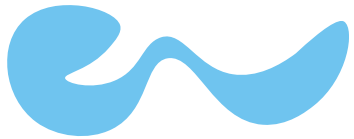
AWSLambdaでSVG to PNG 生成APIを生やした

みずあめ(@mizuameisgod)

@Education-JAWS ＃0 〜Welcome Students! 〜
2024/11/21

自己紹介

本名 佐藤 良
年齢 18歳
活動名 みずあめ
X @mizuameisgod
Web mizuame.works



mizuame.works

・筑波大学情報学群情報科学類1年

・本職は業務委託でUnrealEngineのテクニカルアーティストをやっています

・インターンでゲーム関連のR&Dもやってます

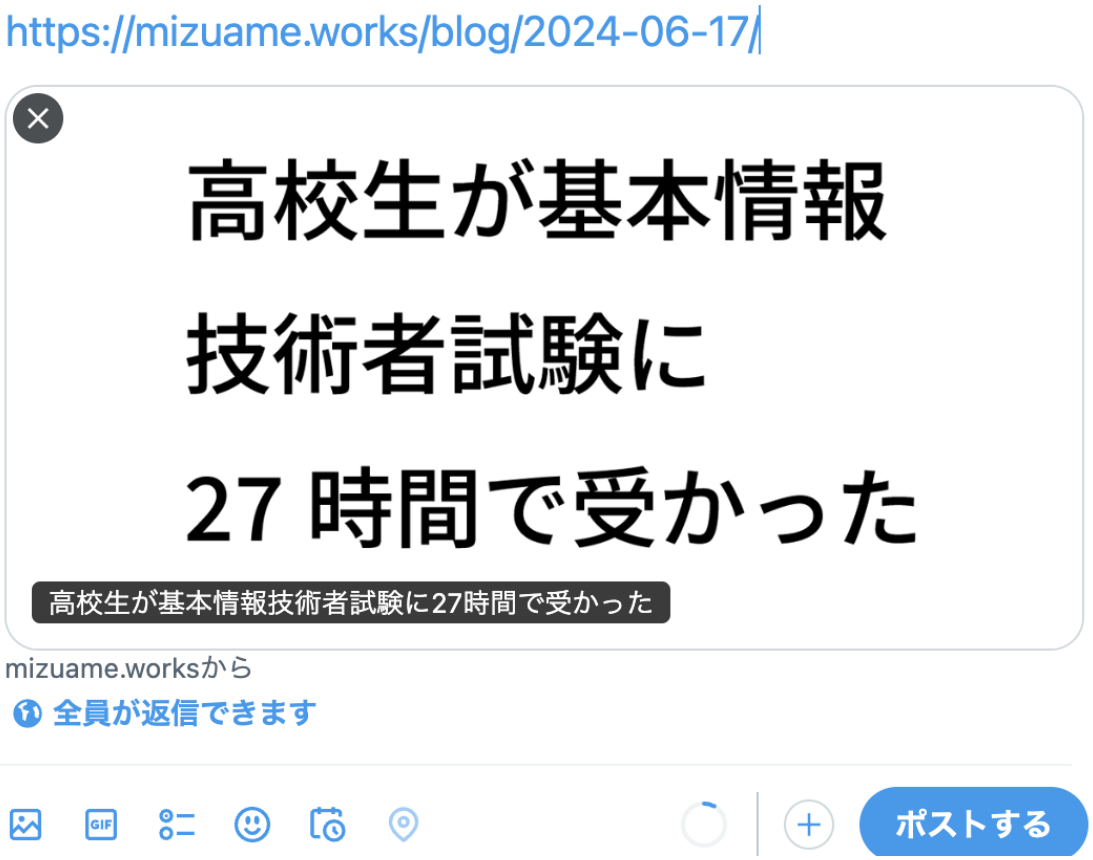
・オンラインゲームの制作がメイン

・最近はなぜかインフラもやってます

・Webは初心者、よくわかってない

・大学の友人の洗脳により、最近はオンプレに気持ちがあります 興味⇨k8s

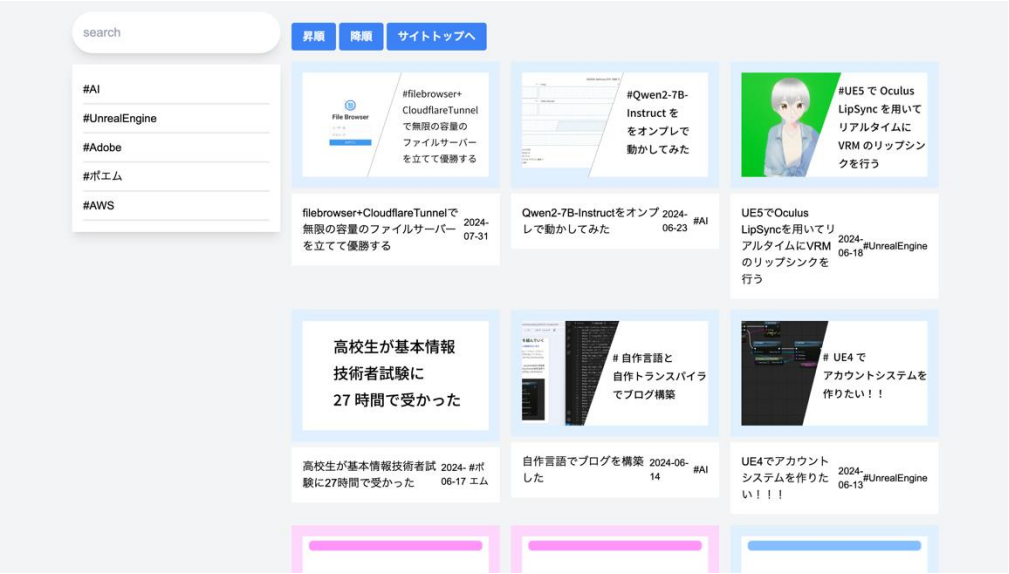
OGPとは



Open Graph Protcol
SNS等で共有した際に画像が表示されるあれ

.png.jpeg等が指定できるが、
svgは指定できない。

メインブログのOGP



メインブログのOGPは一回一回イラレでOGPを作っています
そこそこの頻度でこだわったOGPを設定したいと思っているので、
これは正しい

ブログ2欲しいな…

Qiitaとかの感じをイメージ

- メインブログより気軽に書けるブログも欲しい
- 思想強めの内容を書きたい
- はてなブログ匿名ダイアリーな感じだと良さげ
- 一記事自体の作成にコストをかけたくないので、OGPを自動生成しよう

