公開API説明資料

AWS で CI/CD パイプラインを作成する方法

目次

公開APIの特徴

参考:
APIデザイニングパターン(JJ Geewax著, 松田晃一訳)

公開APIは「公開性」、「硬直性」を意識した開発が重要である

公開性

- 悪意のある攻撃に晒されている
- パブリックなユーザーが存在する

硬直性

- 柔軟性が低い
- 変更の難易度が高い

公開API開発の際の注意点

参考:
APIデザイニングパターン(JJ Geewax著, 松田晃一訳)

公開APIを開発する際は「セキュリティ」、「スロットリング」、「互換性」に注意する

公開性

- 悪意のある攻撃に晒されている
- パブリックなユーザーが存在する

硬直性

- 柔軟性が低い
- 変更の難易度が高い

- **セキュリティへの意識(API Key/Token)**
- **頻度・量の制限(スロットリング)**
- **互換性の意識(バージョン管理)**

- 既存のパターンに則った正しい設計をはじめから行う

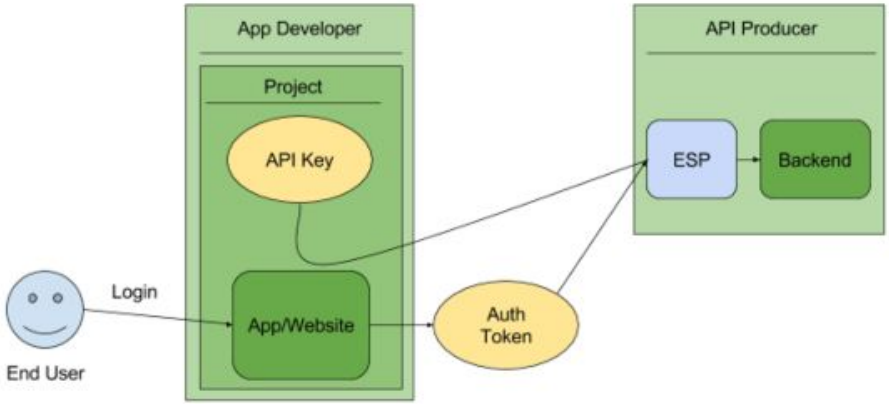
API Key認証とは

参考:
https://architecting.hateblo.jp/entry/2020/03/27/033758#

事前生成した強度の高い認証鍵をクライアントがRequest内に含めて送る認証方式全般を指す

- 様々なサービスで簡易的な認証手段として広く利用されている
  - Amazonでは認証鍵をX-API-KEYヘッダに含める
  - Google Cloudでは認証鍵をkeyクエリ文字列に含める
- 身近なものだと例えば**AWS API Gateway**を使って実装可能

Google Cloud Platformでの使用例



出典：
https://cloud.google.com/endpoints/docs/openapi/when-why-api-key?hl=ja

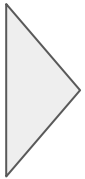
API Key認証使用上の注意

シンプルだが安全性は低いため、機能やRequest元制限、他の認証/認可と組み合わせて使用する

特徴
- DBで管理され、比較的管理期間も長い

使用するメリット
- シンプルな認証が実現可能
- （多少の）セキュリティの向上が見込める

使用するデメリット
- 安全性は低い
- キーを盗聴されて盗まれる可能性がある



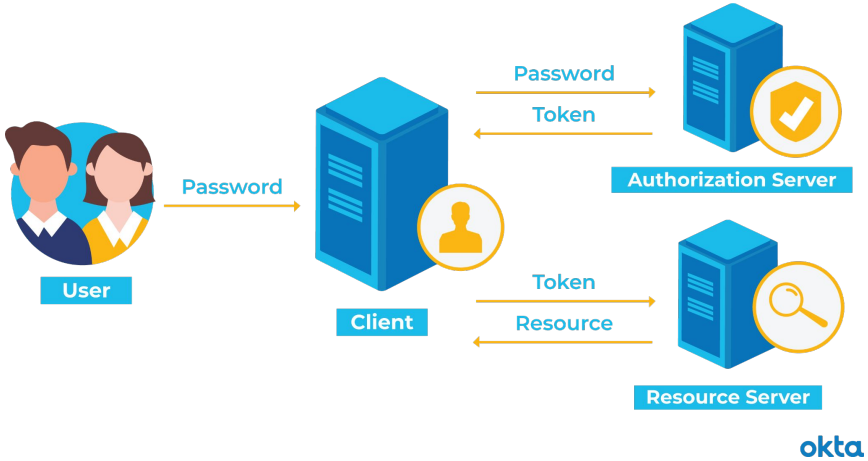
Request送信元に制限を設けることが一般的
- IPアドレス
- Refererヘッダ情報
- アプリ種別（Androidアプリ、iOSアプリ等）

