HOLYJS SPRING 2026

FSD — это беда, спасет только FDA

15.05

Дмитрий Дин

HOLY JS/...

Далее/ × Subquery/

Про меня

Дмитрий Дин

- Евангелист Svelte, Дзена-Python и простого кода
- Автор opensource фреймворков для веба и ботов
- Тревел-блогер и пилот дронов

TeamLead TypeScript, SvelteKit	11.2024
Middle Full-Stack TypeScript, SvelteKit	02.2023
Верстальщик писем (Совместительство)	07.2022
Junior Full-Stack Ruby, RoR + Vue	04.2021
Выпускник (1 поток) Python, Django	09.2016

Далее/

СБЕР МАРКЕТИНГ

education

МИСИС УНИВЕРСИТЕТ

Яндекс Лицей

Про Далее

21
год
на рынке

570+
человек в
команде

1
место в Ruward
Awards

1
место в Adindex
web production

2.2 млрд
Оборот компании
в 2025

50+
Активных
клиентов

3
место в рейтинге
Рунета

5
больших
направлений

Зачем архитектура?

01. Предсказуемость

Каждый разработчик пишет в одной нотации и по одним правилам, таким образом мы минимизурем bus factor

Зачем архитектура?

01. Предсказуемость

Каждый разработчик пишет в одной нотации и по одним правилам, таким образом мы минимизурем bus factor

02. Устойчивость

Новые фичи не ломают старые, изменения вносятся быстро и легко расширяются

Зачем архитектура?

01. Предсказуемость

Каждый разработчик пишет в одной нотации и по одним правилам, таким образом мы минимизурем bus factor

02. Устойчивость

Новые фичи не ломают старые, изменения вносятся быстро и легко расширяются

03. Независимость

Разработчики работают параллельно. Конфликты при слияниях минимизированы

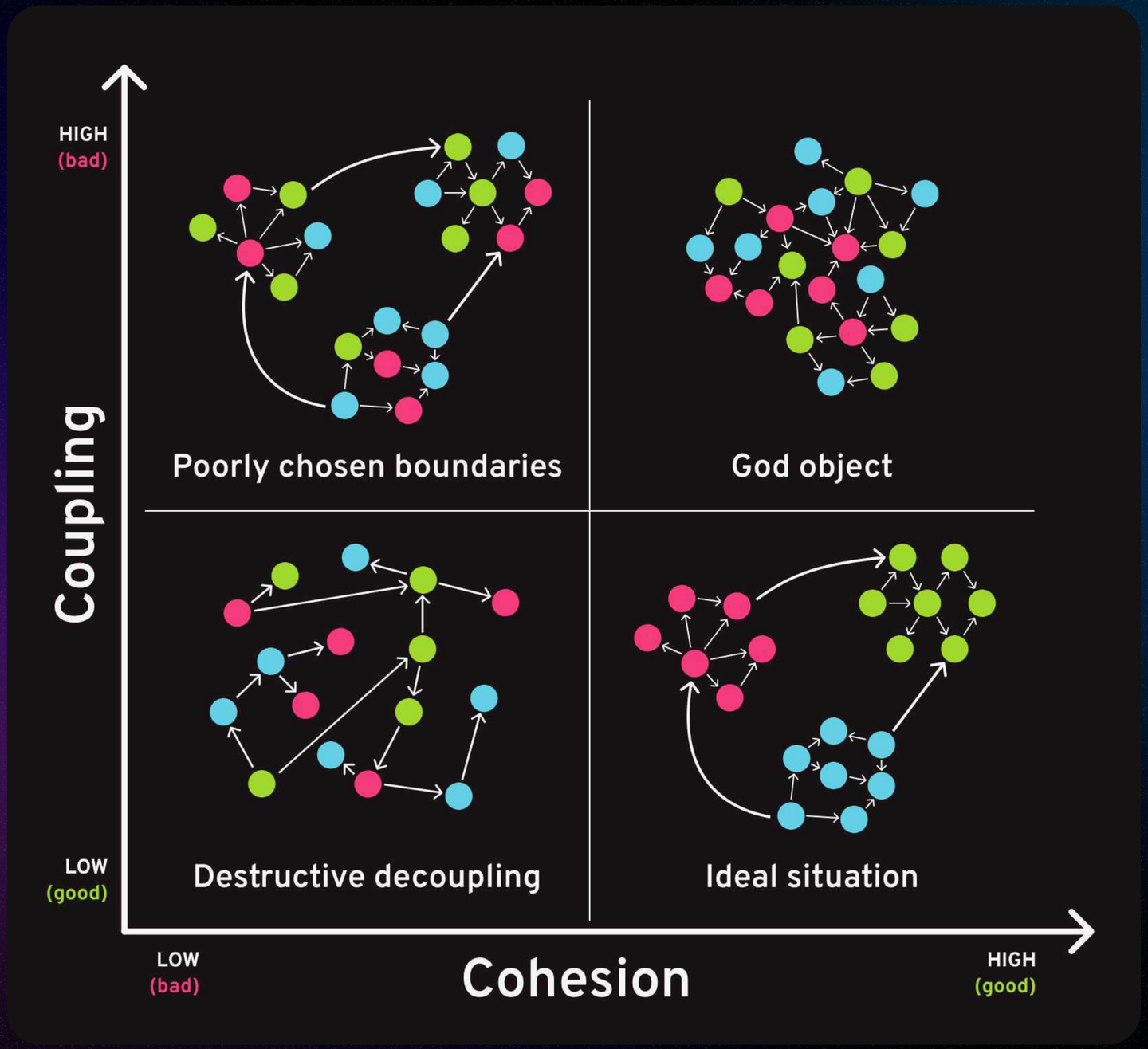
07 —

Определяемся с терминами

Архитектура и структура

Что есть архитектура

Фокусируемся на определении и сужаем контекст



Кординаты архитектур

Скажем, что цветом обозначена бизнес-логика

Ваше мнение может не совпадать, это нормально

О чем говорим

Код

Сервисы

FSD

Clean

Микросервисы

Hexagonal

Микрофронтенды

MVC

DDD

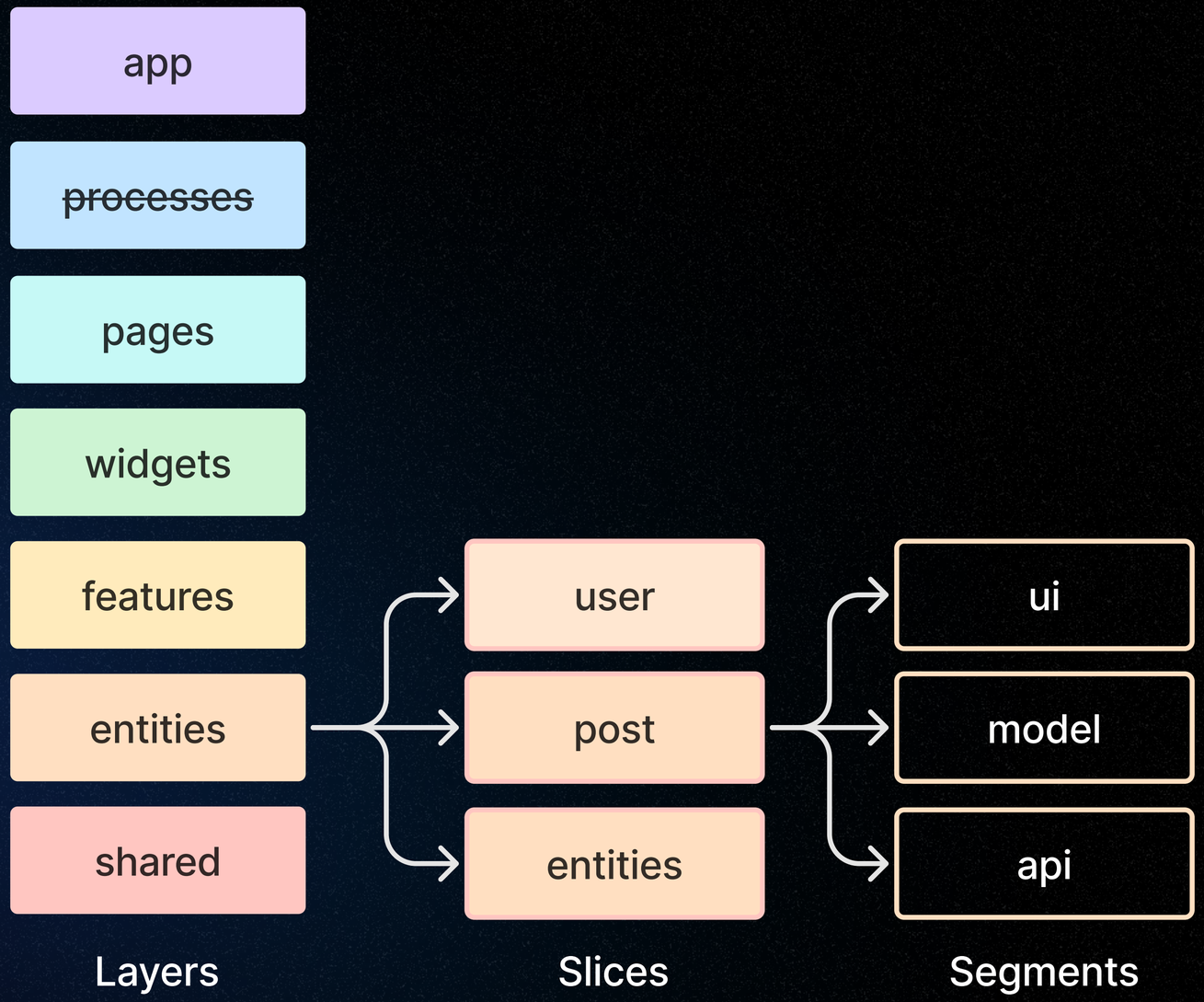
Монолит

Уровень абстракции

FSD

Feature Sliced Design

FSD — Feature Sliced Design



Причины популярности

1

React + SPA

Популяризация компонентного подхода и полноценное отделение от бекенда

2

Первенец

Рост сложности веб-приложений и отсутствие общепризнанных вариантов

3

Маркетинг

Большое сообщество вокруг React способствовало развитию и популяризации

Причины популярности

1

React + SPA

Популяризация компонентного подхода и полноценное отделение от бекенда

2

Первенец

Рост сложности веб-приложений и отсутствие общепризнанных вариантов

3

Маркетинг

Большое сообщество вокруг React способствовало развитию и популяризации

Причины популярности

1

React + SPA

Популяризация компонентного подхода и полноценное отделение от бекенда

2

Первенец

Рост сложности веб-приложений и отсутствие общепризнанных вариантов

3

Маркетинг

Большое сообщество (СНГ) вокруг React способствовало развитию и популяризации

Эволюция FSD

v0.1 - 2018

Самая ранняя версия, наследует Atomic Design
Есть два слоя:

- ui/ - чистые компоненты
- features/ - бизнес логика

v1.0 - 2020

Идеи файлового роутинга

- api/ - сущности и запросы
- features/ - фичи с моделями
- lib/ - utils, helpers
- pages/ - файловый роутинг
- ui/ - чистые компоненты

v2.0 - 2023

app/ > ~~processes/~~ > pages/ > features/ > entities/ > shared/

Более четкое разделение слоев. Требования к зависимостям

v2.1 - 2024

Back to the Past
Снова возвращаемся к страничной модели, вместо композиции “атомов”

Эволюция FSD

v0.1 - 2018

Самая ранняя версия, наследует Atomic Design
Есть два слоя:

- ui/ - чистые компоненты
- features/ - бизнес логика

v1.0 - 2020

Идеи файлового роутинга

- api/ - сущности и запросы
- features/ - фичи с моделями
- lib/ - utils, helpers
- pages/ - файловый роутинг
- ui/ - чистые компоненты

v2.0 - 2023

app/ > ~~processes/~~ > pages/ > features/ > entities/ > shared/

Более четкое разделение слоев. Требования к зависимостям

v2.1 - 2024

Back to the Past
Снова возвращаемся к страничной модели, вместо композиции “атомов”

Эволюция FSD

v0.1 - 2018

Самая ранняя версия, наследует Atomic Design
Есть два слоя:

- ui/ - чистые компоненты
- features/ - бизнес логика

v1.0 - 2020

Идеи файлового роутинга

- api/ - сущности и запросы
- features/ - фичи с моделями
- lib/ - utils, helpers
- pages/ - файловый роутинг
- ui/ - чистые компоненты

v2.0 - 2023

app/ > ~~processes/~~ > pages/ > features/ > entities/ > shared/

Более четкое разделение слоев. Требования к зависимостям

v2.1 - 2024

Back to the Past
Снова возвращаемся к страничной модели, вместо композиции “атомов”

Эволи

v0.1 - 2018

Самая ранняя версия, наследует Atomic Design
Есть два слоя:

- ui/ - чистые компоненты
- features/ - бизнес логика

v2.1 - 2024

Back to the Past
Снова возвращаемся к страничной модели, вместо композиции “атомов”

Эволюция FSD

v0.1 - 2018

Самая ранняя версия, наследует Atomic Design
Есть два слоя:

- ui/ - чистые компоненты
- features/ - бизнес логика

v1.0 - 2020

Идеи файлового роутинга

- api/ - сущности и запросы
- features/ - фичи с моделями
- lib/ - utils, helpers
- pages/ - файловый роутинг
- ui/ - чистые компоненты

v2.0 - 2023

app/ > ~~processes/~~ > pages/ > features/ > entities/ > shared/

Более четкое разделение слоев. Требования к зависимостям

v2.1 - 2024

Back to the Past
Снова возвращаемся к страничной модели, вместо композиции “атомов”

Плюсы FSD

1

Первая (frontend) архитектура

Мотивирует задуматься о необходимости архитектуры в проекте и выработать насмотренность

2

Определения и термины

Вводит понятия абстракций, сущностей, функциональных слоев и правила их взаимодействия

3

Лучше, чем свалка компонентов

На безрыбье и рак рыба, лучше структурировать хаос по хоть каким-то правилам, чем оставлять как есть

Плюсы FSD

1

Первая (frontend) архитектура

Мотивирует задуматься о необходимости архитектуры в проекте и выработать насмотренность

2

Определения и термины

Вводит понятия абстракций, сущностей, функциональных слоев и правила их взаимодействия

3

Лучше, чем свалка компонентов

На безрыбье и рак рыба, лучше структурировать хаос по хоть каким-то правилам, чем оставлять как есть

Плюсы FSD

1

Первая (frontend) архитектура

Мотивирует задуматься о необходимости архитектуры в проекте и выработать насмотренность

2

Определения и термины

Вводит понятия абстракций, сущностей, функциональных слоев и правила их взаимодействия

3

Лучше, чем свалка компонентов

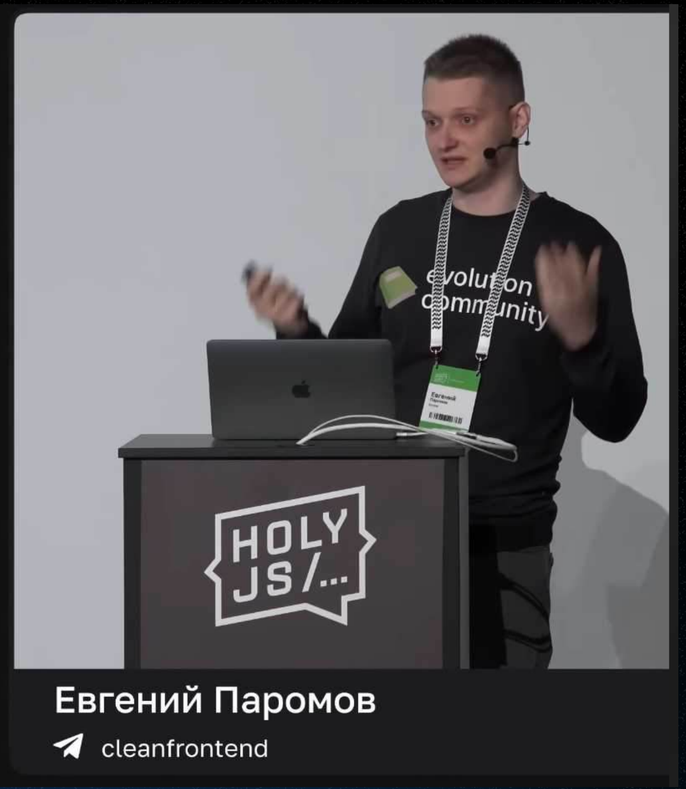
На безрыбье и рак рыба, лучше структурировать хаос по хоть каким-то правилам, чем оставлять как есть

Плюсы FSD

Почему я полюбил FSD

- Миддл с FSD напишет лучше
- Достаточно прост
- Адаптирован под Front-end
- Популяризировал тему архитектуры во Front-end
- Вынес знания об архитектуре на уровень “стека”
  - Поэтому я начал использовать FSD во всех проектах

87 Evrone › Евгений Паромов @paromovevg



3 главных недостатка FSD после 3 лет использования

https://www.youtube.com/watch?v=yRH0O4Fn53U

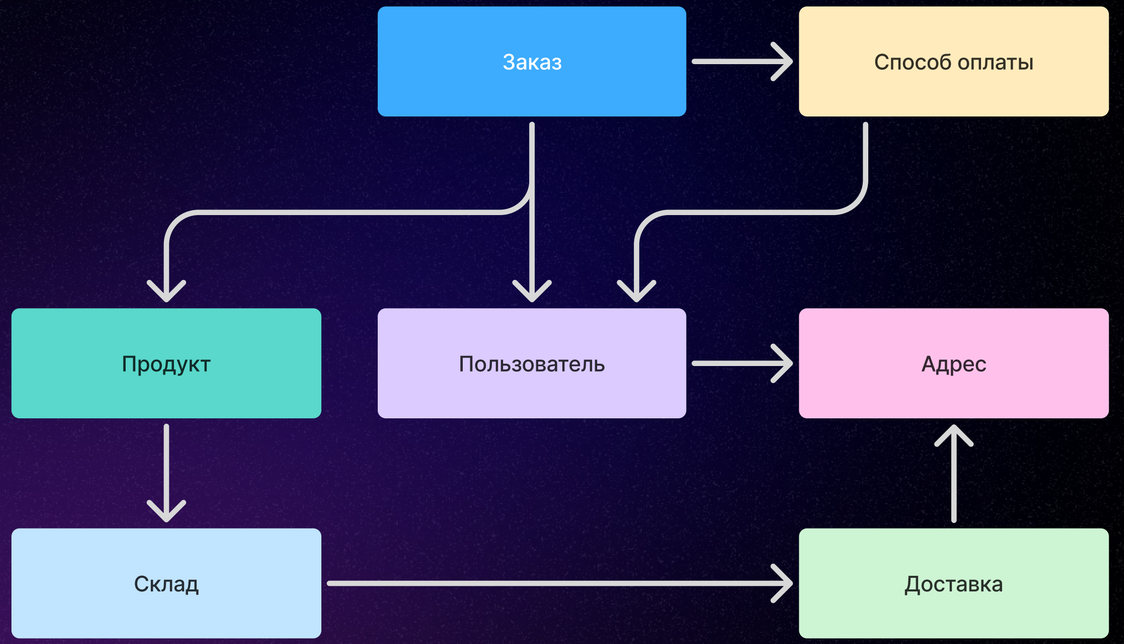
Реальные проблемы

Палки в колеса? А может лучше ненужные абстракции?

