インプレスR&D［NextPublishing］

技術の泉 SERIES
E-Book / Print Book

ReactとPythonでAPI販売サービスを作ろう

高橋 太郎 著

実践的Reactアプリ
開発学習書!

impress R&D
An Impress Group Company

GIJUTSU no IZUMI SERIES
泉
POWERD by NEXTPUBLISHING
技術の泉 SERIES

インプレスR&D［NextPublishing］

技術の泉 SERIES
E-Book / Print Book

ReactとPythonでAPI販売サービスを作ろう

高橋 太郎 著

実践的Reactアプリ開発学習書!

impress R&D

GIJUTSU no IZUMI SERIES 泉 POWERED by NEXTPUBLISHING
技術の泉


An Impress
Group Company
SERIES

電子書籍閲覧に関するご注意

本書では、プログラムリストに専用の等幅フォントを使用しています。ビューアによって以下の作業が必要になります。

・Kindle Paperwhiteの場合：フォント設定画面で「出版者のフォント」を選択

・kobo Androidアプリの場合：フォント画面で「オリジナル」を選択

目次

はじめに

執筆の経緯

筆者はサーバーサイドエンジニアです。いつものように社内向けの仕事をしていたら、あるとき「これは世の中に出したら売れるんじゃないか」と思えるAPIができました。そこで自作のAPIを販売する方法を調査してみましたが、既存のサービスはどれもいまいちでした。[1]「それなら自分で作ってしまおう」そう考えて作ったのがこちらのAPI販売サービス（図1）です。[2]

図1: FM Mail

正確にはこのサービスは筆者の練習用で、次に作るサービスが本命です。ただ練習用とはいえ、認証やDB接続、クレジットカード決済機能まで備えた本格的なWebアプリケーションです。捨ててしまうのはもったいないので、会社の許可をとってオープンソース化し、開発中に得られた知見を共有することにしました。[3]

開発はZenn[4]で記事を書きながら進めていきました。開発に集中すれば3か月ほどで作れる規模のアプリケーションですが、業務[5]の合間にドキュメントを書きながら進めていったため、完成まで半年ほどかかってしまいました。そのころにはZennの記事も本が一冊作れそうなくらいたまっていましたが、開発しながら手探り状態の中で書いていたため、振り返ってみると記述のアラも目立ちます。そこで記事の内容を全面的に書き直し、一冊の本として再編成したのが本書です。

想定する読者

本書は次に該当する方を対象としています。ひとつでも当てはまる方は、ぜひお読みいただけると幸いです。

・Reactで実践的なWebアプリケーションを作ってみたい

・サーバーサイドエンジニアだがモダンフロントエンドに興味がある

・自作のAPIを販売したい

・サーバーレスアーキテクチャに興味がある

・Webサービスに決済機能を追加する方法を知りたい

本書は入門書ではありません。できるだけ詳しく説明するように心がけてはいますが、ページの都合もあり、あまりに基礎的な部分、たとえばNode.jsやPythonのインストール方法などは記載していません。入門書が必要な方はほかの書籍をお買い求めください。筆者のお勧めは「りあクト！ TypeScriptで始めるつらくないReact開発 第3.1版」[6]です。こちらの本にはReact開発に必要なことがほとんど載っています。筆者は実際にReactに関する書籍はこの3冊（3部作なので）しか読んでいなかったのですが[7]、本書のWebアプリケーションを最後まで作り上げることができました。一方、Pythonについてはお勧めの入門書は特にありません。筆者の場合は必要に応じてインターネットで調べているうちに、なんとなく書けるようになっていました。

本書のWebアプリケーションを開発するには、GitHub、AWS、Google、Netlify、Stripeのアカウントがそれぞれ必要となります。事前にご用意ください。フロントエンドのデプロイ先はNetlifyで、バックエンドのAPI群のデプロイ先はAWSです。

本書で開発するWebサービスについて

Claris社が販売するFileMakerという開発プラットフォームはご存じでしょうか。FileMakerは古くからあるローコード開発ツールの一種です。プログラミングが苦手な人でも簡単に本格的なアプリケーションケーションを作ることができるため、一部で根強い人気があります。筆者の職場でもFileMakerが使われていて、業務に合わせて作られた多数のアプリケーションケーションが毎日使われています。

FileMakerには数多くの機能が用意されていますが[8]、不思議なことにメールを受信する機能はありません。メールを送信する機能はあるにもかかわらずです。FileMakerでメールを受信したいユーザーというのは昔から結構存在するため、そういった人々のために、サードパーティーのプラグインが作られてきました。弊社でもMailit[9]というプラグインを使っていましたが、メンテナンス性にやや難があったため、ある時期から筆者が作成したAPIに切り替えました。これが上述した「売れるんじゃないか」と考えたAPIのひとつです。

本書では、FileMaker用のメール受信APIを販売するWebサービス[10]を作っていきます。FileMakerの市場規模は小さすぎるので、残念ながらこのWebサービスを公開しても十分な利益は出ないでしょう。[11]しかし、販売するAPI部分は簡単に差し替えることができます。需要のありそうなAPIを作ったことがある方やアイデアのある方は、本書を参考にWebサービスを作ってAPIを販売してみてはいかがでしょうか。

なお、本書のWebアプリケーションを開発するのにFileMakerは必要ありません。

筆者の開発環境

本書のWebアプリケーションは次の環境で開発しています。

・macOS Monterey （Intel Mac）

・Node.js 16.13.0

・Python 3.9.10

・VS Code

Appleシリコン搭載Mac（M1 Max）でビルドできることは確認済みです。[1][2]WindowsやLinuxでの動作確認はしておりませんので、あらかじめご了承ください。

サンプルコード

GitHubにサンプルコードを用意しました。

https://github.com/sikkimtemi/How_to_create_API_sales_service

フォルダ名は章の数字に対応しています。章が進むごとに機能が追加されていくので、筆者の開発作業を追体験できるようになっています。サンプルコードを動かしながら本書をお読みいただくことで、より理解が深まるでしょう。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、著者はいかなる責任も負いません。

商標

本書に登場するシステム名や製品名は、関係各社の商標または登録商標です。また本書では™や©、®といったマークは省略しています。

1. 楽天Rapid APIやAWS Marketplaceといった既存サービスが存在します。
2. https://fmmail.netlify.app/
3. もともと技術情報は積極的に発信してきましたが、このプロジェクトではシークレットキーなどの公開できない情報以外は100%公開しています。
4. https://zenn.dev/sikkim
5. 筆者はいわゆる「情シス」の中の人なので、社員の「何もしていないのにパソコンが動かなくなったんですけど」に対応するのが主な仕事です。ちなみに、大抵の場合は再起動すると直ります。
6. https://oukayuka.booth.pm/
7. もちろん本の知識だけでは足りないので、インターネットからの情報収集は必須です。また、現在は他の本も読んでいます。
8. https://help.claris.com/ja/pro-help/content/script-steps-reference.html
9. https://www.dacons.net/mailit
10. FileMaker用のメール受信サービスなので、「FM Mail」と名付けました。ただ、次のバージョンからブランド名がFileMakerではなくなるらしいです。
11. せいぜい子どものお小遣い程度の売上にしかならないはずです。運用の手間を考慮すると赤字ですね。
12. FM Mailのビルド時間はIntel Mac（Core i7, メモリ16GB）で44秒、M1 Mac（M1 Max, メモリ32GB）で8秒でした。Mac Studioは速いですね。

第1章　フロントエンドの環境構築

1.1　技術選定をしよう

本章では、フロントエンド開発の基盤となる開発環境を準備します。この構成はあとから変更するのが難しいため、最初の技術選定がとても重要になります。筆者が悩み抜いて選んだ構成がこちらです。[1]

・React（JavaScriptライブラリ）
・TypeScript（開発言語）
・Vite（ビルドツール）
・ESLint（静的解析ツール）
・Prettier（コードフォーマッター）
・Tailwind CSS（CSSフレームワーク）

1.1.1　React

React、Vue、Angularを3大フレームワーク[2]といいます。シェアはReactが圧倒的に1位で、その差はさらに広がりつつあります。ユーザー数が多いということは情報収集しやすいということです。筆者は以前Vue.jsを使っていましたが、しばらく前から何かを検索しようとするとReactの情報ばかりヒットするようになり、Reactを知らないことに危機感を覚え始めていました。そこでこのプロジェクトではReactを使おうと、心に決めていました。

1.1.2　TypeScript

Reactの開発言語はJavaScriptとTypeScriptから選べます。特に理由がない限りはTypeScriptにしましょう。JavaScriptでは実際に動かすまでエラーに気付かないことがありますが、TypeScriptなら開発中に気付く場合が多くて助かります。

1.1.3　Vite

ReactにはCreate React AppというMeta社謹製のビルドツールがありますが、Viteを使った方がはるかに速くビルドできます。プログラマーの3大美徳のひとつは「短気」です。ビルドする時間

は短いほどよいのです。シェアも圧倒的に高いので情報は集めやすいです。

1.1.4　ESLintとPrettier

ESLintとPrettierはきれいなコードを保つためのツールです。ESLintはコーディング規約に違反した箇所を指摘してくれます。コーディング規約を守らないとエラーになってコミットできないので、チームにルールを徹底させることができます。個人で開発する場合でも、コーディング規約に準拠したコードを書けるようになるのは大きなメリットです。Prettierはフォーマッターです。適当に書いたコードでもきれいに整形してくれます。VS Codeと組み合わせると、ファイルの保存時に自動的に整形してくれるのでとても楽です。

1.1.5　Tailwind CSS

Tailwind CSSはとても人気のあるCSSフレームワークです。筆者はCSSを書くのが苦手なので、なるべくCSSを書かずに済む方法を探し求めていました。Tailwind CSSのおかげで、本書ではCSSをまったく書かずに済みました。

1.2　環境構築手順

それでは、実際に手を動かしながら環境を構築していきましょう。GitHubに環境構築済みのリポジトリも用意してあります。[3]

1.2.1　Viteで新規プロジェクトを生成する（React + TypeScript）

Viteは標準でReactとTypeScriptに対応しています。次のようにコマンドを実行して、対話形式で設定していくと、新規プロジェクトが生成されます。

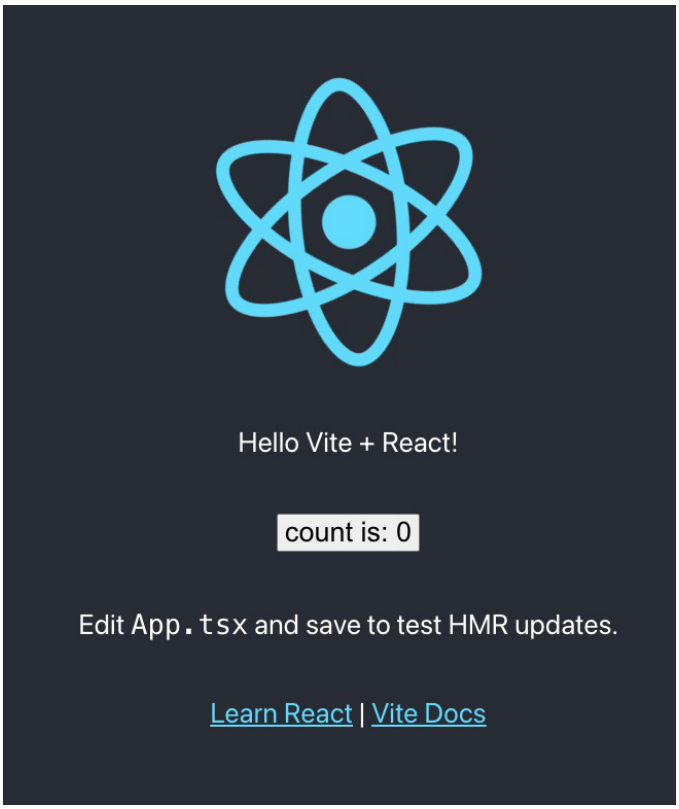
```
npm create vite@latest
 Project name: … {任意のプロジェクト名}
 Select a framework: › react
 Select a variant: › react-ts
```

いったん動かしてみましょう。

```
cd {プロジェクト名}
npm install
npm run dev
```

ブラウザでhttp://localhost:3000にアクセスして、図1.1のように表示されればOKです。

図1.1: Hello Vite + React!



1.2.2　ESLintの設定

ESLintの設定は、フロントエンド開発において特に難しいもののひとつです。同時にとても重要でもあります。ESLintに限らず、開発環境の設定を理解せずに使うのはたいへん危険なので、がんばって理解していきましょう。[4]

まずはモジュールをインストールしましょう。

```
npm i -D eslint
```

次に初期設定します。対話形式で多くの質問に答える必要があります。

```
npm init @eslint/config


How would you like to use ESLint?:
 To check syntax, find problems, and enforce code style
```

```
What type of modules does your project use?:
 JavaScript modules (import/export)

Which framework does your project use?: React

Does your project use TypeScript?: Yes

Where does your code run?: Browser

How would you like to define a style for your project?:
 Use a popular style guide

Which style guide do you want to follow?:
 Airbnb: https://github.com/airbnb/javascript

What format do you want your config file to be in?:
 JavaScript

  Checking peerDependencies of eslint-config-airbnb@latest
  The config that you've selected requires the following dependencies:
    eslint-plugin-react@^7.28.0  @typescript-eslint/eslint-plugin@latest  eslint-
config-airbnb@latest   eslint@^7.32.0   ||   ^8.2.0   eslint-plugin-import@^2.25.3
eslint-plugin-jsx-a11y@^6.5.1    eslint-plugin-react-hooks@^4.3.0    @typescript-
eslint/parser@latest

Would you like to install them now with npm?: Yes
```

最後の質問にYesと答えると、モジュールが自動でインストールされます。npmを使っている場合は問題ありませんが、yarnを使っている場合は、package-lock.jsonを削除してインストールし

直す必要があります。

全ての質問に正しく答えると、次の.eslintrc.jsファイルが生成されます。

```
module.exports = {
  env: {
    browser: true,
    es2021: true,
  },
  extends: [
    'plugin:react/recommended',
    'airbnb',
  ],
  parser: '@typescript-eslint/parser',
  parserOptions: {
    ecmaFeatures: {
      jsx: true,
    },
    ecmaVersion: 'latest',
    sourceType: 'module',
  },
  plugins: [
    'react',
    '@typescript-eslint',
  ],
  rules: {
  },
};
```

ESLintは、さまざまなコーディングルールを追加のプラグインとしてインストールできます。しかし、プラグインをインストールしただけではルールが有効になりません。有効にするには.eslintrc.jsに記

述を追加する必要があります。必要なルールを適用するため、次のように書き換えましょう。

```
module.exports = {
  env: {
    browser: true,
    es2021: true,
  },
   extends: [
     'plugin:react/recommended',
     'airbnb',
+    'airbnb/hooks',
+    'plugin:import/errors',
+    'plugin:import/warnings',
+    'plugin:import/typescript',
+    'plugin:@typescript-eslint/recommended',
+    'plugin:@typescript-eslint/recommended-requiring-type-checking',
   ],
   parser: '@typescript-eslint/parser',
   parserOptions: {
    ecmaFeatures: {
       jsx: true,
     },
     ecmaVersion: 'latest',
+    project: './tsconfig.eslint.json',
     sourceType: 'module',
+    tsconfigRootDir: __dirname,
   },
   plugins: [
+    'import',
+    'jsx-a11y',
```

```
      'react',
+     'react-hooks',
      '@typescript-eslint'
    ],
+   root: true,
    rules: {
+     'lines-between-class-members': [
+       'error',
+       'always',
+       {
+         exceptAfterSingleLine: true,
+       },
+     ],
+     'no-void': [
+       'error',
+       {
+         allowAsStatement: true,
+       },
+     ],
+     'padding-line-between-statements': [
+       'error',
+       {
+         blankLine: 'always',
+         prev: '*',
+         next: 'return',
+       },
+     ],
+     '@typescript-eslint/no-unused-vars': [
+       'error',
+       {
+         vars: 'all',
```

```
+        args: 'after-used',
+        argsIgnorePattern: '_',
+        ignoreRestSiblings: false,
+        varsIgnorePattern: '_',
+      },
+    ],
+    'import/extensions': [
+      'error',
+      'ignorePackages',
+      {
+        js: 'never',
+        jsx: 'never',
+        ts: 'never',
+        tsx: 'never',
+      },
+    ],
+    'react/jsx-filename-extension': [
+      'error',
+      {
+        extensions: ['.jsx', '.tsx'],
+      },
+    ],
+    'react/jsx-props-no-spreading': [
+      'error',
+      {
+        html: 'enforce',
+        custom: 'enforce',
+        explicitSpread: 'ignore',
+      },
+    ],
+    'react/react-in-jsx-scope': 'off',
```

```
+  },
+  overrides: [
+    {
+      files: ['*.tsx'],
+      rules: {
+        'react/prop-types': 'off',
+      },
+    },
+  ],
+  settings: {
+    'import/resolver': {
+      node: {
+        paths: ['src'],
+      },
+    },
   },
 };
```

急に長くなってたいへんですが、ひとつずつ確認していきましょう。まず.eslintrc.jsの構文について説明します。次の表をご覧ください。

eslintrc.jsの設定項目	説明
extends	各プラグインルールの推奨の共有設定をプラグイン開発者が提供しているので、ここで指定する。*順番が重要*。競合する設定はあとから記述されたものによって上書きされる。
parserOptions	ESLintのパーサへ渡すオプションを設定する。
parserOptions.project	プロジェクトのTypeScriptコンパイル設定ファイルのパスをパーサに教えるための設定。tsconfig.jsonではなく、tsconfig.eslint.jsonという別ファイルを用意して渡している。こうしないとパーサがローカルにインストールされたnpmパッケージのファイルまでパースしてしまう。
parserOptions.tsconfigRootDir	相対パスの起点。
plugins	読み込ませる追加ルールのプラグインを設定する。ここに記述しないとプラグインは有効にならないので要注意。
root	ESLintはデフォルトの挙動として親ディレクトリの設定ファイルまで読み込んでしまう。trueにすることで、その挙動を抑制している。

rules	各ルールの適用の可否やエラーレベルを設定する。主にextendsで読み込んだ共有設定を書き換える場合に用いる。
overrides	任意のglobパターンにマッチするファイルのみ、ルールの適用を上書きする。ここでは*.tsxファイルに対してのみreact/prop-typesを無効化するのに利用している。
settings	任意の実行ルールに適用される追加の共有設定。詳しくは後述。

settingsでは次の問題を解決しています。複雑なので順を追って説明します。

・前提としてtsconfig.jsonでsrc/配下のファイルを絶対パスでインポートできるようにしている

・このままではeslint-plugin-importがその絶対パスを解決できずにエラーとなる

・eslint-plugin-importは内部でeslint-import-resolver-nodeというモジュール解決プラグインを使用している

・eslint-import-resolver-nodeに対して、そのパスにsrcを追加することでエラーを解消している

さらにtsconfig.eslint.jsonも追加しましょう。

```
{
  "extends": "./tsconfig.json",
  "include": [
    "src/**/*.js",
    "src/**/*.jsx",
    "src/**/*.ts",
    "src/**/*.tsx"
  ],
  "exclude": [
    "node_modules"
  ]
}
```

追加・変更したルールの説明はこちらです。内容はほぼ「りあクト！」からの引用ですが、モジュールのバージョンアップに伴い、若干変更しています。

ルール	説明

lines-between-class-members	クラスメンバーの定義の間に空行を入れるかどうかを定義するルール。1行記述のメンバーのときは空行を入れなくていいように緩めている。
no-void	void 演算子の（式としての）使用を禁ずるルール。Effect Hook内で非同期処理を記述する際、@typescript-eslint/no-floating-promisesルールに抵触するのを回避するのにvoid文を記述する必要があるため、文としての使用のみを許可している。
padding-line-between-statements	任意の構文の間に区切りの空行を入れるかどうかを定義するルール。ここではreturn文の前に常に空行を入れるよう設定している。
@typescript-eslint/no-unused-vars	使用していない変数の定義を許さないルール。ここでは変数名を「_」にしたときのみ許容するように設定。
import/extensions	インポートの際のファイル拡張子を記述するかどうかを定義するルール。npmパッケージ以外のファイルについて .js、.jsx、.ts、.tsx のファイルのみ拡張子を省略し、ほかのファイルは拡張子を記述させるように設定。
react/jsx-filename-extension	JSXのファイル拡張子を制限するルール。eslint-config-airbnbで.jsxのみに限定されているので、.tsxを追加。
react/jsx-props-no-spreading	JSXでコンポーネントを呼ぶときのpropsの記述にスプレッド構文を許さないルール。eslint-config-airbnbにて全て禁止されているが、<Foo {...{ bar, baz } /}> のように、個々のpropsを明記する書き方のみ許容するように設定。
react/react-in-jsx-scope	JSX記述を使用する場合にreactモジュールをReactとしてインポートすることを強制するルール。新しいJSX変換形式を用いる場合はインポートが不要になるためこの設定を無効化。
react/prop-types	コンポーネントのpropsに型チェックを行うためのpropTypesプロパティの定義を強制するルール。eslint-config-airbnbで設定されているが、TypeScriptの場合は不要なので、ファイル拡張子が.tsxの場合に無効化するよう設定を上書き。

余計なファイルがESLintの対象にならないように、.eslintignoreも追加しましょう。

```
build/
public/
**/coverage/
**/node_modules/
**/*.min.js
*.config.js
.*lintrc.js
```

さらに関数定義をアロー関数式に統一するためのルールを追加適用します。まずはプラグインをインストールしましょう。

```
npm i -D eslint-plugin-prefer-arrow
```

.eslintrc.jsのpluginsとrulesを次のように修正します。

```
plugins: [
    'import',
    'jsx-a11y',
+   'prefer-arrow',
    'react',
    'react-hooks',
    '@typescript-eslint',
  ],
+   'prefer-arrow/prefer-arrow-functions': [
+     'error',
+     {
+       disallowPrototype: true,
+       singleReturnOnly: false,
+       classPropertiesAllowed: false,
+     },
+   ],
```

これだけだとreact/function-component-definitionと競合してしまうので、rulesに次の記述を追加します。

```
+   'react/function-component-definition': [
+     'error',
+     {
+       namedComponents: 'arrow-function',
+       unnamedComponents: 'arrow-function',
+     },
+   ],
```

ESLint単体の設定はこれで完了です。[5]

1.2.3 Prettierの設定

PrettierはESLintと比べたら設定項目が少なくて楽です。まずはモジュールをインストールしましょう。

```
npm i -D prettier eslint-config-prettier
```

次に、.eslintrc.jsのextendsにprettier用の記述を追加します。上でも述べたとおり、extendsは順番が非常に重要です。prettierの記述は最後にくるように設定しましょう。

```
extends: [
  'plugin:react/recommended',
  'airbnb',
  'airbnb/hooks',
  'plugin:import/errors',
  'plugin:import/warnings',
  'plugin:import/typescript',
  'plugin:@typescript-eslint/recommended',
  'plugin:@typescript-eslint/recommended-requiring-type-checking',
+ 'prettier',
],
```

最後に.prettierrcを追加します。[6]

```
{
  "singleQuote": true,
  "trailingComma": "all",
```

```
  "endOfLine": "auto"
}
```

1.2.4　.eslintrc.jsの最終形

.eslintrc.jsの設定が全て終わったので、全体を載せておきます。

```
module.exports = {
  env: {
    browser: true,
    es2021: true,
  },
  extends: [
    'plugin:react/recommended',
    'airbnb',
    'airbnb/hooks',
    'plugin:import/errors',
    'plugin:import/warnings',
    'plugin:import/typescript',
    'plugin:@typescript-eslint/recommended',
    'plugin:@typescript-eslint/recommended-requiring-type-checking',
    'prettier',
  ],
  parser: '@typescript-eslint/parser',
  parserOptions: {
    ecmaFeatures: {
      jsx: true,
    },
    ecmaVersion: 'latest',
    project: './tsconfig.eslint.json',
    sourceType: 'module',
```

```
    tsconfigRootDir: __dirname,
  },
  plugins: [
    'import',
    'jsx-a11y',
    'prefer-arrow',
    'react',
    'react-hooks',
    '@typescript-eslint'
  ],
  root: true,
  rules: {
    'lines-between-class-members': [
      'error',
      'always',
      {
        exceptAfterSingleLine: true,
      },
    ],
    'no-void': [
      'error',
      {
        allowAsStatement: true,
      },
    ],
    'padding-line-between-statements': [
      'error',
      {
        blankLine: 'always',
        prev: '*',
        next: 'return',
```

```
    },
  ],
  'prefer-arrow/prefer-arrow-functions': [
    'error',
    {
      disallowPrototype: true,
      singleReturnOnly: false,
      classPropertiesAllowed: false,
    },
  ],
  'react/function-component-definition': [
    'error',
    {
      namedComponents: 'arrow-function',
      unnamedComponents: 'arrow-function',
    },
  ],
  '@typescript-eslint/no-unused-vars': [
    'error',
    {
      vars: 'all',
      args: 'after-used',
      argsIgnorePattern: '_',
      ignoreRestSiblings: false,
      varsIgnorePattern: '_',
    },
  ],
  'import/extensions': [
    'error',
    'ignorePackages',
    {
```

```
        js: 'never',
        jsx: 'never',
        ts: 'never',
        tsx: 'never',
      },
    ],
    'react/jsx-filename-extension': [
      'error',
      {
        extensions: ['.jsx', '.tsx'],
      },
    ],
    'react/jsx-props-no-spreading': [
      'error',
      {
        html: 'enforce',
        custom: 'enforce',
        explicitSpread: 'ignore',
      },
    ],
    'react/react-in-jsx-scope': 'off',
  },
  overrides: [
    {
      files: ['*.tsx'],
      rules: {
        'react/prop-types': 'off',
      },
    },
  ],
  settings: {
```

```
    'import/resolver': {
      node: {
        paths: ['src'],
      },
    },
  },
};
```

1.2.5　VS Code機能拡張のインストール

まず.vscode/extensions.jsonを追加します。ここに記載したのは、筆者お勧めのVS Code機能拡張です。

```
{
  // See http://go.microsoft.com/fwlink/?LinkId=827846 to learn about workspace recommendations.
  // Extension identifier format: ${publisher}.${name}. Example: vscode.csharp

  // List of extensions which should be recommended for users of this workspace.
  "recommendations": [
    "dbaeumer.vscode-eslint",
    "esbenp.prettier-vscode",
    "oderwat.indent-rainbow",
    "VisualStudioExptTeam.vscodeintellicode",
    "wix.vscode-import-cost"
  ],
  // List of extensions recommended by VS Code that should not be recommended for users of this workspace.
  "unwantedRecommendations": []
}
```

機能拡張をインストールしていないVS Codeでプロジェクトを開くと、機能拡張をインストールするかどうかを確認するダイアログが出ます。Yesをクリックしてインストールしてください。少なくともESLintとPrettierの機能拡張は必ず入れましょう。

1.2.6　VS Codeの設定

VS Codeでファイルを保存したときに、自動でlintとフォーマットが走るようにします。.vscode/settings.jsonを追加してください。

```
{
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": true
  },
  "editor.formatOnSave": false,
  "eslint.packageManager": "npm",
  "typescript.enablePromptUseWorkspaceTsdk": true,
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "[graphql]": {
    "editor.formatOnSave": true
  },
  "[javascript]": {
    "editor.formatOnSave": true
  },
  "[javascriptreact]": {
    "editor.formatOnSave": true
  },
  "[json]": {
    "editor.formatOnSave": true
  },
  "[typescript]": {
    "editor.formatOnSave": true
```

```
  },
  "[typescriptreact]": {
    "editor.formatOnSave": true
  },
}
```

ここまでの設定が正しくできているか確認します。App.tsxを次のように修正してください。VS Codeは事前に再起動しておくと無難です。設定が正しければエラーにはなりません。また、ファイル保存時に自動的に整形されることも確認しましょう。

```
import { useState, FC } from 'react';
import logo from './logo.svg';
import './App.css';

const App: FC = () => {
  const [count, setCount] = useState(0);

  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>Hello Vite + React!</p>
        <p>
          <button type="button" onClick={() => setCount((c) => c + 1)}>
            count is: {count}
          </button>
        </p>
        <p>
          Edit <code>App.tsx</code> and save to test HMR updates.
        </p>
```

```
          <p>
            <a
              className="App-link"
              href="https://reactjs.org"
              target="_blank"
              rel="noopener noreferrer"
            >
              Learn React
            </a>
            {' | '}
            <a
              className="App-link"
              href="https://vitejs.dev/guide/features.html"
              target="_blank"
              rel="noopener noreferrer"
            >
              Vite Docs
            </a>
          </p>
        </header>
      </div>
    );
  };

  export default App;
```

1.2.7　コミット前にlintを自動実行する設定

　これでファイルを保存するたびにlintが走るようになりましたが、念のため、コミット前にもlintが実行されるようにしましょう。まずsimple-git-hooksとlint-stagedをインストールします。

```
npm i -D simple-git-hooks lint-staged
```

package.jsonのscriptを次のように修正します。

```
  "scripts": {
    "dev": "vite",
    "build": "tsc && vite build",
    "preview": "vite preview",
+   "lint": "eslint 'src/**/*.{js,jsx,ts,tsx}'",
+   "lint:fix": "eslint --fix 'src/**/*.{js,jsx,ts,tsx}'",
+   "lint:conflict": "eslint-config-prettier 'src/**/*.{js,jsx,ts,tsx}'",
+   "preinstall": "typesync || :",
+   "prepare": "simple-git-hooks > /dev/null"
  },
```

ついでにlintを手動実行する設定と、npm install時にtypesyncを自動実行する設定も追加しています。

さらに、package.jsonの最後に次の記述を追加しましょう。

```
+ "simple-git-hooks": {
+   "pre-commit": "npx lint-staged"
+ },
+ "lint-staged": {
+   "src/**/*.{js,jsx,ts,tsx}": [
+     "prettier --write --loglevel=error",
+     "eslint --fix --quiet"
+   ],
+   "{public,src}/**/*.{html,gql,graphql,json}": [
+     "prettier --write --loglevel=error"
```

```
+   ]
+ }
```

筆者の開発環境ではsimple-git-hooksの実行モジュールに実行権限が設定されずPermission deniedエラーになる場合がありました。その場合は、次のコマンドで実行権限を付与すれば解決します。

```
chmod +x node_modules/simple-git-hooks/cli.js
```

1.2.8　Tailwind CSSの設定

最後に、Tailwind CSSを設定します。まずはモジュールをインストールしましょう。

```
npm i -D tailwindcss postcss autoprefixer
```

