



ACCESS テックブック 2

ACCESSのエンジニアが語るおいしい開発ライフ

プロローグ

本書について

本書「ACCESS テックブック 2」は、株式会社 ACCESS に所属するエンジニアによって書かれた技術同人誌で、2019 年 9 月の技術書典 7 に出典したテックブックの 2 冊目です。1 冊目の続きの章もあれば、新しい題材を取り扱う章もあります。各章はそれぞれ完結しているので、好きなページからお読みください。

株式会社 ACCESS には、ブラウザエンジン、IoT、スマホアプリ、クラウド、ネットワーク、ハードウェア、ドローンと、とても幅広い技術分野に対し、専門的な深い知識を持ったマニアックなエンジニア達が在籍しています。

株式会社 ACCESS について

日本でインターネットの歩みが始まった 1980 年代、「すべてのモノをネットにつなぐ」という企業ビジョンとともに、株式会社 ACCESS は誕生しました。ACCESS は、このビジョンを DNA として成長し、インターネットの普及とともに「ネットにつなぐ技術」を進化させ続けてきました。IT 革命元年と呼ばれた 1999 年には、世界で初めて「携帯電話をネットにつなぐ技術」の実用化に成功。企業躍進の起爆剤となりました。さまざまなイノベーションを経て、IoT の時代がいよいよ幕を開けようとしています。創業時より思い描いていたビジョンが現実のものになる今、ACCESS は「ネットにつなぐ」技術で、世界により豊かな社会と暮らしを創造し、人々の次の未来の実現を目指します。

<https://www.access-company.com/recruit/>

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。

目次

プロローグ	1
本書について	1
株式会社 ACCESS について	1
免責事項	1
第 1 章 Flow でクリーンアーキテクチャーを最適化する	6
1.1 はじめに	6
1.2 単方向データフロー	7
1.2.1 双方向データバインディングとは	7
1.2.2 単方向データフローとは	7
1.2.3 単方向にするメリットとは	8
1.2.4 Android アプリの変化	9
1.2.5 iOS アプリの変化	9
1.3 クリーンアーキテクチャーと単方向データフロー	10
1.3.1 Presenter を State に置き換え	11
1.3.2 結局 ViewModel は必要なのか	12
1.4 Kotlin Coroutines Flow	14
1.4.1 コールドストリーム	14
1.4.2 ホットストリーム	14
1.4.3 Kotlin のストリーム事情	15
1.4.4 Flow の基本的な使い方	15
1.4.5 SharedFlow と StateFlow	18
1.4.6 Flow と LiveData、Rx との比較	21
1.4.7 Flow 移行のメリット	21
1.5 Kotlin Coroutines Flow を Swift で observe する	23
1.5.1 環境構築	23

1.5.2	実装	25
1.5.3	プレビューと実行	30
1.6	Kotlin Coroutines Flow を Swift の Combine に変換する	34
1.7	あとがき	36

第 2 章 SvelteKit + FastAPI + vercel + heroku でやる気があれば誰でも簡単フルスタックエンジニア 37

2.1	はじめに	37
2.2	対象	38
2.3	想定環境	38
2.4	前置き・座学など	39
2.4.1	Svelte とは?	39
2.4.2	FastAPI とは?	40
2.4.3	vercel とは?	40
2.4.4	heroku とは?	41
2.4.5	アプリケーションの構成	42
2.5	フロントエンド開発の準備	45
2.6	SvelteKit をテンプレートから学ぶ	46
2.6.1	とりあえず起動してみる。	46
2.6.2	Svelte の主要機能を理解する	48
2.6.3	その他にも...	58
2.6.4	他のことをやらせてみよう	59
2.7	デプロイ	59
2.7.1	アカウント作成	59
2.7.2	GitHub にソースをあげる	60
2.7.3	Vercel のプロジェクト作成	60
2.8	バックエンド開発の準備	64
2.8.1	サンプルをダウンロード	64
2.9	heroku に登録	65
2.9.1	pgadmin のインストールと利用	68
2.10	FastAPI をテンプレートから学ぶ	73
2.10.1	リクエスト/レスポンスの実装	74
2.11	ドキュメント生成	77
2.11.1	ルーティング	79
2.11.2	Dependencies による依存性注入	80

2.11.3	補足説明	82
2.11.4	やってみよう追加課題	83
2.11.5	デプロイ	83
2.12	最後に	86
第 3 章	Android のファイルストレージに関する仕様の整理	88
3.1	はじめに	88
3.2	Android のファイルストレージに関する仕様の変遷	88
3.2.1	Android 4.4	89
3.2.2	Android 5.0	89
3.2.3	Android 6.0	89
3.2.4	Android 7.0	90
3.2.5	Android 8.0	90
3.2.6	Android 9.0	90
3.2.7	Android 10	90
3.2.8	Android 11	91
3.2.9	Android 12	92
3.3	Android のファイルストレージのユースケース	92
3.3.1	メディアファイル	93
3.3.2	メディア以外のファイル	94
3.4	おわりに	95
3.5	参考資料	96
第 4 章	業務効率化に貢献する e-sport	97
4.1	はじめに	97
4.2	タイピングの技術領域	97
4.3	用語	98
4.4	タイピングソフト及び各ルールへの適応	98
4.5	初速・正確率・スピード	99
4.5.1	初速を上げる (短くする)	99
4.5.2	正確率	100
4.5.3	スピード	100
4.6	ワードへの最適化	101
4.7	??? (未知の領域)	102
4.8	最後に	102

4.9	おまけ	102
第 5 章	デザインにおける「センス」は誰にでも身につけられるという話	104
5.1	そもそも「センス」ってなんだろう？	104
5.2	「センスが良いもの」を分解するとどうなる？	105
5.2.1	「センスの良さ」を掘り下げる	105
5.2.2	言語化した「センスの良さ」を知識の応用で再現する	106
5.3	センスの基である「観察・言語化・知識」はどうすれば身につくのか？	107
5.3.1	①「良い」と感じたものを掘り下げる	107
5.3.2	②「らしさ」を構成する要素を分解してみる	108
5.3.3	③トレースしてみる	108
5.3.4	④デッサンしてみる	108
5.4	センスを「使いこなす」には？	108
5.4.1	センスは適材適所で使わないと意味がない	109
5.4.2	そのセンスが「客観的なものであるか」を確認する	109
5.5	「センス」は誰にでも身につけられる	109
第 6 章	C#において abstract(抽象) クラス、Interface はどう使い分けるべきか	110
6.1	abstract(抽象) クラスとは	110
6.2	Interface とは	112
6.3	抽象クラス、Interface の違い	113
6.3.1	抽象クラスは実装を持つことができる	113
6.3.2	Interface は複数継承できる	114
6.4	抽象クラスは必要なのか	115
6.5	まとめ	116
第 7 章	プロジェクトマネージャがいきいきと躍動する働きやすい環境を作る	117
著者紹介		122

第 1 章

Flow でクリーンアーキテクチャーを最適化する

1.1 はじめに

tonionagauzzi です。弊社の一部チームでは、モバイルアプリ開発にクリーンアーキテクチャーを採用しています。2019 年に執筆した ACCESS テックブックでは、クリーンアーキテクチャーを使ってみた感想を紹介しました。

クリーンアーキテクチャーには、プロジェクトが大きくなって新しい人が介入しても、基本の設計ポリシーが保たれるという利点があります。しかし、モジュールやクラスの多さに伴ってデータの流がわかりづらくなるため、開発時間やレビュー時間、学習コストは、クリーンアーキテクチャー採用前と比べてさほど変わりませんでした。

本書では、それから数年経って、どのような改善を施しているかを紹介します。

一言で言えば、Unidirectional Data Flow（以下、単方向データフロー）をより強固にします。クリーンアーキテクチャーの思想を守りつつ、データの流をわかりやすくするためです。

本書の前半では、Kotlin Coroutines Flow（以下、Flow）を使い、簡単な例を掲載しながらそれについて説明します。

そして後半では、単方向データフローのロジックを Kotlin Multiplatform Mobile（以下、KMM）を使い Kotlin で実装し、UI 部分を Swift で実装する iOS アプリの例を掲載します。KMM は Android/iOS の両方がターゲットですが、Android はロジックと同じ Kotlin で労せず Flow を使えます。一方 iOS は Kotlin の Flow を Swift で受け取る必要があります。そこをどうすればよいのかを調査しました。

なお、クリーンアーキテクチャーの概要は、本書では触れていません。ACCESS テックブックの第 1 章、もしくは数ある書籍や検索をご活用ください。

1.2 単方向データフロー

最初に、単方向データフローについて詳しく説明します。

2010 年代のモバイルアプリ開発では、Two-way Data Binding（以下、双方向データバインディング）が主流でした。とくに Android は、Data Binding や View Binding が公式から出ていたため、その傾向は顕著でした。iOS も、Rx 系を使って擬似的 or 結果的に双方向を実現する手法が多く使われていました。

ですが、2020 年代に入ると SwiftUI/Combine や Jetpack Compose などの導入が進み、Web Frontend の Flux 系に似た単方向データフローが浸透し始めています。

1.2.1 双方向データバインディングとは

単方向データフローと対の概念である双方向データバインディングは、ある値がユーザー入力と連動し、また通信結果とも連動している状態のことです。値は ViewModel に置かれることが多いです。



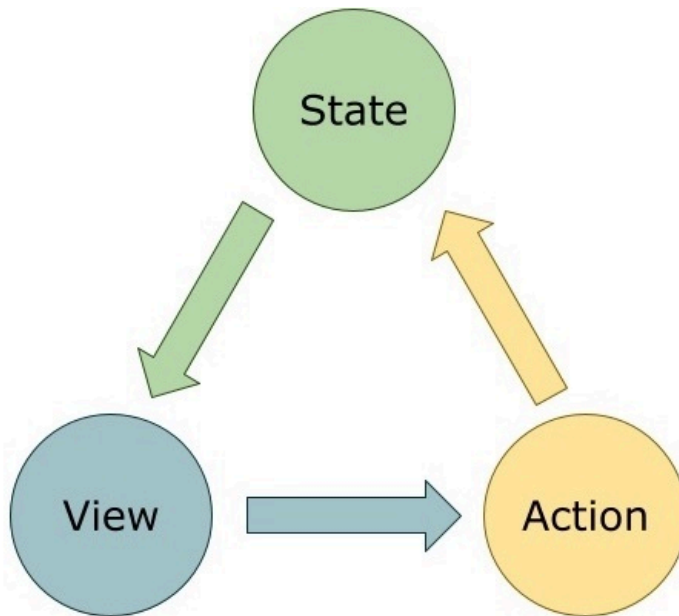
▲図 1.1 双方向データバインディングのデータの流れ

このように各境界が双方向に繋がっており、UseCase の変更が View に、View の変更が UseCase に伝播しやすく、責務を分割しながらコードを整理できるのが特徴です。

見かけのコード量は少なくできますが、各境界でのデータ型の変換や、View のパーツごとにバインディングするための実装が必要で、全体のコード量は減りづらい性質があります。

1.2.2 単方向データフローとは

単方向データフローは、ユーザーが View に与えた入力を Action として受け取り、値は直接更新しません。Action はロジックを経て State の値を更新し、State が View に描画されるというサイクルの関係です。



▲図 1.2 単方向データフローのデータの流れ

単方向データフローでは、データは常に同じ方向へ流れます。逆流は禁止されています。Action は View からの命令、State は Action の結果、そして View は State の結果です。

1.2.3 単方向にするメリットとは

一見すると、双方向データバインディングのほうが階層化されており理解しやすそうです。しかし、単方向化には色々とメリットがあります。

1. 単一化/カプセル化…Mutable な State を 1 箇所にまとめられる。双方向の場合、最新の状態をどこに持たせるかが曖昧で、散らばりやすい
2. 共有…1 つの State を複数の子 View で使いまわしやすい。双方向の場合、状態の共有は ViewModel の共有や View 間のデータ渡しでやりくりする傾向があり、実装が複雑化する
3. 分離…図 1.1 より図 1.2 のほうがノードに接する矢印の数が少なく、切り出しや拡張が容易で、デバッグやテストをしやすく、1 つの変化による副作用も少なく抑えられる

もちろんメリットだけでなく、単方向化のデメリットも存在します。

1. 単一化/カプセル化…Mutable な State が 1 箇所なので、適用できるデザインパターンが限られる
2. 共有…View のライフサイクルに合わせて必要ない State の監視を止めなければ、メモリが枯渇したりランタイムエラーを起こすことがある
3. 分離…View のコード量や階層が増え、パーツごとに描画用の Variable と入力用の Listener が必要

MVVM や双方向な MVC に慣れた人は、単方向の思想に慣れないかもしれません。プロジェクトごとに実装スタイルの差が出やすいのも特徴です。

1.2.4 Android アプリの変化

Android アプリの場合、双方向が主流だった時代は BindingAdapter がよく使われ、推奨アーキテクチャが MVVM なので、双方向データバインディングは暗黙的に推奨されていました。

これはアプリ規模が大きくなるにつれ、アプリ独自のバインディングが多数実装されることを意味しました。やりすぎるとブラックボックス化が起きたり、Android 初心者からは学習ハードルが高いと感じられていた部分です。

また、接続がアノテーション任せな上に双方向なので、デバッグやバグ調査がしづらい問題もありました。

2021 年 7 月、待望の Jetpack Compose が登場し、単方向データフローが公式推奨されました。UI と State が分離され、先述の複雑性からある程度解放されました。

また、デメリットの 1 つであるライフサイクル問題も、`remember` 宣言子を使い State の生存期間を View ライフサイクルに合わせることで回避できます。

1.2.5 iOS アプリの変化

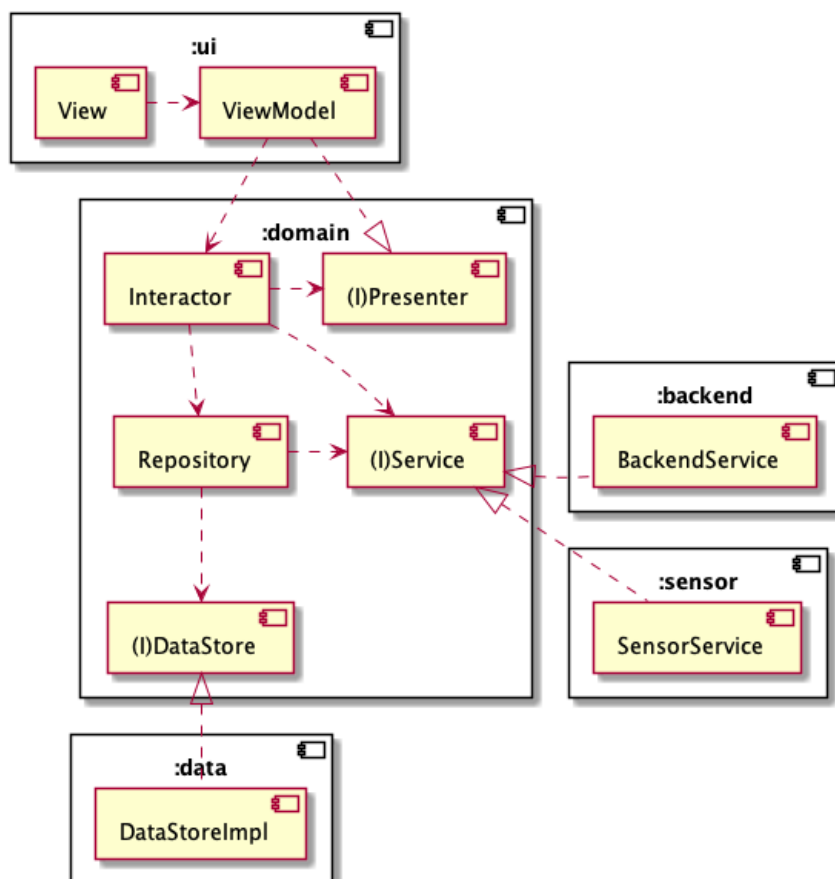
iOS アプリは、MVC や MVC+VM のようなアーキテクチャがよく使われてきました。

しかし、Rx 系をうまく使わないとあっという間に Controller が肥大化したり、外部のフレームワークに頼らざるを得ないのが積年の問題でした。一部を Router に切り出した、VIPER を導入したりなどの模索が続きました。

2019 年、Swift UI/Combine が登場し、単方向データフローの公式サポートが始まりました。非公式ですが TCA も登場し、着々と置き換えが進んでいます。

1.3 クリーンアーキテクチャーと単方向データフロー

続いて、クリーンアーキテクチャーと単方向データフローの親和性に関する考察です。
次の図は、私たちのチームで当初使っていたクリーンアーキテクチャーの設計図です。



▲ 図 1.3 クリーンアーキテクチャーの依存関係（白矢印は非依存の継承または Subscribe）

domain から他へインターフェイスを提供し、「この通りに実装して結果をください」（to backend, data, sensor）、「Presenter 型のオブジェクトを作ってくれば結果を代入します」（to UI）とし、domain から他への依存を無くするのがクリーンアーキテクチャーの特徴です。

この図だと、データは ViewModel から Interactor、Presenter へと流れ、そして ViewModel に帰って来るので、その部分はキッチリ単方向データフローを守っています。

しかし、ViewModel に置かれた状態は、View と domain の両方から参照でき、更新も可能です。人によっては Interactor や Repository に状態を持たせるかもしれません。

つまり、中途半端に単方向も双方向も取り入れているので、プロジェクトが大きくなればデータの流れが複雑化し、保守が困難になるリスクを抱えています。

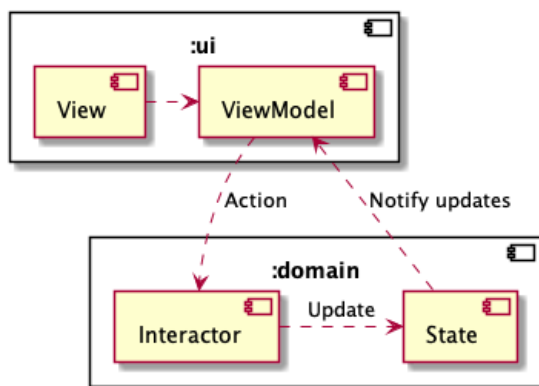
冒頭で述べたとおり、私たちがすべき改善は、どちらかに舵を回しきることだと思っています。単方向データフローがより強固になるよう、データの流れをわかりやすくします。

1.3.1 Presenter を State に置き換え

Presenter を採用していると、最新の状態をどこに置くかが曖昧になります。

また、Action から View に戻るまでのルートを自由に決められるので、私の参画している案件でも Application や AppDelegate が Presenter を継承して、データの行き先がよくわからなくなっているロジックが多少ありました。

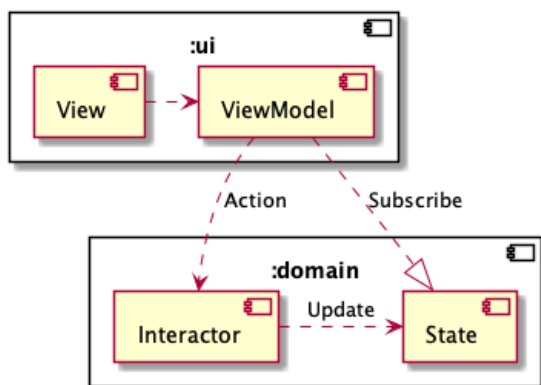
その Presenter を無くして、UI から domain へ向かう矢印を Action と定義し、domain から UI には State の更新を通知するように変えます。



▲ 図 1.4 単方向データフローを適用した UI と domain 間の依存関係

図 1.2 っぽくなりました。ところが、このままではクリーンアーキテクチャーに反しています。Notify updates のところで、domain が UI に依存しているからです。

これは、UI 側から State を Subscribe することで回避可能です。

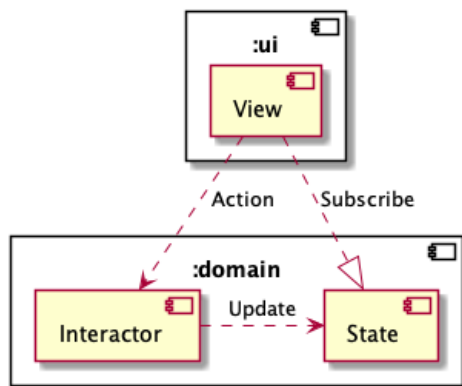


▲図 1.5 単方向データフローを適用し DIP を守った UI と domain 間の依存関係

さて、データの流れは View と ViewModel の間にもあります。

ViewModel から View へは、依存こそありませんが、データは流れます。つまり、View と ViewModel 間のデータの流れはまだ双方向です。このままでは、ViewModel のテストがしづらいです。

1.3.2 結局 ViewModel は必要なのか



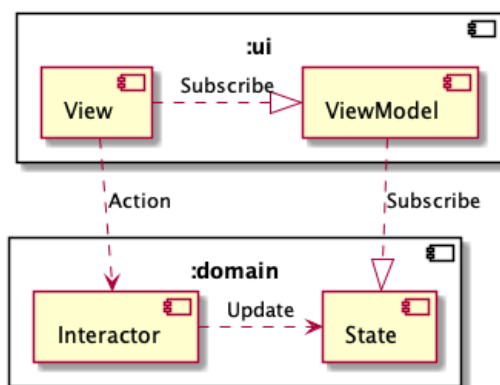
▲図 1.6 ViewModel を廃止した UI と domain 間の依存関係

思いきって ViewModel を取ってみました。すると、View に以下の責務が集中します。

1. 画面の表示
2. State の監視
3. State から画面パーツ向けの型変換
4. 画面の更新
5. 画面遷移

これなら、2 と 3 を ViewModel として分けての方がまだ読みやすいでしょうし、テストのしづらさに至っては悪くなっています。

では、View からのイベント発火で直接 Interactor を呼び、ViewModel は上記 2 と 3 に専念させればどうでしょうか。



▲図 1.7 View から直接 Action を発火する場合の UI と domain 間の依存関係

スッキリしました。

この場合の ViewModel は、Subscribe したデータを View 向けに変換して View に送り出す役割ですが、iOS の TCA では ViewStore、CyberAgent Advanced Technology Studio が提供する Unio では ViewStream と呼ばれています。

ViewModel という呼び方が適切かどうかは諸説あると思いますが、コンバーターとしての役割は View から切り離れたほうがよいです。

1.4 Kotlin Coroutines Flow

ここまでは OS を限定せず記述しましたが、ここからは実際に Android アプリで Flow を使う例を紹介します。

Flow とは、Kotlin Coroutines の非同期処理用ライブラリです。Rx や Promise に似た記述ができ、コールドストリームであることが特徴です。

1.4.1 コールドストリーム

Subscribe されたらはいじめて動きだす、Observable なストリームです。ストリームとは、データを連続して送り出す型を言います。

上司が来たらはいじめて働き出すぐうたら社員をイメージしてください。上司がいる間は、状況の変化をちゃんと逐次報告します。1 人の上司にのみ報告するというのも特徴です。そして、上司がいなくなったらすぐに自分から働くのをやめます。

社員だとうるさなと思うコールドな働き方ですが、プログラムとして、必要ないときに働かないのは実は強力な利点なのです。必要なときだけリソースを食い、不要になったら開放してくれるからです。

しかし、1 人の上司にしか報告しないという特徴は、Observer が 2 つ登録されると、2 つのコールドストリームが必要なことを意味します。これはメモリ効率を考えればまいちなところでは。

1.4.2 ホットストリーム

反対の型がホットストリームです。こちら Observable ですが、Subscriber がいなくても値を発行し、データを送り出します。Publisher と呼ばれることが多くあります。

こちらは上司が来る前から働ける頑張り屋さんです。上司がいる間、状況の変化を逐次報告するのはコールドと同じですが、複数の上司がいても同じ報告を 1 人で請け負います。そして、上司が止めてくれるまで、上司がいなくても働き続けます。

ちゃんと止めてあげないと必要ないときも働き続けてしまうのですが、Observer がいくつ登録されても、使うリソースが変わらないのは利点です。

これを応用すると、1 つのコールドストリームを受信し、複数の Subscriber に送信するという中継地点の役割も担えます。

1.4.3 Kotlin のストリーム事情

元々 Kotlin には、Channel というホットストリームが存在していました。しかし、suspend の非同期処理をシーケンシャルに繋げたい場合、コールドストリームのほうが望ましく、それは遅れて登場した Flow を待つ必要がありました。

実際に Flow が登場すると、非同期処理を直感的に実装でき、安全で習得しやすく使いやすいからと好評で、移行が進みはじめました。

1.4.4 Flow の基本的な使い方

Android アプリで、100ms 毎に 0 から 100 までカウントする処理を、Flow を使って双方向データバインディングで実装します。

CounterUseCase.kt

```
class CounterUseCase {
    suspend fun countStream(): Flow<Int> = flow {
        repeat(100) { count ->
            delay(100)
            emit(count) // 0 1 2 3 4 ... 99
        }
    }
}
```

