1 fargate-cli-handson

Fargate service/task のデプロイを CLI で実施します。

文頭に (見解) と書いてあるものは筆者の知識に基づく見解であり、事実誤認等が含まれる可能性があります。鵜呑みにしないでください。

1.1 概要

Fargate タイプの ECS クラスタと、稼働するネットワーク (VPC, Subnet, RouteTable, Gateway等) の構築、 フェイシングするロードバランサ (ALB) が構成済みであることを前提として、以下の手順を踏むことで初回デプロイが完了する。

1. 開発環境でアプリケーションのイメージをビルド
2. ECR にビルドしたイメージを push
3. タスク定義を登録
4. Service を作成

また、初回以降でアプリケーションの変更を伴うデプロイを行う手順は次の通り。

1. 開発環境でアプリケーションのイメージをビルド
2. ECR にビルドしたイメージを push
3. タスク定義の新しいリビジョンを登録
4. Service を更新

1.2 事前条件／前提条件

Day 1 ハンズオンの handson/boyacky.yaml スタックが実行済みであること

ここで実行するコマンドは、特に断りのない限り Cloud9 Console の上で実行するものとします。

また、追加作業として以下の内容を実施します。これらは作業が必要なタイミングで随時指示します。

- Cloud9 を新規に立ち上げる
  - Default VPC でもOK

- ECR レポジトリ “boyacky/web-app” の作成 (Day1 4.1 Amazon ECR にリポジトリを作成)
- ECS クラスタ “boyacky-cluster” を Fargate 起動タイプで作成 (Day1 6.1 ECS クラスターの作成)
- ALB ターゲットグループの作成
  - ターゲットの作成は不要
- ALB リスナーの作成
- IAM サービスが利用するタスクロールの作成 (Day1 7.1 タスク用のIAMロール作成)
  - DynamoDB full-access
- IAM ワンショットタスクが利用するタスクロールの作成
  - DynamoDB full-access
  - Comprehend full-access
  - S3 full-access
- S3 バケットの作成 (osenchi batch output)

1.3 事前作業

次の環境変数をセットします。

```
# Cloud9 console
export AWS_ACCOUNT=$(curl -s http://169.254.169.254/latest/dynamic/instance-identity/dc
export AWS_REGION=$(curl -s http://169.254.169.254/latest/dynamic/instance-identity/doc
```

ECR レポジトリの名前をシェル変数にセット

```
export DOCKER_IMAGE_NAME=boyacky/web-app
export DOCKER_REMOTE_REPOSITORY=${AWS_ACCOUNT}.dkr.ecr.${AWS_REGION}.amazonaws.com/${DC
```

1.3.1 Cloud9 環境の起動

作業するリージョンで Cloud9 環境を起動します。

- 名前: 任意
- Environment type: EC2
- Instance type: t2.micro
- Platform: Amazon Linux 2

起動が完了したら、この環境にログインします。以降の作業はこの環境のコンソール上で実施します。

コンソール上で以下のレポジトリを clone してください。

```
git clone https://github.com/hassaku63/boyacky.git
# or
git clone git@github.com:hassaku63/boyacky.git
```

1.3.2 ALB の設定

サービス(コンテナ)に向けるためのターゲットグループ、およびリスナーを設定します。

CloudFormation によって “boyacky-alb” という名前の ALB が作成されています。ここに ECS (awsvpc ネットワークモード) のサービスをルーティングするための設定を入れていきます。

1.3.2.1 ターゲットグループの作成

EC2 - Load Balancing - Target Groups の画面に移動し、ターゲットグループを作成します。

- Target Type: IP Address
- Target group name: tg-boyacky-app
- Protocol/Port: HTTP, 8080
- VPC: スタックで作成したVPCを選択 (boyacky-vpc)

- Health checks: デフォルト値

ターゲット登録の画面は編集不要

1.3.2.2 リスナーの登録

EC2 - Load Balancing - Load balancers の画面に移動し、 “boyacky-alb” のリスナーを登録します。

- Protocol/Port: HTTP, 80
- Action
  - Forward to: tg-boyacky-app (wight=1)