次に、初期処理を実行します。

```
npx tailwindcss init -p
```

これによりtailwind.config.jsとpostcss.config.jsが生成されます。tailwind.config.jsを次のように書き換えましょう。

```
module.exports = {
  content: ['./index.html', './src/**/*.{js,jsx,ts,tsx}'],
  theme: {
    extend: {},
  },
  plugins: [],
};
```

src/index.cssを次の内容で上書きします。

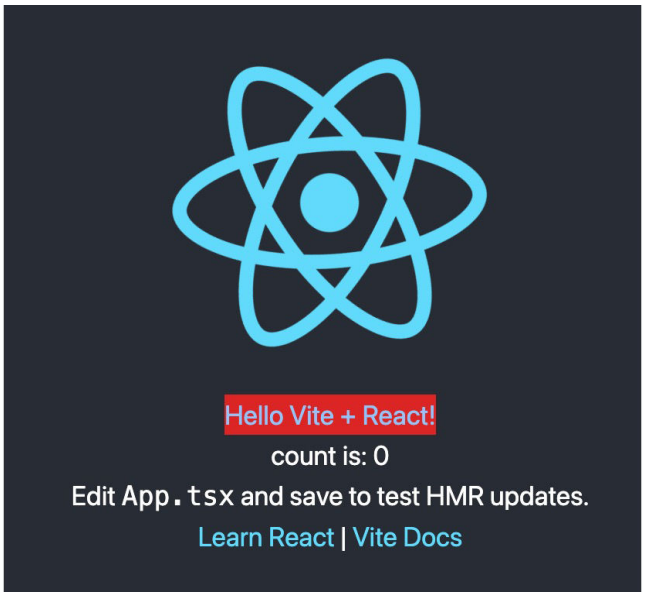
```
@tailwind base;

@tailwind components;

@tailwind utilities;
```

ここで動作確認してみましょう。App.tsxの一部を次のように書き換えてみてください。

```
-       <p>Hello Vite + React!</p>

+       <p className="text-blue-300 bg-red-600">Hello Vite + React!</p>
```

表示が次のようになっていれば、TailwindCSSは適用されています。

図1.2: TailwindCSSの動作確認



classNameの内容を自動でソートするプラグインも入れておきましょう。

```
npm i -D prettier-plugin-tailwindcss
```

このプラグインは設定不要なので、VS Codeを再起動するだけで適用されます。再起動したら、App.tsxをいったん編集状態にしてから保存してください。プラグインが適用されていれば、先ほど修正した箇所が次のように整形されます。classNameの内容がソートされましたね。

```
<p className="bg-red-600 text-blue-300">Hello Vite + React!</p>
```

これでフロントエンドの準備は完了です。お疲れさまでした。この開発環境には基本的なものしか入っていないので、カスタマイズすることでさまざまなプロジェクトに応用できます。

1. 「悩んだわりに普通の構成」とかいわないでください。
2. Reactは正確にはフレームワークではなくライブラリですが、比較の文脈上フレームワークと呼んでいます。
3. https://github.com/sikkimtemi/vite-react-template
4. インターネット上に落ちているソースコードや設定ファイルを、内容を理解しないまま使って痛い目にあう、というパターンを筆者は何度も見てきました。
5. あくまで「単体の」設定です。もう少しESLintの設定は続きます。
6. Prettierの設定はこれだけです。ESLintと比べたらかわいいものですね。

第2章　バックエンドの環境構築

2.1　PythonでもLinterとフォーマッターを使いたい

PythonはJavaScriptほど書き方の自由度が高くないので、Linterやフォーマッターを使わなくてもそこまでひどい表記ゆれは起こりません。実際、筆者はつい最近までVS Codeのデフォルト設定で満足していました。しかし、フロントエンド開発でESLintとPrettierによる自動整形の恩恵に慣れてしまうと、Pythonでも、自動整形機能がほしくなりました。実際には開発終盤で導入した設定ですが、筆者のPython用設定を紹介します。

PythonにはPEP8[1]という、公式のスタイルガイドが存在します。LinterやフォーマッターもPEP8に準拠しているものがほとんどです。ただ、ツールによって微妙な違いがあります。いくつか試したところ、筆者にはFlake8[2]とblack[3]が合っていました。

2.1.1　Flake8の導入

Flake8は次のツールのラッパーです。

・PyFlakes

・pycodestyle

・Ned Batchelder's McCabe script

複数のLinterをまとめて使いやすくしたツールと考えてよいでしょう。インストールは次のようにします。

```
pip install flake8
```

2.1.2　blackの導入

blackはフォーマッターです。設定できるオプションが少ないという特長があります。[4]Pythonのフォーマッターとしてかなり人気があるようです。インストールは次のようにします。

```
pip install black
```

2.1.3　VS Codeの設定

VS CodeのデフォルトのLinterはPylintなので、Flake8に変更する必要があります。VS Code全体の設定を変更してもよいですが、プロジェクトごとに設定したい場合は、プロジェクトルートに.vscode/settings.jsonを配置しましょう。筆者の設定は次のとおりです。1行の文字数以外はデフォルト設定で使っています。

```
{
  "editor.formatOnSave": true,
  "python.linting.flake8Enabled": true,
  "python.linting.flake8Args": ["--max-line-length", "88"],
  "python.formatting.provider": "black",
  "python.formatting.blackArgs": ["--line-length", "88"]
}
```

この設定により、PythonのLinterがFlake8になり、ファイルのセーブ時に自動でフォーマットされます。

2.2　Chaliceを使う準備

本書のバックエンドは、全てサーバレスで構築します。バックエンドのAPIはAWS LambdaとAPI Gatewayを組み合わせたREST APIの構成になっています。Chaliceはこの構成のAPIを簡単に構築できるフレームワークです。AWSの公式ハンズオン資料[5]がわかりやすいので、一度読みながら手を動かしてみることをお勧めします。

2.2.1　Chaliceのインストール

Chaliceのインストールは、次のようにします。

```
pip install chalice
```

2.2.2　AWSの認証情報を設定

Chaliceにはコマンド実行により、ローカルの環境をAWSに自動でデプロイする機能があります。そのため、ローカル環境に認証用のファイルを配置する必要があります。もしAWS CLIを使ったことがある場合は、すでに設定済みのはずです。

まず、AWSで作業用のIAMを表示してください。作業用のIAMがない場合は新規作成してください。本書のWebアプリケーションを開発するには、少なくとも次のポリシーが必要です。必要に応じてポリシーをアタッチもしくはカスタマイズしてください。

・IAMFullAccess

・AmazonDynamoDBFullAccess

・AmazonAPIGatewayAdministrator

・AdministratorAccess-Amplify

・AWSLambda_FullAccess

次に認証情報を開いて、アクセスキーを作成してください。アクセスキーを作成したら、表示またはダウンロードした情報を用いて、~/.aws/credentialsファイルと~/.aws/configファイルを作成してください。

credentialsファイルの例を次に示します。

```
[default]
aws_access_key_id=XXXXXXXXXXXXXXXXXX
aws_secret_access_key=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

configファイルの例を次に示します。

```
[default]
region=ap-northeast-1
output=json
```

以上でバックエンドの準備は完了です。[6]

1. https://peps.python.org/pep-0008/
2. https://pypi.org/project/flake8/
3. https://pypi.org/project/black/
4. Prettierと同じ設計思想を感じますね。
5. https://aws.amazon.com/jp/blogs/startup/event-report-chalice-handson/
6. フロントエンドの環境構築と比べるとあっさりしすぎているので、本章を省略するかかなり迷いました。

第3章　モックアップを作ろう

本章ではモックアップを作ります。モックアップとは、具体的な実装を伴わない見た目だけの画面のことで、これがあると第三者の理解が深まります。口頭や文章ではいくら説明しても理解してもらえなかった機能が、モックアップを見せたら一発で理解してもらえることはよくあります。

開発者にとってもモックアップは役に立ちます。見た目が完成していれば、あとはそこに機能を実装するだけでリリースできますし、残りの作業がどれくらいあるのか見積もりやすくなります。

それに、ReactとTailwind CSSでモックアップを作るのは、とても楽しい作業です。Tailwind CSSのUIコンポーネントをコピー&ペーストして組み立てていくと、あっという間に本格的な画面ができあがっていくので、達成感がありますよ。

3.1　セマンティクスを意識しよう

UIコンポーネントのコードをコピー&ペーストする前に、注意していただきたいことがひとつあります。それは、セマンティクスを意識するということです。セマンティクスとは、コードのもつ「意味」です。コードをコピー&ペーストすれば、見栄えのよいデザインは再現できますが、セマンティクスを意識したコードまでは書けません。HTMLにはセマンティックタグと呼ばれる特別なタグ群が定義されていて、これらを適切に設定すると、コンピュータが理解しやすいコードになります。たとえばメインコンテンツを<main>タグで囲んでおけば、視覚障害者がスクリーンリーダーを通じてメインコンテンツを探しやすくなります。詳しくはMDNドキュメント[1]をお読みください。

私見ですが<header>、<footer>、<main>、<nav>、<section>あたりを適切に使えていれば大丈夫です。

3.2　Tailwind CSSでUIコンポーネントを作る手順

Tailwind CSSのUIコンポーネントをMITライセンスで公開しているページの中から、使い勝手のよさそうなものをピックアップしました。

・Flowrift[2]

・Tailblocks[3]

・Kometa UI Kit[4]

・Tail-kit[5]

・Meraki UI[6]

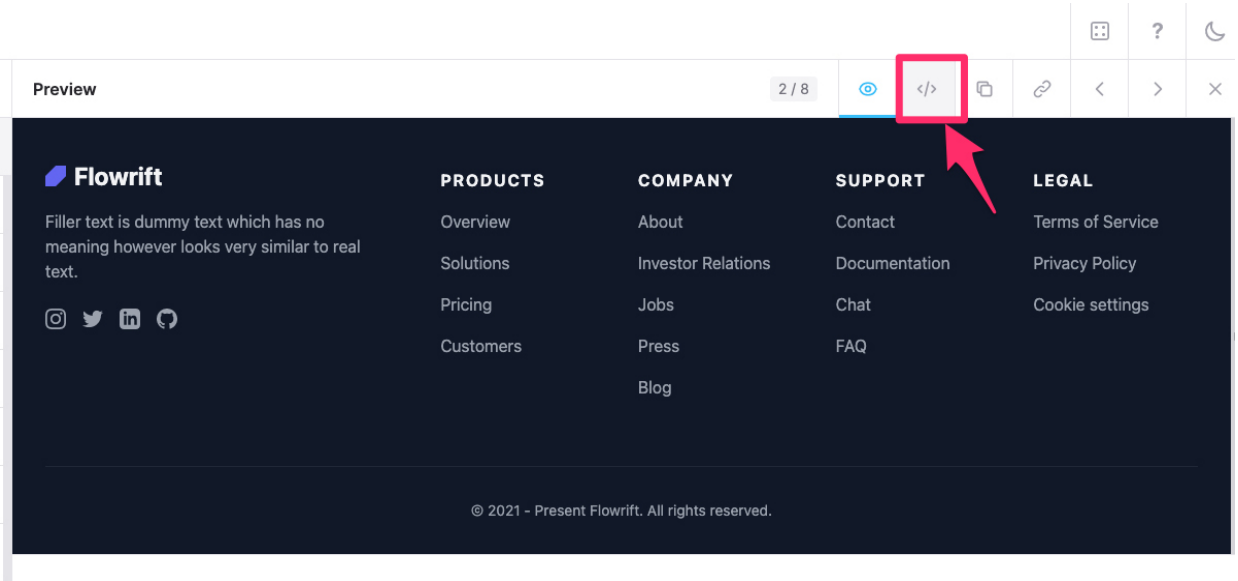
・Preline UI[7]

これらを活用して、モックアップを作っていきましょう。また、さまざまなサービスのUIをまとめて比較できる、Webframe[8]というサイトも役に立ちます。筆者には使いこなせませんでしたが、Figma[9]を使って、事前にデザインしてみてもよいでしょう。

3.2.1　Footerの実装（作業例）

例として、Flowriftに掲載されている次のFooter[10]（図3.1）を実装してみましょう。

図3.1: FlowriftのFooter



矢印で示したアイコンをクリックするとコードが表示されます。

図3.2: Footerのコード

コードが表示されたら、コピーしましょう。

次に、components/Footer.tsxを新規作成します。/* ここにペースト */と書いてあるところに先ほどコピーしたコードをペーストしてください。

```
import { FC } from 'react';


const Footer: FC = () => (
  /* ここにペースト */
);
export default Footer;
```

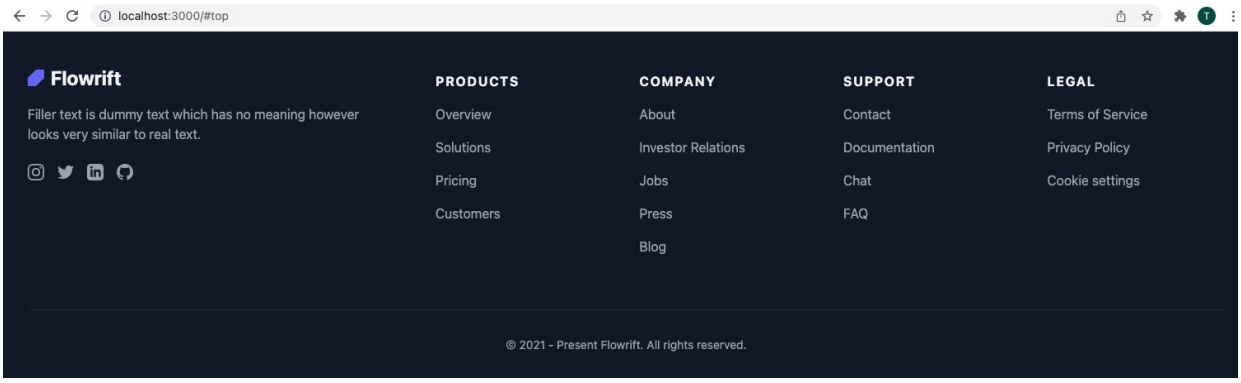
HTML形式をJSX形式に変更するため、class=をclassName=に全置換しましょう。また、<!-- -->形式のコメントは削除します。もし残しておきたいコメントがあったら、JavaScriptのコメント形式に修正しておきましょう。

まだエラーは残っていますが、ここまで修正すれば表示を確認できます。App.tsxを次のように修正してみましょう。

```
import { FC } from 'react';
import Footer from './components/Footer';


const App: FC = () => (
  <footer>
    <Footer />
  </footer>
);


export default App;
```

npm run devで開発サーバーを立ち上げてhttp://localhost:3000にアクセスすると、次のように表示されます。

図3.3: 表示確認



これでFooterが実装できました。

この手順を繰り返していけば、ランディングページくらいならすぐに作れます。ランディングページ完成時点のApp.tsxはこちらです。

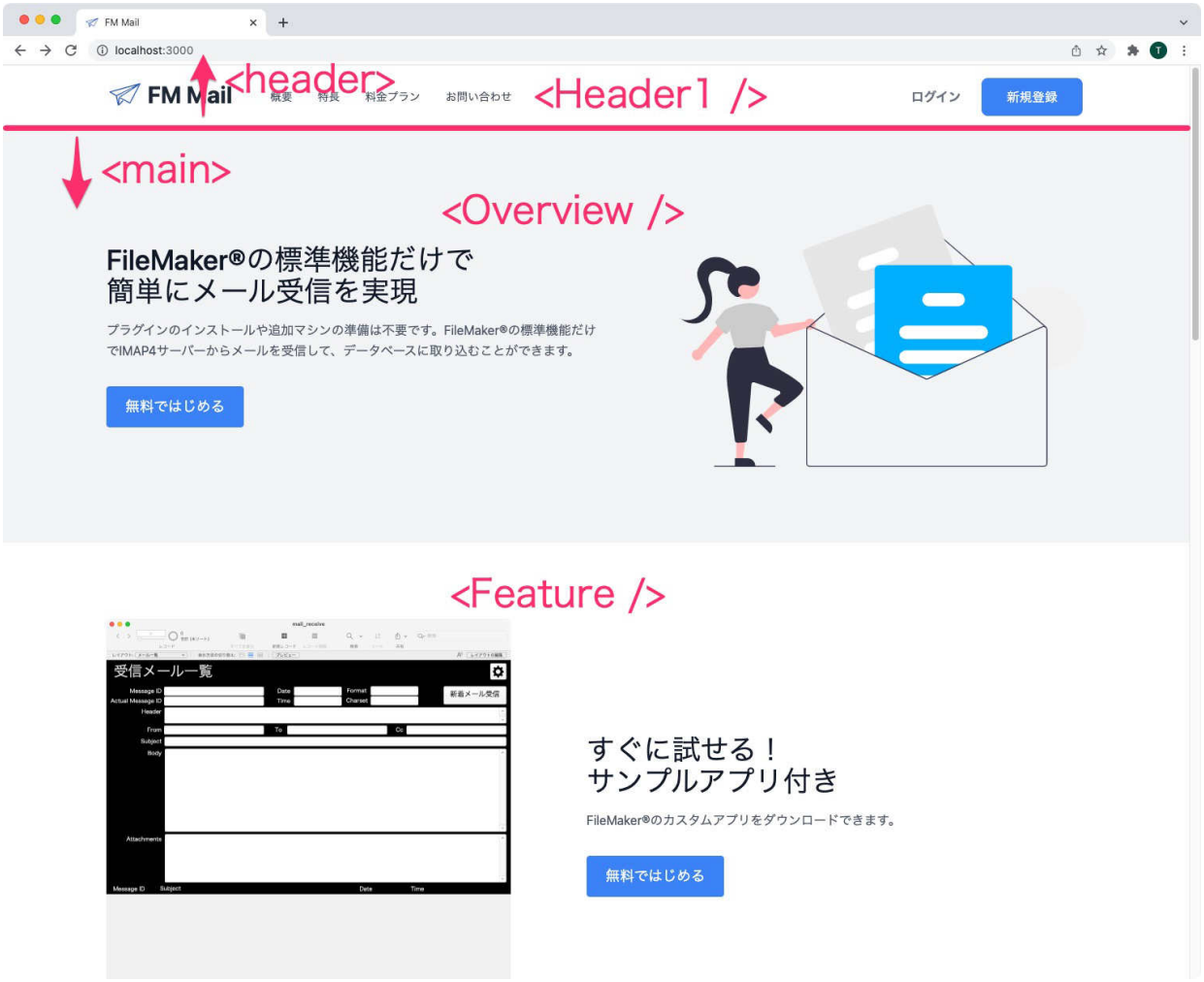
```
import { FC } from 'react';
import Header1 from './components/Header1';
import Footer from './components/Footer';
import Overview from './components/Overview';
import Feature from './components/Feature';
import Price from './components/Price';
import Contact from './components/Contact';

// ランディングページ
const App: FC = () => (
  <>
    <header>
      <Header1 />
    </header>
    <main>
      <Overview />
      <Feature />
      <Price />
```

```
        <Contact />
      </main>
      <footer>
        <Footer />
      </footer>
    </>
  );


  export default App;
```

<header>、<main>、<footer>というセマンティックタグが設定されていることに注目してください。これらのタグを取り除いてもページの見た目はまったく変わりませんが、コンピュータがコンテンツの構造を理解できなくなってしまうので、必ず設定しましょう。

現時点でランディングページの見た目は、次のようになっています。イラストはunDraw[11]を利用しました。個別のコンポーネントの実装は、後述するサンプルコードで確認してください。

図3.4: ランディングページとセマンティックタグ、コンポーネントの関係


FM Mail
localhost:3000
<header>
FM Mail
概要
特長
料金プラン
お問い合わせ
<Header1 />
ログイン
新規登録
<main>
<Overview />
FileMaker®の標準機能だけで
簡単にメール受信を実現
プラグインのインストールや追加マシンの準備は不要です。FileMaker®の標準機能だけでIMAP4サーバーからメールを受信して、データベースに取り込むことができます。
無料ではじめる
<Feature />
受信メール一覧
すぐに試せる！
サンプルアプリ付き
FileMaker®のカスタムアプリをダウンロードできます。
無料ではじめる

3.3　React Routerでルーティングを設定しよう

コピー&ペーストして整形しただけの状態だと、<a>タグのあたりで軒並み次のanchor-is-validというエラーが発生しているはずです。

The href attribute requires a valid value to be accessible. Provide a valid, navigable address as the href value. If you cannot provide a valid href, but still need the element to resemble a link, use a button and change it with appropriate styles.

外部の特定ページへのリンクであれば、hrefを適切に設定することでこのエラーは解消します。一方、Reactアプリケーション内部で画面遷移を実現するには、ルーティングライブラリを導入する必要があります。Reactのルーティングライブラリとしては、React Router[12]が有名です。本書でもReact Routerの最新バージョンであるバージョン6を使用しますが、このライブラリは過去に2回破壊的な変更をしています。1回目はバージョン3から4へ上がったときで、2回目はバージョン5から6へ上がったときです。いずれの場合も設定方法がガラッと変わったので、今後のバージョンアップには気を付けましょう。

また、画面内遷移を可能にするReact Router Hash Link[13]も導入します。それではまとめてインストールしましょう。

```
npm i react-router-dom@6 @xzar90/react-router-hash-link
```

オリジナルのReact Router Hash LinkはReact Routerバージョン6に対応していないので、フォークされたバージョン6対応版をインストールしています。

Viteで構築した場合、ルーティングの設定はmain.tsxに記述します。モックアップ作成時のmain.tsxは次のとおりです。

```
import React from 'react';
import ReactDOM from 'react-dom/client';
```

```
import './index.css';
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import App from './App';
import Canceled from './routes/Canceled';
import CanceledAuth from './routes/CanceledAuth';
import Doc from './routes/Doc';
import Login from './routes/Login';
import MyPage from './routes/MyPage';
import NoMatch from './routes/NoMatch';
import PrivacyPolicy from './routes/PrivacyPolicy';
import Signup from './routes/Signup';
import Thanks from './routes/Thanks';
import ThanksAuth from './routes/ThanksAuth';
import Terms from './routes/Terms';
import Tokusyouhou from './routes/Tokusyouhou';
import ShowApiKey from './routes/ShowApiKey';
import Upgrade from './routes/Upgrade';
import UserInfo from './routes/UserInfo';
import Welcome from './routes/Welcome';

const rootElement = document.getElementById('root');
if (!rootElement) throw new Error('Failed to find the root element');
const root = ReactDOM.createRoot(rootElement);
root.render(
  <React.StrictMode>
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<App />} />
        <Route path="api_key" element={<ShowApiKey />} />
        <Route path="cancel" element={<Canceled message="" />} />
        <Route
```

```
            path="canceled_upgrade"
            element={<CanceledAuth message="アップグレードは" />}
          />
          <Route path="doc" element={<Doc />} />
          <Route path="login" element={<Login />} />
          <Route path="mypage" element={<MyPage />} />
          <Route path="privacy_policy" element={<PrivacyPolicy />} />
          <Route path="signup" element={<Signup />} />
          <Route path="terms" element={<Terms />} />
          <Route path="thanks" element={<Thanks message="お問い合わせ" />} />
          <Route
            path="thanks_upgrade"
            element={<ThanksAuth message="ご契約" />}
          />
          <Route path="tokusyouhou" element={<Tokusyouhou />} />
          <Route path="upgrade" element={<Upgrade />} />
          <Route path="userinfo" element={<UserInfo />} />
          <Route path="welcome" element={<Welcome />} />
          <Route path="*" element={<NoMatch />} />
        </Routes>
      </BrowserRouter>
    </React.StrictMode>,
  );
```

pathとelementの関係ですが、たとえば次の記述ではhttp://localhost:3000/api_keyにアクセスすると、ShowApiKeyコンポーネントの内容が表示されます。

```
<Route path="api_key" element={<ShowApiKey />} />
```

次のようにpathに*を指定している行がありますが、これは404ページ（図3.5）です。

```
<Route path="*" element={<NoMatch />} />
```

マッチするpathが存在しないリクエストは、全てこのNoMatch.tsxに飛ばされます。

図3.5: 404ページ

Reactアプリケーション内部からこれらのpathを呼び出すには、次のように記述します。まずは普通のリンクの例です。

```
<Link to="/signup" >
  新規登録
</Link>
```

次は画面内リンクの例です。

```
<HashLink
  smooth
  to="/#Price"
```

```
>
  料金プラン
</HashLink>
```

toに設定している/#Priceは、ルートのページ内でIDにPriceが設定された箇所を表します。具体的にはsrc/components/Price.tsx内の次の箇所に該当します。また、smoothを設定するとスクロール表現が入るので、画面内遷移だとわかりやすくなります。好みに応じて設定してください。

```
const Price: FC = () => (
  <section id="Price" className="body-font overflow-hidden text-gray-600">
    <div className="container mx-auto px-5 py-24">
```

このid="Price"が画面内リンクのターゲットとなります。

3.4　長い文章はマークダウンで書こう

有料のWebサービスを公開する場合、利用規約とプライバイシーポリシー、特定商取引法に基づく表記が必要です。実際に読む人は少ないですが、いざというときにサービス提供者やユーザーを守る盾になるものです。とても重要なので、必ず作成しましょう。

いずれも長い文章になりがちで、なおかつ利用規約は頻繁に改定されます。いちいちタグを付けて修飾していたら効率が悪いですし、保守性も下がります。そこで、マークダウンで記述できるようにしましょう。Reactでマークダウンを使うにはreact-markdownを使います。それでは、インストールしてみましょう。

```
npm i react-markdown
```

使い方はマークダウンを<ReactMarkdown>で囲むだけです。これでブラウザ出力時にはマークダウンがHTMLに変換されます。react-markdownはデザインについてはノータッチなので、Tailwind CSSで別途設定する必要があります。しかし、タグごとに細かく設定するのは面倒なので、マークダウンを自動で修飾してくれる@tailwindcss/typography[14]を導入します。まずはインストールしましょう。

```
npm i -D @tailwindcss/typography
```

次に、tailwind.config.jsのpluginsに設定を追加します。

```
module.exports = {
  content: ['./index.html', './src/**/*.{js,jsx,ts,tsx}'],
  theme: {
    extend: {},
  },
```

```
  plugins: [require('@tailwindcss/typography')],
};
```

それでは、プラバシーポリシーを作成してみましょう。文章は「Webサイトの利用規約」のひな型[15]をベースにしています。

```
import ReactMarkdown from 'react-markdown';

const body = `
# プライバシーポリシー
_____（以下，「当社」といいます。）は，本ウェブサイト上で提供するサービス（以下，「本サービス」といいます。）における，ユーザーの個人情報の取扱いについて，以下のとおりプライバシーポリシー（以下，「本ポリシー」といいます。）を定めます。

## 第1条（個人情報）
「個人情報」とは，個人情報保護法にいう「個人情報」を指すものとし，生存する個人に関する情報であって，当該情報に含まれる氏名，生年月日，住所，電話番号，連絡先その他の記述等により特定の個人を識別できる情報及び容貌，指紋，声紋にかかるデータ，及び健康保険証の保険者番号などの当該情報単体から特定の個人を識別できる情報（個人識別情報）を指します。

（中略）

## 第10条（お問い合わせ窓口）
本ポリシーに関するお問い合わせは，下記の窓口までお願いいたします。

https://www.example.com/contact

以上
`;
```

```
const PrivacyPolicyContent = () => (
  <section className="bg-white py-6 sm:py-8 lg:py-12">
    <div className="prose mx-auto max-w-screen-md justify-center px-4 md:px-8">
      <ReactMarkdown>{body}</ReactMarkdown>
    </div>
  </section>
);
export default PrivacyPolicyContent;
```

タグのclassNameに設定されたproseという項目が重要です。このクラス名が指定された範囲で、@tailwindcss/typographyが有効になります。文章は変数にベタ書きしていますが、必要に応じて別ファイルに分離するとよいでしょう。

プライバシーポリシーページ全体は次のようにしました。

```
import { FC } from 'react';
import Header1 from '../components/Header1';
import PrivacyPolicyContent from '../components/PrivacyPolicyContent';
import Footer from '../components/Footer';

// プライバシーポリシー
const PrivacyPolicy: FC = () => (
  <>
    <header>
      <Header1 />
    </header>
    <main>
      <PrivacyPolicyContent />
    </main>
    <footer>
      <Footer />
```

```
      </footer>
    </>
  );


  export default PrivacyPolicy;
```

実際の表示はこのようになります。

図3.6: プライバシーポリシー

何もしないでここまできれいに表示されます。

なお、react-markdownは単体ではテーブル表記に対応していません。マークダウンでテーブル表記を使いたい場合はremark-gfmというプラグインを追加する必要があります。src/components/DocContent.tsxで利用しているので、興味のある方はソースコードを確認してください。

3.5　共通するレイアウトをまとめよう

　上述したランディングページとプライバシーポリシーのページの一部を再度表示するので、見比べてみてください。まずはランディングページです。

```
const App: FC = () => (
  <>
    <header>
      <Header1 />
    </header>
    <main>
      <Overview />
      <Feature />
      <Price />
      <Contact />
    </main>
    <footer>
      <Footer />
    </footer>
  </>
);
```

　次はプライバシーポリシーのページです。

```
const PrivacyPolicy: FC = () => (
  <>
    <header>
      <Header1 />
```

```
    </header>
    <main>
      <PrivacyPolicyContent />
    </main>
    <footer>
      <Footer />
    </footer>
  </>
);
```

関数名と<main>タグの中身以外は共通ですね。同じ記述を何度も書くのは効率が悪いので、まとめてしまいましょう。それぞれ次のように書き換えます。まずはランディングページです。

```
const App: FC = () => (
  <PublicLayout>
    <Overview />
    <Feature />
    <Price />
    <Contact />
  </PublicLayout>
);
```

次はプライバシーポリシーのページです。

```
const PrivacyPolicy: FC = () => (
  <PublicLayout>
    <PrivacyPolicyContent />
  </PublicLayout>
);
```

あらたに出てきた<PublicLayout>はsrc/layouts/PublicLayout.tsxで、内容は次のとおりです。

```
import { FC, ReactNode } from 'react';
import Header1 from '../components/Header1';
import Footer from '../components/Footer';

type Props = { children: ReactNode };

const PublicLayout: FC<Props> = ({ children }) => (
  <>
    <header>
      <Header1 />
    </header>
    <main>
      {children}
    </main>
    <footer>
      <Footer />
    </footer>
  </>
);

export default PublicLayout;
```

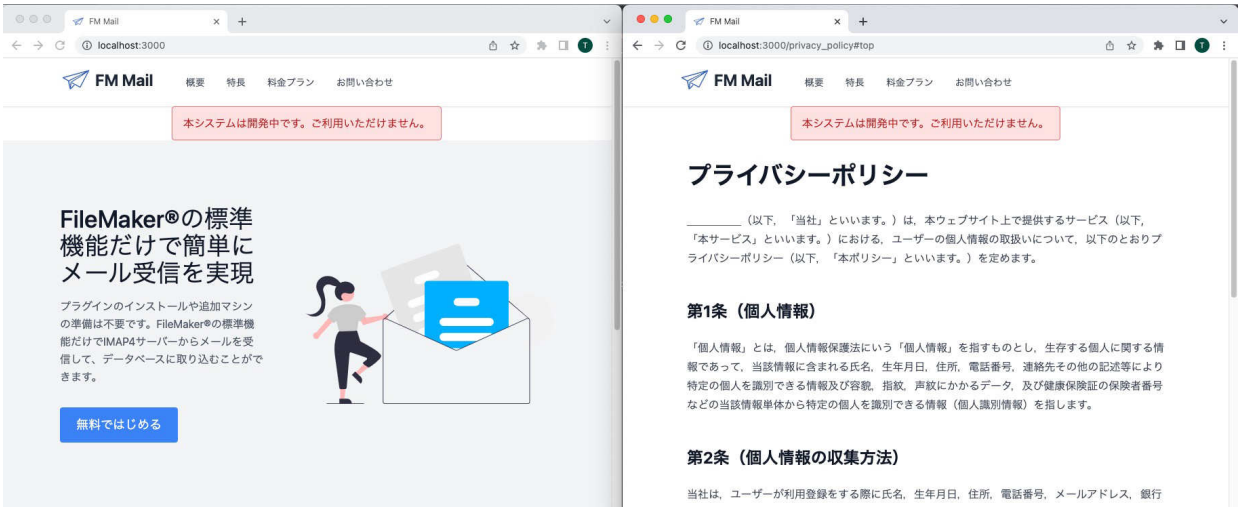
こうすることで、共通するレイアウトを一か所にまとめることができます。本書のWebアプリケーションでは、認証前の画面で使用するPublicLayout.tsxと認証後の画面で使用するAuthenticatedLayout.tsxというふたつのレイアウトを用意しています。

それでは、共通化の恩恵を体感してみましょう。PublicLayout.tsxを次のように書き換えます。内容は注意文言の追加です。

