A - 2

Lambda だけじゃもったいない。サーバーレス開発の第二歩目を踏み出そう

橋本 拓弥
株式会社サーバーワークス

Takuya Hashimoto
Serverworks Co., Ltd

Speaker

■ 名前

橋本 拓弥 / Takuya Hashimoto

■ 所属

株式会社サーバーワークス
プロセスエンジニアリング部
コーポレートエンジニア

■ Favorite

- AWS Step Functions
- AWS Lambda

■ SNS

Twitter: @hassaku_63

GitHub: hassaku63

Speaker

■ 普段の仕事

開発よりの社内 SE

主に業務改善関係（バックオフィス多め）

- CRM (Salesforce)
- Python
- AWS Step Functions + Lambda
- Serverless Framework

■ SNS

Twitter: @hassaku_63

GitHub: hassaku63

対象視聴者

- サーバーレスに興味を持っていること
  - 生産性 up したい／インフラリソースの運用を手放したい
- サーバーレス & 開発の経験値が少ない（初学者向け）
  - AWS Lambda のチュートリアル程度が履修済みだとベター（デプロイツール問わず）
  - Lambda Function のデプロイ手順がイメージできるとより Good
- 簡単な Bot くらいは作れるが、その先をどう進んでいけばいいのか？
  - 開発のイメージが持てない
  - 登場人物や道具（＝各種サービス・ツール）、多すぎでは、、、？

話すこと／話さないこと

■話すこと

「覚えることが多い」問題の的を絞るための（我流プラクティスに基づく）情報提供

- 多彩な各種 AWS サービスの中で、どこから手を出していくか
- 周辺ツール（デプロイツールなど）やコーディングに関係する Tips を知る

■ 話さないこと

- AWS Lambda 以外の AWS サービスを幅広く紹介

Agenda

- はじめに（前提の共有）
- 学習ステップの一例
- AWS CloudFormation と Serverless Framework
- ソースコードのモジュール構成
- Logging
- Test
- よくある（ありそうな）疑問点について

前提事項

■ 開発環境

- MacOS + VSCode
- Python (>= 3.6)
- Serverless Framework
- AWS Step Functions + AWS Lambda を用いた構成がメイン

■ 私の経験値（当時）

- AWS … 認定試験の出題範囲程度の理解はある
- Python … 普通に読み書きはできる程度
- Serverless Framework … ほぼ未経験、AWS CloudFormation は多少

学習ステップの一例

学習ステップ

1. 主要概念を押さえる
    1. Serverless Framework + 各種 AWS サービス
    2. Python + 関連パッケージ (boto3, unittest など)
    3. イベント駆動
2. やってみる
3. それっぽい「書き方」を覚える
4. 応用してみる

学習ステップ

1. 主要概念を押さえる
    1. Serverless Framework + 各種 AWS サービス
    2. Python + 関連パッケージ (boto3, unittest など)
    3. イベント駆動なアーキテクチャ
2. やってみる
3. それっぽい「書き方」を仕入れる
4. 応用してみる

主要概念

押さえておくべき優先順位は、

1. AWS Lambda + Serverless Framework (AWS CloudFormation)
2. AWS Step Functions
3. Amazon Simple Storage Service (Amazon S3)
4. Amazon DynamoDB
5. Amazon Simple Queue Service (Amazon SQS)
   & Amazon Simple Notification Service (Amazon SNS)
6. Amazon Kinesis

主要概念

押さえておくべき優先順位は、

1. AWS Lambda & Serverless Framework (AWS CloudFormation)
2. AWS Step Functions
3. Amazon Simple Storage Service (Amazon S3)
4. Amazon DynamoDB
5. Amazon Simple Queue Service (Amazon SQS)
   & Amazon Simple Notification Service (Amazon SNS)
6. Amazon Kinesis

