仕事でバックエンド開発するときに考えていること

技育祭2022 勉強会

自己紹介

- 名前: 鈴木 進也
  - yanyanと呼ばれています
- 新卒2年目
- 趣味
  - valorant
  - FF14 (最近始めました)
  - キーボードで散財

自己紹介

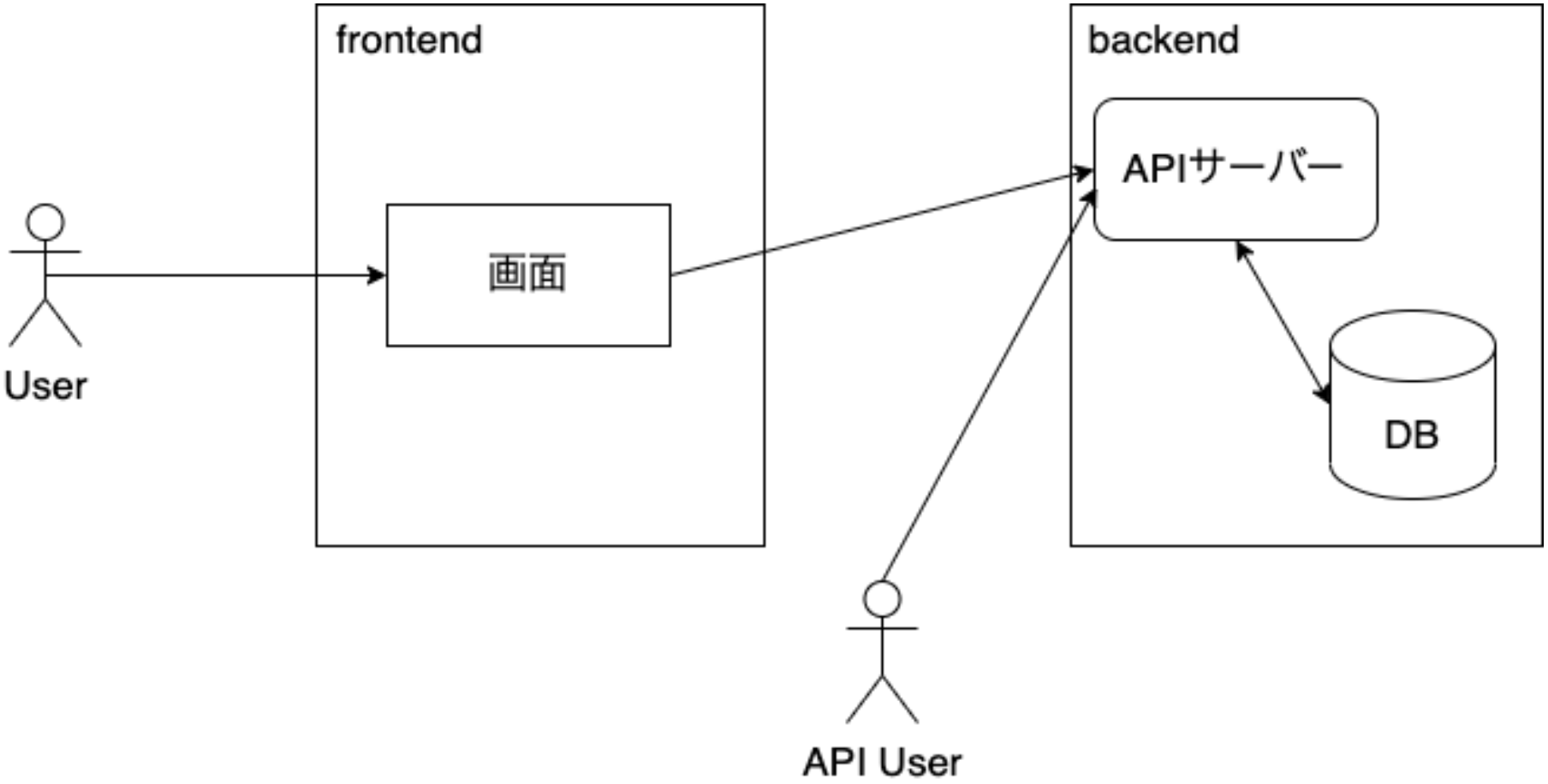
- 株式会社CARTA HOLDINGS
  - 株式会社fluct 開発本部
  - ネット広告の配信、運用支援などをやっている会社
- GoでAPIサーバーを書いたり、データエンジニアリングをしています

今日の資料、サンプルコードはGithubに置いてあります

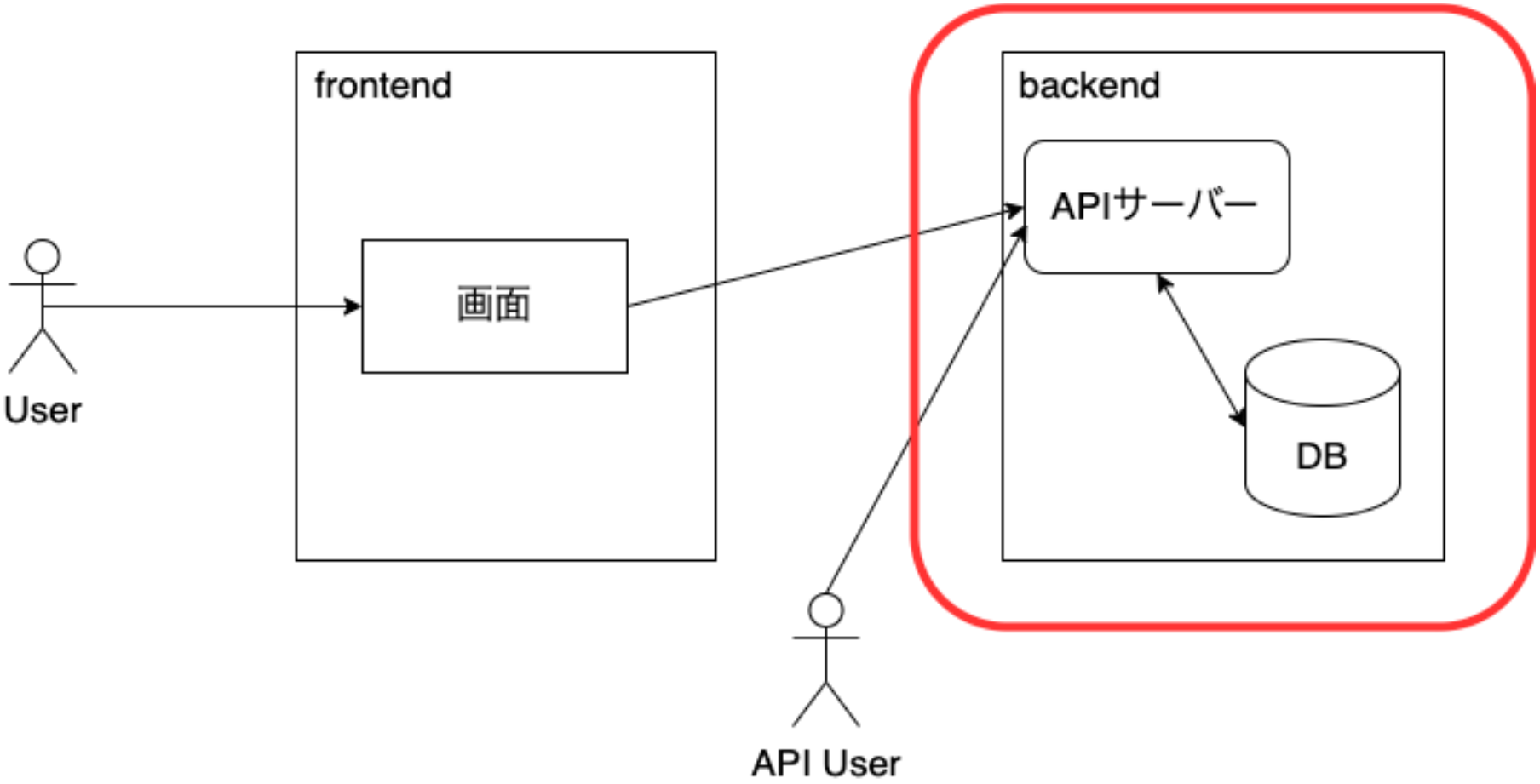
https://github.com/shinya-ml/geeksai-backend-study