図 1.7 の Interactor が、無加工のデータを非同期に送ります。

CounterViewModel.kt

```
class CounterViewModel: ViewModel() {
    val showing = MutableLiveData<String>()

    fun showCountEvenNumbersSquared() {
        viewModelScope.launch(Dispatchers.Main) {
            useCase.countStream()
                .drop(1)
                .filter { it % 2 == 0 }
                .map { (it * it).toString() }
                .take(5)
                .collect { count.value = it } // 4 16 36 64 100
        }
    }
}
```


第 1 章 Flow でクリーンアーキテクチャーを最適化する

ViewModel では、それを表示向けに加工します。

Presenter の代わりに Flow の Observer がおり、データが流れてきたら `drop` や `take` など Flow のさまざまなオペレーターを使い、加工や除外を行います。

MainActivity.kt

```
counterViewModel.showCountEvenNumbersSquared()
```

activity_main.xml

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{counterViewModel.showing}" />
```

View は、計算実行と表示を担当します。

データの流れは、図 1.1 と同等になります。`showing` が最新の値を持つ変数で、それをどこからでも更新できるのが不安材料です。

では、単方向データフローにするとどう変わるでしょうか。

CounterState.kt

```
sealed class CounterState {
    data class Init(val count: Int = 0) : CounterState()
    data class Success(val count: Int) : CounterState()
    data class Error(val exception: Exception) : CounterState()
}
```

図 1.2 に適合するため、State クラスを作成します。

ただのカウンターに `Success` や `Error` を持たせるのは若干大げさですが、何かの処理を行い結果を返す場合の基本パターンです。

CounterUseCase.kt

```
class CounterUseCase {
    private val mutableState = MutableStateFlow(CounterState.Init()
        as CounterState)
    val state: StateFlow<CounterState> = mutableState
    suspend fun countStream() {
        repeat(100) { count ->
            delay(100)
            mutableState.emit(count) // 0 1 2 3 4 ... 99
        }
    }
}
```

Interactor が、無加工のデータを非同期に送ります。

先ほどと違うのは、Flow インスタンスを State クラス型で外部へ公開している点です。

StateFlow は、Flow を継承した状態管理用のホットストリームな Flow で、LiveData に似たものです。詳しくは後述しますが、State を導入する上でもっとも容易な手段なので採用しています。

データ操作は MutableStateFlow でないと行えませんが、データ更新をどこからでも行えるのはリスクのある設計なので、MutableStateFlow 型は非公開にします。そのため、CounterUseCase のみが State の更新が可能です。

なお、countStream() が直接 StateFlow を返すようにしていない理由は、後述の SharedFlow と StateFlow セクションの注意点で説明します。

CounterViewModel.kt

```
class CounterViewModel: ViewModel(
    useCase: CounterUseCase
) {
    private val mutableShowing = MutableStateFlow(String)
    val showing: StateFlow<String> = mutableShowing

    init {
        useCase.state
            .drop(1)
            .filter { it is Success && it.list % 2 == 0 }
            .map { it is Success && (it * it).toString() }
            .take(5)
            .onEach { new ->
                (new as? Success)?.count?.let {
                    mutableShowing.value = it // 4 16 36 64 100
                }
            }
    }
}
```

```
        .launchIn(viewModelScope)
    }

    fun showCountEvenNumbersSquared() {
        viewModelScope.launch(Dispatchers.Main) {
            useCase.countStream()
        }
    }
}
```

ViewModel が、それを表示向けに加工します。

先ほどと違うのは、ここでも `StateFlow` を外部へ公開している点です。今まで説明した依存関係に従い、公開先は View です。

MainActivity.kt

```
counterViewModel.showCountEvenNumbersSquared()
```

activity_main.xml

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{counterViewModel.showing}" />
```

View は、計算実行と表示を担当します。

データの流れは、図 1.7 と同等になります。`state` や `showing` は、適切なコンポーネントへのみ更新を許しています。

サンプルコードに出てきた `ViewModelScope` と `Dispatcher` は、Android で Flow を使う上では理解を欠かせないキーワードですが、本書では省略します。

1.4.5 SharedFlow と StateFlow

さて、先ほど出てきた `StateFlow`、そして異なる性質を持つ `SharedFlow` の 2 つについて説明します。

Flow での値の更新は、上記 UseCase の `flow { ... }` ラムダ式中でしかできません。他の場所からは値を更新できず、`.value` のように値の参照もできません。

Subscribe している数だけ `flow { ... }` ラムダ式が呼ばれてしまうのも特徴です。

それでは状態保持や処理リソースの節約には向いてないからと、ホットストリームな Flow として登場したのが、ここで紹介する SharedFlow と StateFlow です。

SharedFlow とは

複数箇所での Subscribe でデータや状態を共有できる Flow で、処理リソースの節約に向いています。

単なる Flow と違う点は以下です。

```
sharedFlow.onEach {  
    println("1")  
}.launchIn(scope)  
  
sharedFlow.onEach {  
    println("2")  
}.launchIn(scope)
```

- このように複数 Subscribe していても flow { ... } ラムダ式側は 1 回しか呼ばれない
- 処理開始/Subscribe 終了のタイミングを選択できるが、これを適切に指定しないと Subscribe され続ける
 - LiveData に変換し、observe 引数に LifecycleOwner を設定すれば、表示中だけの Subscribe も可能
- 色々高機能
 - replay: Subscribe した瞬間、過去の n 回の値を受信する
 - buffer: 複数 Subscribe かつ処理に時間がかかるとき、1 回目に行われた処理をバッファリングして、2 回目以降を早くしてくれる

1 つだけの Flow インスタンスをすべての場所で参照し、監視は必要な間だけ動作させたり、永続的に監視しつつ replay で最後に発行された 10 個を常に監視するなどのトリックが可能です。

StateFlow とは

状態保持に特化した SharedFlow です。LiveData に似ていますが、LiveData は Android、SharedFlow は Kotlin のフレームワークです。とはいえ、LiveData の後継機能として使うこともできます。

- 初期値が必須
- 現在の状態を .value で受け取れる

- MutableStateFlow を使えば、.value への代入も可能
 - その際 Coroutines Scope は不要
- launchIn で直近の値を 1 つ受信する
- 同じ値の代入は受信しない
- wait など挟まず連続して値が変更されたとき、最後の値しか受信しない
 - つまり状態が保持されないと状態変化とみなされない

sharedFlow では、View を開いたタイミングで Flow がサーバー通信などの処理中なら、直近の値をどう表示するのかで迷います。しかし、stateFlow では.value に直近の値がキャッシュされているので、迷わずに済みます。

初期化方法

初期化は、MutableSharedFlow、MutableStateFlow を使って行うか、shareIn、stateIn を使って Flow から変換します。shareIn は sharedFlow インスタンスを、stateIn は stateFlow インスタンスを返します。

注意点

関数の戻り値で shareIn や stateIn をしてはなりません。そうすると、関数の呼び出しごとに新しい SharedFlow または StateFlow が作成され、リソースのムダ使いになります。

また、ユーザー認証など 1 つの入力に対して 1 つの処理結果をシーケンシャルに返す Flow は、ホットストリーム化して複数の Subscriber がいると誤動作する可能性があります。shareIn や stateIn で安易に共有してはならないパターンです。

開始/終了タイミングの指定

shareIn や stateIn は Flow をホットストリーム化するので、開始/終了タイミングを指定できます。SharingStarted オプションで、

1. shareIn や stateIn の際に開始して永続的に有効な Eagerly
2. Subscribe が行われた際に開始して永続的に有効な Lazily
3. Subscribe が行われた際に開始して Subscribe されている間だけ有効な WhileSubscribed

を選択できます。

結局どれがよいのか

Flow、SharedFlow、StateFlow という選択肢があり、SharedFlow/StateFlow の場合は開始/終了タイミングも選択できます。ですが、どれを選択すべきかは、場合によって異なります。

大事なのは、要件に応じたものを適切に使い分けることです。

1. Subscribe 場所の結果に狂いが生じないこと
2. リソースのムダ使いにならないこと

を念頭に置いて判断しましょう。

1.4.6 Flow と LiveData、Rx との比較

Android アプリでは、LiveData や RxJava から Flow への置き換えが少しずつ進んでいます。ですが、Flow の何が良いのかわからないまま周りを気にして使っている方や、新しいからなんとなく使っている方も少なくないと思います。

そのような方のために、従来手法と比較し、Flow を使うメリットを説明します。

従来手法の問題点

Rx 系を使う上で避けては通れない問題が、OS のライフサイクルへの適合やオーバーヘッド対策を盛り込むこと、あるいはそれらの考慮抜けによるバグです。

LiveData は Android Jetpack の一部なので、Android のライフサイクルやメモリ/キャッシュとの親和性は抜群です。初心者のぶっつけ実装でもその類のバグが起きにくいことは、LiveData の優秀さの証明です。もちろん、LifecycleOwner の指定間違いなど致命的なものもあります。

一方、次々と起こる状態変化の `subscribe` や、Model → ViewModel のデータ変換部分に注目すると、そこは複雑化やバグの温床を抱えたままです。

1.4.7 Flow 移行のメリット

1. 構造化された並行性

Flow にあって LiveData にない主な 1 つは、`map` の再計算やデータ変換を `flowOn` により簡単に他スレッドへ投げ、結果だけを UI 側で受け取れることです。

コールドストリームなので、次々と状態変化が起きてもムダなくスレッドを使い、破棄もしてくれます。

2. さまざまな演算子

map、filter、onEach、reduce など、LiveData にはない多くの演算子で効率的なデータ変換を標準サポートしてくれます。

3. テスタビリティ

テストのしやすさも Flow に軍配が上がります。

LiveData + Coroutines のテストはピタゴラスイッチのようなもので、他のテストの干渉を受けないよう上手く作らないと Fail が起きたり、observe のチェーンになってしまいます。

Flow では flowOn する際の dispatcher に TestCoroutineDispatcher を使い、runBlockingTest で走らせると、必ず決まった順序でテストが可能で、無用意な影響を排除できます。

その順序をチェックする collectIndexed など、便利なオペレーターも揃っています。一部だけモックを DI することも容易です。

まとめ

LiveData から Flow に移行すると、ムダのない非同期処理が書け、いろんな演算子も使え、テストも捗ります。

ただし、Flow は Android ではなく Kotlin の機能なので、Android で使う場合 flowWithLifecycle などの導入を忘れないようにしましょう。

そして、Flow が LiveData や Rx より必ず便利とは限りません。作ろうとしているものが何か次第です。上記のメリットを生かせると判断すれば完全 Flow で、確信がなければ UI 部分は LiveData、ロジック部分は Flow を使うのが無難でしょう。

Android の Flow には asLiveData という変換オペレーターがありますし、反対に LiveData も asFlow を持っているので、公式が共存を推奨していると言ってもよいでしょう。

1.5 Kotlin Coroutines Flow を Swift で observe する

ここまで Flow について説明しましたが、たとえば UI 以外を Kotlin Multiplatform Mobile (以下、KMM) で実装し、UI は Swift で実装する iOS アプリの場合、**Kotlin の Flow は Swift でも受け取れるのか**という疑問が起きます。

結論としては受け取れますし、Flow の強みである各種オペレーターも、そのままとはいきませんが Swift の事情に合わせる形で容易に利用できます。

では、それを確かめます。

1.5.1 環境構築

- 必須要件：Android Studio 4.2 以上 / Xcode 11.3 以上 / macOS Mojave 10.14.4 以上
- 私の環境：Android Studio Arctic Fox 2020.3.1 / Xcode 13.1 (13A1030d) / macOS Big Sur 11.5.2

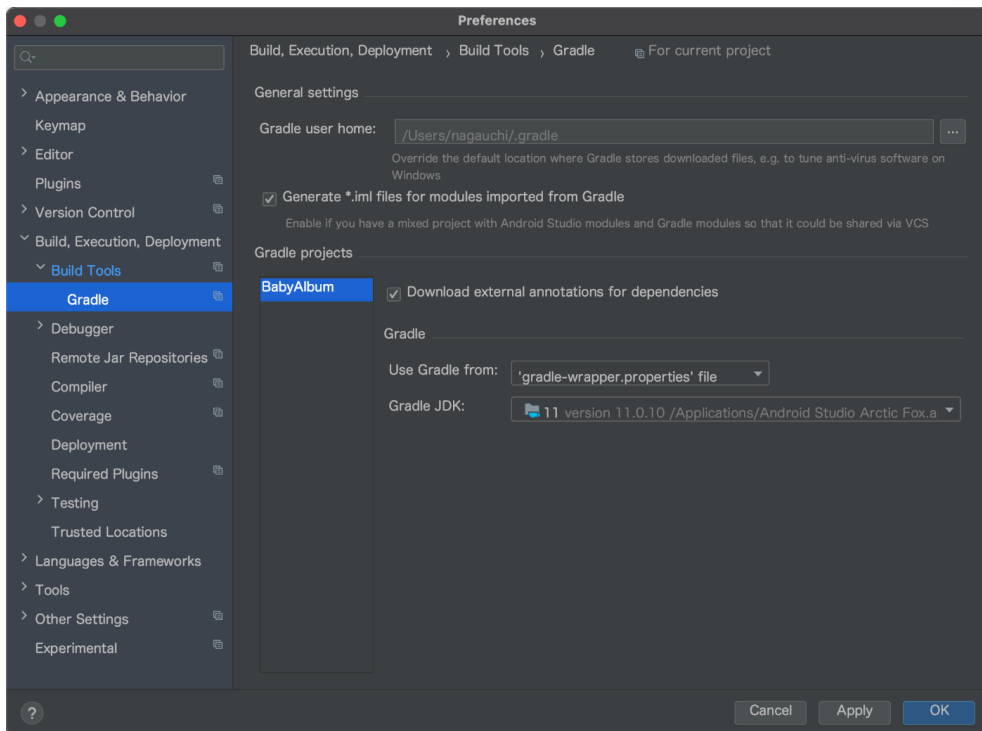
せっかくなので、KMM アプリを 1 から作る手順を載せます。

1. Android Studio で、Android Studio → Preferences → Plugins で、Kotlin Multiplatform Mobile をインストール
2. Android Studio で、File → New → New Project、KMM Application を選択
3. パッケージ名など入れる
4. Add sample tests for Shared module にチェックを入れ（本記事ではテストコード書かないですが今後のため）、iOS framework distribution は Regular framework とする
5. フォルダー階層を Android から Project に変える
6. iOS のデバッグ設定を Edit Configurations から行う
7. Execution Target を好みの Simulator に指定
8. Simulator が出てこない場合は Xcode を開き Window → Devices and Simulators → Simulators タブの＋ボタンで追加

これで準備完了です。

ちなみに Gradle Setting で JDK 1.8 だとビルド時に `Android Gradle plugin requires Java 11 to run. You are currently using Java 1.8.` というエラーになるので、Android Studio → Preferences → Build, Execution, Deployment → Build Tools → Gradle で Gradle JDK を 11 以上に設定します。

第 1 章 Flow でクリーンアーキテクチャーを最適化する



▲図 1.8 Gradle JDK の設定

最後に、`shared/build.gradle.kts` に `common` で使う依存関係を追加します。

```
val commonMain by getting {
    dependencies {
        implementation("org.jetbrains.kotlin:kotlin-stdlib:1.6.0")
        implementation(
            "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.5.2-native-mt"){
            version {
                strictly("1.5.2-native-mt")
            }
        }
    }
}
```

執筆時点では `native-mt` を付与しないと、iOS からの Coroutines 呼び出しで強制終了が発生します。

後々依存関係を追加する際にバージョンの不整合を起こさないため、`strictly` も設定します。

1.5.2 実装

この章で作るのは、赤ちゃんに見せるアルバムアプリです。

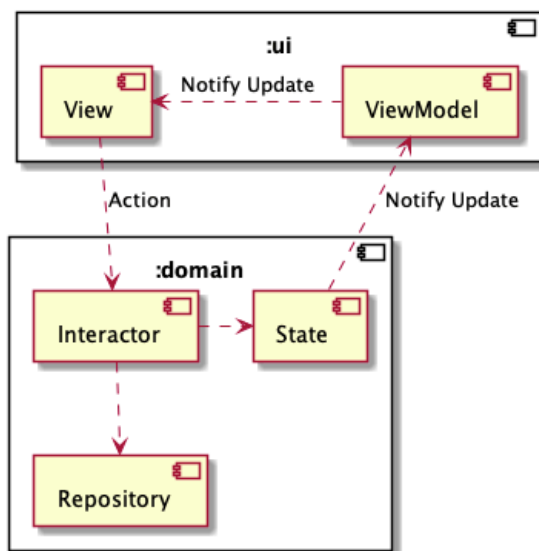
1 歳の娘が、自分やいとこの写真をスマホで見るのが大好きなのです。しかし、OS 標準のフォトアプリで写真を見せていると、手を伸ばして触れ、いろんな操作をしてしまいます。

たとえば写真を削除した上でゴミ箱を空にしたり、共有メニューから画像を誰かに送信したり、なんてことがあると困るのです。

よって、アルバムアプリの仕様は次のようにします。

1. 15 秒ごとに異なる画像をランダムに表示する
2. ユーザー入力以外のホームボタン以外一切受け付けない

設計は図 1.9 の通りとします。Repository より先はありません。



▲ 図 1.9 赤ちゃん用アルバムアプリの設計

サーバーやデータ保存まで作り込むと、本書の記事が長くなり伝えたい本質を見失うので、以下のようにとてもシンプルなプログラム仕様にします。

1. Interactor と Repository は、15 秒ごとに [0-4] の数字をランダムで State に反映

第 1 章 Flow でクリーンアーキテクチャーを最適化する

2. ViewModel は、State を Subscribe し、数字に応じた画像名を View 向けの `@Published` 変数に反映
3. View は、`@Published` 変数に SwiftUI でバインドし、変数の画像名の画像をアセットから見つけて表示

その通りコードを書いていきます。

Kotlin のソースコードは、すべて `shared` の `commonMain` 階層下です。

SwiftStateFlow.kt

まずは、本記事のメインテーマとなる `StateFlow` を Swift で使うための `SwiftStateFlow` を実装します。

```
package com.vitantonio.nagauzzi.babyalbum

import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.Job
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.launchIn
import kotlinx.coroutines.flow.onEach

class SwiftStateFlow<T>(private val kotlinStateFlow: StateFlow<T>) : Flow<T> {
    by kotlinStateFlow {
        val value = kotlinStateFlow.value
        var job: Job? = null

        fun observe(continuation: ((T) -> Unit)) {
            kotlinStateFlow.onEach {
                continuation(it)
            }.launchIn(
                CoroutineScope(Dispatchers.Main + Job()).also { job = it }
            )
        }

        fun close() {
            job?.cancel()
            job = null
        }
    }
}
```

呼び元は `continuation` クロージャの引数でデータを受け取ります。

Flow ではなく `StateFlow` を選択したのは、前回と同じ値を連続発行しないように `state.value` を `Interactor` 内で参照するからです。

呼び元が破棄されるときに `close` を呼ぶと処理を中止できます。後述の `AlbumViewModel` の `deinit` で呼びます。

PublishNumber.kt (Interactor)

次に、KMM 側の Interactor/Repository/State を実装します。

```
package com.vitantonio.nagauzzi.babyalbum.domain.interactor

import com.vitantonio.nagauzzi.babyalbum.SwiftStateFlow
import com.vitantonio.nagauzzi.babyalbum.domain.repository.NumberRepository
import com.vitantonio.nagauzzi.babyalbum.domain.state.NumberState
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

class PublishNumber(
    private val repository: NumberRepository
) {
    private val mutableState = MutableStateFlow(NumberState.Init(0) as NumberState)
    private val swiftMutableState = SwiftStateFlow(mutableState)

    val state: StateFlow<NumberState>
        get() = mutableState
    val swiftState: SwiftStateFlow<NumberState>
        get() = swiftMutableState

    fun execute(min: Int, max: Int, times: Int) =
        CoroutineScope(Dispatchers.Default).launch {
            repeat(times) {
                delay(15000)
                mutableState.value = NumberState.Updated(
                    repository.getChangedRandom(min, max,
                        before = state.value.number)
                )
            }
        }
}
```

NumberRepository.kt (Repository)

```
package com.vitantonio.nagauzzi.babyalbum.domain.repository

class NumberRepository {
    fun getChangedRandom(min: Int, max: Int, before: Int): Int {
        val random = (min..max).random()
        return if (random == before) {
            getChangedRandom(min, max, before)
        } else {
            random
        }
    }
}
```

NumberState.kt (State)

```
package com.vitantonio.nagauzzi.babyalbum.domain.state

sealed class NumberState(open val number: Int) {
    data class Init(override val number: Int) : NumberState(number)
    data class Updated(override val number: Int) : NumberState(number)
}
```

この程度なら Interactor/Repository/State に分けるメリットがありませんが、行く行くはサーバーから画像をダウンロードして ByteArray などの型で Interactor に渡し、State は Init/Success/Failure に分かれることを想定しているので、今のうちに分けました。

AlbumViewModel.swift (ViewModel)

さて、ここからは Swift 側です。

KMM のプロジェクトを作った時点で、SwiftUI の View が自動生成されているので、ViewModel を作ります。

```
import shared

class AlbumViewModel: ObservableObject {
    private let photos = (1...5).map { "BabyImage\($0)" }
    private let interactor: PublishNumber

    @Published var photoName: String

    init(interactor: PublishNumber) {
        self.interactor = interactor
        self.photoName = self.photos[0]
        interactor.swiftState.observe { newState in
            if 0...4 ~= newState!.number {
                self.photoName = self.photos[Int(newState!.number)]
            } else {
                fatalError("newNumber isn't supported number")
            }
        }
    }

    deinit {
        self.interactor.swiftState.close()
    }
}
```

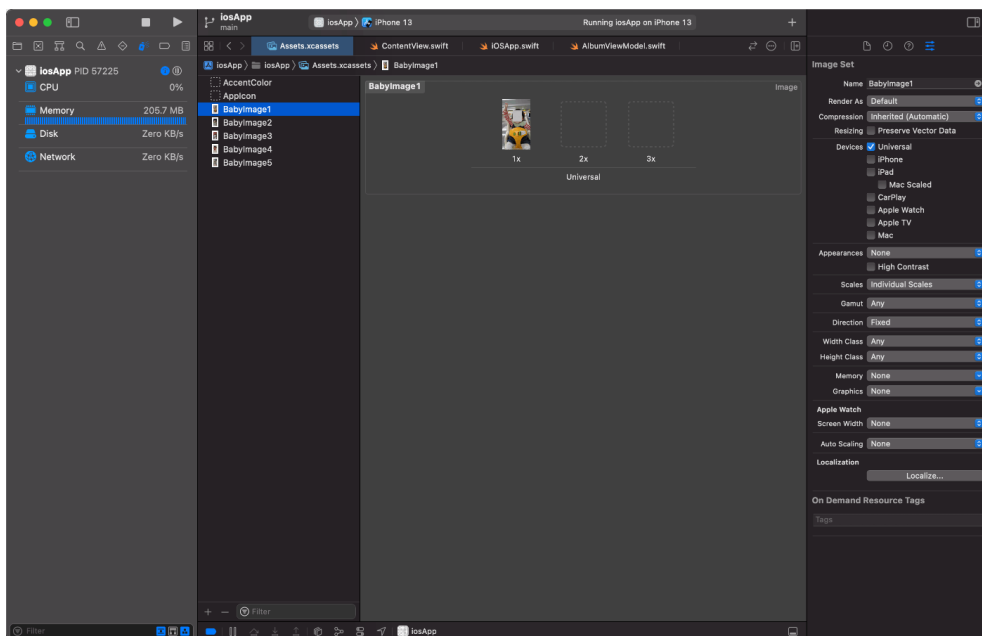
先頭の import shared で、KMM で作った SwiftStateFlow を使えたり PublishNu

mber にアクセスできます。

そして、observe が Flow を監視し、新しい State を受信する部分です。受信した数値を画像名に変えて photoName に入れます。

photoName には @Published が付いており、この変数が View に対する Publisher の役割を果たしてくれます。

photos は配列で画像名を持っていますが、実際の画像は BabyImage1～BabyImage5 を Assets.xcassets に登録済みです。



▲図 1.10 Assets.xcassets

ContentView.swift (ViewModel)

最後に、ContentView をデフォルトから変更します。これが設計図の View にあたります。

```
import shared

struct ContentView: View {
    @ObservedObject var viewModel: AlbumViewModel
```

```
let interactor: PublishNumber

init() {
    self.interactor = PublishNumber(repository: NumberRepository())
    self.viewModel = AlbumViewModel(interactor: interactor)
}

var body: some View {
    ZStack {
        Color.black
            .ignoresSafeArea()
        Image(viewModel.photoName)
            .resizable()
            .aspectRatio(contentMode: .fill)
    }.task {
        self.interactor.execute(min: 0, max: 4, times: 100)
    }
}
```

余談ですが、Repository を PublishNumber に依存させず外から渡す仕組みなので、たとえば KMM 側で OS 共通の通信処理を実装していたがやっぱり Swift 側で Alamofire などを使いたくなった場合、DataSource オブジェクトを Swift 側で用意し、Repository に詰めて KMM 側に渡すことで容易に変更が可能です。