кф “Лемони Сникет: 33 несчастья”

Отсутствие кросс-связей

Бизнес логика всегда имеет связи между сущностями и процессы для них

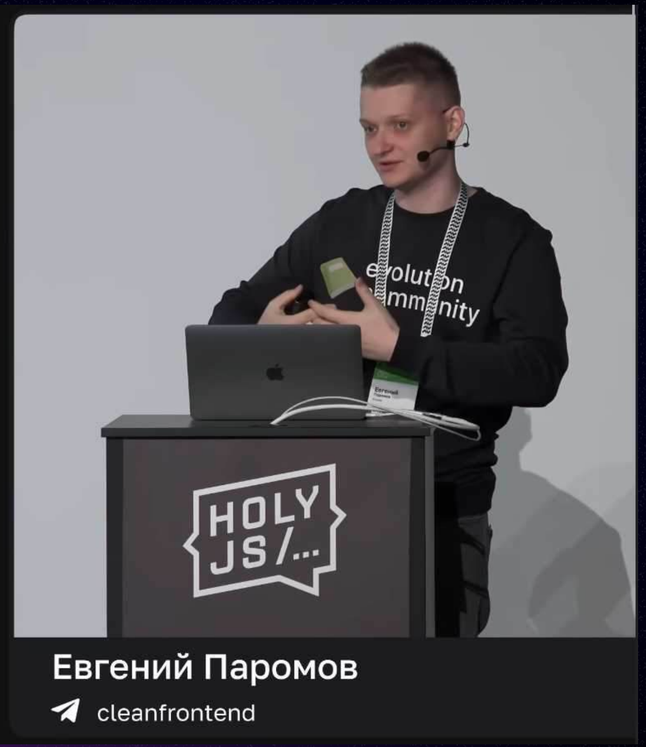


Такое без костылей в FSD не построить, потому что запрещены кросс-импорты одного слоя

Отсутствие кросс-связей

К чему это приводит?

- У людей когнитивный диссонанс
- Entities - описание бизнес сущностей
- Но бизнес сущности **всегда связаны**
  - FSD вынуждает нас провести границу **между плотно связанными модулями**



3 главных недостатка FSD после 3 лет использования

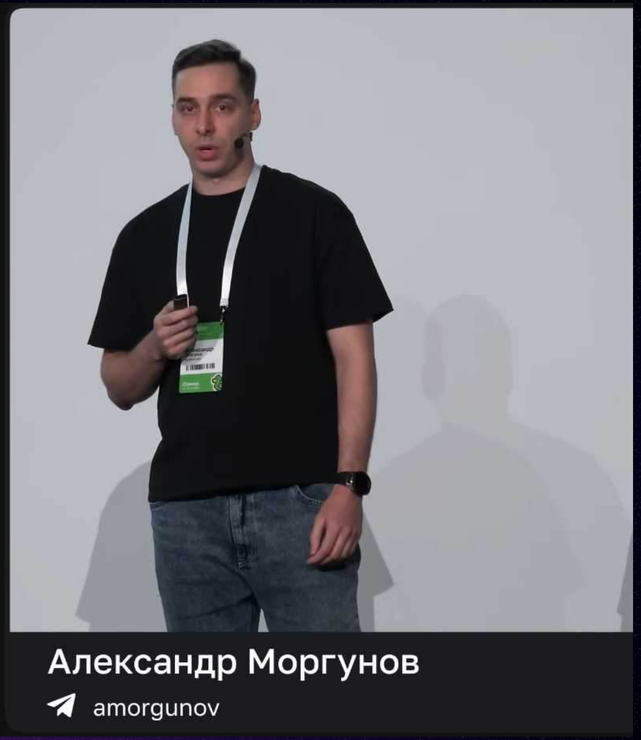
https://www.youtube.com/watch?v=H_rJ0zB8rqc

Отсутствие кросс-связей

1 - Кросс импорты

- Разрешить прямые кросс импорты 👎
- Выносить общее между двумя слайсами на верхлежащие слои (вернемся дальше) 👍👎
- Использовать паттерн инверсии зависимостей (*DIC* или *service locator*) 👍
- Для вьюх использовать паттерн *render slot* 👍
- Явное указание прямых зависимостей между слайсами (@x)

58



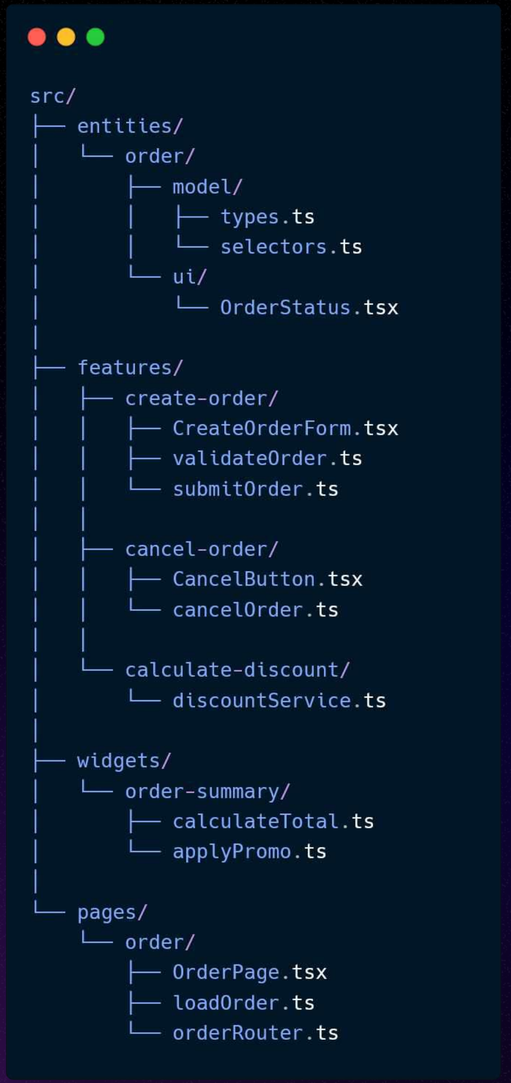
Разбираемся в Feature-Sliced Design

https://www.youtube.com/watch?v=H_rJ0zB8rqc

High Coupling & Low Cohesion

Логика сильно размазывается по слоям при этом остается зависимой друг от друга и связной

Простой процесс работы пользователя с заказом превратился в спагетти-папки

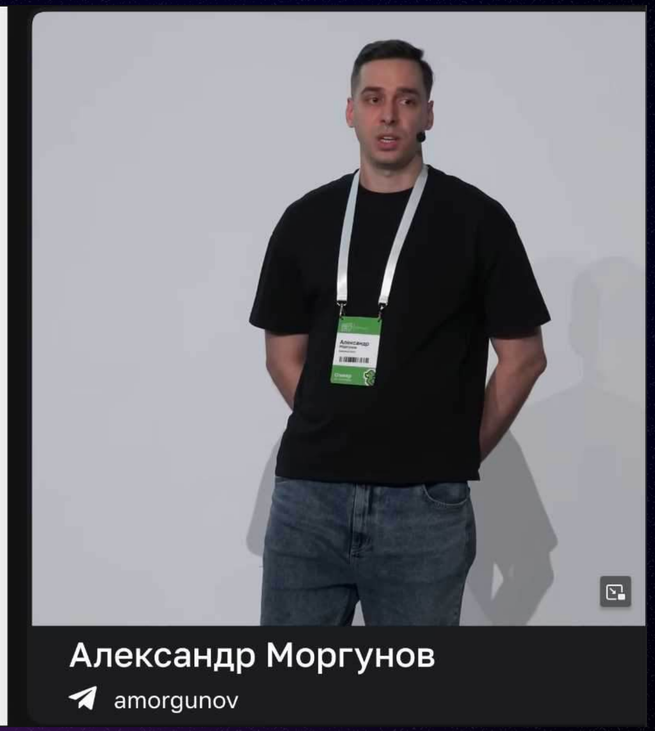


High Coupling & Low Cohesion

2 - Размазывание кода по слоям

- Сильное размазывание кода одного модуля по проекту приводит к *destructive decoupling*
- Для решения использовать *local-first* паттерн - все держим рядом, пока это не понадобится где-то еще

81



Разбираемся в Feature-Sliced Design

https://www.youtube.com/watch?v=H_rJ0zB8rqc

Ненужная абстракция

widget или entity или feature

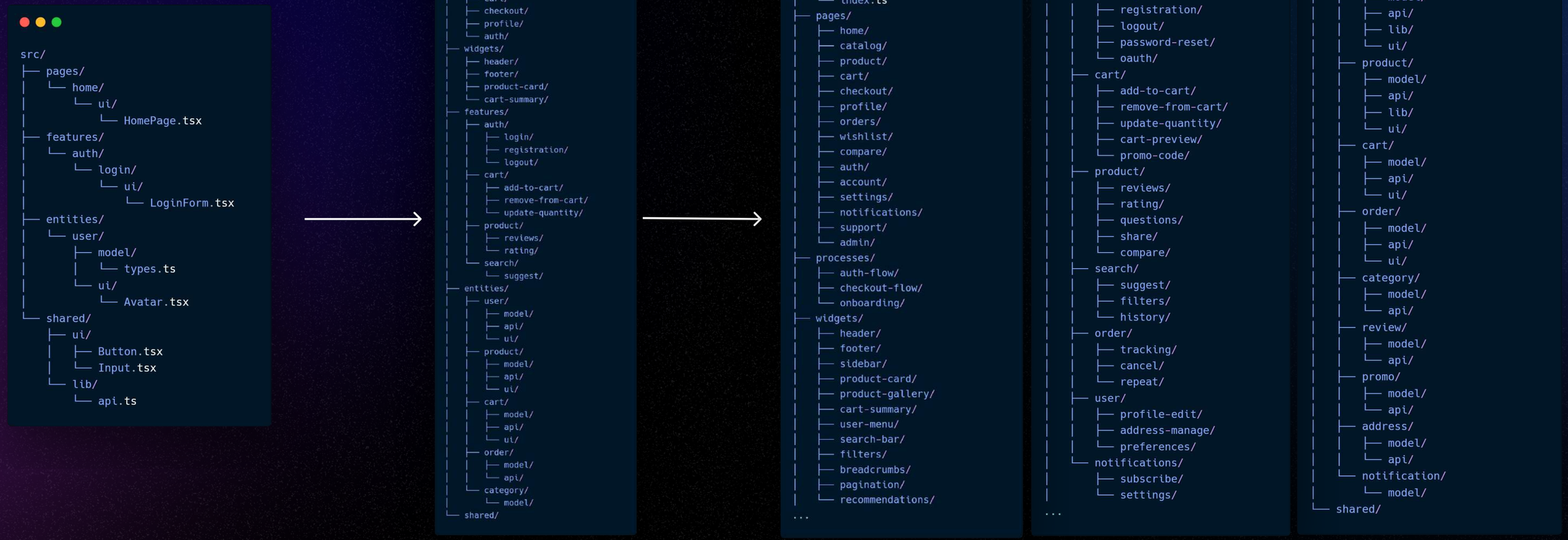
Ненужная абстракция

widget или **entity** или **feature**

80% споров vs **20%** возможно переиспользуемого кода

Где-то грустит брошенный слой processes

Линейный рост сложности



До 5 страниц

До 20 страниц

30+ страниц

Эволюция проекта

Это важно!

36 сек

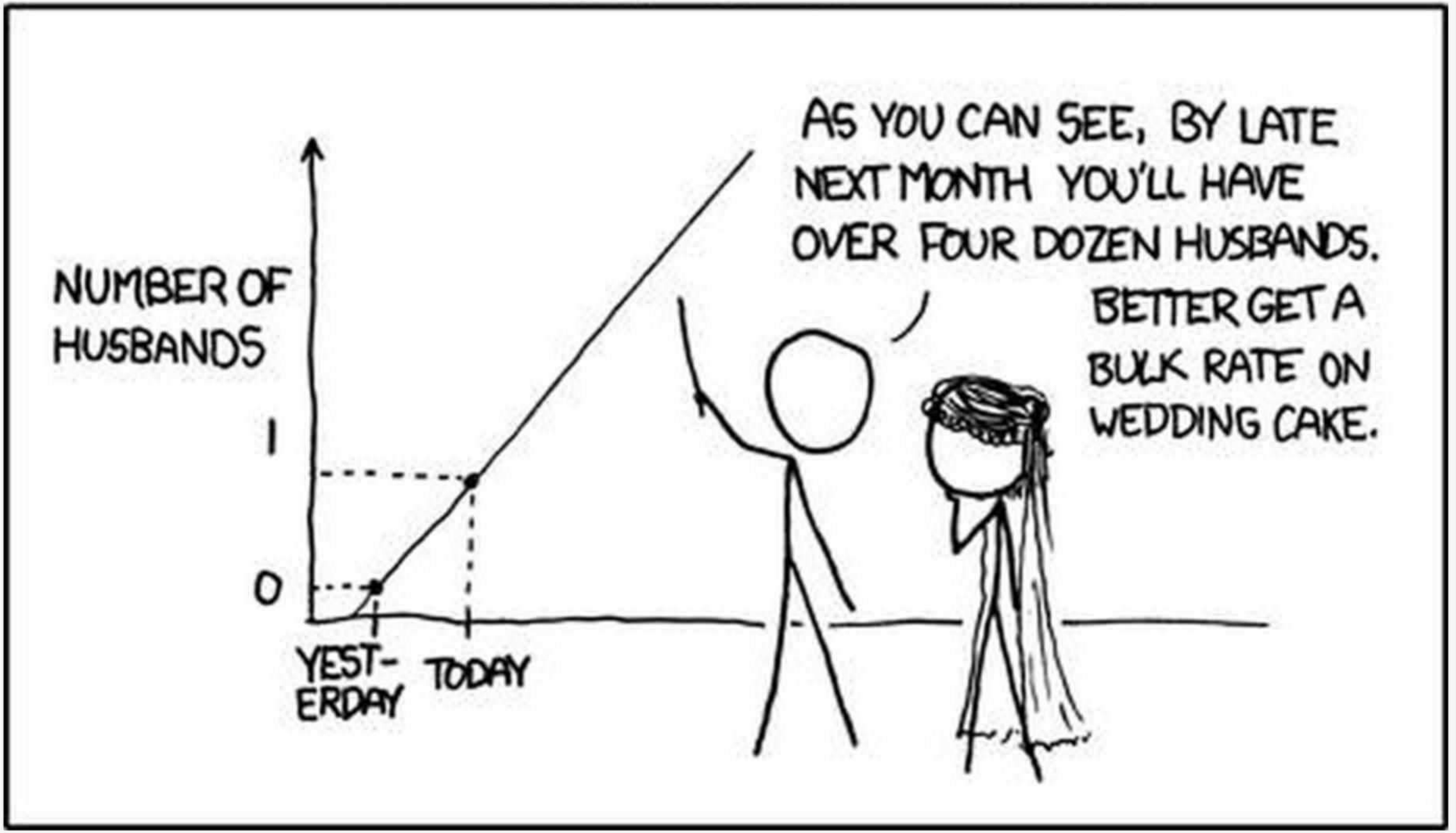
Это важно!

36 сек * 100 навигаций

Это важно!

$$36_{\text{сек}} * 100_{\text{навигаций}} = 1_{\text{час}}$$

MY HOBBY: EXTRAPOLATING



Это важно!

