

複雑なJavaScriptアプリケーション  
を考えながら作る話

# 自己紹介

- Name : **azu**
- Twitter : @azu\_re
- Website: [Web scratch](#), [JSer.info](#)



#jsprimerを書いています

JavaScript入門書に興味ある人はウォッチ 

## ⚠ 注意 ⚠

- 作成するアプリケーションによって必要な構造は異なります
- 今回の話はある程度の規模で複雑性を持つクライアントサイド
  - ライブラリ抜きで数万LOC >=
- 長期的にメンテナンスや変更が発生するアプリケーション
- サーバサイドレンダリングはしないクライアントアプリケーション

# 3行でOK

- 複雑なJavaScriptアプリケーションを作るにあたりドメインモデルをどう実装するか悩んだ
- 色々試行錯誤した結果としてAlmin.jsを作った
- 半年ぐらい議論しながら開発してできたガイドライン、React Component実装ガイド、CSSの実装ガイドとかの参考資料はここに置いてある
- [github.com/azu/large-scale-javascript](https://github.com/azu/large-scale-javascript)

# 対象

- Flux(util)やReduxを使って何か作った事がある人
- Flux実装を書いたことがある人

# 目的

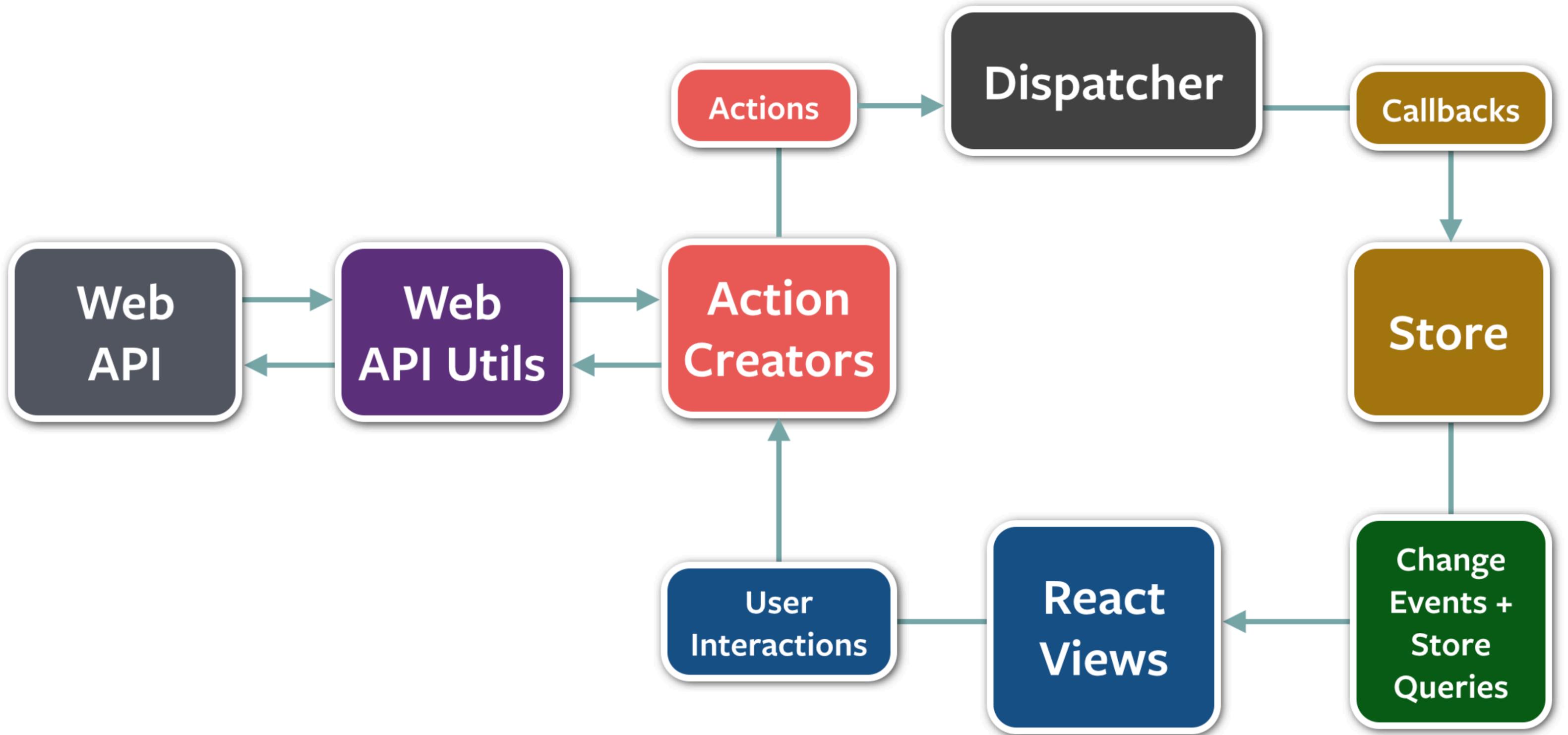
- 難しいものを簡単には作れない
- 難しいものは考えて作るしかない
- 考えて作っていくためには、議論できる言語化されたコードが必要
- また、構造化することでメンテナンス性を高める
- ルールは明確に、でも最初から明確なワケではない
- どうやってそれらを行っていったかについて

# 構造化の目的

- 長期的に動くものを書くため
- 属人性が高くないように、議論して開発できる構造が必要
- 最初から完成している設計はない
  - 立ち上げ時は方向性を決めてコアドメインを作る
  - 20151110 ドメイン駆動設計によるサービス開発
- ドメインモデルも時間で変化する、そのため考え続けないといけない

Flux

/fɪˈlʌks/



# 10分で実装するFlux

[azu.github.io/slide/react-meetup/flux.html](http://azu.github.io/slide/react-meetup/flux.html)

# Fluxのデータフローはわかった

- でも、ドメイン、ロジックはどこに書くの？
- ドメインモデルはどこにいるのか？
  - ここでいうドメインモデルはデータと振る舞いを持ったモデル
- ActionCreator or Store?
- StoreはViewに対するStateを管理する場所にも見える
  - Reduxの中にも明確な答えがあるわけではない

# Fluxの中でのロジック

Stores contain the application state and logic.

– *Flux | Application Architecture for Building User Interfaces*

Storeはデータとロジックを持つ

一旦ここまでの話をチームで考えてみる

How to work as a Team @  
2016/02/23 #reject\_sushi

# 目的

- 新規でそこそこ複雑なウェブページを作る(アプリに近い)
- ある程度柔軟に拡張でき、メンテできるような設計にしたい
- 無難に**React** + 何かでちゃんと設計して作っていききたい
- この設計部分をどう決めていくのかという話

# 現状

- チームに**React/Flux/Redux**を触ったことがない人が多い
- どれが(主に**View**以外の設計)ベストかは分からない
  - **Flux**的な部分の話

# サンプルづくり

→ Flux Util

→ Redux

→ Fluxible

→ material-flux

→ でそれぞれ作った

→ **View(React)**は全て同じものを参照して、**View**と他はちゃんと分離出来ることも証明してみせた

直面していると思われる混乱を  
描いてみてください

Fluxと両立することができる???

Flux設計

ドメイン

語彙豊かなモデル

エラー/ローディング

グの簡潔さ

ユーザーやステークホルダーが  
あなたの混乱を説明するとしたら、  
どんな単語を使うと思いますか？

メンテナンス

分かりやすさ

語彙豊か

テストしやすさ

エラー表示

ドメイン

ローディング

面倒臭さ

トレードオフ

## ここまで

- Fluxは一方通行のデータフローを定めているのは分かった
- 「ActionはUseCaseと呼んだ方が直感的だなー」
- Storeの役割が直感的ではないという意見
  - 「StoreはただのStateを持つObservableな箱とした方が分かりやすい」
- 「Actionを受け取りStoreで処理するときにドメインはどこに書くのが不安になる」

少しFluxの見方を変える

# ドメインモデル

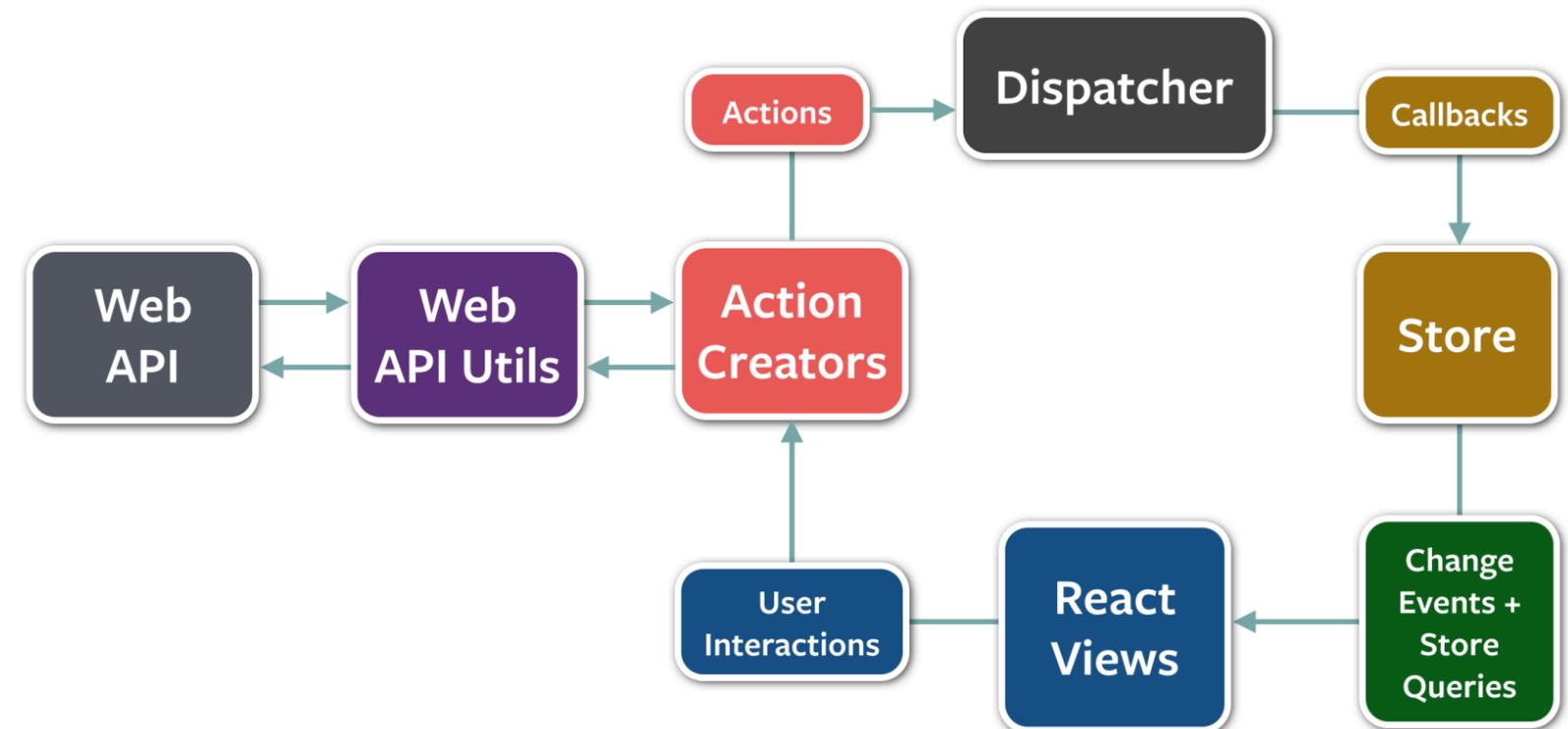
ドメインモデル	プレゼンテーション、アプリケーション、ドメイン、インフラストラクチャの4つのレイヤーに基づき、DDD スタイルで設計された階層化アーキテクチャ。とりわけ、モデルが特殊なオブジェクトモデルであることが想定される
コマンド/クエリ責務分離 (CQRS)	コマンド部分とクエリ部分を処理するための並列セクションを持つ二重の階層化アーキテクチャ。これらのセクションは別々に設計することが可能で、DDD やクライアント/サーバーなど、異なるサポートアーキテクチャを使用することもできる
イベントソーシング	ほとんどの場合は、CQRS にヒントを得て、単純なデータではなくイベントのロジックに焦点を合わせた階層化アーキテクチャ。イベントはファーストクラスのデータとして扱われる。その他の問い合わせ可能な情報は格納されているイベントから推測される

# ドメインモデルをなぜ作りたいの？

- UIじゃなくてコードを読んで挙動がわかるようにするため
- レイヤーを分けて実装するためのパターンでしかない
- それぞれを適切に分解し、適切な注意を払って実装したいだけ
  - ロジックはロジックに集中しよう
  - 永続化やキャッシュは別レイヤーで考えられるように分けよう 等
- そのユースケースに出てくる用語をドメインの振る舞いとして書くため
  - `user.buy(productItem)`

