このコンテンツは公開から3年以上経過しており内容が古い可能性があります
最新情報についてはサービス別資料もしくはサービスのドキュメントをご確認ください

[AWS Black Belt Online Seminar]

AWS CloudFormation deep dive

サービスカットシリーズ

Senior Solutions Architect
大村 幸敬
2020/10/06

AWS 公式 Webinar
https://amzn.to/JPWebinar

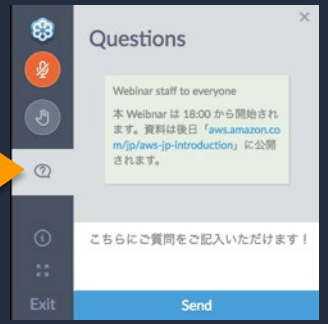
過去資料
https://amzn.to/JPArchive

AWS Black Belt Online Seminar とは

- **「サービス別」「ソリューション別」「業種別」のそれぞれのテーマに分かれて、アマゾン ウェブ サービス ジャパン株式会社が主催するオンラインセミナーシリーズです。**

- **質問を投げることができます！**
- 書き込んだ質問は、主催者にしか見えません
- 今後のロードマップに関するご質問は
  お答えできませんのでご了承下さい

① 吹き出しをクリック
② 質問を入力
③ Sendをクリック



Twitter ハッシュタグは以下をご利用ください
#awsblackbelt

内容についての注意点

- 本資料では2020年10月06日現在のサービス内容および価格についてご説明しています。最新の情報はAWS公式ウェブサイト(https://aws.amazon.com)にてご確認ください。

- 資料作成には十分注意しておりますが、資料内の価格とAWS公式ウェブサイト記載の価格に相違があった場合、AWS公式ウェブサイトの価格を優先とさせていただきます。

- 価格は税抜表記となっています。日本居住者のお客様には別途消費税をご請求させていただきます。

- AWS does not offer binding price quotes. AWS pricing is publicly available and is subject to change in accordance with the AWS Customer Agreement available at http://aws.amazon.com/agreement/. Any pricing information included in this document is provided only as an estimate of usage charges for AWS services based on certain information that you have provided. Monthly charges will be based on your actual use of AWS services, and may vary from the estimates provided.

大村 幸敬 (おおむら ゆきたか)

シニア ソリューションアーキテクト

- これからクラウドを使いはじめる エンタープライズ企業をサポート
- 運用系サービス & DevOps系サービス

好きなAWSのサービス： **AWS CLI, AWS CDK**

本セッションでお伝えすること

- 実運用でよく発生するユースケース別の使い方
- CloudFormation の 深い機能の使いかた
- リソースプロバイダの作りかた

話さないこと

- 基本的な使い方、一般的なユースケース
- こちらをご参照ください
  - AWS Blackbelt オンラインセミナー CloudFormation (2020/08開催)
    https://aws.amazon.com/jp/blogs/news/webinar-bb-aws-cloudformation-2020/
  - AWS Hands-on for Beginners -
    AWS環境のコード管理 AWS CloudFormationでWebシステムを構築する
    https://pages.awscloud.com/JAPAN-event-OE-Hands-on-for-Beginners-cfn-2020-reg-event-LP.html?trk=aws_introduction_page

Agenda

1. よくあるユースケース別使いかた
2. Deepな機能の使いかた
3. リソースプロバイダを使った独自リソースの作りかた
4. よくある質問

※本資料では CloudFormation = CFn と略記します

aws

よくあるユースケース別使いかた

よくあるユースケース

1. **各アカウントに基本設定（ベースライン）を展開する**
2. 手動で作った既存リソースをCFnの管理下に入れる
3. 既存のスタックを分割する（テンプレートリファクタリング）
4. Ansibleと組み合わせてサーバ構成を管理する
5. 1つのテンプレートから複数の環境をデプロイする

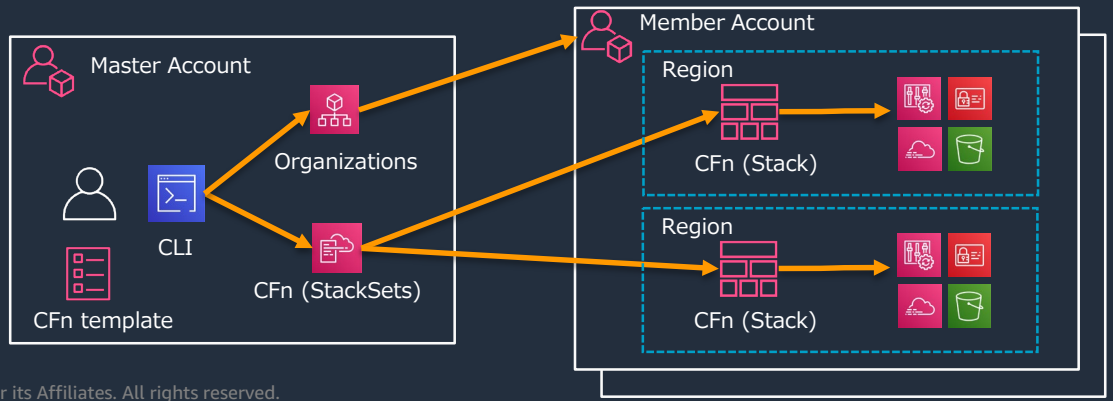
1. 各アカウントにベースラインを展開する

実現したいこと

- マルチアカウント環境で、アカウント作成時に基本の設定を展開したい
- ベースラインとして監査用ログ、構成のチェック、基本のIAMRoleを設定
- CloudFormationを使って効率よく展開したい

実装方針

- アカウント作成はOrganizations + CLI を使用（CFn未対応のため）
- CloudFormation StackSetsを使って対象アカウントにStackを作成