36 сек * 100 навигаций = 1 час

1 час * 5 дней * 4 недели

Это важно!

$$36_{\text{сек}} * 100_{\text{навигаций}} = 1_{\text{час}}$$

$$1_{\text{час}} * 5_{\text{дней}} * 4_{\text{недели}} = \underline{20}_{\text{часов/месяц}}$$

Введем понятие

П(const) и П(n)

Оценим рост проекта при его развитии

Семантика вложености

1

Абстракция П(7)

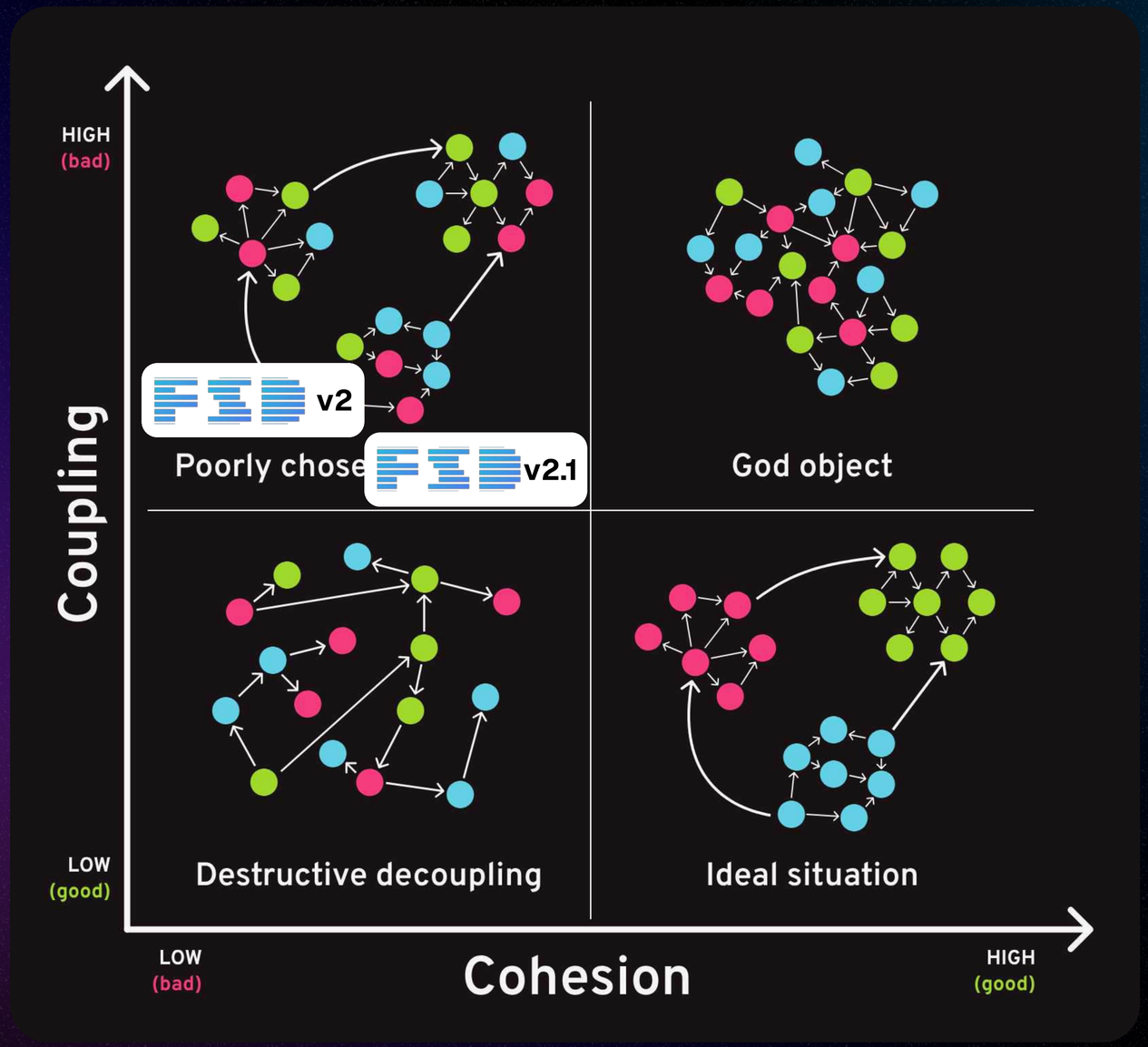
2

Сущность П(3n)

3

Функиональность П(3)

Позиционируем FSD



В итоге мы работаем с архитектурой, которая заметно усложняет связи и код при росте проекта

Это же подкрепляется сложной ментальной моделью, которую каждый интепретирует и трактует по-своему

Ваше мнение может не совпадать, это нормально

Вернемся в прошлое

Подсмотрим чужой опыт

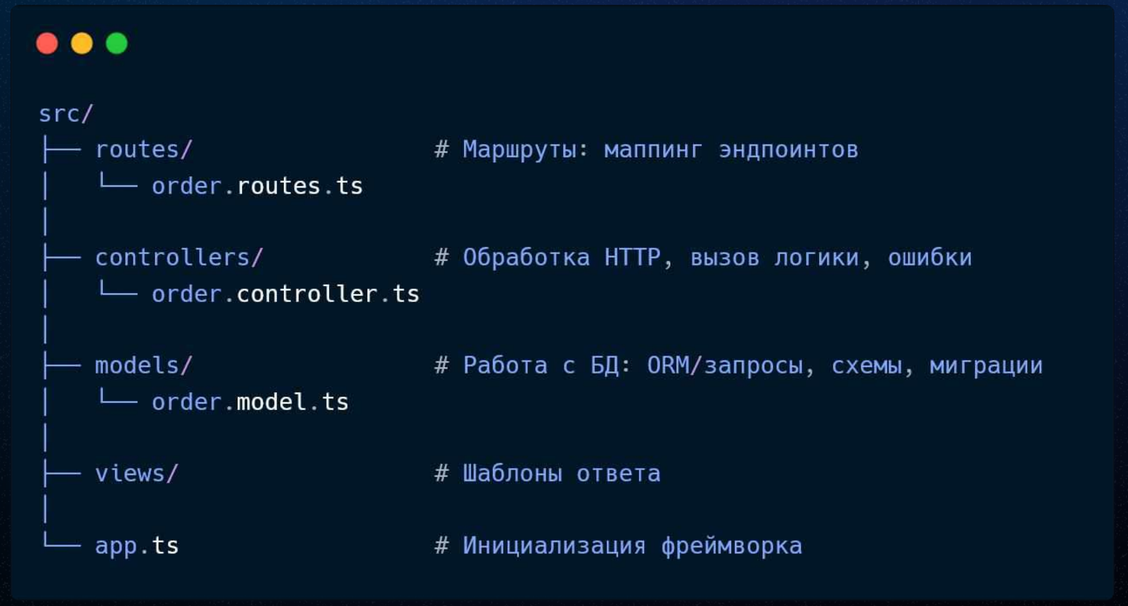
Что там на бекендах

Все хорошее - давно забытое старое.
Короткий экскурс туда и обратно

Про MVC

Базовый паттерн большинства backend-фреймворков (Express, Laravel, Django, Rails, Spring MVC)

Появился в ~1970



Про MVC

Плюсы

- Простота
- Чёткое разделение
- Десятилетия развития и применения
- Идеален для маленьких CRUD-сервисов

Минусы

- Толстый Controller и тесная опора на HTTP
- Контроллеры плохо связываются
- Плохо масштабируется без доп слоев
- Подходит для бекенда, но не для фронта
- Плоская реализация ведет к росту П(3n)

Про MVC

Плюсы

- Простота
- Чёткое разделение
- Десятилетия развития и применения
- Идеален для маленьких CRUD-сервисов

Минусы

- Толстый Controller и тесная опора на HTTP
- Контроллеры плохо связываются
- Плохо масштабируется без доп слоев
- Подходит для бекенда, но не для фронта
- Плоская реализация ведет к росту П(3n)

Про DDD

Бизнес-моделирование в коде.
Применяется в бекендах сложных enterprise систем.
Скорее больше подход, чем архитектура

Появился в 2003

```
src/
├── contexts/
│   ├── sales/                    # Контекст
│   │   ├── domain/               # Бизнес-правила
│   │   │   ├── entities/
│   │   │   ├── value-objects/
│   │   │   ├── aggregates/
│   │   │   ├── events/
│   │   │   └── services/
│   │   │
│   │   ├── application/          # Сценарии использования
│   │   │   ├── commands/
│   │   │   ├── queries/
│   │   │   ├── handlers/
│   │   │   └── dtos/
│   │   │
│   │   ├── infrastructure/       # Реализация внешних зависимостей
│   │   │   ├── persistence/
│   │   │   ├── messaging/
│   │   │   └── external/
│   │   │
│   │   └── interfaces/           # Точки входа в контекст
│   │       ├── http/
│   │       ├── cli/
│   │       └── workers/
│   │
│   └── inventory/                # Другой контекст
│       └── ...
│
└── shared/                       # Общие примитивы
```

Про DDD

Плюсы

- Согласованность терминологии и правил
- Управление сложностью
- Границы доменов
- Идеален для больших enterprise решений

Минусы

- Высокий порог входа
- Требуется domain expert
- Избыточен для простых CRUD сервисов
- Плоская реализация ведет к росту П(n)

Про DDD

Плюсы

- Согласованность терминологии и правил
- Управление сложностью
- Границы доменов
- Идеален для больших enterprise решений

Минусы

- Высокий порог входа
- Требуется domain expert
- Избыточен для простых CRUD сервисов
- Плоская реализация ведет к росту П(n)

Про Clean

Развитие идей Hexagonal и Onion.
Полностью следует SOLID принципам.
Подробно разделяет код по слоям и позволяет их подменять

Появился в 2012

```
src/
── entities/                    # Бизнес-сущности
|   ├── user/
|   ├── order/
|   └── cart/
├── use_cases/                  # Сценарии использования
|   ├── create_order/
|   ├── payment/
|   └── delivery/
|
├── interface_adapters/         # Преобразование данных и управление потоком
|   ├── controllers/            # HTTP контроллеры
|   ├── presenters/             # Форматирование ответа под UI/API
|   └── gateways/               # Интерфейсы для внешних систем
|
├── frameworks_drivers/         # Внешний мир
|   ├── web/                    # Роутеры, middleware, фреймворки
|   ├── persistence/            # Реализации репозиториев
|   ├── external/               # Интеграции 3rd-party API
```

Про Clean

Плюсы

- Полная независимость от стека технологий
- Высокая unit-тестируемость
- Четкие границы и интерфейсы легко подменять реализации
- Идеален для больших решений с зоопарком источников и потребителей

Минусы

- Крутая кривая обучения
- Высокий бойлерплейт
- Подходит для бекенда, но не для фронта
- Масштабирование бизнес-логики зависит от выбранного подхода

Про Clean

Плюсы

- Полная независимость от стека технологий
- Высокая unit-тестируемость
- Четкие границы и интерфейсы легко подменять реализации
- Идеален для больших решений с зоопарком источников и потребителей

Минусы

- Крутая кривая обучения
- Высокий бойлерплейт
- Подходит для бекенда, но не для фронта
- Масштабирование бизнес-логики зависит от выбранного подхода

Выводы

1. Бекенд архитектуры прошли долгую эволюцию и хорошо делят зоны ответственности
2. Принципы SOLID работают в архитектуре также, как в ООП
3. Архитектуры используют бекенд-специфичные слои, такие как репозитории, представления
4. Компоненты и реактивность на фронтенде все меняют

Выводы

1. Бекенд архитектуры прошли долгую эволюцию и хорошо делят зоны ответственности
2. Принципы SOLID работают в архитектуре также, как в ООП
3. Архитектуры используют бекенд-специфичные слои, такие как репозитории, представления
4. Компоненты и реактивность на фронтенде все меняют

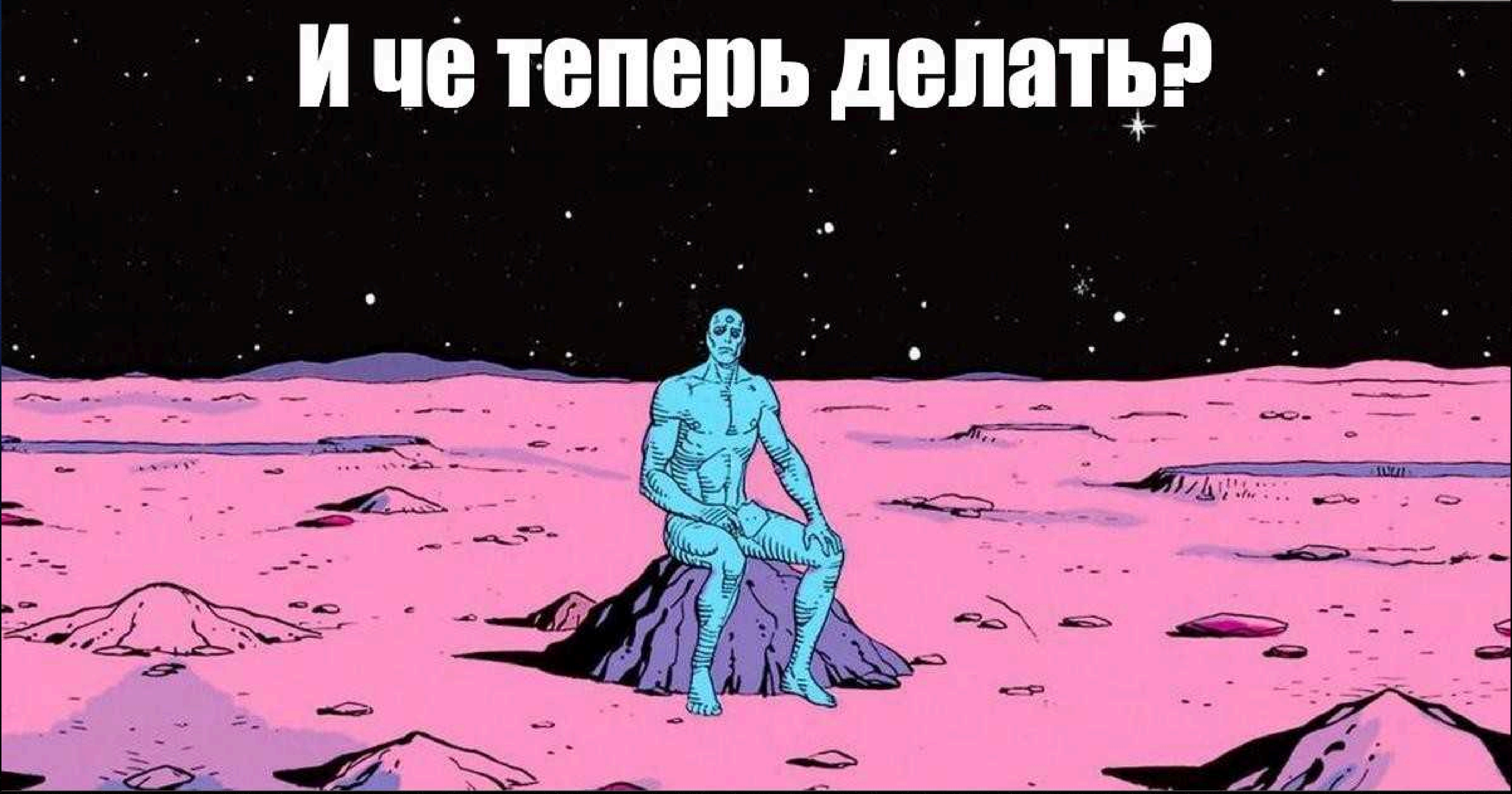
Выводы

1. Бекенд архитектуры прошли долгую эволюцию и хорошо делят зоны ответственности
2. Принципы SOLID работают в архитектуре также, как в ООП
3. Архитектуры используют бекенд-специфичные слои, такие как репозитории, представления
4. Компоненты и реактивность на фронтенде все меняют

Выводы

1. Бекенд архитектуры прошли долгую эволюцию и хорошо делят зоны ответственности
2. Принципы SOLID работают в архитектуре также, как в ООП
3. Архитектуры используют бекенд-специфичные слои, такие как репозитории, представления
4. Компоненты и реактивность на фронтенде все меняют

И че теперь делать?



Немного контекста

1.

2 сервиса — 200к строк

Платина — закрытая аналитическая платформа **SubQuery** - мощный инструмент дата-инженерии

2.

Монолиты на мета-фреймворках

Проекты изначально написаны на **SvelteKit + TypeScript**
Все интеграции и бизнес логика находятся в одном проекте

3.

Full-Stack команда

В нашей команде нет явного разделения на бекенд и фронтенд. Ответственность поделена по **бизнес-логике** сервисов

Немного контекста

1.

2 сервиса — 200к строк

Платина — закрытая аналитическая платформа **SubQuery** - мощный инструмент дата-инженерии

2.

Монолиты на мета-фреймворках

Проекты изначально написаны на **SvelteKit + TypeScript**
Все интеграции и бизнес логика находятся в одном проекте

3.

Full-Stack команда

В нашей команде нет явного разделения на бекенд и фронтенд. Ответственность поделена по **бизнес-логике** сервисов

Немного контекста

1.

2 сервиса — 200к строк

Платина — закрытая аналитическая платформа **SubQuery** - мощный инструмент дата-инженерии

2.

Монолиты на мета-фреймворках

Проекты изначально написаны на **SvelteKit + TypeScript**
Все интеграции и бизнес логика находятся в одном проекте

3.