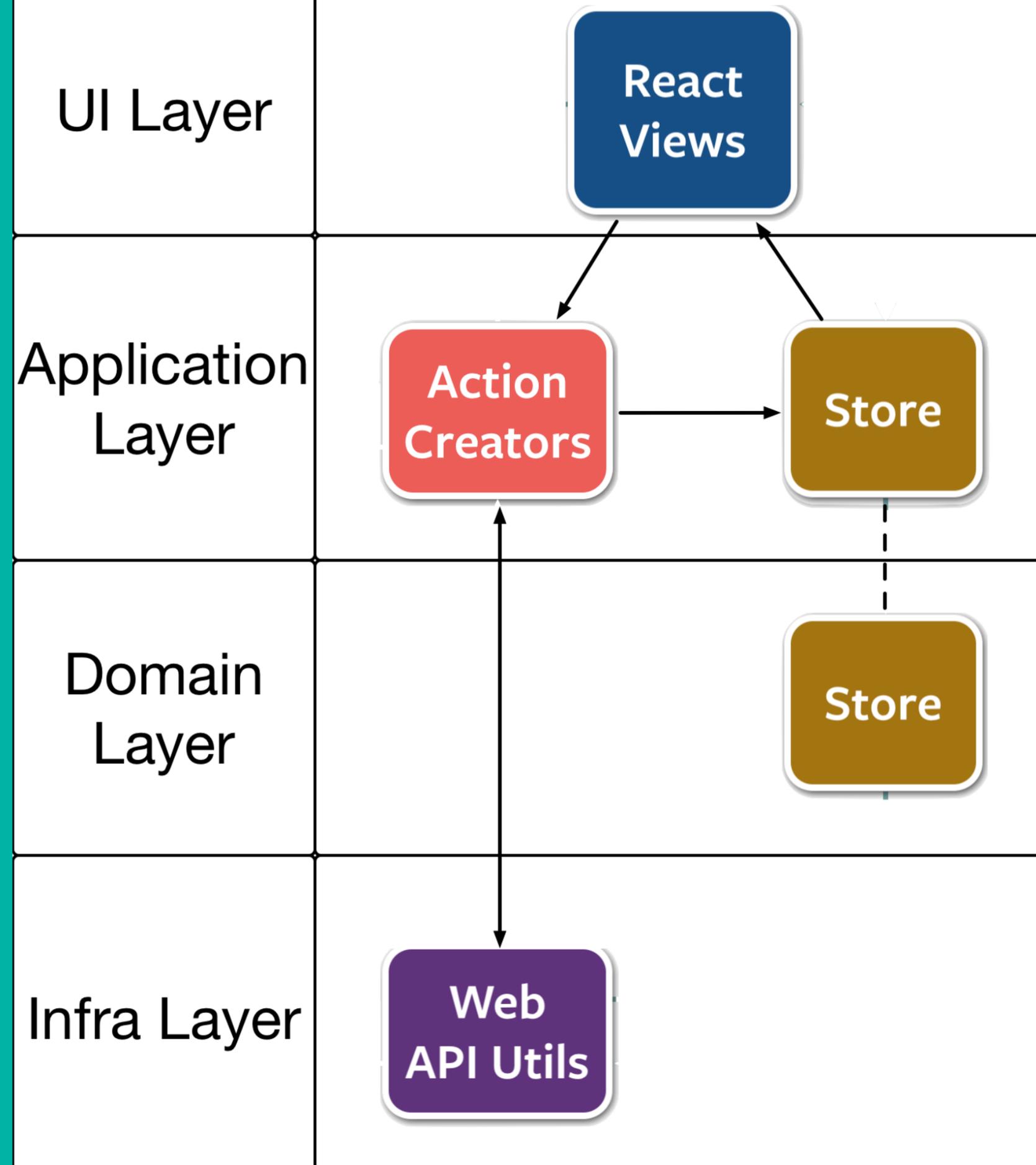
# Fluxをドメインモデルに置き直してみる

- View(プレゼンテーション層)
- ActionCreator(アプリケーション層)
- Store(アプリケーション層)
- Store ???(ドメイン層)
- Web API(インフラストラクチャ層)



# Fluxとドメインモデル

- Fluxでは明確なDomain Layerがない
- Storeがアプリケーションのドメイン、状態(State)を合わせ持つ
- ActionCreatorがInfraを使うので、永続化の問題をどうするか別途考える必要がある
- WebSocketで繋ぐAPIとかを考えると分かりやすい

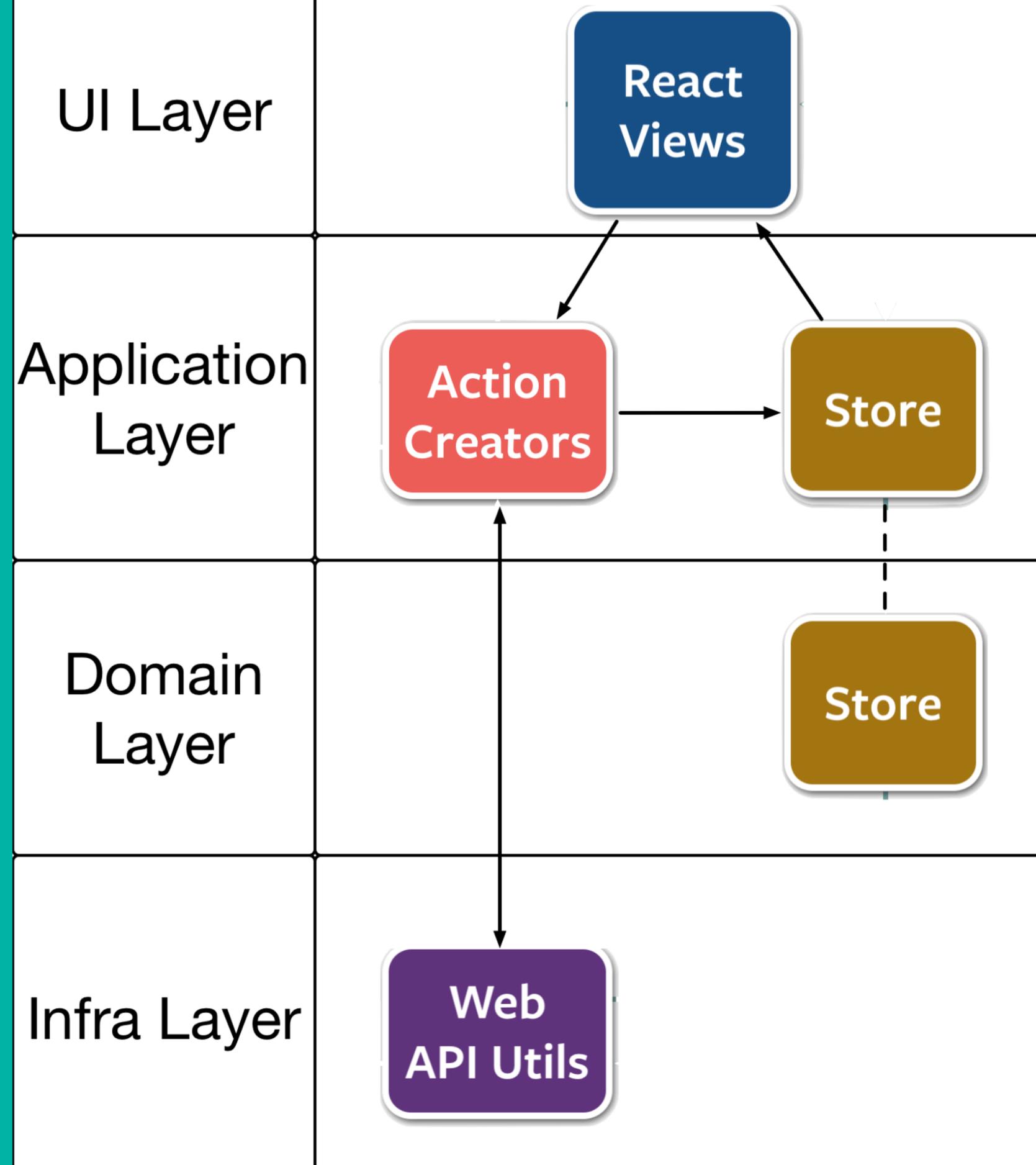


# Fluxのいいところ(確認)

- データフローが一方通行になる
  - それによりデータの流れが追いやすい
  - 複雑さが減りバグを減らしやすい構造

# Fluxの曖昧なところ

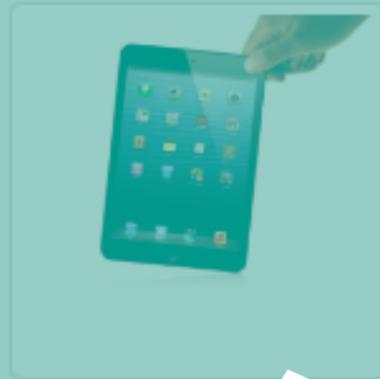
- Domain Layerが曖昧
- Storeがシステムの状態とViewの状態とロジックを含んでいる
- Fluxのまま構造化をするなら、Storeの中で構造化が必要
- Storeを構造化する例: FluxとDDDの統合方法 - かとじゅんの技術日誌



# Storeの役割

- FluxのStoreは2つの側面を持っている
  - Actionを受け取りデータを更新(**Write**)
  - Viewの要求に対してデータを返す(**Read**)
- せっかく一方通行なのに、Storeがやることが2つある
- この2つを出来る限り切り離したい

## Flux Shop Demo (material-flux)



iPad 4 Mini - €500.01

Add to cart

### Your Cart

iPad 4 Mini - €500.01 x 1

H&M T-Shirt White - €10.99 x 1

Charli XCX - Sucker CD - €19.99 x 5

Total: €610.95

Checkout

# ショッピングカート

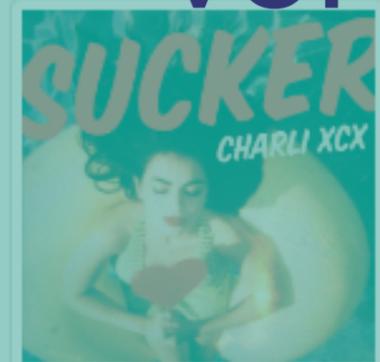


H&M T-Shirt White - €10.99

Add to cart

# Storeが曖昧な例

# voronianski/flux-comparison



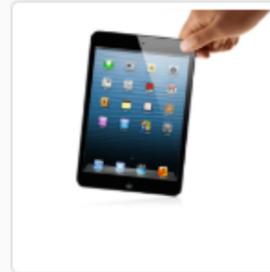
Charli XCX - Sucker CD - €19.99

Sold Out

# ショッピングカート

- ショッピングカートはStoreの2つの役割を見るいい例
- 商品の在庫とカートの中身を同時に扱わないと行けない問題を含んでいる

## Flux Shop Demo (material-flux)



iPad 4 Mini - €500.01

Add to cart



H&M T-Shirt White - €10.99

Add to cart



Charli XCX - Sucker CD - €19.99

Sold Out

### Your Cart

iPad 4 Mini - €500.01 x 1  
H&M T-Shirt White - €10.99 x 1  
Charli XCX - Sucker CD - €19.99 x 5  
Total: €610.95

Checkout

# ショッピングカートの Store

- ProductStore
  - アイテム
  - 在庫数
- CartStore
  - カートのアイテム × 数
  - 合計金額

Flux Shop Demo (material-flux)



iPad 4 Mini - €500.01

Add to cart

## ProductStore



H&M T-Shirt White - €10.99

Add to cart



Charli XCX - Sucker CD - €19.99

Sold Out

Your Cart

iPad 4 Mini - €500.01 x 1  
H&M T-Shirt White - €10.99 x 1  
Charli XCX - Sucker CD - €19.99 x 5  
Total: €610.95

Checkout

## CartStore

# 前提の話

- 前提
  - Viewにロジックは書かない
  - CartStoreからカートの状態を取れる、ProductStoreからは商品の状態が取れる(StoreはViewへのマッピングもしてる)
  - 在庫がない場合はカートに入れることはできない(Sold Out)

# 依存の問題

- CartStoreにアイテムを追加するときに、ProductStoreに在庫があるかを確認しないといけない
- CartStoreはProductStoreに依存している

Flux Shop Demo (material-flux)

The screenshot displays a 'Flux Shop Demo (material-flux)' interface. On the left, a 'ProductStore' is shown with three items: 'iPad 4 Mini - €500.01' (with an 'Add to cart' button), 'H&M T-Shirt White - €10.99' (with an 'Add to cart' button), and 'Charli XCX - Sucker CD - €19.99' (with a 'Sold Out' button). On the right, a 'CartStore' is shown with a 'Your Cart' section containing the items: 'iPad 4 Mini - €500.01 x 1', 'H&M T-Shirt White - €10.99 x 1', and 'Charli XCX - Sucker CD - €19.99 x 5', with a 'Total: €610.95' and a 'Checkout' button. A blue arrow points from the 'CartStore' to the 'ProductStore', indicating that the CartStore depends on the ProductStore for item availability.

# 依存の問題

- Storeを完全に独立したものとして実装してしまうと問題が起きる
- "カートにアイテムを入れる"というAction
  - ProductStoreから在庫を減らす
  - CartStoreにアイテムを追加する
- Product -> Cartの順番で行わないと在庫がないのにカートにアイテムが入るなどの問題を起こしがち

# 問題の解決方法

- Redux
  - Storeは1つ = Single source of truth
  - 1つなので、CartはProductを知っている
- Facebook/flux
  - `waitFor` - dispatchされたActionの処理順を明示する
- 他: [voronianski/flux-comparison](#)

# 複雑な問題の一端

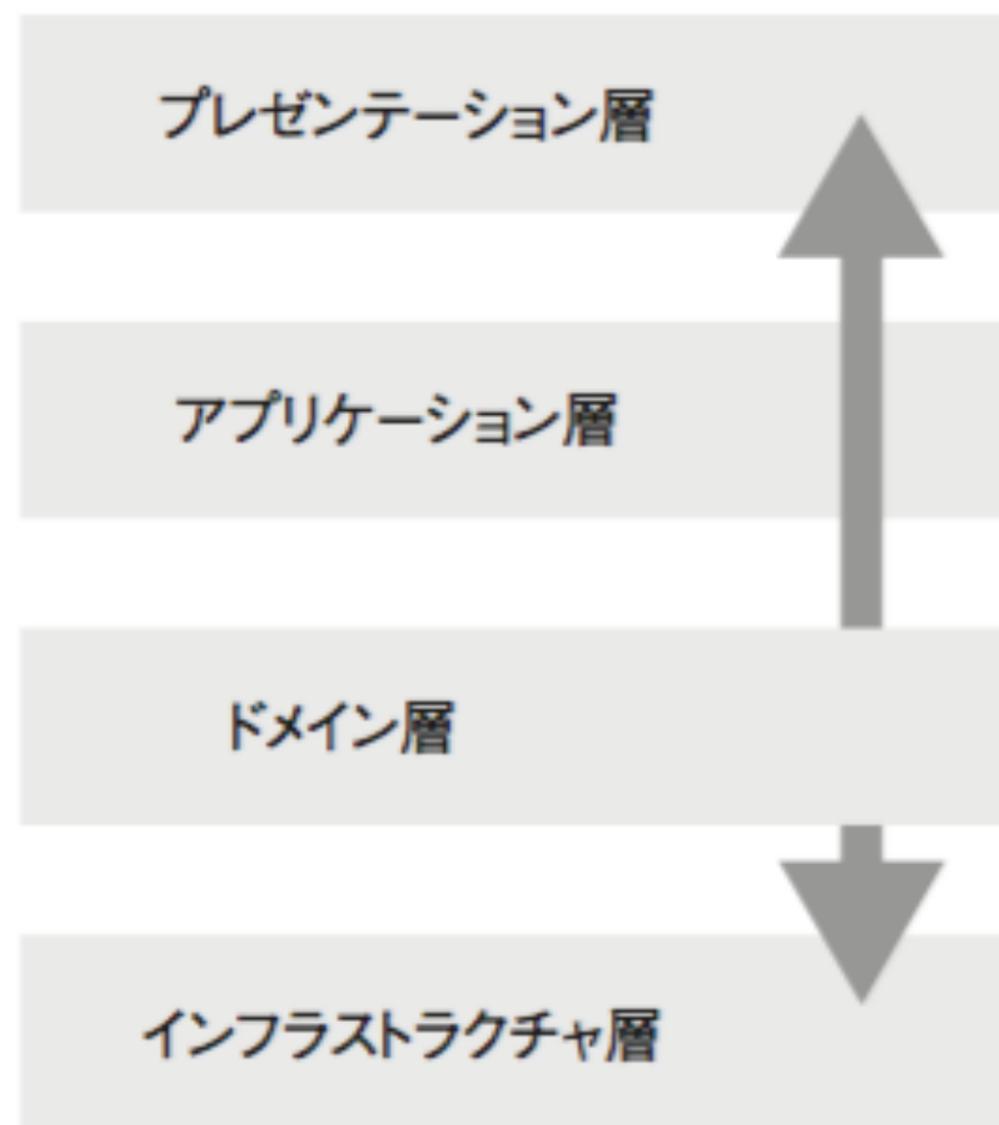
- 次の2つをことを1つのモデル(Storeのこと)で行っているのが複雑な原因
  - **N: Actionを受け取りデータを更新(Write)**
  - **M: Viewの要求に対してデータを返す(Read)**
- 1つモデルで2つの事をやると複雑さは**掛け算**となる
  - 複雑さが  **$N \times M$**  になる
- それぞれに1つずつのモデルを用意すれば複雑さは**足し算**になる
  - 複雑さが  **$N + M$**  になる(代わりにモデルは2つになる)

