令和7年度
「専門職業人材の最新技能アップデートのための専修学校リカレント教育（リ・スキリング）推進事業」

クラウドネイティブシステム開発資料教材
問題編

本クラウドネイティブシステム開発資料教材問題編は、文部科学省の教育政策推進事業委託費による委託事業として、一般社団法人全国専門学校情報教育協会が実施した令和７年度「専門職業人材の最新技能アップデートのための専修学校リカレント教育（リ・スキリング）推進事業」の成果物です。

情報技術者の技能アップデートのためのリカレント教育推進事業

目次

演習1　AWSクラウドネイティブ基盤実装 1

演習1-0　AWS CDK準備 8

演習1-1　ネットワーク構築 12

演習1-1 Step1　ネットワーク構築 – VPC / サブネット / SG 14

演習1-1 Step2　ネットワーク構築 – VPCエンドポイント 20

演習1-1 Step3　ネットワーク構築 – ALB/WAF 25

演習1-2　コンテナ構築 33

演習1-2 Step1　コンテナ構築 - ECR 34

演習1-2 Step2　コンテナ構築 – イメージpush 37

演習1-2 Step3　コンテナ構築 – ECS 40

演習1-3　DB構築 47

演習1-3 Step1　DB構築 - RDS 49

演習1-3 Step2　DB構築 – テーブル作成 / データ投入 54

演習1-4　CI/CDパイプライン構築 58

演習1-4　Step1　CI/CDパイプライン構築 - CodeConnections 61

演習1-4　Step2　CI/CDパイプライン構築 – IAM 64

演習1-4　Step3　CI/CDパイプライン構築 - GitHub 69

演習1-4 Step4　CI/CDパイプライン構築 - CodeBuild 77

演習1-4 Step5　CI/CDパイプライン構築 - CodeDeploy 83

演習1-4 Step6　CI/CDパイプライン構築 - CodePipeline 88

演習2　ロードバランサーのHTTPS移行 94

演習3　CI/CDパイプラインのユニットテスト追加 105

演習4 - 総合演習 115

演習4 - 総合演習　ーネットワーク構築ー 122

演習4 - 総合演習　ーコンテナ構築ー 125

演習4 - 総合演習　ーDB構築ー 129

演習4 - 総合演習　ーGitHub準備ー 136

演習4 - 総合演習　ーCICDパイプライン構築ー 141

確認テスト 148

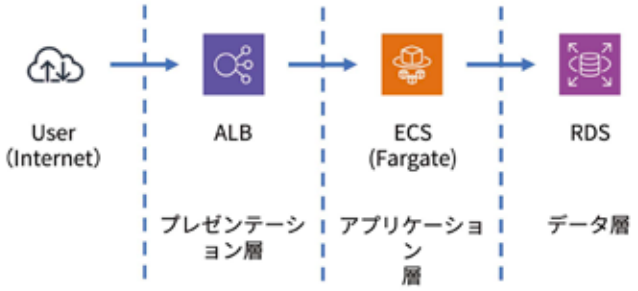
演習1

AWSクラウドネイティブ基盤実装

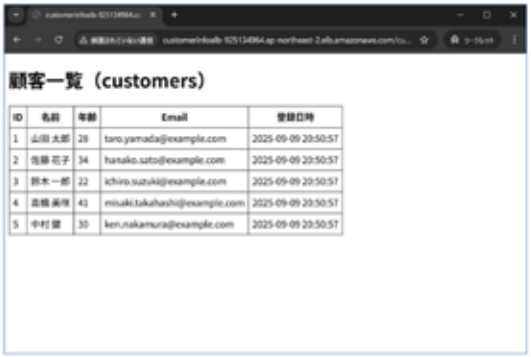
演習1　AWSクラウドネイティブ基盤実装

演習概要

演習1では、AWSに三層アーキテクチャ（ALB＋ECS＋RDS）構成のWebアプリケーションを構築します。
Webアプリケーションは、顧客情報を登録・一覧表示する簡易な管理システムを作成します。
全ての環境構築はAWS CDK（Infrastructure as Code）で自動化し、開発からデプロイまでを一貫して体得します。



構築する三層アーキテクチャ



Webアプリケーション画面

演習1　AWSクラウドネイティブ基盤実装

システム概要

顧客情報管理システムは、 AWSサービスを活用したWebベースの顧客情報を管理するアプリケーションです。コンテナ化されたサーバーレスアーキテクチャと自動CI/CDパイプラインにより、高可用性と保守性を両立したモダンなクラウドネイティブシステムを実現します。

主要な機能

顧客データ管理
データベースに登録した顧客情報をWebブラウザで表示します。

RDSデータベース連携
Amazon RDSと連携し、顧客データを管理します。またSecrets Managerを利用し、DB接続情報をセキュアに管理します。

セキュリティ対策
WAF/ALBによるL7アプリケーションの保護、リソース配置やユーザ権限により外部からのアクセス制御します。

CI/CDパイプライン
GitHub連携したパイプラインを構築し、ビルド・デプロイを自動化します。またBlue/Greenデプロイによる無停止更新を実現します。

演習1　AWSクラウドネイティブ基盤実装

ユーザー操作フロー

アプリケーションおよびインフラ基盤の構築完了後は、Webブラウザからアクセスすると、DBに登録した顧客情報が表示されるようになります。

ユーザー操作フロー

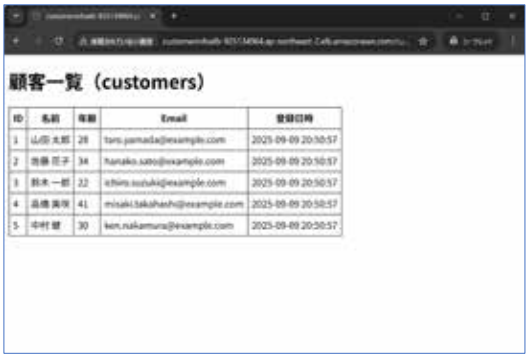
ブラウザアクセス
ユーザーがブラウザから顧客管理システムにアクセス

ALB
ロードバランサーでリクエストを受け付け、ファイアウォールでセキュリティチェック

ECS(Fargate)
サーバレスコンテナがアプリケーションロジックを実行

RDS
RDSから顧客データを取得

HTML表示
顧客情報をHTML形式でレスポンス



HTML表示
（Webアプリケーション画面）

演習1　AWSクラウドネイティブ基盤実装

CI/CDパイプラインフロー

CI/CDパイプラインの構築が完了すると、GitHubによって資材が管理され、コンテナビルドおよびデプロイの自動化を実現することができます。

パイプラインフロー

GitHub
ユーザーがGitHubに資材を登録すると、CI/CDパイプラインを自動でキックします。

CodePipeline
CI/CDパイプラインを起動し、実行・管理します。

CodeBuild
CI部分。GitHubの資材をベースに、Dockerイメージをビルドします。

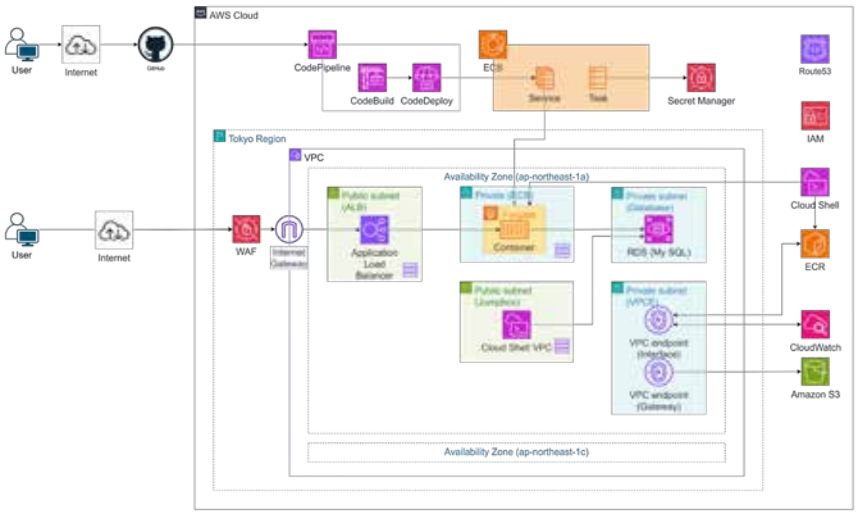
CodeDeploy
CD部分。ビルドしたコンテナイメージをコンテナ基盤ECSにデプロイします。

ECS(Fargate)
サーバレスコンテナが起動します。
Blue/Greenデプロイにより、無停止でアプリケーションが切り替わります。

演習1　AWSクラウドネイティブ基盤実装

全体構成図

演習1を通じて構築するアプリケーションおよびインフラ基盤の全体構成を示します。



演習1　AWSクラウドネイティブ基盤実装

演習1の目的・背景

- 演習概要
  AWS CDKを活用して、**クラウドネイティブなシステムを構築**しましょう。

- 目的
  - AWS CDKを用いた実践的なクラウドネイティブなインフラ構築スキル（IaCベース）を習得します。
  - 段階的なシステム構築を通じてクラウドネイティブなアーキテクチャの理解を深めます。
  - CI/CDパイプラインの実装によるDevOps実践方法の体得を目指します。

- 背景
  クラウドネイティブなシステム開発において、
  インフラのコード化（IaC）およびアプリケーションのDevOps化は不可欠なスキルとなっています。
  従来のシステム開発と異なり、クラウド化が進み、インフラとアプリの境界も曖昧になってきました。
  そのため、クラウドネイティブな開発者は**「基盤からアプリケーションまで」**を一貫して扱える**フルスタックなスキル**が求められています。

  本演習を通じて、クラウドネイティブな基盤およびアプリケーションを自動構築し、**フルスタックなスキルを体得**しましょう。

演習1　AWSクラウドネイティブ基盤実装

演習の進め方

AWS CDKを活用して、段階的にクラウドネイティブなシステムを構築しましょう。
演習1は大きく5段階（準備含む）に分けて進めていきます。

1-0　AWS CDKの準備	開発環境をセットアップし、CDKを初期化 CloudShell/CDK/CloudFormationなど
1-1　ネットワーク構築	セキュアなネットワーク基盤を構築 VPC/サブネット/SG/ALB/WAF
1-2　コンテナ構築	リポジトリ、コンテナ基盤を構築し、アプリケーションをデプロイ ECR/ECS/Fargate
1-3　DB構築	DB基盤を構築し、データ連動可能なシステムを構築 RDS/SecretsManager
1-4　CI/CDパイプライン構築	GitHubと連動したパイプラインを構築し、B/Gデプロイを実現 GitHub/CodePipeline/CodeBuild/CodeDeploy

演習1　AWSクラウドネイティブ基盤実装

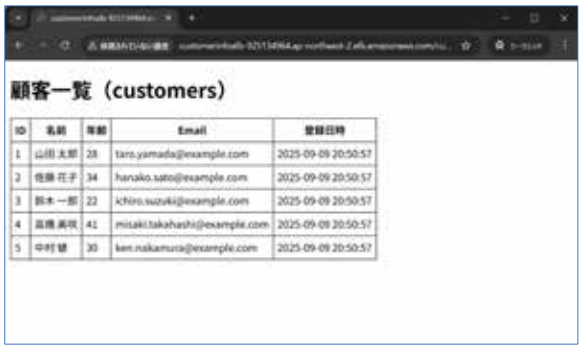
Webアプリケーション画面

アプリケーションは演習1-2、1-4で用意し、2段階的にアップグレードします。

演習1-2
コンテナ基盤およびイメージを準備し、Nginxベースのコンテナを立ち上げます。この段階ではNginxのデフォルト画面がWeb表示されます。



演習1-4
顧客情報管理システムを実装し、コンテナとして起動します。DBに登録された顧客情報を取得して、Webブラウザに表示します。

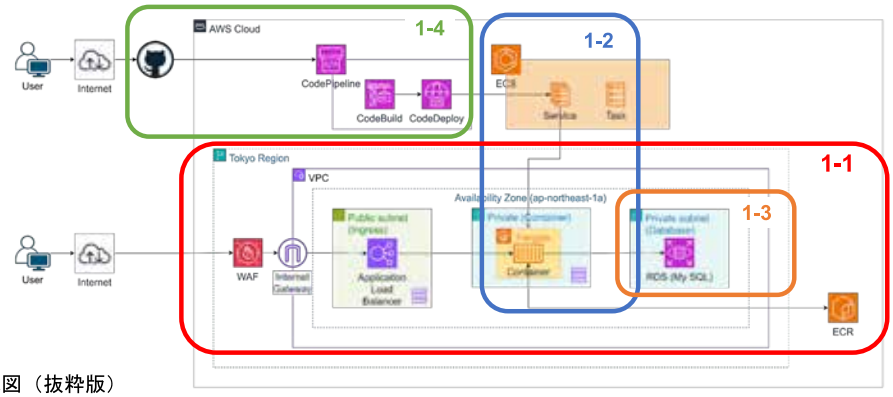


演習1　AWSクラウドネイティブ基盤実装

演習内容と全体構成図の関係性

演習1と全体構成図を紐づけると、以下の流れでシステムを構築します。

1-1　**ネットワーク構築**（VPC・サブネットなどのネットワーク基盤およびロードバランサー、FWの実装）
1-2　**コンテナ構築**（ECS Fargate、ECRを利用し、サーバレスなコンテナ基盤およびアプリケーションをデプロイ）
1-3　**DB構築**（RDSの構築およびアプリケーションが参照するデータの投入）
1-4　**CI/CDパイプライン構築**（GitHub、Codeシリーズを利用し、CI/CDパイプラインでのB/Gデプロイを実現）



全体構成図（抜粋版）

演習1　AWSクラウドネイティブ基盤実装

システム全体の共通設定値

演習1で共通して利用するリソース名を提示します。
すべての演習で共通して使用する設定値を以下にまとめます。

分類	項目	値
リージョン・AZ	リージョン	ap-northeast-1（東京）
	アベイラビリティーゾーン（AZ）	ap-northeast-1a / 1c
	AZ数	2（耐障害性確保のため）
リソース命名規則	VPC	AppVPC
	CDKプロジェクト	ecs-demo

演習1　AWSクラウドネイティブ基盤実装

参考）本講座で利用する主なAWSツール①

■ 本講座で利用するAWSツールを紹介します。

ネットワーク／インフラ構成



リージョン
AWSの物理的なデータセンター群が配置された地理的なエリア。世界中のリージョンを利用可能。



Amazon Virtual Private Cloud（VPC）
AWS上に構築する仮想ネットワーク。
独立したIP空間やルーティングを定義可能。



サブネット（パブリック/プライベート）
VPC内で定義されるIPアドレス範囲の小区画。
パブリック/プライベートに分けて使う。



アベイラビリティーゾーン（AZ）
同一リージョン内で独立性のある物理データセンター群。高可用性なシステム設計に活用。



Route53
ドメイン登録、DNSルーティング、ヘルスチェックなどを提供するスケーラブルなDNSサービス。



ELB（Elastic Load Balancing）
アプリケーションのトラフィックを複数のインスタンスやサービスへ分散するロードバランサ。



インターネットゲートウェイ
VPC内のサブネットに属するリソースにインターネットアクセスを提供するコンポーネント。



VPCエンドポイント
VPC内からS3やRDSなどAWSサービスに対してインターネットを経由せずに接続可能。

演習1　AWSクラウドネイティブ基盤実装

参考）本講座で利用する主なAWSツール②

- 本講座で利用するAWSツールを紹介します。

コンピュート／コンテナ



ECS（Elastic Container Service）
コンテナのデプロイ・スケーリング・管理を行うマネージドサービス。



Fargate
ECS/EKSの実行環境として利用できるサーバーレスなコンテナ実行基盤。インフラ管理が不要。



ECR（Elastic Container Registry）
Docker互換のコンテナイメージレジストリ。ビルドしたイメージを保存・取得してECSなどで使用。



EC2（Elastic Compute Cloud）
スペックを自由に選択できる仮想サーバ。スケーラブルで、多彩な料金体系を持つ。

データストア



RDS（Relational Database Service）
MySQLやPostgreSQL、Auroraなどのリレーショナルを提供するサービス。



S3（Simple Storage Service）
オブジェクトストレージ。画像、動画、バックアップ、ログなどさまざまなデータを保存可能。



Secrets Manager
パスワードやAPIキーなどのシークレット情報を
安全に保管・自動ローテーション可能。

演習1　AWSクラウドネイティブ基盤実装

参考）本講座で利用する主なAWSツール③

- 本講座で利用するAWSツールを紹介します。

CI/CD（DevOps）



CodePipeline
Code系サービスや外部ツールを連携してCI/CDパイプラインを構成・自動化するサービス。



CodeBuild
ソースコードをビルドして、成果物を生成するマネージドなビルドサービス。



CodeConnections
AWSと外部のソースプロバイダーを安全に接続するためのマネージドサービス。



CodeDeploy
ECS、EC2、Lambdaなどに対して、安全かつ段階的にアプリケーションをデプロイするサービス。



CloudShell
ブラウザからAWS CLIが使えるLinux環境。IAM権限を引き継ぎ、検証やCLI操作が可能。

演習1　AWSクラウドネイティブ基盤実装

参考）本講座で利用する主なAWSツール④

- 本講座で利用するAWSツールを紹介します。

セキュリティ・認証



IAM
ユーザー、グループ、ロールに対してアクセス権限を管理するための認証・認可サービス。



WAF（Web Application Firewall）
悪意のあるリクエスト（SQLインジェクション、XSSなど）を検出・遮断するL7ファイアウォール。

管理・モニタリング・自動化



CloudWatch
AWSリソースやアプリケーションのログ・メトリクス・アラームなどを集約・監視するサービス。



CloudFormation
YAMLなどのテンプレートでAWSリソースを定義し、インフラを自動構築・更新できるIaCツール。

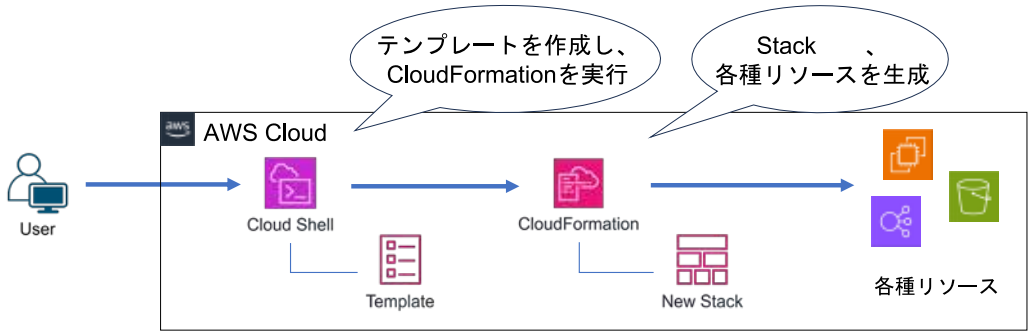
演習1-0

AWS CDK準備

演習1-0　AWS CDK準備

演習概要

- AWS CDKを利用し、CloudShellからCloudFormationにスタックを作る準備を整えます。
- 前提
  - AWSアカウントがあり、作業用ユーザが作成されていること。
  - 作業用ユーザのIAMに「**AdministratorAccess**」のポリシーが付与されていること。
  - AWS CDKは複数の言語がサポートされていますが、本演習では「**TypeScript**」を利用します。



演習1-0　AWS CDK準備

CDKプロジェクト　環境設定値

演習1-0における環境設定値は以下の通りです。

分類	項目	値
AWS CLI	リージョン	ap-northeast-1（東京）
	AWS CLIバージョン	2.28.12 以上
AWS CDK CLI	CDKバージョン	2.1011.0 以上
	TypeScriptバージョン	5.9.2 以上
リソース命名規則	CDKプロジェクト	ecs-demo
	CloudShell 作業ディレクトリ	~/environment/ecs-demo

演習1-0　AWS CDK準備

CDKプロジェクト　作成

- AWSコンソール > CloudShellを起動します。
- CloudShellからコマンドラインを実行し、AWS CDKプロジェクトを作成・初期化します。
  以降、~/environment/ecs-demoでコードを管理します（GitHubのみ別ディレクトリで管理）。

```
## ディレクトリ作成 & 移動
~ $ mkdir -p ~/environment/ecs-demo
~ $ cd  ~/environment/ecs-demo

## AWS CDK初期化（TypeScript）
ecs-demo $ cdk init app --language typescript

## パッケージインストールおよびコンパイル
ecs-demo $ sudo npm install
ecs-demo $ sudo npm install -g typescript
ecs-demo $ sudo npm run build

## AWSアカウントID の設定
ecs-demo $ AWS_ACCOUNT_ID=$(aws ¥
  sts get-caller-identity --query Account --output text)
```

```
## パラメータ確認（正しい値が表示されていること）
ecs-demo $ echo $AWS_ACCOUNT_ID   # 12桁のID
ecs-demo $ echo $AWS_REGION          # 例：ap-northeast-1

## ブートストラップ（リージョンごと一度だけ）
ecs-demo $ cdk bootstrap ¥
  aws://$AWS_ACCOUNT_ID/$AWS_REGION

## CDKコマンド
ecs-demo $ cdk version    # CDK Versionの表示を確認
```

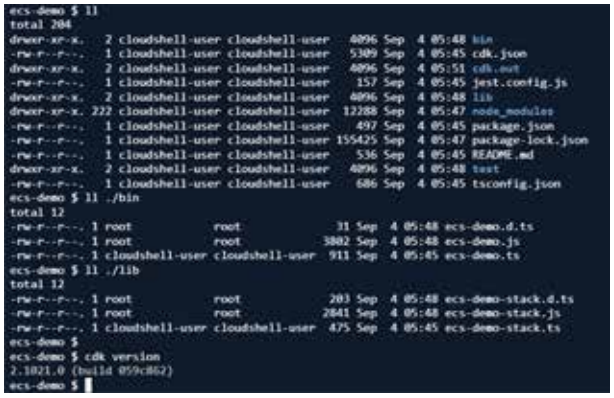
CloudShell - 実行コマンドライン

演習1-0　AWS CDK準備

CDKプロジェクト　動作確認

- 演習1-0で以下の様なディレクトリが生成されていれば、演習1-0は完了です。
  合わせてCloudFormationを確認して、**「CDKToolkit」**が生成されていることを確認してください。

CDKプロジェクトのディレクトリ構成



CloudFormation / CDKToolkitスタック



演習1-0　AWS CDK準備

参考）CDKプロジェクト構成およびCDKコマンド

- 演習では、bin/、lib/配下のファイルを編集・追加して進めます。

CDKプロジェクトのディレクトリ構成

```
ecs-demo/
├── bin/ecs-demo.ts
├── lib/ecs-demo-stack.ts
├── test/
├── cdk.json
├── package.json
└── tsconfig.json
```

bin/、lib/配下のファイルを育てます。

各ディレクトリ・ファイルの役割

- **bin/**
  CDKアプリケーションのエントリーポイント。
  ここからスタックを呼び出す。
- **lib/**
  実際のAWSリソースを定義するファイル。
  スタック（VPC, ALB, ECS など）をここに書く。
- **cdk.json**
  cdk deploy などのコマンド実行時に読み込まれる設定ファイル。
- **package.json**
  依存ライブラリ（aws-cdk-lib など）を管理。

- **test/**
  ユニットテスト用（省略可）。

演習1-0　AWS CDK準備

参考）CDKコマンド

- 最後に、頻出するCDKコマンドを記載します。

```
## 初期化
$ cdk init app --language <言語名>

## ビルド / コンパイル
$ npm install    # 必要なライブラリをインストール
$ npm run build  # TypeScript をコンパイル

## デプロイ前の状態確認
$ cdk list                  # デプロイ可能なスタック一覧を表示
$ cdk synth          #CloudFormation テンプレートを出力
$ cdk diff           # 既存環境との差分を表示

## デプロイ / 削除
$ cdk deploy <スタック名>              #AWSにデプロイ
$ cdk destroy <スタック名>             #スタックの削除
```

よく利用するCDKコマンド

演習1-0　AWS CDK準備

合格判定基準

- 演習1-0の合格判定基準は以下の通りです。

CDK準備

- AWSアカウントがあり、CloudShellにログインできている
- 作業ユーザにIAM:AdministratorAccessが付与されている
- CDK初期化およびパッケージインストールが完了している
- CDKコマンドが利用可能である（cdk versionなど）
- CloudFormationのスタック「CDKToolkit」が生成され、実行完了している

演習1-1

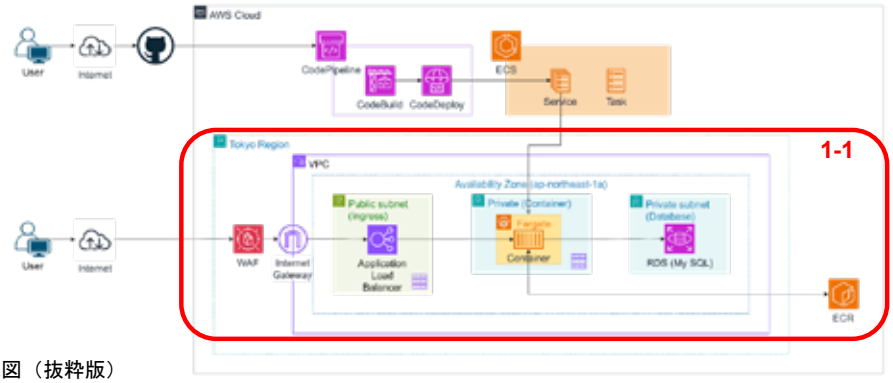
ネットワーク構築

演習1-1　ネットワーク構築

演習概要

演習1-1では、AWSにネットワーク基盤を構築します。

1-1　ネットワーク構築（VPC・サブネットなどのネットワーク基盤およびロードバランサー、FWの実装）
1-2　コンテナ構築（ECS Fargate、ECRを利用し、サーバレスなコンテナ基盤およびアプリケーションをデプロイ）
1-3　DB構築（RDSの構築およびアプリケーションが参照するデータの投入）
1-4　CI/CDパイプライン構築（GitHub、Codeシリーズを利用し、CI/CDパイプラインでのB/Gデプロイを実現）



全体構成図（抜粋版）

演習1-1　ネットワーク構築

演習概要

- AWS CDKで段階的にネットワーク環境を構築しましょう。演習1-1は3段階で構築を進めます。各Stepの要件を確認し、CDKでコード定義しましょう。

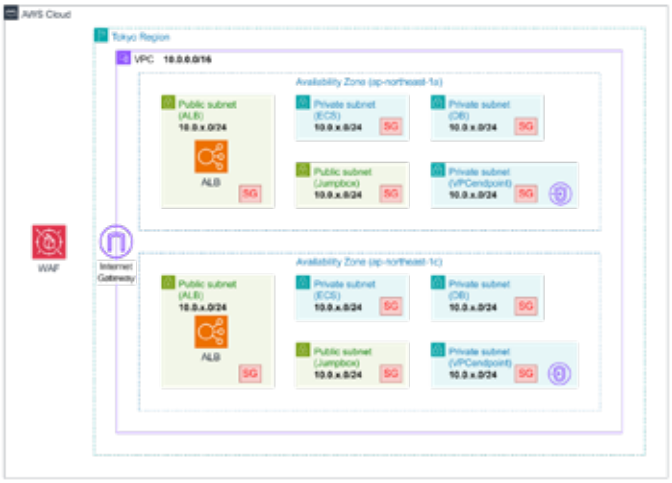
Step1　**VPC、サブネット**
複数AZにまたがるネットワーク基盤

セキュリティグループ（SG）
各サブネットのインバウンド/アウトバウンド通信を制御

Step2　**VPCエンドポイント**
VPCからAWS内サービスへのプライベート接続

Step3　**ALB**
トラフィック分散と可用性向上を実現するロードバランサー

WAF
L7ファイアウォール。外部トラフィックのセキュリティ対策およびアクセス制御



演習1-1 Step1

ネットワーク構築 – VPC / サブネット / SG

演習1-1　ネットワーク構築

Step1：VPC/AZ/サブネット　演習概要

- 東京リージョンにVPCを用意し、2AZ/10サブネットを構築しましょう。
  設定値は以下の通りです。
  今回はNAT-GWは用意しません。



VPC

項目	値
VPC名	AppVpc
VPC CIDR	10.0.0.0/16
リージョン	ap-northeast-1 (東京)
AZ	ap-northeast-1a / 1c

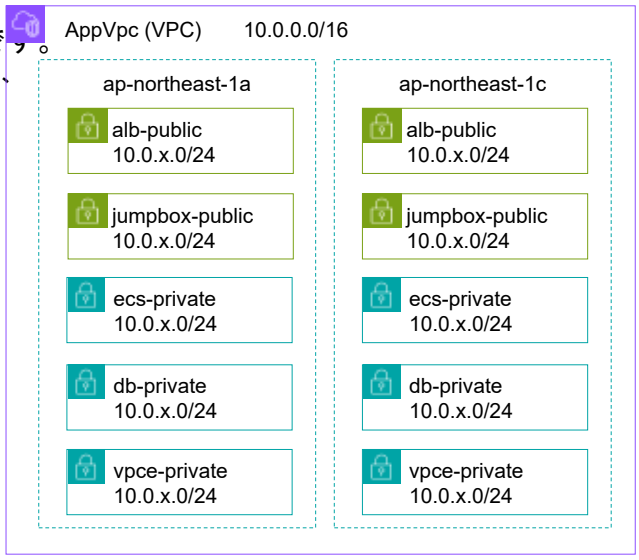
サブネット

項目	値
パブリックサブネット（4個）	CIDR：10.0.x.0 /24
プライベートサブネット（6個）	CIDR：10.0.x.0 /24
リージョン	ap-northeast-1 (東京)
AZ	ap-northeast-1a / 1c

演習1-1　ネットワーク構築

Step1：サブネット　設計

- サブネットは用途やサービス単位×Public or Privateで分割するのが基本です。
- サブネットの用途および構成は以下の通りです。
  ※今回、各サブネットのCIDRはAWS側で払い出すため、設計不要です。



Public サブネット

- alb-public (2個)：　外部からのリクエストを受付
- jumpbox-public (2個)：　踏み台アクセス用

Private サブネット

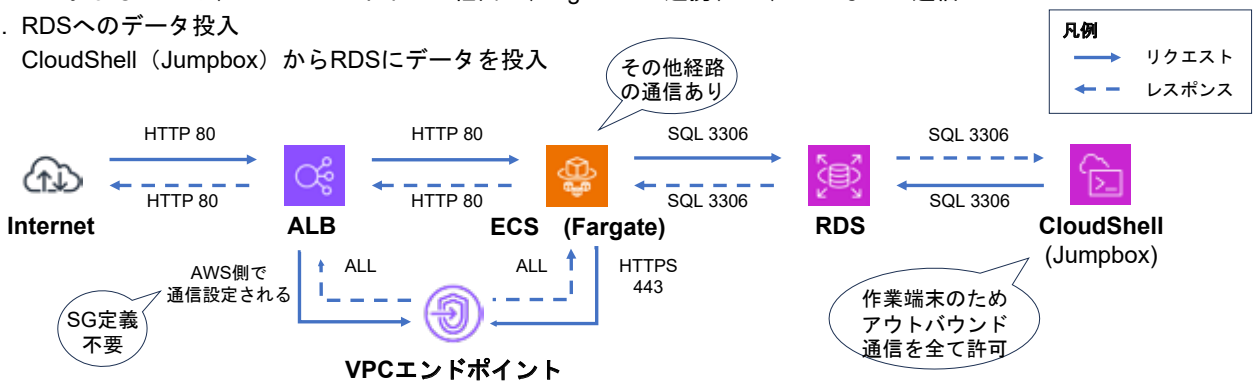
- ecs-private (2個)：　ECSコンテナワークロード配置
- db-private (2個)：　RDS（データベース）配置
- vpce-private (2個)：　VPCエンドポイント配置

演習1-1　ネットワーク構築

Step1：セキュリティグループ（SG）　演習概要

- つづいて、サブネット間の通信要件を整理して、各サブネットのSGを設計しましょう。
  演習1の主な通信要件は以下の通りです。

1. 外部より顧客情報を参照する（HTTP通信）
   Internet → ALB → ECS(Fargate) → RDS(MySQL)の経路でリクエストを処理
2. AWSサービス間のプライベート通信
   ALBおよびECSは、VPCエンドポイント経由で、log/metrics連携、S3、ECRなどと通信
3. RDSへのデータ投入
   CloudShell（Jumpbox）からRDSにデータを投入



演習1-1　ネットワーク構築

Step1：セキュリティグループ（SG）　設計

■ サブネット間の通信要件から整理したSGは以下の通りです。

SG	サブネット	主要リソース	インバウンド		アウトバウンド	
			ポート	宛先	ポート	宛先
AlbSg	alb-public	ALB	80	0.0.0.0/0	80	EcsSg
EcsSg	ecs-private	ECS	80	AlbSg	ALL	ALL
JumpSg	jumpbox-public	CloudShell	—	—	ALL	ALL
DbSg	db-private	RDS	3306	EcsSg JumpSg	—	—
VpceSg	vpce-private	VPCエンドポイント	443	EcsSg	ALL	ALL

演習1-1　ネットワーク構築

Step1：VPC/AZ/サブネット/SG　コード作成

■ AWS CDKを利用して、VPC、サブネット、ルートテーブル、SGを一括生成していきます。
ここから実際のコード作成に入りますので、CloudShellを立ち上げましょう。

■ CloudShellのCDKプロジェクトのディレクトリに移動し、コードを作成してください。
※次ページ以降に、作成するコードのサンプルを提示します。
※演習1-1のみ、サンプルの操作手順を掲載します。1-2以降は同様の手順でCloudShellを操作してください。

```
// フォルダ移動
~$ cd ~/environment/ecs-demo

// CDKアプリの定義変更（中身を全て修正）
ecs-demo $ nano bin/ecs-demo.ts
ecs-demo $ cat bin/ecs-demo.ts

// VPC、サブネット、SGのスタック定義
ecs-demo $ nano lib/net-stack.ts
ecs-demo $ cat lib/net-stack.ts
```

ecs-demo/
├─ bin/ecs-demo.ts　★修正
├─ lib/ecs-demo-stack.ts
├─ lib/net-stack.ts　★新規作成
・・・
└─ tsconfig.json

※一部ファイル/ディレクトリの記載は割愛

CDKディレクトリ構成

演習1-1　ネットワーク構築

Step1：VPC/AZ/サブネット/SG　コード作成

- **bin/ecs-demo.tsを修正**してください。
  このファイルでは、lib/から呼び出すスタックを定義します。スタックを追加する度に追記して利用します。

```
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';

// CDKの初期化、デプロイ環境の設定
const app = new App();
const env = {
  account: process.env.CDK_DEFAULT_ACCOUNT,
  region : process.env.CDK_DEFAULT_REGION,
};

// VPC / Subnets / SecurityGroups
const net = new NetStack(app, 'NetStack', { env });
```

スタック名	用途
App	CDK全体を表すクラス名
NetStack	VPC、サブネット、SGを定義

NetStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	ー	デプロイ先アカウント/リージョン

演習1-1　ネットワーク構築

Step1：VPC/AZ/サブネット/SG　コード作成

- lib/net-stack.tsを作成し、**VPC/サブネット/SG**のスタックを定義してください。

```
// クラス等のインポート
import * as cdk from 'aws-cdk-lib';
import { Stack, StackProps, CfnOutput } from 'aws-cdk-lib';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import { Construct } from 'constructs';

// クラス宣言、他スタックに公開するプロパティ
export class NetStack extends Stack {
  public readonly vpc: ec2.Vpc;
  public readonly ecsSg: ec2.SecurityGroup;
 ※追記してください

// スタック初期化
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);

  // VPC＋サブネット作成
  ※追記してください
  // セキュリティグループ作成（SG）
  ※追記してください
  // 出力
  ※追加してください
```

演習1-1　ネットワーク構築

参考）Step1：VPC/AZ/サブネット/SG　コード作成

VPC/サブネット作成　サンプルコード

```
import * as cdk from 'aws-cdk-lib';
import { Stack, StackProps, CfnOutput } from 'aws-cdk-lib';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import { Construct } from 'constructs';

export class VpcStack extends Stack {
  public readonly vpc: ec2.Vpc;

  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);

    // VPCとサブネット作成
    this.vpc = new ec2.Vpc(this, 'sampleVpc', {
      ipAddresses: ec2.IpAddresses.cidr('10.0.0.0/16'),
      maxAzs: 2,
      subnetConfiguration: [
        { name: 'public-subnet-a',  subnetType: ec2.SubnetType.PUBLIC, cidrMask: 24 },                    // Publicサブネット定義
        { name: 'private-subnet-b', subnetType: ec2.SubnetType.PRIVATE_ISOLATED, cidrMask: 24 },    // Privateサブネット定義
      ],
      natGateways: 0,     // NAT-GW定義
    });

    // 出力（確認用）
    new CfnOutput(this, 'VpcId', { value: this.vpc.vpcId });
  }
}
```

演習1-1　ネットワーク構築

参考）Step1：VPC/AZ/サブネット/SG　コード作成

セキュリティグループ作成　サンプルコード

```
import * as cdk from 'aws-cdk-lib';
import { Stack, StackProps, CfnOutput } from 'aws-cdk-lib';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import { Construct } from 'constructs';

export class SgStack extends Stack {
  public readonly vpc: ec2.Vpc;
  public readonly albSg: ec2.SecurityGroup;

  constructor(scope: Construct, id: string, props: SgStackProps) {
    super(scope, id, props);

    // ALB用SG
    this.albSg = new ec2.SecurityGroup(this, 'AlbSg', {
      vpc: props.vpc,
      description: 'SG for ALB',
      allowAllOutbound: false,  //アウトバウンド通信を全て許可とするか制御
    });
    this.albSg.addIngressRule(ec2.Peer.anyIpv4(), ec2.Port.tcp(80), 'Allow HTTP from Internet');  //インバウンド通信許可
    this.albSg.addEgressRule(this.ecsSg, ec2.Port.tcp(80), 'ALB to ECS:80');  //アウトバウンド通信許可

    // 出力
    new CfnOutput(this, 'AlbSgId', { value: this.albSg.securityGroupId });
  }
}
```

演習1-1　ネットワーク構築

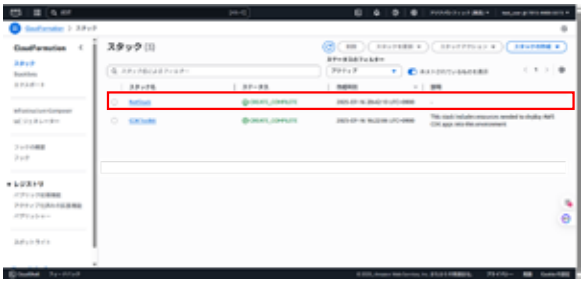
Step1：VPC/AZ/サブネット/SG　コード実行

■ CloudShellからAWS CDKでNetStackをデプロイし、VPC/サブネット/SGを作成しましょう。
cdk deployが成功し、CloudFormationの走行が完了すると、各リソースが作成されます。

```
// CDKで実行するライブラリの確認
ecs-demo $ cdk list

// VPCスタックのデプロイ
ecs-demo $ cdk deploy NetStack
```

■ CloudFormation確認
- AWSコンソール > CloudFormation > スタック
- **NetStackの実行完了**を確認しましょう。



CloudFormation – スタック

演習1-1　ネットワーク構築

Step1：VPC/AZ/サブネット/SG　動作確認

■ 各種リソースが定義通りに作成されていれば、Step1は完了です。
AWSコンソールから各種リソースを確認しましょう。



VPCダッシュボード

VPC
- VPCダッシュボード > VPC
- 新規VPC（NetStack/VpcApp）を確認

サブネット
- VPCダッシュボード > サブネット
- 新規サブネット（Public:4、Private:6）を確認

ルートテーブル
- VPCダッシュボード > ルートテーブル
- 新規ルートテーブル（10個）が自動生成を確認

IGW
- VPCダッシュボード > インターネットゲートウェイ（IGW）
- 新規IGW（1個）が自動生成されていることを確認

SG
- VPCダッシュボード > セキュリティグループ
- 新規SG（5個）を確認
  ※インバウンド/アウトバウンド通信の設定も確認

ルートテーブル、IGWは、定義しなくても自動生成されます

演習1-1　ネットワーク構築

Step1：VPC/AZ/サブネット/SG　クリーンナップ

■ CloudShellからCloudFormationのスタックおよびVPC関連リソースを全て削除します。
※これ以降、各章およびStepで各スタックを都度デプロイが必要です。

```
// CDKでNetStackを削除
ecs-demo $ cdk destroy NetStack
```

■ **リソース削除の確認**

①AWSコンソール > CloudFormation
- スタック（NetStack）の削除を確認

②AWSコンソール > VPCダッシュボード
- 以下リソースの削除を確認
  VPC/サブネット/ルートテーブル/IGW/SG



CloudFormation – スタック



VPCダッシュボード

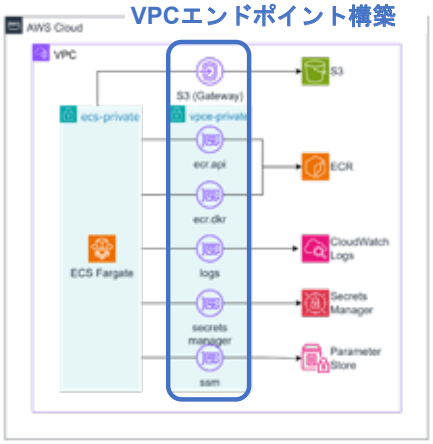
演習1-1 Step2

ネットワーク構築 – VPCエンドポイント

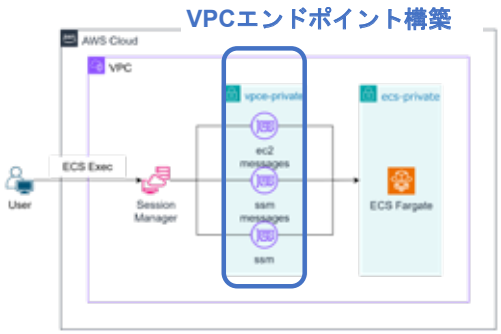
演習1-1　ネットワーク構築

Step2：VPCエンドポイント　演習概要

- VPCエンドポイントを設定することで、**インターネットを経由せずに**AWSリソースにアクセス可能となります。ECS（Fargate）の運用に必要なVPCエンドポイントを設計します。