1.4 ECR レポジトリの作成

`ecr create-repository` コマンドで Docker レポジトリを作成します。

```
aws ecr create-repository --repository-name boyacky/web-app
```

成功した場合は次のような json が戻ります。

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:<region>:<aws-account>:repository/boyacky/web-app
        "registryId": "<aws-account>",
        "repositoryName": "boyacky/web-app",
        "repositoryUri": "<aws-account>.dkr.ecr.<region>.amazonaws.com/boyacky/web-app"
        "createdAt": 1000000000.0,
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": false
        }
    }
}
```

1.5 アプリケーションの最新版を ECR に push

Docker レジストリ (ECR) にログインします。

```
aws ecr get-login-password --region ${AWS_REGION} | docker login --username AWS --passw
```

アプリケーションのソースディレクトリに移動した後、イメージをビルドします。ここでは、ビルドを実行した時点での日時をタグとして付与することにします。

```
export IMAGE_TAG=$(date +%Y%m%d-%H%M%S)
docker build -t ${DOCKER_IMAGE_NAME}:${IMAGE_TAG} .
```

ECR にイメージを push します。

```
docker tag ${DOCKER_IMAGE_NAME}:${IMAGE_TAG} ${DOCKER_REMOTE_REPOSITORY}:${IMAGE_TAG}
docker push ${DOCKER_REMOTE_REPOSITORY}:${IMAGE_TAG}
```

1.6 ECS クラスターの作成

ECS クラスターを Fargate 起動タイプで作成します。

```
aws ecs create-cluster \
  --cluster-name boyacky-cluster \
  --capacity-providers FARGATE
```

正しく実行できれば、以下のような json が戻ります。

```
{
    "cluster": {
        "clusterArn": "arn:aws:ecs:<region>:xxxxxxxxxxxx:cluster/boyacky-cluster",
        "clusterName": "boyacky-cluster",
        "status": "PROVISIONING",
        "registeredContainerInstancesCount": 0,
        "runningTasksCount": 0,
        "pendingTasksCount": 0,
        "activeServicesCount": 0,
        "statistics": [],
        "tags": [],
        "settings": [
            {
                "name": "containerInsights",
                "value": "disabled"
            }
        ],
        "capacityProviders": [
            "FARGATE"
        ],
        "defaultCapacityProviderStrategy": [],
        "attachmentsStatus": "UPDATE_IN_PROGRESS"
    }
}
```


1.7 ECS 用の IAM ロールを作成

コンテナインスタンスが使用するロールと、コンテナ自身が使用するロールの2種類が必要です。

いずれもマネジメントコンソールから作成します。ARN を後で利用するので、手元に控えてください。

1.7.1 タスク実行ロール

IAM ロールに `ecsTaskExecutionRole` という名前のロールがあれば、作業不要

※ なければ Amazon ECS task execution IAM role の記述通りに作成

1.7.2 タスクロール

- Choose a use case: Elastic Container Service
- Select your use case: Elastic Container Service Task (Allows ECS tasks to call AWS services on your behalf.)
- Attached permissions policies
    - 以下のポリシーを追加
        - AmazonDynamoDBFullAccess
        - AmazonS3FullAccess
        - ComprehendFullAccess
- Role Name: 任意 (例: boyacky-ecs-task-role)

1.8 タスク定義の登録

AWS CLI `ecs register-task-definition` を使って、タスク定義を登録します。

実際に使用する json とパラメータ仕様を突合しながら、何をしているのか確認してみましょう。

- register-task-definition パラメータ仕様 … params-resgister-task-definition.md
- 入力パラメータとして使用する json の雛形 … cli-input-templates/task-definition-input.json

1.8.1 タスク定義の登録 (CLI)

引数の数が多く、かつ引数のいくつか（特に必須の `--container-defeinitions` ）ため、 `--cli-input-json` を用いて実行します。

`register-task-definition` の入力とするJSON を作成します。

cli-input-templates/task-definition-input.json に雛形となるファイルがあります。 この json ファイルにはプレースホルダとなる変数 (`${ECS_TASK_ROLE_ARN}}` など) が含まれており、 `envsubst` コマンドを利用することでけれらのプレースホルダに対応する環境変数の値を適用できます。各自の AWS 環境に応じて値をセットし、以下のコマンドを実行しましょう。

環境変数のセット