主要概念

トピック	補足コメント
AWS Lambda & Serverless Framework	何をするにも使うので最初に押さえる ・チュートリアル + 簡単な Bot アプリの作成などで開発の感覚を掴めば OK ・テンプレートの書き方に関するお作法は、早めに “型” を知っておくと捗る（後述）
AWS Step Functions	ちょっと複雑なことをやりたくなったときに使える、非常に頼れる手札 1つのロジックを複数のサブルーチンに分割するのに近い感覚で使う 以下のことを押さえておけば、実戦投入して十分活用できる（はず） ・Task, Choice が使える程度 ・入出力の “Path” の概念を、動かしながら把握する
Amazon S3	Get/Put を SDK から実行できる程度に知っていれば OK （使用機会も多く、馴染みやすいのでさほど苦労しないはず）

イベント駆動なアーキテクチャ

個人的にざっくり解釈すると；

→　イベントハンドラとその仲介役となるサービスを組み合わせて、
　　「イベント」を転がしていくピタゴラスイッチを構築すること

（ ※サーバーレス界隈で見聞きする「イベント駆動」という言葉に対する個人的な解釈）

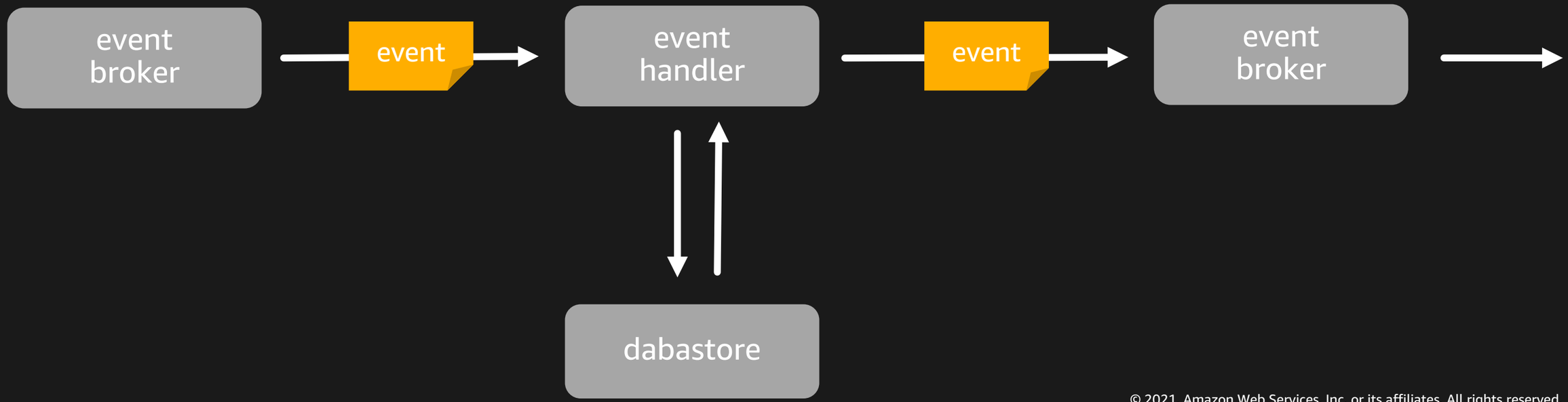
登場人物	説明
イベントソースと 仲介役（ブローカー）	イベント（≒データ）の発生源あるいは仲介役となるサービス郡 Amazon SQS, Amazon Amazon SNS, Amazon Kinesis, Amazon EventBridge, AWS Step Functions など
イベントハンドラ	イベントを受け取って処理するロジック部分 必要に応じて後続（ブローカー）に処理結果を投げる AWS Lambda, AWS Step Functions など

イベント駆動なアーキテクチャ

ざっくり思想と登場人物を押さえておけば OK

以下の3つを組み合わせて、「イベント」を転がすピタゴラスイッチを構築する

（ ※個人の解釈です）



イベント駆動なアーキテクチャ

最初からあまり難しく考えることはないが、気持ち程度に意識しておくとよい

- ハンドラの実装はできるだけ受け取る or 投げる相手を意識しない
    - 特定の誰かから受け取ることを前提とした実装をできるだけ避ける
- ポーリングのような同期的な設計・実装は基本的に避ける
    - ハンドラの間にブローカーを挟んで疎結合にすることを検討する
        - 前工程の結果を Amazon S3 への Object Put イベント、あるいは Amazon SQS/Amazon SNS のようなサービスを挟めて受け取れないか？
- 個々のハンドラはステートレスな実装であること