```
import { FC, ReactNode } from 'react';
import Header1 from '../components/Header1';
import Footer from '../components/Footer';

type Props = { children: ReactNode };

const PublicLayout: FC<Props> = ({ children }) => (
  <>
    <header>
      <Header1 />
    </header>
    <main>
+     <div className="flex justify-center text-center">
+       <div
+         className="relative rounded border border-red-400 bg-red-100 px-4 py-3
text-red-700"
+         role="alert"
+       >
+         本システムは開発中です。ご利用いただけません。
+       </div>
+     </div>
      {children}
    </main>
    <footer>
      <Footer />
    </footer>
  </>
);

export default PublicLayout;
```

すると、ランディングページとプライバシーポリシーの表示が両方とも変わります。

図3.7: ランディングページとプライバシーポリシー

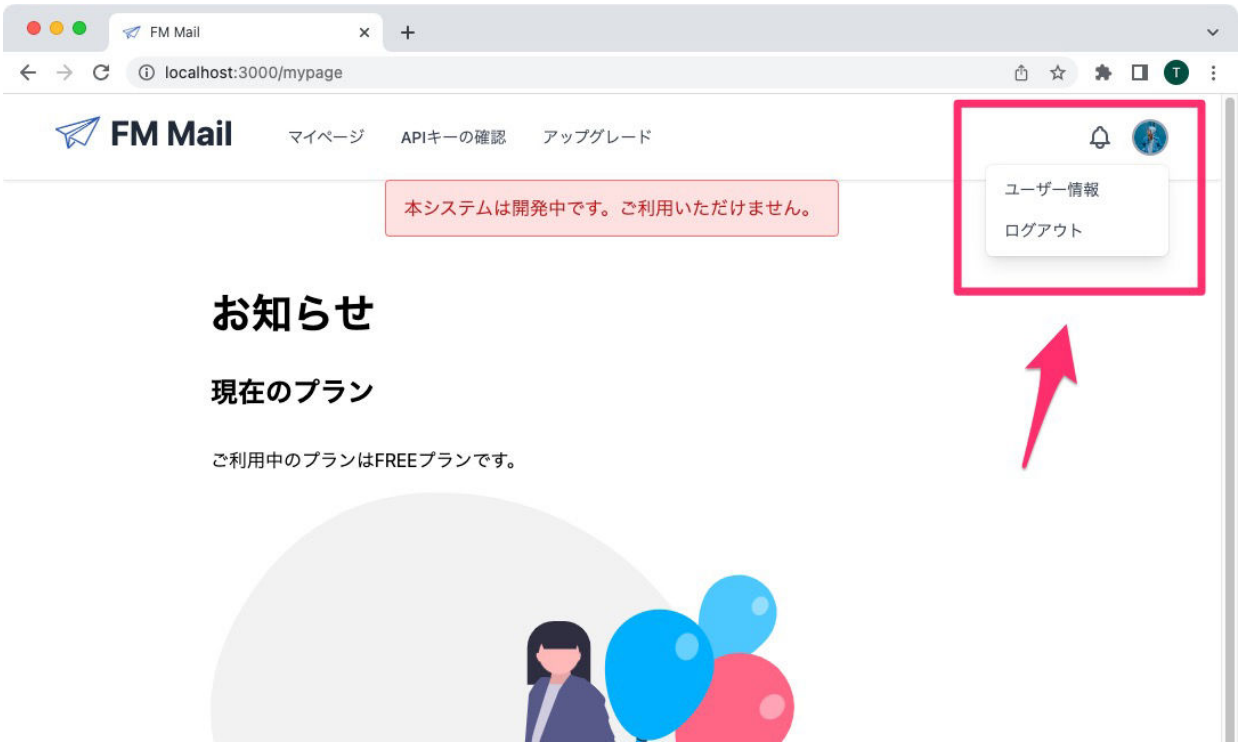


PublicLayout.tsxを利用している画面はほかにもあるので、実際には10個近い画面が同時に修正されます。5章では認証処理を実装しますが、AuthenticatedLayout.tsxを修正することで、同様に複数ページをまとめて修正します。

3.6　プルダウンメニューの表示制御

　サインイン後の画面右上にはユーザーアイコンと通知アイコンがあり、クリックするとプルダウンメニューが開いたり閉じたりします。この部分の実装について説明します。

図3.8: ユーザーアイコンとユーザーメニュー



　Header2.tsxの一部を次に示します。プルダウンの制御はisNoticeOpenとisUserOpenで行っています。

```
const Header2: FC = () => {
  // 通知プルダウンの制御用
  const [isNoticeOpen, setIsNoticeOpen] = useState(false);

  // ユーザーメニュープルダウンの制御用
```

```
  const [isUserOpen, setIsUserOpen] = useState(false);


  // 通知アイコンをクリックしたときの処理
  const handleNoticeClick = () => {
    setIsNoticeOpen((t) => !t);
    setIsUserOpen(false);
  };


  // ユーザーアイコンをクリックしたときの処理
  const handleUserClick = () => {
    setIsNoticeOpen(false);
    setIsUserOpen((t) => !t);
  };


  // コンポーネントの外側をクリックしたときの処理
  const handleOutsideClick = () => {
    setIsNoticeOpen(false);
    setIsUserOpen(false);
  };
```

ユーザーメニューの表示部分は、次のようになっています。

```
<div className="relative inline-block">
  <button
    id="isUserOpenButton"
    type="button"
    className="flex items-center focus:outline-none"
    aria-label="toggle profile dropdown"
    onClick={() => handleUserClick()}
  >
```

```
     <div className="h-8 w-8 overflow-hidden rounded-full border-2 border-gray-
400">
      <img
        src="https://images.unsplash.com/photo-1517841905240-472988babdf9?ixlib=
rb-1.2.1&ixid=eyJhcHBfaWQiOjEyMDd9&auto=format&fit=crop&w=334&q=80"
        className="h-full w-full object-cover"
        alt="avatar"
      />
    </div>
      <h3 className="mx-2 text-sm font-medium text-gray-700 dark:text-gray-200
md:hidden">
      ユーザーメニュー
    </h3>
  </button>
  <div
    className={`absolute right-0 mt-2 w-40 origin-top-right rounded-md bg-white
shadow-lg ring-1 ring-black ring-opacity-5 dark:bg-gray-800 ${
      isUserOpen ? '' : 'hidden'
    }`}
  >
    <ul className="py-1" aria-labelledby="dropdownButton">
      <li>
        <Link
          to="/userinfo"
             className="block w-full py-2 px-4 text-left text-sm text-gray-700
hover:bg-gray-100  dark:text-gray-200  dark:hover:bg-gray-600  dark:hover:text-
white"
        >
          ユーザー情報
        </Link>
      </li>
```

```
      <li>
        <button
          type="button"
             className="block w-full py-2 px-4 text-left text-sm text-gray-700
hover:bg-gray-100 dark:text-gray-200 dark:hover:bg-gray-600 dark:hover:text-
white"
        >
          <Link to="/">ログアウト</Link>
        </button>
      </li>
    </ul>
  </div>
</div>
```

onClickでhandleUserClick()を呼び出している箇所と、isUserOpenのTrue/Falseでclassname内のhiddenを制御している箇所がポイントです。通知アイコンも同様の構造です。ここまでは簡単ですね。

それでは、次の場合はどうすればよいでしょうか。メニューの外側をクリックしたら、プルダウンを閉じたい場合の表示制御です。

図3.9: 外側をクリックしてメニューを閉じたい

まず、表示制御したい範囲全てを<OutsideClickHandler>で囲みます。実際のコードでは、通知メニューとユーザーメニューを全て囲んでいます。

```
<OutsideClickHandler onClickOutside={() => handleOutsideClick()}>
  // ここに表示制御したい内容を記述する
</OutsideClickHandler>
```

src/components/OutsideClickHandler.tsxの内容は次のとおりです。

```
import React, { FC, useEffect, useRef, ReactNode } from 'react';


type Props = {
  children: ReactNode;
  onClickOutside: any;
};
```

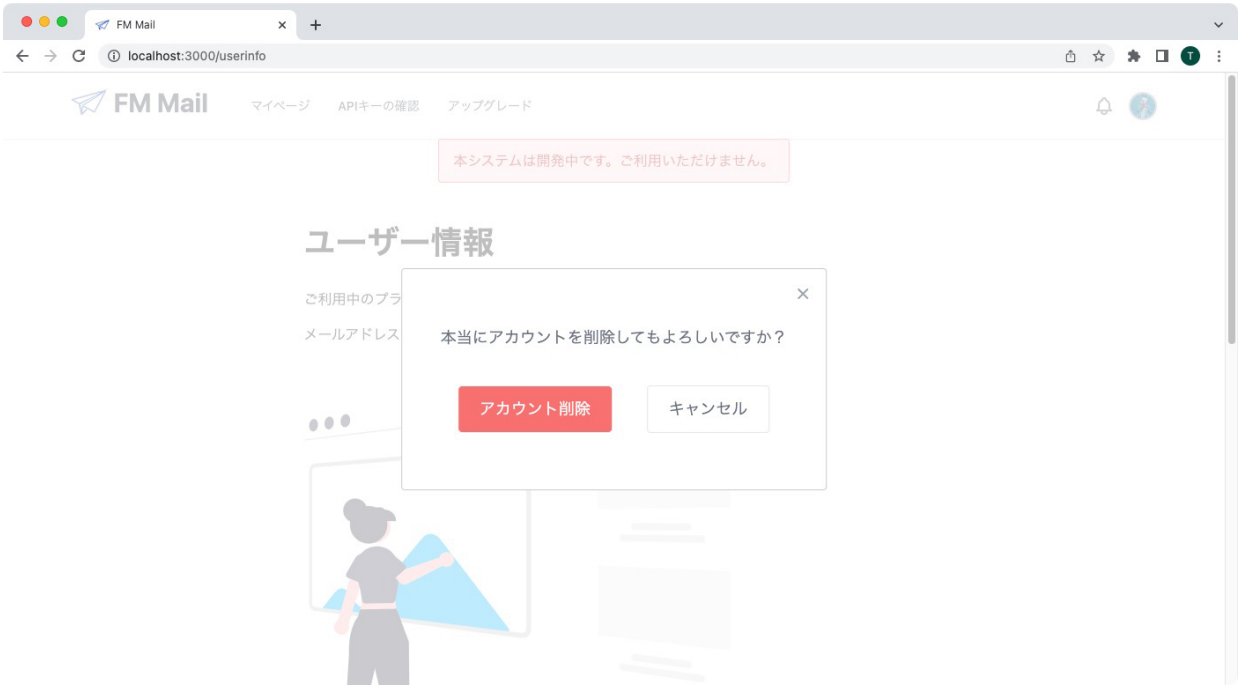
```
// コンポーネントの外側に対するクリックイベントをハンドリングする
const OutsideClickHandler: FC<Props> = ({ children, onClickOutside }) => {
  const ref = useRef<HTMLDivElement>(null);

  useEffect(() => {
    const handleClickOutside = (event: { target: any }) => {
      // eslint-disable-next-line @typescript-eslint/no-unsafe-argument
      if (ref.current && !ref.current.contains(event.target)) {
        // eslint-disable-next-line @typescript-eslint/no-unsafe-call
        onClickOutside();
      }
    };
    document.addEventListener('mousedown', handleClickOutside);

    return () => {
      document.removeEventListener('mousedown', handleClickOutside);
    };
  }, [onClickOutside]);

  return <div ref={ref}>{children}</div>;
};
export default OutsideClickHandler;
```

useRefでdivのDOM要素にアクセスし、currentプロパティとイベントを比較して、divの外側をクリックしたのか、内側をクリックしたのか判断しています。外側をクリックした場合は、propsで渡された関数を実行します。これで任意のコンポーネントの外側をクリックした場合のイベントを制御できるようになりました。

3.7　モーダルの表示制御

次の図は、アカウント削除時の確認ダイアログです。モーダルと呼ばれるUIですね。

図3.10: アカウント削除の確認ダイアログ



モーダルの実装方法はいくつもありますが、本書ではreact-modal[16]を使います。まずはインストールしましょう。

```
npm i react-modal
```

モーダルが使われているのはUserInfoContent.tsxです。表示制御用フラグの定義と表示スタイルの設定箇所は次のとおりです。

```
  // モーダルの表示制御用
  const [isOpen, setIsOpen] = useState<boolean>(false);

  // モーダルの表示スタイル（画面中央に表示）
  const customStyle = {
    content: {
      top: '50%',
      left: '50%',
      right: 'auto',
      bottom: 'auto',
      marginRight: '-50%',
      transform: 'translate(-50%, -50%)',
    },
  };
```

モーダルの表示箇所は次のとおりです。

```
<Modal
  isOpen={isOpen}
  style={customStyle}
  onRequestClose={() => setIsOpen(false)}
  ariaHideApp={false}
>
  <button
    type="button"
        className="absolute top-3 right-2.5 ml-auto inline-flex items-center rounded-lg bg-transparent p-1.5 text-sm text-gray-400 hover:bg-gray-200 hover:text-gray-900 dark:hover:bg-gray-800 dark:hover:text-white"
    onClick={() => setIsOpen(false)}
  >
```

```
    <svg
      className="h-5 w-5"
      fill="currentColor"
      viewBox="0 0 20 20"
      xmlns="http://www.w3.org/2000/svg"
    >
      <path
        fillRule="evenodd"
           d="M4.293 4.293a1 1 0 011.414 0L10 8.586l4.293-4.293a1 1 0 111.414
1.414L11.414 10l4.293 4.293a1 1 0 01-1.414 1.414L10 11.414l-4.293 4.293a1 1 0
01-1.414-1.414L8.586 10 4.293 5.707a1 1 0 010-1.414z"
        clipRule="evenodd"
      />
    </svg>
  </button>
  <div className="p-6 text-center">
     <h3 className="mb-5 mt-5 text-lg font-normal text-gray-500 dark:text-gray-
400">
      本当にアカウントを削除してもよろしいですか？
    </h3>
    <button
      data-modal-toggle="popup-modal"
      type="button"
       className="m-5 inline-flex rounded border-0 bg-red-400 py-3 px-6 text-lg
text-white hover:bg-red-500 focus:outline-none active:bg-red-600"
    >
      アカウント削除
    </button>
    <button
      data-modal-toggle="popup-modal"
      type="button"
```

```
        className="m-5 inline-flex rounded border border-gray-200 bg-white py-3
px-6 text-lg text-gray-500 hover:bg-gray-200 focus:outline-none active:bg-gray-
300 "
      onClick={() => setIsOpen(false)}
    >
      キャンセル
    </button>
  </div>
</Modal>
```

全体を<Modal>で囲み、ボタンにdata-modal-toggleを設定しているのがポイントです。

3.8　サンプルコードを動かしてみよう

　モックアップに使われている技術は以上です。実際にサンプルコード[17]を動かして、確認してみてください。見た目の割に使われている技術は簡単なものばかりです。なお、モックアップの時点では遷移元が存在しない画面もいくつか存在しますが、main.tsxのpathを直接入力すれば表示できます。存在しないpathを指定すると、404ページのNoMatch.tsxが表示されます。

1. https://developer.mozilla.org/ja/docs/Learn/Accessibility/HTML
2. https://flowrift.com/
3. https://tailblocks.cc/
4. https://kitwind.io/products/kometa/components
5. https://www.tailwind-kit.com/components
6. https://merakiui.com/components/
7. https://preline.co/examples.html
8. https://webframe.xyz/
9. https://www.figma.com/
10. https://flowrift.com/c/footer/Zqs6g?view=preview
11. https://undraw.co/search
12. https://reactrouter.com/
13. https://www.npmjs.com/package/@xzar90/react-router-hash-link
14. https://tailwindcss.com/docs/typography-plugin
15. https://kiyaku.jp/hinagata/privacy.html
16. https://www.npmjs.com/package/react-modal
17. https://github.com/sikkimtemi/How_to_create_API_sales_service/tree/main/3-Mockups/fm_mail_frontend

第4章　メール受信APIを作ろう

本章では、FileMaker用のメール受信APIを作成します。API販売サービスの商品にあたるプログラムですね。APIに認証・認可を組み込む方法や、実行回数を制限する方法も説明します。

4.1　APIを販売するとはどういうことか

本書はAPI販売サービスの作り方の本ですが、そもそもAPIを販売するとはどういうことでしょうか。さまざまな形態が考えられますが、本書ではインターネット上にAPIを利用できる環境を用意し、それを利用する権利を販売します。この場合APIはインターネット上にあるため、もし誰でも無制限にAPIを利用できたら、有料版を買ってもらえません。したがって、APIの利用に何らかの制限をかける必要があります。一般的には認証・認可の仕組みを入れて特定の人しか利用できなくしたり、有料版にしかない機能を提供したりします。本書では次の方針としました。

・認可にはAPIキーを用いる
・APIの実行回数に上限を設ける
・評価用の無料版と実運用可能な有料版を用意する
・ユーザー登録で無料版のAPIキーを発行する
・Stripeのサブスクリプション契約で有料版のAPIキーを発行する

無料版と有料版の違いは、シンプルにAPI実行回数だけとしました。メール受信はさほど頻繁に行う必要がないので、無料版の実行回数は月100回、有料版は月10万回としました。それから本書のWebサービスは練習用なので、使っていないときはお金がかからないようにしたいです。

なお、ユーザー登録に連動してAPIキーを自動で発行する機能の実装方法は、6章で説明します。Stripeと連携して有料版のAPIキーを発行する方法は、8章で説明します。

4.1.1　クラウドサービスの技術選定

GCP、AWS、Azureをはじめとして、クラウドサービスはたくさんありますが、上述の条件を全て満たすものは限られます。Google App Engine（GAE）にするか、AWSのLambda + API Gatewayにするかで悩みましたが、最終的に後者を選びました。APIの実行回数制限を簡単に設定できることが決め手となりました。

4.2　Chaliceで楽々デプロイ

APIの実装はChaliceを使います。Chaliceを使うと、本当に簡単にAPIを作ることができます。

4.2.1　メール受信APIの実装

それでは、実際にAPIを作っていきましょう。まずfetch_mail_apiというChaliceプロジェクトを新規作成します。ChaliceのインストールとAWSの認証情報設定が完了していない場合は、2章に戻って環境構築してください。

```
chalice new-project fetch_mail_api
```

プロジェクトのファイル群が生成されたら、app.pyを次のように書き換えてください。

```
import json
from chalice import Chalice
from imap2dict import MailClient



app = Chalice(app_name="fetch_mail_api")



@app.route("/fetch_mail", methods=["POST"], api_key_required=True)
def fetch_mail():
    # リクエストパラメータの解析
    request_params = app.current_request.json_body
    # hostname, user_id, passwordは必須
    host_name = request_params["host_name"]
    user_id = request_params["user_id"]
```

```
    password = request_params["password"]
    # search_optionとtimezoneは省略可能
    search_option = (
        request_params["search_option"]
        if "search_option" in request_params
        else "UNSEEN"
    )
    timezone = (
              request_params["timezone"] if "timezone" in request_params else
"Asia/Tokyo"
    )

    # メールを受信
    cli = MailClient(host_name, user_id, password)
    messages = cli.fetch_mail(search_option=search_option, timezone=timezone)
    resp = {"status": "OK", "messages": messages}

    # 結果を返す
    return json.dumps(resp, ensure_ascii=False)



@app.route("/delete_mail", methods=["DELETE"], api_key_required=True)
def delete_mail():
    # リクエストパラメータの解析
    request_params = app.current_request.json_body
    # hostname, user_id, passwordは必須
    host_name = request_params["host_name"]
    user_id = request_params["user_id"]
    password = request_params["password"]
    # daysは省略可能
    days = request_params["days"] if "days" in request_params else 90
```

```
    # メールを削除
    cli = MailClient(host_name, user_id, password)
    delete_count = cli.delete_mail(days=days)

    # 結果を返す
    return {"delete_count": delete_count}
```

requirements.txtは次のように記述します。

```
imap2dict==0.1.2
```

requirements.txtを作成したらインストールしましょう。

```
pip install -r requirements.txt
```

imap2dictはIMAP4サーバーからメールを取り込み、Pythonの辞書形式に変換するライブラリです。[1]このライブラリは筆者が作成し、PyPIに登録しました。PyPIへの登録手順はZennの記事にしたので、興味のある方はお読みください。[2]

それでは、内容を説明します。最初はアプリケーションの初期化です。fetch_mail_apiという名前にしています。次のようにすると、デバッグモードになります。

```
app = Chalice(app_name="fetch_mail_api")
app.debug = True
```

次はメール受信用のfetch_mailメソッドを定義している箇所です。HTTPのPOSTでアクセスすることとAPIキーが必要であることを宣言しています。

```
@app.route("/fetch_mail", methods=["POST"], api_key_required=True)
def fetch_mail():
```

リクエストパラメータは次のように取得します。

```
    # リクエストパラメータの解析
    request_params = app.current_request.json_body
    # hostname, user_id, passwordは必須
    host_name = request_params["host_name"]
```

IMAPサーバーからメールを受信し、結果を返す処理は次のとおりです。メール受信は結構複雑な処理が必要[3]ですが、面倒な処理は全てimap2dictライブラリに追い出したので、わずかな行数でメールデータをレスポンスに格納して返すことができるようになりました。

```
    # メールを受信
    cli = MailClient(host_name, user_id, password)
    messages = cli.fetch_mail(search_option=search_option, timezone=timezone)
    resp = {"status": "OK", "messages": messages}

    # 結果を返す
    return json.dumps(resp, ensure_ascii=False)
```

メールデータの削除も構造は受信と同じです。削除なので、HTTPメソッドはDELETEにしています。[4]

```
@app.route("/delete_mail", methods=["DELETE"], api_key_required=True)
def delete_mail():
```

リクエストパラメータを取得してライブラリを呼び出すところは同じなので、以降の説明は省略します。

4.2.2　ローカルで動作確認

それでは、ローカルで動作確認してみましょう。

```
chalice local
```

デフォルトでは8000番ポートが使われます。サーバーが起動したらcurlでアクセスしてみましょう。ローカルで動かす場合、認証・認可は不要です。なお、事前にテスト用のメールアカウントを用意しておくと安全です。

```
curl -X POST -H 'Content-Type: application/json' \
-d '{"host_name":"{IMAPサーバーのホスト名}", "user_id":"{メールアカウントのID}", "password":"
{メールアカウントのパスワード}"}' \
http://127.0.0.1:8000/fetch_mail
```

IMAPサーバーに未読メールが存在する場合は、メールデータがJSON形式で返ります。未読メールが存在しない場合は、次のレスポンスが返ります。

```
{"status": "OK", "messages": []}
```

リクエストパラメータは次のとおりです。

リクエストパラメータのキー	説明
host_name	IMAP4サーバーのホスト名。
user_id	IMAP4アカウントのユーザーID。
password	IMAP4アカウントのパスワード。
search_option	省略可。IMAP4の検索オプション。省略時はUNSEENが設定され、未読メッセージのみが取得対象となる。例としてUID 5:8と指定するとUIDが5から8のメールデータが検索対象となる。

timezone	省略可。クライアントのタイムゾーン。省略時はAsia/Tokyoが設定される。

レスポンスは次の形式で返ってきます。

レスポンスのキー	説明
messages	メッセージのデータが配列で格納される。
messages[n].uid	メッセージのUID。
messages[n].subject	メッセージのタイトル。
messages[n].body	メッセージ本文。
messages[n].from	メッセージのFrom。
messages[n].to	メッセージのTo。
messages[n].cc	メッセージのCC。
messages[n].date	メッセージの送信日付。
messages[n].time	メッセージの送信時刻。
messages[n].format	メッセージのフォーマット（text/plainもしくはtext/html）。
messages[n].msg_id	メッセージID。
messages[n].charset	文字コード（utf-8など）。
messages[n].header	メッセージのヘッダ。
messages[n].attachments	添付ファイルのデータが配列で格納される。
messages[n].attachments[m].file_name	添付ファイルのファイル名。
messages[n].attachments[m].file_obj	添付ファイルのオブジェクトがbase64エンコードされたテキストデータ。

メール削除機能もあります。メール削除機能のリクエストパラメータは次のとおりです。

リクエストパラメータのキー	説明
host_name	IMAP4サーバーのホスト名。
user_id	IMAP4アカウントのユーザーID。
password	IMAP4アカウントのパスワード。
days	省略可。数値型で日数を指定するとIMAP4サーバー上からその日数より前に届いたメッセージが削除される。省略時は90が設定される。

メール削除機能のレスポンスは次のとおりです。

レスポンスのキー	説明

delete_count	削除したメッセージの件数。

4.2.3　APIのデプロイ

ローカルで動作確認できたら、AWSにデプロイしましょう。Chaliceでデプロイするのはとても簡単です。

```
chalice deploy
```

デプロイに成功すると、デプロイ先のURLが表示されます。デプロイ先のURLは、次のコマンドでも確認できます。

```
chalice url
```

デプロイすると、次のリソースが自動で作られます。AWSコンソールで確認しておきましょう。

・Lambda関数

・API Gateway

・IAM

4.2.4　APIキーの作成

それでは、デプロイ先のURLにアクセスしてみましょう。ローカルでテストしたときと同じリクエストパラメータで、URLをデプロイ先のURLに変更してcurlコマンドを実行してみてください。ちなみに、デプロイ先のURLはhttps://hogehoge.execute-api.ap-northeast-1.amazonaws.com/apiという形式です。筆者はよく最後の/apiを書き忘れてエラーになるので、注意しましょう。

```
curl -X POST -H 'Content-Type: application/json' \
-d '{"host_name":"{IMAPサーバーのホスト名}", "user_id":"{メールアカウントのID}", "password":"
{メールアカウントのパスワード}"}' \
{デプロイ先のURL}/fetch_mail
```

すると、次のようなレスポンスが返るはずです。

```
{"message":"Forbidden"}
```

fetch_mailメソッドにはapi_key_required=Trueが設定されているため、デプロイ先にアクセスするには、APIキーが必要となります。それでは、APIキーを手動で生成してみましょう。AWSにサインインして、API Gatewayにアクセスしてください。先ほどデプロイしたfetch_mail_apiがあるはずなので、クリックしてください。そして「APIキー」→「アクション」→「APIキーの作成」で適当な名前のAPIキーを作成してください。公式ドキュメント[5]も参考にしてください。

図4.1: fetch_mail_apiの設定

APIキーの「表示」をクリックすると、内容を確認できます。

図4.2: APIキーの表示

4.2.5　使用量プランの作成

APIキーを作成したら、次は使用量プランを作成します。APIを使用量プランに紐付け、使用量プランをAPIキーに紐付けることで、初めてAPIにアクセスできます。使用量プランは図4.3を参考に作成してください。「クォータを有効にする」をチェックして、月のリクエスト数を100回、および100,000回に設定した使用量プランをそれぞれ作成してください。実験用にリクエスト数を極端に少なくした使用量プランも作ってみましょう。

図4.3: 使用量プランの例

次に「APIステージの追加」をクリックして、先ほどデプロイしたAPIと紐付けてください。

使用量プランを作成したらAPIキーの設定に戻り、「使用量プランに追加」をクリックして、先ほど作成した使用量プランを追加してください。

4.2.6　APIキーを用いた動作確認

それでは、作成したAPIキーを試してみましょう。次のcurlコマンドを実行してください。

```
curl -X POST -H 'Content-Type: application/json' \
-H '"x-api-key":"{作成したAPIキー}"' \
```

```
-d '{"host_name":"{IMAPサーバーのホスト名}", "user_id":"{メールアカウントのID}", "password":"
{メールアカウントのパスワード}"}' \
{デプロイ先のURL}/fetch_mail
```

今度は、ローカルで動作確認したときと同じレスポンスが返ったはずです。次はリクエスト回数を極端に少なくした使用量プランとAPIキーを紐付けて、上限に達するまでリクエストしてみましょう。上限に達すると、次のようなレスポンスが返ります。