ECS FargateとAWSリソース間の VPCエンドポイント



ECS FargateにECS Execでログインする場合に必要なVPCエンドポイント（任意）

演習1-1　ネットワーク構築

Step2：VPCエンドポイント　設計

- VPCエンドポイントの設定値は以下の通りです。

VPCエンドポイント名	種別	サービス	CDK定数	用途	必須/任意	適用サブネット
S3Gateway	Gateway	com.amazonaws.${region}.s3	S3	S3アクセス	必須	ecs-private vpce-private jumpbox-public
EcrApiEp	Interface	com.amazonaws.${region}.ecr.api	ECR	ECR API呼び出し	必須	vpce-private
EcrDkrEp	Interface	com.amazonaws.${region}.ecr.dkr	ECR_DOCKER	ECR イメージ取得	必須	vpce-private
SecretsManagerEp	Interface	com.amazonaws.${region}.secretsmanager	SECRETS_MANAGER	Secrets Manager 参照（DB参照用）	必須	vpce-private
LogsEp	Interface	com.amazonaws.${region}.logs	CLOUDWATCH_LOGS	CloudWatch Logs 送信	必須	vpce-private
SsmEp	Interface	com.amazonaws.${region}.ssm	SSM	SSM API　(ECS Exec / Session Manager)	任意	vpce-private
SsmMessagesEp	Interface	com.amazonaws.${region}.ssmmessages	SSM_MESSAGES	ECS Exec	任意	vpce-private
Ec2MessagesEp	Interface	com.amazonaws.${region}.ec2messages	EC2_MESSAGES	ECS Exec	任意	vpce-private

演習1-1　ネットワーク構築

Step2：VPCエンドポイント　コード作成

- AWS CDKを利用して、VPCエンドポイント作成のコードを作成してください。

```
// フォルダ移動
~$ cd ~/environment/ecs-demo

// CDKアプリの定義修正
ecs-demo $ nano bin/ecs-demo.ts
ecs-demo $ cat bin/ecs-demo.ts

// VPCエンドポイントのスタック定義
ecs-demo $ nano lib/vpce-stack.ts
ecs-demo $ cat lib/vpce-stack.ts
```

```
ecs-demo/
├─ bin/ecs-demo.ts    ★修正
├─ lib/ecs-demo-stack.ts
├─ lib/net-stack.ts
├─ lib/vpce-stack.ts    ★新規作成
・・・
└─ tsconfig.json
```

CDKディレクトリ構成

演習1-1　ネットワーク構築

Step2：VPCエンドポイント　コード作成

- **bin/ecs-demo.tsを修正**してください。VPCエンドポイント用スタックを呼び出すコード定義を追加します。

```
・・・略・・・
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';  //★スタック追加

・・・略・・・
// VPC Endpoints（追記）
new VpceStack(app, 'VpceStack', {
  env,
  vpc: net.vpc,
  vpceSg: net.vpceSg,
  ecsSg: net.ecsSg,
  jumpSg: net.jumpSg,
});
```

スタック名	用途
App	CDK全体を表すクラス名
NetStack	VPC、サブネット、SGを定義
VpceStack	VPCエンドポイントを定義

VpceStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	ー	デプロイ先アカウント/リージョン
ネットワーク	vpc	net.vpc	配置先VPC/サブネットを参照するため
セキュリティ	vpceSg	net.veceSg	vpce-privateサブネットに付与するSG
	ecsSg	net.ecsSg	ecs-privateサブネットに付与するSG
	jumpSg	net.jumpSg	jumpbox-publicサブネットに付与するSG

演習1-1　ネットワーク構築

Step2：VPCエンドポイント　コード作成

■ **lib/vpce-stack.ts**を作成し、**VPCエンドポイント**のスタックを定義してください。

```
// インポート
import * as cdk from 'aws-cdk-lib';
import { Stack, StackProps } from 'aws-cdk-lib';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import { Construct } from 'constructs';

// インターフェース定義
export interface VpceStackProps extends StackProps {
  vpc: ec2.IVpc;
  vpceSg: ec2.ISecurityGroup;
  ecsSg: ec2.ISecurityGroup;
  jumpSg: ec2.ISecurityGroup;
}

// スタック初期化
export class VpceStack extends Stack {
  constructor(scope: Construct, id: string, props: VpceStackProps) {
    super(scope, id, props);
    const { vpc, vpceSg, ecsSg, jumpSg } = props;
```

```
    // サブネット: vpce-private、ecs-private、jumpbox-publicを設定
    const vpceSubnets: ec2.SubnetSelection = {
      subnetGroupName: 'vpce-private',
      onePerAz: true,
    };

    ※追記してください（ecs-private、jumpbox-public分）

    // Gatewayエンドポイント作成（S3）
    ※追記してください

    // Interfaceエンドポイント作成（ECR API）
    ※追記してください

    // Interfaceエンドポイント作成（ECR Docker）
    ※追記してください

    // Interfaceエンドポイント分（xxx）
    ※残りのエンドポイント分を追記してください
  }
}
```

演習1-1　ネットワーク構築

参考）Step2：VPCエンドポイント　コード作成

VPCエンドポイント作成　サンプルコード

```
// インポート
import * as cdk from 'aws-cdk-lib';
import { Stack, StackProps } from 'aws-cdk-lib';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import { Construct } from 'constructs';

// インターフェース定義
export interface VpceStackProps extends StackProps {
  vpc: ec2.IVpc;
  vpceSg: ec2.ISecurityGroup;
  ecsSg: ec2.ISecurityGroup;
}

// スタック初期化
export class VpceStack extends Stack {
  constructor(scope: Construct, id: string, props: VpceStackProps) {
    super(scope, id, props);
    const { vpc, vpceSg, ecsSg } = props;

    // サブネット: vpce-private、ecs-privateを設定
    const vpceSubnets: ec2.SubnetSelection = {
      subnetGroupName: 'vpce-private',
      onePerAz: true,
    };

    const ecsSubnets: ec2.SubnetSelection = {
      subnetGroupName: 'ecs-private',
      onePerAz: true,
    };
```

```
    // Gatewayエンドポイント作成（S3）
    new ec2.GatewayVpcEndpoint(this, 'S3Gateway', {
      vpc,
      service: ec2.GatewayVpcEndpointAwsService.S3,
      subnets: [vpceSubnets, ecsSubnets],
    });

    // 5. Interfaceエンドポイント作成
    // Interface エンドポイント（ECR API）
    new ec2.InterfaceVpcEndpoint(this, 'EcrApiEp', {
      vpc,
      service: ec2.InterfaceVpcEndpointAwsService.ECR,
      subnets: vpceSubnets,
      securityGroups: [vpceSg],
    });
  }
}
```

演習1-1　ネットワーク構築

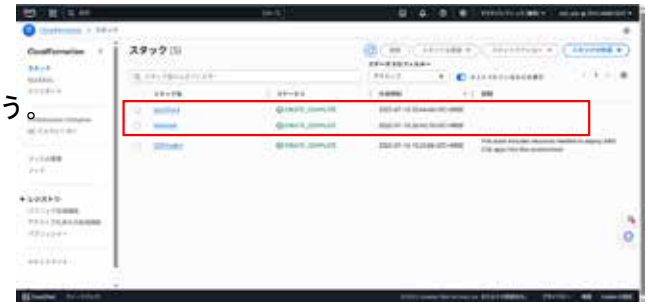
Step2：VPCエンドポイント　動作確認

- CloudShellからAWS CDKでVpceStackをデプロイして、VPCエンドポイントを作成しましょう。これまでに作ったStackを順番に起動していく必要があるので注意してください。

```
// CDKで実行するライブラリの確認
ecs-demo $ cdk list

// 各スタックのデプロイ
ecs-demo $ cdk deploy NetStack
ecs-demo $ cdk deploy VpceStack
```

- CloudFormation確認
  - AWSコンソール > CloudFormation > スタック
  - **NetStack、VpceStackの実行完了**を確認しましょう。

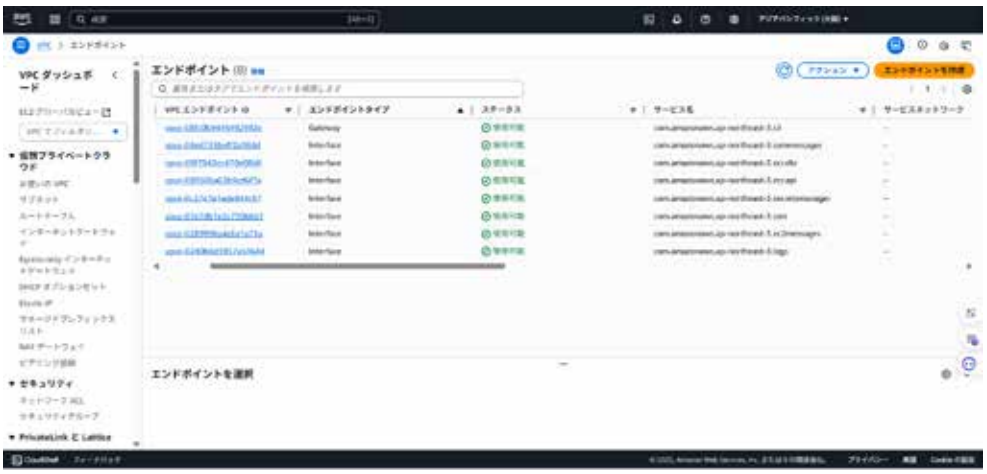


演習1-1　ネットワーク構築

Step2：VPCエンドポイント　動作確認

- VPCエンドポイントが定義通りに作成されていれば、Step2は完了です。

VPCエンドポイント
- VPCダッシュボード > エンドポイント
- 新規エンドポイントを確認（Gateway:1、Interface:7）



演習1-1　ネットワーク構築

Step2：VPCエンドポイント　クリーンナップ

■ CloudShellからCloudFormationのスタックおよびVPC関連リソースを全て削除します。

```
// CDKで全Stackを一括削除
ecs-demo $ cdk destroy --all
```

■ **リソース削除の確認**

①AWSコンソール > CloudFormation
- 2つのスタック削除を確認

②AWSコンソール > VPCダッシュボード
- 以下リソースの削除を確認
  VPC/サブネット/ルートテーブル/IGW/SG/VPCエンドポイント



CloudFormation – スタック



VPCダッシュボード

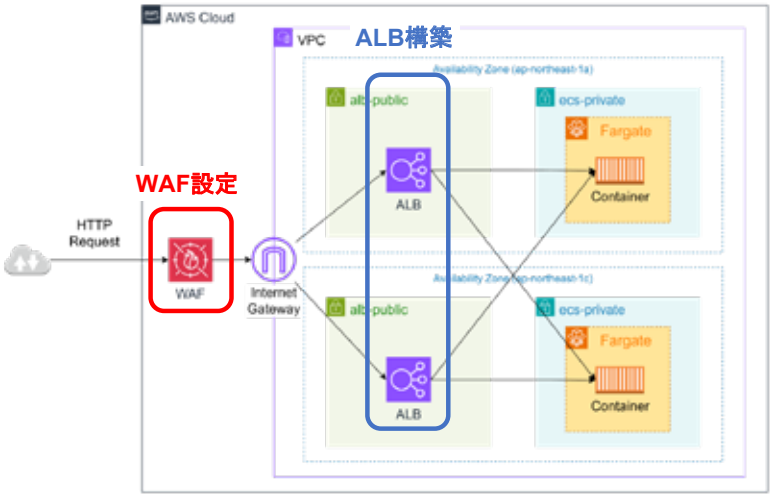
演習1-1 Step3

ネットワーク構築 – ALB / WAF

演習1-1　ネットワーク構築

Step3：ALB/WAF　演習概要

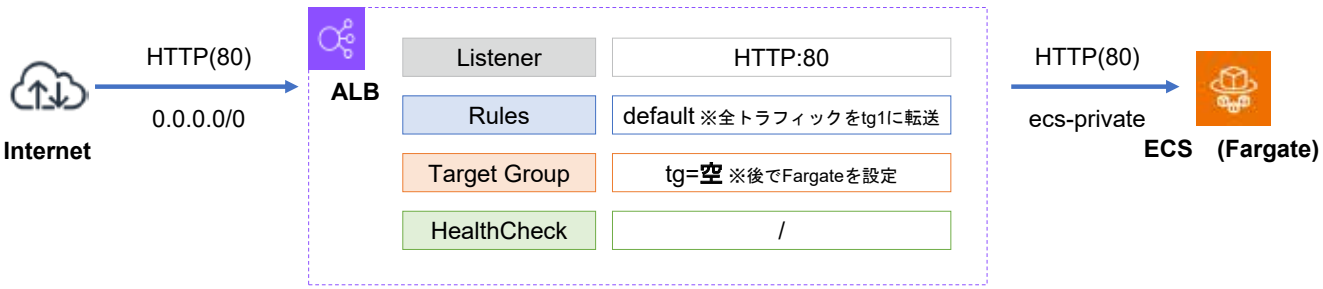
- 外部からのHTTPリクエストをECS Fargateで処理できるように、ロードバランサーとファイアウォールを設置します。Step3では、ロードバランサーはALB、ファイアウォールはWAFを利用しましょう。



演習1-1　ネットワーク構築

Step3：ALB　設計

- HTTPリクエスト(port:80)で受け、Fargateに転送します。
  ただしECS Fargateが現時点で存在しないため、ターゲットグループは空とします。



演習1-1　ネットワーク構築

Step3：ALB　設計

■ ALBの設定値は以下の通りです。

分類	項目	値
ALB	ALB名	CustomerInfoAlb
	配置先サブネット	alb-public
リスナー	プロトコル	HTTP
	ポート	80
	デフォルトターゲットグループ	tg
ターゲットグループ	ターゲットグループ名(tg)	AlbTg
	ターゲットタイプ	IP　※ECS Fargateのため
	ターゲット	なし ※ECS構築時にターゲットを上書きます
ヘルスチェック	パス	/
	ヘルスチェック間隔	30sec

演習1-1　ネットワーク構築

Step3：WAF　設計

■ 外部リクエストに対するセキュリティ対策としてWAF（L7ファイアウォール）を設定します。
今回はWebACLにAWS管理のマネージドルールグループとカスタムルールを設定します。



演習1-1　ネットワーク構築

Step3：WAF　設計

■ WAFの設定値は以下の通りです。

分類	項目	値
WebACL	WebACL名	AlbWebAcl
	デフォルトアクション	Allow
ルールA	ルール名	BlockBadBotUA
	優先度	0
	アクション	Block
	ステートメント	ByteMatchStatement
	対象フィールド	HTTP Header: User-Agent
	条件	CONTAINS “BadBot”
ルールB	ルール名	AWSManagedCommonRuleSet
（ルールグループB）	優先度	1
	アクション	OverrideAction: None
	ステートメント	ManagedRuleGroupStatement
	ベンダー	AWS
	ルールグループ	AWSManagedRulesCommonRuleSet

演習1-1　ネットワーク構築

Step3：ALB/WAF　コード作成

■ AWS CDKを利用して、**ALBとWAFを作成するコード**を作成してください。

```
// 作業ディレクトリへの移動
~ $ cd  ~/environment/ecs-demo

// ALB、WAFのスタック定義
ecs-demo $ nano lib/alb-stack.ts
ecs-demo $ cat lib/alb-stack.ts

// 既存定義の変更（追記）
ecs-demo $ nano bin/ecs-demo.ts
ecs-demo $ cat bin/ecs-demo.ts
```

```
ecs-demo/
├─ bin/ecs-demo.ts     ★修正
├─ lib/ecs-demo-stack.ts
├─ lib/net-stack.ts
├─ lib/vpce-stack.ts
├─ lib/alb-stack.ts      ★新規作成
・・・
└─ tsconfig.json
```

CDKディレクトリ構成

演習1-1　ネットワーク構築

Step3：ALB/WAF　コード作成

- **bin/ecs-demo.tsを修正**してください。AlbStackを呼び出すコード定義を追加します。

```
・・・略・・・
import { VpceStack } from '../lib/vpce-stack';
import { AlbStack } from '../lib/alb-stack';  // ★スタック追加

・・・略・・・

// ALB Stack　★ブロック追加
const alb = new AlbStack(app, 'AlbStack', {
  env,
  vpc: net.vpc,
  albSg: net.albSg,
});
```

スタック名	用途
App	CDK全体を表すクラス名
NetStack	VPC、サブネット、SGを定義
VpceStack	VPCエンドポイントを定義
AlbStack	ALB、WAFを定義

AlbStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	ー	デプロイ先アカウント/リージョン
ネットワーク	vpc	net.vpc	配置先VPC/サブネットを参照するため
セキュリティ	albSg	net.albSg	alb-publicサブネットに付与するSG

演習1-1　ネットワーク構築

Step3：ALB/WAF　コード作成

- **lib/alb-stack.ts**を作成し、**ALBとWAFのスタック**を定義してください。

```
// クラス等のインポート
import { Stack, StackProps, CfnOutput, Duration } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2   from 'aws-cdk-lib/aws-ec2';
import * as elbv2 from 'aws-cdk-lib/aws-elasticloadbalancingv2';
import * as wafv2 from 'aws-cdk-lib/aws-wafv2';

// クラス宣言、他スタックに公開するプロパティ
export interface AlbStackProps extends StackProps {
  vpc       : ec2.IVpc;
  albSg     : ec2.ISecurityGroup;
  albSubnets?: ec2.SubnetSelection;
}

// スタック初期化
export class AlbStack extends Stack {
  public readonly albDnsName: string;
  public readonly targetGroup: elbv2.ApplicationTargetGroup;

  constructor(scope: Construct, id: string, props: AlbStackProps) {
    super(scope, id, props);

// 右に続く
```

```
if (!props.albSg) {
    throw new Error('AlbStack requires a pre-created albSg from NetStack');
  }

  // サブネット選択（同一 AZ 重複を防ぐ）
  const subnetSel = props.albSubnets ??
    props.vpc.selectSubnets({ subnetGroupName: 'alb-public' });

  // ALBの作成
  ※追記してください

  // WAFの作成
  ※追記してください

    // 出力
    this.albDnsName = alb.loadBalancerDnsName;
  new CfnOutput(this, 'AlbDnsName',   { value: alb.loadBalancerDnsName });
  new CfnOutput(this, 'AlbWebAclArn', { value: webAcl.attrArn          });
  new CfnOutput(this, 'AlbTgArn',     { value: tg.targetGroupArn       });
  }
}
```

演習1-1　ネットワーク構築

参考）Step3：ALB/WAF　コード作成

ALB作成　サンプルコード

```
  // ALB作成
  const alb = new elbv2.ApplicationLoadBalancer(this, 'Alb', {
    vpc: props.vpc,
    internetFacing: true,
    securityGroup: props.albSg,
    loadBalancerName: 'CustomerInfoAlb',
    vpcSubnets: props.subnetSel,
  });

  // ターゲットグループ設定
  const tg = new elbv2.ApplicationTargetGroup(this, 'AlbTg', {
    vpc: props.vpc,
    port: 80,
    protocol: elbv2.ApplicationProtocol.HTTP,
    targetType: elbv2.TargetType.IP,    // Fargateのタスクを登録する想定
    healthCheck: { path: '/', interval: Duration.seconds(30) },    //ヘルスチェック
  });

  this.targetGroup = tg;

  // リスナー作成
  alb.addListener('HttpListener', {
    port: 80,
    protocol: elbv2.ApplicationProtocol.HTTP,
    defaultTargetGroups: [tg],                    //デフォルトターゲットグループ
  });
```


演習1-1　ネットワーク構築

参考）Step3：ALB/WAF　コード作成

WAF作成　サンプルコード

```
  // WebACL作成
    const webAcl = new wafv2.CfnWebACL(this, "Acl", {
      scope: "REGIONAL",
      defaultAction: { allow: {} },
      visibilityConfig: { cloudWatchMetricsEnabled: true, sampledRequestsEnabled: true, metricName: "Acl" },

      rules: [
        // ルール作成
        {name: "BlockBadBotUA", priority: 0, action: { block: {} },
          statement: { byteMatchStatement: {
            fieldToMatch: { singleHeader: { name: "user-agent" } },
            positionalConstraint: "CONTAINS", searchString: "BadBot",
            textTransformations: [{ priority: 0, type: "NONE" }]
          }},
          visibilityConfig: { cloudWatchMetricsEnabled: true, sampledRequestsEnabled: true, metricName: "BlockBadBotUA" }
        },

        // AWSマネージドルール（共通）
        {name: "AWSManagedCommonRuleSet", priority: 1, overrideAction: { none: {} },
          statement: { managedRuleGroupStatement: { vendorName: "AWS", name: "AWSManagedRulesCommonRuleSet" } },
          visibilityConfig: { cloudWatchMetricsEnabled: true, sampledRequestsEnabled: true, metricName: "AWSCommonRuleSet" }
        }
      ]
    });

    // WebACLとALBの紐づけ
    new wafv2.CfnWebACLAssociation(this, "AclAttach", { resourceArn: alb.loadBalancerArn, webAclArn: webAcl.attrArn });
  }
}
```

演習1-1　ネットワーク構築

Step3：ALB/WAF　動作確認

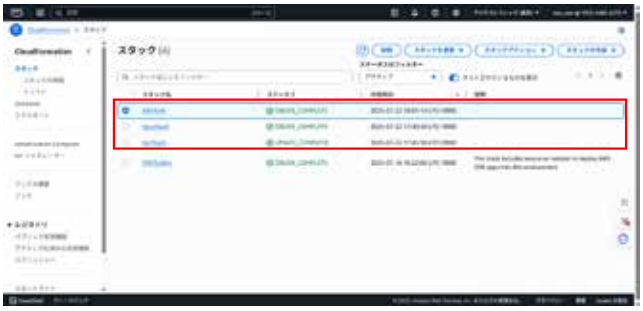
■ CloudShellからAWS CDKでAlbStackをデプロイして、ALB+WAFを作成しましょう。
これまでに作ったStackを順番に起動していく必要があるので注意してください。

```
// CDKで実行するライブラリの確認
ecs-demo $ cdk list

// 各スタックのデプロイ
ecs-demo $ cdk deploy NetStack
ecs-demo $ cdk deploy VpceStack
ecs-demo $ cdk deploy AlbStack
```

■ CloudFormation確認

- AWSコンソール > CloudFormation > スタック
- **3つのスタックの実行完了**を確認しましょう。



演習1-1　ネットワーク構築

Step3：ALB/WAF　動作確認

■ ALB、WAF(WebACL) が定義通りに作成されていれば、Step3は完了です。

ALB

- EC2 > ロードバランサー
- 新規ロードバランサー（1個）を確認



WAF

- WAF & Shield > WebACLs
- 新規WebACL（1個）を確認



演習1-1　ネットワーク構築

Step3：ALB/WAF　クリーンナップ

■ CloudShellからCloudFormationのスタックおよびVPC関連リソースを全て削除します。

```
// CDKで全リソースを削除
ecs-demo $ cdk destroy --all
```



CloudFormation – スタック

■ リソース削除の確認

①AWSコンソール > CloudFormation
- 3つのスタックの削除を確認

②AWSコンソール > 各種リソース
- 以下リソースの削除を確認
  - VPC/サブネット/ルートテーブル/IGW/SG/VPCエンドポイント
  - ALB/WAF

演習1-1　ネットワーク構築

合格判定基準

■ 演習1-1の合格判定基準は以下の通りです。

Step1（VPC/AZ/サブネット/SG）

- VPC/AZ：　CIDR:10.0.0.0/16 で2AZ・10サブネットが作成されている
- サブネット：alb-public/ecs-private/db-private/vpce-private/jumpbox-public が2個ずつ作成されている
- SG：　入出力が要件表どおりに設定されている（ALB→ECS:80、ECS→RDS:3306 etc.）
- 出力：　CfnOutputでVPC ID / SG IDを確認できる

Step2（VPCエンドポイント）

- VPCエンドポイント：　Gateway型1つ（S3）、Interface型7つ（ECR API/ECR DKR/Logs/…）が作成済み
- Vpce SGに 443/tcp from ECS SG
- VPCエンドポイントが利用可能なこと（aws ec2 describe-vpc-endpoints で Status=available）

Step3（ALB/WAF）

- ALB：ALBが1つ作成され、Listener:80 → 空TG と紐づけができている
- WAF：WebACLに BadBot（UA一致）＋ AWSManagedRulesCommonRuleSet のルールが紐づいていること
- AclAssociationが存在すること（ALBにWebACLが紐付いている）
- 外部インターネットからアクセスでき、WAF/ALBにリクエストが到達すること

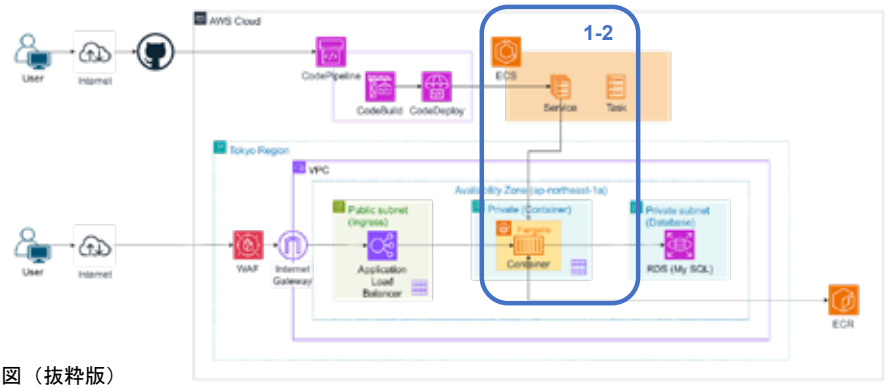
演習1-2
コンテナ構築

演習1-2　コンテナ構築

演習概要

演習1-2では、1-1で構築したネットワーク基盤上にコンテナ基盤を構築します。

1-1　ネットワーク構築（VPC・サブネットなどのネットワーク基盤およびロードバランサー、FWの実装）
1-2　コンテナ構築（ECS Fargate、ECRを利用し、サーバレスなコンテナ基盤およびアプリケーションをデプロイ）
1-3　DB構築（RDSの構築およびアプリケーションが参照するデータの投入）
1-4　CI/CDパイプライン構築（GitHub、Codeシリーズを利用し、CI/CDパイプラインでのB/Gデプロイを実現）



全体構成図（抜粋版）

演習1-2　コンテナ構築

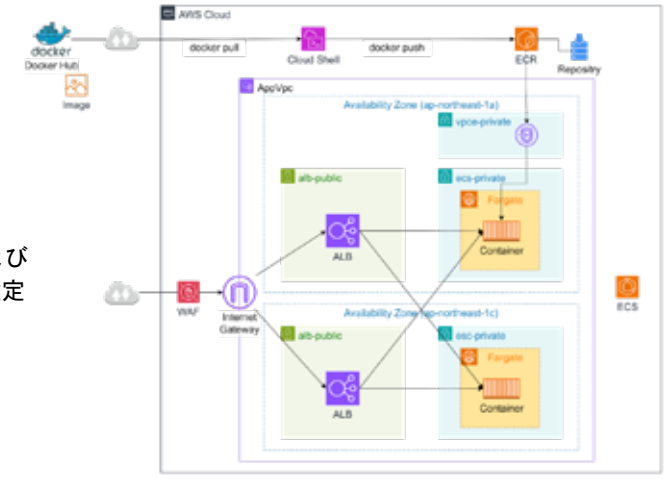
演習概要

- AWS CDKを利用してコンテナ環境を構築してください。演習1-2は3段階で構築を進めます。
  演習1-1で構築したネットワークの上でコンテナが起動するようになります。

Step1　**ECR**
Dockerイメージを保存するリポジトリ作成

Step2　**イメージpush**
Dockerイメージ作成およびECRへの登録

Step3　**ECS**
サーバーレスコンテナ(Fargate)実行環境および
コンテナ起動に必要なタスク、サービスの設定



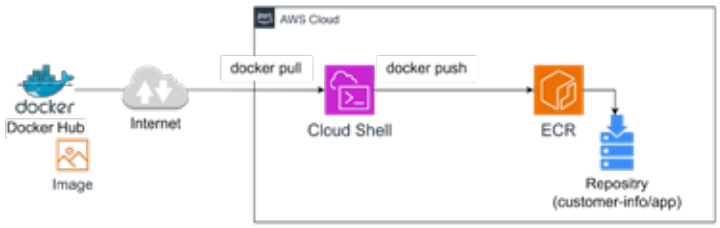
演習1-2 Step1

コンテナ構築 - ECR

演習1-2　コンテナ構築

Step1：ECR　演習概要/設計

- Dockerイメージをビルドして格納するためのECRリポジトリ(**customer-info/app**)を作成してください。Step1では、ECRリポジトリを作成し、 CloudShellからdocker pushでイメージを格納します。



- ECRの設定値は以下の通りです。

分類	項目	値
ECR	リポジトリ名	customer-info/app
	脆弱性スキャン	true
	誤削除防止	RETAIN
	ライフサイクルポリシー	7日間

演習1-2　コンテナ構築

Step1：ECR　コード作成

- AWS CDKを利用して、**ECRスタックを作成するコード**を作成してください。
  ※演習1-2より操作手順の記載は最低限にします。

- コード作成のポイント

bin/ecs-demo.ts

ECRスタック(EcrStack)を呼び出す定義を追加しましょう
- importの追加
- EcrStackを呼び出すブロックを追加

lib/ecr-stack.ts

ECRスタックを呼び出す定義ファイルを作成しましょう
- customer-info/app リポジトリを作成
- リポジトリに適用する設定（脆弱性スキャンなど）を追加

```
ecs-demo/
├── bin/ecs-demo.ts      ★修正
├── lib/ecs-demo-stack.ts
├── lib/net-stack.ts
├── lib/vpce-stack.ts
├── lib/alb-stack.ts
├── lib/ecr-stack.ts     ★新規作成
・・・
```

CDKディレクトリ構成

演習1-2　コンテナ構築

Step1：ECR　コード作成

■ **bin/ecs-demo.tsを修正**してください。EcrStackを呼び出すコード定義を追加します。

```
#!/usr/bin/env node
・・・略・・・
※importを追記してください。

・・・略・・・

// EcrStack ★ブロック追加
※追加してください。
```

スタック名	用途
App	CDK全体を表すクラス名
NetStack	VPC、サブネット、SGを定義
VpceStack	VPCエンドポイントを定義
AlbStack	ALB、WAFを定義
EcrStack	ECR(リポジトリ)を定義

EcrStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	ー	デプロイ先アカウント/リージョン

演習1-2　コンテナ構築

Step1：ECR　コード作成

■ **lib/ecr-stack.ts**を作成し、**EcrStack**を定義してください。

ECR作成　サンプルコード

```
// クラス等のimport
import * as ecr from 'aws-cdk-lib/aws-ecr'

// クラスの宣言
export class EcrStack extends cdk.Stack {
  public readonly repository: ecr.Repository;

  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // リポジトリ作成
    this.repository = new ecr.Repository(this,'SampleRepo',{
        repositoryName : 'sample',                    // 作成するリポジトリ名
        imageScanOnPush: true,                        // 自動スキャン
        removalPolicy  : cdk.RemovalPolicy.RETAIN,  // 誤削除防止
      },
    );

    // ライフサイクルポリシー
    this.repository.addLifecycleRule({
      description : 'Delete images older than 3 days',
      maxImageAge : cdk.Duration.days(3),             // イメージの保存期間
      tagStatus   : ecr.TagStatus.ANY,                // 削除対象タグ
    });

// 右に続く
```

```
// 出力（他スタックで参照する値を設定）
    new cdk.CfnOutput(this, 'SampleRepoUri',{        // 作成したURIを外部公開
      value      : this.repository.repositoryUri,
      description: 'ECR repository URI for sample',
      exportName : 'sampleRepoUri',
    });
  }
}
```

演習1-2　コンテナ構築

Step1：ECR　動作確認

- CloudShellからAWS CDKでEcrStackをデプロイして、ECRにリポジトリを作成しましょう。
  ECRにcustomer-info/appが作成されていれば、Step1は完了です。

ECR

- ECR > Private registry > Repositories
- **customer-info/appリポジトリ**を確認



CloudFormation

- CloudFormation
- スタック（EcrStack）の実行完了を確認



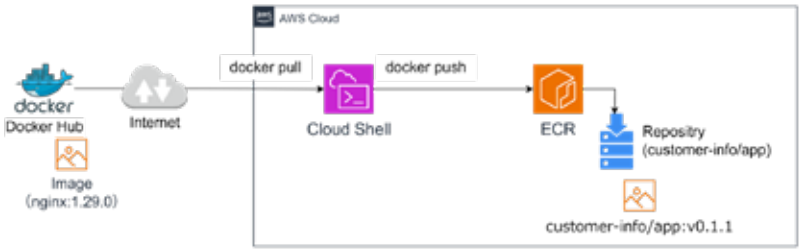
演習1-2 Step2

コンテナ構築 – イメージpush

演習1-2　コンテナ構築

Step2：イメージpush　演習概要

- CloudShellでDockerイメージを取得・ビルドして、ECRにdocker pushしてください。
  ベースイメージはNginxとし、Step1で作成したリポジトリにpushします。



- ECRに登録するイメージは以下の通りです。

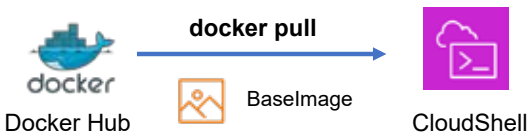
分類	項目	値
ベースイメージ	イメージ名	nginx
	タグ	1.29.0
ECR	Push先リポジトリ	customer-info/app
	タグ	v0.1.1

演習1-2　コンテナ構築

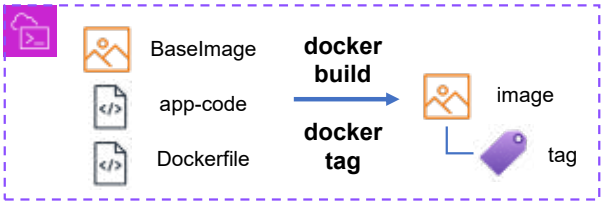
Step2：イメージpush　設計

- ECRにイメージを登録するまでの流れは以下の通りです。

① 外部レジストリからイメージを取得

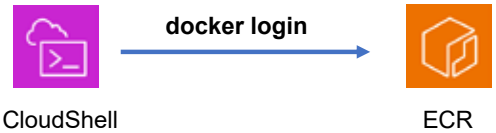


② イメージをビルド（タグ付け）



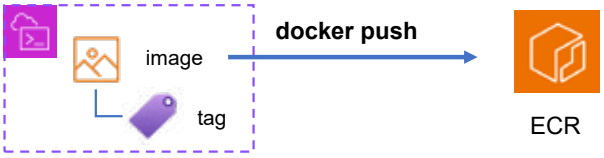
③ ECRにログイン

※ECRの認証トークン取得を含む



④ ECRにイメージを登録

※事前に格納先のリポジトリを作成する必要があります。



演習1-2　コンテナ構築

Step2：イメージpush　CLI操作

■ イメージpushまでの大まかな流れは以下の通りです。
流れに従って、イメージを取得・ビルドし、ECRに登録してください。

1. 外部レジストリからイメージを取得（docker pull）
2. イメージをビルド（docker build）
3. ビルドしたイメージにタグ付け（docker tag）
4. ECRにログイン（aws ecr）
5. ECRにイメージ登録（docker push）

■ 参考）CloudShellからECRにログインするコマンド

```
// AWSアカウントIDの設定
AWS_ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)
REGION=${AWS_DEFAULT_REGION:-ap-northeast-1}

// ECRログイン
ECR_URL=${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_REGION}.amazonaws.com
aws ecr get-login-password | docker login --username AWS --password-stdin ${ECR_URL}
```

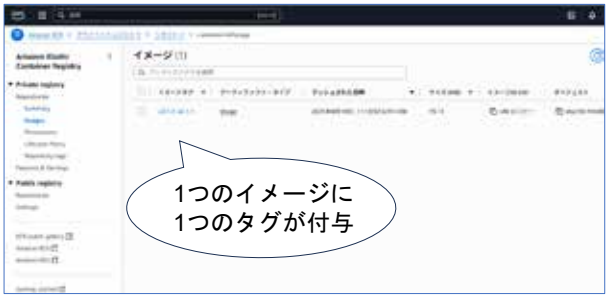
演習1-2　コンテナ構築

Step2：イメージpush　動作確認

■ ECRのリポジトリにイメージが登録されたことを確認できれば、Step2は完了です。

ECR

- ECR > Private registry > Repositories
- **customer-info/appにイメージ1つ（タグ1つ）**が登録されていることを確認
- イメージを選択すると、脆弱性スキャンの結果が出力されることを確認





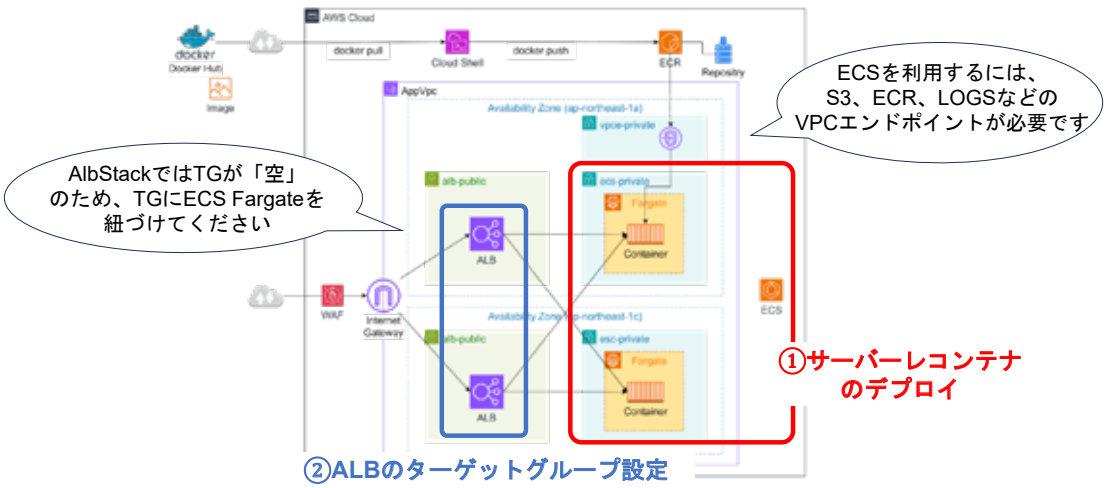
演習1-2 Step3

コンテナ構築 – ECS

演習1-2　コンテナ構築

Step3：ECS　演習概要

- AWS CDKでECSの設定を追加し、ECS Fargateでタスク(コンテナ)をデプロイします。
  イメージはStep2でECRに登録されたcustomer-info/app:v0.1.1を利用します。



演習1-2　コンテナ構築

Step3：ECS　演習概要

- ECSを利用する際に必要なVPCエンドポイントは以下の通りです。
  演習1-1 Step2で設定済ですが、再掲します。

VPCエンドポイント名	種別	サービス	CDK定数	用途	必須/任意	適用サブネット
S3Gateway	Gateway	com.amazonaws.${region}.s3	S3	S3アクセス	必須	ecs-private vpce-private
EcrApiEp	Interface	com.amazonaws.${region}.ecr.api	ECR	ECR API呼び出し	必須	vpce-private
EcrDkrEp	Interface	com.amazonaws.${region}.ecr.dkr	ECR_DOCKER	ECR イメージ取得	必須	vpce-private
SecretsManagerEp	Interface	com.amazonaws.${region}.secretsmanager	SECRETS_MANAGER	Secrets Manager 参照（DB参照用）	必須	vpce-private
LogsEp	Interface	com.amazonaws.${region}.logs	CLOUDWATCH_LOGS	CloudWatch Logs 送信	必須	vpce-private
SsmEp	Interface	com.amazonaws.${region}.ssm	SSM	SSM API　(ECS Exec / Session Manager)	任意	vpce-private
SsmMessagesEp	Interface	com.amazonaws.${region}.ssmmessages	SSM_MESSAGES	ECS Exec	任意	vpce-private
Ec2MessagesEp	Interface	com.amazonaws.${region}.ec2messages	EC2_MESSAGES	ECS Exec	任意	vpce-private

演習1-2　コンテナ構築

Step3：ECS　設計

- ECSの設定値は以下の通りです。

ECS設定値（1/2）

分類	項目	パラメータ	値
クラスター	クラスター名	clusterName	ecs-app-cluster
	メトリクスログ連携	containerInsights	true
CloudWatchロググループ	ロググループ名	logGroupName	/ecs/customer-info
	保存期間	rentention	1week
タスク定義	タスクに割り当てるCPU/MEM	cpu / memoryLimits	256 / 512
	タスク定義ファミリー	family	customer-info-task
コンテナ定義	イメージ	image	customer-info/app:v0.1.1
	ポートマッピング	portMappings	80/tcp
	ログ出力	Logging	/ecs/customer-info
	ヘルスチェック	Healthcheck	TCP 80番ポートをLISTEN 30秒間隔でチェック タイムアウト5秒、リトライ3回 起動後60秒間は失敗を無視

演習1-2　コンテナ構築

Step3：ECS　設計

■ ECSの設定値は以下の通りです。

ECS設定値（2/2）

分類	項目	パラメータ	値
Fargateサービス	サービス	serviceName	customer-info-service
	タスク数	desiredCount	2
	セキュリティグループ	securityGroups	EcsSg
	サブネット	vpcSubnets	ecs-private
	ECS Exec有効化	enableExecuteCommand	true
	起動直後のヘルチェ開始時間	healthCheckGracePeriod	60sec
ALB　※	ALBのターゲットグループ	attachToApplicationTargetGroup	AlbTgArn
出力	ECSクラスタのARN	ClusterArn	cluster.clusterArn
	Fargateのサービス名	ServiceName	service.serviceName
	タスク定義のファミリー名	TaskFamily	taskDef.family

※…ALBのTGをECS Fargateにすることで、FargateのIPアドレスが登録され、
リクエストがALBからECS Fargateに転送されるようになります。

演習1-2　コンテナ構築

Step3：ECS　コード作成

■ AWS CDKを利用して、**ECSスタックを作成するコード**を作成してください。

■ コード作成のポイント

bin/ecs-demo.ts

ECSスタック(EcsStack)を呼び出す定義を追加しましょう
- importの追加
- EcsStackを呼び出すブロックを追加

lib/ecs-stack.ts

ECSスタックを呼び出す定義ファイルを作成しましょう
- 他StackからEcsStackに渡すパラメータの設定が必要
- 設定項目が多いですが1つずつ定義する必要があります
  クラスター、ロググループ、タスク定義、コンテナ、サービス、出力
- AlbTgのターゲットにECS Fargateを設定してください
  ECSのタスク作成後にターゲットをTgに割り当てることになるため、
  ecs-stack.ts内でtgのターゲットを上書きます

```
ecs-demo/
├─ bin/ecs-demo.ts      ★修正
├─ lib/ecs-demo-stack.ts
├─ lib/net-stack.ts
├─ lib/vpce-stack.ts
├─ lib/alb-stack.ts
├─ lib/ecr-stack.ts
├─ lib/ecs-stack.ts     ★新規作成
・・・
```