やってみる

技術的な不安（不確実性）が少ないものがよい

社内システムで、業務効率化のネタがおすすめ

- 複数の社内システム間をつなぎ合わせて何かしら自動化するもの
  - e.g) プロジェクト管理等の社内システムとチャットの連携
  - バックエンドのみ、フロントエンド不要
- 自前で DB 設計しないで済むもの（必要なデータは社内システムが全部持っている）
- 障害等のよるトラブル・サービス停止によるビジネスインパクトが少ないもの

AWS CloudFormation と Serverless Framework

Serverless Framework

AWS CloudFormation によるデプロイをより手軽に扱えるように
ラップした OSS の IaC ツール

※「Serverlees Framework の使い方まとめ」で検索

Serverless Framework

AWS CloudFormation と比べて嬉しいこと;

- ツール単体でのテンプレートの記述力が高い
  - テンプレート変数を利用可能（特に CLI オプションとの連携が楽）
  - .env からパラメータを読み出すことができる
- package -> deploy の流れはコマンド一発で OK
- 有用なプラグインが多いためショートカットがしやすい

おすすめプラグイン

■ Serverless Step Functions

- AWS Step Functions を構築する場合のデファクト
- AWS CloudFormation の記法ほぼそのままなのでクセが少ない

■ Serverless Python Requirements

- virtualenv, pyenv, Pipenv, Poetory などを利用してインストールした依存関係を自動的にデプロイパッケージにバンドルできる
- docker 環境でバンドルを実行することも可能（Mac や Windows の開発環境で OS ネイティブの so に依存するパッケージを利用する場合に利用する）
- バンドルパッケージのスリム化機能

テンプレートの書き方

テンプレートの書き方

■ リソースの命名規則

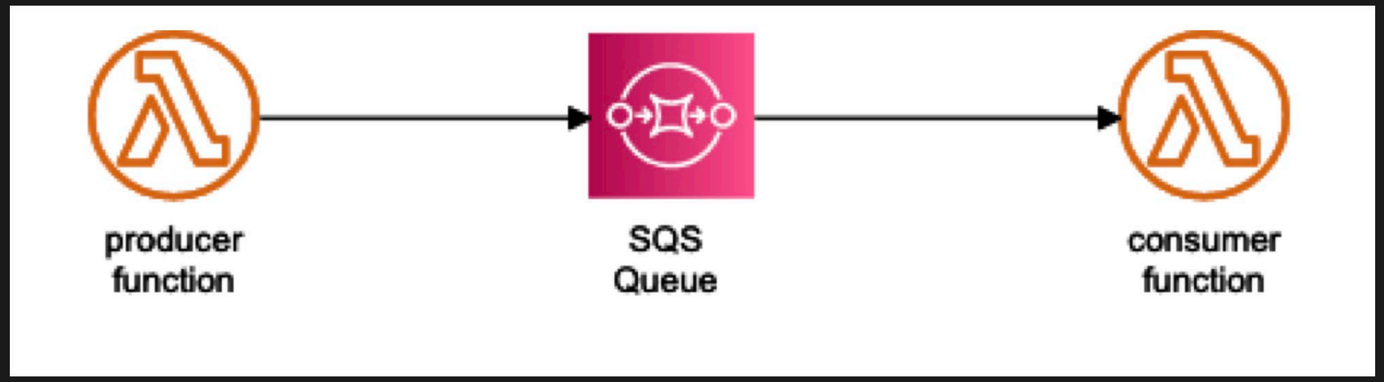
- サービスやデプロイ環境の単位でマルチアカウント戦略を徹底しているなら雑でもOK
  - 命名規則は意図しない名称被りの防止や画面上での識別性・視認性性向上のため
- 1つの AWS アカウント内で複数のサービス／デプロイ環境が混在する状況であれば、
  ある程度人間向けの可読な表現、かつ同居人との一意性が確保できる名前付けを考慮する
  - e.g) *${Service}-${DeployEnv}-${ResouceName}<-${Region}>*
- AWS CloudFormation の Parameter よりは、Serverless Framework の変数機構を利用する
  - 変数の値は .env や CLI オプションから供給