Full-Stack команда

В нашей команде нет явного разделения на бекенд и фронтенд. Ответственность поделена по **бизнес-логике** сервисов

Fractals & Domains

FSD vs FDA

FDA

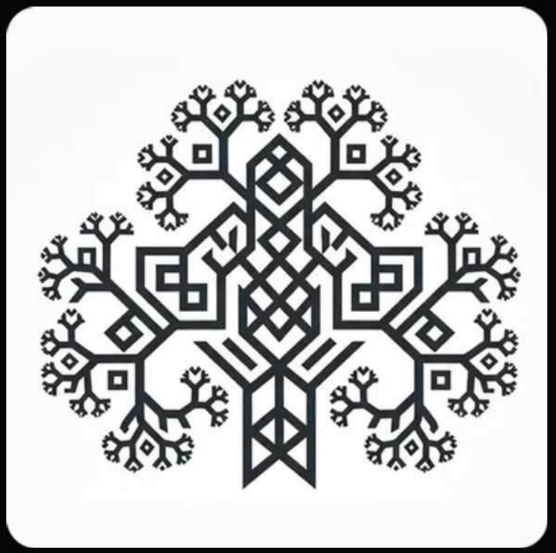
Фрактально-Доменная Архитектура

FDA (Fractal Domain Architecture)

Фрактальная архитектура для современных приложений

Начать изучение →

GitHub репозиторий



Основные принципы

Фрактальность

Одинаковая структура на всех уровнях: компоненты → подмодули → домены → приложения

Разделение на слои

Четкое разделение: репозитории → модель → контроллер → UI

Терминология

Домены

- Задаются требованиями бизнеса
- Прозрачны для разработчиков, менеджеров и заказчиков
- Могут иметь публичное api для кросс-доменных связей
- Определяют структуру приложения

Слои

- Задаются разработчиками
- Расширяются по мере возрастания функциональной сложности приложения
- Данные перетекают как правило последовательно от слоя к слою
- Отвечают за архитектуру внутри доменов

Терминология

Домены

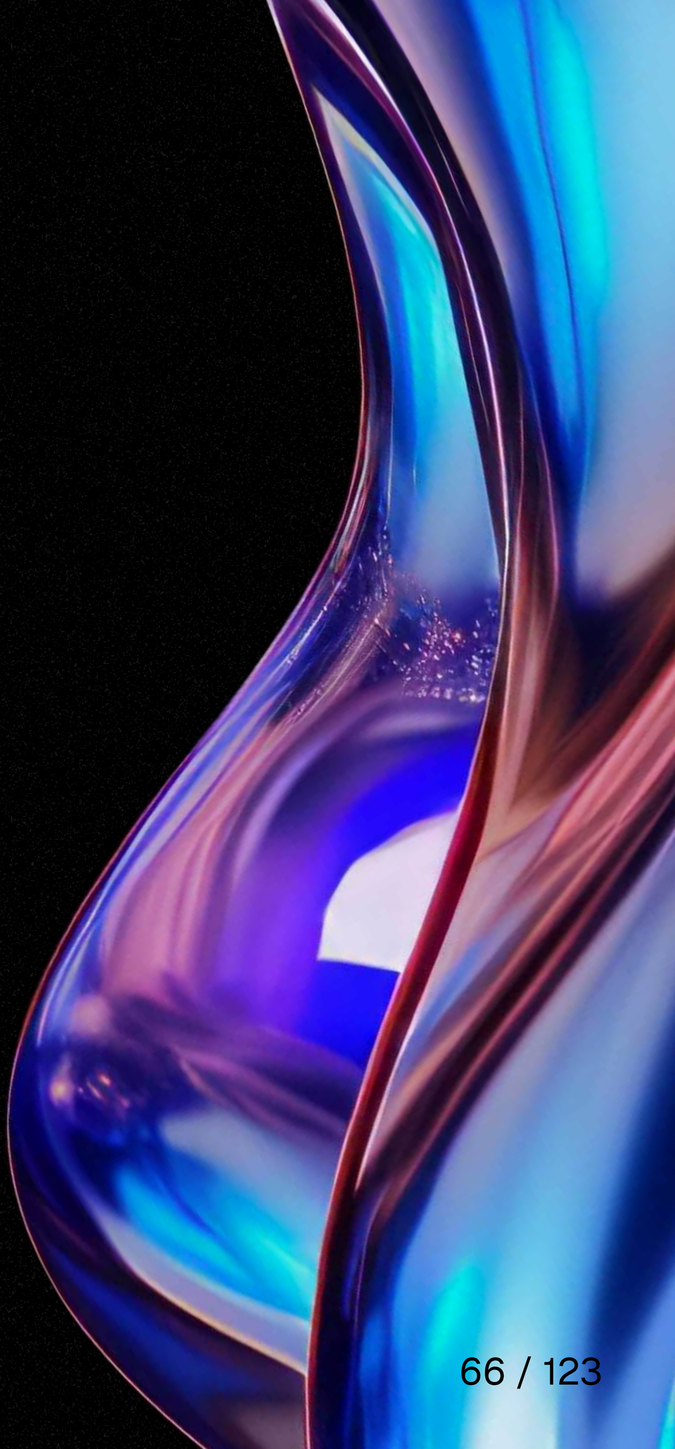
- Задаются требованиями бизнеса
- Прозрачны для разработчиков, менеджеров и заказчиков
- Могут иметь публичное api для кросс-доменных связей
- Определяют структуру приложения

Слои

- Задаются разработчиками
- Расширяются по мере возрастания функциональной сложности приложения
- Данные перетекают как правило последовательно от слоя к слою
- Отвечают за архитектуру внутри доменов

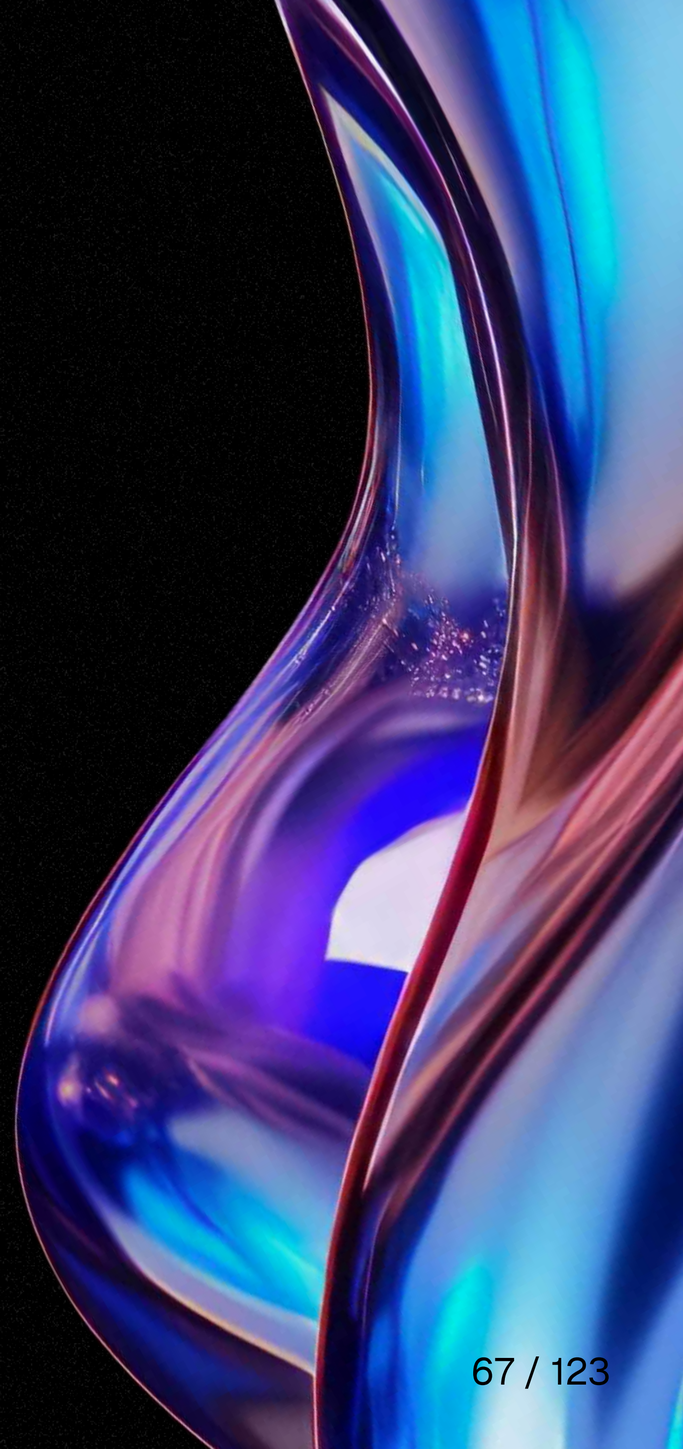
Принципы

1. Две категории для кода - функциональные слои и бизнес логика
2. Бизнес логика должна быть инкапсулирована и определять домены
3. Домены содержат функциональные слои и поддомены
4. Module first подход



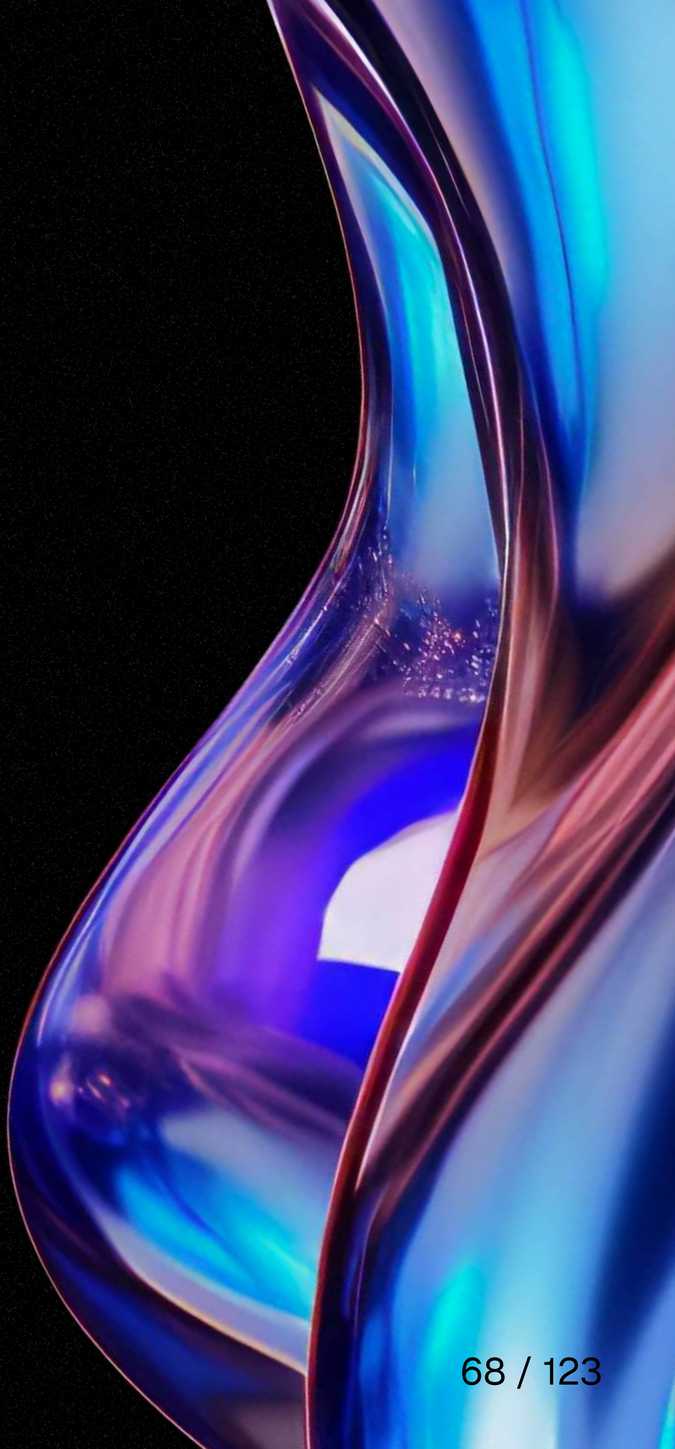
Принципы

1. Две категории для кода - функциональные слои и бизнес логика
2. Бизнес логика должна быть инкапсулирована и определять домены
3. Домены содержат функциональные слои и поддомены
4. Module first подход



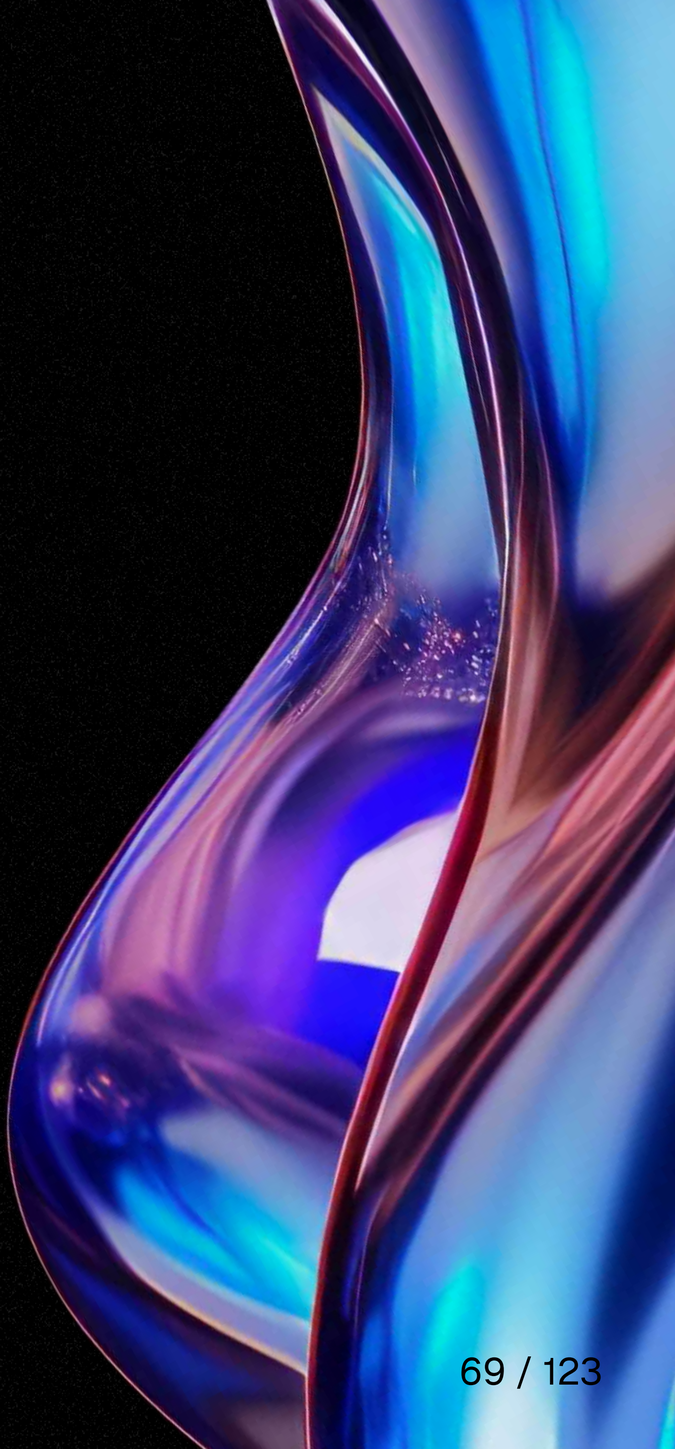
Принципы

1. Две категории для кода - функциональные слои и бизнес логика
2. Бизнес логика должна быть инкапсулирована и определять домены
3. Домены содержат функциональные слои и поддомены
4. Module first подход



Принципы

1. Две категории для кода - функциональные слои и бизнес логика
2. Бизнес логика должна быть инкапсулирована и определять домены
3. Домены содержат функциональные слои и поддомены
4. Module first подход




Да, я слышал про теорию разделения властей
Разделяй и властвуй

Что есть домен?

Модуль

Совокупность сущностей и страниц. Можно вынести в микросервис

Сущность

Совокупность взаимодействий и операций над объектом

Страница

Совокупность компонентов для действий пользователя

Компонент

Отображение и стили, методы, работа с данными пользователя. Слоты

Гранулярность

Что есть домен?

Модуль

Совокупность сущностей и страниц. Можно вынести в микросервис

Сущность

Совокупность взаимодействий и операций над объектом

Страница

Совокупность компонентов для действий пользователя

Компонент

Отображение и стили, методы, работа с данными пользователя. Слоты

Гранулярность

Что есть домен?

Модуль

Совокупность сущностей и страниц. Можно вынести в микросервис

Сущность

Совокупность взаимодействий и операций над объектом

Страница

Совокупность компонентов для действий пользователя

Компонент

Отображение и стили, методы, работа с данными пользователя. Слоты

08 —

Что есть домен?