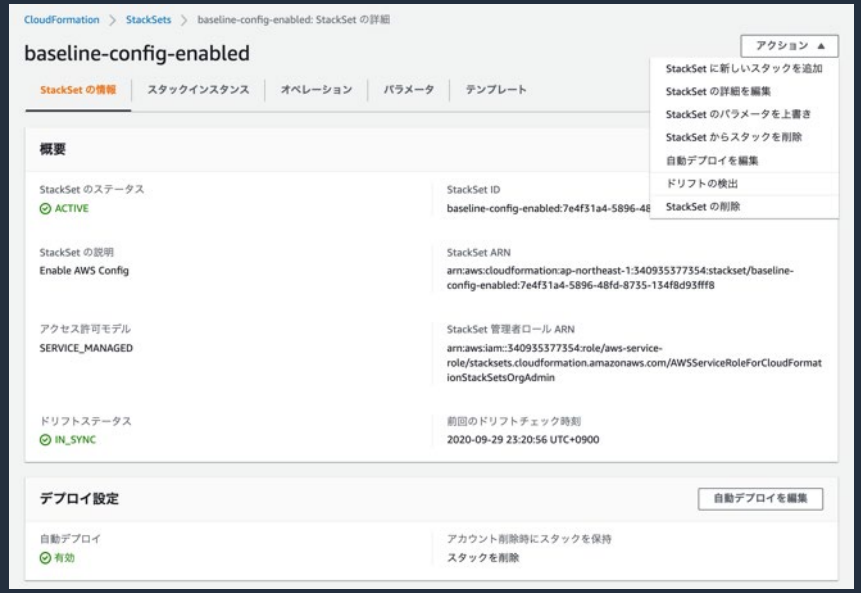


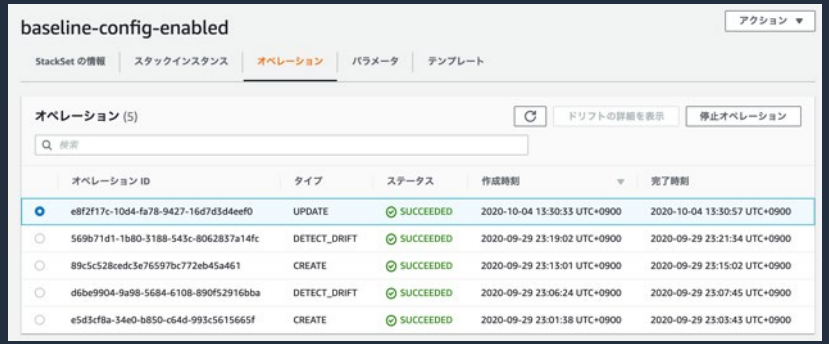
CFn StackSets 画面例 - 作成1

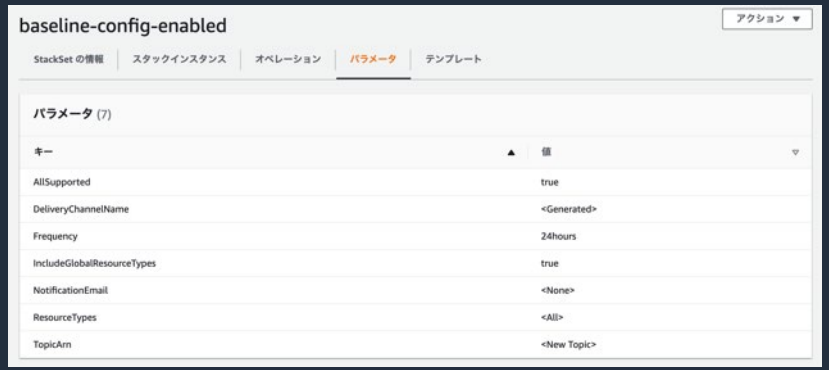
CFn StackSets 画面例 - 作成2

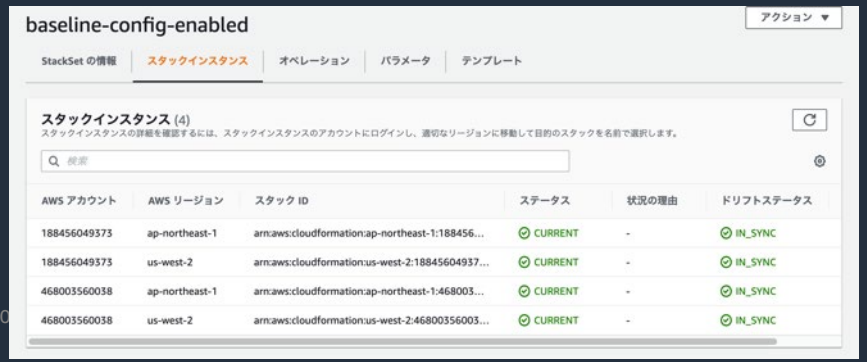
CFn StackSets 画面例 - 作成3

CFn StackSets 画面例 詳細画面

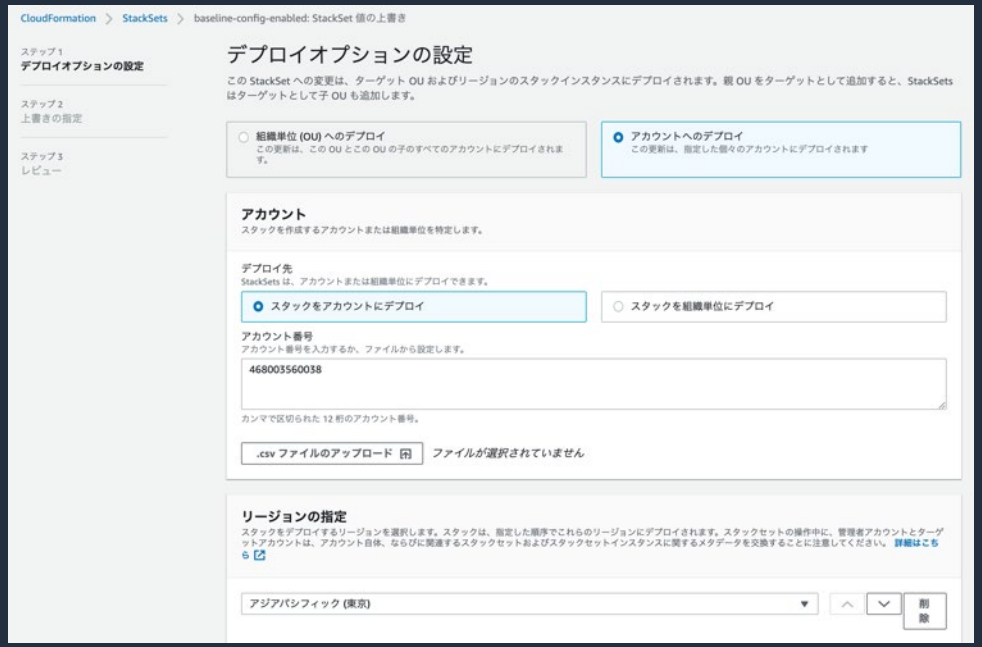


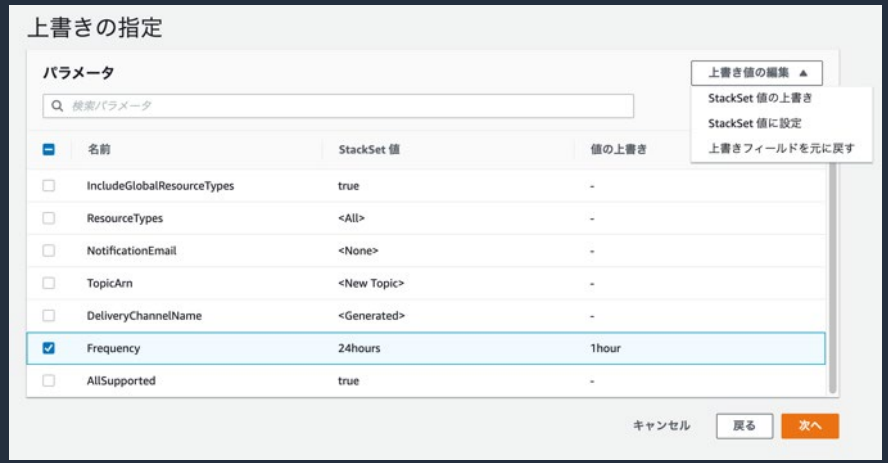






CFn StackSets 画面例 パラメータの上書き





StackSetsを使ったベースラインの展開

- デプロイ対象は複数のアカウントの複数のリージョンを指定
  - OUまたは個別アカウントのいずれかを指定することが多い
  - Organizationsがあると指定とロールの管理が楽だが必須ではない

- テンプレートとパラメータの指定
  - テンプレートは同じものを使用
  - パラメータはデプロイ対象に対して一括設定だが、特定のアカウントxリージョンを個別にデプロイする際にパラメータを上書きできる

- 組織やOUへのアカウント追加/削除をトリガにスタックを展開/削除可能
  - 1つのアカウントに対して指定する自動デプロイStackSetsは１つだけがよい（StackSetsの依存関係を記述できない）

ベースラインの例

- AWS ControlTowerの設定（参考）
  - ControlTowerドキュメントに記載のCFnテンプレート
    https://docs.aws.amazon.com/ja_jp/controltower/latest/userguide/guardrails-reference.html
  - ControlTowerが生成するスタック
    https://docs.aws.amazon.com/ja_jp/controltower/latest/userguide/account-factory.html#account-factory-considerations
    ControlTower環境を作るとCFn経由でテンプレートを参照できます

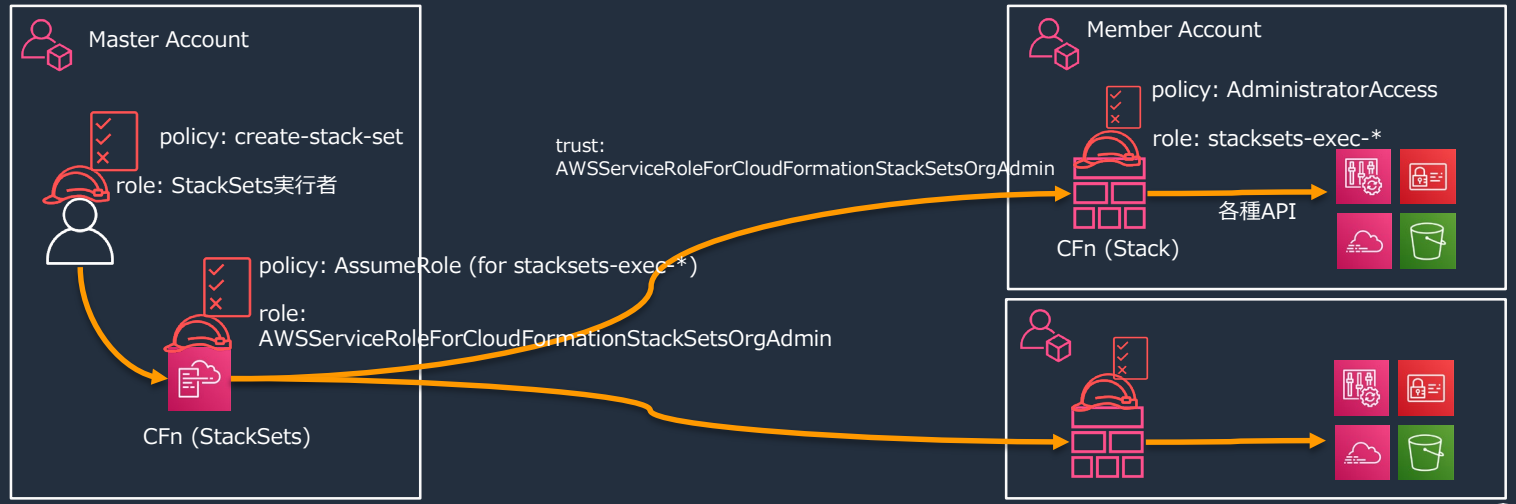
- ロギングの有効化
  - CloudTrail および AWS Configの有効化
  - AWS ConfigRulesでCloudTrailが有効であることを確認
    - サンプルテンプレートがStackSetsから利用可能
      https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/stacksets-sampletemplates.html

- 各アカウントの管理用 IAM Role (例: baseline-admin ロール)
  - 用途に合わせてAdministratorAccessや、ReadOnlyAccessポリシーを持つユーザを作成
  - ただし…Organizations SCP （または追加のポリシー）で以下を制限
    - Role名に baseline-* がついたロールの変更禁止（権限昇格の禁止）
    - CloudTrailおよびConfigの設定変更を禁止

    https://docs.aws.amazon.com/ja_jp/controltower/latest/userguide/mandatory-guardrails.html#cloudtrail-configuration-changes
    https://docs.aws.amazon.com/ja_jp/controltower/latest/userguide/mandatory-guardrails.html#config-disallow-changes

StackSetsの実行権限（マネージド）

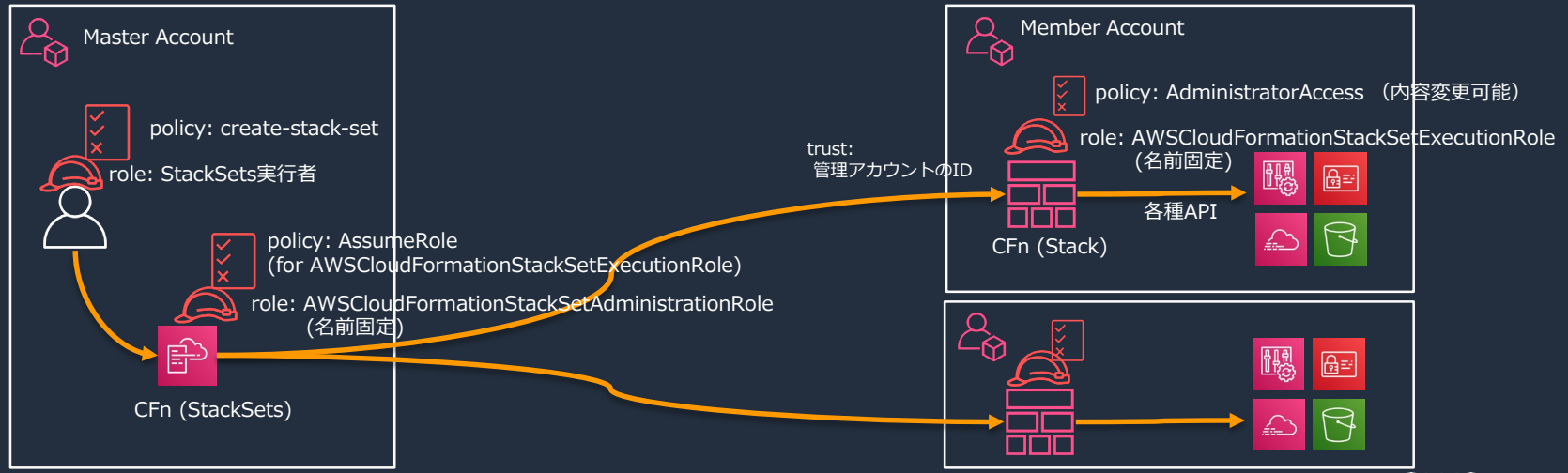
- Organizationsで「信頼されたアクセス」を有効にするとStackSetsのマネージドロールが自動生成される
  - StackSets管理アカウント側: AWSServiceRoleForCloudFormationStackSetsOrgAdmin
    - ポリシー: stacksets-exec-* ロールを引き受ける
    - 信頼関係: stacksets.cloudformation.amazonaws.com を信頼
  - ターゲットアカウント側: stacksets-exec-*
    - ポリシー: AdministratorAccess
    - 信頼関係: AWSServiceRoleForCloudFormationStackSetsOrgAdmin ロールを信頼
    - AWSServiceRoleForCloudFormationStackSetsOrgMember ロールがこれを作成する



https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/stacksets-orgs-enable-trusted-access.html

