DEC 23, 2015 / TAGGED WITH AMAZON ECS (HTTPS://TORIS.IO/TAGS/AMAZON-ECS/), DOCKER (HTTPS://TORIS.IO/TAGS/DOCKER/), GITLAB CI (HTTPS://TORIS.IO/TAGS/GITLAB-CI/)