最初から KMM を適用するスコープを自在に決められるのも嬉しいです。

クリーンアーキテクチャーの大きな利点は、こういった差し替えのしやすさです。

1.5.3 プレビューと実行

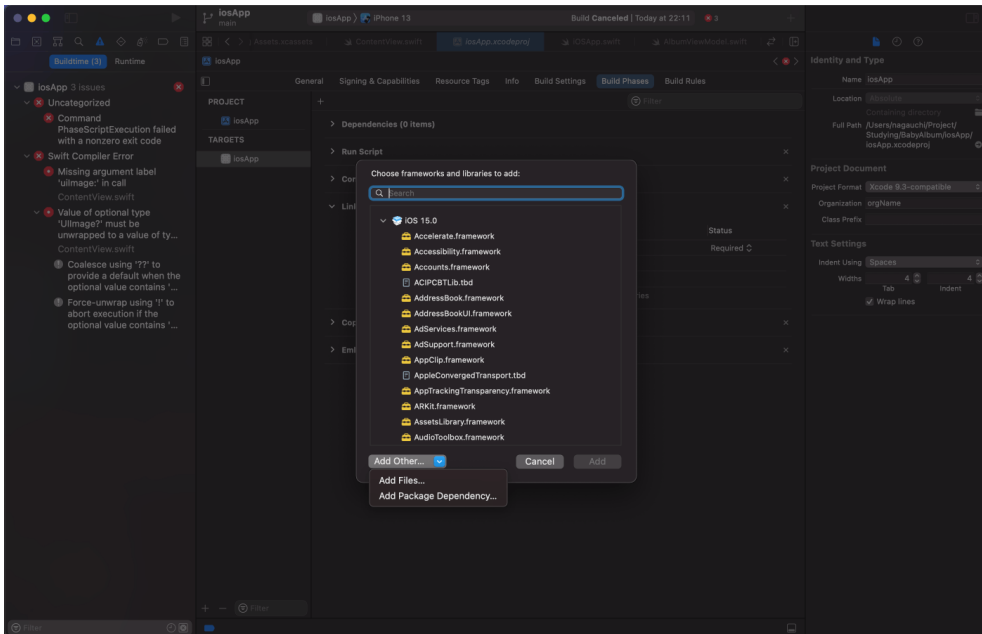
ここで、SwiftUI のプレビューが動いていないことに気づきました。

Android Studio 側で Run するとビルドできますが、Xcode 側では `shared.framework` をインポートしていないので、そのビルドエラーによって SwiftUI のプレビューや Xcode 側でのデバッグ、エラー調査などができないのです。

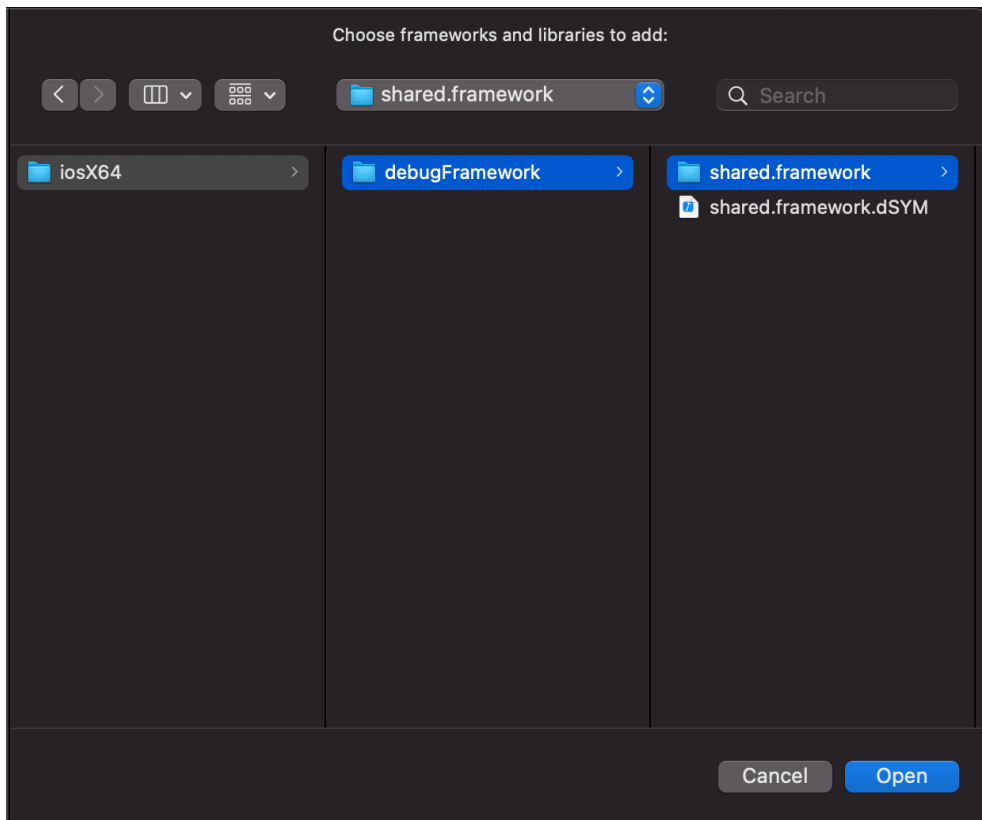
Xcode でそれらができると iOS エンジニアとしては開発効率が上がるので、直します。

iosApp の TARGETS → Build Phases → Link Binary With Libraries で、`./BabyAlbum/shared/build/bin/iosX64/debugFramework/shared.framework` を追加します。

1.5 Kotlin Coroutines Flow を Swift で observe する



▲図 1.11 Framework の追加

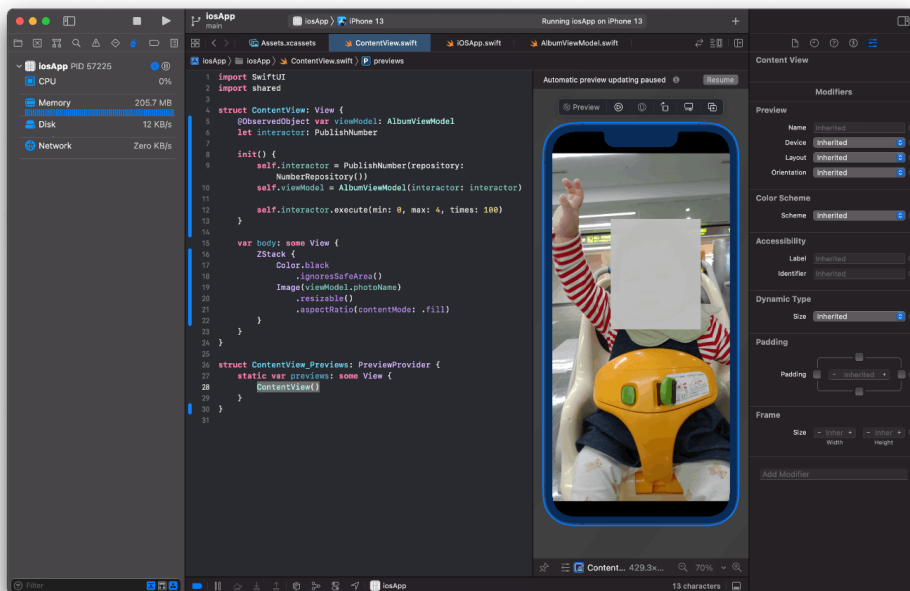


▲図 1.12 shared.framework の選択

ただし、動かす iOS 端末の CPU アーキテクチャーに応じた Framework を取り込む必要があるので、正式なビルドは Android Studio から行うのがよいです。

ここまでで、赤ちゃん用アルバムアプリの仕様を満たしました。

1.5 Kotlin Coroutines Flow を Swift で observe する



▲図 1.13 SwiftUI のプレビュー

SwiftUI のプレビューが動作しており、iOS 端末にインストールして実行すると、15 秒毎に違う画像が表示されます。これは SwiftStateFlow が 15 秒毎の Flow のデータ送出を Swift にうまく伝えているからです。

1.6 Kotlin Coroutines Flow を Swift の Combine に変換する

さて、ここまでで満足かというところ、本当に作りたかったものはこれじゃない感があります。

SwiftStateFlow でも、Flow の強みであるさまざまなオペレーター（map、filter、onEach、reduce など）を使いたいです。

とはいえ、Flow と同じオペレーターを Swift で 1 から実装するのは大変です。shared.h に定義されている Kotlinx_coroutines_core 系の型を使って Flow を操作するのも、根気やリスクが伴いそうです。

また、無理して Flow に合わせるのではなく、Swift なら Swift らしい実装をしたいと思います。

そこで、Swift 標準の非同期フレームワークである **Combine** の出番です。

AlbumViewModel を、以下のように書き直します。

AlbumViewModel.swift (ViewModel)

```
import Combine
import shared

class AlbumViewModel: ObservableObject {
    private let photos = (1...5).map { "BabyImage\($0)" }
    private let interactor: PublishNumber
    private var cancellable: Cancellable?

    @Published var photoName: String

    init(interactor: PublishNumber) {
        self.interactor = interactor
        self.photoName = self.photos[0]
        self.cancellable = NumberStatePublisher(stateFlow: interactor.swiftState)
            .map { newValue in
                self.photos[Int(newValue.number)]
            }
            .assign(to: \.photoName, on: self)
    }

    deinit {
        self.cancellable?.cancel()
    }
}

public struct NumberStatePublisher: Publisher {
    public typealias Output = NumberState
```

```

public typealias Failure = Never

private let stateFlow: SwiftStateFlow<Output>

public init(stateFlow: SwiftStateFlow<Output>) {
    self.stateFlow = stateFlow
}

public func receive<S: Subscriber>(subscriber: S)
    where S.Input == Output, S.Failure == Failure {
    let subscription = NumberStateSubscription(stateFlow: stateFlow,
        subscriber: subscriber)
    subscriber.receive(subscription: subscription)
}

final class NumberStateSubscription<S: Subscriber>: Subscription
    where S.Input == NumberState, S.Failure == Never {
    private let stateFlow: SwiftStateFlow<S.Input>
    private var subscriber: S?

    public init(stateFlow: SwiftStateFlow<S.Input>, subscriber: S) {
        self.stateFlow = stateFlow
        self.subscriber = subscriber

        stateFlow.observe { newValue in
            _ = subscriber.receive(newValue!)
        }
    }

    func cancel() {
        subscriber = nil
        stateFlow.close()
    }

    func request(_ demand: Subscribers.Demand) {}
}

```

Combine の仕組みは省略しますが、必要な Publisher と Subscription を作りました。

重要なのは、`func startSubscribe()` の部分です。受け取った数字を `map` で文字列加工して、直接 `photoName` にバインドしています。

この数行で、ロジックから来たデータを View 向きに加工して再描画まで行っています。Combine のオペレーターを使用できる大きな利点です。

改良前の ViewModel もコード量は少ないのですが、State を表示向けに変換する処理を適宜追加する必要があるため、複雑な実装になると `observe` 部分がじわじわと膨れ上がることが予想されます。

また、Publisher を `@Published` 変数に `assign` すると `Cancellable` オブジェクトが取れるので、その参照を取っておき、ViewModel が解放される際に監視を止めます。改良前は直接 `Interactor` を介して `SwiftState` の `cancel` を呼ぶ必要があり、クリーンアーキテクチャーを意識しているにもかかわらず、責務の境界が曖昧になっていました。

1.7 あとがき

前回のテックブックの記事を書いたとき、クリーンアーキテクチャーの運用経験は 1 年未満で、ちょうど iOS の SwiftUI が発表されたタイミングでした。よって、数年後に事情が変わるであろうことは想像に難しくなく、後日談や続編のような記事を書きたいと当初から思っていました。

今回、技術書典 12 に出典するチャンスを頂き、それを現実のものにできました。本書を書くにあたって、支援してくださった技術書典同好会と株式会社 ACCESS の皆さん、ご協力くださったすべての方々へこの場を借りて感謝を申し上げます。

本書では、Android/iOS のクロスプラットフォームで使われることを想定しています。しかし、Flow や Combine のような非同期フレームワークを持つ環境であれば、言語や OS を問わず柔軟に応用できると感じています。

また、モバイルアプリ界限には引き続きパラダイムシフトが度々起こると予想しているので、各時代に応じた最新のトレンドアーキテクチャーを個人でも当社としても引き続き追究し、発信を続けたいと考えています。

第 2 章

SvelteKit + FastAPI + vercel + heroku でやる気があれば誰でも簡単フルスタックエンジニア

2.1 はじめに

こんにちは。野々山太郎 (@n2-freevas) です。

ACCESS は、Chromium などを用いたブラウザベンダとしての側面を強く持つ一方、ブラウザ以外の開発業務では、Web・アプリ開発にも注力しており、通常の開発業務でも最新フレームワークを全面的に取り入れる前向きな姿勢を持っています。

さて、Web 開発といえば PHP や Ruby などの言語を想像する御仁もおられると思います。Web フレームワークは流行り廃りが激しく、1 つの言語をとっても同種のフレームワークが複数あったりと競合状態で、それぞれに若干の得手不得手がありますが実際使ってみないと違いがよくわからない、というのも正直なところです。そんな中、昨今業界にじわじわ食い込んで来ている 2 つのフレームワークがあります。

それが、**SvelteKit** と **FastAPI** です。

前者がフロントエンドのフレームワーク、後者がバックエンド API サーバーのフレームワークになります。

とりわけ、どちらも簡単に初学者に優しく、短いコードでモノを書け、実業務で運用しても十分な可用性を持ち合わせておる大変ありがたいフレームワークです。

本章では、SvelteKit と FastAPI がどんなものか、実際に Web 開発の経験が浅い読者に **Web 初学者に開発経験を与えたい** という意図を盛り込んだ内容を口伝していく所存です。

また、本章の内容を追うことで、開発環境をすぐさま用意でき、フロントとバックエンド開発をそこそ網羅でき、実際に成果物をネットにあげることができます。そして、それ以降に続く「各々がやりたい開発」へと導けたのなら幸いです。

また、この記事で紹介するデプロイの方法であれば、**成果物の運用はすべて無料でやれる**ことが特徴です。

2.2 対象

実装についての説明はありますが、言語仕様などのフレームワークと直接関係のない部分は触れません。よって、

- HTML / CSS / JS は触ったことがある
 - ここが未経験の方は、「HTML&CSS と Web デザインが 1 冊でちゃんと身につく本 (技術評論社)」*¹を買って、基本を身につけましょう
- Python をすでに知ってる or 知らないで調べつつ触る気概がある

が対象になります。

2.3 想定環境

macOS Big Sur ver:11.4

なお、同じ環境が用意できなくても、この記事で紹介する内容の範疇であれば、Windows だろうが mac だろうが、さほど違いはないと思います。自分が用意したセットをこのマシンで作ったので、参考までに。

また、ブラウザは Google Chrome を使ってる前提で進めています。

*¹ <https://gihyo.jp/book/2022/978-4-297-12510-3>

2.4 前置き・座学など

2.4.1 Svelte とは？



▲図 2.1

Svelte は、React.js や Vue.js などといったメジャータイトルに比べ歴史が浅いですが、結論からいうと、ライバルフレームワークを蹴散らして主流に躍り出るポテンシャルを秘めた、すさまじい開発体験を与えてくれるフロントエンドの開発手段です。

また、Svelte をベースにフレームワーク化を施した **SvelteKit** は、2021 年 3 月に beta 版が公開され、今に至ります^{*2}。

SvelteKit は、実際に商用サイトで使っている企業も続々登場していることもあり、もししたら 5 年後とかには、採用面接などで、「svelte の経験は？」と聞かれる時代になるかもしれません。

特徴として、

- ・ ソースコードは React よりも遥かに短いソースコードで書ける。(1/3 で書けると言われる。)
- ・ svelte 固有の構文が少ない。JS を知っていればだいぶ有利。
- ・ (プラグインなどいるが)SSA、SPA 両対応
- ・ ソースのディレクトリ構造が、そのまま URL の階層構造になるシンプルさ
- ・ 状態管理、State の処理を、かなり楽に書ける
- ・ それでいて、動作が遅いといったことはない。

といった具合でしょうか。

^{*2} <https://svelte.dev/blog/sveltekit-beta>

2.4.2 FastAPI とは？



▲図 2.2

FastAPI は、Python の WebAPI フレームワーク。

2018,19 ごろに登場した若手で、Python でサーバのソースコードがスラスラと書けて、API に欲しい機能や実装が比較的簡単に導入できるのでおすすめです。

昨今の主流である、json でやりとりするタイプの API を実装する場合は、FastAPI の持つデフォルトのサポートが、大いに役立つでしょう。

そもそも、Python という言語はそれ自体が、そこそこ早く、書き心地がよく、日本語のドキュメントが揃っているなので、初心者にとってつけの言語です。

他言語の対抗馬として、「Go(iris や gin)」や「node.js(express)」があります。特に「Go」は、若手の注目言語ということもありめっぽう熱いですが、初心者に優しいドキュメントの数で言うと Python に軍配が上がるので、この記事では FastAPI を選択します。

また、世間の太鼓判では「node.js や Go に匹敵する速さ」という評価であり、僕もそう感じていることが、技術選択の後押しになっています。

2.4.3 vercel とは？

svelte のソースコードをデプロイするときに使います。本当に楽にデプロイできるので盲信して使っています。



▲図 2.3

2.4.4 heroku とは？

FastAPI をデプロイするときに使います。この手のサービスにしては珍しく、DB まで無料で使えるのが大きいです。もちろん制限はありますが。

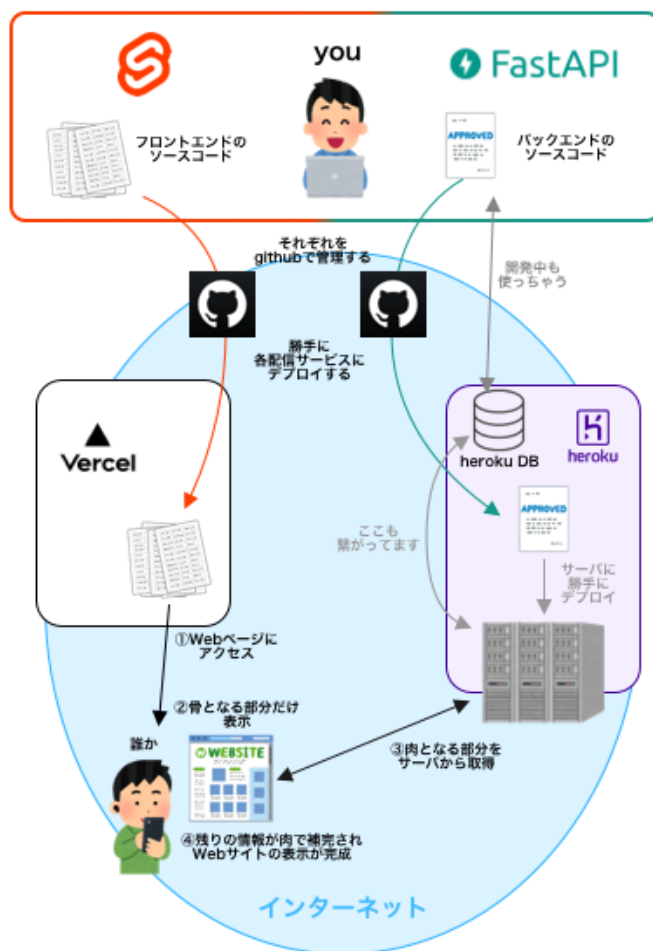
(ただ、もし、あなたのアプリケーションが、1 万人以上が使うような大きいアプリケーションになったら、heroku の弱点が露呈してくるので、大人しく Amazon Web Service を使うことをお勧めします。)



▲図 2.4

2.4.5 アプリケーションの構成

完全に初学者向けの内容ですが、一応、Web アプリケーションの構成について説明を記します。今回のアプリケーションは、王道中の王道である「Restful な」「JSON 形式の」アプリを目指します。



▲図 2.5

上図が、王道中の王道の構成である Restful なアプリケーションの構成図です。Restful

アプリは、**WebAPI** というものが軸になって、いろんな物事を進めていきます。

世の中にあるほとんどのアプリケーションは、デバイス単体で完結しない機能を有していることがほとんどです。(例えば、他の人と交流する機能とか。) それを実装するときには、もちろん通信を行って情報を補完していきます。通信の登場人物は、たいてい **サーバ** と、**フロントエンド (PC・スマホ)**、そしてそれら同士の取り決めである **プロトコル (情報の送信方法)** です。

サーバは「**情報の生成や整理** をして、それを **フロントに送る** 役割」、フロントは「その情報を **人に魅せるための処理** & **ユーザの操作をサーバに送る** 処理」というふうに役割分担しています。

その「情報」というのはどうやって送るでしょうか？ それは大抵、**json** と呼ばれる形式の情報体を、HTTPS 通信で送ります。

API というのは、フロントの命令によってサーバから json を引き出す一連の処理を指し、言い換えれば、フロントエンドがサーバから json を得るために訪問する窓口的な存在なのです。

百聞は一見に如かず。とりあえず API を叩いてみます。世の中に API は星の数ほどありますが、今回は CatAPI^{*3}という高尚な趣味をした API を訪問しましょう。

^{*3} <https://docs.thecatapi.com/>

下図は、CatAPI を叩くと得られる json です。

```
[
  {
    breeds: [],
    id: "e2t",
    url: "https://cdn2.thecatapi.com/images/e2t.jpg",
    width: 500,
    height: 375
  },
  {
    breeds: [],
    id: "MTkwMjc4Mw",
    url: "https://cdn2.thecatapi.com/images/MTkwMjc4Mw.jpg",
    width: 480,
    height: 360
  },
  {
    breeds: [],
    id: "2UxCy6KIh",
    url: "https://cdn2.thecatapi.com/images/2UxCy6KIh.jpg",
    width: 499,
    height: 498
  }
]
```

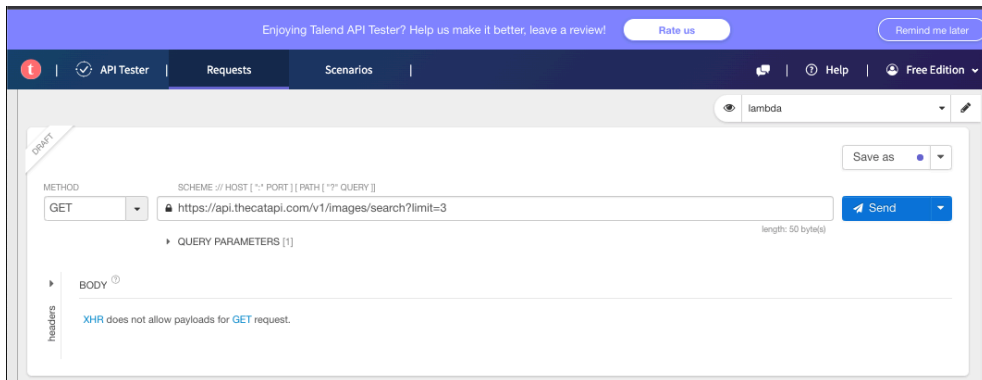
▲図 2.6

この API は特にセキュリティ的なものを用意していない、フリーな API なので、どんな輩でも叩き放題です。

試しに皆さんも、叩いてみましょう。

API を叩く方法も星の数ほどありますが、今回は Talend API Tester^{*4}を使います。リンクの指示に従い、Chrome に拡張機能を入れて、起動して、<https://api.thecatapi.com/v1/images/search?limit=3> と打ち込んで、Send を押しましょう。すると、先程の画像のような json が返ってきます。

^{*4} <https://chrome.google.com/webstore/detail/talend-api-tester-free-ed/aejoelaoggembcahagimdiliamlcdmfm>



▲図 2.7

CatAPI 公式ドキュメント^{*5}を見ると、/search の後に limit や page などと付加情報を与えると、json の返り方が変わってくるみたいです。

この付加情報のルールなどは、当然 **catAPI** の実装者が決めている **API** のルールです。そして、我々はドキュメントなどに書かれている、そのルールにしたがってコマンドを叩くことで、情報を得られるのです。

では、こういう風に作られた **API** というものを、あなた自身が、あなたの思うようにサーバに実装して、それをあなたのアプリ自身が叩いて、そしてその情報を元に、アプリの魅せ方を変える **Web ページ** を組んだらどうなるでしょう。