StackSetsの実行権限（セルフマネージド）

- それぞれ名前固定のロールを作成して必要な権限を渡す（Organizations不要）
- StackSets管理アカウント側: AWSCloudFormationStackSetAdministrationRole（名前固定）
  - ポリシー: AWSCloudFormationStackSetExecutionRoleを 引き受けられるように設定
  - 信頼関係: cloudformation.amazonaws.comを信頼
- ターゲットアカウント側: AWSCloudFormationStackSetExecutionRole（名前固定）
  - ポリシー: ターゲットアカウントでリソース作成に必要な権限を付与。通常はAdministratorAccess
  - 信頼関係: 管理側アカウントIDを信頼 (arn:aws:iam::admin_account_id:root)



https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/stacksets-prereqs-self-managed.html

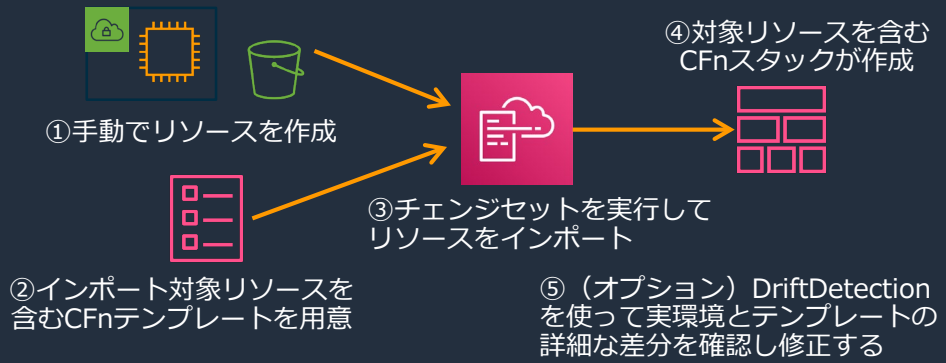
よくあるユースケース

1. 各アカウントに基本設定（ベースライン）を展開する
2. **手動で作った既存リソースをCFnの管理下に入れる**
3. 既存のスタックを分割する（テンプレートリファクタリング）
4. Ansibleと組み合わせてサーバ構成を管理する
5. 1つのテンプレートから複数の環境をデプロイする

CloudFormation Resource Import

手動で作った既存リソースをインポートしてCFnの管理下に入れる機能

- CFn Registry に対応したリソースが利用可能（順次追加中）
- ベースとなるテンプレートを用意
  - 必須プロパティを指定
  - 対象リソースに削除ポリシーを設定（DeletionPolicy: Retain）
  - インポート後 DriftDetection を行って細かな差分を調整する
- インポート時にテンプレートのリソースと実リソースIDとを紐付け



https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/resource-import.html

Former2を使ったインポート用テンプレートの生成

※Former2は3rd Partyによる
オープンソースのツールです

メリット

- Import用のテンプレートをFormer2で作ると効率よくインポートが可能
  - Former2 = 既存リソースからCFnテンプレートを生成するOSSツール
  - Import後のDriftDetection→テンプレート修正がほぼ不要
  - 生成したテンプレートに"DeletionPolicy: Retain"の追加が必要

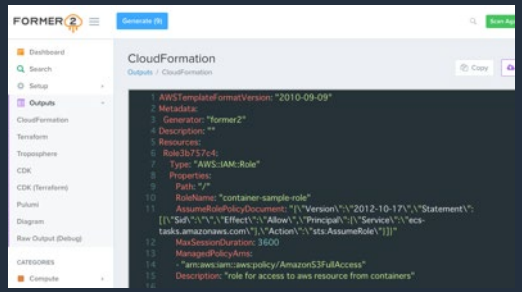
デメリット

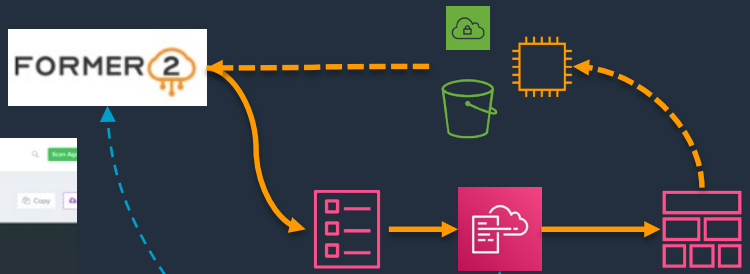
- テンプレート定義と既存リソース識別子の紐付けは引き続きマニュアル作業が必要であり、大量のリソースを一括でインポートするより小さい単位で処理することを推奨
- Importに対応していないリソースがあるため、必ずしも既存の構成を容易にStack管理化におけるわけではない

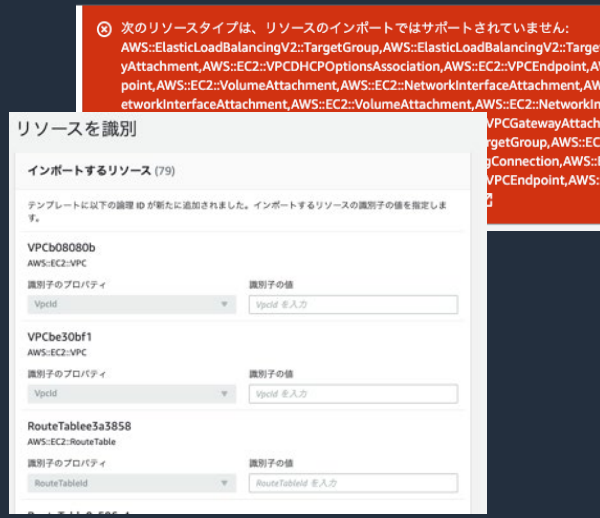
ガイダンス

- Importではなく再作成が可能ならそちらを推奨
- 一般的にサーバー系よりサーバーレス系のほうがインポートは容易（依存関係が少ない）

①Former2で既存リソースから
テンプレートを作成







②テンプレートに対応するリソースを指定して
リソースインポート

CFn Resource Import が可能なリソース

- ドキュメントを参照のこと（最新情報は英語版を）
  https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/resource-import-supported-resources.html

- 状況
  - リソースそのものはインポート可能になっているものが多いが、リソース間の関係を定義するものなどでインポートができないものが存在
  - CFn リソース定義のRegistryへの移行が進行中
    レジストリに登録されればインポートが可能になる（2020/6アナウンス）
    https://aws.amazon.com/jp/about-aws/whats-new/2020/06/aws-cloudformation-resource-import-supports-cloudformation-registry-types/

- 対応リソースの追加要望はオープンロードマップへ！
  https://github.com/aws-cloudformation/aws-cloudformation-coverage-roadmap/projects/1

よくあるユースケース

1. 各アカウントに基本設定（ベースライン）を展開する
2. 手動で作った既存リソースをCFnの管理下に入れる
3. **既存のスタックを分割する（テンプレートリファクタリング）**
4. Ansibleと組み合わせてサーバ構成を管理する
5. 1つのテンプレートから複数の環境をデプロイする

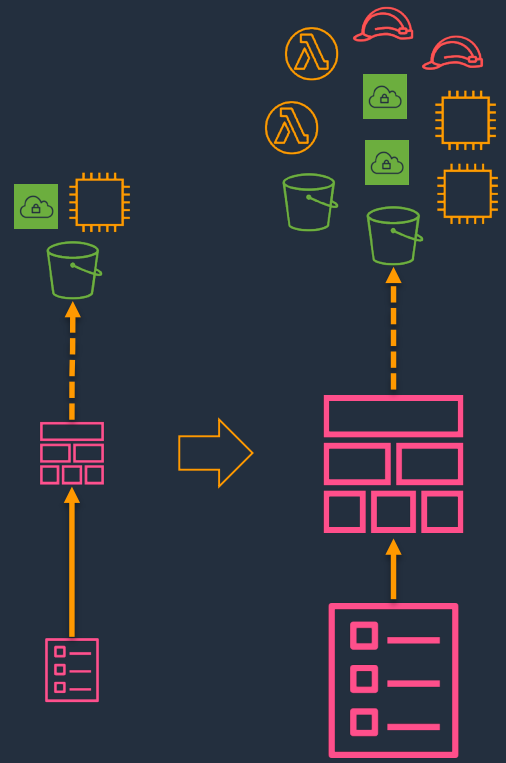
既存のスタックを分割する（テンプレートリファクタリング）

課題

- 最初に作ったスタックに関係するリソースを追加していったらスタックが肥大化してしまった
  - 変更に時間がかかる、スタック変更の影響範囲が大きい、クォータの上限に達しそうなどの問題
- 当初別のスタックで作っていたリソースを別のスタックで管理するようにしたい