```
#
# 確認用。これまでの作業で既にセット済みの変数を確認します
#
echo $AWS_ACCOUNT
echo $AWS_REGION
echo $DOCKER_IMAGE_NAME
echo $DOCKER_REMOTE_REPOSITORY # 次の値と等価 ${AWS_ACCOUNT}.dkr.ecr.${AWS_REGION}.amazor
echo $IMAGE_TAG

#
# 各自の環境に合わせて、セットしてください
#
export ECS_TASK_ROLE_ARN=  # タスクロールの ARN
export ECS_TASK_EXEXCUTION_ROLE_ARN=  # タスク実行ロール（コンテナインスタンスが利用する方のロー
```

`--cli-input-json` の入力に使用する json を生成します。カレントディレクトリを `handson/` に移動して、以降の作業を実施してください。

```
# envsubst を使用して、プレースホルダに環境変数を埋め込んで出力
envsubst < cli-input-templates/task-definition-input.json > cli-inputs/boyacky-taskdef.
```

`boyacky-taskdef.json` を開いて、プレースホルダ (`${xxx}` 形式) が残っていないことを確認してください。

json に問題がなければ、 `ecs register-task-definition` でタスク定義を登録します。

```
aws ecs register-task-definition --cli-input-json file://cli-inputs/boyacky-taskdef.jsc
```

正常に Task difinition が作成された場合の出力例は cli-output-samples/register-task-definition.json のようになります。

出力の json から、 `taskDefinition.taskDefinitionArn` の値を確認し、 `TASKDEF_ARN` 環境変数にセットします。

```
# TODO: CLI の戻り値の、taskDefinition.taskDefinitionArn をコピペ
# {
#     "taskDefinition": {
#         "taskDefinitionArn": "arn:aws:ecs:<your-region>:<your-aws-account>:task-defir
#         "containerDefinitions": [
#             {
#                 // ...
export TASKDEF_ARN=<your-taskdef-arn>
```

ここまでの終了時点で、以下の環境変数がセットされていることを確認してください。

```
cat <<EOF | column -t
echo AWS_ACCOUNT = ${AWS_ACCOUNT}
echo AWS_REGION = ${AWS_REGION}
echo DOCKER_IMAGE_NAME = ${DOCKER_IMAGE_NAME}
echo DOCKER_REMOTE_REPOSITORY = ${DOCKER_REMOTE_REPOSITORY}
echo IMAGE_TAG = ${IMAGE_TAG}
echo ECS_TASK_ROLE_ARN = ${ECS_TASK_ROLE_ARN}
echo ECS_TASK_EXEXCUTION_ROLE_ARN = ${ECS_TASK_EXEXCUTION_ROLE_ARN}
echo TASKDEF_ARN = ${TASKDEF_ARN}
EOF
```

また、マネジメントコンソールで “boyacky” というタスク定義が作成されていることを確認してみましょう。

1.8.1.1 Column: CLI で JSON 形式の入力を使用する

雛形となる JSON の生成を `--generate-cli-skeleton` で行えます。AWS CLI の共通オプションとしてサポートされています。

利用するサブコマンドの引数が多い場合や、今回のように json 形式の引数が含まれる場合に検討してみましょう。

Note: AWS CLI v2 では yaml 形式もサポート されました。

```
# ecs register-task-definition の JSON 入力のスケルトンを生成
aws ecs register-task-definition --generate-cli-skeleton input
```

かなり多くのキーを持った JSON が生成されますが、最低限必要なものに絞ればそれほどのボリュームはありません。 CLI ドキュメントのサンプル JSON を引用します。