もうわかりましたね。Web アプリケーションがやっていることは、単純にそういうことなのです。

2.5 フロントエンド開発の準備

本節より、フロントエンド開発を進めていきます。

まずは、homebrew, node.js, python3 を用意してください。以下の記事に従えば、あらゆる準備が完了します。

Qiita: Homebrew のインストール^{*6}

TechAcademy: 手順を分かりやすく解説！ Node.js のインストール方法^{*7}

Python3 インストール (Mac 編)^{*8}

^{*5} <https://docs.thecatapi.com/pagination>

^{*6} <https://qiita.com/zaburo/items/29fe23c1ceb6056109fd>

^{*7} <https://techacademy.jp/magazine/16050>

^{*8} <http://netsu-n.mep.titech.ac.jp/~Kawaguchi/python/install-mac/>

Python3 インストール (Win 編)*⁹

初学者の方にとって、この事前準備は苦しいものになるかもしれません。私からは誠悦ながら、この事前準備のゴールを示しておきます。インストール作業が終わったら、以下のコマンドを打って、以下のように出てくれば OK です。

```
$brew --version
Homebrew 3.3.5 (これ)
Homebrew/homebrew-core (git revision 0eb6718c519; last commit 2021-11-29)
Homebrew/homebrew-cask (git revision bec9ce12ba; last commit 2021-11-29)

$ npm --version
6.14.13 (これ)

$ python3 -V
Python 3.8.12 (これ)
```

2.6 SvelteKit をテンプレートから学ぶ

では、基礎的な API の挙動を楽しめるアプリを作ってみましょう。

僕が用意したテンプレート↓があるので、まずこれで遊びましょう。GitHub からダウンロードしてください。

GitHub: [n2-freevas svelte-sample](https://github.com/n2-freevas/svelte-sample)*¹⁰

もし、GitHub 初心者であれば、以下の記事から、Clone と呼ばれるダウンロードの流れを行ってください。

Qiita: [GitHub～Clone から Push までの流れ](https://qiita.com/ogss34/items/5370df0f7bba6c32723e)*¹¹

2.6.1 とりあえず起動してみる。

svelte-sample 側の、ディレクトリ上で、

```
// 必要なライブラリのインストール
// (予めpackage.json/dependenciesに定義しておいたライブラリを自動で全部installする。)
// (SvelteKitなども含まれている。)

$ npm install
```

*⁹ <http://netsu-n.mep.titech.ac.jp/~Kawaguchi/python/install-win/>

*¹⁰ <https://github.com/n2-freevas/svelte-sample>

*¹¹ <https://qiita.com/ogss34/items/5370df0f7bba6c32723e>

```
// SvelteKitの起動  
// (package.json/scripts/dev に書いてあるコマンドを打ったのと同義)  
  
$ npm run dev
```

(this will be run only when your dependencies or config have changed)

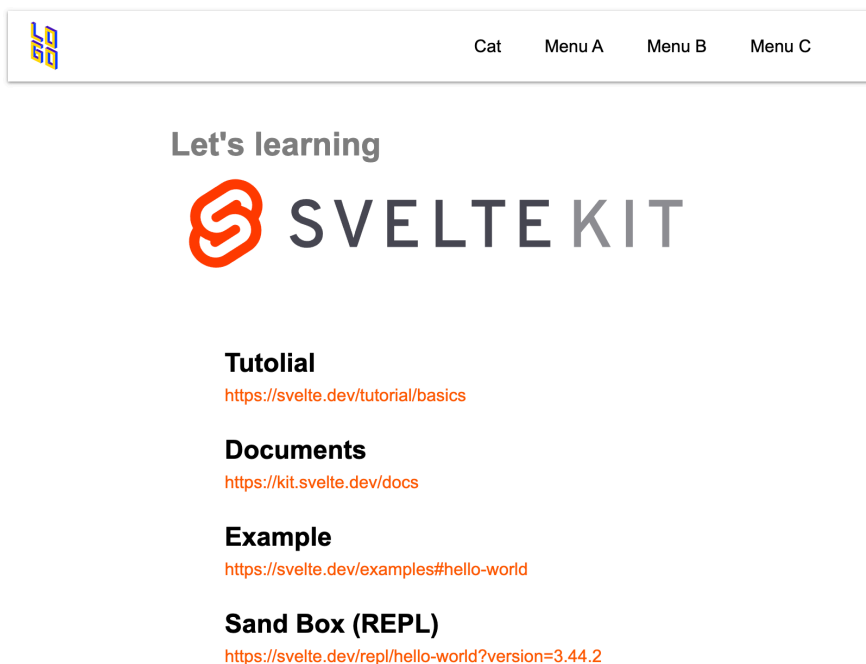
SvelteKit v1.0.0-next.201

local: **http://localhost:3000**
network: not exposed

Use `--host` to expose server to other devices on this network

▲図 2.8

すると、上記のような回線が開くので、`http://localhost:3000` をブラウザでアクセス



▲図 2.9

出てきた画面は、私があらかじめ実装しておいた Web ページになります。同じように出て来れば、起動成功です。

2.6.2 Svelte の主要機能を理解する

ファイルの書き方

.svelte 拡張子が付いたソースが、svelte のソースコードです。下図は、/src/routes/menu-a/index.svelte のソースです。

```
//javascript実装部分
<script>

</script>
```

```
//それ以外のマークダウンの部分
<h1>MENU A</h1>

//CSS実装部分
<style>
  h1{
    line-height: 70vh;
    text-align: center;
  }
</style>
```

基本的に、svelte は、

- `script`タグで囲まれた、javascript 実装部分
- `style`タグで囲まれた、CSS 実装部分
- それ以外のマークダウンの部分

の 3 つで構成されています。(ちなみに、この 3 つは順不同でよかったはずです。)

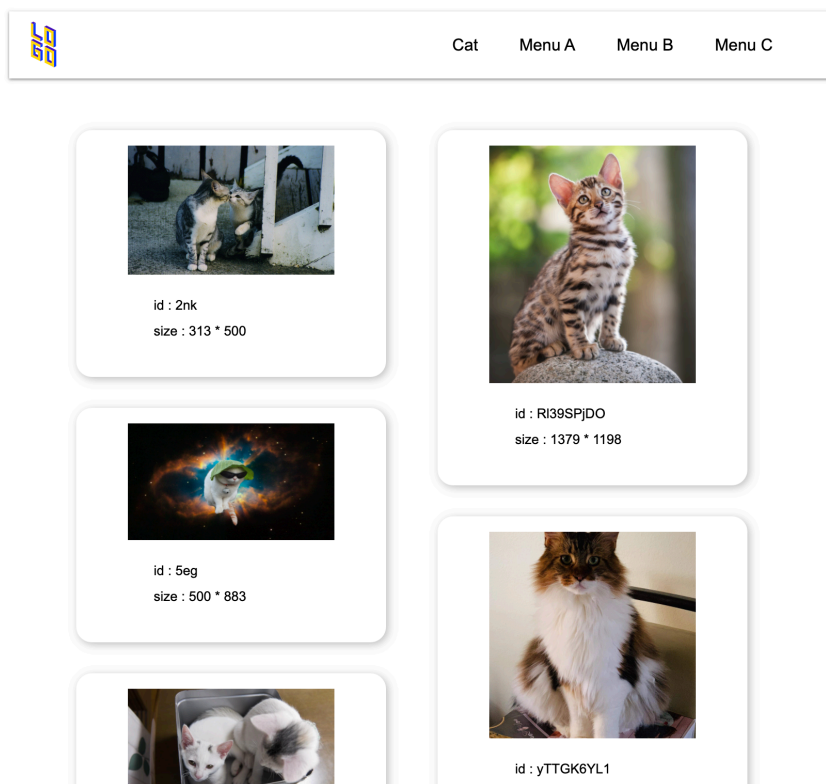
また、`script` と `style` には、`lang='***'` と記載すれば、`typescript` や、`scss` などの拡張言語に変更することもできます。

共通部分の書き方

昨今の Web ページは、共通部分があることがほとんどです。共通部分の例として、`/src/routes/cat/index.svelte` が出力する以下のページをご覧ください。
`http://localhost:3000/cat`

(注: もし、`catApi` にアクセスできない状況だと、ここで `json` を受け取れないので、画面には何も映りません。その場合の対策を用意したので↓、よしなにソースを切り替えてください。)

```
$catsStore = await getCats(10) //もしこれがだめだったら、
$catsStore = await getCatsMock() //これを使おう
```



▲図 2.10

このページは、

- ヘッダーなどの、ページの外枠的なレイアウトを決めている部分
- 猫の画像が並んでる部分の、1 つ分のパーツ

は、共通化しています (要は、同じソースコードをなんべんも書かんでいいようにしている。)

前者は、「どのページでも使い回す」 素材です。

これは、`__layout.svelte` に実装されています。svelte では、`__layout.svelte` と命名されたソースが、`/routes` 直下にある時、そのソースを全てのページの共通パーツとして認識します。^{*12}

^{*12} <https://kit.svelte.dev/docs#layouts>

```

<div class='logo' on:click={()=>goto('/')}>
  <img src='/img/logo.svg' alt='LOGO'>
</div>
<div class='menu-row-list'>
  {#each $mainmenus as menu}
    <HeaderMenuBox name={menu.name} path={menu.path}></HeaderMenuBox>
  {/each}
</div>
</header>
<section id='body'>
  <slot></slot>
</section>
</article>
{/if}
</MediaQuery>

```

▲ 図 2.11

そして、___layout.svelte の任意の部分に<slot/>を記載します。すると、各ページのマークダウンは、この<slot/>以下に描画される扱いを受けます。

後者の、猫 1 匹分のパーツは、俗に「コンポーネント」と呼ばれる単位です。コンポーネントを実装はとても簡単です。まず、

1. src 配下のどこかに、.svelte ファイルでパーツを実装。(当サンプルでは、/src/lib/component/以下に記載し、\$lib/component/... とすれば import できるように設計してあります。)
2. import CatApiComponent from '\$lib/component/Home/CatApiComponent.svelte' といった具合で、コンポーネントをインポート
3. <CatApiComponent 引数... />と、マークダウンの部分に記載。

```

//コンポーネントの実装
<script lang='ts'>
  import type {CatModel} from '$lib/model/Cat'
  ....
  // 補足: export let と書いておくと、コンポーネントは引数を受け付けるようになる。
  export let cat: CatModel

```

```
//コンポーネントを呼び出す側の実装
<script lang='ts'>
  import CatApiComponent from '$lib/component/Home/CatApiComponent.svelte'
  ....
  // cという変数を、引数で用意したcatに渡すと、コンポーネントに情報を渡すことができる！
  <CatApiComponent cat={c} on:catLoveLove={catLoveLoveHandler}/>
```

フロント初學者の方は、ここで、デベロッパツール等を使い、サンプルのソースが、実際にブラウザにどう乗っかってくるのか観察すれば、やっていることがよりわかるはずです！^{*13}

ルーティング

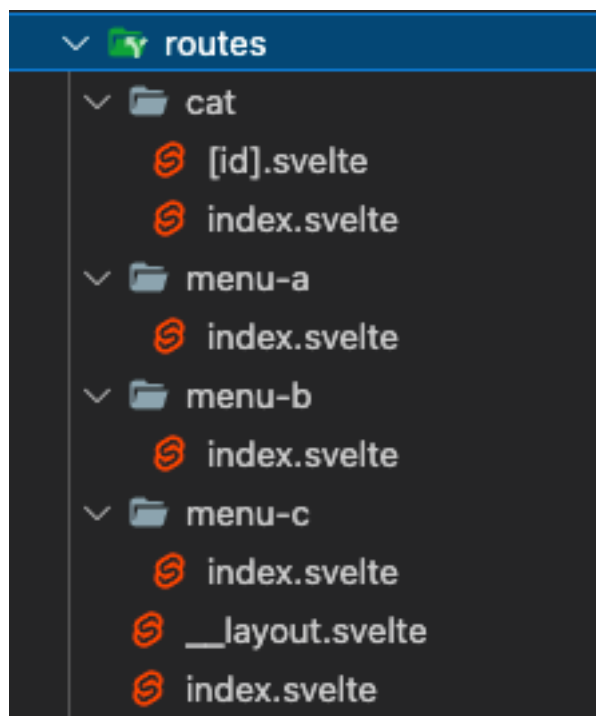
基本的に、.svelte ファイル 1 枚が、ページ 1 枚になります。そして、すでにお察しの方もいるかもしれませんが、

.svelte ファイルの名前・ディレクトリそのものが、ページの URL になります。

注：(index.svelte は、それが属する「フォルダの名前」が URL になります。)(「_」(アンダーバー) がファイル名についでいるフォルダ・ファイルは、ルーティングから無視されます。)

例えば、

^{*13} <https://willcloud.jp/knowhow/dev-tools-01/>



▲図 2.12

このサンプルは、上記のような構成をしているので、

- `http://localhost:3000`
- `http://localhost:3000/cat`
- `http://localhost:3000/cat/{id}`
(`{id}` には、URL セーフならなんでも受け付ける)
- `http://localhost:3000/menu-a`
- `http://localhost:3000/menu-b`
- `http://localhost:3000/menu-c`
- (`__layout.svelte` は無視。)

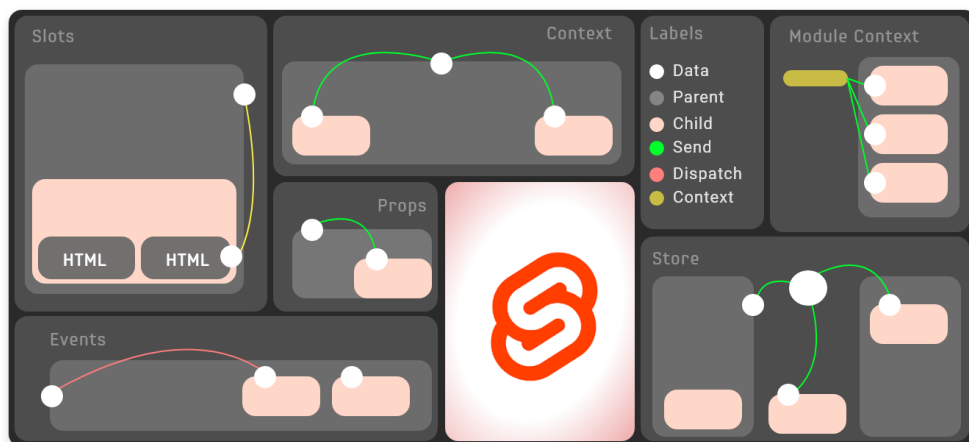
これらが URL として開放されます。

マークダウン部分の制御構文

サンプル内には、`{#each}`や`{#if}`などの、制御構文が登場します。サンプルでは使っていませんが、`{#await}`も登場しました。これらの構文で、マークダウン部分をプログラミング的に制御して、表示項目の繰り返しをしたり、表示の切り替えをすることができます。^{*14}

ストア

フロントエンドのフレームワークでは、ファイル間を跨いだ処理を書きたい時に難儀するものです。その点、SvelteKit はここがすごい強いと感じます。



▲ 図 2.13

上図は、svelte に搭載されている「跨ぎ」の機能一覧と動作の抽象図です。先ほど紹介した`<slot/>`も、跨ぎの一種です。跨ぎに関しては色々ありまして、その場その場で最適なものがあります。が、「とりあえず使っとけ」と推したいのは **store** です。

store は、あなたが作ったアプリケーションが、ブラウザで表示され続けている限り、あらゆる変数の値を保存し、どの画面でも共有することができる機能で、フロントエンドのあらゆる跨ぎを助けてくれます。

そして、SvelteKit がすごい強いと申し立てました最大の理由は、この **store** への書き込み/読み込みが猛烈に楽 だからです。

^{*14} <https://svelte.dev/tutorial/if-blocks>

store を使っている実装は、簡単ですが sample にあります。

```
// storeのための「モデル」の宣言。catAPIから得られる猫モデル
export interface CatModel {
  breeds: []
  height: number
  id: string
  url: URL
  width: number
  /* catAPIの返却値サンプル
    breeds: []
    height: 1024
    id: "c92"
    url: "https://cdn2.thecatapi.com/images/c92.jpg"
    width: 784
  */
}
```

```
import { writable } from 'svelte/store'
// さっき作った 猫モデルをimport
import type { CatModel } from '$lib/model/Cat'

// 猫モデルの配列[]を、storeが読み書きできるように定義し、その結果をcatsStoreに代入
export const catsStore = writable<CatModel[]>([])
```

これにて、catsStore という変数が、あらゆる.svelte ファイルで読み書き可能になりました。試しに、読み取りを行っているソースが以下のソースだ。catsStore をインポートし、

```
<script lang="ts">
  ...
  import { getCats } from '$lib/api/catApi'
  import { catsStore } from '$lib/store/CatStore';
  import { onMount } from 'svelte';
  ...

  onMount( async()=>{
    //storeへのアクセスは、「$」マークをつける。
    if($catsStore.length == 0){
      //getCats関数(自作)が、catAPIで10匹の猫を取得するので、catsStoreに代入(書き込み)
      $catsStore = await getCats(10)
      //catsStoreのコンソール表示(読み込み)
      console.log($catsStore)
    }
  })
}
```



```
//ストアをマークダウン部分でも使え、catsStoreが変更されたら、この部分も連動して変更される。  
(再描画という。)  
{#each $catsStore as c, i}  
  <CatApiComponent cat={c} on:catLoveLove={catLoveLoveHandler}/>  
{/each}
```

といった具合です。Model の定義などが若干礼節的に感じる方もいらっしゃるですが、面倒がほとんどないです。

store は大抵の場合 `object` か `array` なので、javascript 特有の、代入時のシャローコピーの仕様^{*15}を理解していれば、store の操作でつまづくことはないでしょう。

dispatcher

store が強力すぎるので、霞んでしまいがちだが、dispatcher も、コンポーネント間共有においては、同様に強力です。dispatcher が使えるシチュエーションは限られていて、**あるコンポーネントが親に何か伝えたいとき** に使うモジュールです。サンプルだと、`/cat/index.svelte` と、そこが呼び出している `CatApiComponent` で使われている。

```
<script lang='ts'>  
  import type {CatModel} from '$lib/model/Cat'  
  //dispatcherの定義  
  import { createEventDispatcher } from 'svelte'  
  const dispatch = createEventDispatcher()  
  
  export let cat: CatModel  
  
  function onClickHandler(){  
    //こんな具合で、親に何を伝えたいかを定義。  
    dispatch('catLoveLove',{  
      id: cat.id  
    })  
  }  
</script>  
// 補足：エレメントにon:カトカ と書くと、そのエレメントに、カトカの動作をしたら〜する、を実装できる。  
// この場合は、「クリックしたら onClickHandlerを実行」ができる。  
<article on:click={onClickHandler}>  
  <section class='components-box'>
```

^{*15} <https://qiita.com/JetNel0/items/15087313a88f1986d20c>

```
<script lang="ts">
  import CatApiComponent from '$lib/component/Home/CatApiComponent.svelte'
  import { goto } from '$app/navigation';
  ...

  function catLoveLoveHandler(event: CustomEvent){
    //svelteの、gotoモジュールを使って、localhost:3000/cat/{id}へジャンプ
    goto(`cat/${event.detail.id}`)
  }

  ...
  //ここで、「on:定義したdispatcherの名前」とすると、子供の連絡を受け取る体制が取れる。
  <CatApiComponent cat={c} on:catLoveLove={catLoveLoveHandler}/>
```

上記の例は、サンプルで dispatcher を使いたいがために用意した ゴミ コードなので、若干魅力が伝わりづらいかも。(やるなら普通に、猫コンポーネントが goto すれば、いいよね。)

dispatcher は、コンポーネントに何個でも置けるので、コンポーネントが多くの機能を内包する場合は、強力な機能である。

API 叩く部分

このサンプルの /cat は、catAPI を叩いて、猫の画像情報などを取得し、それをコンポーネントに当てはめ表示している。svelte とは関係ないが、API を叩く部分がある程度用意しておいたので、見ていこう。

```
import axios from 'axios' // (API叩くライブラリ)
import type { CatModel } from '$lib/model/Cat' //猫モデルをここでも使います。

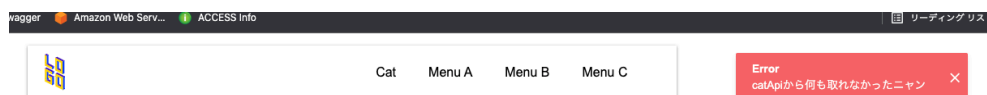
// catApiを叩くことしか脳のないcatApiオブジェクトを定義
const catApi = axios.create({
  baseURL: "https://api.thecatapi.com/v1",
  headers: {
    'Content-Type': 'application/json',
  },
})

//catApiをREST APIの 「GET」 メソッドで送信する関数の定義。
const cat_get = async (url, request?) => {
  const res = await catApi.get(url, { params: request })
  return res.data
}

// catApiの/image/search APIを叩く関数
export const getCats = async (limit:number = 3): Promise<CatModel[]>=> {
  try{
    // @ts-ignore
    let data: CatModel[] = await cat_get('/images/search',{limit})
    console.log(JSON.stringify(data)) //受け取ったデータを、デベロッパツールで見て
```

```
みよう。  
    return data  
  }  
  catch (error) {  
    throw error  
  }  
}
```

ちなみに、サンプルでは、easytoast (自作。svelte-toast^{*16}のラッパ) を使って、トーストを表示するようにしておきました。下図の、右上に出てきてるやつですね。



▲図 2.14

2.6.3 その他にも...

このサンプルで最も複雑なことをさせている部分は、多分__layout.svelte です。

- ヘッダーを表示する
- ヘッダーのナビゲーションを行う
- ヘッダーのフレキシブルレイアウト
- スマホ版のレイアウト (横幅 700px 以下) の時の、ハンバーガメニューの実装
- スクロールイベントを起こすと、ヘッダーを勝手にしまう処理
- isometric なロゴ (自作・著作権フリーだよ)

このソースでは、これらの機能を実装するために、いくつか svelte の助けを得ています。例えば、

```
<MediaQuery query="(min-width: 701px)" let:matches>  
{if matches}  
  ...  
{/if}  
</MediaQuery>
```

^{*16} <https://github.com/zerodevx/svelte-toast>

↑ MediaQuery タブは、外部モジュールの `svelte-media-query` を使ってます。これは、フレキシブルレイアウトを実現する手段の一つです。これは CSS のメディアクエリ機能とほぼ同じですが、フレキシブル化の際に、表示したい要素を増減させたいときに強力な機能です。

```
<svelte:window on:scroll={scrollEvent} bind:scrollTop={scrollTop}/>
```

↑ これは、svelte が用意した window インターフェース^{*17}のラップみたいなもので、window のイベントを検知して、指定した関数を発火してくれます。

```
<header class='{isHideHeader ? "hide":""}'>
```

↑ svelte のマークダウン部分では、こんな感じで、三項演算子を使って、その要素にクラスを付与するか否かの実装もできます。これは、CSS アニメーションの達人たちにとって非常に強力な機能です。

2.6.4 他のことをやらせてみよう

`menu-a`, `menu-b`, `menu-c` は、空きのファイルにしてある。ここまでの機能は、実務でも普通に使うレベルの実装なので、これらの機能を使って、何か作ってみましょう!!!