CDKディレクトリ構成

演習1-2　コンテナ構築

Step3：ECS　コード作成

■ **bin/ecs-demo.tsを修正**してください。EcsStackを呼び出すコード定義を追加します。

```
#!/usr/bin/env node
・・・略・・・
※importを追記してください。

・・・略・・・

## EcsStack //ブロック追加
※追加してください。
```

スタック名	用途
App	CDK全体を表すクラス名
NetStack	VPC、サブネット、SGを定義
VpceStack	VPCエンドポイントを定義
AlbStack	ALB、WAFを定義
EcrStack	ECR(リポジトリ)を定義
EcsStack	ECS(クラスタ/タスク/サービス)を定義

EcsStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	ー	デプロイ先アカウント/リージョン
ネットワーク	vpc	net.vpc	配置先VPC/サブネットを参照するため
セキュリティ	ecsSg	net.ecsSg	ECSサービスに付与するSG
コンテナリポジトリ	repo	ecr.repository	取得元ECRリポジトリ
ALB連携	targetGroup	alb.targetGroup	登録先ターゲットグループ

演習1-2　コンテナ構築

Step3：ECS　コード作成

■ **lib/ecs-stack.ts**を作成し、**EcsStack**を定義してください。

```
// クラス等のimport
import { Duration, Stack, StackProps, CfnOutput, RemovalPolicy } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ecs  from 'aws-cdk-lib/aws-ecs';
import * as ec2  from 'aws-cdk-lib/aws-ec2';
import * as logs from 'aws-cdk-lib/aws-logs';
import * as elbv2 from 'aws-cdk-lib/aws-elasticloadbalancingv2';
import * as ecr from 'aws-cdk-lib/aws-ecr';

// 外部からのパラメータ参照
export interface EcsStackProps extends StackProps {
  vpc         : ec2.IVpc;
  ecsSg       : ec2.ISecurityGroup;
  repo        : ecr.IRepository;
  targetGroup : elbv2.ApplicationTargetGroup;
}

// クラス定義
export class EcsStack extends Stack {
  constructor(scope: Construct, id: string, props: EcsStackProps) {
    super(scope, id, props);

// 右に続く
```

```
  // ECS Cluster作成
  const cluster = new ecs.Cluster(this, 'AppCluster', {
    vpc: props.vpc,
    clusterName: 'ecs-app-cluster',  //ECSクラスタ名
    containerInsights: true,
  });
  // CloudWatchロググループ
    ※追加してください

  // タスク定義
  ※追加してください

  // Fargateサービス定義
  ※追加してください

  // ALBターゲットグループの登録
  ※追加してください

  // 出力
new CfnOutput(this, 'ClusterArn', { value: cluster.clusterArn });
  new CfnOutput(this, 'ServiceName', { value: service.serviceName });
  new CfnOutput(this, 'TaskFamily',  { value: taskDef.family });
  }
}
```

演習1-2　コンテナ構築

参考）Step3：ECS　コード作成

ECS作成　サンプルコード

```
// CloudWatchロググループ設定
const logGroupName = '/ecs/customer-info';

let logGroup: logs.ILogGroup;

try {
  logGroup = logs.LogGroup.fromLogGroupName(this, 'ExistingLogGroup', logGroupName);
} catch {
  // 存在しなければ作成
  logGroup = new logs.LogGroup(this, 'NewLogGroup', {
    logGroupName,
    retention: logs.RetentionDays.ONE_WEEK,
    removalPolicy: RemovalPolicy.RETAIN,
  });
}
// タスク定義
const taskDef = new ecs.FargateTaskDefinition(this, 'TaskDef', {
  cpu: 256,
  memoryLimitMiB: 512,
  executionRole: execRole,
  taskRole: taskRole,
  family: 'customer-info-task',
});
taskDef.addContainer('app', {
  image: ecs.ContainerImage.fromEcrRepository(props.repo, 'v0.1.1'),
  logging: ecs.LogDrivers.awsLogs({
    streamPrefix: 'customer-info',
    logGroup,
  }),
  portMappings: [{ containerPort: 80 }],
```

```
  // ヘルスチェック
  healthCheck: {
    command: [
      'CMD-SHELL',
      "awk 'NR>1 && $2 ~ /:0050$/ && $4==\"0A\"{f=1} END{exit f?0:1}' /proc/net/tcp"
    ],
    interval: Duration.seconds(30),
    timeout : Duration.seconds(5),
    retries : 3,
    startPeriod: Duration.seconds(60),
  },
  environment: {
    ENVIRONMENT: 'production',
    APP_NAME: 'customer-info',
  },
});

// サービス定義
this.service = new ecs.FargateService(this, 'AppService', {
  cluster: this.cluster,
  taskDefinition: taskDef,
  desiredCount: 2,
  serviceName: 'customer-info-service',
  securityGroups: [props.ecsSg],
  vpcSubnets: props.vpc.selectSubnets({ subnetGroupName: 'ecs-private' }),
  enableExecuteCommand: true,
  healthCheckGracePeriod: Duration.seconds(60),
  deploymentController: { type: ecs.DeploymentControllerType.CODE_DEPLOY },
});

// ALBターゲットグループ登録
this.service.attachToApplicationTargetGroup(props.targetGroup);
```

演習1-2　コンテナ構築

Step3：ECS　動作確認

- AWS CDKで**EcsStack**をデプロイして、ECS Fargateを立ち上げましょう。
  事前に他スタックをデプロイして、ネットワーク環境を構築してから実施してください。

```
// AWSアカウントID の設定
export AWS_ACCOUNT_ID=$(aws sts get-caller-identity \
  --query Account --output text)

// パラメータ確認（正しい値が表示されていること）
echo $AWS_ACCOUNT_ID  # 12桁のID
echo $AWS_REGION      # 例：ap-northeast-1

// CDKで実行するライブラリの確認
cdk list

// 各種リソースのデプロイ
cdk deploy NetStack              #VPC、サブネットなどの作成
cdk deploy VpceStack         #VPCエンドポイントの作成
cdk deploy AlbStack              #ALB、WAFの作成
cdk import EcrStack              #ECRの設定をインポート　★コマンドが違うため注意★
cdk deploy EcsStack              #ECSタスク定義・サービスの作成
```

補足）$ cdk import EcrStack

EcrStackは一度デプロイしたら、誤削除防止のため、リポジトリを残すことができます。

この場合、再度cdk deploy EcrStackを実行するとエラーになるため、一度cdk deployするだけで問題ありません。

ただ他スタックに出力値（ECRリポジトリ）を連携するため、**$ cdk import EcrStack** を実行して、設定値だけ他スタックに連携してください。

演習1-2　コンテナ構築

Step3：ECS　動作確認

■ 各リソースが作成されたことを確認します。
CloudFormationのスタックが動作して、Fargateコンテナを構築できました。

■ CloudFormation確認
- AWSコンソール > CloudFormation
- 全スタックの実行完了を確認

■ ECS確認
- Elastic Container Service
  1. クラスターが1つ作成されていること
  2. タスク定義が1つ作成されていること
  3. サービス1つ、タスク2つが作成されていること

■ ALB確認
- AWSコンソール > EC2
  1. ターゲットグループ > 登録済ターゲットに2つのIPが登録されていること
  2. ロードバランサー > CustomerInfoAlbのリソースマップに2つのターゲットが登録されていること

ECS



ALB



演習1-2　コンテナ構築

Step3：ECS　動作確認

■ CloudShellからALBにアクセスして、コンテナのヘルスチェックを確認しましょう。

```
// 1. ALB の DNS 名と TG ARN を CloudFormation の出力から取得
ALB_DNS=$(aws cloudformation describe-stacks \
  --stack-name AlbStack \
  --query "Stacks[0].Outputs[?OutputKey=='AlbDnsName'].OutputValue" \
  --output text)

TG_ARN=$(aws cloudformation describe-stacks \
  --stack-name AlbStack \
  --query "Stacks[0].Outputs[?OutputKey=='AlbTgArn'].OutputValue" \
  --output text)

echo "ALB_DNS = $ALB_DNS"
echo "TG_ARN  = $TG_ARN"

// 2. ALB → Fargate タスクのヘルスチェック状態を確認（Healthyと返却される）
aws elbv2 describe-target-health --target-group-arn "$TG_ARN" \
  --query "TargetHealthDescriptions[*].TargetHealth.State"

// 3. CloudShellからALB 経由で 200 OK が返るか確認
curl -s -o /dev/null -w '%{http_code}\n' "http://$ALB_DNS/"
```

演習1-2　コンテナ構築

Step3：ECS　動作確認

■ ブラウザからアプリケーション(Nginx)が表示されるか確認できれば、演習1-2は完了です。

```
// 4. URL確認
echo http://$ALB_DNS/
```

※以下の様なURLが表示されるのでコピーしてください。
http://CustomerInfoAlb-xxx.ap-northeast-1.elb.amazonaws.com/

■ **ブラウザ検索画面**

- ブラウザからコピーしたURLを検索して、以下の様な表示がされれば成功です。
  NginxのFargateコンテナが起動していることを確認できました。



演習1-2　コンテナ構築

Step3：ECS　クリーンナップ

■ CloudShellからCloudFormationのスタックおよび作成したリソースを全て削除します。

```
// CDKで全リソースを削除
ecs-demo $ cdk destroy --all
```

■ リソース削除の確認

①CloudFormation
- 全スタックの削除を確認
  ※Stackの削除に失敗する場合は、GUIでCloudFormationのスタックを強制削除してください

②Elastic Container Service
- タスク定義、クラスタ、サービス、タスクの削除を確認

③その他
- VPC、SG、ALB、WAF、など全リソースの削除を確認
- ECRリポジトリおよびイメージは残るため、残っていてOKです

演習1-2　コンテナ構築

合格判定基準

■ 演習1-2の合格判定基準は以下の通りです。

Step1（ECR）

- customer-info/app のECRリポジトリが作成されている
- 脆弱性スキャン（imageScanOnPush: true）が有効化されている
- ライフサイクルポリシー（例：3日以上経過したイメージを削除）が設定されている
- CloudFormation出力に SampleRepoUri が表示されている

Step2（イメージpush）

- CloudShellでNginxイメージをビルド(docker pull / docker build)し、タグを設定できている(docker tag)
- docker pushを実行でき、ECR(customer-info/app)に1つのイメージとタグが登録されている
- ECRの脆弱性スキャン結果を確認できる

Step3（ECS）

- ECSクラスター（ecs-app-cluster）が作成されている
- タスク定義が1つ作成されている（イメージ参照先がECRリポジトリになっている）
- サービスが作成され、タスクが2つ稼働している
- ALBのターゲットグループにFargateタスクのIPが登録されている
- aws elbv2 describe-target-health で healthy が返る
- http://$ALB_DNS/ でNginxのページが表示される

演習1-3

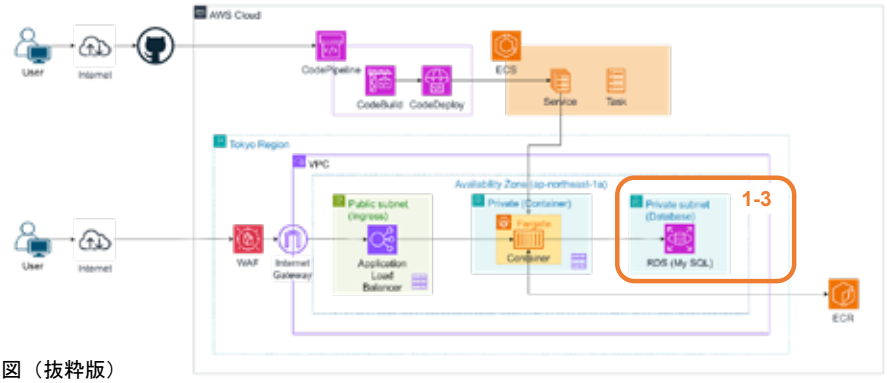
DB構築

演習1-3　DB構築

演習概要

演習1-3では、1-1で構築したネットワーク基盤上にDB環境を構築します。

1-1　ネットワーク構築（VPC・サブネットなどのネットワーク基盤およびロードバランサー、FWの実装）
1-2　コンテナ構築（ECS Fargate、ECRを利用し、サーバレスなコンテナ基盤およびアプリケーションをデプロイ）
1-3　DB構築（RDSの構築およびアプリケーションが参照するデータの投入）
1-4　CI/CDパイプライン構築（GitHub、Codeシリーズを利用し、CI/CDパイプラインでのB/Gデプロイを実現）



全体構成図（抜粋版）

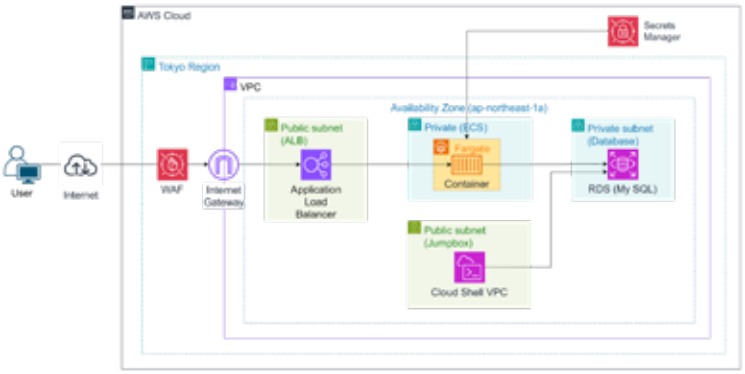
演習1-3　DB構築

演習概要

■ AWS CDKを利用してDBを構築してください。演習1-3は2段階で進めましょう。

Step1　**RDS**
顧客情報を登録するためのデータベースを構築
データベースの接続情報をSecrets Managerで管理

Step2　**テーブル作成 / データ投入**
データベースのテーブルを作成し、
初期データを投入して顧客情報を登録



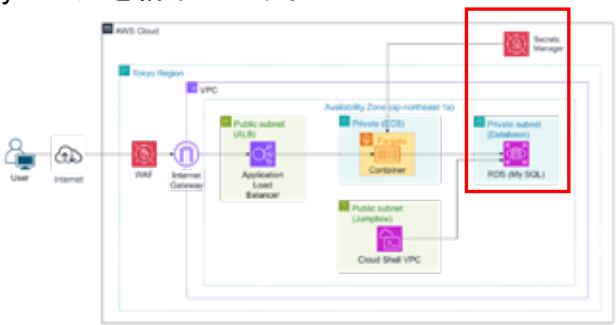
演習1-3 Step1

DB構築 - RDS

演習1-3　DB構築

Step1：RDS　演習概要

■ 顧客情報を登録するデータベースを構築してください。
Step1では、RDS（MySQL）を構築します。



■ 要件

- Amazon RDS for MySQLを利用して、プライベートサブネット（db-private）上にRDSを構築する
- RDSのインスタンスおよび初期データベースをCDKで構築する
- DB接続情報は秘匿情報としてSecrets Managerで管理し、ECS Fargateから呼び出せるようにする
  ※管理者ユーザー(admin)以外に、ECS FargateからRDSにアクセスするユーザー(app_user)のシークレットを作る
- DBインスタンスは単一　　AZで起動させる（通常は2AZ以上が望ましいが、有料になるため単一AZとします）
- RDS関連のログは、CloudWatch Logsに連携する

演習1-3　DB構築

Step1：RDS　設計

■ RDS（MySQL）の設計値は以下の通りです。

RDS設定値（1/2）

分類	項目	パラメータ	値
Secrets Manager	シークレット名（管理者ユーザー）	secretName	admin-db-credentials
	管理者ユーザー	username	admin
	管理者ユーザーパスワード	※SecretsManagerが自動生成※	※SecretsManagerが自動生成※
	シークレット名（アプリユーザー）	secretName	customer-info-app-credentials
	アプリユーザー	Username	app_user
	アプリユーザーパスワード	※SecretsManagerが自動生成※	※SecretsManagerが自動生成※
インスタンス	エンジンタイプ	engine	MySQL
	バージョン	version	ver8.0.42
	DB識別子（インスタンス名）	instanceIdentifier	application-db
	VPC名 / サブネット	vpc / vpcSubnets	proc.vpc（AppVpc）/ db-private
	インスタンスタイプ	instanceType	t3.micro
	セキュリティグループ	securityGroups	props.dbSg（DbSg）
	マルチAZ	multiAz	false（1AZ）

演習1-3　DB構築

Step1：RDS　設計

■ RDS（MySQL）の設計値は以下の通りです。

RDS設定値（2/2）

分類	項目	値	
ストレージ	ストレージタイプ	storageType	gp3
	容量（最小）	allocatedStorage	20GiB
	容量（最大）	maxAllocatedStorage	100GiB
その他	削除保護	deletionProtection	False
	CDK削除時の挙動	removalPolicy	destroy
	初期データベース	databaseName	customer_info

■ 補足）演習で作成するRDSのユーザーについて

1. **admin**
   CDKや手動CLIで利用する管理者ユーザーです。
   初期データベース構築やRDSにアクセスしてテーブル作成やデータ投入を行う際に、利用します。
2. **app_user**
   アプリケーションがRDSにアクセスする際に利用する一般ユーザーです。
   基本データ操作（SELECT / INSERT / UPDATE）を実行可能とします。

演習1-3　DB構築

Step1：RDS　コード作成

■ AWS CDKを利用して、**RDSスタックを作成するコード**を作成してください。

■ コード作成のポイント

bin/ecs-demo.ts

ECSスタック(EcsStack)を呼び出す定義を追加しましょう
- importの追加（Ecsの前にRdsを配置）
- EcsStackの前に、RdsStackを呼び出すブロックを追加

lib/rds-stack.ts

RDSスタックを呼び出す定義ファイルを作成しましょう
- 他StackからRdsStackに渡すパラメータの設定が必要
- RDSのSecretを作成して、Secrets Managerで管理する設定を追加
- RDSの各種パラメータを設定し、十分なリソースを確保
- 演習なので、スタック削除時、RDSを削除して構いません。

```
ecs-demo/
├─ bin/ecs-demo.ts     ★修正
├─ lib/ecs-demo-stack.ts
├─ lib/net-stack.ts
├─ lib/vpce-stack.ts
├─ lib/alb-stack.ts
├─ lib/ecr-stack.ts
├─ lib/ecs-stack.ts
├─ lib/rds-stack.ts      ★新規作成
・・・
```

CDKディレクトリ構成

演習1-3　DB構築

Step1：RDS　コード作成

■ **bin/ecs-demo.tsを修正**してください。RdsStackを呼び出すコード定義を追加します。

```
#!/usr/bin/env node
・・・略・・・
※importを追記してください。

・・・略・・・

// RdsStack //ブロック追加
※追記してください。

//EcSStack
・・・略・・・
```

スタック名	用途
App	CDK全体を表すクラス名
NetStack	VPC、サブネット、SGを定義
VpceStack	VPCエンドポイントを定義
AlbStack	ALB、WAFを定義
EcrStack	ECR(リポジトリ)を定義
RdsStack	RDS(MySQL)のインスタンスを定義
EcsStack	ECS(クラスタ/タスク/サービス)を定義

RdsStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	ー	デプロイ先アカウント/リージョン
ネットワーク	vpc	net.vpc	配置先VPC/サブネットを参照する
セキュリティ	vpceSg	net.dbSg	db-privateサブネットに付与するSG

演習1-3　DB構築

Step1：RDS　コード作成

- **lib/rds-stack.ts**を作成し、**RdsStack**を定義してください。

RDS作成　サンプルコード

```
// クラス等のimport
import * as ecr from 'aws-cdk-lib/aws-ecr'
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ec2 from 'aws-cdk-lib/aws-ec2';
import * as rds from 'aws-cdk-lib/aws-rds';
import * as secretsmanager from 'aws-cdk-lib/aws-secretsmanager';
// インタフェース定義
export interface RdsStackProps extends cdk.StackProps {
  vpc: ec2.IVpc;
  dbSg: ec2.ISecurityGroup;
}
// クラスの宣言
export class RdsStack extends cdk.Stack {
  public readonly dbInstance: rds.DatabaseInstance;
  public readonly dbSecret: secretsmanager.ISecret;
  public readonly appSecret: secretsmanager.ISecret;
  public readonly dbHost: string;
  constructor(scope: Construct, id: string, props: RdsStackProps) {
    super(scope, id, props);
    // シークレットの作成（Secrets Managerで管理）
    this.dbSecret = new rds.DatabaseSecret(this, 'AdminDbSecret', {
      secretName: 'admin-db-credentials',
      username: 'admin',
    });
    const appSecret = new secretsmanager.Secret(this, 'CustomerInfoAppSecret',  {
      secretName: 'customer-info-app-credentials',
      generateSecretString: {
        secretStringTemplate: JSON.stringify({ username: 'app_user' }),
        generateStringKey: 'password',
        excludePunctuation: true,
      },
    });
```

```
// MySQL インスタンス
    this.dbInstance = new rds.DatabaseInstance(this, 'ApplicationDb', {
      engine: rds.DatabaseInstanceEngine.mysql({
        version: rds.MysqlEngineVersion.VER_8_0_42,
      }),
      instanceIdentifier: 'application-db
      vpc: props.vpc,
      vpcSubnets: { subnetGroupName: 'db-private' },
      instanceType: new ec2.InstanceType(process.env.DB_INSTANCE_TYPE ?? 't3.micro'),
      securityGroups: [props.dbSg],
      credentials: rds.Credentials.fromSecret(this.dbSecret),
      multiAz: false,
      allocatedStorage: 20,
      maxAllocatedStorage: 100,
      deletionProtection: false,
      removalPolicy: cdk.RemovalPolicy.DESTROY,
      databaseName: 'customer_info
    });

    this.dbHost = this.dbInstance.dbInstanceEndpointAddress;

    // 出力（他スタックで参照する値を設定）
    new cdk.CfnOutput(this, 'ApplicationDbEndpoint', {
      value: this.dbInstance.dbInstanceEndpointAddress,
    });
    new cdk.CfnOutput(this, 'CustomerInfoDbSecretName', {
      value: this.dbSecret.secretName,
    });
  }
}
```

演習1-3　DB構築

Step1：RDS　動作確認

- CloudShellからAWS CDKでRdsStackをデプロイして、RDSインスタンスを構築しましょう。
  RDSに「application-db」、Secrets Managerに2つのシークレットが作成されていれば、GUIでの確認は終わりです。

RDS

- Aurora and RDS > データベース
- DB識別子**application-db**を確認



Secrets Manager

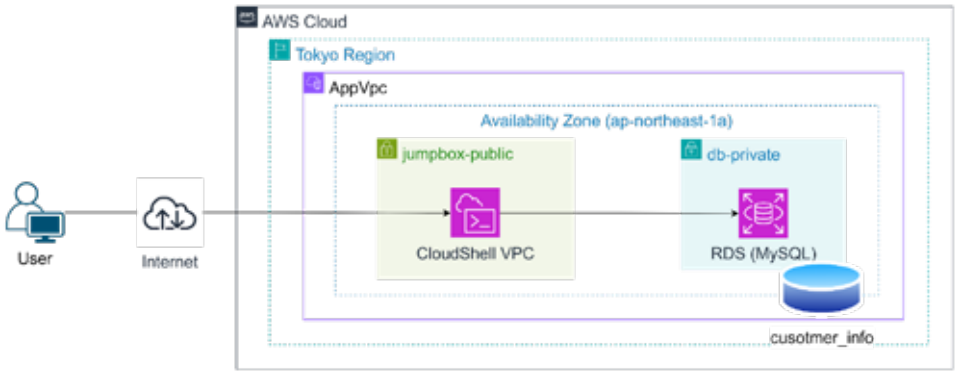
- Secrets Manager > シークレット
- 2つのシークレットの生成を確認
  - **admin-db-credentials**
  - **customer-info-app-credentials**



演習1-3　DB構築

Step1：RDS　動作確認

■ RDSインスタンスにアクセスして、初期データベースを確認してください。
演習ではCloudShell VPCからアクセスします。
※踏み台として新規EC2を立てて、RDSにアクセスするなど別の手段でも確認することは可能です。



※ALB、Fargateなどその他リソースの記載は省略

演習1-3　DB構築

Step1：RDS　動作確認

CloudShell VPC

- CloudShell > アクション > Create VPC environment
- VPC / サブネット(jumpbox-public) / SG(jumpSG) を選択して、「Create」すると利用可能になります。





演習1-3　DB構築

Step1：RDS　動作確認

- CloudShell VPCからRDSインスタンスにアクセスし、初期データベースを確認できたら、Step1は完了です。以下の流れに従って確認することができます。
  1. CloudShell VPCにアクセス
  2. MySQLのインストール
  3. Secrets ManagerでDB接続情報を確認
  4. MySQLクライアントでログイン
  5. 初期データベース（customer_info）を確認（SHOW DATABASES）

- 参考）CloudShell VPCからRDS(MySQL)にログインするコマンド

```
// Secrets ManagerからDB接続情報を確認
$ aws secretsmanager get-secret-value --secret-id application-db-credentials \
  --query SecretString  --output text | jq .

//  MySQLクライアントでログイン
// Secrets Managerで管理しているDB接続情報を利用します
$ mysql -h <host> -u <username> -p
Enter password: <password>
```

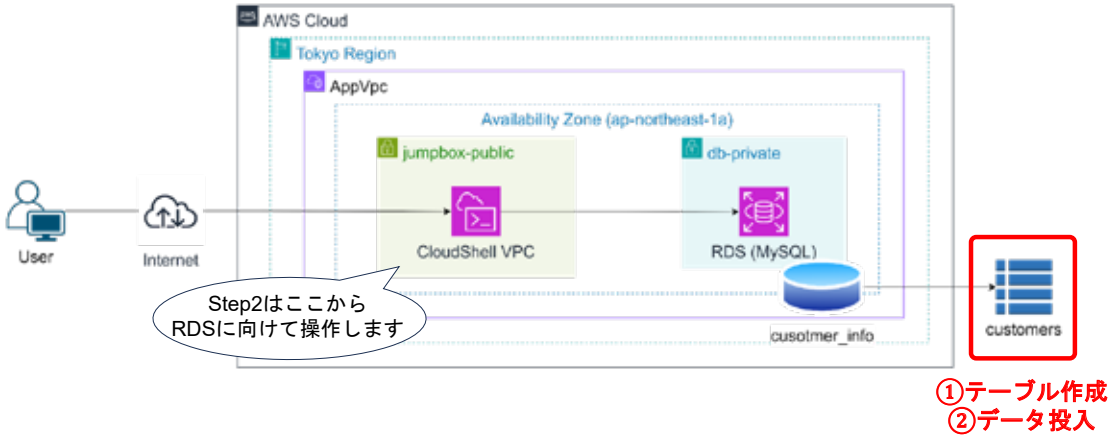
演習1-3 Step2

DB構築 – テーブル作成 / データ投入

演習1-3　DB構築

Step2：テーブル作成 / データ投入　演習概要

- RDSに新規テーブルを作成し、顧客データを登録してください。
  演習ではjumpbox-publicのCloudShell VPCからの操作を想定しています。



演習1-3　DB構築

Step2：テーブル作成 / データ投入　設計

- 初期DB(customer_info)にテーブル(customers)を作成し、データを登録しましょう。
  テーブルおよび登録データの値は以下の通りです。

テーブル設計　（customers）

カラム	データ型	制約	説明
id	INT	PRIMARY KEY, AUTO_INCREMENT	項番ID
name	VARCHAR(50)	NOT NULL	名前
age	INT	-	年齢
email	VARCHAR(100)	UNIQUE	メールアドレス
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	登録日時

登録データ

id	name	age	email	created_at
1	山田太郎	28	taro.yamada@example.com	2025-09-09 10:00:00
2	佐藤花子	34	hanako.sato@example.com	2025-09-09 10:01:00
3	鈴木一郎	22	ichiro.suzuki@example.com	2025-09-09 10:02:00
4	高橋美咲	41	misaki.takahashi@example.com	2025-09-09 10:03:00
5	中村健	30	ken.nakamura@example.com	2025-09-09 10:04:00

演習1-3　DB構築

Step2：テーブル作成 / データ投入　設計

- 合わせて、アプリケーションが参照するユーザー(app_user)も作成しましょう。
  設定値は以下の通りです。

ユーザー設計（app_user）

項目	設定値	説明
ユーザー名	app_user	アプリケーションが利用するMySQLユーザー
ホスト	%	アクセス可能なデータベース
テーブル	customer_info.*	アクセス可能なテーブル （customer_infoの全テーブルを対象）
権限	SELECT/INSERT/UPDATE	付与する操作権限

演習1-3　DB構築

Step2：テーブル作成 / データ投入　CLI操作

- イメージpushまでの大まかな流れは以下の通りです。
  流れに従って、イメージを取得・ビルドし、ECRに登録してください。

【事前準備】
1. MySQLクライアントでログイン

【ユーザ作成】
2. ユーザー作成（app_user）
3. ユーザー権限付与（SELECT / INSERT / UPDATE）

【テーブル作成 / データ投入】
4. データベース選択（customer_info）
5. テーブル作成（customers）
6. データ投入（登録データ5行分をINSERT）
7. データ確認

演習1-3　DB構築

Step2：テーブル作成 / データ投入　CLI操作

■ 参考）SQLコマンド

```
// ユーザー作成（サンプル）
// '<password>'は、SecretsManagerのcustomer-info-app-credentialsのpasswordに合わせてください。
CREATE USER 'app_user'@'%' IDENTIFIED BY '<password>';

// 権限付与
GRANT SELECT, INSERT, UPDATE
ON customer_info.* TO 'app_user'@'%';

FLUSH PRIVILEGES;  // 権限の即時反映

// テーブル作成（サンプル）
CREATE TABLE customers (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(50) NOT NULL
);

// データ投入（サンプル）
INSERT INTO customers (name, age, email) VALUES
('山田 太郎', 28, 'taro.yamada@example.com'),
('中村 健', 30, 'ken.nakamura@example.com');
```

演習1-3　DB構築

Step2：テーブル作成 / データ投入　動作確認

■ customerテーブルにデータを5件登録されたことを確認して、Step2は完了です。

CloudShell VPC

- customersテーブルをselectし、以下の様にデータが登録されていること

```
MySQL [customer_info]> SELECT * FROM customers;
+----+---------------+------+-----------------------------+---------------------+
| id | name          | age  | email                       | created_at          |
+----+---------------+------+-----------------------------+---------------------+
|  1 | 山田 太郎     |   28 | taro.yamada@example.com     | 2025-09-09 20:50:57 |
|  2 | 佐藤 花子     |   34 | hanako.sato@example.com     | 2025-09-09 20:50:57 |
|  3 | 鈴木 一郎     |   22 | ichiro.suzuki@example.com   | 2025-09-09 20:50:57 |
|  4 | 高橋 美咲     |   41 | misaki.takahashi@example.com | 2025-09-09 20:50:57 |
|  5 | 中村 健       |   30 | ken.nakamura@example.com    | 2025-09-09 20:50:57 |
+----+---------------+------+-----------------------------+---------------------+
5 rows in set (0.001 sec)
```

演習1-3　DB構築

合格判定基準

■ 演習1-3の合格判定基準は以下の通りです。

Step1（RDS）

- RDSインスタンス（application-db）が作成されている
- Secrets Managerにシークレットが2つ作成されている
- CloudShell VPCを起動し、RDSインスタンスに接続できる
- 初期データベース（customer_info）が作成されている

Step2（テーブル作成 / データ投入）

- MySQLにログインできている
- app_userが作成され、適切な権限が付与されている
- 初期データベース（customer_info）に新規テーブル（customers）が作成できている
- customersテーブルに5件のデータを登録できている

演習1-4

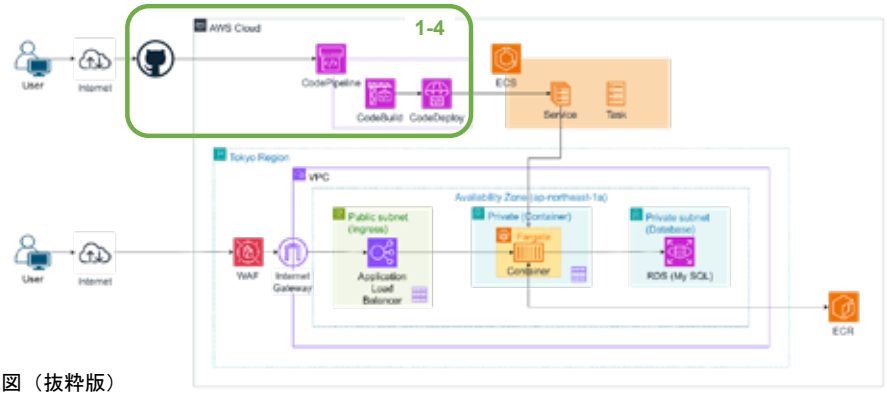
CI/CDパイプライン構築

演習1-4　CI/CDパイプライン構築

演習概要

演習1-4では、CI/CDパイプラインを構築します。

1-1　ネットワーク構築（VPC・サブネットなどのネットワーク基盤およびロードバランサー、FWの実装）
1-2　コンテナ構築（ECS Fargate、ECRを利用し、サーバレスなコンテナ基盤およびアプリケーションをデプロイ）
1-3　DB構築（RDSの構築およびアプリケーションが参照するデータの投入）
1-4　CI/CDパイプライン構築（GitHub、Codeシリーズを利用し、CI/CDパイプラインでのB/Gデプロイを実現）

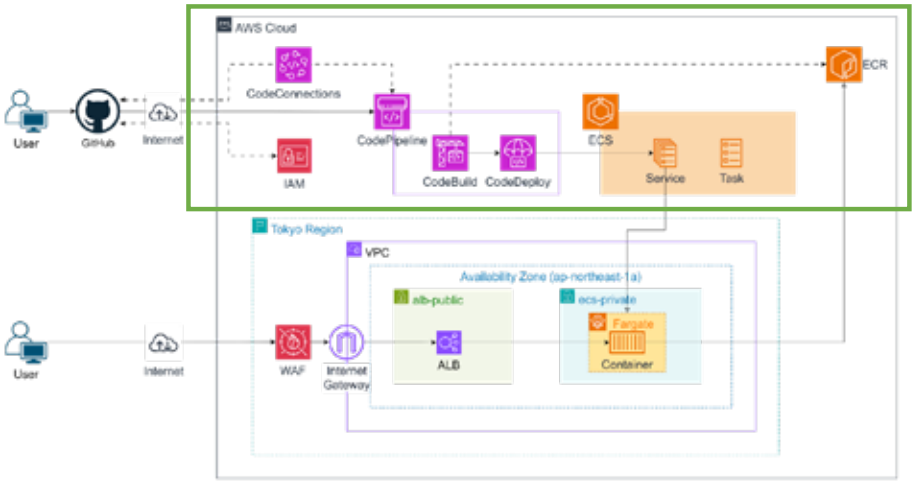


全体構成図（抜粋版）

演習1-4　CI/CDパイプライン構築

演習概要

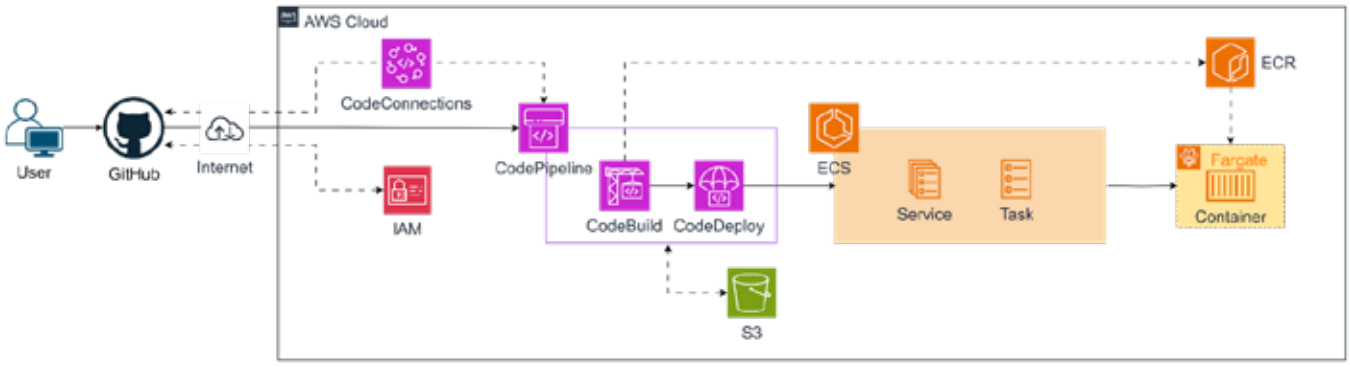
- AWS CDKを利用してCI/CDパイプラインを構築してください。
- GitHub～ECR/ECSを連携し、GitHubリポジトリのコード変更をトリガーに、CodePipelineを起動し、ビルド～デプロイの自動化を実現します。



演習1-4　CI/CDパイプライン構築

演習概要　CI/CDパイプラインアーキテクチャ

- GitHub～ECR/ECSを連携し、GitHubリポジトリのコード変更をトリガーに、CodePipelineが起動し、ビルド～デプロイが全て自動化されます。これによりECR FargateのタスクのB/Gデプロイを実現します。

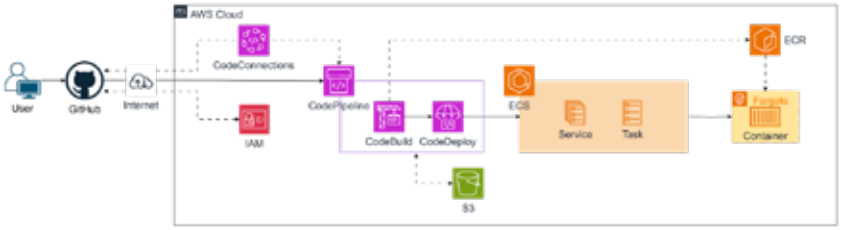


CI/CDパイプライン構成のみ抜粋

演習1-4　CI/CDパイプライン構築

演習概要　CI/CDパイプラインアーキテクチャ

- CI/CDパイプラインにおける各サービスの役割は以下の通りです。



CodePipeline
GitHubのコード変更をトリガーにパイプラインを実行
ソース確認、ビルド、デプロイまでを統合的に管理

CodeBuild
GitHubのソースコードをベースにコンテナイメージをビルドし、イメージをECRに格納

CodeDeploy
CodeBuildで生成されたイメージをECSにデプロイ
タスク(コンテナ)はB/Gデプロイメントで切り替え

GitHub
ソースコードを管理するリポジトリ。AWSとOIDC連携し、コード変更をトリガーにCI/CDパイプラインを起動

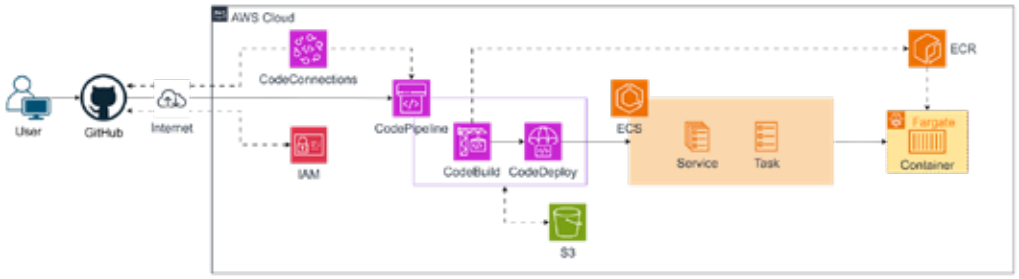
IAM
IAMロールを管理し、GitHubやCodeシリーズが他のAWSサービスを操作するためのポリシーを付与

CodeConnections
CodePipelineからGitHubのコードを直接参照させるため、AWSとGitHubを安全に接続

演習1-4　CI/CDパイプライン構築

演習概要

- 演習1-4は6段階で構築を進めます。



Step1 **CodeConnections**
AWSとGitHubの接続設定

Step2 **IAM**
各リソースにポリシー（アクセス制御）を付与

Step3 **GitHub**
GitHubのリポジトリ作成およびGit資材登録

Step4 **CodeBuild**
Dockerイメージのビルド、プッシュを定義（CI部分）

Step5 **CodeDeploy**
ECSへのB/Gデプロイ設定を定義（CD部分）

Step6 **CodePipeline**
gitHubをトリガーにCI/CDパイプラインを動作

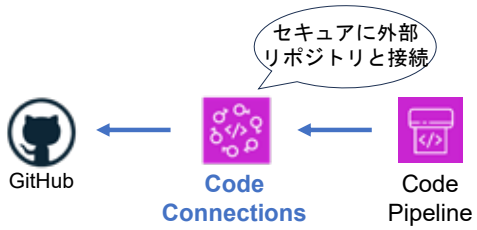
演習1-4　Step1

CI/CDパイプライン構築 - CodeConnections

演習1-4　CI/CDパイプライン構築

Step1：CodeConnections　演習概要/設計

■ CI/CDパイプラインのリポジトリして**GitHub**を利用します。
CodePipelineでCI/CDパイプライン実行時、GitHubのリポジトリを直接参照させるため、
CodeConnectionsを設定して、AWSとGitHubを安全に接続してください。



項目	パラメータ	値
Connection名	connectionName	CustomerInfoGitHub
接続先プロバイダー	providerType	GitHub

■ 前提

- GitHubアカウントがあること
  お持ちでない場合は、新規アカウントを作成してください
- AWSのコードリポジトリサービスである、CodeCommitは新規サービス受付停止のため、
  GitHubをリポジトリとして採用します（GitLab等でも）。

演習1-4　CI/CDパイプライン構築

Step1：CodeConnections　コード作成

■ AWS CDKを利用して、**CodeConnectionsを作成するコード**を作成してください。
※一度設定すればいいため、GUIで手動作成してもよいです。