とりあえず色々な実装をみてみた

- vercel/og

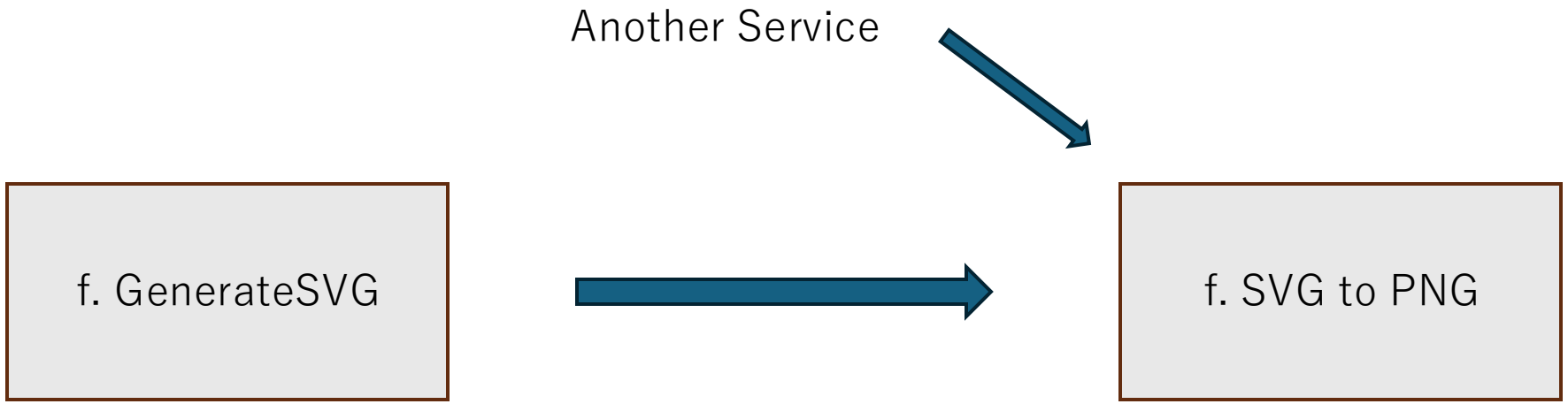
調べるとこれが一番使われているようだった。
内部的な仕組みだと

1, satoriというライブラリでHTML to SVG
2, resvgでSVG to PNG

といった感じで最初にSVGを作り、それをPNGにするらしい

SVG to PNGのAPIを生やした方が汎用性高そう

- 変なものをよく作る人間なので、OGP生成用として一括りにするのではなく、SVG生成とPNGへのコンバートをそれぞれ独立させた方が良さそうとなった。(他のサービス、システムでも使えるので)



Cloudflare WorkersかAWS Lambdaか

今回に関しては、AWS Lambdaに軍配が上がると考えた。

なぜか？

Workersは無料枠内だと、Memory128MB制限、Limit0.01s
Lambdaは無料枠内だと、 Memory128MB~10GB、Limit900s
(100 万件の無料リクエストと、1 か月あたり 40 万 GB 秒無料)

※そもそも論、Workersは重い処理向けのサービスではない

Cloudflare WorkersかAWS Lambdaか

加えて、AWS Lambdaではバイナリが実行できるのも大きい
(色々間違っている可能性があります)

例えばresvgをWorkersで動かす実装には、resvg-jsというwasmにコンパイルされたものを使っている(Workersはバイナリを実行できない、元はRust実装)

加えてWorkersにはパッケージ1MB以下制限がある
→色々動かせないnpmパッケージがある

結果的に、今回はAWS Lambdaを選択

どっちも良い点、悪い点があるため状況によって選択すると良い

実装

ランタイム
Node.js 20.x

ハンドラ
index.handler

アーキテクチャ
x86_64

Sharpという画像処理モジュールを使用

→裏がlibvips
→libvipsの裏はC実装、バイナリを叩いている

→LambdaはAmazon Linuxで動いている
→windowsPCの手元で動くものはぽしゃる
(windows用のバイナリが落ちてくる)