Модуль

Совокупность сущностей и страниц. Можно вынести в микросервис

Сущность

Совокупность взаимодействий и операций над объектом

Страница

Совокупность компонентов для действий пользователя

Компонент

Отображение и стили, методы, работа с данными пользователя. Слоты

Гранулярность

Теперь о слоях

Модель

Работа с сущностями БД. Использует ORM или коннекторы к репозиториям

Эндпоинты

Это контроллеры из MVC Обработка запросов, хендлинг ошибок

Реактивность

Обработка входящих данных и действий пользователя

Отображение

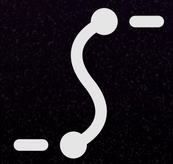
Интерфейс с верстской и стилями на вашем любимом фреймворке

Поток данных

Теперь о слоях

Модель

Работа с сущностями БД.
Использует ORM или
коннекторы к репозиториям



Эндпоинты

Это контроллеры из MVC
Обработка запросов,
хендлинг ошибок

Реактивность

Обработка входящих
данных и действий
пользователя

Отображение

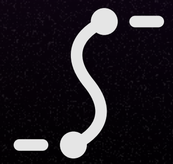
Интерфейс с верстkой и
стилями на вашем
любимом фреймворке

Поток данных

Теперь о слоях

Модель

Работа с сущностями БД. Использует ORM или коннекторы к репозиториям



Эндпоинты

Это контроллеры из MVC Обработка запросов, хендлинг ошибок

Реактивность

Обработка входящих данных и действий пользователя

Отображение

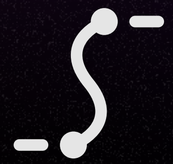
Интерфейс с верстккой и стилями на вашем любимом фреймворке

Поток данных

Теперь о слоях

Модель

Работа с сущностями БД.
Использует ORM или
коннекторы к репозиториям



Эндпоинты

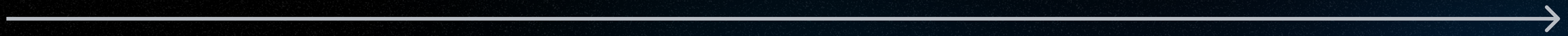
Это контроллеры из MVC
Обработка запросов,
хендлинг ошибок

Реактивность

Обработка входящих
данных и действий
пользователя

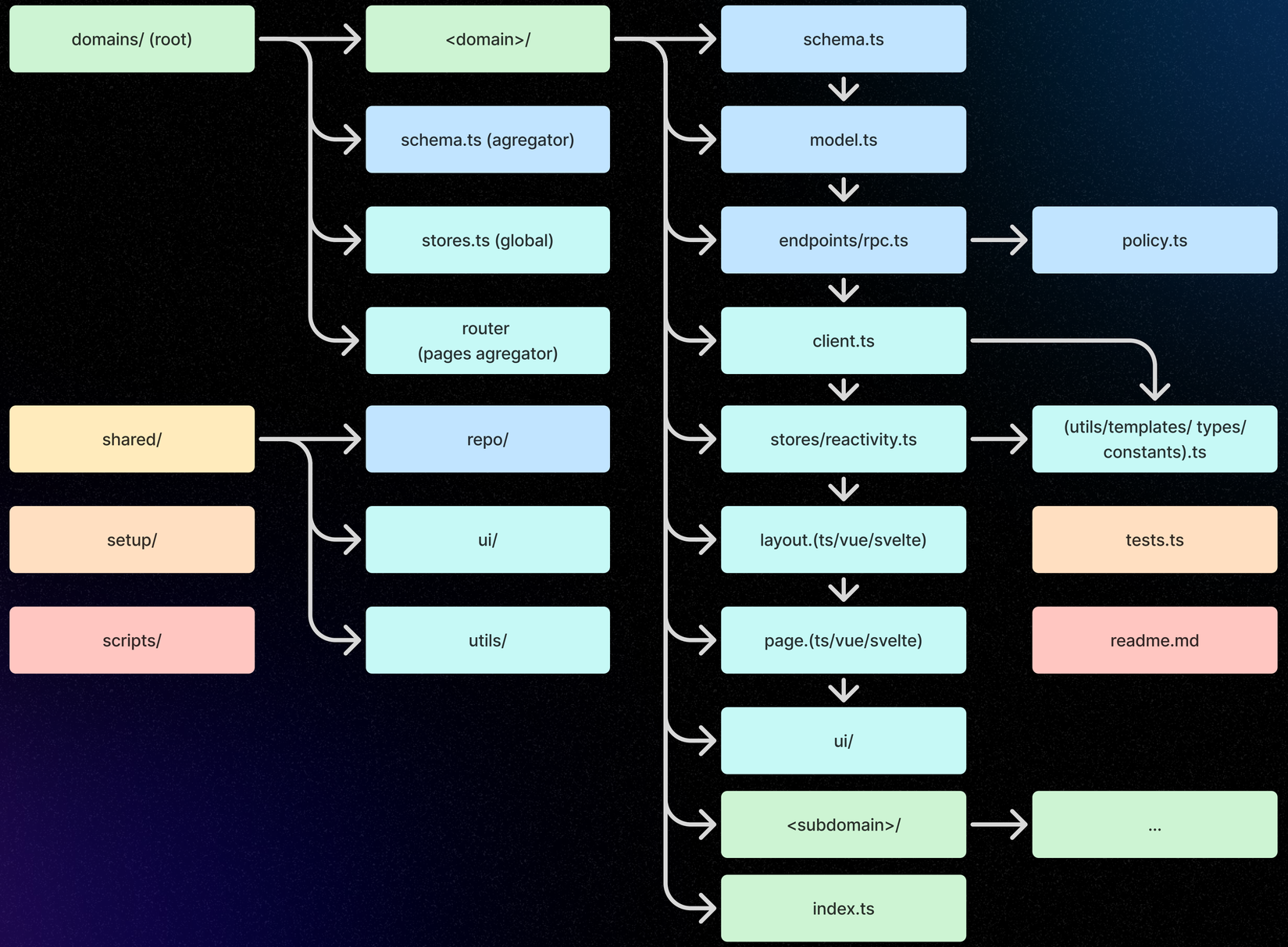
Отображение

Интерфейс с верстkой и
стилями на вашем
любимом фреймворке



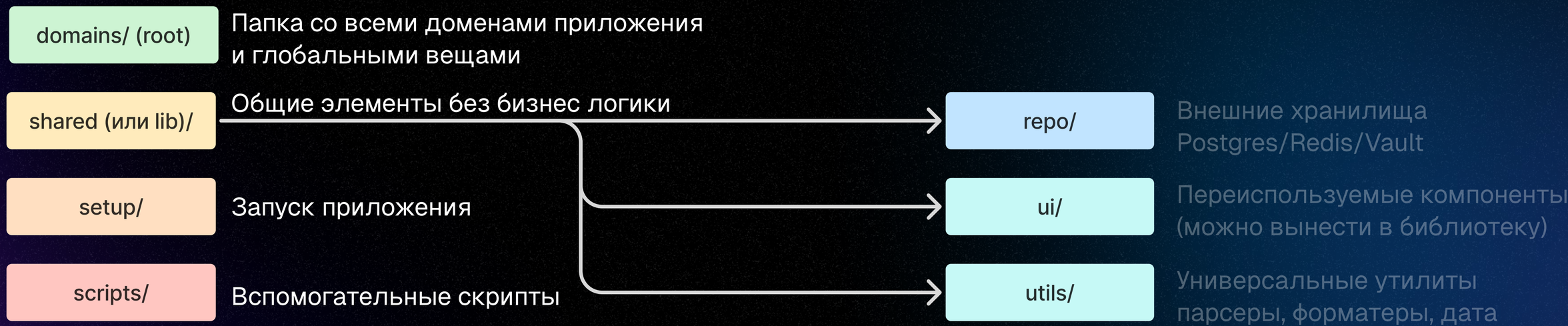
Поток данных

Как это выглядит

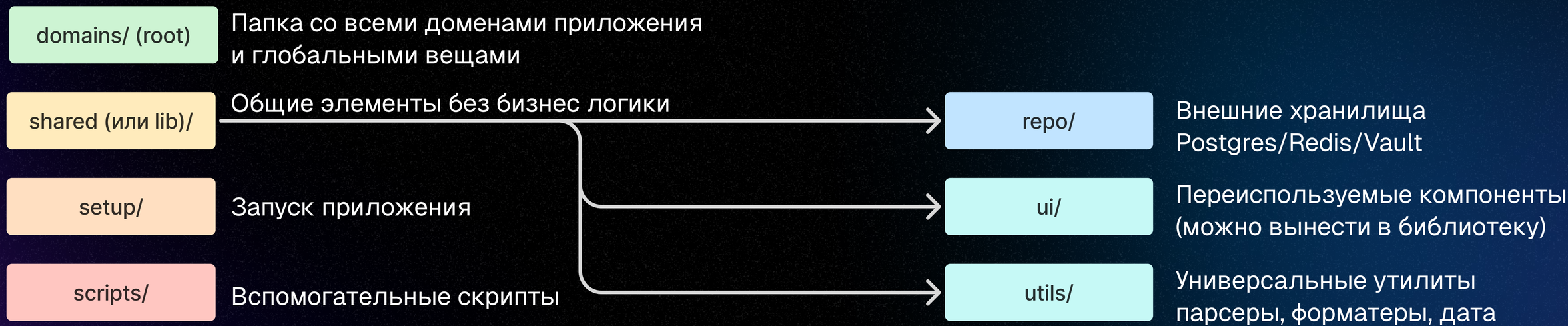



Бу! Испугался не бойся я друг я
тебе не обижу.

Верхний уровень



Верхний уровень



Первый уровень доменов

Список всех доменов приложения (название папки = название домена)

Агрегатор всех схем, если ORM поддерживает мультифайловую схему

Глобальное состояние веб-приложения: текущий пользователь, тема, язык

Если нет файлового роутера, то собираем все страницы тут

Первый уровень доменов

Список всех доменов приложения (название папки = название домена)

Агрегатор всех схем, если ORM поддерживает мультифайловую схему

Глобальное состояние веб-приложения: текущий пользователь, тема, язык

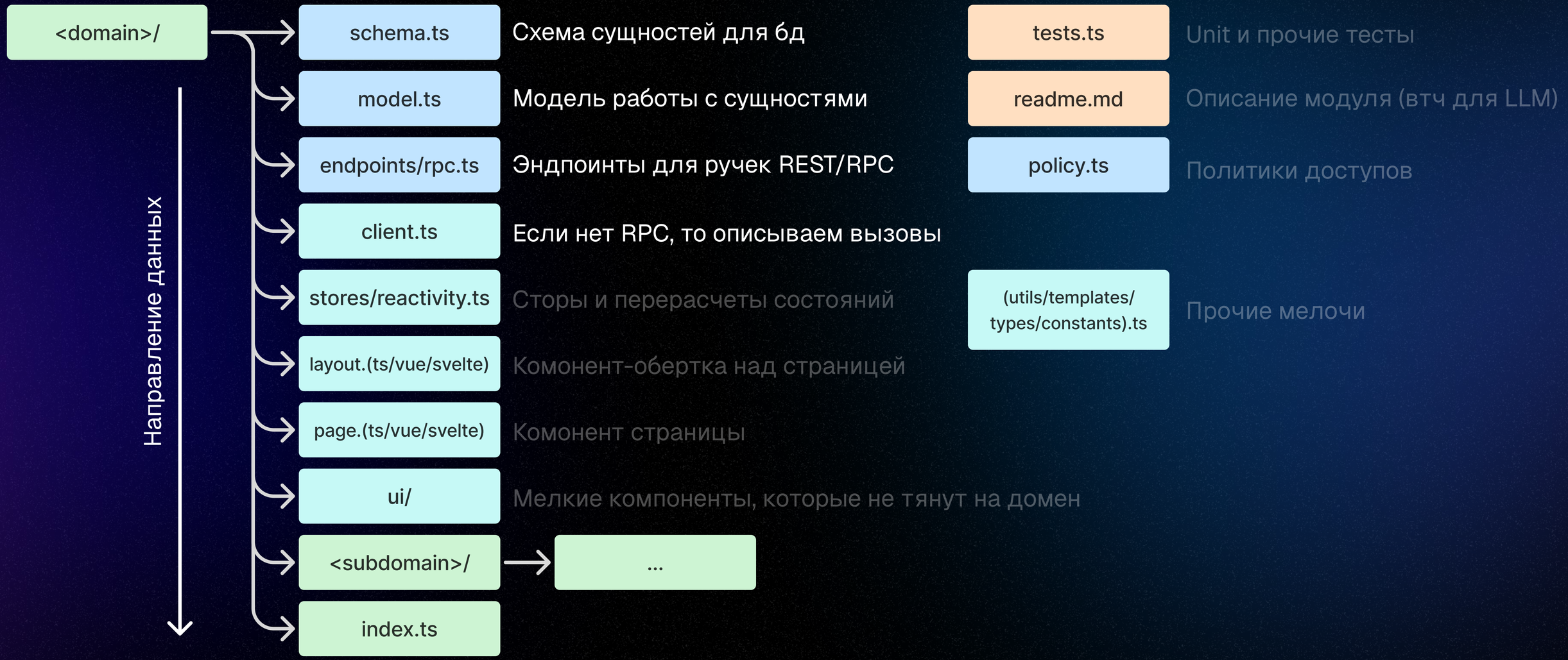
Если нет файлового роутера, то собираем все страницы тут

“move files around until it feels right”