2.7 デプロイ

ここまでの実装を、vercel にデプロイして、世界に公開しよう。

2.7.1 アカウント作成

vercel の公式のサービスサイト^{*18}から、ログインしましょう。特別な理由がないなら、課金はしなくていいです。

また、GitHub のアカウントがない方は、作成しましょう^{*19}。

^{*17} <https://developer.mozilla.org/ja/docs/Web/API/Window>

^{*18} <https://vercel.com/>

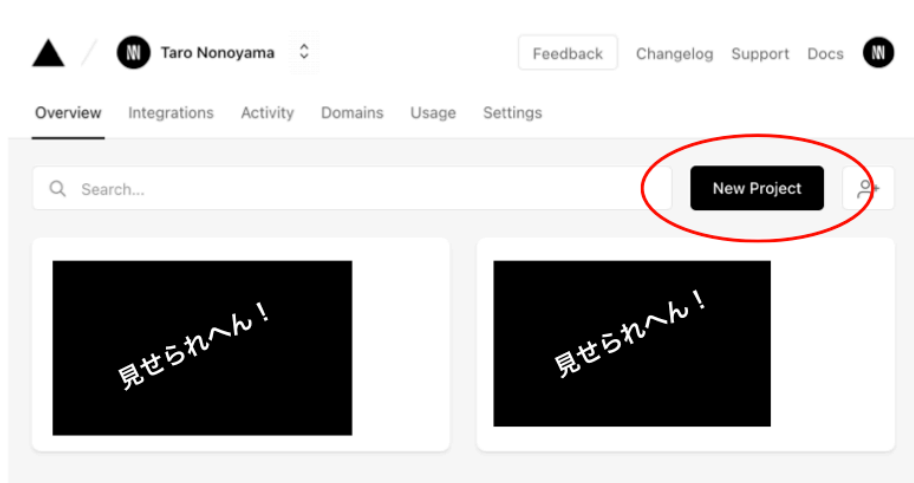
^{*19} <https://github.com/>

2.7.2 GitHub にソースをあげる

作ったソースを、GitHub にあげましょう。間違って `n2-freevas/svelte-sample` に上げないでください。

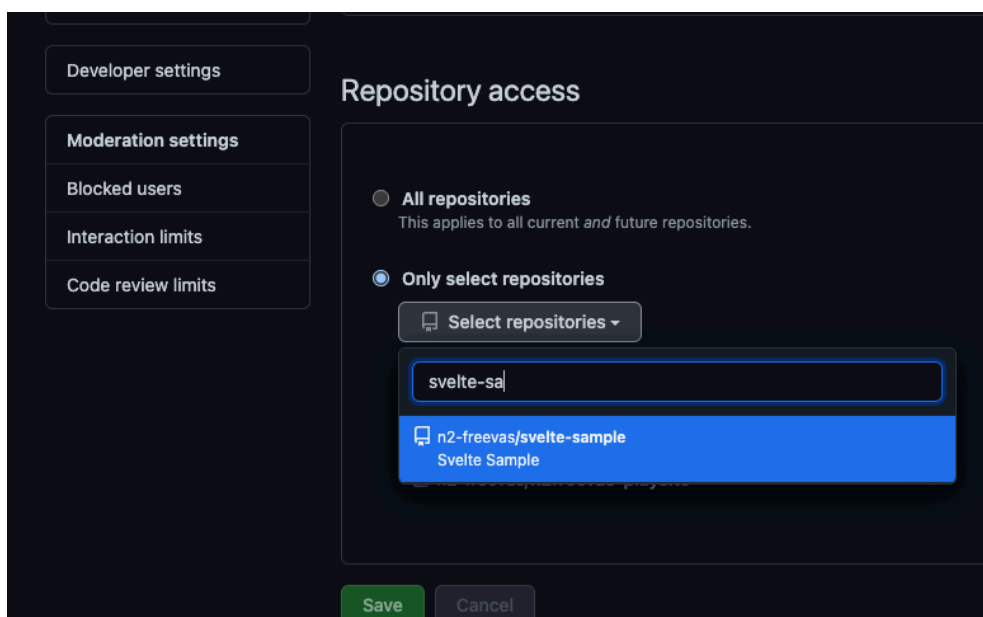
2.7.3 Vercel のプロジェクト作成

Vercel にログイン後、下図のような、ダッシュボードの「New Project」から、



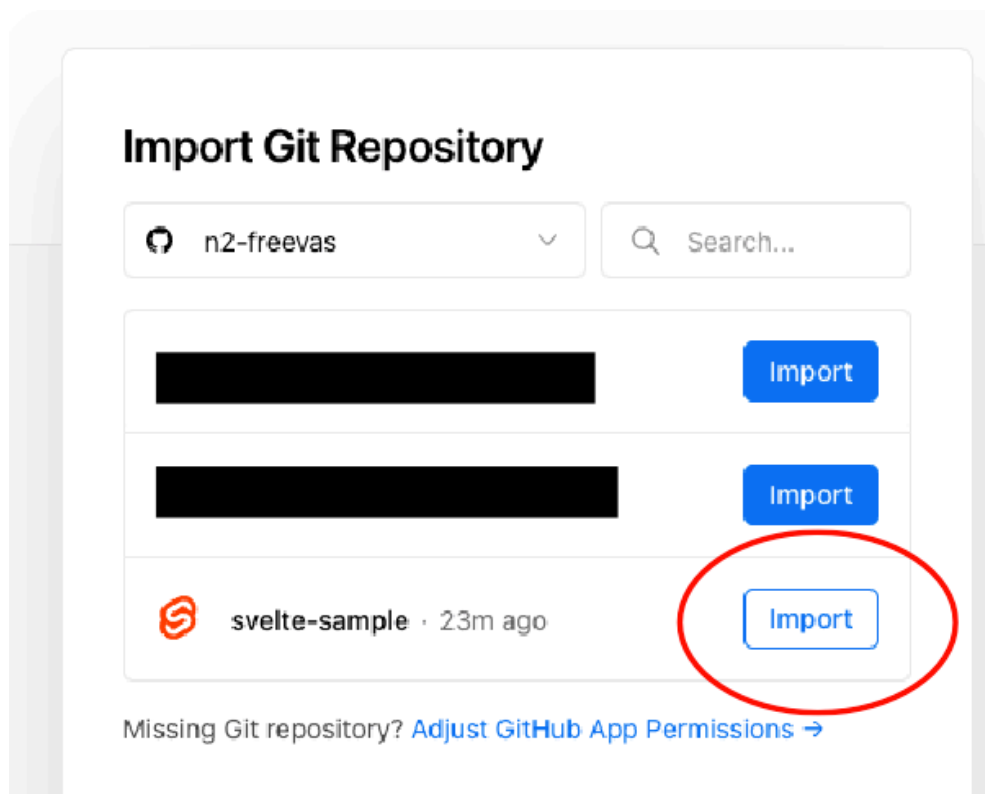
▲図 2.15

Adjust GitHub App Permissions →をクリックして、GitHub の特定のリポジトリと、vercel を連携する。



▲ 図 2.16

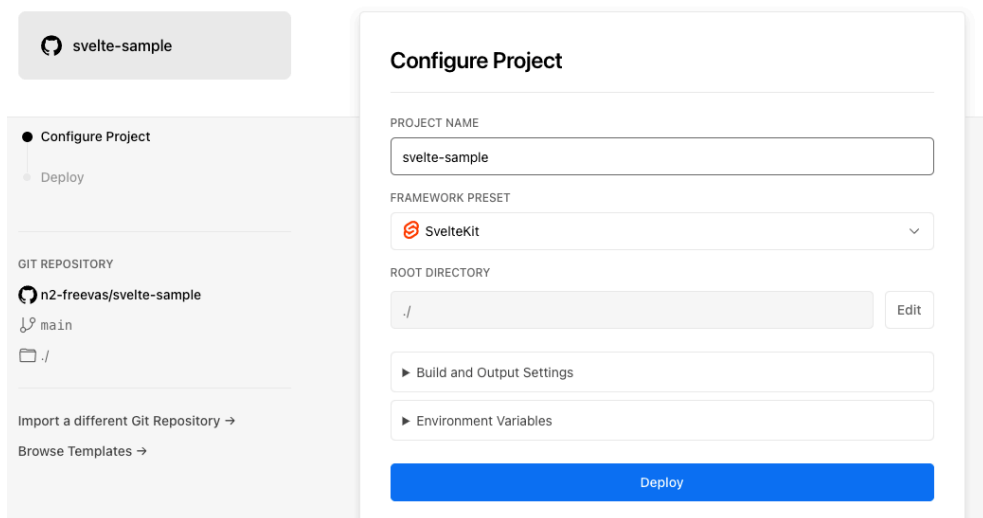
リポジトリを選択して Save したら、vercel に戻るので、連携したリポジトリを import



▲ 図 2.17

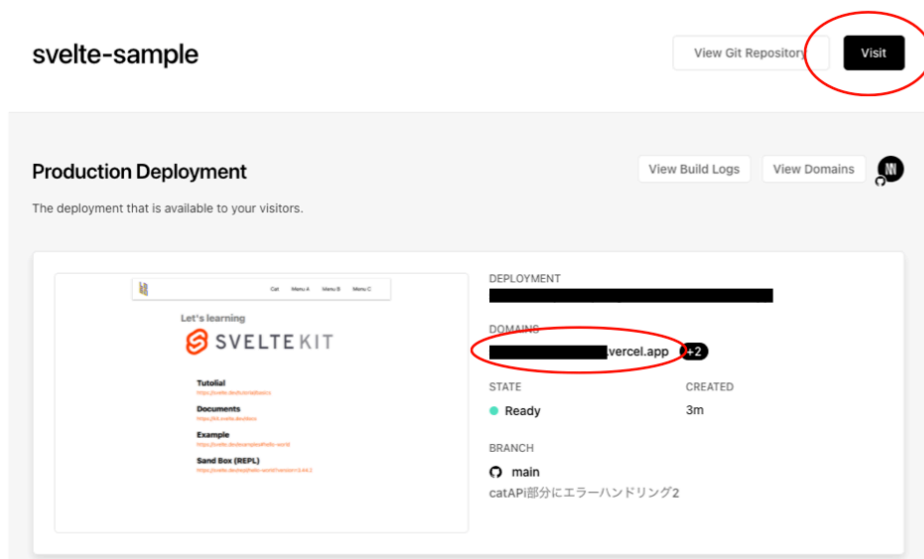
そのまま Deploy で OK です。(ちなみに、ここの build 作業は、ローカルでも確認できます。(`$ npm run build`)

ローカルのこれが通らないなら、基本的に vercel でも通らないので、注意してください。また、本サンプルは vercel にあげるための設定をしているので、それを変更しない限りは、他のデプロイサービスではデプロイに失敗します。



▲図 2.18

デプロイに成功すると盛大に紙吹雪が舞った後、いくつか進むと以下のような画面に入ります。この時点で、勝ち確です。「Visit」を押すか、DOMAINS に書かれているドメインをコピーして、`https://{ドメイン}` で、デプロイしたソースが見れます。



▲ 図 2.19

(Vercel、簡単すぎて怖いです。)

この方法で GitHub と連携しておけば、GitHub にソースをあげたら、vercel に勝手にデプロイされます。もし不都合があれば、設定を変えておこう。

2.8 バックエンド開発の準備

さて、前節までで、SvelteKit の冒険と、Vercel の驚異的なデプロイ能力を垣間見たことでしょう。以降は、サーバーサイドを開発していきます。

2.8.1 サンプルをダウンロード

この URL^{*20} から、GitHub に上げたサンプルにアクセスし、クローンします。

なお、FastAPI に関しては、作者の方が直々にプロジェクトソースコードを公開しており^{*21}、ある程度仕組みをわかっているなら、これを元にプロジェクトを立ち上げてもらいかもしれません。

^{*20} <https://github.com/n2-freevas/fastapi-easy-sample>

^{*21} <https://github.com/tiangolo/full-stack-fastapi-postgresql>

ダウンロードしたら、まず、README.md の指示に従って、ローカルの環境構築を行ないます。(\$ uvicorn のコマンドは、まだ実行しなくていいです。サーバを立ち上げてしまいます。)

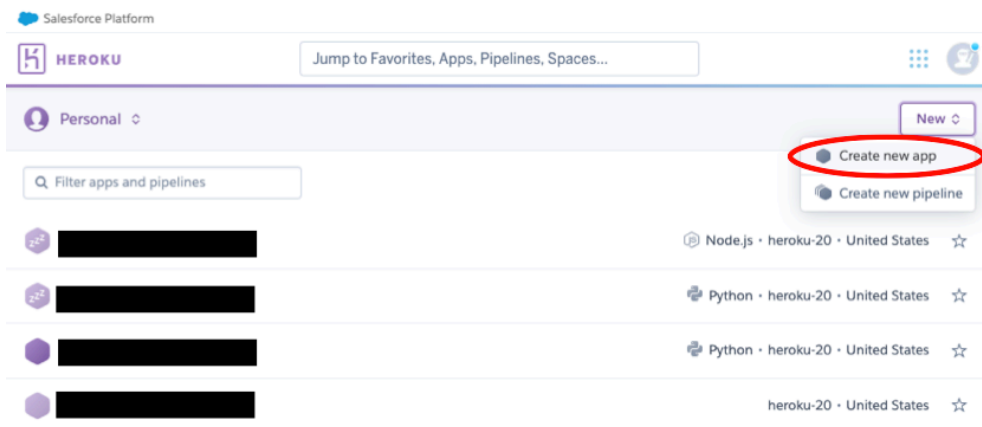
2.9 heroku に登録

前半戦での説明通り、heroku にサーバの実装をデプロイします。ここで行うことは3つで、

- heroku に登録
- heroku で「アプリケーション」を作成
- heroku で、PostgreSQL のデータベースを無料で貸与

です。

まず、heroku のサービスサイト^{*22}にて、アカウントを作成 & ログインを行います。そうしたら、



▲ 図 2.20


◆ Create new app を押して、

^{*22} <https://signup.heroku.com/jp>

第2章 SvelteKit + FastAPI + vercel + heroku でやる気があれば誰でも簡単フルスタックエンジニア



Create New App

App name



fastapi-sample-for-n2-freevas is available

Choose a region

 United States 

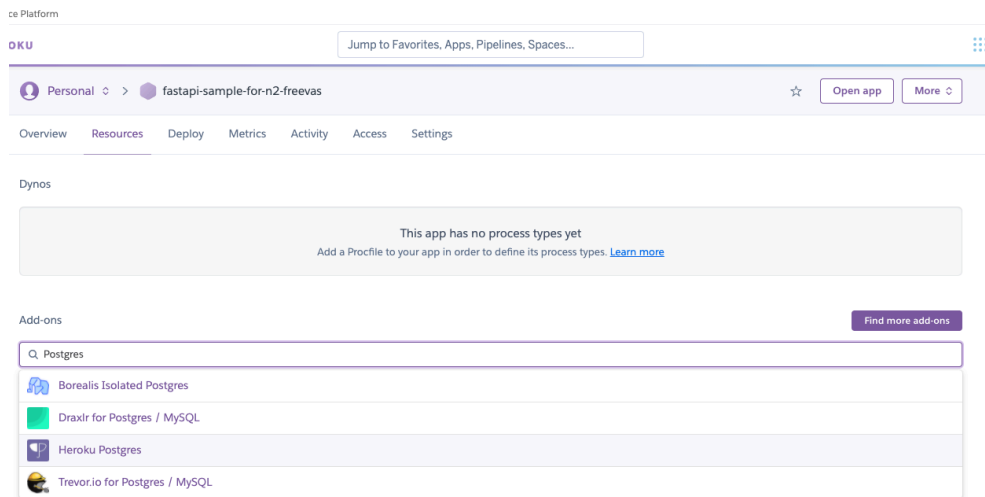
Add to pipeline...

Create app

▲図 2.21

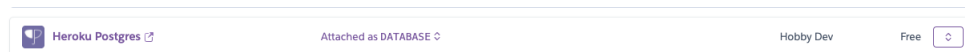
適当なアプリケーション名とサーバの置き場所を指定して、[Create app]

2.9 heroku に登録



▲図 2.22

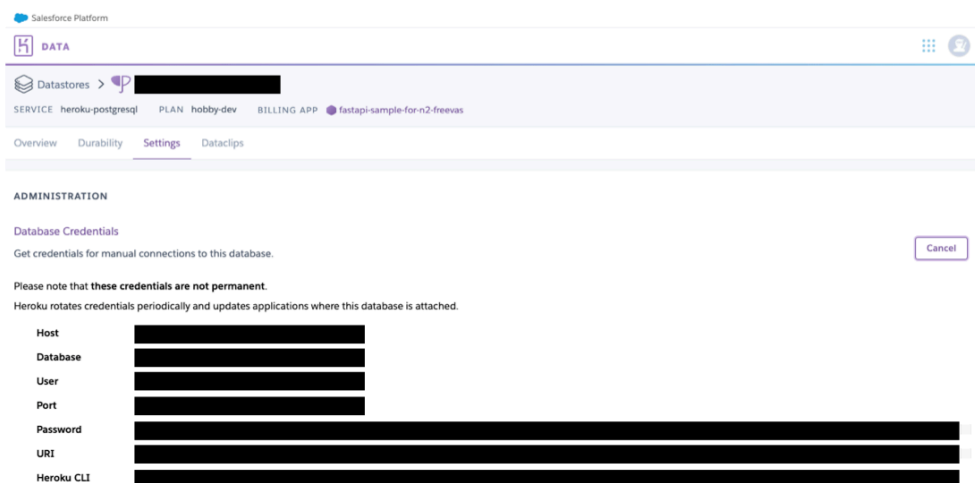
次に、Resource > Add-on の検索窓から、「Heroku Postgres」を探す。> 決定 Hobby プランが無料です。



▲図 2.23

登録すると、Heroku Postgres のコンソールに行けるようになるので、

第2章 SvelteKit + FastAPI + vercel + heroku でやる気があれば誰でも簡単フルスタックエンジニア



▲ 図 2.24

コンソール移動 > Setting > Database Credentials > View credential で、DB への接続情報を GET できます。この URI を、サンプルソースコードの、`/config/db_config.py` の、「ここに URI 入れて！」部分に入れてください。注意として、URI の先頭は `postgres:` ですが、今回使うソースでは、`postgresql:`と書き換えてください。

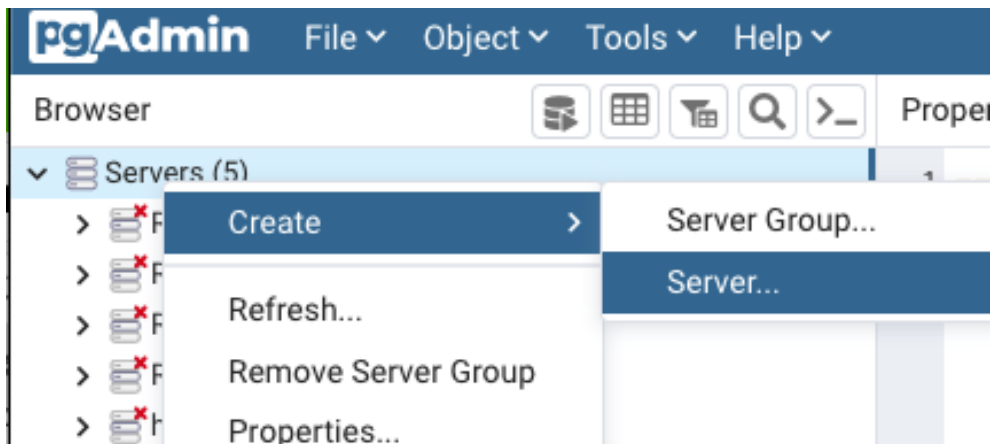
```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

engine1 = create_engine(
    "postgresql://...残りはそのまんま..."
)
```

2.9.1 pgadmin のインストールと利用

DB の中身を覗くことは実は結構難儀します。そこで、pgadmin を使って、可視化します。インストールは下記 URL^{*23}の記事、Qiita: pgAdmin インストール手順 を参考にするのが良いでしょう。

^{*23} https://qiita.com/pyon_kiti_jp/items/01d6150e46bd66be29f0



▲図 2.25

インストールができましたら、Server を右クリック > Create > Server すると、設定画面が出てくるので、先ほど Heroku Postgres で出てきた Credential の情報を入れていきます。

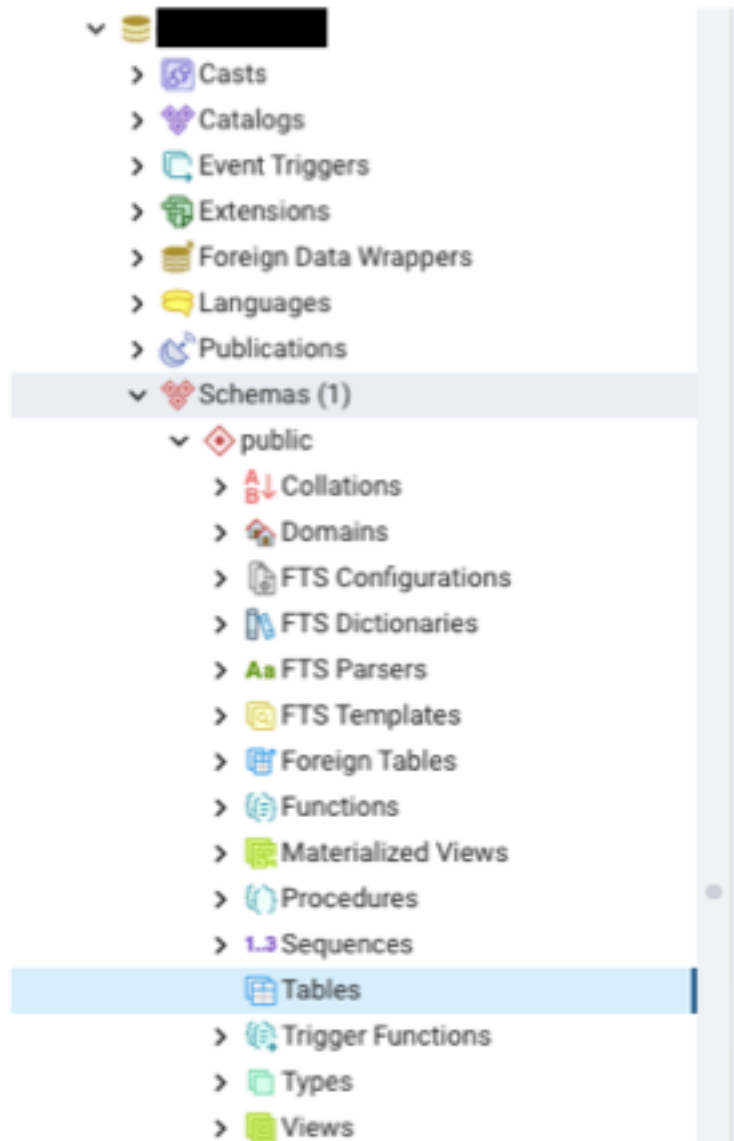
- General タブ
 - Name に、好きな名前を設定 (pgadmin4 上での、表示名になる)
- Connection タブ
 - Host name/address に、「Host」の記載を入力 (ほとんどの場合、ec2-から始まるやつ)
 - Port に、「Port」の記載を入力 (ほとんどの場合 5432)
 - Maintenance database に、「Database」の記載を入力
 - Username に、「User」の記載を入力
 - Password に、「Password」の記載を入力。そしてセキュリティ意識高い系でもない僕たちのような人は、特別な理由がない限り、そんなクソ長いパスワード忘れるので下のチェックマークを ON にして、入力を省略できるようにしよう

Save を押すと、左メニューバーに作った DB 接続が表示されるので、プルダウンしていきます。



▲図 2.26

{ 作った名前 } > Databases > {Maintenance database の名前までスクロール} > Schemas > Tables > 何も表示されないが正解



▲図 2.27

ちなみに、Databases 移行に死ぬほど DB が並んでいるのは、Heroku が一つの接続に、複数の DB を登録して、その 1 つ 1 つをユーザに対して間借りする形でやってるからだ

第2章 SvelteKit + FastAPI + vercel + heroku でやる気があれば誰でも簡単フルスタックエンジニア

思われます。課金したら変わるのかな？

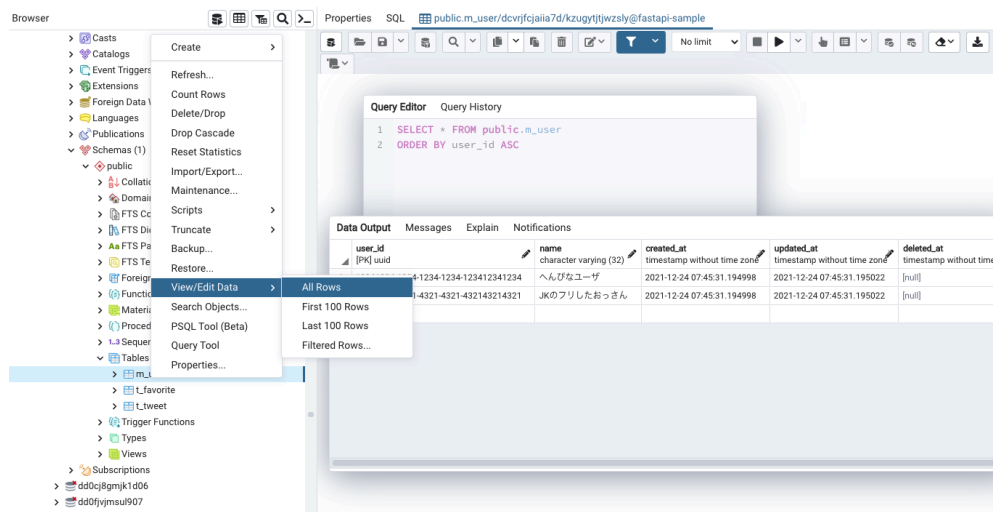
ここで、DB 関連で、注意事項をいくつか。

- 低料金プランでは、最大接続数が決まっています。Hobby はたった 20 接続しか確保されません。
- 「間借り」しているので、ちょっと重たいです。
- 登録できるレコードは、Hobby で上限 10000 レコードです。上限を超えると、メール通知ののちに、1 週間程度の猶予期間が始まり、そのあとは、消えるかなんらかの対処がなされます。

次に、用意しましたサンプルデータを突っ込みます。./createtable.py を実行すれば、あなたの用意した Heroku Postgres DB に勝手にサンプルデータを流し込みます。少量なので帯域とかは気にしなくて大丈夫。

```
python createtable.py
or
python3 createtable.py
```

そうすると、DB にデータが入られるので、早速、pgadmin4 で確認してみましょう。



▲ 図 2.28

Table > どれかのテーブル (3 つ入っているはず) > 右クリック > View/Edit Data >

All Rows で、ユーザらしきものが2つ出れば成功です。

最後に、README.md にも記載しております、起動シーケンスを実行して、サーバをあなたの PC で立ててみましょう。

```
uvicorn main:app --reload
```

を実行して、

```
INFO:      Will watch for changes in these directories:
...
INFO:      Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:      Started reloader process [27454] using watchgod
INFO:      Started server process [27456]
INFO:      Waiting for application startup.
INFO:      Application startup complete.
```