今日話すこと

ざっくりとしたWebアプリケーションの構成



ざっくりとしたWebアプリケーションの構成



この部分を作るときに考えていることを話します

お題目

他にも考えることは色々あるが、今回は以下のことについて考える

- 認知負荷の話
- バックエンドアプリケーションのアーキテクチャの話
- API設計について
- テストの話
- 思想を言語化する

認知負荷の話

認知負荷とは

人が学習する際にかかる記憶領域に対する負荷

- 開発には様々な認知負荷がかかる (コードの意図や、アーキテクチャの理解etc...)
- A Philosophy of Software Design では、ソフトウェアの複雑性が増大している兆候の一つとしてあげられている
- 自分は普段の開発で認知負荷が高くなりすぎていないか？をよく気にしている

なぜ認知負荷を気にしているのか

- 理解が不十分なままコードの修正や書き足しをすると、**より複雑度が高まってしまう**
  - 認知負荷の高いコードは、さらなる認知負荷の上昇をもたらす
- 規模が大きくなるにつれて複雑性の増加は避けられない
  - 工夫して複雑になりすぎないようにすることはできる

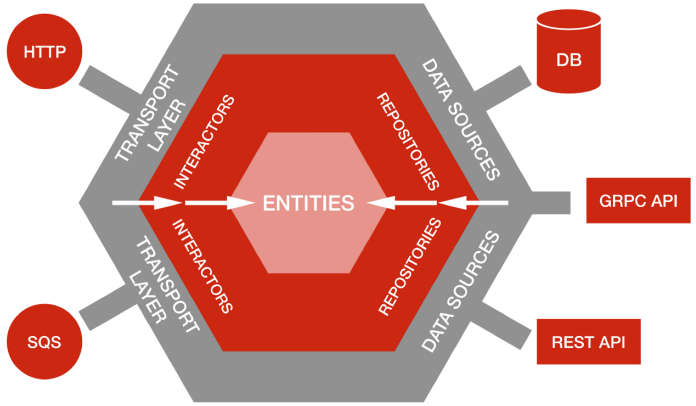
アーキテクチャの話

ここでいうアーキテクチャとは

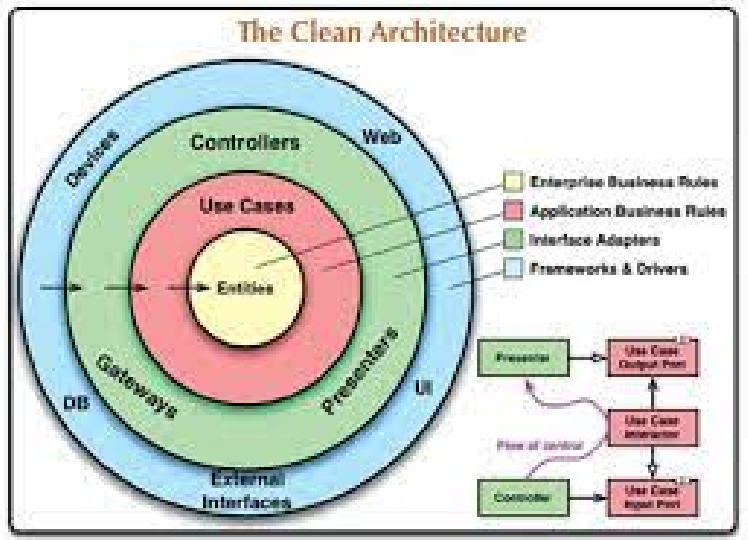
- アプリケーションの実装をレイヤーごとに分けて整理する
- レイヤーに分けることによって以下のことが達成できる
  - 関心事の分離
  - 依存関係の整理

1から作るバックエンドアプリケーションのレイヤー構造をどうやって考えていくか

よく目にするアーキテクチャたち



- レイヤードアーキテクチャ
- ヘキサゴナルアーキテクチャ
- オニオンアーキテクチャ
- クリーンアーキテクチャ
- etc...