```
// sample input json file of register-task-definition
{
    "containerDefinitions": [
        {
            "name": "sleep",
            "image": "busybox",
            "cpu": 10,
            "command": [
                "sleep",
                "360"
            ],
            "memory": 10,
            "essential": true
        }
    ],
    "family": "sleep360"
}
```

実際に Fargate タイプでサービスを稼働させる場合、上記のパラメータでは不十分ですが、足りない／不正なパラメータは CLI がバリデーションしてくれるので、試しながら覚えていきましょう。公式ドキュメントにも タスク定義のサンプル が公開されています。参考にしてください。

1.9 サービスの作成 (デプロイ)

タスク定義 “boyacky” をもとに、サービスを作成します。

サービスの作成は `ecs create-service`, 更新は `ecs update-service` を利用します。

入力として使用する json ファイル各パラメータの意味を確認していきましょう。

- create-service パラメータ仕様 … params-create-service.md
- 使用する json の雛形 … cli-input-templates/create-service-input.json

1.9.1 サービス作成 (CLI) の実行

まずは、開始条件を満たしていることを確認しましょう。

- “boyacky” という名前でタスク定義が作成されていること
- 以下の環境変数がセットされていること

```
# 環境変数のチェック
cat <<EOF | column -t
AWS_ACCOUNT = ${AWS_ACCOUNT}
AWS_REGION = ${AWS_REGION}
DOCKER_IMAGE_NAME = ${DOCKER_IMAGE_NAME}
DOCKER_REMOTE_REPOSITORY = ${DOCKER_REMOTE_REPOSITORY}
IMAGE_TAG = ${IMAGE_TAG}
ECS_TASK_ROLE_ARN = ${ECS_TASK_ROLE_ARN}
ECS_TASK_EXEXCUTION_ROLE_ARN = ${ECS_TASK_EXEXCUTION_ROLE_ARN}
TASKDEF_ARN = ${TASKDEF_ARN}
EOF
```

次に、サービス作成に必要な追加パラメータをセットします。 ターゲットグループ、サブネット、セキュリティグループの情報を変数にセットします。次の変数を使ってください。

```
export TARGET_GROUP_ARN=xxx  # ALB ターゲットグループのARN
export SUBNET_1=subnet-xxx  # デプロイ先のプライベートサブネット(1)
export SUBNET_2=subnet-xxx  # デプロイ先のプライベートサブネット(2)
export TASK_SECURITY_GROUP=sg-xxx  # タスクにアタッチするSG ("boyacky-web-sg" で作成されてる
```

json 入力のテンプレートファイルから、実際の入力ファイルを作成します。

```
envsubst < cli-input-templates/create-service-input.json > cli-inputs/create-service-ir
```

生成された *cli-inputs/create-service-input.json* に、ターゲットグループ／サブネット／セキュリティグループの情報がセットされていることを確認してください。

問題なければ `ecs create-service` を実行しましょう。

```
aws ecs create-service --cli-input-json file://cli-inputs/create-service-input.json | t
```

サービス作成のリクエストが正常に受け付けられれば、cli-output-samples/create-service-output.json のような形式の json が戻ります。

1.10 サービスの更新

ユーザーが増えたのでアプリケーションコンテナをスケールアウトすることにしました。サービスの更新によってこの対応を行います。

スケールアウトはコンテナイメージの変更を伴わない変更であるため、 `update-service` コマンドの実行で完結します。

Doc: AWS CLI - aws.ecs.update-service

基本的に引数は `create-service` で使えるものと共通していますので、詳細は省きます。ここで実行するコマンドは以下の通りです。

```
aws ecs update-service \
  --cluster boyacky \
  --service boyacky-webapp-svc \
  --desired-count 2
```

`cluster`, `service` が必須引数となります。リクエストが正常に受け付けられた場合の出力例は cli-output-samples/update-service-output.json のようになります。

1.11 ワンショットタスクの実行

一回きりの処理（バッチなど）を実行する場合は、サービスからではなく、タスクを直接開始します。

タスク実行に関わるサブコマンド (API) は `start-task`, `run-task` の2つがありますが、今回利用するのは `ecs run-task` です。

subcommand	description
ecs run-task	タスクを作成し、実行する
ecs start-task	run-task と基本的には同じもの。タスクをどのコンテナインスタンスの上で実行するか、明示的に指定可能

また、ECS Run Task API は結果整合です。状態の変更がただちに AWS 側のリソースに反映されるとは限らないことに留意してください。Run Task の後続処理として何某かの処理を挟む必要がある場合は、Run Task 実行後に describe-tasks を繰り返し実行し、タスクの状態を確認します (exponential backoff で、間隔の最大値は5分ほど)。 desctibe-tasks が意図した応答を返した場合でも、後続処理の前に待機時間を挟んでおくとよいでしょう。

run-task のパラメータ仕様は params-run-task.md, 入力となる json の雛形は cli-input-templates/run-task-input.json です。

1.11.1 タスクの実行 (CLI)

感情分析サービスの “osenchi” をバッチ実行するタスクを実装しました(osenchi.py)。このバッチは、利用者のぼやきを取得し、 Comprehend で感情分析を行い、結果を S3 に出力します。このバッチを ECS のタスクで実行してみましょう。

タスク実行のために必要な環境変数をセットします。