Application startup complete が出たら、事前準備はこれにて終了です！

2.10 FastAPI をテンプレートから学ぶ



▲図 2.29

お待たせしました。インフラ整備は、さっきのでほとんど終わりです。いよいよサーバの実装です。

流し込んだデータを見てもらえればわかる通り、サンプルでは、Twitter っぽいシステムを構築しようとしています。FastAPI の魅力はなんといっても、その早さ (実装のシンプルさ) と速さ (処理速度的な) です。

では、それらの機能を、実装と共に見ていきます。

2.10.1 リクエスト/レスポンスの実装

リクエスト/レスポンス は、いわば API の主要な概念であり、定義です。

FastAPI はかなり楽にそれを定義できます。/api/twitter_modoki/__init__.py の、このコードをみていきましょう。

```
twitterModokiRouter = APIRouter() #一旦ここは気にしない

# .(メソッド名) ('{パス}', response_model={レスポンスの形式})
@twitterModokiRouter.get('/tweet/list', response_model=List[TweetResponseModel])
def 全てのツイートを取得するAPI( # APIの関数名が、自動生成ドキュメントのAPIの説明文になります。
    user_id: UUID = None, # クエリパラメータ。 型チェック(もしくは変換)・軽いバリデーションもしてくれる
    offset: int = 0,      # クエリパラメータ, デフォルト値を設定すると自動的にOptionalになる。
    limit: int = 100,     # クエリパラメータ
    session: Session= Depends(get_session) # APIの開始時にget_sessionが呼び出され、終了時にはget_sessionのfinallyを実行する。
):
    try:
        .....
```

このコードで、以下の機能が実装されました。

- `http://{ドメイン}/tweet/list` に窓口開設
- 窓口では `user_id`、`limit`、`offset` の3つのパラメータを任意で受け付ける。型チェックや、軽いバリデーションなども行う。(例えば、`user_id` は UUID 形式出ないとエラーを吐く。`limit`、`offset` は、数値型になりうるならパラメータから勝手に数値型に変形する。)
- 3つのパラメータは、任意である。(= `***` と書くと、その値がデフォルトになり、パラメータが `Optional`・任意になる)
- 窓口は、レスポンスの形式を `TweetResponseModel` の配列: `List[TweetResponseModel]` で返すように約束。
- `getsession` 関数 (DB の接続を確保する関数、自作) を、レスポンスを返すまで取り回せるようにする。この例では、「APIが終わったら、DBの接続をきる」という実装がこれだけで済む。(詳しくは、`getsession` の実装を参照)

どうでしょうか。これらの実装が、数行で済んでいることがまず驚きである。

では、サーバを起動して、

```
uvicorn main:app --reload
```

API を呼んでみよう。

```
// コマンドラインからAPIを呼ぶなら
curl -X GET http://127.0.0.1:8000/twitter-modoki/tweet/list?user_id=12341234-1234-1234-1234-123412341234&offset=1&limit=10
```

API Tester で呼んでいる様子↓



▲図 2.30

すると、こんなのが帰ってくる。

```
{
  "favorites": 0,
  "tweet_id": "11111111-1111-1111-1111-111111111113",
  "user_name": "へんびなユーザ",
  "tweet_text": "意味不明すぎて草",
  "favorites": 0
},
{
  "tweet_id": "11111111-1111-1111-1111-111111111111",
  "user_name": "へんびなユーザ",
  "tweet_text": "ハーゲンダッツって高級感の割に300円でマ?",
  "favorites": 0
}
]
```

▲図 2.31

パラメータを指定しなければ、

第2章 SvelteKit + FastAPI + vercel + heroku でやる気があれば誰でも簡単フルスタックエンジニア

```
curl -X GET http://127.0.0.1:8000/twitter-modoki/tweet/list
```

こんなふうToListが帰ってくる。

```
{
  "tweet_id": "11111111-1111-1111-1111-111111111113",
  "user_name": "へんびなユーザ",
  "tweet_text": "意味不明すぎて草",
  "favorites": 0
},
{
  "tweet_id": "11111111-1111-1111-1111-111111111111",
  "user_name": "へんびなユーザ",
  "tweet_text": "ハーゲンダッツって高級感の割に300円でマ?",
  "favorites": 0
},
{
  "tweet_id": "11111111-1111-1111-1111-111111111112",
  "user_name": "JKのフリしたおっさん",
  "tweet_text": "犬ってゆうのわ。。\n英語で「dog」\n逆から読むと。。\n「god」\nそう神。。\nいみわかんない。。。",
  "favorites": 1
}
]
```

lines nums length: 760 bytes

Top Bottom Collapse Open 2Request Copy Download

▲図 2.32

これらは、データを「取得」するAPI。(たいていHTTPメソッドでは「GET」で定義される。)では、「POST」の実装はどうだろう。

```
# api/twitter-modoki/__init__.py
@twitterModokiRouter.post('/tweet')
def ツイートを1つ登録するAPI(
    body: TweetRequestBody,
    # POSTメソッドの場合、このようにbodyに格納される予定のjsonオブジェクトの定義と結びつ
    # けると、
    # 勝手に必要な情報を構造体として抽出してくれる
    session: Session = Depends(get_session)
):
    try:

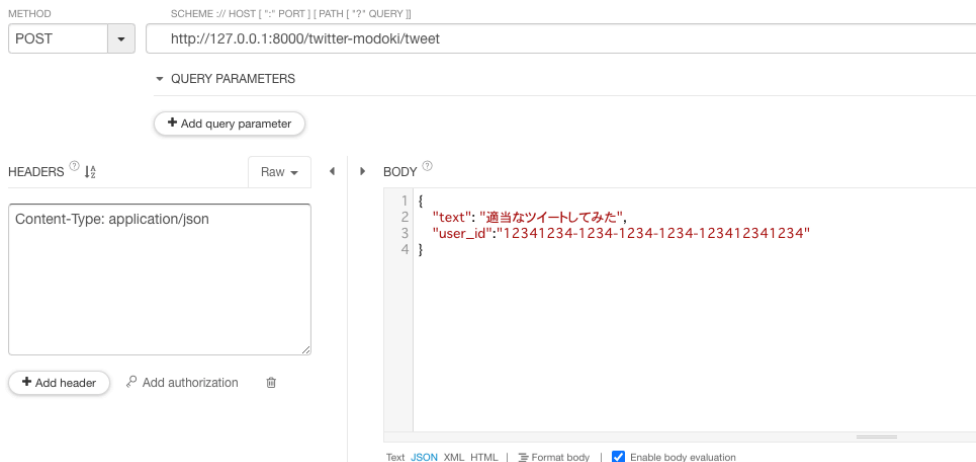
    -----

# schema/request.py
class TweetRequestBody(BaseModel):
    text: str = Field(...)
```

```
user_id: UUID = Field(...)
reply_tweet_id: Optional[UUID] = Field(...
```

GET と同じく、引数部分に何やら `TweetRequestBody` と書くと、POST メソッドで頻繁に使われる「BODY パラメータ」を取得できる。

API を叩く側は、以下のように、BODY パラメータに構造体を書き込んでリクエストを送出する。



▲図 2.33

通信経路を通る際は、BODY パラメータはただの文字列と化すので、サーバ側はその文字列を構造体に直すのに難儀するはずだったが、この実装は、`TweetRequestBody` の定義に沿った構造をパラメータから勝手に作り出し、`body` に代入する。あれまー

サンプルでは、GET, POST メソッドを使ったパターンを紹介したが、基本的にリクエスト/レスポンスのすべての実装は、これだけで賄える。

2.11 ドキュメント生成

`fastapi` の真なる強みとして、ドキュメントの自動生成機能がある。

ドキュメントの生成は自動で行われ、そのドキュメントを見たい時は、`@<tt><{http://localhost:8000/docs }`

つまり、API のドメインの後ろに `/docs` をつけて、アクセスしに行く。これだけ。

生成されたドキュメントの、`/twitter-modoki/tweet/list` の記述がこちら。

GET

/twitter-modoki/tweet/list 全てのツイートを取得するApi

Parameters

Name	Description
user_id string(\$uuid) (query)	<div>user_id</div>
offset integer (query)	<div>Default value : 0</div> <div>0</div>
limit integer (query)	<div>Default value : 100</div> <div>100</div>

Responses

Code	Description
200	<div>Successful Response</div> <div>Media type</div> <div>application/json</div> <div>Controls Accept header.</div> <div>Example Value Schema</div> <div>[{ "tweet_id": "12341234-1234-1234-1234-123412341234", "user_name": "ユーザ1", "tweet_text": "Twitter Modoki!", },]</div>

▲ 図 2.34

さっきの、リクエスト/レスポンスの実装をやっていれば、実務レベルにも耐えうる立派なドキュメントが完成します。これで残業しなくて済むね！

これには、見た目以上にたくさんのメリットがあります。色々言い換えながら説明するのなら、こんな感じでしょうか↓

- ドキュメントをわざわざ別で作らなくていい。
- サーバーの実装ができれば、即座にドキュメントが生成される
- フロントの実装者は、サーバと同じドメインにアクセスすればドキュメントが手に入る
- ドキュメントと実装にズレが生じることがない！！！！ (神)

デフォだと、swagger 形式のドキュメントが生成されますが、redoc が好きな人は、/docs じゃなくて、/redoc とすればそうなります。スゴイネー

また、ドキュメント自動生成機能は、もちろん OFF にすることも可能です。

```
app = FastAPI(docs_url=None, redoc_url=None, openapi_url=None) #オフにする
```

2.11.1 ルーティング

api のルーティングが簡単な方が、実装にも幅が出るし、バージョンングも楽ちんです。FastAPI の、APIRouter、include_router() は、楽ちんを実現してくれます。

```
from fastapi import FastAPI
from api.twitter_modoki.v1 import twitterModokiRouter as v1
from api.twitter_modoki.v2 import twitterModokiRouter as v2

app = FastAPI()

app.include_router(v1, prefix="/twitter-modoki", tags=['TwitterModoki'])
app.include_router(v2, prefix="/twitter-modoki/v2", tags=['TwitterModoki2'])
```

ちなみに main.py は、このサーバー実装のスタートポイントです。app というのが全ての母体で、app に紐付ける形で、router というのをどんどん繋げていきます。prefix の定義もできるので、API をお手軽にグルーピングしたい場合は、このサンプルみたく、バージョンで API を分けたい時にも役立ちます。tags は、自動生成ドキュメントのグループ名になります。

2.11.2 Dependencies による依存性注入

では、API の v2 バージョンで、v1 よりセキュリティを強化したり、リクエストをしやすくしたりしましょう。依存性注入 (DI) を用いて。DI って何？ という方々も、実装を見れば言いたいことがわかります。

さて、v2 では、簡単なセキュリティ要件に対応したようです。

- フロントは、HTTP ヘッダーに、user_id と、認証キー (authorization) が必ず入れる。
- user_id が DB に存在するか、存在チェックをしないといけなくなった
- authorization が違うなら、API にアクセスできなくなった。

これらを、DI で実装しよう。

実は、ここまでで DI っぽい実装が一つ登場しています。

```
session: Session = Depends(get_session)
```

此奴です。この、Depends という fastapi のモジュールが不可思議奇奇怪怪な存在で、DI の魔法に簡単に誘ってくれます。

```
@twitterModokiRouter.get(
    '/tweet/list',
    response_model=List[TweetResponseModel],
    #これにより、2つの関数がAPI処理実行前に必ず行われるようになった。
    dependencies=[Depends(required_header), Depends(required_authorization)]
)
def 全てのツイートを取得するAPI(
    #user_idはHTTP Headerからとる。
    # A = Header(...)で、ヘッダーからカラムAを勝手に探して取る。
    user_id: UUID = Header(...),
    offset: int = 0,
    limit: int = 100,
    session: Session= Depends(get_session)
):
    try:
        .....
```

次に、dependencies に紐付けた 2 つの関数の実装をみてみよう。

```

from fastapi import Header, HTTPException
from sqlalchemy.orm.session import Session
from model.UserModel import UserModel
from config.db_config import SessionLocal
from uuid import UUID

def required_authorization(
    authorization: str = Header(...)
):
    if authorization == 'fast-api-token-barebare':
        pass
    else:
        #キーが違うなら、リクエストをRejectする。
        raise HTTPException(status_code=432, detail="Access invalid")

def required_header(
    user_id: UUID = Header(...)
):
    session: Session = SessionLocal()
    try:
        #get()によって、該当するidをもつユーザが0つ、あるいは2つ以上取れた時、エラーを吐く。
        session.query(UserModel).get(user_id)
    except:
        #ので、リクエストをRejectできる。
        raise HTTPException(status_code=432, detail="Access invalid")
    finally:
        session.close()

```

こんな感じ、単純に、要件にそうチェックを走らせている。おそらく上級者の方は、「Header(...) ってこんなとこまで来ても中身引っ張れるんだ。。。」と驚愕かもしれないが、できてしまうものはできてしまうのである。

これらが通れば、API 関数の↓の部分

```

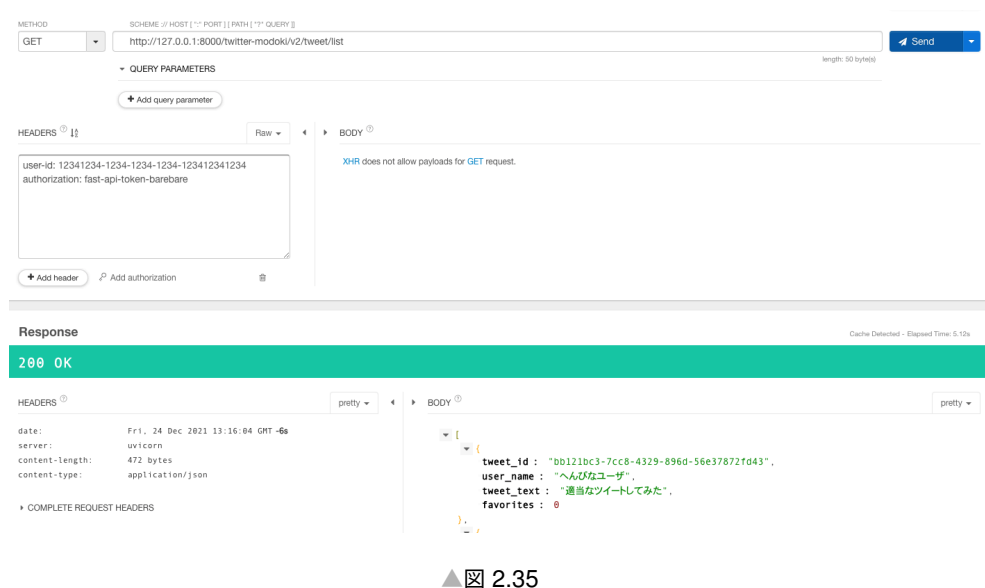
# A = Header(...)で、ヘッダーからカラムAを勝手に探して取る。
user_id: UUID = Header(...),

```

で Header から user_id をとつても OK という感じである。

Header に必要情報を記載して送信。これで、v2 の要件達成である。

第 2 章 SvelteKit + FastAPI + vercel + heroku でやる気があれば誰でも簡単フルスタックエンジニア



そして、この構文なら、

```
dependencies=[Depends(required_header), Depends(required_authorization)]
```

この記述を他の API 関数に取り入れるだけで、認証をかけたい API に同じ処理を実装することができるのです。

2.11.3 補足説明

このサンプルが構成についての説明欄を設けました。

- /api
 - 窓口です。窓口の説明や機能は、おそらく今までに紹介した機能で十分だと思います。
- /service
 - /api から、1:1 対応で呼び出します。いわば API のメイン処理部分。
 - /repository と組み合わせて、DB から欲しい情報を整理して、レスポンスの型に沿ったデータを作り出します。
- /model

- サーバでいう Model とは、データベースに入ってるテーブルの定義の写し鏡です。
- SQLAlchemy という ORM マップ (写し鏡をやってくれるライブラリ) を使っているので、ORM マップのルール通りに Model を実装しています。
- /schema
 - リクエスト、レスポンスの型を定義しています。/api でも頻繁に使っているので、おなじみかと。
- /repository
 - /DB にやらせたい操作を、汎用性持たせつつ書いてます。
 - /service から共通パーツを切り分けた・と捉えてもいいでしょう。
- /config
 - 各種設定。

2.11.4 やってみよう追加課題

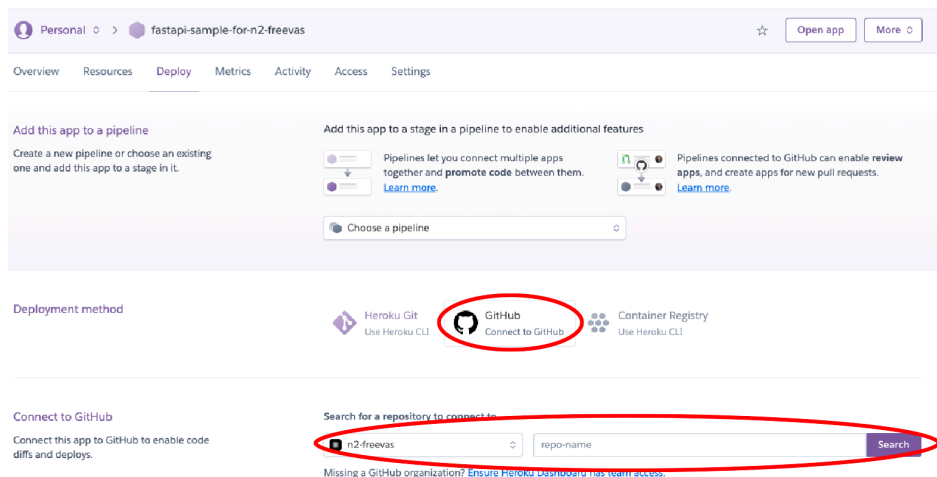
- フォロー・フォロワーを実装しよう
- /tweet/list が、フォロワーの内容しか出てこないように実装しよう
- Twitter Modoki アプリのフロントを実装して、結合してみよう。
- OAuth2^{*24}を使って、ログインの実装をしよう

2.11.5 デプロイ

さて、最後の仕事です。実装をデプロイしましょう。

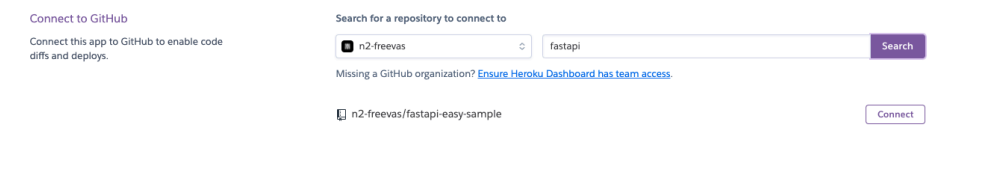
^{*24} <https://fastapi.tiangolo.com/ja/tutorial/security/oauth2-jwt/>

第2章 SvelteKit + FastAPI + vercel + heroku でやる気があれば誰でも簡単フルスタックエンジニア



▲ 図 2.36

フロントエンドの時よろしく、自分のリポジトリにソースをあげた後、heroku の Deploy メニューから、



▲ 図 2.37

リポジトリを検索して Connect

Automatic deploys

Enables a chosen branch to be automatically deployed to this app.

You can now change your main deploy branch from "master" to "main" for both manual and automatic deploys, please follow the instructions [here](#).

Automatic deploys from `main` are enabled

Every push to `main` will deploy a new version of this app. **Deploys happen automatically:** be sure that this branch in GitHub is always in a deployable state and any tests have passed before you push. [Learn more](#).

☐ Wait for CI to pass before deploy

Only enable this option if you have a Continuous Integration service configured on your repo.

Disable Automatic Deploys

Manual deploy

Deploy the current state of a branch to this app.

Deploy a GitHub branch

This will deploy the current state of the branch you specify below. [Learn more](#).

Choose a branch to deploy

`main` **Deploy Branch**

▲図 2.38

あとはよしなに Deploy の設定を行います。とりあえず Automatic deploy の設定をして、

Manual deploy

Deploy the current state of a branch to this app.

Deploy a GitHub branch

This will deploy the current state of the branch you specify below. [Learn more](#).

Choose a branch to deploy

`main` **Deploy Branch**

Receive code from GitHub

Build `main 946f59e2`

```

-----> Building on the Heroku-20 stack
-----> Determining which buildpack to use for this app
-----> Python app detected
-----> No Python version was specified. Using the buildpack default: python-3.9.9
-----> To use a different version, see: https://devcenter.heroku.com/articles/python-runtimes
-----> Installing python-3.9.9
-----> Installing pip 21.3.1, setuptools 57.5.0 and wheel 0.37.0
-----> Installing 50Lite3
  
```

☒ Autoscroll with output [View build log](#)

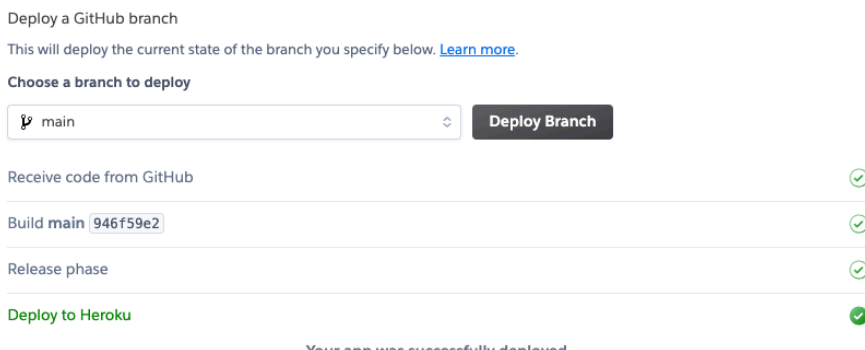
Release phase

Deploy to Heroku

▲図 2.39

初回だけは、Manual deploy を実行。
Procfile (heroku 用のコマンドソース) に、デプロイに必要な定義をしておいたので、あとはデプロイを待つだけ。

第 2 章 SvelteKit + FastAPI + vercel + heroku でやる気があれば誰でも簡単フルスタックエンジニア



▲図 2.40

終わったら、Setting > ちょっとスクロール > Domains に、ドメインが貼られています。これが、`http://localhost:8000/`の代わりになり、デプロイした API はこのドメインの配下に配備されます。

こうなってしまうと、世界中のどこにしようと、ネットワークさえあればこのドメインで自分の API にアクセスできます。

2.12 最後に

いかがでしたでしょうか！ sveltekit + fastapi、2つのフレームワークの強力さを体感できたことはもちろん、昨今のデプロイサービスは恐ろしいほど簡単に世界に配信をかけられてしまうことも伝わったことでしょう。

2021 年、私のように、エンジニアを志す若者にとって、すごくいい時代です。膨大なソースコード資産、俗にいう「巨人」の「肩の上」に成り立つツールで、こんなにも快適で、簡単で、実現できる物事の幅も広い開発体験ができるのですから。

そして何より、簡単なフレームワークというのは、HTTP/TCP や、ブラウザや、その他ベースの基幹技術を理解する上での登竜門でもあります。

なので、誰に罵倒されようと、簡単で、楽で、面白くて、いいのです。遠慮なく巨人の肩の上で、やりたいことやってしましましょう。



▲ 図 2.41

第 3 章

Android のファイルストレージに関する仕様の整理

3.1 はじめに

モバイルアプリを開発していると、しばしば画像・動画・音声といったメディアファイルや PDF などのドキュメントファイルを扱う場面に遭遇します。こと Android OS においては比較的柔軟にファイルを扱うことができるものの、OS バージョンアップの度に大きく仕様変更が入ることが多々あり開発者泣かせになっている節があります。

本章では、そんな複雑化している Android OS のファイルストレージへのアクセスに関して、Android 4.4 から Android 12 までの仕様の変遷をご紹介した後、ユースケースに応じた実装について簡単にご紹介します。

3.2 Android のファイルストレージに関する仕様の変遷

公式ドキュメントの情報を基に、Android 4.4 以降におけるファイルストレージへのアクセスに関する仕様の変遷をまとめました。(紙面の都合上、Android 4.4 以前についての情報は割愛させていただいております...)