[画像の出典] Ready for changes with Hexagonal Architecture, Clean Architecture

彼らは銀の弾丸ではない

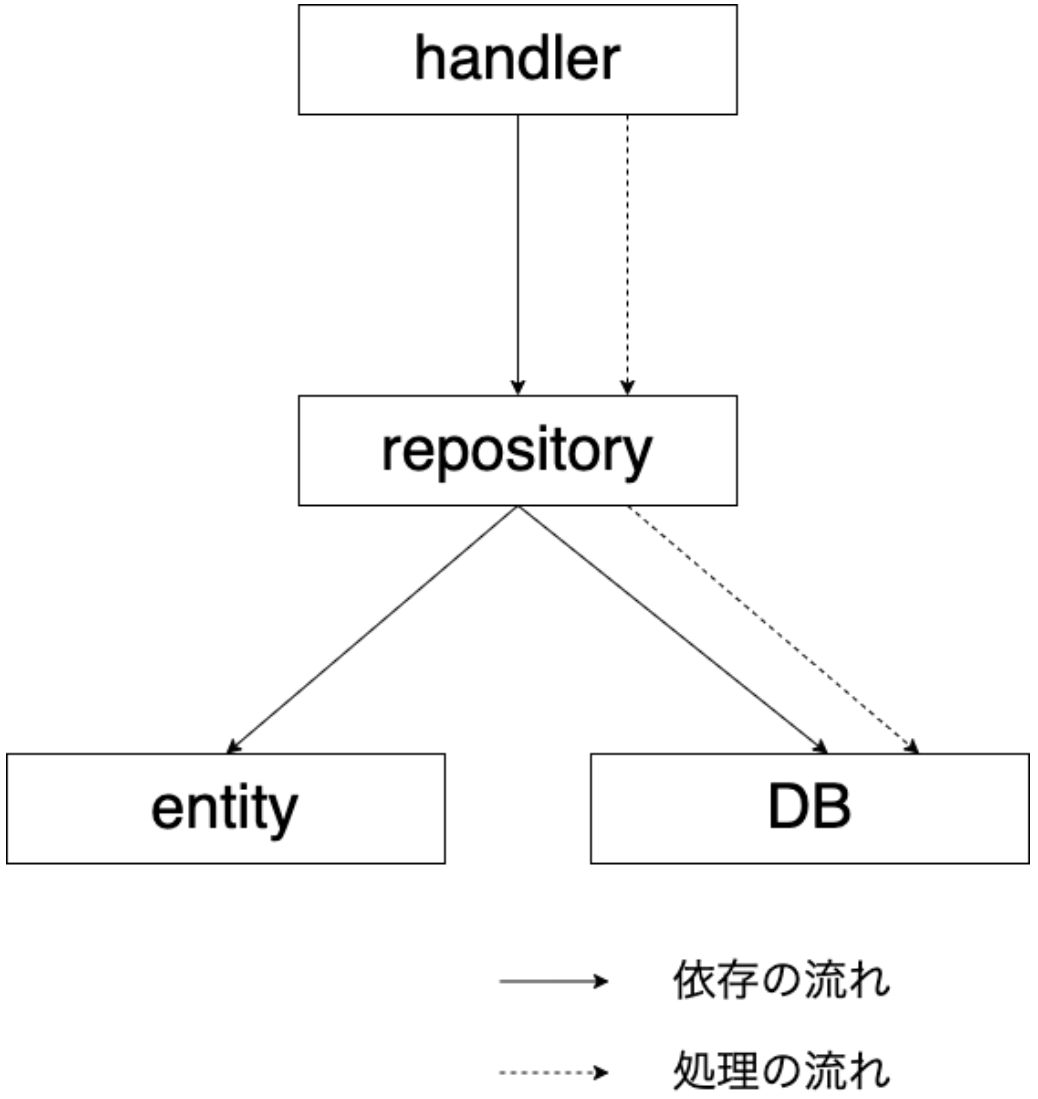
いかなるアプリケーションでも、このアーキテクチャを適用しとけばよいというわけではない

必要なときに必要な変更をする

- 良いとされるアーキテクチャは、開発が進むにつれて変わっていくもの
- アプリケーションの規模が小さい段階から、壮大なアーキテクチャにしようとすると大体つらい
  - ほとんどなにもしてないレイヤーが生まれる
  - 意味のない抽象化 (具体的な実装が1個しかないとか)
  - なぜそのレイヤーが存在しているのかわからない = 認知負荷が高い
- 自分たちにとって大事な考えを守りつつ、**必要に応じて**層を足したり抽象化をすればよい