テンプレートの書き方 - example

環境変数などを絡めてリソース名称や Function に渡すパラメータなどを定義するサンプル

> https://github.com/hassaku63/sls-env-example

- .env で宣言した環境変数を Serverless Framework に読み取らせて、テンプレート内で扱う
- settings モジュールを導入し、Function ハンドラが参照する設定値を集約する




```
.
├── app
|   ├── __init__.py
|   ├── functions
|   └── libs
|       └── __init__.py
└── libs
    ├── __init__.py
    └── settings.py
```

テンプレートの書き方 - example

serverless.yml

```
resources:
  Resources:
    MyQueue:
      Type: AWS::SQS::Queue
      Properties:
        QueueName: ${self:provider.environment.MY_QUEUE_NAME}
        # notice: VisibilityTimeout must be greeter than lambda
        VisibilityTimeout: 45
```

キューの名前を環境変数から与える

テンプレートの書き方 - example

serverless.yml

```yaml
provider:
  name: aws
  runtime: python3.8
  profile: ${opt:profile, default}
  stage: ${opt:stage, dev}
  region: ${opt:region, us-east-1}
  timeout: 30
  environment:
    # To tell lambda function to AWS AccountId by use pseudo-parameters plugin.
    # AccountId is used to construct SQS Queue URL
    AWS_ACCOUNT_ID: '#{AWS::AccountId}'
    # MY_QUEUE_NAME is used to construct Queue URL in lambda.
    # And also convenient to reference resource ARN in IAM Policy statement
    MY_QUEUE_NAME: ${self:service}-my-queue-${self:provider.stage}
```

テンプレートの書き方 - example

serverless.yml

```yaml
provider:
  name: aws
  runtime: python3.8
  profile: ${opt:profile, default}
  stage: ${opt:stage, dev}
  region: ${opt:region, us-east-1}
  timeout: 30
  environment:
    # To tell lambda function to AWS Accou
    # AccountId is used to construct SQS Q
    AWS_ACCOUNT_ID: '#{AWS::AccountId}'
    # MY_QUEUE_NAME is used to construct Queue URL
    # And also convenient to reference resource ARN in IAM Policy statement
    MY_QUEUE_NAME: ${self:service}-my-queue-${self:provider.stage}
```

リソース識別子となる情報を環境変数で宣言
（Lambda の実行環境から参照可能）

service や stage といった変数を使うことで
他のスタック／ステージとのリソース名衝突を避ける

テンプレートの書き方 - example

libs/settings.py

- serverless.yml で定義した環境変数の読み出しを settings で行う
- コード上で必要なリソース参照名（この例では SQS）を settings に集約
  - ハンドラロジックからは settings を介して設定値を読み込む
- Feature toggle 的な機能を制御するフラグもここにまとめたり
  - e.g) 本番環境以外では Mock実装に差し替えておきたい機能

```
# SQS
MY_QUEUE_NAME = os.environ.get('MY_QUEUE_NAME', '')
```

テンプレートを書くときに意識していること

記述が少々煩雑になっても、循環参照が起きづらいように書く

- デプロイに失敗した CloudFormation テンプレートのデバックが一番つらい
    - 循環参照はその中でも割とやりがち、かつ解消が面倒
    - IAM ポリシーの記述などでリソース参照が絡むと起こりがち
- 基本的に CloudFormation の参照関数 (Ref, GetAtt) は濫用しない
- 各リソースのプロパティ間であちこちパラメータ参照が飛んでいる状況が（循環参照的に）好ましくない
    - リソースの名前は環境変数で宣言しておく
    - テンプレート上で ARN への参照が必要な場合は、Ref, GetAtt で参照を張るよりは
      リソース名の文字列と AWS Cloudformation の「疑似変数」の組み合わせで文字列を構成する