解決策

- リソースを残したままスタックを削除し、改めて別のテンプレートでResource Importを行うことでスタックをリファクタリングする



既存のスタックを分割する（テンプレートリファクタリング）

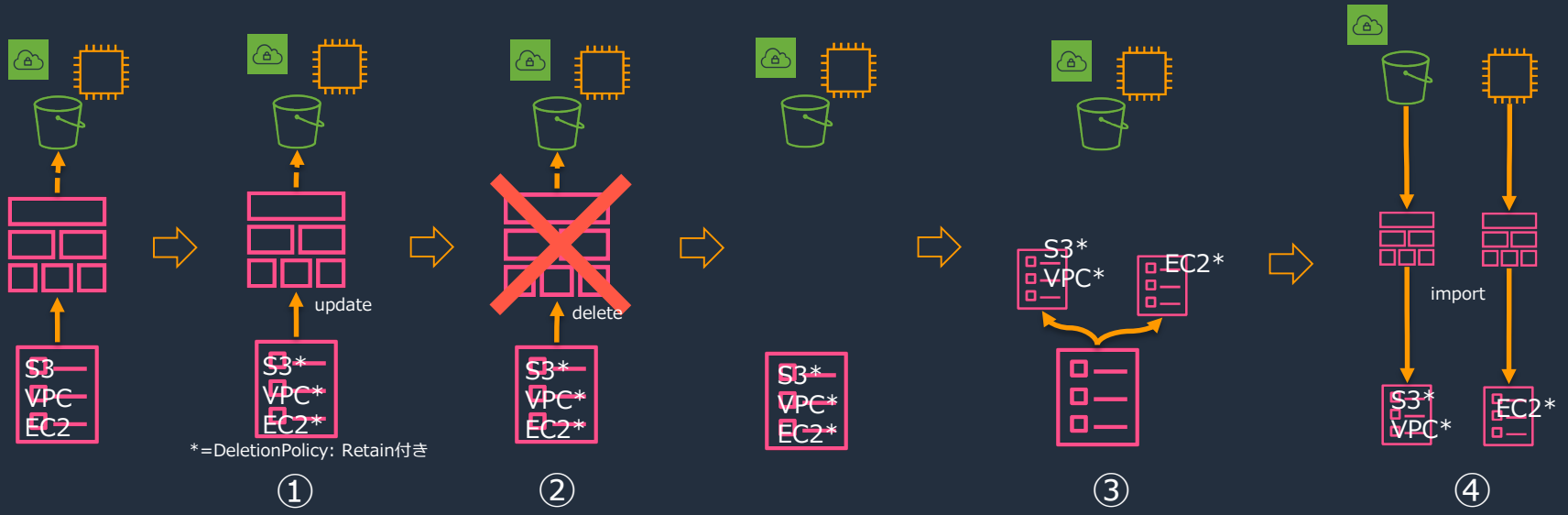
リファクタリングの方法

① テンプレートの全リソースに "DeletionPolicy: Retain" を指定してスタックを更新する（リソース自体は変更なし）

② スタックを削除する
- DeletionPolicyによりリソースは削除されることなく、スタックのみが削除される

③ 既存テンプレートを編集して希望するサイズに分ける
- 参照がスタック間をまたぐ場合はクロススタックリファレンスを使用
- ネストスタックとして構成することも可能
  https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/resource-import-nested-stacks.html

④ それぞれのスタックをインポートする

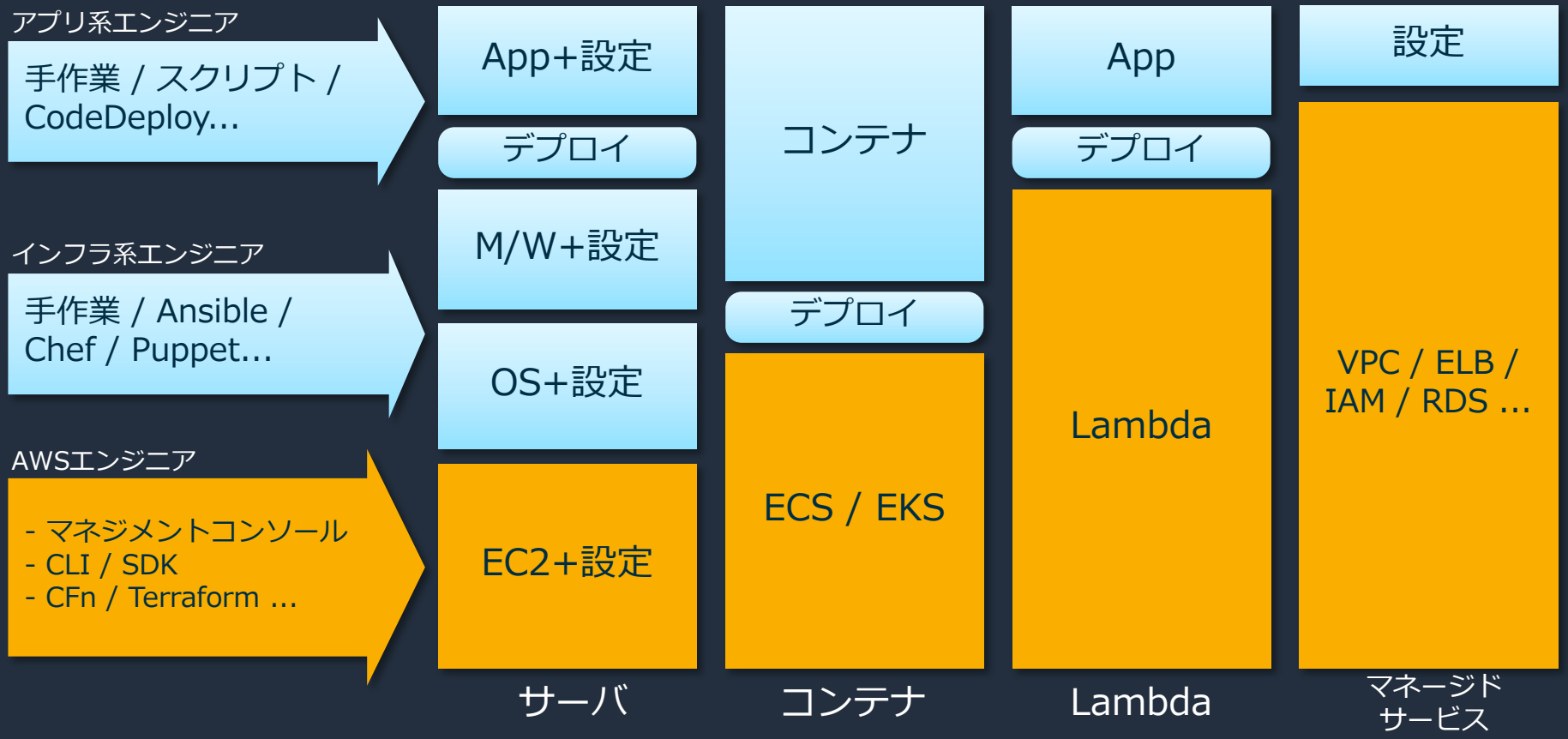


https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/refactor-stacks.html

よくあるユースケース

1. 各アカウントに基本設定（ベースライン）を展開する
2. 手動で作った既存リソースをCFnの管理下に入れる
3. 既存のスタックを分割する（テンプレートリファクタリング）
4. **Ansibleと組み合わせてサーバ構成を管理する**
5. 1つのテンプレートから複数の環境をデプロイする

リソースタイプによる構成管理方法の違い

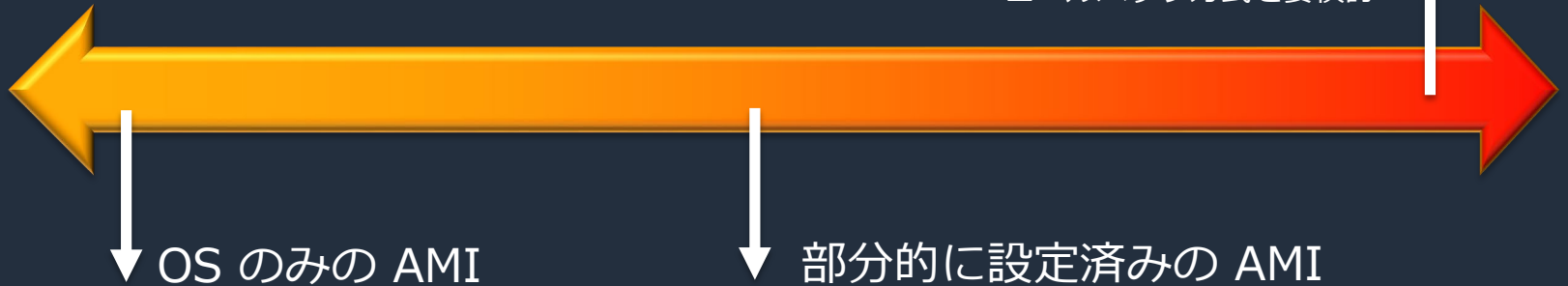


サーバコンフィグレーションのアプローチ

- サーバコンフィグレーションのAMI利用アプローチは連続的
- ニーズに合ったバランスを見出す
- AMI作成には EC2 ImageBuilderが使える
  https://aws.amazon.com/jp/blogs/news/webinar-bb-aws-ec2-image-builder-2020/

完全な AMI

- アプリケーションとすべての依存関係がAMIに導入済み
- 利用可能になるまでの時間が短い
- AMI構築時間がより長い
- 存続期間がより短い
- ロールバック方式を要検討



OS のみの AMI

- 起動後にすべての設定を実施
- 起動時の最新状態へアップグレード可能
- AMI構築時間を短縮可能
- 起動から利用可能まで時間がかかる

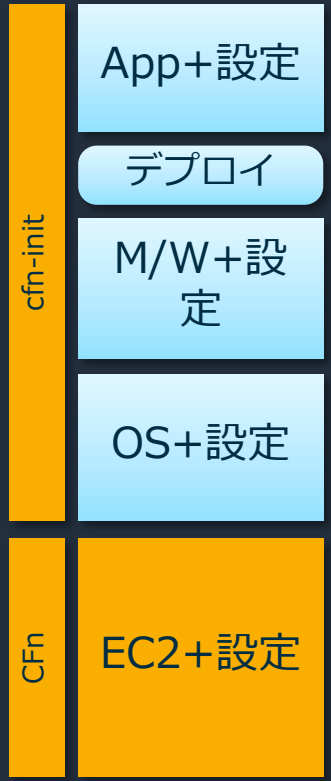
部分的に設定済みの AMI

- AMIに共通環境のみが導入済み
- 個別に変更される箇所のみ起動時に設定
- 存続期間がより長い
- ロールバックが比較的容易

CFn + cfn-init によるサーバ構成管理

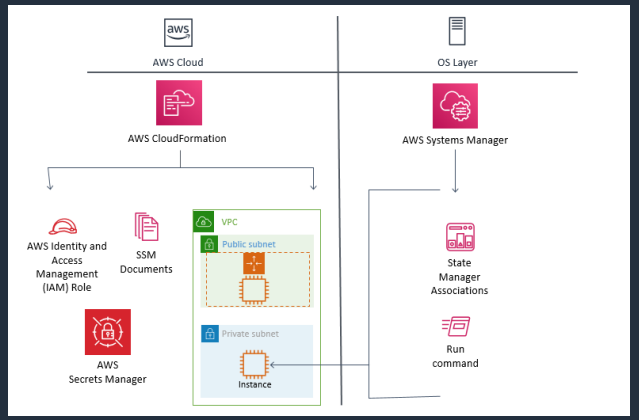
- EC2 UserData にスクリプトを記述する方式
  - 初期起動時に指定したスクリプトを実行する
  - テンプレートにサーバの起動処理をまとめて記載可能
  - 慣れ親しんだシェルスクリプトが使える
  - 従来から利用されてきており情報が豊富

- 課題
  - インスタンス初期起動時しか動作できず起動後のメンテナンスは他の仕組みが必要
  - CFnテンプレート内にコードを書くことで複雑な処理や多数のサーバの管理ではメンテナンス性に課題
  - 処理中のエラーメッセージがインスタンスローカルに保持される



CFnとAnsibleを組み合わせてサーバ構成を管理する

- CFn + SSM* StateManager (AWS-ApplyAnsiblePlaybooks)
  *SSM = SystemsManager
- CFn で SSM StateManagerを作成
  - SSMマネージドインスンタンスに対しStateManagerが定期的にAnsibleを実行
    - Playbookと設定が一致するよう継続的に維持および保証
    - 実行ログはS3に保存
- CFn で 作成したEC2をStateManagerの管理対象にする
  - 特定タグを付与 etc.