APIキー認証ユーザには簡単な機能のみ開放し、他の機能はOAuth等強い認証を使用している事業者が多い

アクセストークン認証とは

認証鍵が動的に生成されて時間とともに変化する方式で、APIキー認証より安全性が高い

- トークン認証のプロセス
  1. トークンの要求
  2. ユーザーの検証
  3. トークンの発行
  4. トークンの保持



出典：
https://www.okta.com/jp/identity-101/what-is-token-based-authentication/

頻度・量の制限（スロットリング）とは

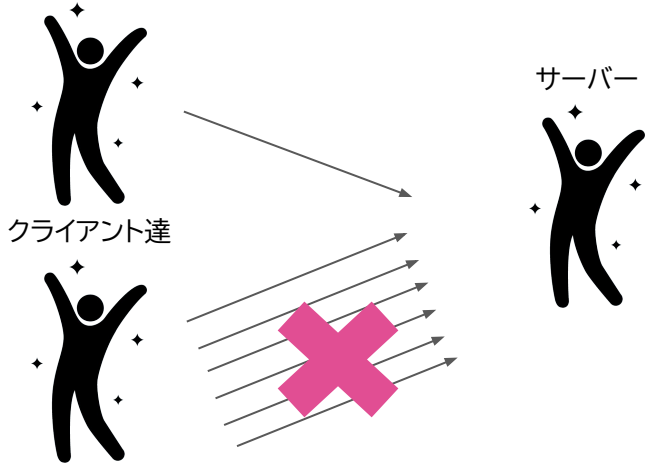
参考:
https://future-architect.github.io/articles/20200121/
https://techblog.zozo.com/entry/zozotown-api-gateway-throttling

何らかのリソースに対して使用量の上限を設定し、上限を超えるものについてはその使用を制限するような処理

リクエストに対するスロットリング

- 一定時間内に受信可能なリクエスト数を制限する
  - 制限を上回るリクエストがなされた際には受信を拒否しエラーコードを返却する
- 一定時間経過後、これらのリクエスト制限は解除される

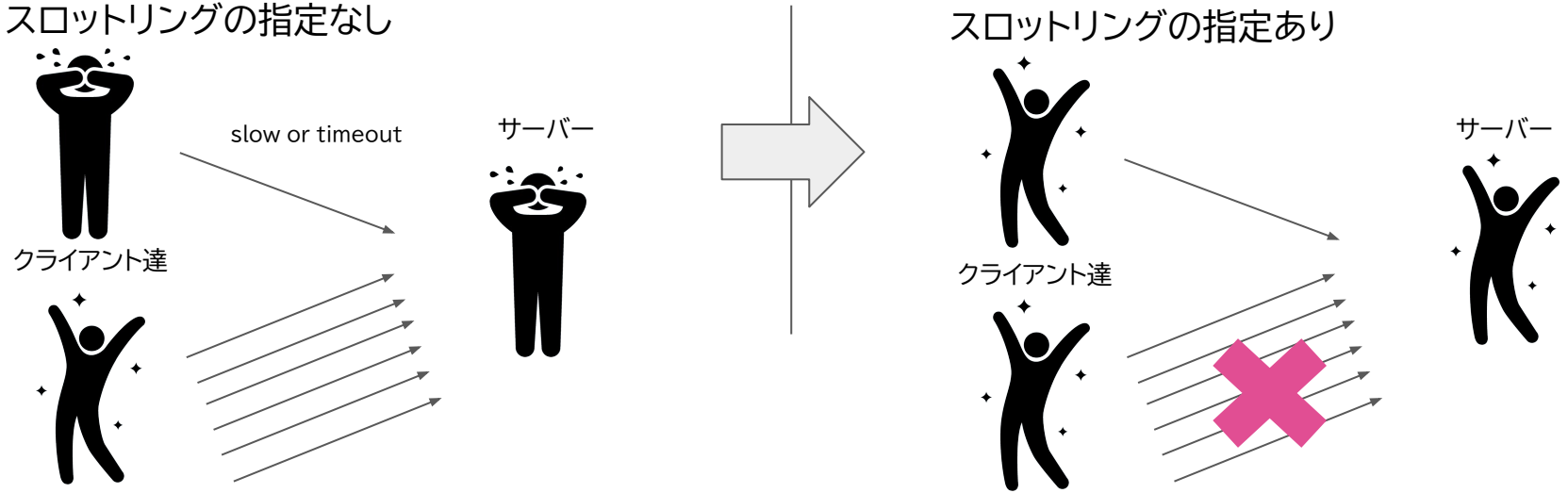
リクエストに対するスロットリングイメージ



頻度・量の制限（スロットリング）のメリット

参考:
https://future-architect.github.io/articles/20200121/
https://techblog.zozo.com/entry/zozotown-api-gateway-throttling