# 複雑さの掛け算( $N \times M$ )をなくしたい

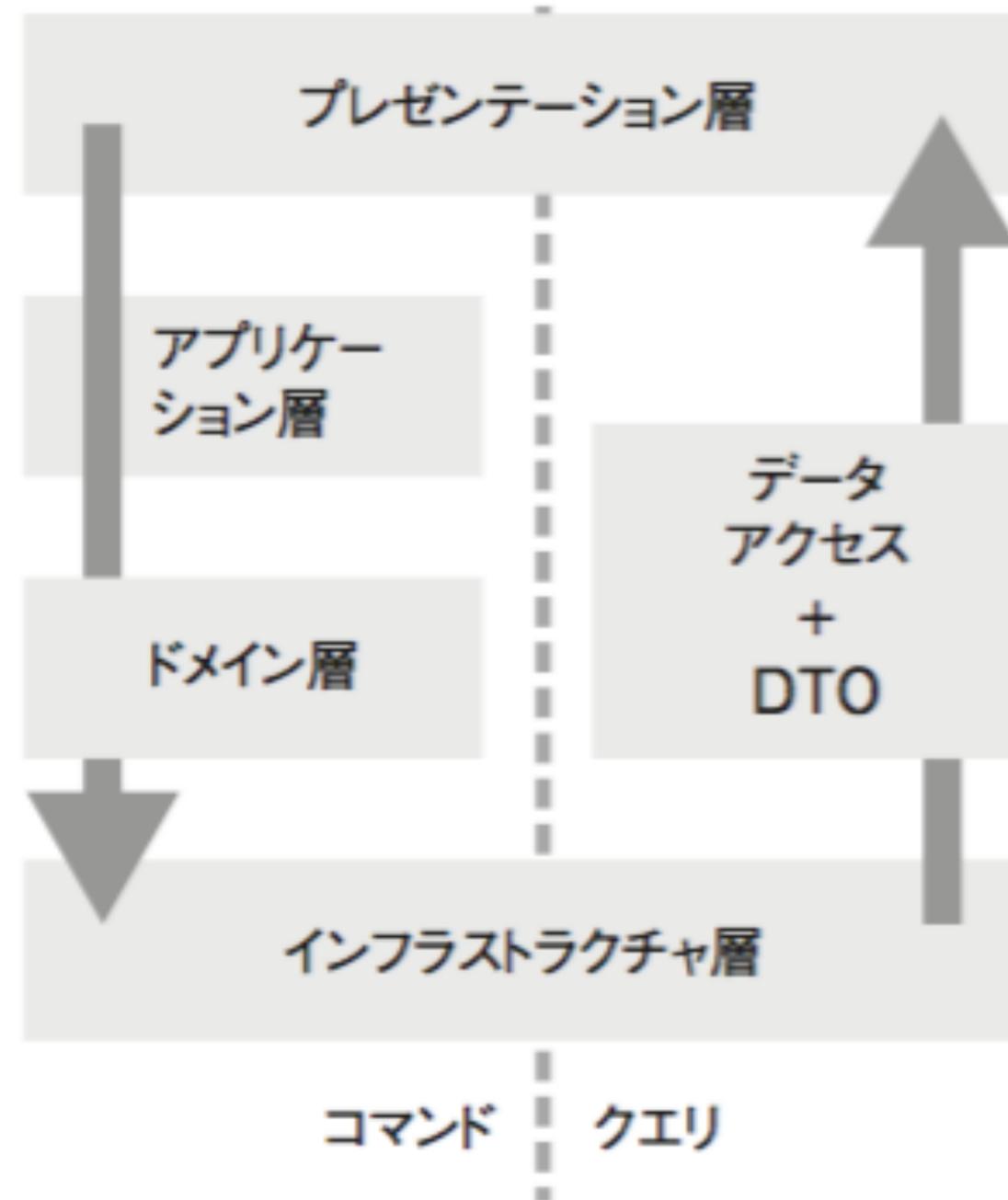
- Storeにデータを書き込むときに次の2つ(WriteとRead)を同時に考えてしまってる
  - 受け取ったActionをどうStoreで処理するか(ビジネスロジック)
  - View(Component)向けにどういう形のオブジェクトを返すか
- この2つを同時ではなく、一旦分けて考えられる状況を作ろう
  - => **CQRS**という考え方

CQRS(コマンドクエリ責務分離)

## Domain Model



## CQRS



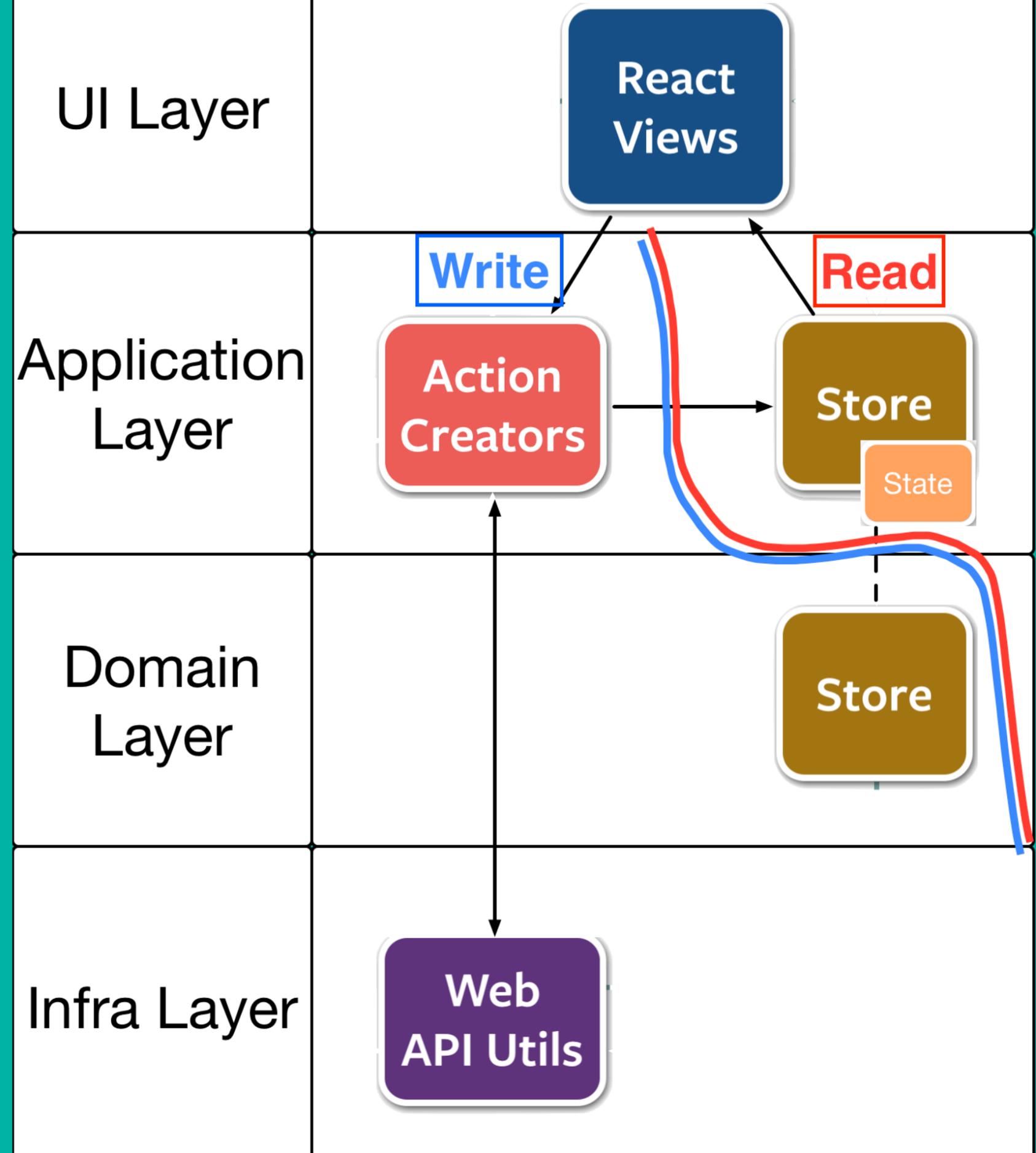
▲図 10-1 : Domain Model と CQRS の視覚的な比較

# CQRS(コマンドクエリ責務分離)

- Command Query Responsibility Segregation
- 構造をコマンド(**Write**)とクエリ(**Read**)で縦に割る
  - Actionを受け取りデータを更新(**Write**)
  - Viewの要求に対してデータを返す(**Read**)
- クエリ(**Read**)は読み取りのみなので**Write**よりは単純
- 詳しくは.NETのエンタープライズアプリケーションアーキテクチャ

# FluxのWriteとRead

- Storeの中で構造化するため上手く割り切れない
- Action(Command)がStoreに書き込む
- ViewがStoreからState(Query)を読み取る
- => Storeの役割を変えないとやりにくい



やっと本題



**Almin.js**  
Almin

# ここからの話

- **Almin.js**を作るまでに考えた設計の概念的な話
  - 理想的な形をクライアントサイドで動く現実の形に落とす話
- コードの解説ではないです
  - ドキュメントを見て
  - <https://almin.js.org>

# 考えるポイント

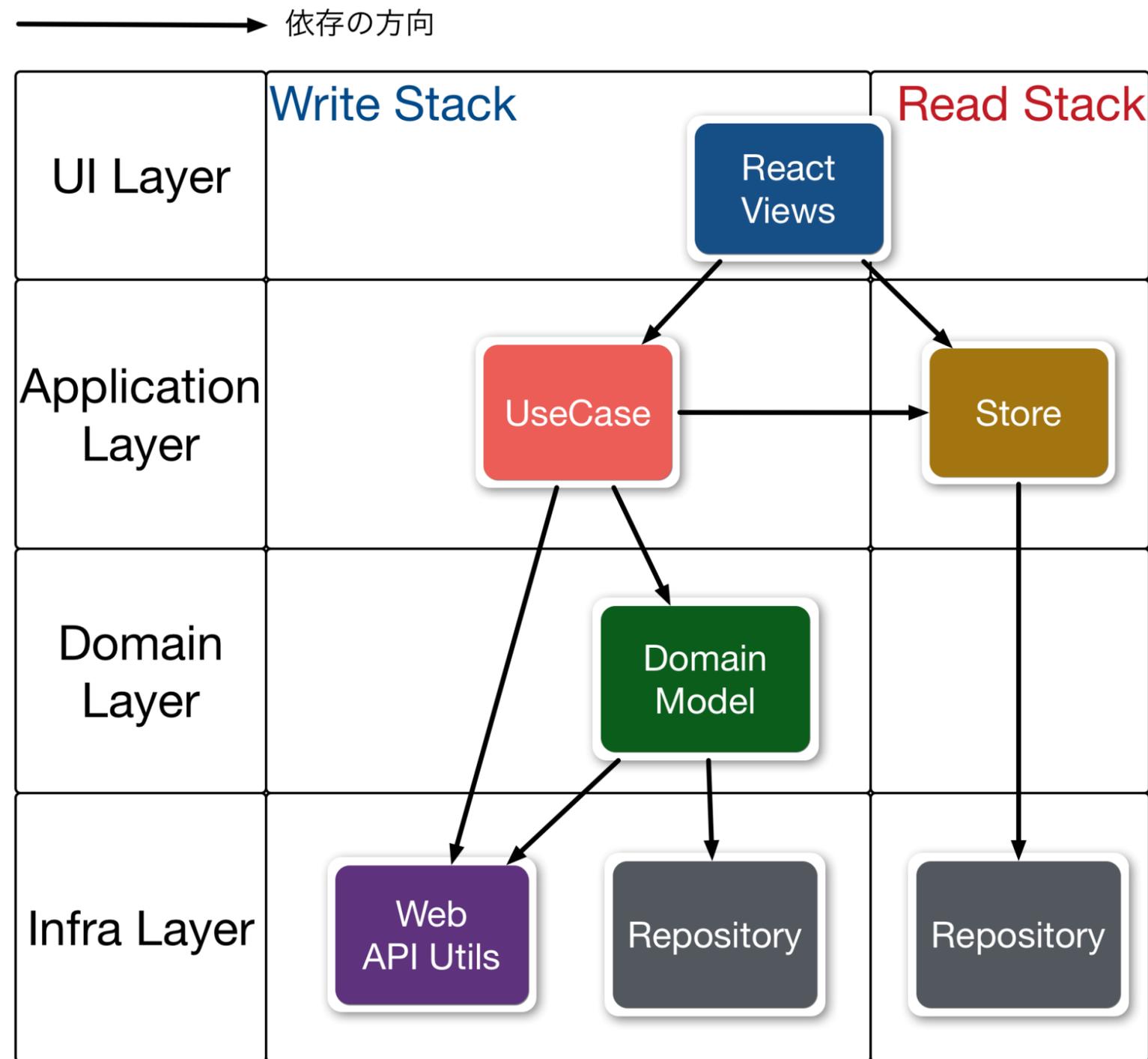
- Write StackとRead Stackを分離 = CQRS
  - Write × Readの複雑さを掛け算ではなく足し算にする
  - **Write × Read => Write + Read** <sup>注</sup>
- ドメインモデルを扱える構造を作る
- クライアントサイドで問題点となるのはオブジェクトの永続化
  - シングルトンがでてくる問題