CFn + SSM StateManager + Ansible

App+設定

デプロイ

M/W+設定

OS+設定

CFn

EC2+設定

サンプル: https://bit.ly/36uMGxB

https://aws.amazon.com/jp/blogs/news/using-state-manager-over-cfn-init-in-cloudformation-and-its-benefits/

よくあるユースケース

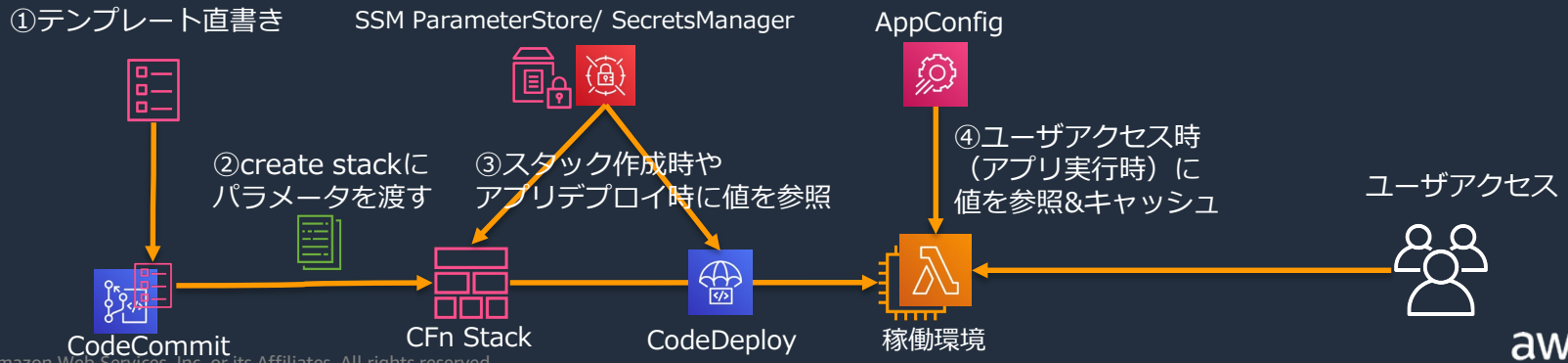
1. 各アカウントに基本設定（ベースライン）を展開する
2. 手動で作った既存リソースをCFnの管理下に入れる
3. 既存のスタックを分割する（テンプレートリファクタリング）
4. Ansibleと組み合わせてサーバ構成を管理する
5. **1つのテンプレートから複数の環境をデプロイする**

1つのテンプレートから複数の環境をデプロイする

ポイントは環境に依存するパラメータがどの時点で確定するか
- パラメータの値がいつ決まるのかによって設定箇所が変わる
- パラメータ（コンフィグレーション）自体のパイプラインが必要な場合もある

方式

① CFnテンプレートに記載	- テンプレート作成時に決定
② CFnパラメータで指定	- CFnデプロイ時に決定
③ ParameterStore/SecretsManager	- CFnやアプリのプロイ時に決定
④ AppConfig	- ユーザアクセス時に決定



よくあるユースケース まとめ

1. 各アカウントに基本設定（ベースライン）を展開する
2. 手動で作った既存リソースをCFnの管理下に入れる
3. 既存のスタックを分割する（テンプレートリファクタリング）
4. Ansibleと組み合わせてサーバ構成を管理する
5. 1つのテンプレートから複数の環境をデプロイする

Deepな機能の使いかた

Deepな機能の使いかた

1. **任意の処理を追加する（Custom Resources）**
2. スタック作成時にテンプレートを加工する（マクロ）
3. スタック作成権限とリソースの保護
4. CDKとCFnの使い分け
5. CodePipelineからCFnスタックをデプロイする
6. CFnでECSのBlue/Greenデプロイを実施する

任意の処理を追加する（Custom Resources）

- CFnの中からLambdaを実行する（SNS経由で外部サービスに処理させることも可能）
  - CFn未対応のリソースをAPI経由で操作したり、手続き的な処理を追加したりできる
  - CFnの処理に合わせてResourceTypeがCreate/Update/Deleteのいずれかにセットされたイベントが送られるので、整合性が保たれるようコードの実装が必要
  - CFn上に状態が記録されないため、できればリソースプロバイダ（後述）による実装がおすすめ
  - 実装を容易にするヘルパーツールが提供されている (crhelper)
    https://aws.amazon.com/jp/blogs/infrastructure-and-automation/aws-cloudformation-custom-resource-creation-with-python-aws-lambda-and-crhelper/

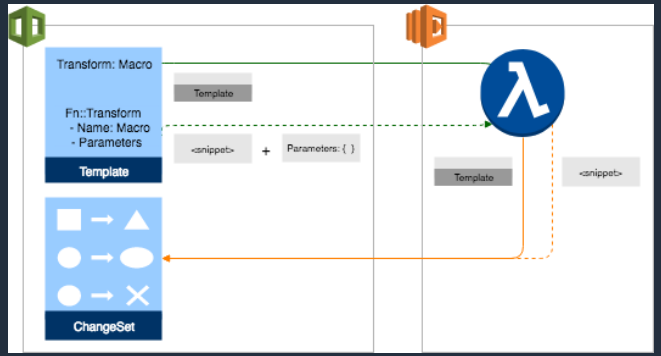
https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/template-custom-resources.html

Deepな機能の使いかた

1. 任意の処理を追加する（Custom Resources）
2. **スタック作成時にテンプレートを加工する（マクロ）**
3. スタック作成権限とリソースの保護
4. CDKとCFnの使い分け
5. CodePipelineからCFnスタックをデプロイする
6. CFnでECSのBlue/Greenデプロイを実施する

スタック作成時にテンプレートを加工する（マクロ）

- テンプレートの記述量を減らすため
  Lambdaでテンプレートを加工する
  - CFnスタック作成処理実行前に処理
  - テンプレートに Transform を指定して
    どのマクロを使用するか宣言する
  - CDKと異なりCFnの中の機能として実現



- AWSが提供するマクロ
  https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/transform-section-structure.html
  - AWS::Serverless (SAM) - サーバーレスアプリケーション用の簡素化記述
  - AWS::SecretsManager - シークレットローテーション用Lambdaを指定
  - AWS::Include - テンプレートに定型コンテンツを挿入する
  - AWS::CodeDeployBlueGreen - ECSのBlue/Greenデプロイ定義用

- 独自にマクロを定義できるのが AWS::CloudFormation::Macro リソース
  - テンプレートを加工するLambda関数をユーザ自身が開発・デプロイする

https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/template-macros.html

Deepな機能の使いかた

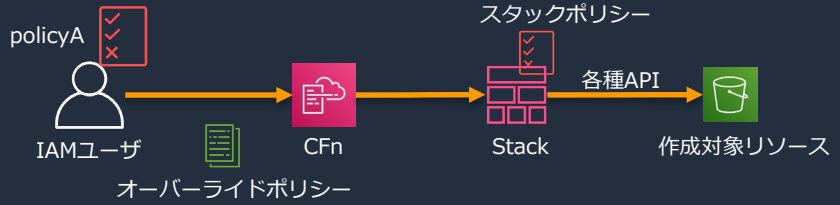
1. 任意の処理を追加する（Custom Resources）
2. スタック作成時にテンプレートを加工する（マクロ）
3. **スタック作成権限とリソースの保護**
4. CDKとCFnの使い分け
5. CodePipelineからCFnスタックをデプロイする
6. CFnでECSのBlue/Greenデプロイを実施する

スタック作成権限とリソースの保護

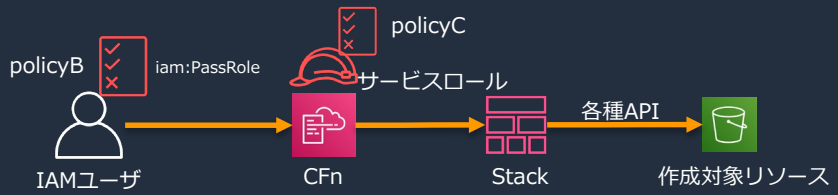
- CFn実行時の権限は create-stack APIを呼び出したユーザ/ロールの権限と同じ
  https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/using-iam-template.html
  - サービスロール指定により、別権限でStackを作成させることが可能
    - ユーザが対象ロールを iam:PassRole できる権限が必要
      https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/using-iam-servicerole.html

- リソースの保護
  - スタック削除保護　　- スタックの削除操作を禁止する
  - スタックポリシー　　- リソースレベルのアクセス許可(*)
  - リソースの保護　　- テンプレートに指定するDeletionPolicy

*) スタック操作時にオーバーライドポリシーを指定して上書き可能
https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/protect-stack-resources.html