```
{"message":"Limit Exceeded"}
```

本章で作成した使用量プランは、6章と8章で利用します。手動で作成したAPIキーと実験用の使用量プランはもう使わないので、削除してかまいません。

1. https://pypi.org/project/imap2dict/
2. https://zenn.dev/sikkim/articles/490f4043230b5a
3. メールの解析やUIDの取得、添付ファイルの変換など、実装はかなり面倒でした。
4. RESTful APIのセオリーに従うならメール受信ではGETを使うべきですが、POSTの方がリクエストパラメータの取り回しがよいので、筆者はPOSTを多用しています。
5. https://docs.aws.amazon.com/ja_jp/apigateway/latest/developerguide/api-gateway-setup-api-key-with-console.html

第5章　Cognitoで認証しよう

本章では、3章で作成したモックアップに認証機能を追加します。次の方針に沿って実装していきます。

・Amazon Cognitoを利用する

・Googleアカウントでユーザー登録／サインインする

・IDとパスワードによる認証は行わない

認証機能を実装するときは、個人的に次のことに気を付けています。

・独自実装しない

・個人情報はなるべく持たない

・パスワードを自分で管理しない

認証は本当に難しいので、できるだけCognitoやFirebase Authなど既存の仕組みを利用しましょう。パスワードの平文保存などはもってのほかですが、仮に暗号化していてもリスクは残ります。今回はGoogleにパスワードの管理を押し付けることで、自サービスの責任を減らしています。

図5.1: 認証処理の概要

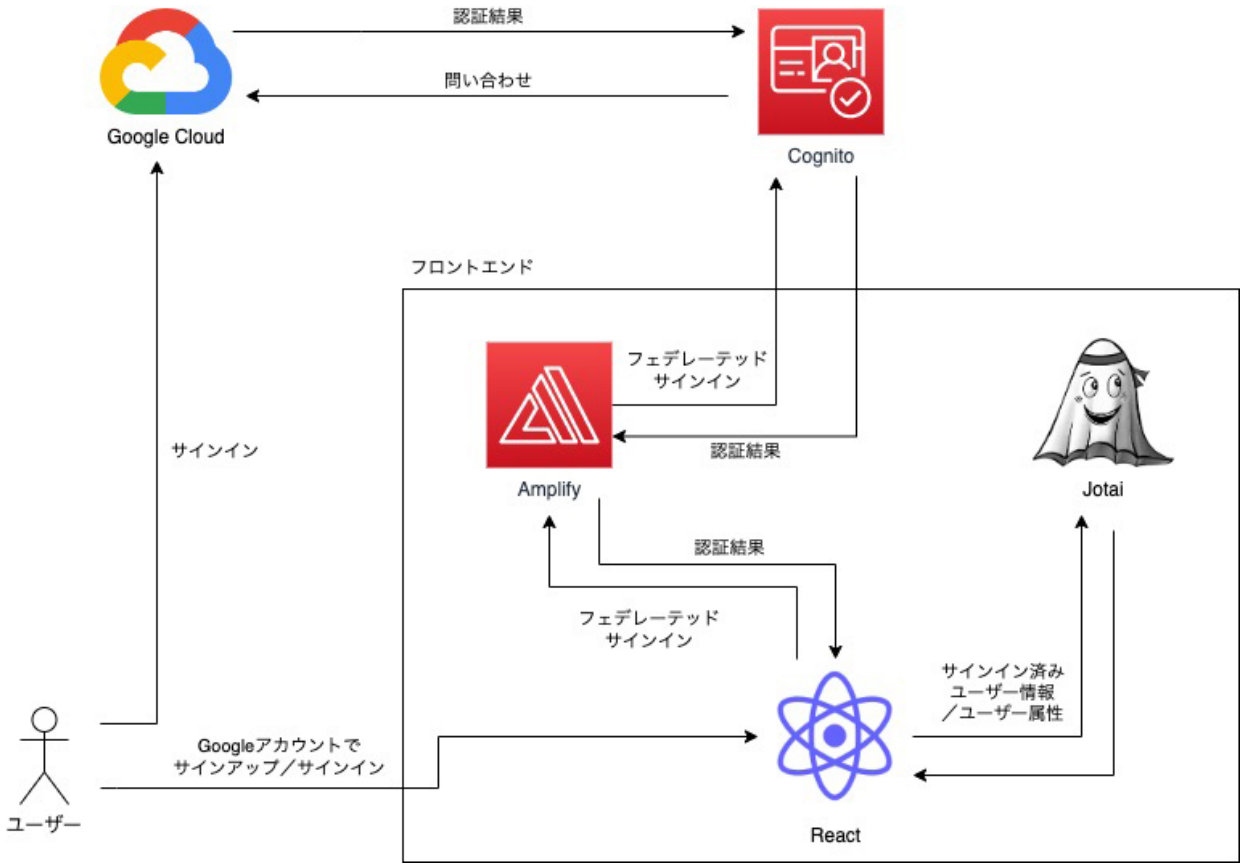


図5.1はこれから実装する認証処理の概要を示しています。実装を進めながらこの図を見返すと、理解が深まるでしょう。

5.1　Cognitoユーザープールを設定しよう

それでは、Cognitoで認証する準備をしましょう。Cognitoで認証するには、ユーザープールを作成する必要があります。ただし、今回はGoogleアカウントでログインするので、先にGCPで同意画面と認証情報を用意します。

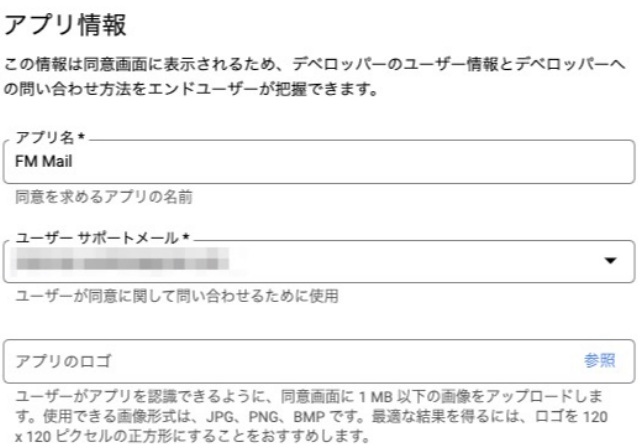
5.1.1　OAuth同意画面の設定

GCPのコンソールで「APIとサービス」→「OAuth同意画面」にアクセスしてください。

図5.2: OAuth同意画面の作成

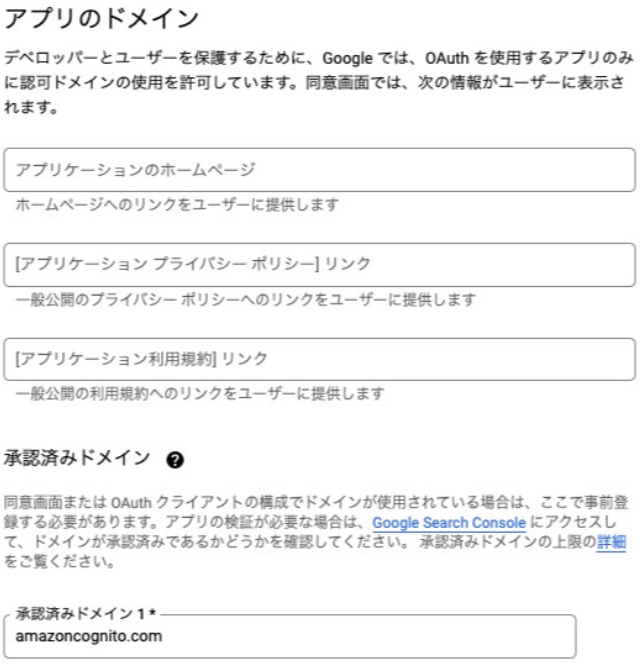
User Typeは「外部」を選択してください。「作成」をクリックすると「アプリ登録の編集」画面が表示されます。項目が多いので、分割して説明します。まずはアプリ情報です。

図5.3: アプリ情報



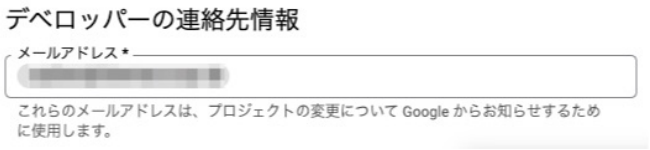
アプリ名は適当に設定してください。ユーザーサポートメールは選択肢から選択してください。次はアプリのドメインです。

図5.4: アプリのドメイン



承認済みドメインにamazoncognito.comを追加してください。ここで設定しなくてもOAuthクライアントIDの作成時に自動的に設定されるので、とばしてもかまいません。最後はデベロッパーの連絡先情報です。

図5.5: デベロッパーの連絡先情報



自分のメールアドレスを入力し、「保存して次へ」をクリックしてください。その次のスコープ画面とテストユーザー画面では何も入力する必要はありません。

5.1.2 OAuthクライアントIDの作成

次は、OAuthクライアントIDを作成します。「認証情報」→「認証情報を作成」→「OAuthクライアントID」でOAuthクライアントIDの作成画面を開いてください。

図5.6: OAuthクライアントIDの作成

「アプリケーションの種類」は「ウェブアプリケーション」を選択してください。すると、次の項目が入力可能になります。

図5.7: OAuthクライアントIDの作成の続き

名前は適当に入力してください。「承認済みのJavaScript生成元」は次のように入力してください。まだCognitoユーザープールは作成していないので、作成予定の名前でかまいません。もしユーザープールを作成済みの場合は「アプリケーションの統合」タブでCognitoドメインを確認できるので、その値を入力してください。

```
https://{Cognitoドメインのプレフィックス}.auth.{AWSのリージョン}.amazoncognito.com
```

「承認済みのリダイレクトURI」は次のように入力してください。

```
{「承認済みのJavaScript生成元」に入力した値}/oauth2/idpresponse
```

OAuthクライアントIDが作成されると、「クライアントID」と「クライアントシークレット」が確認可能になります。これらの値はCognitoユーザープールの設定で使用します。

5.1.3　ユーザープールの作成

次は、Cognitoユーザープールを作成します。AWSのコンソールから「Cognito」→「ユーザープールを作成」で作成を開始してください。[1]最初は「サインインエクスペリエンスを設定」画面が表示されます。

図5.8: サインインエクスペリエンスを設定

サインインエクスペリエンスを設定 情報

アプリケーションユーザーは、ユーザー名とパスワードを使用してユーザープールにサインインするか、サードパーティーのアイデンティティプロバイダーを使用してサインインできます。

認証プロバイダー

ユーザーがサインインするときに利用できるプロバイダーを設定します。

プロバイダーのタイプ

ユーザーが Cognito ユーザープール、フェデレーテッドアイデンティティプロバイダー、またはその両方にサインインするかどうかを選択します。Amazon Cognito では、フェデレーティッドユーザーとユーザープールユーザーの料金が異なります。 料金の詳細はこちら

- [x] Cognito ユーザープール
  ユーザーは E メールアドレス、電話番号、またはユーザー名を使用してサインインできます。ユーザー属性、グループメンバーシップ、およびセキュリティ設定はユーザープールに保存され、ユーザープールに設定されます。
- [x] フェデレーテッドアイデンティティプロバイダー
  ユーザーは、Facebook、Google、Amazon、Apple などのソーシャルアイデンティティプロバイダーの認証情報を使用して、または SAML もしくは Open ID Connect を介して外部ディレクトリの認証情報を使用して、サインインできます。ユーザープールのフェデレーティッドユーザーのユーザー属性マッピングとセキュリティを管理できます。

フェデレーティッドサインインのみを許可しますか?
ユーザープールのディレクトリは、フェデレーティッドユーザーのプロファイルを保存するため、ユーザープールに不可欠です。ユーザープールの自己登録を無効にして、管理者のみがユーザーを作成するようにできます。ユーザープールのサインインを禁止する場合、ユーザープールがアプリケーションクライアントのためのアイデンティティプロバイダーとして選択されているのを解除できます。

Cognito ユーザープールのサインインオプション 情報

サインインに使用するユーザープールの属性を選択します。属性を 1 つだけ選択した場合、またはユーザー名と 1 つ以上のその他の属性を選択した場合、ユーザーは選択したすべてのオプションを使用してサインインできます。電話番号と E メールのみを選択した場合、サインアップ時に 2 つのサインインオプションのいずれかを選択するよう求められます。

- [ ] ユーザー名
- [x] E メール
- [ ] 電話番号

⚠ ユーザープールの作成後に Cognito ユーザープールのサインインオプションを変更することはできません。

フェデレーティッドサインインのオプション 情報

設定するサードパーティーのアイデンティティプロバイダーを選択します。フェデレーションプロバイダーはいつでも追加または削除できます。

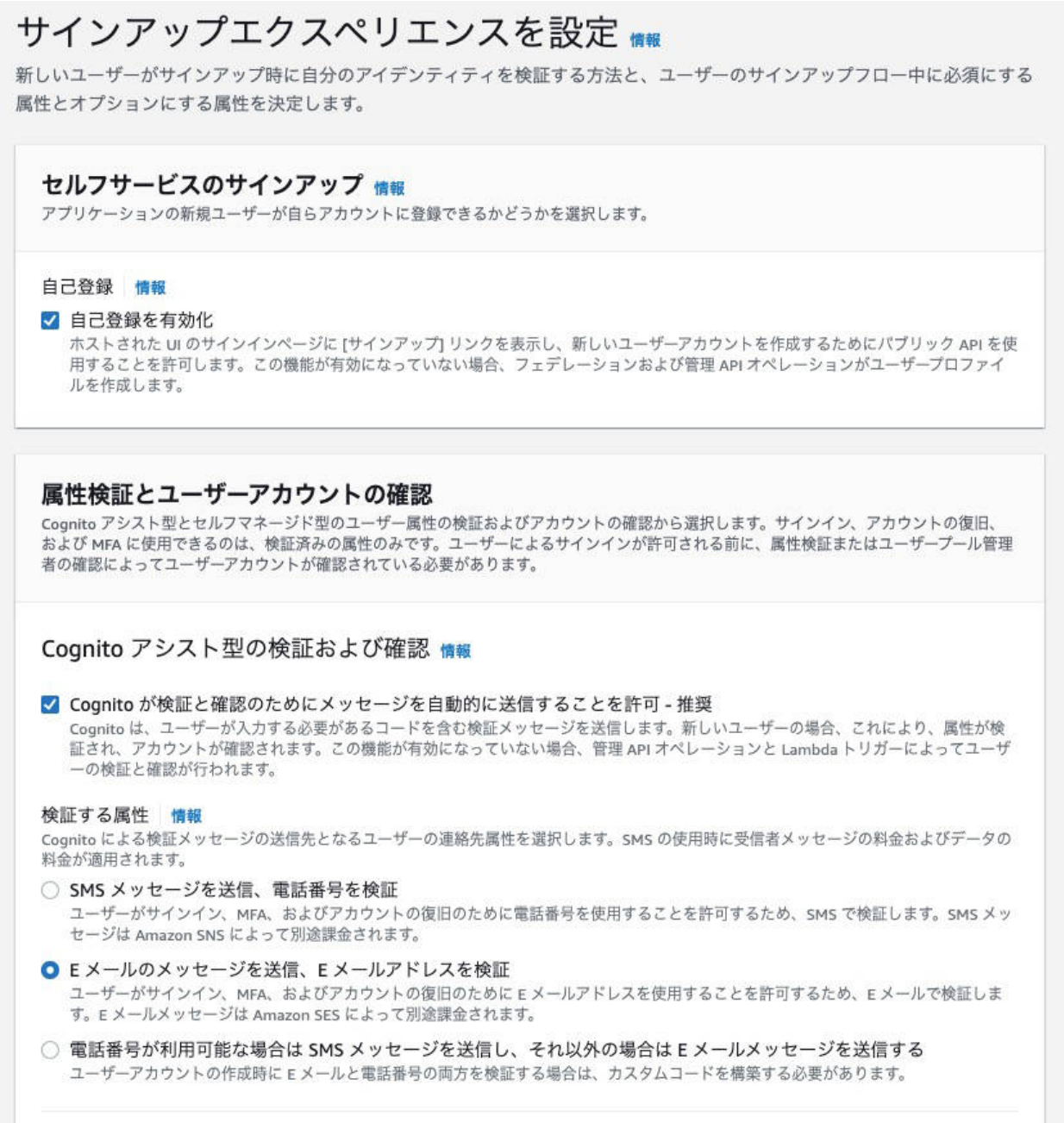
- [ ] **Facebook**
  Facebook アカウントでのサインインを許可します。
- [x] **Google**
  Google アカウントでのサインインを許可します。
- [ ] **Login with Amazon**
  Amazon アカウントでのサインインを許可します。

「フェデレーテッドアイデンティティプロバイダー」と「Eメール」および「Google」をチェックしてください。次は「セキュリティ要件を設定」画面です。

図5.9: セキュリティ要件を設定

パスワードポリシーモードは「Cognitoのデフォルト」にし、多要素認証は「MFAなし」にします。今回はGoogleアカウントのみで認証するので、多要素認証は不要です。[2]次は「サインアップエクスペリエンスを設定」画面です。

図5.10: サインアップエクスペリエンスを設定



「自己登録を有効化」と「Cognitoが検証と確認のためにメッセージを自動的に送信することを許可」をチェックし、「Eメールのメッセージを送信、Eメールアドレスを検証」を選択してください。次は「メッセージ配信を設定」画面です。

図5.11: メッセージ配信を設定

「CognitoでEメールを送信」を選択してください。次は「フェデレーテッドプロバイダーを接続」画面です。

図5.12: フェデレーテッドプロバイダーを接続

フェデレーテッドアイデンティティプロバイダーを接続 情報

フェデレーテッドアイデンティティプロバイダーで認証を設定し、プロバイダーのレスポンスと Cognito トークンの間の属性マッピングを設定します。

Google

Google をユーザープールのアイデンティティプロバイダーとして設定します。アプリケーションの登録と作成には約 10 分かかります。

後で

アプリケーションを Google に登録

Google でデベロッパーアカウントを登録して、プロジェクトと OAuth 2.0 認証情報を作成します。Google のクライアント ID とクライアントシークレットを使用して、ユーザープールのフェデレーションの設定を完了します。 登録手順

このユーザープールで Google フェデレーションを設定

クライアント ID 情報

Google が提供するクライアント ID を入力します。

クライアントのシークレット 情報

Google が提供するクライアントのシークレットを入力します。

許可されたスコープ 情報

Google からリクエストする OAuth 2.0 のスコープを選択します。スコープは、アプリケーションとやりとりを行う Google ユーザー属性のグループです。デフォルトでは、このユーザープールとのフェデレーションに必要なスコープが選択されています。必要に応じて、さらにスコープを追加できます。

profile email openid

スコープはスペースで区切ります。

Google とユーザープール間で属性をマッピング 情報

必須の属性は同等の Google 属性にマッピングされます。追加する各属性は、Google 属性にマッピングする必要があります。

ユーザープール属性	Google 属性
email	email

別の属性を追加

クライアントIDとクライアントシークレットには、「5.1.2 OAuthクライアントIDの作成」で作成したクライアントIDとクライアントシークレットをそれぞれ入力してください。「許可されたスコープ」にはprofile email openidと入力してください。ユーザープール属性のemailに対応するGoogle属性はemailを選択してください。次は「アプリケーションを統合」画面です。長いので前後に分割します。

図5.13: アプリケーションを統合（前半）

アプリケーションを統合 情報

Cognito の組み込みの認証および承認フローを使用して、ユーザープールのためにアプリケーション統合を設定します。

ユーザープール名

ユーザープールのわかりやすい名前を作成します。

ユーザープール名

fm_mail_test

ユーザープール名は 128 文字以下である必要があります。名前には、英数字、スペース、+ = , . @ - といった特殊文字のみを使用できます。

⚠ このユーザープールを作成すると、ユーザープール名を変更できなくなります。

ホストされた認証ページ

ユーザーのサインアップおよびサインインフローのために Cognito のホストされた UI と OAuth 2.0 サーバーを使用するかどうかを選択します。

☑ Cognito のホストされた UI を使用

ホストされたサインアップ、サインイン、および OAuth 2.0 サービスエンドポイントを Amazon Cognito で構築します。この機能が有効になっていない場合は、Cognito API オペレーションを使用してサインアップとサインインを実行します。

フェデレーティッドサインインを使用するユーザープールでは、ホストされた UI で構築された OAuth 2.0 エンドポイントを使用する必要があります。 詳細はこちら

ドメイン 情報

ホストされた UI および OAuth 2.0 エンドポイントのドメインを設定します。ホストされた UI を使用するには、認証エンドポイントが作成されるドメインを選択する必要があります。

ドメインタイプ

◉ Cognito ドメインを使用する

Amazon が所有するドメインで使用する識別プレフィックスを入力します。本番稼働用アプリケーションの場合は、代わりにカスタムドメインを使用することをお勧めします。

○ カスタムドメインを使用

Cognito がホストするサインアップページとサインインページで所有するドメインを入力します。カスタムドメインを使用するには、DNS レコードと AWS Certificate Manager (ACM) 証明書を指定する必要があります。本番稼働用ワークロードにはカスタムドメインを使用することをお勧めします。

Cognito ドメイン

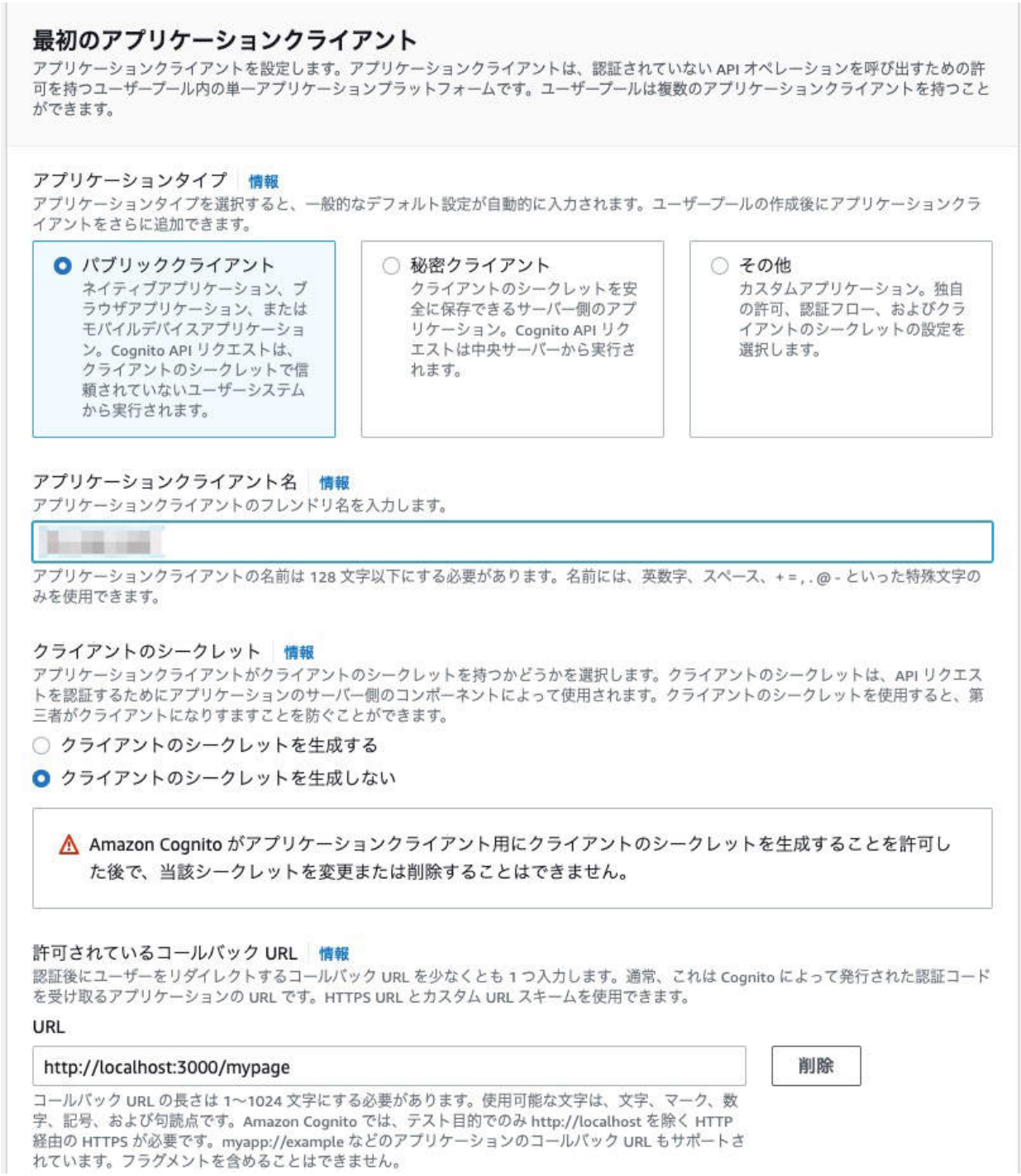
ドメインプレフィックスを入力します。

https:/[redacted] .auth.ap-northeast-1.amazoncognito.com

ドメインプレフィックスには、小文字、英数字、ハイフンのみを使用できます。ドメインプレフィックスには aws、amazon、または cognito というテキストを使用することはできません。ドメインプレフィックスは、現在のリージョン内で一意である必要があります。

ユーザープール名は適当に入力してください。ドメインタイプは「Cognitoドメインを使用する」を選択してください。Cognitoドメインが「5.1.2 OAuthクライアントIDの作成」で「承認済みのJavaScript生成元」に設定した値と同じになるように、ドメインプレフィックスを入力してください。もし予定していた値が使用できない場合は、ほかの値に変更したうえでGCP側のOAuthクライアントIDの設定を修正してください。次は「アプリケーションを統合」画面の後半です。

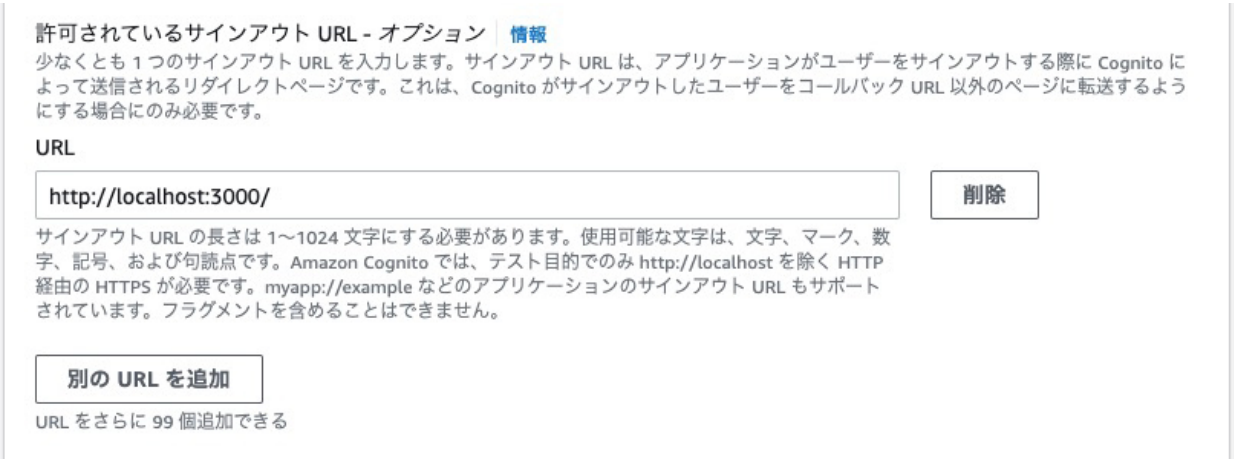
図5.14: アプリケーションを統合（後半）



アプリケーションタイプは「パブリッククライアント」を選択してください。アプリケーションクライアント名は適当に入力してください。また、「クライアントのシークレットを生成しない」を選択してください。「許可されているコールバックURL」にはhttp://localhost:3000/mypageと入力してくださ

い。httpsではなくhttpなので、間違えないようにしましょう。さらに「高度なアプリケーションクライアントの設定」をクリックして、下の方までスクロールしてください。

図5.15: 高度なアプリケーションクライアントの設定



「サインアウトURLを追加」をクリックし、http://localhost:3000/と入力してください。これでユーザープールの設定はいったん完了です。そのままユーザープールを作成してください。

5.2　Amplifyで認証処理を実装しよう

それでは、フロントエンド側で認証処理を実装しましょう。まずAmplifyをインストールします。

```
npm i aws-amplify
```

また、状態管理ライブラリのJotaiもインストールします。

```
npm i jotai
```

AmplifyとはWebアプリケーションを構築するためのオープンソースの開発プラットフォームで、Amazonが開発しています。AWSのさまざまなサービスを統合して、自動で構築してくれるという点ではChaliceと似ていますが、Amplifyはもっと複雑です。本書ではAmplifyをCognitoに接続するためのライブラリとしてのみ使用します。[3]Jotaiについては後述します。

5.2.1　Amplify設定の読み込み

Amplifyの設定は、src/awsExports.tsに記述します。

```
const awsExports = {
  Auth: {
    region: import.meta.env.VITE_REGION,
    userPoolId: import.meta.env.VITE_USER_POOL_ID,
    userPoolWebClientId: import.meta.env.VITE_USER_POOL_WEB_CLIENT_ID,
    oauth: {
      domain: import.meta.env.VITE_OAUTH_DOMAIN,
      scope: ['openid'],
      redirectSignIn: import.meta.env.VITE_OAUTH_REDIRECT_SIGN_IN,
```

```
      redirectSignOut: import.meta.env.VITE_OAUTH_REDIRECT_SIGN_OUT,
      responseType: 'code',
    },
  },
};

export default awsExports;
```

設定値は環境変数に格納します。ローカルでテストする場合は.env.localファイルを用意しましょう。サンプルコード[4]に.env.local.templateというテンプレートファイルを用意したので、コピーして利用してください。

```
cp .env.local.template .env.local
```

テンプレートは次のとおりです。

```
VITE_OAUTH_DOMAIN=xxxxxxxxxxx.auth.ap-xxxxxxxxxx-1.amazoncognito.com
VITE_OAUTH_REDIRECT_SIGN_IN=http://localhost:3000/mypage
VITE_OAUTH_REDIRECT_SIGN_OUT=http://localhost:3000/
VITE_REGION=ap-xxxxxxxxxx-1
VITE_USER_POOL_ID=ap-xxxxxxxxxx-1_xxxxxxxxx
VITE_USER_POOL_WEB_CLIENT_ID=xxxxxxxxxxxxxxxx
```

VITEで環境変数を利用するには、このようにVITE_という接頭語をつける必要があります。各項目の説明は次のとおりです。

環境変数	説明
VITE_OAUTH_DOMAIN	「アプリケーションの統合」タブのCognitoドメインを設定する。
VITE_OAUTH_REDIRECT_SIGN_IN	サインイン後のリダイレクト先。ローカル環境の場合は http://localhost:3000/mypageを設定する。

VITE_OAUTH_REDIRECT_SIGN_OUT	サインアウト後のリダイレクト先。ローカル環境の場合は http://localhost:3000/を設定する。
VITE_REGION	AWSのリージョン。東京リージョンなら ap-northeast-1を設定する。
VITE_USER_POOL_ID	CognitoユーザープールのID。「ユーザープールの概要」で確認可能。
VITE_USER_POOL_WEB_CLIENT_ID	アプリケーションクライアントのID。「アプリケーションの統合」タブで確認可能。

このままだとawsExports.tsでTypeScriptのエラーが出るので、vite-env.d.tsを次のように書き換えましょう。

```
/// <reference types="vite/client" />


interface ImportMetaEnv {
  readonly VITE_OAUTH_DOMAIN: string;
  readonly VITE_OAUTH_REDIRECT_SIGN_IN: string;
  readonly VITE_OAUTH_REDIRECT_SIGN_OUT: string;
  readonly VITE_REGION: string;
  readonly VITE_USER_POOL_ID: string;
  readonly VITE_USER_POOL_WEB_CLIENT_ID: string;
}


interface ImportMeta {
  readonly env: ImportMetaEnv;
}
```

これでエラーは解消し、エディタ上で補完も効くようになります。

最後にAmplifyの設定を読み込むため、main.tsxに次の記述を追加します。

```
Amplify.configure(awsExports);
```

5.2.2　ViteでAmplifyを動かすための設定

この段階でnpm run devを実行し、ローカル環境で動作確認すると、画面が真っ白になります。コンソールを確認すると、Uncaught ReferenceError: global is not definedというエラーが出ているはずです。この問題の解決方法は公式ページに記載されています。[5]まずindex.htmlを次のように書き換えてください。