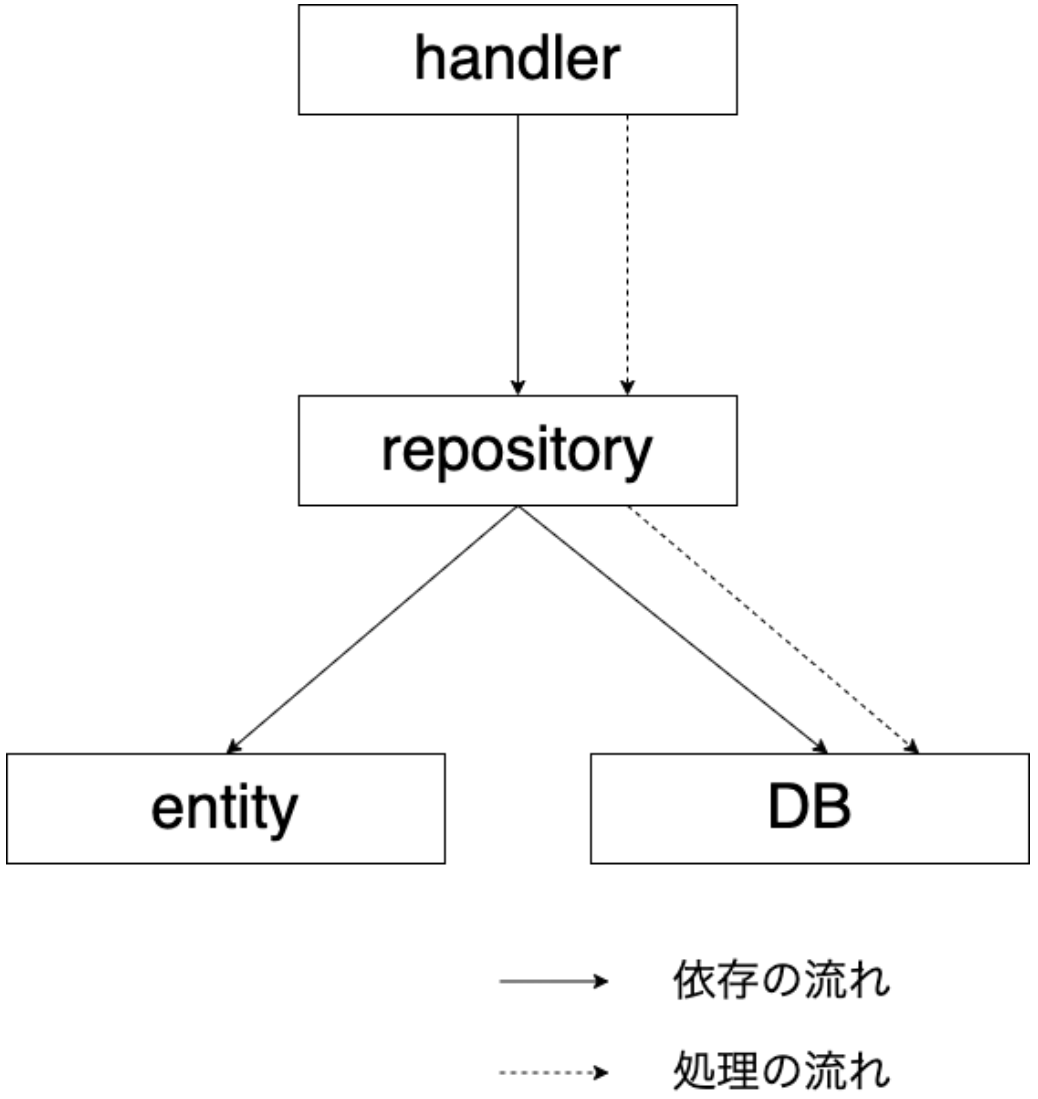
例えば

- **handler**: HTTP リクエストを受け取ってレスポンスを返すマン
- **repository**: DBとやりとりするマン
- **entity**: サービスが扱うオブジェクトを定義するマン