通常のスタック作成権限の関係
- ユーザが持つpolicyAの権限でAPIを呼び出す
- =対象リソースへの権限をユーザが持っている必要がある



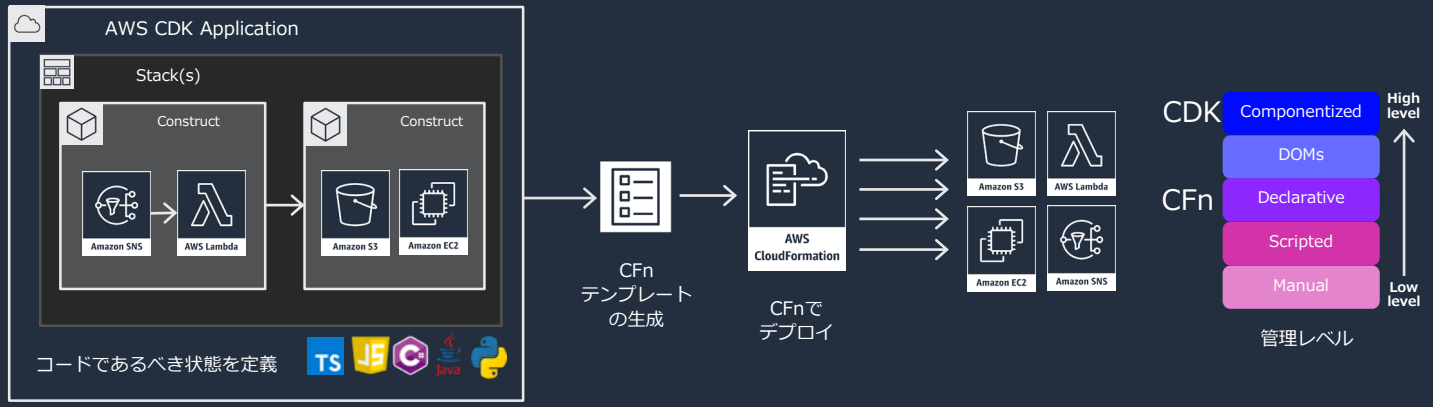
サービスロールを使ったスタック作成権限の関係
- PolicyBは iam:PassRoleでサービスロールを渡す権限が必要
- サービスロールのPolicyCは対象リソースへの権限を持っている必要がある
- サービスロールは cloudformationを信頼
  （スタックポリシーは省略）

Deepな機能の使いかた

1. 任意の処理を追加する（Custom Resources）
2. スタック作成時にテンプレートを加工する（マクロ）
3. スタック作成権限とリソースの保護
4. **CDKとCFnの使い分け**
5. CodePipelineからCFnスタックをデプロイする
6. CFnでECSのBlue/Greenデプロイを実施する

CDKとCFnの使い分け

- CDKはCFnのテンプレートエンジン（テンプレートを楽に書けるようにしてくれる）
- 既存CFnをCDKに置き換えるのは頑張りすぎない
  - 今CFnで管理できているなら無理に変えることのメリットは少ない
  - 新しい部分やメンテに困っているところから使うのがお勧め
- 過度な抽象化は禁物
  - あくまでCFnのテンプレートを楽に書けるというメリットにフォーカス
  - 美しく過度に抽象化されたコードより、CFnテンプレートが概ねイメージできるシンプルなコードがお勧め
  - 抽象化されたL2ライブラリはすべてのリソースに用意されているわけではなく、
    L1ライブラリ（CFnの定義をそのままクラスにしたライブラリ）を使う場面もある。こうなるとCFnの知識も必要。
- SDKは使わないことをお勧め
  - SDKを組み合わせれば手続きも記述できるが、構成管理という目的を考えるとAPIを直接叩く処理は混乱の元



Deepな機能の使いかた

1. 任意の処理を追加する（Custom Resources）
2. スタック作成時にテンプレートを加工する（マクロ）
3. スタック作成権限とリソースの保護
4. CDKとCFnの使い分け
5. **CodePipelineからCFnスタックをデプロイする**
6. CFnでECSのBlue/Greenデプロイを実施する

CodePipelineからCFnスタックをデプロイする

CI/CDパイプラインでCFnをデプロイする最もポピュラーな方法

一般的なフロー

① ソース - ソース一式

② ビルド - パラメータファイルやLambda等の作成

③ アーティファクト - デプロイ対象一式

④ テスト(Test)環境 - スタックを1から作成してテスト→削除

- デプロイ
- テスト完了承認
- 削除

⑤ 本番(Prod)環境 - ③を元にデプロイ準備→既存環境更新

- チェンジセット作成
- 承認
- デプロイ

③

④

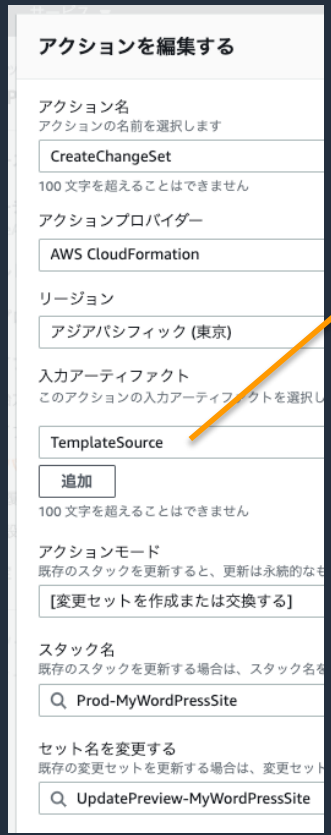
⑤

サンプル: https://bit.ly/3l7sE0a

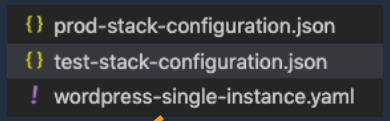
https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/continuous-delivery-codepipeline.html

CodePipelineへのCloudFormationデプロイ設定

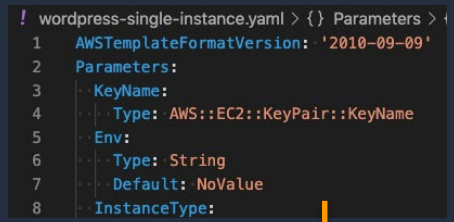
Prodデプロイの設定例



アーティファクトzip内のファイル一覧

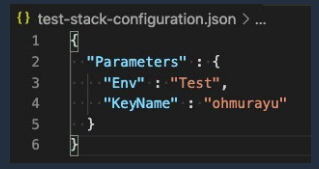


テンプレートの冒頭



- 入力アーティファクト
  - Sourceで定義したS3オブジェクト
  - 通常、ビルドは1回で同じアーティファクトでテスト→本番へデプロイ
- アクションモード
  - スタックの作成/更新
  - スタック削除
  - ChangeSetの作成/更新
  - ChangeSetの実行
  - 故障したスタックの取替
- パラメータの管理
  - テンプレートファイル
  - テンプレート設定ファイル
    - ここにCFn実行時パラメータを指定

Prod用
テンプレート設定ファイル



Test用
テンプレート設定ファイル

https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/continuous-delivery-codepipeline.html

Deepな機能の使いかた

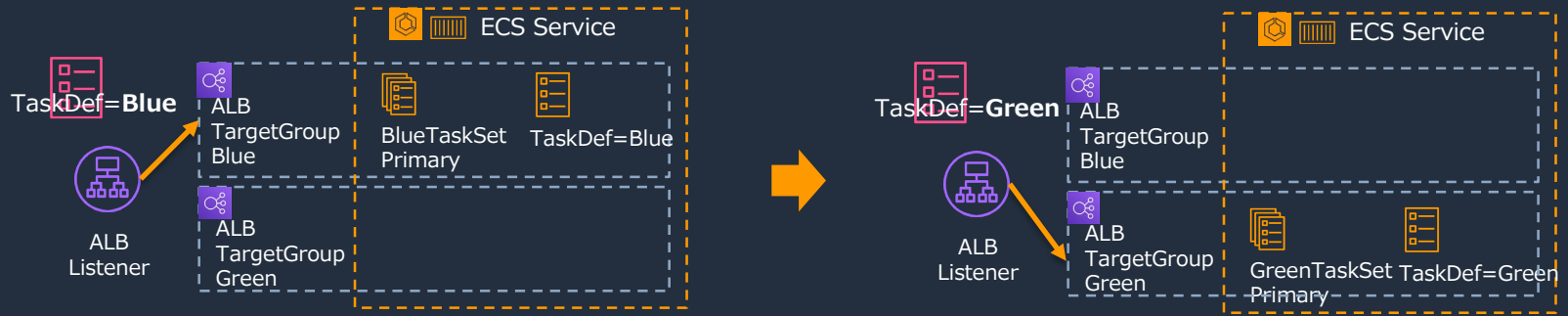
1. 任意の処理を追加する（Custom Resources）
2. スタック作成時にテンプレートを加工する（マクロ）
3. スタック作成権限とリソースの保護
4. CDKとCFnの使い分け
5. CodePipelineからCFnスタックをデプロイする
6. **CFnでECSのBlue/Greenデプロイを実施する**

CFnでECSのBlue/Greenデプロイを実施する

- CFnテンプレートでタスク定義を変更すると
  Blue/Greenでタスクの変更が行われる
  - CFnで新しい状態を定義したとき、その変更過程がBlue/Green処理になる
  - 現状はECSのTaskDefinitionとTeskSetを変更したときに動作する
  - 従来は初期設定をCFnで定義してもCodeDeployでBlue/Greenデプロイを行うと
    CFnの定義と実環境にDriftが発生していた

- 実装方法
  - ECSのExternal Deployment Controllerを使用
  - CFnでは CodeDeploy Transform を使う
  - Blue/Green用に指定されているリソースをCFn内に定義する（下図のとおり）
  - HooksセクションにBlue/Green切り替え処理の定義を記述

```
Hooks:
  CodeDeployBlueGreenHook:
    Properties:
      TrafficRoutingConfig:
        Type: TimeBasedCanary
        TimeBasedCanary:
          StepPercentage: 15
          BakeTimeMins: 1
      Applications:
        - Target:
            Type: AWS::ECS::Service
            LogicalID: ECSDemoService
          ECSAttributes:
            TaskDefinitions:
              - BlueTaskDefinition
              - GreenTaskDefinition
            TaskSets:
              - BlueTaskSet
              - GreenTaskSet
            TrafficRouting:
              ProdTrafficRoute:
                Type: AWS::ElasticLoadBalancingV2::Listener
                LogicalID: ALBListenerProdTraffic
              TargetGroups:
                - ALBTargetGroupBlue
                - ALBTargetGroupGreen
    Type: AWS::CodeDeploy::BlueGreen
```



https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/blue-green.html

Deepな機能の使いかたまとめ

1. 任意の処理を追加する（Custom Resources）
2. スタック作成時にテンプレートを加工する（マクロ）
3. スタック作成権限とリソースの保護
4. CDKとCFnの使い分け
5. CodePipelineからCFnスタックをデプロイする
6. CFnでECSのBlue/Greenデプロイを実施する

aws

リソースプロバイダを使った
独自リソースの作り方

CloudFormationリソースプロバイダ

AWS以外の独自リソースをCloudFormationで管理

- CloudFormation CLIとRPDK*でリソース定義とハンドラを開発（OSS）
- CloudFormation Registryに登録
- テンプレートに定義してCFnでデプロイ可能
- チェンジセット、リソースインポートなどCFnの各種機能が利用可能