■ コード作成のポイント

bin/ecs-demo.ts

IAMスタック(ConnectionStack)を呼び出す定義を追加しましょう

- importの追加
- ConnectionStackを呼び出すブロックを追加

lib/connection-stack.ts

Connectionスタックを呼び出す定義ファイルを作成しましょう。
・接続先はGitHubを指定
・作成したConnectionはGUIから別途承認が必要

```
ecs-demo/
├─ bin/ecs-demo.ts    ★修正
├─ lib/ecs-demo-stack.ts
├─ lib/net-stack.ts
├─ lib/vpce-stack.ts
├─ lib/alb-stack.ts
├─ lib/ecr-stack.ts
├─ lib/ecs-stack.ts
├─ lib/rds-stack.ts
├─ lib/connection-stack.ts    ★新規作成
・・・
```

CDKディレクトリ構成

演習1-4　CI/CDパイプライン構築

Step1：CodeConnections　コード作成

■ **bin/ecs-demo.tsを修正**してください。EcrStackを呼び出すコード定義を追加します。

```
#!/usr/bin/env node
・・・略・・・
※importを追記してください。

・・・略・・・

## IamStack //ブロック追加
※追加してください。
```

スタック名	用途
App	CDK全体を表すクラス名
NetStack	VPC、サブネット、SGを定義
VpceStack	VPCエンドポイントを定義
AlbStack	ALB、WAFを定義
EcrStack	ECR(リポジトリ)を定義
EcsStack	ECS(クラスタ/タスク/サービス)を定義
RdsStack	RDS(MySQL)のインスタンスを定義
ConnectionStack	CodeConnections(GitHub接続)を定義

ConnectionsStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	ー	デプロイ先アカウント/リージョン

演習1-4　CI/CDパイプライン構築

Step1：CodeConnections　コード作成

■ **lib/connections-stack.ts**を作成し、**ConnectionStack**を定義してください。

CodeConnections作成　サンプルコード

```
// クラス等のimport
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as codeconnections from 'aws-cdk-lib/aws-codestarconnections';
// クラスの宣言
export class ConnectionStack extends cdk.Stack {
  public readonly connectionArn: string;
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // CodeConnectionsの生成
    const conn = new codeconnections.CfnConnection(this, 'GitHubConnection', {
      connectionName: 'xxx',    // 任意の管理名
      providerType: 'xxx',    // 接続先プロバイダ
    });
    this.connectionArn = conn.attrConnectionArn;

    // 出力
    new cdk.CfnOutput(this, 'GitHubConnectionArn', {
      value: this.connectionArn,
      exportName: 'GitHubConnectionArn',
    });
  }
}
```

演習1-4　CI/CDパイプライン構築

Step1：CodeConnections　動作確認

- CDKから**ConnectionStack**を実行してください。
  スタック実行後、**GUIから手動承認**しましょう。

- AWSコンソール > CodePipeline > 左ペイン「設定」 > 「接続」 を選択してください。
  保留中となっているGitHubとの接続を選択後、「保留中の接続を更新」しGitHubと接続しましょう。

承認前



承認後



演習1-4　Step2

CI/CDパイプライン構築 – IAM

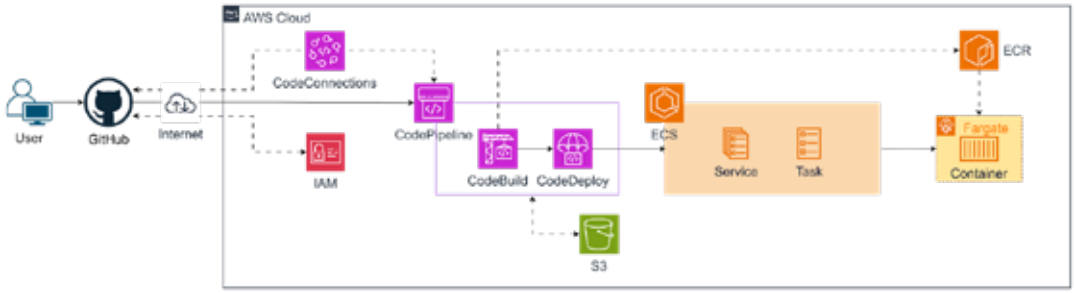
演習1-4　CI/CDパイプライン構築

Step2：IAM　演習概要

- 各サービスが他のAWSサービスを利用するためには、IAMロールとポリシーを割り当てる必要があります。
- Step1では、CI/CDパイプラインの流れを踏まえて、各サービスが正しく実行できるように、必要最小限のIAMポリシーを付与してください。

対象サービス

- CodeDeploy
- CodeBuild
- CodePipeline
- GitHub
- ECS
- Secrets Manager



演習1-4　CI/CDパイプライン構築

Step2：IAM　設計

- 各リソースに割り当てるIAMロールとポリシーは以下です。ポリシーは必要最低限にするのが理想です。

IAM設定値（1/2）

付与サービス	ロール名	AssumePrincipal	用途	権限（ポリシー内容）
CodeDeploy	CodeDeployServiceRole	codedeploy.amazonaws.com	ECS Blue/Green デプロイを実行	- AWSCodeDeployRoleForECS（AWS管理ポリシー）
CodeBuild	CodeBuildServiceRole	codebuild.amazonaws.com	CodeBuildがECRにpush / Logs出力 / S3のArtifacts参照 / kmsの暗号化・復号化を利用する	- logs:*（Logs出力） - ECR認証トークン取得 ecr:GetAuthorizationToken - ECR操作(対象Repoを限定 - ecrRepoArn) ecr:BatchCheckLayerAvailability/InitiateLayerUpload/ UploadLayerPart/CompleteLayerUpload/PutImage/ BatchGetImage/GetDownloadUrlForLayer - s3:GetObject/PutObject/GetBucketLocation/ListBucket - kms:Decrypt/Encrypt/GenerateDataKey*/DescribeKey
CodePipeline	CodePipelineServiceRole	codepipeline.amazonaws.com	Source/Build/Deploy をオーケストレーション S3のアーティファクト操作 CodeConnectionsの利用	- CodeBuild / CodeDeploy起動 codebuild:StartBuild codedeploy:CreateDeployment/Get*/ RegisterApplicationRevision - ロール譲歩 iam:PassRole (Build/Deployロールのみ) - CodeConnectionsの利用許可 codestar-connections:UseConnection - s3:GetObject/PutObject/GetObjectVersion/ListBucket - kms:Decrypt/Encrypt/GenerateDataKey*/DescribeKey

演習1-4　CI/CDパイプライン構築

Step2：IAM　設計

■ 各リソースに割り当てるIAMロールとポリシーは以下です。ポリシーは必要最低限にするのが理想です。

IAM設定値（2/2）

付与サービス	ロール名	AssumePrincipal	用途	権限（ポリシー内容）
GitHub (GitHub Actions)	GitHubOIDCRole	OIDC Provider (token.actions.githubusercontent.com)	GitHub ActionsからAWSを操作 （Pipeline起動など）	- パイプライン実行(対象Pipelineを指定) codepipeline:StartPipelineExecution
ECS タスク (Fargate)	EcsTaskExecutionRole	ecs-tasks.amazonaws.com	ECSのタスク起動時に ECRからのイメージPull / Logsを出力	- AWS 管理ポリシー service-role/AmazonECSTaskExecutionRolePolicy
ECS タスク (アプリ)	AppTaskRole	ecs-tasks.amazonaws.com	アプリの処理でAWS リソースにアクセスする際の実行ロール	- 特になし
Secrets Manager	ー （シークレットリソース）	ー	DB認証情報を格納するシークレットリソース。必要に応じて、AppTask、EcsTaskExecutionロールに権限を付与	- secretsmanager:GetSecretValue AppTaskRole / EcsTaskExecutionRole に付与

演習1-4　CI/CDパイプライン構築

Step2：IAM　コード作成

■ AWS CDKを利用して、**IAMスタックを作成するコード**を作成してください。

■ コード作成のポイント

bin/ecs-demo.ts

IAMスタック(IamStack)を呼び出す定義を追加しましょう

- importの追加
- IamStackを呼び出すブロックを追加

lib/iam-stack.ts

IAMスタックを呼び出す定義ファイルを作成しましょう

- IAMロールを作成後、ロールにポリシーを付与します
  ポリシーが不足した場合、呼び出したサービスを動かせずエラーとなります
- IAMロール名とリソースが同じであれば、複数ポリシーをまとめて定義できます
- 作成したIAMロールを他スタックで参照できるよう公開しましょう

ecs-demo/
├─ bin/ecs-demo.ts　★修正
├─ lib/ecs-demo-stack.ts
├─ lib/net-stack.ts
├─ lib/vpce-stack.ts
├─ lib/alb-stack.ts
├─ lib/ecr-stack.ts
├─ lib/ecs-stack.ts
├─ lib/rds-stack.ts
├─ lib/connection-stack.ts
├─ lib/iam-stack.ts　★新規作成
・・・

CDKディレクトリ構成

演習1-4　CI/CDパイプライン構築

Step2：IAM　コード作成

■ **bin/ecs-demo.tsを修正**してください。IamStackを呼び出すコード定義を追加します。

```
#!/usr/bin/env node
・・・略・・・
※importを追記してください。

・・・略・・・

## IamStack //ブロック追加
※追加してください。
```

スタック名	用途
App	CDK全体を表すクラス名
NetStack	VPC、サブネット、SGを定義
VpceStack	VPCエンドポイントを定義
AlbStack	ALB、WAFを定義
EcrStack	ECR(リポジトリ)を定義
EcsStack	ECS(クラスタ/タスク/サービス)を定義
RdsStack	RDS(MySQL)のインスタンスを定義
ConnectionStack	CodeConnections(GitHub接続)を定義
IamStack	IAMロールを定義

演習1-4　CI/CDパイプライン構築

Step2：IAM　コード作成

■ **bin/ecs-demo.ts**から、IamStackに渡すパラメータは以下の通りです。

IamStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	ー	デプロイ先アカウント/リージョン
ECRリポジトリ	ecrRepoName	'customer-info/app'	CodeBuild が push/pull する ECR リポジトリ名
パイプライン	pipelineName	'CustomerInfoPipeline'	CodePipeline名
GitHub	ghOwner ※	<GitHubアカウント>	GitHubのアカウント名
	ghRepo	'customer-info'	GitHubのリポジトリ名
CodeConnections	gitHubConnectionArn	conn.connectionArn	CodeConnections
RDS	appSecretArn	rds.appSecret.secretArn	アプリケーションユーザ(app_user)の認証情報

※ghOwner・・・GitHubアカウントを設定します。
ご自身のGitHubアカウントを設定してください。

演習1-4　CI/CDパイプライン構築

Step2：IAM　コード作成

■ **lib/iam-stack.ts**を作成し、**IamStack**を定義してください。

IAM作成　サンプルコード

```
// クラス等のimport
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as iam from 'aws-cdk-lib/aws-iam';

// IamStackに渡す外部パラメータ
export interface IamStackProps extends cdk.StackProps {
  ecrRepoName: string;
  pipelineName: string;
  ghOwner: string;
  ghRepo: string;
  githubOidcRoleArn?: string;
  gitHubConnectionArn?: string;
}
// クラスの宣言
export class IamStack extends cdk.Stack {
  public readonly codeDeployRole: iam.Role;  //CodeDeploy用ロールを作成・公開
    ※他ロールの定義を追加してください

  // ポリシーで利用するコンストラクタの定義
  constructor(scope: Construct, id: string, props: IamStackProps) {
    super(scope, id, props);

    const account = cdk.Stack.of(this).account;
    const region  = cdk.Stack.of(this).region;
    const ecrRepoArn  = `arn:aws:ecr:${region}:${account}:repository/${props.ecrRepoName}`;
    const pipelineArn = `arn:aws:codepipeline:${region}:${account}:${props.pipelineName}`;

      // 右に続く
```

```
    // CodeDeployRoleの作成
    this.codeDeployRole = new iam.Role(this, 'CodeDeployRole', {
      assumedBy: new iam.ServicePrincipal('codedeploy.amazonaws.com'),
      description: 'Allows CodeDeploy to perform ECS Blue/Green with ALB',
    });
    this.codeDeployRole.addManagedPolicy(
      iam.ManagedPolicy.fromAwsManagedPolicyName('AWSCodeDeployRoleForECS')
    );

    ※他ロールの作成およびポリシーを適用してください

    // Secrets Manager（アプリ用）
    if (props.appSecretArn) {
      const appSecret = secretsmanager.Secret.fromSecretCompleteArn(this, 'AppSecret',
props.appSecretArn);
      appSecret.grantRead(this.appTaskRole);
      appSecret.grantRead(this.ecsTaskExecutionRole);
    }

  // 出力（他スタックで参照する値を設定）
    new cdk.CfnOutput(this, 'CodeDeployRoleArn',      { value: this.codeDeployRole.roleArn });
    ※他ロールを出力項目に追加してください
  }
}
```

演習1-4　CI/CDパイプライン構築

Step2：IAM　コード実行

■ CDKから**IamStack**を実行してください。新しくロールが作成されていればStep2は完了です。

■ AWSコンソール > IAM > 左ペイン「ロール」 を選択します。
新しく7つのロールが作成されていることを確認しましょう。

作成されるロールは以下です。

- CodeBuildRole
- CodeDeployRole
- CodePipelineRole
- GitHubOIDCRole　※1
- ECSTaskExecutionRole
- AppTaskRole
- CustomAWSCDKOpenIdConnect ProviderCustomRes　※2



※1・・・GitHubOIDCRoleは名前を指定しています
※2・・・OIDCプロバイダー利用時に自動で
作成されるカスタムリソースです。
CDKコードでは定義していないロールです。

演習1-4　Step3

CI/CDパイプライン構築 - GitHub

演習1-4　CI/CDパイプライン構築

Step3：GitHub　演習概要

- アプリケーションコードとコンテナ化設定を含むGitHubリポジトリを作成し、CI/CDパイプラインを起動できる準備を整えてください。

- 今回はCloudShell での操作を前提としていますが、ローカル環境（VSCodeなど）やGitHubで直接リポジトリ作成およびコード編集しても問題ありません。ご自身のやりやすい方法でリポジトリを準備してください。
- CloudShellを用いたGitHub演習の流れは以下の通りです。
  GitHubとCloudShellで作業します。

1. GitHub PAT取得（GitHub作業）
2. GitHub 認証（CloudShell作業）
3. リポジトリ作成（CloudShell作業）
4. 変数設定（GitHub作業）
5. コード作成（CloudShell作業）
6. コード登録（CloudShell作業）



演習1-4　CI/CDパイプライン構築

Step3：GitHub　AWSとの接続

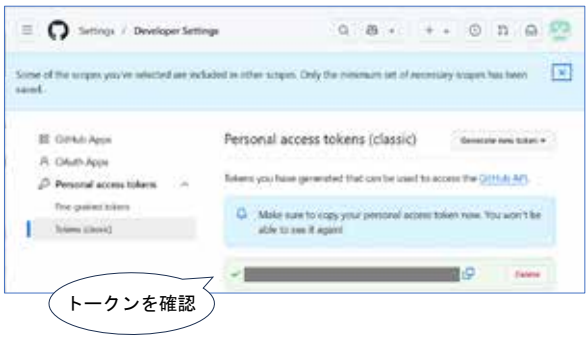
■ AWS CloudShellを利用して、GitHubのリポジトリを作成する手順を説明します。
作成するリポジトリは「**customer-info**」です。

1. GitHub PAT取得

- GitHubにログインし、トークン(PAT*)を取得・コピーしてください。
CloudShellからGitHubに接続する際に使用します。
*PAT：Personal Access Token

1. GitHubにログインし、Settingsを開きます。
2. Developer settings > Personal access tokens
> Tokens (classic) > Generate new token をクリックします。
3. トークンの名前（Note）と有効期限（Expiration）を設定し、
必要なスコープ（権限）を選択します。
スコープ：repo、workflow、read:org、admin:repo_hook
あたりを選択しておく
4. Generate token をクリックしてトークンを生成します。
5. **生成されたトークンは再表示できないため、**
必ずコピーして安全な場所に保管してください。

GitHub PAT生成画面



演習1-4　CI/CDパイプライン構築

Step3：GitHub　AWSとの接続

2. GitHub認証、リポジトリ作成

```
## 変数定義
ecs-demo $ GH_REPO=customer-info       # GitHubのリポジトリ名（今回はcustomer-info）
ecs-demo $ GH_OWNER=<your-org>             # GitHub ユーザー or Organizationを確認して入力
ecs-demo $ GH_TOKEN="ghp_xxxxxxxx“       # Personal Access Token

## GitHub CLI のインストール
ecs-demo $ sudo dnf install -y dnf-plugins-core
ecs-demo $ sudo dnf config-manager \
  --add-repo https://cli.github.com/packages/rpm/gh-cli.repo
ecs-demo $ sudo dnf install -y gh

## GitHubにログイン（PATで認証）
ecs-demo $ echo $GH_TOKEN | gh auth login --with-token
```

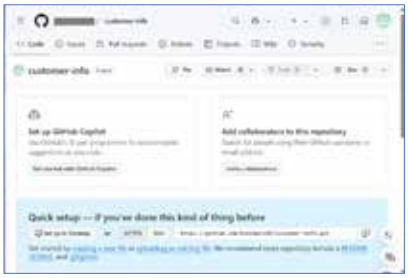
3. リポジトリ作成

```
## GitHub リポジトリ作成
ecs-demo $ gh repo create $GH_OWNER/$GH_REPO --public --confirm

## クローンして初期構成を作成
ecs-demo $ mkdir -p ~/github && cd ~/github
github $ git clone https://github.com/$GH_OWNER/$GH_REPO.git
github $ cd customer-info
```

GitHub customer-infoリポジトリ

GitHubにcustomer-infoリポジトリが
作成されます。現段階は「**空**」です。



演習1-4　CI/CDパイプライン構築

Step3：GitHub　AWSとの接続

4. GitHubへの変数登録

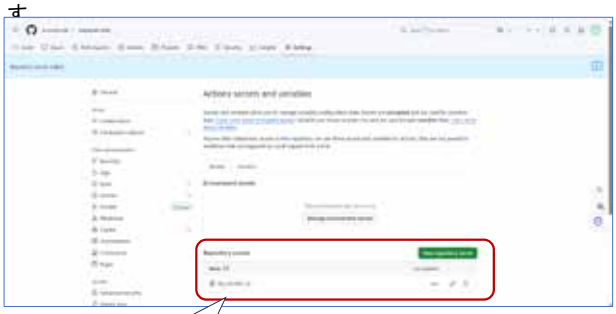
- customer-infoリポジトリのGitHub Action動作時に必要な変数をGitHubのSecretに登録します。

1. customer-infoリポジトリ > 上部メニューからSettingを選択
2. 左ペイン > Secrets and variables > Actions を選択
3. Secrets セクションの右上「New repository secret」を選択
4. 以下を入力して、「Add secret」を選択

項目	入力内容
Name	AWS_ACCOUNT_ID
Secret	<あなたの 12桁の AWS アカウント ID>

customer-info リポジトリのSecret

GitHubのcustomer-infoリポジトリにSecretが作成されます



作成したSecretが表示されます

演習1-4　CI/CDパイプライン構築

Step3：GitHub　リポジトリ設計

- GitHubに作成するリポジトリは**customer-info**とし、配下に各種ファイルを作成してください。
  Dockerfileでコンパイルできるように準備を進めてください。

- GitHubリポジトリの構成は以下の通りです。

```
customer-info/
├─ .github/
│   └─ workflows/
│       └─ ci.yml              # GitHub Actions
├─ app/                                  # アプリ本体
│   └─ app.py
├─ nginx/                # Nginx 設定
│   └─ nginx.conf
├─ Dockerfile
├─ requirements.txt
├─ .dockerignore                 # ビルド高速化用
├─ .gitignore                    # Python / Docker / VSCode
など
└─ README.md             # プロジェクト概要・ローカル実行方法
```

GitHub Actions(ci.yml)を定義してください。

Dockerfileでコンパイルできるように、アプリケーションやDockerfileなどを用意します。
実装するアプリケーションによって、多少構成が変わっても大丈夫です。

GitHubリポジトリ構成（customer-info）

演習1-4　CI/CDパイプライン構築

Step3：GitHub　リポジトリ設計

■ customer-infoリポジトリのmainブランチへのコードマージを契機に、CodePipelineが起動するようにしましょう。 GitHub Actions(ci.yml)を定義してください。

GitHub Actions(ci.yml)の役割

- customer-infoのmainブランチへのコードpushをトリガーとします。
  それ以外のリポジトリ、ブランチでは動作させません。
- GitHub OIDCトークンを発行し、AWSクレデンシャルを設定します。
- AWS CodePipelineを起動させます。
  つまりコードpushすれば、CI/CDパイプラインが起動する状態にします。

■ 前提

- ブランチ戦略は**GitHub Flow**（featureブランチとmainブランチのみ）を採用します。
- ci.ymlの役割はAWS OIDC連携し、CodePipelineを起動するだけとします。
  ※ci.ymlでもビルドやチェックはできますが、AWS CodePipeline / CodeBuildで同等の処理ができるため、
  ci.ymlでは実装しません。
- CodePipelineで設定するパイプライン名は「CustomerInfoPipeline」とします。

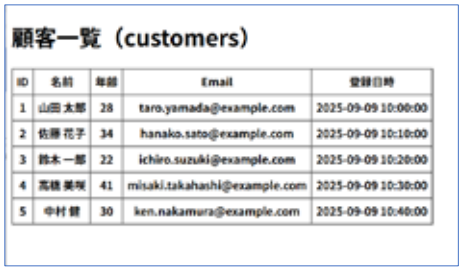
演習1-4　CI/CDパイプライン構築

Step3：GitHub　アプリケーション設計

■ Dockerfileでビルドできるシンプルなwebアプリケーションを準備してください。
準備したコードはcustomer-info配下にpushしてください。

アプリケーション要件

- 単一ファイルで動作するアプリケーションを実装してください
- 言語：　flask(python)　※別言語でもOK
- ブラウザからHTTP通信で閲覧
- ルーティング要件
  - /：ヘルスチェック用エンドポイント
    常に「200 OK」を返却
  - /customers：HTMLを返却（サンプル画像参照）
    演習1-3で構築したDB(mysql)から顧客情報を取得し、
    一覧を画面に表示


顧客一覧（customers）

ID	名前	年齢	Email	登録日時
1	山田 太郎	28	taro.yamada@example.com	2025-09-09 10:00:00
2	佐藤 花子	34	hanako.sato@example.com	2025-09-09 10:10:00
3	鈴木 一郎	22	ichiro.suzuki@example.com	2025-09-09 10:20:00
4	高橋 美咲	41	misaki.takahashi@example.com	2025-09-09 10:30:00
5	中村 健	30	ken.nakamura@example.com	2025-09-09 10:40:00

サンプル画像

■ 次ページから各ファイルのコードを掲載しますが、**独自にアプリケーションを用意いただいても大丈夫**です。
CloudShellで各ファイルを作成してGitHubにpushしましょう。

演習1-4　CI/CDパイプライン構築

Step3：GitHub　コード作成（参考）

5. GitHub/customer-infoのコード作成（1/5）

```
// customer-info内のディレクトリ作成
github $ mkdir -p app nginx .github/workflows

// 各種ファイル作成
// Dockerfile
github $ cat > Dockerfile <<'DOCKER'
FROM python:3.11-slim

RUN apt-get update && apt-get install -y nginx curl

WORKDIR /app
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

COPY app/app.py /app/app.py
COPY nginx/nginx.conf /etc/nginx/sites-enabled/default

# デフォルトのindex.htmlは削除
RUN rm -f /usr/share/nginx/html/index.html

CMD service nginx start && gunicorn -b 127.0.0.1:5000 app:app

DOCKER
```

```
// nginx/nginx.conf
github $ cat > nginx/nginx.conf <<'NGINX'
server {
    listen 80;

    location / {
        proxy_pass http://127.0.0.1:5000;
    }

    location /customers {
        proxy_pass http://127.0.0.1:5000/customers;
    }
}
NGINX

// .gitignore
github $ cat > .gitignore <<'GI'
__pycache__/
*.py[cod]
.venv/
.env
*.log
.vscode/
GI
```

演習1-4　CI/CDパイプライン構築

Step3：GitHub　コード作成（参考）

5. GitHub/customer-infoのコード作成（2/5）

```
// requirements.txt
github $ cat > requirements.txt <<'REQ'
Flask
gunicorn
pymysql
Boto3

REQ

// .dockerignore
github $ cat > .dockerignore <<'DI'
__pycache__
*.pyc
*.pyo
.git
.vscode
DI
```

```
// README.md
github $ cat > README.md <<'MD'
# customer-info

Flask + Nginx + Gunicorn のサンプルアプリ。

# ローカル起動
```bash
docker build -t customer-info:dev .
docker run -p 8080:80 \
  -e DB_HOST=localhost -e DB_USER=user -e
DB_PASSWORD=pass -e DB_NAME=sample \
  customer-info:dev
# http://localhost:8080 へアクセス
MD
```
```

演習1-4　CI/CDパイプライン構築

Step3：GitHub　コード作成（参考）

5. GitHub/customer-infoのコード作成（3/5）

```
// .github/workflows/ci.yml
github $ cat > .github/workflows/ci.yml <<'YAML'
name: CI (Kick CodePipeline)

on:
  push:
    branches: [ main ]
  pull_request:

permissions:
  id-token: write
  contents: read

jobs:
  kick-pipeline:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

// 右に続く
```

```
      - name: Configure AWS credentials via OIDC
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::${{ secrets.AWS_ACCOUNT_ID
}}:role/GitHubOIDCRole
          role-session-name: GitHubActionsSession
          aws-region: ap-northeast-1

  - name: Start CodePipeline
        run: aws codepipeline start-pipeline-execution --name
CustomerInfoPipeline

YAML
```

演習1-4　CI/CDパイプライン構築

Step3：GitHub　コード作成（参考）

5. GitHub/customer-infoのコード作成（4/5）

```
// app/app.py（4つに分割して表示）
github $ cat > app/app.py <<'PY'
from flask import Flask
import pymysql
import boto3
import json
import os

# PyMySQL を MySQLdb として使う
pymysql.install_as_MySQLdb()

app = Flask(__name__)

# ===== Secrets Manager から DB 認証情報を取得 =====
def load_db_credentials():
    secret_name = os.environ.get("DB_SECRET_NAME", "customer-
info-app-credentials")
    region_name = os.environ.get("AWS_REGION", "ap-northeast-1")
    client = boto3.client("secretsmanager",
region_name=region_name)
    secret_value = client.get_secret_value(SecretId=secret_name)
    return json.loads(secret_value["SecretString"])
```

```
# 起動時にキャッシュ
DB_CREDS = load_db_credentials()
DB_HOST  = os.environ.get("DB_HOST")
DB_NAME  = os.environ.get("DB_NAME", "customer_info")

def get_connection():
    return pymysql.connect(
        host=DB_HOST,
        user=DB_CREDS["username"],
        password=DB_CREDS["password"],
        database=DB_NAME,
        port=int(DB_CREDS.get("port", 3306)),
        charset="utf8mb4",
        cursorclass=pymysql.cursors.DictCursor,
        connect_timeout=5,
        autocommit=True,
    )

# === ヘルスチェック用: DB非依存 ===
@app.route("/")
def index():
    return "<h1>OK</h1><p>Service is up.</p>", 200
```

演習1-4　CI/CDパイプライン構築

Step3：GitHub　コード作成（参考）

5. GitHub/customer-infoのコード作成（5/5）

```
// app/app.pyの続き
# === DB参照 ===
@app.route("/customers")
def customers():
    try:
        with get_connection() as conn, conn.cursor() as cur:
            cur.execute("""
                SELECT id, name, age, email, created_at
                FROM customers
                ORDER BY id ASC
            """)
            rows = cur.fetchall()

        html = [
            "<!doctype html><meta charset='utf-8'>",
            "<h1>顧客一覧（customers）</h1>",
            "<table border='1' cellpadding='6' cellspacing='0'>",
            "<tr><th>ID</th><th>名前</th><th>年齢
</th><th>Email</th><th>登録日時</th></tr>"
        ]
```

```
        for r in rows:
            html.append(
                f"<tr><td>{r['id']}</td><td>{r['name']}</td><td>{r['age']}</td
>"
                f"<td>{r['email']}</td><td>{r['created_at']}</td></tr>"
            )
        html.append("</table>")
        return "\n".join(html), 200

    except Exception as e:
        return f"<pre>DB接続エラー：{e}</pre>", 500
```

演習1-4　CI/CDパイプライン構築

Step3：GitHub　コード作成（参考）

6. GitHub/customer-infoへのコード登録

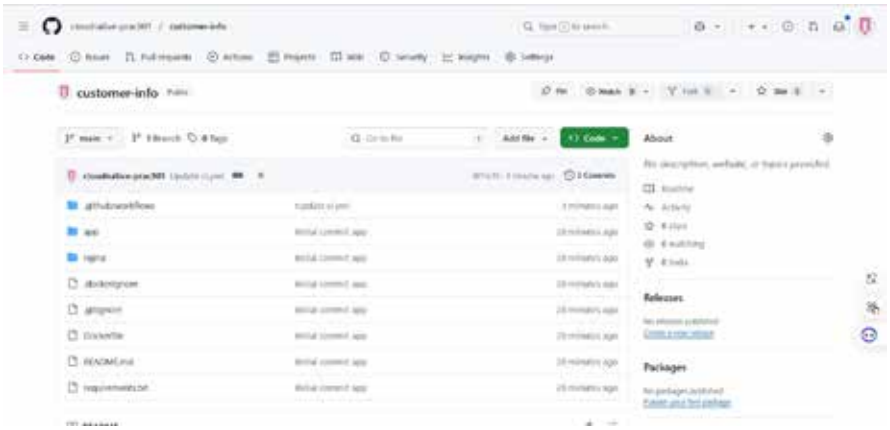
```
// GitHubにpush
github $ git config --global user.name "<Your Name>"
github $ git config --global user.email "<your.email@example.com>"

github $ git add .
github $ git commit -m "Initial commit app"
github $ git push -u origin main
  Username for 'https://github.com':                # GitHub ユーザー
  Password for 'https://xxx@github.com':            # PATを入力
```

演習1-4　CI/CDパイプライン構築

Step3：GitHub　動作確認

- git pushが完了すると、customer-infoリポジトリのmainブランチに、
  作成した各ファイルが反映されるのを確認してください。

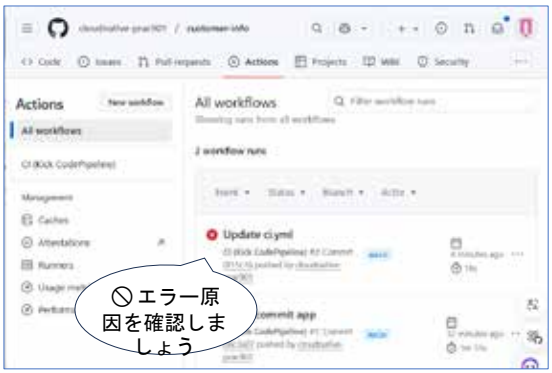


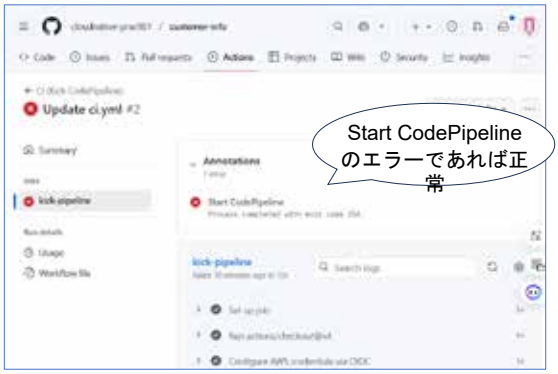
GitHub customer-infoリポジトリ

演習1-4　CI/CDパイプライン構築

Step3：GitHub　動作確認

- ci.yml登録後にファイルをpushするとGitHub Actionsが動作します。
  動作結果からAWSのOIDC認証が通ったことを確認できれば、Step3は完了です。
  - customer-infoリポジトリ > Actions　を選択し、**workflowの実行結果**を確認しましょう。
    - 「Configure AWS credentials via OIDC」の正常終了✅　＝　OIDCの認証が通った　ことになります。
    - 「Start CodePipeline」のエラー🚫は、CodePipelineを実装していないことが原因のため、正常です。





演習1-4　CI/CDパイプライン構築

合格判定基準

■ 演習1-4-①の合格判定基準は以下の通りです。

Step1（CodeConnections）

- CodePipeline の「接続」に GitHub接続が作成されているか
- 接続が「利用可能」状態になっているか（保留中の承認が済んでいるか）

Step2（IAM）

- IAM ロールが7種類作成されている
  （CodeBuildRole, CodeDeployRole, CodePipelineRole, GitHubOIDCRole, ECSTaskExecutionRole, AppTaskRole、 CustomAWSCDKOpenIdConnectProviderCustomRes）
- GitHubOIDCRoleのみ名前を固定して作成できている
- 各ロールに必要最小限のポリシーが正しく付与されている

Step3（GitHub）

- GitHubにcustomer-info リポジトリが作成されている
- アプリ・Dockerfile・ci.yml などの各ファイルが main ブランチに push されている
- GitHub Actions の workflow（ci.yml）が main ブランチ push をトリガーに起動することを確認できている
- Actions 実行結果で OIDC 認証が通っている（CodePipelineのスタートはエラーでよいです）

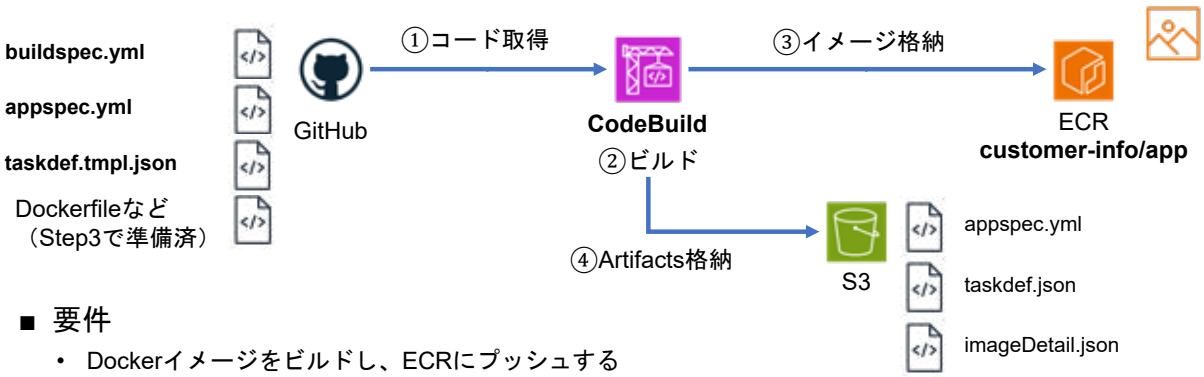
演習1-4 Step4

CI/CDパイプライン構築 - CodeBuild

演習1-4　CI/CDパイプライン構築

Step4：CodeBuild　演習概要

■ CI/CDパイプラインを構築するにあたり、最初にCIパイプライン部分を準備します。
CodeBuildプロジェクトおよびbuildspec.ymlを実装してください。



■ 要件
- Dockerイメージをビルドし、ECRにプッシュする
- CodeBuildは、GitHubのbuildspec.ymlを読み込んで、ビルドを行う
- ECSはBlue/Greenデプロイを前提とし、CodeDeployに連携するため、buildspec.ymlの成果物(Artifacts)としてimageDetail.jsonを生成する
- ECSに渡すappspec.yml、taskdef.tmpl.jsonはテンプレートとして実装する

演習1-4　CI/CDパイプライン構築

Step4：CodeBuild　設計

■ CodeBuildが実行するビルド手順を**buildspec.yml等に定義し**、GitHubに配置しましょう。
GitHub(customer-info)に配置することで、CodeBuild起動時に自動で読み込まれます。

■ buildspec.yml　要件

pre_build
認証・準備作業を実行。ECRログインや環境変数の設定など、ビルド前の準備をします。
- AWS認証を済ませ、ECRにログイン
- タグを生成、イメージURI生成

Build
Dockerイメージのビルド、テストの実行などを行います。
- docker buildの実行

post_build
ビルド後の成果物の処理を実行。ECRへのプッシュ、メタデータファイル生成などを行います。
- ECRにイメージプッシュ
- イメージURIをimageDetail.jsonに出力
- taskdef.tmpl.jsonの値を埋めて、taskdef.jsonを出力

```
customer-info/
├── .github/
│   └── workflows/
│       └── ci.yml                  # GitHub Actions
├── app/                                    # アプリ本体
│   └── app.py
├── nginx/                  # Nginx 設定
│   └── nginx.conf
├── Dockerfile
├── requirements.txt
├── .dockerignore                   # ビルド高速化用
├── .gitignore                      # Python / Docker / VSCode など
├── README.md                       # プロジェクト概要・ローカル実行方法
├── buildspec.yml                   # CodeBuildが参照するビルド設計書
└── deploy/
    └── ecs/
        ├── appspec.yml             # CodeDeployが参照するアプリ仕様
        └── taskdef.tmpl.json       # CodeDeployが参照するタスク定義
```

演習1-4　CI/CDパイプライン構築

Step4：CodeBuild　コード作成

■ GitHubのcustomer-info配下に、buildspec.ymlを定義してください。

buildspec.yml作成　サンプルコード（1/2）

```
version: 0.2
env:
  variables:
    ECR_REPO: "customer-info/app"          # ECR リポジトリ名
    IAM_STACK_NAME: "IamStack"             # IamStack の実スタック名
    RDS_STACK_NAME: "RdsStack"                     # RdsStack の実スタック名
    DEPLOY_DIR: "deploy/ecs"                       # リポ直下の /deploy/ecs に変更
  git-credential-helper: no
phases:
  # pre_build（ECRログイン、タグ生成などを実施）
  pre_build:
    commands:
      - set -eu
      - ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)
      - REGION="${AWS_DEFAULT_REGION}"
      - echo "[pre_build] ACCOUNT_ID=${ACCOUNT_ID} REGION=${REGION}"
      # ECR login
      - aws ecr get-login-password --region "$REGION" | docker login --username AWS --
password-stdin "${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com"

      # Pipelineから渡される変数でロール/SecretARN を解決
      - EXEC_ROLE_ARN="${EXEC_ROLE_ARN:-}"
      - TASK_ROLE_ARN="${TASK_ROLE_ARN:-}"
      - DB_SECRET_ARN="${DB_SECRET_ARN:-}"
      - DB_HOST="${DB_HOST:-}"
```

```
      # イメージタグ生成（main 固定ポリシー）
      - TS=$(date +%Y%m%d%H%M%S)
      - IMAGE_TAG="main-${TS}-b${CODEBUILD_BUILD_NUMBER:-0}"
      - test -n "${ECR_REPO}" || { echo "ECR_REPO is empty"; exit 1; }
      -
IMAGE_URI="${ACCOUNT_ID}.dkr.ecr.${REGION}.amazonaws.com/${ECR_REPO}:${IMAGE_
TAG}"
      - echo "[pre_build] IMAGE_URI=${IMAGE_URI}"
      # 配置ファイルの存在チェック
      - test -f "${DEPLOY_DIR}/appspec.yml" || { echo "not found:${DEPLOY_DIR}/appspec.yml";
exit 1; }
      - test -f "${DEPLOY_DIR}/taskdef.tmpl.json" || { echo "not
found:${DEPLOY_DIR}/taskdef.tmpl.json"; exit 1; }

  # build（docker buildなど）
  build:
    commands:
      - set -eu
      - CTX="${BUILD_CONTEXT:-.}"
      - DF="${DOCKERFILE_PATH:-${CTX}/Dockerfile}"
      - test -f "${DF}" || { echo "Dockerfile not found:${DF}"; exit 1; }
      - echo "[build] docker build ${DF} -> ${IMAGE_URI}"
      - docker build --pull -f "${DF}" -t "${IMAGE_URI}" "${CTX}"
```

演習1-4　CI/CDパイプライン構築

Step4：CodeBuild　コード作成

■ GitHubのcustomer-info配下に、buildspec.ymlを定義してください。

buildspec.yml作成　サンプルコード（2/2）

```
  # post_build（ECRプッシュ、imageDetail.jsonの出力）
  post_build:
    commands:
      - set -eu
      - echo "[post_build] push ${IMAGE_URI}"
      - docker push "${IMAGE_URI}"
      # CodeDeploy(ECS Blue/Green) 用の imageDetail.json（DEPLOY_DIR に出力）
      - printf '{"ImageURI":"%s"}\n' "${IMAGE_URI}" > imageDetail.json
      - echo "imageDetail.json:" && cat imageDetail.json

      # jq/gettext インストール（yum/apt 両対応）
      - |
        if command -v yum >/dev/null 2>&1; then
          sudo yum -y install jq gettext >/dev/null
        else
          sudo apt-get update -y >/dev/null
          sudo apt-get install -y jq gettext-base >/dev/null
        fi

      # taskdef.tmpl.json → taskdef.json（envsubst でプレースホルダ埋め込み）
      - export EXEC_ROLE_ARN TASK_ROLE_ARN REGION DB_SECRET_ARN DB_HOST
      - envsubst < "${DEPLOY_DIR}/taskdef.tmpl.json" > "${DEPLOY_DIR}/taskdef.json"
```

```
      # DB未導入（CFn Outputs が None/空）の場合は secrets を削除して登録
      - |
        if [ -z "${DB_SECRET_ARN}" ] || [ "${DB_SECRET_ARN}" = "None" ]; then
          echo "[post_build] DB_SECRET_ARN is empty => remove secrets from taskdef"
          jq '(.containerDefinitions[0]) |= del(.secrets)' "${DEPLOY_DIR}/taskdef.json" >
"${DEPLOY_DIR}/taskdef.tmp" && mv "${DEPLOY_DIR}/taskdef.tmp"
"${DEPLOY_DIR}/taskdef.json"
        fi
      - echo "[done] artifacts in ${DEPLOY_DIR}"

# アーティファクト出力
artifacts:
  files:
    - deploy/ecs/appspec.yml
    - deploy/ecs/taskdef.json
    - imageDetail.json
  discard-paths: no
```

演習1-4　CI/CDパイプライン構築

Step4：CodeBuild　コード作成

- appspec.yml は CodeDeploy(ECS) がデプロイ時に参照するテンプレートです。
  GitHubの/deploy/ecs配下に、appspec.yamlを定義してください。

appspec.yml　サンプルコード

```
version: 0.0
Resources:
 - TargetService:
     Type: AWS::ECS::Service
     Properties:
       TaskDefinition: <TASK_DEFINITION>  # CodeDeploy が登録したTaskDef ARN
を差込
       LoadBalancerInfo:
         ContainerName: "app"    # ECSサービスのコンテナ名と一致させる
         ContainerPort: 80                  # ECSサービスのコンテナポートと一致させ
る
```