```
<!DOCTYPE html>
<html lang="ja">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/src/favicon.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta name="robots" content="noindex">
    <title>FM Mail</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.tsx"></script>
+   <script>
+     window.global = window;
+     window.process = {
+       env: { DEBUG: undefined },
+     };
+     var exports = {};
+   </script>
  </body>
</html>
```

これで上述のエラーは解消します。次はビルド時のエラーを解消します。現時点でnpm run buildを実行すると'request' is not exported byで始まるエラーによりビルドが失敗します。この

エラーを解消するにはvite.config.tsを次のように書き換えます。

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
+ resolve: {
+   alias: [
+     {
+       find: './runtimeConfig',
+       replacement: './runtimeConfig.browser',
+     },
+   ]
+ },
});
```

公式ページではさらにtsconfig.jsonでcompilerOptionsのskipLibCheckをtrueにするという記載があります。この設定を省略しても、本書のアプリケーションは問題なく動きます。型チェックはなるべくスキップしたくないので、筆者は無視しています。

5.2.3 サインインとサインアウト

Amplifyの設定を読み込んで正常に起動できるようになったので、次はサインインとサインアウトを実装しましょう。サインインは次のように実装します。

```
import { Auth } from 'aws-amplify';

...
```

```
    <button
      type="button"
      onClick={() => Auth.federatedSignIn({ customProvider: 'Google' })}
    >
      Googleアカウントでログイン
    </button>

  ...
```

Google等の外部のソーシャルIDでサインインする場合は、このようにfederatedSignInを使用します。簡略化しているので、サンプルコードで実際の使い方を確認してください。この処理はsrc/components/LoginForm.tsxとsrc/components/SignupForm.tsxに実装されています。

federatedSignInはサインアップも兼ねています。ユーザーが登録されていない状態で実行すると、自動的に登録とサインインが完了します。UI上は新規登録画面とログイン画面を分けていますが、内部的には同じ処理を呼んでいます。

サインアウトは次のように実装します。

```
import { Auth } from 'aws-amplify';

...

    <button
      type="button"
      onClick={() => Auth.signOut()}
    >
      ログアウト
    </button>

...
```

対応するサンプルコードはsrc/components/Header2.tsxです。

これでサインインとサインアウトは実装できました。いったん動作確認してみましょう。ログイン画面を表示して、「Googleアカウントでログイン」をクリックしてください。

図5.16: ログイン

複数のGoogleアカウントでサインインしている場合は、次のようにアカウントの選択画面が表示されます。

図5.17: アカウントの選択

アカウントを選択すると次のマイページに遷移します。単一のGoogleアカウントでサインインしている場合は、直接マイページに遷移します。

図5.18: マイページ

federatedSignInメソッドには遷移先を記載していないのに、マイページに遷移するのが不思議ではありませんか。これはサインイン後の遷移先がsrc/awsExports.tsのredirectSignInから読み込まれているためです。redirectSignInは環境変数で設定しているので、環境変数のVITE_OAUTH_REDIRECT_SIGN_INに設定されたhttp://localhost:3000/mypageがサインイン後の遷移先になります。

次はサインアウトしてみましょう。ユーザーアイコンをクリックしてメニューを開き、「ログアウト」を選択してください。

図5.19: ログアウト

サインアウトすると、ランディングページに遷移します。

図5.20: サインアウト後の遷移先

画面遷移のメカニズムはサインインと同様です。src/awsExports.tsのredirectSignOutに設定された環境変数VITE_OAUTH_REDIRECT_SIGN_OUTが遷移先となります。今はローカル環境で実行しているのでhttp://localhost:3000/を設定していますが、デプロイ後はインターネットからアクセス可能なURLを設定することになります。詳しくは7章で説明します。

5.2.4　保護したいページにアクセス制限をかけよう

サインインとサインアウトはできるようになりましたが、現時点ではURLを直打ちすれば、全てのページにアクセスできる状態です。誰でも自由にマイページやAPIキーの確認ページを表示できてしまうのはよくありません。保護したいページにサインインなしでアクセスした場合は、ログイン画面へ遷移するようにしましょう。

3章で作成したモックアップにはあらかじめsrc/layouts/AuthenticatedLayout.tsxというファイルが用意されており、保護対象のページから呼び出されています。現時点では次のようになっています。

```
import { FC, ReactNode } from 'react';
import Header2 from '../components/Header2';
import Footer from '../components/Footer';

type Props = { children: ReactNode };

const AuthenticatedLayout: FC<Props> = ({ children }) => (
  <>
    <header>
      <Header2 />
    </header>
    <main>
      <div className="flex justify-center text-center">
        <div
          className="relative rounded border border-red-400 bg-red-100 px-4 py-3
text-red-700"
          role="alert"
        >
          本システムは開発中です。ご利用いただけません。
        </div>
      </div>
      {children}
```

```
    </main>
    <footer>
      <Footer />
    </footer>
  </>
);

export default AuthenticatedLayout;
```

これを次のように書き換えます。

```
import { FC, ReactNode, useEffect, useState } from 'react';
import { Auth } from 'aws-amplify';
import { useAtom } from 'jotai';
import { Navigate } from 'react-router-dom';
import Header2 from '../components/Header2';
import Footer from '../components/Footer';
import Spinner from '../components/Spinner';
import stateCurrentUser from '../atom/User';
import type { CognitoUser } from '../atom/User';

type Props = { children: ReactNode };

const AuthenticatedLayout: FC<Props> = ({ children }) => {
  // サインイン中のユーザー情報
  const [user, setUser] = useAtom(stateCurrentUser);

  // 読込中フラグ
  const [isLoading, setIsLoading] = useState<boolean>(true);
```

```
// 要ログインフラグ
const [loginRequired, setLoginRequired] = useState<boolean>(false);

// サインイン済みかどうかチェックする
useEffect(() => {
  // awaitを扱うため、いったん非同期関数を作ってから呼び出している
  const checkSignIn = async () => {
    try {
      // サインイン済みのユーザー情報を取得する
      // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
      const currentUser: CognitoUser = await Auth.currentAuthenticatedUser();
      // ユーザー情報をJotaiで管理（これをトリガーにもうひとつのEffect Hookが動く）
      setUser(currentUser);
    } catch (e) {
      // 認証に失敗したらサインアウトしてからログイン画面に遷移させる
      await Auth.signOut();
      setLoginRequired(true);
    }
  };

  // Promiseを無視して呼び出すことを明示するためvoidを付けている
  void checkSignIn();
}, [setUser]);

// サインイン済みチェックが終わったらローディング表示をやめる
useEffect(() => {
  if (user || loginRequired) setIsLoading(false);
}, [user, loginRequired]);

// ローディング表示
if (isLoading) {
```

```
    return (
      <main>
        <Spinner />
      </main>
    );
  }

  // 要ログインの場合はログイン画面に遷移
  if (loginRequired) {
    return <Navigate to="/login" replace />;
  }

  return (
    <>
      <header>
        <Header2 />
      </header>
      <main>
        <div className="flex justify-center text-center">
          <div
              className="relative rounded border border-red-400 bg-red-100 px-4
py-3 text-red-700"
            role="alert"
          >
            本システムは開発中です。ご利用いただけません。
          </div>
        </div>
        {children}
      </main>
      <footer>
        <Footer />
```

```
        </footer>
      </>
    );
  };

  export default AuthenticatedLayout;
```

ややこしい部分もあるので、時系列にそって説明します。まずはユーザー情報と各種フラグの初期化です。

```
  // サインイン中のユーザー情報
  const [user, setUser] = useAtom(stateCurrentUser);

  // 読込中フラグ
  const [isLoading, setIsLoading] = useState<boolean>(true);

  // 要ログインフラグ
  const [loginRequired, setLoginRequired] = useState<boolean>(false);
```

ユーザー情報の状態管理はJotaiを用いています。Jotaiについては次の節で説明します。useStateで読込中フラグと要ログインフラグも定義しています。初期状態で読込中フラグはtrue、要ログインフラグはfalseになっていることを覚えておいてください。

最初に表示されるのは、ローディング表示のSpinnerです。Webでよく見かける、輪っかがぐるぐる回るあれです。

```
  // ローディング表示
  if (isLoading) {
    return (
      <main>
```

```
        <Spinner />
      </main>
    );
  }
```

Spinnerの実装は、Zennの作者であるcatnoseさんの記事[6]を参考にしています。
次は、サインイン済みのチェックをしている箇所です。

```
  // サインイン済みかどうかチェックする
  useEffect(() => {
    // awaitを扱うため、いったん非同期関数を作ってから呼び出している
    const checkSignIn = async () => {
      try {
        // サインイン済みのユーザー情報を取得する
        // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
        const currentUser: CognitoUser = await Auth.currentAuthenticatedUser();
        // ユーザー情報をJotaiで管理（これをトリガーにもうひとつのEffect Hookが動く）
        setUser(currentUser);
      } catch (e) {
        // 認証に失敗したらサインアウトしてからログイン画面に遷移させる
        await Auth.signOut();
        setLoginRequired(true);
      }
    };

    // Promiseを無視して呼び出すことを明示するためvoidを付けている
    void checkSignIn();
  }, [setUser]);
```

currentAuthenticatedUserは、サインイン済みのユーザー情報を取得するメソッドです。サインインしている場合はそのまま次のsetUserが実行されて、ユーザー情報が更新されます。

サインインしていない場合はエラーが発生してcatch節に入り、サインアウトと要ログインフラグの更新が行われます。ここでわざわざサインアウトしているのは、無効にしたユーザーでサインインした場合もcurrentAuthenticatedUserが失敗するためです。その場合、サインアウトしないと無限ループすることがあるため、ここで明示的にサインアウトさせています。

サインイン済みのユーザー情報の取得に成功した場合も、失敗して要ログインフラグが立った場合も、次の処理が実行されます。

```
useEffect(() => {
  if (user || loginRequired) setIsLoading(false);
}, [user, loginRequired]);
```

この処理で読込中フラグがオフになるので、ローディング表示から抜けて次の処理に移ります。

```
if (loginRequired) {
  return <Navigate to="/login" replace />;
}
```

この時点で要ログインフラグが立っていたら、ログイン画面へ遷移します。要ログインフラグは初期状態でfalseなので、ユーザー情報の取得に成功していたらこの分岐には入らず、本来の表示処理が実行されます。

それでは実際に試してみましょう。サインアウトした状態でhttp://localhost:3000/mypageにアクセスしてください。自動的にhttp://localhost:3000/loginに遷移してログイン画面が表示されるはずです。

5.2.5　Jotaiで状態管理しよう

ユーザー情報は画面遷移のたびにAuthenticatedLayout.tsxで毎回取得します。しかし、ほかのコンポーネントでもユーザー情報を使いたい場合があります。ユーザー情報画面[7]を見てくださ

い（図5.21）。

図5.21: ユーザー情報画面

メールアドレスの表示部分は、現時点では次のように固定値となっています。

```
<h1 className="mb-8 text-4xl font-bold">ユーザー情報</h1>
<p className="mb-4">ご利用中のプランは FREE プランです。</p>
<p className="mb-4">メールアドレスは sample@example.com です。</p>
```

この箇所をサインイン中のユーザー情報を用いて、動的に表示してみましょう。コンポーネント間でデータを受け渡す方法はいくつもありますが、本書では状態管理ライブラリのJotaiを用います。最近はMeta社謹製の状態管理ライブラリであるRecoilが人気ですが、JotaiはRecoilに似ている上に、Recoilよりもシンプルで軽く使いやすいという特長があります。本書のWebアプリケーションに必要な機能はどちらのライブラリでも提供されているので、軽くて使いやすいJotaiを選びました。

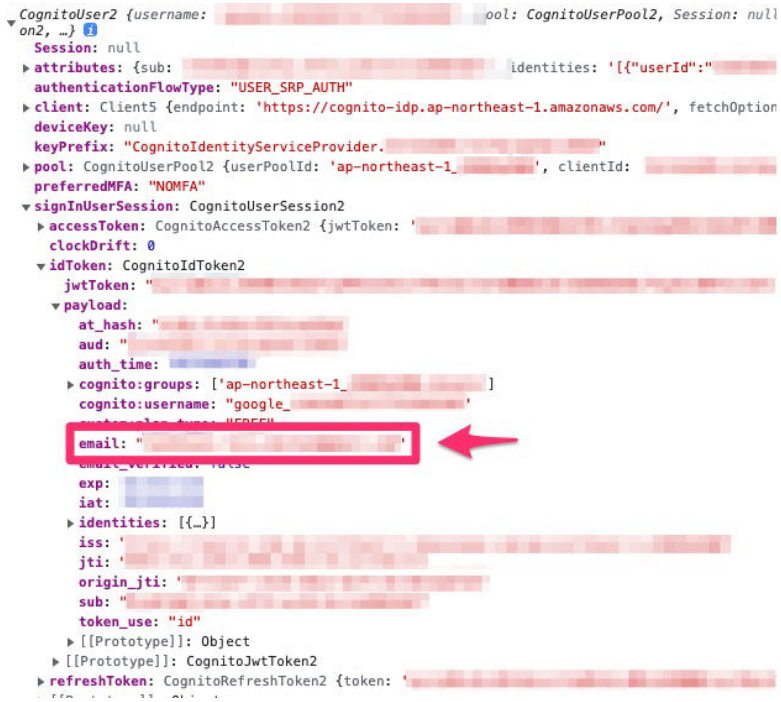
JotaiやRecoilでは、状態を管理するためにAtomと呼ばれるものを用意する必要があります。本書のWebアプリケーションにおいて、ユーザー情報を格納するAtomはsrc/atom/User.tsです。内容は次のとおりです。

```
import { atom } from 'jotai';

type Payload = { email: string };
type IdToken = {
  jwtToken: string;
  payload: Payload;
};
type SignInUserSession = { idToken: IdToken };
export type CognitoUser = {
  signInUserSession: SignInUserSession;
  username: string;
  userDataKey: string;
};

const stateCurrentUser = atom<CognitoUser | null>(null);

export default stateCurrentUser;
```

CognitoUserの構造はいったんコンソールに出力して調べ（図5.22）、必要なものだけ定義しました。今回使いたいメールアドレスはCognitoUser > SignInUserSession > IdToken >

Payload > Payload > emailと、結構深いところにいます。ちなみに、このAtomは1行目を書き換えるだけで、Recoilでもそのまま使えます。

図5.22: CognitoUserの構造



それでは、Jotaiを使ってユーザー情報の管理をしてみましょう。ユーザー情報を格納する処理は前節ですでに登場しました。Jotaiに関連するところだけ抜き出すと、次のとおりです。

```
import { Auth } from 'aws-amplify';
import { useAtom } from 'jotai';
import stateCurrentUser from '../atom/User';
import type { CognitoUser } from '../atom/User';

const [user, setUser] = useAtom(stateCurrentUser);

const currentUser: CognitoUser = await Auth.currentAuthenticatedUser();
setUser(currentUser);
```

useStateに似ているので、Reactを使ったことのある人なら直感的に理解できますね。次はユーザー情報画面を構成しているsrc/components/UserInfoContent.tsxを書き換えます。

```
import { FC, useState } from 'react';
+import { useAtom } from 'jotai';
import Modal from 'react-modal';
import Image from '../svg/undraw_browsing_re_eycn.svg';
import Spacer from './Spacer';
+import stateCurrentUser from '../atom/User';

const UserInfoContent: FC = () => {
+ const [user] = useAtom(stateCurrentUser);
+ const email = user?.signInUserSession.idToken.payload.email;

  // モーダルの表示制御用
  const [isOpen, setIsOpen] = useState<boolean>(false);

  // モーダルの表示スタイル（画面中央に表示）
  const customStyle = {
    content: {
      top: '50%',
      left: '50%',
      right: 'auto',
      bottom: 'auto',
      marginRight: '-50%',
      transform: 'translate(-50%, -50%)',
    },
  };

  return (
```

```
    <section className="bg-white py-6 sm:py-8 lg:py-12">
      <div className="mx-auto max-w-screen-md px-4 md:px-8">
        <h1 className="mb-8 text-4xl font-bold">ユーザー情報</h1>
        <p className="mb-4">ご利用中のプランは FREE プランです。</p>
+       <p className="mb-4">メールアドレスは {email} です。</p>
        <div className="w-5/6 md:w-1/2 lg:w-full lg:max-w-lg">
          <img
            className="rounded object-cover object-center"
            src={Image}
            alt="ユーザー情報"
          />
        </div>
```

値を更新する必要がない場合は、このように記述します。

```
const [user] = useAtom(stateCurrentUser);
```

表示を確認してみましょう。

図5.23: 修正後のユーザー情報画面

サインイン中のユーザー情報が表示されました。Jotaiを使えばこのように、簡単な手順で状態を共有できます。[8]

1. パンくずリストの表示では「Amazon Cognito > ユーザープール > ユーザープールを作成」になっています。この順番でもアクセス可能です。
2. 通常のIDとパスワードで認証する場合は多要素認証を有効にしましょう。
3. Amplifyをフルに使えば、バックエンドやデプロイ先まで自動で用意してくれます。しかし、余計なものまで大量に作られるので、筆者の好みではありません。EasyではあってもSimpleではないツールという印象です。
4. https://github.com/sikkimtemi/How_to_create_API_sales_service/tree/main/5-Authentication/fm_mail_frontend
5. https://ui.docs.amplify.aws/react/getting-started/troubleshooting
6. https://zenn.dev/catnose99/articles/19a05103ab9ec7
7. http://localhost:3000/userinfo
8. Recoilも簡単ですが、ルートコンポーネントをタグで囲む必要があるので、Jotaiよりひと手間かかります。

第6章　APIキーを自動で発行しよう

本章ではユーザー登録と連動して自動でAPIキーを発行する仕組みを作り、フロントエンド側で表示します。4章で作成したメール受信APIと使用量プランを使います。

6.1　使用量プランの確認

　AWSのAPI Gatewayにアクセスして、4章で作成したメール受信APIをクリックし、使用量プラン（図6.1）を確認してください。

図6.1: 使用量プランの確認

　のちほどこのIDを利用するので、控えておいてください。

6.2　DynamoDBテーブルの作成

　フロントエンド側で表示するAPIキーはDynamoDBに格納します。AWSで「DynamoDB」→「テーブル」→「テーブルの作成」からテーブルを作成してください。

図6.2: DynamoDBテーブルの作成

　テーブル名は適当でかまいません。筆者はFM_Mail_API_Keyにしました。パーティションキーはUserIdにしてください。ここにはCognitoのユーザーID（ユーザー名）が入ります。ソートキーはTypeにしてください。ここには有料版と無料版を識別する「FREE」もしくは「PRO」という文字列が入ります。あとはデフォルト設定で作成してください。

6.3　バックエンドの実装

それでは、バックエンドから実装していきましょう。バックエンドではCognitoのユーザー登録をトリガーに次の処理を行います。

・APIキーの発行

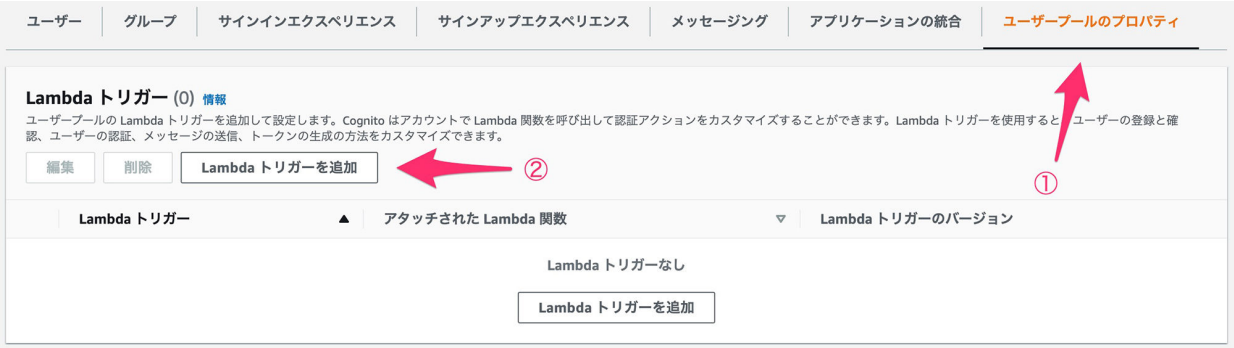
・使用量プランの適用

・DynamoDBにAPIキーを登録

6.3.1　Lambdaトリガーの登録

「Amazon Cognito」→「ユーザープール」から5章で作成したユーザープールを開いてください。

図6.3: Lambdaトリガーを追加



次に「ユーザープールのプロパティ」タブを開き、「Lambdaトリガーを追加」をクリックしてください。

図6.4: Lambdaトリガーの設定

「Lambdaトリガーを追加」画面が表示されたら、「サインアップ」と「確認後トリガー」をそれぞれ選択して「Lambda関数の作成」をクリックしてください。

6.3.2　Lambda関数の作成

Lambdaの画面が表示されたら「関数の作成」をクリックしてください。

図6.5: 関数の作成

関数名は適当に入力してください。ランタイムは「Python 3.9」を選択してください。アーキテクチャは「x86_64」を選択してください。

関数の内容は次のとおりです。

```
import boto3

import os


# 環境変数

REGION_NAME = os.environ["REGION_NAME"]

DYNAMODB_TABLE = os.environ["DYNAMODB_TABLE"]

REST_API_ID = os.environ["REST_API_ID"]

USAGE_PLAN_ID = os.environ["USAGE_PLAN_ID"]


dynamodb = boto3.resource("dynamodb", region_name=REGION_NAME)

table = dynamodb.Table(DYNAMODB_TABLE)
```

```
apigateway_cli = boto3.client("apigateway")



def lambda_handler(event, context):
    # ユーザー名を取得
    user_name = event["userName"]

    # APIキーを発行
    result = apigateway_cli.create_api_key(
        name="fm_mail_free_" + user_name,
        enabled=True,
        stageKeys=[{"restApiId": REST_API_ID, "stageName": "api"}],
    )

    # 発行したAPIキーの値とIDを取得
    api_key = result["value"]
    api_key_id = result["id"]

    # APIキーに使用量プランを適用
    apigateway_cli.create_usage_plan_key(
        usagePlanId=USAGE_PLAN_ID, keyId=api_key_id, keyType="API_KEY"
    )

    # DynamoDBにAPIキーを登録
    with table.batch_writer() as batch:
        batch.put_item(
            Item={
                "UserId": user_name,
                "Type": "FREE",
                "ApiKey": api_key,
                "ApiKeyId": api_key_id,
```

```
            }
        )

    # eventを返さないとCognito側で「Unrecognizable lambda output」というエラーになる
    return event
```

環境変数を用いるので、「設定」→「環境変数」から定義してください。

図6.6: 環境変数

環境変数の設定値は次のとおりです。

環境変数	設定値
DYNAMODB_TABLE	6.2で作成したDynamoDBのテーブル名。
REGION_NAME	AWSのリージョン名。東京リージョンならap-northeast-1を設定する。
REST_API_ID	メール受信APIのID。API GatewayのAPI一覧で確認可能。
USAGE_PLAN_ID	使用量プランのID。API Gatewayの使用量プランで確認可能。

次は、このLamda関数に必要な権限をアタッチしましょう。関数に対応するIAMが自動生成されているはずなので、探してください。

図6.7: IAMの許可ポリシー

次のポリシーをアタッチしてください。[1]

・AmazonDynamoDBFullAccess

・AmazonAPIGatewayAdministrator

・AmazonCognitoPowerUser

それでは、関数の内容を見ていきましょう。最初はboto3の初期処理です。

```
dynamodb = boto3.resource("dynamodb", region_name=REGION_NAME)

table = dynamodb.Table(DYNAMODB_TABLE)

apigateway_cli = boto3.client("apigateway")
```

DynamoDBとAPI Gatewayを操作するクライアントを用意しています。次はAPIキーの発行です。

```
    # ユーザー名を取得
    user_name = event["userName"]


    # APIキーを発行
    result = apigateway_cli.create_api_key(
        name="fm_mail_free_" + user_name,
        enabled=True,
        stageKeys=[{"restApiId": REST_API_ID, "stageName": "api"}],
    )
```

Cognitoの新規ユーザー登録をトリガーとするLambda関数では、このようにイベントからユーザー名を取得できます。Cognitoのユーザー名は名称こそuserNameですが、実際の値はgoogle_123456789012345678901234567890といった形式の一意の値で、変更もできません。どちらかというとユーザーIDに近い存在なので、本書ではユーザーID代わりに用います。[2]

APIキーの発行はcreate_api_keyメソッドで行います。APIキーの名称にユーザー名を含めることで、ユーザーとAPIキーを紐付けています。

次は使用量プランの適用です。

```
    # 発行したAPIキーの値とIDを取得
    api_key = result["value"]
    api_key_id = result["id"]

    # APIキーに使用量プランを適用
    apigateway_cli.create_usage_plan_key(
        usagePlanId=USAGE_PLAN_ID, keyId=api_key_id, keyType="API_KEY"
    )
```

APIキーの値とIDはcreate_api_keyメソッドの戻り値から取得できます。取得したAPIキーのIDと環境変数に設定した使用量プランのIDを用いて、create_usage_plan_keyメソッドで両者を紐付けます。

次はDynamoDBへの登録です。

```
    # DynamoDBにAPIキーを登録
    with table.batch_writer() as batch:
        batch.put_item(
            Item={
                "UserId": user_name,
                "Type": "FREE",
```

```
                "ApiKey": api_key,
                "ApiKeyId": api_key_id,
            }
        )
```

batch_writerメソッドとput_itemメソッドを組み合わせて、レコードを作成しています。
主な処理は以上ですが、最後に重要な行が残っています。

```
    return event
```

引数で受け取ったイベントを戻り値として返しています。これを忘れると、フロントエンドで実行する初回のAuth.federatedSignInメソッドが失敗します。

6.3.3　Cognitoのユーザー属性を使ってみよう

Cognitoユーザープールには、ユーザー属性と呼ばれるリソースが存在します。メールアドレスや姓名などのあらかじめ用意された項目のほかに、カスタム属性を追加して自由に使うこともできます。それではユーザーが無料版と有料版のどちらを利用しているか識別するplan_typeというカスタム属性を追加し、ユーザー登録時にFREEという値が自動で格納されるようにしてみましょう。

カスタム属性を追加するには「Amazon Cognito」→「ユーザープール」で対象のユーザープールを開き、「サインアップエクスペリエンス」タブを開きます。

図6.8: サインアップエクスペリエンスタブ

下の方にスクロールするとカスタム属性が出てくるので、「カスタム属性を追加」をクリックしてください。

図6.9: カスタム属性を追加

plan_typeと入力して変更を保存してください。注意文言にあるとおり、一度作成したカスタム属性は変更や削除できません。慎重に行動しましょう。[3]

それでは、先ほど作成したLamda関数を次のように書き換えてください。

```
import boto3

import os


# 環境変数

REGION_NAME = os.environ["REGION_NAME"]

DYNAMODB_TABLE = os.environ["DYNAMODB_TABLE"]

REST_API_ID = os.environ["REST_API_ID"]
```

```
USAGE_PLAN_ID = os.environ["USAGE_PLAN_ID"]


dynamodb = boto3.resource("dynamodb", region_name=REGION_NAME)
table = dynamodb.Table(DYNAMODB_TABLE)
apigateway_cli = boto3.client("apigateway")
+ cognito_cli = boto3.client("cognito-idp")



def lambda_handler(event, context):
    # ユーザー名を取得
    user_name = event["userName"]

+   # ユーザープールIDを取得
+   user_pool_id = event["userPoolId"]
+
+   # ユーザープールのカスタム属性を更新
+   cognito_cli.admin_update_user_attributes(
+       UserPoolId=user_pool_id,
+       Username=user_name,
+       UserAttributes=[
+           {"Name": "custom:plan_type", "Value": "FREE"},
+       ],
+   )

    # APIキーを発行
    result = apigateway_cli.create_api_key(
        name="fm_mail_free_" + user_name,
        enabled=True,
        stageKeys=[{"restApiId": REST_API_ID, "stageName": "api"}],
    )
```

```
    # 発行したAPIキーの値とIDを取得
    api_key = result["value"]
    api_key_id = result["id"]

    # APIキーに使用量プランを適用
    apigateway_cli.create_usage_plan_key(
        usagePlanId=USAGE_PLAN_ID, keyId=api_key_id, keyType="API_KEY"
    )

    # DynamoDBにAPIキーを登録
    with table.batch_writer() as batch:
        batch.put_item(
            Item={
                "UserId": user_name,
                "Type": "FREE",
                "ApiKey": api_key,
                "ApiKeyId": api_key_id,
            }
        )

    # eventを返さないとCognito側で「Unrecognizable lambda output」というエラーになる
    return event
```

Cognitoをboto3経由で操作するクライアントは次のように準備します。

```
cognito_cli = boto3.client("cognito-idp")
```

ユーザー属性の更新は次のように行います。

```
    # ユーザープールIDを取得
    user_pool_id = event["userPoolId"]

    # ユーザープールのカスタム属性を更新
    cognito_cli.admin_update_user_attributes(
        UserPoolId=user_pool_id,
        Username=user_name,
        UserAttributes=[
            {"Name": "custom:plan_type", "Value": "FREE"},
        ],
    )
```

属性の名称がcustom:plan_typeになっていることに注目してください。カスタム属性には自動的にcustom:という接頭語が追加されます。

以上で、Lamdaトリガーの実装は完了です。Cognitoユーザープールでユーザーを削除し、新規登録してみましょう。APIキーが自動で発行され、DynamoDBに登録されていれば成功です。

6.3.4 DynamoDBから安全に情報を取り出す

DynamoDBにAPIキーの値を登録できるようになったので、今度は取り出せるようにしましょう。ただし、APIキーは秘密にすべき値です。他人のAPIキーを取得できない仕組みを構築する必要があります。筆者がかなり頭を悩ませて、「これなら安全かな」と思えるようにしたAPIがこちらです。

```
import boto3
from boto3.dynamodb.conditions import Key
import json
import os
from chalice import Chalice, CognitoUserPoolAuthorizer
```

```
app = Chalice(app_name="dynamodb_api")


# 環境変数
USER_POOL_ARN = os.environ.get("USER_POOL_ARN")
USER_POOL_NAME = os.environ.get("USER_POOL_NAME")
DYNAMODB_TABLE = os.environ.get("DYNAMODB_TABLE")
REGION_NAME = os.environ.get("REGION_NAME")


# Cognitoで認証する
authorizer = CognitoUserPoolAuthorizer(USER_POOL_NAME, provider_arns=[USER_POOL_
ARN])


# DynamoDBに接続
dynamodb = boto3.resource("dynamodb", region_name=REGION_NAME)
table = dynamodb.Table(DYNAMODB_TABLE)



@app.route("/apikey", authorizer=authorizer, cors=True)
def get_my_api_key():
    # 認証情報からUserNameを取り出す
    context = app.current_request.context
    user_name = context["authorizer"]["claims"]["cognito:username"]

    # DynamoDBからユーザーに紐づくAPIキーの情報を取り出す
    result = table.query(KeyConditionExpression=Key("UserId").eq(user_name))

    resp = {"status": "OK", "result": result, "UserName": user_name}

    # 結果を返す
    return json.dumps(resp, ensure_ascii=False)
```