---

<sup>注</sup> Write と Readが共に複雑でなければ、掛け算の方が簡単なのは自明です

# 全体像(Simple版)

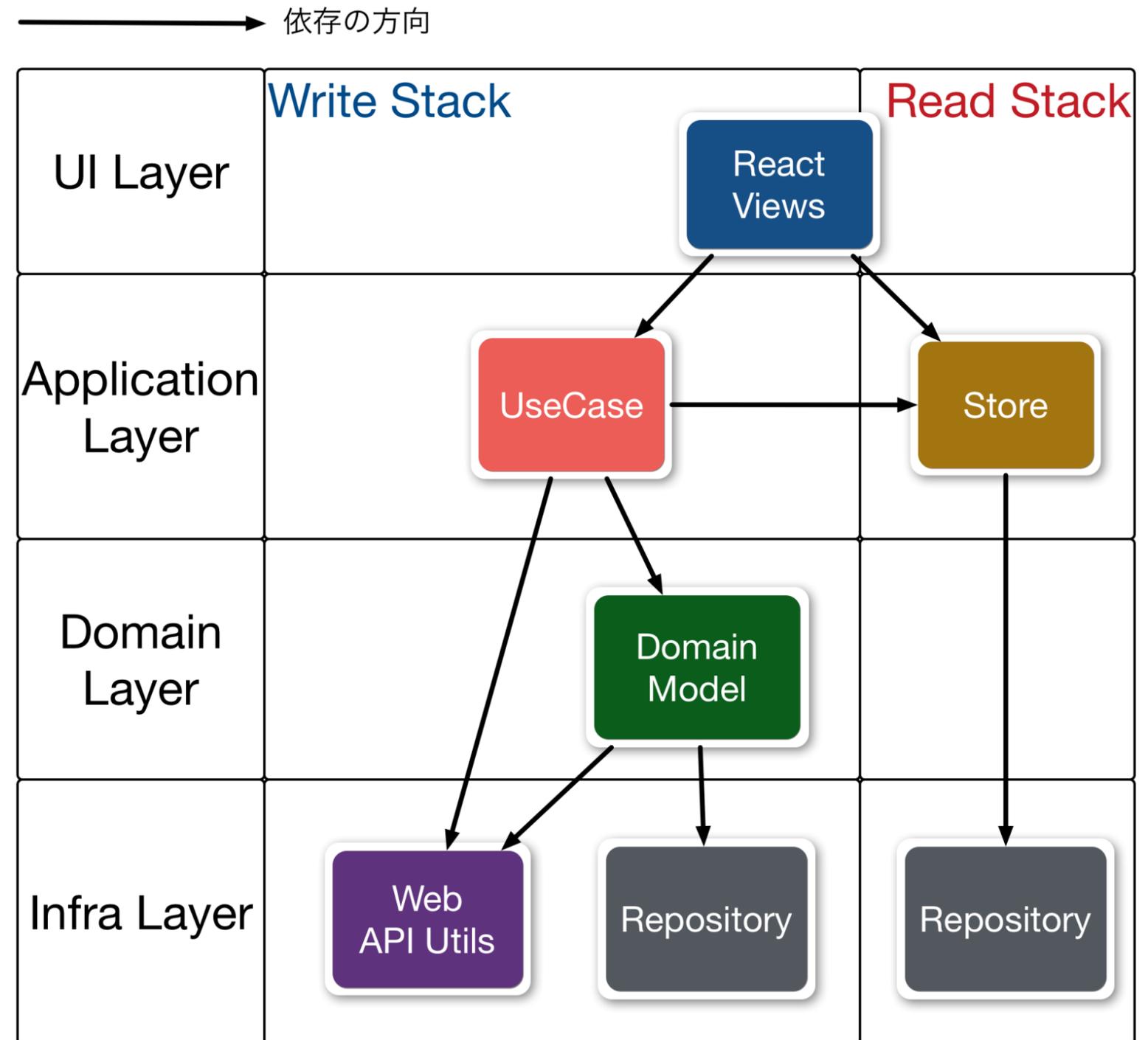
## 画像は概念イメージ



依存関係逆転の原則(DIP)で依存の方向を変える前

# 登場人物

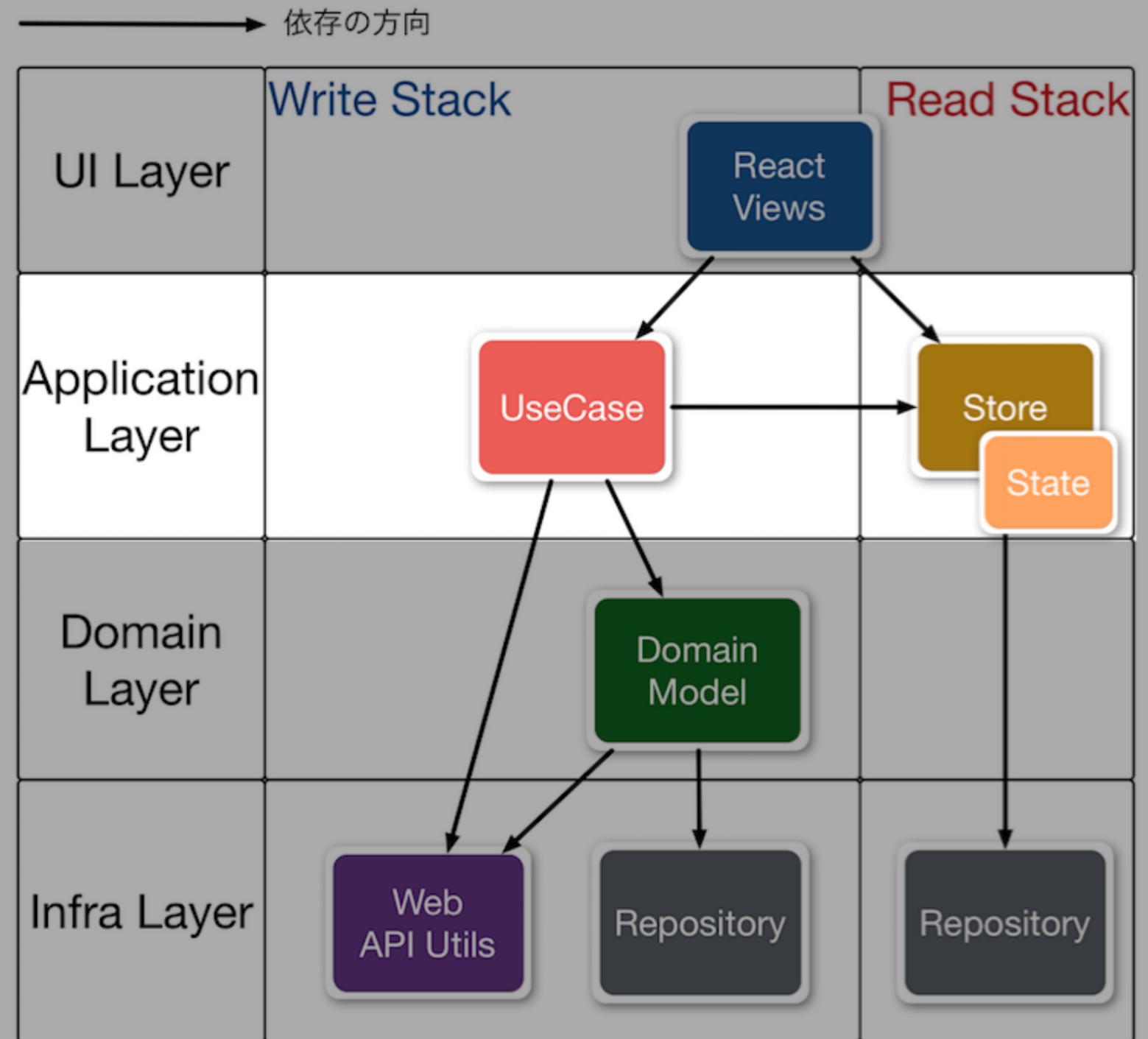
- View(React Component)
- Write Stack
  - UseCase
  - Domain
  - Repository ← 同一かも
- Read Stack
  - Store
  - Repository → 同一かも



依存関係逆転の原則(DIP)で依存の方向を変える前

# Alminが提供してるのこれだけ

- いわゆるFluxライブラリと対して変わらない
- しかしこの構造を強く意識作り、ドキュメント
- ここからの話はあくまでパターンにすぎないので、ライブラリに依存した何かではないはず



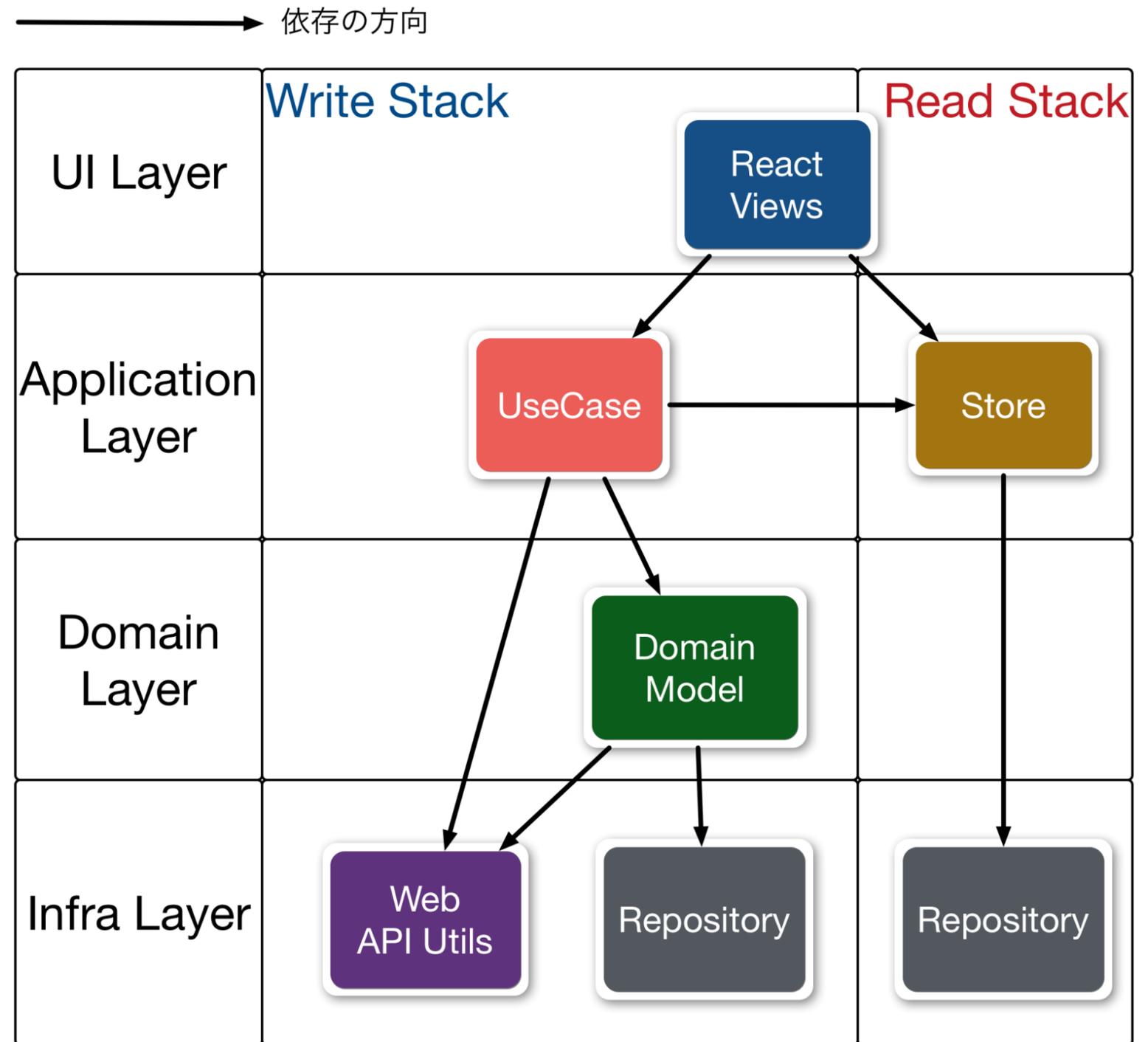
View

# View

- Reactを使う
- PostCSS使ってる
- 以上

Write Stack(コマンド)

# Write Stack



依存関係逆転の原則(DIP)で依存の方向を変える前

# Start from the Use Cases

The best place to start when trying to understand a new domain is by mapping out use cases.

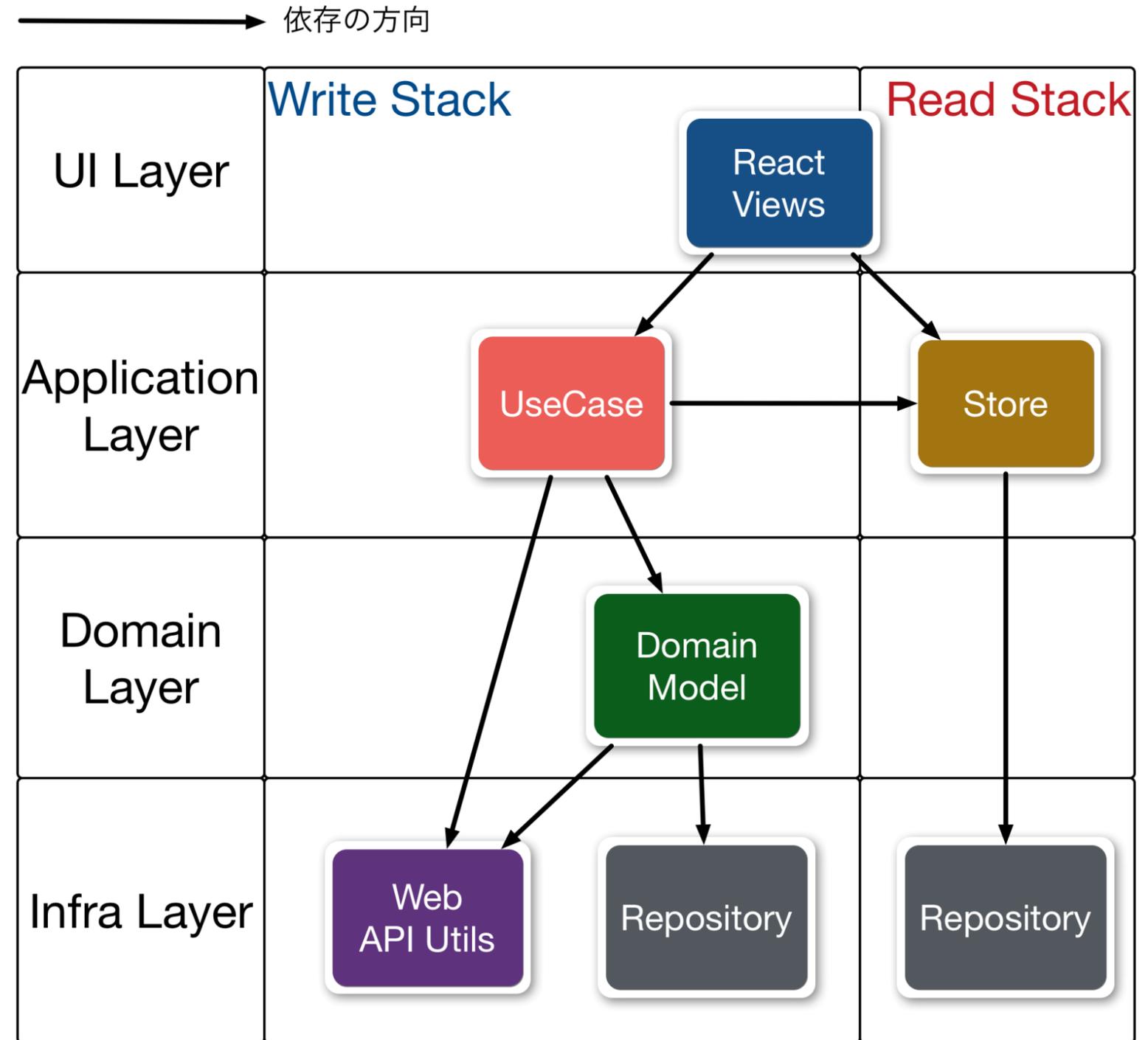
– *Patterns, Principles, and Practices of Domain-Driven Design*