演習1-4　CI/CDパイプライン構築

Step4：CodeBuild　コード作成

- taskdef.tmpl.json はCodeDeployがECSタスク定義を設定するためのテンプレートです。
  GitHubの/deploy/ecs配下に、taskdef.tmpl.jsonを定義してください。
  ※buildspec.yml実行時、このテンプレートを参照して、taskdef.jsonを出力します。

taskdef.tmpl.json　サンプルコード

```
{
  "family": "customer-info-task",
  "networkMode": "awsvpc",
  "requiresCompatibilities": ["FARGATE"],
  "cpu": "256",
  "memory": "512",
  "executionRoleArn": "${EXEC_ROLE_ARN}",
  "taskRoleArn":      "${TASK_ROLE_ARN}",
  "containerDefinitions": [
    {
      "name": "app",
      "image": "<IMAGE1_NAME>",
      "essential": true,
      "portMappings": [{ "containerPort": 80, "protocol": "tcp" }],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "/ecs/customer-info",
          "awslogs-region": "${REGION}",
          "awslogs-stream-prefix": "customer-info"
        }
      },
      "environment": [
        { "name": "DB_HOST", "value": "${DB_HOST}"}
      ],
      "healthCheck": {
          "command": ["CMD-SHELL","curl -f http://localhost:80/ || exit 1"],
          "interval": 30, "timeout": 5, "retries": 3, "startPeriod": 15
        },
        "secrets": [
          { "name": "DB_USER", "valueFrom": "${DB_SECRET_ARN}:username::" },
          { "name": "DB_PASS", "valueFrom": "${DB_SECRET_ARN}:password::" }
        ]
      }
    ],
    "runtimePlatform": { "operatingSystemFamily": "LINUX", "cpuArchitecture": "X86_64" }
}
```

演習1-4　CI/CDパイプライン構築

Step4：CodeBuild　コード作成

■ AWS CDKを利用して、**CodeBuildを作成するコード**を作成してください。

■ コード作成のポイント

bin/ecs-demo.ts

CodeBuildスタック(BuildStack)を呼び出す定義を追加しましょう

- importの追加
- BuildStackStackを呼び出すブロックを追加

lib/build-stack.ts

Buildスタックを呼び出す定義ファイルを作成しましょう。

- Docker build/pushのために、privileged: trueを付与
- ビルドする際にGitHubのbuildspec.yml を読み込むように指定
- IamStackで生成した実行ロールを取り込む

```
ecs-demo/
├─ bin/ecs-demo.ts ★修正
├─ lib/ecs-demo-stack.ts
├─ lib/net-stack.ts
├─ lib/vpce-stack.ts
├─ lib/alb-stack.ts
├─ lib/ecr-stack.ts
├─ lib/ecs-stack.ts
├─ lib/rds-stack.ts
├─ lib/connection-stack.ts
├─ lib/iam-stack.ts
├─ lib/build-stack.ts　★新規作成
・・・
```

CDKディレクトリ構成

演習1-4　CI/CDパイプライン構築

Step4：CodeBuild　コード作成

■ **bin/ecs-demo.tsを修正**してください。BuildStackを呼び出すコード定義を追加します。

```
#!/usr/bin/env node
・・・略・・・
※importを追記してください。

・・・略・・・

// CodeBuild ★ブロック追加
※追加してください。
```

スタック名	用途
App	CDK全体を表すクラス名
NetStack	VPC、サブネット、SGを定義
VpceStack	VPCエンドポイントを定義
AlbStack	ALB、WAFを定義
EcrStack	ECR(リポジトリ)を定義
RdsStack	RDS(MySQL)のインスタンスを定義
EcsStack	ECS(クラスタ/タスク/サービス)を定義
ConnectionStack	CodeConnections(GitHub接続)を定義
IamStack	IAMロールを定義
BuildStack	CodeBuildプロジェクトを定義

BuildStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	ー	デプロイ先アカウント/リージョン
IAM	codeBuildRoleArn	iam.codeBuildRole.roleArn	CodeBuild実行ロールARN
ECR	ecrRepoName	'customer-info/app'	ECRリポジトリ名

演習1-4　CI/CDパイプライン構築

Step4：CodeBuild　コード作成

- **lib/build-stack.ts**を作成し、**BuildStack**を定義してください。

BuildStack作成　サンプルコード

```
// インポート
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as codebuild from 'aws-cdk-lib/aws-codebuild';
import * as iam from 'aws-cdk-lib/aws-iam';
// インタフェース定義
export interface BuildStackProps extends cdk.StackProps {
  codeBuildRoleArn: string;        // CodeBuildRoleのARN
  ecrRepoName?: string;            // ECRリポジトリ名
  buildSpecFile?: string;          // buildspec.ymlのパス
}
// クラス宣言
export class BuildStack extends cdk.Stack {
  public readonly project: codebuild.IProject;
  public readonly projectName: string;
  constructor(scope: Construct, id: string, props: BuildStackProps) {
    super(scope, id, props);
    // デプロイ先のアカウント、リージョンのセット
    const account = cdk.Stack.of(this).account;
    const region  = cdk.Stack.of(this).region;
    // ECRリポジトリの設定
    const repoName    = props.ecrRepoName ?? 'customer-info/app';
    const ecrRegistry = `${account}.dkr.ecr.${region}.amazonaws.com`;
    // CodeBuildRoleのインポート
    const role = iam.Role.fromRoleArn(this, 'ImportedCodeBuildRole', props.codeBuildRoleArn, {
      mutable: false,
    });
```

```
    // CodeBuildプロジェクトの作成（CodePipelineから起動される）
    const project = new codebuild.PipelineProject(this, 'Project', {
      projectName: 'customer-info-app',
      role,
      environment: {
        buildImage: codebuild.LinuxBuildImage.STANDARD_7_0,
        privileged: true, // Docker build/pushを行うための必須設定
      },
      // リポジトリ直下の buildspec.yml を利用
      buildSpec: codebuild.BuildSpec.fromSourceFilename(props.buildSpecFile ?? 'buildspec.yml'),
      // buildspec から参照する環境変数
      environmentVariables: {
        ECR_REPO:       { value: repoName },  // ECRのリポジトリ名
        AWS_ACCOUNT_ID: { value: account },
        AWS_REGION:     { value: region  },
      },
    });
    this.project = project;
    this.projectName = project.projectName;
    // 出力
    new cdk.CfnOutput(this, 'CodeBuildProjectName', { value: project.projectName });
    new cdk.CfnOutput(this, 'CodeBuildProjectArn',  { value: project.projectArn  });
    new cdk.CfnOutput(this, 'EcrRepoName',          { value: repoName            });
    new cdk.CfnOutput(this, 'EcrRegistry',          { value: ecrRegistry         });
  }
}
```

演習1-4　CI/CDパイプライン構築

Step4：CodeBuild　動作確認

- BuildStackの動作確認をします。CodeBuildにcustomer-info-appプロジェクトが作成されていることを確認して、Step4完了です。
  - CodeBuild > ビルドプロジェクト > customer-info-app　で確認できます。
  - ビルドプロジェクト自体の動作確認は、Step6 Pipeline実装時に合わせて確認します。



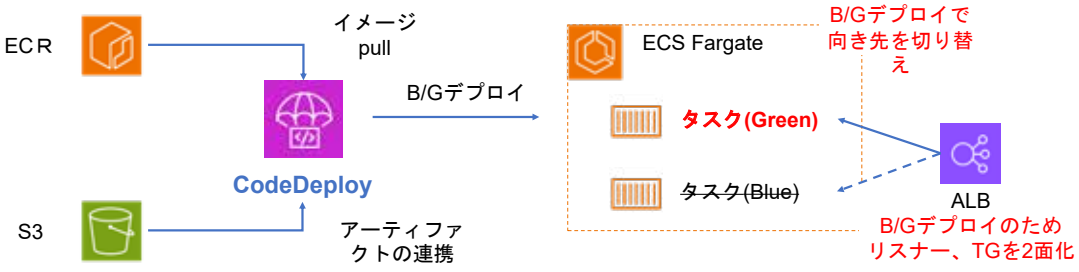
演習1-4 Step5

CI/CDパイプライン構築 - CodeDeploy

演習1-4　CI/CDパイプライン構築

Step5：CodeDeploy　演習概要

■ 次にCDパイプライン部分を準備します。**CodeDeploy**を**実装**してください。
ECSスタックデプロイしたタスクを、B/Gデプロイで入れ替えるのを実現します。



■ 要件

- ECRのリポジトリcustomer-info/appからイメージを取得し、ECS Fargateをデプロイする
- ECS FargateはBlue/Greenデプロイを前提とし、既存のタスクを入れ替えるリリース方式とする
- B/Gデプロイ失敗時はロールバックすること

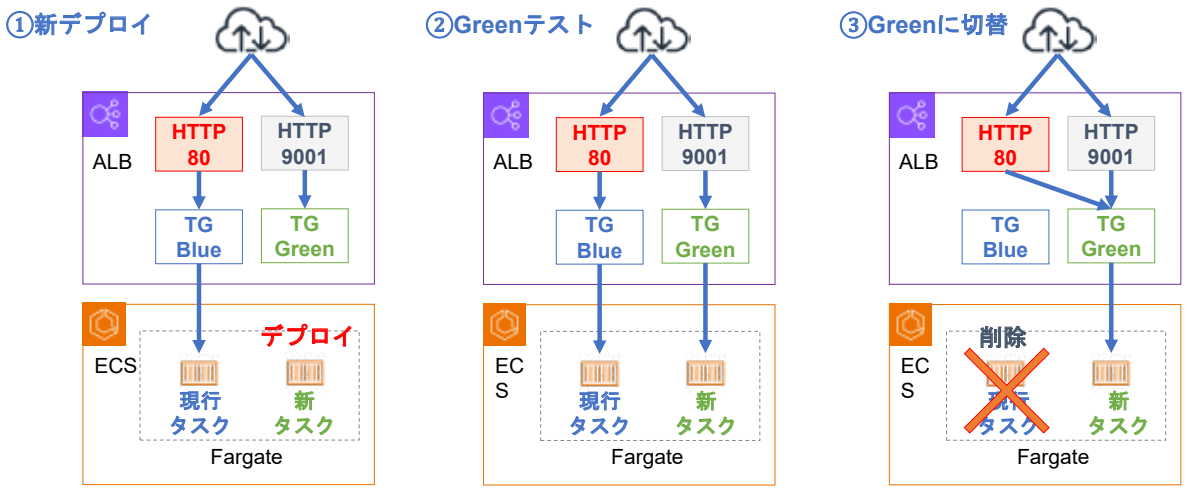
その他

- Blue/Greenデプロイを実現するため、ALB(リスナー、tgなど含む)を2面化すること
- CodeDeployはCodePipelineから起動させる（Step6で実装します）

演習1-4　CI/CDパイプライン構築

Step5：CodeDeploy　演習概要

- CodeDeployはB/Gデプロイ（Blue/Greenデプロイメント）を実現することができます。
  タスク切り替えのため、ALBのリスナー、ターゲットグループを2つに増やします。



補足）　HTTP:80 ⇒ prodリスナー、　HTTP:9001 ⇒ testリスナー

演習1-4　CI/CDパイプライン構築

Step5：CodeDeploy　コード作成

- AWS CDKを利用して、**CodeDeployを作成するコード**を作成してください。
- コード作成のポイント

bin/ecs-demo.ts

CodeDeployスタック(DeployStack)を呼び出す定義を追加しましょう

lib/deploy-stack.ts

Deployスタックを呼び出す定義ファイルを作成しましょう。

- ビルドする際にGitHubのbuildspec.yml を読み込むように指定
- IamStackで生成した実行ロールを取り込む

lib/alb-stack.ts

Albを2面化になるよう定義ファイルを修正してください。

- tgとリスナーを追加、追加するテストリスナーのポートは9001とします

lib/net-stack.ts

AlbSgのインバウンドにテストリスナのポート（9001）を追加してください。

```
ecs-demo/
├─ bin/ecs-demo.ts ★修正
├─ lib/ecs-demo-stack.ts
├─ lib/net-stack.ts　★修正
├─ lib/vpce-stack.ts
├─ lib/alb-stack.ts　★修正
├─ lib/ecr-stack.ts
├─ lib/ecs-stack.ts
├─ lib/rds-stack.ts
├─ lib/connection-stack.ts
├─ lib/iam-stack.ts
├─ lib/build-stack.ts
├─ lib/deploy-stack.ts　★新規作成
・・・
```

CDKディレクトリ構成

演習1-4　CI/CDパイプライン構築

Step5：CodeDeploy　コード作成

- **bin/ecs-demo.tsを修正**してください。DeployStack、EcsStackを呼び出すコード定義を追加します。

```
#!/usr/bin/env node
・・・略・・・
※importを追記してください。

・・・略・・・

// ECS / Fargate
※ALBのTG変数名に合わせて、targetGroupの変数を変更してください。

・・・略・・・

// CodeDeploy ★セクション追加
※追加してください。
```

スタック名	用途
App	CDK全体を表すクラス名
NetStack	VPC、サブネット、SGを定義
VpceStack	VPCエンドポイントを定義
AlbStack	ALB、WAFを定義
EcrStack	ECR(リポジトリ)を定義
RdsStack	RDS(MySQL)のインスタンスを定義
EcsStack	ECS(クラスタ/タスク/サービス)を定義
ConnectionStack	CodeConnections(GitHub接続)を定義
IamStack	IAMロールを定義
BuildStack	CodeBuildプロジェクトを定義
DeployStack	CodeDeployアプリおよびデプロイメントを定義

EcsStackに渡すパラメータ（変更点のみ）

※ALBの定義変更に伴い、パラメータの値を変更します。

項目	パラメータ	値（変更前）	値（変更後）
ALB	targetGroup	alb.targetGroup	alb.tgBlue

演習1-4　CI/CDパイプライン構築

Step5：CodeDeploy　コード作成

- bin/ecs-demo.tsを修正してください。
  DeploymentStackに渡すパラメータは以下の通りです。

DeployStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	ー	デプロイ先アカウント/リージョン
ECS	clusterName	ecs.cluster.clusterName	ECSクラスター名
	serviceName	ecs.service.serviceName	ECSサービス名
ALB	prodListenerArn	alb.listenerProd.listenerArn	ALBリスナー（本番リスナー）
	testListenerArn	alb.listenerTest.listenerArn	ALBリスナー（テストリスナー）
	tgBlueName	alb.tgBlue.targetGroupName	ALBターゲットグループ（Blue側）
	tgGreenName	alb.tgGreen.targetGroupName	ALBターゲットグループ（Green側）
IAM	codeDeployRoleArn	iam.codeDeployRole.roleArn	IAMロールの引き渡し
CodeDeploy	applicationName	'CustomerInfoEcsApp'	CodeDeployアプリケーション名
	deploymentGroupName	'CustomerInfoDG'	CodeDeployデプロイメントグループ名

演習1-4　CI/CDパイプライン構築

Step5：CodeDeploy　コード作成

- lib/alb-stack.tsを修正してください。
  targetGroup(tg)で定義していた部分をtgBlue(tgBlue)、tgGreen(tgGreen)に修正し、
  ECSやCodeDeployに引き渡してください。

AlbStackの修正パラメータ

項目	修正前	修正後
ターゲットグループ	targetGroup	tgBlue / tgGreen
変数	tg	tgBlue / tgGreen
リスナー:ポート	listener:80	listenerProd:80 / listenerTest:9001
リスナーのデフォルトTG	listener=[tg]	listenerProd=[tgBlue] / listenerTest=[tgGreen]
出力	AlbTgArn: tg.targetGroupArn	ProdListenerArn: this.listenerProd.listenerArn TestListenerArn: this.listenerTest.listenerArn TgBlueName: this.tgBlue.targetGroupName TgGreenName: this.tgGreen.targetGroupName TgBlueArn: this.tgBlue.targetGroupArn TgGreenArn: this.tgGreen.targetGroupArn

- lib/net-stack.tsを修正してください。
  alb-publicに適用するSGにテストリスナー用ポート（HTTP:9001）を追加するのみです。

演習1-4　CI/CDパイプライン構築

Step5：CodeDeploy　コード作成

- lib/deploy-stack.tsを作成し、DeployStackを定義してください。

DeployStack作成　サンプルコード（1/2）

```
// クラス等のimport
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as codedeploy from 'aws-cdk-lib/aws-codedeploy';
import * as iam from 'aws-cdk-lib/aws-iam';

// BuildStackに渡す外部パラメータ
export interface DeployStackProps extends cdk.StackProps {
  clusterName: string;
  serviceName: string;
  prodListenerArn: string;
  testListenerArn: string;
  tgBlueName: string;
  tgGreenName: string;
  codeDeployRoleArn: string;
  applicationName: string;
  deploymentGroupName: string;
}

// クラスの宣言
export class DeployStack extends cdk.Stack {
  public readonly application: codedeploy.CfnApplication;
  public readonly deploymentGroup: codedeploy.CfnDeploymentGroup;

  // コンストラクタの定義
  constructor(scope: Construct, id: string, props: DeployStackProps) {
    super(scope, id, props);

// 右に続く
```

```
// CodeDeployのアプリケーション作成
  this.application = new codedeploy.CfnApplication(this, 'EcsApplication', {
    applicationName: props.applicationName,
    computePlatform: 'ECS',
  });
  // サービスロールの読み込み
  const serviceRole = iam.Role.fromRoleArn(this, 'CdServiceRole', props.codeDeployRoleArn, {
    mutable: false,
  });
  // デプロイメントグループの定義
  this.deploymentGroup = new codedeploy.CfnDeploymentGroup(this, 'EcsDeploymentGroup', {
    applicationName: this.application.ref,
    deploymentGroupName: props.deploymentGroupName,
    serviceRoleArn: serviceRole.roleArn,

    ecsServices: [{ clusterName: props.clusterName, serviceName: props.serviceName }],

  // ALB・リスナー・TGの設定
  loadBalancerInfo: {
    targetGroupPairInfoList: [
      {
        prodTrafficRoute: { listenerArns: [props.prodListenerArn] },
        testTrafficRoute: { listenerArns: [props.testListenerArn] },
        targetGroups: [{ name: props.tgBlueName }, { name: props.tgGreenName }],
      },
    ],
  },
// 次ページに続く
```

演習1-4　CI/CDパイプライン構築

Step5：CodeDeploy　コード作成

■ **lib/deploy-stack.ts**を作成し、**DeployStack**を定義してください。

DeployStack作成　サンプルコード（2/2）

```
// デプロイメント方式（Blue/Greenデプロイメント）
    deploymentStyle: { deploymentType: 'BLUE_GREEN', deploymentOption:
'WITH_TRAFFIC_CONTROL' },
    blueGreenDeploymentConfiguration: {
      terminateBlueInstancesOnDeploymentSuccess: { action: 'TERMINATE',
terminationWaitTimeInMinutes: 5 },
      deploymentReadyOption: { actionOnTimeout: 'CONTINUE_DEPLOYMENT' },
    },

    // 自動ロールバック設定
    autoRollbackConfiguration: {
      enabled: true,
      events: ['DEPLOYMENT_FAILURE', 'DEPLOYMENT_STOP_ON_ALARM',
'DEPLOYMENT_STOP_ON_REQUEST'],
    },
  });

  // 出力
  this.deploymentGroup.addDependency(this.application);
  new cdk.CfnOutput(this, 'EcsAppName', { value: this.application.applicationName! });
  new cdk.CfnOutput(this, 'EcsDeploymentGroupName', { value: props.deploymentGroupName
});
  }
}
```

演習1-4　CI/CDパイプライン構築

Step5：CodeDeploy　動作確認

■ DeployStackの前にNetStack、AlbStackをデプロイして、修正内容を確認しましょう。
AlbSgにインバウンドルール(tcp:9001)、ALBにリスナー、TGが2つずつ存在していれば大丈夫です。

■ SG（AlbSg）

- VPCダッシュボード > セキュリティグループ > AlbSg
- インバウンドルールにtcp:9001が追加されていることを確認



■ ALB（CustomerInfoAlb）

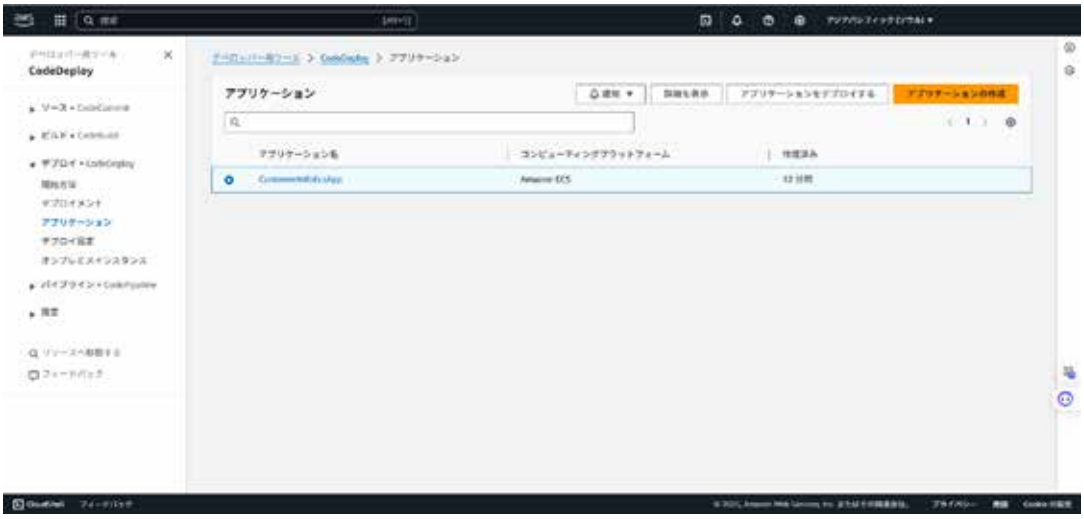
- EC2 > ロードバランサー
- CustomerInfoAlbのリソースマップを開き、HTTP:80 → ターゲット:AlbSta-TgBlu → ターゲット2つが接続されていることを確認



演習1-4　CI/CDパイプライン構築

Step5：CodeDeploy　動作確認

- 続いてDeployStackの動作確認をします。
  CodeDeployのアプリケーションにCustomerInfoEcsStackが登録されていれば、Step5完了です。



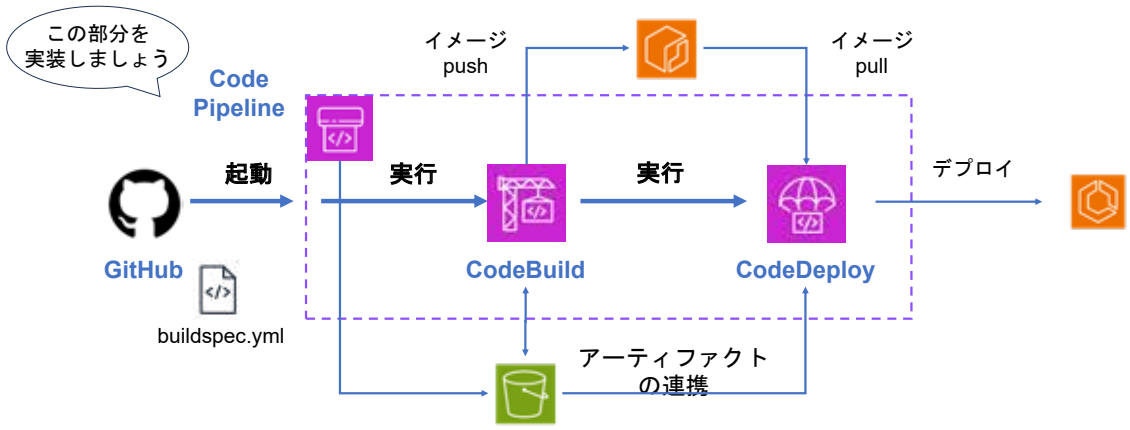
演習1-4 Step6

CI/CDパイプライン構築 - CodePipeline

演習1-4　CI/CDパイプライン構築

Step6：CodePipeline　演習概要

■ 最後にCI/CDパイプライン部分を統合していきます。**CodePipelineを実装**してください。
Step1～5で実装してきたものを組み合わせてCI/CDパイプラインを完成させましょう。



演習1-4　CI/CDパイプライン構築

Step6：CodePipeline　演習概要

■ CodePipelineの要件は以下の通りです。

- gitHubのソースマージをトリガーとして、CodePipelineが起動する（Step3 ci.ymlで実装済）
- CodePipelineは「Source」「Build」「Deploy」の3ステージで構成する
  - Sourceステージ：　GitHubリポジトリ(customer-info)からのソース取得
  - Buildステージ：　CodeBuildプロジェクトを起動し、ECRにイメージを格納（Step4で実装済）
  - Deployステージ：　CodeDeployのECSアプリケーション/デプロイメントを利用し、B/Gデプロイを実現（Step5で実装済）



- CodeDeployには、以下3ファイルを読み込ませて、B/Gデプロイをさせる
  - GitHub上のテンプレート2種（appspec.yml、taskdef.json）
  - CodeBuildで作成したimageDetail.json

- S3バケットは一意の名前で作成し、Codeシリーズ内で、Artifactsを連携する
  - S3バケットはCDKの自動的に任せても良いです
  - S3バケットはグローバルで一意の名前にする必要があり、他アカウント含め同名は重複不可なため注意してください

演習1-4　CI/CDパイプライン構築

Step6：CodePipeline　コード作成

- AWS CDKを利用して、**CodePipelineを作成するコード**を作成してください。
- コード作成のポイント

bin/ecs-demo.ts

Pipelineスタック(PipelineStack)を呼び出す定義を追加しましょう

lib/pipeline-stack.ts

Pipelineスタックを呼び出す定義ファイルを作成しましょう。

- ビルドする際にGitHubのbuildspec.yml を読み込むように指定
- IamStackで生成した実行ロールを取り込む

```
ecs-demo/
├─ bin/ecs-demo.ts ★修正
├─ lib/ecs-demo-stack.ts
├─ lib/net-stack.ts
├─ lib/vpce-stack.ts
├─ lib/alb-stack.ts
├─ lib/ecr-stack.ts
├─ lib/ecs-stack.ts
├─ lib/rds-stack.ts
├─ lib/connection-stack.ts
├─ lib/iam-stack.ts
├─ lib/build-stack.ts
├─ lib/deploy-stack.ts
├─ lib/pipeline-stack.ts　★新規作成
・・・
```

CDKディレクトリ構成

演習1-4　CI/CDパイプライン構築

Step6：CodePipeline　コード作成

- **bin/ecs-demo.tsを修正**してください。PipelineStackを呼び出すコード定義を追加します。

```
#!/usr/bin/env node
・・・略・・・
※importを追記してください。

・・・略・・・

// CodePipeline ★セクション追加
※追加してください。
```

スタック名	用途
App	CDK全体を表すクラス名
NetStack	VPC、サブネット、SGを定義
VpceStack	VPCエンドポイントを定義
AlbStack	ALB、WAFを定義
EcrStack	ECR(リポジトリ)を定義
RdsStack	RDS(MySQL)のインスタンスを定義
EcsStack	ECS(クラスタ/タスク/サービス)を定義
ConnectionStack	CodeConnections(GitHub接続)を定義
IamStack	IAMロールを定義
BuildStack	CodeBuildプロジェクトを定義
DeployStack	CodeDeployアプリおよびデプロイメントを定義
PipelineStack	CodePipelineパイプラインを定義

演習1-4　CI/CDパイプライン構築

Step6：CodePipeline　コード作成

- **bin/ecs-demo.tsを修正**してください。PipelineStackに渡すパラメータは以下の通りです。

PipelineStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	ー	デプロイ先アカウント/リージョン
CodePipeline	pipelineName	'CustomerInfoPipeline'	パイプライン名
IAM	codeBuildRoleArn	iam.codeBuildRole.roleArn	IAMロール（codeBuildロール）
	codeDeployRoleArn	iam.codeDeployRole.roleArn	IAMロール（codeDeployロール）
	codePipelineRoleArn	iam.codePipelineRole.roleArn	IAMロール（codePiplelineロール）
	ecsTaskExecutionRoleArn	iam.ecsTaskExecutionRole.roleArn	IAMロール（ecsTaskExecutionロール）
	ecsTaskRoleArn	iam.appTaskRole.roleArn	IAMロール（appTaskロール）
GitHub	gitHubConnectionArn	conn.connectionArn	CodeConnectionsのARN
	gitHubOwner	'<xxx>'	GitHubアカウント
	gitHubRepo	'customer-info'	GitHubリポジトリ
	gitHubBranch	'main'	GitHubブランチ
ECR	ecrRepoName	'customer-info/app'	ECRリポジトリ
CodeDeploy	ecsAppName	'CustomerInfoEcsApp'	ECSアプリケーション
	ecsDeploymentGroupName	'CustomerInfoDG'	ECSデプロイメントグループ
RDS	dbSecretArn	rds.appSecret.secretArn	RDS アプリケーションシークレットのARN
	dbHost	rds.dbHost	RDS DBホスト

演習1-4　CI/CDパイプライン構築

Step6：CodePipeline　コード作成

- **lib/pipeline-stack.ts**を作成し、**PipelineStack**を定義してください。

PipelineStack作成　サンプルコード（1/2）

```
// クラス等のimport
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as codebuild from 'aws-cdk-lib/aws-codebuild';
import * as codepipeline from 'aws-cdk-lib/aws-codepipeline';
import * as cpactions from 'aws-cdk-lib/aws-codepipeline-actions';
import * as codedeploy from 'aws-cdk-lib/aws-codedeploy';
import * as iam from 'aws-cdk-lib/aws-iam';

// 外部パラメータの読み込み
export interface PipelineStackProps extends cdk.StackProps {
  pipelineName: string;
  // Roles
  codeBuildRoleArn: string;
  codePipelineRoleArn: string;
  // GitHub (CodeStar Connections)
  gitHubConnectionArn: string;
  gitHubRepo: string;
  gitHubBranch: string;
  // Buildで使う（必要ならCodeBuild環境変数に渡す）
  ecrRepoName: string;
  // CodeDeploy / ECS Blue-Green（既存）
  ecsAppName: string;
  ecsDeploymentGroupName: string;
  // ECS Task Roles
  ecsTaskExecutionRoleArn: string;
  ecsTaskRoleArn?: string;
  dbSecretArn?: string;          // RDS APPシークレット
  dbHost?: string;               // RDS DBホスト
}
// 右に続く
```

```
// クラスの宣言
export class PipelineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props: PipelineStackProps) {
    super(scope, id, props);

    // 既存ロールのimport
    const codeBuildRole = iam.Role.fromRoleArn(
      this, 'ImportedCodeBuildRole', props.codeBuildRoleArn, { mutable: false },
    );
    const codePipelineRole = iam.Role.fromRoleArn(
      this, 'ImportedCodePipelineRole', props.codePipelineRoleArn, { mutable: false },
    );
    // 既存CodeBuild プロジェクトの設定
    const buildProject = codebuild.Project.fromProjectName(
      this, 'BuildProject', 'customer-info-app',
    );
    // 既存 CodeDeploy App / DeploymentGroupの設定
    const app = codedeploy.EcsApplication.fromEcsApplicationName(
      this, 'EcsApp', props.ecsAppName,
    );
    const dg = codedeploy.EcsDeploymentGroup.fromEcsDeploymentGroupAttributes(
      this, 'EcsDG', { application: app, deploymentGroupName: props.ecsDeploymentGroupName
},
    );

    // Artifacts（CDKにバケット自動作成させる：名前は自動サフィックスで変動OK）
    const sourceOutput = new codepipeline.Artifact('SourceArtifact');
    const buildOutput  = new codepipeline.Artifact('BuildArtifact');

// 次ページに続く
```

演習1-4　CI/CDパイプライン構築

Step6：CodePipeline　コード作成

PipelineStack作成　サンプルコード（2/2）

```
// パイプライン本体
  const pipeline = new codepipeline.Pipeline(this, 'Pipeline', {
    pipelineName: props.pipelineName,
    role: codePipelineRole,
    stages: [
      // ── Source: GitHub（CodeStar Connections）
      {
        stageName: 'Source',
        actions: [
          new cpactions.CodeStarConnectionsSourceAction({
            actionName: 'GitHub_Source',
            owner:  props.gitHubRepo.split('/')[0],
            repo:   props.gitHubRepo.split('/')[1],
            branch: props.gitHubBranch,
            connectionArn: props.gitHubConnectionArn,
            output: sourceOutput,
            triggerOnPush: true,
          }),
        ],
      },
      // ── Build: Docker Build & Push → imageDetail.json のみ成果物へ
      {
        stageName: 'Build',
        actions: [
          new cpactions.CodeBuildAction({
            actionName: 'Docker_Build',
            project: buildProject,
            input: sourceOutput,
            outputs: [buildOutput], // imageDetail.json（※buildspec側で出力）
            environmentVariables: {
              ECR_REPO:       { value: props.ecrRepoName },
              EXEC_ROLE_ARN:  { value: props.ecsTaskExecutionRoleArn },
              TASK_ROLE_ARN:  { value: props.ecsTaskRoleArn ??
props.ecsTaskExecutionRoleArn },
// 右に続く
```

```
              DB_HOST:        { value: props.dbHost },
              DB_SECRET_ARN:  { value: props.dbSecretArn ?? '' },
            },
            role: codeBuildRole,
          }),
        ],
      },
      // ── Deploy: テンプレートはSourceから、imageDetailはBuildから受け取って置換
      {
        stageName: 'Deploy',
        actions: [
          new cpactions.CodeDeployEcsDeployAction({
            actionName: 'ECS_BlueGreen',
            deploymentGroup: dg,
            // GitHub上に外出ししたテンプレートを参照（例：deploy/ecs 配下）
            appSpecTemplateFile:        sourceOutput.atPath('deploy/ecs/appspec.yml'),
            taskDefinitionTemplateFile: sourceOutput.atPath('deploy/ecs/taskdef.json'),
            // Build成果物の imageDetail.json で TaskDef の <IMAGE1_NAME> を置換
            containerImageInputs: [
              {
                input: buildOutput,
                taskDefinitionPlaceholder: 'IMAGE1_NAME',
              },
            ],
          }),
        ],
      },
    ],
  });
  // 出力
  new cdk.CfnOutput(this, 'PipelineName', { value: pipeline.pipelineName });
  new cdk.CfnOutput(this, 'ArtifactBucketName', { value: pipeline.artifactBucket.bucketName });
  }
}
```

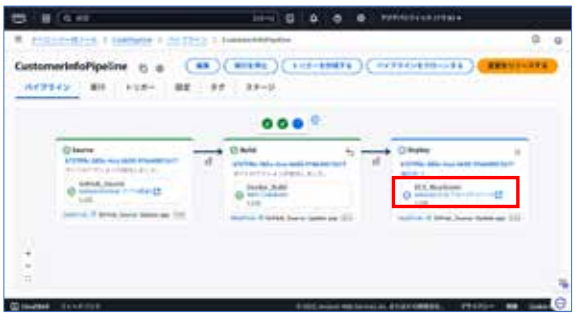
演習1-4　CI/CDパイプライン構築

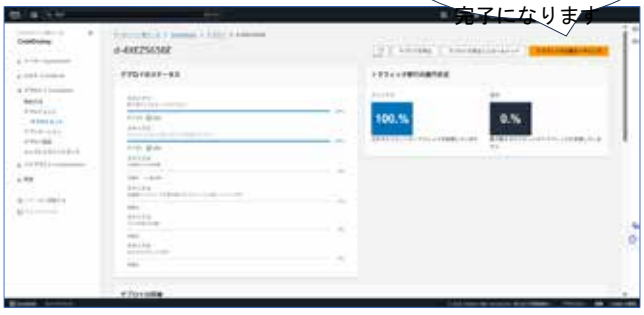
Step6：CodePipeline　動作確認

- CDKからCICDパイプラインのスタックを実行します。
  途中でCodeDeployの手動承認が発生しますので注意してください。

- CodePipeline > パイプライン から実行中のパイプラインを選択します。
  パイプラインがSource→Build→Deployと進んだら、
  実行中のCodeDeployを確認しましょう。

最後は手動です
アクティブになったら、
ボタンを押してB/Gデプロイが
完了になります



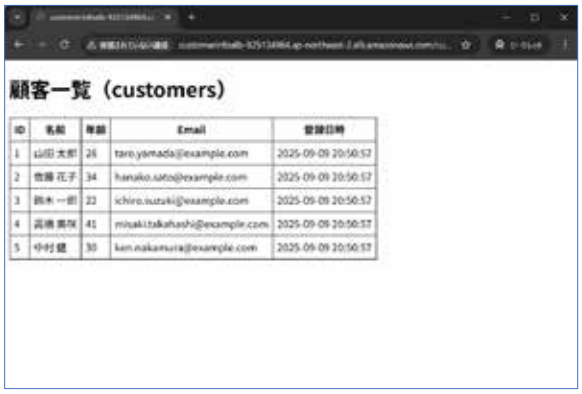


演習1-4　CI/CDパイプライン構築

Step6：CodePipeline　動作確認

■ パイプラインの完了を確認し、ブラウザ画面が変わることを確認して演習1-4完了です。
※キャッシュ等が残っていて表示が変わらない場合があるため、別ブラウザなどで確認してください。





CodePipelineでB/Gデプロイ後の顧客情報表示画面
（URIIは<ALB_DNS>/customers）

演習1-4　CI/CDパイプライン構築

Step6：CodePipeline　クリーンナップ

■ CloudShellからCloudFormationのスタックおよび作成したリソースを全て削除します。
各サービス画面から削除してください。

```
// CDKで全リソースを削除
ecs-demo $ cdk destroy --all
```

■ リソース削除の確認

①CloudFormation
- 全スタックの削除を確認

②ネットワーク / コンテナ環境
- VPC、SG、ALB、WAF、など全リソースの削除を確認
- ECS：タスク定義、クラスタ、サービス、タスクの削除を確認

③CI/CDパイプライン
- CodePipeline/Build/Deployの各種定義の削除を確認

演習1-4　CI/CDパイプライン構築

合格判定基準

■ 演習1-4-②の合格判定基準は以下の通りです。

Step4（CodeBuild）

- CodeBuildにcustomer-info-appプロジェクトが作成されている
- ECRのcustomer-info/appリポジトリにビルドしたイメージが登録されている
- GitHubにbuildSpec.yml、appsepc.yml、taskdef.tmpl.jsonの3ファイルが正しく配置されている

Step5（CodeDeploy）

- アプリケーション(CustomerInfoEcsApp)、
- デプロイメントグループが作成され、環境設定、Load Balancingが設定通りになっている
- ALBのテストリスナー、ターゲットグループが2つずつ作成され、AlbSgのインバウンドルールにtcp:9001が追加されている
- ALBのセキュリティグループのインバウンドルールにHTTP9001が登録されている

Step6（CodePipeline）

- パイプラインが作成されている（Source → Build → Deploy の3ステージ構成）
- GitHub の main ブランチへの push をトリガーにワークフローが起動し、CodePipelineが起動する
- CodePipelineの処理が進み、Source、Buildが完了し、CodeDeployが実行中になる
- CodeDeployの手動承認が発生し、承認後にB/Gデプロイが進み、パイプライン(Deploy)が完了する
- ALB のターゲットグループの向き先がTG(Green) に切り替わり、ECSのタスク(Green)に紐づくことを確認する
- デプロイ後、ブラウザから新しいアプリケーション画面にアクセスできる

演習2

ロードバランサーのHTTPS移行

演習2　ロードバランサーのHTTPS移行

演習概要・目的・背景

■ 演習概要
AWS上でHTTP対応となっているALBおよびアプリケーションを、 AWS Certificate Manager(ACM)を用いてHTTPSに移行し、セキュアなシステムを構築しましょう。

■ 目的
- AWS Certificate Manager(ACM)を活用した証明書の発行・適用手順を学びます。
- ALBのHTTP→HTTPSのリダイレクト設定を通じて、セキュアな通信経路の設計を習得します。

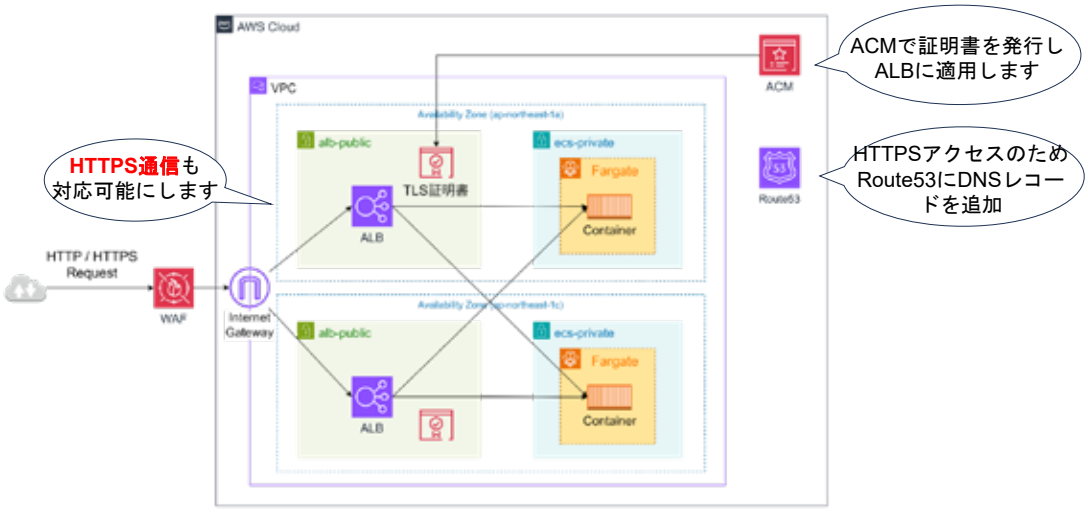
■ 背景
クラウドネイティブなシステムでは、インターネットを介した通信においてHTTPS化は必須です。
ALBを用いたアプリケーションにおいても、ACM証明書を適用することで、HTTPS(443ポート) 通信を実現できます。
本演習を通じて、HTTPS移行をIaCで自動化し、セキュアな通信を実現しましょう。

■ 前提
- 演習1-4まで完了しており、AWS上にクラウドネイティブなシステムを構築できていることが前提になります。
  ※演習1-2まで完了していても取り組むことは可能です。
- 本演習は**独自ドメインを取得していること**が前提となります。

