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
解答編

本クラウドネイティブシステム開発資料教材解答編は、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1-1　解答編

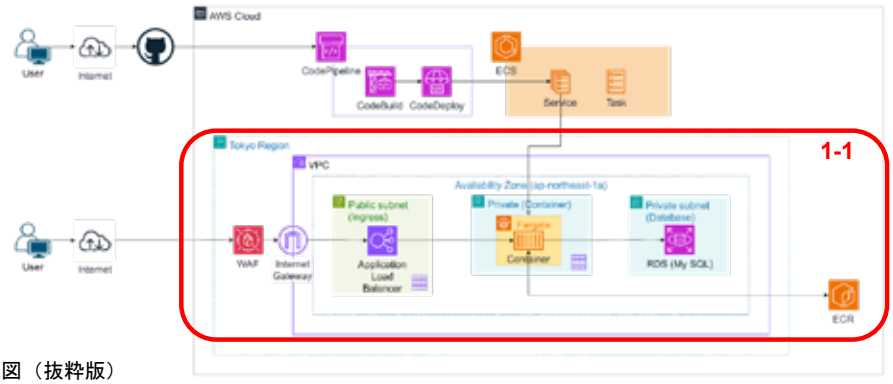
ネットワーク構築

演習1-1　ネットワーク構築　解答編

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　解答編

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。

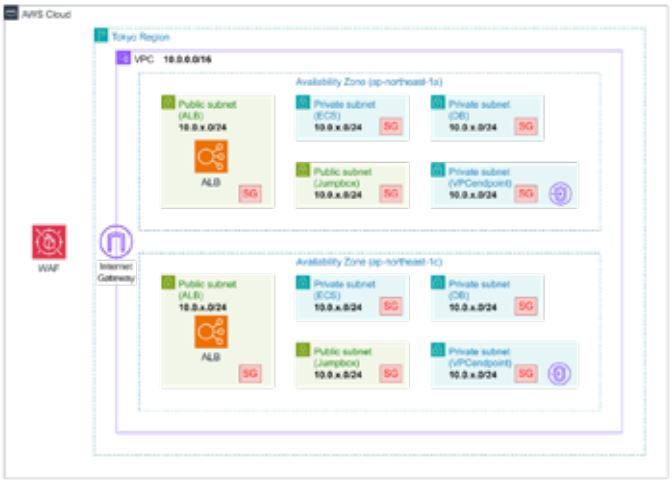
Step1　**VPC、サブネット**
複数AZにまたがるネットワーク基盤

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　解答編

ネットワーク構築 – VPC / サブネット / SG

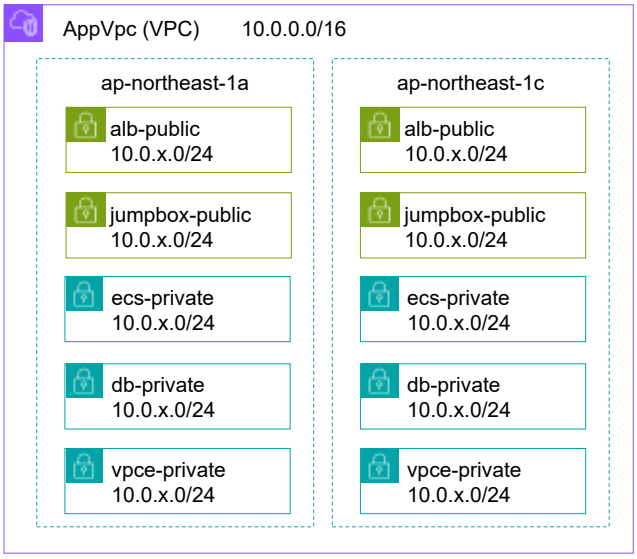
演習1-1　ネットワーク構築　解答編

Step1：VPC　設計

■ VPC/AZの設定値は以下の通りです。

VPC/AZ

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



演習1-1　ネットワーク構築　解答編

Step1：サブネット　設計

■ サブネットの設計値は以下の通りです。

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

サブネット種別	個数	Public/Private	用途
alb-public	2個	Public	外部からのリクエスト受付用
jumpbox-public	2個	Public	踏み台アクセス用
ecs-private	2個	Private	ECSコンテナ配置用
db-private	2個	Private	RDS配置用
vpce-private	2個	Private	VPCエンドポイント配置用

演習1-1　ネットワーク構築　解答編

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の設定値は以下の通りです。

通信要件

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

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　解答編

Step1：VPC/AZ/サブネット/SG　CDKコード

■ VPC/AZ/サブネット/SGの設計値を基に、CDKプロジェクトのコードを作成します。

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

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

CDKディレクトリ構成

NetStack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

演習1-1　ネットワーク構築　解答編

Step1：VPC/AZ/サブネット/SG　CDKコード

- bin/ecs-demo.tsのコードは以下の通りです。

```
#!/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 });
```

演習1-1　ネットワーク構築　解答編

Step1：VPC/AZ/サブネット/SG　CDKコード

- lib/net-stack.tsのコードの全体構成は以下の通りです。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	クラス宣言	他スタック向けにプロパティの公開
3	スタック初期化	NetStackを初期化
4	VPC＋サブネット作成	VPC、サブネット10個を構築
5	セキュリティグループ作成	セキュリティグループを5つ作成
6	セキュリティグループ間の通信ルール	セキュリティグループ間の通信ルールを追加
7	出力	設定値の出力

演習1-1　ネットワーク構築　解答編

Step1：VPC/AZ/サブネット/SG　CDKコード

VPC/サブネット/SG作成　解答例（1/4）

```
// 1. 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';

// 2. 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;
  public readonly vpceSg: ec2.SecurityGroup;
  public readonly albSg: ec2.SecurityGroup;
  public readonly dbSg: ec2.SecurityGroup;
  public readonly jumpSg: ec2.SecurityGroup;

  // 3. スタック初期化
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
```

ポイント

1. インポート
- 利用するライブラリを漏れなく明記してください。

2. クラス宣言
- VPCやSGは他スタックで利用するため、プロパティを外部に公開します。

演習1-1　ネットワーク構築　解答編

Step1：VPC/AZ/サブネット/SG　CDKコード

VPC/サブネット/SG作成　解答例（2/4）

```
    // 4. VPC＋サブネット作成
    this.vpc = new ec2.Vpc(this, 'AppVpc', {
      ipAddresses: ec2.IpAddresses.cidr('10.0.0.0/16'),
      maxAzs: 2,
      subnetConfiguration: [
        { name: 'alb-public',
          subnetType: ec2.SubnetType.PUBLIC, cidrMask: 24
        },
        { name: 'jumpbox-public',
          subnetType: ec2.SubnetType.PUBLIC, cidrMask: 24
        },
        { name: 'ecs-private',
          subnetType: ec2.SubnetType.PRIVATE_ISOLATED, cidrMask: 24
        },
        { name: 'vpce-private',
          subnetType: ec2.SubnetType.PRIVATE_ISOLATED, cidrMask: 24
        },
        { name: 'db-private',
          subnetType: ec2.SubnetType.PRIVATE_ISOLATED, cidrMask: 24
        },
      ],
      natGateways: 0,
    });
```

ポイント

4. VPC+サブネット作成
- VPCとサブネットをまとめて作成できます。
- maxAzs:2
  AZは少なくとも2つが望ましいです。
- natGateway:0
  NAT-GWは今回作成しません。
- NAT-GWを利用しないprivateサブネットには、subnetType:PRIVATE_**ISOLATED**を設定しましょう。

演習1-1　ネットワーク構築　解答編

Step1：VPC/AZ/サブネット/SG　CDKコード

VPC/サブネット/SG作成　解答例（3/4）

```
    // 5. セキュリティグループの作成
    this.albSg = new ec2.SecurityGroup(this, 'AlbSg', {              // alb-public用SG
      vpc: this.vpc,
      description: 'Security Group for ALB',
      allowAllOutbound: false,
    });
    this.ecsSg = new ec2.SecurityGroup(this, 'EcsSg', {              // ecs-private用SG
      vpc: this.vpc,
      description: 'Security Group for ECS Service',
      allowAllOutbound: true,
    });
    this.dbSg = new ec2.SecurityGroup(this, 'DbSg', {                // db-private用SG
      vpc: this.vpc,
      description: 'Security Group for RDS',
      allowAllOutbound: false,
    });
    this.jumpSg = new ec2.SecurityGroup(this, 'JumpSg', {            // jumpbox-public用SG
      vpc: this.vpc,
      description: 'Security Group for JumpBox',
      allowAllOutbound: true,
    });
    this.vpceSg = new ec2.SecurityGroup(this, 'VpceSg', {            // vpce-private用SG
      vpc: this.vpc,
      description: 'Security Group for VPC Endpoints',
      allowAllOutbound: true,
    });
```

ポイント

5. セキュリティグループの作成

- allowAllOutBoundは全アウトバウンド通信を許可する設定です。
  アウトバウンドを最小限に絞る必要があるサブネットはfalseにして、アウトバウンド通信を個別設定するのが望ましいです。

演習1-1　ネットワーク構築　解答編

Step1：VPC/AZ/サブネット/SG　CDKコード

VPC/サブネット/SG作成　解答例（4/4）

```
    // 6. セキュリティグループ間の通信ルール
    // ALB: InternetからHTTP/HTTPSを許可
    this.albSg.addIngressRule(ec2.Peer.anyIpv4(), ec2.Port.tcp(80), 'Allow HTTP from
Internet');
    this.albSg.addEgressRule(this.ecsSg, ec2.Port.tcp(80), 'ALB-to-ECS');
    // ECS: ALBからHTTPアクセスを許可
    this.ecsSg.addIngressRule(this.albSg, ec2.Port.tcp(80), 'ALB-to-ECS');
    // DB: ECSおよびJumpBoxからMySQLアクセスを許可
    this.dbSg.addIngressRule(this.ecsSg, ec2.Port.tcp(3306), 'ECS-to-DB');
    this.dbSg.addIngressRule(this.jumpSg, ec2.Port.tcp(3306), 'JumpBox-to-DB');
    // VPC Endpoint: ECSからHTTPSアクセスを許可
    this.vpceSg.addIngressRule(this.ecsSg, ec2.Port.tcp(443), 'ECS-to-VPCE');

    // 7. 出力
    new CfnOutput(this, 'VpcId',   { value: this.vpc.vpcId });
    new CfnOutput(this, 'EcsSgId',   { value: this.ecsSg.securityGroupId });
    new CfnOutput(this, 'AlbSgId',   { value: this.albSg.securityGroupId });
    new CfnOutput(this, 'JumpSgId',  { value: this.jumpSg.securityGroupId });
    new CfnOutput(this, 'DbSgId',    { value: this.dbSg.securityGroupId });
    new CfnOutput(this, 'VpceSgId',  { value: this.vpceSg.securityGroupId });
  }
}
```

ポイント

6. セキュリティグループ間の通信ルール

- 通信要件で整理した個別通信ルールを最低限に漏れなく設定する必要があります。
- プロトコルの変更（例えばHTTP→HTTPS）時は、許可するプロトコルを追加するなどしてください。
- インバウンド通信はaddIngressRule、アウトバウンド通信はaddEgressRuleで設定できます。

演習1-1　ネットワーク構築　解答編

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）が自動生成されていることを確認

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

ルートテーブル、IGWは、定義しなくても自動生成されます

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

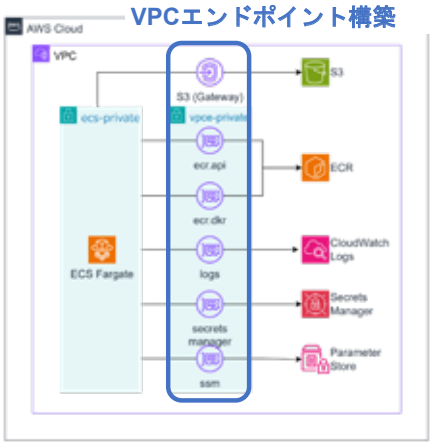
演習1-1 Step2　解答編

ネットワーク構築 – VPCエンドポイント

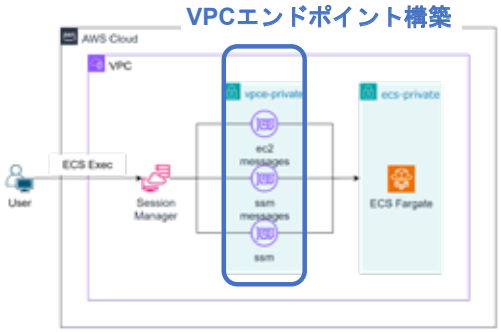
演習1-1　ネットワーク構築

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　解答編

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。
  S3のみGateway型、残りはInterface型です。

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　解答編

Step2：VPCエンドポイント　CDKコード

■ VPCエンドポイントの設計値を基に、CDKプロジェクトのコードを作成します。

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

```
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

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　解答編

Step2：VPCエンドポイント　CDKコード

■ bin/ecs-demo.tsのコードは以下の通りです。

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

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 });

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

ポイント

スタックの順番

- 前後関係のあるStackが出てくるため、順番を考慮してブロックを追加してください。
- 例：NetStack→VpceStack
  NetStackで構築したVPC、SGを利用して、VpceStackでVPCエンドポイントを作成します。

演習1-1　ネットワーク構築　解答編

Step2：VPCエンドポイント　CDKコード

■ lib/vpce-stack.tsのコードの全体構成は以下の通りです。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インターフェース定義	他スタックから渡されるパラメータ
3	スタック初期化	VpceStackを初期化、サブネットの選択
4	エンドポイント作成 (Gateway型)	Gateway型エンドポイントの作成（S3）
5	エンドポイント作成 (Interface型)	Interface型エンドポイントの作成 （ECR, Secrets Manager, Logs, SSM 等）

演習1-1　ネットワーク構築　解答編

Step2：VPCエンドポイント　CDKコード

VPCエンドポイント　解答例（1/4）

```
// 1. 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';

// 2. 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;
}

// 3. クラス宣言
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、jumpbox-publicを設定
    const vpceSubnets: ec2.SubnetSelection = {
      subnetGroupName: 'vpce-private', onePerAz: true,
    };
    const ecsSubnets: ec2.SubnetSelection = {
      subnetGroupName: 'ecs-private', onePerAz: true,
    };
```

ポイント

1. インポート

- 利用するライブラリを漏れなく明記しましょう。

2. インターフェース定義

- NetStackで作成した、VPC、SGなどを外部から読み込みます。

3. クラス宣言

- VPCエンドポイント作成時にサブネットを指定するため、ここでサブネットの定義をしておきます。
- onePerAzをセットすることで、一つのサブネットを指定可能となり、AZ内で重複エラーとなるのを防止できます。

演習1-1　ネットワーク構築　解答編

Step2：VPCエンドポイント　CDKコード

VPCエンドポイント　解答例（2/4）

```
const jumpSubnets: ec2.SubnetSelection = {
  subnetGroupName: 'jumpbox-public', onePerAz: true,
};

// 4. Gatewayエンドポイント作成（S3）
new ec2.GatewayVpcEndpoint(this, 'S3Gateway', {
  vpc,
  service: ec2.GatewayVpcEndpointAwsService.S3,
  subnets: [vpceSubnets, ecsSubnets, jumpSubnets],
});

// 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],
});

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

ポイント

4. Gatewayエンドポイント作成

- S3へのアクセスはGateway型エンドポイントで実現できます。
- vpce-private、ecs-private、jumpbox-publicからS3を見られるようにしておきます。

5. Interfaceエンドポイント作成

- ECSを利用するには複数のエンドポイントが必要です。Interface型が複数必要なので、漏れなく設定してください。

演習1-1　ネットワーク構築　解答編

Step2：VPCエンドポイント　CDKコード

VPCエンドポイント　解答例（3/4）

```
// Interface エンドポイント（Secrets Manager）
new ec2.InterfaceVpcEndpoint(this, 'SecretsManagerEp', {
  vpc,
  service: ec2.InterfaceVpcEndpointAwsService.SECRETS_MANAGER,
  subnets: vpceSubnets,
  securityGroups: [vpceSg],
});

// Interface エンドポイント（CloudWatchLogs）
new ec2.InterfaceVpcEndpoint(this, 'LogsEp', {
  vpc,
  service: ec2.InterfaceVpcEndpointAwsService.CLOUDWATCH_LOGS,
  subnets: vpceSubnets,
  securityGroups: [vpceSg],
});
```

ポイント

5. Interfaceエンドポイント作成（続き）

- エンドポイントの種類により、指定するサービス名が異なるため、入力ミスがないように注意してください。

演習1-1　ネットワーク構築　解答編

Step2：VPCエンドポイント　CDKコード

VPCエンドポイント　解答例（4/4）

```
    // Interface エンドポイント（SSM / ECS Exec）
    new ec2.InterfaceVpcEndpoint(this, 'SsmEp', {
      vpc,
      service: ec2.InterfaceVpcEndpointAwsService.SSM,
      subnets: vpceSubnets,
      securityGroups: [vpceSg],
    });

    new ec2.InterfaceVpcEndpoint(this, 'SsmMessagesEp', {
      vpc,
      service: ec2.InterfaceVpcEndpointAwsService.SSM_MESSAGES,
      subnets: vpceSubnets,
      securityGroups: [vpceSg],
    });

    new ec2.InterfaceVpcEndpoint(this, 'Ec2MessagesEp', {
      vpc,
      service: ec2.InterfaceVpcEndpointAwsService.EC2_MESSAGES,
      subnets: vpceSubnets,
      securityGroups: [vpceSg],
    });
  }
}
```

ポイント

SSM / Exec

- ECS execを利用するために利用する設定です。利用しない場合は不要です。
- 本運用を考えた場合、ECSコンテナ内に入れるルートはなくした方が良いでしょう。

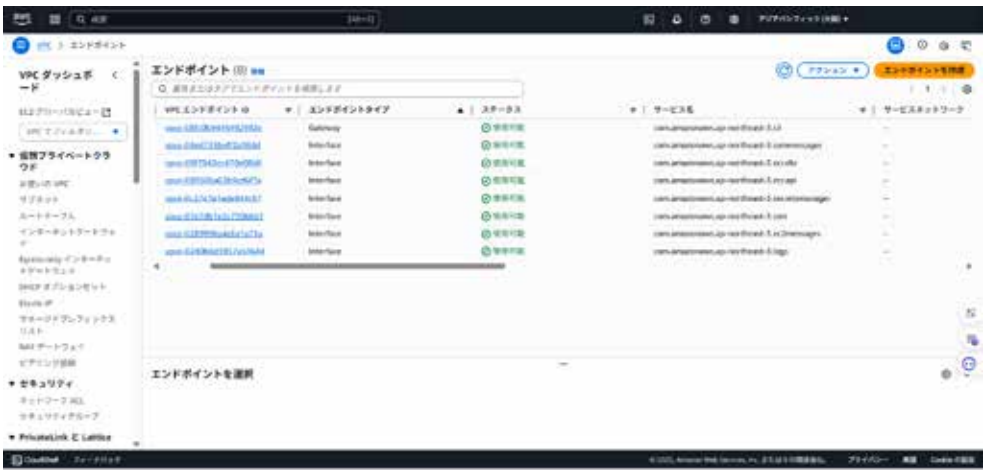
演習1-1　ネットワーク構築　解答編

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）



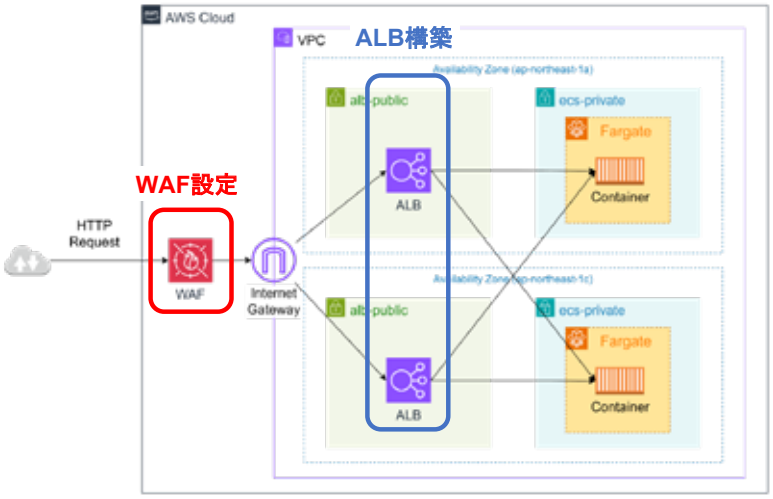
演習1-1 Step3　解答編

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

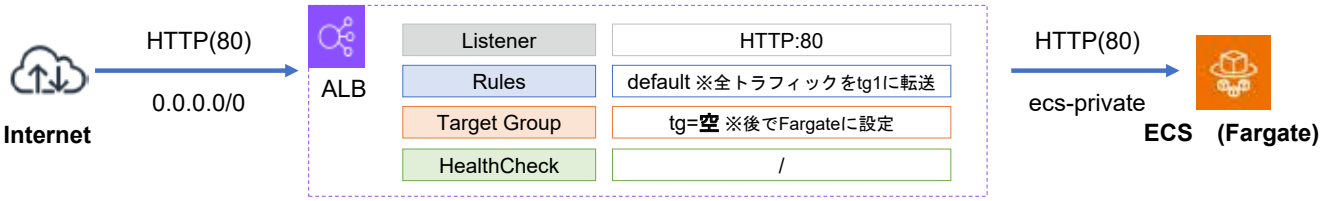
- 外部からのHTTPリクエストをECS Fargateで処理できるように、ロードバランサーとファイアウォールを設置します。Step3では、ロードバランサーはALB(L7)、ファイアウォールはWAFで設定します。



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。



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：ポート	HTTP：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1つずつ設定します。

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　解答編

Step3：ALB/WAF　CDKコード

■ ALB/WAFの設計値を基に、CDKプロジェクトのコードを作成します。

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

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

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
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　解答編

Step3：ALB/WAF　CDKコード

- bin/ecs-demo.tsのコードは以下の通りです。

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

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 });
```

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

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

演習1-1　ネットワーク構築　解答編

Step3：ALB/WAF　CDKコード

- lib/alb-stack.tsのコードの全体構成は以下の通りです。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インターフェース定義	他スタックから渡されるパラメータ
3	公開プロパティ	他スタックが参照するプロパティの公開
4	スタック初期化	VpceStackを初期化、サブネットの選択
5	ALB作成	ALBを構築
6	ターゲットグループ作成	ターゲットグループを設定（ターゲットは**空**）
7	HTTPリスナー作成	HTTPリスナーを作成
8	WAFルール作成 （BadBotブロック）	WAFのルールAを作成 User-Agentに”BadBot”を含むアクセスを遮断
9	WAFルール作成 （マネージドルール）	WAFのルールBを作成 AWSマネージド共通ルールの適用
10	WebACL作成	WAF WebACLを作成
11	ALBとWebACL関連付け	WebACLをALBにアタッチ
12	出力	設定値の出力

演習1-1　ネットワーク構築　解答編

Step3：ALB/WAF　CDKコード

ALB/WAF　解答例（1/6）

```
// 1. 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';

// 2. インターフェース定義
export interface AlbStackProps extends StackProps {
  vpc       : ec2.IVpc;
  albSg     : ec2.ISecurityGroup;
  albSubnets?: ec2.SubnetSelection;   // 指定が無ければ 'alb-public' グループを自動選択
}

// 3. 公開プロパティ
export class AlbStack extends Stack {
  public readonly albDnsName: string;
  public readonly targetGroup: elbv2.ApplicationTargetGroup;
```

ポイント

1. インポート

- elbv2、wafv2を忘れずにインポートしましょう。

2. インターフェース定義

- albSg、alb-publicのインストールが必要です。これらはNetStackで定義したものを取り込みます。

3. 公開プロパティ

- targetGroupは、ECS Fargateのタスクに紐づけるため、外部に公開する必要があります。

演習1-1　ネットワーク構築　解答編

Step3：ALB/WAF　CDKコード

ALB/WAF　解答例（2/6）

```
  // 4. スタック初期化
  constructor(scope: Construct, id: string, props: AlbStackProps) {
    super(scope, id, props);

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

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


    // 5. Application Load Balancerの作成
    const alb = new elbv2.ApplicationLoadBalancer(this, 'Alb', {
      vpc: props.vpc,
      internetFacing: true,
      securityGroup: props.albSg,
      loadBalancerName: 'CustomerInfoAlb',
      vpcSubnets: subnets,
    });
```

ポイント

4. スタック初期化

- if(!props.albSg) throw　を設定することで、albSgの注入失敗を早期検出します。
- ALBはalb-publicの各AZ1本にし、同一AZの重複を避けるようにしました。

5. ALB作成

- internetFacing:trueで外向けに設定します。
- 必要なら削除保護やアクセスログを有効化することで、監査トラブルシュートにも使うことが可能です。

演習1-1　ネットワーク構築　解答編

Step3：ALB/WAF　CDKコード

ALB/WAF　解答例（3/6）

```
// 6. ターゲットグループ作成
const tg = new elbv2.ApplicationTargetGroup(this, 'AlbTg', {
  vpc: props.vpc,
  port: 80,
  protocol: elbv2.ApplicationProtocol.HTTP,
  targetType: elbv2.TargetType.IP,
  healthCheck: { path: '/', interval: Duration.seconds(30) },
});

this.targetGroup = tg;

// 7. HTTPリスナーの作成
const listener = alb.addListener('HttpListener', {
  port: 80,
  protocol: elbv2.ApplicationProtocol.HTTP,
  defaultTargetGroups: [tg],
});
```

ポイント

6. ターゲットグループ作成

- FargateはIPで管理されるため、TargetType:IPを指定します。
- healthCheckのpathは”/”としていますが、アプリの作りに合わせてパスを設定してください。/healthcheckや/healthなどが一般的です。

7. HTTPリスナーの作成

- HTTP:80でリッスンするよう設定します。
- HTTPS対応する場合は、ACMの証明書の適用が必要です。演習2でHTTPS対応について学習します。

演習1-1　ネットワーク構築　解答編

Step3：ALB/WAF　CDKコード

ALB/WAF　解答例（4/6）

```
// 8. WAFルール作成（BadBotブロック）
const badBotRule: wafv2.CfnWebACL.RuleProperty = {
  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',
  },
};
```

ポイント

8. WAFルール作成（BadBotブロック）

- priorityはルールの優先度に応じて設定どの順番でルールを適用するか意図して決定します。
- actionはブロックとしているが、allow / countなど、ルール抵触時の挙動を指定可能です。
- visibilityConfigにて、CloudWatchにメトリクス連携が可能です。（任意）

演習1-1　ネットワーク構築　解答編

Step3：ALB/WAF　CDKコード

ALB/WAF　解答例（5/6）

```
// 9. WAFルール作成（AWSマネージドルール）
const awsManagedCommon: wafv2.CfnWebACL.RuleProperty = {
  name: 'AWSManagedCommonRuleSet',
  priority: 1,
  overrideAction: { none: {} },
  statement: {
    managedRuleGroupStatement: {
      vendorName: 'AWS',
      name: 'AWSManagedRulesCommonRuleSet',
    },
  },
  visibilityConfig: {
    cloudWatchMetricsEnabled: true,
    sampledRequestsEnabled: true,
    metricName: 'AWSCommonRuleSet',
  },
};
```

ポイント

9. WAFルール作成（マネージドルール）

- overrideActionはnoneとして、通信をスルーさせています。
- まずはどんな攻撃が来ているかcountで確認して、必要に応じてルール化してブロックするなどの運用が望ましいです。

演習1-1　ネットワーク構築　解答編

Step3：ALB/WAF　CDKコード

ALB/WAF　解答例（6/6）

```
// 10. WebACL作成
const webAcl = new wafv2.CfnWebACL(this, 'AlbWebAcl', {
  scope: 'REGIONAL',
  defaultAction: { allow: {} },
  visibilityConfig: {
    cloudWatchMetricsEnabled: true,
    sampledRequestsEnabled: true,
    metricName: 'AlbWebAcl',
  },
  rules: [badBotRule, awsManagedCommon],
});

// 11. ALB と WebACL の関連付け
new wafv2.CfnWebACLAssociation(this, 'WebAclAssociation', {
  resourceArn: alb.loadBalancerArn,
  webAclArn  : webAcl.attrArn,
});

// 12. 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        });
  }
}
```

ポイント

10. WebACL作成

- rulesで、WebACLに紐づけるルールを選択します。
- WebACLおよびルールグループに紐づけられるルールには上限があります。
- WCU:5000に抵触しないよう適用するルールを調整・管理してください

11. ALBとWebACLの関連付け

- ALBに紐づけられるWebACLは1つのみです。
- ALB、WebACLを両方作成してから関連付けする。作成の順序を考慮してください。

演習1-1　ネットワーク構築　解答編

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　解答編

Step3：ALB/WAF　動作確認

- CloudShellからALBおよびWAFの動作確認することができます。

```
// ALBの変数設定
ecs-demo $ ALB_DNS=$(aws cloudformation describe-stacks \
  --stack-name AlbStack \
  --query "Stacks[0].Outputs[?OutputKey=='AlbDnsName'].OutputValue" \
  --output text)
ecs-demo $ echo "ALB DNS = $ALB_DNS"

// ALBの正常性確認（503が返ればOK）
ecs-demo $ curl -s -o /dev/null -w "%{http_code}\n" http://$ALB_DNS/

// WAFの動作確認 – badBotルール（403が返ればOK）
ecs-demo $ curl -s -o /dev/null -w "%{http_code}\n" \
  -H "User-Agent: BadBot Scanner" http://$ALB_DNS/

