Rails on Docker

銀座Rails#22 2020/06/12(Fri)

神速(@sinsoku)

自己紹介

- 名前: 神速
- 会社: メドピア株式会社
- GitHub: @sinsoku (画像右上)
- Twitter: @sinsoku_listy (画像右下)
- Rails歴: 約7年

みんな🍺🍶を飲みながら気楽に聞いて

Dockerとは何か？🐳

コンテナ仮想化🤔

Docker 25 users

出典: フリー百科事典『ウィキペディア（Wikipedia）』

Docker（ドッカー[2]）は、コンテナ仮想化を用いてアプリケーションを開発・配置・実行するためのオープンソースソフトウェアあるいはオープンプラットフォームである[3]。

Dockerはコンテナ仮想化を用いたOSレベルの仮想化（英語版）によりアプリケーションを開発・実行環境から隔離し、アプリケーションの素早い提供を可能にする。かつその環境自体をアプリケーションと同じようにコード（イメージ）として管理可能にする[4]。Dockerを開発・テスト・デプロイに用いることで「コードを書く」と「コードが製品として実行される」間の時間的ギャップを大きく短縮できる[5]。

目次 [非表示]

Docker

作者	Solomon Hykes
開発元	Docker, Inc.
初版	2013年3月13日
最新版	19.03.11 - 2020年6月1日（5日前）[1][±]

私もよく分かってない😇

過去のインフラ構築の課題🚧

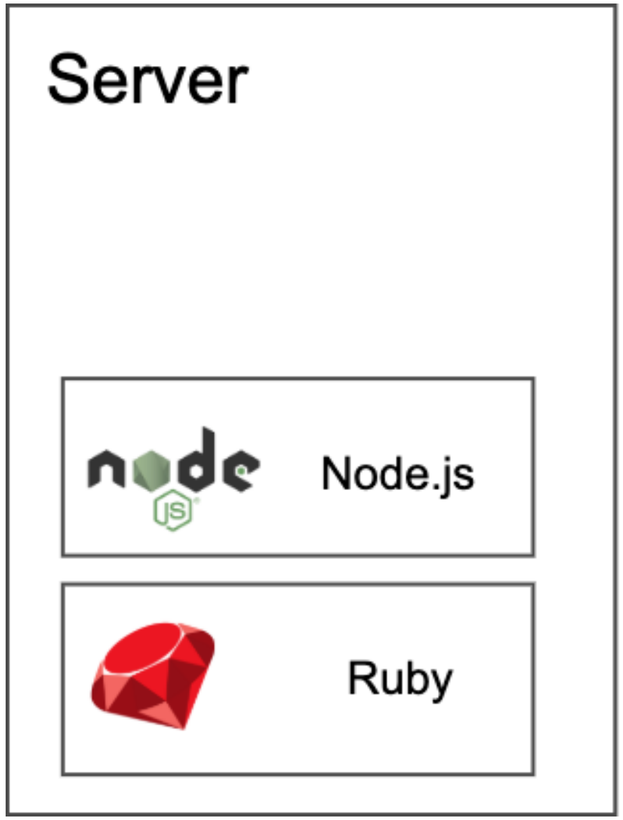
Dockerで何を解決できるか？🐳

Chef, Ansible期

- インフラの構築手順をコード化
  - Infrastructure as Code（IaC）
- バージョン管理できる
- レビューできる

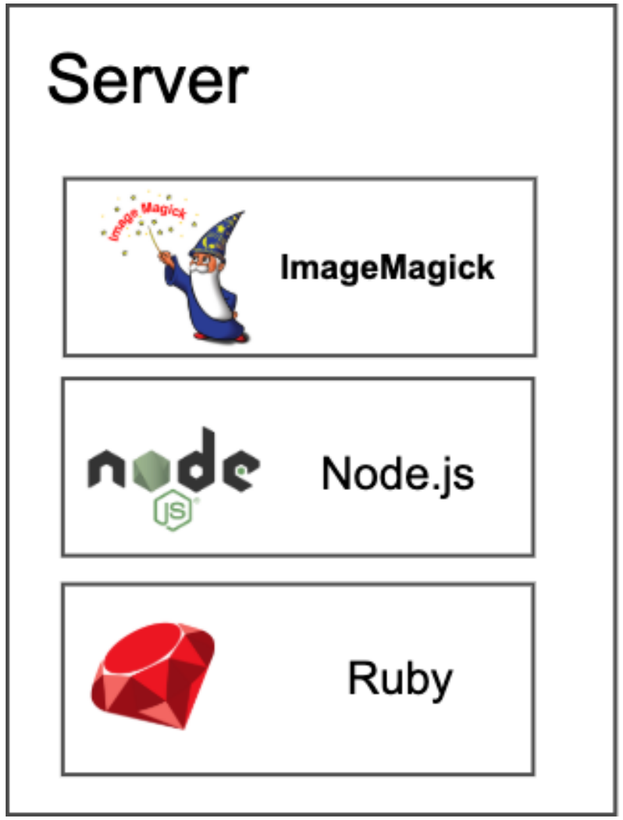
Ansibleの例（playbook.yml）

```
- hosts: all
  become: yes
  tasks:
  - apt:
      pkg:
        - ruby
        - nodejs
      update_cache: yes
```



Ansibleの例（playbook.yml）

```
- hosts: all
  become: yes
  tasks:
  - apt:
      pkg:
        - ruby
        - nodejs
        - imagemagick
      update_cache: yes
```