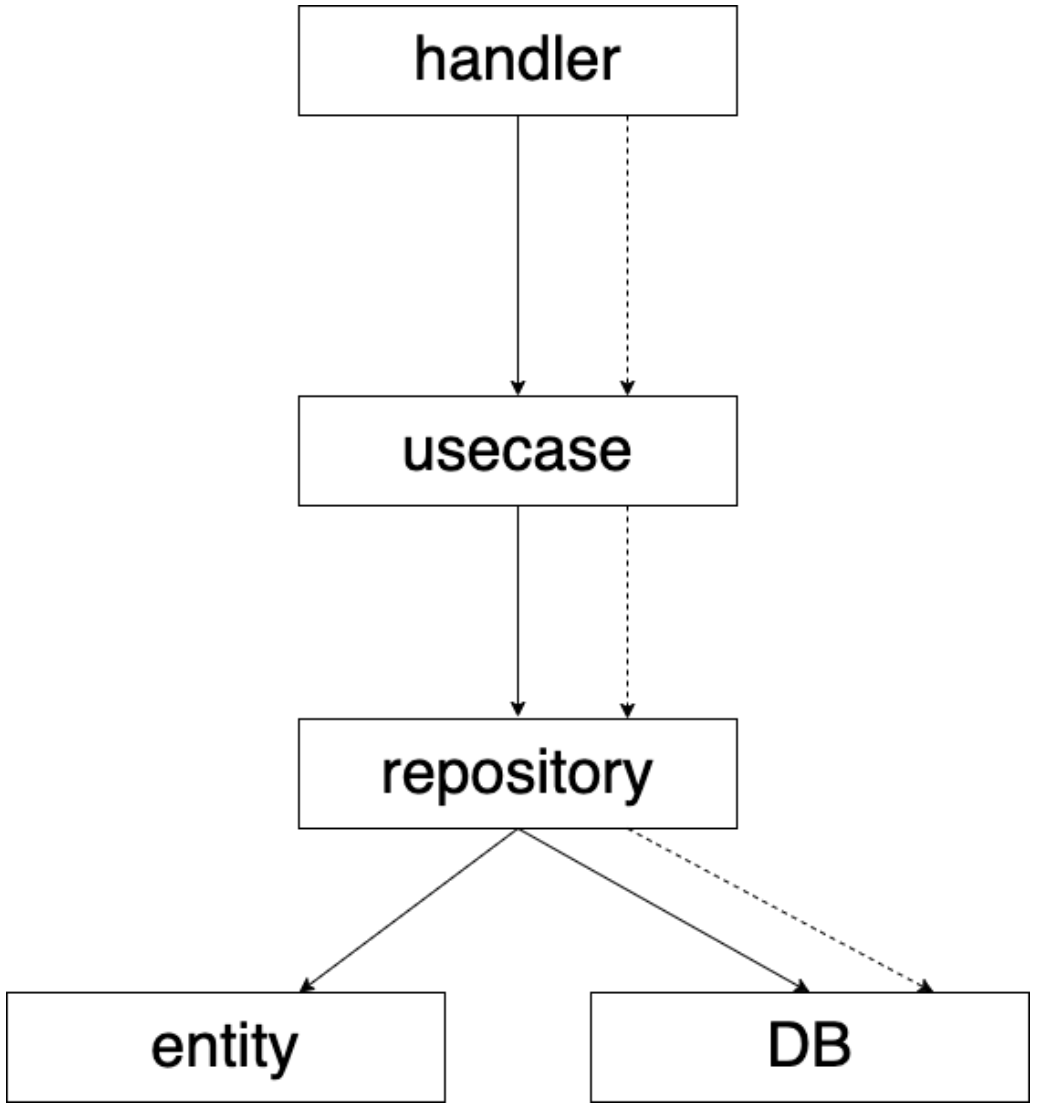
例えば

- 単に来たリクエストに応じてCRUDするだけならこれくらい素朴でもいい
- 開発したいことに応じてアーキテクチャも変化させていく



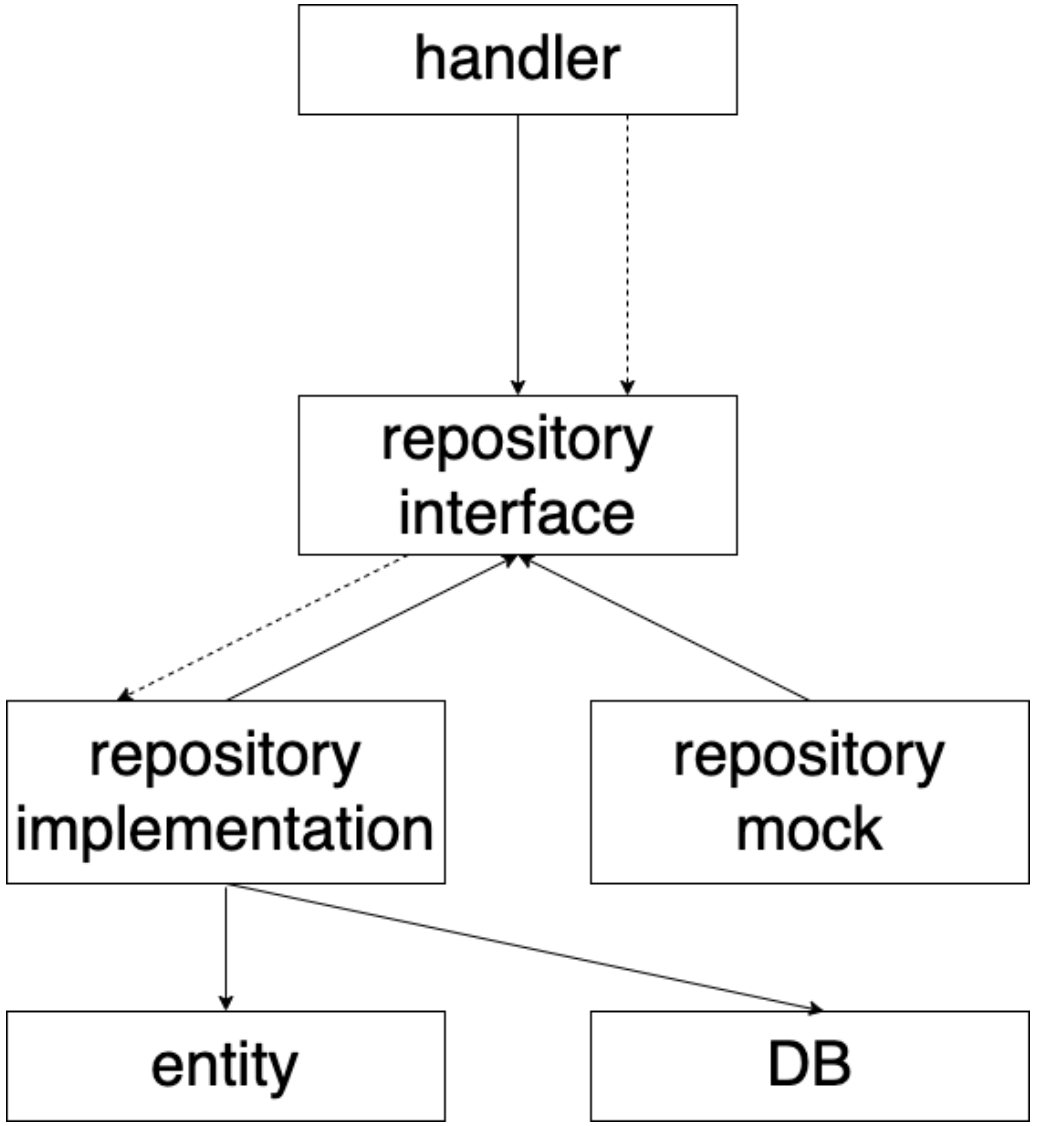
扱う関心事が増えた

- ビジネスロジックを書く層がほしい！
    - あとから足せば良い



抽象化したい

- repositoryに依存する層のユニットテストをしたい！
  - repositoryの部分はフェイクに差し替えたい
- インターフェースに依存する形にする
  - 具象が１個だけなら抽象化する必要もない



大事な考え

私がアーキテクチャの構造を考えるときに守りたいこと

1. 関心事の分離
2. 依存の流れを1方向にする

これらを守りながら、その時々でベストな設計を模索する

何を大事にするのかは人とか開発するサービスの特性によって変わってくる

関心事の分離

- 関心事とは
  - 働きかける対象
  - e.g.) DBとのやりとり、HTTP req/resについてetc...

関心事の分離

- まずは存在する関心事を言語化することが大事
- 1レイヤーが複数の関心事を扱わないようにする
    - e.g.) ファットコントローラー

サンプルコード を見てみよう

各層が１つの関心事しか扱わないとどう嬉しい？

- 認知負荷が低い
  - 触りたい実装がどこにあるかが把握しやすい
  - e.g.) DB周りはrepository層をみればおｋ
- 変更しやすい
  - 変更するためにいじらなければならない箇所が明確になる
- 壊れたときに直しやすい
  - 壊れた原因が特定しやすい

依存関係

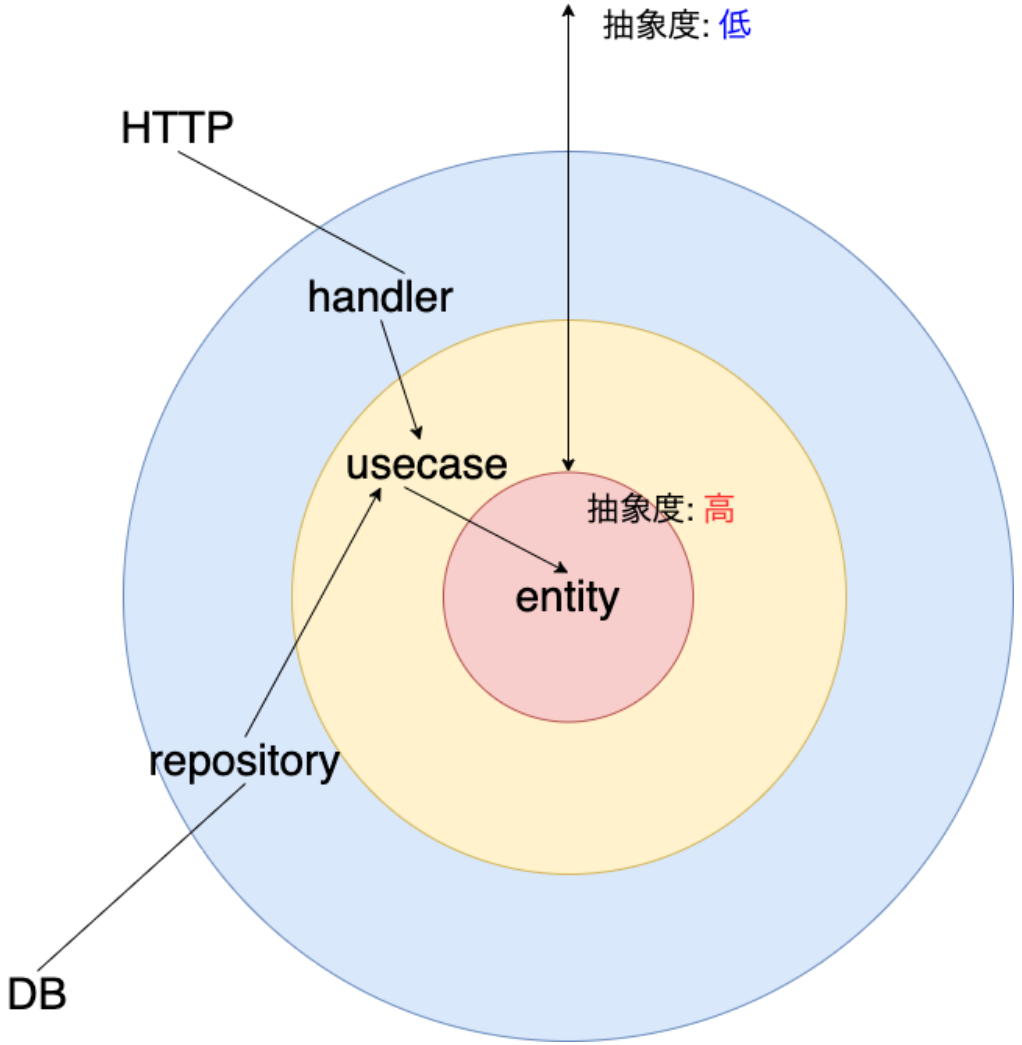
- レイヤー構造を成すので、レイヤー間に依存関係が生まれる
- 依存とは
  - 依存される側の知識が依存する側に漏れ出ている状態
  - メソッドの呼び出しに必要な引数とか
- 依存される側に変更が入ると、する側も影響を受ける
  - あるモジュールが依存したりされたりしまくっている (密結合) と辛い