// WAFの動作確認 – AWSマネージドルール/XSS（403が返ればOK）
ecs-demo $ curl -s -o /dev/null -w "%{http_code}\n" -d '<script>alert(1)</script>' \
     -H "Content-Type: application/x-www-form-urlencoded" "http://$ALB_DNS/submit"
```

演習1-2　解答編

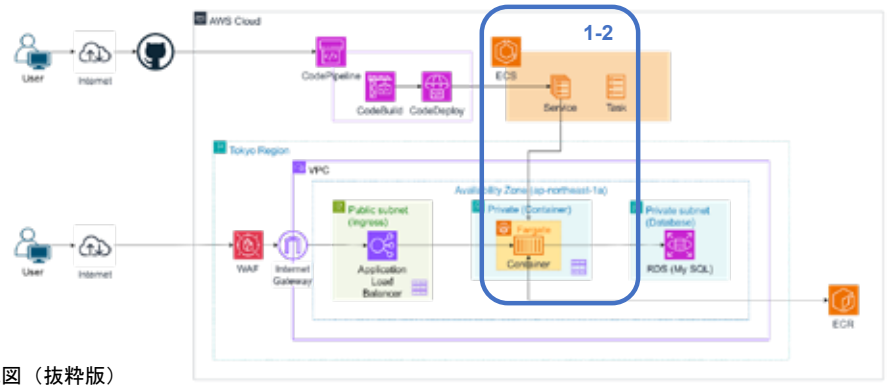
コンテナ構築

演習1-2　コンテナ構築　解答編

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　解答編

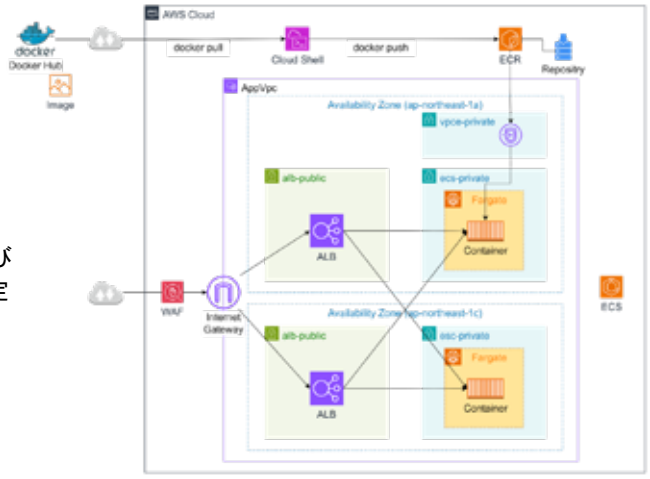
演習概要

- AWS CDKを利用して、コンテナ環境を3段階で構築します。
  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**
サーバレスーコンテナ(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



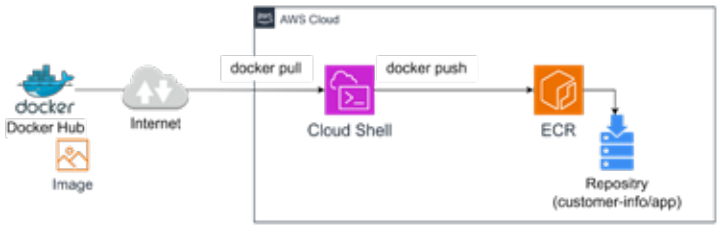
演習1-2 Step1　解答編

コンテナ構築 - ECR

演習1-2　コンテナ構築　解答編

Step1：ECR　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　解答編

Step1：ECR　CDKコード

- ECRの設計値を基に、CDKプロジェクトのコードを作成します。

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

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

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　解答編

Step1：ECR　CDKコード

- bin/ecs-demo.tsのコードは以下の通りです。

```
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';
import { AlbStack } from '../lib/alb-stack';
import { EcrStack } from '../lib/ecr-stack';  //★追加

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 });
```

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

// ALB / WAF
const alb = new AlbStack(app, 'AlbStack', {
  env,
  vpc: net.vpc,
  albSg: net.albSg,
});

// ECR Stack ★追加
const ecr = new EcrStack(app, 'EcrStack', { env });
```

演習1-2　コンテナ構築　解答編

Step1：ECR　CDKコード

- lib/ecr-stack.tsのコードの全体構成は以下の通りです。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	スタック初期化	EcrStackを初期化
3	リポジトリ作成	customer-info/appリポジトリの作成
4	ライフサイクル	リポジトリのライフサイクルルールを設定
5	出力	設定値の出力

演習1-2　コンテナ構築　解答編

Step1：ECR　CDKコード

ECRリポジトリ作成　解答例（1/2）

```
// 1. インポート
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as ecr from 'aws-cdk-lib/aws-ecr';

// 2. スタック初期化
export class EcrStack extends cdk.Stack {
  public readonly repository: ecr.Repository;

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

    // 3. リポジトリ作成
    this.repository = new ecr.Repository(this,'CustomerInfoAppRepo',{
        repositoryName : 'customer-info/app',              // 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
      },
    );
```

ポイント

1. インポート
- ecrをインポート

2. スタック初期化
- ECRリポジトリ名は、ECSタスク実行時に参照するため、外部に公開します。

3. リポジトリ作成
- imageScanOnPushをtrueにすることで、docker push時に脆弱性スキャンが動作。
- removalPolicyでスタック削除時の挙動を制御できます。RETAINにすることで、スタック削除時、リポジトリおよび登録済イメージを残せます。
- 既に同名のリポジトリが存在する場合、エラーとなるので注意してください。

演習1-2　コンテナ構築　解答編

Step1：ECR　CDKコード

ECRリポジトリ作成　解答例（2/2）

```
    // 4. ライフサイクル
    this.repository.addLifecycleRule({
      description : 'Delete images older than 7 days',
      maxImageAge : cdk.Duration.days(7),
      tagStatus   : ecr.TagStatus.ANY,
    });

    // 5. 出力
    new cdk.CfnOutput(this, 'CustomerInfoAppRepoUri',{
      value      : this.repository.repositoryUri,
      description: 'ECR repository URI for customer-info/app',
      exportName : 'customerInfoAppRepoUri',
    });
  }
}
```

ポイント

4. ライフサイクル
- リポジトリ内のイメージのライフサイクルを定義します。今回はイメージを7日間保持としています。
- tagStatusの設定により、タグの有無でライフサイクルルールを適用するか選択できます。

- システムの実運用に合わせてライフサイクルは設定する必要があるので注意してください。
  例えば、
  - 直近N個イメージを保持する
  - 商用タグのイメージは長期保存する
  - 開発タグのイメージは3日で削除

  などのルールを設けることができます。

演習1-2　コンテナ構築　解答編

Step1：ECR　動作確認

- CloudShellからAWS CDKでEcrStackをデプロイし、
  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

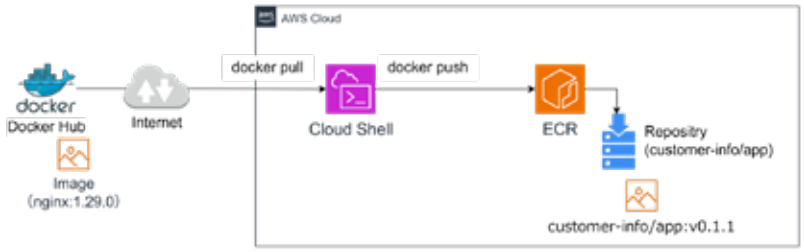
演習1-2 Step2　解答編

コンテナ構築 – イメージpush

演習1-2　コンテナ構築　解答編

Step2：イメージpush　設計

- CloudShellでDockerイメージを取得・ビルドして、ECRにdocker pushします。
  ベースイメージはNginxとし、customer-info/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	customer-info/app
	タグ	v0.1.1

演習1-2　コンテナ構築　解答編

Step2：イメージpush　作業手順

- CloudShellで、Nginxのベースイメージを取得し、ECRに登録します。

```
// Nginxイメージをpull
ecs-demo $ docker pull nginx:1.29.0
ecs-demo $ docker images          //nginxのイメージが存在すること

// Nginxのイメージにタグ付け
ecs-demo $ AWS_ACCOUNT_ID=$(aws sts get-caller-identity --query "Account" --output text)
ecs-demo $ ECR_URL=${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_REGION}.amazonaws.com
ecs-demo $ docker tag nginx:1.29.0 $ECR_URL/customer-info/app:v0.1.1          //Nginxイメージにタグ付け
ecs-demo $ docker images          //タグ付けしたイメージが追加されていること

// ECRにログイン
ecs-demo $ aws ecr get-login-password | docker login --username AWS --password-stdin ${ECR_URL}

// ECRにイメージをpush
ecs-demo $ docker push $ECR_URL/customer-info/app:v0.1.1
```

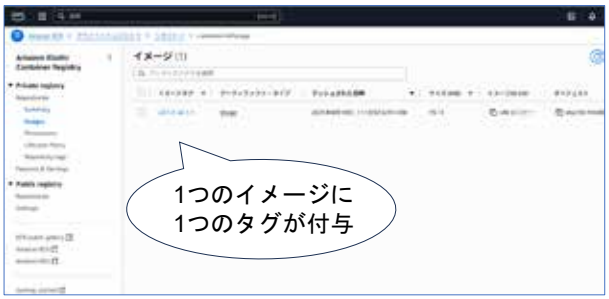
演習1-2　コンテナ構築　解答編

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



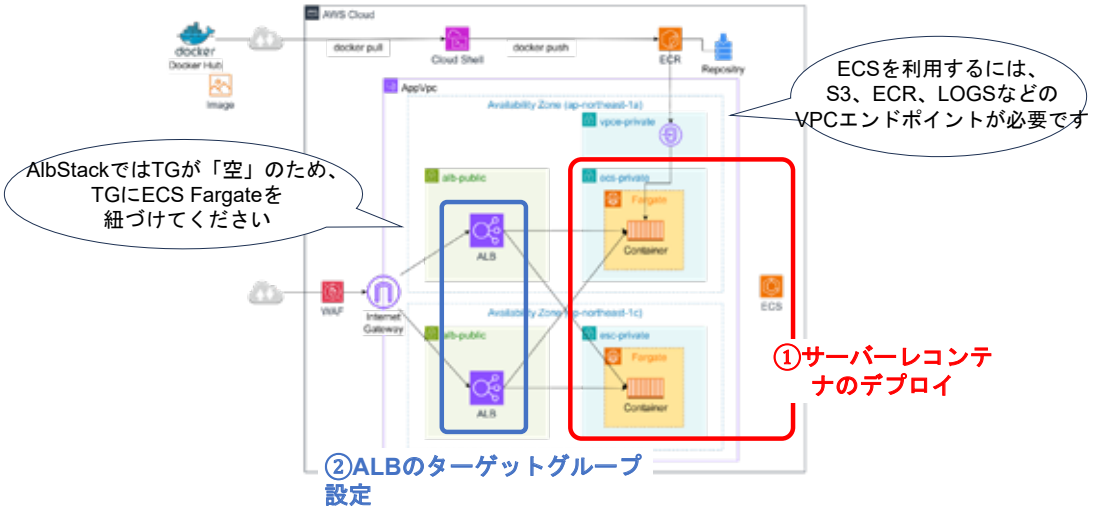
演習1-2 Step3　解答編

コンテナ構築 – ECS

演習1-2　コンテナ構築　解答編

Step3：ECS　設計

- 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ECRのcustomer-info/app:v0.1.1を利用します。



演習1-2　コンテナ構築　解答編

Step3：ECS　設計

- 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。
  これらVPCエンドポイントはVpceStackで設定しています。

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　解答編

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　解答編

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
	起動直後のヘルスチェック開始時間	healthCheckGracePeriod	60sec
ALB　※	ALBのターゲットグループ	attachToApplicationTargetGroup	AlbTg
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　解答編

Step3：ECS　CDKコード

■ 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。

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 Fargateのタスク定義、サービスを定義

```
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

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　解答編

Step3：ECS　CDKコード

■ bin/ecs-demo.tsのコードは以下の通りです。

```ts
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';
import { EcrStack } from '../lib/ecr-stack';
import { EcsStack } from '../lib/ecs-stack';   // ★追加

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 });

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

```ts
// ALB / WAF
const alb = new AlbStack(app, 'AlbStack', {
  env,
  vpc: net.vpc,
  albSg: net.albSg,
});

// ECR
const ecr = new EcrStack(app, 'EcrStack', { env });

// ECS / Fargate  ★追加
const ecs = new EcsStack(app, 'EcsStack', {
  env,
  vpc       : net.vpc,
  ecsSg     : net.ecsSg,
  repo      : ecr.repository,
  targetGroup: alb.targetGroup,
});
```

演習1-2　コンテナ構築　解答編

Step3：ECS　CDKコード

■ lib/ecs-stack.tsのコードの全体構成は以下の通りです。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インタフェース定義	他スタックから渡されるパラメータ
3	スタック初期化	EcrStackを初期化
4	ECSクラスタ作成	ECSクラスタを作成
5	IAMロール設定	ECSにIAMロールを付与（Excution、Task）
6	タスク定義	ECS Fargateのタスク定義を定義
7	サービス定義	ECS Fargateのサービスを定義
8	ALBターゲットグループ登録	ALBのターゲットグループにECS Fargateを追加
9	オートスケール設定	ECS Fargateのオートスケールを設定（任意）
10	出力	設定値の出力

演習1-2　コンテナ構築　解答編

Step3：ECS　CDKコード

ECSタスク定義/サービス作成　解答例（1/6）

```
// 1. インポート
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 iam  from 'aws-cdk-lib/aws-iam';
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';
import * as cw  from 'aws-cdk-lib/aws-cloudwatch';

// 2. インタフェース定義
export interface EcsStackProps extends StackProps {
  vpc         : ec2.IVpc;
  ecsSg       : ec2.ISecurityGroup;
  repo        : ecr.IRepository;
  targetGroup : elbv2.ApplicationTargetGroup;
}

// 3. スタック初期化
export class EcsStack extends Stack {
  public readonly cluster: ecs.Cluster;
  public readonly service: ecs.FargateService;
  constructor(scope: Construct, id: string, props: EcsStackProps) {
    super(scope, id, props);
```

ポイント

1. インポート
- importするサービスに「Duration」「RemovalPolicy」を追加しています。タスクのヘルスチェック、削除時の挙動を制御するために利用します。

2. インタフェース定義
- ECS起動に必要な外部パラメータを読み込み設定しています。

3. スタック初期化
- CodeDeployで利用するため、ECSクラスター、ECSサービスを外部プロパティとして公開します。

演習1-2　コンテナ構築　解答編

Step3：ECS　CDKコード

ECSタスク定義/サービス作成　解答例（2/6）

```
    // 4. ECSクラスタ作成
    this.cluster = new ecs.Cluster(this, 'AppCluster', {
      vpc: props.vpc,
      clusterName: 'ecs-app-cluster',
      containerInsights: true,
    });

    // 5. IAMロール設定
    const execRole = new iam.Role(this, 'TaskExecutionRole', {
      assumedBy: new iam.ServicePrincipal('ecs-tasks.amazonaws.com'),
      managedPolicies: [
        iam.ManagedPolicy.fromAwsManagedPolicyName('service-
role/AmazonECSTaskExecutionRolePolicy'),
      ],
    });
    execRole.addToPolicy(new iam.PolicyStatement({
      actions: [
        'logs:CreateLogGroup', 'logs:CreateLogStream',
        'logs:PutLogEvents', 'logs:DescribeLogStreams',
      ],
      resources: ['*'],
    }));
    const taskRole = new iam.Role(this, 'AppTaskRole', {
      assumedBy: new iam.ServicePrincipal('ecs-tasks.amazonaws.com'),
    });
```

ポイント

4. ECSクラスタ作成

- 今回はECSクラスター名を一意に設定しています。
- containerInsights:trueにすることでコンテナに特化したモニタリングが可能になります。ただし、 CloudWatch Container Insightは有料です。

5. IAM設定

- ECS実行に必要なIAMロールを付与します。
  - Exec：ECRからイメージをpull、CloudWatch Logsなどを実行。
  - Task：アプリがAWS APIを叩く際に利用

演習1-2　コンテナ構築　解答編

Step3：ECS　CDKコード

ECSタスク定義/サービス作成　解答例（3/6）

```
    // 6. 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',
      }),
      portMappings: [{ containerPort: 80 }],
```

ポイント

6. タスク定義

- Fargateのタスク定義します。
- Fargateのリソースをcpu/memoryLimitで指定します。サービスに合わせて、最適なリソース値を設定してください。
- タスクに付与するロールもセットします。
- imageでECRから引っ張るイメージ:タグを指定します。演習1-2 Step2でECRに格納したcustomer-info/appのイメージを呼び出します。

演習1-2　コンテナ構築　解答編

Step3：ECS　CDKコード

ECSタスク定義/サービス作成　解答例（4/6）

```
    // 6. タスク定義（続き）
    // ヘルスチェック（80番ポートがLISTEN中かを /proc で判定）
    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',
    },
  });
```

ポイント

6. タスク定義

- ECSエージェントからコンテナのヘルスチェックを行います。
- /proc/net/tcpで簡易なヘルスチェックを実現しています。他にもアプリにcurlが入っていれば、curlでヘルスチェックするなどが可能です。
- ヘルスチェックの開始タイミングやチェック間隔などはアプリの起動時間をテストしたうえで、調整するのが良いでしょう。
  ※例えば、DBを含むタスクは起動時間が長時間化しやすく、ヘルスチェックに引っかかり、コンテナ再起動を繰り返すことがあります。

演習1-2　コンテナ構築　解答編

Step3：ECS　CDKコード

ECSタスク定義/サービス作成　解答例（5/6）

```
    // 7. 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 },
    });

    // 8. ALBターゲットグループ登録
    this.service.attachToApplicationTargetGroup(props.targetGroup);
```

ポイント

7. サービス定義

- タスク定義を指定して、タスクを起動します。
- enableExecuteCommand:trueにすることでタスクにECS execを実行できます。セキュリティの観点から、本番環境ではfalseを推奨します。
- ここで設定するhealthCheckGracePeriodは、ALBからのヘルスチェック開始に対する猶予時間になります。コンテナ内のヘルスチェックとは異なります。
- いずれCodeDeployでB/Gデプロイするため、deploymentControllerタイプをCODE_DEPLOYに設定します。

8. ALBターゲットグループ登録

- 既存TGにECS Fargateを登録します。
- ALB側で設定したヘルスチェック(今回は"/"を設定)とアプリケーションパスを一致させることが大切です。一致しない場合、ALBのヘルスチェックに失敗しリクエストが振り分けられません。

演習1-2　コンテナ構築　解答編

Step3：ECS　CDKコード

ECSタスク定義/サービス作成　解答例（6/6）

```
    // 9. オートスケール設定（任意）
    const scalable = this.service.autoScaleTaskCount({ minCapacity: 2, maxCapacity: 4 });

    // 9-1 CPU 50% でターゲット追跡
    scalable.scaleOnCpuUtilization('Cpu50', {
      targetUtilizationPercent: 50,
      scaleInCooldown : Duration.seconds(60),
      scaleOutCooldown: Duration.seconds(60),
    });

    // 9-2 ALB リクエスト 100 req/tgt/sec
    scalable.scaleOnRequestCount('Req100', {
      requestsPerTarget: 100,
      targetGroup: props.targetGroup,
      scaleInCooldown : Duration.seconds(60),
      scaleOutCooldown: Duration.seconds(60),
    });

    // 10. 出力
    new CfnOutput(this, 'ClusterArn', { value: this.cluster.clusterArn });
    new CfnOutput(this, 'ServiceName', { value: this.service.serviceName });
    new CfnOutput(this, 'TaskFamily',  { value: taskDef.family });
  }
}
```

ポイント

9. オートスケール（任意）

- 任意でタスクのオートスケールを設定できます。要件には含めていませんが、コンテナであればオートスケールも合わせて設定する方が望ましいです。
- minCapacity / maxCapacityで起動するタスク数の最小値、最大値を設定できます。最初はサービス定義のdesiredCount数でタスクが起動しますが、その後min/maxCapacityの値で起動数が制御されます。
- 条件は複数設定でき、タスク特性に合わせて設定してください。CPU/MEM使用率やリクエスト数をトリガーにオートスケールするのが一般的です。

演習1-2　コンテナ構築　解答編

Step3：ECS　動作確認

■ 各スタックを実行時リソースの作成を確認します。
CloudFormationのスタックが動作して、Fargateコンテナを構築できました。

■ 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

■ CloudFormation確認

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

■ 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　解答編

Step3：ECS　動作確認

■ CloudShellからALBにアクセスし、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　解答編

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1-3　解答編

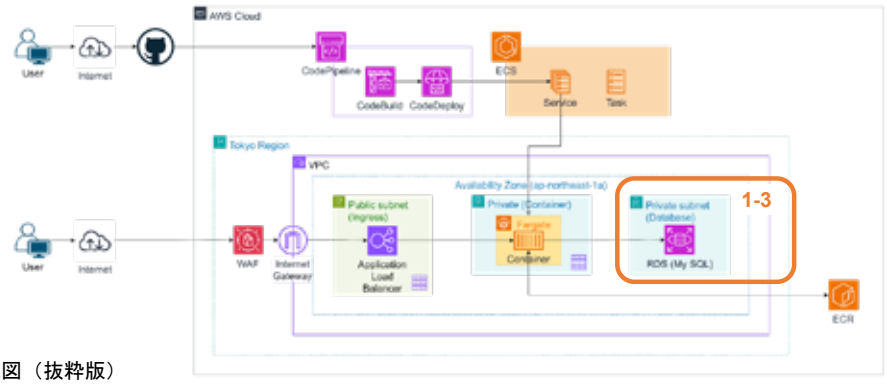
DB構築

演習1-3　DB構築　解答編

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）

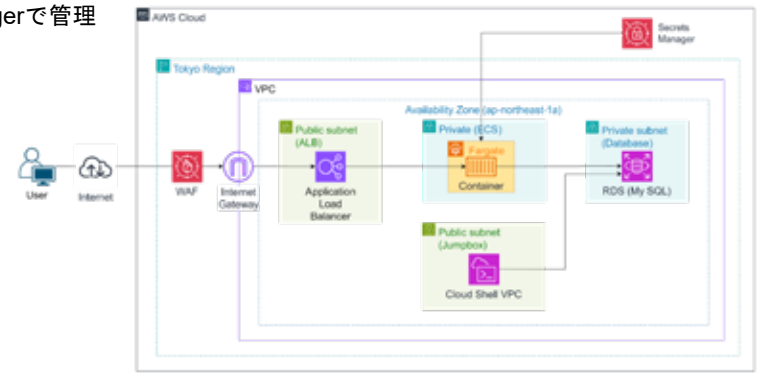
演習1-3　DB構築　解答編

演習概要

- AWS CDKを利用してDBを構築します。

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　解答編

DB構築 - RDS

演習1-3　DB構築　解答編

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　解答編

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）のみを実行可能とします。

演習1-3　DB構築　解答編

Step1：RDS　CDKコード

- AWS CDKを利用して、**RDSスタックを作成するコード**を作成します。

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 Fargateのタスク定義、サービスを定義

```
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

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　解答編

Step1：RDS　CDKコード

- bin/ecs-demo.tsのコードは以下の通りです。

```ts
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';
import { AlbStack } from '../lib/alb-stack';
import { EcrStack } from '../lib/ecr-stack';
import { RdsStack } from '../lib/rds-stack';    // ★追加
import { EcsStack } from '../lib/ecs-stack';

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 });

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

```ts
// ALB / WAF
const alb = new AlbStack(app, 'AlbStack', {
  env,
  vpc: net.vpc,
  albSg: net.albSg,
});
// ECR
const ecr = new EcrStack(app, 'EcrStack', { env });

// RDS       ★追加
const rds = new RdsStack(app, 'RdsStack', {
  env,
  vpc: net.vpc,
  dbSg: net.dbSg,
});

// ECS / Fargate
const ecs = new EcsStack(app, 'EcsStack', {
  env,
  vpc       : net.vpc,
  ecsSg     : net.ecsSg,
  repo      : ecr.repository,
  targetGroup: alb.targetGroup,
});
```

演習1-3　DB構築　解答編

Step1：RDS　CDKコード

■ lib/rds-stack.tsのコードの全体構成は以下の通りです。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インタフェース定義	他スタックから渡されるパラメータ
3	スタック初期化	RdsStackを初期化
4	シークレット作成(admin)	RDSの管理者ユーザ(admin)のシークレット情報をSecret Managerに生成・登録
5	シークレット作成(app_user)	RDSの管理者ユーザ(app_user)のシークレット情報をSecret Managerに生成・登録
6	RDS MySQL作成	RDS(MySQL)のインスタンスを作成
7	出力	設定値の出力

演習1-3　DB構築　解答編

Step1：RDS　CDKコード

RDS作成　解答例（1/4）

```
// 1. インポート
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';

// 2. インタフェース定義
export interface RdsStackProps extends cdk.StackProps {
  vpc: ec2.IVpc;
  dbSg: ec2.ISecurityGroup;
}

// 3. スタック初期化
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);
```

ポイント

1. インポート

- rdsだけでなく、シークレット登録のためsecretsmanagerもインポートします。

3. スタック初期化

- DBインスタンス、DB接続情報、DBホストはECSやCI/CDパイプラインで利用するため、外部プロパティとして公開します。

演習1-3　DB構築　解答編

Step1：RDS　CDKコード

RDS作成　解答例（2/4）

```
// 4. シークレット作成（admin）
this.dbSecret = new rds.DatabaseSecret(this, 'AdminDbSecret', {
  secretName: 'adnmin-db-credentials',
  username: 'admin',
});

// 5. シークレット作成（app_user）
const appSecret = new secretsmanager.Secret(this, 'CustomerInfoAppSecret', {
  secretName: 'customer-info-app-credentials',
  generateSecretString: {
    secretStringTemplate: JSON.stringify({ username: 'app_user' }),
    generateStringKey: 'password',
    excludePunctuation: true,
  },
});

this.appSecret = appSecret;
```

ポイント

4. シークレット作成(admin)

- 管理者ユーザー(admin)のシークレットを作成し、Secrets Managerで管理します。
  ※データベース接続情報はセキュアに管理することが望ましいです。
- CDK動作時はadminでインスタンス、初期データベースを構築します。

5. シークレット作成(app_user)

- アプリユーザー(app_user)のシークレットを作成します。ただし、シークレット作成のみでRDSにapp_user自体は現段階で作成しません。
- app_userは独自でシークレットを用意する形となるため、adminと比較して設定行数が多くなります。

演習1-3　DB構築　解答編

Step1：RDS　CDKコード

RDS作成　解答例（3/4）

```
// 6. RDS MySQLインスタンス作成
this.dbInstance = new rds.DatabaseInstance(this, 'ApplicationDb', {
  engine: rds.DatabaseInstanceEngine.mysql({
    version: rds.MysqlEngineVersion.VER_8_0_42,
  }),
  instanceIdentifier: 'application-db',      // RDS インスタンス名
  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,
  storageType: rds.StorageType.GP3,
  allocatedStorage: 20,
  maxAllocatedStorage: 100,
  deletionProtection: false,
  removalPolicy: cdk.RemovalPolicy.DESTROY,
  databaseName: 'customer_info',             // 初期データベース名
});

this.dbHost = this.dbInstance.dbInstanceEndpointAddress;
```

ポイント

6. RDS MySQL インスタンス作成

- AWS RDSはAurora、Postgres、MariaDBなどもDBエンジンとして選択できます。実際に開発する際は、最適なものを選択してください。
- credentialsにDB接続情報（シークレット）を選択することで、Secrets Managerでセキュアにシークレットを管理できます。
- multiAz:falseで単一AZでRDSを構築していますが、実運用では複数AZにレプリケーションするのが一般的です。
- DB削除時は全て削除する設定としていますが、本番環境ではデータ管理の方法を検討の上、最適に設定してください。

演習1-3　DB構築　解答編

Step1：RDS　CDKコード

RDS作成　解答例（4/4）

```
    // 7. 出力
    new cdk.CfnOutput(this, 'ApplicationDbEndpoint', {
      value: this.dbInstance.dbInstanceEndpointAddress,
    });
    new cdk.CfnOutput(this, 'AdminDbSecretName', {
      value: this.dbSecret.secretName,
    });
    new cdk.CfnOutput(this, 'AppSecretName', {
      value: this.appSecret.secretName,
    });
    new cdk.CfnOutput(this, 'AppSecretArn', {
      value: this.appSecret.secretArn,
    });
  }
}
```

ポイント

7. 出力

- AppSecretのシークレット名、ARNを外から参照できるよう、以下を出力しています。
  - AppSecretName
  - AppSecretArn

演習1-3　DB構築　解答編

Step1：RDS　動作確認

■ CloudShellからAWS CDKでRdsStackをデプロイし、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

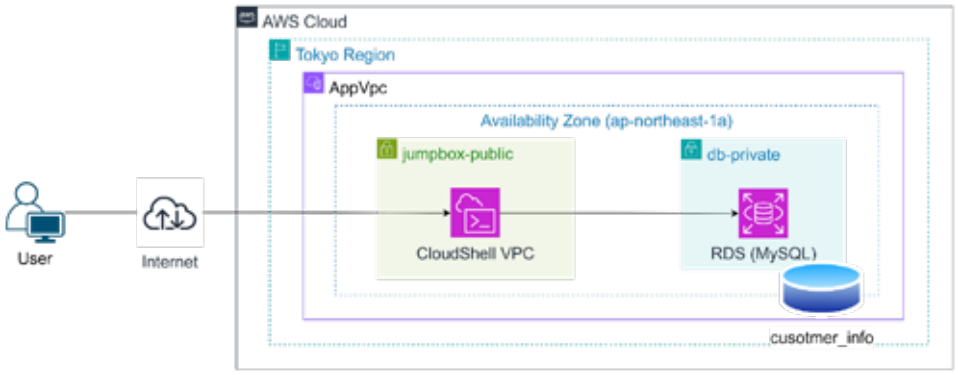
Secrets Manager

- 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　解答編

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　解答編

Step1：RDS　動作確認

- CloudShell VPCからRDSに接続し、初期データベースを確認できれば、Step1は完了です。

```
// MySQLのインストール
~ $ sudo dnf makecache
~ $ sudo dnf install -y mariadb105  // ★MySQLパッケージがないため、MySQLを含むMariaDB105を代替でインストール

// Secrets ManagerからDB接続情報を確認
~ $ aws secretsmanager get-secret-value \
  --secret-id admin-db-credentials \
  --query SecretString \
  --output text | jq .
{
  "password": “xxxxxxxxx",
  "dbname": “customer_info",
  "engine": "mysql",
  "port": 3306,
  "dbInstanceIdentifier": “application-db",
  "host": “application-db.xxxxx.ap-northeast-1.rds.amazonaws.com",
  "username": "admin"
}
```

```
// MySQLクライアントでログイン
// DB接続情報のhost、username、 passwordを利用します
~ $ mysql -h <host> -u <username> -p
Enter password: <password>

// 初期データベースの確認
MySQL [(none)]> SHOW DATABASES;

+----------------------+
| Database             |
+----------------------+
| customer_info        | // ★初期データベースが構築済
| information_schema   |
| mysql                |
| performance_schema   |
| sys                  |
+----------------------+
5 rows in set (0.023 sec)
```