*RPDK=Resource Provider Development Kit

AWSネイティブリソース

```
Resources:
  MyServerInstance:
    Type: “AWS::EC2::Instance”
    Properties:
      InstanceType: t2.micro
```

カスタムリソース

```
Resources:
  AMIInfo:
    Type: “Custom::AMIInfo”
    Properties:
      ServiceToken: !GetAtt AMIIf.Arn
```

リソースプロバイダ

```
Resources:
  MyResource:
    Type: “AnyCompany::Service::Resource”
    Properties:
      AnyProperty: 1000
```

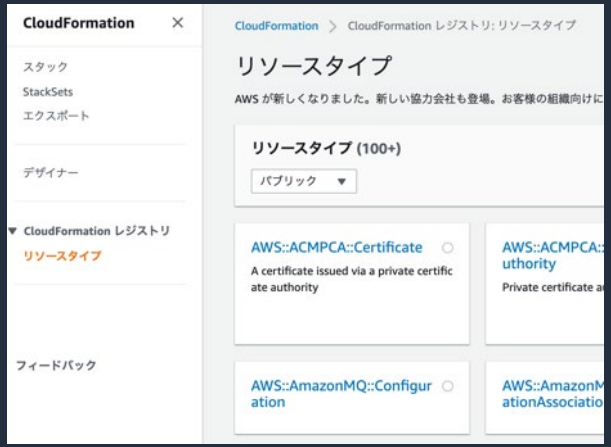

https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/registry.html

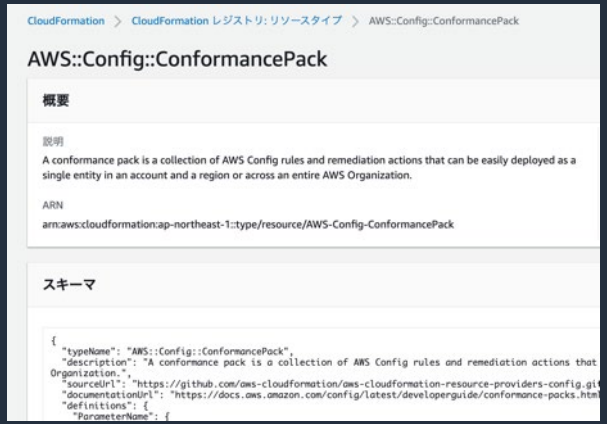
CloudFormationレジストリ

- 独自に作成したCFnリソース定義を登録する
  - 3rd PartyリソースがCFnで管理できる
  - パブリック（AWSのネイティブ）リソースも移行中
    - 既存のテンプレートやスタックは変更不要
    - 現在519（東京リージョン）

- リソースプロバイダスキーマ
  - リソースの設計書に相当するスキーマ
  - 設定可能なプロパティなどを定義する
  - マネジメントコンソールで一覧を確認可能

- CFnレジストリ登録により他の機能と統合される
  - Drift Detection が可能
  - Resource Import が可能
  - AWS Config による構成情報追跡が可能





https://aws.amazon.com/about-aws/whats-new/2020/10/aws-cloudformation-drift-detection-supports-cloudformation-registry-resource-types/
https://aws.amazon.com/about-aws/whats-new/2020/06/aws-cloudformation-resource-import-supports-cloudformation-registry-types/
https://aws.amazon.com/about-aws/whats-new/2020/06/aws-config-integrates-aws-cloudformation-registry/

リソースプロバイダ実装の流れ

実装の流れ

1. スキーマを定義する
2. ハンドラを実装する
3. ビルドする
4. テストする
5. レジストリに登録する
6. CFnで利用する

実装例（サンプルコード）

- Unicorn Maker
  - https://github.com/askulkarni2/unicorn-maker
  - モック用のREST APIを提供する crudcrud.com をCFnで操作する
  - Python(GA), go(GA), TypeScript(Non-Official) のコードがある
- Java(GA) はオフィシャルドキュメントにWalkThroughコードがある
  https://docs.aws.amazon.com/cloudformation-cli/latest/userguide/resource-type-walkthrough.html

https://docs.aws.amazon.com/cloudformation-cli/latest/userguide/what-is-cloudformation-cli.html
https://github.com/aws-cloudformation/cloudformation-cli-python-plugin

Pythonによる実装例 1

1. 事前準備
   - Cloud9環境(Ubuntu)
   - Python 3.7（RPDK設定と合わせる）
   - AWS CLI
   - AWS SAM CLI

```
$ sudo apt install python3
$ sudo update-alternatives --install /usr/bin/python python /usr/bin/python3.7 0
$ sudo update-alternatives --config python -（3.7を選択）
$ python --version
Python 3.7.5
$ sudo apt install python3.7-venv
$ brew upgrade aws-sam-cli
$ sam --version
SAM CLI, version 1.4.0
$ aws --version
aws-cli/1.18.149 Python/3.6.9 Linux/5.4.0-1025-aws botocore/1.18.8
```

2. コード、CFn CLIとプラグイン
   - unicorn-makerのclone
   - venvの設定
   - cloudformation-cli*の導入
     (バージョン整合注意)

```
$ git clone https://github.com/askulkarni2/unicorn-maker.git
$ cd unicorn-maker/python
$ python -m venv .env
$ . .env/bin/activate
$ pip install --upgrade pip
$ pip install cloudformation-cli-python-plugin
$ vi requirements.txt  -（loudformation-cli-python-libのバージョンを2.1.2へ）
$ pip install -r requirements.txt
$ cfn --version
cfn 0.1.11
$ pip list | grep form
cloudformation-cli                0.1.11
cloudformation-cli-python-lib     2.1.2
cloudformation-cli-python-plugin  2.1.1
```

Pythonによる実装例 2

スキーマを一から定義する手順
（今回はリポジトリに含まれており不要）

1. cfn init で初期ファイルを作成
   - スキーマの雛形
   - RPDK設定ファイル
     .rpdk-config

2. スキーマを作成し
   cfn validate でスキーマ検証
   - 仕様はドキュメントを参照

3. cfn generate で
   スキーマからハンドラとモデルを生成

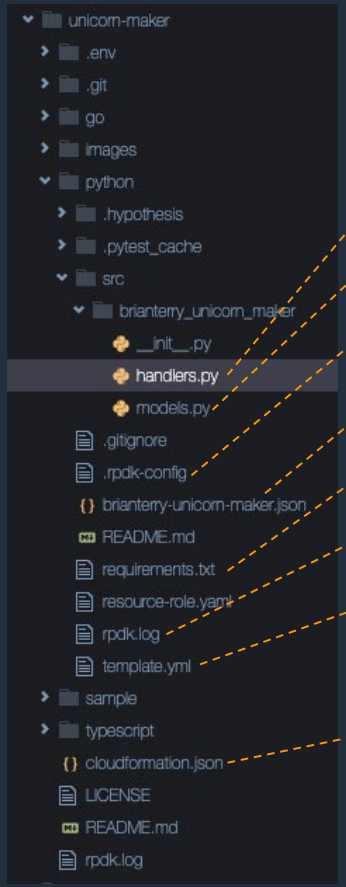
```
(.env) Admin:~/environment/unicorn-maker/sample (master) $ cfn init
Initializing new project
What's the name of your resource type?
(Organization::Service::Resource)
>> MyOrg::MyService::MyResource
Select a language for code generation:
[1] python36
[2] python37
(enter an integer):
>> 2
Use docker for platform-independent packaging (Y/n)?
This is highly recommended unless you are experienced
with cross-platform Python packaging.
>> n
Initialized a new project in /home/ubuntu/environment/unicorn-maker/sample
```

```
(.env) Admin:~/environment/unicorn-maker/sample (master) $ cat myorg-myservice-myresource.json
{
    "typeName": "MyOrg::MyService::MyResource",
    "description": "An example resource schema demonstrating some basic constructs and validation rules.",
    "sourceUrl": "https://github.com/aws-cloudformation/aws-cloudformation-rpdk.git",
    "definitions": {
        "InitechDateFormat": {
            "$comment": "Use the `definitions` block to provide shared resource property schemas",
            "type": "string",
            "format": "date-time"
        },
        "Memo": {
            "type": "object",
```

```
(.env) Admin:~/environment/unicorn-maker/sample (master) $ cfn validate
Resource schema for MyOrg::MyService::MyResource is valid
(.env) Admin:~/environment/unicorn-maker/sample (master) $ cfn generate
Generated files for MyOrg::MyService::MyResource
(.env) Admin:~/environment/unicorn-maker/sample (master) $
```


aws

Pythonによる実装例 3



ファイル一覧

- ハンドラ(Lambda)の本体
- モデルコード（自動生成）
- RPDKの設定
- スキーマ定義ファイル
- パッケージリスト
- RPDKのログ
- ローカルテスト用のSAMテンプレート（自動生成）
- 作成したリソースをデプロイするためのテスト用CFnテンプレート

crudcrud.comの自分用APIパスを指定

```
crud_crud_id = "4a564439a22e4880000000082e5a84c9"
api_endpoint = f"https://crudcrud.com/api/{crud_crud_id}/unicorns"
```

```
{
    "typeName": "Brianterry::Unicorn::Maker",
    "language": "python37",
    "runtime": "python3.7",
    "entrypoint": "brianterry_unicorn_maker.handlers.resource",
    "testEntrypoint": "brianterry_unicorn_maker.handlers.test_entrypoint",
    "settings": {
        "use_docker": false,
        "protocolVersion": "2.0.0"
    }
}
```

cloudformation-cli-python-lib==2.1.2に変更

```
"Resources": {
    "Unicorn": {
        "Type": "Brianterry::Unicorn::Maker",
        "Properties": {
            "Name": {
                "Ref": "Name"
            },
            "Color": {
                "Ref": "Color"
            }
        }
    }
},
```