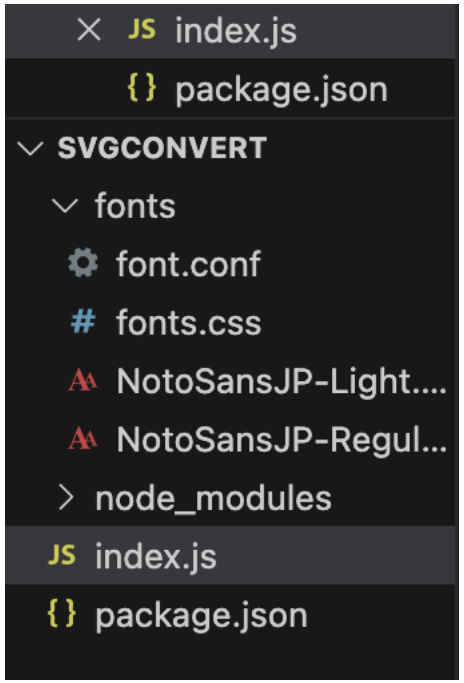
Dockerで環境を作り、それをzipにした

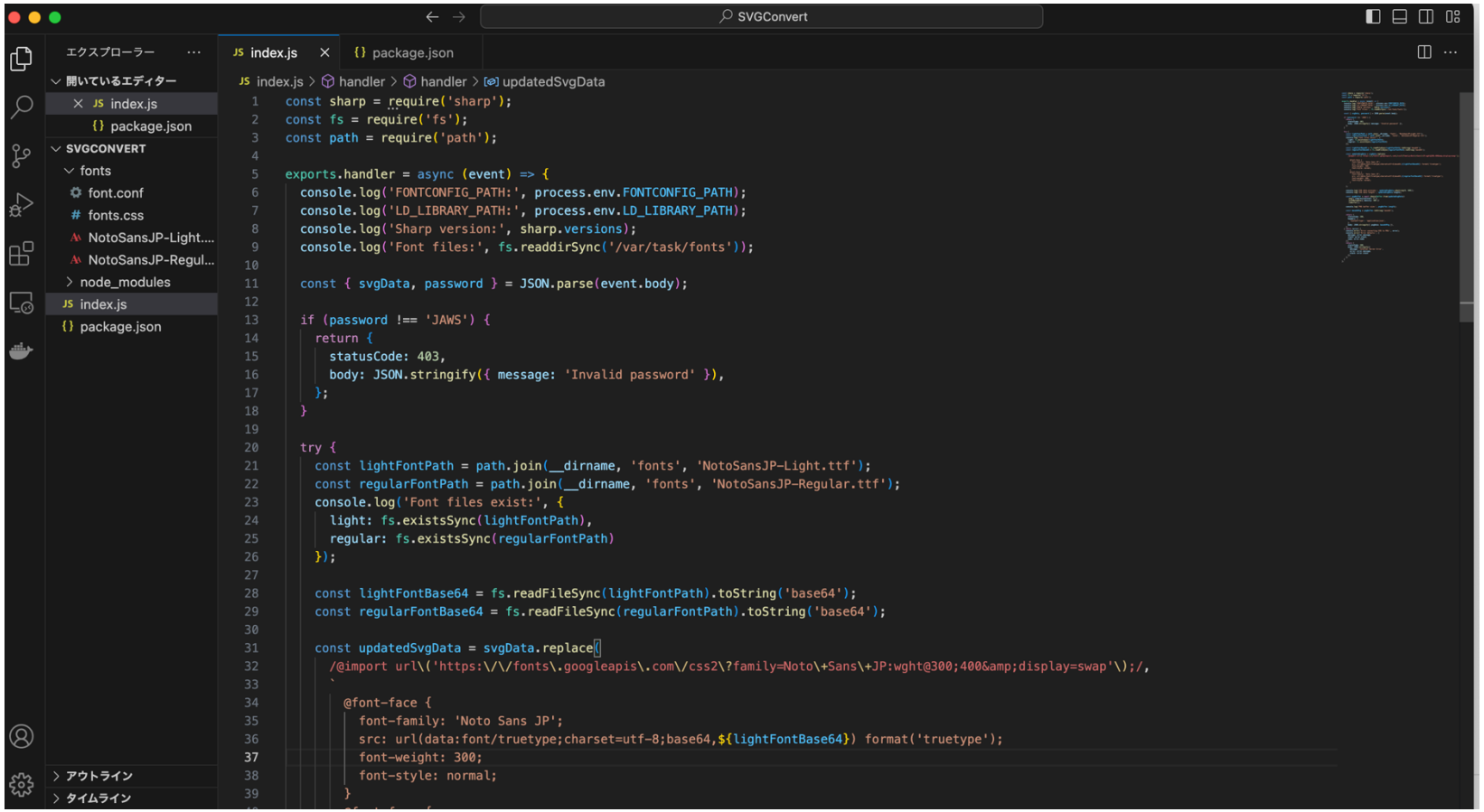
実装

超重要　index.jsがある階層でzip化

これで1時間はまった

実装


index.js
package.json
SVGCONVERT
fonts
font.conf
fonts.css
NotoSansJP-Light....
NotoSansJP-Regul...
node_modules
index.js
package.json


SVGConvert
エクスプローラー
開いているエディター
index.js
package.json
SVGCONVERT
fonts
font.conf
fonts.css
NotoSansJP-Light....
NotoSansJP-Regul...
node_modules
index.js
package.json
アウトライン
タイムライン
index.js
package.json
index.js > handler > handler > updatedSvgData
const sharp = require('sharp');
const fs = require('fs');
const path = require('path');
exports.handler = async (event) => {
console.log('FONTCONFIG_PATH:', process.env.FONTCONFIG_PATH);
console.log('LD_LIBRARY_PATH:', process.env.LD_LIBRARY_PATH);
console.log('Sharp version:', sharp.versions);
console.log('Font files:', fs.readdirSync('/var/task/fonts'));
const { svgData, password } = JSON.parse(event.body);
if (password !== 'JAWS') {
return {
statusCode: 403,
body: JSON.stringify({ message: 'Invalid password' }),
};
}
try {
const lightFontPath = path.join(__dirname, 'fonts', 'NotoSansJP-Light.ttf');
const regularFontPath = path.join(__dirname, 'fonts', 'NotoSansJP-Regular.ttf');
console.log('Font files exist:', {
light: fs.existsSync(lightFontPath),
regular: fs.existsSync(regularFontPath)
});
const lightFontBase64 = fs.readFileSync(lightFontPath).toString('base64');
const regularFontBase64 = fs.readFileSync(regularFontPath).toString('base64');
const updatedSvgData = svgData.replace(
/@import url\('https:\/\/fonts\.googleapis\.com\/css2\?family=Noto\+Sans\+JP:wght@300;400&display=swap'\);/,
`
@font-face {
font-family: 'Noto Sans JP';
src: url(data:font/truetype;charset=utf-8;base64,${lightFontBase64}) format('truetype');
font-weight: 300;
font-style: normal;
}