演習2　ロードバランサーのHTTPS移行

演習概要

■ AWS上でHTTP対応となっているALBおよびアプリケーションを、 AWS Certificate Manager(ACM)を用いて、HTTPSに移行し、セキュアなシステムを構築しましょう。



演習2　ロードバランサーのHTTPS移行

演習概要

■ CDKで管理できるよう演習1で構築したシステムをベースにHTTPS対応させます。
ACMでTLS証明書を発行し、ALBに適用してください。要件は以下の通りです。

ACM要件

- ACMでTLS証明書を発行
- ACMはALBと同一リージョンで発行

ALB要件

- ALBでHTTPSを処理できるようACMで発行したTLS証明書を適用
- HTTPS(443)のリスナーを用意し、HTTPSを受信可能にする
- HTTPS通信はALBで終端させ、ALB→ECS FargateはHTTPでフォワード（演習1から変更なし）
- HTTP(80)で受信したリクエストは、HTTPSにリダイレクト(301)
  ※本来HTTP(80)は閉じた方がいいですが、リダイレクト学習のため残します。

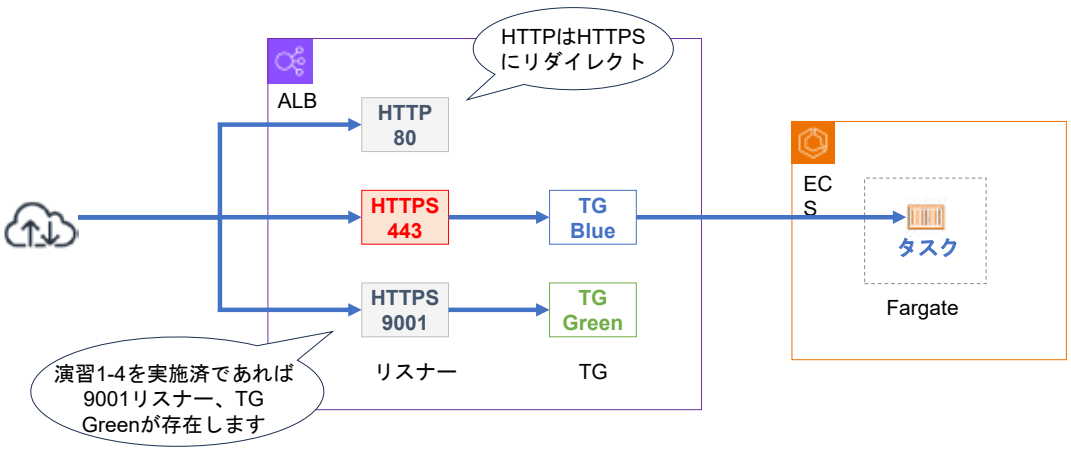
その他要件

- ALBセキュリティグループ(AlbSg)のインバウンド通信にHTTPS:443を許可
- ECSセキュリティグループ(EcsSg)の通信要件は、 HTTP:80のまま変更なし
- Route53に独自ドメイン（ホストゾーン）を登録している
- Route53にACMが提供する検証用CNAME、ALBのHTTPSアクセスのためのAレコードが登録されている

演習2　ロードバランサーのHTTPS移行

ALB　設計

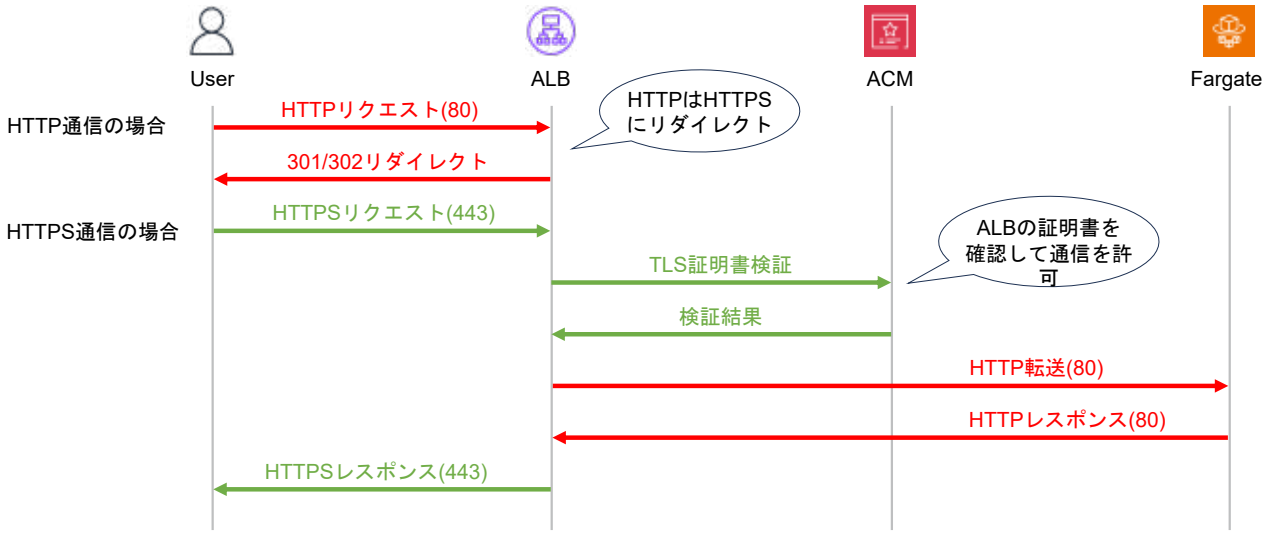
■ HTTPSリクエスト(port:443)で受け、TG経由でFargateに転送します。
HTTP(port:80)で受けた場合は、HTTPSにリダイレクトされるようにリスナーを設定します。



演習2　ロードバランサーのHTTPS移行

ALB　設計

■ ALBをHTTPSに移行するにあたり、HTTPとHTTPSの通信フローを補足説明します。



演習2　ロードバランサーのHTTPS移行

ALB　設計

■ ALBの設定値は以下の通りです。

ALB全体

分類	項目	値
ALB	ALB名	CustomerInfoAlb
	配置先サブネット	alb-public

リスナー

リスナー名	ポート	プロトコル	証明書	ターゲットグループ	用途 / 挙動
HttpListenerRedirect	80	HTTP	なし	なし（Redirectのみ）	HTTPアクセスを受けて HTTPS:443 に 301 リダイレクト
HttpsListenerProd	443	HTTPS	ACM証明書	Blue / Green ※prodTrafficRouteで管理	本番トラフィックを受けるリスナー。B/G 切替対象
HttpsListenerTest	9001	HTTPS	ACM証明書	Blue / Green ※testTrafficRouteで管理	テストトラフィックを受けるリスナー。B/G 切替対象

ターゲットグループ

ターゲットグループ	ポート	プロトコル	ターゲットタイプ	ヘルスチェックパス	初期割り当て	ターゲット
TgBlue	443	HTTPS	IP	/	HttpListenerProd	※現時点では空
TgGreen	443	HTTPS	IP	/	HttpListenerTest	※現時点では空

演習2　ロードバランサーのHTTPS移行

ACM　設計

■ ACMの設定値は以下の通りです。

項目	設定値	説明
証明書タイプ	公開証明書 (Public Certificate)	ALBで利用するため、必ず「**パブリック**」証明書
リージョン	ap-northeast-1（ALBと同一リージョン）	ALB と同じリージョンで発行する必要あり
ドメイン名	※設定したいドメイン名 例：app.example.com	公開するサブドメイン
ホストゾーン名	※Route53に登録済のドメイン 例：example.com	Route53管理の親ドメイン **※ドメイン名は事前にRoute53に登録してください**
検証方式	DNS検証 (DNS Validation)	Route53で管理しているドメインならCNAMEを自動作成
検証用CNAME	※ACM指定のCNAME	Route53 HostedZoneに追加される。これで検証が完了
証明書ARN	acm.certificateArn	ALBスタックに渡す値。証明書発行後に自動で設定される
有効期限	自動更新（90日ごとに ACM が自動更新）	手動作業は不要
利用リスナー	HTTPS:443　（Prodリスナー） HTTPS:9001（Testリスナー）	1枚の証明書を複数リスナーに適用可能

演習2　ロードバランサーのHTTPS移行

HTTPS移行　コード作成

■ AWS CDKを利用して、**ALBをHTTPSに移行するためのコード**を作成してください。

■ コード作成のポイント

bin/ecs-demo.ts

Acmスタック(AcmStack)を呼び出す定義を追加

- AlbスタックにACM証明書を引き渡してください

lib/acm-stack.ts

TLS証明書を生成する定義ファイルを作成

- Route53にTLS証明書用のCNAMEレコードを登録
- IamStackで生成した実行ロールを取り込む

lib/alb-stack.ts

ALBのリスナー生成の定義を修正

- HTTPSリスナーを作成し、tgBlue、tgGreenに紐付け
- HTTPリスナーはport:443にリダイレクト
- Route53の独自ドメイン名をALBに紐付け(Aレコード)

ecs-demo/
├─ bin/ecs-demo.ts ★修正
├─ lib/ecs-demo-stack.ts
├─ lib/net-stack.ts　★修正
├─ lib/vpce-stack.ts
├─ lib/alb-stack.ts　★修正
├─ lib/ecr-stack.ts
├─ lib/ecs-stack.ts
├─ lib/rds-stack.ts
├─ lib/connection-stack.ts
├─ lib/iam-stack.ts
├─ lib/build-stack.ts
├─ lib/deploy-stack.ts
├─ lib/pipeline-stack.ts
├─ lib/acm-stack.ts　★新規作成
・・・

CDKディレクトリ構成

演習2　ロードバランサーのHTTPS移行

HTTPS移行　コード作成

■ **bin/ecs-demo.tsを修正**してください。AcmStackを呼び出すコード定義を追加します。

```
#!/usr/bin/env node
・・・略・・・
※importを追記してください。

・・・略・・・

// ACM　　★ブロック追加
※追記してください。

// ALB / WAF
※パラメータを追加してください。
・・・略・・・
```

スタック名	用途
App	CDK全体を表すクラス名
NetStack	VPC、サブネット、SGを定義
VpceStack	VPCエンドポイントを定義
AcmStack	TLS証明書を定義
AlbStack	ALB、WAFを定義
EcrStack	ECR(リポジトリ)を定義
RdsStack	RDS(MySQL)のインスタンスを定義
EcsStack	ECS(クラスタ/タスク/サービス)を定義
ConnectionStack	CodeConnections(GitHub接続)を定義
IamStack	IAMロールを定義
BuildStack	CodeBuildプロジェクトを定義
DeployStack	CodeDeployアプリおよびデプロイメントを定義
PipelineStack	CodePipelineパイプラインを定義

演習2　ロードバランサーのHTTPS移行

HTTPS移行　コード作成

■ bin/ecs-demo.tsにおいて、AcmStack、AlbStackに渡すパラメータは以下の通りです。

AcmStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	―	デプロイ先アカウント/リージョン
DNS	domainName	※設定したいドメイン名 例：app.example.com	証明書を発行する対象のドメイン名（FQDN）
	hostedZoneName	※Route53に登録したホストゾーン 例：example.com	DNS検証に使うRoute53のホストゾーン

AlbStackに渡すパラメータ

項目	パラメータ	値	役割
環境	env	―	デプロイ先アカウント/リージョン
VPC	vpc	net.vpc	配置先VPC/サブネットを参照するため
SG	albSg	net.albSg	alb-publicサブネットに付与するSG
ACM	certificateArn	acm.certificateArn	ACMで生成したTLS証明書（追加）
DNS	domainName	※設定したいドメイン名 例：app.example.com	証明書を発行する対象のFQDN（追加）
	hostedZoneName	※Route53に登録したホストゾーン 例：example.com	DNS検証に使うRoute53のホストゾーン情報（追加）

演習2　ロードバランサーのHTTPS移行

HTTPS移行　コード作成

■ 参考）独自ドメインを用意頂き、事前にRoute53にホストゾーン（パブリック）を登録してください。

Route53

- Route53 > ホストゾーン
- ホストゾーンの作成を実行し、独自ドメインを登録してください
  ※例：　example.com



演習2　ロードバランサーのHTTPS移行

HTTPS移行　コード作成

■ **lib/net-stack.ts**を修正し、AlbSgのインバウンドルールにHTTPS:443を許可してください。

セキュリティグループ　通信要件

SG	サブネット	主要リソース	インバウンド		アウトバウンド		備考
			ポート	宛先	ポート	宛先	
AlbSg	alb-public	ALB	80 **443** 9001	0.0.0.0/0 **0.0.0.0/0** 0.0.0.0/0	80	EcsSg	HTTPS:443を追加
EcsSg	ecs-private	ECS	80	AlbSg	ALL	ALL	
JumpSg	jumpbox-public	CloudShell	—	—	ALL	ALL	
DbSg	db-private	RDS	3306	EcsSg JumpSg	—	—	
VpceSg	vpce-private	VPCエンドポイント	443	EcsSg	ALL	ALL	

演習2　ロードバランサーのHTTPS移行

HTTPS移行　コード作成

■ **lib/acm-stack.ts**を作成し、**ACMのスタック**を定義してください。

ACM証明書作成　サンプルコード

```
// インポート
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as acm from 'aws-cdk-lib/aws-certificatemanager';
import * as route53 from 'aws-cdk-lib/aws-route53';

// インタフェース定義
export interface AcmStackProps extends cdk.StackProps {
  domainName: string;        // ドメイン名　例: app.example.com
  hostedZoneName: string;    // ホストゾーン名　例: example.com
}

// クラス初期化
export class AcmStack extends cdk.Stack {
  public readonly certificateArn: string;

  constructor(scope: Construct, id: string, props: AcmStackProps) {
    super(scope, id, props);
```

```
    // HostedZoneをlookup
    const zone = route53.HostedZone.fromLookup(this, 'HostedZone', {
      domainName: props.hostedZoneName,
    });

    // 証明書作成
    const cert = new acm.Certificate(this, 'AlbCert', {
      domainName: props.domainName,
      validation: acm.CertificateValidation.fromDns(zone),
    });

    this.certificateArn = cert.certificateArn;
  }
}
```

演習2　ロードバランサーのHTTPS移行

HTTPS移行　コード作成

■ **lib/alb-stack.ts**を修正し、HTTP/HTTPSリスナーの定義を変更してください。

ALB　リスナー作成　サンプルコード

```
・・・インポート部分の追加分をサンプル提示・・・
import * as route53 from 'aws-cdk-lib/aws-route53';
import * as targets from 'aws-cdk-lib/aws-route53-targets';

・・・リスナー作成部分のみサンプルを提示・・・
  // HTTPSリスナー作成　の定義
  this.listenerSample = alb.addListener('HttpsListenerSample', {
    port: 443,
    protocol: elbv2.ApplicationProtocol.HTTPS,
    certificates: [{ certificateArn: props.certificateArn }],
    sslPolicy: elbv2.SslPolicy.RECOMMENDED_TLS,
    defaultTargetGroups: [this.targetGroup],
  });

  // HTTPリスナー（リダイレクト）の定義
  alb.addListener('HttpListenerRedirectSample', {
    port: 80,
    defaultAction: elbv2.ListenerAction.redirect({
      protocol: 'HTTPS',
      port: '443',
      permanent: true,
    }),
  });
```

```
  // Route53 レコード作成（リスナー作成後に設定）
  const zone = route53.HostedZone.fromLookup(this, 'AlbZone', {
    domainName: props.hostedZoneName,
    privateZone: false,
  });

  // domainName = "app.example.com"
  // hostedZoneName = "example.com"
  // → recordName = "app"
  const recordName =
props.domainName.endsWith(`.${props.hostedZoneName}`)
    ? props.domainName.slice(0, props.domainName.length -
props.hostedZoneName.length - 1)
    : props.domainName;

  // AレコードをALBへAlias
  new route53.ARecord(this, 'AlbARecord', {
    zone,
    recordName,
    target: route53.RecordTarget.fromAlias(new
targets.LoadBalancerTarget(alb)),
    ttl: Duration.minutes(1),
  });
```

演習2　ロードバランサーのHTTPS移行

HTTPS移行　動作確認

■ ACMスタックを実行し、証明書が「発行済み」になっていれば、OKです。
※ALBスタック実行する前に行ってください。

AWS Certificate Manager（ACM）

- Certificate Manager > 証明書一覧
- 新規証明書の発行を確認



演習2　ロードバランサーのHTTPS移行

HTTPS移行　動作確認

■ Route53のホストゾーンにCNAME、Aレコードが登録されていることを確認してください。

Route53

- Route53 > ホストゾーン >
<独自ドメイン>を選択

- CNAMEおよびAレコードが1つ
ずつ追加作成されていること
を確認



演習2　ロードバランサーのHTTPS移行

HTTPS移行　動作確認

- ALBスタックを実行し、リスナーとターゲットグループが定義通りに作成されていることを確認してください。
  ※NetStack、VpecStackは実行済という前提です。

ALB

- EC2 > ロードバランサー > CustomerInfoAlbを確認

- リスナー3つ、ターゲットグループ2つ
- リスナーとターゲットグループの紐づけ、
  プロトコルが正しいことを確認
  - HTTPS : 443 – TG_Blue
  - HTTPS : 9001 – TG_Green
  - HTTP : 80 – なし（リダイレクト）



演習2　ロードバランサーのHTTPS移行

HTTPS移行　動作確認

- ECS Fargateを実行し、HTTPおよびHTTPS通信でアクセスできることを確認できれば、演習2は完了です。
  - 証明書を適用したため、登録した<ドメイン名>（https://<ドメイン名>）でブラウザから閲覧可能です。
  - HTTPでアクセスした場合、自動的にHTTPSにリダイレクトされることを確認してください。



演習2　ロードバランサーのHTTPS移行

HTTPS移行　クリーンナップ

■ CloudShellからCloudFormationのスタックおよび関連リソースを全て削除します。

```
// CDKで全リソースを削除
ecs-demo $ cdk destroy --all
```

■ リソース削除の確認

①CloudFormation
- 全スタックの削除を確認
  ※Stackの削除に失敗する場合は、GUIでCloudFormationのスタックを強制削除してください

②Route53
- AcmStack起動時に追加されたCNAMEレコードが消えないため、GUIから該当レコードを削除してください。
  ※Aレコードは削除されます。

③その他
- VPC、SG、ALB、WAF、など全リソースの削除を確認
- ECSタスク定義、クラスター、サービス、タスクの削除を確認
- ECRリポジトリおよびイメージは残るため、残っていてOKです

演習2　ロードバランサーのHTTPS移行

合格判定基準

■ 演習2の合格判定基準は以下の通りです。

Route53 / ACM

- Route53のホストゾーンに独自ドメインが登録されている
- ホストゾーンに新規CNAME、Aレコードが登録されている
- ACMに新規証明書が登録されている

VPC / ALB

- AlbSgのインバウンドルールに443が追加されている
- ALBのリスナーが3つ、ターゲットグループが2つずつ作られている
- リスナーとターゲットグループが正しく紐付いている（HTTP:80は443にリダイレクト）
- （ECS Fargate起動後）HTTPS:443 リスナー、tgBluにタスク(IP)が紐づいている

ECS Fargate

- 設定したドメイン名（ https://<ドメイン名>）でWeb画面が閲覧できる　　※ALBのDNSではない
- httpでアクセスした場合、httpsに自動的にリダイレクトされる

演習3

CI/CDパイプラインのユニットテスト追加

演習3　CI/CDパイプラインのユニットテスト追加

演習概要・目的・背景

■ 演習概要
演習1で構築したCI/CDパイプラインに**テストステージを追加**します。パイプラインにテストステージを追加することで、アプリケーション品質を担保しつつ、自動デプロイまで行う仕組みを実現します。

■ 目的
- 品質担保のためのゲートを導入し、パイプラインの拡張を図ります。
- シンプルなテストを通じて、テスト自動化を体験します。

■ 背景
CI/CDパイプラインの役割は、コード管理からデプロイまでを自動化することにありますが、デプロイ前にアプリケーションの品質を担保することが肝要です。
演習1-4時点のパイプラインは、ビルドしてそのままデプロイという流れになっており、アプリケーションのバグに気づかないままリリースされるリスクが残っています。そこでテストステージを追加して、アプリケーションの品質を担保しましょう。

■ 前提
- 演習1-4まで完了しており、AWS上にクラウドネイティブなシステムを構築できていることが前提になります。

演習3　CI/CDパイプラインのユニットテスト追加

演習概要

CI/CDパイプラインのフローを確認しましょう。これまでソース確認、ビルド、デプロイのみでしたが、CodeBuildで**ユニットテストを実施して、**ビルドする流れに変更します。

パイプラインフロー

GitHub
ユーザーがGitHubに資材を登録すると、CI/CDパイプラインを自動でキックします。

CodePipeline
CI/CDパイプラインを起動し、各ステージの実行・管理します。

CodeBuild
CI部分。**GitHubの資材をユニットテストし、**Dockerイメージをビルドします。

CodeDeploy
CD部分。ビルドしたコンテナイメージをコンテナ基盤ECSにデプロイします。

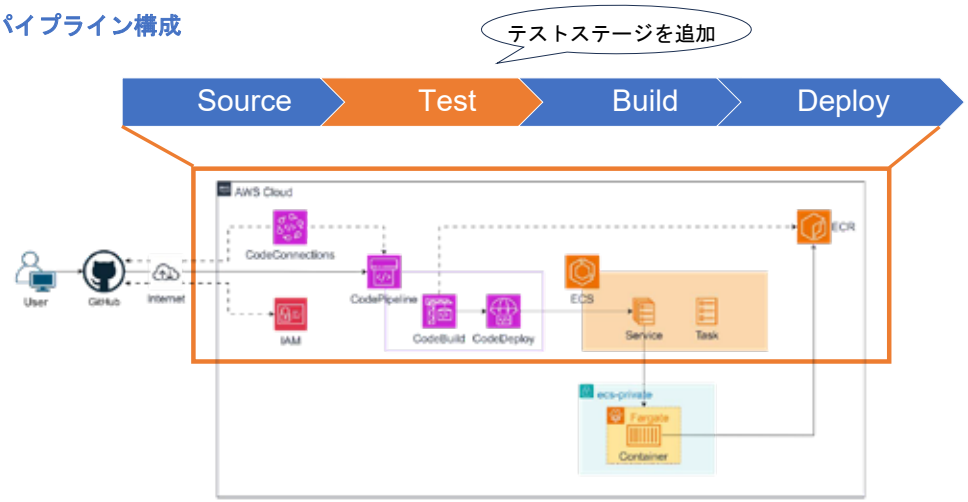
ECS(Fargate)
サーバレスコンテナが起動します。
Blue/Greenデプロイにより、無停止でアプリケーションが切り替わります。

演習3　CI/CDパイプラインのユニットテスト追加

演習概要

- 既存の CI/CDパイプラインを拡張して、テストステージを追加しましょう。
- CodePipelineにテストステージを追加し、CodeBuildの中でユニットテストを実行させます。

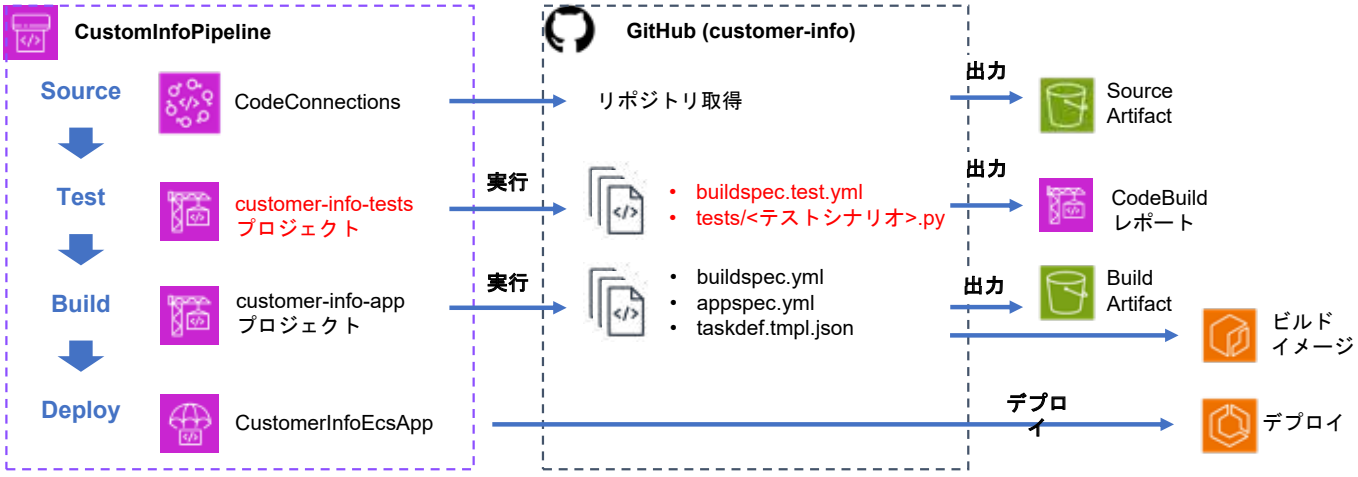
CI/CDパイプライン構成



演習3　CI/CDパイプラインのユニットテスト追加

演習概要

■ CodePipelineにテストステージを追加し、CodeBuildのテスト用プロジェクトを呼び出すように変更します。呼び出すプロジェクトは「customer-info-tests」とします。



演習3　CI/CDパイプラインのユニットテスト追加

演習概要

■ テストステージ追加に伴い、CI/CDパイプラインに追加要件が発生します。

CI/CDパイプライン要件（追加分）

- CodePipeline
  - **Testステージを追加**し、「Source、Test、Build、Deploy」の4ステージに構成を変更する
  - Testステージ：CodeBuildからビルドイメージに対してユニットテストを実施
  - Testステージの完了をもって、Buildステージに進めるようにする

- CodeBuild
  - CodeBuildにTest用プロジェクトを追加する
  - CodePipelineからTest用プロジェクトを呼び出しユニットテストを実行（結合/総合テストは対象外）

- ユニットテスト
  - Test用にbuildspec.test.ymlを準備し、Test用プロジェクトから起動する
  - 簡単なテストシナリオを用意し、ユニットテスト（pytest）を実行する
  - ユニットテストの実行結果はCodeBuildレポート（JUnit）として出力する

- その他要件
  - CodeBuildのレポート機能を利用できるよう、IAMのCodeBuildロールに権限を追加する

演習3　CI/CDパイプラインのユニットテスト追加

CodeBuild　設計

- CodeBuildの設定値は以下の通りです。
  新たなビルドプロジェクトcustomer-info-testsの設定が追加になります。

分類	項目	値	備考
共通	OS	LinuxBuildImage.STANDARD_7_0	OS/ランタイム
	IAMロール	CodeBuildRole	ECRなど操作のためのロール
	ECRリポジトリ	customer-info/app	
customer-info-app	特権モード（Privileged）	true	Docker buildなどで必須
	buildspec	buildspec.yml	利用するbuildspecファイル
	出力	BuildArtifact	imageDetail.json、taskdef.json などを出力
customer-info-tests	特権モード（Privileged）	false	不要
	buildspec	buildspec.test.yml	利用するbuildspecファイル
	出力	CodeBuild Reports	JUnitXMLテストレポート

演習3　CI/CDパイプラインのユニットテスト追加

IAM　設計

- CodeBuildのレポート機能を利用するため、CodeBuildロールに権限を2つ追加してください。

IAM設定値（1/2）

付与サービス	ロール名	AssumePrincipal	用途	権限（ポリシー内容）
CodeBuild	CodeBuildServiceRole	codebuild.amazonaws.com	CodeBuildがECRにpush / Logs出力 / S3のArtifacts参照 / kmsの暗号化・復号化を利用する	- logs:*（Logs出力） - ECR認証トークン取得 ecr:GetAuthorizationToken - ECR操作(対象Repoを限定 - ecrRepoArn) ecr:BatchCheckLayerAvailability/InitiateLayerUpload/ UploadLayerPart/CompleteLayerUpload/PutImage/ BatchGetImage/GetDownloadUrlForLayer - s3:GetObject/PutObject/GetBucketLocation/ListBucket - kms:Decrypt/Encrypt/GenerateDataKey*/DescribeKey - CodeBuildレポート機能　★追加 codebuild:*Report*/BatchPut*
CodePipeline	CodePipelineServiceRole	codepipeline.amazonaws.com	Source/Build/Deploy をオーケストレーション S3のアーティファクト操作 CodeConnectionsの利用	- CodeBuild / CodeDeploy起動 codebuild:StartBuild codedeploy:CreateDeployment/Get*/ RegisterApplicationRevision - ロール譲歩 iam:PassRole (Build/Deployロールのみ) - CodeConnectionsの利用許可 codestar-connections:UseConnection - s3:GetObject/PutObject/GetObjectVersion/ListBucket

演習3　CI/CDパイプラインのユニットテスト追加

IAM　設計

■ CodeBuildのレポート機能を利用するため、CodeBuildロールに権限を2つ追加してください。

IAM設定値（2/2）

付与サービス	ロール名	AssumePrincipal	用途	権限（ポリシー内容）
CodeDeploy	CodeDeployServiceRole	codedeploy.amazonaws.com	ECS Blue/Green デプロイを実行	- AWSCodeDeployRoleForECS（AWS管理ポリシー）
GitHub (GitHub Actions)	GitHubOIDCRole	OIDC Provider (token.actions.githubusercontent.com)	GitHub ActionsからAWSを操作（Pipeline起動など）	- パイプライン実行(対象Pipelineを指定) codepipeline:StartPipelineExecution
ECS タスク (Fargate)	EcsTaskExecutionRole	ecs-tasks.amazonaws.com	ECSのタスク起動時に ECRからのイメージPull / Logsを出力	- AWS 管理ポリシー service-role/AmazonECSTaskExecutionRolePolicy
ECS タスク (アプリ)	AppTaskRole	ecs-tasks.amazonaws.com	アプリの処理でAWS リソースにアクセスする際の実行ロール	- 特になし
Secrets Manager	AppSecret （条件付与）	—	props.appSecretArn が指定された場合、シークレットを参照可能にする	- secretsmanager:GetSecretValue AppTaskRole / EcsTaskExecutionRole に付与

演習3　CI/CDパイプラインのユニットテスト追加

buildspec.test.yml　設計

■ CodeBuildのテスト用プロジェクト実行時に参照する**buildspec.test.yml**を作成し、GitHubに配置します。ビルドとは内容が異なるため、buildspec.ymlとは別ファイルを用意します。

buildspec.test.yml　要件

pre_build
アプリのユニットテスト前の準備をします。
- アプリケーション、、試験関連ライブラリのインストール
- ソースコードのパス設定
- テスト用にダミーのDB認証情報を設定

Build
pytestを実行してユニットテストを実行します。
- pytestの実行（ユニットテストシナリオ分）

Reports
pytestの実行結果を出力します。
- pytest_reportsという名称でレポートグループを作成
- CodeBuildのレポートに実行結果をアップロード

```
customer-info/
├─ .github/
│  └─ workflows/
│     └─ ci.yml                    # GitHub Actions
├─ app/                                  # アプリ本体
│  └─ app.py
├─ nginx/                          # Nginx 設定
│  └─ nginx.conf
├─ tests/                                # テストシナリオ格納先
│  └─ <テストシナリオ>             # ユニットテストシナリオ
├─ Dockerfile
├─ requirements.txt                # インストールパッケージ一覧
├─ .dockerignore                   # ビルド高速化用
├─ .gitignore                            # Python / Docker / VSCode など
├─ README.md                       # プロジェクト概要・ローカル実行方法
├─ buildspec.test.yml              # CodeBuild(テスト)が参照するビルド設計書
├─ buildspec.yml                   # CodeBuild(ビルド)が参照するビルド設
計書
└─ deploy/ecs/
      ├─ appspec.yml               # CodeDeployが参照するアプリ仕様
      └─ taskdef.tmpl.json         # CodeDeployが参照するタスク定義
```

演習3　CI/CDパイプラインのユニットテスト追加

テストシナリオ　設計

■ CodeBuildで実行するユニットテストのシナリオについても設計します。

テストシナリオ設計

- 前提
  - CI/CDパイプラインのTestステージで実行され、最低限の品質を担保するために実施します。
  - buildspec.test.ymlを実行して、テストシナリオを実行させます。テストにはpytestを利用します。
  - アプリはAWSやRDSに接続せず、モックと環境変数で完結させます。
  - テストエラー時はCI/CDパイプラインを停止。成功時のみ、Build、Deployステージに進みます。

- テスト項目
  - Flaskアプリケーションの疎通および最低限の単体テストを実施します。
    ※試験項目表の作成が目的ではないため、テストは最低限準備する形とします。
    　今回は、「"/"へのアクセス」、「ダミーでのDB接続情報の取得」を試験項目とします。
  - 実際のAWSへの接続（Secrets ManagerやRDS）は実施しません。
    またパフォーマンステストやE2Eテストも実施しません。

- 出力
  - pytest_reportsというレポートグループを作成します。
  - test-results/junit.xml を CodeBuild のレポートとしてアップロード

演習3　CI/CDパイプラインのユニットテスト追加

GitHub　コード作成

■ GitHubのcustomer-info配下に、buildspec.test.ymlを定義してください。

buildspec.test.yml作成　サンプルコード

```
version: 0.2
phases:
  // pre_build
  pre_build:
    commands:
      - pip install -r requirements.txt
      - pip install pytest
      // app.pyのパスを指定
      - export PYTHONPATH="${CODEBUILD_SRC_DIR}:${CODEBUILD_SRC_DIR}/app:${PYTHONPATH}"
      // 認証情報ロード（ダミー値）
      - export DB_CREDENTIALS_JSON='{"username":"test_user","password":"test_pass","port":3306}'
  // build
  build:
    commands:
      // pytestの実施（test_*.py or *_test.pyを自動で検索）
      - pytest -q --junitxml=test-results/junit.xml

  // reports
reports:
  pytest_reports:
    files: [test-results/junit.xml]  // 出力ファイル
    file-format: JUNITXML
```

演習3　CI/CDパイプラインのユニットテスト追加

GitHub　コード作成

■ /tests配下に、1つめの試験シナリオ「test_health.py」を定義してください。
※このシナリオはアプリが最低限起動していることを確認します。
Flaskアプリを読み込んで、GET /が200で返ってくるのを確認します。

/tests/test_health.py　サンプルコード

```
import os, sys, types

# app/ を import パスに追加（CodeBuild でもローカルでも動くように保険）
sys.path.insert(0, os.path.abspath("app"))

# boto3 を Secrets Managerだけダミー化
class _FakeSecretsClient:
    def get_secret_value(self, SecretId):
        # テスト用のダミー資格情報を返す
        return {"SecretString":
'{"username":"test_user","password":"test_pass","port":3306}'}

def _fake_boto3_client(service_name, region_name=None):
    assert service_name == "secretsmanager"
    return _FakeSecretsClient()

# boto3 モジュールを差し替え（最低限のAPIだけ持つ偽オブジェクト）
sys.modules['boto3'] =
```

```
# アプリのimport
import app as target  # app/app.py が読み込まれる（
PYTHONPATH=.../app 前提）
app = target.app

# テスト
def test_index_returns_ok():
    client = app.test_client()
    res = client.get("/")
    assert res.status_code == 200
    assert b"OK" in res.data
```

演習3　CI/CDパイプラインのユニットテスト追加

GitHub　コード作成

■ /tests配下に、2つめの試験シナリオ「test_db_credentials.py」を定義してください。
※このシナリオは認証情報を正しくロードできるかを確認します。
load_db_credentials() が正しく認証情報を返せるか検証します。

/tests/test_db_credentials.py　サンプルコード

```
import os, sys, types, importlib

# app/ を import パスに追加
sys.path.insert(0, os.path.abspath("app"))

# boto3 をダミー化
class _FakeSecretsClient:
    def get_secret_value(self, SecretId):
        # テストで検証したい値を返す
        return {"SecretString":
'{"username":"app_user","password":"secret","port":3306}'}

def _fake_boto3_client(service_name, region_name=None):
    assert service_name == "secretsmanager"
    return _FakeSecretsClient()

sys.modules['boto3'] =
types.SimpleNamespace(client=_fake_boto3_client)
```

```
# import。app/app.py のトップレベルが実行される前にモック済み
target = importlib.import_module("app")  # = app/app.py

def test_load_db_credentials_returns_mock():
    creds = target.load_db_credentials()
    assert creds["username"] == "app_user"
    assert creds["password"] == "secret"
    assert int(creds.get("port", 0)) == 3306
```

演習3　CI/CDパイプラインのユニットテスト追加

CDK　コード作成

■ AWS CDKを利用して、**CI/CDパイプラインにテストステージを追加**してください。
※今回サンプルコードはありません。

■ コード作成のポイント

lib/iam-stack.ts

- CodeBuildRoleに2つのCodeBuild操作権限を付与してください

lib/build-stack.ts

- 既存のCodeBuildプロジェクトを参考に、テスト用のCodeBuildプロジェクト作成のセクションを追加してください。
  environmentVariablesは不要です。
- 合わせて外部から参照できるよう公開してください

lib/pipeline-stack.ts

- SourceとBuildの間にTestステージを追加してください（順番に注意）
- Testステージでテスト用CodeBuildプロジェクトを呼び出しましょう。
  Buildステージに近いですが、outputs、environmentValiablesは不要です。

bin/ecs-demo/ts

- 今回は修正しません。

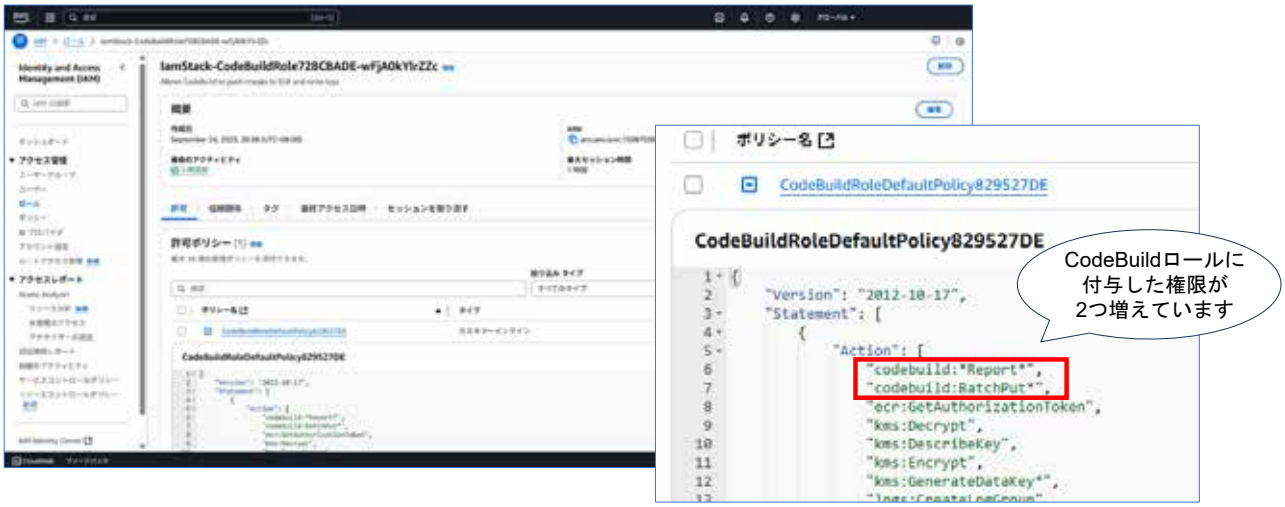
```
ecs-demo/
├── bin/ecs-demo.ts
├── lib/ecs-demo-stack.ts
├── lib/net-stack.ts
├── lib/vpce-stack.ts
├── lib/alb-stack.ts
├── lib/ecr-stack.ts
├── lib/ecs-stack.ts
├── lib/rds-stack.ts
├── lib/connection-stack.ts
├── lib/iam-stack.ts    ★修正
├── lib/build-stack.ts    ★修正
├── lib/deploy-stack.ts
├── lib/pipeline-stack.ts    ★修正
├── lib/acm-stack.ts
・・・
```

CDKディレクトリ構成

演習3　CI/CDパイプラインのユニットテスト追加

パイプラインユニットテスト追加　動作確認

■ まずIamStackを実行し、CodeBuildRoleに「codebuild:*Report*」「codebuild:BatchPut*」が追加されていることを確認してください。



演習3　CI/CDパイプラインのユニットテスト追加

パイプラインユニットテスト追加　動作確認

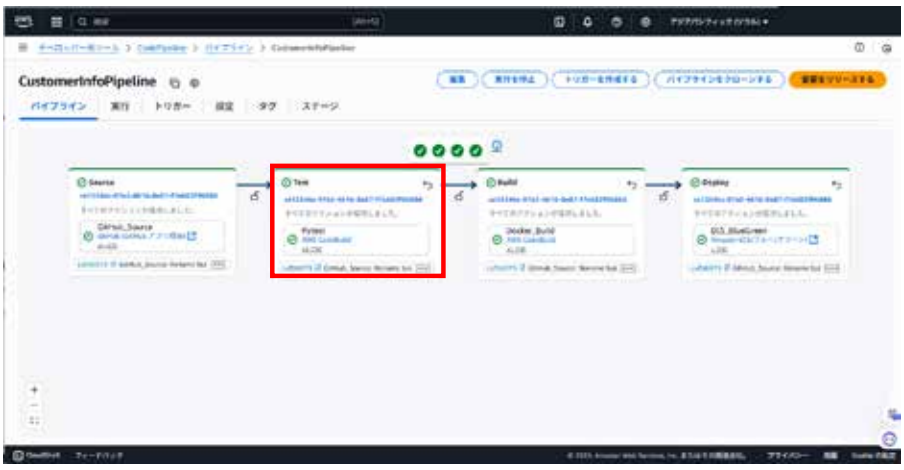
- 次にBuildStackをデプロイし、CodeBuildのビルドプロジェクトに「customer-info-tests」が存在することを確認しましょう。
  - CodeBuild > ビルドプロジェクト > customer-info-tests　で確認できます。