このAPIは4章のメール受信APIと同じく、Chaliceで実装しています。今回は環境変数を用います。Chaliceの環境変数は.chalice/config.jsonに記載します。サンプルコードにconfig.json.templateを用意したので、コピーして利用してください。

```
{
  "version": "2.0",
  "app_name": "dynamodb_api",
  "stages": {
    "dev": {
      "api_gateway_stage": "api",
      "environment_variables": {
        "USER_POOL_ARN": "arn:aws:cognito-idp:ap-xxxxxxxxxxx:xxxxxxxxx:userpool/
ap-xxxxxxx-1_xxxxxxxxxx",
        "USER_POOL_NAME": "xxxxxxxxxxxxxxx",
        "DYNAMODB_TABLE": "xxxxxxxxxxxxxxx",
        "REGION_NAME": "ap-xxxxxxxxxx-1"
      }
    }
  }
}
```

環境変数の設定値は次のとおりです。

環境変数	設定値
USER_POOL_ARN	ユーザープールのARN。「ユーザープールの概要」で確認可能。
USER_POOL_NAME	ユーザープール名。「ユーザープールの概要」で確認可能。
DYNAMODB_TABLE	DynamoDBのテーブル名。
REGION_NAME	AWSのリージョン名。

このAPIはCognitoにサインインしたフロントエンドから接続するので、認証・認可にはCognitoを用います。ChaliceでCognitoを利用するには、次のようにします。

```
authorizer = CognitoUserPoolAuthorizer(USER_POOL_NAME, provider_arns=[USER_POOL_
ARN])


...


@app.route("/apikey", authorizer=authorizer, cors=True)
def get_my_api_key():
```

これで、Cognitoにサインインしていないユーザーからのアクセスは無効になります。そして次のようにすると、Cognitoの認証情報からユーザー名を取り出せます。

```
    context = app.current_request.context
    user_name = context["authorizer"]["claims"]["cognito:username"]
```

最後にユーザー名を用いて、DynamoDBからデータを取り出します。

```
    result = table.query(KeyConditionExpression=Key("UserId").eq(user_name))
```

このAPIのポイントは、リクエストパラメータが存在しないことです。もしユーザー名をパラメータとして渡すことができたら、どうなるか想像してみてください。Cognitoのユーザー名は数字で構成されているため、他人のユーザー名を類推しやすい特徴があります。ブルートフォースアタックをかけられたら簡単にAPIキーが流出してしまいますね。このAPIでは認証情報からユーザー名を取り出すことで、自分自身のAPIキーしか取り出せないようにしているわけです。

それでは、ローカルで動作確認してみましょう。次のコマンドでローカルサーバーを起動してください。

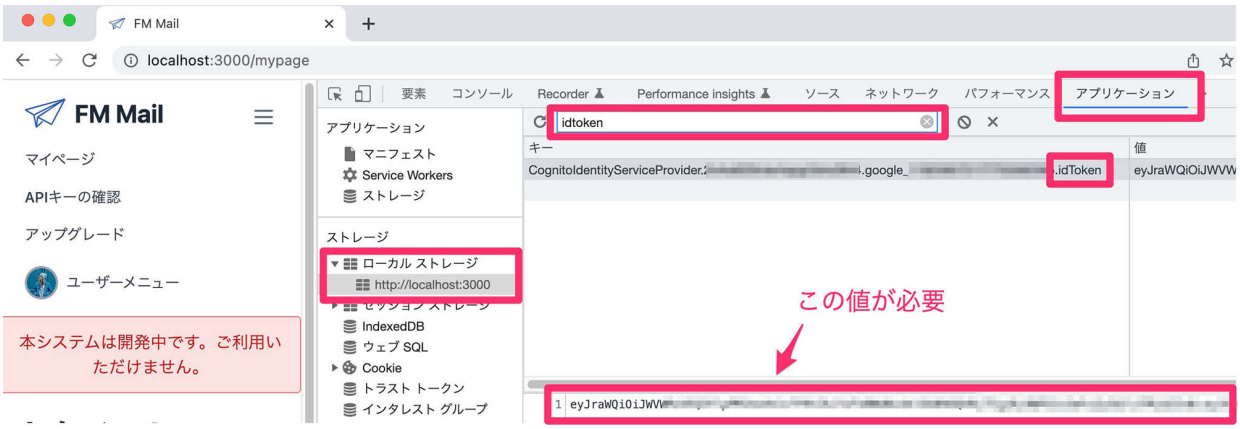
```
chalice local
```

curlコマンドでアクセスしてみましょう。

```
curl -X GET \
-H 'Content-Type:application/json' \
-H 'Authorization:Bearer {CognitoのidToken}' \
http://localhost:8000/apikey
```

CognitoのidTokenはブラウザのDevToolsで確認できます。

図6.10: idTokenの確認



ちなみにidTokenはとても長いので、実際のcurlコマンドは図6.11のようにターミナルを目いっぱい使います。

図6.11: 実際のcurlコマンド

APIキーを含むレスポンスが返ってきたら成功です。デプロイしましょう。

```
chalice deploy
```

デプロイしたら権限を追加しましょう。IAMが自動生成されているので、探してください。

図6.12: IAMの許可ポリシー

図のようにAmazonDynamoDBFullAccessポリシーをアタッチしてください。

それでは、デプロイ先のURLでも動作確認しましょう。

```
curl -X GET \
-H 'Content-Type:application/json' \
-H 'Authorization:Bearer {CognitoのidToken}' \
{デプロイ先のURL}/apikey
```

ローカルと同じレスポンスが返れば成功です。以上で、バックエンド側の準備は完了です。

6.4　フロントエンドの実装

フロントエンド側では、次の処理を行います。

・Cognitoの属性を取得して表示する

・APIを呼び出して結果を表示する

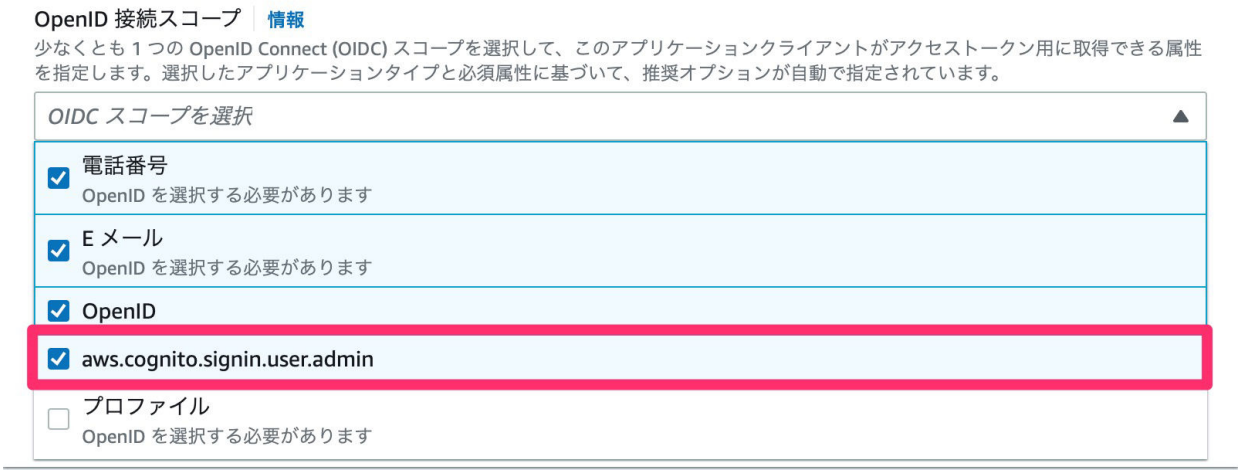
6.4.1　Cognitoのユーザー属性を取得する準備

フロントエンドからCognitoのユーザー属性にアクセスするには、AWS側でいくつか準備が必要です。「Amazon Cognito」→「ユーザープール」で対象のユーザープールを開き、「アプリケーションの統合」タブをクリックしてください。下までスクロールすると「アプリケーションクライアントのリスト」があるので、対象のアプリケーションクライアント名をクリックしてください。「属性の読み取りおよび書き込み許可」の「編集」をクリックしてください。

図6.13: 属性の読み取りおよび書き込み許可

先ほど追加したcustom:plan_typeの読み取り許可をチェックして、保存してください。次は「ホストされたUI」の「編集」をクリックしてください。

図6.14: OpenID接続スコープ



OpenID接続スコープにaws.cognito.signin.user.adminを追加し、変更を保存してください。AWS側の準備作業は以上です。

6.4.2 フロントエンドでユーザー属性を取得する

次はフロントエンド側の修正です。まずsrc/awsExports.tsのscopeを次のように修正してください。

```
const awsExports = {
  Auth: {
    region: import.meta.env.VITE_REGION,
    userPoolId: import.meta.env.VITE_USER_POOL_ID,
    userPoolWebClientId: import.meta.env.VITE_USER_POOL_WEB_CLIENT_ID,
    oauth: {
      domain: import.meta.env.VITE_OAUTH_DOMAIN,
-     scope: ['openid'],
+     scope: ['openid', 'aws.cognito.signin.user.admin'],
      redirectSignIn: import.meta.env.VITE_OAUTH_REDIRECT_SIGN_IN,
      redirectSignOut: import.meta.env.VITE_OAUTH_REDIRECT_SIGN_OUT,
      responseType: 'code',
```

```
    },
  },
};

export default awsExports;
```

先ほどAWS側で追加したOpenID接続スコープに対応しています。次はJotai用のAtomを作ります。本書では、ユーザー属性用のAtomはsrc/Atom/UserAttribute.tsとします。

```
import { atom } from 'jotai';

export type CognitoUserAttribute = {
  email: string | undefined;
  planType: string | undefined;
};

const stateUserAttribute = atom<CognitoUserAttribute | null>(null);

export default stateUserAttribute;
```

planTypeのほかにemailが定義されています。5章ではCognitoのユーザー情報からメールアドレスを取得しましたが、メールアドレスはユーザー属性からでも取得可能です。ユーザー属性のメールアドレスはユーザーが変更可能なので、もしユーザー情報編集機能を実装するならこちらを利用しましょう。本書ではユーザー情報編集機能は実装しません。

Atomを用意したので、サインイン時にユーザー属性を格納しましょう。src/layouts/AuthenticatedLayout.tsxを次のように修正します。

```
import { FC, ReactNode, useEffect, useState } from 'react';
import { Auth } from 'aws-amplify';
```

```
import { useAtom } from 'jotai';
import { Navigate } from 'react-router-dom';
import Header2 from '../components/Header2';
import Footer from '../components/Footer';
import Spinner from '../components/Spinner';
import stateCurrentUser from '../atom/User';
+import stateUserAttribute from '../atom/UserAttribute';
import type { CognitoUser } from '../atom/User';
+import type { CognitoUserAttribute } from '../atom/UserAttribute';

type Props = { children: ReactNode };

const AuthenticatedLayout: FC<Props> = ({ children }) => {
  // サインイン中のユーザー情報
  const [user, setUser] = useAtom(stateCurrentUser);

+ // ユーザー属性
+ const [userAttribute, setUserAttribute] = useAtom(stateUserAttribute);

  // 読込中フラグ
  const [isLoading, setIsLoading] = useState<boolean>(true);

  // 要ログインフラグ
  const [loginRequired, setLoginRequired] = useState<boolean>(false);

  // サインイン済みかどうかチェックする
  useEffect(() => {
    // awaitを扱うため、いったん非同期関数を作ってから呼び出している
    const checkSignIn = async () => {
      try {
        // サインイン済みのユーザー情報を取得する
```

```
          // eslint-disable-next-line @typescript-eslint/no-unsafe-assignment
          const currentUser: CognitoUser = await Auth.currentAuthenticatedUser();
+         // Cognitoのユーザー属性を取得する
+         const userAttributes = await Auth.userAttributes(currentUser);
+         // カスタム属性を取り出す
+         const email = userAttributes.find((obj) => obj.Name === 'email')?.Value;
+         const planType = userAttributes.find(
+           (obj) => obj.Name === 'custom:plan_type',
+         )?.Value;
+         const myAttribute: CognitoUserAttribute = {
+           email,
+           planType,
+         };
          // ユーザー情報をJotaiで管理（これをトリガーにもうひとつのEffect Hookが動く）
          setUser(currentUser);
+         // ユーザー属性をJotaiで管理
+         setUserAttribute(myAttribute);
        } catch (e) {
          // 認証に失敗したらサインアウトしてからログイン画面に遷移させる
          await Auth.signOut();
          setLoginRequired(true);
        }
      };

...
```

ユーザー属性はuserAttributesメソッドで取得できます。個々の属性値は取得結果のオブジェクトからcustom:plan_typeのような属性名で検索して取得しています。最後は表示箇所の修正です。src/components/UserInfoContent.tsxを次のように修正します。

```
import { FC, useState } from 'react';
import { useAtom } from 'jotai';
import Modal from 'react-modal';
import Image from '../svg/undraw_browsing_re_eycn.svg';
import Spacer from './Spacer';
import stateUserAttribute from '../atom/UserAttribute';

const UserInfoContent: FC = () => {
  // ユーザー属性からemailと現在のプランを取り出す
  const [userAttribute] = useAtom(stateUserAttribute);
  const email = userAttribute?.email;
  const planType = userAttribute?.planType;

...

        <p className="mb-4">ご利用中のプランは{planType}プランです。</p>
        <p className="mb-4">メールアドレスは {email} です。</p>

...
```

これで、ユーザー属性の取得と表示ができるようになりました。

6.4.3 フロントエンドでAPIキーを表示する

次はDynamoDBからAPIキーを取得して表示しましょう。本書ではHTTPクライアントライブラリとしてkyを用います。まずはインストールしましょう。

```
npm i ky
```

次にsrc/components/ApiKeyTable.tsxを書き換えます。

```
import { useEffect, useState, FC } from 'react';
import { useAtom } from 'jotai';
import ky from 'ky';
import stateCurrentUser from '../atom/User';
import Spinner from './Spinner';
import ApiKeyInfo from './ApiKeyInfo';
import type { ApiKeyItem } from './ApiKeyInfo';

type Result = { Items: ApiKeyItem[] };
type Resp = { result: Result };

const ApiKeyTable: FC = () => {
  // サインイン中のユーザー情報
  const [user] = useAtom(stateCurrentUser);

  // 読込中フラグ
  const [isLoading, setIsLoading] = useState<boolean>(true);

  // APIキー
  const [apiKeys, setApiKeys] = useState<ApiKeyItem[] | null>(null);

  // DynamoDBアクセス用URL
  const url = `${import.meta.env.VITE_DYNAMODB_BASE_URL}/apikey`;

  // DynamoDBからAPIキーを取得する
  useEffect(() => {
    // awaitを扱うため、いったん非同期関数を作ってから呼び出している
    const getApiKeys = async () => {
      try {
        if (user) {
```

```
          // Lambda経由でDynamoDBにアクセスする
          const res: Resp = await ky
            .get(url, {
              headers: {
                                                          Authorization:    `Bearer
${user.signInUserSession.idToken.jwtToken}`,
              },
            })
            .json();
          if (res.result.Items) setApiKeys(res.result.Items);
        }
      } catch (e) {
        // APIキー取得に失敗したらnullをセット
        setApiKeys(null);
      }
    };

    // Promiseを無視して呼び出すことを明示するためvoidを付けている
    void getApiKeys();
  }, [url, user]);

  // APIキーを取得できたらローディング表示をやめる
  useEffect(() => {
    if (apiKeys) setIsLoading(false);
  }, [apiKeys]);

  // ローディング表示
  if (isLoading) {
    return <Spinner />;
  }
```

```
  return (
    <table className="mb-8 rounded-lg bg-white p-4 shadow">
      <thead>
        <tr>
             <th  className="dark:border-dark-5 whitespace-nowrap border-b-2 p-4
font-normal text-gray-900">
            No.
          </th>
             <th  className="dark:border-dark-5 whitespace-nowrap border-b-2 p-4
font-normal text-gray-900">
            プラン種別
          </th>
             <th  className="dark:border-dark-5 whitespace-nowrap border-b-2 p-4
font-normal text-gray-900">
            APIキー
          </th>
             <th  className="dark:border-dark-5 whitespace-nowrap border-b-2 p-4
font-normal text-gray-900">
            コピー
          </th>
        </tr>
      </thead>
      <tbody>
        {apiKeys &&
          apiKeys.map((item: ApiKeyItem, index: number) => (
            <ApiKeyInfo item={item} index={index} key={item.ApiKey} />
          ))}
      </tbody>
    </table>
  );
```

```
};
export default ApiKeyTable;
```

構造的には5章の認証処理と似ています。図6.11で見た長いcurlコマンドと同じ処理は、次の箇所で行っています。

```
    const getApiKeys = async () => {
      try {
        if (user) {
          // Lambda経由でDynamoDBにアクセスする
          const res: Resp = await ky
            .get(url, {
              headers: {
                 Authorization: `Bearer ${user.signInUserSession.idToken.jwtToke
n}`,
              },
            })
            .json();
          if (res.result.Items) setApiKeys(res.result.Items);
        }
      } catch (e) {
        // APIキー取得に失敗したらnullをセット
        setApiKeys(null);
      }
    };
```

アクセス先のURLは環境変数に格納しています。次の.env.local.templateを参考に、.env.localを修正してください。

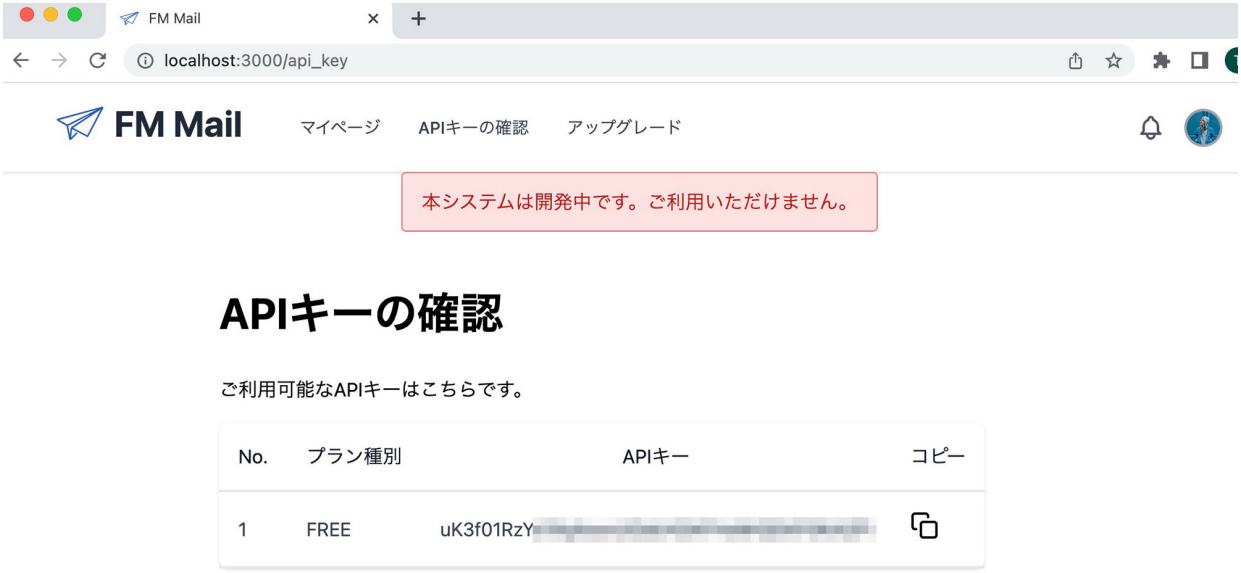
```
VITE_DYNAMODB_BASE_URL=https://xxxxxxxxxxx.execute-api.ap-xxxxxxxxxx-1.amazonaw
s.com/api
VITE_OAUTH_DOMAIN=xxxxxxxxxxx.auth.ap-xxxxxxxxxx-1.amazoncognito.com
VITE_OAUTH_REDIRECT_SIGN_IN=http://localhost:3000/mypage
VITE_OAUTH_REDIRECT_SIGN_OUT=http://localhost:3000/
VITE_REGION=ap-xxxxxxxxxx-1
VITE_USER_POOL_ID=ap-xxxxxxxxxx-1_xxxxxxxxx
VITE_USER_POOL_WEB_CLIENT_ID=xxxxxxxxxxxxxxxx
```

VITE_DYNAMODB_BASE_URLが追加されています。src/vite-env.d.tsも同様に修正が必要です。

```
/// <reference types="vite/client" />

interface ImportMetaEnv {
  readonly VITE_DYNAMODB_BASE_URL: string;
  readonly VITE_OAUTH_DOMAIN: string;
  readonly VITE_OAUTH_REDIRECT_SIGN_IN: string;
  readonly VITE_OAUTH_REDIRECT_SIGN_OUT: string;
  readonly VITE_REGION: string;
  readonly VITE_USER_POOL_ID: string;
  readonly VITE_USER_POOL_WEB_CLIENT_ID: string;
}

interface ImportMeta {
  readonly env: ImportMetaEnv;
}
```

動作確認してみましょう。サインインしてAPIの確認ページ[4]を開いてください。

図6.15: APIキーの確認ページ



スピナーが表示されたあと、APIキーが表示されたら成功です。

1. AmazonCognitoPowerUserはこの時点では不要です。次の節でカスタム属性を更新する処理を追加する際に必要になります。
2. Cognitoユーザープールには「ユーザーID（Sub）」という値もありますが、こちらは使い方がさっぱりわかりません。公式ドキュメントでもユーザー名のことをユーザーIDと表記している箇所があったりして、最初はかなり混乱しました。
3. 慎重に行動しなかった筆者のユーザープールには、使われることのないカスタム属性が今も残っています。
4. http://localhost:3000/api_key

第7章　Netlifyでいったん公開してみよう

　無料版の機能はだいたい実装できたので、いったん公開してみましょう。デプロイ先の候補はいろいろありますが、本書ではNetlifyを利用します。

7.1　ビルドしてみよう

　今まではローカルのNode.jsで動かしてきましたが、Netlifyでホストできるのは静的なファイルのみです。ローカルでも静的なファイルをビルドして、ホストしてみましょう。ビルドは次のように行います。

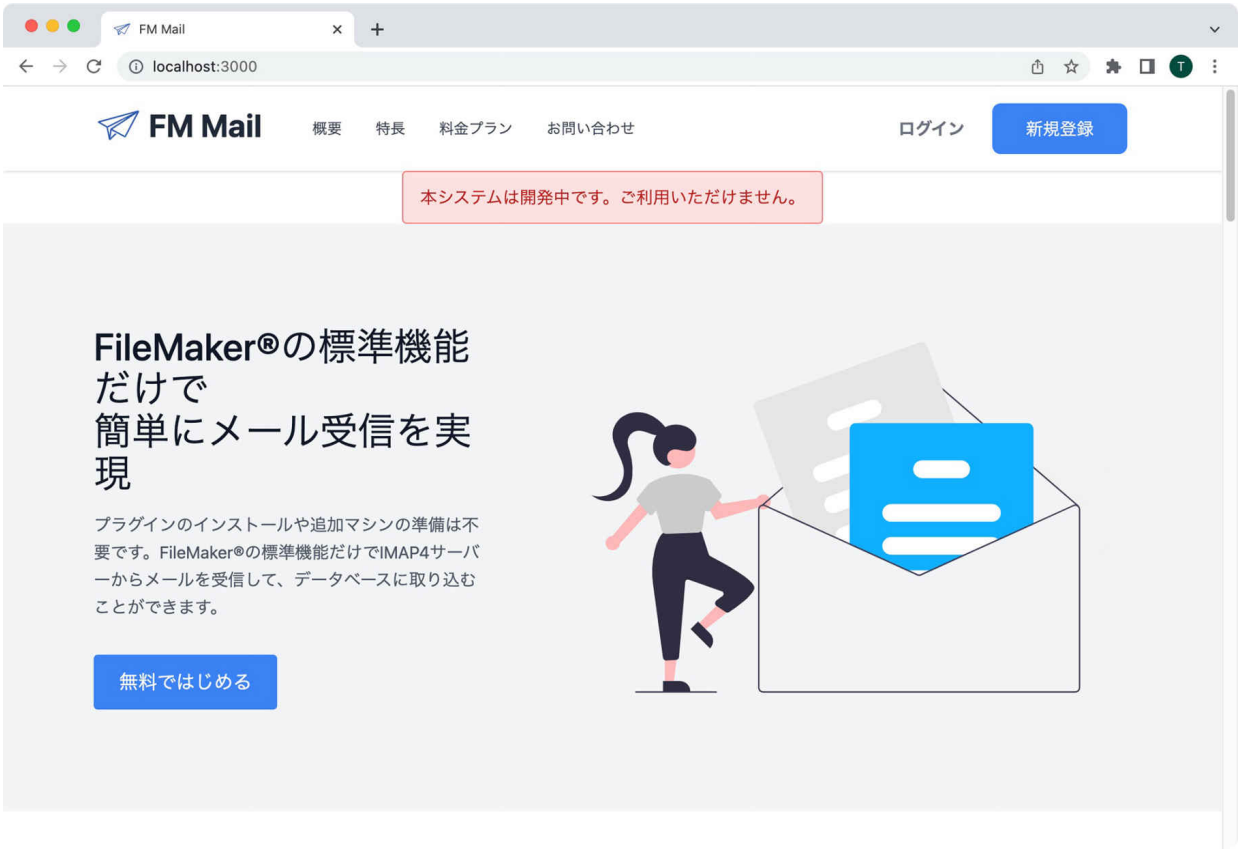
```
npm run build
```

　ビルドに成功すると、distディレクトリにindex.htmlファイルとassetsディレクトリが生成されます。Pythonの簡易HTTPサーバーで動作確認してみましょう。

```
cd dist
python3 -m http.server 3000
```

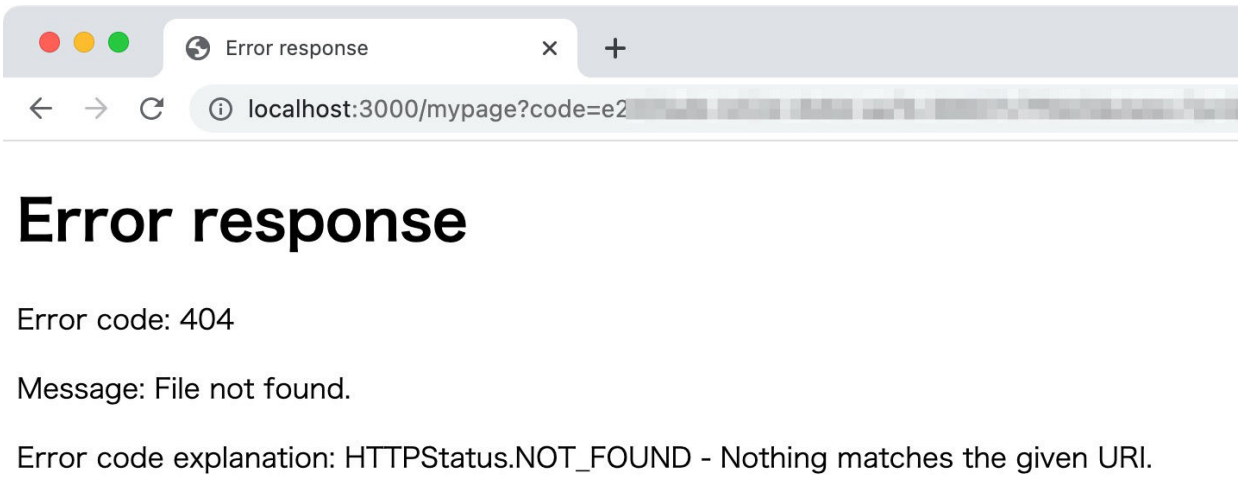
　ブラウザで表示してみると、問題なく動いているように見えます。

図7.1: 簡易HTTPサーバーによる動作確認



ところが、ログインすると404エラーが発生します。

図7.2: 404エラー



簡易HTTPサーバーは現在distディレクトリをホストしています。そして、distディレクトリにはindex.htmlとassetsしか存在しません。ログインするとマイページに遷移するのでhttp://localhost:3000/mypageがリクエストされます。ところが、mypageというファイルやフォルダはdist配下には存在しないので、404エラーになるわけです。この問題を解決するには、ルート以外へのアクセスをルートへリダイレクトします。Pythonでは少し手間がかかるので、本書ではNode.jsのservorを用います。Pythonの簡易HTTPサーバーを停止してから次のコマンドを実行してください。

```
npx servor . index.html 3000
```

今度はサインインしても、404エラーにはならなかったはずです。APIキーが取得できることも確認しておきましょう。

7.2　Netlifyにデプロイしよう

ビルドしたファイルがローカルで動くことが確認できたので、次はNetlifyにデプロイします。

7.2.1　Netlifyのリダイレクト設定

Netlifyでdistディレクトリのファイル群をホストすると、ローカルで試した簡易HTTPサーバーと同じ問題が生じます。そこで、Netlify用のリダイレクト設定を追加しましょう。プロジェクトルートにnetlify.tomlを配置してください。

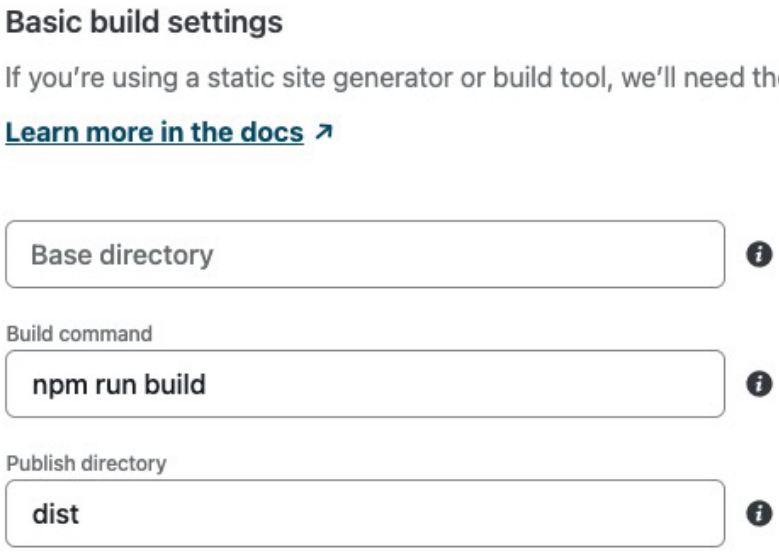
```
[[redirects]]
  from = "/*"
  to = "/index.html"
  status = 200
```

これで、全てのアクセスがルートのindex.htmlにリダイレクトされます。

7.2.2　Gitリポジトリと連携

それでは、Netlifyへデプロイしてみましょう。Netlifyにサインインしたら「Add new site」→「Import an existing project」でGitリポジトリと連携してください。NetlifyはGitリポジトリを変更すると、自動でビルドしてくれます。ビルド設定は次のようにしましょう。