依存の流れを1方向にする

- 依存の流れを交通整理する
- 具体的な関心事をもつレイヤー -> 抽象的な関心事を持つレイヤーという依存の流れを守る

具象 -> 抽象へと依存させる

- 円の外側は具体的な技術的関心事
- 内側は抽象的なビジネスのコアを成す関心事
- 外 -> 内という向きで依存させる



抽象が具象に依存するとどうなる？

つまり、具体的な知識が内側のビジネスロジックやオブジェクトに漏れ出る

- e.g.) DBの知識がusecase層で必要になる

具体的な関心事の知識が漏洩すると...

- 円の外側に対する変更で内側も影響を受ける
- 技術の差し替えが難しくなる
    - REST -> GraphQLに移行したいとかが辛い

まとめ

- アプリケーションにレイヤー構造を設けることで依存関係が整理される
- 関心ごとを適切に分けよう
- 守りたいルールは遵守しながら、サービスの成長に合わせてアーキテクチャは変化させていこう

API設計について

APIスタイル

API設計の際に選択肢として出てくるやつら

- REST
  - リソースベースのURI
  - JSON形式でデータをやりとりする
  - 長いこと使われてきてる
- gRPC
  - Protobuf形式でデータをやりとりする
  - マイクロサービス間の通信とかで使われている
- GraphQL
  - クエリ言語＋クエリに対するサーバーサイド実装
  - 最近使われ始めている

大前提

- ユースケースに応じて使い分けよう
  - 銀の弾丸などない
- GraphQLはRESTの上位互換であるとか、そんなことはない
  - RESTを使ったほうがいい場合もある

どういう軸で考えるのか

- APIの利用者
  - 誰が使うんだっけ
  - どのくらい使われるんだっけ
- ユースケースの数
  - 多様な利用者がいてユースケースも様々なんだよねーとか
- サービス的になにを重要視するか
  - APIとしての柔軟性？
  - パフォーマンス？ etc...

ざっくりとした私の所感

- REST
  - リソースベースでエンドポイントを記述するので、1つのAPIでいろんなユースケースに対応しようとすると辛くなりがち
  - 1APIのユースケースが単純ならわかりやすい
- GraphQL
  - クエリによって利用者側が柔軟に欲しいデータを記述できるのでユースケースが多様な場合にいい
  - クエリの形式と返ってくるデータの形式がほぼ一緒なので直感的
- gRPC
  - パフォーマンス重視ならこれかなー
  - 内部向けのAPIとかなら、型もかっちり書けるしいい

余談: 仕事でGraphQLを使っています

- 顧客向けWebアプリケーションの開発でGraphQLを採用した
- バックエンドの実装はGoでgqlgenというライブラリを利用している
  - スキーマ定義からリゾルバーのメソッドやモデルの構造体を生成してくれる
  - ブラウザ上でGraphQLのクエリが叩けるプレイグラウンド環境の用意もいいかんじにしてくれる
- スキーマ設計やロギング、ドキュメンテーションなど、これからやっていきなことはたくさんある

良いAPIとは？

正しい使い方をするのが簡単で、間違った使い方をするのが難しい

- APIを使う側のことを考えて設計する
- 適切にドキュメンテーションをする
- 命名の一貫性
- レスポンスの設計

例えば

APIスタイルによって気をつけたいことも変わってくる

REST, gRPCなら...

- エンドポイントのURIはわかりやすくなっているか
- クエリパラメータやリクエストボディの設計etc...

GraphQLなら...

- スキーマ設計
  - 命名の一貫性やわかりやすさ
  - nullが妥当に使えているか
- Production Ready GraphQLという本がおすすめ

テストの話

バックエンドにおけるテストは色々ある

- ユニットテスト
  - モジュール単体のテスト
- インテグレーションテスト
  - 複数のモジュールを跨いだテスト
  - repository - DB間のテストのような、アプリケーションの外側とのテストも含む

Q.どのテストを書く？

A.全部書けばええやん

~~A.全部書けばええやん~~

なぜテストを書きたい?

- リリース前にバグに気づく
- 変更することに対する安全性、容易性
- テスト対象のコードの理解を助ける
- etc...

つまり、開発における様々な不安を取り除く

どこにテストを書きたい？

テストを書くことによって不安を取り除きたい箇所

どこにテストを書きたい

テストを書くことによって不安を取り除きたい箇所

- リリース後に壊れるとサービス的に致命的な箇所
  - お金が絡んだりして、後から直すのが辛いとか
- サービス的に大事なロジックが書かれている
  - ビジネスロジックとか