インフラは冪等になった🔧

神速
@sinsoku_listy

ansible で冪等を掴んだと思ったら、夢だった

午後2:30 · 2016年12月21日 · YoruFukurou

ツイートアクティビティを表示

3 いいねの数

神速
@sinsoku_listy

Ansible なのでべき等です！（常にべき等になるわけではない

午後2:03 · 2018年1月9日 · Twitter for Mac

ツイートアクティビティを表示

1 いいねの数

冪等を保つのはとても難しい👻

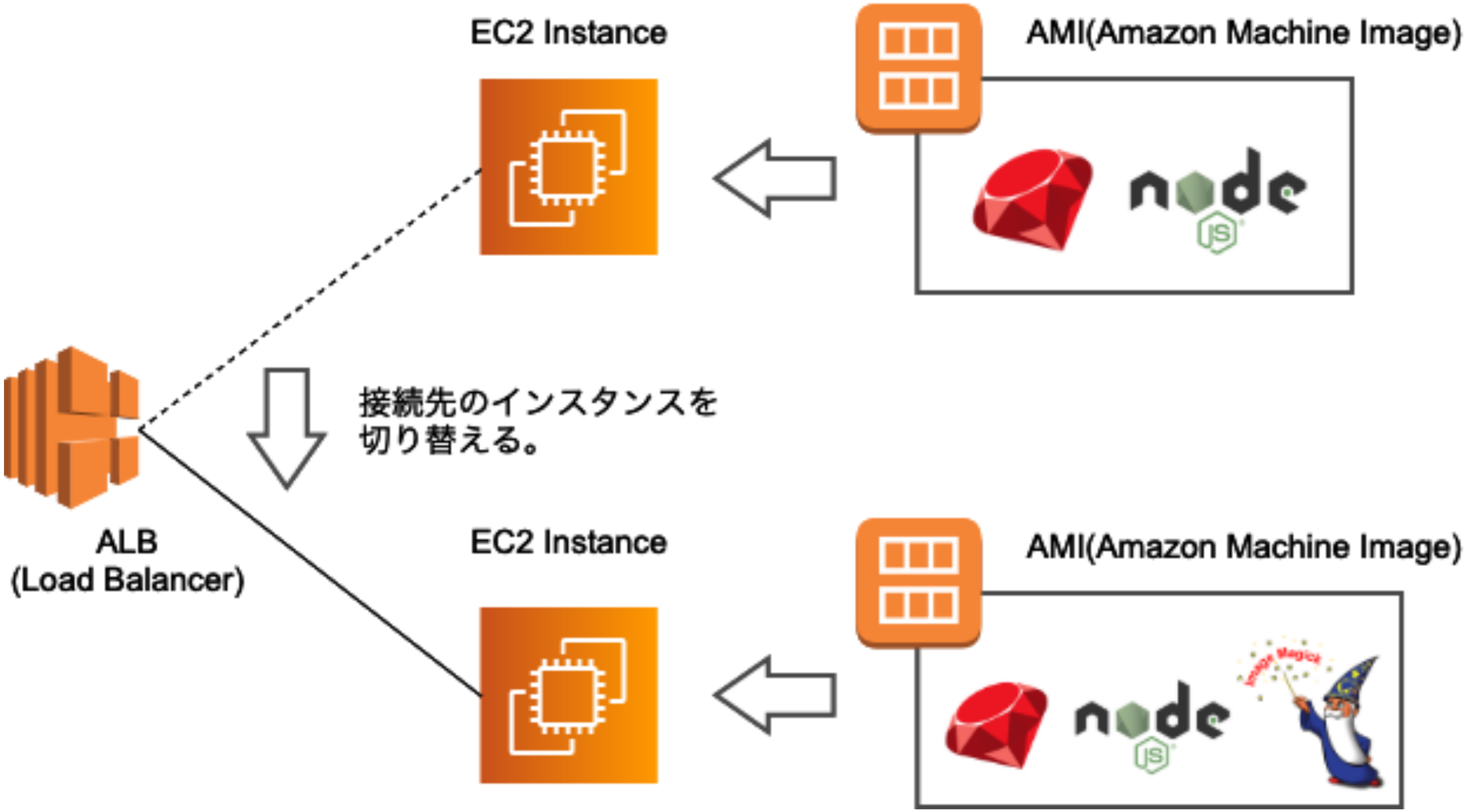
- 初回、2回目の両パターンの記述
- apt-getは冪等じゃない
  - リポジトリの状態で変わる

気づいたらstg/prdでパッチバージョンが違う🔥

Packer, AMI期

構成を変更するたびにマシンイメージを毎回ゼロから作成する。

- Immutable Infrastructure（不変のインフラ）
- Disposable Infrastructure（廃棄可能なインフラ）


EC2 Instance
AMI(Amazon Machine Image)
接続先のインスタンスを
切り替える。
ALB
(Load Balancer)
EC2 Instance
AMI(Amazon Machine Image)

多くの問題が解決した🎉

また新たな課題が出てきた...

マシンイメージの問題点

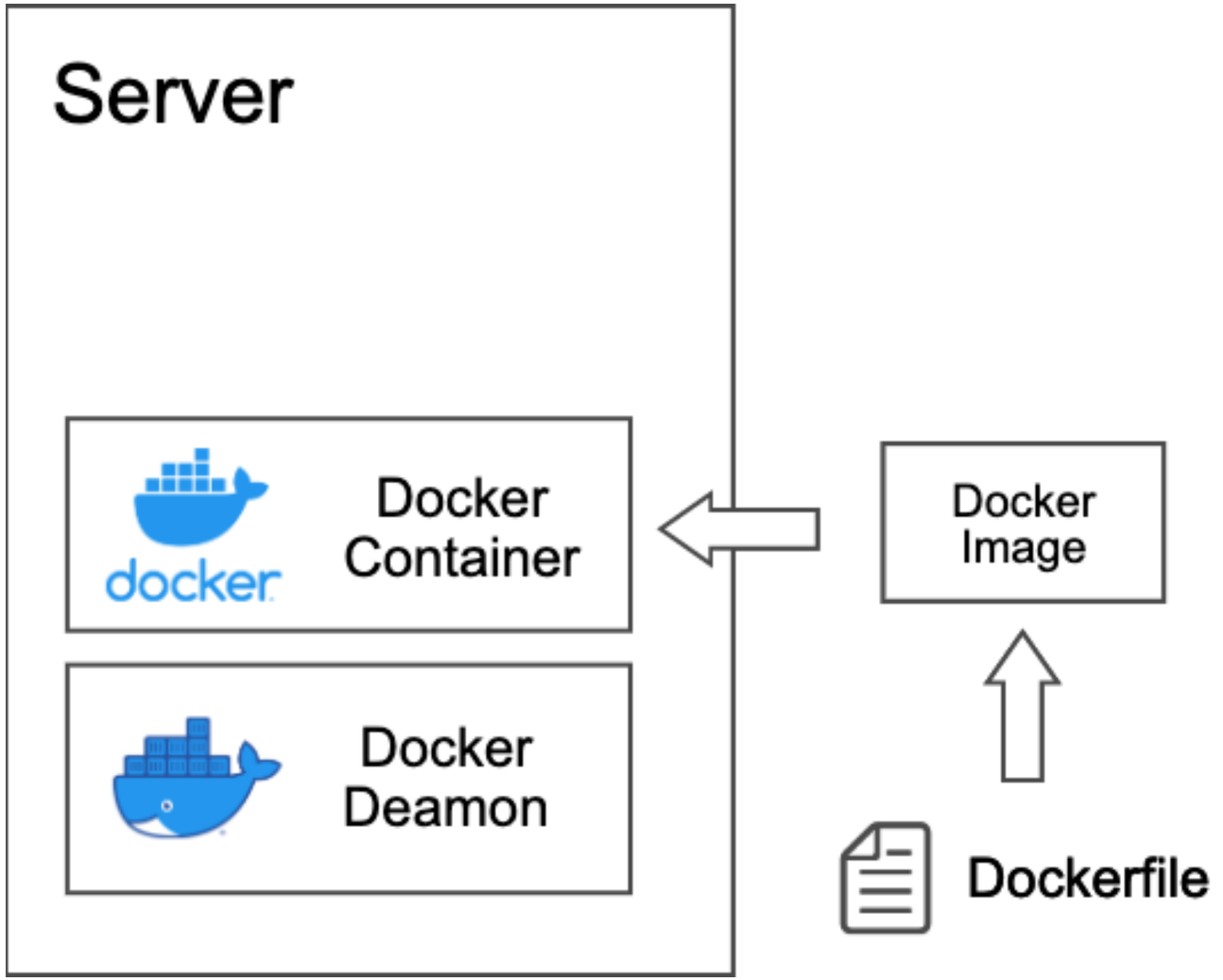
- ビルドに時間がかかる
- 容量が重く、起動が遅い
- ポータビリティが低い
  - 手元で動作確認できない
  - 他クラウドに移行しづらい