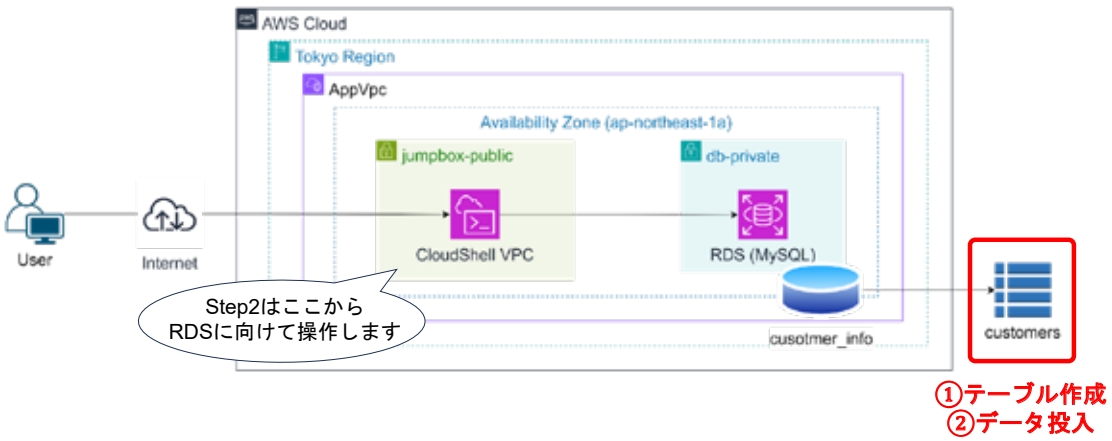
演習1-3 Step2　解答編

DB構築 – テーブル作成 / データ投入

演習1-3　DB構築　解答編

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　解答編

Step2：テーブル作成 / データ投入　設計

■ 初期データベース(customer_info)にテーブル(customers)を作成し、データを登録します。
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)も作成します。
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　解答編

Step2：テーブル作成 / データ投入　CLI操作

■ まずアプリケーション用のMySQLユーザーとしてapp_userを作成します。

```
// adminでMySQLログインおよび初期データベース確認済の前提です

// ユーザー確認（app_userが存在しないことを確認）
MySQL [(none)]> SELECT  host, user FROM mysql.user;

// ユーザー作成
// '<password>'は、SecretsManagerの
// customer-info-app-credentialsのpasswordに合わせてください。
MySQL [(none)]> CREATE USER 'app_user'@'%' IDENTIFIED BY
'<password>';

// ユーザー権限付与
MySQL [(none)]> GRANT SELECT, INSERT, UPDATE ON
customer_info.* TO 'app_user'@'%';
// 権限の即時反映
MySQL [(none)]> FLUSH PRIVILEGES;

// ユーザー確認（app_userが存在することを確認）
MySQL [(none)]> SELECT  host, user FROM mysql.user;

// ユーザー権限確認
MySQL [(none)]> SHOW GRANTS FOR 'app_user'@'%';
```

ポイント

ユーザー作成

- <password>はSecrets Managerのcustomer-info-app-credentialsのパスワードと一致させてください。

ユーザー権限付与

- 本番環境では、アプリケーションに対して適切な操作権限を付与してください。例えば参照するだけならSELECTに絞る。

動作確認

- 適切にユーザーが作成されていれば、app_userでMySQLにログインできるようになります。必要に応じて確認してください。

演習1-3　DB構築　解答編

Step2：テーブル作成 / データ投入　CLI操作

■ 続いて、customersテーブルおよびデータ登録(5件)を確認できれば、Step2は完了です。

```
// adminでMySQLログインおよび初期データベース確認済の前提です
// データベース選択
MySQL [customer_info]> USE customer_info;

// テーブル作成（customers）
MySQL [customer_info]> CREATE TABLE customers (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  age INT,
  email VARCHAR(100) UNIQUE,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
// customersテーブル構成確認
MySQL [customer_info]> DESCRIBE customers;

// データ投入（登録データ5行分をINSERT）
MySQL [customer_info]> INSERT INTO customers (name, age, email)
VALUES
('山田 太郎', 28, 'taro.yamada@example.com'),
('佐藤 花子', 34, 'hanako.sato@example.com'),
('鈴木 一郎', 22, 'ichiro.suzuki@example.com'),
('高橋 美咲', 41, 'misaki.takahashi@example.com'),
('中村 健', 30, 'ken.nakamura@example.com');
// データ確認
MySQL [customer_info]> SELECT * FROM customers;
```

ポイント

customerテーブル作成

- テーブル作成時、各カラムの条件(PRIMARY KEYなど)を忘れず設定してください。

データ登録

- データはまとめて登録しても、一件ずつ投入しても構いません。

実行結果（SELECT * FROM customers;）

```
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1-4　解答編

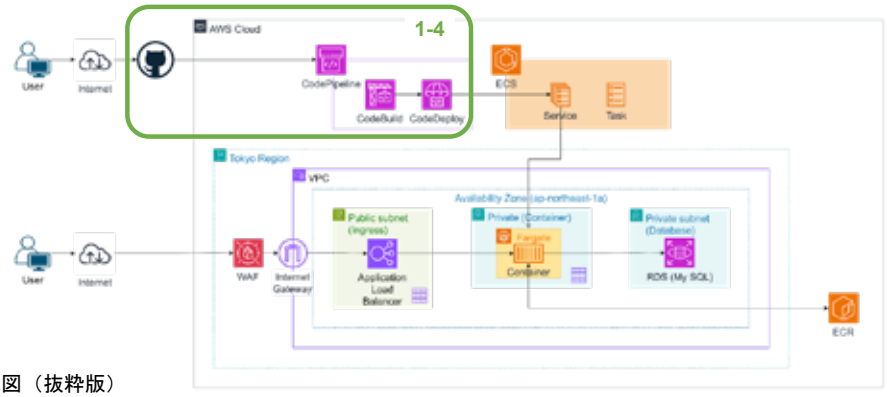
CI/CDパイプライン構築

演習1-4　CI/CDパイプライン構築　解答編

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）

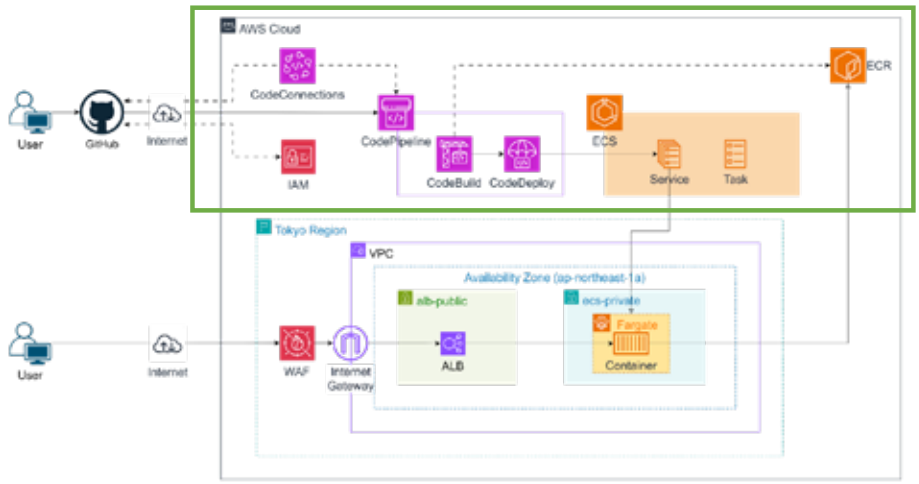


全体構成図（抜粋版）

演習1-4　CI/CDパイプライン構築　解答編

演習概要

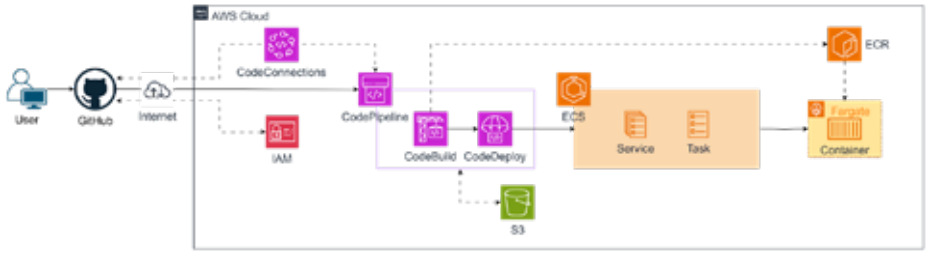
- AWS CDKを利用してCICDパイプラインを構築します。
- GitHub～ECR/ECSを連携し、GitHubリポジトリのコード変更をトリガーに、CodePipelineを起動し、ビルド～デプロイの自動化を実現します。



演習1-4　CI/CDパイプライン構築　解答編

演習概要

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



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

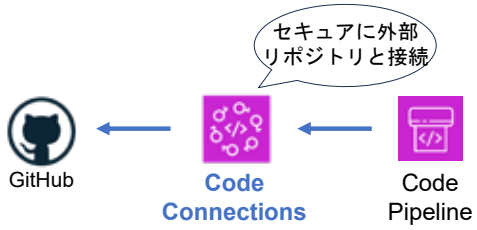
演習1-4　Step1　解答編

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　解答編

Step1：CodeConnections　設計

■ CI/CDパイプラインの実装にあたり、CodePipelineから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を安全に接続します。



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　解答編

Step1：CodeConnections　CDKコード

- CodeConnectionsの設計値を基に、CDKプロジェクトのコードを作成します。

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
ConnectionStack	CodeConnections(GitHub接続)を定義

```
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/connection-stack.ts    ★新規作成
・・・
```

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　解答編

Step1：CodeConnections　CDKコード

- bin/ecs-demo.tsのコードは以下の通りです。

```ts
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';
import { AlbStack } from '../lib/alb-stack';
import { EcrStack } from '../lib/ecr-stack';
import { RdsStack } from '../lib/rds-stack';
import { EcsStack } from '../lib/ecs-stack';
import { ConnectionStack } from '../lib/connection-stack';//★追加

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 });

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

```ts
// ALB / WAF
const alb = new AlbStack(app, 'AlbStack', {
  env,
  vpc: net.vpc,
  albSg: net.albSg,
});

// ECR
const ecr = new EcrStack(app, 'EcrStack', { env });

// RDS
const rds = new RdsStack(app, 'RdsStack', {
  env,
  vpc: net.vpc,
  dbSg: net.dbSg,
});

// ECS / Fargate
const ecs = new EcsStack(app, 'EcsStack', {
  env,
  vpc       : net.vpc,
  ecsSg     : net.ecsSg,
  repo      : ecr.repository,
  targetGroup: alb.targetGroup,
});

// CodeConnection Stack  ★追加
const conn = new ConnectionStack(app, 'ConnectionStack', { env });
```

演習1-4　CI/CDパイプライン構築　解答編

Step1：CodeConnections　CDKコード

■ lib/connection-stack.tsの全体構成は以下の通りです。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	クラス宣言	他スタック向けにプロパティの公開
3	スタック初期化	NetStackを初期化
4	コネクション作成	GitHub接続用のコネクション作成
5	出力	設定値の出力

演習1-4　CI/CDパイプライン構築　解答編

Step1：CodeConnections　CDKコード

CodeConnecton作成　解答例

```
// 1. インポート
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as codestar from 'aws-cdk-lib/aws-codestarconnections';

// 2. クラス宣言、他スタックに公開するプロパティ
export class ConnectionStack extends cdk.Stack {
  public readonly connectionArn: string;

  // 3. スタック初期化
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // 4. コネクション作成
    const conn = new codestar.CfnConnection(this, 'GitHubConnection', {
      connectionName: 'CustomerInfoGitHub',
      providerType: 'GitHub',
    });
    this.connectionArn = conn.attrConnectionArn;

    // 5. 出力
    new cdk.CfnOutput(this, 'GitHubConnectionArn', {
      value: this.connectionArn,
      exportName: 'GitHubConnectionArn',
    });
  }
}
```

ポイント

1. インポート

- codestarをインポートします。サービス名はCodeConnectionsに変更となりましたが、CDKのモジュール名はcodestarのままです。

2. スタック初期化

- CodePipelineから利用するため、作成したコネクションを外部展開します。

4. コネクション作成

- providerType: 'GitHub'とし、GitHubのコネクションを作成します。作成しても保留状態のため、GUIで手動承認を忘れず実施してください。

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、**GUIから手動承認**すれば、Step1完了です。
- 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」すると、GitHubと接続可能になります。

承認前

承認後

演習1-4　Step2　解答編

CI/CDパイプライン構築 – IAM

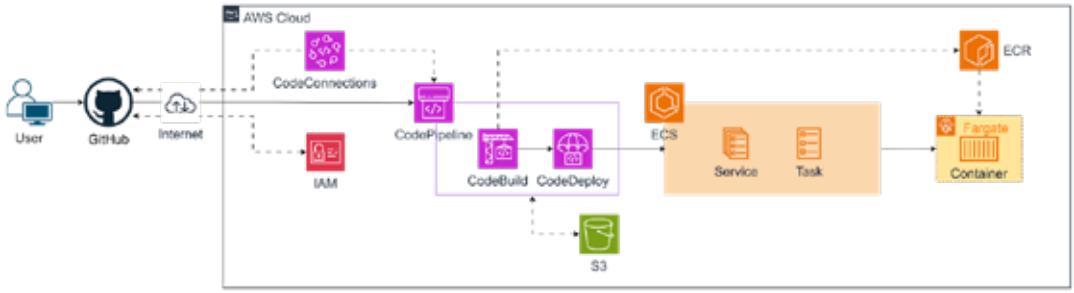
演習1-4　CI/CDパイプライン構築　解答編

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　解答編

Step2：IAM　設計

- IAMロールとポリシーは以下の通りです。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　解答編

Step2：IAM　設計

■ IAMロールとポリシーは以下の通りです。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)	EcsTaskExecution Role	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　解答編

Step2：IAM　CDKコード

■ AWS CDKを利用して、**IAMスタックを作成するコード**を作成します。

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 Fargateのタスク定義、サービスを定義
ConnectionStack	CodeConnections(GitHub接続)を定義
IamStack	IAMロールを定義

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　解答編

Step2：IAM　CDKコード

- **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　解答編

Step2：IAM　CDKコード

- bin/ecs-demo.tsのコードは以下の通りです。

```
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';
import { AlbStack } from '../lib/alb-stack';
import { EcrStack } from '../lib/ecr-stack';
import { RdsStack } from '../lib/rds-stack';
import { EcsStack } from '../lib/ecs-stack';
import { ConnectionStack } from '../lib/connection-stack';
import { IamStack } from '../lib/iam-stack';  // ★追加

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 });

・・・省略・・・
```

```
// CodeConnection
const conn = new ConnectionStack(app, 'ConnectionStack', { env });

// IAM  ★追加
const iam = new IamStack(app, 'IamStack', {
  env,
  ecrRepoName : 'customer-info/app',
  pipelineName: 'CustomerInfoPipeline',
  // ★GitHubアカウント名はご自身の環境に合わせて変更してください
  ghOwner     : 'xxx',  // GitHubアカウント
  ghRepo      : 'customer-info',  // GitHubリポジトリ
  gitHubConnectionArn: conn.connectionArn,
  appSecretArn: rds.appSecret.secretArn,  // アプリ用の認証情報
});
```

演習1-4　CI/CDパイプライン構築　解答編

Step2：IAM　CDKコード

■ lib/iam-stack.tsの全体構成は以下の通りです。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インタフェース定義	他スタックから渡されるパラメータ
3	スタック初期化	IamStackを初期化
4	CodeBuildロール作成	CodeBuildロールを作成し、ポリシーを付与
5	CodeDeployロール作成	CodeDeployロールを作成し、ポリシーを付与
6	TaskExecutionロール作成	ECS TaskExecutionロールを作成し、ポリシーを付与
7	AppTaskロール作成	ECS AppTaskロールを作成し、ポリシーを付与
8	SecretsManagerロール作成	SecretsManagerを参照する場合、一部ロールにポリシーを付与
9	CodePipelineロール作成	CodePipelineロールを作成し、ポリシーを付与
10	GitHub OIDCプロバイダー / ロール作成	GitHub OIDCプロバイダーおよびOIDCロールを作成し、ポリシーを付与
11	出力	設定値の出力

演習1-4　CI/CDパイプライン構築　解答編

Step2：IAM　CDKコード

IAMロール作成　解答例（1/9）

```
// 1. インポート
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as iam from 'aws-cdk-lib/aws-iam';
import * as secretsmanager from 'aws-cdk-lib/aws-secretsmanager';

// 2. インタフェース定義
export interface IamStackProps extends cdk.StackProps {
  ecrRepoName: string;
  pipelineName: string;
  ghOwner: string;              // GitHub org/user
  ghRepo: string;               // GitHub repo
  gitHubConnectionArn?: string;   // Use CodeConnections
  appSecretArn?: string;          // アプリ用DBユーザーのSecret ARN（任意）
}
```

ポイント

1. インポート

- iam以外に、シークレットのインポートを伴うため、SecretsManagerもインポートします。

2. インタフェース定義

- ECRとPipelineのARN作成のため、ecrRepoName、pipelineNameを受け取ります。
- アプリ（ECS）にDB読み取り権限を付与するため、シークレットARNを受け取っています。

演習1-4　CI/CDパイプライン構築　解答編

Step2：IAM　CDKコード

IAMロール作成　解答例（2/9）

```
// 3. スタック初期化
export class IamStack extends cdk.Stack {
  public readonly codeBuildRole: iam.Role;
  public readonly codeDeployRole: iam.Role;
  public readonly codePipelineRole: iam.Role;
  public readonly ecsTaskExecutionRole: iam.Role;
  public readonly appTaskRole: iam.Role;
  public readonly githubOidcRole: iam.Role;

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

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

ポイント

3. スタック初期化

- public readonly xxxRole: iam.Role;
  生成したIAMロールは他サービスに付与するため、外部公開します。
- ecrRepoArn、pipelineArn
  account、regionから、ECRとPipelineのARNを動的に生成しています。

演習1-4　CI/CDパイプライン構築　解答編

Step2：IAM　CDKコード

IAMロール作成　解答例（3/9）

```
    // 4. CodeBuildロール作成
    this.codeBuildRole = new iam.Role(this, 'CodeBuildRole', {
      assumedBy: new iam.ServicePrincipal('codebuild.amazonaws.com'),
      description: 'Allows CodeBuild to push images to ECR and write logs',
    });

    // CloudWatchLogs
    this.codeBuildRole.addToPolicy(new iam.PolicyStatement({
      actions: ['logs:CreateLogGroup','logs:CreateLogStream','logs:PutLogEvents'],
      resources: ['*'],
    }));

    // ECR
    this.codeBuildRole.addToPolicy(new iam.PolicyStatement({
      actions: ['ecr:GetAuthorizationToken'],
      resources: ['*'],
    }));
    this.codeBuildRole.addToPolicy(new iam.PolicyStatement({
      actions: [
        'ecr:BatchCheckLayerAvailability','ecr:InitiateLayerUpload','ecr:UploadLayerPart',
        'ecr:CompleteLayerUpload','ecr:PutImage','ecr:BatchGetImage','ecr:GetDownloadUrlForL
ayer',
      ],
      resources: [ecrRepoArn],
    }));
```

ポイント

4. CodeBuildロール作成

- ロールを作成して、ロールにポリシーを付与するという順番でロールを作り上げます。
- CodeBuildには、CloudWatchLogs / ECR / S3 / KMS を操作するためのポリシーを付与します。
- ECR向けはGetAuthorizationTokenのみresourceを'*'としています。
- 他の操作(PutImageなど)はresourceにecrRepoArnを指定しています。ECRリポジトリ(customer-info/app)に対してのみ操作できるよう対象を絞ることで、権限を最小にしています。

演習1-4　CI/CDパイプライン構築　解答編

Step2：IAM　CDKコード

IAMロール作成　解答例（4/9）

```
    // 4. CodeBuildロール作成（続き）
    // Artifact S3/KMS
    this.codeBuildRole.addToPolicy(new iam.PolicyStatement({
      actions: ['s3:GetObject','s3:PutObject','s3:GetBucketLocation','s3:ListBucket'],
      resources: ['*'],
    }));
    this.codeBuildRole.addToPolicy(new iam.PolicyStatement({
      actions: ['kms:Decrypt','kms:Encrypt','kms:GenerateDataKey*','kms:DescribeKey'],
      resources: ['*'],
    }));

    // 5. CodeDeployロール作成
    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')
    );
```

ポイント

4. CodeBuildロール作成（続き）

- S3、KMSはCodeシリーズのサービス間で連携するArtifactをS3でやり取りするために付与します
- ArtifactがKMSで暗号化された状態でS3に保存される場合を考慮して、暗号/複号できるようKMSの権限も追加しています。

5. CodeDeployロール作成

- AWSCodeDeployRoleForECSはAWS管理ポリシーです。これをポリシーとして付与することで、一通りECSに対する操作ができるようになります。

演習1-4　CI/CDパイプライン構築　解答編

Step2：IAM　CDKコード

IAMロール作成　解答例（5/9）

```
    // 6. ECS TaskExecutionロール作成
    this.ecsTaskExecutionRole = new iam.Role(this, 'EcsTaskExecutionRole', {
      assumedBy: new iam.ServicePrincipal('ecs-tasks.amazonaws.com'),
      description: 'Execution role for ECS tasks to pull images & write logs',
    });
    this.ecsTaskExecutionRole.addManagedPolicy(
      iam.ManagedPolicy.fromAwsManagedPolicyName('service-
role/AmazonECSTaskExecutionRolePolicy')
    );

    // 7. ECS AppTaskロール作成
    this.appTaskRole = new iam.Role(this, 'AppTaskRole', {
      assumedBy: new iam.ServicePrincipal('ecs-tasks.amazonaws.com'),
      description: 'Application task role for ECS tasks',
    });

    // 8. Secrets Manager（アプリ用）
    if (props.appSecretArn) {
      const appSecret = secretsmanager.Secret.fromSecretCompleteArn(this, 'AppSecret',
props.appSecretArn);
      appSecret.grantRead(this.appTaskRole); // secretsmanager:GetSecretValue など
      appSecret.grantRead(this.ecsTaskExecutionRole);
    }
```

ポイント

6. ECS TaskExecutionロール作成

- service-role/AmazonECSTaskExecution RolePolicyはAWS管理ポリシーです。ECR pull/logsをできるようになります。

7. ECS AppTaskロール作成

- ロールは作成しますが、特別な要件がないため、権限の付与はしません。

8. Secrets Managerロール作成

- DB参照のためアプリケーションのシークレットを参照する場合のみ、Secrets Managerを読み取る権限をappTaskおよびecsTaskExecutionロールに付与します。

演習1-4　CI/CDパイプライン構築　解答編

Step2：IAM　CDKコード

IAMロール作成　解答例（6/9）

```
    // 9. CodePipelineロール作成
    this.codePipelineRole = new iam.Role(this, 'CodePipelineRole', {
      assumedBy: new iam.ServicePrincipal('codepipeline.amazonaws.com'),
      description: 'Allows CodePipeline to orchestrate Source/Build/Deploy',
    });
    this.codePipelineRole.addToPolicy(new iam.PolicyStatement({
      actions: ['codebuild:StartBuild','codebuild:BatchGetBuilds'],
      resources: ['*'],
    }));
    this.codePipelineRole.addToPolicy(new iam.PolicyStatement({
      actions:
['codedeploy:CreateDeployment','codedeploy:Get*','codedeploy:RegisterApplicationRevision'],
      resources: ['*'],
    }));
    // Artifact S3/KMS
    this.codePipelineRole.addToPolicy(new iam.PolicyStatement({
      actions: ['s3:GetObject','s3:PutObject','s3:GetBucketLocation','s3:ListBucket'],
      resources: ['*'],
    }));
    this.codePipelineRole.addToPolicy(new iam.PolicyStatement({
      actions: ['kms:Decrypt','kms:Encrypt','kms:GenerateDataKey*','kms:DescribeKey'],
      resources: ['*'],
    }));
```

ポイント

9. CodePipelineロール作成

- CodeBuild、CodeDeployを呼び出すポリシーを付与しています。BuildStart、CreateDeploymentが該当するポリシーです。
- その他はCodeBuild、CodeDeployの進捗や結果を受け取るためのポリシーです。
- Codeシリーズ間でArtifactをやりとりするため、S3、KMSの操作権限も合わせて付与します。

演習1-4　CI/CDパイプライン構築　解答編

Step2：IAM　CDKコード

IAMロール作成　解答例（7/9）

```
    // 9. CodePipelineロール作成（続き）
    // PassRole を委譲先サービスに限定
    this.codePipelineRole.addToPolicy(new iam.PolicyStatement({
      actions: ['iam:PassRole'],
      resources: [this.codeBuildRole.roleArn, this.codeDeployRole.roleArn],
      conditions: { StringEquals: { 'iam:PassedToService':
['codebuild.amazonaws.com','codedeploy.amazonaws.com'] } }
    }));

    // CodeConnection
    if (props.gitHubConnectionArn) {
      this.codePipelineRole.addToPolicy(new iam.PolicyStatement({
        actions: ['codestar-connections:UseConnection','codeconnections:UseConnection'],
        resources: [props.gitHubConnectionArn],
      }));
    }
```

ポイント

9. CodePipelineロール作成（続き）

- PassRoleを設定することで、呼び出したCodeBuild、CodeDeployに各ロールの利用を許可しています。
  BuildStart、CreateDeploymentはAPIをキックするのみで、ロールの利用許可が別で必要となります。
- CodeConnectionsを利用してGitHubにアクセスしに行くため、UseConnectionの利用ポリシーも付与しています。

演習1-4　CI/CDパイプライン構築　解答編

Step2：IAM　CDKコード

IAMロール作成　解答例（8/9）

```
// 10. GitHub OIDCプロバイダー/ロール作成
// OIDCプロバイダー作成
const provider = new iam.OpenIdConnectProvider(this, 'GitHubOIDC', {
  url: 'https://token.actions.githubusercontent.com',
  clientIds: ['sts.amazonaws.com'],
});
// main ブランチのみ許可
const subPattern = `repo:${props.ghOwner}/${props.ghRepo}:ref:refs/heads/main`;
// OIDCロール作成
this.githubOidcRole = new iam.Role(this, 'GitHubOIDCRole', {
  roleName: 'GitHubOIDCRole',
  description: 'GitHub Actions OIDC role (main branch only)',
  assumedBy: new iam.WebIdentityPrincipal(provider.openIdConnectProviderArn, {
    StringEquals: { 'token.actions.githubusercontent.com:aud': 'sts.amazonaws.com' },
    StringLike:   { 'token.actions.githubusercontent.com:sub': subPattern }
  }),
});
// Pipeline
this.githubOidcRole.addToPolicy(new iam.PolicyStatement({
  actions: ['codepipeline:StartPipelineExecution'],
  resources: [pipelineArn],
}));
```

ポイント

10. GitHub OIDCプロバイダー/ロール作成

- OIDCプロバイダーを設定することで、GitHubからAWSにアクセスできるようになります。
- OIDCロールはGitHubから指定するため、roleNameでロール名を固定しています。
- 今回はGitHub / customer-infoリポジトリのmainブランチからのみ動作するように設定しています。
- AWSに対しては、GitHubからCodePipelineをキックする権限のみ与えています。

演習1-4　CI/CDパイプライン構築　解答編

Step2：IAM　CDKコード

IAMロール作成　解答例（9/9）

```
    // 10. 出力
    new cdk.CfnOutput(this, 'CodeBuildRoleArn',        { value: this.codeBuildRole.roleArn });
    new cdk.CfnOutput(this, 'CodeDeployRoleArn',       { value: this.codeDeployRole.roleArn });
    new cdk.CfnOutput(this, 'CodePipelineRoleArn',     { value: this.codePipelineRole.roleArn });
    new cdk.CfnOutput(this, 'ECSTaskExecutionRoleArn', { value:
this.ecsTaskExecutionRole.roleArn });
    new cdk.CfnOutput(this, 'AppTaskRoleArn',          { value: this.appTaskRole.roleArn });
    new cdk.CfnOutput(this, 'GitHubOIDCRoleArn',       { value: this.githubOidcRole.roleArn });
    new cdk.CfnOutput(this, 'GitHubOIDCRoleName',      { value: this.githubOidcRole.roleName
});
    new cdk.CfnOutput(this, 'TargetPipelineArn',       { value: pipelineArn });
    new cdk.CfnOutput(this, 'TargetEcrRepoArn',        { value: ecrRepoArn });
  }
}
```

ポイント

11. 出力

- 特にありません

演習1-4　CI/CDパイプライン構築　解答編

Step2：IAM　コード実行

- CDKから**IamStack**を実行し、新規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　解答編

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　解答編

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ます。

- 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　解答編

Step3：GitHub　AWSとの接続

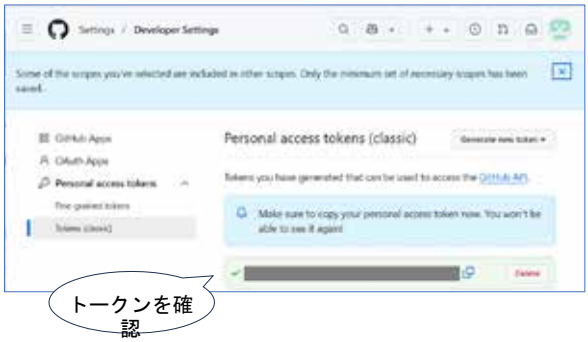
- AWS CloudShellを利用して、GitHubのリポジトリを作成します。
  GitHubに「**customer-info**」を作成します。

1. GitHub PAT取得

- GitHubにログインし、トークン(PAT*)を取得・コピーします。
  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 (classic) をクリック
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　解答編

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
```

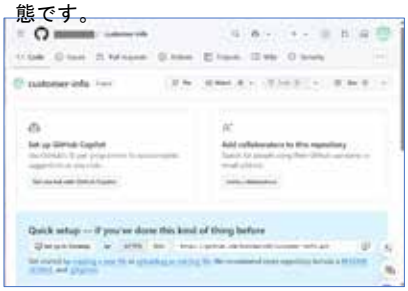
3. リポジトリ作成

```
// 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。現段階では**「空」**の状態です。



演習1-4　CI/CDパイプライン構築　解答編

Step3：GitHub　AWSとの接続

4. GitHubへの変数登録

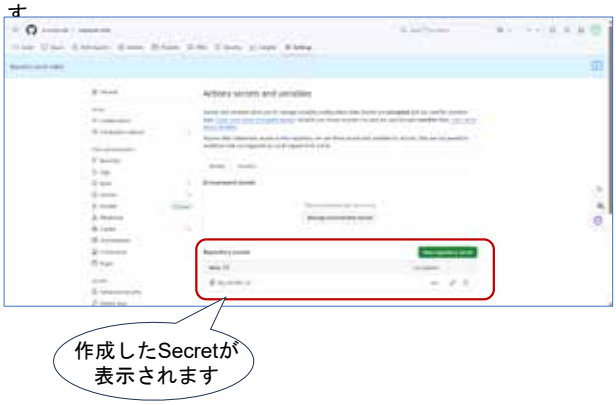
- customer-infoリポジトリのGitHub Action動作時に必要な変数をGitHubのSecretに登録します。

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



演習1-4　CI/CDパイプライン構築　解答編

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ます。
  Dockerfileでコンパイルできるようにします。

- 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)を定義します。

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　解答編

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ます。
準備したコードはcustomer-info配下にpushします。

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
- 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
  - /: 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
  - /healthcheck: ヘルスチェック用エンドポイント
    常に「200 OK」を返却


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

演習1-4　CI/CDパイプライン構築　解答編

Step3：GitHub　アプリケーションコード

■ GitHubに格納する各ファイルについて解説します。 /app/app.pyのコードは以下の通りです。

/app/app.py　解答例（1/4）