参考

https://gist.github.com/hassaku63/3447a4381e8001367093ddc61c5ae13f

- 最新版の Serverless Framework に対応したサンプル
  - 環境変数の取り回し
  - CloudFormation 疑似パラメータの使用
  - “ステージ” をリソースの命名規則に利用する

ソースコードのモジュール構成

モジュール構成

モジュール構成の思想は（おそらく）
一般的な Web アプリとそう変わらない

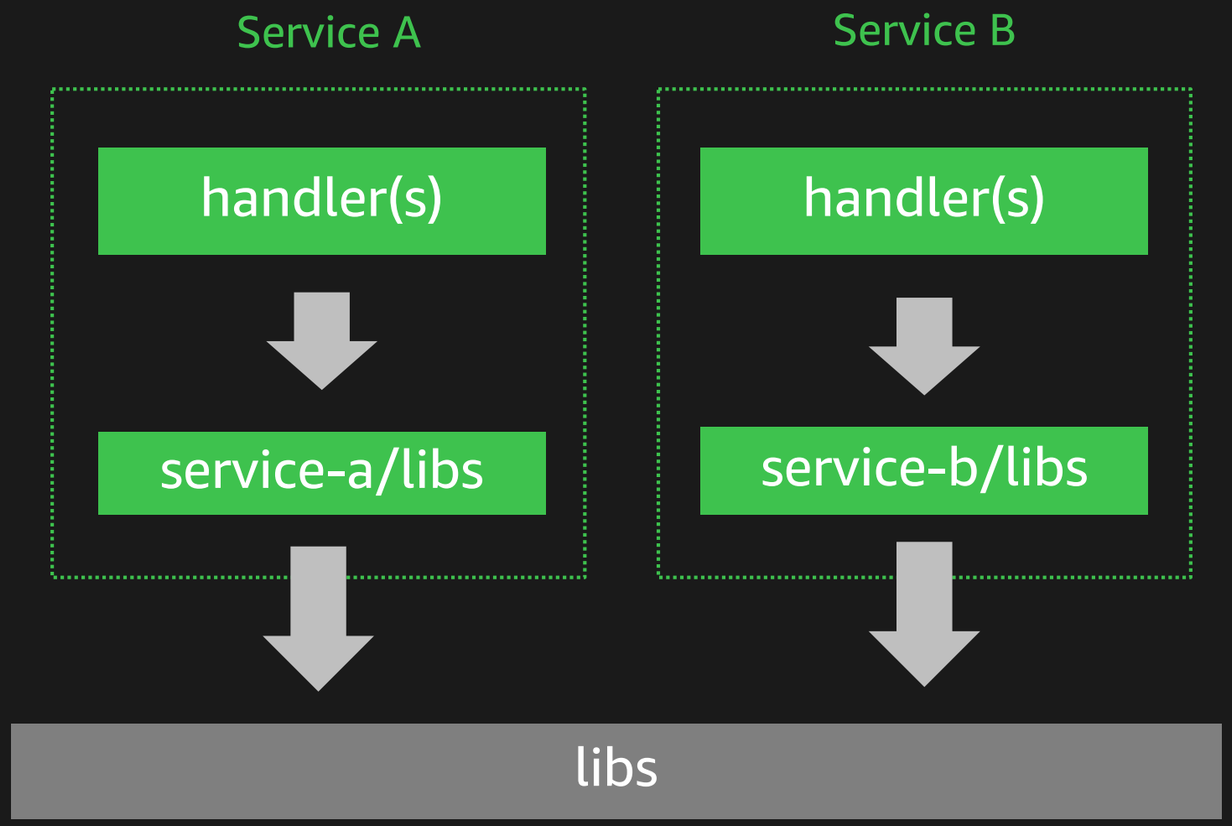
- サービス単位でトップ階層を切る
- サービス横断で共通する低レベルなラッパー層を
  置く場所として libs/** を使用
  - AWS サービスやサードパーティなど
    外部連携にあたるものをラップしておく
- ハンドラに近いコードほど抽象度の高い
  インタフェースになるようにする

```
➜  sample-app tree -d
.
├── libs
├── service_a
│   ├── functions
│   │   ├── sfn
│   │   └── sqs
│   └── libs
└── service_b
    ├── functions
    │   ├── api
    │   └── cwevents
    └── libs
```