Docker期

- マシンイメージに比べて軽量
- CPU/メモリの使用効率が良い
- ポータビリティが高い
  - 本番と同じイメージが手元で動く

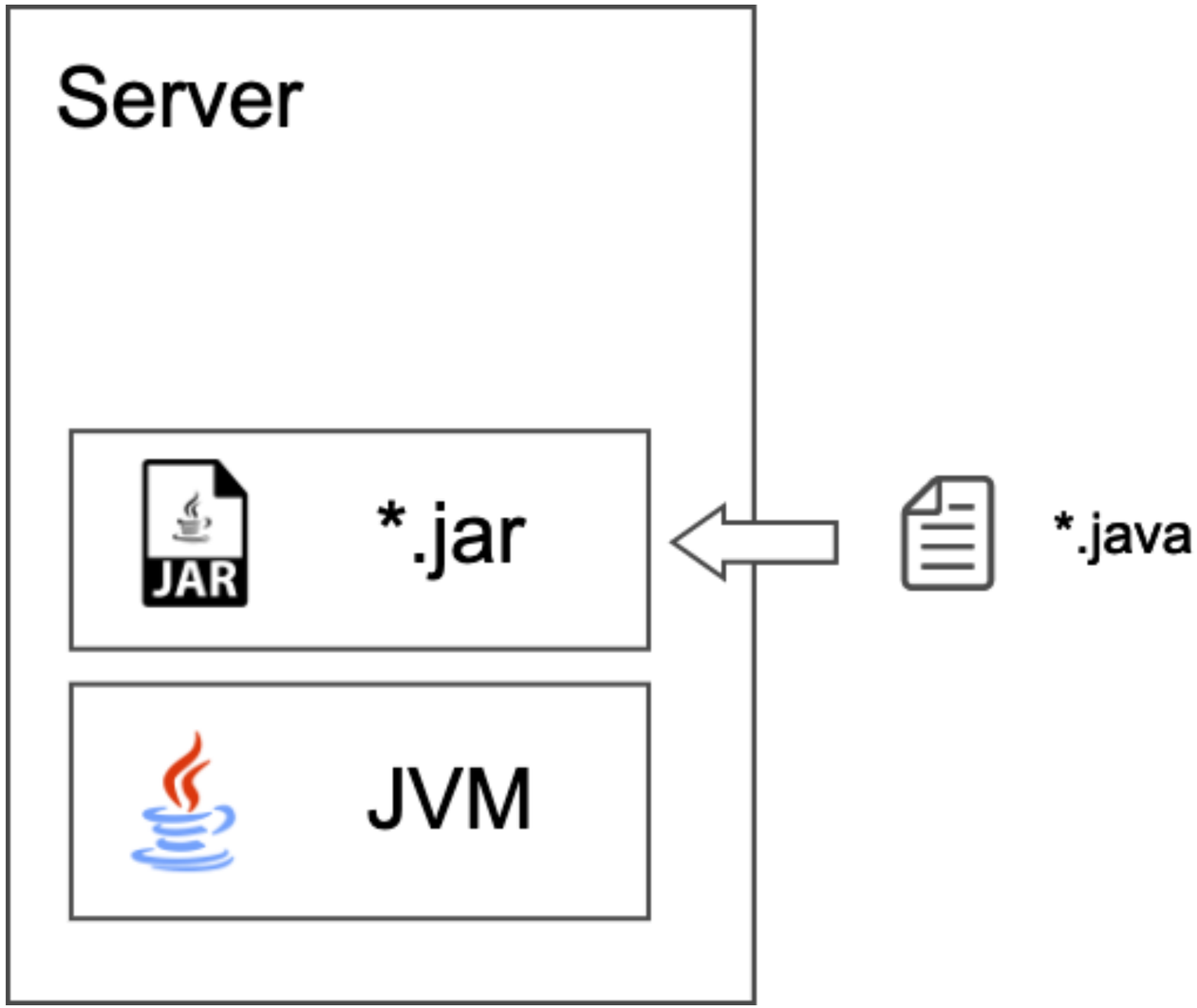
Dockerの概要

- Dockerfileからイメージを作る
- イメージからコンテナを起動する
- Dockerがある場所で動く
- **"Build once, run anywhere"**