```
export EXPORT_BUCKET=xxx  # 感情分析ジョブの出力となる S3 バケット名
```

環境変数がセットされていることを確認します。

```
# 環境変数のチェック
cat <<EOF | column -t
AWS_ACCOUNT = ${AWS_ACCOUNT}
AWS_REGION = ${AWS_REGION}
DOCKER_IMAGE_NAME = ${DOCKER_IMAGE_NAME}
DOCKER_REMOTE_REPOSITORY = ${DOCKER_REMOTE_REPOSITORY}
IMAGE_TAG = ${IMAGE_TAG}
ECS_TASK_ROLE_ARN = ${ECS_TASK_ROLE_ARN}
ECS_TASK_EXEXCUTION_ROLE_ARN = ${ECS_TASK_EXEXCUTION_ROLE_ARN}
TASKDEF_ARN = ${TASKDEF_ARN}
TARGET_GROUP_ARN = ${TARGET_GROUP_ARN}
SUBNET_1 = ${SUBNET_1}
SUBNET_2 = ${SUBNET_2}
TASK_SECURITY_GROUP = ${TASK_SECURITY_GROUP}
EXPORT_BUCKET = ${EXPORT_BUCKET}
EOF
```

入力として使用する json ファイルは cli-input-templates/run-task-input.json です。この json ファイルと envsubst コマンドを使って、 CLI の入力に使用する json を生成します。

```
# envsubst を使用して、プレースホルダに環境変数を埋め込んで出力
envsubst < cli-input-templates/run-task-input.json > cli-inputs/run-task-input.json
```

生成した json ファイルを入力として、 `ecs run-task` を実行します。

```
aws ecs run-task --cli-input-json file://cli-inputs/run-task-input.json | tee cli-outpu
```

タスク実行のリクエストが正常に受け付けられれば、cli-output-samples/run-task-output.json のよ

うな形式の json が戻ります。

1.12 演習課題

Boyacky の稼働が開始され、本格的に運用を考える必要が出てきました。さしあたってアプリケーションの監視設定が何も入っていないため、まずはログ監視の導入を検討しています。

当面の方針として、アプリケーションログは CloudWatch Logs に集約することにしました。
サービスを更新や、必要に応じてその他の AWS 環境の設定を変更し、ログ監視の仕組みを実装してください。

- ゴール
  - アプリケーションのログを CloudWatch Logs から閲覧できること
  - 手順書のドラフトとなる情報が残るように、作業内容のメモを残す(できれば)
- 条件
  - ECS のサービス更新に関わる作業は AWS CLI で行う

目安： 1.0 〜 1.5h（解説含）

1.13 サービスの削除 (アンデプロイ)

サービスを消す前に、タスクの起動数を 0 にする必要があります。

```
aws ecs update-service \
  --service boyacky-webapp-svc \
  --cluster boyacky-cluster \
  --desired-count 0
```

サービスの停止にはしばらくかかります。`describe-services` を実行して、状態を確認してみましょう。

```
aws ecs describe-services \
  --services boyacky-webapp-svc \
  --cluster boyacky-cluster \
  --query "services[0] | { status: status, desiredCount: desiredCount, runningCount: ru
```

runningCount が 0 になっていれば、サービスを削除できます。 `delete-service` を実行しましょう。

```
aws ecs delete-service \
  --service boyacky-webapp-svc \
  --cluster boyacky-cluster
```