テストを書かないという選択

特段不安がないとか、テストのコスパ悪そうだな〜って思った箇所には私はテストを書かない

- テストコードにもメンテナンスコストはかかる
- 自動テストにかかる時間が長くなると人々はテストしなくなる

-> テストしたいところだけテストする

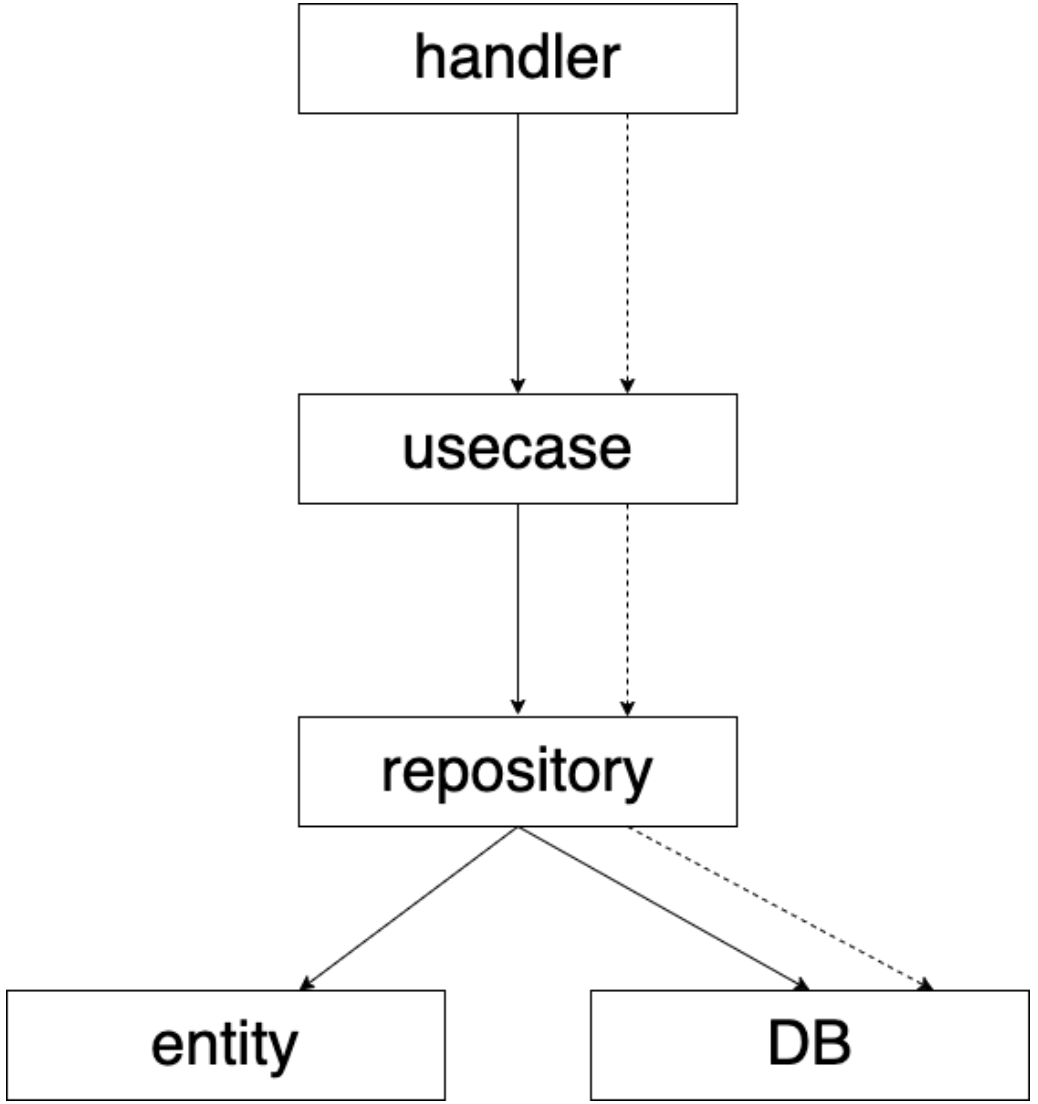
例

こういうレイヤー構造で以下のことを考えてみる

- なんのテストを書きたいか
- なんのテストは書かないか

各レイヤーの関心事

- handler: HTTP req/res
- usecase: ビジネスロジック
- repository: DBとのやりとり
- entity: ビジネスオブジェクト