Javaに似てる

- ソースコードからJARを作る
- JARをサーバにデプロイする
- JVMがある場所で動く
- **"Write once, run anywhere"**



🐳 実際にコンテナを動かす

例. シンプルなWEBrickサーバ

```
# app.rb
require "webrick"

srv = WEBrick::HTTPServer.new
srv.mount('/', WEBrick::HTTPServlet::FileHandler, 'index.html')
srv.start
```

Dockerfile

```
FROM ruby:2.7.1
WORKDIR /app

COPY index.html ./
COPY app.rb ./

EXPOSE 80
CMD ["ruby", "app.rb"]
```

ビルドと起動

```
$ ls
Dockerfile  app.rb      ginza.html  index.html
```

・ イメージのビルド

```
$ docker build -t plain_server .
```

・ コンテナの起動

```
$ docker run -p 8080:80 plain_server
```

```
$ curl localhost:8080
<!DOCTYPE html>
<html lang="ja">
<head>
  <title>Index</title>
</head>
<body>
  <h1>Hello, Docker</h1>
</body>
</html>
```

ボリュームマウント

```
$ docker run -p 8080:80 \
    -v `pwd`/ginza.html:/app/index.html \
    plain_server

$ curl localhost:8080
<!DOCTYPE html>
<html lang="ja">
<head>
  <title>Index</title>
</head>
<body>
  <h1>Hello, Ginza.rb</h1>
</body>
</html>
```

ここまでのまとめ

- docker build -t <name> . でビルドする
  - Dockerfileがイメージの設計図
- `$ docker run IMAGE` でコンテナを起動する
- コンテナの起動時にPortやVolumeを指定できる

少し休憩(10分くらいのはず）[1]

[1] https://medpeer.co.jp/recruit/

🐳でRailsを動かす方法

production用Railsの復習

DBを起動しておく。

```
$ brew services start postgresql
```

rails newからこんな感じ。

```
$ rails new foo --database postgresql && cd foo
$ RAILS_ENV=production BUNDLE_WITHOUT=development:test bundle install
$ RAILS_ENV=production bin/rails assets:precompile
$ RAILS_ENV=production bin/rails db:prepare
$ RAILS_ENV=production RAILS_SERVE_STATIC_FILES=1 bin/rails s
```

これを🐳で動くようにする

Dockerfile v1

```
FROM ruby:2.7.1
WORKDIR /app
ENV RAILS_ENV production
ENV BUNDLE_WITHOUT development:test

RUN curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | apt-key add - \
  && echo "deb https://dl.yarnpkg.com/debian/ stable main" | tee /etc/apt/sources.list.d/yarn.list

RUN apt-get update && apt-get install -y yarn
COPY . .
RUN bundle install
RUN SECRET_KEY_BASE=dummy bin/rails assets:precompile

EXPOSE 3000
CMD ["bin/rails", "server"]
```

$ cp .gitignore .dockerignoreも要る

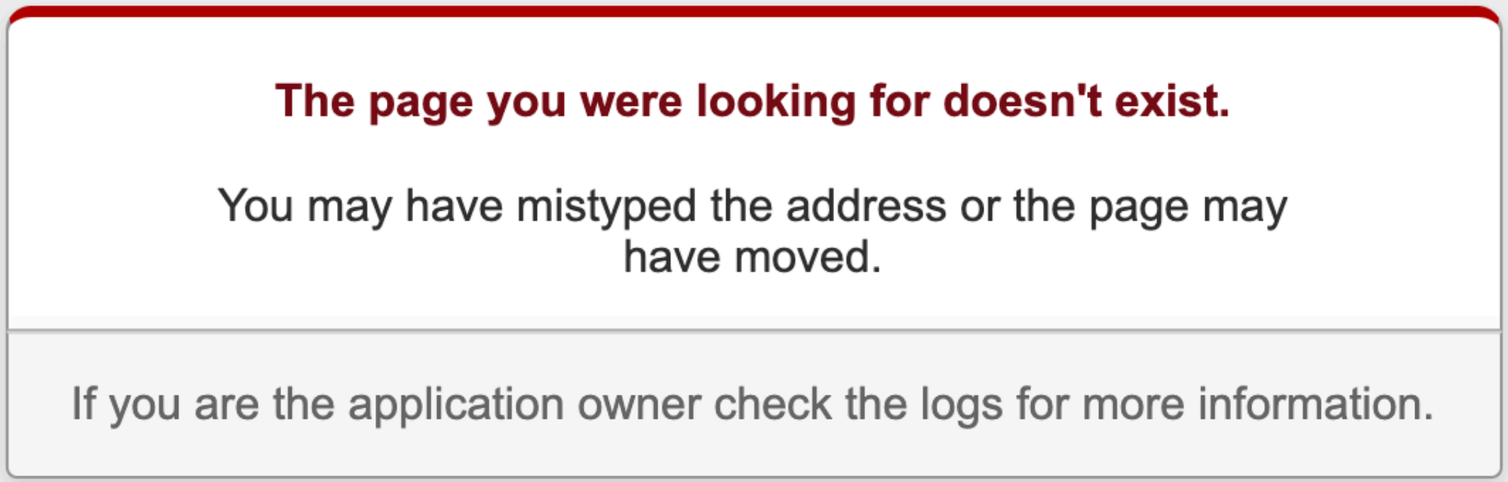
```yaml
# docker-compose.yml
version: '3.7'

services:
  db:
    image: postgres:11.6-alpine
    volumes:
      - db:/var/lib/postgresql/data
  web:
    build: .
    depends_on:
      - db
    ports:
      - "3000:3000"
    stdin_open: true
    tty: true
    environment:
      DATABASE_URL: 'postgres://postgres:@db'
      RAILS_LOG_TO_STDOUT: 1
      RAILS_SERVE_STATIC_FILES: 1
      RAILS_MASTER_KEY:

volumes:
  db:
```

dbとwebを起動

<key>はconfig/master.keyの中の値。

```
$ export RAILS_MASTER_KEY=<key>
$ docker-compose up -d db
$ docker-compose run web bin/rails db:prepare
$ docker-compose up web
```

localhost:3000でエラー画面が出れば成功



🐳のダイエット

レイヤーキャッシュ

```
FROM ruby:2.7.1
WORKDIR /app
ENV RAILS_ENV production
ENV BUNDLE_WITHOUT development:test

RUN curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | apt-key add - \
  && echo "deb https://dl.yarnpkg.com/debian/ stable main" | tee /etc/apt/sources.list.d/yarn.list

RUN apt-get update && apt-get install -y yarn
COPY . .
RUN bundle install
RUN SECRET_KEY_BASE=dummy bin/rails assets:precompile

EXPOSE 3000
CMD ["bin/rails", "server"]
```