モジュール構成

モジュール構成の思想は（おそらく）
一般的な Web アプリとそう変わらない

- サービス単位でトップ階層を切る
- サービス横断で共通する低レベルなラッパー層を
  置く場所として libs/** を使用
  - AWS サービスやサードパーティなど
    外部連携にあたるものをラップしておく
- ハンドラに近いコードほど抽象度の高い
  インタフェースになるようにする



Logging

JSON 形式でロギングを行う

簡易的にでも JSON 形式を出すロガーを作っておく

- CloudWatch Logs Insights を利用して
  フィールド名指定のクエリが可能になる
- Python ユーザーかつ新規プロジェクトであれば
  Lambda Powertools Python [1] が推奨

```python
class JsonFormatter(logging.Formatter):
    def format(self, record: logging.LogRecord):
        rdict = vars(record)
        if isinstance(rdict['msg'], Mock):
            rdict['msg'] = repr(rdict['msg'])
        return json.dumps(rdict)


def get_logger(name, level=None) -> logging.Logger:
    if level is None:
        level = LOG_LEVEL
    log = logging.getLogger(name)
    log.setLevel(level)
    handler = logging.StreamHandler()
    handler.setFormatter(JsonFormatter())
    log.addHandler(handler)
    return log
```

Example: 最小限の JSON ロガーのサンプル実装

[1] Lambda Powertools Python
https://awslabs.github.io/aws-lambda-powertools-python/latest/

モジュール構成

モジュール構成の思想は（おそらく）
一般的な Web アプリとそう変わらない

- サービス単位でトップ階層を切る
- サービス横断で共通する低レベルなラッパー層を
  置く場所として libs/** を使用
  - AWS サービスやサードパーティなど
    外部連携にあたるものをラップしておく
- ハンドラに近いコードほど抽象度の高い
  インタフェースになるようにする

```
.
├── cli
├── issuer      … サービスのディレクトリ
|   ├── functions   … Lambda ハンドラ置き場
|   |   └── sfn     … Step Functionsで使うハンドラ
|   ├── libs    … issuer サービスで使う汎用モジュール群
|   |   └── templates
|   └── templates   … テンプレートファイル置き場
├── libs    … 全サービスが共通利用する汎用モジュール群
|   ├── aws
|   ├── settings.py
|   ├── sfdc
|   └── stage.py
└── tests
```

Test

Best Practice ?? 🤔

- 王道的なものは（知る限り）まだない
  - 構成図レベルでバリエーションが幅広いため、画一的な議論は難しい？
- 基本はユニットテストをしっかりやること
  - テストピラミッドの足場から固めていくこと
  - ロジックの具体的な実装はハンドラ関数から隠蔽して、ロジックをテストする
  - モジュール構成の工夫をしつつ機能抽象度で雑にレイヤ分割
  - DI ライクなアプローチを取り入れる（次のスライトで紹介）

Unit Test

何らかの Client を扱う機能は、Client を外部注入可能なインタフェースを用意しておく

```python
# テスト対象のコード
def target_function(arg: int):
    """内部用のインタフェースに渡すだけで実質的仕事は client の初期化のみ
    """
    return _target_function(arg, get_foo_client())


def _target_function(arg: int, foo_client):
    """client を注入可能なモジュール内部用のインタフェース

    unittest ではこちらをテストする
    """
    param = {
        # ...
    }
    resp = foo_client.some_function(**param)

    return resp
```

```python
# テストコード
from unittest import TestCase
from unittest.mock import MagicMock


class TestFoo(TestCase):
    def test_target_function(self):
        # set mock
        mock_client = MagicMock(name='mock_foo_client')
        mock_client.some_function.return_value = {
            # ...
        }

        self.assertEqual(
            _target_function('test-input', mock_client),
            {},  # expect value
        )
```