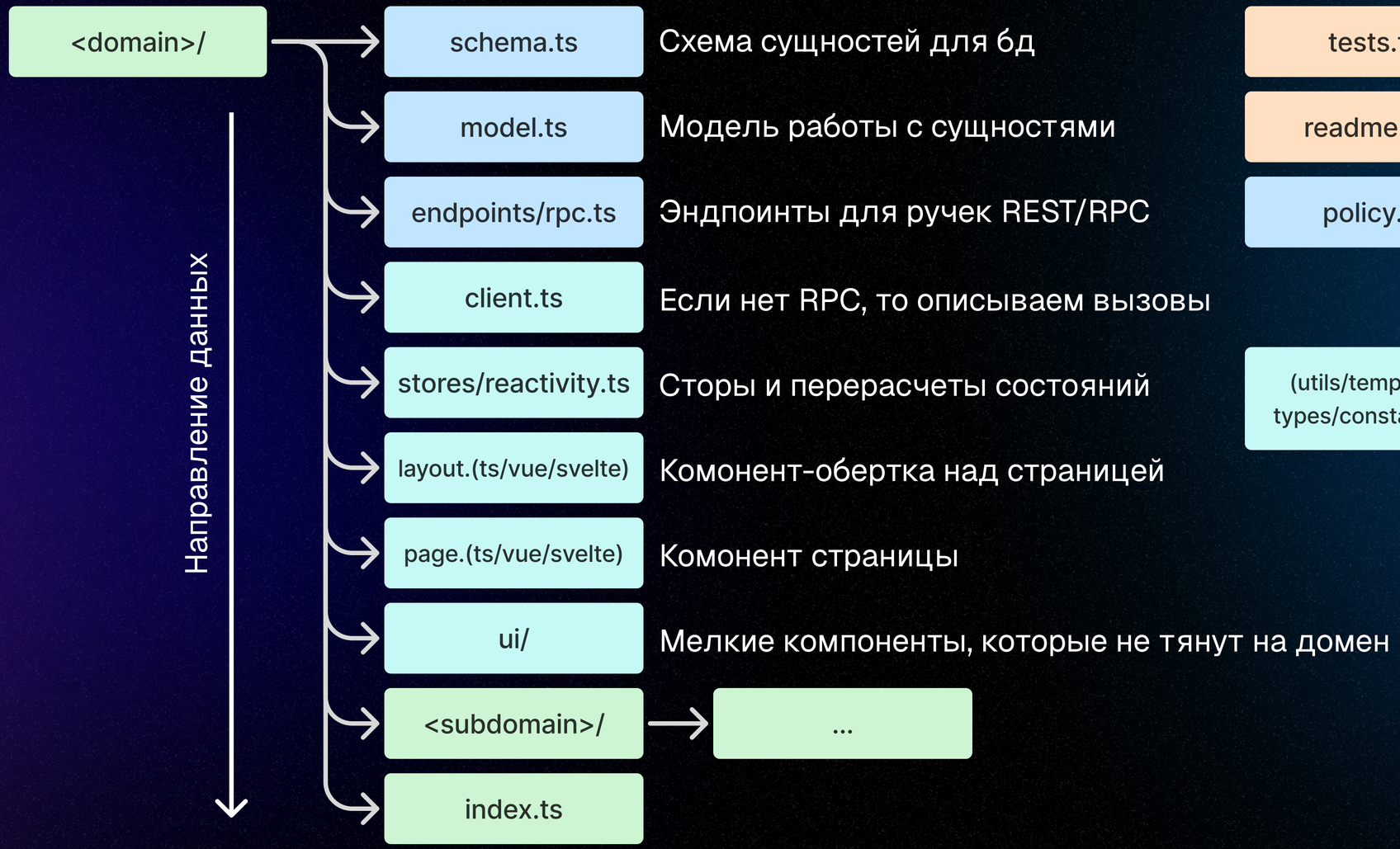
— Дэн Абрамов

https://react-file-structure.surge.sh

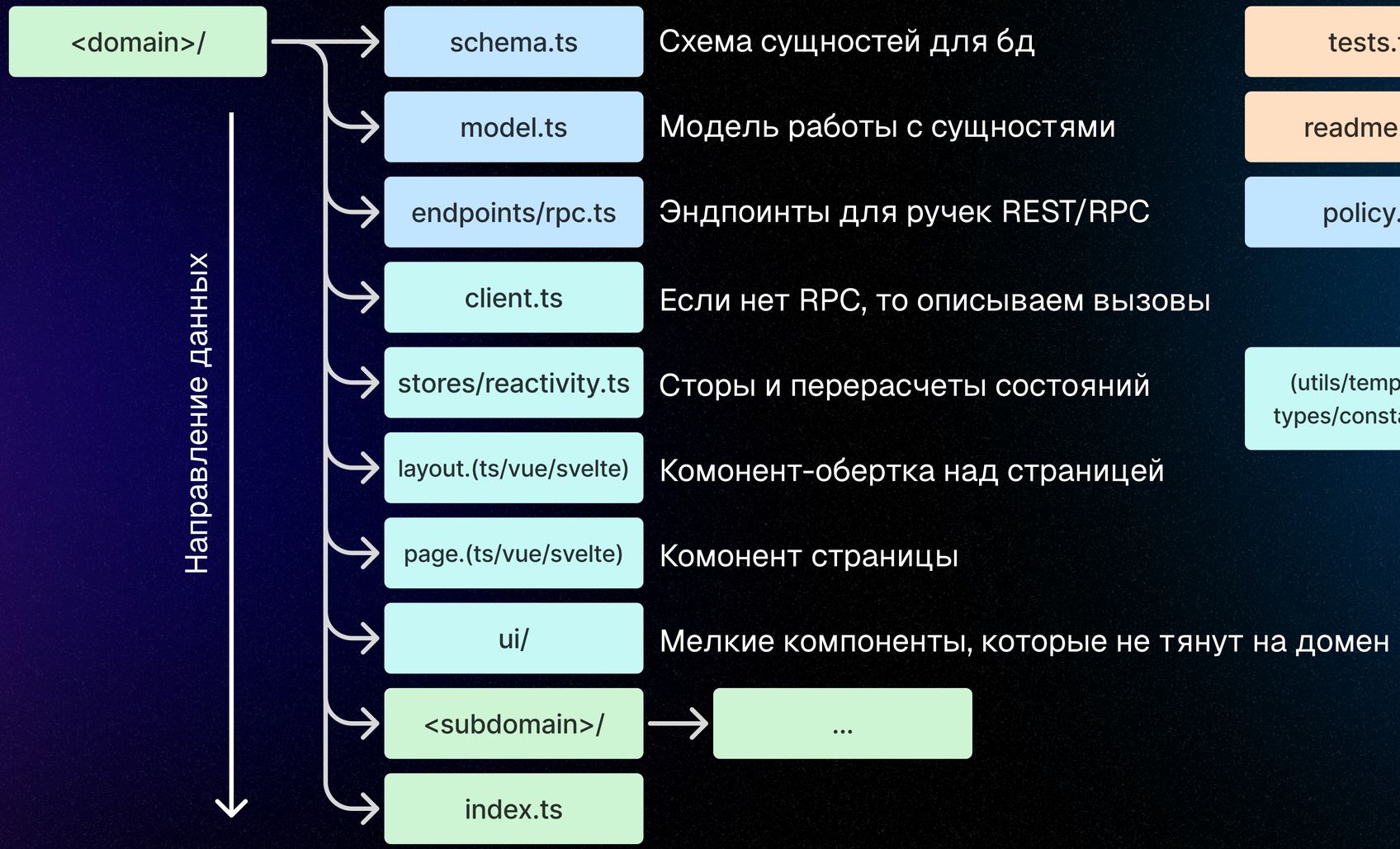
Домен


<domain>/
Направление данных
schema.ts
Схема сущностей для бд
model.ts
Модель работы с сущностями
endpoints/rpc.ts
Эндпоинты для ручек REST/RPC
client.ts
Если нет RPC, то описываем вызовы
stores/reactivity.ts
Сторы и перерасчеты состояний
layout.(ts/vue/svelte)
Комонент-обертка над страницей
page.(ts/vue/svelte)
Комонент страницы
ui/
Мелкие компоненты, которые не тянут на домен
<subdomain>/
...
index.ts
tests.ts
Unit и прочие тесты
readme.md
Описание модуля (втч для LLM)
policy.ts
Политики доступов
(utils/templates/
types/constants).ts
Прочие мелочи

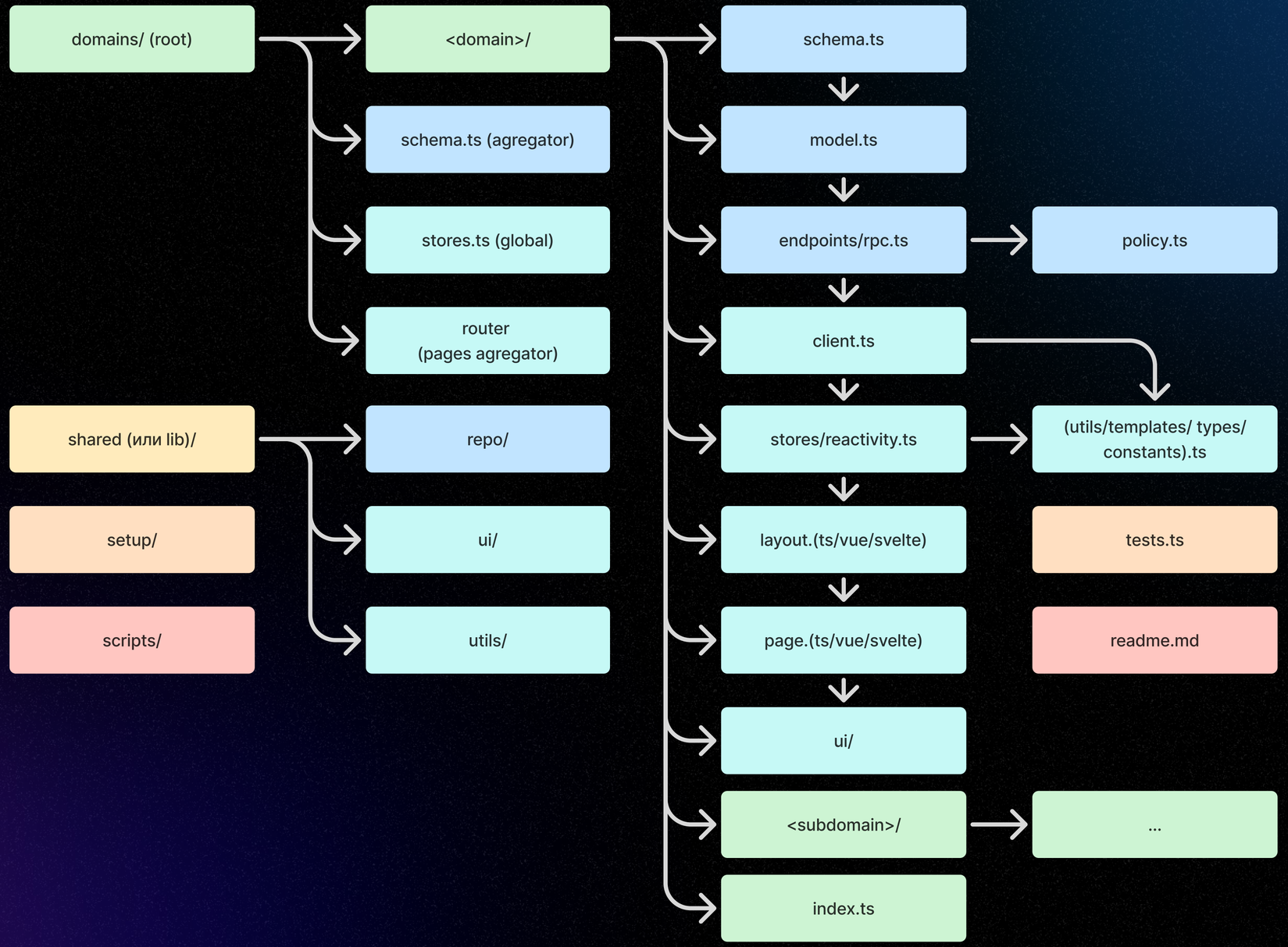
Домен


<domain>/
Направление данных
schema.ts
Схема сущностей для бд
model.ts
Модель работы с сущностями
endpoints/rpc.ts
Эндпоинты для ручек REST/RPC
client.ts
Если нет RPC, то описываем вызовы
stores/reactivity.ts
Сторы и перерасчеты состояний
layout.(ts/vue/svelte)
Комонент-обертка над страницей
page.(ts/vue/svelte)
Комонент страницы
ui/
Мелкие компоненты, которые не тянут на домен
<subdomain>/
...
index.ts
tests.ts
Unit и прочие тесты
readme.md
Описание модуля (втч для LLM)
policy.ts
Политики доступов
(utils/templates/
types/constants).ts
Прочие мелочи

Домен


<domain>/
Направление данных
schema.ts
Схема сущностей для бд
model.ts
Модель работы с сущностями
endpoints/rpc.ts
Эндпоинты для ручек REST/RPC
client.ts
Если нет RPC, то описываем вызовы
stores/reactivity.ts
Сторы и перерасчеты состояний
layout.(ts/vue/svelte)
Комонент-обертка над страницей
page.(ts/vue/svelte)
Комонент страницы
ui/
Мелкие компоненты, которые не тянут на домен
<subdomain>/
...
index.ts
tests.ts
Unit и прочие тесты
readme.md
Описание модуля (втч для LLM)
policy.ts
Политики доступов
(utils/templates/
types/constants).ts
Прочие мелочи

Как это выглядит



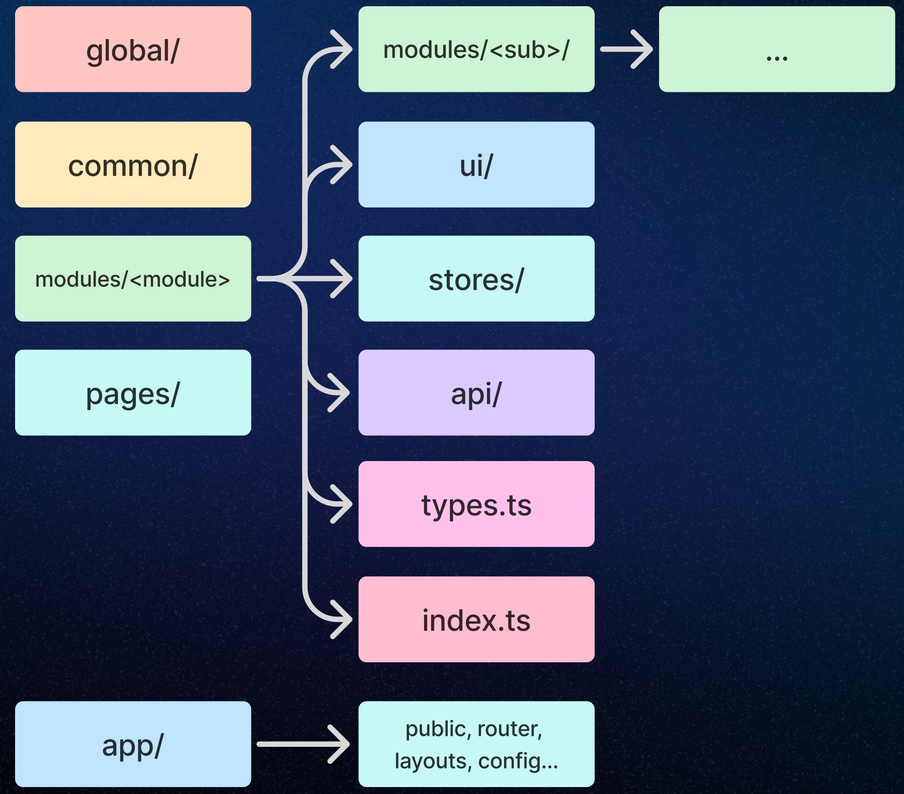
FEOD

Разделение бизнес логики

Разделение функциональных слоев

Основная идея - модульность

Очень похоже на FDA



Семантика вложености

1

Управление приложением П(4)

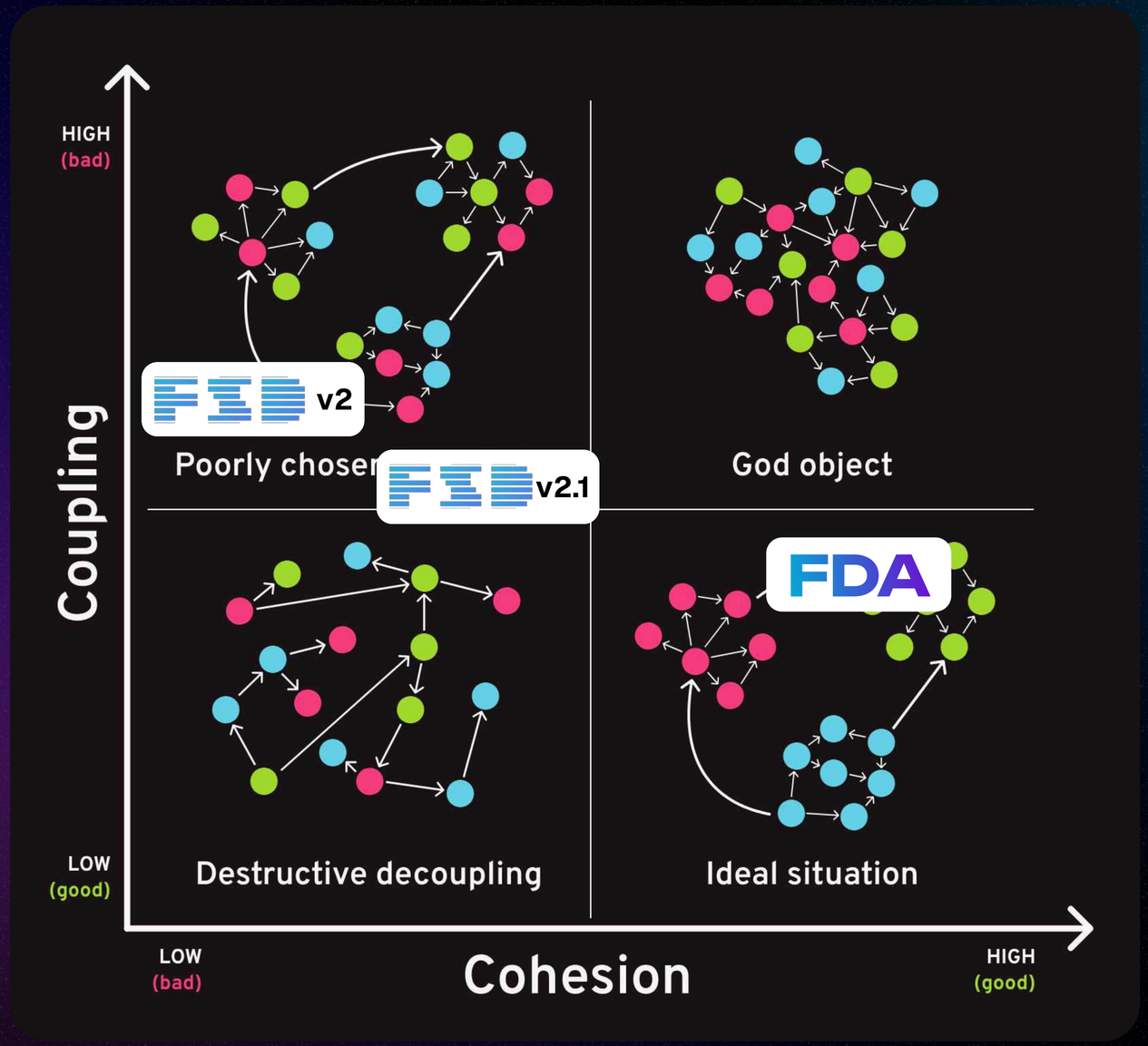
2

Сущность + Абстракиця П(√n)

3

Функиональность П(~10)

Позиционируем FDA



Простая ментальная модель, высокое сцепление бизнес логики.

Публичные интерфейсы дают слабую связность между разными модулями

Однако при неправильном делении есть риск улететь в God Object

Ваше мнение может не совпадать, это нормально

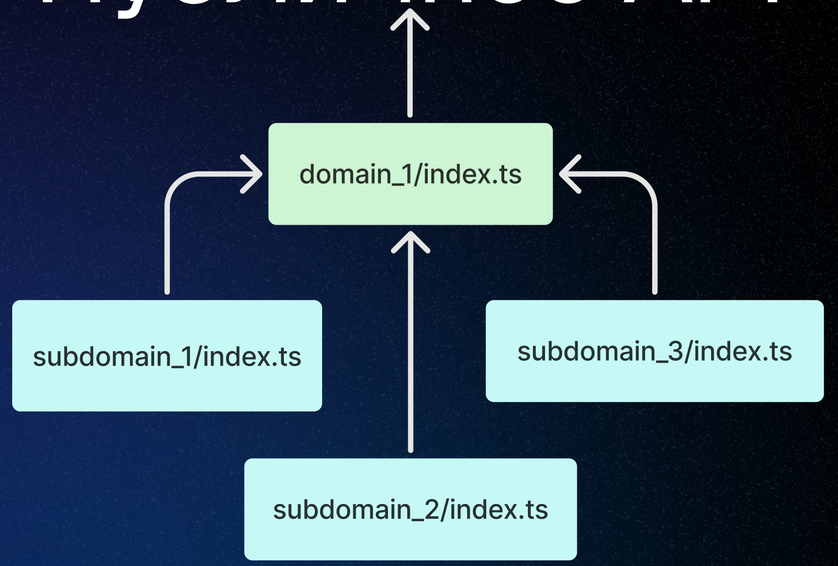
Связи между доменами

```ts
// index.ts

// Импортируем из нижестоящих доменов
import { userStore } from './user/stores.ts'
import { USER_ROLES } from './user/constants.ts'
import { default as Profile } from './user/ui/Profile.svelte'

// Делаем публичным для других
export { userStore, USER_ROLES, Profile }
```