Dockerfile v2

```
FROM ruby:2.7.1
WORKDIR /app
ENV RAILS_ENV production
ENV BUNDLE_WITHOUT development:test

RUN curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | apt-key add - \
  && echo "deb https://dl.yarnpkg.com/debian/ stable main" | tee /etc/apt/sources.list.d/yarn.list

RUN apt-get update && apt-get install -y yarn
COPY Gemfile .
COPY Gemfile.lock .
RUN bundle install
COPY package.json .
COPY yarn.lock .
RUN yarn install
COPY . .
RUN SECRET_KEY_BASE=dummy bin/rails assets:precompile

EXPOSE 3000
CMD ["bin/rails", "server"]
```

マルチステージビルド

```
FROM ruby:2.7.1 as builder

COPY Gemfile
COPY Gemfile.lock .
RUN bundle install

FROM ruby:2.7.1

COPY --from=builder /usr/local/bundle /usr/local/bundle

EXPOSE 3000
CMD ["bin/rails", "server"]
```

Dockerfile v3

```
FROM ruby:2.7.1 as base
WORKDIR /app
ENV RAILS_ENV production
ENV BUNDLE_WITHOUT development:test

FROM base as builder
RUN curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | apt-key add - \
  && echo "deb https://dl.yarnpkg.com/debian/ stable main" | tee /etc/apt/sources.list.d/yarn.list

RUN apt-get update && apt-get install -y yarn
COPY Gemfile .
COPY Gemfile.lock .
RUN bundle install
COPY package.json yarn.lock .
RUN yarn install
COPY . .
RUN SECRET_KEY_BASE=dummy bin/rails assets:precompile

# 続く
```

Dockerfile v3

```
FROM base

COPY . .
COPY --from=builder /usr/local/bundle /usr/local/bundle
COPY --from=builder /builder/public/assets ./public/assets
COPY --from=builder /builder/public/packs ./public/packs

EXPOSE 3000
CMD ["bin/rails", "server"]
```

サーバの起動時にNode.jsとnode_modulesは不要⭐

Alpine Linux

- 軽量なLinuxディストリビューション
- 最小限のものしか入ってない
- パッケージ管理はapk

Dockerfile v4

```
FROM ruby:2.7.1-alpine as base
WORKDIR /app
ENV RAILS_ENV production
ENV BUNDLE_WITHOUT development:test

FROM base as builder
RUN apk update && apk-add --update \
      build-base postgresql-dev yarn tzdata git

COPY Gemfile .
COPY Gemfile.lock .
RUN bundle install
COPY package.json yarn.lock .
RUN yarn install
COPY . .
RUN SECRET_KEY_BASE=dummy bin/rails assets:precompile

# 後半は同じなので省略
```

BuildKit[2]を使用する

Docker 18.09から本体に統合されたツール。

- ビルドを並列に行い高速化
- キャッシュや機密情報に関する機能強化
- DOCKER_BUILDKIT環境変数を設定する
  - `$ DOCKER_BUILDKIT=1 docker build -t foo .`

[2] https://docs.docker.com/develop/develop-images/build_enhancements/

本番🐳のまとめ

- ビルド高速化
  - レイヤーキャッシュ、BuildKit
- 軽量化
  - マルチステージビルド、Alpine Linux

開発環境で🐳を使う

開発環境で🐳を使うメリット

- 開発者の環境を統一できる
  - DBのバージョン違いのトラブルを避ける
- 複数のDBやサーバを簡単に起動できる
  - マイクロサービスな案件

本番🐳との違い

- インストールするgemが異なる
- npm, node_modulesが必要
  - eslint, jest, ...etc
- 開発用のツール（Chrome, curlなど）が必要

本番🐳を開発環境として使うのは厳しい

開発でCircleCIのイメージを使う

```
$ docker pull circleci/ruby:2.7.1-node-browsers
```

- Node.js, Yarnが入っている
- ブラウザが入っている
- jqが入っている
- CircleCIで落ちるテストを再現できる

```
# docker-compose.development.yml
services:
  web:
    image: circleci/ruby:2.7.1-node-browsers
    stdin_open: true
    tty: true
    volumes:
      - .:/app
    environment:
      BOOTSNAP_CACHE_DIR: '/usr/local/bundle'
      HISTFILE: '/app/log/.bash_history'
      EDITOR: vi
      DATABASE_URL: 'postgres://postgres:@db'
    depends_on:
      - db
    command: ["bin/rails", "server", "-b", "0.0.0.0"]
    expose: ["3000"]
    ports:
      - "3000:3000"
    working_dir: /app
    user: root
```

Docker for Mac遅い問題

- MacのVolueマウントが遅い
- `bundle install`が遅い
- `yarn install`が遅い
- `assets:precompile`が遅い

もう何もかも遅い🐢

```yaml
# docker-compose.development.yml
services:
  web:
    image: circleci/ruby:2.7.1-node-browsers
    stdin_open: true
    tty: true
    volumes:
      - ..:/app:cached
      - rails_cache:/app/tmp/cache
      - bundle:/usr/local/bundle
      - node_modules:/app/node_modules
      - packs:/app/public/packs
      - packs_test:/app/public/packs-test
    environment:
      BOOTSNAP_CACHE_DIR: '/usr/local/bundle'
      HISTFILE: '/app/log/.bash_history'
      EDITOR: vi
      DATABASE_URL: 'postgres://postgres:@db'
    depends_on:
      - db
    command: ["bin/rails", "server", "-b", "0.0.0.0"]
    expose: ["3000"]
    ports:
      - "3000:3000"
    working_dir: /app
    user: root
```

bibendi/dip[3]

DockerでRailsを開発するときに便利なツール。

- `docker-compose`のラッパー
- `dip.yml`にコマンドを記述する
  - `dip rails s`
  - `dip rubocop --parallel`