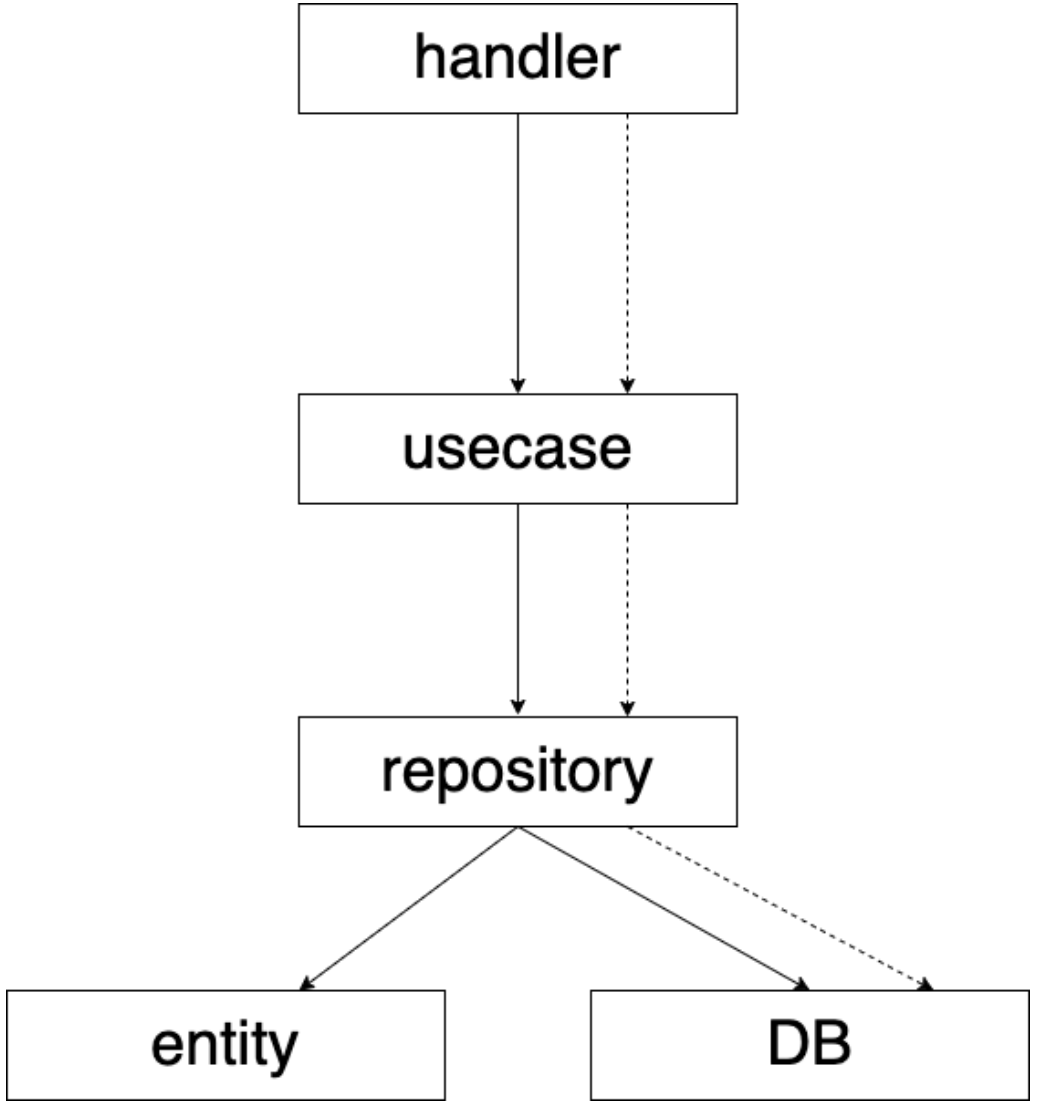


なんのテストを書きたいか

あくまで例で、サービスの特徴によって変わる

単純な構造なので書きたいテストはそんなに多くない

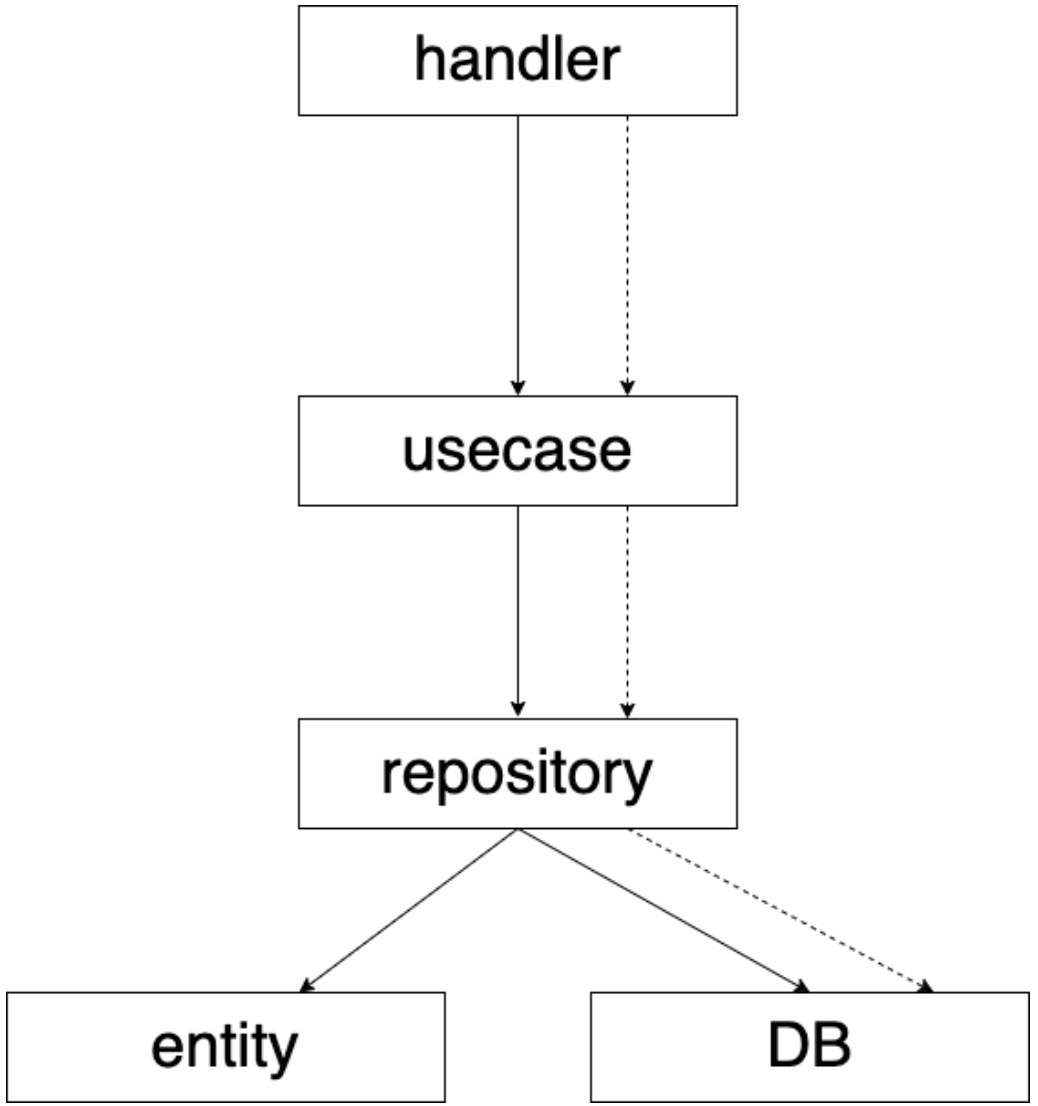
- usecase層のユニットテスト
- (ロジックがあれば) entity層のユニットテスト
- handler ~ repository まで一気通貫のインテグレーションテスト



なんのテストを書かないか

あくまで例で、サービスの特(ry

- handlerのユニットテスト
- repositoryのユニットテスト
- repository - DB 間のインテグレーションテスト



handlerのユニットテスト

HTTP request/responseが関心事

- それ以外の殆どの処理は他の層に委譲している
- つまり、ほとんどロジックがない薄い層 -> テストしたいことがない

この層にテストしたくなるようなロジックがいたら、関心事の分離がうまくできていないかもしれない

テストは意図が大事

- なぜテストを書き、何をテストしたいのか
- 意図がわからないテストは、後々辛い
  - プロダクションコードの変更でテストがコケたとき、直しづらい
  - そのテストがなぜ存在しているのかわからないとメンテもされない

本当にテストしたいところにテストを書こう

思想を言語化しよう

Whyはコードを読んでもわからない

- なぜこのアーキテクチャにした?
- なぜこの言語を選んだ？
- なぜGraphQLを選んだ？

こうしたWhyに対する答えは、意図的に言語化しないと残らない

Whyを言語化しておくことはなぜ大事なのか

- 理解の助けになる
- 後から反省する材料になる
- アーキテクチャやテストに手を入れる際、既存のものの意図を知ることは大事
    - すでにあるものがなぜこうなっているかを知った上で、どう変化させていくかを考える

Design Doc

システムを作り始める前に書く地図のようなもの

- これから作るシステムが目指すゴール
- どういう設計で作るのか
- システムがスコープとしないこと、やらないと決めたこと

などを書く

- システムを作るにあたって必要な意思決定が言語化される
- ここに、意思決定に至ったWhyも書く

Design Docを見ればシステムの目指すゴール、意思決定のwhyが分かる状態にする

Architecture Decision Record (ADR)

特定の意思決定に関することを記述する

- 背景
- なぜこの意思決定をしたのか
- 他にどんな選択肢があったのか

作り始めてから行われる変化の意思決定はADRで言語化するとわかりやすい

おわりに

- 認知負荷を意識して開発する
- アーキテクチャもテストも必要だと思ったことをやればよい
- なぜやる (やらない) のかが大事
- コードでは伝わらないことは、積極的に言語化していこう