なお、Android には「内部ストレージ」と「外部ストレージ」という概念があります。これはそれぞれ以下のような意味合いになります：

- 内部ストレージ (Internal storage): アプリ専用のファイルを保存する領域
- 外部ストレージ (External storage): 端末や SD カード内にある他アプリと共有されたファイルの保存領域

内部ストレージ/外部ストレージは、端末の内蔵ストレージ/SD カードのような区分と

は直接関係していません。また、「アプリ固有のストレージ」「共有ストレージ」という概念もあり、それぞれ以下のような意味合いになります。

- アプリ固有のストレージ (App-specific storage): 内部ストレージと外部ストレージにそれぞれ設けられたアプリ専用の保存領域
- 共有ストレージ (Shared storage): 外部ストレージに設けられている、画像などのメディアファイルやドキュメントファイルのための保存領域

3.2.1 Android 4.4

<https://developer.android.com/about/versions/android-4.4#StorageAccess>

- 外部ストレージの読み込みには `READ_EXTERNAL_STORAGE` 権限が必要になりました。
 - 内部ストレージの読み書きに権限は不要です。
- Storage Access Framework(SAF) が実装されました。
 - 以下の Intent で Android 標準の File Picker が起動し、ファイル単位で読み取り、書き込みの権限を付与できるようになりました。
 - `ACTION_OPEN_DOCUMENT`
 - `ACTION_CREATE_DOCUMENT`

3.2.2 Android 5.0

<https://developer.android.com/about/versions/android-5.0#Storage>

- SAF の機能が拡張され、`ACTION_OPEN_DOCUMENT_TREE` インテントを用いてディレクトリ全体に権限を付与できるようになりました。

3.2.3 Android 6.0

<https://developer.android.com/about/versions/marshmallow/android-6.0-changes>

- 実行時パーミッションが実装されました。
- それに伴い、`WRITE_EXTERNAL_STORAGE` 権限を付与するためには実行時パーミッションによる権限付与が必要になりました。

3.2.4 Android 7.0

<https://developer.android.com/about/versions/nougat/android-7.0-changes#permfilesys>

- ファイルのパーミッションに関していくつか追加で制限が入りました。
- アプリのプライベートディレクトリのパーミッションが 0700 になりました。
- `file://` で始まる URI を使用したファイルの共有に `FileProvider` の利用が推奨されるようになりました。
- `DownloadManager.COLUMN_LOCAL_FILENAME` が使用不可になりました。

3.2.5 Android 8.0

<https://developer.android.com/about/versions/oreo/android-8.0-changes>

ファイルストレージに関する仕様の変更はありません。

3.2.6 Android 9.0

<https://developer.android.com/about/versions/pie/android-9.0-changes-28#per-app-selinux>

- アプリ固有のファイルへの外部のアプリからのアクセスに制限が入りました。
- 他のアプリとのファイル共有については `ContentProvider` の使用が推奨されるようになりました。

3.2.7 Android 10

<https://developer.android.com/about/versions/10/privacy/changes#scoped-storage>

- 対象範囲別ストレージが実装されました。
 - 共有ストレージ内のファイルは `Storage Access Framework` か `Media Store API` を通してしかアクセスできなくなりました。
 - ただし、移行措置として `TargetSdkVersion <= 29` のアプリにおいては `requestLegacyExternalStorage` オプションを `true` にしていれば対象範囲別の利用を回避できるようになっています。
- `Media Store API` に `MediaStore.Downloads` テーブルが追加されました。

- この変更により、Media Store API の テーブルは以下の 4 種となりました。
- `MediaStore.Images`
- `MediaStore.Video`
- `MediaStore.Audio`
- `MediaStore.Downloads`

3.2.8 Android 11

<https://developer.android.com/about/versions/11/privacy/storage>

- Android 11 では、ファイルストレージの仕様に関して大幅な変更が加えられました。(寧ろここがメインなのではというくらいに)
- 対象範囲別ストレージの強制化
 - `TargetSdkVersion >= 30` のアプリにおいては `requestLegacyExternalStorage` プロパティが使用不可になりました。
 - `requestLegacyExternalStorage=false` が記述されていても無視されます。
- アプリ固有のディレクトリを `Context#getExternalFilesDirs()` メソッドによって得られるパス以外の場所に保存できなくなりました。
- メディアファイルに Java File API と `fopen()` などのネイティブメソッド経由でアクセスできるようになりました。
- Android 9.0 で制限が入った他アプリからのアプリ固有データへのアクセスの制限が強化され、外部/内部ストレージともに外部アプリからのアクセスが一切不可になりました。
- Storage Access Framework でアクセス権を付与できるディレクトリに制限が入りました。なお、これらの制限は `TargetSdkVersion >= 30` のときにのみ有効になります。
 - `ACTION_OPEN_DOCUMENT_TREE` Intent を使用した場合、以下のディレクトリには権限を付与できなくなりました。
 - 内部ストレージのルートディレクトリ
 - SD カードのルートディレクトリ
 - Download ディレクトリ
- `ACTION_OPEN_DOCUMENT_TREE` または `ACTION_OPEN_DOCUMENT` を使用した場合、以下のディレクトリ配下のファイルには権限を付与できなくなりました。
 - `Android/data/` とその配下のディレクトリ
 - `Android/obb/` とその配下のディレクトリ

- SAF 以外の方法でメディア以外のファイルにアクセス可能となる権限として、`MANAGE_EXTERNAL_STORAGE` が追加されました。
 - Play Store にこの権限を付与したアプリをリリースするには、一定の基準をクリアする必要があります。
- Media Store API を用いる場合は `WRITE_EXTERNAL_STORAGE` 権限と `WRITE_MEDIA_STORAGE` が不要となりました。
 - これらの権限はもはや Android11 で不要になったとも言えそうです

3.2.9 Android 12

<https://developer.android.com/about/versions/12/summary>

- 新機能追加が主で、既存の実装に対していますぐ修正が必要になるような変更は含まれていません。
- Media Store API の `MediaStore.Audio` テーブルに `Recordings/` ディレクトリが追加されました。
- Media Store API の `getMediaUri` メソッドにおいて `MediaDocumentsProvider` 形式の URI がサポートされるようになりました。
- `MANAGE_MEDIA` 権限が追加されました。
 - `MANAGE_EXTERNAL_STORAGE` 権限が付与されている場合に、追加の確認ダイアログを表示させないようにする権限のようです。
- `StorageManager` に `getManageSpaceActivityIntent` が追加されました。
 - 他のファイルマネージャーなどにアクティビティを公開できるようです。

3.3 Android のファイルストレージのユースケース

執筆時点での最新 OS である Android 12 を基準としています。

Android アプリで扱うファイルには主に以下の2種類があります。

(ここでいう"ファイル"には構造化 DB や Key-Value Preference のファイルは含まれません。)

- メディアファイル
 - 画像、動画、オーディオ、ダウンロードファイル
- メディア以外のファイル
 - 上記以外のすべてのファイル

- 構造化 DB や Preference のファイルは除く

3.3.1 メディアファイル

メディアファイルへのアクセスには Media Store API を利用します。

<https://developer.android.com/training/data-storage/shared/media>

ファイルの追加

MediaStore API のコレクションにファイルを追加するには、ContentResolver の `insert()` メソッドを利用します。

以下のコードは MediaStore.Audio のテーブルに新たなファイルを追加する例です。

```
// ContentResolver を取得する
val resolver = applicationContext.contentResolver

// 外部ストレージ上のコレクションを取得する
// Android 10 以降では MediaStore.VOLUME_EXTERNAL_PRIMARY からの取得が推奨される
val collection = if (Build.VERSION_CODES.Q <= Build.VERSION.SDK_INT) {
    MediaStore.Audio.Media.getContentUri(MediaStore.VOLUME_EXTERNAL_PRIMARY)
} else {
    MediaStore.Audio.Media.EXTERNAL_CONTENT_URI
}

// コレクションに反映する情報を定義する
val details = ContentValues().apply {
    put(MediaStore.Audio.Media.DISPLAY_NAME, "example.mp3")
}

// ContentResolver に新たなファイルを登録する
val uri = resolver.insert(collection, details)
```

ファイルの読み込み

MediaStore からファイルを読み込む際には、ContentResolver の `query()` メソッドを利用してファイルの URI を取得した上で InputStream を利用します。

以下のコードは MediaStore.Images のテーブルから画像を取得する例です。

```
val resolver = applicationContext.contentResolver

// ファイルのクエリ
resolver.query(
    MediaStore.Images.Media.EXTERNAL_CONTENT_URI,
    null, null, null, null
```

```
)?.use { cursor ->
    val idColumn = cursor.getColumnIndexOrThrow(MediaStore.Images.Media._ID)
    // cursor から順に情報を取得する
    while (cursor.moveToNext()) {
        val id = cursor.getLong(idColumn)
        val uri = ContentUris.withAppendedId(
            MediaStore.Images.Media.EXTERNAL_CONTENT_URI, id)
        // List に URI を追加する
        imageUris.add(uri)
    }
}
```

3.3.2 メディア以外のファイル

Storage Access Framework を利用します。

<https://developer.android.com/guide/topics/providers/document-provider>

ファイルの書き込み

ファイルを書き込むには ACTION_CREATE_DOCUMENT インテントを呼び出します。

```
const val REQUEST_CODE = 1

private fun createFile(pickerInitialUri: Uri) {
    val intent = Intent(Intent.ACTION_CREATE_DOCUMENT).apply {
        addCategory(Intent.CATEGORY_OPENABLE)
        type = "application/pdf"
        putExtra(Intent.EXTRA_TITLE, "invoice.pdf")
        // ファイルピッカーで最初に開く場所を EXTRA_INITIAL_URI で指定できる
        putExtra/DocumentsContract.EXTRA_INITIAL_URI, pickerInitialUri)
    }
    startActivityForResult(intent, REQUEST_CODE)
}
```

ファイルの読み込み

ファイルを読み込むには ACTION_OPEN_DOCUMENT インテントを呼び出します。

```
const val REQUEST_CODE = 2

fun openFile(pickerInitialUri: Uri) {
    val intent = Intent(Intent.ACTION_OPEN_DOCUMENT).apply {
        addCategory(Intent.CATEGORY_OPENABLE)
        type = "application/pdf"
        // ファイルピッカーで最初に開く場所を EXTRA_INITIAL_URI で指定できる
        putExtra/DocumentsContract.EXTRA_INITIAL_URI, pickerInitialUri)
    }
```

```

    }

    startActivityForResult(intent, REQUEST_CODE)
}

```

ディレクトリの操作

任意のディレクトリに対してアクセス権限を付与するには、ACTION_OPEN_DOCUMENT_TREE インテントを呼び出します。

```

const val REQUEST_CODE = 3

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    setContentView(R.layout.activity_main)
    prefSetting = PreferenceManager.getDefaultSharedPreferences(this)
    path_button.setOnClickListener {
        val intent = Intent(Intent.ACTION_OPEN_DOCUMENT_TREE)
        startActivityForResult(intent, REQUEST_CODE)
    }
}

override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == REQUEST_CODE && resultCode == Activity.RESULT_OK) {
        val uri = data?.data ?: return

        // 権限の永続化
        contentResolver.takePersistableUriPermission(
            uri,
            Intent.FLAG_GRANT_READ_URI_PERMISSION
                or Intent.FLAG_GRANT_WRITE_URI_PERMISSION
        )

        // ファイルの保存 or 読み出しの処理
        // もしくは URI を SharedPreferences 等に保存して他クラスで処理することも可能
    }
}

```

3.4 おわりに

本章では、Android OS のファイルストレージに関して、Android 4.4 から Android 12 までの仕様の変遷をご紹介した後、ユースケースに応じた実装について簡単にご紹介しました。ファイルストレージに関する仕様は OS のバージョンアップによる影響が大きい領域ですが、本記事が読者の皆様の一助となりましたら幸いです。

※ 本記事は、株式会社 ACCESS Advent Calendar 2021 に投稿した記事を加筆修正したものです。

3.5 参考資料

[1] データ ストレージとファイル ストレージの概要 | Android Developers

<https://developer.android.com/training/data-storage>

[2] android/storage-samples

<https://github.com/android/storage-samples>

公式のサンプルリポジトリです。

[3] [DroidKaigi 2021] メディアアクセス古今東西 / Now and Future of Media Access

<https://speakerdeck.com/e10dokup/now-and-future-of-media-access>

Media Store API でファイルを扱う方法に関してより詳しく解説されている資料です。

第 4 章

業務効率化に貢献する e-sport

4.1 はじめに

ソフトウェアエンジニアは、多くの時間をタイピングに使っていると思います。

それも、ソースコードではなく、文章を書くために。

これは意外かもしれませんが、ソースコードを書いているということはそのプログラムの仕様書や設計書を書いているわけですし、コード内にドキュメントやコメントを書いたり、コミットメッセージを書いたり、コードレビューをする/してもらうときには文章で会話をするし、その他にも色々……と考えれば自然なことです。

つまり何が言いたいかというと、**タイピングを極めれば業務効率爆上がり！みんなタイピングやろうぜ！**

4.2 タイピングの技術領域

タッチタイピングができればそれで終わり、みたいに思っていないですか？

実はその先にはとても深い沼世界が広がっているのです。

タイピングを極める上で考える要素はこんなにたくさんあります：

- キーボード・椅子・机などのハードウェア
- 姿勢と体調管理

- 基本動作 (タッチタイピング)
- キー配置・入力形式
- タイピングソフト及び各ルールへの適応
- 初速・正確率・スピード
- ワードへの最適化
- ??? (未知の領域)

前半の4つを解説しても商品紹介みたいになってしまうので、今回は後半の4つ (実質3つ) を解説していこうと思います。

4.3 用語

と、その前にいくつか用語を説明しておきます。適宜見返して読むのをオススメします。

■**打鍵** キーを押すことです。キーを押して離すまでを含むこともあります。

■**kpm** key per minute の略。「1分あたり何打鍵」という単位で、打鍵速度を表すのに使います。シフトキーとの同時押しなどを考慮して cpm (character per minute) という単位を使う場合もあるようです。

■**初速** ワードが表示されてから最初の正しい打鍵をするまでの時間です。

■**正確率** 全体の打鍵数に対する正しい打鍵数の割合です。

■**ワード** お題となる文章のことを指します。必ずしも1単語とは限りません。

■**ワードセット** 出題されるワードの集まり。大抵のソフトではワードセットが決まっています。

4.4 タイピングソフト及び各ルールへの適応

タイピングソフトやルールによって特有の要素があったりします。例えば

- 何度も挑戦してハイスコアを競うなら試行回数ゲー、一発勝負なら安定性が必要
- 漢字変換が必要だと辞書ゲーになりうる
- 出題ワードが固定だと練習した人が有利
- ワードごとにスペースを押すソフトだと慣れが必要

などなど。要は特定のソフトに慣れてる人はそのソフトでは強いということです。

自分のやりやすいソフトや練習目的に合ったソフトを選ぶと良いでしょう。

4.5 初速・正確率・スピード

これらの数値を良くするのがタイピングを練習する目標になります。1 つずつ見ていきましょう。

4.5.1 初速を上げる (短くする)

この初速 = ワードが表示されてから打鍵するまでの時間には、下記が含まれます:

- ワードが表示されてから認識するまでの時間
- ワードを認識してから読みを考える時間
- 読みから打鍵・運指を考える時間
- 実際に指を動かして打鍵するまでの時間

ワードが表示されてから認識するまでの時間は、環境 (部屋の明るさやディスプレイの明るさ等) に多少左右されるものの、他に比べれば誤差レベルでしょう。

ワードを認識してから読みを考える時間は、難しい漢字やあまり使わない言葉などがあると読めずに長くなります。ワードセットが固定のルールなら、そういった語彙を勉強しなくても**ワードを覚えるだけでここがグッと短縮**できます。さらに、1 文字ずつではなく単語・文節・文単位といった具合に**視野を広げて先読み**することで実質的に 0 にすることもできます。

読みから打鍵・運指を考える時間、ここが脱初心者のポイントだったりします。**日常での会話や文章を読むときに、どう打鍵するかを考える (あわよくばエア・タイピングする) 習慣をつけておくことで、脳内に短絡回路が形成されて短くなります。**更に深い話もあるのですが、ワードへの最適化のところで解説します。

実際に指を動かして打鍵するまでの時間は、ホームポジションを守る (あるいは工夫する) ことで少し短くできます。一般的なホームポジションでやるのが基本的にはオススメです、小指を動かすのが苦手な人や、人差し指・中指だけが異常に速く動かせる人などはそれに応じた自分のホームポジションを作っておくと良いです。

ワードが短ければ短いほど初速の重要性は高くなります。

4.5.2 正確率

あらゆる技能で言えることですが、ゆっくり正確にやることができない限り、実力がついたとは言い難いでしょう。

感覚的な説明になってしまいますが、**正確率はただ練習しているだけでは上がりません**。スピードは無視して、とにかく正確に打つことを意識して練習する必要があります。

「ミスしたら即終了」という機能を持ったタイピングソフトもあります。「何回連続でノーミス」とか「一週間正確率 95% 以上を維持」といった目標を立てて練習すると良いと思います。

正確率重視とスピード重視で一週間ずつ交互に練習している、という人もいます。

4.5.3 スピード

先述した、**視野を広げて先読み**することができるようになると平均速度が上がります。

瞬間最高速度を上げるには、同じワードをひたすら打ち続ける練習が良いです。同じ動きを続けていると脳内に短絡回路が形成されて速くなります。簡単に言うと**慣れると速くなる**ということです。

また、実際の指の動きを感じながら練習することも重要です。**頭で考えた動きと実際の動きが一致したときに速く正確に打てる**というのは当然ですが、まさにスポーツらしい要素といえるでしょう。

慣れると速くなる、ということは、逆に言えばただやってるだけで速くなるということでもあります。頭打ちになるまではとにかく練習を続けることが大事です。

そして限界に達したときこそが、「ワードへの最適化」というタイピングの世界の深淵に飛び込むときになります (もちろん先取りしても構いません)。

4.6 ワードへの最適化

タイピングをしていて打ちづらいワードを見たことはありませんか？ まずこれに気づくことが最適化の第一歩です。

例えば「あいうえお」と「あああああ」を比べてみましょう。文字数・打鍵数はどちらも同じですが、圧倒的に「あああああ」のほうが遅くなるはずです。これはつまり**同じ指で連打するのは遅く、複数の指でほぼ同時に打鍵するのが速い**ということです。

同じキーを連打するときは割とどうしようもないのですが(なので「あああああ」はクソワード)、異なるキーを同じ指で入力しているときは改善の余地があります。

例えば「き」をローマ字入力、つまり「KI」を打つとき、QWERTY 配列で一般的な打鍵をしようとするとう右手中指を連続で使う必要があります。そうではなく、中指と薬指で打鍵したり、人差し指と中指で打鍵したりすることで速くなります。

僅かな差に見えるかもしれませんが、最適化できる対象は意外と多く、これが積み重なると大きな違いになります。

例えば「で」は「き」と同様の最適化ができます。「ふ」「ん」「ちょ」「か」など、打つキーを変えるレベルで最適化できるものもあります(それぞれ FU/HU, NN/XN, CHO/TYO, KA/CA)。

もう少し視野を広げて、「キウイ」や「吐息」ではどうでしょう？「ズッキーニ」や「筆記試験」では？ おそらく、同じ「KI」の入力でも**ワードによって最速となる運指は違う**のではないかと思います。

これがワードへの最適化です。人によってやりやすい指の動きは異なるので、客観的な正解は無く、かつ多くの人の場合、非常に複雑になります。

打ちづらいワードに気づき、最適化を考えて練習し、頭と指で覚える。これを繰り返していくことで、少しずつ、結果的には大きくスピードが上がるのです。

4.7 ??? (未知の領域)

とまあ偉そうに語っていますが、筆者の私は e-typing 週間最高で 600 弱・タイプウェル 国語 R で ZJ 程度をうろついている雑魚タイパーであり、トップレベルのタイパーはその 1.3 倍ほどの速さを持っています。

きっと私も知らない未知の領域がそこには広がっているはずです。タイピングなんもわからん。

4.8 最後に

タイピングは直接業務効率に貢献するだけでなく、その考え方も業務に活かせることがわかります:

- 視野を広げて先読みする
- ただやっているだけではミスは減らない
- 慣れれば速くなる
- やりづらいポイントに気づき、改善していく

ということで、タイピングやろうぜ！

4.9 おまけ

登録不要で、ブラウザで動くオススメのタイピングソフトを紹介します:

- e-typing
 - <https://www.e-typing.ne.jp/>
 - 通称エタイ
 - 初速あり、正確率重視
 - 毎週ワードセットが変更される
 - 長文や英語、かなタイピングもあり幅広い
- タイピング速度測定
 - <https://typing.tanonews.com/>
 - 通称タイ速
 - 初速なし、速度重視
- FoxTyping

- <https://whitefox-lugh.github.io/FoxTyping/>
- 猛者タイパーが作ったタイピングソフト
- 初速なし、ワードはある程度ランダム生成
- リザルトの情報量が多く練習によい
- typeracer
 - <https://play.typeracer.com/>
 - グローバルなタイピング対戦ソフト
 - 実質初速なし、ミスタイプに修正が必要
 - 多言語対応、日本語だと変換も必要
 - オンラインで複数人とリアルタイム対戦できる

第5章

デザインにおける「センス」は誰にでも身につけられるという話

普段の仕事で様々な方とやりとりをする中で「私は全然センスがなく、デザインとかわからないんです」「デザインに興味があるんですが、センスがないから無理なんですよね」といった言葉を聞くことがあります。その度に思います。絶対にそんなことない。一部の人間離れした天才たちについてはわかりませんが、一般的なデザイン業務に必要とされる、いわゆる「センス」は誰にでも身につけられます。

なぜなら私自身が「自分には絶望的にセンスがない」と思っていた人間で、それでも現在デザイナーとして働くことができています。

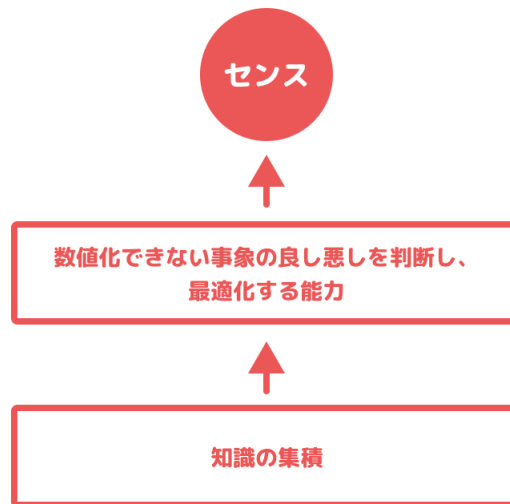
5.1 そもそも「センス」ってなんだろう？

アートディレクター・水野学さんのご著書である「センスは知識からはじまる」ではセンスとは「**数値化できない事象の良し悪しを判断し、最適化する能力**」と定義されています。そして、その能力とは「**知識の集積**」である、ということが書かれています。こんなことも言及されています。

感覚とは知識の集合体です。その書体を「美しいな」と感じる背景には、これまで僕が美しいと思ってきた、ありとあらゆるものたちがあります。（水野学 2014）

つまり、何かを美しい、素敵だ、センスが良い、などと思う感覚は何もないところから生まれたものではなく、**今までに自分が意識的・無意識的に感じてきた経験の蓄積からくる感覚**である、ということが示されています。

一見捉え所のない「センス」は、このように具体的なものとして提示することができるのです。こうするだけで、以前よりも手が届きそうに感じるのではないのでしょうか。



5.2 「センスが良いもの」を分解するとどうなる？

何かを良い、と思う感覚は今までに自分が意識的・無意識的に感じてきた経験の蓄積からくる感覚である、ということをもう少し具体的に考えてみます。

5.2.1 「センスの良さ」を掘り下げる

例として、「センスが良い」と思う Android アプリがあったとします。

なぜ「センスが良い」と思ったのでしょうか？

- 見た目がスッキリとしている？
- 色がきれい？
- 使いやすい？
- 今っぽくてカッコいい？

上記はほんの一例ですが、他にも色々な理由があるはずです。

仮に、センスが良いと思った理由が「見た目がスッキリとしている」ということだと気づけたとしましょう。さらに掘り下げてみます。

「見た目がスッキリとしている」とはどういうことでしょうか？

- 余白が適切にとられている？
- 情報設計が適切にされている？
- 色使いに一貫性があるって見やすい？
- 文字間が適切にとられており読みやすい？

などの理由に分解できると思います。もちろん、いろんな理由が組み合わさっている場合もあります。

なぜ「余白が適切にとられている」と感じたのでしょうか？

- 自分が普段使い慣れているプラットフォームのデザインガイドラインに沿った margin がとられているから
- 情報のグルーピングが margin によって適切にとられているから（同じ情報同士の margin は狭める、別の情報同士の margin は広げる、というように）