Публичное API



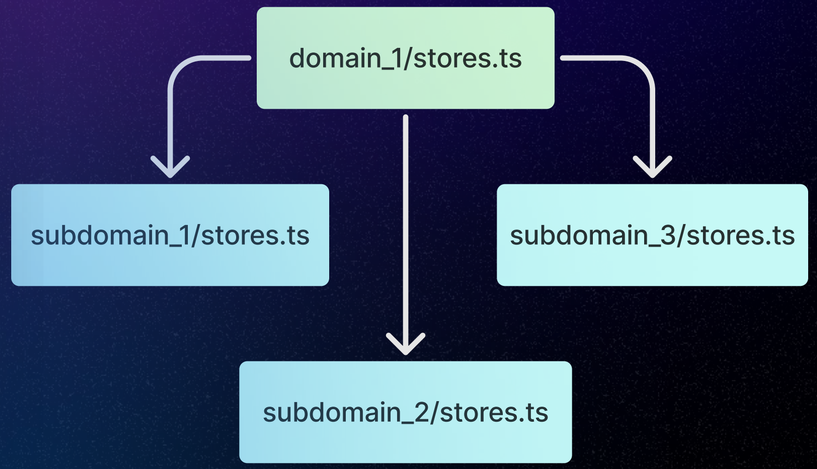
Связи между доменами

```
// subdomain/stores.ts

// Импортируем из родительского домена
import { userStore } from '../stores.ts'

// Используем как контекст для данного субдомена
const profile = derived(userStore, ($userStore) => ...)
```

Контекст



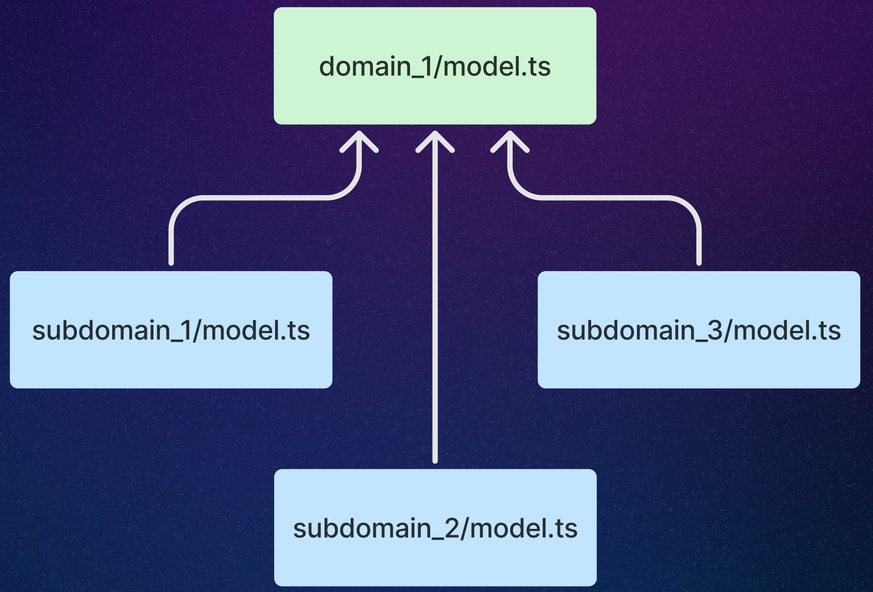
Связи между доменами

```
// order/model.ts

import { cleanUp } from './cart/model.ts'
import { generateBill } from './payment/model.ts'
import { updateSchedule } from './delivery/model.ts'

async function createOrder() {...}
```

Композиция



Связи между доменами

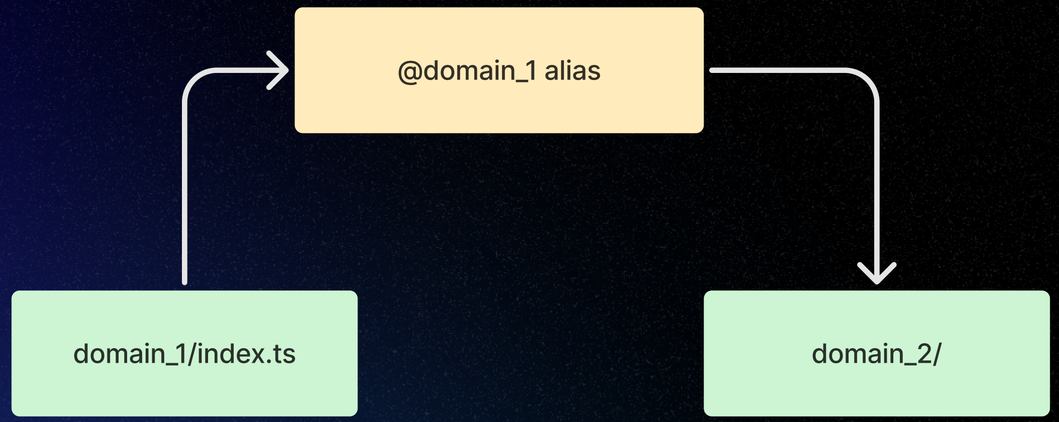
```
// domain_2/stores.ts

// Импортируем из соседнего домена
import { userStore } from '@domain_1'
```

```
// vite.config.js

const config = {
  ...
  alias: {
    "@domain_1": "src/domains/domain_1"
  }
}
```

Кросс связи



Преимущества

Рамки контекста

Разработчик ограничен контекстом в рамках задачи. Исправления вносятся точечно в рамках домена

Conflict-free

Каждый разработчик ответственен за свой домен и за счет этого его исправления не конфликтуют с другими

Функциональная изоляция

Логика распределяется внутри домена по слоям и имеет линейный поток данных и четкую классификацию

Быстрый онбординг

Достаточно отдать нужный домен на том или ином уровне в зависимости от грейда. Также позволяет познакомиться с бизнес-процессами в приложении

Масштабирование вглубь

Домен может неограничено расширятся вглубь под новые требования и фичи. При этом не усложняя более верхние уровни

Легкая миграция

Можно внедрять постепенно с мелких доменов. Также позволяет проще перенести миграцию в микросервис/микрофронтенд

Преимущества

Рамки контекста

Разработчик ограничен контекстом в рамках задачи. Исправления вносятся точечно в рамках домена

Conflict-free

Каждый разработчик ответственен за свой домен и за счет этого его исправления не конфликтуют с другими

Функциональная изоляция

Логика распределяется внутри домена по слоям и имеет линейный поток данных и четкую классификацию

Быстрый онбординг

Достаточно отдать нужный домен на том или ином уровне в зависимости от грейда. Также позволяет познакомиться с бизнес-процессами в приложении

Масштабирование вглубь

Домен может неограничено расширятся вглубь под новые требования и фичи. При этом не усложняя более верхние уровни

Легкая миграция

Можно внедрять постепенно с мелких доменов. Также позволяет проще перенести миграцию в микросервис/микрофронтенд

Преимущества

Рамки контекста

Разработчик ограничен контекстом в рамках задачи. Исправления вносятся точечно в рамках домена

Conflict-free

Каждый разработчик ответственен за свой домен и за счет этого его исправления не конфликтуют с другими

Функциональная изоляция

Логика распределяется внутри домена по слоям и имеет линейный поток данных и четкую классификацию

Быстрый онбординг

Достаточно отдать нужный домен на том или ином уровне в зависимости от грейда. Также позволяет познакомиться с бизнес-процессами в приложении

Масштабирование вглубь

Домен может неограничено расширятся вглубь под новые требования и фичи. При этом не усложняя более верхние уровни

Легкая миграция

Можно внедрять постепенно с мелких доменов. Также позволяет проще перенести миграцию в микросервис/микрофронтенд

Преимущества

Рамки контекста

Разработчик ограничен контекстом в рамках задачи. Исправления вносятся точечно в рамках домена

Conflict-free

Каждый разработчик ответственен за свой домен и за счет этого его исправления не конфликтуют с другими

Функциональная изоляция

Логика распределяется внутри домена по слоям и имеет линейный поток данных и четкую классификацию

Быстрый онбординг

Достаточно отдать нужный домен на том или ином уровне в зависимости от грейда. Также позволяет познакомиться с бизнес-процессами в приложении

Масштабирование вглубь

Домен может неограничено расширятся вглубь под новые требования и фичи. При этом не усложняя более верхние уровни

Легкая миграция

Можно внедрять постепенно с мелких доменов. Также позволяет проще перенести миграцию в микросервис/микрофронтенд

Преимущества

Рамки контекста

Разработчик ограничен контекстом в рамках задачи. Исправления вносятся точечно в рамках домена

Conflict-free

Каждый разработчик ответственен за свой домен и за счет этого его исправления не конфликтуют с другими

Функциональная изоляция

Логика распределяется внутри домена по слоям и имеет линейный поток данных и четкую классификацию

Быстрый онбординг

Достаточно отдать нужный домен на том или ином уровне в зависимости от грейда. Также позволяет познакомиться с бизнес-процессами в приложении

Масштабирование вглубь

Домен может неограничено расширятся вглубь под новые требования и фичи. При этом не усложняя более верхние уровни

Легкая миграция

Можно внедрять постепенно с мелких доменов. Также позволяет проще перенести миграцию в микросервис/микрофронтенд

Преимущества

Рамки контекста

Разработчик ограничен контекстом в рамках задачи. Исправления вносятся точечно в рамках домена

Conflict-free

Каждый разработчик ответственен за свой домен и за счет этого его исправления не конфликтуют с другими

Функциональная изоляция

Логика распределяется внутри домена по слоям и имеет линейный поток данных и четкую классификацию

Быстрый онбординг

Достаточно отдать нужный домен на том или ином уровне в зависимости от грейда. Также позволяет познакомиться с бизнес-процессами в приложении

Масштабирование вглубь

Домен может неограничено расширятся вглубь под новые требования и фичи. При этом не усложняя более верхние уровни

Легкая миграция

Можно внедрять постепенно с мелких доменов. Также позволяет проще перенести миграцию в микросервис/микрофронтенд

Слабые стороны

Избыточность

Может быть слишком подробным и большим для малых проектов с простым функционалом, например лендингов

Неявные домены

Создание новых доменов нужно тщательно контролировать, ревьюить и согласовывать

Мета-фреймворк

В большей степени подходит под мета-фреймворки SvelteKit/Next/Nuxt с файловым роутингом. В меньшей степени для SPA

Иная парадигма

Легко пересесть, если иметь представления про DDD, Clean. Тяжелее, если до этого применялся только FSD

Слабые стороны

Избыточность

Может быть слишком подробным и большим для малых проектов с простым функционалом, например лендингов

Неявные домены

Создание новых доменов нужно тщательно контролировать, ревьюить и согласовывать

Мета-фреймворк

В большей степени подходит под мета-фреймворки SvelteKit/Next/Nuxt с файловым роутингом. В меньшей степени для SPA

Иная парадигма

Легко пересесть, если иметь представления про DDD, Clean. Тяжелее, если до этого применялся только FSD

Слабые стороны

Избыточность

Может быть слишком подробным и большим для малых проектов с простым функционалом, например лендингов

Неявные домены

Создание новых доменов нужно тщательно контролировать, ревьюить и согласовывать

Мета-фреймворк

В большей степени подходит под мета-фреймворки SvelteKit/Next/Nuxt с файловым роутингом. В меньшей степени для SPA

Иная парадигма

Легко пересесть, если иметь представления про DDD, Clean. Тяжелее, если до этого применялся только FSD

Слабые стороны

Избыточность

Может быть слишком подробным и большим для малых проектов с простым функционалом, например лендингов

Неявные домены

Создание новых доменов нужно тщательно контролировать, ревьюить и согласовывать

Мета-фреймворк

В большей степени подходит под мета-фреймворки SvelteKit/Next/Nuxt с файловым роутингом. В меньшей степени для SPA

Иная парадигма

Легко пересесть, если иметь представления про DDD, Clean. Тяжелее, если до этого применялся только FSD

А какие проблемы мы закрыли в FSD

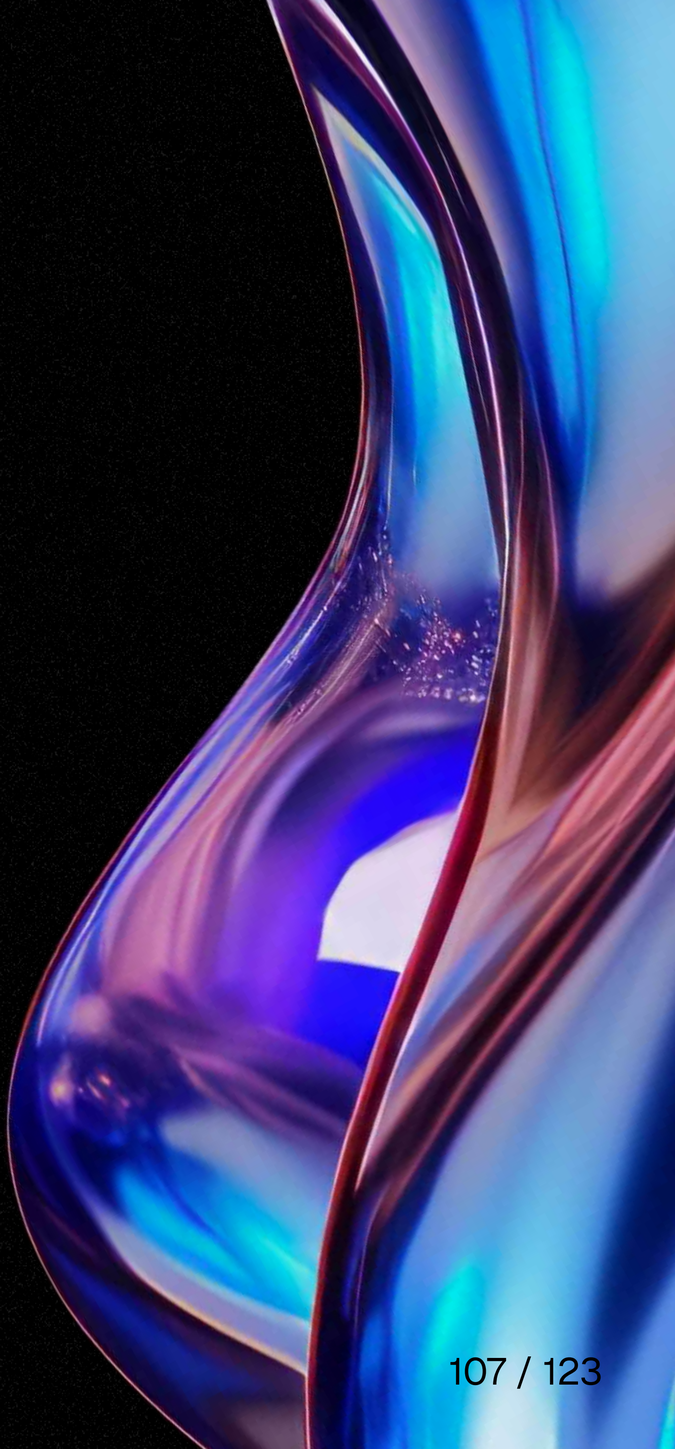
1. Ненужные абстракции —> бизнес-требования толкуются точнее, а деление приносит реальную пользу

2. Запрет кросс импортов —> FDA их разрешает и выставляет требования

А какие проблемы мы закрыли в FSD

3. Слабое сцепление —> домены объединяют логику с ними проще работать

4. Линейный рост —> инкапсулируем сущности благодаря фракталам, снижаем разрастание