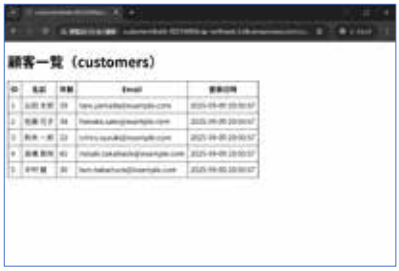


演習3　CI/CDパイプラインのユニットテスト追加

パイプラインユニットテスト追加　動作確認

- PipelineStackを実行しましょう。CustomerInfoPipelineにTestステージが追加され、パイプラインが走行完了することを確認してください。
  ブラウザからもWeb画面を閲覧できればOKです。
  ※演習1-4 Step6と同様、Deployステージは、CodeDeployのGUIから手動承認が必要です。





演習3　CI/CDパイプラインのユニットテスト追加

パイプラインユニットテスト追加　動作確認

- 最後に、CodeBuildのレポートグループを開き、Testの実行結果を確認できれば、演習3は完了です。最新のTest結果についてエラーが0件であれば、正常にテストが完了しています。
  - CodeBuild > レポートグループ > レポートグループ > customer-info-tests-pytest_reports　で確認できます。







演習3　CI/CDパイプラインのユニットテスト追加

パイプラインユニットテスト追加　クリーンナップ

- CloudShellからCloudFormationのスタックおよび関連リソースを全て削除します。

```
// CDKで全リソースを削除
ecs-demo $ cdk destroy --all
```

- リソース削除の確認

①CloudFormation
- 全スタックの削除を確認
  ※Stackの削除に失敗する場合は、GUIでCloudFormationのスタックを強制削除してください

②Route53
- AcmStack起動時に追加されたCNAMEレコードが消えないため、GUIから該当レコードを削除してください。
  ※Aレコードは削除されます。

③その他
- VPC、SG、ALB、WAF、など全リソースの削除を確認
- ECSタスク定義、クラスター、サービス、タスクの削除を確認
- ECRリポジトリおよびイメージは残っていてOKです

演習3　CI/CDパイプラインのユニットテスト追加

合格判定基準

- 演習3の合格判定基準は以下の通りです。

IAM

- CodeBuildRoleに権限が2つ追加されている（codebuild:*Report*、codebuild:BatchPut*）
- その他ロール、権限が変わらず生成されている

CodeBuild/GitHub

- GitHubにbuildspec.test.ymlやテストシナリオが登録できている
- ビルドプロジェクトに「customer-info-tests」が生成されている
- レポートグループに「customer-info-test-pytest_reports」が生成されている
- （CodePipeline走行完了後）レポート結果が表示され、履歴含めて確認できる

CodePipeline

- CustomerInfoPipelineにTestステージが追加され、Source→Test→Build→Deployの順でステージが表示される
- パイプラインの走行が完了し、全ステージが正常に完了している

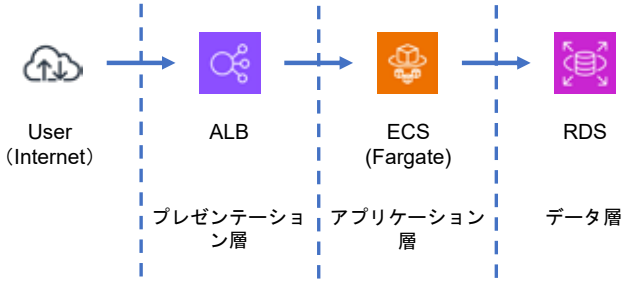
演習4

総合演習

演習4　総合演習

演習概要

演習4では、新規のWebアプリケーションとして、今日の運勢をおみくじで占うシステムを作成します。
こちらも三層アーキテクチャ（ALB＋ECS＋RDS）構成となっています。演習1と同様に全ての環境構築はAWS CDKで自動化し、複数アプリケーションの開発からデプロイまでを一括管理できるようにします。



構築する三層アーキテクチャ



Webアプリケーション画面

演習4　総合演習

システム概要

「おみくじ占い」は、おみくじ占いを提供するWebアプリケーションです。
こちらもサーバーレスアーキテクチャと自動CI/CDパイプラインで、クラウドネイティブシステムを実現します。
演習1で構築した「顧客情報管理システム」と同じAWS環境およびCDKで管理できるようにします。

主要な機能

おみくじ占い
「占う」を選択すると、データベースからランダムに取得した「おみくじ」をブラウザに表示します。

RDSデータベース連携
Amazon RDSと連携し、おみくじの情報を管理します。またSecrets Managerを利用し、DB接続情報をセキュアに管理します。顧客情報管理システムと同じDBで管理。

セキュリティ対策
WAF/ALBによるL7アプリケーションの保護、リソース配置やユーザ権限により外部からのアクセス制御します。

CI/CDパイプライン
GitHub連携したパイプラインを構築し、ビルド・デプロイを自動化します。またBlue/Greenデプロイによる無停止更新を実現します。

演習4　総合演習

ユーザー操作フロー

アプリケーションおよびインフラ基盤の構築完了後は、Webブラウザで、おみくじ占いが出来るようになります。

ユーザー操作フロー

ブラウザアクセス
ブラウザからおみくじ占いシステムにアクセス

ALB
ロードバランサーでリクエストを受け、
ファイアウォールでセキュリティチェック

ECS(Fargate)
コンテナがアプリケーションロジックを実行し、
おみくじ占いのサイトをHTMLで返却

RDS
占う選択時、RDSからおみくじデータを取得

HTML表示
おみくじの結果をHTML形式でレスポンス





HTML表示
（Webアプリケーション画面）

演習4　総合演習

CI/CDパイプラインフロー

CI/CDパイプラインのフローは演習1と同じです。
GitHubによって資材が管理され、コンテナビルドおよびデプロイの自動化を実現することができます。

パイプラインフロー

GitHub
ユーザーがGitHubに資材を登録すると、CI/CDパイプラインを自動でキックします。

CodePipeline
CI/CDパイプラインを起動し、実行・管理します。

CodeBuild
CI部分。GitHubの資材をベースに、Dockerイメージをビルドします。

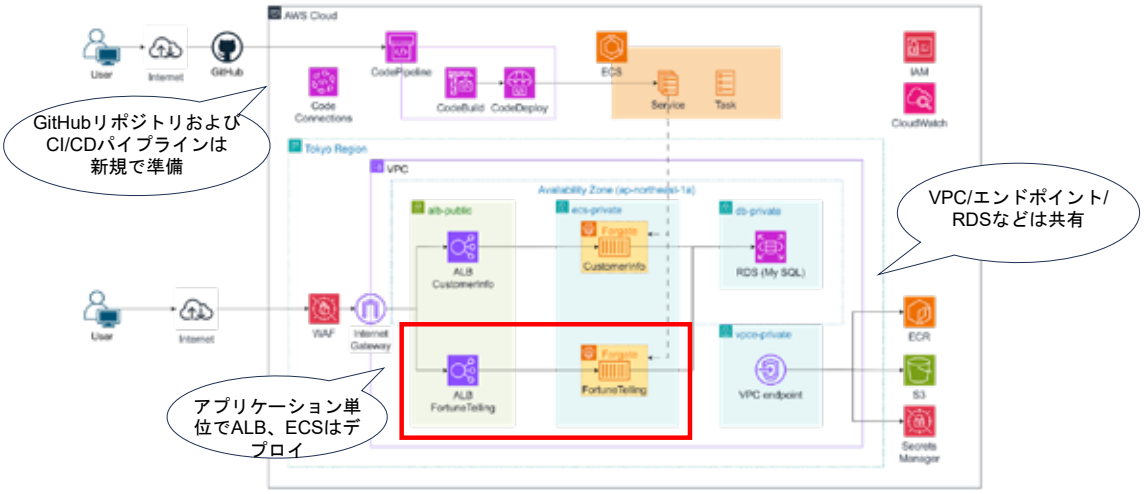
CodeDeploy
CD部分。ビルドしたコンテナイメージをコンテナ基盤ECSにデプロイします。

ECS(Fargate)
サーバレスコンテナが起動します。
Blue/Greenデプロイにより、無停止でアプリケーションが切り替わります。

演習4　総合演習

全体構成図

- 演習1～3で構築したクラウドネイティブ基盤に、「おみくじ占い」アプリケーションをデプロイしてください。既存AWSおよびCDKを拡張し、新しいアプリケーションを独立してデプロイしましょう。



演習4　総合演習

演習4の目的・背景

- 演習概要
  演習1～3で構築したAWSクラウドネイティブ基盤を活用し、「おみくじ占い」アプリケーションを追加します。
  既存CDKプロジェクトにスタック定義を追加し、複数アプリケーションを一元的に管理する流れを体験します。

- 目的
  - CDKを拡張し、複数アプリを展開できる仕組みを習得します。
  - 共通リソースとアプリ固有リソースを分離し、クラウドネイティブ設計のベストプラクティスを学びます。

- 背景
  クラウドネイティブ基盤上で複数アプリケーションをデプロイ・運用することは一般的です。
  しかし、全リソースを共有すればセキュリティや障害影響が拡大し、逆に全てを分割するとコストが増大します。
  そのため、セキュリティ・コスト・運用保守性の観点から、リソースごとに共有か分離かを適切に判断する必要があります。
  本演習を通じて、リソース再利用と独立性のバランスを取る設計思想を体得します。

- 前提
  - 演習1までを完了し、AWS上にクラウドネイティブなシステムを構築できていること。
  - 演習2・3を実施していることが望ましいですが、未実施でも本演習を進めることは可能です。

演習4　総合演習

システム要件

■ 演習4のシステム要件は以下の通りです。

ネットワーク要件

- リージョン/AZ：　東京リージョン、2AZ（1a/1c）を利用してください
- VPC/サブネット/VPCエンドポイント：既存環境を利用してください
- SG：新規ALB用に新しいSGを用意してください
- ALB：アプリケーション単位で用意してください（新規ALBを1つ追加する）
- ターゲットグループ：各アプリケーションで2つずつ用意し、各アプリケーションが単独でB/Gデプロイできるようにしてください
- WAF：既存ルールから変更ありませんが、各ALBスタックでWAFを定義してアタッチしてください
- ACM：おみくじ占いアプリケーションはHTTP通信とするため、新規TLS証明書作成およびALBへの適用は不要です。

コンテナ要件

- ECR：新規リポジトリを用意し、おみくじ占いアプリケーション用のコンテナイメージを登録できるようにしてください
- ECS：クラスター/タスク定義/サービスはアプリケーション単位で分割して用意してください

DB要件

- RDS：既存RDSは共有で利用し、スキーマとテーブルはアプリケーション単位で用意してください
- Secrets Manager：adminユーザーは共有して良いですが、アプリケーション単位でアプリユーザーを用意してください

演習4　総合演習

システム要件

■ 演習4のシステム要件は以下の通りです。

アプリケーション要件

- 簡易なおみくじ占いアプリケーションが実装します
- 演習1で用意した顧客情報表示機能とは別の独立したアプリケーションを用意します
- 両方のアプリケーションをデプロイし、B/Gデプロイできるようにします
- 次ページで詳細な要件を説明します

CI/CDパイプライン要件

- GitHub：おみくじ占いアプリケーション用のリポジトリを用意してください
- CodeConnections：GitHubとのコネクションは共有してよいです
- CI/CDパイプライン：おみくじ占いアプリケーション用に新規で用意してください
  （CodePipeline/CodeBuild/CodeDeploy）
  テストステージは不要とします。ソース、ビルド、デプロイの３ステージでパイプラインを構成してください

その他要件

- IAM：既存ロールを適用してください。新規ロール作成やポリシー追加は不要です。
- CloudWatchLogs：アプリケーション単位で別にログを管理してください

演習4　総合演習

アプリケーション要件

- おみくじ占いアプリケーションを実装し、おみくじ占いのサイトを立ち上げてください。
- アプリケーションはDockerfileでビルドできるようにコードを準備しましょう。

アプリケーション要件

- 概要：おみくじ占いアプリケーション
- 単一ファイルで動作するアプリケーションを実装してください
- 言語：　flask(python)　※別言語でもOK
- ブラウザからHTTP通信で閲覧
- ルーティング要件
  - /：ヘルスチェック用エンドポイント
    常に「200 OK」を返却
  - /fortune：占いのトップページ（HTML）を表示。「今日の日付」と「占う」ボタンを用意。
    「占う」ボタンを押下すると、結果ページ（/result）に遷移。
  - /result：占いの結果を表示。RDSに登録したデータを参照し、おみくじ番号、今日の運勢、開運の一言をランダムで表示。
    「Topに戻る」を押下すると、Topページ（/fortune）に戻る。

- 後程アプリケーションのコードを掲載しますが、**独自にアプリケーションを用意いただいても大丈夫**です。作成したコードはGitHubで管理しましょう。

演習4　総合演習

アプリケーション要件

- おみくじ占いアプリケーションのブラウザ画面を表示します。
  トップ画面で占いボタンを押すと、今日の運勢がランダムに表示されるというシンプルなものです。



Top画面　　結果画面

演習4　総合演習

CDKスタック構成

- 複数アプリケーションを同一CDKで管理するにあたり、各スタックを流用するか分割するか、の推奨案を記載しています。下記を参考に、おみくじ占いアプリケーションをデプロイできるように各スタックを用意してください。

CDKスタックの推奨構成（1/2）

分類	スタック	推奨	理由
全体	ecs-demo.ts	共通 **（修正あり）**	binで共通→ALB→TG→ECS→CI/CDの依存をアプリごとに分割。
ネットワーク構築	net-stack.ts	共通 **（修正あり）**	VPC/サブネット/IGW/NAT は基盤。ALBはどちらも alb-public を使用。
	vpce-stack.ts	共通 （変更なし）	ECR/Logs/Secrets/SSM などは共通で使用。
	acm-stack.ts	共通 （変更なし）	既存ALBにのみ証明書をアタッチ。 新規ALBはHTTPのみで証明書不要。
	alb-stack.ts alb2-stack.ts	分割 **（アプリ別に2つ用意）**	アプリケーション毎の依存を考慮し、ALBスタックを分割。 alb-stack.ts（HTTPS/80リダイレクト＋443＋9001） alb-stack2.ts（HTTPのみ：80＋9001）。 Route53/WAFの付与もここで。
コンテナ構築	ecr-stack.ts	共通 **（修正あり）**	リポジトリをアプリ別に増設。 customer-info/app, fortune-telling/appの2つ。
	ecs-stack.ts ecs2-stack.ts	分割 **（アプリ別に2つ用意）**	各アプリケーションのECSスタックを準備。 2つ用意したALB、SGを各アプリケーションに適用

演習4　総合演習

CDKスタック構成

CDKスタックの推奨構成（2/2）

分類	スタック	推奨	理由
DB構築	rds-stack.ts	共通 **（修正あり）**	RDBは2アプリケーションで共有のため変更なし。 ただし各アプリケーションのシークレットを作成するよう修正。
その他	iam-stack.ts	共通 **（修正あり）**	既存のロールに付与するポリシーは変更なし。 ただし各ロールを適用するリソースについて、おみくじ占いアプリケーションに関連する項目を対象リソースに追加。
パイプライン構築	connection-stack.ts	共通 （変更なし）	GitHub CodeConnections は複数パイプラインで共有可。
	build-stack.ts build2-stack.ts	分割 **（アプリ別に2つ用意）**	アプリケーションによって定義が異なるため、アプリケーション単位でビルドスタックを用意。
	deploy-stack.ts deploy2-stack.ts	分割 **（アプリ別に2つ用意）**	アプリケーションによって定義が異なるため、アプリケーション単位でデプロイスタックを用意。
	pipeline-stack.ts pipeline2-stack.ts	分割 **（アプリ別に2つ用意）**	アプリケーションによって定義が異なるため、アプリケーション単位でパイプラインスタックを用意

※提示しているCDKスタックの構成は推奨案です。分割もしくは共通で用意したいなどあれば、自由に構成を変更頂けます。
※演習4で追加するスタック定義のファイルはxxx2-stack.tsという名称にしています。
　より分かりやすい名称に変更頂いても構いません。

演習4

総合演習　ーネットワーク構築ー

演習4　総合演習

ネットワーク構築　SG

■ 通信要件は以下の通りです。おみくじ占いアプリケーション用の通信要件が発生します。

通信要件

1. 外部リクエスト通信（顧客情報管理用）
   Internet → ALB → ECS(Fargate) → RDS(MySQL)の経路でリクエストを処理
   - HTTP：80/HTTPS：443通信が可能。HTTPの場合、HTTPSにリダイレクト
   - B/Gデプロイ時はテストリスナーからHTTPS：9001で疎通確認
   - 演習2を実施していない場合は、HTTP通信のみ可能な状態

2. 外部リクエスト通信（おみくじ占い用）
   Internet → ALB → ECS(Fargate) → RDS(MySQL)の経路でリクエストを処理
   - HTTP：80での通信が可能。HTTPS：443は対応不要
   - B/Gデプロイ時はテストリスナーからHTTPS：9001で疎通確認

新規アプリケーションの通信要件が発生

1. AWSサービス間のプライベート通信
   ALBおよびECSは、VPCエンドポイント経由で、log/metrics連携、S3、ECRなどと通信

1. RDSへのデータ投入
   CloudShell VPC（Jumpbox）からRDSにデータを投入

演習4　総合演習

ネットワーク構築　SG

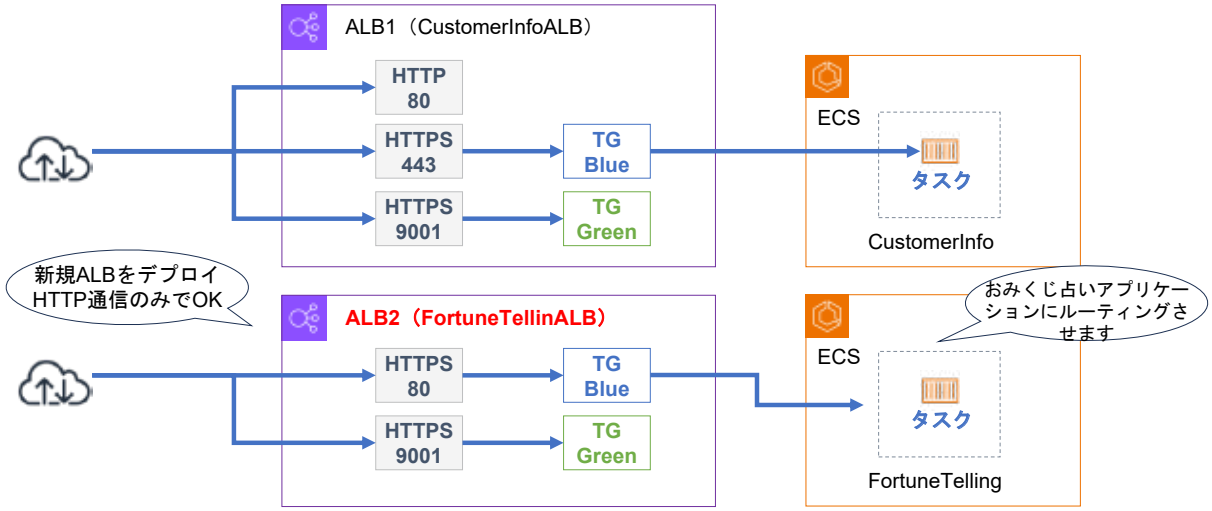
- SGの設計は以下の通りです。おみくじ占いアプリケーション用に新規SGを準備しましょう。
  既存のAlbSgを流用することも可能ですが、通信要件が異なるため、Alb2Sgを新たに用意します。

SG	サブネット	主要リソース	インバウンド		アウトバウンド	
			ポート	宛先	ポート	宛先
AlbSg	alb-public	ALB	80 / 443 / 9001	0.0.0.0/0	80	EcsSg
Alb2Sg	**alb-public**	**ALB**	**80 / 9001**	**0.0.0.0/0**	**80**	**EcsSg**
EcsSg	ecs-private	ECS	80	AlbSg **Alb2Sg**	ALL	ALL
JumpSg	jumpbox-public	CloudShell	—	—	ALL	ALL
DbSg	db-private	RDS	3306	EcsSg JumpSg	—	—
VpceSg	vpce-private	VPCエンドポイント	443	EcsSg	ALL	ALL

演習4　総合演習

ネットワーク構築　ALB

- おみくじ占いアプリケーション用に、新規ALBを生成してください。
  2つめのALBはHTTP通信のみとし、B/Gデプロイできるようターゲットグループを2つ用意しましょう。



演習4　総合演習

ネットワーク構築　ALB

■ 新規ALBの設定値は以下の通りです。HTTP通信のみ受け付けるものとします。

ALB全体

分類	項目	値
ALB	ALB名	FortuneTellingAlb
	配置先サブネット	alb-public

リスナー

リスナー名	ポート	プロトコル	証明書	ターゲットグループ	用途 / 挙動
HttpListenerProd	80	HTTP	なし	Blue / Green ※prodTrafficRouteで管理	本番トラフィックを受けるリスナー。B/G 切替対象
HttpListenerTest	9001	HTTP	なし	Blue / Green ※testTrafficRouteで管理	テストトラフィックを受けるリスナー。B/G 切替対象

ターゲットグループ

ターゲットグループ名	ポート	プロトコル	ターゲットタイプ	ヘルスチェックパス	初期割り当て	ターゲット
TgBlue	80	HTTP	IP	/	HttpListenerProd	なし
TgGreen	80	HTTP	IP	/	HttpListenerTest	なし

演習4　総合演習

ネットワーク構築　コード作成

■ AWS CDKを利用して、**2つめのALB生成に関連するコード**を作成してください。

■ コーディングのポイント

bin/ecs-demo.ts

2つめのALBスタックを呼び出す定義を追加してください。

- Alb2Stack(lib/alb2-stack.ts)を呼び出すセクションを追加

lib/net-stack.ts

2つめのALB向けSGを生成する定義を追加してください。

- Alb2Sgを用意し、alb-publicサブネットに適用
- SG要件を反映してください

lib/alb2-stack.ts

2つめのALBを生成する定義ファイルを新規作成してください。

- 既存AlbStackを流用するとコード作成が容易です
- HTTPリスナーを2つ、ターゲットグループを2つ作成
- 初期は本番リスナーをtgBlue、テストリスナーをtgGreenに紐づけ
- 2つめのALBに関連するプロパティを外部公開
- WAFは新しくWAFルールを作る形で構いません。

演習4　総合演習

ネットワーク構築　動作確認

■ 各種リソースが定義通りに作成されていれば、ネットワーク構築の修正は完了です。
AWSコンソールから各種リソースを確認しましょう。

SG

- VPC > セキュリティグループ
- 新規SG（Alb2Sg）を確認



ALB

- EC2 > ロードバランサー
- 新規ALB（FortuneTellingAlb）を確認



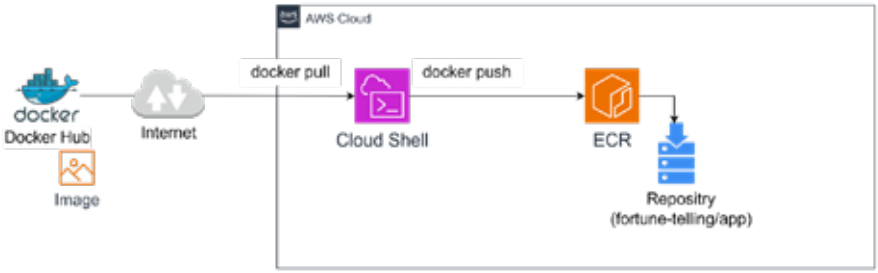
演習4

総合演習　ーコンテナ構築ー

演習4　総合演習

コンテナ構築　ECR

- おみくじ占いアプリケーションのDockerイメージを格納するECRリポジトリ(**fortune-telling/app**)を作成してください。



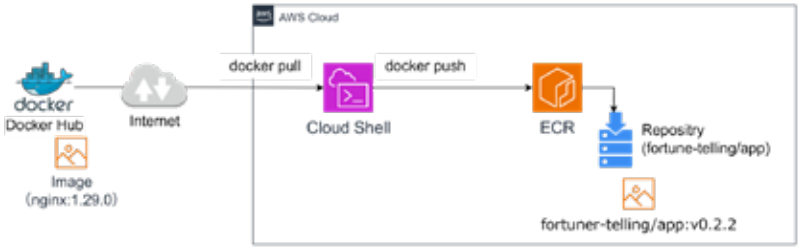
- ECRの設定値は以下の通りです。

分類	項目	値
ECR	リポジトリ名	fortune-telling/app
	脆弱性スキャン	true
	誤削除防止	RETAIN
	ライフサイクルポリシー	7日間

演習4　総合演習

コンテナ構築　イメージpush

- CloudShellでDockerイメージを取得・ビルドして、ECRにdocker pushしてください。
  現段階のベースイメージはNginxとし、**fortune-telling/appリポジトリにpush**します。



- ECRに登録するイメージは以下の通りです。

分類	項目	値
ベースイメージ	イメージ名	nginx
	タグ	1.29.0
ECR	Push先リポジトリ	fortune-telling/app
	タグ	v0.2.2

演習4　総合演習

コンテナ構築　ECS

- おみくじ占いアプリケーション用にECS Fargateを設定します。設定値は以下の通りです。
  クラスター名やALBなどは異なりますが、ECS関連の設定は同じです。

ECS設定値（1/2）

分類	項目	パラメータ	値
クラスター	クラスター名	clusterName	fortune-telling-cluster
	メトリクスログ連携	containerInsights	true
CloudWatchロググループ	ロググループ名	logGroupName	/ecs/fortune-telling
	保存期間	rentention	1week
タスク定義	タスクに割り当てるCPU/MEM	cpu / memoryLimits	256 / 512
	タスク定義ファミリー	family	fortune-telling-task
コンテナ定義	イメージ	image	fortune-telling/app:v0.2.2
	ポートマッピング	portMappings	80/tcp
	ログ出力	Logging	/ecs/fortune-telling
	ヘルスチェック	Healthcheck	TCP 80番ポートをLISTEN 30秒間隔でチェック タイムアウト5秒、リトライ3回 起動後60秒間は失敗を無視

演習4　総合演習

コンテナ構築　ECS

ECS設定値（2/2）

分類	項目	パラメータ	値
Fargateサービス	サービス	serviceName	fortune-telling-service
	タスク数	desiredCount	2
	セキュリティグループ	securityGroups	EcsSg
	サブネット	vpcSubnets	ecs-private
	ECS Exec有効化	enableExecuteCommand	true
	起動直後のヘルチェ開始時間	healthCheckGracePeriod	60sec
ALB　※	ALBのターゲットグループ	attachToApplicationTargetGroup	Alb2TgArn　**※Alb2を指定**
出力	ECSクラスタのARN	ClusterArn	cluster.clusterArn
	Fargateのサービス名	ServiceName	service.serviceName
	タスク定義のファミリー名	TaskFamily	taskDef.family

※…ALBのTGをECS Fargateにすることで、FargateのIPアドレスが登録され、
リクエストがALBからECS Fargateに転送されるようになります。

演習4　総合演習

コンテナ構築　コード作成

■ AWS CDKを利用して、**おみくじ占いアプリケーションをデプロイ**してください。
現段階では、Nginxが立ち上がります。

■ コーディングのポイント

bin/ecs-demo.ts

おみくじ占いアプリケーションをデプロイする定義を追加してください。
- Ecs2Stack(lib/ecs2-stack.ts)を呼び出すセクションを追加
- 2つのEcsスタックに渡すパラメータ（コンテナリポジトリ）の値に注意

lib/ecr-stack.ts

おみくじ占い用リポジトリを生成する定義ファイルを追加してください。
- ECRにfortune-telling/appリポジトリを作成
- customer-info/appも同一スタックで管理
- リポジトリのプロパティに注意（fortune-telling/app分も公開）

fortune-telling/app

ECRのfortune-telling/appリポジトリにDockerイメージを格納してください。
- 演習1-2と同様、Nginxをdocker buildしてECR格納
- 既にcustomer-info/appが存在する場合、

※このタイミングでアプリケーションコードおよびDockerfileでビルドして、
正式なアプリケーションイメージを用意して頂いても良いです

lib/ecs2-stack.ts

おみくじ占いアプリケーションをデプロイする定義ファイルを作成してください。
- fortune-telling/appのイメージをデプロイ
- ALB、SG、TGは演習4で新たに作成したものを適用

演習4　総合演習

コンテナ構築　動作確認

■ 各種リソースが定義通りに作成されていれば、コンテナ構築の修正は完了です。
AWSコンソールから各種リソースを確認しましょう。

ECR

- ECR > Private registry > Repositories
- **fortune-telling/appリポジトリ**を確認



ECS

- Elastic Container Service
  1. クラスター(**fortune-telling-cluster**)が作成されていること
  2. タスク定義が1つ作成されていること
  3. クラスター内に、1サービス、2タスクが作成されていること



演習4　総合演習

コンテナ構築　動作確認

■ アプリケーションの動作も確認したら、次のステップに進みましょう。

ALB

- EC2 > ロードバランサー
- 新規ALBのリソースマップでIPの紐付きを確認



Webブラウザ

- http://$ALB_DNS/ を検索して、fortune-tellingについても、NginxのWelcome画面が表示がされればOKです。



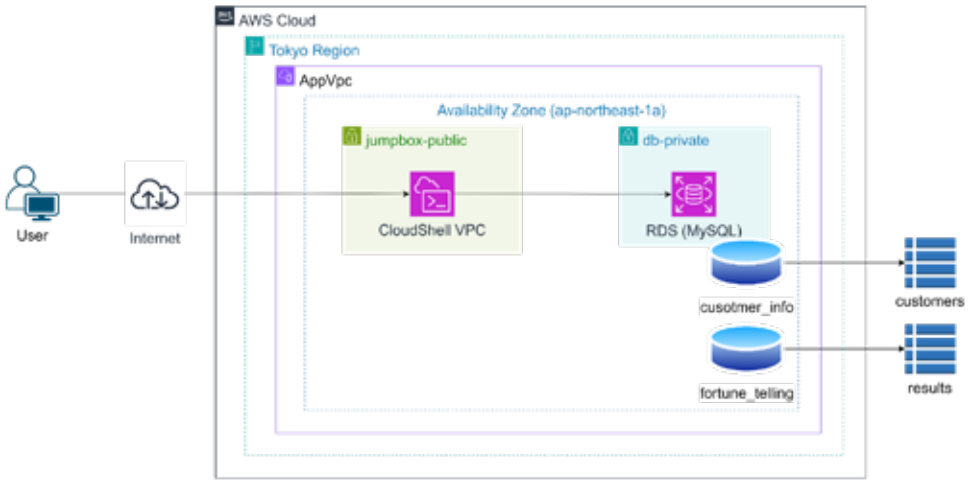
演習4

総合演習　―DB構築―

演習4　総合演習

DB構築　RDS

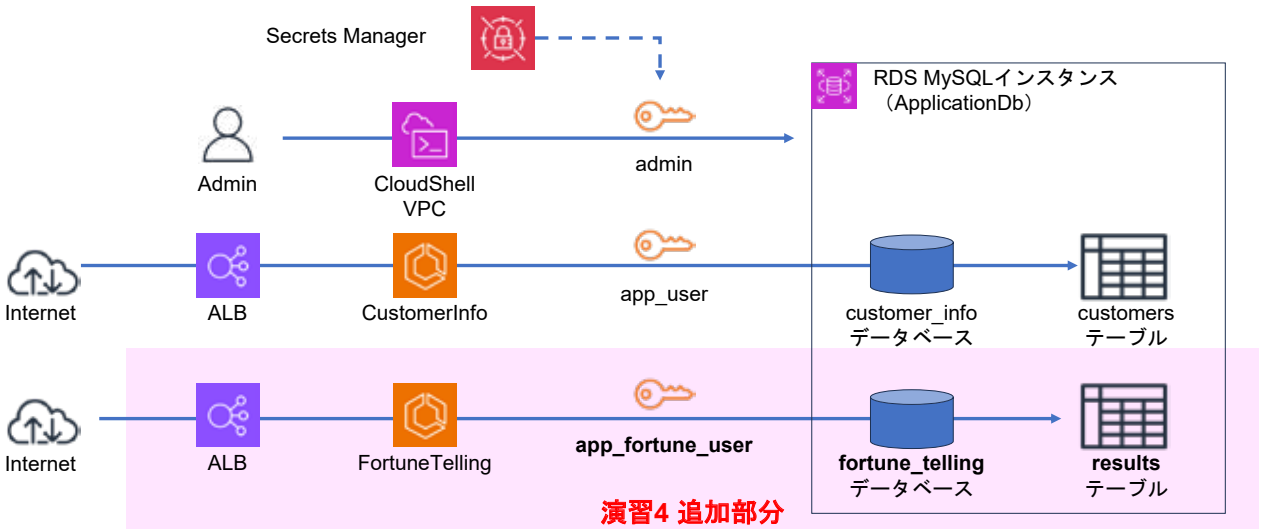
- 既存RDSインスタンスに新規データベースおよび新規テーブルを作成し、占い結果データを登録してください。また、おみくじ占いアプリケーション用のシークレットも作成してください。



演習4　総合演習

DB構築　RDS

- RDSへのアクセスおよびアプリケーションとDBの関係性は以下の通りです。



演習4　総合演習

DB構築　RDS

- SecretsManagerで管理する新規ユーザー分のシークレットを追加します。
  設計値は以下の通りです。

Secrets Manager設定値

分類	項目	パラメータ	値
Secrets Manager	シークレット名（管理者ユーザー）	secretName	admin-db-credentials
	管理者ユーザー	username	admin
	管理者ユーザーパスワード	※SecretsManagerが自動生成※	※SecretsManagerが自動生成※
	シークレット名（アプリユーザー）	secretName	customer-info-app-credentials
	アプリユーザー	Username	app_user
	アプリユーザーパスワード	※SecretsManagerが自動生成※	※SecretsManagerが自動生成※
	シークレット名（アプリユーザー）	secretName	fortune-telling-app-credentials
	アプリユーザー	Username	app_fortune_user
	アプリユーザーパスワード	※SecretsManagerが自動生成※	※SecretsManagerが自動生成※

演習4　総合演習

DB構築　RDS

- RDSインスタンスは共通で利用します。初期データベースは一つしか作成できないため、初期に作成するデータベースはcustoemer_infoのままとします。RDS関連の設計値は以下の通りです。

RDS設定値

分類	項目	値	
インスタンス	エンジンタイプ	engine	MySQL
	バージョン	version	ver8.0.42
	DB識別子（インスタンス名）	instanceIdentifier	application-db
	VPC名 / サブネット	vpc / vpcSubnets	proc.vpc（AppVpc）/ db-private
	インスタンスタイプ	instanceType	t3.micro
	セキュリティグループ	securityGroups	props.dbSg（DbSg）
	マルチAZ	multiAz	false（1AZ）
ストレージ	ストレージタイプ	storageType	gp3
	容量（最小）	allocatedStorage	20GiB
	容量（最大）	maxAllocatedStorage	100GiB
その他	削除保護	deletionProtection	False
	CDK削除時の挙動	removalPolicy	destroy
	初期データベース	databaseName	customer_info

演習4　総合演習

DB構築　テーブル作成 / データ投入

- 新規データベース(fortune_telling)およびテーブル(results)を作成し、データを登録してください。
  データベースおよびテーブルの値は以下の通りです。

データベース設計（fortune_telling）

項目	内容	説明
データベース名	fortune_telling	占いアプリ用のデータベース。運勢データなどを格納
文字コード	Utf8mb4	絵文字や漢字を扱うための文字コード
照合順序	utf8mb4_0900_ai_ci	大文字小文字を区別しない比較方式
テーブル数	1	占い結果を格納するテーブルを1つ用意

テーブル設計　（results）

カラム	データ型	制約	説明
id	BIGINT	PRIMARY KEY, AUTO_INCREMENT	通し番号。自動採番される一意のID
number	VARCHAR(10)	NOT NULL	おみくじ番号 （例：一、二、三など漢数字）
fortune_rank	VARCHAR(10)	NOT NULL	運勢（例：大吉、吉、小吉、凶など）
message	VARCHAR(255)	NOT NULL	開運のメッセージ内容
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	データ登録日時。自動で登録

演習4　総合演習

DB構築　テーブル作成 / データ投入

- 新規データベース(fortune_telling)およびテーブル(results)を作成し、データを登録してください。
  テーブルに登録するデータは以下の通りです。

登録データ（サンプル）

id	number	fortune_rank	message	created_at
1	一	大吉	新しいアイデアに挑戦すると吉！	2025-09-09 10:00:00
2	二	中吉	朝の散歩で運気アップ。	2025-09-09 10:01:00
3	三	小吉	メール整理でチャンス到来。	2025-09-09 10:02:00
4	四	末吉	焦らず準備を整えると好転。	2025-09-09 10:03:00
5	五	凶	無理は禁物。こまめな休憩を。	2025-09-09 10:04:00
6	六	大吉	ありがとうを先に言うと良縁。	2025-09-09 10:05:00
7	七	中吉	机を片付けると運気が巡る。	2025-09-09 10:06:00
8	八	小吉	丁寧な言葉遣いが鍵。	2025-09-09 10:07:00
9	九	末吉	背伸びより継続を選ぼう。	2025-09-09 10:08:00
10	十	凶	夜更かしは控えて体調第一。	2025-09-09 10:09:00

演習4　総合演習

DB構築　テーブル作成 / データ投入

- おみくじ占いアプリケーションが利用するユーザー(app_fortune_user)も作成しましょう。
  設定値は以下の通りです。

ユーザー設計（app_fortune_user）

項目	設定値	説明
ユーザー名	app_fortune_user	fortune_tellingアプリケーションが利用するMySQLユーザー
ホスト	%	アクセス可能なデータベース
テーブル	fortune_telling.*	アクセス可能なテーブル （fortune_tellingの全テーブルを対象）
権限	SELECT/INSERT/UPDATE	付与する操作権限

演習4　総合演習

DB構築　コード作成

- AWS CDKを利用して、**おみくじ占いアプリケーションが参照するデータベースを準備**してください。
- コーディングのポイント

bin/ecs-demo.ts

変更ありません。

lib/rds-stack.ts

新規シークレットを生成する定義を追加してください。

- アプリケーション単位でシークレットを用意
- RDSインスタンスは共有
- 新規シークレットは外部プロパティとして公開

RDS（application-db）

applicatoin-dbに新規データベース、テーブル、データを登録してください。

- CloudShell VPCからMySQLで操作
- app_fortune_userを追加で作成
- forutne_tellingデータベース、resultsテーブルを作成
- 占い結果をresultsテーブルに登録

演習4　総合演習

DB構築　動作確認

- RDSに「application-db」、Secrets Managerに3つのシークレットが作成されていることを確認しましょう。

RDS

- Aurora and RDS > データベース
- DB識別子**application-db**を確認
  ※既存から変更なし



Secrets Manager

- Secrets Manager > シークレット
- 3つのシークレットの生成を確認
  - admin-db-credentials
  - customer-info-app-credentials
  - **fortune-telling-app-credentials（新規）**



演習4　総合演習

DB構築　動作確認

- CloudShell VPCからapplication-dbにアクセスして、新規ユーザー/データベース/テーブルの作成、およびテーブルへのデータ登録まで済んだら、DB構築パートは完了です。

```
# データベース表示
MySQL [(none)]> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| customer_info      |
| fortune_telling    |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
6 rows in set (0.002 sec)

# テーブル確認
MySQL [fortune_telling]> DESCRIBE results;
+--------------+--------------+------+-----+-------------------+-------------------+
| Field        | Type         | Null | Key | Default           | Extra             |
+--------------+--------------+------+-----+-------------------+-------------------+
| id           | bigint       | NO   | PRI | NULL              | auto_increment    |
| number       | varchar(10)  | NO   |     | NULL              |                   |
| fortune_rank | varchar(10)  | NO   |     | NULL              |                   |
| message      | varchar(255) | NO   |     | NULL              |                   |
| created_at   | timestamp    | YES  |     | CURRENT_TIMESTAMP | DEFAULT_GENERATED |
+--------------+--------------+------+-----+-------------------+-------------------+
5 rows in set (0.002 sec)
```

演習4　総合演習

DB構築　動作確認

- CloudShell VPCからapplication-dbにアクセスして、新規ユーザー/データベース/テーブルの作成、およびテーブルへのデータ登録まで済んだら、DB構築パートは完了です。

```
# データ確認
MySQL [fortune_telling]> SELECT * FROM results;
+----+--------+-------------+-----------------------------------------------+---------------------+
| id | number | fortune_rank | message                                      | created_at          |
+----+--------+-------------+-----------------------------------------------+---------------------+
|  1 | 一     | 大吉        | 新しいアイデアに挑戦すると吉！               | 2025-10-07 08:44:23 |
|  2 | 二     | 中吉        | 朝の散歩で運気アップ。                       | 2025-10-07 08:44:23 |
|  3 | 三     | 小吉        | メール整理でチャンス到来。                   | 2025-10-07 08:44:23 |
|  4 | 四     | 末吉        | 焦らず準備を整えると好転。                   | 2025-10-07 08:44:23 |
|  5 | 五     | 凶          | 無理は禁物。こまめな休憩を。                 | 2025-10-07 08:44:23 |
|  6 | 六     | 大吉        | ありがとうを先に言うと良縁。                 | 2025-10-07 08:44:23 |
|  7 | 七     | 中吉        | 机を片付けると運気が巡る。                   | 2025-10-07 08:44:23 |
|  8 | 八     | 小吉        | 丁寧な言葉遣いが鍵。                         | 2025-10-07 08:44:23 |
|  9 | 九     | 末吉        | 背伸びより継続を選ぼう。                     | 2025-10-07 08:44:23 |
| 10 | 十     | 凶          | 夜更かしは控えて体調第一。                   | 2025-10-07 08:44:23 |
+----+--------+-------------+-----------------------------------------------+---------------------+
10 rows in set (0.001 sec)

# ユーザー確認
MySQL [(none)]> SHOW GRANTS FOR 'app_fortune_user'@'%';
+-----------------------------------------------------------------------------+
| Grants for app_fortune_user@%                                               |
+-----------------------------------------------------------------------------+
| GRANT USAGE ON *.* TO `app_fortune_user`@`%`                                |
| GRANT SELECT, INSERT, UPDATE ON `fortune_telling`.* TO `app_fortune_user`@`%` |
+-----------------------------------------------------------------------------+
2 rows in set (0.004 sec)
```

演習4　総合演習

合格判定基準

- 演習4前半の合格判定基準は以下の通りです。

ネットワーク構築

- ☐ 新規ALB用に新しくAlb2Sgが作成できている
- ☐ 新規アプリケーション用のALBをデプロイされ、リスナー2つとターゲットグループ2つが正しく紐づいている
- ☐ 両方のALBにWAFがアタッチされている