[3] https://github.com/bibendi/dip

```yaml
# dip.yml
version: '4.1'

interaction:
  rails:
    description: Run Rails commands
    service: web
    command: bundle exec rails
    subcommands:
      s:
        description: Run Rails server at http://localhost:3000
        service: web
        compose:
          run_options: [service-ports, use-aliases]

provision:
  - dip compose down --volumes
  - dip compose up -d db redis
  - dip bash -c ./bin/setup
```

本番🐳をローカルで動かす

dip.ymlでdocker-compose.ymlを複数指定できる。

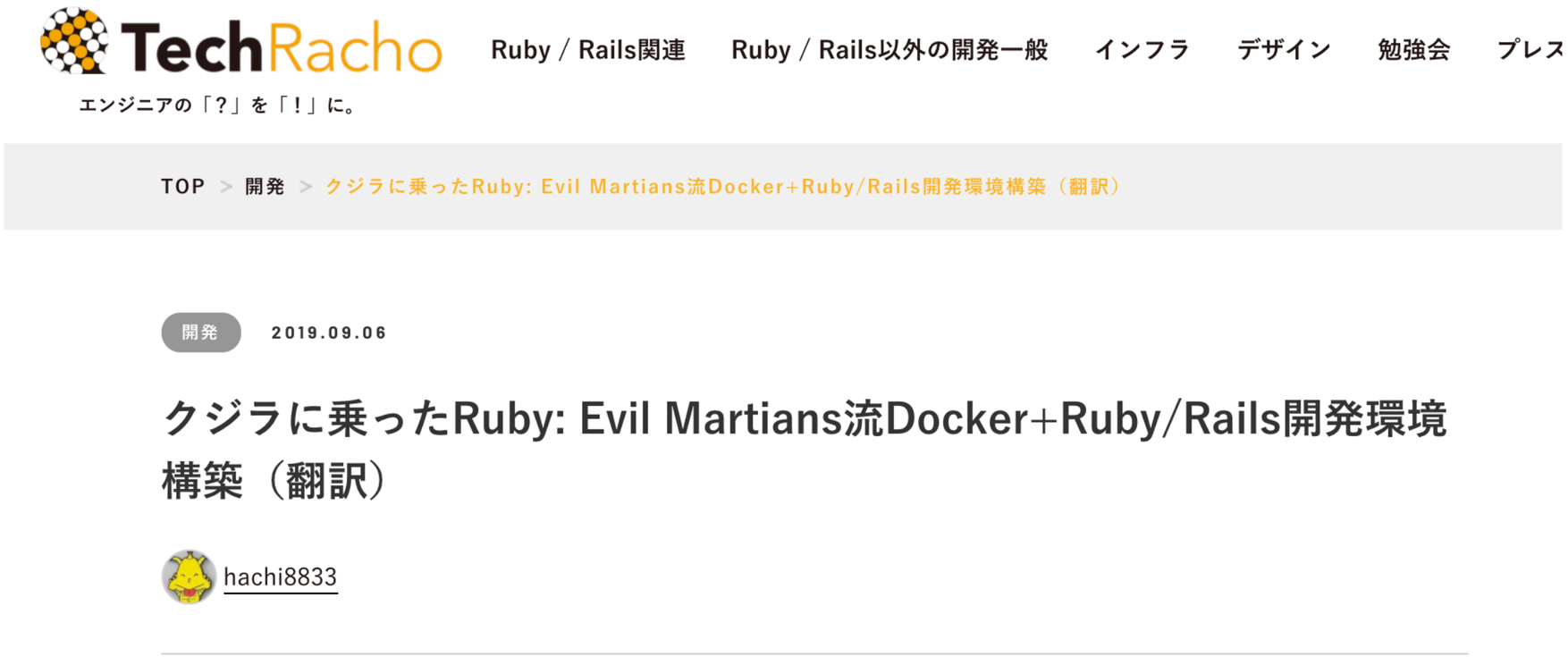
```
environment:
  COMPOSE_EXT: development

compose:
  files:
    - docker-compose.yml  # db, redis
    - docker-compose.$COMPOSE_EXT.yml # web, worker
```

```yaml
# docker-compose.staging.yml
services:
  web: &web
    build: .
    image: ginza-rails:staging
    stdin_open: true
    tty: true
    environment:
      BOOTSNAP_CACHE_DIR: '/usr/local/bundle'
      HISTFILE: '/app/log/.bash_history'
      EDITOR: vi
      RAILS_ENV: staging
      DATABASE_URL: 'postgres://postgres:@db'
      REDIS_URL: 'redis://redis'
      RAILS_LOG_TO_STDOUT: 1
      RAILS_SERVE_STATIC_FILES: 1
    depends_on:
      - db
      - redis
    ports:
      - "3000:3000"

  sidekiq:
    <<: *web
    command: ['bundle', 'exec', 'sidekiq']
    expose: []
    ports: []
```