Integration / E2E

現在の実業務では、E2E のみテストケースを切って手作業で対応

■ Integration Test

- 粒度が細かすぎると費用対効果が薄い上に開発スピードが損なわれる
  - Lambda ハンドラ1つ取っても permission の兼ね合いなど外部影響で失敗要素がある
  - サーバーレス開発だと構成レベルから変更するケースが普通にある
- AWS 以外のサービス（主に SaaS 製品）とどう付き合うか？
  - サービスの仕様や予算・データ移行の手間などなど、テスト用途に適した環境が用意できないことも
  - 実環境を（支障ない範囲で）叩くか、mock response を差し込めるような実装に変え対応するか

■ E2E Test

- テスト専用の AWS アカウントを作って実際にデプロイしてしまうのが最もシンプル
- AWS 以外のサービスと（略）

よくある（ありそうな）疑問点

コンテナベースの実行環境について

SAM Local や localstack などのコンテナの実行環境とどのように付き合うか？

- 最初から進んで使うようなものではない
  - ロジックに対するユニットテストを充実させる方がよほど重要
  - Integration レベルのテストにはおそらく合わない
- ローカルの開発環境で素早く検証を回す目的では使えそう
  - デプロイを待っている時間はなんやかんやで長い（集中力が切れる）
- 自動テストの枠組みに取り込む意義はあまり感じない
  - 実際にデプロイした方がシンプルな上に確実
  - そのコンテナ環境の構成もメンテし続けるんですか...？

ピタゴラスイッチの構成理解がつらい

実際そう

- とはいえ普通に組み上げたら必然そうなる
- 結局はディレクトリ構成の設計やドキュメンテーションで補っていく
    - 「この Lambda ハンドラはどのような役割で、どう呼ばれるか？」が見えると Good

シークレットの扱い

API Key やクライアント証明書などの扱い

- 最初は dotenv と Lambda Function の環境変数で OK
- 慣れてきたら Parameter Store のパラメータに移行する

AWS Step Functions はそんなに重要？

個人的には yes.

- AWS Step Functions は「デプロイでき、視覚化もサポートするフローチャートサービス」
  - マネジメントコンソールでフローチャートを視覚化
  - フローチャートを読むことで、そのサービスが「やりたいこと」の大枠を読解できる
  - Amazon SNS への pushlish など、No Code でサービス連携できる選択肢がある
- 1本の Lambda Function に多くの仕事をさせたくない
  - エラー発生時の原因特定が難しくなる
  - Invocation timeout のハンドリングが難しくなる
  - サブルーチン化と同じ感覚で積極的に分割するくらいでちょうどよい
- ログを有効化しておくことで、障害調査の情報源が増える
  - 「実行」単位でエラーを捉えることができる
  - コンソール上ではフローチャート上のどこで失敗したかが一目瞭然
- 失敗したジョブを手動でリカバリする場合のリトライが非常に簡単

おわりに

脱初心者してきた時期の振り返り

サーバーレス開発はエントリーの障壁が高い（ように感じる）

- 覚えるべき「最低限」の水準が高く、実態をイメージしづらい
  - アプリケーション開発のスキル
  - 各種 AWS サービスの知識
  - IaC ツールの知識
  - サーバーレス的な作法（e.g: AWS Lambda で長時間の処理を走らせてはいけない）

脱初心者してきた時期の振り返り

結局（我々初心者は）どうすれば前に進めるのか？

- 学び始めのうちは「目が散る」ことを極力避けて学習すべき
  - 断片的な情報収集は、初心者には時として害になる（手が止まってしまう）
  - 複数の不確実なものを、一度に相手にしようとしないこと
- かと言って、体系的なコンテンツや事例等の情報源は
  一般的な（サーバーレスではない）アプリケーション開発ほど豊富でない

まとめ

- 雑多に Tips 的なものを紹介してみた
  - イメージを持つきっかけになれば幸いです
- 本番環境できっちり運用していくために「覚えた方がよいこと」はまだまだある
  - 無数の「した**ほうがいい**こと」は適度に無視して、1つ1つやっていきましょう
  - 私なりの道筋は提案してみたつもりです。 Enjoy Serverless.

Thank you!