コンテナ構築

- ☐ ECRに新規リポジトリ（fortune-telling/app）が生成され、イメージが格納されている
- ☐ ECSに新規クラスター / タスク定義 / サービスが登録され、fortune-tellingのタスクを実行できている

DB構築

- ☐ 既存RDSインスタンスに新規ユーザー / データベース / テーブルが生成されている
- ☐ resultsテーブルに占い結果のデータを登録できている（サンプルデータを利用する場合は10件）
- ☐ 新規アプリケーション用シークレットが生成されている

演習4

総合演習　ーGitHub準備ー

演習4　総合演習

GitHub準備

- GitHubに、おみくじ占いアプリケーションおよびパイプライン関連の資材を準備しましょう。
- まず新規リポジトリ**fortune-telling**を作成し、配下に各種ファイルを作成してください。
  リポジトリ構成や役割は演習1-4とほぼ変わりませんが、必要に応じてコードを修正してください。

```
fortune-telling/
├── .github/
│   └── workflows/
│       └── ci.yml                  # GitHub Actions
├── app/                                    # アプリ本体
│   └── app.py
├── Dockerfile
├── requirements.txt
├── .dockerignore                   # ビルド高速化用
├── .gitignore                              # Python / Docker / VSCode など
├── README.md                       # プロジェクト概要・ローカル実行方法
├── buildspec.yml                   # CodeBuildが参照するビルド設計書
└── deploy/
    └── ecs/
        ├── appspec.yml                     # CodeDeployが参照するアプリ仕様
        └── taskdef.tmpl.json           # CodeDeployが参照するタスク定義
```

ポイント

- customer-info固有の設定を含む場合は、fortune-telling相当の設定値に変更してください。
- こちらから提示する新規アプリケーションを利用する場合、Nginxを使用しないため、nginx/nginx.conflは用意しません。
- パイプラインのテストステージは実装しません。そのため、buildspec.test.ymlや/testおよびテストシナリオは用意しません。
- 大きな修正が入るのは、app/app.pyとDockerfileです。

演習4　総合演習

GitHub準備

■ おみくじ占いアプリケーションの要件を再掲します。
Dockerfileでビルドできるようにコードを準備してください。

アプリケーション要件

- 概要：おみくじ占いアプリケーション
- 単一ファイルで動作するアプリケーションを実装してください
- 言語：　flask(python)　※別言語でもOK
- ブラウザからHTTP通信で閲覧
- ルーティング要件
  - /：ヘルスチェック用エンドポイント
    常に「200 OK」を返却
  - /fortune：占いのトップページ（HTML）を表示。「今日の日付」と「占う」ボタンを用意。
    「占う」ボタンを押下すると、結果ページ（/result）に遷移。
  - /result：占いの結果を表示。RDSに登録したデータを参照し、おみくじ番号、今日の運勢、開運の一言をランダムで表示。
    「Topに戻る」を押下すると、Topページ（/fortune）に戻る。

■ 後程アプリケーションのコードを掲載しますが、**独自にアプリケーションを用意いただいても大丈夫**です。作成したコードはGitHubで管理しましょう。

演習4　総合演習

GitHub準備

■ おみくじ占いアプリケーションのブラウザ画面を表示します。
トップ画面で占いボタンを押すと、今日の運勢がランダムに表示されるというシンプルなものです。



Top画面　　結果画面

演習4　総合演習

参考）GitHub準備

- おみくじ占いアプリケーションの関連するファイルについて、サンプルコードを掲載します。

Dockerfile　サンプルコード

```
# ランタイム
FROM python:3.12-slim

# OSパッケージ（MySQL, curl など）
RUN apt-get update -y \
 && apt-get install -y --no-install-recommends \
      build-essential default-libmysqlclient-dev ca-certificates curl \
 && rm -rf /var/lib/apt/lists/*

WORKDIR /app

# 依存関係
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# アプリ本体
COPY app/app.py /app/app.py
ENV PORT=80
EXPOSE 80

# Gunicornで起動（0.0.0.0:80）
CMD ["gunicorn", "-b", "0.0.0.0:80", "--workers", "2", "app:app"]
```

演習4　総合演習

参考）GitHub準備

- おみくじ占いアプリケーションの関連するファイルについて、サンプルコードを掲載します。

app/app.py　サンプルコード（1/3）

```
# app.py
from flask import Flask, render_template_string, url_for, redirect, request
from datetime import datetime, timezone, timedelta
import os, json
import pymysql
import boto3

# PyMySQL を MySQLdb として使う
pymysql.install_as_MySQLdb()

app = Flask(__name__)

# ===== 設定 =====
AWS_REGION     = os.getenv("AWS_REGION", "ap-northeast-2")
DB_HOST        = os.getenv("DB_HOST")  # 例: xxxxxx.ap-northeast-
1.rds.amazonaws.com
DB_NAME        = os.getenv("DB_NAME", "fortune_telling")
DB_SECRET_NAME = os.getenv("DB_SECRET_NAME", "fortune-
telling-app-credentials")
TZ_JST         = timezone(timedelta(hours=9))
```

```
# ===== Secrets Manager から DB 認証情報を取得　=====
_sm = boto3.client("secretsmanager", region_name=AWS_REGION)
_secret = _sm.get_secret_value(SecretId=DB_SECRET_NAME)
_creds  = json.loads(_secret["SecretString"])  # {"username": "...",
"password": "..."}

def get_conn():
    return pymysql.connect(
        host=DB_HOST,
        user=_creds["username"],
        password=_creds["password"],
        database=DB_NAME,
        charset="utf8mb4",
        cursorclass=pymysql.cursors.DictCursor,
        autocommit=True,
    )
```

演習4　総合演習

参考）GitHub準備

- おみくじ占いアプリケーションの関連するファイルについて、サンプルコードを掲載します。

app/app.py　サンプルコード（2/3）

```
# ===== HTML（トップ画面 /fortune）=====
TOP_HTML = """
<!doctype html><html lang="ja"><head>
<meta charset="utf-8"><meta name="viewport" content="width=device-
width,initial-scale=1">
<title>今日の運勢</title>
<style>
body{font-family:sans-serif;text-align:center;margin:2em;}
button{padding:.6em 1.2em;font-
size:1em;background:#0ea5e9;color:#fff;border:0;border-
radius:6px;cursor:pointer;}
</style></head><body>
<h1>今日の運勢</h1>
<p>{{ date_text }}</p>
<form method="post" action="{{ url_for('result') }}">
<button type="submit">占う</button>
</form>
</body></html>
"""
```

```
# ===== HTML（/result）=====
RESULT_HTML = """
<!doctype html><html lang="ja"><head>
<meta charset="utf-8"><meta name="viewport" content="width=device-
width,initial-scale=1">
<title>今日の運勢 - 結果</title>
<style>
body{font-family:sans-serif;text-align:center;margin:2em;}
p{margin:.5em 0;}
button{padding:.5em 1em;border:0;border-
radius:6px;background:#eee;cursor:pointer;}
</style></head><body>
<h1>今日の運勢（結果）</h1>
<p>おみくじ番号：{{ number }}番</p>
<p>今日の運勢：{{ fortune_rank }}</p>
<p>開運の一言：{{ message }}</p>
<form action="{{ url_for('fortune') }}" method="get">
<button type="submit">Topに戻る</button>
</form></body></html>
"""
```


演習4　総合演習

参考）GitHub準備

- おみくじ占いアプリケーションの関連するファイルについて、サンプルコードを掲載します。

app/app.py　サンプルコード（3/3）

```
# ===== ルーティング =====
# / はヘルスチェック用
@app.get("/")
def health():
    return {"status": "ok"}, 200

# /fortune はトップ画面
@app.get("/fortune")
def fortune():
    today = datetime.now(TZ_JST)
    date_text = f"{today.month}月{today.day}日の運勢"
    return render_template_string(TOP_HTML, date_text=date_text)

# /result は結果画面（ボタンクリックでPOST想定／GETも許可）
@app.route("/result", methods=["GET", "POST"])
def result():
    sql = "SELECT number, fortune_rank, message FROM results
ORDER BY RAND() LIMIT 1"
```

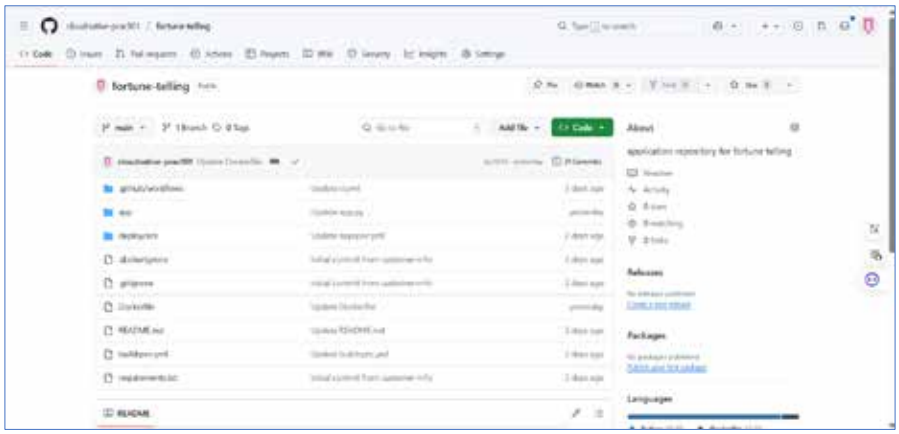
```
row = None
    try:
        with get_conn() as conn, conn.cursor() as cur:
            cur.execute(sql)
            row = cur.fetchone()
    except Exception:
        # DB未準備時のフォールバック
        row = {"number": "11", "fortune_rank": "中吉", "message": "深呼吸し
てペースを整えよう。"}
    # テーブルが空のときのフォールバック
    if not row:
        row = {"number": "11", "fortune_rank": "中吉", "message": "深呼吸し
てペースを整えよう。"}

    return render_template_string(
        RESULT_HTML,
        number=row["number"],
        fortune_rank=row["fortune_rank"],
        message=row["message"],
    )
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=int(os.getenv("PORT", 80)))
```

演習4　総合演習

GitHub準備　動作確認

- fortune-tellingリポジトリのmainブランチに各ディレクトリおよび各ファイルが正しく反映されていることを確認してください。

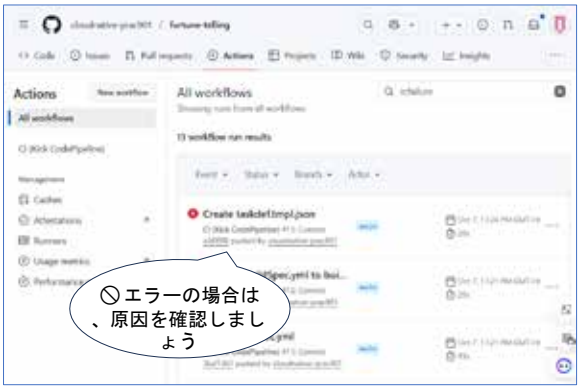


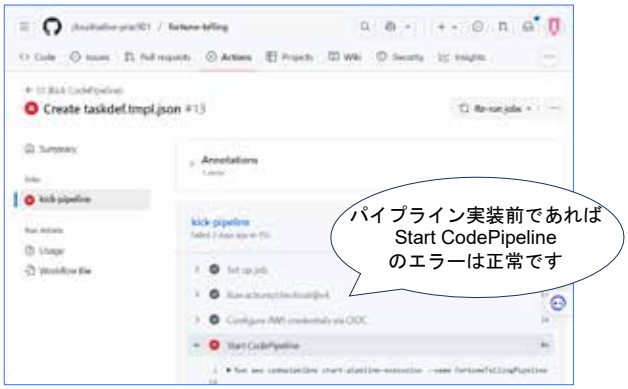
fortune-tellingリポジトリ

演習4　総合演習

GitHub準備　動作確認

- ci.yml登録後にファイルをpushするとGitHub Actionsが動作します。
  動作結果からAWSのOIDC認証が通ったことを確認できれば、GitHubの準備は完了です。
  - fortune-tellingリポジトリ > Actions　を選択し、**workflowの実行結果**を確認しましょう。
    - 「Configure AWS credentials via OIDC」の正常終了✅　＝　OIDCの認証が通った　ことになります。
    - CodePipeline実装前であれば、「Start CodePipeline」がエラー🚫なのは正常です。





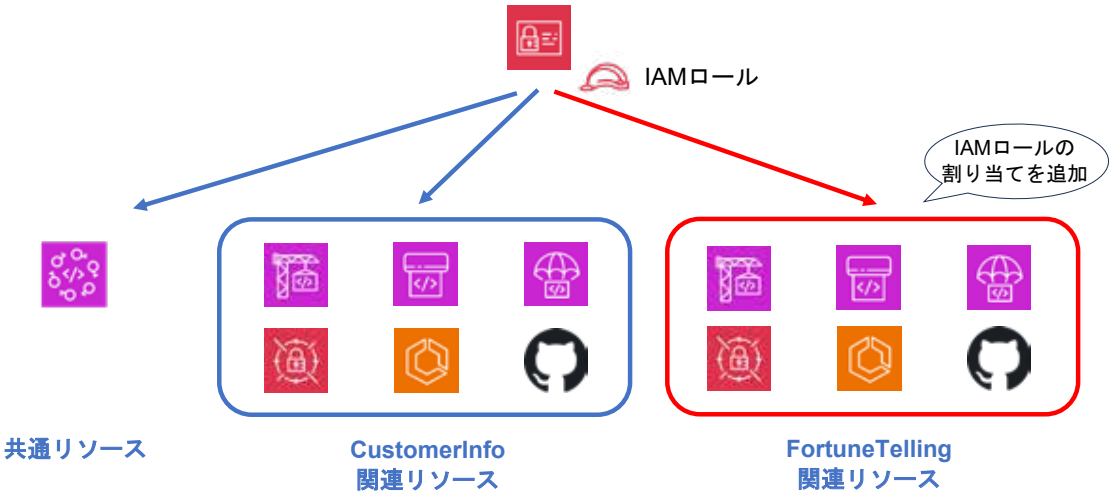
演習4

総合演習　ーCI/CDパイプライン構築ー

演習4　総合演習

CI/CDパイプライン構築　IAM

- おみくじ占いアプリケーションの実装に伴いリポジトリやパイプラインが追加となるため、各IAMロールを割り当てるリソースを増やしてください。各ロールに紐づけるポリシーは変更ありません。



演習4　総合演習

CI/CDパイプライン構築　IAM

- おみくじ占いアプリケーションの実装に伴いリポジトリやパイプラインが追加となるため、各IAMロールを割り当てるリソースを増やしてください。各ロールに紐づけるポリシーは変更ありません。

IAM設定値（1/2）

付与サービス	ロール名	AssumePrincipal	用途	権限（ポリシー内容）
CodeBuild	CodeBuildServiceRole	codebuild.amazonaws.com	CodeBuildがECRにpush / Logs出力 / S3のArtifacts参照 / kmsの暗号化・復号化を利用する	- logs:*（Logs出力） - ECR認証トークン取得 ecr:GetAuthorizationToken - ECR操作(対象Repoを限定 - ecrRepoArn) ecr:BatchCheckLayerAvailability/InitiateLayerUpload/ UploadLayerPart/CompleteLayerUpload/PutImage/ BatchGetImage/GetDownloadUrlForLayer - s3:GetObject/PutObject/GetBucketLocation/ListBucket - kms:Decrypt/Encrypt/GenerateDataKey*/DescribeKey
CodePipeline	CodePipelineServiceRole	codepipeline.amazonaws.com	Source/Build/Deploy をオーケストレーション S3のアーティファクト操作 CodeConnectionsの利用	- CodeBuild / CodeDeploy起動 codebuild:StartBuild codedeploy:CreateDeployment/Get*/ RegisterApplicationRevision - ロール譲歩 iam:PassRole (Build/Deployロールのみ) - CodeConnectionsの利用許可 codestar-connections:UseConnection - s3:GetObject/PutObject/GetObjectVersion/ListBucket - kms:Decrypt/Encrypt/GenerateDataKey*/DescribeKey

演習4　総合演習

CI/CDパイプライン構築　IAM

- おみくじ占いアプリケーションの実装に伴いリポジトリやパイプラインが追加となるため、各IAMロールを割り当てるリソースを増やしてください。各ロールに紐づけるポリシーは変更ありません。

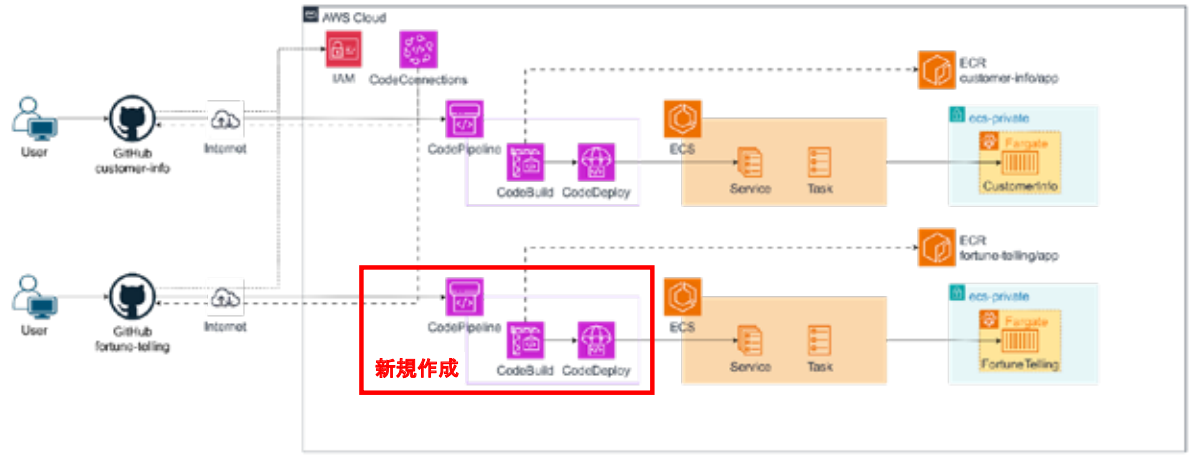
IAM設定値（2/2）

付与サービス	ロール名	AssumePrincipal	用途	権限（ポリシー内容）
CodeDeploy	CodeDeployServiceRole	codedeploy.amazonaws.com	ECS Blue/Green デプロイを実行	- AWSCodeDeployRoleForECS（AWS管理ポリシー）
GitHub (GitHub Actions)	GitHubOIDCRole	OIDC Provider (token.actions.githubusercontent.com)	GitHub ActionsからAWSを操作（Pipeline起動など）	- パイプライン実行(対象Pipelineを指定) codepipeline:StartPipelineExecution
ECS タスク (Fargate)	EcsTaskExecutionRole	ecs-tasks.amazonaws.com	ECSのタスク起動時に ECRからのイメージPull / Logsを出力	- AWS 管理ポリシー service-role/AmazonECSTaskExecutionRolePolicy
ECS タスク (アプリ)	AppTaskRole	ecs-tasks.amazonaws.com	アプリの処理でAWS リソースにアクセスする際の実行ロール	- 特になし
Secrets Manager	—（シークレットリソース）	—	DB認証情報を格納するシークレットリソース。必要に応じて、AppTask、EcsTaskExecutionロールに権限を付与	- secretsmanager:GetSecretValue AppTaskRole / EcsTaskExecutionRole に付与

演習4　総合演習

CI/CDパイプライン構築　パイプライン

- おみくじ占いアプリケーション（FortuneTelling）用のCI/CDパイプラインを構築してください。
  Pipeline/Build/Deployスタックを追加しますが、各スタックの構成は変更ありません。



演習4　総合演習

CI/CDパイプライン構築　パイプライン

- おみくじ占いアプリケーションに伴う、CI/CDパイプラインに関連するパラメータを確認しましょう。
  変更となるパラメータを抜粋して記載します。なおパイプラインの動作内容は変更しません。

CodeDeploy

項目	パラメータ	値	役割
スタック	-	Deploy2Stack	新規アプリケーション用DeployStack
CodeDeploy	applicationName	'FortuneTellingEcsApp'	CodeDeployアプリケーション名
	deploymentGroupName	'FortuneTellingDG'	CodeDeployデプロイメントグループ名
ECS	clusterName	ecs2.cluster.clusterName	ECSクラスター名
	serviceName	ecs2.service.serviceName	ECSサービス名
ALB	prodListenerArn	alb2.listenerProd.listenerArn	FortuneTellingAlb　本番リスナーARN
	testListenerArn	alb2.listenerTest.listenerArn	FortuneTellingAlb　テストリスナーARN
	tgBlueName	alb2.tgBlue.targetGroupName	FortuneTellingAlb　ターゲットグループ(Blue)
	tgGreenName	alb2.tgGreen.targetGroupName	FortuneTellingAlb　ターゲットグループ(Green)

演習4　総合演習

CI/CDパイプライン構築　パイプライン

■ おみくじ占いアプリケーションに伴う、CI/CDパイプラインに関連するパラメータを確認しましょう。
変更となるパラメータを抜粋して記載します。なおパイプラインの動作内容は変更しません。

CodeBuild

項目	パラメータ	値	役割
スタック	-	Build2Stack	新規アプリケーション用BuildStack
CodeBuild	projectName	'fortune-telling-app'	CodeBuildビルドプロジェクト名
ECR	ecrRepoName	'customer-info/app'	ECR リポジトリ名

CodePipeline

項目	パラメータ	値	役割
スタック	-	Pipeline2Stack	新規アプリケーション用PipelineStack
CodePipeline	pipelineName	'FortuneTellingPipeline'	パイプライン名
GitHub	gitHubOwner	'<xxx>'	GitHubアカウント
	gitHubRepo	'fortune-telling'	GitHubリポジトリ
ECR	ecrRepoName	'fortune-telling/app'	ECR リポジトリ名
CodeDeploy	ecsAppName	'FortuneTellingEcsApp'	ECSアプリケーション
	ecsDeploymentGroupName	'FortuneTellingDG'	ECSデプロイメントグループ
RDS	dbSecretArn	rds.fortuneAppSecret.secretArn	RDS アプリケーションシークレットのARN

演習4　総合演習

CI/CDパイプライン構築　コード作成

■ AWS CDKを利用して、**新規アプリケーション用のパイプラインを構築**してください。

■ コーディングのポイント

bin/ecs-demo.ts

おみくじ占いアプリケーション用パイプラインのスタックを追加してください。

- CodeBuild/CodeDeploy/CodePipelineの3スタックを追加
- おみくじ占いアプリケーションのGitHubリポジトリやパイプラインの追加に伴いIamスタックに渡すパラメータを追加

lib/iam-stack.ts

既存ロールを付与する対象リソースを追加してください。

- 各ロールに紐づけるポリシーは変更なし
- 各ロールを付与する対象リソースに、新規アプリケーションのGitHubリポジトリやパイプラインを追加
- 新規アプリケーションのGitHubリポジトリやパイプラインは、外部パラメータとして追加で受け取る

lib/build2-stack.ts

新規アプリケーション用のビルドスタックを新規作成してください。

- コード構成は既存のbuild-stack.tsから変更なし
- スタック名、ビルドプロジェクト名、ECRリポジトリ名などは要修正

lib/deploy2-stack.ts

新規アプリケーション用のデプロイスタックを新規作成してください。

- 既存のdeploy-stack.tsをベースにしてください。
- テスト用ビルドプロジェクトの定義があれば、削除してください。
- スタック名は要修正

lib/pipeline2-stack.ts

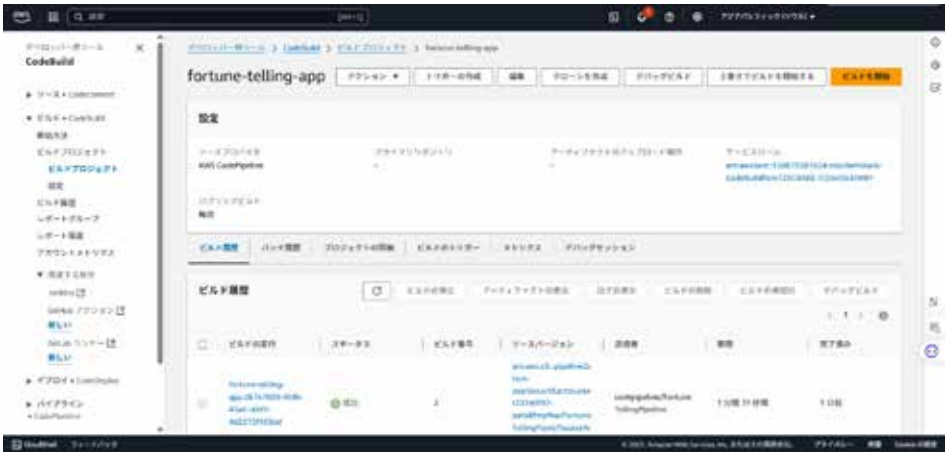
新規アプリケーション用のパイプラインスタックを新規作成してください。

- 既存のpipeline-stack.tsをベースにしてください。
- テストフェーズの定義があれば、削除してください。
- スタック名、パイプライン名、ビルドプロジェクト名は要修正

演習4　総合演習

CI/CDパイプライン構築　動作確認

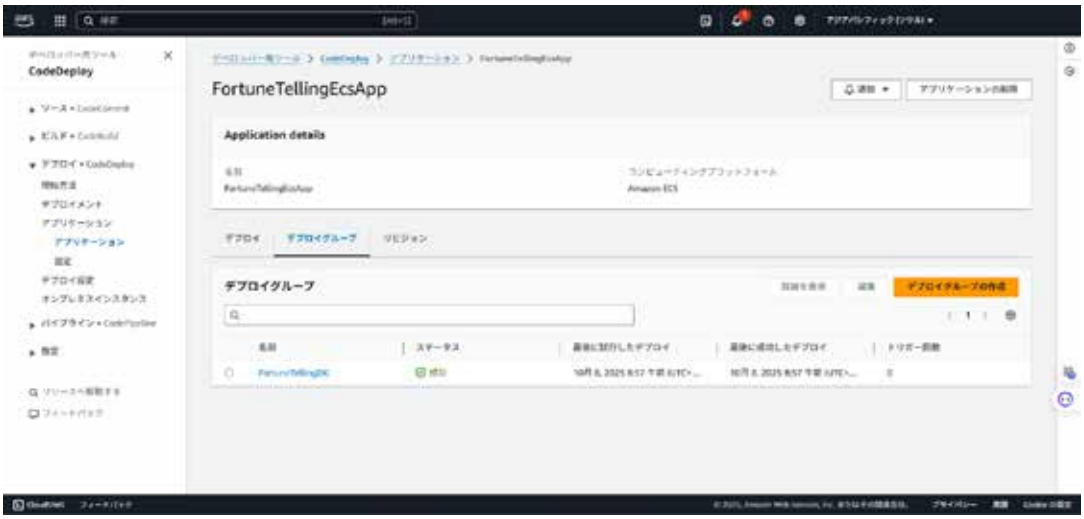
- Build2Stackの動作確認をします。CodeBuildにfortune-tellling-appプロジェクトが作成されていることを確認してください。
  - CodeBuild > ビルドプロジェクト > fortune-telling-app　で確認できます。
  - ビルドプロジェクト自体の動作確認は、Pipeline実装時に合わせて確認します。



演習4　総合演習

CI/CDパイプライン構築　動作確認

- 続いてDeployStackの動作確認をします。
  CodeDeployのアプリケーションにFortuneTellingEcsStackが登録されていることを確認してください。



演習4　総合演習

CI/CDパイプライン構築　動作確認

- CDKからPipeline2Stackを実行しましょう。
  途中でCodeDeployの手動承認が発生しますので注意してください。

- CodePipeline > パイプライン から実行中の「FortuneTellingPipeline」を選択します。
  パイプラインがSource→Build→Deployと進んだら、
  実行中のCodeDeployを確認しましょう。





演習4　総合演習

CI/CDパイプライン構築　動作確認

- CodePipelineの実行完了を確認し、おみくじ占いアプリケーションが正常動作すれば、演習4は完了です。合わせて、顧客情報表示機能もB/Gデプロイでき、動作することを見ておきましょう。



Top画面
URI：<ALB_DNS>/fortune

結果画面
URI：<ALB_DNS>/result

演習4　総合演習

合格判定基準

■ 演習4 後半の合格判定基準は以下の通りです。

GitHub準備

- 新規リポジトリ（fortune-telling）が生成され、各ディレクトリ / ファイルの登録が済んでいる
- おみくじ占いアプリケーションのコード（app.py）が実装できている
- GitHub Actionsが起動し、OIDC認証が通り、CodePipelineをキックできている

CI/CDパイプライン構築

- Buildにおみくじ占い用のビルドプロジェクトが新しく作成されている
- CodeDeployにおみくじ占い用のアプリケーション、デプロイメントグループが新しく作成されている
- FortuneTellingPipelineが生成され、Source/Build/Deployステージの走行完了している
- B/Gデプロイが完了し、占いサイトが正しく表示・機能している
- おみくじ占いアプリケーションが起動した状態で、既存の顧客情報表示機能のB/Gデプロイができる

確認テスト

演習1-0 確認テスト

【問題1】クラウドネイティブな開発で重視される概念として最も適切なものはどれ？

A. 手作業による構築
B. コード化と自動化
C. 物理サーバの冗長化
D. 固定インフラの維持

【問題2】AWS CDKの主な目的はどれか？

A. AWSリソースをコードで定義・自動構築
B. WebアプリのUI作成
C. GitHubリポジトリの作成
D. CloudFrontのキャッシュ制御

【問題3】 CDK初期化後に生成されるスタック名として**自動的に作成されるもの**は？

A. CloudFormationStack
B. CDKToolkit
C. BootstrapStack
D. DefaultStack

【問題4】cdk bootstrap コマンドの役割として正しいのは？

A. CDKアプリをコンパイルする
B. IAMユーザを生成する
C. CloudFormation実行用の基盤スタックを作成する
D. VPCを初期構築する

【問題5】CDKコマンドで「デプロイ可能なスタック一覧」を表示するコマンドは？

A. cdk list
B. cdk diff
C. cdk synth
D. cdk deploy

【問題6】本演習で扱う三層アーキテクチャ構成として正しい組み合わせは？

A. S3＋Lambda＋DynamoDB
B. Route53＋EC2＋CloudWatch
C. API Gateway＋Lambda＋S3
D. ALB＋ECS（Fargate）＋RDS

==========

【問題1】
　正解：B

【問題2】
　正解：A

【問題3】
　正解：B

【問題4】
　正解：C

【問題5】
　正解：A

【問題6】
　正解：D

演習1-1 確認テスト

【問題1】AWS CDKを利用する主な目的はどれか？

A. ネットワーク通信を監視する
B. AWSリソースをコードで定義し再利用する
C. アプリのUI設計を行う
D. コスト削減レポートを自動生成する

【問題2】NAT Gatewayを作成しない場合、Privateサブネットに設定すべき型は？

A. PRIVATE_WITH_EGRESS
B. PRIVATE_WITH_NAT
C. PRIVATE_ISOLATED
D. PUBLIC

【問題3】VPCエンドポイントを利用することで実現できることはどれか？

A. インターネット経由の高速通信
B. パブリックアクセス制御の解除
C. AWSサービスへのプライベート接続
D. CloudFront経由でのキャッシュ利用

【問題4】セキュリティグループで「allowAllOutbound: false」を指定する主な目的は？

A. すべての通信を許可する
B. インバウンド通信を無効化する
C. VPCエンドポイントを強制する
D. 通信を制限し最小権限化する

【問題5】Fargateで「TargetType:IP」を指定するのはなぜか？

A. FargateタスクがIP単位で管理されるため
B. EC2インスタンスを直接指定するため
C. Lambdaをターゲットにするため
D. DNS名を指定するため

【問題6】WAFとALBの関連付けで注意すべき点として正しいものは？

A. 複数のWebACLを1つのALBに紐づけ可能
B. WebACLはALB作成前にのみ適用できる
C. 1つのALBには1つのWebACLしか関連付けできない
D. WebACLは自動ですべてのALBに適用される

【問題1】
正解：B

【問題2】
正解：C

【問題3】
正解：C

【問題4】
正解：D

【問題5】
正解：A

【問題6】
正解：C

演習1-2 確認テスト

【問題1】ECRリポジトリをCDKで作成する際、imageScanOnPush: true の目的は？

A. push時に自動スキャンで脆弱性を検知するため
B. push後にイメージを削除するため
C. pull時の検証をスキップするため
D. CloudWatchログに自動登録するため

【問題2】’aws ecr get-login-password | docker login’の目的は？

A. Docker Hubへログイン
B. AWS CLIを再認証
C. ECRへのDockerログイン認証
D. IAMユーザーを作成

【問題3】 Dockerタグ付けで $ECR_URL/customer-info/app:v0.1.1 と指定する理由は？

A. 複数リージョンにデプロイするため
B. ECRリポジトリとバージョンを識別するため
C. push先をS3に変更するため
D. 複数タスクで共有するため

【問題4】サービス定義でhealthCheckGracePeriod: Duration.seconds(60)を指定する理由は？

A. 起動直後のヘルスチェック失敗を防ぐため
B. ALB側の監視を無効化するため
C. CPUスロットリングを回避するため
D. 再デプロイを高速化するため

【問題5】ALBのターゲットグループにECSを紐づける目的として正しいものは？

A. ECSがALBを監視するため
B. ALBからのリクエストをECSコンテナへ転送するため
C. ログをALBへ送信するため
D. Fargateを自動スケールさせるため

【問題6】CloudShellからALB経由でアプリにアクセスし、200が返ることを確認する理由は？

A. ECSログを取得するため
B. ALBとECSの疎通確認を行うため
C. ALBの設定を削除するため

D. タグの整合性を確認するため

==========

【問題1】
正解：A

【問題2】
正解：C

【問題3】
正解：B

【問題4】
正解：A

【問題5】
正解：B

【問題6】
正解：B

演習1-3 確認テスト

【問題1】RDS構築時に「Secrets Manager」を使う主な理由は？

A. パスワードをコードに直書きするため
B. DB接続情報をセキュアに管理するため
C. ユーザー情報を暗号化せずに保存するため
D. ログイン制御を簡略化するため

【問題2】CDKで作成するadminユーザーの用途として正しいものは？

A. 初期データベース構築や管理操作用
B. アプリケーションアクセス用
C. CI/CDパイプライン専用
D. ALB疎通確認用

【問題3】 app_userのシークレット作成時にgenerateSecretStringを使う理由は？

A. 固定文字列を使うため
B. パスワードを自動生成して安全に保管するため
C. Secrets Managerを無効化するため
D. 複数リージョンに複製するため

【問題4】本番環境でRDSインスタンスの設定で、removalPolicy: DESTROYを使う際の注意点は？

A. 削除防止が自動で有効になる
B. データが永続化される
C. 自動バックアップが無効化される
D. スタック削除時にDBデータも削除される

【問題5】顧客データを追加するSQL文として正しいものは？

A. INSERT INTO customers VALUES ('Taro', 30, 'taro@example.com');
B. ADD INTO customers ...
C. INSERT INTO customers (name, age, email) VALUES ('Taro', 30, 'taro@example.com');
D. UPDATE customers ...

【問題6】app_userで削除権限を持たせないようにするにはどの権限セットが適切？

A. GRANT SELECT, INSERT, UPDATE ON customer_info.* TO 'app_user';
B. GRANT ALL PRIVILEGES ON customer_info.* TO 'app_user';
C. GRANT DELETE ON customer_info.* TO 'app_user';
D. GRANT CREATE, DROP ON customer_info.* TO 'app_user';

==========

【問題1】

正解：B

【問題2】

正解：A

【問題3】

正解：B

【問題4】

正解：D

【問題5】

正解：C

【問題6】

正解：A

演習1-4 確認テスト

【問題1】CI/CDパイプライン全体に共通する設計思想として最も正しいものは？

A. 手動実行による制御を強化する
B. 手動レビューを必須とする
C. IAMロールを共有して簡素化する
D. コード変更からデプロイまでの流れを自動化・標準化する

【問題2】GitHub OIDCロールの設定目的は？

A. GitHubからAWSを安全に操作できるようにする
B. CloudFormationを高速化する
C. Secrets Managerを削除する
D. ECSのメトリクスを送信する

【問題3】 CodeBuildで参照される設定ファイルはどれ？

A. .gitignore
B. buildspec.yml
C. ci.yml
D. taskdef.tmpl.json

【問題4】iam:PassRole ポリシーの目的は？

A. GitHub接続を切断するため
B. IAMロールを削除するため
C. ユーザー作成を自動化するため
D. ロールを他サービスに委譲するため

【問題5】GitHub Actionsのci.ymlにおけるon: push: branches: [main]の意味は？

A. すべてのブランチで実行
B. GitHubのpull request時に実行
C. mainブランチへのpush時にのみ実行
D. CodeDeployの完了後に実行

【問題6】CodeDeployがECSサービスを切り替える際、ターゲットグループが2つ必要な理由は？

A. テスト用と本番用で別々にロードバランサを管理するため
B. Blue環境とGreen環境を同時に存在させて安全に切替えるため
C. セキュリティグループを分離するため
D. ALBのゾーン冗長性を確保するため

==========

【問題1】
　正解：D

【問題2】
　正解：A

【問題3】
　正解：B

【問題4】
　正解：D

【問題5】
　正解：C

【問題6】
　正解：B

演習2 確認テスト

【問題1】HTTPS対応のために追加利用するAWSサービスは？

A. AWS Certificate Manager
B. AWS Shield
C. AWS Secrets Manager
D. Amazon Inspector

【問題2】ACMで証明書を発行する際、DNS検証を採用する主な理由は？

A. 検証を自動化できるから
B. セキュリティ強度が低いため
C. HTTP検証より遅いから
D. AWS CLIでしか設定できないから

【問題3】HTTPSリスナーのsslPolicyにRECOMMENDED_TLSを設定する目的は？

A. 暗号化を無効化するため
B. 独自証明書を強制するため
C. 最新かつ推奨の暗号スイート構成を自動適用するため
D. S3との通信に最適化するため

【問題4】HTTPからHTTPSへ移行するにあたり、ALBのセキュリティグループに追加するべきインバウンドルールは？

A. TCP 22
B. TCP 443
C. UDP 53
D. TCP 25

【問題5】本番環境でHTTP(80)を閉じることが推奨される理由は？

A. パフォーマンスが低下するため
B. セキュリティリスクを減らすため
C. ECSコンテナが80を使うため
D. RDSと競合するため

==========

【問題1】

正解：A

【問題2】

正解：A

【問題3】
正解：C

【問題4】
正解：B

【問題5】
正解：B

演習3 確認テスト

【問題1】テストステージ追加後のパイプライン構成として正しいものは？

A. Source → Build → Deploy → Test
B. Source → Test → Build → Deploy
C. Source → Build → Test → Deploy
D. Test → Source → Build → Deploy

【問題2】buildspec.test.ymlの主な役割は？

A. ECSの起動設定を定義する
B. CodePipelineの承認ルールを設定する
C. RDSスキーマを作成する
D. pytestの実行とレポート出力を定義する

【問題3】CodeBuildのレポート機能を使うために追加されたIAM権限は？

A. iam:PassRole
B. codebuild:StartBuild
C. codebuild:*Report*、codebuild:BatchPut*
D. codedeploy:CreateDeployment

【問題4】Test用CodeBuildプロジェクトでprivileged: falseとする理由は？

A. セキュリティ上の最小権限運用のため
B. Dockerビルドを高速化するため
C. pytestがrootで動作しないため
D. RDSアクセスを制限するため

【問題5】テストに失敗した場合、パイプラインはどう動作する？

A. Testステージをスキップして続行
B. Buildステージを再試行
C. Testステージでパイプラインが停止
D. ECSがロールバックする

【問題6】pytestの実行結果をCodePipeline全体の品質ゲートとして活用する理由は？

A. 手動確認を省くためだけ
B. ECSのスケーリングを最適化するため
C. RDS更新を高速化するため
D. デプロイ前に自動的に異常を検知し、次ステージへの進行を防ぐため

==========

【問題1】
正解：B

【問題2】
正解：D

【問題3】
正解：C

【問題4】
正解：A

【問題5】
正解：C

【問題6】
正解：D

演習4 確認テスト

【問題1】CDKで複数アプリを展開する場合に重要な考え方は？

A. 全リソースを共通化する
B. 各スタックを一つに統合
C. アプリ単位で独立性を保ちながら共通リソースを再利用
D. アプリケーション単位で全スタックを追加

【問題2】複数アプリケーションで共有されるAWSリソースとして正しいものは？

A. ALB
B. ECSタスク
C. VPC・エンドポイント
D. CodeBuild

【問題3】 bin/ecs-demo.tsにおいて複数アプリを管理するために推奨される変更点は？

A. アプリケーション単位で各スタックを用意する
B. パイプラインは1つのスタックにまとめる
C. ECSサービスを共有化する
D. 各スタックのパラメータを配列化して渡すよう修正

【問題4】Secrets Managerの設定で正しい説明はどれ？

A. adminユーザーは共有し、appユーザーはアプリ単位で登録
B. アプリ単位でappユーザーを登録
C. 1つのシークレットに複数アプリの情報を混在
D. Secrets Managerは不要

【問題5】アプリケーションの追加に伴う、IAMロールの変更に関する説明で正しいものは？

A. 新しいアプリケーション用にIAMロールをすべて再定義
B. 既存ロールを流用し、対象リソースのみ追加
C. 対象リソースを全て（＊）にし、適用範囲を拡大
D. 全ロールを共通ポリシー化

【問題6】おみくじ占いアプリケーション用のパイプラインに対する主な対応内容として正しいものは？

A. 既存パイプライン構成を流用し、リソースパラメータを変更する
B. 既存パイプラインを共有して、両アプリケーションのB/Gデプロイを可能にする

C. GitHubリポジトリを共有し、新しいパイプラインからも参照可能にする
D. デプロイステージを削除する

==========

【問題1】
正解：C

【問題2】
正解：C

【問題3】
正解：D

【問題4】
正解：A

【問題5】
正解：B

【問題6】
正解：A

令和７年度「専門職業人材の最新技能アップデートのための専修学校リカレント教育（リ・スキリング）推進事業」
情報技術者の技能アップデートのためのリカレント教育推進事業

クラウドネイティブシステム開発資料教材
問題編

令和８年２月

一般社団法人全国専門学校情報教育協会

〒164-0003 東京都中野区東中野 1-57-8 辻沢ビル３F

電話：03-5332-5081 FAX.03-5332-5083

●本書の内容を無断で転記、掲載することは禁じます。