本番🐳をローカルで動かす

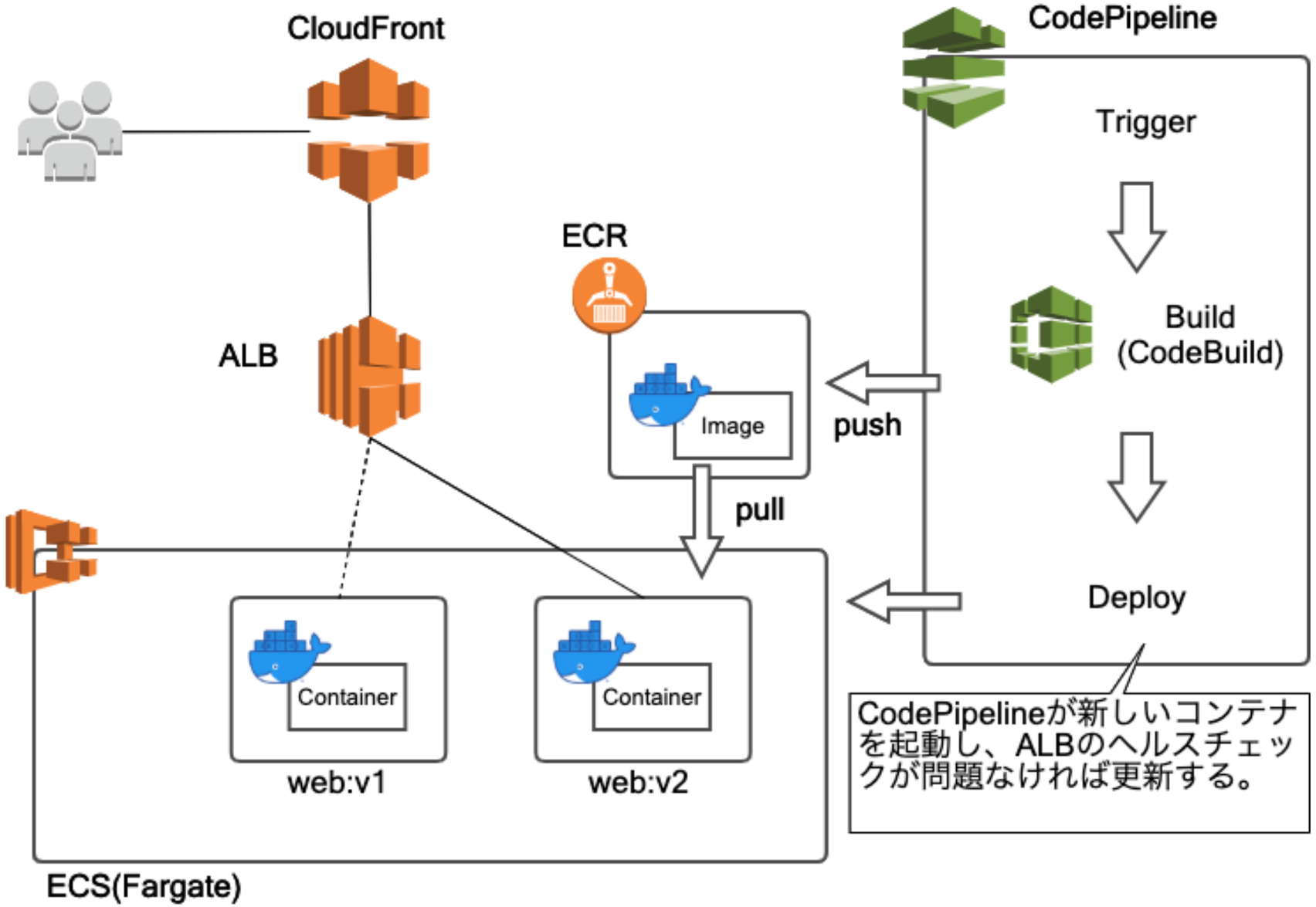
```
$ COMPOSE_EXT=staging dip compose build
$ COMPOSE_EXT=staging dip rails db:prepare
$ COMPOSE_EXT=staging dip rails s
```

Evil Martians流Docker+Ruby/Rails開発環境構築（翻訳）[4]



[4] https://techracho.bpsinc.jp/hachi8833/2019_09_06/79035

実際のAWSでの運用


CloudFront
CodePipeline
Trigger
ECR
Build
(CodeBuild)
ALB
Image
push
pull
Deploy
Container
Container
web:v1
web:v2
CodePipelineが新しいコンテナを起動し、ALBのヘルスチェックが問題なければ更新する。
ECS(Fargate)

✓ assetsファイルを置くためのs3バケットを用意

デプロイ時にユーザーがアクセスすると、以下のフローで
assetsファイルが404エラーになる可能性がある。

```
旧新のコンテナが起動中にアクセスが来る
-> LBで新appがヒットし、新js/cssを含むhtmlを返す
-> CloudFrontにreqきて、packsなのでALBに流れる
-> LBで不幸にも旧appがヒットし、js/cssが返せない
```

この問題を解決するため、assetsファイルをS3に置いて、
CloudFrontから配信する方針に変更します。

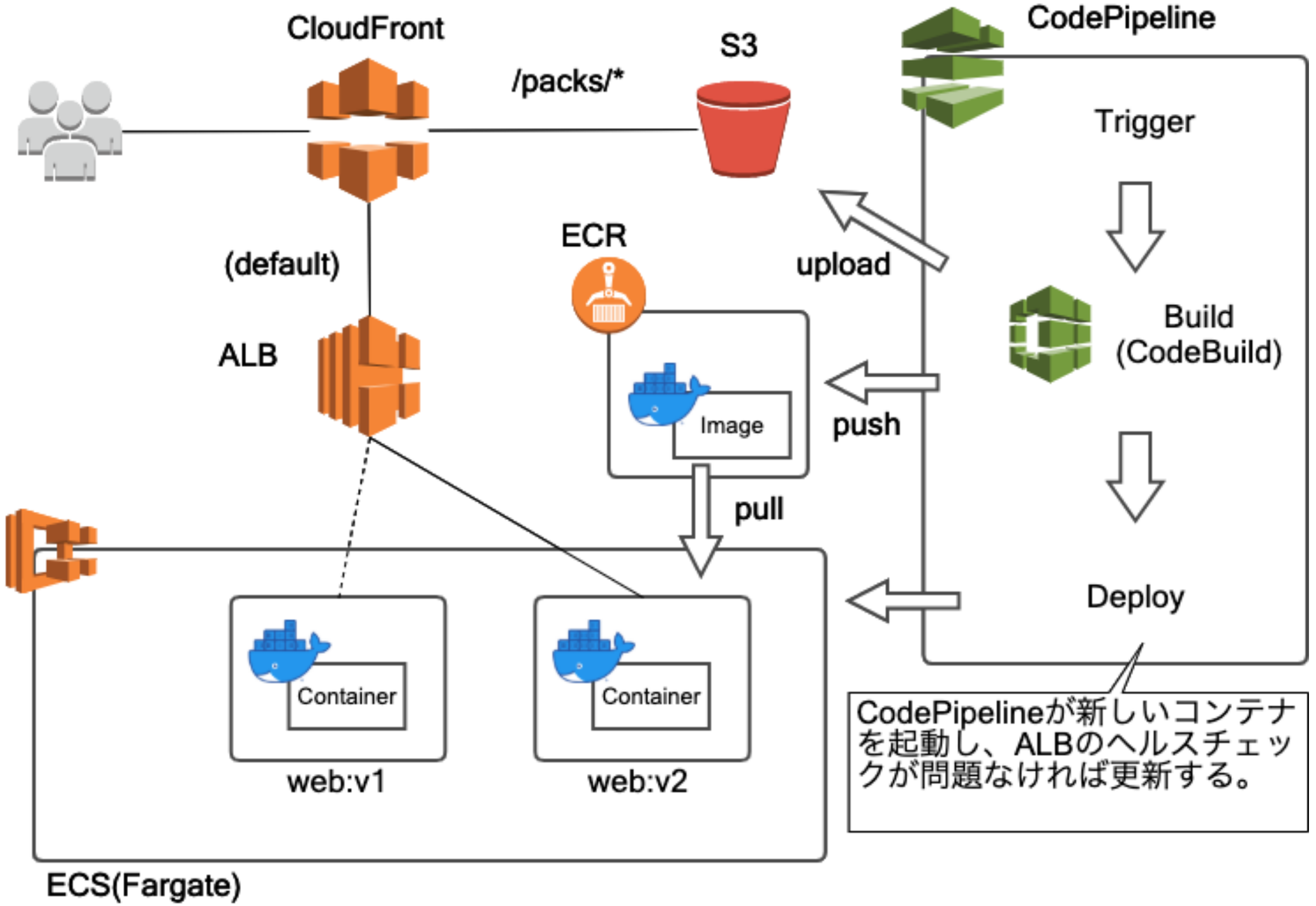
まずはassetsを置くところの動作確認をしたいので、
このコミットではS3バケットだけを追加します。