```
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"])
```

ポイント

- Flaskを利用した簡易なWebアプリを作ります。リクエストに対し、SecretsManagerで取得した認証情報を使って、RDS(MySQL)に接続して、 Web画面にテーブル内容を表示します。

- Flaskを初期化し、SecretsManagerからDB情報を取得しています。
- DB_SECRET_NAMEを参照して、username / passwordを取得しています。

演習1-4　CI/CDパイプライン構築　解答編

Step3：GitHub　アプリケーションコード

/app/app.py　解答例（2/4）

```
# 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")

# RDS　MySQLとのコネクション作成
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,
    )
```

ポイント

- DBに関する情報と取得した認証情報から、get_connectionで、MySQLとのコネクションを生成しています。

演習1-4　CI/CDパイプライン構築　解答編

Step3：GitHub　アプリケーションコード

/app/app.py　解答例（3/4）

```
# === 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

# === 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()
```

ポイント

- app.route(“/”)でシンプルに稼働状況をレスポンスするページを用意しています。DB接続を伴うパスとヘルスチェックのパスは分けた方が良いです。
- 今回ALBのヘルスチェックのパスは”/”に設定にしていますが、通常は”/healthcheck”をヘルスチェックさせる方が良いでしょう。
- app.route(“/customers”)で、DB情報を一覧で出力するWebページを用意しています。デプロイ後、<ALB_DNS>/customersでブラウズすると顧客情報一覧のページが表示されます。
- cur.executeでcustomersテーブルの情報を検索して配列に格納します。
- htmlで返却するWebページのHTMLをべた書きしています。

演習1-4　CI/CDパイプライン構築　解答編

Step3：GitHub　アプリケーションコード

/app/app.py　解答例（4/4）

```
# HTMLの表作成
        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
```

ポイント

- forで配列を回して、取得してテーブル情報をHTMLとして表示します。今回、customersテーブルには5件登録しているため、5行分の情報が表示される想定です。
- DB接続に失敗した際は、Exceptionで500エラーを返します。

演習1-4　CI/CDパイプライン構築　解答編

Step3：GitHub　アプリケーションコード

■ Dockerfileのコードは以下の通りです。

Dockerfile　解答例

```
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
```

ポイント

- Dockerfileの内容をベースにコンテナイメージを固めていきます。
- 今回アプリケーションのベースイメージは軽量なpython:3.11-slimを利用します。他Versionのイメージを利用しても大丈夫です。
  ビルドタイミングでベースイメージが変わるのを防ぐため、本番環境での”latest”の利用は控えてください。
- requirements.txtに書かれたパッケージをインストールして、app.pyとnginx.confを/etc配下のフォルダに配置しています。
- 最後CMDで、nginx をバックグラウンド、Flask アプリを実行する gunicorn をフォアグラウンドで動かします。gunicorn はコンテナ内部で 127.0.0.1:5000 を listen しており、外部からのリクエストはまず Nginx (80番ポート) が受け取り、Nginxから gunicorn (5000 番ポート) へリバースプロキシされます。

演習1-4　CI/CDパイプライン構築　解答編

Step3：GitHub　アプリケーションコード

■ nginx/nginx.conf、.gitignoreのコードは以下の通りです。

nginx/nginx.conf　解答例

```
server {
    listen 80;

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

.gitignore　解答例

```
__pycache__/
*.py[cod]
.venv/
.env
*.log
.vscode/
```

ポイント

nginx/nginx.conf

- Nginxのルーティングを設定します。
- 外部リクエストは80番ポートで受け付けます。
- proxy_passでコンテナ内部にリバースプロキシするようにしています。ECSのヘルスチェック用に/healthcheckルートを分けて用意しています。

.gitignore

- GitHubリポジトリに含めないファイルを定義しています。
- pythonのキャッシュやログ、エディタ情報などを管理対象外に指定しています。

演習1-4　CI/CDパイプライン構築　解答編

Step3：GitHub　アプリケーションコード

■ requirement.txt、.dockerignoreのコードは以下の通りです。

requirements.txt　解答例

```
Flask
gunicorn
pymysql
boto3
```

.dockerignore　解答例

```
__pycache__
*.pyc
*.pyo
.git
.vscode
```

ポイント

requirement.txt

- Pythonプロジェクトで利用するパッケージを一覧にしています。今回は、Python、MySQL、AWSサービスの操作に必要なパッケージを4つ定義しています。
- 各パッケージのバージョンを指定しても良いです。

.dockerignore

- Dockerイメージに含めないファイルを定義しています。
- キャッシュやログなどを含めないようここでは設定しています。

演習1-4　CI/CDパイプライン構築　解答編

Step3：GitHub　アプリケーションコード

■ /workflows/ci.ymlのコードは以下の通りです。

/workflows/ci.yml　解答例（1/2）

```
name: CI (Kick CodePipeline)

on:
  push:
    branches: [ main ]
  pull_request:
permissions:
  id-token: write
  contents: read
```

ポイント

- GitHub Actionsのワークフローを定義します。今回はOIDC認証して、CodePipelineを起動するだけのワークフローです。ワークフローで試験やビルドもできますが、CodeBuildに任せています。
- “on”でmainブランチへのpush、pull requestをトリガーにワークフローが動くように設定しています。運用フローに応じて、「どのブランチで」「どんな操作」をされた時にワークフローが起動するかを検討して設定してください。
- premissionsを設定することで、OIDCでロールを引き受けられます。特にid-token:write は必須です。

演習1-4　CI/CDパイプライン構築　解答編

Step3：GitHub　アプリケーションコード

/workflows/ci.yml　解答例（2/2）

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

ポイント

- role-to-assumeで、GitHubのOIDCトークンで AWSのGitHubOIDCRoleを利用できるようになり、GitHubからAWSを操作可能になります。
- 最後にCodePipelineをキックして、ワークフローは終了です。

演習1-4　CI/CDパイプライン構築　解答編

Step3：GitHub　アプリケーションコード

6. GitHub/customer-infoへのコード登録

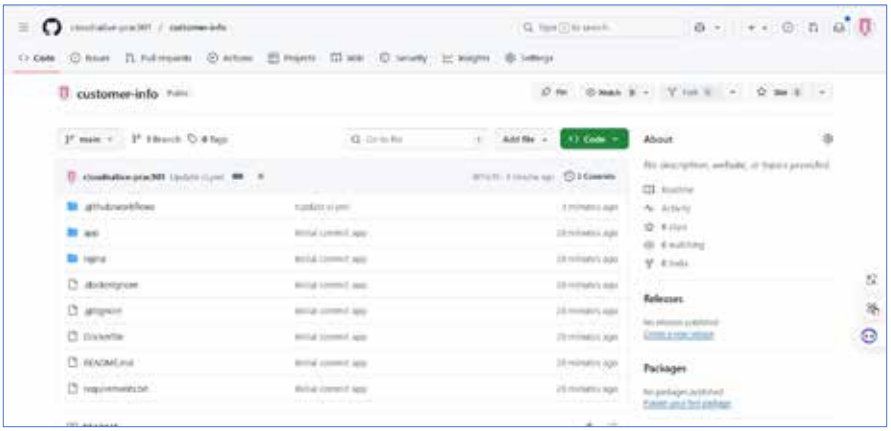
```
// GitHubにpush
github $ git config --global user.name "<Your Name>"
github $ git config --global user.email "<your.email@example.com>"

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　解答編

Step3：GitHub　動作確認

- git pushが完了すると、customer-infoリポジトリのmainブランチに各ファイルが反映されます。



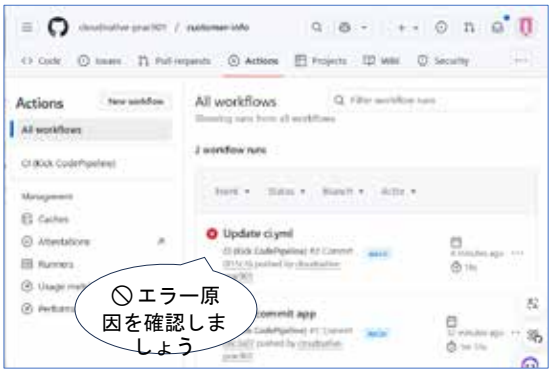
GitHub customer-infoリポジトリ

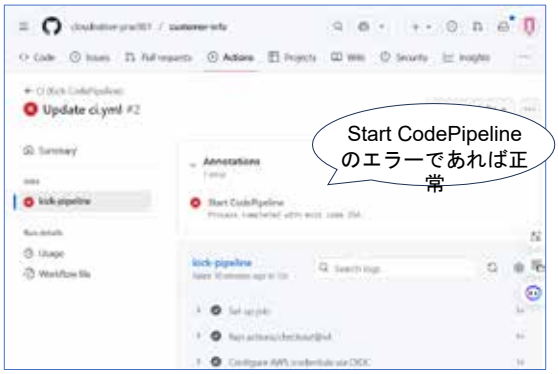
演習1-4　CI/CDパイプライン構築　解答編

Step3：GitHub　動作確認

■ ci.yml登録後にファイルをpushするとGitHub Actionsが自動で動作します。
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の走行結果**を確認します。
  - 「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。





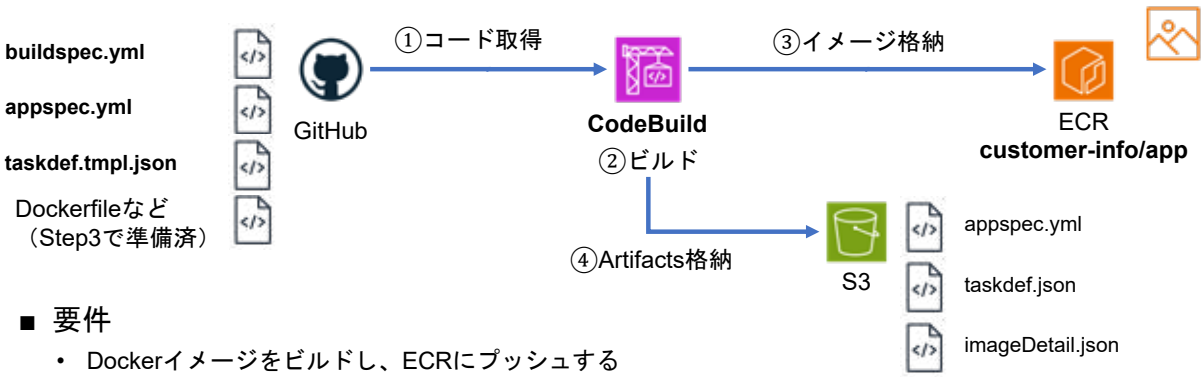
演習1-4 Step4　解答編

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　解答編

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を実装します。



■ 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　解答編

Step4：CodeBuild　設計

■ CodeBuildが実行するビルド手順を**buildspec.yml等に定義し**、GitHubに配置します。
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　解答編

Step4：CodeBuild　GitHubコード

■ buildspec.ymlの全体構成は以下の通りです。

コード構成

#	構成	役割
1	env	環境変数の宣言
2	pre_build	ビルド前の工程（ECRログイン、タグ生成などを実施）
3	└ECRログイン	ECRにログイン
4	└ロール/Secretの解決	ロール、シークレットをCloudFormationから取得
5	└イメージタグの生成	ビルドしたコンテナイメージに付与するタグの生成
6	build	ビルド工程（Dockerビルドし、コンテナイメージを作成）
7	post_build	ビルド後の工程（ECRプッシュ、imageDetail.jsonの出力）
8	└imageDetail.json生成	CodeDeployが参照するimageDetail.jsonを生成
9	└taskdef.json生成	CodeDeployが参照するtaskdef. jsonを生成
10	artifacts	アーティファクトを出力

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　GitHubコード

buildspec.yml　解答例（1/4）

```
version: 0.2
env:
  variables:
    ECR_REPO: "customer-info/app“              # ECR リポジトリ名
    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:-}"
```

ポイント

env

- git-credential-helper:no
  Gitの資格情報ヘルパは利用せず、CodePipeline側で処理させます。

pre_build

- イメージビルドの事前処理を行います。
- set –eu
  スクリプト実行を安全に実行するおまじないです。
- ECRにログインし、イメージpushする準備を整えます。
- パイプラインから渡された変数を設定して、後処理で利用します。

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　GitHubコード

buildspec.yml　解答例（2/4）

```
    # 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/${ECR_REPO}:${IMAG
E_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} -> ${IMAGE_URI}"
    - docker build -f "${DF}" -t "${IMAGE_URI}" "${CTX}"
```

ポイント

pre_build（続き）

- イメージに付与するタグを生成します。今回はブランチ名、マージした日時、CodeBuildの通番を付与しています。
  本番運用では、タグの命名規則に従って生成してください。例えば、デプロイ環境、セマンティックバージョン、日時などをタグに付与します。
- IMAGE_URIはECRに保存されるイメージURIになるよう生成しています。
- appspec.yml、taskdef.tmpl.jsonの存在有無を確認し、なければ処理を終了させています。

build

- Dockerfileのパスを作り、docker buildを実行します。
- docker buildと同時に、タグも付与しています。

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　GitHubコード

buildspec.yml　解答例（3/4）

```
 # 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"
```

ポイント

post_build

- まず docker push によりビルドしたイメージを ECR にプッシュし、その URI を CodeDeploy に渡すために imageDetail.json を生成します。
- 続いて、後続処理に必要となる jq と gettext（envsubst コマンド用）を CodeBuild 実行環境にインストールします。環境によりyumかaptが変わりうるため、分岐処理を用意しています。
- アプリのタスク定義はテンプレートファイル taskdef.tmpl.json として管理しており、変数プレースホルダを含んでいます。export した変数を envsubstで置換することで、taskdef.json を生成します。
- テンプレートファイルや S3 アーティファクトの配置パスが buildspec.yml や CodeBuild 設定と食い違っていたり、変数の置換が正しく行われないと、CodeDeploy が taskdef.json を読み取れずエラーになることがあるため注意が必要です。

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　GitHubコード

buildspec.yml　解答例（4/4）

```
    # DB未導入の場合は 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}"
artifacts:
  files:
    - deploy/ecs/appspec.yml
    - deploy/ecs/taskdef.json
    - imageDetail.json
  discard-paths: no
```

ポイント

post_build(続き)

- アプリケーション用のDBシークレットが存在しない場合、taskdef.jsonから、.containerDefinitions[0]を削除することで、SecretsManager連携なしでもECSタスクを起動できるようにしています。

artifacts

- S3アーティファクトの出力先を設定しています。
- Discard-paths:noとしています。yesにした場合、S3/buildArtifactの直下に全アーティファクトが出力されるようになります。

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　GitHubコード

■ 続いて、appspec.ymlのコードは以下の通りです。

appspec.yml　解答例

```
version: 0.0
Resources:
  - TargetService:
      Type: AWS::ECS::Service          # ECSのサービスをターゲット指定
      Properties:
        TaskDefinition: <TASK_DEFINITION>  # CodeDeployのタスク定義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
```

ポイント

- appspec.ymlで、CodeDeployに、どのサービスにどのコンテナ/ポートでトラフィックを流すかを定義しています。
- TaskDefinitionでtaskdef.jsonで生成されたタスク定義の引数を設定します。
- LoadBalancerInfoでALBに紐づけるコンテナ名とポートを指定して、トラフィックの切り替えを実現します。

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　GitHubコード

■ 続いて、taskdef.tmpl.jsonのコードは以下の通りです。

taskdef.tmpl.json　解答例（1/2）

```
{
  "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"
     }
    },
```

ポイント

- ECSのタスク定義用のテンプレートファイルで、変数を置換して、taskdef.jsonを生成するのに利用しています。
- タスク定義に必要な項目を設定しており、タスク定義名から、Fargateのリソースやロールを指定しています。
- containerDefinitionsで起動するコンテナを定義しています。起動するイメージのURI、ポートマッピングなどを行います。
- なお、logConfigurationを設定しているので、CloudWatch Logsにログが連携されるため、タスクに関するログを確認することができます。
  コンテナが起動しない場合などトラブルシュートに利用してください。

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　GitHubコード

taskdef.tmpl.json　解答例（2/2）

```
    "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" }
}
```

ポイント

- コンテナ内のヘルスチェックを設定しています。
  ヘルスチェックの間隔などはアプリケーション毎に調整してください。
- 今回のアプリケーションはDB参照するため、SecretsManagerで管理されているRDSのシークレット情報を引き渡すようにしています。
  （今回はapp_userのシークレット）
- シークレットの引き渡しができないと、DB接続できないので忘れず設定してください。
- runtimePlatformでFargateのOS / アーキテクチャを指定しています。Fargateの場合、Linux / X86_64がデフォルトです。

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　CDKコード

- AWS CDKを利用して、**Buildスタックを作成するコード**を作成します。

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

```
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

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　解答編

Step4：CodeBuild　CDKコード

- bin/ecs-demo.tsのコードは以下の通りです。

```ts
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';
import { AlbStack } from '../lib/alb-stack';
import { EcrStack } from '../lib/ecr-stack';
import { RdsStack } from '../lib/rds-stack';
import { EcsStack } from '../lib/ecs-stack';
import { ConnectionStack } from '../lib/connection-stack';
import { IamStack } from '../lib/iam-stack';
import { BuildStack } from '../lib/build-stack';  // ★追加

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 });

・・・省略・・・
```

```ts
// CodeConnection
const conn = new ConnectionStack(app, 'ConnectionStack', { env });

// IAM
const iam = new IamStack(app, 'IamStack', {
  env,
  ecrRepoName : 'customer-info/app',
  pipelineName: 'CustomerInfoPipeline',
  ghOwner     : 'xxx',  // GitHubアカウント
  ghRepo      : 'customer-info',  // GitHubリポジトリ
  gitHubConnectionArn: conn.connectionArn,
  appSecretArn: rds.appSecret.secretArn,  // アプリ用の認証情報
});

// CodeBuild　★追加
const build = new BuildStack(app, 'BuildStack', {
  env,
  codeBuildRoleArn: iam.codeBuildRole.roleArn,
  ecrRepoName: 'customer-info/app',
});
```

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　CDKコード

■ lib/build-stack.tsの全体構成は以下の通りです。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インタフェース定義	他スタックから渡されるパラメータ
3	スタック初期化	BuildStackを初期化
4	IAMロール形式確認	IAMロールARNの形式確認（任意）
5	ECRリポジトリ設定	ECRリポジトリを設定
6	CodeBuildロールのインポート	CodeBuildロールをインポート
7	CodeBuildプロジェクト作成	CodeBuildのビルドプロジェクトを作成する
8	出力	設定値の出力

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　CDKコード

CodeBuild作成　解答例（1/4）

```
// 1. 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';
import { Token } from 'aws-cdk-lib'; // Token 判定に使用

// 2. 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
}

// 3. スタック初期化
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;
```

ポイント

1. インポート
- Codebuild、iamをインポートします。
- トークン関連の処理を伴うため、tokenもインポートしています。

2. インタフェース定義
- CodeBuildを動かす際に必要なcodeBuildRoleArnを読み込みます。

3. スタック初期化
- 今回は単一AWSアカウントですが、CodePipeline/CodeBuildとECR / ECSが別AWSアカウントになる場合もあります。複数AWSアカウントが混在する環境では、デプロイ先のアカウント / リージョンを変数にセットして利用してください。

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　CDKコード

CodeBuild作成　解答例（2/4）

```
    // 4. IAMロールARNの形式確認
    const arnRegex = /^arn:aws:iam::\d{12}:role\/.+$/;
    const validateArnIfLiteral = (label: string, value: string) => {
      if (!Token.isUnresolved(value) && !arnRegex.test(value)) {
        throw new Error(`${label} is invalid: ${value}`);
      }
    };
    validateArnIfLiteral('codeBuildRoleArn', props.codeBuildRoleArn);

    // 5. ECRリポジトリの設定
    const repoName    = props.ecrRepoName ?? 'customer-info/app';
    const ecrRegistry = `${account}.dkr.ecr.${region}.amazonaws.com`;

    // 6. CodeBuildRoleのインポート
    const role = iam.Role.fromRoleArn(this, 'ImportedCodeBuildRole', props.codeBuildRoleArn,
{
      mutable: false,
    });
```

ポイント

4. IAMロールARN形式確認（任意）

- 手入力でARNが渡された場合に形式を検証し、手入力のミスを早めに検知するようにしています。またtokenの場合は、スキップとしています。

5. ECRリポジトリの設定

- ECRリポジトリ名とURIを設定します。

6. CodeBuildRoleインポート

- IamStackで作成したCodeBuildロールをインポートすることで、docker build/push、ECR、S3などのアクセス・操作権限が設定されます。

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　CDKコード

CodeBuild作成　解答例（3/4）

```
    // 7. 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;
```

ポイント

7. CodeBuildプロジェクト作成

- CodeBuildのプロジェクトを生成します。
- environmentで、ビルド環境および権限を制御しています。
  - STANDARD_7_0はAmazonLinux2023ベースのコンテナイメージで、これベースにビルドが進みます。
  - ビルドは特権モードで起動しています。特権モードでないと、docker build、docker pushができないため必須ま設定です。
- buildspecがGitHubリポジトリのどこにあるかを指定して実行させます。リポジトリ直下に配置するのが普通です。
- environmentVariablesは、buildspec.yml実行時に自動的に引渡す値を設定しました。

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　CDKコード

CodeBuild作成　解答例（4/4）

```
    // 8. 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         });
  }
}
```

ポイント

9. 出力

- 特にありません

演習1-4　CI/CDパイプライン構築　解答編

Step4：CodeBuild　動作確認

■ BuildStackを実行し、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。

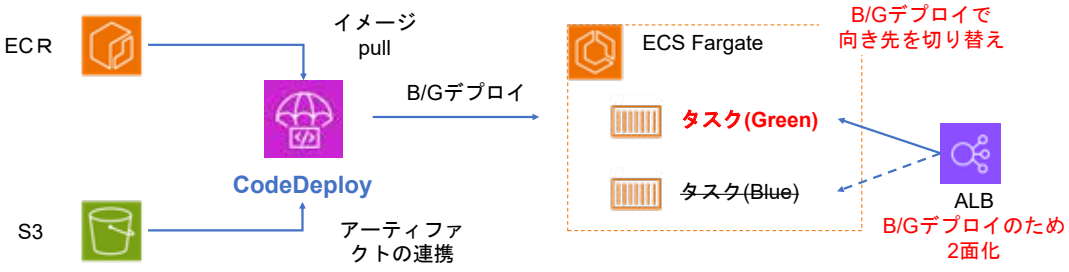
演習1-4 Step5　解答編

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　解答編

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**します。
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　解答編

Step5：CodeDeploy　CDKコード

- AWS CDKを利用して、**DeployStackを作成するコード**を作成します。

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

```
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
・・・
```

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

- **bin/ecs-demo.ts**から、DeployStack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 本番用リスナー
	testListenerArn	alb.listenerTest.listenerArn,	ALB テスト用リスナー
	tgBlueName	alb.tgBlue.targetGroupName	ALB ターゲットグループ（Blue）
	tgGreenName	alb.tgGreen.targetGroupName	ALB ターゲットグループ（Green）
IAM	codeDeployRoleArn	iam.codeDeployRole.roleArn	CodeDeploy用IAMロール
CodeDeploy	applicationName	'CustomerInfoEcsApp'	CodeDeploy アプリケーション名
	deploymentGroupName	'CustomerInfoDG'	CodeDeploy デプロイグループ名

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

■ bin/ecs-demo.tsのコードは以下の通りです。

taskdef.tmpl.json　解答例（1/2）

```
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';
import { AlbStack } from '../lib/alb-stack';
import { EcrStack } from '../lib/ecr-stack';
import { RdsStack } from '../lib/rds-stack';
import { EcsStack } from '../lib/ecs-stack';
import { ConnectionStack } from '../lib/connection-stack';
import { IamStack } from '../lib/iam-stack';
import { BuildStack } from '../lib/build-stack';
import { DeployStack } from '../lib/deploy-stack';  // ★追加

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 });

・・・省略・・・
```

```
・・・省略・・・

// RDS
const rds = new RdsStack(app, 'RdsStack', {
  env,
  vpc: net.vpc,
  dbSg: net.dbSg,
});


// ECS / Fargate
const ecs = new EcsStack(app, 'EcsStack', {
  env,
  vpc       : net.vpc,
  ecsSg     : net.ecsSg,
  repo      : ecr.repository,
  targetGroup: alb.tgBlue,                //★プロパティ変更
  //targetGroup: alb.targetGroup,        //★プロパティ変更のためコメントアウト
});

・・・省略・・・
```

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

bin/ecs-demo.ts　解答例（2/2）

```
・・・省略・・・

// CodeBuild
const build = new BuildStack(app, 'BuildStack', {
  env,
  codeBuildRoleArn: iam.codeBuildRole.roleArn,
  ecrRepoName: 'customer-info/app',
});

// CodeDeploy　★追加
const deploy = new DeployStack(app, 'DeployStack', {
  env,
  clusterName: ecs.cluster.clusterName,
  serviceName: ecs.service.serviceName,
  prodListenerArn: alb.listenerProd.listenerArn,
  testListenerArn: alb.listenerTest.listenerArn,
  tgBlueName: alb.tgBlue.targetGroupName,
  tgGreenName: alb.tgGreen.targetGroupName,
  codeDeployRoleArn: iam.codeDeployRole.roleArn,
  applicationName: 'CustomerInfoEcsApp',
  deploymentGroupName: 'CustomerInfoDG',
});
```

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

■ 次に、lib/net-stack.tsの修正箇所は以下の通りです。

NetStack　修正例

```
・・・省略・・・

   // 6. セキュリティグループ間の通信ルール
   // ALB: InternetからHTTP/HTTPSを許可
   this.albSg.addIngressRule(ec2.Peer.anyIpv4(), ec2.Port.tcp(80), 'Allow HTTP from
Internet');
   //★追加（B/Gデプロイ - テストリスナ用ポート）
   this.albSg.addIngressRule(ec2.Peer.anyIpv4(), ec2.Port.tcp(9001), 'Allow test listener
(9001) from Internet');
   this.albSg.addEgressRule(this.ecsSg, ec2.Port.tcp(80), 'ALB-to-ECS');

・・・省略・・・
```

ポイント

6. セキュリティグループ間の通信ルール

- ALBのテストリスナーの設定に伴い、ALBのSGにtcp(9001)の通信を許可する必要があります。そこで、AlbSgのインバウンドルールに、テストリスナー用のポート:tcp(9001)を追加します。
- その他部分は修正不要です。

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

■ 続いて、lib/alb-stack.tsの修正箇所は以下の通りです。

AlbStack　修正例（1/4）

```
・・・省略・・・

// 3. 公開プロパティ
export class AlbStack extends Stack {
  public readonly albDnsName: string;
  //public readonly targetGroup: elbv2.ApplicationTargetGroup; // ★tgを2つに変更するためコメ
ントアウト
  // ★B/Gデプロイ用に、本番およびテスト用のリスナー、ターゲットグループを2つずつ定義
  public readonly listenerProd: elbv2.ApplicationListener; // ★本番(80)
  public readonly listenerTest: elbv2.ApplicationListener; // ★テスト(9001)
  public readonly tgBlue : elbv2.ApplicationTargetGroup;   // ★初期：本番
  public readonly tgGreen: elbv2.ApplicationTargetGroup;   // ★初期：テスト

・・・省略・・・
```

ポイント

3. 公開プロパティ

- 既存ターゲットグループ(targetGroup)を削除し、代わりに、B/Gデプロイ用のリスナー2つ、ターゲットグループ2つを公開します。
- targetGroupをそのまま利用することも可能ですが、B/Gデプロイ用に分かりやすくするため、tgBlue、tgGreenを定義して利用する形としました。

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

AlbStack　修正例（2/4）

```
・・・省略・・・

  // 6. 空ターゲットグループ（後で Fargate を登録）
  // ★初期ターゲットグループ(tg)は不要となるため、コメントアウト
  //const tg = new elbv2.ApplicationTargetGroup(this, 'AlbTg', {
  //  vpc: props.vpc, port: 80, protocol: elbv2.ApplicationProtocol.HTTP,
  //  targetType: elbv2.TargetType.IP,
  //  healthCheck: { path: '/', interval: Duration.seconds(30) },
  //});
  //this.targetGroup = tg;

  // ★Blue/Green 用ターゲットグループ（HTTP:80）
  this.tgBlue = new elbv2.ApplicationTargetGroup(this, 'TgBlue', {
    vpc: props.vpc, port: 80, protocol: elbv2.ApplicationProtocol.HTTP,
    targetType: elbv2.TargetType.IP,
    healthCheck: { path: '/', interval: Duration.seconds(30) },
  });
  this.tgGreen = new elbv2.ApplicationTargetGroup(this, 'TgGreen', {
    vpc: props.vpc, port: 80, protocol: elbv2.ApplicationProtocol.HTTP,
    targetType: elbv2.TargetType.IP,
    healthCheck: { path: '/', interval: Duration.seconds(30) },
  });

・・・省略・・・
```

ポイント

6. ターゲットグループ作成

- 既存ターゲットグループを削除し、Blue用Green用のターゲットグループを2つ生成します。
- B/Gデプロイするアプリケーションも Fargateのため、 targetTypeはIPを指定するのは変わりません。
- ALBのヘルスチェック先のパスは、”/healthcheck”が一般的です。
- ただ今回初期デプロイするアプリがNginxで”/healthcheck”を持たないため、”/”をヘルスチェック先に設定しています。

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

AlbStack　修正例（3/4）

```
・・・省略・・・

  // 7. HTTPリスナーの作成
  //const listener = alb.addListener('HttpListener', {
  //  port: 80, protocol: elbv2.ApplicationProtocol.HTTP,
  //  defaultTargetGroups: [tg],
  //});
  // ★Blue/Green 用リスナー
  // ★本番リスナー (80) : 初期はBlueを本番に適用
  this.listenerProd = alb.addListener('HttpListenerProd', {
    port: 80,
    protocol: elbv2.ApplicationProtocol.HTTP,
    defaultTargetGroups: [this.tgBlue],
  });
  // ★テストリスナー (9001) : 初期はGreenをテストに適用
  this.listenerTest = alb.addListener('HttpListenerTest', {
    port: 9001,
    protocol: elbv2.ApplicationProtocol.HTTP,
    defaultTargetGroups: [this.tgGreen],
  });

・・・省略・・・
```

ポイント

7. HTTPリスナーの作成

- 既存リスナーの代わりに、本番用 / テスト用のリスナーを作成します。
- 初期のターゲットグループとリスナーの紐づけは以下です。B/Gデプロイを行うたびに紐づけが分かります。
  - 本番リスナーとBlue側ターゲットグループ
  - テストリスナーとGreen側ターゲットグループ