実装

フォントが
反映されない

→パスでフォント
を直接指定しよう

→Lambdaの
パスって？

□□□□□□□□□□□□□□□□□□□□

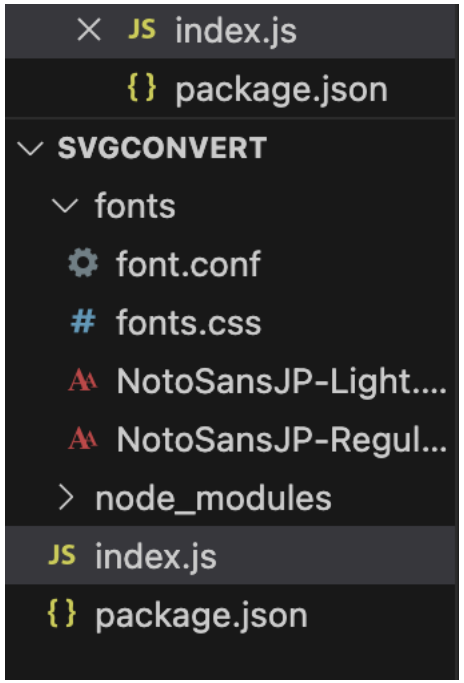
実装

結論：/var/task に展開される

コードで `import` または `require` ステートメントを使用すると、Node.js ランタイムはモジュールが見つかるまで `NODE_PATH` パス内のディレクトリを検索します。デフォルトでは、ランタイムが最初に検索する場所は、.zip デプロイパッケージを解凍してマウントするディレクトリ (`/var/task`) です。ランタイムに含まれるライブラリのバージョンをデプロイパッケージに含める場合、そのバージョンが、ランタイムに含まれるバージョンよりも優先されます。デプロイパッケージ内の依存関係も、レイヤー内の依存関係よりも優先されます。

https://docs.aws.amazon.com/ja_jp/lambda/latest/dg/nodejs-package.html

実装



環境変数 (2)

以下の環境変数はデフォルトの Lambda サービスキーを使用して保存時に暗号化されました。

環境変数を検索

キー	値
FONTCONFIG_FILE	/var/task/fonts/font.conf
FONTCONFIG_PATH	/var/task/fonts

```
<?xml version="1.0"?>
<!DOCTYPE fontconfig SYSTEM "fonts.dtd">
<fontconfig>
 <dir>/var/task/fonts</dir>
 <cachedir>/tmp/fontconfig-cache</cachedir>
 <config>
  <rescan>
   <int>30</int>
  </rescan>
 </config>
</fontconfig>
```