# UseCase

アクターがシステムに対して何をしたいかを  
書く場所

# UseCase

- ViewからUseCaseを発行(ActionCreatorと類似)
- ドメインを使った**処理の流れ**を記述する
- ここに処理の内容を全部書くとトランザクションスクリプト
- UseCaseと対になるFactoryを持つてる
  - Factoryはテストのため(コンストラクタによる依存解決)



依存関係逆転の原則(DIP)で依存の方向を変える前

## UseCaseの例

「TodoListに新しいTodoを追加する」というユースケース

- `TodoRepository`から`TodoList`のインスタンスを取り出す
- `TodoList`に作った`TodoItem`を追加する
- `TodoRepository`に`TodoList`を保存する

# AlminのUseCase

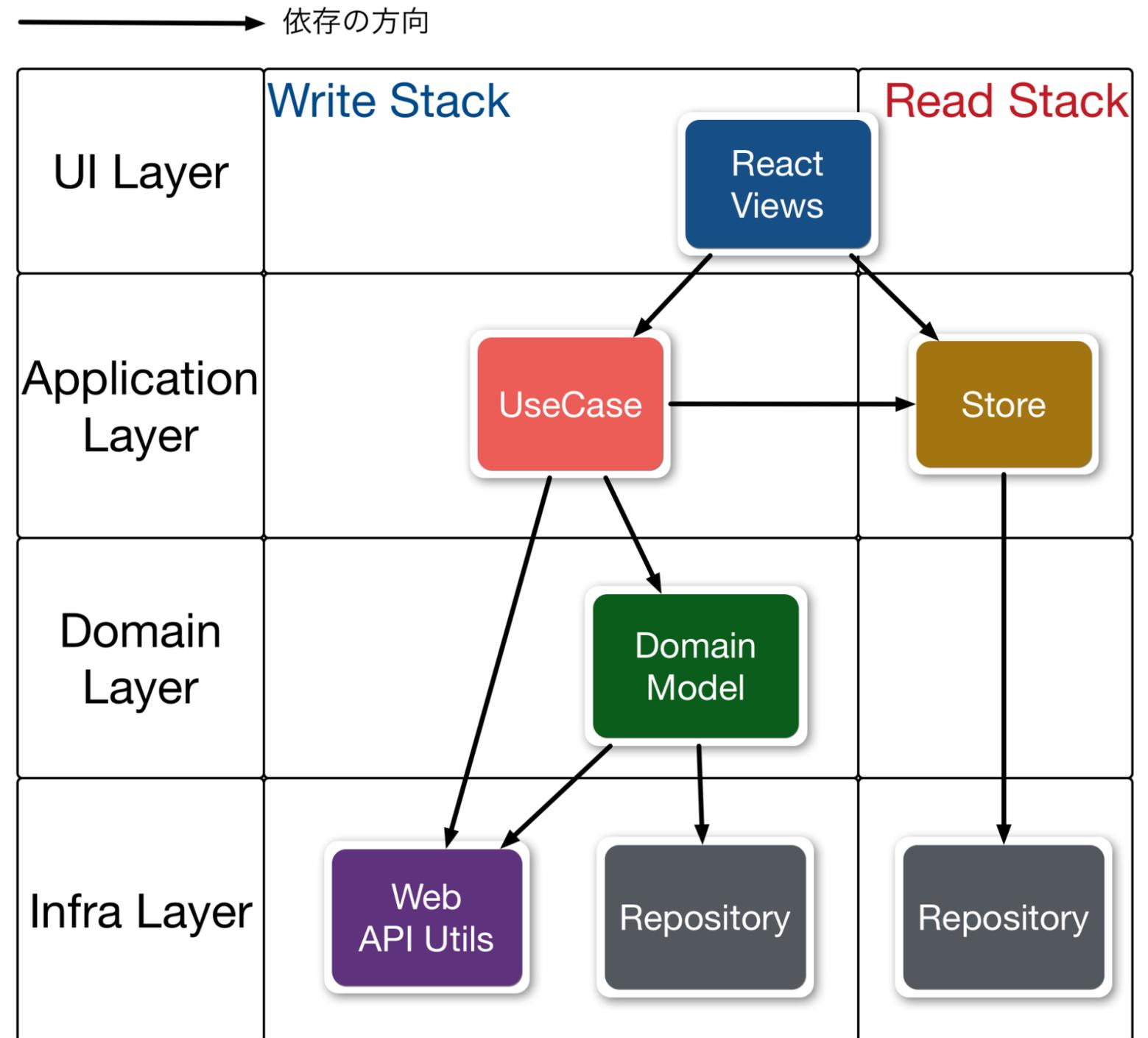
```
import {UseCase} from "almin";
export class AddTodoItemUseCase extends UseCase {
  execute(title) {
    // ユースケースの内容を書く
    // TodoListにTodoItemを追加するというロジック
    // ここに全部書きちゃうとトランザクションスクリプトっぽい
  }
}
```

# Domain Model

# Domain Model

- 作ろうとしているものを表現するオブジェクト<sup>ⓧ</sup>
- モデルクラス
- ここでは、データと振る舞いを持ったクラス
- できるだけPOJO(Plain Old JavaScript Object)である

<sup>ⓧ</sup> Storeは入れ物、Stateは中身という考え方



依存関係逆転の原則(DIP)で依存の方向を変える前

# モデルとは...

## 重要

私たちの考えでは、MVC パターンの**モデル**は、ソフトウェアの歴史において最も誤解されている概念の 1 つです。1980 年代に考案された MVC は、アプリケーションパターンとして出発し、アプリケーション全体の設計に使用することができました。それは何から何まで一つのトランザクションスクリプトとして作成された、モノリシックシステム時代のことでした。マルチレイヤーシステムとマルチティアシステムの到来により、MVC の役割は変化しましたが、その意義は失われませんでした。MVC は依然として強力なパターンですが、単一のモデルという発想はもはや通用しなくなっています。MVC の **Model** は「ビューで操作されるデータ」として定義されていました。これは現在の MVC が基本的にはプレゼンテーションパターンであることを意味します。

## 4.まとめ 確認

- MVC系のModel ≠ DDDのDomainModel
- DDDのDomainModel ≠ PoEAAのDomainModel
- MVC系のModel ≠ PoEAAのDomainModel
- MVVMのViewModel ≠ MVCのViewModel
- MVCのModel ≠ MVVMのModel
- WPF MVVMのModel ≠ JS MVVMのModel

# モデルの例: Todo

- TodoList: TodoItemを管理する
- TodoItem: TodoItemのオブジェクト

TodoList に TodoItem を追加する

```
function addNewTodo(title){  
  // TODO: 毎回TodoListを作ってるのはおかしいけど...  
  const todoList = new TodoList();  
  const todoItem = new TodoItem({title});  
  todoList.addItem(todoItem);  
}
```

# TODOを追加するUseCaseをモデルを使って 書く

```
import {UseCase} from "almin";
export class AddTodoItemUseCase extends UseCase {
  execute(title) {
    const todoList = new TodoList();
    const todoItem = new TodoItem({title});
    todoList.addItem(todoItem);
  }
}
```

# モデルの永続化

- モデルをPOJOで書けることは分かる
- モデルはどこでだれが永続化するの?
  - どこでインスタンス化して、どうやってインスタンス化したものを再度取り出すのか
- => **Repository**が永続化を考える層
- モデルは自身の永続化の方法をしらない(関心がない)

Repository

# Repositoryとは

ここでは、ドメインモデルの永続化に対処する概念/実装のこと

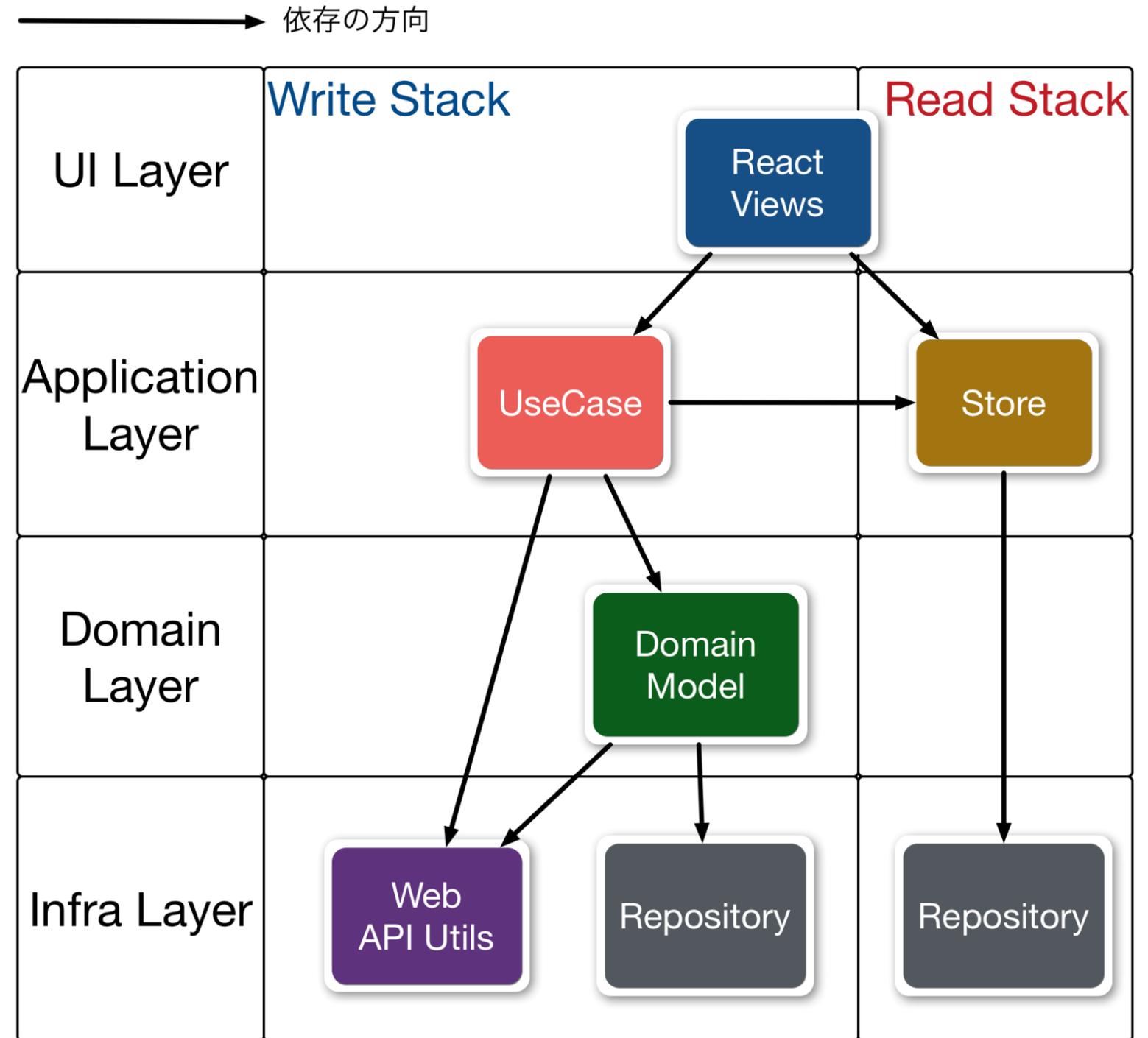
## 注

リポジトリはかなり包括的な用語です。まわりを見回してみると、相反する定義や、強く推奨される実装が見つかるかもしれません。どうやら、異なる概念に対して同じ用語が使用されているようです。DDD との関連では、リポジトリは本章で説明したとおりのものであり、エンティティと（理想的には）集約ルートに代わって永続化に対処するクラスです。これについては、第 14 章で詳しく説明します。

# Repository

- ドメインモデルのインスタンスを永続化する場所 図r
- Repositoryパターン
- Repository自体はシングルトン！でインスタンス化する
- `findById(id)/save(model)/delete(model)` などのAPIを持つケースが多い

図r 概念にすぎず、データや処理の流れを表すものではありません

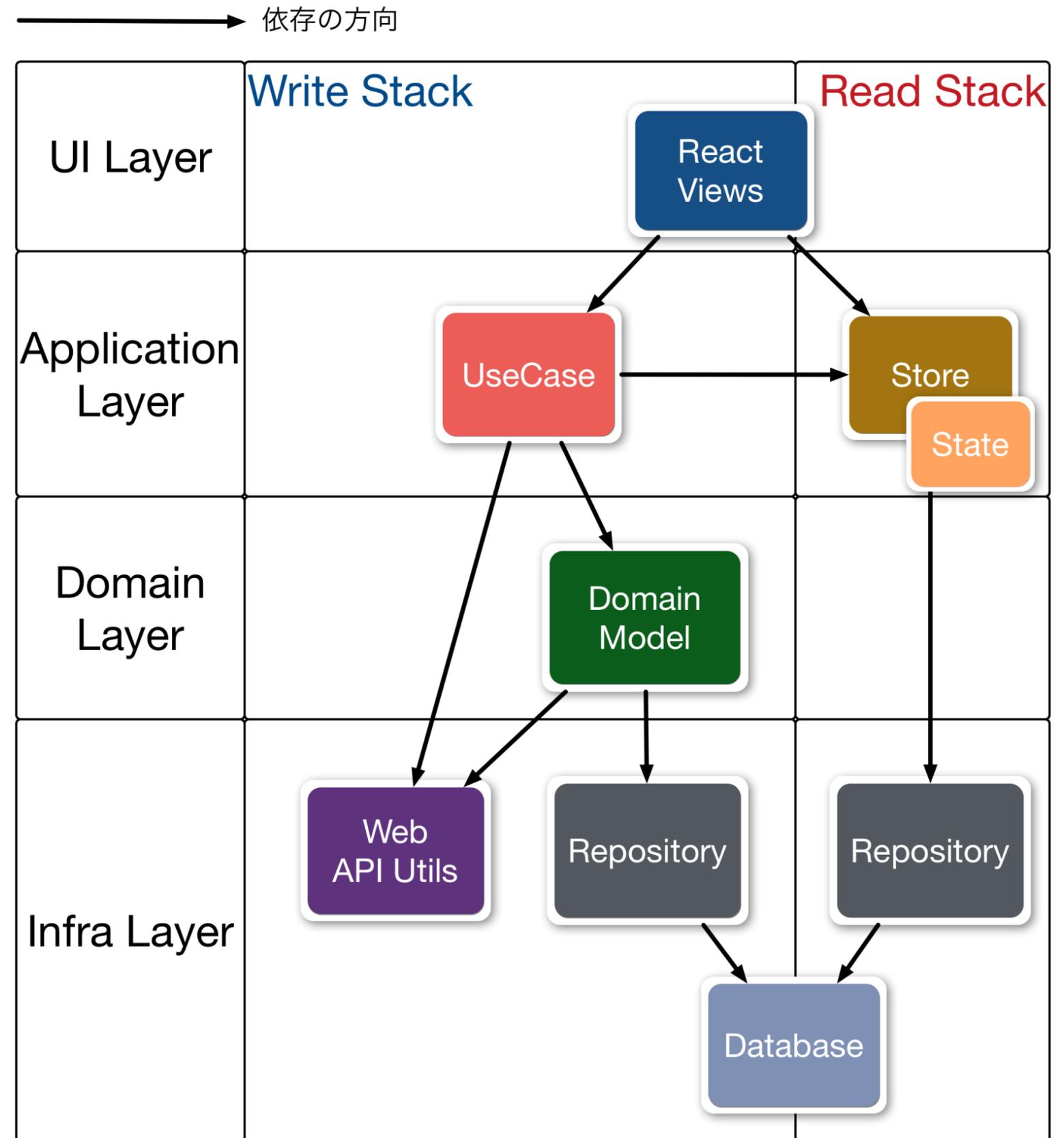


依存関係逆転の原則(DIP)で依存の方向を変える前

Repositoryの保存先?