スロットリングの指定はセキュリティの向上，システムの負荷軽減，多くのユーザーの快適さに繋がる

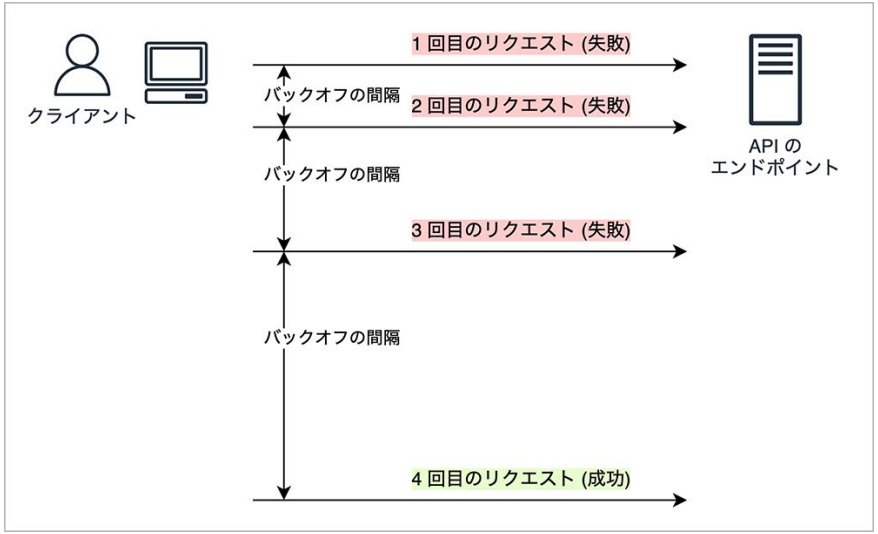


スロットリングのあるAPIへの効率的なリクエスト再試行

「ジッター＋指数バックオフ」アルゴリズムを使用する

指数バックオフ
- リクエストが拒否されるたびに、次のリクエストまでの時間を2倍にする

ジッター
- リクエスト時間に足し合わせるためのランダムな時間
- クライアントごとにリクエストのタイミングをばらつかせるために付与する



出典：
https://aws.amazon.com/jp/builders-flash/202211/way-to-operate-api-3/?awsf.filter-name=*all

バージョン管理とは

既にAPIを使用している人に不自由を感じさせることなくAPIを更新する手法である

何も意識していないバージョン管理

- Ver.1
  - リリースバージョン
- Ver.2
  - 大規模な変更
- Ver.3
  - 軽微なバグ修正のみ

いつ使えなくなるかわからず不便

互換性を意識したバージョン管理(一例)

- Ver.1. 0. 0
  - リリースバージョン
- Ver.1. 0. 1
  - バグ修正パッチ
- Ver.2. 0. 0
  - 大規模な変更(メジャー変更)

どこで動かなくなるのか一目瞭然

バージョン管理する際に意識する点

プロジェクトごとの後方互換性を定義する

後方互換性とは？

- **行った変更により既存のコードが動かなくなるか否か**
- 利用者によって後方互換の定義の度合いは異なる

後方互換性チェックポイント

- ❏ 新しい機能の追加
- ❏ バグ修正
- ❏ リファクタリング等の機能の最適化によるコードの変化
- ❏ セマンティクスの変化(リソース・フィールドの追加)

バージョン管理する際に意識する点

バージョン管理方式を選択する

バージョン管理方式選択の目安

- トレードオフを意識する
  - ユーザビリティ（使い勝手） or フレキシビリティ（柔軟性）
  - バージョン方式の「単純さ」 or「細かさ」
  - バージョンの「安定性」 or 「新機能」

代表的なバージョン管理方式

- 永久安定方式
- アジャイルインスタビリティ
  - プレビュー版, 現行版, 非推奨版でサイクルを回す
- セマンティックバージョン管理方式
  - 1.1.1等, 三つの数字でバージョンを管理

実装におけるバージョニング

参考:
https://www.django-rest-framework.org/api-guide/versioning/#hostnameversioning

バージョンの指定方法を決定する

- パスに入れる

```
https://example.com/v1/books
```

- クエリパラメータを使用する

```
https://example.com/books/?version=v1
```

- ホストネームによるバージョニング

```
https://v1.example.com/books/
```

- Acceptヘッダに入れる

```
GET /books/ HTTP/1.1
Host: example.com
Accept: application/json; version=1.0
```