参考URL

ECSのデプロイ時に一定確率で静的ファイルが404になる問題を回避する
https://qiita.com/hareku/items/6be1b71e58033b9739fd

buildspec.ymlから抜粋

```
# Upload assets to s3
- CONTAINER_ID=$(docker create $REPOSITORY_URL:$IMAGE_TAG)
- docker cp $CONTAINER_ID:/app/public/packs public/packs
- aws s3 sync public/packs s3://${s3_bucket}/packs
- docker rm $CONTAINER_ID
```


CloudFront
/packs/*
S3
CodePipeline
Trigger
(default)
ECR
upload
ALB
Build
(CodeBuild)
Image
push
pull
Deploy
Container
Container
web:v1
web:v2
ECS(Fargate)
CodePipelineが新しいコンテナを起動し、ALBのヘルスチェックが問題なければ更新する。

更なるビルドの高速化

更なるビルドの高速化（？）

1. Cache Mount
2. `assets:precompile`の省略

Cache Mount

イメージのビルド時にボリュームマウントを使う機能。

experimentalなので、Dockerfileの1行目に以下を記述する。

```
# syntax=docker/dockerfile:experimental
```

Cache Mount

以下のように書くことで、Gemfile.lockを更新しても2回目のビルドが高速になる。

```
RUN --mount=type=cache,target=/usr/local/bundle \
      bundle install
```

ローカルで劇的な効果（詳細は後述）

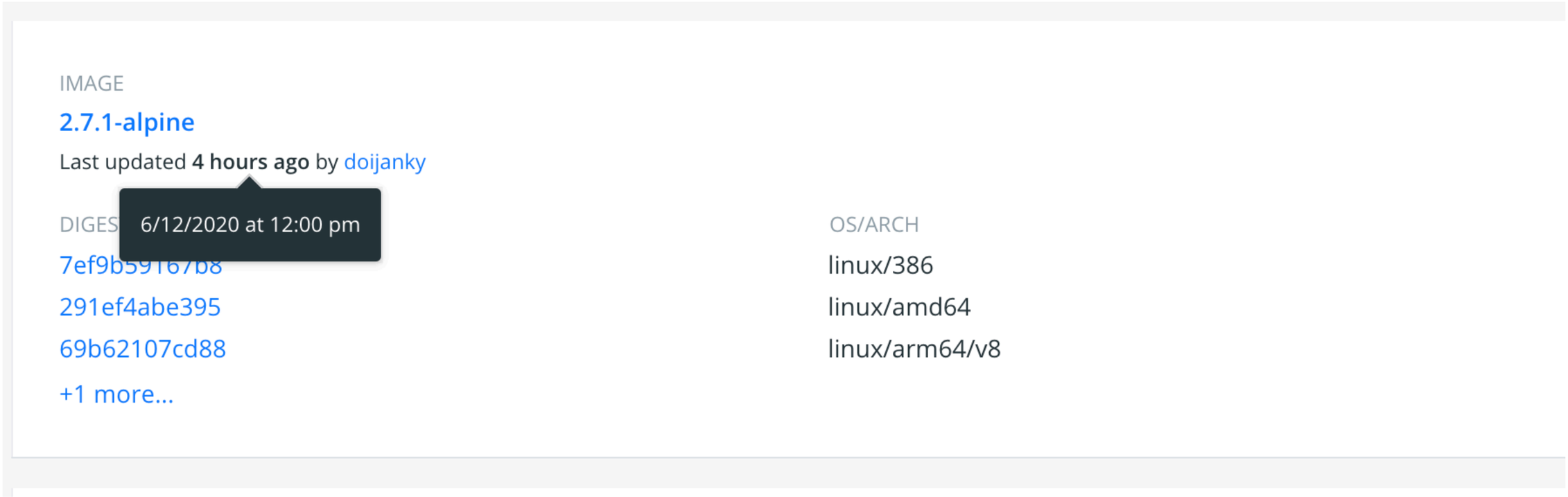
Dockerfileでassets:precompileしない

```
$ docker build -t <IMAGE> .
$ docker run -v `pwd`/public/assets:/app/public/assets \
    -v `pwd`/public/packs:/app/public/packs \
    <IMAGE> bin/rails assets:precompile
$ aws s3 sync public/assets s3://<bucket>/assets
$ aws s3 sync public/packs s3://<bucket>/packs
```

どちらも課題が残る

1. Cache Mount
   - CodeBuildはDockerボリュームのキャッシュが効かない
   - ベースイメージの更新でキャッシュが使えなくなる
2. `assets:precompile`の省略
   - manifest.jsonが無いとViewのレンダリングでエラー

本日昼、stagingビルドが死にました😇



「FROM ruby:2.7.1-alpine」って書いてました。

🐳との戦いはまだ続く...

TO BE CONTINUED...

ご静聴ありがとうございました

We are hiring[6]

MedPeer

今しかできない挑戦がここにある

[6] https://medpeer.co.jp/recruit/