# Database

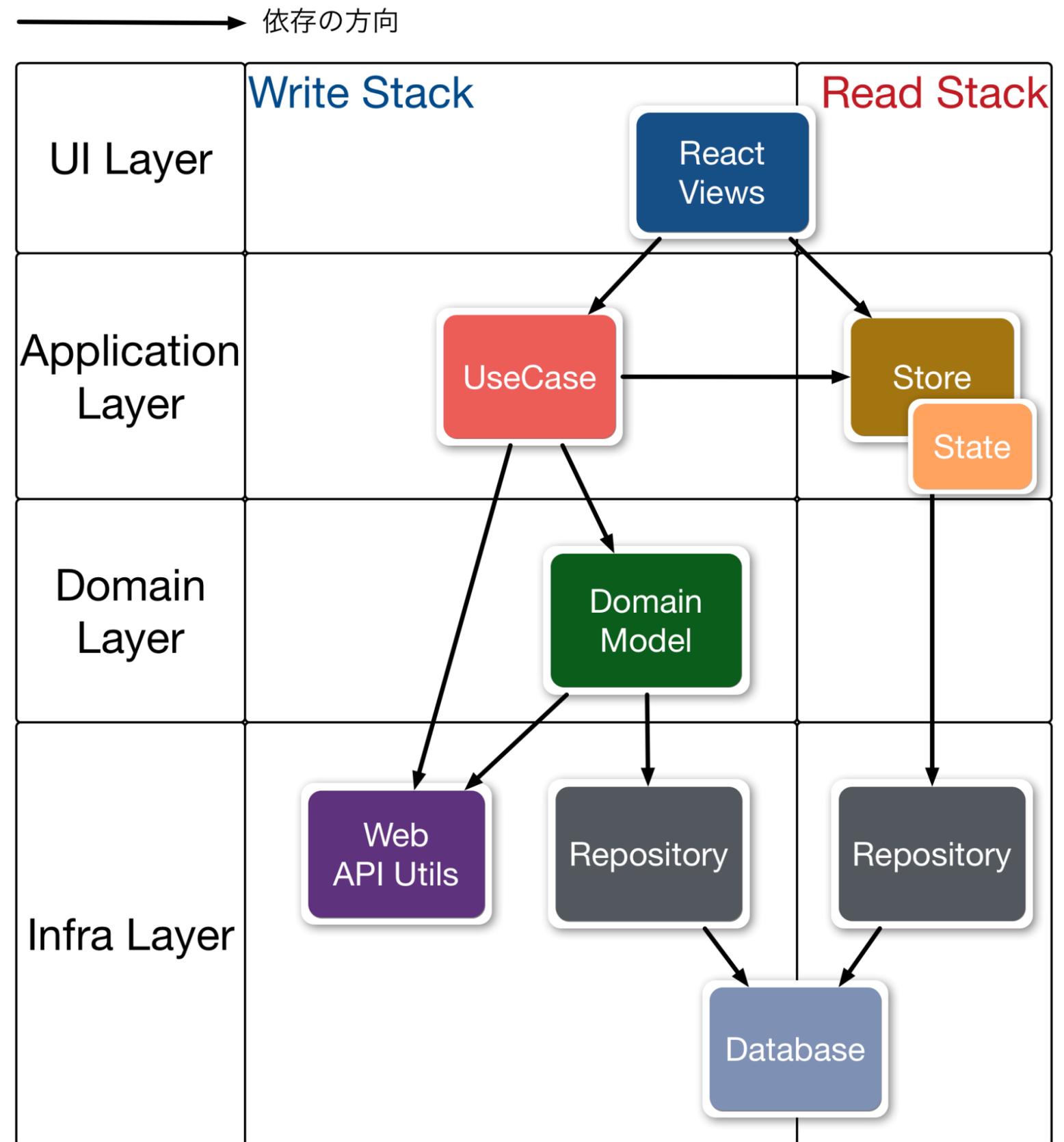
- Repositoryの保存先は実装毎に違う
- メモリ上でいいならただのMapオブジェクトでいい
  - localStorageとかIndexedDBなど色々
- AlminのサンプルだとDatabaseが変更の検知はRepositoryでやったり



```
import {UseCase} from "almin";
export class AddTodoItemUseCase extends UseCase {
  constructor({todoListRepository}) {
    super();
    this.todoListRepository = todoListRepository;
  }
  execute(title) {
    // RepositoryからTodoListのインスタンスを取得
    const todoList = this.todoListRepository.findById(todoListId);
    const todoItem = new TodoItem({title});
    todoList.addItem(todoItem);
    // RepositoryにTodoListを保存する
    this.todoListRepository.save(todoList);
  }
}
```

# Repository自体のインスタンス化の問題

- クライアントサイドJavaScriptでは永続化が難しい
- どこでインスタンス化するの?問題
  - それへの現実解としてシングルトンが出てくる
- DomainはRepositoryに依存してはいけない
- => 依存関係逆転の原則(DIP)



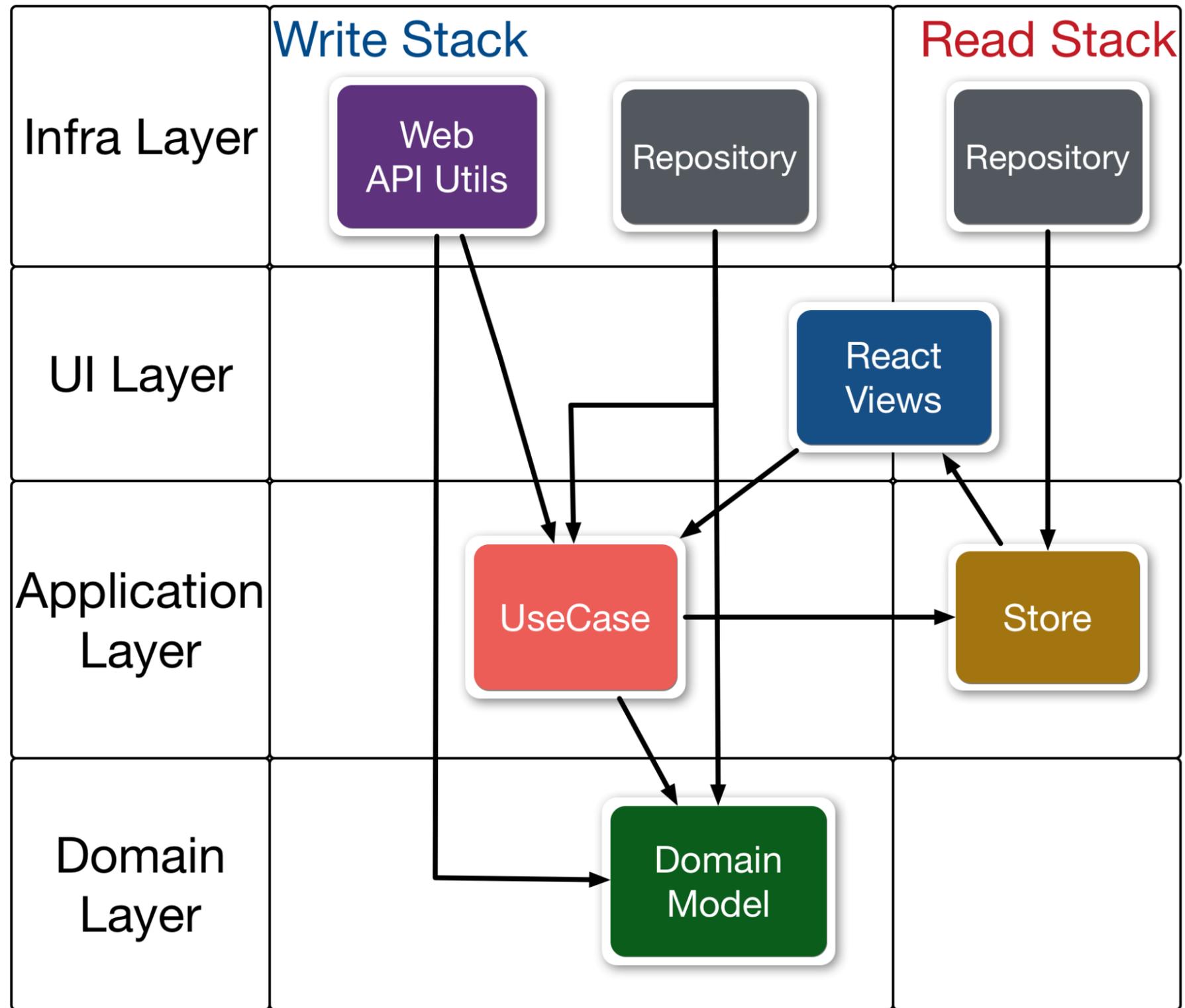
# 依存関係逆転の原則(DIP)

上位のモジュールは下位のモジュールに依存してはならない。どちらのモジュールも「抽象」に依存すべきである。  
「抽象」は実装の詳細に依存してはならない。実装の詳細が「抽象」に依存すべきである。

— 依存関係逆転の原則 (DIP) - *Strategic Choice*

*DIP:the Dependency Inversion Principle*

→ 依存の方向



依存関係逆転の原則(DIP)適応後

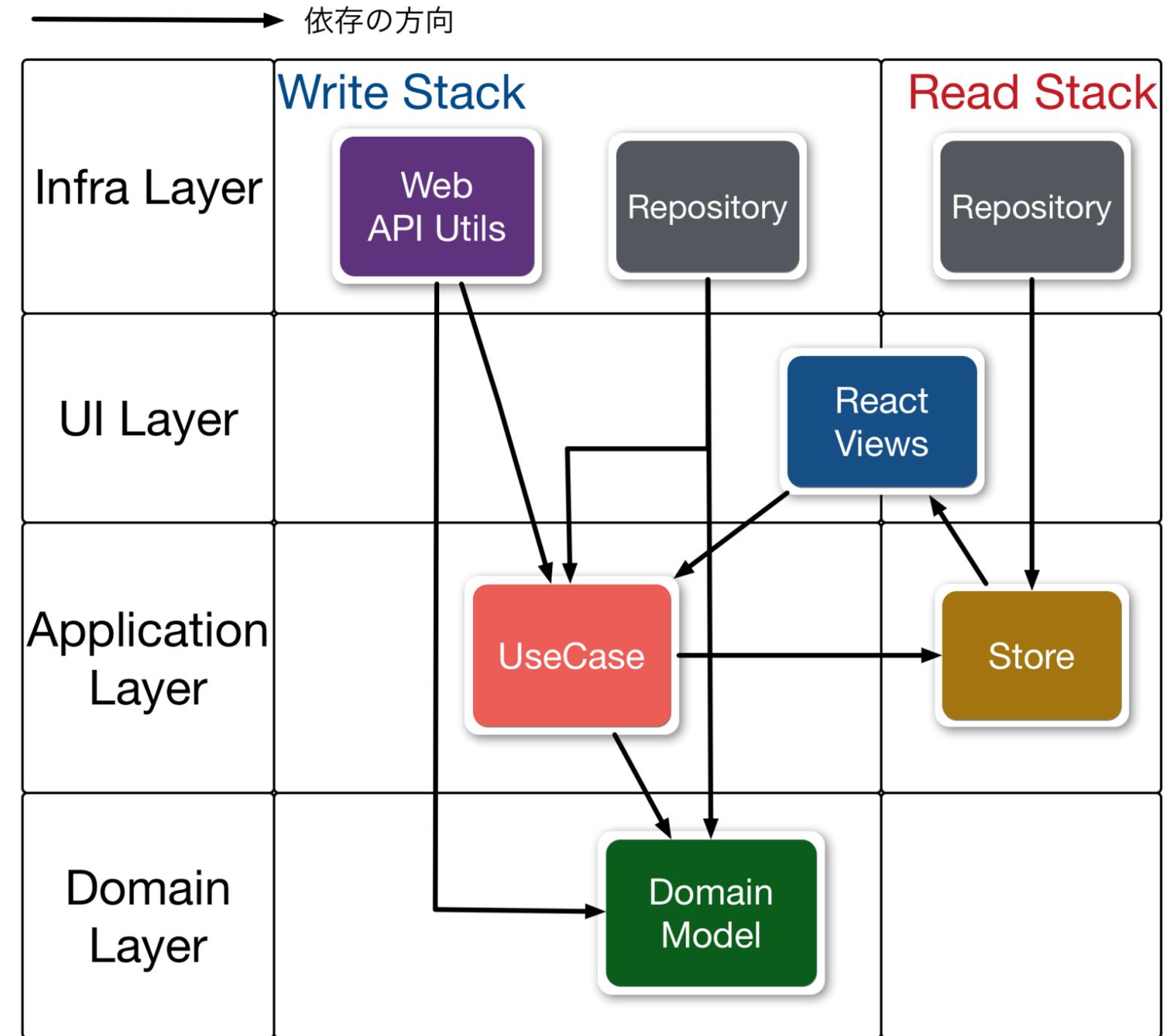
```
import {UseCase} from "almin";
// シングルトンを渡すだけのFactoryクラス

import todoListRepository from "../infra/ToDoRepository"
export class AddTodoItemUseCaseFactory {
    static create() {
        return new AddTodoItemUseCase({ todoListRepository });
    }
}
// テストする際は直接`UseCase`クラスを使う

export class AddTodoItemUseCase extends UseCase {
    constructor({ todoListRepository }) {
        super();
        this.todoListRepository = todoListRepository;
    }
    execute({ title }) {
        // ... ユースケースの内容
    }
}
```