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

AlbStack　修正例（4/4）

```
・・・省略・・・

    // 12. 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        });
    // ★本番/テスト用リスナーの出力を追加
    new CfnOutput(this, 'ProdListenerArn',   { value: this.listenerProd.listenerArn });
    new CfnOutput(this, 'TestListenerArn',   { value: this.listenerTest.listenerArn });
    // ★B/Gデプロイ用ターゲットグループの名前およびARNを追加
    new CfnOutput(this, 'TgBlueName',        { value: this.tgBlue.targetGroupName });
    new CfnOutput(this, 'TgGreenName',       { value: this.tgGreen.targetGroupName });
    new CfnOutput(this, 'TgBlueArn',         { value: this.tgBlue.targetGroupArn });
    new CfnOutput(this, 'TgGreenArn',        { value: this.tgGreen.targetGroupArn });
  }
}
```

ポイント

12. 出力

- targetGroupの代わりに、TG2つ、リスナー2つをCfnOutPutに出力します。
- 設定すれば、「3. 公開プロパティ」と「12.出力」のどちらからでも変数を外部から引くことは可能ですが、どちらから外部参照させるのか方針を予め決めておくとコードが書きやすくなります。

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

■ 最後はlib/deploy-stack.tsです。全体構成は以下の通りです。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インタフェース定義	他スタックから渡されるパラメータ
3	スタック初期化	DeployStackを初期化
4	CodeDeploy アプリケーション作成	CodeDeployのアプリケーションを作成
5	CodeDeploy デプロイメントグループ作成	CodeDeployのデプロイメントグループを作成
6	デプロイ方式	B/Gデプロイの指定および切り替え方法を設定
7	ECSサービスとALBの紐付け	ECSサービスとALBのリスナー、ターゲットグループを接続
8	出力	設定値の出力

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

CodeBuild作成　解答例（1/4）

```
// 1. インポート
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';

// 2. インタフェース定義
export interface DeployStackProps extends cdk.StackProps {
  clusterName: string;  // ECSクラスター名
  serviceName: string;  // ECSサービス名
  prodListenerArn: string;  // 本番リスナー
  testListenerArn: string;  // テストリスナー
  tgBlueName: string;       // ターゲットグループ(Blue)
  tgGreenName: string;      // ターゲットグループ(Green)
  codeDeployRoleArn: string;  // CodeDeploy用IAMロール
  applicationName: string;    // CodeDeploy アプリケーション名
  deploymentGroupName: string; // CodeDeploy デプロイメントグループ名
}

// 3. スタック初期化
export class DeployStack extends cdk.Stack {
  public readonly application: codedeploy.CfnApplication;
  public readonly deploymentGroup: codedeploy.CfnDeploymentGroup;
  constructor(scope: Construct, id: string, props: DeployStackProps) {
    super(scope, id, props);
```

ポイント

2. インタフェース定義

- ECSでB/Gデプロイするため、ECSおよびALB関連のパラメータを中心に外部から読みこみます。

3. スタック初期化

- 外部参照できるようCodeDeployに作成するアプリケーションおよびデプロイメントグループを外部公開します。

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

CodeBuild作成　解答例（2/4）

```
    // 4. CodeDeployのApplication作成
    this.application = new codedeploy.CfnApplication(this, 'EcsApplication', {
      applicationName: props.applicationName,
      computePlatform: 'ECS',
    });
    // CodeDeployサービスロールの参照
    const serviceRole = iam.Role.fromRoleArn(
      this,
      'CdServiceRole',
      props.codeDeployRoleArn,
      { mutable: false }
    );

    // 5. CodeDeployのデプロイメントグループ作成(ECS Blue/Green)
    this.deploymentGroup = new codedeploy.CfnDeploymentGroup(this,
'EcsDeploymentGroup', {
      applicationName: this.application.ref,
      deploymentGroupName: props.deploymentGroupName,
      serviceRoleArn: serviceRole.roleArn,
      // 失敗時のロールバック
      autoRollbackConfiguration: {
        enabled: true,
        events: ['DEPLOYMENT_FAILURE', 'DEPLOYMENT_STOP_ON_ALARM',
'DEPLOYMENT_STOP_ON_REQUEST'],
      },
```

ポイント

4. CodeDeployのアプリケーション作成

- computePlatform:ECSを忘れず選択してください。デプロイ先をECSに指定します。
- IamStackで作成したロールをCodeDeployに紐づけします。

5. CodeDeploy デプロイメントグループ作成

- デプロイに失敗した場合は、前のタスクにロールバックさせます。”events”でどのイベントの時にロールバックするかを設定しています。
  - DEPLOYMENT_FAILURE
    →デプロイ失敗時
  - DEPLOYMENT_STOP_ON_ALARM
    →CloudWatch Alarmの発報時
  - DEPLOYMENT_STOP_ON_REQUEST
    →デプロイメントの手動停止時

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

CodeBuild作成　解答例（3/4）

```
    // 6. デプロイ方式（Blue/Green + トラフィックコントロール）
    deploymentStyle: {
      deploymentOption: 'WITH_TRAFFIC_CONTROL',
      deploymentType: 'BLUE_GREEN',
    },

    // Blue/Green の切替挙動
    blueGreenDeploymentConfiguration: {
      terminateBlueInstancesOnDeploymentSuccess: {
        action: 'TERMINATE',
        terminationWaitTimeInMinutes: 5,
      },
      deploymentReadyOption: {
        actionOnTimeout: 'STOP_DEPLOYMENT', // 承認ゲート入れない場合は
CONTINUE_DEPLOYMENT
        waitTimeInMinutes: 300,                    // 承認を待つ時間
      },
    },

    // 監視アラーム（必要なら enabled: true にして Alarm を渡す）
    alarmConfiguration: { enabled: false },
```

ポイント

6. デプロイ方式

- WITH_TRAFFIC_CONTROL、BLUE_GREENを指定することで、本番/テストリスナーを使ったトラフィック制御を可能とし、ダウンタイムなしでのB/Gデプロイメントを実現します。
- terminateBlueInstancesOnDeploymentSuccessでデプロイ成功時のBlue側のインスタンスの扱いを決められます。今回はTERMINATEで削除されます。
- actionOnTimeoutでデプロイ前の承認有無を選択できます。本番運用を考えると、STOP_DEPLOYMENTとして、手動承認プロセスを挟んだ方が良いです。
- デプロイに関して監視設定を入れるのであれば、alarmConfigurationをtrueにして、アラーム設定できるようにしてください。

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

CodeBuild作成　解答例（4/4）

```
    // 7. ECS サービスと ALB（Listener/TG）を接続
    ecsServices: [{ clusterName: props.clusterName, serviceName: props.serviceName }],
    loadBalancerInfo: {
      targetGroupPairInfoList: [{
        prodTrafficRoute: { listenerArns: [props.prodListenerArn] },
        testTrafficRoute: { listenerArns: [props.testListenerArn] },
        targetGroups: [{ name: props.tgBlueName }, { name: props.tgGreenName }],
      }],
    },
  });

  // 8. 出力
  this.deploymentGroup.addDependency(this.application);
  new cdk.CfnOutput(this, 'EcsAppName', { value: this.application.applicationName! });
  new cdk.CfnOutput(this, 'EcsDeploymentGroupName', { value:
props.deploymentGroupName });
  }
}
```

ポイント

7. ECSサービスとALBの接続

- サービスとALBを紐づけるにあたり、本番リスナーとテストリスナーをトラフィックルートに設定し、それぞれBlue/Greenのターゲットグループに紐づけています。

演習1-4　CI/CDパイプライン構築　解答編

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す。
  AlbSgにインバウンドルール2つ、ALBにリスナー、TGが2つずつ登録されていれば大丈夫です。

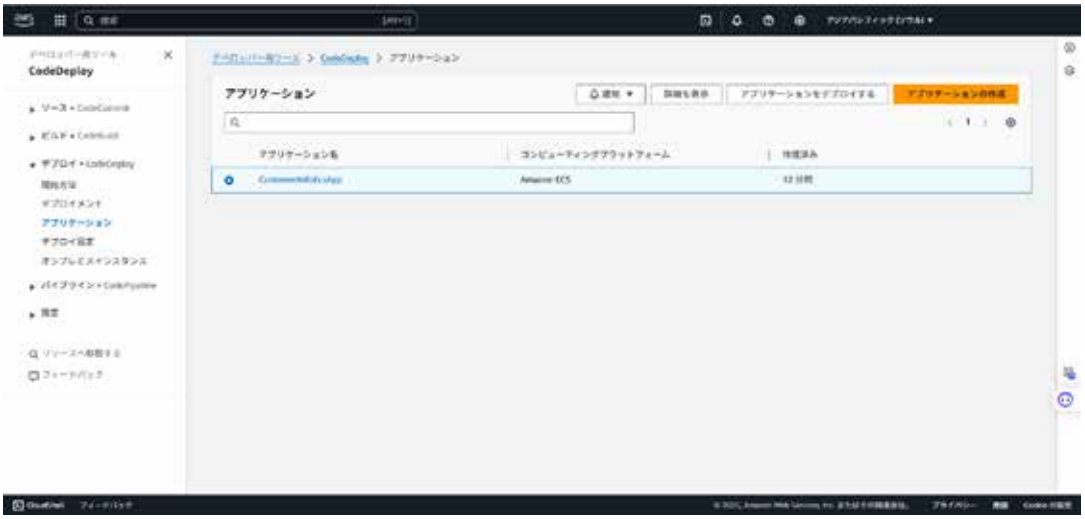
- SG（AlbSg）
  - VPCダッシュボード > セキュリティグループ > AlbSg
  - インバウンドルールにtcp:9001が追加されていることを確認

- 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　解答編

Step5：CodeDeploy　動作確認

- 続いて、DeployStackを実行し、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。



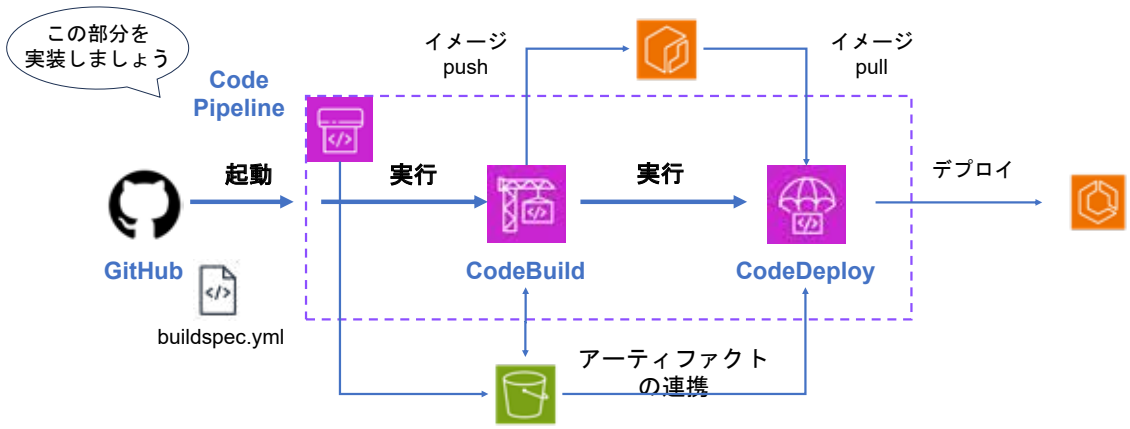
演習1-4 Step6　解答編

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　解答編

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　解答編

Step6：CodePipeline　CDKコード

- AWS CDKを利用して、**PipelineStackを作成するコード**を作成します。

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

```
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
・・・
```

演習1-4　CI/CDパイプライン構築　解答編

Step5：CodeDeploy　CDKコード

- **bin/ecs-demo.ts**から、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	—	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（codePiplepline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　解答編

Step6：CodePipeline　CDKコード

- bin/ecs-demo.tsのコードは以下の通りです。

```
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';
import { AlbStack } from '../lib/alb-stack';
import { EcrStack } from '../lib/ecr-stack';
import { RdsStack } from '../lib/rds-stack';
import { EcsStack } from '../lib/ecs-stack';
import { ConnectionStack } from '../lib/connection-stack';
import { IamStack } from '../lib/iam-stack';
import { BuildStack } from '../lib/build-stack';
import { DeployStack } from '../lib/deploy-stack';
import { PipelineStack } from '../lib/pipeline-stack';  //★追加

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 });
・・・省略・・・
```

```
・・・省略・・・

// CodeDeploy
・・・省略・・・

// CodePipeline ★追加
new PipelineStack(app, 'PipelineStack', {
  env,
  pipelineName              : 'CustomerInfoPipeline',
  codeBuildRoleArn                  : iam.codeBuildRole.roleArn,
  codeDeployRoleArn         : iam.codeDeployRole.roleArn,
  codePipelineRoleArn       : iam.codePipelineRole.roleArn,
  ecrRepoName                       : 'customer-info/app',
  gitHubConnectionArn       : conn.connectionArn,
  gitHubOwner               : '<xxx>',    // GitHubアカウント名を修正
  gitHubRepo                : 'customer-info',
  gitHubBranch              : 'main',
  ecsAppName                        : 'CustomerInfoEcsApp',
  ecsDeploymentGroupName    : 'CustomerInfoDG',
  ecsTaskExecutionRoleArn           : iam.ecsTaskExecutionRole.roleArn,
  ecsTaskRoleArn                    : iam.appTaskRole.roleArn,
  dbSecretArn                                 :
rds.appSecret.secretArn,
  dbHost                            : rds.dbHost,
});
```

演習1-4　CI/CDパイプライン構築　解答編

Step6：CodePipeline　CDKコード

- lib/pipeline-stack.tsの全体構成は以下の通りです。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インタフェース定義	他スタックから渡されるパラメータ
3	スタック初期化	PipelineStackを初期化、既存ロールのインポート
4	CodePipeline作成	CodePipelineのパイプライン作成
5	Sourceステージ	GitHubとの接続およびリソース確認
6	Buildステージ	Dockerビルドおよびイメージプッシュ
7	Deployステージ	Artifactの置換およびデプロイメントの起動
8	IAMロール権限付与	CodeDeployへのIAMロール権限付与
9	出力	設定値の出力

演習1-4　CI/CDパイプライン構築　解答編

Step6：CodePipeline　CDKコード

CodePipeline作成　解答例（1/6）

```
// 1. 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 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';

// 2. インタフェース定義
export interface PipelineStackProps extends cdk.StackProps {
  pipelineName: string;
  codeBuildRoleArn: string;                    // IAMロールの参照
  codeDeployRoleArn: string;                   // IAMロールの参照
  codePipelineRoleArn: string;                 // IAMロールの参照
  ecsTaskExecutionRoleArn: string;          // IAMロールの参照
  ecsTaskRoleArn?: string;                     // IAMロールの参照
  gitHubConnectionArn: string;                 // CodeConnection
  gitHubOwner: string;                         // GitHubアカウント
  gitHubRepo: string;                          // GitHubリポジトリ
  gitHubBranch: string;                        // GtiHubブランチ
  ecrRepoName: string;                      // ECRリポジトリ
  ecsAppName: string;                       // ECSアプリケーション
  ecsDeploymentGroupName: string;           // ECSデプロイメントグループ
  dbSecretArn?: string;                        // RDS APPシークレット
  dbHost?: string;                          // RDS DBホスト
}
```

ポイント

1. インポート

- CodePipelineだけでなく、CodeBuild、CodeDeployを呼び出すため、合わせてインポートを設定しています。

2. インタフェース定義

- CodePipelineで利用するパラメータを設定します。GitHub、CodeBuild、CodeDeploy、ECR、ECS、RDSなども関連してくるため、多くのパラメータを引っ張っています。

演習1-4　CI/CDパイプライン構築　解答編

Step6：CodePipeline　CDKコード

CodePipeline作成　解答例（2/6）

```
// 3. スタック初期化
export class PipelineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props: PipelineStackProps) {
    super(scope, id, props);
    // 既存ロールを 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（Privileged: ON 前提）
    const buildProject = codebuild.Project.fromProjectName(
      this, 'BuildProject', 'customer-info-app',
    );
    // 既存 CodeDeploy アプリケーション / デプロイメントグループ
    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');
```

ポイント

3. スタック初期化

- 既存IAMロールのインポートをしつつ、各種設定値を設定していきます。
- Codeシリーズは設定値などをArtifactで連携します。そのための設定を、sourceOutput、buildOutputで入れます。
- ロール設定時、mutable:falseとしてIAMロールの上書きを防止しています。

演習1-4　CI/CDパイプライン構築　解答編

Step6：CodePipeline　CDKコード

CodePipeline作成　解答例（3/6）

```
// 4. CodePipeline作成
const pipeline = new codepipeline.Pipeline(this, 'Pipeline', {
  pipelineName: props.pipelineName,
  role: codePipelineRole,
  stages: [

    // 5. Source: GitHub（CodeStar Connections）
    {
      stageName: 'Source',
      actions: [
        new cpactions.CodeStarConnectionsSourceAction({
          actionName: 'GitHub_Source',
          owner:  props.gitHubOwner,
          repo:   props.gitHubRepo,
          branch: props.gitHubBranch,
          connectionArn: props.gitHubConnectionArn,
          output: sourceOutput,
          triggerOnPush: true,
        }),
      ],
    },
```

ポイント

5. Sourceステージ

- CodeConnections経由でGitHubにアクセスするために各種設定値を入れて動作させます。
- triggerOnPush:trueにすることで、GitHubからwebhookでパイプラインが起動するのを許可しています。これにより、コードマージをトリガーにパイプラインが動きます。
- SourceフェーズのArtifactはsourceOutputとして、S3に保存されます。CodeBuildではこのsourceOutputを利用して動きます。

演習1-4　CI/CDパイプライン構築　解答編

Step6：CodePipeline　CDKコード

CodePipeline作成　解答例（4/6）

```
    // 6. Build: Dockerビルド、プッシュ
    {
      stageName: 'Build',
      actions: [
        new cpactions.CodeBuildAction({
          actionName: 'Docker_Build',
          project: buildProject,
          input: sourceOutput,
          outputs: [buildOutput],
          environmentVariables: {
            // CodeBuildに渡すパラメータ
            ECR_REPO:       { value: props.ecrRepoName },
            EXEC_ROLE_ARN:  { value: props.ecsTaskExecutionRoleArn },
            TASK_ROLE_ARN:  { value: props.ecsTaskRoleArn ??
props.ecsTaskExecutionRoleArn },
            DB_HOST:        { value: props.dbHost },
            DB_SECRET_ARN:  { value: props.dbSecretArn ?? '' },
          },
        }),
      ],
    },
```

ポイント

6. Buildステージ

- sourceOutputを入力として起動させています。またBuildフェーズの成果物をbuildOutputとして出力し、Deployフェーズに引き渡します。
- environmentVariables
CodeBuildの起動に必要なパラメータはここで渡します。今回アプリケーションでDB参照するため、DB_HOST、DB_SECRET_ARNを接続情報として合わせて引き渡しています。

演習1-4　CI/CDパイプライン構築　解答編

Step6：CodePipeline　CDKコード

CodePipeline作成　解答例（5/6）

```
      //7. Deploy: artifact置換、デプロイメント起動
      {
        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/ 配下）
            appSpecTemplateFile:        sourceOutput.atPath('deploy/ecs/appspec.yml'),
            taskDefinitionTemplateFile: buildOutput.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',
              },
            ],
          }),
        ],
      },
    ],
  });
```

ポイント

7. Deployステージ

- codeDeployを起動します。
- 以下ファイルを参照しながらタスク定義とアプリケーションを作成します。Sourceフェーズ、buildフェーズの成果物としてS3に格納されたappspec.yml、taskdef.jsonを利用するため、参照するOutputと格納先のパスを間違えないようにしてください。格納先はbuildspec.ymlで定義しています。
  - appSpecTemplateFile
  - taskDefinitionTemplateFile
- デプロイするコンテナイメージはbuildOutputのimageDetail.jsonのURIに書かれています。
- そのURIを引っ張り、IMAGE1_NAMEという変数でtaskdef.jsonに書き変えてデプロイします。変数名は合わせてください。

演習1-4　CI/CDパイプライン構築　解答編

Step6：CodePipeline　CDKコード

CodePipeline作成　解答例（6/6）

```
    // 8. IAMロール権限追加
    // CodeDeployのIAMロールへの権限追加
    const cdRole = iam.Role.fromRoleArn(
      this, 'ImportedCodeDeployRole', props.codeDeployRoleArn, { mutable: false }
    );
    // S3 読み取り許可（アーティファクト取得用）
    pipeline.artifactBucket.grantRead(cdRole);
    // KMS 暗号化されていた場合のみ Decrypt も付与
    if (pipeline.artifactBucket.encryptionKey) {
      pipeline.artifactBucket.encryptionKey.grantDecrypt(cdRole);
    }

    // 9. 出力
    new cdk.CfnOutput(this, 'PipelineName', { value: pipeline.pipelineName });
    new cdk.CfnOutput(this, 'ArtifactBucketName', { value: pipeline.artifactBucket.bucketName
});
  }
}
```

ポイント

8. IAMロール権限追加

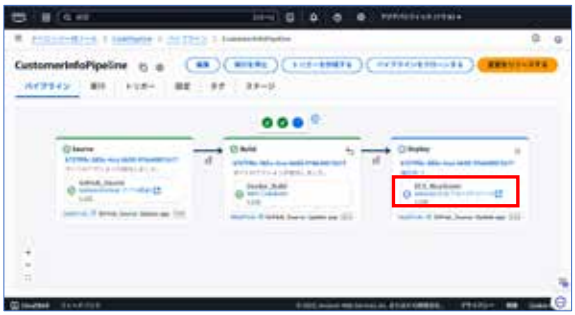
- 今回はPipelineStack内でArtifactBucketを新規作成している。そのため、このタイミングで、作成したS3バケットの参照権限を付与しています。
- S3バケットが暗号化されている場合を考慮して、grantDecryptを呼べるようにしています。

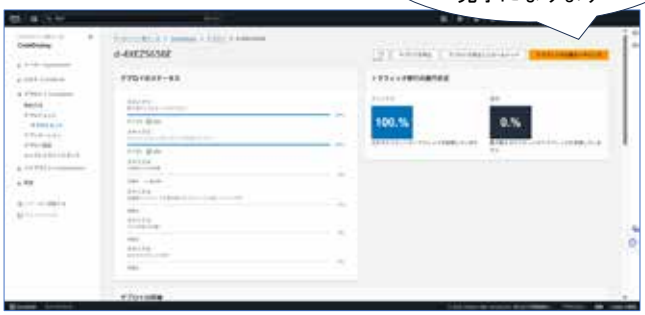
演習1-4　CI/CDパイプライン構築　解答編

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を手動承認します。

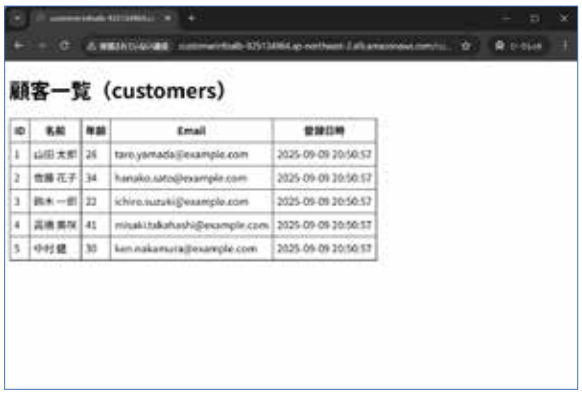




演習1-4　CI/CDパイプライン構築　解答編

Step6：CodePipeline　動作確認

- パイプライン完了を確認し、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でビルドしたイメージをB/G
デプロイ
（URIは<ALB_DNS>/customers）

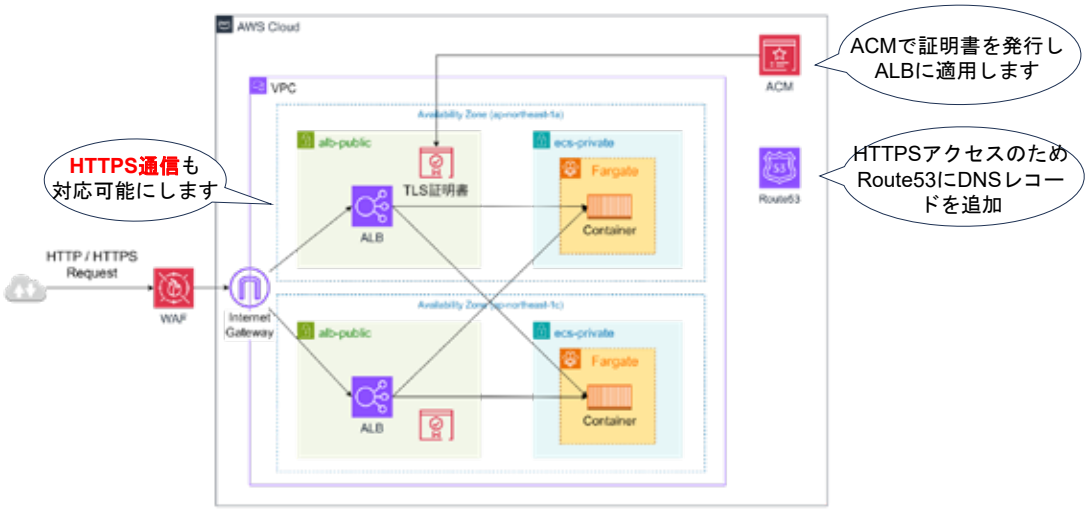
演習2 - 解答編

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　解答編

演習概要

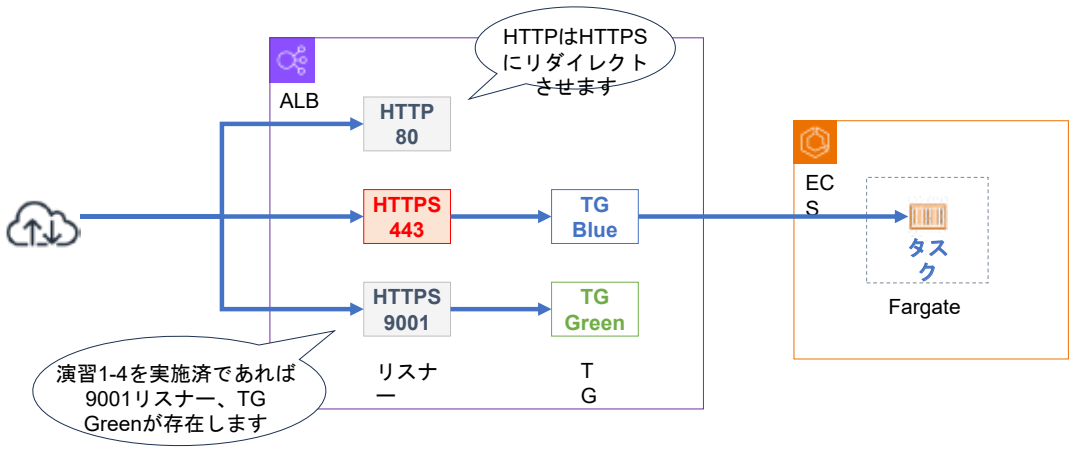
- AWS上でHTTP対応となっているALBおよびアプリケーションを、 AWS Certificate Manager(ACM)を用いて、HTTPS対応させ、セキュアなシステムを構築しましょう。



演習2　ロードバランサーのHTTPS移行　解答編

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　解答編

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　解答編

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　解答編

HTTPS移行　CDKコード

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

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

```
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　解答編

HTTPS移行　CDKコード

- 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	ー	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	ー	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　解答編

HTTPS移行　CDKコード

- bin/ecs-demo.tsのコードは以下の通りです。
  ACMで証明書発行後にALBを作成するため、AlbStackの前にAcmStackを定義します。

```
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';
import { AcmStack } from '../lib/acm-stack';  //★追加
import { AlbStack } from '../lib/alb-stack';
import { EcrStack } from '../lib/ecr-stack';
import { RdsStack } from '../lib/rds-stack';
import { EcsStack } from '../lib/ecs-stack';
import { ConnectionStack } from '../lib/connection-stack';
import { IamStack } from '../lib/iam-stack';
import { BuildStack } from '../lib/build-stack';
import { DeployStack } from '../lib/deploy-stack';
import { PipelineStack } from '../lib/pipeline-stack';

・・・省略・・・

// VPC Endpoints

・・・省略・・・
```

```
// ACM ★追加
const acm = new AcmStack(app, 'AcmStack', {
  env,
  domainName: '<公開したいサブドメイン>',  // 公開サブドメイン
  hostedZoneName: '<Route53に登録済みホストゾーン>',  // ホストゾーン
});

// ALB / WAF
const alb = new AlbStack(app, 'AlbStack', {
  env,
  vpc: net.vpc,
  albSg: net.albSg,
  // ★追加（3パラメータ）
  certificateArn: acm.certificateArn,  // 証明書ARN
  domainName: '<公開したいサブドメイン>',  // 公開サブドメイン
  hostedZoneName: '<Route53に登録済みホストゾーン>',  // ホストゾーン
});

・・・省略・・・
```

演習2　ロードバランサーのHTTPS移行　解答編

HTTPS移行　CDKコード

■ 続いて、lib/net-stack.tsのコードは以下の通りです。

net-stack.ts　解答例

```
・・・省略・・・

    // 6. セキュリティグループ間の通信ルール
    // ALB: InternetからHTTP/HTTPSを許可
    this.albSg.addIngressRule(ec2.Peer.anyIpv4(), ec2.Port.tcp(80), 'Allow HTTP
from Internet');
    this.albSg.addIngressRule(ec2.Peer.anyIpv4(), ec2.Port.tcp(443), 'Allow
HTTPS from Internet'); // ★追加　HTTPS移行用ポート
    this.albSg.addIngressRule(ec2.Peer.anyIpv4(), ec2.Port.tcp(9001), 'Allow test
listener (9001) from Internet');
    this.albSg.addEgressRule(this.ecsSg, ec2.Port.tcp(80), 'ALB-to-ECS');

・・・省略・・・
```

ポイント

6. セキュリティグループ間の通信ルール

- ALBのセキュリティグループにインバウンドルールとして、HTTPS:443を追加します。
- 演習ではHTTP:80を残していますが、セキュリティ的にHTTP接続は非推奨です。本番運用ではHTTP:80を蓋閉じすることをお勧めします。
- その他項目（VPC/サブネット/その他SG）の変更はありません。

演習2　ロードバランサーのHTTPS移行　解答編

HTTPS移行　CDKコード

■ lib/acm-stack.tsの全体構成は以下の通りです。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インタフェース定義	他スタックから渡されるパラメータ
3	スタック初期化	AcmStackを初期化、既存ロールのインポート
4	HostedZone情報取得	Route53に登録されたホストゾーン情報の取得
5	証明書作成	ACMでTLS証明書を作成し、Route53にCNAMEレコードを登録
9	出力	設定値の出力

演習2　ロードバランサーのHTTPS移行　解答編

HTTPS移行　CDKコード

ACM証明書作成　解答例（1/2）