図7.3: Basic build settings



ビルドコマンドはnpm run build、公開ディレクトリはdistです。サンプルコードのようにモノレポ構成になっている場合は、ベースディレクトリも設定する必要があります。設定が終わったらデプロイしましょう。しばらくするとビルドが終わり、サイトが公開されます。必要に応じて「Domains」設定でサイト名を変更しておきましょう。

7.2.3　AWS側の設定追加

現時点では、Cognitoにサインインできるのはローカル環境のみです。Netlifyでもサインイン可能にしましょう。Cognitoのユーザープールを開き、「ホストされたサインアップページとサインインページ」でNetlifyのURLを追加してください。

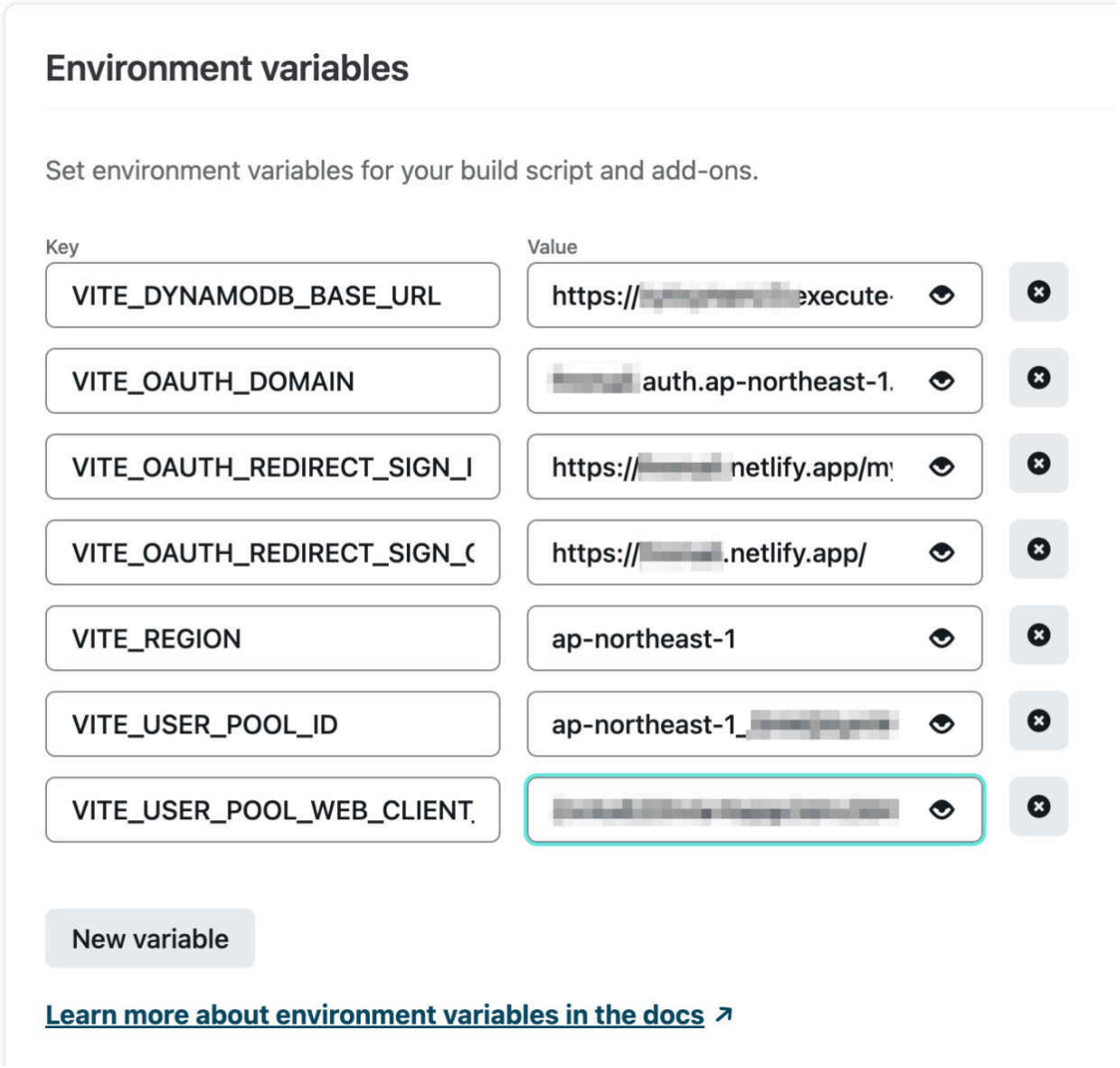
図7.4: ホストされたサインアップページとサインインページ

「許可されたコールバックURL」は、後ろに/mypageを付けるのを忘れないでください。

7.2.4　Netlifyで環境変数を設定

今度はNetlifyに戻り、環境変数を設定しましょう。Netlifyの環境変数設定は「Build & deploy」設定の中ほどにあります。

図7.5: Netlifyの環境変数



基本的には.env.localと同じ内容を設定します。ただし、VITE_OAUTH_REDIRECT_SIGN_INとVITE_OAUTH_REDIRECT_SIGN_OUTはNetlifyのURLに変更してください。

環境変数を設定したら、必要に応じて再度デプロイして動作確認してみましょう。サインインとAPIキーの表示ができれば成功です。

7.3　問い合わせフォームを使おう

Netlifyは、簡単な記述で問い合わせフォームを設置できるのが魅力のひとつです。ただし、ReactなどのSPAでは少し手間がかかります。

まず、src/components/Contact.tsxの<form>タグを次のように書き換えます。

```
      <form
        className="mx-auto grid max-w-screen-md gap-4"
        name="contact"
        method="POST"
        data-netlify="true"
        action="/dummy_thanks.html"
      >
```

data-netlify="true"という記述がポイントです。静的なHTMLの場合はこれだけで問い合わせフォームが有効になります。残念ながらNetlifyはJavaScriptを解析してくれないので、ダミーのHTMLファイルを用意する必要があります。次のpublic/dummy_form.htmlを作成してください。

```
<!-- NetlifyのFormを利用するための静的HTML -->
<form
  name="contact"
  data-netlify="true"
  netlify-honeypot="bot-field"
  action="/dummy_thanks.html"
  hidden
>
  <input type="text" name="name" />
```

```
    <input type="text" name="company" />
    <input type="email" name="email" />
    <input type="text" name="name" />
    <textarea name="message"></textarea>
  </form>
```

同様に、public/dummy_thanks.htmlも作成します。

```
<!DOCTYPE html>
<html lang="ja">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <!-- 本当のthanksページに遷移させる -->
    <meta http-equiv="refresh" content="0;url=/thanks" />
    <title>FM Mail</title>
  </head>
  <body>
    <br />
  </body>
</html>
```

dummy_form.htmlの役割は、Netlifyにフォームの存在と項目名を教えることです。dummy_thanks.htmlは問い合わせフォーム送信後のリダイレクトを受けて、本当のサンクスページへさらにリダイレクトします。

dummy_form.htmlをデプロイすると、「Site settings」→「Forms」で問い合わせフォームの通知先を設定可能になります。筆者はメールとSlackを通知先に設定しています。

設定が完了したら、動作確認してみましょう。

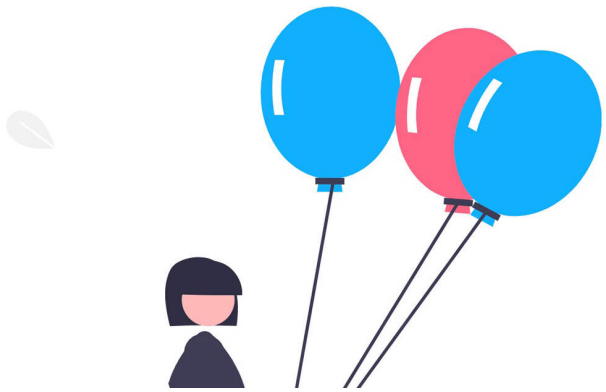
図7.6: お問い合わせフォーム

ランディングページ下方のお問い合わせフォームに適当な内容を入力して、送信してください。サンクスページが表示され、入力した内容が設定した通知先に通知されれば成功です。

図7.7: サンクスページ

本システムは開発中です。ご利用いただけません。

お問い合わせありがとうございます！



7.4　OGP画像を設定しよう

現時点ではTwitterなどにURLを載せて紹介しても、文字しか表示されません。そこでindex.htmlを次のように修正して、OGP画像を設定しましょう。OGP用の画像はpublicディレクトリに配置します。

```
<!DOCTYPE html>
<html lang="ja">
+ <head prefix="og: http://ogp.me/ns#">
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml" href="/src/favicon.svg" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta name="robots" content="noindex">
+   <meta property="og:url" content="https://{デプロイ先のサブドメイン}.netlify.app/" />
+   <meta property="og:type" content="website" />
+   <meta property="og:description" content="FileMakerの標準機能だけでメール受信を可能にするWebサービス" />
+      <meta  property="og:image"  content="https://{デプロイ先のサブドメイン}.netlify.app/FM_Mail_OGP.png" />
+   <meta property="og:title" content="FM Mail" />
    <title>FM Mail</title>
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.tsx"></script>
    <script>
      window.global = window;
      window.process = {
```

```
          env: { DEBUG: undefined },
        };
        var exports = {};
      </script>
    </body>
  </html>
```

すると、次のようにリンクが画像付きで表示されます。

図7.8: OGP表示例

第8章　Stripeでサブスクリプションを実装しよう

本章ではStripeのAPIを利用して、クレジットカード決済によるサブスクリプションを実装します。主な内容は次のとおりです。

・決済画面の呼び出し

・クレジットカード決済に伴う有料版APIの発行

・カスタマーポータルの呼び出し

・サブスクリプション解約に伴う有料版APIの削除

Stripeにはわかりやすいドキュメントが用意されています。[1]事前に読んでおくことをお勧めします。

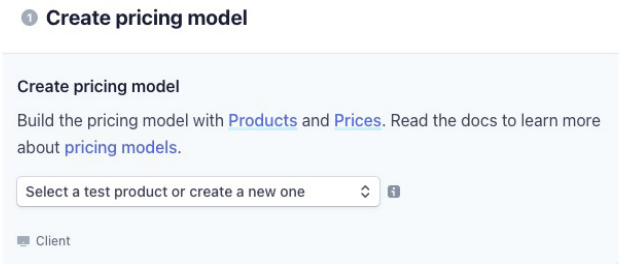
8.1　Stripe側の準備

まずStripe側の設定をします。ここで説明するのは、本書のWebサービス実装に最低限必要な箇所のみです。アカウントの作り方やビジネス設定については説明しないので、必要に応じてご自身で調査してください。なお、本書のStripe設定は全てテスト環境のものです。

8.1.1　商品の登録

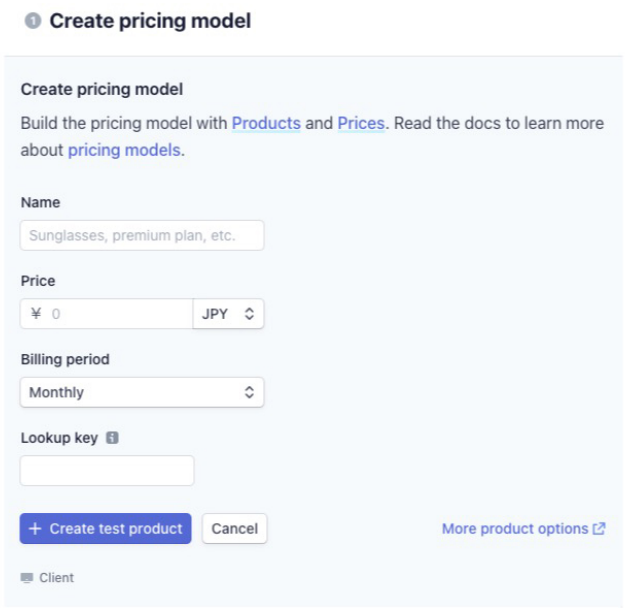
Stripeのドキュメントで「サブスクリプション」→「クイックスタート」を表示してください。[2]上の方に「1 Create pricing model」があるはずです。

図8.1: Create pricing model（選択前）



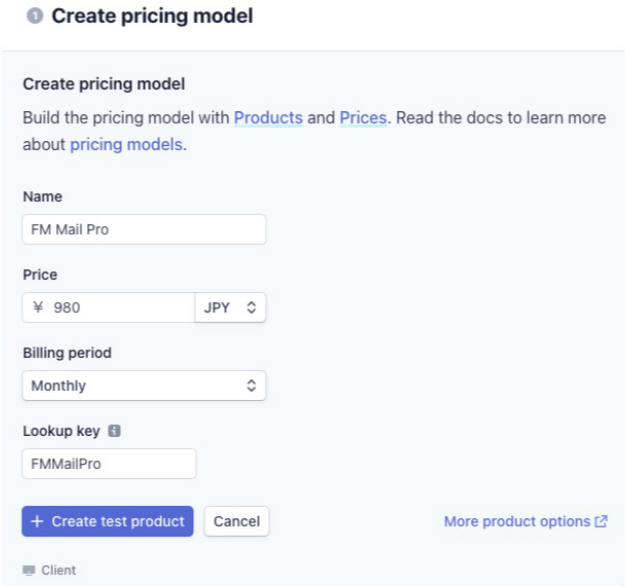
選択肢の中から「Add new test product」を選択してください。表示が次のように変わります。

図8.2: Create pricing model（選択後）



商品名と金額は適当に入力してください。Lookup keyはFMMailProと入力してください。

図8.3: Create pricing model（入力後）



入力したら「Create test product」をクリックして商品を登録してください。登録した内容はStripeのポータルから確認できます。

商品はStripeのポータルでも登録できますが、その場合は検索キー（Lookup key）が付与されません。検索キーをあとから付与したい場合は、StripeのAPIを呼び出す必要があります。Pythonのサンプルコードは次のとおりです。もし検索キーが付与されていなかったら、このコードを実行してください。

```
import stripe

stripe.api_key = "sk_test_xxxxxxxxxxxxxxxxxxxxxxxx"

stripe.Price.modify("price_xxxxxxxxxxxxxxxxxxxxxx", lookup_key="FMMailPro")
```

8.1.2　カスタマーポータルの設定

次はカスタマーポータルの設定です。「設定」→「カスタマーポータル」を表示してください。長いので前後に分けます。

図8.4: カスタマーポータルの設定（機能の設定）

前半は機能の設定です。「顧客によるサブスクリプションのキャンセルを許可」を有効にしてください。また「今すぐキャンセル」も選択してください。[3]

図8.5: カスタマーポータルの設定（ビジネス設定）

見出し

顧客に対して表示するメッセージをカスタマーポータル全体でカスタマイズしましょう。

FM Mailの請求情報を表示しています。

リンク

利用規約
顧客がサブスクリプションの変更を確認したとき、または支払い方法を追加したときに表示されます。空白の場合は、代わりにアカウントの詳細が使用されます。

https:/[redacted]netlify.app/terms

プライバシーポリシー
顧客がサブスクリプションの変更を確認したとき、または支払い方法を追加したときに表示されます。空白の場合は、代わりにアカウントの詳細が使用されます。

https:/[redacted]netlify.app/privacy_policy

デフォルトのリダイレクトリンク
アカウント管理の終了後に顧客をどこにリダイレクトするかを選択します。API を使用して、より詳細に制御することもできます。もっと知る

https:/[redacted]netlify.app/mypage

後半はビジネス設定です。カスタマーポータルに表示する文言と各種リンクを設定します。

8.2　AWS側の準備

AWS側ではStripe関連の情報を格納するために、Cognitoユーザープールのカスタム属性とDynamoDBのテーブルを追加します。

8.2.1　Cognitoユーザープールのカスタム属性追加

「Amazon Cotnito」→「ユーザープール」で対象のユーザープールを開き、「サインアップエクスペリエンス」タブでカスタム属性を追加してください。

図8.6: カスタム属性の追加

追加するのはstripe_customer_idです。追加の手順は6.3.3で詳しく説明しています。接頭語のcustom:は自動で追加されるので、登録時は入力しないでください。

次は「アプリケーションの統合」タブでアプリケーションクライアントを開き、「属性の読み取りおよび書き込み許可」を開いてください。

図8.7: 属性の読み取りおよび書き込み許可

stripe_customer_idには読み取り許可のみを与えてください。もし書き込み許可を与えるとユーザーによるなりすましが可能になり、セキュリティリスクが生じるので注意してください。

8.2.2 DynamoDBテーブルの追加

DynamoDBにはテーブルをふたつ追加します。最初のテーブルはFM_Mail_Customerテーブルです。

図8.8: FM_Mail_Customerテーブル

パーティションキーはCustomerIdとしてください。ソートキーは不要です。

次のテーブルはFM_Mail_Stripeテーブルです。

図8.9: FM_Mail_Stripeテーブル

パーティションキーはSessionIdとしてください。こちらもソートキーは不要です。

AWS側の準備は以上です。

8.3　バックエンドの実装

　バックエンドはサブスクリプション契約をして有料版にアップグレードするための処理とサブスクリプションを解約して、ダウングレードするための処理に分かれています。いずれもChaliceで実装します。実装後は次のコマンドを実行して、デプロイしてください。

```
chalice deploy
```

8.3.1　アップグレード用の処理

　アップグレード用の処理はfm_mail_create_api_key_proディレクトリに格納されています。バックエンドとしては、本書でもっとも複雑な処理です。かなり長いので、全体像はサンプルコードで確認してください。

　また、デプロイ後は対応するIAMのロールに権限を追加してください。

図8.10: アップグレード処理の許可ポリシー

　アップグレード処理に必要な権限は、次のとおりです。

・AmazonDynamoDBFullAccess

・AmazonAPIGatewayAdministrator

・AmazonCognitoPowerUser

　それでは、コードの内容を先頭から説明していきます。

```
USER_POOL_ARN = os.environ.get("USER_POOL_ARN")
USER_POOL_NAME = os.environ.get("USER_POOL_NAME")
USER_POOL_ID = os.environ.get("USER_POOL_ID")
DYNAMODB_API_KEY_TABLE = os.environ.get("DYNAMODB_API_KEY_TABLE")
DYNAMODB_CUSTOMER_TABLE = os.environ.get("DYNAMODB_CUSTOMER_TABLE")
DYNAMODB_STRIPE_TABLE = os.environ.get("DYNAMODB_STRIPE_TABLE")
REGION_NAME = os.environ.get("REGION_NAME")
STRIPE_API_KEY = os.environ.get("STRIPE_API_KEY")
CHALICE_DOMAIN = os.environ.get("CHALICE_DOMAIN")
MY_DOMAIN = os.environ.get("MY_DOMAIN")
REST_API_ID = os.environ["REST_API_ID"]
USAGE_PLAN_ID = os.environ["USAGE_PLAN_ID"]
```

最初は環境変数の読み込みです。アップグレード処理では、多くの環境変数が使われています。設定値は次のとおりです。

環境変数	設定値
USER_POOL_ARN	ユーザープールのARN。「ユーザープールの概要」で確認可能。
USER_POOL_NAME	ユーザープール名。「ユーザープールの概要」で確認可能。
USER_POOL_ID	ユーザープールのID。「ユーザープールの概要」で確認可能。
DYNAMODB_API_KEY_TABLE	APIキーを格納するDynamoDBのテーブル名。
DYNAMODB_CUSTOMER_TABLE	Stripeの顧客情報を格納するDynamoDBのテーブル名。
DYNAMODB_STRIPE_TABLE	StripeのセッションIDを格納するDynamoDBのテーブル名。
REGION_NAME	AWSのリージョン名。
STRIPE_API_KEY	StripeのAPIキー。Stripeの開発ダッシュボードで確認可能。
CHALICE_DOMAIN	このAPIのデプロイ先URL。chalice urlコマンドで確認可能。
MY_DOMAIN	Webサービスのデプロイ先URL。本書の手順どおり実装している場合はNetlifyのURLとなる。
REST_API_ID	メール受信APIのID。API GatewayのAPI一覧で確認可能。
USAGE_PLAN_ID	使用量プランのID。API Gatewayの使用量プランで確認可能。有料版のIDを設定する。

Chaliceの環境変数はconfig.jsonに記載します。サンプルコードにはconfig.json.templateを用意したので、コピーして利用してください。

次は各種リソースの初期処理です。

```
# Stripe初期設定
stripe.api_key = STRIPE_API_KEY

# Cognitoで認証する
authorizer = CognitoUserPoolAuthorizer(USER_POOL_NAME, provider_arns=[USER_POOL_
ARN])

# DynamoDBに接続
dynamodb = boto3.resource("dynamodb", region_name=REGION_NAME)
api_key_table = dynamodb.Table(DYNAMODB_API_KEY_TABLE)
customer_table = dynamodb.Table(DYNAMODB_CUSTOMER_TABLE)
stripe_table = dynamodb.Table(DYNAMODB_STRIPE_TABLE)

# API Gatewayの設定用クライアント
apigateway_cli = boto3.client("apigateway")

# Cognitoの設定用クライアント
cognito_cli = boto3.client("cognito-idp")
```

内容はコメントのとおりですので、説明は省略します。次はアップグレードボタンのクリック時に呼び出される処理です。

```
@app.route("/create-checkout-session/{lookup_key}/{user_name}", cors=True)
def create_checkout_session(lookup_key, user_name):
    # 検索キーから価格を取得する
            prices    =    stripe.Price.list(lookup_keys=[lookup_key],    expand=
```

```
["data.product"])


    # ワンタイムキーを生成する
    one_time_key = str(uuid.uuid4())


    # Stripeのセッションを作成する（success_urlにワンタイムキーが含まれているのがポイント）
    checkout_session = stripe.checkout.Session.create(
        line_items=[
            {
                "price": prices.data[0].id,
                "quantity": 1,
            },
        ],
        mode="subscription",
        success_url=CHALICE_DOMAIN
        + "/create-api-key/{CHECKOUT_SESSION_ID}/"
        + one_time_key,
        cancel_url=MY_DOMAIN + "/canceled_upgrade",
    )


    # セッションIDとUserNameをDynamoDBに登録する
    dt_now_jst = datetime.datetime.now(datetime.timezone(datetime.timedelta(hour
s=9)))
    with stripe_table.batch_writer() as batch:
        batch.put_item(
            Item={
                "SessionId": checkout_session.id,
                "UserName": user_name,
                "PaidFlag": False,
                "OneTimeKey": one_time_key,
                "CreatedAt": dt_now_jst.strftime("%Y-%m-%d %H:%M:%S"),
```

```
                "UpdatedAt": dt_now_jst.strftime("%Y-%m-%d %H:%M:%S"),
            }
        )

    # Stripeに遷移する
    return Response(
        status_code=302, body="", headers={"Location": checkout_session.url}
    )
```

このcreate-checkout-sessionメソッドで実施している主な処理は次のとおりです。

・Stripeのセッションを作成

・セッションIDとCognitoのユーザー名をDynamoDBに登録

・Stripeの決済画面へ遷移

また、このメソッドでは引数をふたつ使用しています。lookup_keyは販売する商品に対応するキーです。本書で扱う商品はひとつしかありませんが、この引数により複数の商品にも対応可能です。user_nameはCognitoのユーザー名です。

Stripeのセッションを作成する際、UUIDのワンタイムキーを用いてsuccess_urlに設定しています。success_urlはクレジットカード決済に成功したあとで遷移するURLです。セッションIDはユーザーからも見えますが、ワンタイムキーは決済が成功するまでユーザーからは見えません。

次は決済成功後に呼び出される処理です。ここで先ほど登場したワンタイムキーが不正防止に用いられています。

```
@app.route("/create-api-key/{session_id}/{one_time_key}", cors=True)
def create_api_key(session_id, one_time_key):
    # チェックアウトセッションの状態を確認
    checkout_session = stripe.checkout.Session.retrieve(session_id)
    payment_status = checkout_session.payment_status

    # 支払い済みでなければキャンセルページに遷移
    cancel_url = MY_DOMAIN + "/canceled_upgrade"
```

```
    if payment_status != "paid":
                return  Response(status_code=302,  body="",  headers={"Location":
cancel_url})

    # カスタマーIDを取得
    customer_id = checkout_session.customer

    # DynamoDBからセッションIDに紐づく情報を取り出す
    result = stripe_table.get_item(Key={"SessionId": session_id})
    user_name = result["Item"]["UserName"]
    paid_flag = result["Item"]["PaidFlag"]
    one_time_key_dynamo_db = result["Item"]["OneTimeKey"]

    # ワンタイムキーが一致しない場合、もしくはAPIキー発行済みの場合はキャンセルページに遷移
    if one_time_key != one_time_key_dynamo_db or paid_flag:
                return  Response(status_code=302,  body="",  headers={"Location":
cancel_url})

    # APIキーを発行
    result = apigateway_cli.create_api_key(
        name="fm_mail_pro_" + user_name,
        enabled=True,
        stageKeys=[{"restApiId": REST_API_ID, "stageName": "api"}],
    )

    # 発行したAPIキーの値とIDを取得
    api_key = result["value"]
    api_key_id = result["id"]

    # APIキーに使用量プランを適用
    apigateway_cli.create_usage_plan_key(
```

```
        usagePlanId=USAGE_PLAN_ID, keyId=api_key_id, keyType="API_KEY"
    )

    # DynamoDBにAPIキーを登録
    with api_key_table.batch_writer() as batch:
        batch.put_item(
            Item={
                "UserId": user_name,
                "Type": "PRO",
                "ApiKey": api_key,
                "ApiKeyId": api_key_id,
            }
        )

    # DynamoDBにStripeのカスタマーIDとAPIキーIDを登録
    with customer_table.batch_writer() as batch:
        batch.put_item(
            Item={
                "CustomerId": customer_id,
                "UserId": user_name,
                "ApiKeyId": api_key_id,
            }
        )

    # DynamoDBの支払い済みフラグを更新
    dt_now_jst = datetime.datetime.now(datetime.timezone(datetime.timedelta(hour
s=9)))
    stripe_table.update_item(
        Key={"SessionId": session_id},
              ExpressionAttributeNames={"#PaidFlag":  "PaidFlag",  "#UpdatedAt":
"UpdatedAt"},
```

```
        ExpressionAttributeValues={
            ":PaidFlag": True,
            ":UpdatedAt": dt_now_jst.strftime("%Y-%m-%d %H:%M:%S"),
        },
        UpdateExpression="SET #PaidFlag = :PaidFlag, #UpdatedAt = :UpdatedAt",
    )

    # ユーザープールのカスタム属性を更新
    cognito_cli.admin_update_user_attributes(
        UserPoolId=USER_POOL_ID,
        Username=user_name,
        UserAttributes=[
            {"Name": "custom:plan_type", "Value": "PRO"},
            {"Name": "custom:stripe_customer_id", "Value": customer_id},
        ],
    )

    # サンクスページに遷移する
    success_url = MY_DOMAIN + "/thanks_upgrade"

    return Response(status_code=302, body="", headers={"Location": success_url})
```

このcreate_api_keyメソッドでは、セッションIDとワンタイムキーが引数として渡されます。セッションIDを用いると、Stripeから支払いステータスとカスタマーIDを取得できます。支払いステータスが支払い済み（paid）でない場合は、キャンセルページに遷移して終了します。

さらにセッションIDを用いて、DynamoDBからCognitoのユーザー名と支払い済みフラグ、ワンタイムキーを取り出します。引数で渡したワンタイムキーとDynamoDBから取り出したワンタイムキーが一致しなければ、キャンセルページに遷移して終了します。また支払い済みフラグが立っていた場合はAPIが発行済みなので、同様にキャンセルページに遷移して終了します。

次はAPIの発行処理です。使用量プランのIDが異なるだけで、処理の内容は6章の無料版APIキーの発行とまったく同じです。Cognitoのユーザー名は先ほどDynamoDBから取り出した値を用いています。

次はDynamoDBの支払い済みフラグを更新します。最後にユーザープールのカスタム属性を更新し、サンクスページに遷移して終了です。カスタム属性はplan_typeをPROに変更し、stripe_customer_idには冒頭で取得したカスタマーIDを保存しています。ここで保存したカスタマーIDは、カスタマーポータルの呼び出しで使用します。

次はカスタマーポータルの呼び出し処理です。

```
@app.route("/create-billing-portal-by-user", authorizer=authorizer, cors=True)
def create_billing_portal_by_user():
    # 認証情報からUserNameを取り出す
    context = app.current_request.context
    user_name = context["authorizer"]["claims"]["cognito:username"]

    # ユーザープールのカスタム属性からStripeのカスタマーIDを取得
            user_info   =   cognito_cli.admin_get_user(UserPoolId=USER_POOL_ID,
Username=user_name)
    user_attributes = user_info["UserAttributes"]
    customer_id = [
                    x["Value"]  for  x  in  user_attributes  if  x["Name"]  ==
"custom:stripe_customer_id"
    ][0]

    # 請求ポータルのURLを生成する
    return_url = MY_DOMAIN + "/upgrade"
    portal_session = stripe.billing_portal.Session.create(
        customer=customer_id,
        return_url=return_url,
    )
```

```
    billing_portal_url = portal_session.url
    resp = {
        "status": "OK",
        "billing_portal_url": billing_portal_url,
    }

    # 結果を返す
    return json.dumps(resp, ensure_ascii=False)
```

このメソッドのみCognitoの認証が必要です。引数はなく認証情報からユーザー名を取り出して利用します。ユーザー名を用いてCognitoのカスタム属性を取得し、StripeのカスタマーIDを取り出します。カスタマーIDを用いてカスタマーポータルのURLを生成し、JSON形式で返します。

以上が、アップグレード用のバックエンド処理の説明となります。

8.3.2 ダウングレード用の処理

ダウングレード用の処理はfm_mail_stripe_webhookディレクトリに格納されています。その名のとおり、StripeのWebhookとして機能するAPIです。内容は次のとおりで、アップグレード用のAPIと比べると、かなり単純です。

```
import boto3
import os
import stripe
from chalice import Chalice

app = Chalice(app_name="fm_mail_stripe_webhook")

# 環境変数
USER_POOL_ID = os.environ.get("USER_POOL_ID")
DYNAMODB_API_KEY_TABLE = os.environ.get("DYNAMODB_API_KEY_TABLE")
DYNAMODB_CUSTOMER_TABLE = os.environ.get("DYNAMODB_CUSTOMER_TABLE")
```

```
REGION_NAME = os.environ.get("REGION_NAME")
STRIPE_ENDPOINT_SECRET = os.environ["STRIPE_ENDPOINT_SECRET"]


# DynamoDBに接続
dynamodb = boto3.resource("dynamodb", region_name=REGION_NAME)
api_key_table = dynamodb.Table(DYNAMODB_API_KEY_TABLE)
customer_table = dynamodb.Table(DYNAMODB_CUSTOMER_TABLE)


# API Gatewayの設定用クライアント
apigateway_cli = boto3.client("apigateway")


# Cognitoの設定用クライアント
cognito_cli = boto3.client("cognito-idp")



@app.route("/webhook", methods=["POST"])
def webhook():
    event = None
    # リクエストパラメータの解析
    payload = app.current_request.raw_body
    sig_header = app.current_request.headers["stripe-signature"]

    try:
        # イベントの解析
        event = stripe.Webhook.construct_event(
            payload, sig_header, STRIPE_ENDPOINT_SECRET
        )
    except ValueError as e:
        raise e
    except stripe.error.SignatureVerificationError as e:
        raise e
```

```
# サブスクリプションの解約以外は何もしない
if event["type"] != "customer.subscription.deleted":
    return {"success": True}

# StripeのカスタマーID
customer_id = event["data"]["object"]["customer"]

# DynamoDBからカスタマーIDに紐づくCognitoユーザーIDを取り出す
result = customer_table.get_item(Key={"CustomerId": customer_id})
user_id = result["Item"]["UserId"]
api_key_id = result["Item"]["ApiKeyId"]

# APIキーを削除
result = apigateway_cli.delete_api_key(apiKey=api_key_id)

# DynamoDBからPRO版のAPIキーを削除
api_key_table.delete_item(Key={"UserId": user_id, "Type": "PRO"})

# ユーザープールのカスタム属性を更新
cognito_cli.admin_update_user_attributes(
    UserPoolId=USER_POOL_ID,
    Username=user_id,
    UserAttributes=[
        {"Name": "custom:plan_type", "Value": "FREE"},
        {"Name": "custom:stripe_customer_id", "Value": customer_id},
    ],
)

return {"success": True}
```