フォントをただパスで指定しただけでもこける

font.confというファイルを作り、指定する。

font.conf自体も環境変数で指定

実装

DDoS対策をした方が良い

レート 情報
1

バースト 情報
1

レート制限をかけましょう

実装

どこかに公開していなくても
DDoSを受けるらしい

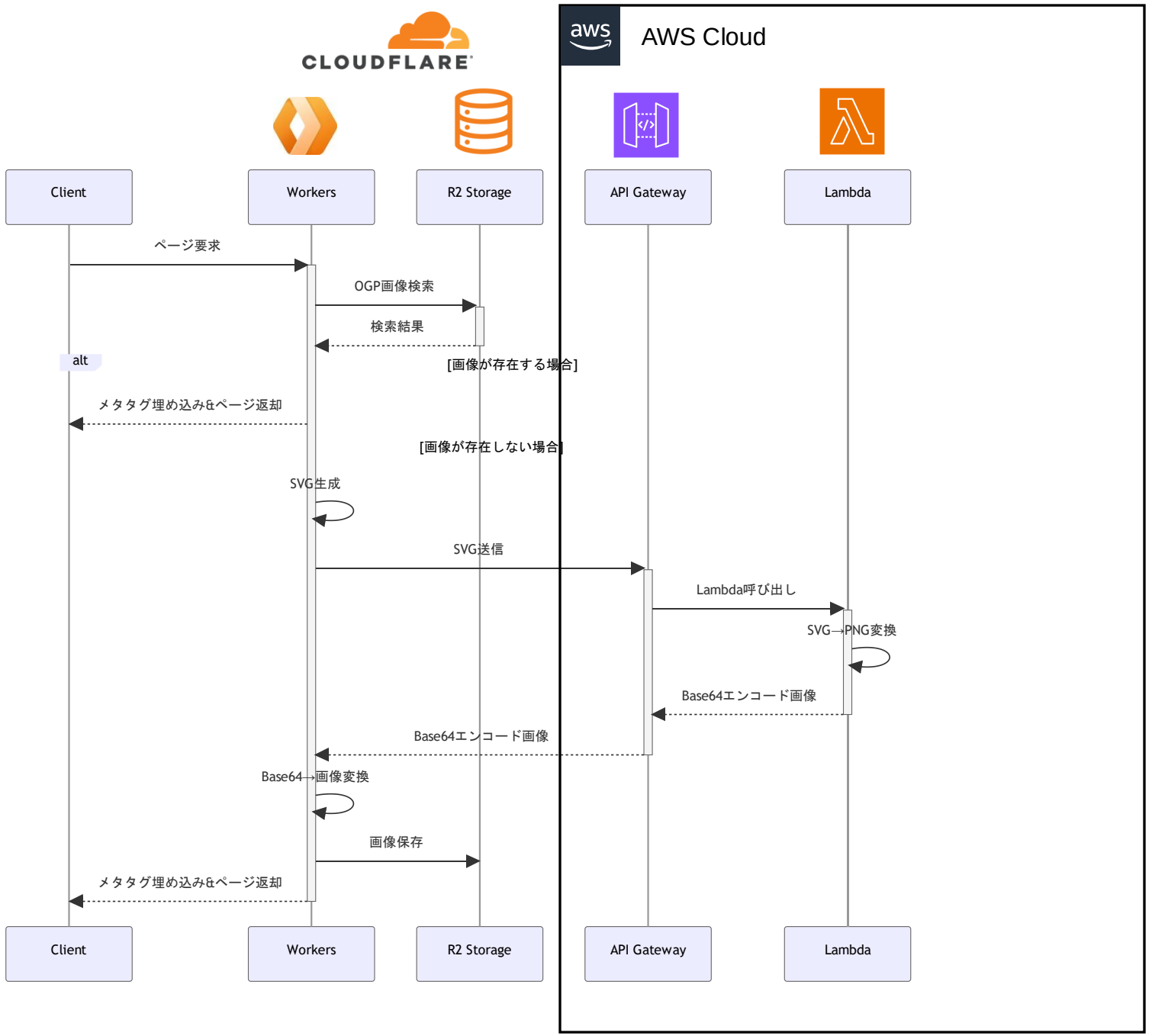
https://{restapi_id}.execute-api.{region}.amazonaws.com/{stage_name}/

ある程度URLは推測可能なため、攻撃は発生しゆる

→ステージ名初期値の「default」は使用しないなど

引用 : https://qiita.com/naka345/items/98582cfb45ab09c3df9f

OverView


CLOUDFLARE
aws
AWS Cloud
Client
Workers
R2 Storage
API Gateway
Lambda
ページ要求
OGP画像検索
検索結果
alt
[画像が存在する場合]
メタタグ埋め込み&ページ返却
[画像が存在しない場合]
SVG生成
SVG送信
Lambda呼び出し
SVG→PNG変換
Base64エンコード画像
Base64エンコード画像
Base64→画像変換
画像保存
メタタグ埋め込み&ページ返却
Client
Workers
R2 Storage
API Gateway
Lambda

まとめ

汎用性とコストを考えた結果、
マルチベンダーになった

Thank you for watching!!!