```
// 1. 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';

// 2. インタフェース定義
export interface AcmStackProps extends cdk.StackProps {
  domainName: string;                  // 例: app.test-jp.com
  hostedZoneName: string;    // 例: test-jp.com
}

// 3. 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);
```

ポイント

1. インポート
- ACMとRoute53を操作するため、acmとroute53をインポートしています。
- ACMの証明書はリージョンに依存するため、

2. インタフェース定義
- Route53に問い合わせに行くためのホストゾーン名と生成したいドメイン名を外部プロパティとして読み込みます。

演習2　ロードバランサーのHTTPS移行　解答編

HTTPS移行　CDKコード

ACM証明書作成　解答例（2/2）

```
    // 4. HostedZone情報取得
    const zone = route53.HostedZone.fromLookup(this, 'HostedZone', {
      domainName: props.hostedZoneName,
    });

    // 5. 証明書作成
    const cert = new acm.Certificate(this, 'AlbCert', {
      domainName: props.domainName,
      validation: acm.CertificateValidation.fromDns(zone),
    });

    this.certificateArn = cert.certificateArn;

    // 6. 出力
    new cdk.CfnOutput(this, 'CertificateArn', {
      value: cert.certificateArn,
      exportName: `${cdk.Stack.of(this).stackName}-CertificateArn`,
    });
  }
}
```

ポイント

4. HostedZone情報取得
- Route53からホストゾーンの情報を取得します。
- fromLookupを使用し、hostedZoneNameで、Route53の既存ホストゾーンを検索し、ゾーンIDやゾーン情報を取得します。

5. 証明書作成
- domainNameに基づくTLS証明書を発行して、Route53にCNAMEレコードを作成します。
- CertificateValidation.fromDns(zone)により、CDKが自動でRoute53にCNAMEレコードを作成します。これにより、ACMでDNSを自動検証するようになります。
- ACMの証明書はリージョンに依存するので注意が必要です。スタック全体のenvで指定したリージョンに作成されます。

演習2　ロードバランサーのHTTPS移行　解答編

HTTPS移行　CDKコード

■ lib/alb-stack.tsの全体構成は以下の通りです。次ページからコードの修正箇所を説明します。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インターフェース定義	他スタックから渡されるパラメータ
3	公開プロパティ	他スタックが参照するプロパティの公開
4	スタック初期化	VpceStackを初期化、サブネットの選択
5	ALB作成	ALBを構築
6	ターゲットグループ作成	ターゲットグループを設定（ターゲットは**空**）
7	HTTPリスナー作成	HTTPリスナー(80)を作成　80→443にリダイレクト　★修正
8	HTTPSリスナー作成	HTTPSリスナー(443、9001)を作成　★追加
9	Route53 レコード作成	Route53のALB向けのAレコードを作成し、ドメインを解決　★追加
10	WAFルール作成（BadBotブロック）	WAFのルールAを作成　User-Agentに”BadBot”を含むアクセスを遮断
11	WAFルール作成（マネージドルール）	WAFのルールBを作成　AWSマネージド共通ルールの適用
12	WebACL作成	WAF WebACLを作成
13	ALBとWebACL関連付け	WebACLをALBにアタッチ

演習2　ロードバランサーのHTTPS移行　解答編

HTTPS移行　CDKコード

ALB作成　解答例（1/4）

```
// 1. 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';
import * as route53 from ‘aws-cdk-lib/aws-route53’;           // ★追加
import * as targets from 'aws-cdk-lib/aws-route53-targets';  // ★追加

// 2. インタフェース定義
export interface AlbStackProps extends StackProps {
  vpc           : ec2.IVpc;
  albSg         : ec2.ISecurityGroup;
  albSubnets?   : ec2.SubnetSelection;
  certificateArn : string;          // TLS証明書ARN   ★追加
  hostedZoneName : string;    // ホストゾーン名   例: example.com   ★追加
  domainName     : string;    // ドメイン名   例: app.example.com   ★追加
}

・・・省略（3～6）・・・
```

ポイント

1. インポート

- Route53の参照およびRoute53のDNS登録のため、route53、targetsをインポートに追加しています。

2. インタフェース定義

- ALBリスナーに適用する証明書ARN、そして、Route53の検索およびDNSレコード登録用を外部インタフェースとして読み込むよう、設定を追加しました。

演習2　ロードバランサーのHTTPS移行　解答編

HTTPS移行　CDKコード

ALB作成　解答例（2/4）

```
// 7. HTTPリスナー作成
// HTTP用リスナー（HTTP(80)はHTTPS(443)へ301リダイレクト）　★追加
alb.addListener('HttpListenerRedirect', {
  port: 80,
  defaultAction: elbv2.ListenerAction.redirect({
    protocol: 'HTTPS',
    port: '443',
    permanent: true, // 301を返却
  }),
});
```

ポイント

7. HTTPリスナー作成

- HTTP:80をリッスンし、HTTPリクエストをターゲットグループに流さず、HTTPS:443にリダイレクトするように変更しています。
- permanent: true を設定することで301を返却し、HTTPを強制的にHTTPSにリダイレクトするようになります。

演習2　ロードバランサーのHTTPS移行　解答編

HTTPS移行　CDKコード

ALB作成　解答例（3/4）

```
// 8. HTTPSリスナー作成（Blue/Green 用）
// 本番リスナー (443) : 初期はBlueを本番に適用（★追加）
this.listenerProd = alb.addListener('HttpsListenerProd', {
  port: 443,
  protocol: elbv2.ApplicationProtocol.HTTPS,
  certificates: [{ certificateArn: props.certificateArn }],
  sslPolicy: elbv2.SslPolicy.RECOMMENDED_TLS,
  defaultTargetGroups: [this.tgBlue],
});

// テストリスナー (9001) : 初期はGreenをテストに適用（★ポートは変更なし
)
this.listenerTest = alb.addListener('HttpsListenerTest', {
  port: 9001,
  protocol: elbv2.ApplicationProtocol.HTTPS,
  certificates: [{ certificateArn: props.certificateArn }],
  sslPolicy: elbv2.SslPolicy.RECOMMENDED_TLS,
  defaultTargetGroups: [this.tgGreen],
});
```

ポイント

8. HTTPSリスナー作成

- HTTP: 80/9001で本番/テストリスナーとしていましたが、HTTPS: 443/9001に変更しました。
- certificatesでACMの証明書を指定して適用します。先程作成した証明書をARNで紐づけます。
- sslPolicyはどの暗号化方式を許可するかを決めることができます。RECOMENTDED_TLSを選択することで、TLS1.2以上の強度が高めの方式だけを許可し、セキュリティが高まります。
- 本番リスナーとテストリスナーの設定はほぼ同じで、プロトコルと紐づくターゲットグループのみ異なります。

演習2　ロードバランサーのHTTPS移行　解答編

HTTPS移行　CDKコード

ALB作成　解答例（4/4）

```
  // 9. Route53 レコード作成　★追加
  const zone = route53.HostedZone.fromLookup(this, 'AlbZone', {
    domainName: props.hostedZoneName,
    privateZone: false,  // パブリックゾーンのみ検索
  });

  // 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),
  });

・・・省略（10～14）・・・
```

ポイント

9. Route53レコード作成

- Route53のホストゾーンにALB向けのAレコードを作成し、domainNameで指定したアドレスでALBのDNS名を解決できるようになります。
- fromLookupでホストゾーン情報を取得します。これはAcmStackと同じ挙動です。
- route53.ARecordで、Aレコードを作成しています。
  ホストゾーンにAレコードを登録するにあたり、サブドメイン部分（recordName）を抽出して、設定しています。直接recordNameを定義しても良いです。
- 今回はIPv4通信のため、Aレコードのみ登録してします。IPv6通信の場合は、AAAAレコードが必要になります。必要なレコードタイプを登録してください。

演習2　ロードバランサーのHTTPS移行　解答編

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になります。
  - 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3 - 解答編

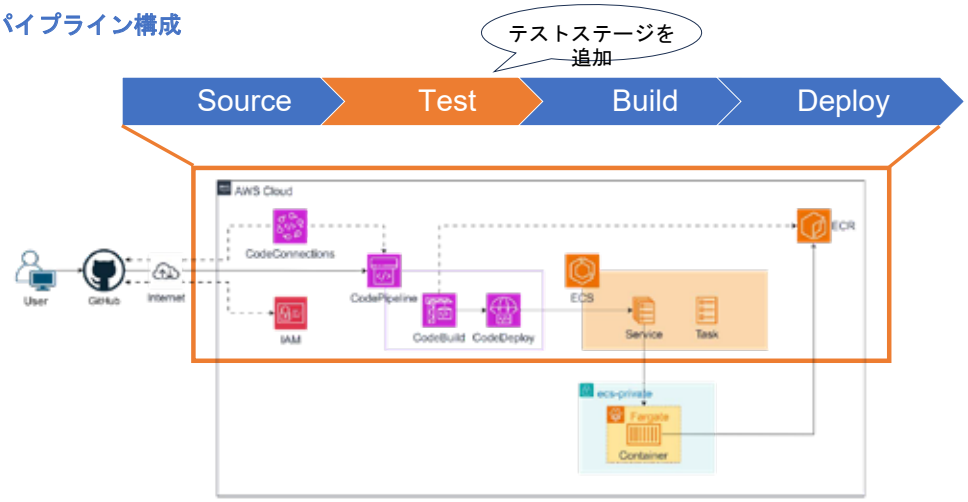
CI/CDパイプラインのユニットテスト追加

演習3　CI/CDパイプラインのユニットテスト追加　解答編

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　解答編

演習概要

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

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。

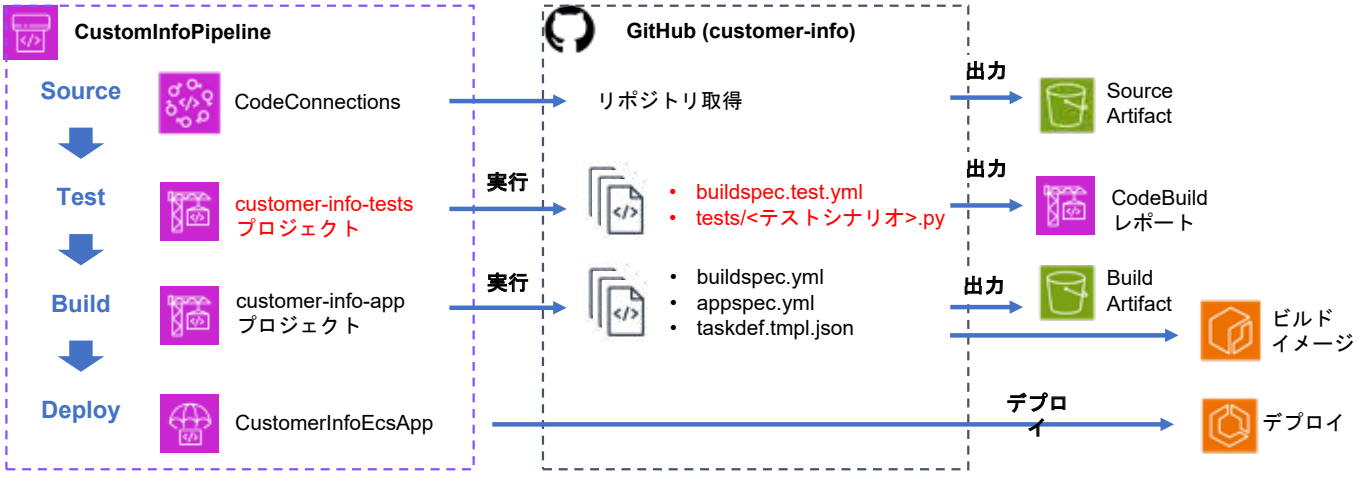
CodeDeploy
CD部分。ビルドしたコンテナイメージをコンテナ基盤ECSにデプロイします。

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　解答編

CodePipeline　設計

■ 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　解答編

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
  - 追加したTest用プロジェクトをCodePipelineから呼び出してユニットテストを実行する（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　解答編

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　解答編

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　解答編

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　解答編

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。
- アプリケーションおよび試験関連のライブラリインストール
- 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　解答編

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　解答編

パイプラインユニットテスト追加　GitHubコード

■ まず、buildspec.test.ymlのコードは以下の通りです。

buildspec.test.yml作成　解答例

```
version: 0.2
phases:
  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:${PYTHONPAT
H}"
      // 認証情報ロード（ダミー値）
      - export
DB_CREDENTIALS_JSON='{"username":"test_user","password":"test_pass","port":3306}'
  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:
  pytest_reports:
    files: [test-results/junit.xml]  // 出力ファイル
    file-format: JUNITXML
```

ポイント

pre_buildフェーズ
- ユニットテストの準備をします。requirement.txtのパッケージおよびpytestをインストールします。
- PYTHONPATHで、 app.pyを格納している/app配下も含め、pythonが検索するパスを指定しています。
- DB_CREDENTIALS_JSONでダミーのDB接続情報を作成し、アプリケーションで利用します。

Buildフェーズ
- pytestを実行し、JUnit形式でレポートを生成しています。JUnitは標準的なレポート形式です。

Reports
- pytestで出力したファイルを指定して、CodeBuildのレポート機能に登録します。

演習3　CI/CDパイプラインのユニットテスト追加　解答編

パイプラインユニットテスト追加　GitHubコード

■ 続いて、試験シナリオ「/tests/test_health.py」のコードは以下の通りです。

/tests/test_health.py　解答例（1/2）

```
import os, sys, types

# app/ を import パスに追加
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):
        # テスト用のダミー認証情報を返す
        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'] = types.SimpleNamespace(client=_fake_boto3_client)
```

ポイント

- sys.path.insert(0, os.path.abspath(“app”))
  app/ディレクトリをモジュール検索パスの先頭に追加しています。これにより、後のimport appを解決しています。
- boto3.client(“secretsmanager”)
  本番のアプリケーションでは、AWSのRDS認証情報を取得するところですが、ユニットテストのため、_FakeSecretsClinetで、ダミーの認証情報を返却するように設定しています。
- sys.modules[‘boto3’]
  ダミーのboto3オブジェクトを生成するように設定し、AWSの操作ではなく、ダミーに対して操作します。

演習3　CI/CDパイプラインのユニットテスト追加　解答編

パイプラインユニットテスト追加　GitHubコード

/tests/test_health.py　解答例（2/2）

```
# アプリのimport
import app as target  # app/app.py の読込（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
```

ポイント

- import app as target
  アプリケーション(app/app.py)を読み込みます。
- res = client.get(“/”)
  Flaskのテストクライアントから”/”エンドポイントにアクセスします。ここでHTTPステータスが200となれば、アプリが正常起動したと判断できます。

演習3　CI/CDパイプラインのユニットテスト追加　解答編

パイプラインユニットテスト追加　GitHubコード

■ 2つ目の試験シナリオ「/tests/test_db_credentials.py 」のコードは以下の通りです。

/tests/test_db_credentials.py　解答例（1/2）

```
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()

# 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'] = types.SimpleNamespace(client=_fake_boto3_client)
```

ポイント

- 前半は、/test/test_health.pyと同じ実装になります。ポイントも同様のため、説明は割愛します。

演習3　CI/CDパイプラインのユニットテスト追加　解答編

パイプラインユニットテスト追加　GitHubコード

/tests/test_db_credentials.py　解答例（2/2）

```
# 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")

# テスト
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
```

ポイント

- Appのimportについては、モジュールを動的にロード可能にしています。
- target.load_db_credentials()
  上記関数の動作確認をします。
  ダミーの認証情報を取得し、想定した値と一致するかを検証しています。

- 今回は2つのシナリオだけですが、その他ユニットテストとして、以下の様なシナリオを追加することも可能です。
  - /customersが想定通りに表示されるか
  - 存在しないルートにアクセスし、404エラーとなるか
  - DB接続処理が正しく動作するか

演習3　CI/CDパイプラインのユニットテスト追加　解答編

パイプラインユニットテスト追加　CDKコード

■ AWS CDKを利用して、**CI/CDパイプラインにユニットテストを追加するためのコード**を作成します。

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

```
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　解答編

パイプラインユニットテスト追加　CDKコード

■ まずは、lib/iam-stack.tsの修正箇所は以下の通りです。

net-stack.ts　解答例

```
・・・省略・・・
  // 4. CodeBuildロール作成
  this.codeBuildRole = new iam.Role(this, 'CodeBuildRole', {
    assumedBy: new iam.ServicePrincipal('codebuild.amazonaws.com'),
    description: 'Allows CodeBuild to push images to ECR and write logs',
  });
    ・・・省略・・・
  // Artifact S3/KMS
    ・・・省略・・・

  // CodeBuild Reports（pytestのJUnit等をレポートとして登録）　★追加
  this.codeBuildRole.addToPolicy(new iam.PolicyStatement({
    actions: [
      'codebuild:*Report*',
      'codebuild:BatchPut*',
    ],
    resources: ['*'],
  }));

・・・省略・・・
```

ポイント

4. CodeBuildロール作成

- CodeBuildのレポート作成用のポリシーを付与するのみでコード修正は完了です。これで、CodeBuildのレポート機能にJUnit XMLを登録可能になります。
- CodeBuildの権限ですが、「*Report*」「BatchPut*」でまとめて付与しています。細かく権限を制御する場合は、codeBuild:CreateReport / UpdateCreateGroupなど 1個ずつ権限を記載して、権限を最小化してください。

演習3　CI/CDパイプラインのユニットテスト追加　解答編

パイプラインユニットテスト追加　CDKコード

■ 続いて、lib/build-stack.tsを修正します。全体構成が以下のように変更となります。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インタフェース定義	他スタックから渡されるパラメータ
3	スタック初期化	BuildStackを初期化
4	IAMロール形式確認	IAMロールARNの形式確認
5	ECRリポジトリ設定	ECRリポジトリを設定
6	CodeBuildロールのインポート	CodeBuildロールをインポート
7	CodeBuildプロジェクト作成	CodeBuildのビルドプロジェクトを作成する
8	Test用CodeBuildプロジェクト作成	テスト用にCodeBuildのビルドプロジェクトを作成する（追加）
9	出力	設定値の出力

演習3　CI/CDパイプラインのユニットテスト追加　解答編

パイプラインユニットテスト追加　CDKコード

■ lib/build-stack.tsの修正箇所は以下の通りです。

build-stack.ts　解答例（1/3）

```
・・・省略・・・
// 2. 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
  testBuildSpecFile?: string;        // buildspec.test.ymlのパス ★追加
}

// 3. スタック初期化
export class BuildStack extends cdk.Stack {
  public readonly project: codebuild.IProject;
  public readonly projectName: string;
  public readonly testProject: codebuild.IProject; // ★テスト用CodeBuild
  public readonly testProjectName: string;        // ★テスト用CodeBuild名

  constructor(scope: Construct, id: string, props: BuildStackProps) {

・・・省略・・・
```

ポイント

2. インタフェース定義

- test用のbuildspec.test.ymlのパスを外部から渡せるように定義を追加しています。本解答例では、パラメータがない場合デフォルトで参照するパスをコード内に記載しているため、ecs-demo.tsからBuildStackに当該パラメータを渡していません。

3. スタック初期化

- test用プロジェクトとプロジェクト名を外部公開するプロパティとして追加しています。

演習3　CI/CDパイプラインのユニットテスト追加　解答編

パイプラインユニットテスト追加　CDKコード

build-stack.ts　解答例（2/3）

```
・・・省略・・・
  // 8. test用CodeBuildプロジェクトの作成（CodePipelineから起動）　★追加
  const testProject = new codebuild.PipelineProject(this, 'UnitTestProject', {
    projectName: 'customer-info-tests',
    role,
    environment: {
      buildImage: codebuild.LinuxBuildImage.STANDARD_7_0,
      privileged: false,  // pytest だけなので特権不要
    },

    // リポジトリ直下の buildspec.test.yml を利用
    buildSpec: codebuild.BuildSpec.fromSourceFilename(
      props.testBuildSpecFile ?? 'buildspec.test.yml',
    ),
  });

  this.project = project;
  this.projectName = project.projectName;
  this.testProject = testProject;                                    // ★追加
  this.testProjectName = testProject.projectName;  // ★追加
```

ポイント

8. Test用CodeBuildプロジェクト作成

- 7. CodeBuildプロジェクト作成に近しいコードでビルドプロジェクトを作成します。
- このプロジェクトは、pytestのみ実行します。ECR pushなどに必要な特権モードは不要なため、privileged:false としています。
  特権モードが不要な場合は、基本falseとして特権モードをつけないことが大切です。
- ビルド設計書は「buildspec.test.yml」を指定します。

演習3　CI/CDパイプラインのユニットテスト追加　解答編

パイプラインユニットテスト追加　CDKコード

build-stack.ts　解答例（3/3）

```
  // 9. 出力
  new cdk.CfnOutput(this, 'CodeBuildProjectName',       {
    value: project.projectName      });
  new cdk.CfnOutput(this, 'CodeBuildProjectArn',        {
    value: project.projectArn       });
  new cdk.CfnOutput(this, 'TestCodeBuildProjectName', {
    value: testProject.projectName }); // ★追加
  new cdk.CfnOutput(this, 'TestCodeBuildProjectArn',    {
      value: testProject.projectArn   }); // ★追加
  new cdk.CfnOutput(this, 'EcrRepoName',                {
      value: repoName                 });
  new cdk.CfnOutput(this, 'EcrRegistry',                {
      value: ecrRegistry              });
 }
}
```

ポイント

9. 出力

- CfnOutputで、テスト用CodeBuildプロジェクトとARNを出力します。

演習3　CI/CDパイプラインのユニットテスト追加　解答編

パイプラインユニットテスト追加　CDKコード

■ 最後に、lib/pipeline-stack.tsを修正します。全体構成が以下のように変更となります。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インタフェース定義	他スタックから渡されるパラメータ
3	スタック初期化	PipelineStackを初期化、既存ロールのインポート
4	CodePipeline作成	CodePipelineのパイプライン作成
5	Sourceステージ	GitHubとの接続およびリソース確認
6	Testステージ	アプリケーションコードのユニットテスト（追加）
7	Buildステージ	Dockerビルドおよびイメージプッシュ
8	Deployステージ	Artifactの置換およびデプロイメントの起動
9	IAMロール権限付与	CodeDeployへのIAMロール権限付与
10	出力	設定値の出力

演習3　CI/CDパイプラインのユニットテスト追加　解答編

パイプラインユニットテスト追加　CDKコード

■ 最後に、lib/pipeline-stack.tsの修正箇所は以下の通りです。

pipeline-stack.ts　解答例（1/2）

```
・・・省略・・・
// 3. スタック初期化
export class PipelineStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props: PipelineStackProps) {
    super(scope, id, props);

    // 既存ロールを import
     ・・・省略・・・

    // 新規 CodeBuild プロジェクト test 用 （pytest用・Docker不要） ★追加
    const testBuildProject = codebuild.Project.fromProjectName(
      this, 'TestBuildProject', 'customer-info-tests',
    );

    // 既存 CodeBuild プロジェクト（Privileged: ON 前提）
    const buildProject = codebuild.Project.fromProjectName(
      this, 'BuildProject', 'customer-info-app',
    );
・・・省略・・・
```

ポイント

3. スタック初期化

- スタック初期化時に、CodeBuildプロジェクトを設定します。
- 既にビルド用のビルドプロジェクトが存在する想定ですので、同様にテスト用ビルドプロジェクトを設定してください。

演習3　CI/CDパイプラインのユニットテスト追加　解答編

パイプラインユニットテスト追加　CDKコード

pipeline-stack.ts　解答例（2/2）

```
・・・省略・・・
  // 4. CodePipeline作成
  const pipeline = new codepipeline.Pipeline(this, 'Pipeline', {
    pipelineName: props.pipelineName,
    role: codePipelineRole,
    stages: [
      // 5. Source: GitHub（CodeStar Connections）

      // 6. Test: pytest実行して品質担保　★追加
      {
        stageName: 'Test',
        actions: [
          new cpactions.CodeBuildAction({
            actionName: 'Pytest',
            project: testBuildProject,
            input: sourceOutput,  // tests/ と buildspec.test.yml を参照
          }),
        ],
      },

      // 7. Build: Dockerビルド、プッシュ
      ・・・省略・・・
```

ポイント

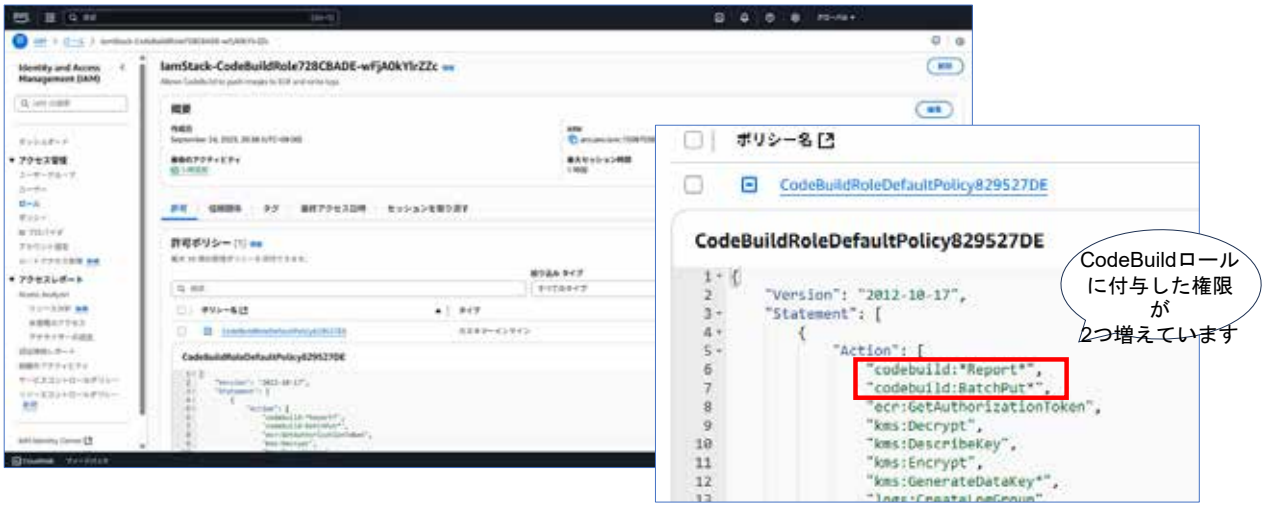
6. Test：pytestを実行

- SourceとBuildステージの間に、Testステージを追加します。stagesの記載順序を守れば、Source→Test→Build→Deployの4ステージでパイプラインが構築されます。
- testBuildProjectでテスト用ビルドプロジェクトを指定することで、CI/CDパイプラインからCodeBuildが呼び出され、ユニットテストを実施します。
- Test失敗時は、デフォルトでパイプラインが停止するようになっています。

演習3　CI/CDパイプラインのユニットテスト追加　解答編

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　解答編

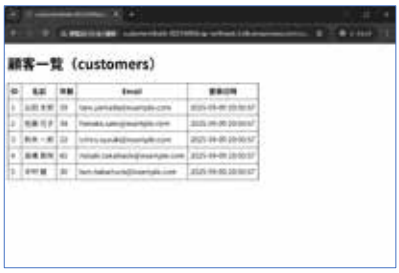
パイプラインユニットテスト追加　動作確認

- 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　解答編

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。Customer　InfoPipeline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　解答編

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結果にてエラーが0件であれば、正常にテストが完了していることを確認できます

Testの詳細も
確認可能です

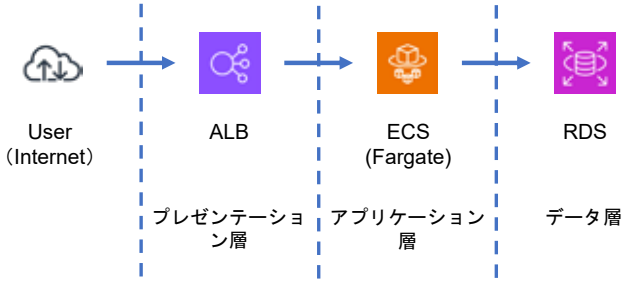
演習4 - 解答編

総合演習

演習4　総合演習　解答編

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　解答編

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　解答編

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　解答編

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。

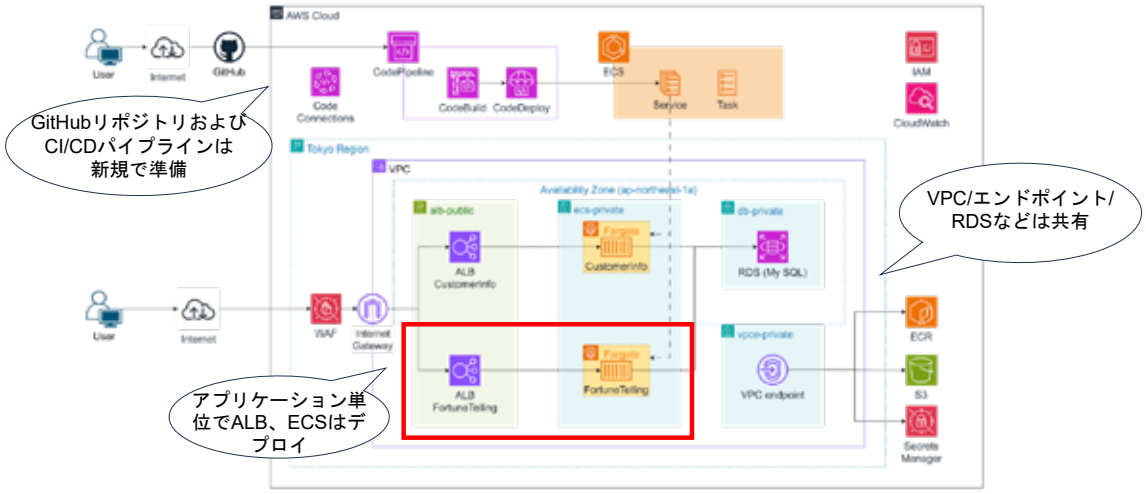
CodeDeploy
CD部分。ビルドしたコンテナイメージをコンテナ基盤ECSにデプロイします。

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　解答編

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　解答編

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。
  既存のCDKにスタック定義を追加・修正し、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、全ての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　解答編

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　解答編

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を実装します
- 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　解答編

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　解答編

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　解答編

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、各スタックを共有するか分割するか、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	共有 **（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	NetStack	共有 **（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。
	VpceStack	共有 （変更なし）	ECR/Logs/Secrets/SSM などは共通で使用。
	AcmStack	共有 （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。
	AlbStack Alb2Stack	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	EcrStack	共有 **（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。
	EcsStack Ecs2Stack	共有 **（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　解答編