デプロイ後は、対応するIAMのロールに権限を追加してください。

図8.11: ダウングレード処理の許可ポリシー

ダウングレード処理に必要な権限は、次のとおりです。アップグレード処理と同じですね。

・AmazonDynamoDBFullAccess

・AmazonAPIGatewayAdministrator

・AmazonCognitoPowerUser

環境変数はアップグレード処理とほとんど共通ですが、STRIPE_ENDPOINT_SECRETだけは独自です。ここには、Webhookの署名シークレットを設定します。

それでは、コードについて説明します。このAPIはStripeのカスタマーポータルでユーザーがサブスクリプションを解約したときに、Stripeから呼び出されます。冒頭でイベントを解析していますが、サブスクリプションの解約時に発生するイベントは、customer.subscription.deletedです。もしWebhook設定を間違えて別のイベントが送られてきた場合は、無視するようになっています。また、Stripe以外からアクセスすると、イベント解析時にstripe.error.SignatureVerificationErrorが発生します。これにより、なりすましによる不正なダウングレードを防止できます。

イベントからカスタマーIDを取り出し、それを用いてDynamoDBからCognitoユーザーIDとAPIキーのIDを取得します。あとは、取得したIDを用いてアップグレードとは逆の処理を実施します。API GatewayからAPIキーを削除し、DynamoDBから有料版のAPIキーを削除し、Cognitoのカスタム属性を更新して終了です。

8.3.3　Webhookの登録

ダウングレード処理はStripe側でWebhookとして登録する必要があります。開発者向けのダッシュボードからWebhookを開いて、エンドポイントを追加してください。

図8.12: エンドポイントを追加

「エンドポイントURL」には、ダウングレード用のAPIのデプロイURLを設定します。デプロイURLはchalice urlコマンドで確認できます。URLの後ろに/webhookを追加するのも忘れないようにしましょう。

「リッスンするイベントの選択」ではcustomer.subscription.deletedを選択します。図8.13は実際の設定例です。

図8.13: Webhook設定の例

ローカルでテストする場合は、Stripe CLIを使うと便利です。Stripe CLIの使い方は本書では説明しません。「Stripe CLIの本」[4]の解説がわかりやすかったです。

8.4　フロントエンドの実装

本章におけるフロントエンドの役割は次のとおりです。

・決済画面の呼び出し

・請求ポータルの呼び出し

順番に見ていきましょう。

8.4.1　決済画面の呼び出し

決済画面の呼び出しは次のとおりです。主要な部分のみを抜き出しているので、実際のコードはサンプルコードのsrc/components/UpgradeContent.tsxで確認してください。

```
const [user] = useAtom(stateCurrentUser);
const username = user?.username;

// Stripe決済用URL
const stripeUrl = `${
  import.meta.env.VITE_STRIPE_BASE_URL
}/create-checkout-session/FMMailPro/${username}`;

...

 <a href={stripeUrl}>
  <button type="button">PROプランに切り替え</button>
 </a>
```

内容はとても単純で、Cognitoのユーザー名を使ってURLを組み立て、<a>タグでリンクを作っているだけです。環境変数のVITE_STRIPE_BASE_URLには、バックエンドで実装したアップグレード用のAPIのURLを設定します。「Stripe決済用URL」の組み立てで登場するFMMailProは

「8.1.1 商品の登録」で確認した料金体系の検索キーです。もし検索キーがちがう値になっていた場合は、この箇所を書き換えてください。

8.4.2　カスタマーポータルの呼び出し

　カスタマーポータルの呼び出しは、アップグレード後のアップグレード画面とユーザー情報画面の2か所から行われます。共通処理となるので、src/function/StripeUtil.tsに実装しています。

```
import ky from 'ky';
import type { CognitoUser } from '../atom/User';

type BillingPortalResponse = {
  billing_portal_url: string;
};

// Stripe請求ポータルを呼び出す
const openBillingPortal = async (user: CognitoUser | null) => {
  // Stripeの請求ポータル呼び出し用URL
  const stripeMyPortalUrl = `${
    import.meta.env.VITE_STRIPE_BASE_URL
  }/create-billing-portal-by-user`;

  if (!user) return;
  const resp: BillingPortalResponse = await ky
    .get(stripeMyPortalUrl, {
      headers: {
        Authorization: `Bearer ${user.signInUserSession.idToken.jwtToken}`,
      },
    })
    .json();
  // Stripe請求ポータルに移動
```

```
  window.location.href = resp.billing_portal_url;
};


export default openBillingPortal;
```

カスタマーポータルの呼び出しはCognitoの認証が必要なので、kyを用いてHTTPヘッダを付与して呼び出しています。APIのレスポンスからカスタマーポータルのURLを取り出し、最後は画面遷移します。

呼び出し元では次のように記述します。

```
<button
  type="button"
  onClick={() => openBillingPortal(user)}
>
  カスタマーポータル
</button>
```

実際の記述はUpgradeContent.tsxやUserInfoContent.tsxで確認してください。

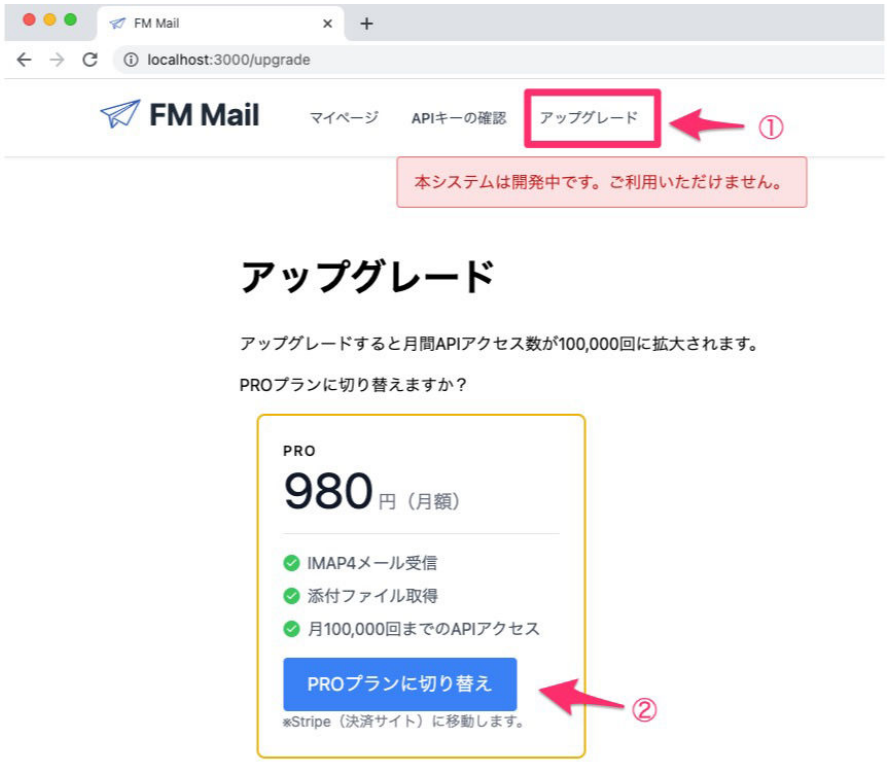
8.5　動作確認

それでは動作確認してみましょう。環境変数を設定するのを忘れないよう気を付けてください。

8.5.1　アップグレードの確認

アップグレード画面は、サインイン後に「アップグレード」をクリックすると表示できます。

図8.14: アップグレード画面（アップグレード前）



「PROプランに切り替え」をクリックすると、バックエンドのcreate_checkout_sessionが呼び出されて、Stripe決済画面に遷移します。

図8.15: Stripe決済画面

テスト環境では、テストカードを利用できます。

図8.16: テスト用のカード情報の例

カード番号に4242 4242 4242 4242を入力すると、決済が成功します。4000 0000 0000 9995を入力すると、決済が失敗します。カード有効期限とCVCは適当でかまいません。

「申し込む」をクリックすると、バックエンドでcreate_api_keyが呼び出されて、サンクスページに遷移します。「APIキーの確認」をクリックしてみましょう。

図8.17: APIキーの確認

このようにプラン種別が「PRO」のAPIキーが表示されていたら成功です。ほかにもマイページやアップグレード画面、ユーザー情報画面が変化しているので確認してみましょう。

8.5.2　ダウングレードの確認

次はダウングレードの動作確認をします。アップグレード画面を表示してください。

図8.18: アップグレード画面（アップグレード後）

アップグレード後はこのような画面になります。「カスタマーポータルを表示」をクリックしてください。バックエンドでcreate_billing_portal_by_userが呼び出されて、Stripeのカスタマーポータル

に遷移します。

図8.19: カスタマーポータル

　カスタマーポータルが表示されたら、「プランをキャンセル」をクリックしてください。もしボタンが表示されていなければ、「8.1.2 カスタマーポータルの設定」を見直しましょう。デフォルトでは、顧客によるサブスクリプションのキャンセルは無効になっています。「プランをキャンセル」をクリックすると、次の確認画面に遷移します。

図8.20: プランをキャンセル

確認画面でも「プランをキャンセル」をクリックしてください。カスタマーポータルに遷移しますが、今度は「プランをキャンセル」ボタンが消えているはずです。このときバックエンドでは、Webhookが呼び出されています。

図8.21: プランをキャンセル後のカスタマーポータル

「○に戻る」（○にはStripeに設定した会社名が入る）をクリックしてください。アップグレード画面に遷移します。「APIキーの確認」をクリックしてください。

図8.22: APIキーの確認

このようにプラン種別が「FREE」のAPIキーだけが表示されていたら成功です。マイページやアップグレード画面、ユーザー情報画面もアップグレード前に戻っているので確認してみましょう。

1. https://stripe.com/docs/

2. https://stripe.com/docs/billing/quickstart

3. ここでは動作確認のしやすさを優先して「今すぐキャンセル」を選択しましたが、実際の運用では顧客の利便性やビジネス観点等を踏まえて十分検討したうえで判断してください。

4. https://zenn.dev/hideokamoto/books/e961b4bad92429

第9章　アカウントの削除に対応しよう

　ようやく最後の章となりました。本章ではアカウントの削除を実装します。たまに、登録は簡単なのにアカウント削除はとても手間がかかるWebサービス[1]に出会うことがありますが、非常に感じが悪いです。本書で作るWebサービスは、簡単にアカウントを削除できるようにしましょう。

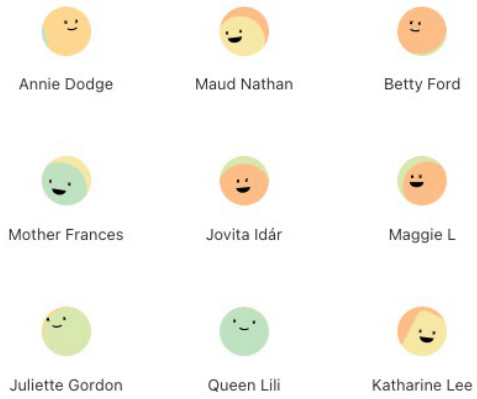
9.1　削除の前にアカウントの識別を可能にしよう

アカウント削除機能を実装する前に、アカウントの識別機能を実装します。サインイン後の画面右上に表示されるユーザーアイコンは、これまで固定値を用いていました。本サービスではユーザー間の交流などはないので、画像のアップロード機能は必要ありませんが、最低限ユーザーを識別できる機能は必要です。

図9.1: これまでのユーザーアイコン

アカウントを識別するだけなら、Identiconが適しています。GitHubでアカウントを作成したときの初期アイコンは典型的なIdenticonです。本書では、Boring Avatars[2]というIdenticonを導入します。

図9.2: Boring Avatars



このように、顔のアイコンになっています。かわいいですね。まずはインストールしましょう。

```
npm i boring-avatars
```

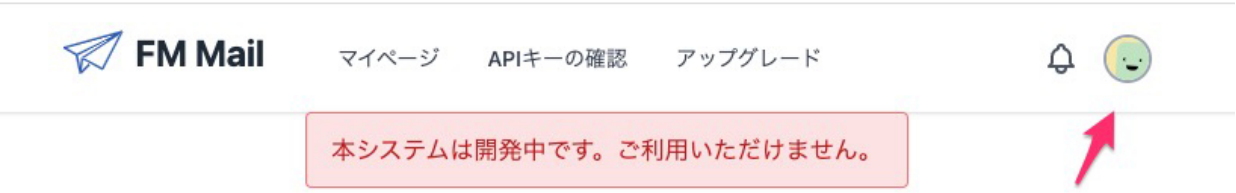
基本的な使い方は次のとおりです。

```
import Avatar from "boring-avatars";


<Avatar
  size={40}
  name="Hoge Huga"
  variant="beam"
  colors={["#FFBD87", "#FFD791", "#F7E8A6", "#D9E8AE", "#BFE3C0"]}
/>
```

nameの値によって、生成される画像が変わります。variantはmarble、beam、pixel、sunset、ring、bauhausの中から選べます。顔のアイコンはbeamです。colorsは使用するカラーパレットです。Boring AvatarsのWebサイト[3]に行くと、パラメータをいろいろ変更して試すことができます。

サンプルコードではsrc/components/Header2.tsxに実装されているので、確認してください。

図9.3: Boring Avatars導入後のユーザーアイコン



筆者のアイコンはこのようになりました。一人一人ちがうはずなので、確かめてみてください。

9.2　アカウント削除処理の実装

それでは、アカウント削除処理を実装していきます。バックエンドから説明します。

9.2.1　バックエンドの実装

バックエンドの主な機能は次のとおりです。

・Cognito認証

・ユーザーに紐づくAPIキーを削除

・DynamoDBからAPIキーを削除

・ユーザーを無効にする

ユーザーを削除ではなく、無効にしたのは理由があります。「5.2.3 サインインとサインアウト」で説明したとおり、AmplifyのfederatedSignInはサインインだけでなく、サインアップも兼ねています。ユーザーが存在しない状態でサインインすると、ユーザー登録とサインインが同時に行われるわけです。もしアカウント削除のバックエンド処理でユーザーを削除してしまうと、アカウント削除の直後でもサインインが可能になってしまいます。実際には新たなユーザーが作られてサインインしているわけですが、「アカウントが削除されていないのではないか」とユーザーに不信感を抱かせかねない挙動です。ユーザーを無効にした場合は完全にサインインできなくなるため、このような問題は生じません。[4]

実装は例によって、Chaliceで行います。

```
import boto3
import os
from chalice import Chalice, CognitoUserPoolAuthorizer


app = Chalice(app_name="fm_mail_delete_user")


# 環境変数
USER_POOL_ARN = os.environ.get("USER_POOL_ARN")
```

```
USER_POOL_NAME = os.environ.get("USER_POOL_NAME")
USER_POOL_ID = os.environ.get("USER_POOL_ID")
DYNAMODB_API_KEY_TABLE = os.environ.get("DYNAMODB_API_KEY_TABLE")
REGION_NAME = os.environ.get("REGION_NAME")

# Cognitoで認証する
authorizer = CognitoUserPoolAuthorizer(USER_POOL_NAME, provider_arns=[USER_POOL_
ARN])

# DynamoDBに接続
dynamodb = boto3.resource("dynamodb", region_name=REGION_NAME)
api_key_table = dynamodb.Table(DYNAMODB_API_KEY_TABLE)

# API Gatewayの設定用クライアント
apigateway_cli = boto3.client("apigateway")

# Cognitoの設定用クライアント
cognito_cli = boto3.client("cognito-idp")



@app.route("/delete-user", authorizer=authorizer, cors=True)
def delete_user():
    # 認証情報からUserNameを取り出す
    context = app.current_request.context
    user_name = context["authorizer"]["claims"]["cognito:username"]

    # DynamoDBからユーザーに紐づくAPIキーの情報を取り出す
    result = api_key_table.get_item(Key={"UserId": user_name, "Type": "FREE"})
    api_key_id = result["Item"]["ApiKeyId"]

    # APIキーを削除
```

```
    result = apigateway_cli.delete_api_key(apiKey=api_key_id)


    # DynamoDBからFREE版のAPIキーを削除
    api_key_table.delete_item(Key={"UserId": user_name, "Type": "FREE"})


    # ユーザーを無効化
    cognito_cli.admin_disable_user(
        UserPoolId=USER_POOL_ID,
        Username=user_name,
    )


    return {"success": True}
```

環境変数は8章のアップグレード処理から流用しています。コメントどおりの処理をしているだけなので、特に難しいところはありませんね。

デプロイしたら、対応するIAMに権限を付与するのを忘れないようにしましょう。

図9.4: アカウント削除処理の許可ポリシー

アタッチする権限は次のとおりです。

・AmazonDynamoDBFullAccess

・AmazonAPIGatewayAdministrator

・AmazonCognitoPowerUser

9.2.2　フロントエンドの実装

フロントエンド側のアカウント削除処理は、src/components/UserInfoContent.tsxに実装されています。アカウント削除の箇所は次のとおりです。

```
  const deleteUser = async () => {
    const url = `${import.meta.env.VITE_DELETE_USER_BASE_URL}/delete-user`;
    try {
      if (user) {
        // ローディング表示開始
        setIsLoading(true);

        // ユーザー削除処理を呼び出す
        await ky
          .get(url, {
            headers: {
                                                    Authorization:    `Bearer
${user.signInUserSession.idToken.jwtToken}`,
            },
          })
          .json();

        // サインアウトする
        // eslint-disable-next-line @typescript-eslint/no-floating-promises
        Auth.signOut();
      }
    } catch (e) {
      setIsLoading(false);
      alert('エラーが発生しました。');
    }
  };
```

本書でも何度か登場したkyを用いて、アカウント削除APIを呼び出しています。アカウントを削除したあとは、サインアウトしています。

上述の関数を呼び出せばアカウントを削除できますが、有料版のユーザーの場合は注意が必要です。もしサブスクリプションを解約せずにアカウントを削除してしまうと、Stripeのカスタマーポータルにアクセスする手段が失われ、自力ではサブスクリプションを解約できなくなってしまいます。そこでボタンの表示箇所を次のようにして、有料版の場合はアカウント削除ができないようにしています。

```
  <div className="mt-5 flex items-center justify-between sm:col-span-2">
    <button
      type="button"
        className={`inline-flex rounded border-0 bg-blue-500 py-3 px-6 text-lg
text-white hover:bg-blue-600 focus:outline-none active:bg-blue-700 ${
        planType === 'PRO' ? '' : 'hidden'
      }`}
      // eslint-disable-next-line @typescript-eslint/no-misused-promises
      onClick={() => openBillingPortal(user)}
    >
      カスタマーポータル
    </button>
    <button
      type="button"
        className={`inline-flex rounded border-0 bg-red-400 py-3 px-6 text-lg
text-white hover:bg-red-500 focus:outline-none active:bg-red-600 ${
        planType === 'FREE' ? '' : 'hidden'
      }`}
      onClick={() => setIsOpen(true)}
    >
      アカウント削除
```

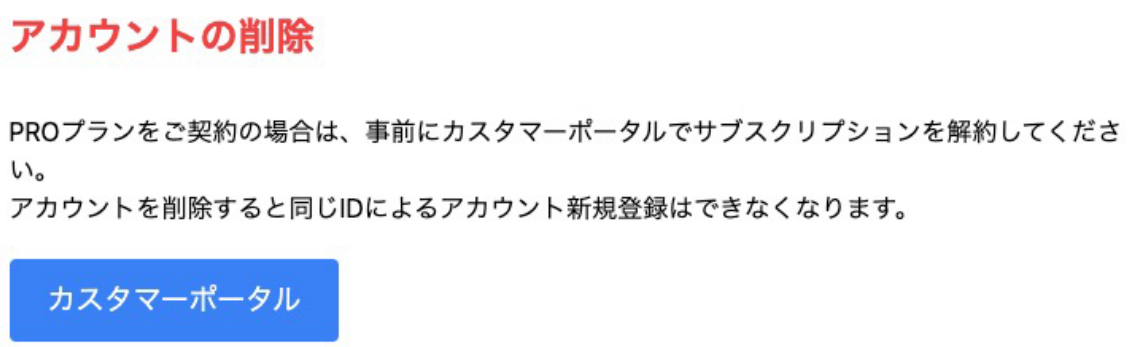
```
      </button>
    </div>
```

これでplanTypeがPROの場合はカスタマーポータルに遷移するボタンが表示され、planTypeがFREEの場合はアカウント削除ボタンが表示されます。

9.3　アカウント削除の動作確認

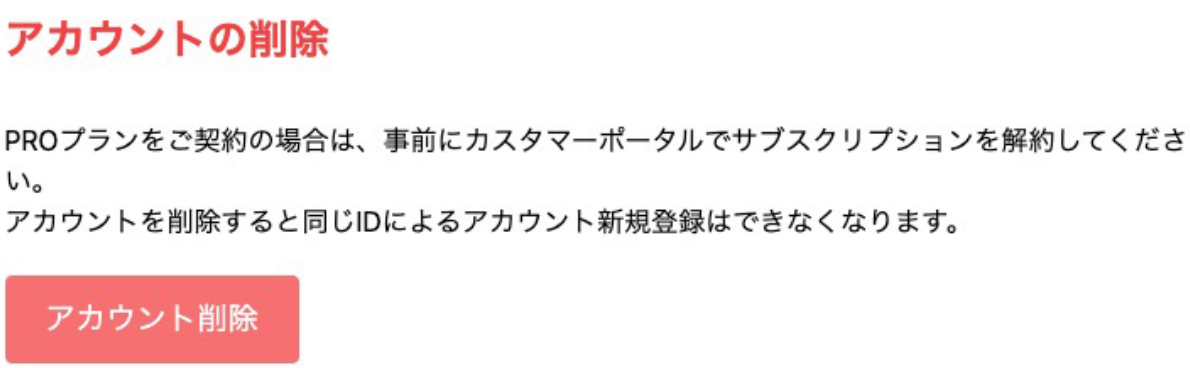
それでは、動作確認してみましょう。サインインしてユーザー情報画面を表示してください。ユーザー情報画面へのリンクは右上のユーザーアイコンをクリックすると現れます。

図9.5: 有料版のユーザー情報画面



有料版の場合はこのように「カスタマーポータル」ボタンが表示されます。8章を参考にカスタマーポータルでサブスクリプションの解約をしてください。

図9.6: 無料版のユーザー情報画面

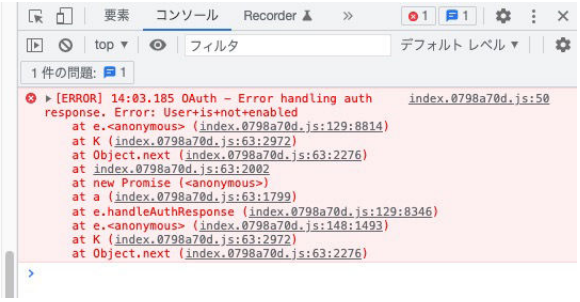


無料版の場合はこのように「アカウント削除」ボタンが表示されます。クリックすると確認ダイアログが表示されます。

図9.7: アカウント削除の確認ダイアログ

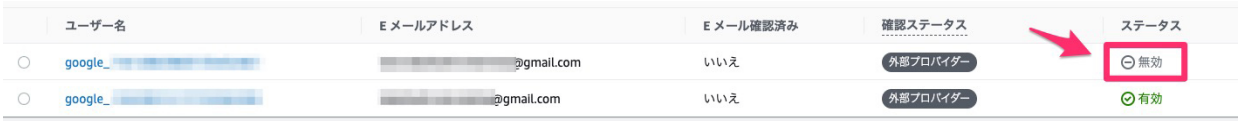
確認ダイアログの「アカウント削除」をクリックすると、バックエンド処理が呼び出されてサインアウトします。サインアウトしたら、先ほど削除したアカウントでもう一度サインインしてみてください。

図9.8: ログイン画面とエラーメッセージ



何度やってもログイン画面に遷移してしまうはずです。コンソールにはこのようにOAuth - Error handling auth response. Error: User+is+not+enabledというエラーメッセージが表示されます。

図9.9: ユーザープールの確認



Cognitoユーザープールを確認すると、該当のユーザーが無効になっていることがわかります。このユーザーを手動で削除すれば、再度ユーザー登録が可能になります。

9.4　最終型の動作確認

以上で全ての機能を実装できました。全てをひとつにまとめた最終型のソースコードを用意したので、参考にしてください。[5]また、筆者がデプロイしたテスト環境はこちらです。[6]

図9.10: 筆者のテスト環境

間違って一般の方に利用されないように「本システムは開発中です。ご利用いただけません」と表示していますが、実際には全ての機能が利用可能です。本書の読者の方であれば実際にユーザー登録して動作確認していただいてかまいません。テストカード番号を用いて有料版へのアップグレードもテスト可能です。[7]本書の実装でつまずいたときは、あるべき姿をこちらで確認してみてください。

1. 登録はネットからできるのに退会は郵送のみとか、もっとひどい場合は退会方法が存在しない場合もありますね。オンラインで行われた契約はオンラインで解約できるようにすることを義務付けるカリフォルニア州法がもっと広まればよいのに。
2. https://www.npmjs.com/package/boring-avatars
3. https://boringavatars.com/
4. 個人情報保護の観点から、実運用する際は無効にしたユーザーを完全に削除するバッチ処理を定期的に動かすとよいでしょう。
5. https://github.com/sikkimtemi/How_to_create_API_sales_service/tree/main/10-Complete
6. https://fmmail.netlify.app/
7. 悪用しないでくださいね。

あとがき

お読みいただきありがとうございます。本書は筆者が初めて執筆した書籍です。執筆だけでなく、ReactやCognito、DynamoDB、Stripeも全て初めてという、初めてづくしの体験でした。そんなmy-first-react-appを書籍化するという暴挙に出たのは、自分が読みたかったから、より正確には1年前の自分に読ませたかったからです。

1年前の筆者はAPI販売サービスを作るために、手探りで調査と実装を繰り返していました。APIキーという、たかだか数十文字の文字列を表示するだけの単純なサービスですが、それでも完成までにやらなければならないことの多さに圧倒されました。当時、本書が存在していたらどれほど楽だったことでしょう。

時間をさかのぼることはできませんが、当時の筆者と似たような状況に置かれている方は今もいるはずです。本書はその方たちのために書いたといっても、過言ではありません。応用しやすい作りにしたつもりなので、ぜひカスタマイズしてオリジナルのWebサービスを作り上げてください。[1]

謝辞

本書をレビューしていただいた@kk-ster様、@naofumi様、ありがとうございました。

Zennを開発した@catnose99様、「Zenn個人開発の限界に挑んだ話」[2]にはたいへん勇気付けられました。また、Zennというプラットフォームがなければ本書は生まれていませんでした。

「りあクト！」の@oukayuka様、3.1版の3部作には筆者の知りたかったことがほとんど載っていて感動しました。

そして時には支え、時には癒やしを与えてくれる3人と1匹の家族の皆、いつもありがとう。

1. そしてビジネスが成功したあかつきには、筆者を技術顧問として高給で雇ってください。
2. https://www.youtube.com/watch?v=DTpGfpLybr0

著者紹介

高橋 太郎（たかはし たろう）

職業はサーバーサイドエンジニア、情報処理安全確保支援士、猫の下僕などを兼任しています。現在はPython、React、FileMakerで開発することが多いです。数年前に横浜から鹿児島に移住しました。

◎本書スタッフ
アートディレクター/装丁：岡田章志＋GY
編集協力：山部 沙織
ディレクター：栗原 翔
〈表紙イラスト〉
鍋料理
社畜系お絵かきマンです。生存はSNSにて。

技術の泉シリーズ・刊行によせて

技術者の知見のアウトプットである技術同人誌は、急速に認知度を高めています。インプレスR&Dは国内最大級の即売会「技術書典」（https://techbookfest.org/）で頒布された技術同人誌を底本とした商業書籍を2016年より刊行し、これらを中心とした『技術書典シリーズ』を展開してきました。2019年4月、より幅広い技術同人誌を対象とし、最新の知見を発信するために『技術の泉シリーズ』へリニューアルしました。今後は「技術書典」をはじめとした各種即売会や、勉強会・LT会などで頒布された技術同人誌を底本とした商業書籍を刊行し、技術同人誌の普及と発展に貢献することを目指します。エンジニアの"知の結晶"である技術同人誌の世界に、より多くの方が触れていただくきっかけになれば幸いです。

株式会社インプレスR&D
技術の泉シリーズ　編集長　山城 敬

●お断り

掲載したURLは2022年10月1日現在のものです。サイトの都合で変更されることがあります。また、電子版ではURLにハイパーリンクを設定していますが、端末やビューアー、リンク先のファイルタイプによっては表示されないことがあります。あらかじめご了承ください。

●本書の内容についてのお問い合わせ先

株式会社インプレスR&D　メール窓口
np-info@impress.co.jp

件名に「『本書名』問い合わせ係」と明記してお送りください。
電話やFAX、郵便でのご質問にはお答えできません。返信までには、しばらくお時間をいただく場合があります。
なお、本書の範囲を超えるご質問にはお答えしかねますので、あらかじめご了承ください。
また、本書の内容についてはNextPublishingオフィシャルWebサイトにて情報を公開しております。
https://nextpublishing.jp/

技術の泉シリーズ

ReactとPythonで
API販売サービスを作ろう

2022年11月25日　初版発行Ver.1.0（リフロー版）

著　者　高橋 太郎

編集人　山城 敬

企画・編集　合同会社技術の泉出版

発行人　井芹 昌信

発　行　株式会社インプレスR&D

〒101-0051

東京都千代田区神田神保町一丁目105番地

https://nextpublishing.jp/

ISBN978-4-295-60145-6

NextPublishing®

◉本書はNextPublishingメソッドによって発行されています。
NextPublishingメソッドは株式会社インプレスR&Dが開発した、電子書籍と印刷書籍を同時発行できるデジタルファースト型の新出版方式です。 https://nextpublishing.jp/