# 依存関係逆転の原則(DIP)

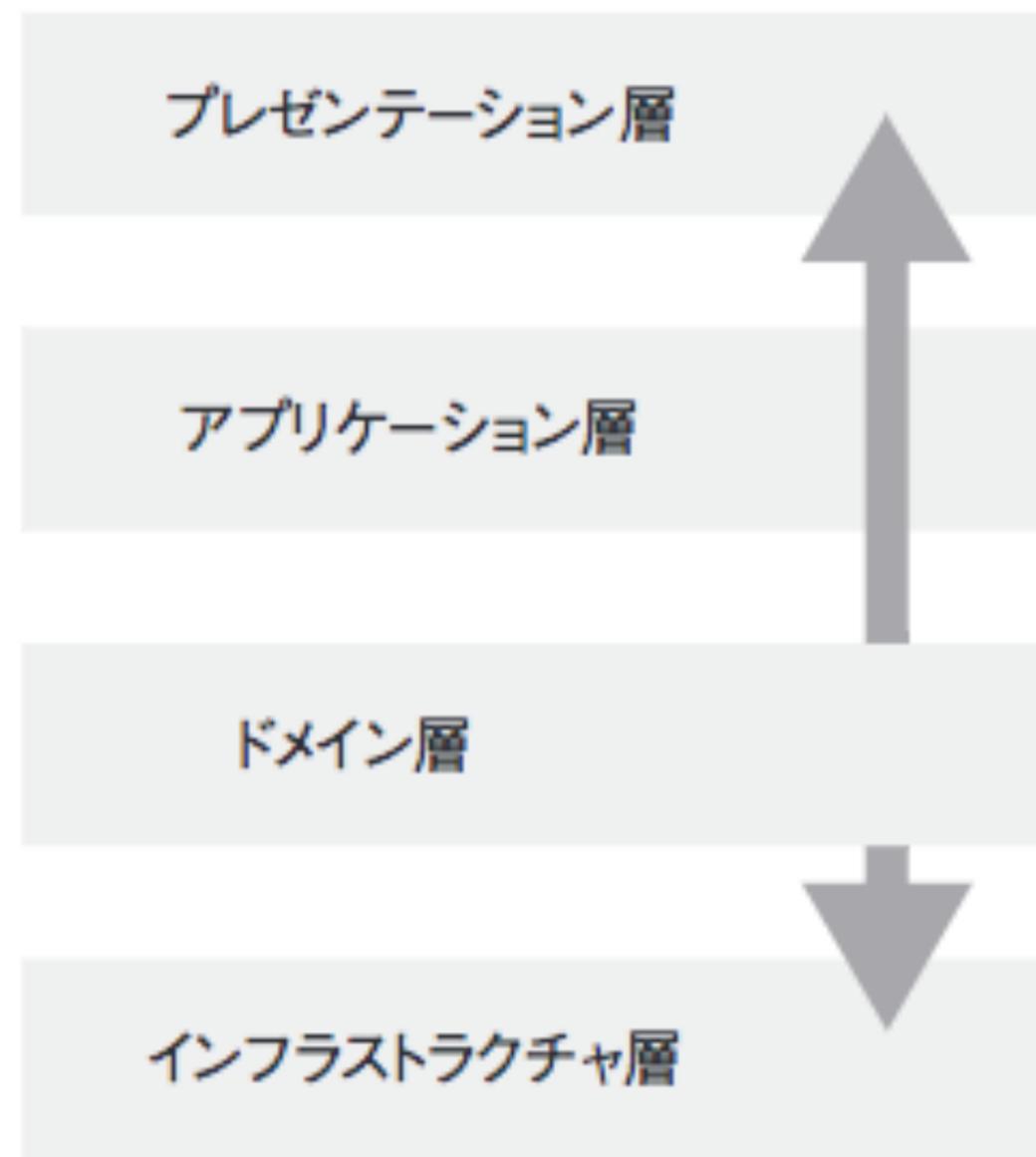
- Factoryが依存をUseCaseへ渡す
- UseCaseやドメインがリポジトリに依存しなくて良い
- ドメインがちゃんと永続化できる
  - シングルトンのリポジトリは常に存在するから
- テスト時はUseCaseのコンストラクタにDIすることでテストもできる



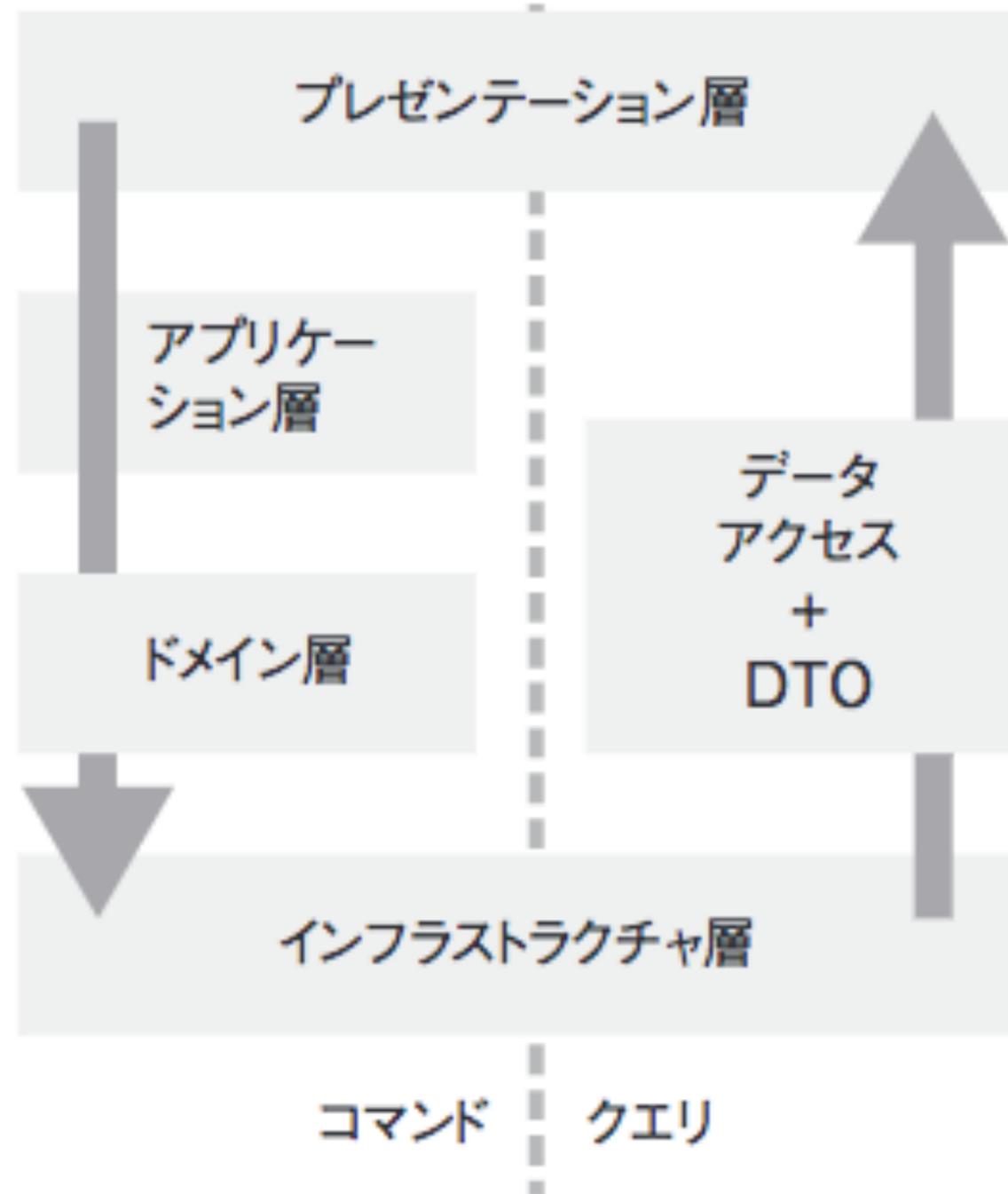
依存関係逆転の原則(DIP)適応後

Read Stack(クエリ)

## Domain Model



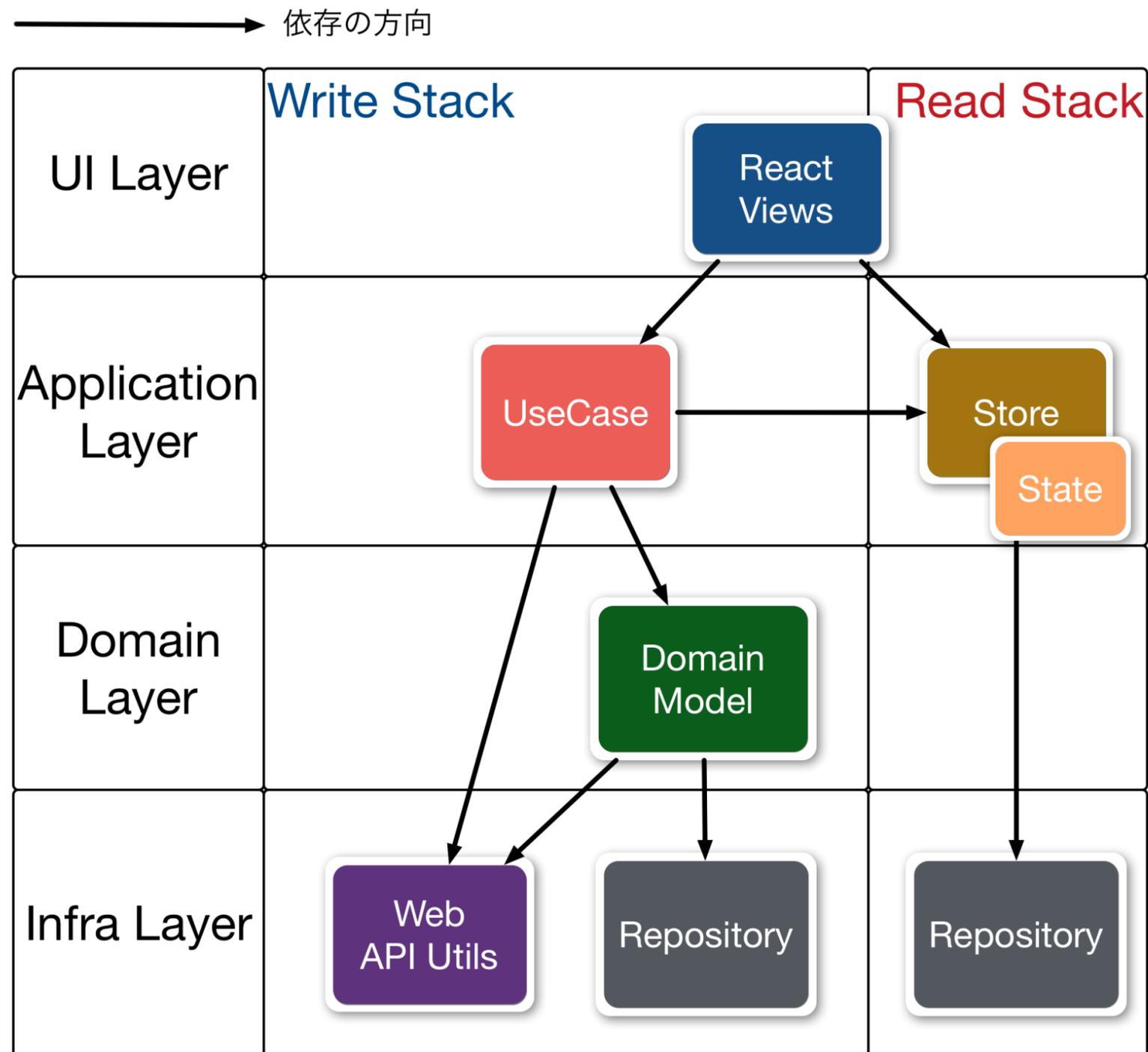
## CQRS



▲ 図 10-1 : Domain Model と CQRS の視覚的な比較

# Write(Command)とRead(Query)の復習

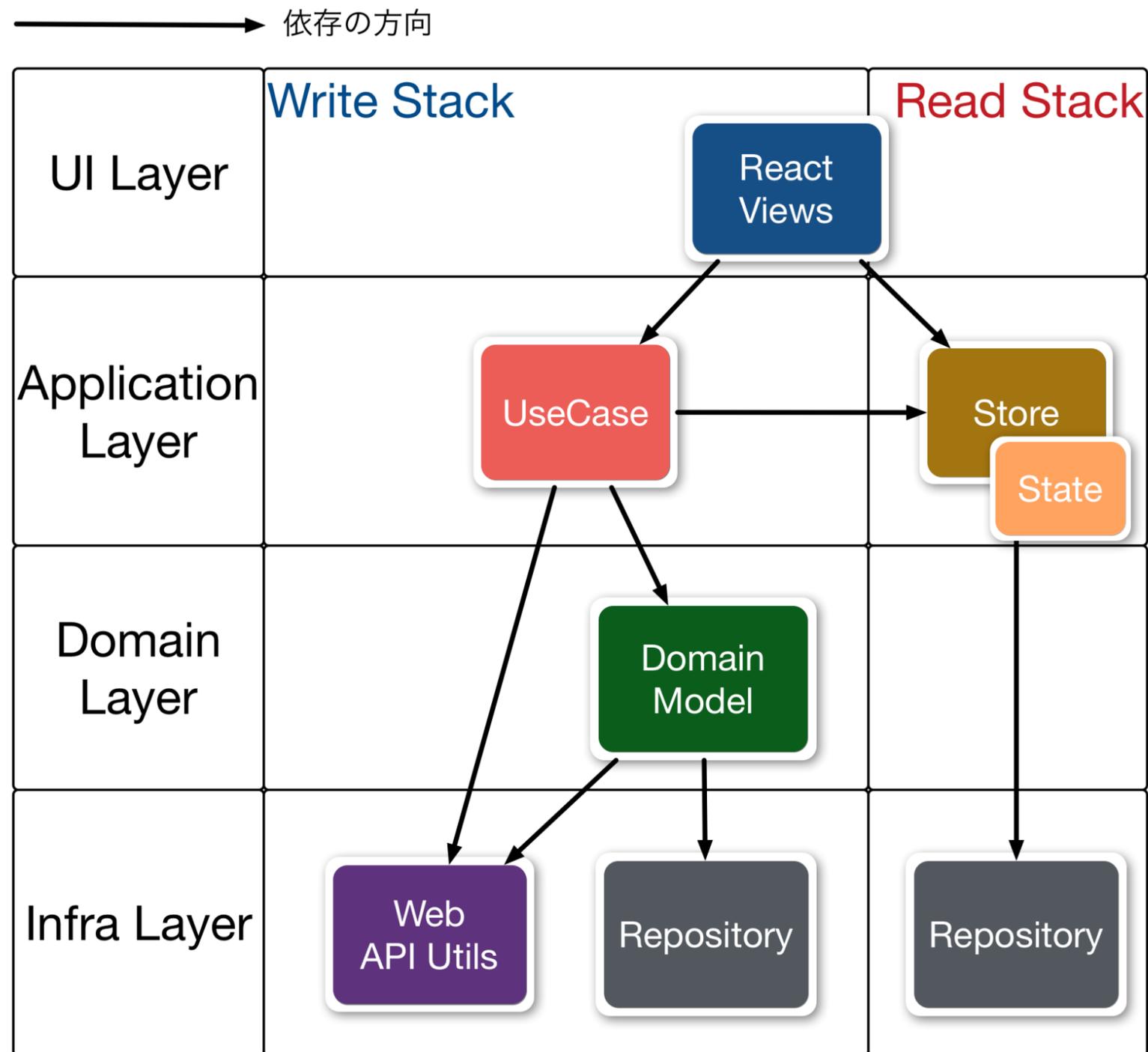
- CQRS (Command Query Responsibility Segregation)
- ざっくり: WriteとReadを層として分けて責務を分離する
- 一方通行のデータフロー
  - UseCaseでドメインを使ってデータ更新(**Write**)
  - Viewの要求に対してデータを返す(**Read**)