などの理由に言語化ができると思います。

5.2.2 言語化した「センスの良さ」を知識の応用で再現する

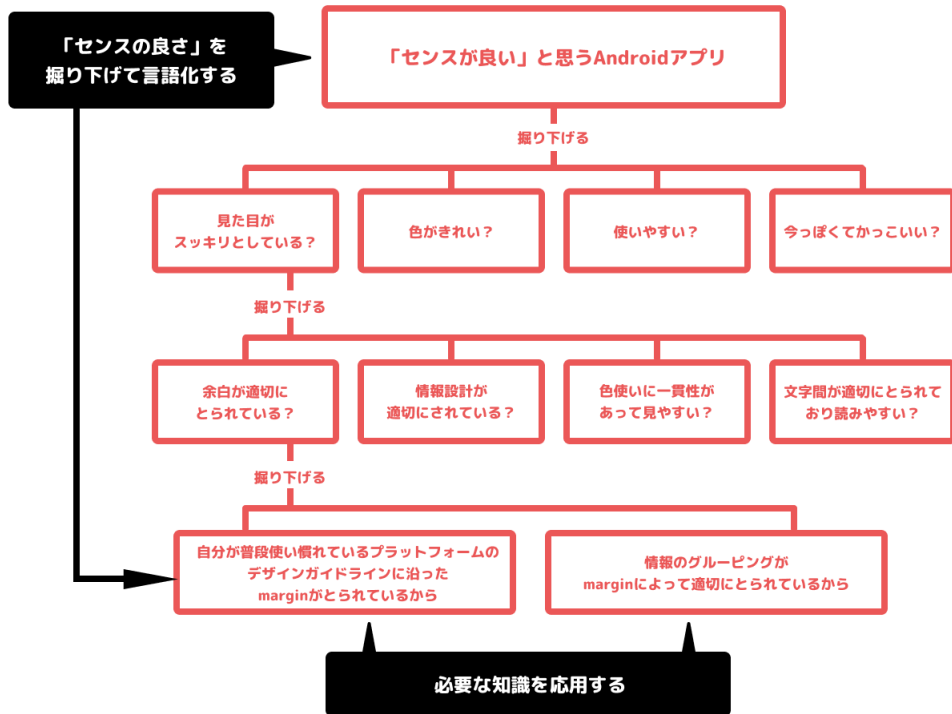
最初に「センスが良い」と感じた理由をここまで言語化できれば、逆算して自分が感じた「センスの良さ」を再現するヒントになるのではないのでしょうか。

「自分が普段使い慣れているプラットフォームのデザインガイドラインに沿った margin がとられているから」を実現するためには MATERIAL DESIGN の margin について書かれていることを応用できるかもしれません。「情報のグルーピングが margin によって適切にとられているから」を実現するにはノンデザイナーズ・デザインブックにある「近接」の項目を参考にできるかもしれません。

同じように、「情報設計が適切にされている」「色使いに一貫性があるって見やすい」「文字間が適切にとられており読みやすい」などを実現するには、それらを可能な限り具体的に言語化し、必要な知識を応用すればいいのです。

つまり、「センスが良い」と感じる部分をじっくり観察し言語化できれば、実はそれらは既出の情報の応用でできていることがわかります。

その観察・言語化・知識を使いこなせる人を総称して、「センスが良い人」と呼ばれているのだらうと思います。



5.3 センスの基である「観察・言語化・知識」はどうすれば身につくのか？

ひとことで言うと訓練です。普段から鍛えていると、目の解像度が段々と上がってくるのがわかります。最初は「なんか変だけど、具体的にどこが変なのかわからない...」と思っていたものが「ここが変だから、こう直せば良くなるだろう」とわかるようになってきます。そうなるためにやってきたことを挙げてみます。

5.3.1 ①「良い」と感じたものを掘り下げる

普段から「良いな〜!」と思ったものをなぜ良いのだろう? と思うと考える癖をつけます。「ここが良いんだな!」とわかったら、それに関連する本などを読み、知識をセットでつけるようにすると奥行きのある情報になります。アプリやデザインに限ったことでなく

「あの人の服装はいつも素敵だな～！ どこが良いんだろう？」みたいなことでも良いと思います。掘り下げて楽しいな、と思うことは自然と習慣づけられるのでおすすめです。

5.3.2 ②「らしさ」を構成する要素を分解してみる

「今っぽい」「スタイリッシュ」「親しみやすい」「信頼できる」「高級感がある」などなど、何かを見た時に生じる感覚があると思います。その「感覚」を呼び起こす要素は何なのかを考えてみましょう。色使い、文字のサイズやフォント、写真などのコンテンツ、情報量の大小、など、色々見つかると思います。また、「今っぽい」と感じるものをいくつか並べて要素を取り出し共通点を見つけることができれば、自分で「今っぽさ」を演出したい時に取り出せる知識となります。

5.3.3 ③トレースしてみる

自分が「良い」と思ったアプリをスクショして並べ、デザインソフトを使ってトレースします。いくつかのアプリをこなすと、自分が感じる「良さ」を抽象化してストックできるようになります。アプリをトレースする際は、**全体の構造化を同時に行う**ことがポイントです。世の中のアプリがどのように全体の一貫性を担保しているのか、ということを学ぶことができます。色々なものを「見る」だけでももちろん勉強になると思いますが、自分は手を動かす方が頭に入りやすいのでこのような方法をとっています。

5.3.4 ④デッサンしてみる

デッサンは一見絵を描く練習のように思えますが（もちろんその目的もありますが）、実は半分以上は「観察」の訓練です。例えばガラス瓶を描く課題があった場合は、ガラス瓶を見つめながら「**ガラス瓶をガラス瓶たらしめる要素はなんだろうか？**」とひたすら考えることになります。光のまわり込み方、影の落ち方、映り込みの入り方、そんな風に「私たちが何をもってガラス瓶と認識するか」を観察し、紙の上に表現するのです。今でも「目が狂ってきたな」と思った時はデッサン教室に行くと、目の解像度がグッと上がるのがわかって楽しいです。

5.4 センスを「使いこなす」には？

もういくつか、大事なポイントがあります。

5.4.1 センスは適材適所で使わないと意味がない

いくら「高級感がある」を上手に表現しても、それが「お手頃感がある」が求められる場面だったら成果につなげることはできません。

5.4.2 そのセンスが「客観的なものであるか」を確認する

例えば「今っぽい」が求められる場面において、自分がどれだけ「今っぽい」と思っている、他の人が「今っぽい」と思わないものであれば意味がありません。仕事の上では「自分がこう思う」と「みんながこう思う」の差を自覚しておくことは非常に重要です。自分の感覚は否定せずに大事にしつつ、「自分」「ターゲットユーザー」「チームメンバー」などの感覚差を把握しておくことでコミュニケーションをスムーズにすることができ、アウトプットの質もあげることができます。

5.5 「センス」は誰にでも身につけられる

デザイナーになりたての頃「自分はセンスがないのでデザイナーとしてやっていくのは無理だな」と思ったことがありました。そんな時に、「センス」というのは生まれつきの才能ではなく、具体的なスキルとして獲得可能である、と知ったことで本当に勇気をもらいました。同じように「自分はセンスがないかも」と悩んでいる人、「センスがあればいいのに」とモヤモヤしている人に届けば嬉しいです。

第 6 章

C#において abstract(抽象) クラス、Interface はどう使い分けるべきか

皆様初めまして。開発本部ソリューション開発第 2 部 1 課の岡田と申します。

現在は新卒一年目で、C#を使う案件に配属されたのですが、大学時代は脳筋 Python コーダーでオブジェクト指向を意識した実装というのをまったくしていませんでした。配属され色々と実装している時に abstract(抽象) クラスと Interface がなぜ必要になるのか、が最初はあまりピンときておらず、手探りで実装していました。最近はやっとわかってきた (気がする) ので、自分自身の理解を書いていきたいと思います。これを読んだ人の何かの助けとなることを望んでいます。

これを読む人の中で C#に詳しくない方も多いため、最初に簡単な解説を。その後どのような時で使うのか、どのように使い分けるのかを書いていきます。

6.1 abstract(抽象) クラスとは

抽象クラスとは、通常のクラスと異なり継承のみに使用される目的で実装されるクラスのことです。普通のクラスと違う点としては、継承のみに使用される目的のためにインスタンス化ができないという点、そして抽象メソッドを持っている点です。

抽象メソッドとは、それ自体は中身や機能を持っておらず「空」のメソッドになっています。空になっているため、派生クラスの方で機能を提供してあげないといけません。では、なぜ抽象メソッドを宣言するのか。それは、**派生先で実装してほしい部分を明確にする**ためです。

抽象クラスを継承したクラスは、すべての抽象メソッドをオーバーライドしなければいけない義務を負います。すべてをオーバーライドしなければインスタンス化ができないために、通常のクラスとして使用することができません（実装されていないメソッドがあるため、できてしまうと問題ですよ）。すべてが正しくオーバーライドできなかった場合は、派生先も抽象クラスとして用いることになってしまいます。

それでは、簡単な実装をみてみましょう。

```
// 抽象クラス
abstract class AbstractHuman
{
    public string Name { get; set; }
    public int Age { get; set; }

    public AbstractHuman(string name = "アクセス太郎", int age = 30)
    {
        this.Name = name;
        this.Age = age;
    }

    public PrintAge()
    {
        Console.WriteLine($"年齢は{this.Age}歳です");
    }

    // 抽象メソッドとして宣言
    public abstract void PrintName();
}

// 派生先のクラス
class Human: AbstractHuman
{
    public override void PrintName()
    {
        Console.WriteLine($"名前は{this.Name}です");
    }
}
```

抽象メソッドを定義するには、メソッド定義に `abstract` 修飾子を指定するだけです。抽象メソッドは、オーバーライドすることを前提としたメソッドであるために、抽象メソッドに中身を与えてしまうことがあった場合コンパイルエラーが発生してしまいます。中身がなかったとしてもブロックを記述してしまうだけでもコンパイルエラーが発生します。

抽象クラスは抽象メソッドを含んでいること、クラスに `abstract` 宣言子を指定する必要があります。しかし、他は基本的なクラスと大きな違いはありません。普通のクラスのように実装を持つことも可能です。派生先クラスでは、抽象メソッドを用いてオーバーライドする必要があるために、`override` 修飾子を追加し実装を追加してあげれば OK です。

使うだけであればそこまで難しいこともないですね。次は `Interface` についてみていき

ます。

6.2 Interface とは

Interface とは、配下のメソッドが**すべて抽象メソッド**になっているクラスのことを言います。abstract クラスでは、フィールドを持つことができますが、Interface では持つことができません。

Interface ではすべてが抽象メソッドであるため、abstract 修飾子を付与する必要はありませんし付けてはいけません。Interface の名前は、Interface であることを明らかにするために「I~」で始まる名前をつけるのが一般的です。実際の例を見てみましょう。

```
interface IHuman
{
    string Name { get; set; }
    int Age { get; set; }
    void PrintName();
    void PrintAge();
}

class Human: IHuman
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Human(string name = "アクセス太郎", int age = 30)
    {
        this.Name = name;
        this.Age = age;
    }

    public void PrintName()
    {
        Console.WriteLine($"名前は{this.Name}です");
    }

    public PrintAge()
    {
        Console.WriteLine($"年齢は{this.Age}歳です");
    }
}
```

このように記述して作成します。抽象クラスの場合は override メソッドで上書きしていましたが、Interface では必要ありません。どうでしょうか。必要な機能を Interface として宣言して明確にして、継承する。シンプルですね。

次は違いについてもう少し詳しくみていきましょう。

6.3 抽象クラス、Interface の違い

抽象クラス、Interface の簡単な実装をみてきましたが、これらの違いはどこにあるのでしょうか。抽象クラスと Interface の大きな差は 2 つあります。

- 抽象クラスは実装を持つことができる
- Interface は複数継承できる

1 つずつみていきましょう

6.3.1 抽象クラスは実装を持つことができる

抽象クラスは抽象メソッドを持つクラスであり、インスタンス化できないクラスであるため、普通のクラスのように実装を含むことができます。Interface はすべてが抽象クラスであるため、実装を持つことができません。

そのため、抽象クラスでは共通化された処理が明確に存在しているが、派生先にて実装が異なるメソッドが存在している場合に用いるのが良いでしょう。実際に抽象クラスを用いて実装されているクラスが Stream クラス (System.IO 名前空間) です。Stream クラスを継承しているのは、FileStream、MemoryStream、BufferdStream クラス等で用いられています。実際の実装を少しみてみましょう。

```
public abstract partial class Stream : MarshalByRefObject, IDisposable
{
    // 中略

    // 以下のように必要となるが使われるクラスごとに処理の内容が異なるもの
    public abstract long Seek(long offset, SeekOrigin origin);

    public abstract void SetLength(long value);

    public abstract int Read(byte[] buffer, int offset, int count);

    //中略

    // 以下のように必要となり処理も共通化できる部分は実装する
    public void CopyTo(Stream destination)
    {
        int bufferSize = GetCopyBufferSize();

        CopyTo(destination, bufferSize);
    }

    public virtual void CopyTo(Stream destination, int bufferSize)
    {
```

```
StreamHelpers.ValidateCopyToArgs(this, destination, bufferSize);

byte[] buffer = ArrayPool<byte>.Shared.Rent(bufferSize);
try
{
    int read;
    while ((read = Read(buffer, 0, buffer.Length)) != 0)
    {
        destination.Write(buffer, 0, read);
    }
}
finally
{
    ArrayPool<byte>.Shared.Return(buffer);
}

//中略
}
```

このように大量の実装を持つようなコードを Interface で表現しようとする、共通化されているべきコードを継承先で記述する必要があるために面倒です。処理が共通化されているにもかかわらず、継承のたびに同じコードを記述することはコードも読みにくくなり差分もわかりにくくなるため、こういう場合には抽象クラスを使うのが良いでしょう。

6.3.2 Interface は複数継承できる

C# においては、クラスの多重継承は許可されていません。それと比較して、Interface は多重継承を許可されています。

オブジェクト指向におけるクラス継承はとてもよくできているのですが、継承の都合上、派生クラスは親クラスのすべてを含んでいる必要があります。そのため、必ずしも派生クラスで必要としない機能に対しても**必ず**オーバーライドしないといけません。必要のない機能を実装することは、コードが冗長になり派生クラスの役割がわかりにくくなるため望ましい状態ではありません。

実際の例をみてみましょう。

```
interface IHoge1
{
    void Hoge1();
}

interface IHoge2
{
    void Hoge2();
}
```

```
class TestClass: IHoge1, IHoge2
{
    public void Hoge1()
    {
        Console.WriteLine("Hoge1")
    }

    public void Hoge2()
    {
        Console.WriteLine("Hoge2")
    }
}
```

このように複数の Interface を継承できます。これにより、機能が複数あるような場合に使うことで多くの機能実装を明確にした上で実装を安心して行うことができます。Interface を実装することで必要な実装は明確になっているため、実装忘れを防いだ上で実装ができます。すでに実装した Interface を利用することもでき、多くのクラスで用いられているが追加で実装が必要な部分が出た場合に新規の Interface を実装することで柔軟に対応することもできます。

6.4 抽象クラスは必要なのか

ここで、ひとつ疑問が浮かびます。それは、Interface が多重継承可能なのであれば抽象クラスをわざわざ実装する必要がないのではないかということです。共通の実装は通常のクラスで宣言しておき、追加で Interface を実装して継承すれば良いのではないかということです。

実際の例を見てみましょう。

```
interface IMetaHuman
{
    void PrintName();
}

class Human
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Human(string name = "アクセス太郎", int age = 30)
    {
        this.Name = name;
        this.Age = age;
    }
}

class MetaHuman : Human, IMetaHuman
```

```
{
    public void PrintName()
    {
        Console.WriteLine($"名前は{this.Name}です");
    }
}
```

こういうふうになれば、継承先で実装が必要な部分は Interface として宣言することで抽象クラスと同様のことを実現できます。

正直なところ、抽象クラスを用いることは少ないです。実際に私が携わっている開発においても Interface をメインに用いて開発を行なっています。しかし、Interface がたくさん存在してくると継承するものが増えてきてしまい、どれを継承すべきなのかわからなくなってくる場合もあります。そのため、Stream クラスのように共通化部分が大きい実装の場合は用いても良いでしょう。

Interface を用いる利点として、テストコード実装の際に関数を mock する場合は Interface を介して実装を行うため Interface で実装していると何かと便利です。ひとつの指針として困った場合は Interface で実装するで問題ないと思います。

6.5 まとめ

今回は、Interface と abstract クラスの使いわけについて書いてみました。自分は最初に実装をしている時に、抽象クラスいらなくねとなり色々調べた結果をまとめてみました。こういうことを気にする人がいるのかわかりませんが、自分自身の考えや理解を書くのは意外と楽しいですね。また機会があれば書いてみたいと思います。

第7章

プロジェクトマネージャがいきいきと躍動する働きやすい環境を作る

本書を手にとってくださりありがとうございます！ この本のこのページまでたどりついたあなたは、ここにしかない面白い技術を貪欲に探し求めているか、あるいはタイトルにある「プロジェクトマネージャ」「働きやすい」といった単語に反応したかのどちらかでしょうか。面白い技術をお探しの方はぜひとも引き続き当社の自慢のエンジニア達が筆を奮った他の章や冊子をお楽しみください。タイトルが気になったという方、ようこそいらっしゃいました。きっとプロジェクトマネージャが働く環境に課題感をお持ちのことでしょう。私たちも同じです。プロジェクトマネージャがいきいきと躍動する働きやすい環境をどうやって作るか一緒に考え、共有していこうではありませんか。

さて、ここで当社と本章執筆者について簡単に紹介します。当社、株式会社 ACCESS は技術の会社と言っても過言ではありません。ビジョンとして“「技術」「知恵」「創造性」と「勇気」で世界を革新し続ける独立系、企画・研究型企业”をうたい、真っ先に「技術」を掲げたうえに、「技術」という用語を“ACCESS の価値の真髄・誇り”と説明するほどなのです。実際に社内では技術力を高めることが奨励されており、技術力の高いエンジニアは社内外から尊敬を集めています。一方で私自身はといえば、2005 年に新卒入社し開発部門に配属されたものの技術がさっぱり分からない落ちこぼれでした。それでも、プロジェクトマネジメントっぽいことや“名前のない仕事”のようなものをする事で何とか他の人とは違う価値を出しここまでやってきました。上司や同僚、環境に恵まれ続けたおかげとしか言いようがありません。

落ちこぼれの私が周りの助けを得て生き延びられた一方で、どんなに優秀な人でも周りからの助けが全くなかったら仕事は困難で険しいものになると思います。当社ではプロジェクトマネージャは基本的に各プロジェクトに一人なので、そのプロジェクト固有の悩

第7章 プロジェクトマネージャがいきいきと躍動する働きやすい環境を作る

みを一人で抱え込んでしまうことがあります。もちろん、チームメンバーや上司が助けてくれますが、いかんせん役割が違うため適切なアドバイスができないこともあります。そのため、プロジェクトマネージャが個別に工夫をこらしており、職人芸のような領域に達していることもあれば、周りの期待に応えきれず苦しい思いをしていることもあります。

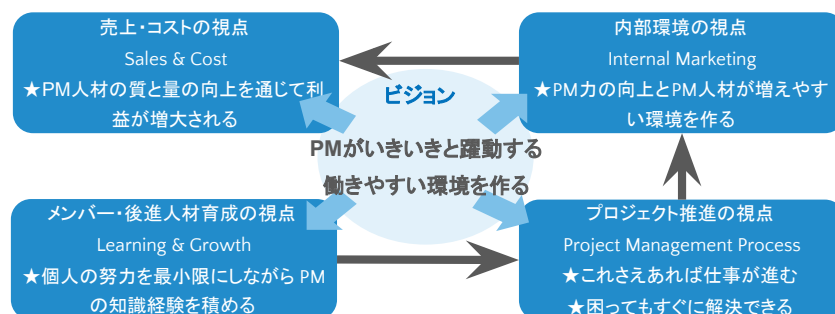
そこで、社内プロジェクトマネージャ達が知見を共有し、助け合い、共に成長するためのグループ「PM Specialist」が作られました。「PM」とは、プロジェクトマネージャの略称であり、プロジェクトを管理・運営する役割を担っています。現役のプロジェクトマネージャだけでなく候補者や退役者、品質管理部門メンバーなど様々な立場の人が自らの意思でグループに入っています。2019年に発足し、メンバーの入れ替わりを経ながら、2021年初めには何週間もかけてグループの目的を改めて議論しました。参加メンバーによってやりたいことや目指す姿がバラバラだったことが分かり、どうまとめればよいか途方に暮れかけましたが、メンバーの一人である熟練者の助けを得てビジョンとして方向づけることができました。

そのビジョンがこちらです。

「PMがいきいきと躍動する働きやすい環境を作る」（図7.1 参照）

PM Specialistのビジョン

■ PMがいきいきと躍動する働きやすい環境を作る

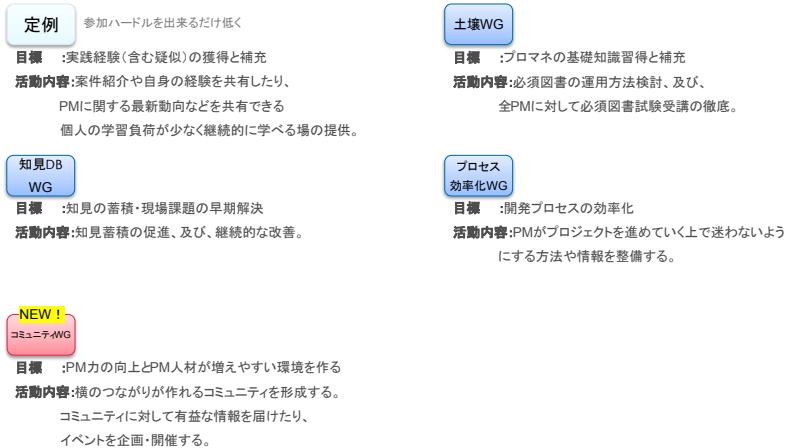


▲図 7.1 PM Specialist のビジョン

このビジョンに基づき PM Specialist の活動を整理し、目的別に 4 つのワーキンググループを構成しました（図 7.2 参照）。各自が希望するワーキンググループに入り、どこに

も所属していないメンバーもいれば複数かけもちしているメンバーもあります。プロジェクトマネージャは多忙なことが多いので、本人ができる時にできる範囲で活動しています。

38期 WGの構成



▲図 7.2 ワーキンググループの構成

これまでに取り組んできたこととして図 7.3 のようなものがあります。

第7章 プロジェクトマネージャがいきいきと躍動する働きやすい環境を作る

取り組み	概要・工夫した点
スターティングガイド (土壌WG)	はじめてプロジェクトマネージャになった方が知っておくべき最低限の情報をまとめたものです。前提知識がなくとも理解できるように用語の解説や手順を示したスクリーンショットを掲載しています。
ACCESS KEEP (知見DB WG)	プロジェクトマネージャ同士が気軽に質問したり、自身が持つ知見の紹介・共有をしたりすることにより、ノウハウを蓄積するデータベース兼フォーラムです。毎月その月の投稿内容を配信し認知度向上に努めています。
必須・推薦図書& 推薦PM資格試験 (土壌WG)	プロジェクトマネージャにおすすめする書籍&資格試験を紹介しています。チーム開発に対する理解度を自己チェックできるテストも作成しました。
PM Newsletter (コミュニティWG)	社内外のプロジェクトマネジメントに関する情報やPM Specialistの活動を毎月全社に向けてNewsletter形式で発信しています。これまでに見積もり手法の紹介やインタビュー記事等を掲載してきました。
PMトーク!! (PM Specialist全体)	毎週持ち回りの発表会をPM Specialist全体に向けて開催しています。発表者の担当プロジェクトや経験から広く学ぶ機会となっています。10名前後の参加者に加え、PM Specialist外の社員も毎回何名か視聴している密かな人気番組です。
PM Meet-up!! (コミュニティWG)	社内のプロジェクトマネージャ全員が一堂に会し、プロジェクトマネジメントに対する課題を共有したり解決策を検討したりするイベントを半期に一度開催しています。昨今のリモートワーク環境により他部門との交流が減っているため、貴重な情報交換の機会となっています。

▲図 7.3 PM Specialist の取り組み事例

このような活動を積極的に進めていますが、当社のプロジェクトマネージャ全員がいきいきと躍動する働きやすいと感じているかという点必ずしもそうではないと思います。複雑な利害関係にあるステークホルダー間の調整や、厳しい納期の調整、エンジニアとのコミュニケーション、社内の諸々の手続きなど、大小さまざまなタスクに埋もれ悩んでいるプロジェクトマネージャもいることと思います。ただ確実に言えるのは、その人を放っておくのではなく助け合いたいと思っている仲間がいて、いつでもドアが開かれているということです。

他の会社ではプロジェクトマネージャが働きやすい環境をどのように作っているのか知りたいです。何か機会がありましたらぜひ教えてください。

著者紹介

第 1 章 Vitantonio Nagauzzi / @tonionagauzzi

Mobile App Engineer, "You decide you're happy or not."

第 2 章 Taro Nonoyama / @n2-freevas or @n2_freevas

Web Engineer / Artist / DJ

第 3 章 Soichi Ikebe / @aqua_ix

Mobile App / Web / xR Engineer

第 4 章 SekiT / <https://github.com/SekiT>

Web server/frontend Engineer

第 5 章 Mika Numata / @numatami

UI / Product / Graphic Designer

第 6 章 Kazushi Okada / @kokada420

Mobile App Engineer / ML Lover

第 7 章 Chinatsu Iwasa (岩佐千夏)

2005 年 4 月新卒入社し開発部門へ。2018 年 12 月内部監査室へ異動。

ACCESS テックブック 2

2022 年 1 月 22 日 ACCESS テックブック 2 v1.0.0

著 者 ACCESS 技術書典同好会
デザイン numatami
編 集 tonionagauzzi、krmtmint（くろみつ）
発行所 ACCESS Co., Ltd.

(C) 2022 ACCESS Co., Ltd.