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	RdsStack	共有 **（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	IamStack	共有 **（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	ConnectionStack	共有 （変更なし）	GitHub CodeConnections は複数パイプラインで共有可。
	BuildStack Build2Stack	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。
	DeployStack Deploy2Stack	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。
	PipelineStack Pipeline2Stack	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はxxx2Stackという名称にしていますが、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 - 解答編

総合演習　ーネットワーク構築ー

演習4　総合演習　解答編

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　解答編

ネットワーク構築　SG

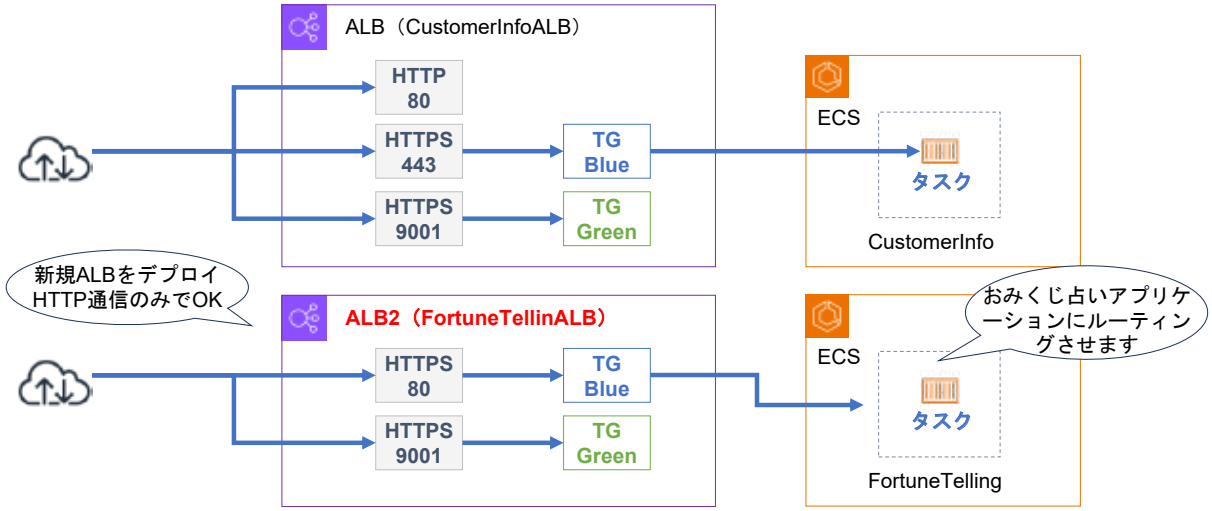
- SGの設計は以下の通りです。おみくじ占いアプリケーション用に新規SGを準備しましょう。
  既存のAlbSgを流用することも可能ですが、通信要件が異なるため、Alb2Sgを新たに用意します。

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　解答編

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　解答編

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　解答編

ネットワーク構築　CDKコード

■ AWS CDKを利用して、おみくじ占いアプリケーションに対応したネットワーク基盤を構築します。
演習4も段階的にスタックを追加・修正していきましょう。

※赤字のスタックおよびbin/ecs-demo.tsを修正します。

スタック名	用途	修正内容
App	CDK全体を表すクラス名	
NetStack	VPC、サブネット、SGを定義	FortuneTellingAlb用のSGの定義を追加
VpceStack	VPCエンドポイントを定義	
AcmStack	TLS証明書を定義	
AlbStack、**Alb2Stack**	ALB、WAFを定義	Alb2Stackを新規追加し、FortuneTellingAlbを生成
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4　総合演習　解答編

ネットワーク構築　CDKコード

- bin/ecs-demo.tsにて、Alb2Stackに渡すパラメータは以下の通りです。
  2つめのALBはHTTP通信のみのため、ACMのTLS証明書等の引き渡しは不要です。

Alb2Stack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
VPC	vpc	net.vpc	配置先VPC/サブネットを参照するため
SG	albSg	net.alb2Sg	alb-publicサブネットに付与するSG 2つめのALB用のSG、**alb2Sg**を渡します

演習4　総合演習　解答編

ネットワーク構築　CDKコード

- bin/ecs-demo.tsのコードは以下の通りです。
  AlbStackの次に、Alb2Stackのセクションを追加しています。

```
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';
import { AcmStack } from '../lib/acm-stack';
import { AlbStack } from '../lib/alb-stack';
import { Alb2Stack } from '../lib/alb2-stack';  //★追加
import { EcrStack } from '../lib/ecr-stack';
import { RdsStack } from '../lib/rds-stack';
import { EcsStack } from '../lib/ecs-stack';
import { ConnectionStack } from '../lib/connection-stack';
import { IamStack } from '../lib/iam-stack';
import { BuildStack } from '../lib/build-stack';
import { DeployStack } from '../lib/deploy-stack';
import { PipelineStack } from '../lib/pipeline-stack';

・・・省略・・・
```

```
// ALB / WAF
const alb = new AlbStack(app, 'AlbStack', {
  env,
  vpc: net.vpc,
  albSg: net.albSg,
  certificateArn: acm.certificateArn,  // 証明書ARN
  domainName: '<公開したいサブドメイン>',  // 公開サブドメイン
  hostedZoneName: '<Route53に登録済みホストゾーン>',  // ホストゾーン
});

// ALB / WAF for FortuneTelling　★追加セクション
const alb = new Alb2Stack(app, 'Alb2Stack', {
  env,
  vpc: net.vpc,
  albSg: net.alb2Sg,  // 2つめのALB用SG
});

・・・省略・・・
```

演習4　総合演習　解答編

ネットワーク構築　CDKコード

■ 続いて、lib/net-stack.tsに新規セキュリティグループを追加します。修正箇所は以下の通りです。

net-stack.ts　解答例（1/2）

```
・・・省略・・・
// 2. 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 alb2Sg: ec2.SecurityGroup;  // ★追加

・・・省略・・・

    // 5. セキュリティグループ
    // AlbSg

    // Alb2Sg  ★ブロック追加
    this.alb2Sg = new ec2.SecurityGroup(this, 'Alb2Sg', {
      vpc: this.vpc,
      description: 'Security Group for ALB2',
      allowAllOutbound: false,
    });

・・・省略・・・
```

ポイント

2. クラス宣言
- 新規ALBのセキュリティグループの追加に伴い、alb2Sgを外部公開します。
- その他パラメータは変更せず、外部公開のままとします。

5. セキュリティグループ
- 新しいセキュリティグループAlb2Sgを追加します。
  定義内容は既存のAlbSgと変わりません。
- 既存AlbSgブロック含め、その他の既存SG定義は残します。

演習4　総合演習　解答編

ネットワーク構築　CDKコード

■ 続いて、lib/net-stack.tsに新規セキュリティグループを追加します。修正箇所は以下の通りです。

net-stack.ts　解答例（2/2）

```
・・・省略・・・
    // 6. セキュリティグループ間の通信ルール
    // ALB2: InternetからHTTPを許可  ★追加
    this.alb2Sg.addIngressRule(ec2.Peer.anyIpv4(), ec2.Port.tcp(80), 'Allow HTTP
from Internet');  // 本番リスナー用ポート
    this.alb2Sg.addIngressRule(ec2.Peer.anyIpv4(), ec2.Port.tcp(9001), 'Allow
test listener (9001) from Internet');  // テストリスナー用ポート
    this.alb2Sg.addEgressRule(this.ecsSg, ec2.Port.tcp(80), 'ALB2-to-ECS');
    // ECS: ALBからHTTPアクセスを許可
    this.ecsSg.addIngressRule(this.albSg, ec2.Port.tcp(80), 'ALB-to-ECS');
    this.ecsSg.addIngressRule(this.alb2Sg, ec2.Port.tcp(80), 'ALB2-to-ECS');  // ★
追加

・・・省略・・・
// 7. 出力
    new CfnOutput(this, 'Alb2SgId',   { value: this.alb2Sg.securityGroupId });  // ★追加
  }
}
```

ポイント

6. セキュリティグループ間の通信ルール
- 新しいセキュリティグループAlb2Sgにインバウンドルールを追加します。
- addIngressRuleでインバウンドルールを設定します。HTTP:80とHTTP:9001の2つのポートを設定してください。
- EcsSgブロックについて、新しいALBからECSへのインバウンド通信（HTTP:80）を追加します。
- その他の既存SGの通信ルール定義は変更せずに残します。

7. 出力
- CfnOutputを利用し、Alb2Sgを追加で出力させます。
- その他出力項目はそのまま残置します。

演習4　総合演習　解答編

ネットワーク構築　CDKコード

- 次に、lib/alb2-stack.tsです。演習1-4 Step5のalb-stack.tsをコピーして新規作成してください。
  コードの構成や内容はほぼ同じです、修正例のように各種変数や名称を修正してください。

alb2-stack.ts　修正例

修正項目	修正前	修正後
スタックインタフェース	AlbStackProps	Alb2StackProps
スタッククラス	AlbStack	Alb2Stack
変数（ロードバランサー）	const alb	const alb2
ロードバランサー名	CustomerInfoAlb	FortuneTellingAlb
変数（WAFルール1）	const batBotRule	const batBotRule2
WAFルール1　名称	BlockBadBotUA	BlockBadBotUA2
変数（WAFルール2）	const awsManagedCommon	const awsManagedCommon2
WAFルール2　名称	AWSManagedCommonRuleSet	AWSManagedCommonRuleSet2
変数（WebACL)	const webAcl	const webAcl2
WebACL名	AlbWebAcl	Alb2WebAcl
WAF適用ルール	rules: [badBotRule, awsManagedCommon],	rules: [badBotRule2, awsManagedCommon2],

演習4　総合演習　解答編

HTTPS移行　CDKコード

- lib/alb2-stack.tsの全体構成は以下の通りです。
  alb-stackでもWAFルールが定義され、同じルールが2重に作成されるため、本運用ではALBとWAFのスタックを分けると良いでしょう。

コード構成

#	構成	役割
1	インポート	CDKやモジュールなどのインポート
2	インターフェース定義	他スタックから渡されるパラメータ
3	公開プロパティ	他スタックが参照するプロパティの公開
4	スタック初期化	VpceStackを初期化、サブネットの選択
5	ALB作成	ALBを構築
6	ターゲットグループ作成	ターゲットグループを設定（ターゲットは**空**）
7	HTTPリスナー作成	HTTPリスナー(80、9001)を作成
8	WAFルール作成（BadBotブロック）	WAFのルールAを作成　User-Agentに”BadBot”を含むアクセスを遮断
9	WAFルール作成（マネージドルール）	WAFのルールBを作成　AWSマネージド共通ルールの適用
10	WebACL作成	WAF WebACLを作成
11	ALBとWebACL関連付け	WebACLをALBにアタッチ
12	出力	設定値の出力

演習4　総合演習　解答編

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

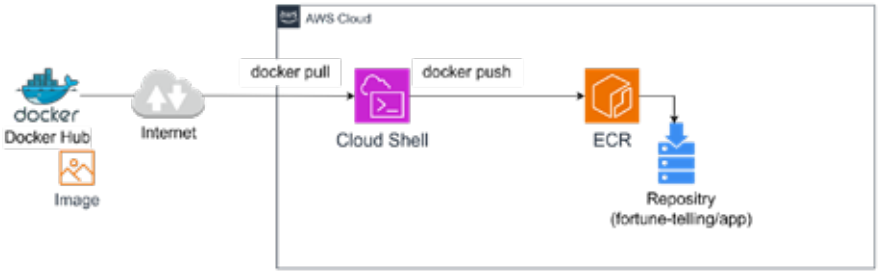
演習4 - 解答編

総合演習　ーコンテナ構築ー

演習4　総合演習　解答編

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。



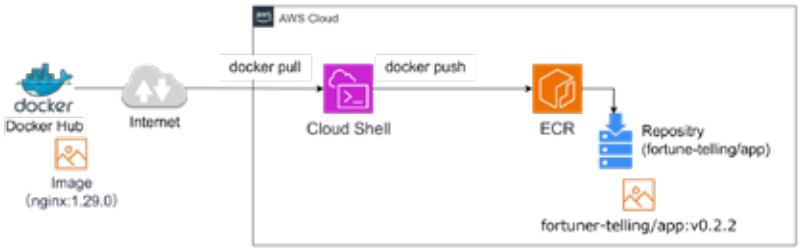
- ECRの設定値は以下の通りです。

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　解答編

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　解答編

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　解答編

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　解答編

コンテナ構築　CDKコード

- AWS CDKを利用して、おみくじ占いアプリケーションに対応したコンテナ基盤を構築します。
  段階的にスタックを追加・修正していきましょう。

※赤字のスタックおよびbin/ecs-demo.tsを修正します。

スタック名	用途	修正内容
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、Alb2Stack	ALB、WAFを定義	
EcrStack	ECR(リポジトリ)を定義	FortuneTellingアプリ用のリポジトリを追加
RdsStack	RDS(MySQL)のインスタンスを定義	
EcsStack、**Ecs2Stack**	ECS(クラスタ/タスク/サービス)を定義	Ecs2Stackを新規追加し、FortuneTellingアプリを起動
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4　総合演習　解答編

コンテナ構築　CDKコード

- bin/ecs-demo.tsにて、Ecs2Stackに渡すパラメータは以下の通りです。

Ecs2Stack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.fortuneTellingRepo	取得元ECRリポジトリ
ALB連携	targetGroup	alb2.tgblue	登録先ターゲットグループ（Alb2/Blue側）

演習4　総合演習　解答編

コンテナ構築　CDKコード

- bin/ecs-demo.tsのコードは以下の通りです。
  EcsStackの次に、Ecs2Stackのセクションを追加しています。

```
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';
import { AcmStack } from '../lib/acm-stack';
import { AlbStack } from '../lib/alb-stack';
import { Alb2Stack } from '../lib/alb2-stack';
import { EcrStack } from '../lib/ecr-stack';
import { RdsStack } from '../lib/rds-stack';
import { EcsStack } from '../lib/ecs-stack';
import { Ecs2Stack } from '../lib/ecs2-stack';  //★追加
import { ConnectionStack } from '../lib/connection-stack';
import { IamStack } from '../lib/iam-stack';
import { BuildStack } from '../lib/build-stack';
import { DeployStack } from '../lib/deploy-stack';
import { PipelineStack } from '../lib/pipeline-stack';

・・・省略・・・
```

```
// ECS / Fargate for CustomerInfoApp
const ecs = new EcsStack(app, 'EcsStack', {
  env,
  vpc       : net.vpc,
  ecsSg     : net.ecsSg,
  repo      : ecr.repository,
  targetGroup: alb.tgBlue,
});

// ECS / Fargate for FortuneTellingApp  ★セクション追加
const ecs2 = new Ecs2Stack(app, 'Ecs2Stack', {
  env,
  vpc       : net.vpc,
  ecsSg     : net.ecsSg,
  repo      : ecr.fortuneTellingRepo,
  targetGroup: alb2.tgBlue,
});

・・・省略・・・
```

演習4　総合演習　解答編

コンテナ構築　CDKコード

- 続いて、lib/ecr-stack.tsのコードに新規リポジトリを追加します。修正箇所は以下の通りです。

ecr-stack.ts　解答例（1/2）

```
・・・省略・・・
// 2. スタック初期化
export class EcrStack extends cdk.Stack {
  public readonly repository: ecr.Repository;  // 顧客情報表示機能用
  public readonly fortuneTellingRepo: ecr.Repository;  // ★おみくじ占い用

    ・・・省略・・・
   // 4. FortuneTellingApp リポジトリ作成　★セクション追加
   this.fortuneTellingRepo = new ecr.Repository(this, 'FortuneTellingAppRepo', {
     repositoryName : 'fortune-telling/app',
     imageScanOnPush: true,
     removalPolicy  : cdk.RemovalPolicy.RETAIN,
   });

    ・・・省略・・・
```

ポイント

2. スタック初期化

- 新規リポジトリの追加に伴い、fortuneTellingRepoを外部公開します。
- 既存の顧客情報表示機能のリポジトリは変わらず公開しますが、このタイミングで、repositoryから、customerInfoRepoなどに名称変更すると変数の命名規則を整えると良いでしょう。

4. FortuneTellingAppリポジトリ作成

- ECRに新しいリポジトリを作成するセクションを追加します。
- 変数およびリポジトリ名称を「fortune-telling/app」相当に変更するのみです。

演習4　総合演習　解答編

コンテナ構築　CDKコード

■ 続いて、lib/ecr-stack.tsのコードに新規リポジトリを追加します。修正箇所は以下の通りです。

ecr-stack.ts　解答例（2/2）

```
    ・・・省略・・・
  // 5. ライフサイクルルール　　★2リポジトリにまとめて設定するためにコードを修正
  const lifecycleRule = {
    description : 'Delete images older than 7 days',
    maxImageAge : cdk.Duration.days(7),
    tagStatus   : ecr.TagStatus.ANY,
  };
  this.repository.addLifecycleRule(lifecycleRule);  // 既存アプリ
  this.fortuneTellingRepo.addLifecycleRule(lifecycleRule);  // 新規アプリ

  // 6. 出力
  // ★新規アプリケーションの出力設定　追加
  new cdk.CfnOutput(this, 'FortuneTellingAppRepoUri', {
    value      : this.fortuneTellingRepo.repositoryUri,
    description: 'ECR repository URI for fortune-telling/app',
    exportName : 'fortuneTellingAppRepoUri',
  });
 }
}
```

ポイント

5. ライフサイクルルール

- 2リポジトリにまとめてライフサイクルルールを設定できるようコードを修正しています。ライフサイクルルールを定義し、各リポジトリにaddLifecycleRuleで適用する構成としました。
- 複数アプリケーションの登録が想定される場合は、都度セクションを追加するのは非効率なため、なるべく修正が容易な形にコードを整理しておくことをお勧めします。

6. 出力

- Fortune-telling/appリポジトリURIを出力対象に追加します。

演習4　総合演習　解答編

コンテナ構築　イメージpush手順

■ 参考までに、CloudShellで、Nginxのベースイメージを取得し、ECRに登録する手順を掲載します。
Pushする先のリポジトリ名、イメージタグ名を間違えないよう注意してください。

```
// Nginxイメージをpull
ecs-demo $ docker pull nginx:1.29.0
ecs-demo $ docker images          //nginxのイメージが存在すること

// Nginxのイメージにタグ付け
ecs-demo $ AWS_ACCOUNT_ID=$(aws sts get-caller-identity --query "Account" --output text)
ecs-demo $ ECR_URL=${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_REGION}.amazonaws.com
ecs-demo $ docker tag nginx:1.29.0 $ECR_URL/fortune-telling/app:v0.2.2            // ★リポ名、Versionに注意
ecs-demo $ docker images             //タグ付けしたイメージが追加されていること

// ECRにログイン
ecs-demo $ aws ecr get-login-password | docker login --username AWS --password-stdin ${ECR_URL}

// ECRにイメージをpush
ecs-demo $ docker push $ECR_URL/fortune-telling/app:v0.2.2            // ★リポ名、Versionに注意
```

演習4　総合演習　解答編

コンテナ構築　CDKコード

■ 次に、lib/ecs2-stack.tsです。演習1-2 Step3のecs-stack.tsをコピーして新規作成してください。
コードの構成や内容は同じため、修正例のように各種変数や名称を一律修正するのみとなります。

ecs2-stack.ts　修正例

修正項目	修正前	修正後
スタックインタフェース	EcsStackProps	Ecs2StackProps
スタッククラス	EcsStack	Ecs2Stack
ECSクラスター名	customer-info-cluster	fortune-telling-cluster
タスク定義名	family: 'customer-info-task'	family: 'fortune-telling-task'
イメージバージョン	'v0.1.1'	'v0.2.2'
ロググループ名	streamPrefix: 'customer-info'	streamPrefix: 'fortune-telling'
アプリケーション名	APP_NAME: 'customer-info'	APP_NAME: 'fortune-telling'
サービス名	serviceName: 'customer-info-service'	serviceName: 'fortune-telling-service'

演習4　総合演習　解答編

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**が存在すること
- 上記リポジトリにタグv0.2.2のイメージが登録されていること

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　解答編

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。

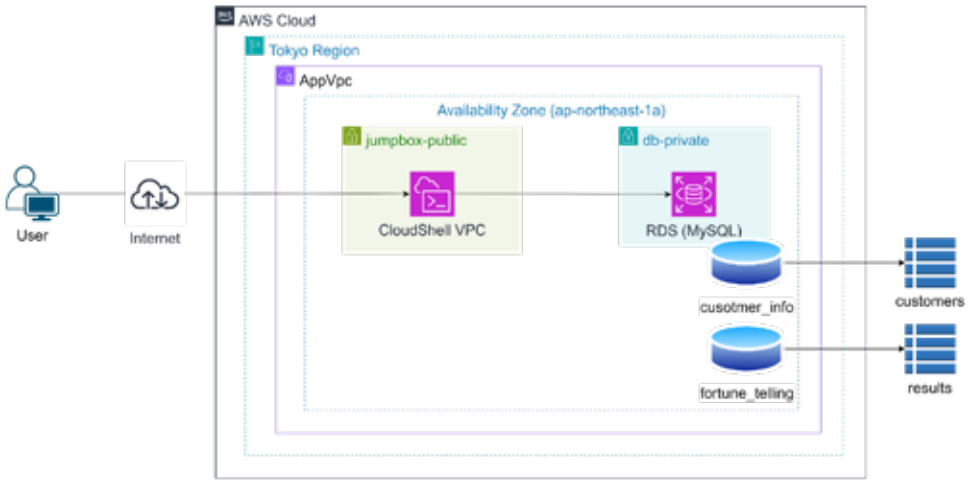
演習4 - 解答編

総合演習　―DB構築―

演習4　総合演習　解答編

DB構築　RDS

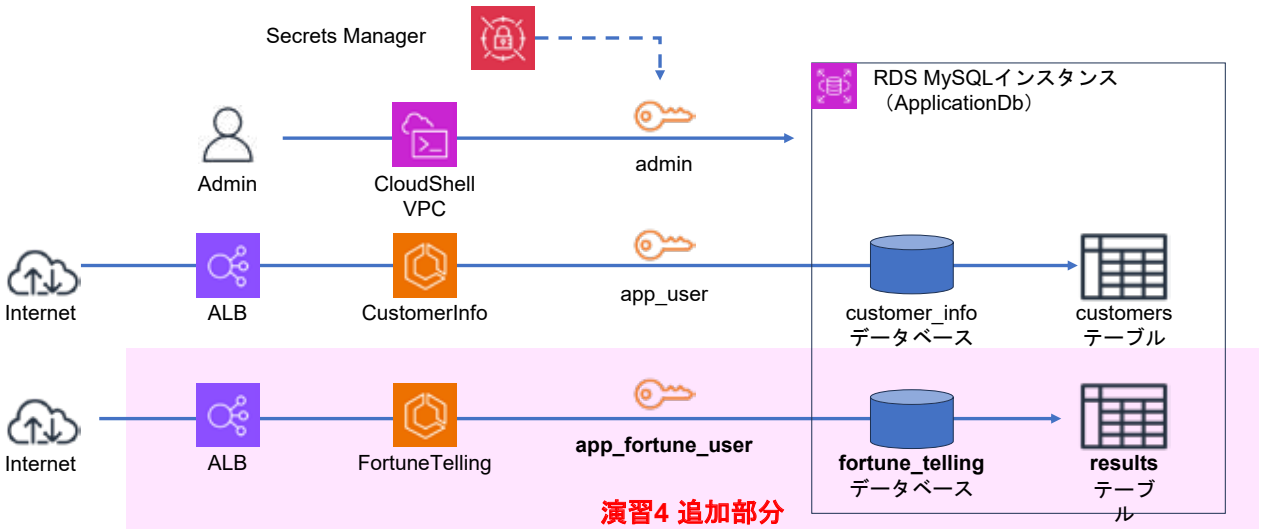
- 既存RDSインスタンスに新規データベースおよび新規テーブルを作成し、占い結果データを登録してください。また、おみくじ占いアプリケーション用のシークレットも作成してください。



演習4　総合演習　解答編

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　解答編

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　解答編

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　解答編

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　解答編

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　解答編

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　解答編

DB構築　CDKコード

- AWS CDKを利用して、おみくじ占いアプリケーションに対応したネットワーク基盤を構築します。
  演習4も段階的にスタックを追加・修正していきましょう。

スタック名	用途	修正内容
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、Alb2Stack	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	おみくじ占い用のシークレット作成を追加
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4　総合演習　解答編

DB構築　CDKコード

■ lib/rds-stack.tsに新規シークレットを追加します。修正箇所は以下の通りです。

rds-stack.ts　解答例（1/2）

```
・・・省略・・・
// 3. スタック初期化
Export class RdsStack extends cdk.Stack {
  public readonly fortuneAppSecret: secretsmanager.Isecret;  // ★追加

    ・・・省略・・・
    // 6. シークレット作成（fortune 用）　★セクション追加
    this.fortuneAppSecret = new secretsmanager.Secret(this,
'FortuneTellingAppSecret', {
      secretName: 'fortune-telling-app-credentials',
      generateSecretString: {
        secretStringTemplate: JSON.stringify({ username: 'app_fortune_user' }),
        generateStringKey: 'password',
        excludePunctuation: true,
      },
    });

    ・・・省略・・・
```

ポイント

3. スタック初期化

- 新規アプリケーション（FortuneTelling）のシークレットの追加に伴い、fortuneAppSecretを外部公開します。
- その他パラメータは変更せず、外部公開のままとします。

6. シークレット作成（fortune用）

- FortuneTelling用のシークレットを作成するセクションを追加します。
- シークレット名および変数以外は、既存のシークレット作成とコード内容は同じです。

演習4　総合演習　解答編

DB構築　CDKコード

■ lib/rds-stack.tsに新規シークレットを追加します。修正箇所は以下の通りです。

rds-stack.ts　解答例（2/2）

```
    ・・・省略・・・

    // 8. 出力
    // ★追加　FortuneTellingアプリのシークレット情報
    new cdk.CfnOutput(this, 'FortuneAppSecretName', {
      value: this.fortuneAppSecret.secretName,
    });
  }
}
```

ポイント

8. 出力

- 新規アプリケーション（fortuneTelling）のシークレット追加に伴い、シークレット名（fortuneApp.secretName）を公開します。
- その他出力項目は変更しません。

演習4　総合演習　解答編

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
- 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　解答編

DB構築　CLI操作

■ 続いて、RDSインスタンスにアプリケーション用MySQLユーザー(app_fortune_user)を作成します。

```
// adminでMySQLログインおよび初期データベース確認済の前提です
// ユーザー確認（app_fortune_userが存在しないことを確認）
MySQL [(none)]> SELECT  host, user FROM mysql.user;

// ユーザー作成
// '<password>'は、SecretsManagerの
// fortune-telling-app-credentialsのpasswordに合わせてください。
MySQL [(none)]> CREATE USER 'app_fortune_user'@'%'
IDENTIFIED BY '<password>';

// ユーザー権限付与
MySQL [(none)]> GRANT SELECT, INSERT, UPDATE ON
fortune_telling.* TO 'app_fortune_user'@'%';
// 権限の即時反映
MySQL [(none)]> FLUSH PRIVILEGES;

// ユーザー確認（app_userが存在することを確認）
MySQL [(none)]> SELECT  host, user FROM mysql.user;

// ユーザー権限確認
MySQL [(none)]> SHOW GRANTS FOR 'app_fortune_user'@'%';
```

ポイント

ユーザー作成

- <password>はSecrets Managerのfortune-telling-app-credentialsのパスワードと一致させてください。

ユーザー権限付与

- 本番環境では、アプリケーションに対して適切な操作権限を付与してください。例えば参照するだけならSELECTに絞る。

動作確認

- 適切にユーザーが作成されていれば、app_fortune_userでMySQLにログインできるようになります。必要に応じて確認してください。

演習4　総合演習　解答編

DB構築　CLI操作

■ 続いて、fortune_tellingデータベースとresultsテーブルを作成します。

```
// adminでMySQLログインおよび初期データベース確認済の前提です
// 新規データベース作成（fortune_telling）
MySQL [None]> CREATE DATABASE fortune_telling
  CHARACTER SET utf8mb4
  COLLATE utf8mb4_0900_ai_ci;

// データベース選択
MySQL [fortune_telling]> USE fortune_telling;

// テーブル作成（results）
MySQL [fortune_telling]> CREATE TABLE results (
  id BIGINT PRIMARY KEY AUTO_INCREMENT,
  number VARCHAR(10) NOT NULL,
  fortune_rank VARCHAR(10) NOT NULL,
  message VARCHAR(255) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

// resultsテーブル構成確認
MySQL [fortune_telling]> DESCRIBE results;
```

ポイント

fortune_tellingデータベース作成

- create databaseで新規データベースを作成します。
- 文字コードや照合順序も合わせて設定します。

resultsテーブル作成

- テーブル作成時、各カラムの条件(PRIMARY KEYなど)を忘れず設定してください。

演習4　総合演習　解答編

DB構築　CLI操作

■ 最後にresultsテーブルにデータを登録できれば、DB構築は完了です。

```
// データ投入（登録データ5行分をINSERT）
MySQL [fortune_telling]> INSERT INTO results (number,
fortune_rank, message) VALUES
('一', '大吉', '新しいアイデアに挑戦すると吉！'),
('二', '中吉', '朝の散歩で運気アップ。'),
('三', '小吉', 'メール整理でチャンス到来。'),
('四', '末吉', '焦らず準備を整えると好転。'),
('五', '凶', '無理は禁物。こまめな休憩を。'),
('六', '大吉', 'ありがとうを先に言うと良縁。'),
('七', '中吉', '机を片付けると運気が巡る。'),
('八', '小吉', '丁寧な言葉遣いが鍵。'),
('九', '末吉', '背伸びより継続を選ぼう。'),
('十', '凶', '夜更かしは控えて体調第一。');

// データ確認
MySQL [fortune_telling]> SELECT * FROM results;
```

ポイント

データ登録

- データは一括登録しても一件ずつ投入しても構いません。
- 必要であれば、初期データ移行を検討しても良いでしょう。

実行結果（SELECT * FROM results;）

演習4　総合演習　解答編

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　解答編

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 - 解答編

総合演習　ーGitHub準備ー

演習4　総合演習　解答編

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.conf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　解答編

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　解答編

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　解答編

GitHub　アプリケーションコード

■ GitHubに格納する各ファイルについて解説します。 /app/app.pyのコードは以下の通りです。

/app/app.py　解答例（1/6）

```
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-1")
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))
```

ポイント

- Flaskを利用した簡易なWebアプリを作ります。リクエストに対し、SecretsManagerで取得した認証情報を使って、RDS(MySQL)に接続して、 Web画面にテーブル内容を表示します。
- Flaskを初期化し、SecretsManagerからDB情報を取得しています。
- 必要な環境変数を設定します。