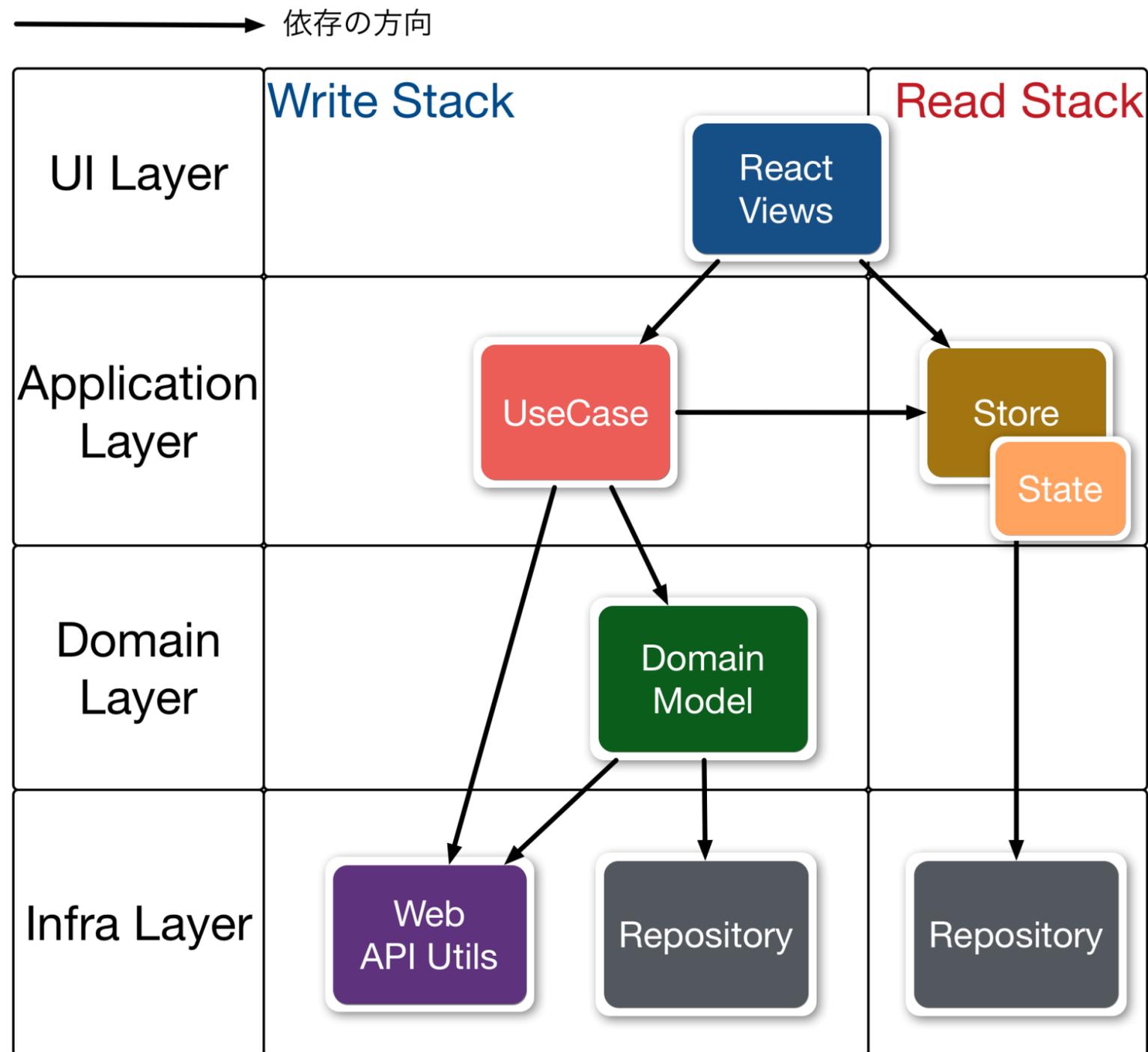
# Read Stack

- ReadはRepository経由でデータを読み込んでView用のデータを作って渡すだけ☒
- 読み取り専用(保存したデータの変更はしない)なので色々簡略化できる
- 縦に別れたので、テスト依存関係が簡略化できる！

☒ Storeは入れ物、Stateは中身という考え方



- Repository
  - Write Stackと同じものを参照するでも良い
- State(Read Modelとか言われる)
  - Writeのドメインから振るまいを消したモデルを作ってもよい
  - ドメインモデル貧血症にわざととしても良い = Viewのためのモデルなので
- Store
  - 実装はFluxのStoreと同じ
  - Stateを格納してる入れ物という感じ



# Store

- StoreはStateを持つオブジェクト☒
- StateをUIに渡してUIはそれを使って更新する
- StoreはStateが更新された事をUIに伝える

☒ Storeは入れ物、Stateは中身という考え方

# ⚠ クライアントサイドの問題 ⚠

永続化するパターン

View -> UseCase -> Domain -> Repository -> State -> View -> ...

Stateを直接更新するパターン

View -> UseCase -> State -> View -> ...

# クライアントサイドで多い問題

- UseCase -> Repositoryを経由したStore/Stateの更新までの流れ
- クライアントサイドではStateを直に更新して、UIにすぐ反映されて欲しいことがある
  - 1F以内にアクションがViewに反映されて欲しいケース
  - ローディング、モーダル、アニメーション、停止ボタン
  - 「ほんのいつとき」が許されないケースはクライアントサイドにはある
  - コンポーネントに閉じ込めるというのあり
- そのため縦(Read/Writeの層)じゃなくて、横のルールも必要

クエリデータベースとコマンドデータベースが同期していない状態では、プレゼンテーション層が古いデータを表示するかもしれませんし、システム全体の整合性が完全に確保されなくなります。データベースの整合性は何らかのタイミングで確保されますが、常に保証されるわけではありません —— これを**結果整合性**と呼びます。

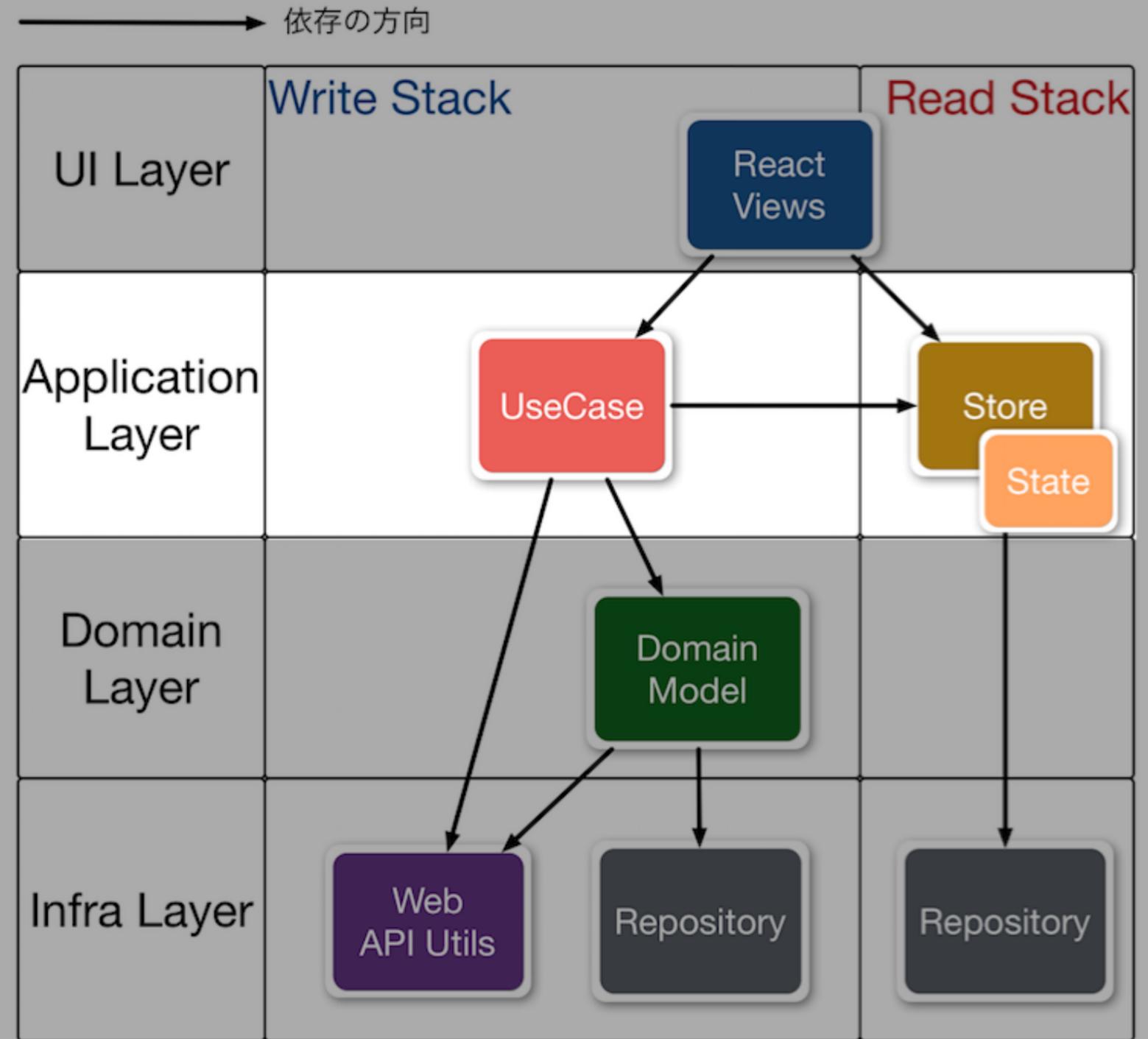
古くなったデータを扱うことが問題かどうかは、状況次第です。

何よりもまず、古いデータを使用するからには理由があるはずです。多くの場合は、スケーラビリティを向上させるために書き込みアクションを高速化することが理由となります。スケーラビリティが重要でなければ、古いデータと結果整合性に甘んじる理由はないでしょう。ましてや、ほんのいつとき古いデータを表示することが許されないアプリケーションなどほとんど存在しないはずです。もっとも、「ほんのいつとき」がどれくらいかは状況によります。

via [.NETのエンタープライズアプリケーションアーキテクチャ](#) 第2版 p299

# UseCase -> Store

- UseCaseからdispatchしたイベントが、Storeに届く横のルート
  - 抜け穴感があるので慎重に取り扱いたい
- FluxやReduxはこのルートが基本的な流れ



# Single source of truth

- Alminでも基本的にはSingle source of truth
- StoreをまとめるStoreGroupという概念を持つてる
  - 一つのアプリはStoreはたくさん存在する
  - Storeが同期的に一斉にemitChangeすると、何回もUIが更新されてしまう
  - StoreGroupは同時に発生したemitChangeを一つにまとめる
    - requestAnimationFrameなどで間引く or UseCaseの実行が終わったら確認する
  - イベントを間引く役 = UI層に近い

# 実装したもの

- [Almin.js = almin.js.org](http://almin.js.org) 
  - このスライドで書いた内容大体そのまま実装
  - Counter
  - TodoMVC
  - Shopping Cart

# まとめ

- Fluxと呼ばれてるものが、CQRSとどのような点で同じで異なるのかを示した
- イベントソーシングは抜いてCQRSについて考えAlminを実装した
- ドメイン/ビジネスロジックをちゃんと考えて実装できるような状況を作った
- コアドメインについてはちゃんと考えて、相談しながら作らないといけない

# まとめ

- アプリケーションの種類毎に適切なアーキテクチャは異なる
- アーキテクチャが良くできていても、ステートフルなDOMという巨大なモデルとの戦いは存在する
- Reactでは吸収できない状態はある `<audio>`とか`<video>`とか`<canvas>`
- 万能なアーキテクチャは存在しない

# まとめ

- 今回はイベントソーシングではなくステートソーシング
- 複雑なものをイベントソーシングで上手くやるイメージがまだない
- Entity自体はImmutableで実装した方が良い(Readでのモデルの共有とか考えるなら尚更)

# まとめのまとめ

- 半年間この考えをベースを実践してみての知見まとめ
- [azu/large-scale-javascript](#): 複雑なJavaScriptアプリケーションを作るために考えること
  - コーディングガイドライン
  - 考え方
  - 参考資料などのまとめ

Write Code Thinking :)

# 時系列

- [10分で実装するFlux](#)
- [How to work as a Team](#)
- [JavaScriptのアーキテクチャ](#)
- [Read/Write Stack | JavaScriptアーキテクチャ](#)
- [Almin.js | JavaScriptアーキテクチャ](#)

# 実装

- [Introduction · Almin.js](#)
  - [CQRSなどを考えて作ったライブラリ](#)
- [almin/example/shopping-cart at master · almin/almin](#)
  - [ショッピングカートを実装したもの](#)
- [azu/presentation-annotator](#)
  - [実践的に考えて実装したもの](#)