Pythonによる実装例 4

1. ビルド
   - cfn submit はレジストリ登録用
   - --dry-runでパッケージングのみ実施

2. テストの準備（別ターミナル）
   - sam local start-lambda
   - ローカルテスト用エンドポイント起動

3. テストを実行
   - cfn test で Contract Testを実施
   - 各ハンドラがイベントに対して
     期待通りに処理しているか
     各シナリオを実行して確認する
   - このサンプルはまだ修正が必要
   - テストの出力、SAM localの出力、
     rpdk.logを見てデバッグ

```
$ cfn submit --dry-run
Starting pip build.
Dry run complete: /home/ubuntu/environment/unicorn-maker/python/brianterry-unicorn-maker.zip
```

```
(.env) Admin:~/environment/unicorn-maker/python (master) $ sam local start-lambda
Starting the Local Lambda Service. You can now invoke your Lambda Functions defined in your temp
2020-10-05 10:12:25  * Running on http://127.0.0.1:3001/ (Press CTRL+C to quit)
Invoking brianterry_unicorn_maker.handlers.resource (python3.7)
Skip pulling image and use local one: amazon/aws-sam-cli-emulation-image-python3.7:rapid-1.4.0.
```

```
(.env) Admin:~/environment/unicorn-maker/python (master) $ cfn test
============================================================================
platform linux -- Python 3.7.5, pytest-6.1.1, py-1.9.0, pluggy-0.13.1 -- /hom
cachedir: .pytest_cache
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/home
Test order randomisation NOT enabled. Enable with --random-order or --random-
rootdir: /home/ubuntu/environment/unicorn-maker/python, configfile: ../../../
plugins: hypothesis-5.37.0, localserver-0.5.0, random-order-1.0.4
collected 16 items

handler_create.py::contract_create_delete PASSED
handler_create.py::contract_invalid_create FAILED
handler_create.py::contract_create_duplicate SKIPPED
handler_create.py::contract_create_read_success PASSED
handler_create.py::contract_create_list_success FAILED
handler_delete.py::contract_delete_read PASSED
handler_delete.py::contract_delete_list FAILED
handler_delete.py::contract_delete_update PASSED
handler_delete.py::contract_delete_delete PASSED
handler_delete.py::contract_delete_create SKIPPED
handler_misc.py::contract_check_asserts_work PASSED
handler_read.py::contract_read_without_create FAILED
handler_update.py::contract_update_read_success PASSED
handler_update.py::contract_update_list_success FAILED
handler_update_invalid.py::contract_update_create_only_property SKIPPED
handler_update_invalid.py::contract_update_non_existent_resource FAILED
```

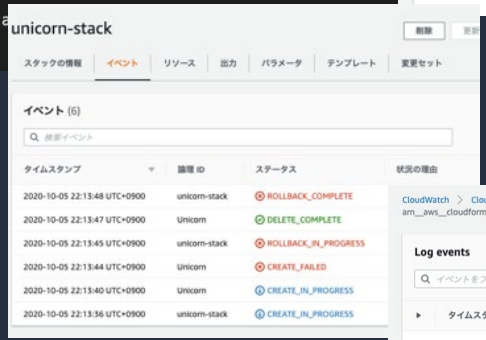
Pythonによる実装例 5

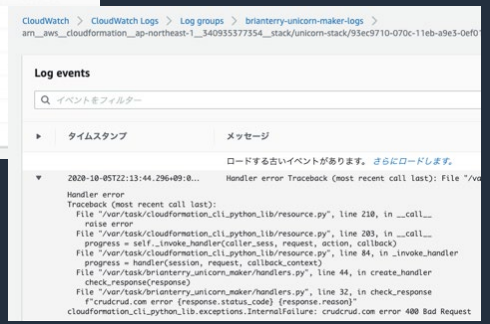
1. 登録
   - cfn submit
   - CFnレジストリに登録される

```
(.env) Admin:~/environment/unicorn-maker/python (master) $ cfn submit
Starting pip build.
Successfully submitted type. Waiting for registration with token '5b1f0692-7863-4879-8abe-3fcc04a4324a' to complete.
Registration complete.
{'ProgressStatus': 'COMPLETE', 'Description': 'Deployment is currently in DEPLOY_STAGE of status COMPLETED; ', 'Type
aker', 'TypeVersionArn': 'arn:aws:cloudformation:ap-northeast-1:3409
atusCode': 200, 'HTTPHeaders': {'x-amzn-requestid': 'e39aa80c-d8a4-4
```

2. CFnで使ってみる
   - マネコンまたはCLIでデプロイ

```
$ aws cloudformation create-stack --template-body file://../cloudformation.json --stack-name unicorn-stack2
{
    "StackId": "arn:aws:cloudformation:ap-northeast-1:340935377354:stack/unicorn-sta
}
```





- 対象リソースがデプロイされれば成功
  （このサンプルはまだ修正が必要）
- ハンドラのログはCloudWatchLogsに
  出力 (brianterry-unicorn-maker-logs)

リソースプロバイダを使った独自リソースの作り方まとめ

- AWS以外の独自リソースをCloudFormationで管理
- CFn CLI と RPDK (Resource Provider Development Kit）で開発
- Go、Python、Javaの プラグインがGA済み
- 作ったリソースは CFn レジストリへ登録して他のリソースと同様に利用

よくある質問

よくある質問

- 使いたいサービスがまだCloudFormationリソースにない
- 循環参照になってセキュリティグループが作成できない
- 依存関係が残っていて削除に失敗する
- CodeDeployでAutoScalingのBlue/Greenデプロイをすると構成が合わなくなる
- CloudFormationのQuota

使いたいサービスがまだCloudFormationリソースにない

課題

- ある機能をCFnで管理したいがそのCFnリソースが存在しないようだ

解決策

- ドキュメントのリソースリファレンス、更新履歴、CFnレジストリを再度確認（英語サイト）
  https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/aws-template-resource-type-ref.html
  https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/ReleaseHistory.html
  - 多くのサービスでは 機能リリースと同時にCloudFormationリソースも
    リリースされるようになっています

- 存在しないものはどうするか？
  - マネジメントコンソールやCLIで作成してCFnが対応したらインポートor再作成
  - 繰り返し作成が必要なら
    - CLIを使ってスクリプト化
    - CFnカスタムリソース+SDKでリソースを作る
    - リソースプロバイダを使って独自にリソースを作る

- オープンロードマップへのインプットをぜひお願いいたします！
  - https://github.com/aws-cloudformation/aws-cloudformation-coverage-roadmap/projects/1

循環参照になってセキュリティグループが作成できない

課題

- CFnテンプレートを書くと循環参照になってしまいスタックが作れない
  - 例）異なるセキュリティグループのアクセス元として
    相互のセキュリティグループを指定する場合

解決策

- SecurityGroupのリソースをそれぞれ定義して
  その上でSecurityGroupIngressのリソースを別作って紐付け

- １つのテンプレート内で表現できない場合は別スタックにするかCLIやカスタムリソースなど手続き型の処理を使う

依存関係が残っていて削除に失敗する

課題
- CFnスタックを削除するとリソースが使用中のため途中でエラーになる
  - CFnで作ったリソースを他の部分で使ってしまう場合
    - ENI周りやセキュリティグループなど
  - CFnで作ったS3バケットにデータが残っている場合

解決策
- 依存しているリソースを削除してスタックを削除する
- 消せないリソースを保持してスタックを削除する

- どこに依存関係があるかわからない場合
  - AWS Configで削除に失敗するリソースを探し、依存関係を調べる
  - CLIを使ってdescribe-*を実行して確認する

参考: VPCの依存関係を確認する方法
https://aws.amazon.com/jp/premiumsupport/knowledge-center/troubleshoot-dependency-error-delete-vpc/

CodeDeployでAutoScalingのBlue/Greenデプロイをすると構成が合わなくなる

状況

- CodeDeployでAutoScaling配下のサーバをBlue/Greenデプロイする
  - 既存のAutoScalingGroup(ASG)と同じ設定で新しいASGを作る
  - 新ASGのサーバに新しいアプリケーションをデプロイする
  - トラフィックを新ASGに送り、旧ASGを削除する

課題

- CFnで作成した旧ASGが存在しなくなる
- CFn管理外の新ASGが作られる

現状の解決策

- 通常はそのまま運用してしまう
  - CodeDeployのBlue/Greenデプロイルールがあるため何が起きているかは把握できる
- ASG内のLaunchConfigurationの変更が必要な場合は、リソースをインポートして変更する

CloudFormationのQuota

課題

- CloudFormationの Quota にかかってエラーになる

解決策

- Quotaに掛かりそうなところがないか確認する
  https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/cloudformation-limits.html

- よくかかるもの (数字は2020/10/06時点)
  - 200 - テンプレートで宣言できるリソースの最大数
  - 51KBytes - CFnに渡せるテンプレートの最大サイズ
  - 460KBytes - S3にアップロードしたテンプレートの最大サイズ
  - 200 - リージョンxアカウントで作成できる最大スタック数*
  - 100 - スタックセットの数*
  - 2000 - スタックセットから作成できるスタックの数*

*=上限緩和可能

よくある質問まとめ

- 使いたいサービスがまだCloudFormationリソースにない
- 循環参照になってセキュリティグループが作成できない
- 依存関係が残っていて削除に失敗する
- CodeDeployでAutoScalingのBlue/Greenデプロイをすると構成が合わなくなる
- CloudFormationのQuota

CFnベストプラクティスもご参照ください

https://docs.aws.amazon.com/ja_jp/AWSCloudFormation/latest/UserGuide/best-practices.html

AWS Blackbelt CloudFormation deep dive まとめ

1. よくあるユースケース別使いかた
2. Deepな機能の使いかた
3. リソースプロバイダを使った独自リソースの作りかた
4. よくある質問

Q&A

- お答えできなかったご質問については
- AWS Japan Blog 「https://aws.amazon.com/jp/blogs/news/」にて
- 後日掲載します。

AWS の日本語資料の場所「AWS 資料」で検索

https://amzn.to/JPArchive

AWS Well-Architected 個別技術相談会

毎週”W-A個別技術相談会”を実施中

- AWSのソリューションアーキテクト(SA)に対策などを相談することも可能
- 申込みはイベント告知サイトから

(https://aws.amazon.com/jp/about-aws/events/)

AWS イベント で[検索]

ご視聴ありがとうございました

AWS 公式 Webinar
https://amzn.to/JPWebina

過去資料
https://amzn.to/JPArchive