演習4　総合演習　解答編

GitHub　アプリケーションコード

■ GitHubに格納する各ファイルについて解説します。 /app/app.pyのコードは以下の通りです。

/app/app.py　解答例（2/6）

```
# ===== Secrets Manager から DB 認証情報を取得（起動時キャッシュ）=====
_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,
    )
```

ポイント

- DB_SECRET_NAMEを参照して、Secrets ManagerからDB認証情報としてusername / passwordを取得します。
- get_conn関数でDB接続します。

演習4　総合演習　解答編

GitHub　アプリケーションコード

■ GitHubに格納する各ファイルについて解説します。 /app/app.pyのコードは以下の通りです。

/app/app.py　解答例（3/6）

```
# ===== 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>
"""
```

ポイント

- HTMLでトップ画面(/fortune画面)を実装します。
- 「占う」ボタンを押すと、/resultsに遷移します。
- 本コードは最低限表示するHTMLです。サンプル画面の様に、ボタンや表示の装飾をして頂いても良いです。

演習4　総合演習　解答編

GitHub　アプリケーションコード

■ GitHubに格納する各ファイルについて解説します。 /app/app.pyのコードは以下の通りです。

/app/app.py　解答例（4/6）

```
# ===== 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>
"""
```

ポイント

- HTMLで/result画面を実装します。
- 後程DBから取得する情報を変数で入れ込んで、表示します。
- 「Topに戻る」ボタンを押すと、/fortuneに遷移します。
- こちらも最低限表示するHTMLです。サンプル画面の様に、ボタンや表示の装飾をして頂いても良いです。

演習4　総合演習　解答編

GitHub　アプリケーションコード

■ GitHubに格納する各ファイルについて解説します。 /app/app.pyのコードは以下の通りです。

/app/app.py　解答例（5/6）

```
# ===== 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
```

ポイント

- app.get(“/”)とapp.get(“/fortune”)はシンプルにGETでHTML画面を表示します。
- 今回ALBのヘルスチェックのパスは”/”に設定にしており、ヘルスチェック200OKを返却します。
- 続いて”/fortune”でTOP_HTMLを表示し、「おみくじ占い」のトップ画面を呼び出します。
- app.route(“/fortune”)でRESULT_HTMLを表示し、占いの結果画面を呼び出します。その前に、SQL文で「resultsテーブル」からランダムに1件だけ、おみくじ番号、運勢、開運メッセージを取得しています。

演習4　総合演習　解答編

GitHub　アプリケーションコード

■ GitHubに格納する各ファイルについて解説します。 /app/app.pyのコードは以下の通りです。

/app/app.py　解答例（6/6）

```
    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)))
```

ポイント

- DB接続時、DBが存在しない、もしくはテーブルのデータが空の場合は、代替の占い結果を用意し、RESULT_HTMLに渡します。

演習4　総合演習　解答編

GitHub　アプリケーションコード

■ Dockerfileのコードは以下の通りです。

Dockerfile　解答例

```
# ランタイム、OSパッケージ（MySQL, curl など）
FROM python:3.12-slim
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"]
```

ポイント

- Dockerfileの内容をベースにコンテナイメージを固めていきます。
- 今回のベースイメージは軽量なpython:3.12-slimを利用します。他Versionのイメージも利用可能です。
- requirements.txtに書かれたパッケージをインストールして、app.pyを/etc配下のフォルダに配置しています。
- アプリケーションはポート80番を公開します。
- 最後CMDで、Flaskアプリを実行するgunicorn を動かします。

演習4　総合演習　解答編

GitHub　アプリケーションコード

■ その他ファイルは概ね変更ありません。customer-infoリポジトリをベースにファイル作成してください。リポジトリやパイプライン、ロググループ等の固有名をfortuneTelling相当に直せばGitHubは準備完了です。

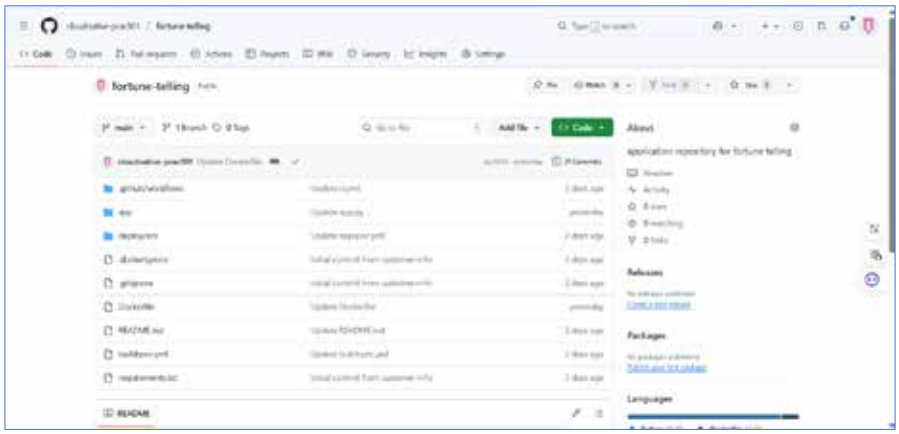
GitHub　各ファイル　修正例

対象ファイル	修正項目	修正前	修正後
.github/workflows/ci.yml	パイプライン名	CustomerInfoPipeline	FortuneTellingPipeline
buildspec.yml	ECRリポジトリ名	ECR_REPO: “customer-info/app"	ECR_REPO: "fortune-telling/app"
	ロググループ名	LOG_GROUP: "/ecs/customer-info"	LOG_GROUP: "/ecs/fortune-telling"
/deploy/ecs/taskdef.tmpl.json	タスク名	"family": "customer-info-task”	"family": "fortune-telling-task"
	ロググループ名	"awslogs-group": "/ecs/customer-info"	"awslogs-group": "/ecs/fortune-telling"
	ログストリーム	"awslogs-stream-prefix": “customer-info"	"awslogs-stream-prefix": "fortune-telling"

演習4　総合演習　解答編

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。

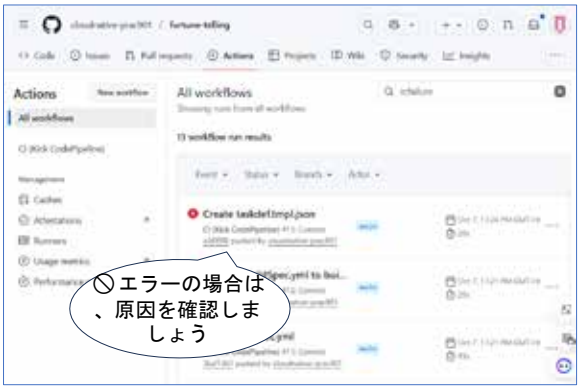


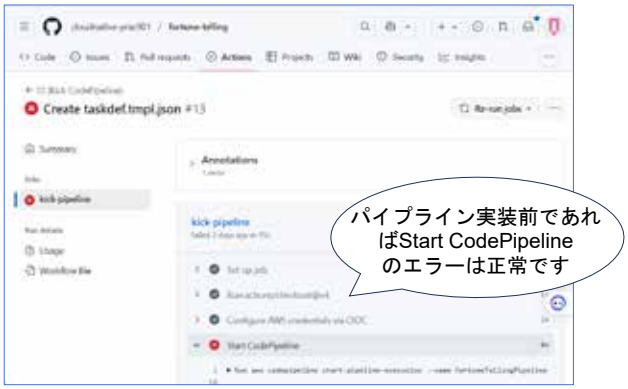
fortune-tellingリポジトリ

演習4　総合演習　解答編

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。





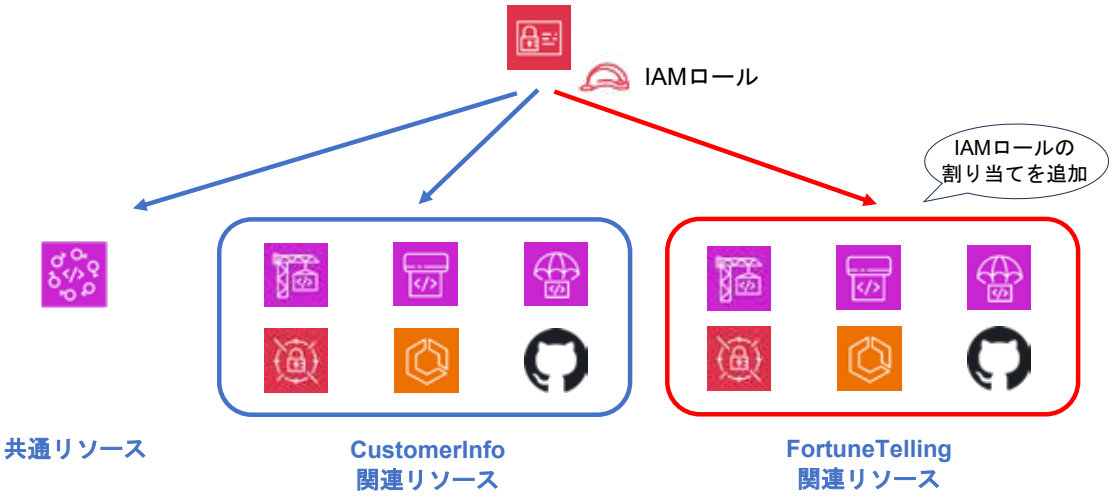
演習4 - 解答編

総合演習　ーCICDパイプライン構築ー

演習4　総合演習　解答編

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　解答編

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 - CodeBuildレポート機能 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4　総合演習　解答編

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。

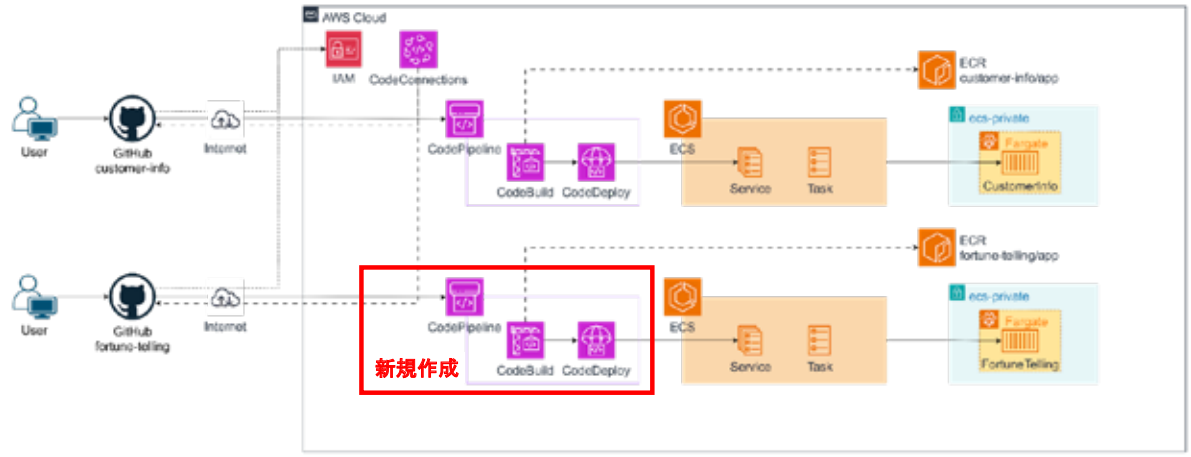
IAM設定値（2/2）

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	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4　総合演習　解答編

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　解答編

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　解答編

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　解答編

コンテナ構築　CDKコード

- AWS CDKを利用して、おみくじ占いアプリケーションに対応したCI/CDパイプラインを構築します。
  段階的なスタックもこれで完了です。

※赤字のスタックおよびbin/ecs-demo.tsを修正します。

スタック名	用途	修正内容
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、Alb2Stack	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、Ecs2Stack	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	IAMロールを付与する対象リソースの追加
BuildStack、**Build2Stack**	CodeBuildプロジェクトを定義	Build2Stackを新規追加し、新規ビルドプロジェクトを定義
DeployStack、**Deploy2Stack**	CodeDeployアプリおよびデプロイメントを定義	Deploy2Stackを新規追加し、新規デプロイ設定を定義
PipelineStack、**Pipeline2Stack**	CodePipelineパイプラインを定義	Pipeline2Stackを新規追加し、新規パイプラインを定義

演習4　総合演習　解答編

コンテナ構築　CDKコード

- bin/ecs-demo.tsにて、各スタックに渡すパラメータを確認します。
  iam-stackに対しては、パラメータが複数になるため、配列で渡すように修正しました。
  ※単純に連携パラメータの数を増やしても構いません（例：ghRepo/ghRepo2、ecrRepoName/ecrRepoName2）

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	ecrRepoNames	'customer-info/app' 'fortune-telling/app'	CodeBuild が push/pull する ECR リポジトリ名
パイプライン	pipelineNames	'CustomerInfoPipeline' 'FortuneTellingPipeline'	CodePipeline名
GitHub	ghOwner ※	<GitHubアカウント>	GitHubのアカウント名
	ghRepos	'customer-info' 'fortune-telling'	GitHubのリポジトリ名
CodeConnections	gitHubConnectionArn	conn.connectionArn	CodeConnections
RDS	appSecretArns	rds.appSecret.secretArn rds.fortuneAppSecret.secretArn	アプリケーションユーザ(app_fortune_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4　総合演習　解答編

コンテナ構築　CDKコード

- bin/ecs-demo.tsにて、Build2Stackに渡すパラメータは以下の通りです。

Build2Stack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	'fortune-telling/app'	ECRリポジトリ名

演習4　総合演習　解答編

コンテナ構築　CDKコード

- bin/ecs-demo.tsにて、Deploy2Stackに渡すパラメータは以下の通りです。
  Ecs2Stack、Alb2Stackからパラメータを渡すため、設定する値（ecs2、alb2）に注意してください。

Deploy2Stack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	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	ALBリスナー（本番リスナー）
	testListenerArn	alb2.listenerTest.listenerArn	ALBリスナー（テストリスナー）
	tgBlueName	alb2.tgBlue.targetGroupName	ALBターゲットグループ（Blue側）
	tgGreenName	alb2.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	'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

演習4　総合演習　解答編

コンテナ構築　CDKコード

- bin/ecs-demo.tsにて、Pipeline2Stackに渡すパラメータは以下の通りです。

Pipeline2Stack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	'FortuneTellingPipeline'	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	'fortune-telling'	GitHubリポジトリ
	gitHubBranch	'main'	GitHubブランチ
ECR	ecrRepoName	'fortune-telling/app'	ECRリポジトリ
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
	dbHost	rds.dbHost	RDS DBホスト

演習4　総合演習　解答編

コンテナ構築　CDKコード

- bin/ecs-demo.tsの修正箇所は以下の通りです。まずIamStackへ渡すパラメータを修正します。
  importの順にBuild2 / Deploy2 / Pipeline2スタックのセクションを追加します。

ecs-demo.ts　解答例（1/2）

```
#!/usr/bin/env node
import 'source-map-support/register';
import { App } from 'aws-cdk-lib';
import { NetStack } from '../lib/net-stack';
import { VpceStack } from '../lib/vpce-stack';
import { AcmStack } from '../lib/acm-stack';
import { AlbStack } from '../lib/alb-stack';
import { Alb2Stack } from '../lib/alb2-stack';
import { EcrStack } from '../lib/ecr-stack';
import { RdsStack } from '../lib/rds-stack';
import { EcsStack } from '../lib/ecs-stack';
import { Ecs2Stack } from '../lib/ecs2-stack';
import { ConnectionStack } from '../lib/connection-stack';
import { IamStack } from '../lib/iam-stack';
import { BuildStack } from '../lib/build-stack';
import { Build2Stack } from '../lib/build2-stack'; // ★追加
import { DeployStack } from '../lib/deploy-stack';
import { Deploy2Stack } from '../lib/deploy2-stack'; // ★追加
import { PipelineStack } from '../lib/pipeline-stack';
import { Pipeline2Stack } from '../lib/pipeline2-stack';  // ★追加
・・・省略・・・
```

```
・・・省略・・・

// IAM ★複数のデータを渡せるように一部パラメータを配列化
const iam = new IamStack(app, 'IamStack', {
  env,
  ecrRepoNames: ['customer-info/app', 'fortune-telling/app'],
  pipelineNames: ['CustomerInfoPipeline', 'FortuneTellingPipeline'],
  ghRepos: [
    { owner: 'xxx', repo: 'customer-info', branches: ['main'] },
    { owner: 'xxx', repo: 'fortune-telling', branches: ['main'] },
  ],
  gitHubConnectionArn: conn.connectionArn,
  appSecretArns: [ rds.appSecret.secretArn,
rds.fortuneAppSecret.secretArn ],  // アプリケーション用の認証情報
});

・・・省略・・・
```

演習4　総合演習　解答編

コンテナ構築　CDKコード

- bin/ecs-demo.tsの修正箇所は以下の通りです。まずIamStackへ渡すパラメータを修正します。
  importの順にBuild2 / Deploy2 / Pipeline2スタックのセクションを追加します。

ecs-demo.ts　解答例（2/2）

```
・・・省略・・・
// CodeBuild for fortune-telling  ★セクション追加
const build2 = new Build2Stack(app, 'Build2Stack', {
  env,
  codeBuildRoleArn: iam.codeBuildRole.roleArn,
  ecrRepoName: 'fortune-telling/app',
});

// CodeDeploy for fortune-telling  ★セクション追加
const deploy2 = new Deploy2Stack(app, 'Deploy2Stack', {
  env,
  clusterName: ecs2.cluster.clusterName,
  serviceName: ecs2.service.serviceName,
  prodListenerArn: alb2.listenerProd.listenerArn,
  testListenerArn: alb2.listenerTest.listenerArn,
  tgBlueName: alb2.tgBlue.targetGroupName,
  tgGreenName: alb2.tgGreen.targetGroupName,
  codeDeployRoleArn: iam.codeDeployRole.roleArn,
  applicationName: 'FortuneTellingEcsApp',
  deploymentGroupName: 'FortuneTellingDG',
});
・・・省略・・・
```

```
・・・省略・・・
// CodePipeline for fortune-telling  ★セクション追加
new Pipeline2Stack(app, 'Pipeline2Stack', {
  env,
  pipelineName       : 'FortuneTellingPipeline',
  codeBuildRoleArn   : iam.codeBuildRole.roleArn,
  codeDeployRoleArn  : iam.codeDeployRole.roleArn,
  codePipelineRoleArn : iam.codePipelineRole.roleArn,
  ecsTaskExecutionRoleArn : iam.ecsTaskExecutionRole.roleArn,
  ecsTaskRoleArn          : iam.appTaskRole.roleArn,
  ecrRepoName        : 'fortune-telling/app',
  gitHubConnectionArn : conn.connectionArn,
  gitHubOwner        : '<xxx>',                // GitHubユーザ名に修正
  gitHubRepo         : 'fortune-telling',
  gitHubBranch       : 'main',
  ecsAppName             : 'FortuneTellingEcsApp',
  ecsDeploymentGroupName  : 'FortuneTellingDG',
  dbSecretArn            : rds.fortuneAppSecret.secretArn,
  dbHost                 : rds.dbHost,
});
```

演習4　総合演習　解答編

コンテナ構築　CDKコード

■ 続いて、lib/iam-stack.tsのコードです。IAMロールにリソースを追加します。修正箇所は以下の通りです。

iam-stack.ts　解答例（1/5）

```
// 2. インタフェース定義
// ★外部から受け取るパラメータを配列で渡す仕様に変更
// ★受け取るパラメータを倍に増やして良い（例：ecrRepoName/ecrRepoName2）
export interface IamStackProps extends cdk.StackProps {
  ecrRepoNames: string[];
  pipelineNames: string[];
  ghRepos: Array<{ owner: string; repo: string; branches?: string[] }>;
  gitHubConnectionArn?: string;
  appSecretArns: string[];}

// 3. スタック初期化
・・・省略・・・

    // ★ARNを配列で生成するよう処理を変更
    const ecrRepoArns = props.ecrRepoNames.map(
      name => `arn:aws:ecr:${region}:${account}:repository/${name}`
    );
    const pipelineArns = props.pipelineNames.map(
      name => `arn:aws:codepipeline:${region}:${account}:${name}`
    );
```

ポイント

2. インタフェース定義

- 外部パラメータを配列で受け取れるように定義を修正しています。今回、ECRリポジトリ名、パイプライン名、GitHubリポジトリ名、Secret名をそれぞれ2つずつ受け取ります。
- 各パラメータを個別に渡す形でも良いですが、アプリケーションの拡張性を考慮すると、パラメータ変数を増やすより、配列データや環境変数ファイルで渡す方が良いでしょう。

3. スタック初期化

- ECRリポジトリ名とパイプライン名のARN生成では、map関数でkey-valueの数だけARNを作って配列変数に格納するように変更しています。
- こちらもシンプルに生成するARNの数を増やしても構いません。

演習4　総合演習　解答編

コンテナ構築　CDKコード

■ 続いて、lib/iam-stack.tsのコードです。IAMロールにリソースを追加します。修正箇所は以下の通りです。

iam-stack.ts　解答例（2/5）

```
    // 4. CodeBuildロール作成
    // ECR
    this.codeBuildRole.addToPolicy(new iam.PolicyStatement({
      actions: ['ecr:GetAuthorizationToken'],
      resources: ['*'],
    }));

    this.codeBuildRole.addToPolicy(new iam.PolicyStatement({
      actions: [
        'ecr:BatchCheckLayerAvailability','ecr:InitiateLayerUpload','ecr:UploadLayerPart',
        'ecr:CompleteLayerUpload','ecr:PutImage','ecr:BatchGetImage','ecr:GetDownloadUrlForLayer',
      ],
      resources: ecrRepoArns,  //★リポジトリARN配列を対象リソースに指定
    }));
```

ポイント

4. CodeBuildロール作成

- ECRのリポジトリ操作ポリシーは、対象リポジトリを設定しているため、resourcesの対象にfortune-telling/appも追加します。resourcesにecrRepoArnsを設定することで、両方のリポジトリに適用されます。
- CodeBuildロールに付与するポリシーは色々ありますが、その他ポリシーの適用先は変更不要です。resources: ['*']としていて、リソース追加がいらないためです。

演習4　総合演習　解答編

コンテナ構築　CDKコード

■ 続いて、lib/iam-stack.tsのコードです。IAMロールにリソースを追加します。修正箇所は以下の通りです。

iam-stack.ts　解答例（3/5）

```
  // 8. Secrets Manager（アプリ単位）
  // ★シークレットARNを配列にし、アプリケーション単位でシークレットの操作権限を付与
  if (props.appSecretArns?.length) {
    props.appSecretArns.forEach((arn, idx) => {
      const secret = secretsmanager.Secret.fromSecretCompleteArn(this, `AppSecret${idx}`, arn);
      secret.grantRead(this.appTaskRole);
      secret.grantRead(this.ecsTaskExecutionRole);
    });
  }
```

ポイント

8. Secrets Manager

- Secrets Managerの操作権限を与える際、操作可能なシークレットを追加します。
- 操作可能なシークレットはARNで指定しているため、配列で渡すように定義を変更することで、複数シークレットに対して操作できるようになります。

演習4　総合演習　解答編

コンテナ構築　CDKコード

■ 続いて、lib/iam-stack.tsのコードです。IAMロールにリソースを追加します。修正箇所は以下の通りです。

iam-stack.ts　解答例（4/5）

```
  // 10. GitHub OIDCプロバイダー（修正箇所のみ）
  // ★複数リポジトリ/ブランチを許可できるように変更
  const subPatterns = props.ghRepos.flatMap(r => {
    const branches = r.branches && r.branches.length ? r.branches : ['main'];
    return branches.map(b => `repo:${r.owner}/${r.repo}:ref:refs/heads/${b}`);
  });
  this.githubOidcRole = new iam.Role(this, 'GitHubOIDCRole', {
    roleName: 'GitHubOIDCRole',
    description: 'GitHub Actions OIDC role for multiple repos',
    assumedBy: new iam.WebIdentityPrincipal(provider.openIdConnectProviderArn, {
      StringEquals: { 'token.actions.githubusercontent.com:aud': 'sts.amazonaws.com' },
      StringLike: { 'token.actions.githubusercontent.com:sub': subPatterns },
    }),
  });
```

ポイント

10. GitHub OIDCプロバイダー

- OIDC認証可能なリポジトリとブランチが複数になるように配列で指定できるようにしています。
  これにより、customer-infoだけでなく、fortune-tellingからもAWSに対してOIDC認証できるようになります。

演習4　総合演習　解答編

コンテナ構築　CDKコード

■ 続いて、lib/iam-stack.tsのコードです。IAMロールにリソースを追加します。修正箇所は以下の通りです。

iam-stack.ts　解答例（5/5）

```
    // 10. GitHub OIDCプロバイダー（修正箇所のみ） 続き
    // 各Pipelineを実行可能にする
    this.githubOidcRole.addToPolicy(new iam.PolicyStatement({
      actions: ['codepipeline:StartPipelineExecution'],
      resources: pipelineArns,  //★パイプラインARN配列を対象リソースに指定
    }));

    // 11. 出力
      ・・・省略・・・

    new cdk.CfnOutput(this, 'TargetPipelineArns',      { value: pipelineArns.join(',')
});  // ★配列に変更
    new cdk.CfnOutput(this, 'TargetEcrRepoArns',       { value:
ecrRepoArns.join(',') });   // ★配列に変更
  }
}
```

ポイント

10. GitHub OIDCプロバイダー

- OIDC認証後、キック可能なパイプラインも配列化したARNで渡すことで、複数のパイプラインをキックできるようになります。

演習4　総合演習　解答編

コンテナ構築　CDKコード

■ 次に、lib/build2-stack.ts、deploy2-stack.tsです。
既存のbuild-stack.ts、deploy-stack.tsをコピーして新規作成してください。
コードの構成や内容は同じため、修正例のように各種変数や名称を一律修正するのみとなります。

build2-stack.ts　修正例

修正項目	修正前	修正後
スタックインタフェース	BuildStackProps	Build2StackProps
スタッククラス	BuildStack	Build2Stack
ECRリポジトリ名	'customer-info/app'	'fortune-telling/app'
ビルドプロジェクト名	projectName: 'customer-info-app'	projectName: 'fortune-telling-app'

deploy2-stack.ts　修正例

修正項目	修正前	修正後
スタックインタフェース	DeployStackProps	Deploy2StackProps
スタッククラス	DeployStack	Deploy2Stack

演習4　総合演習　解答編

コンテナ構築　CDKコード

- 次に、lib/pipeline2-stack.tsです。既存のpipeline-stack.tsを利用してください。
  コードの構成や内容は同じため、修正例のように各種変数や名称を一律修正するのみとなります。

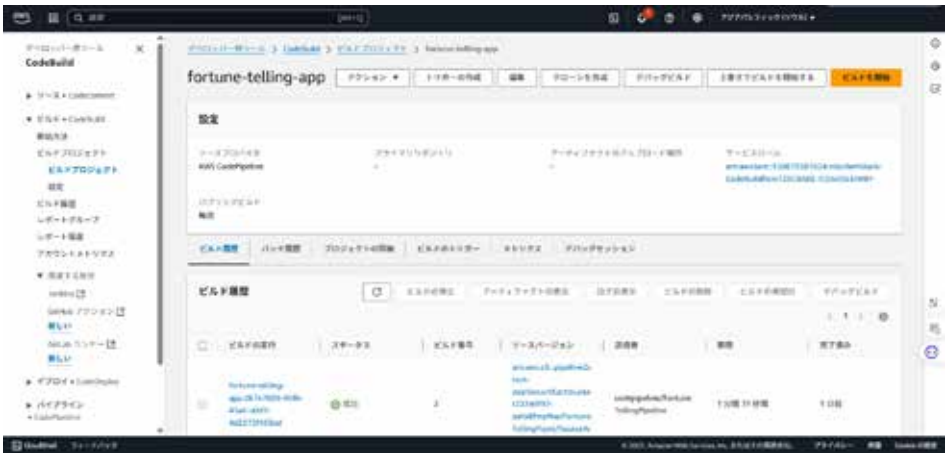
pipeline2-stack.ts　修正例

修正項目	修正前	修正後
スタックインタフェース	PipelineStackProps	Pipeline2StackProps
スタッククラス	PipelineStack	Pipeline2Stack
ビルドプロジェクト名	'customer-info-app'	'fortune-telling-app'
パイプライン名	'CustomerInfoPipeline'	'FortuneTellingPipeline'

演習4　総合演習　解答編

CICDパイプライン構築　動作確認

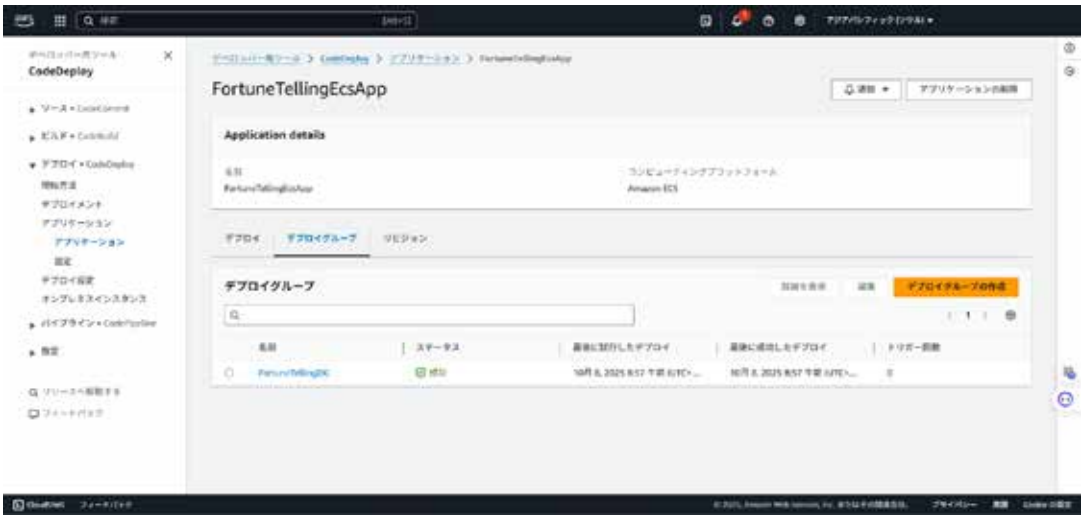
- Build2Stackの動作確認をします。CodeBuildにfortune-tellling-appプロジェクトが作成されていることを確認してください。
  - CodeBuild > ビルドプロジェクト > fortune-telling-app　で確認できます。
  - ビルドプロジェクト自体の動作確認は、Pipeline実装時に合わせて確認します。



演習4　総合演習　解答編

CICDパイプライン構築　動作確認

- 続いてDeploy2Stack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　解答編

CICD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。

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4　総合演習　解答編

CICD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に動作すれば、演習4は完了です。合わせて、顧客情報表示システムも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

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
解答編

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。