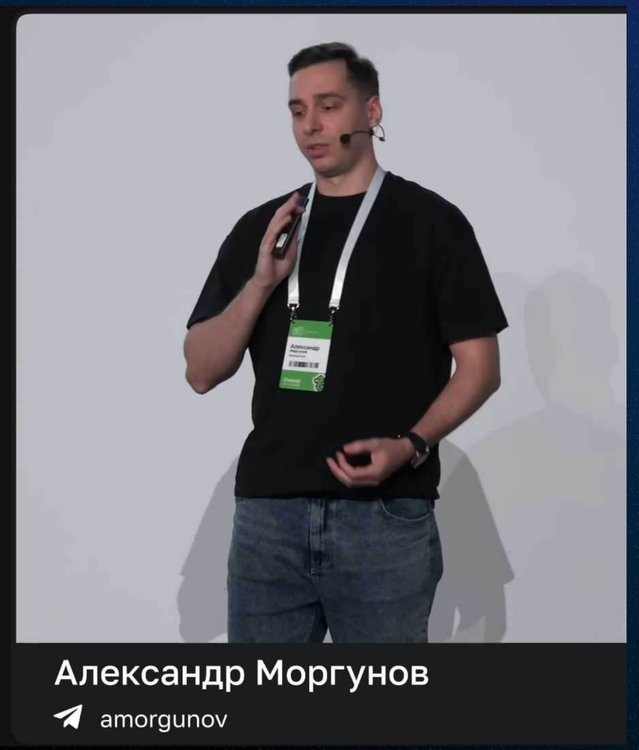


Ничего не напоминает? 😜

Вариант решения проблемы High Coupling & Low Cohesion от пользователя FSD

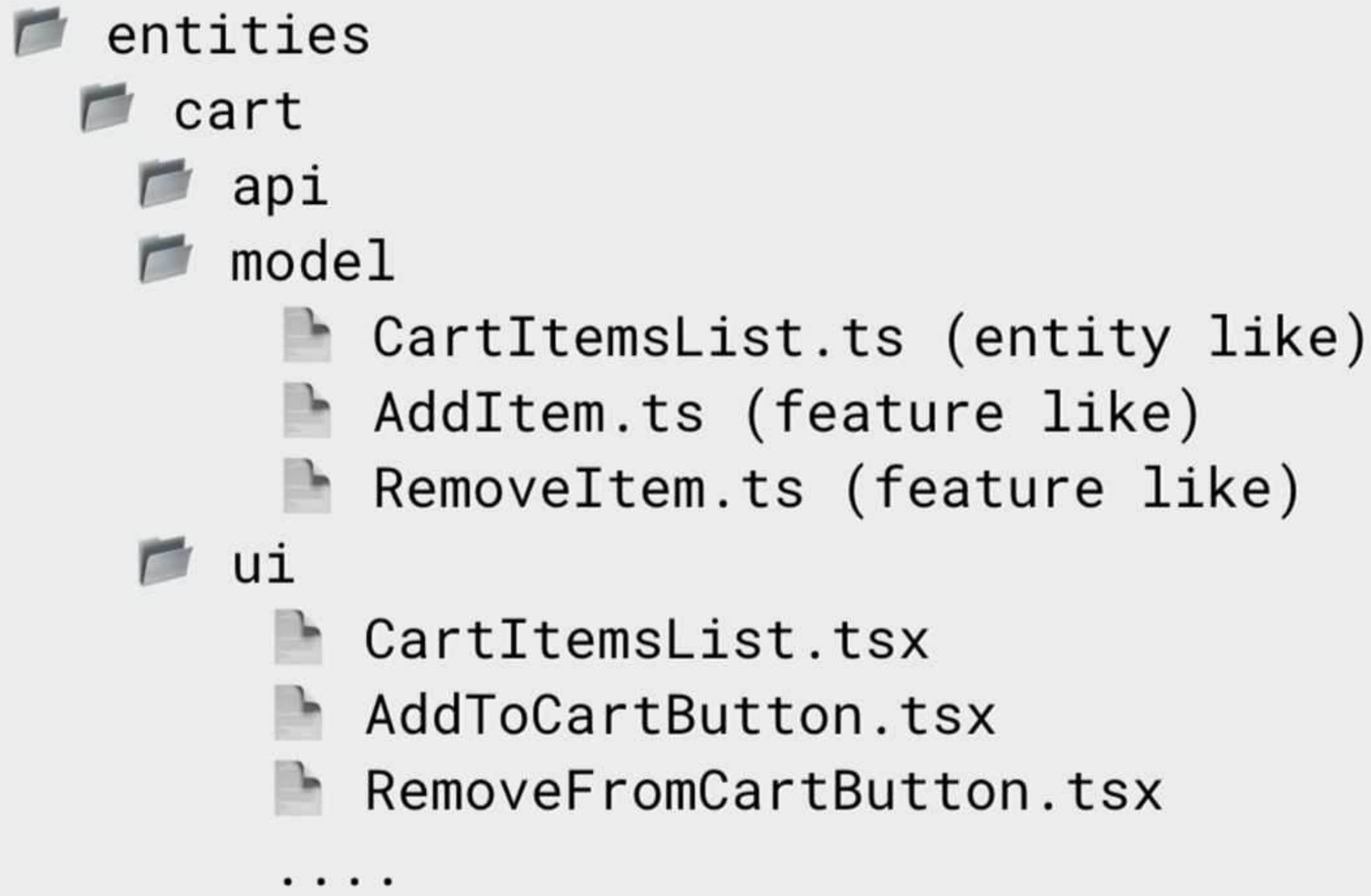
```
entities
  cart
    api
    model
        CartItemsList.ts (entity like)
        AddItem.ts (feature like)
        RemoveItem.ts (feature like)
    ui
        CartItemsList.tsx
        AddToCartButton.tsx
        RemoveFromCartButton.tsx
        ....
```

84



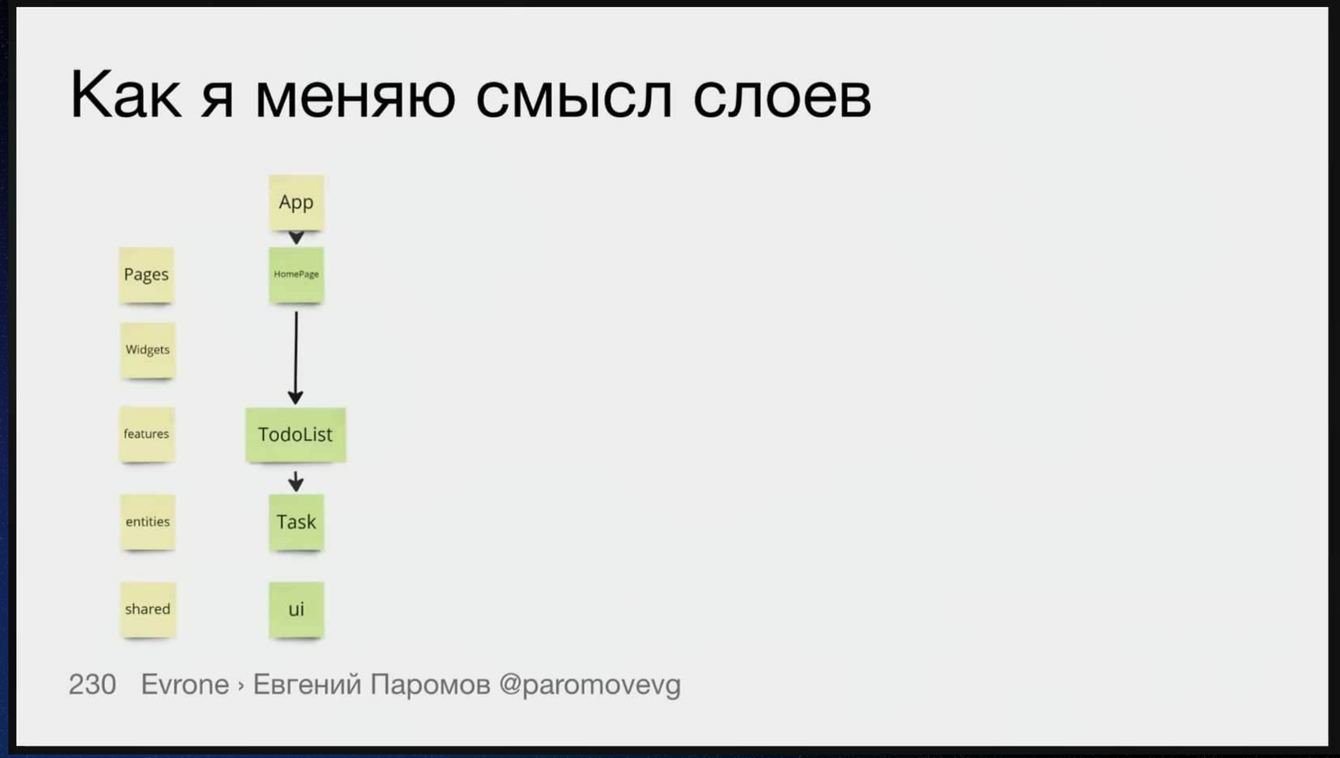
Разбираемся в Feature-Sliced Design

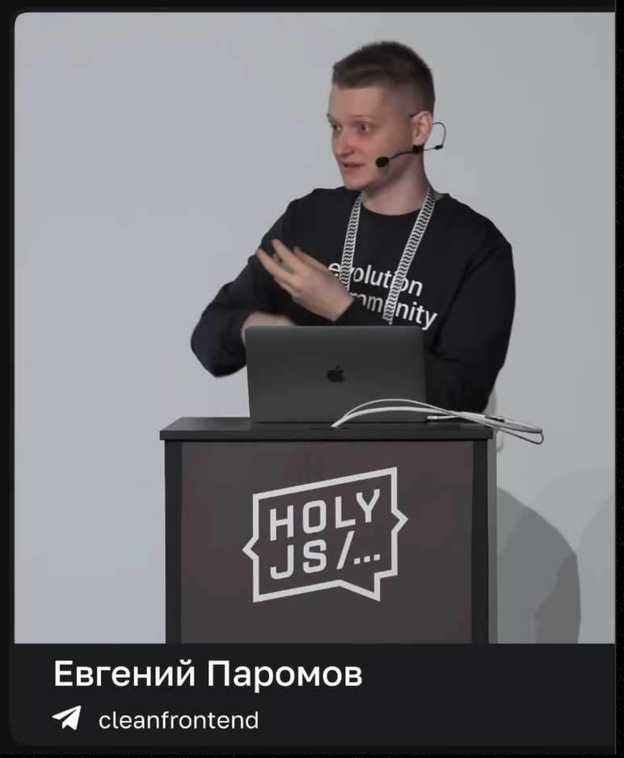
https://www.youtube.com/watch?v=H_rJ0zB8rqc


entities
cart
api
model
CartItemsList.ts (entity like)
AddItem.ts (feature like)
RemoveItem.ts (feature like)
ui
CartItemsList.tsx
AddToCartButton.tsx
RemoveFromCartButton.tsx
....

А вот это? 😜

Вариант решения проблемы High Coupling & Low Cohesion от пользователя FSD





3 главных недостатка FSD после 3 лет использования

https://www.youtube.com/watch?v=H_rJ0zB8rqc

Как я меняю смысл слоев

“Мыши плакали, кололись, но продолжали грызть кактус”

— Народная идиома

Кому подойдет FDA

1

Проект на мета-фреймворке

Архитектура изначально затачивалась под данный тип фреймворков и идеально вписывается в full-stack разработку

2

Огромный легаси клубок

Поэтапно инкапсулируем домены и забываем про них. Постепенно хаос обретет порядок

3

Надежные правила в команде

Если вы хотите зафиксировать гайдлайны,
то нужна архитектура с явными рекомендациями с однозначной трактовкой

Кому подойдет FDA

1

Проект на мета-фреймворке

Архитектура изначально затачивалась под данный тип фреймворков и идеально вписывается в full-stack разработку

2

Огромный легаси клубок

Поэтапно инкапсулируем домены и забываем про них. Постепенно хаос обретет порядок

3

Надежные правила в команде

Если вы хотите зафиксировать гайдлайны,
то нужна архитектура с явными рекомендациями с однозначной трактовкой

Кому подойдет FDA

1

Проект на мета-фреймворке

Архитектура изначально затачивалась под данный тип фреймворков и идеально вписывается в full-stack разработку

2

Огромный легаси клубок

Поэтапно инкапсулируем домены и забываем про них. Постепенно хаос обретет порядок

3

Надежные правила в команде

Если вы хотите зафиксировать гайдлайны,
то нужна архитектура с явными рекомендациями и однозначной трактовкой

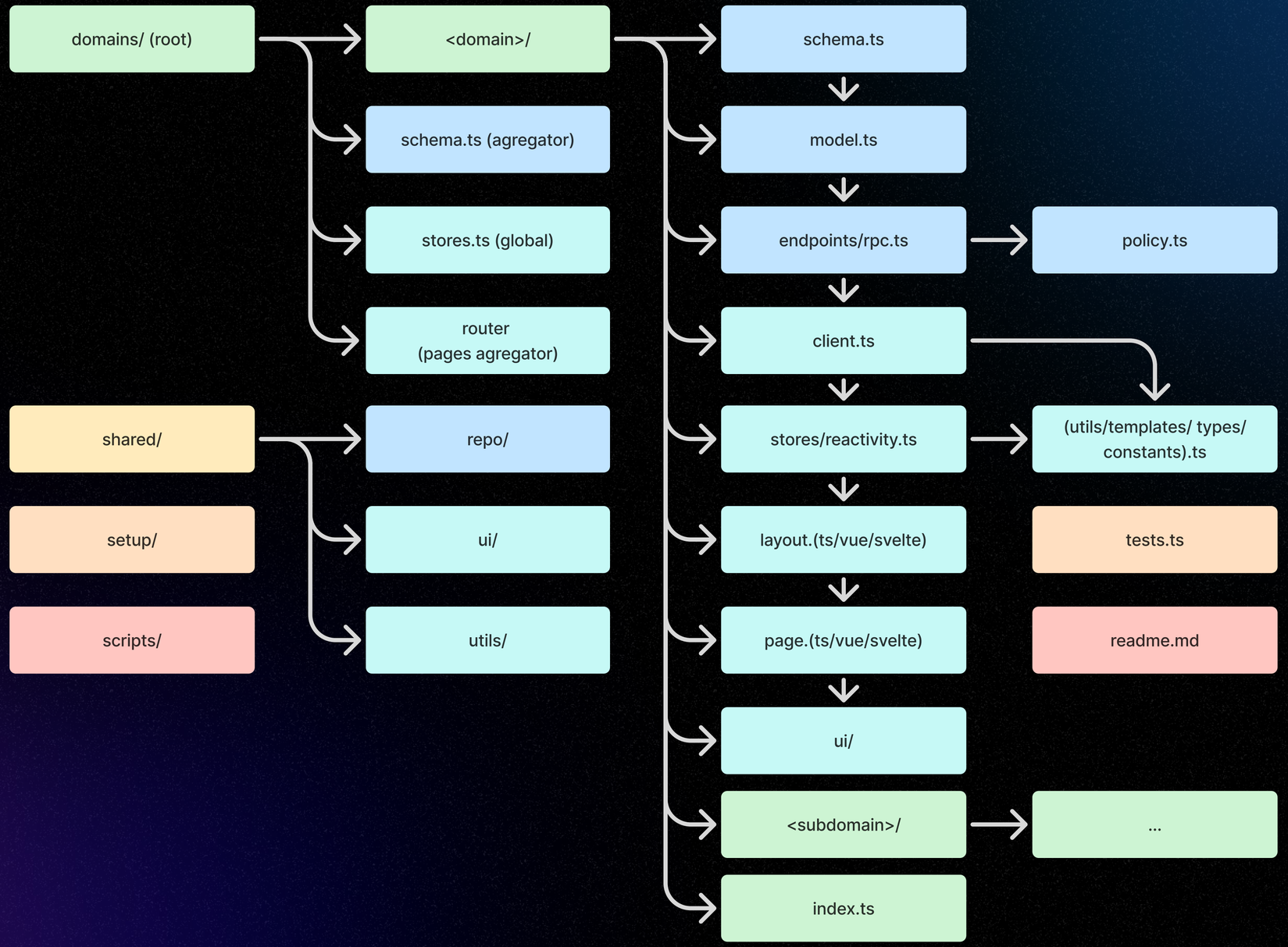
Опыт тимлида

Кодовая база - 200к

Про личный опыт

Положительный DevX для разработчика как драйвер эффективности и мотивации

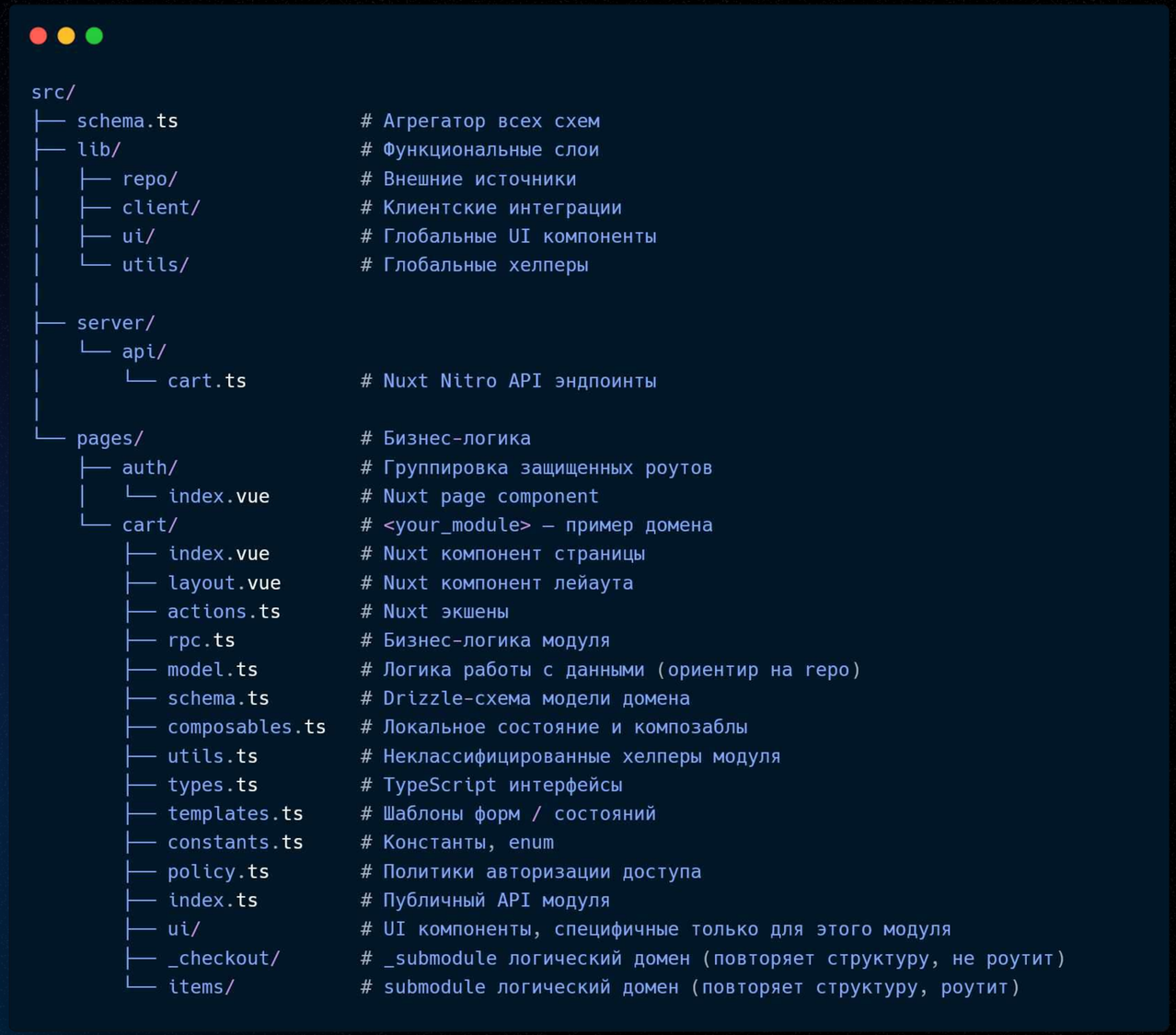
Как это выглядит



SvelteKit

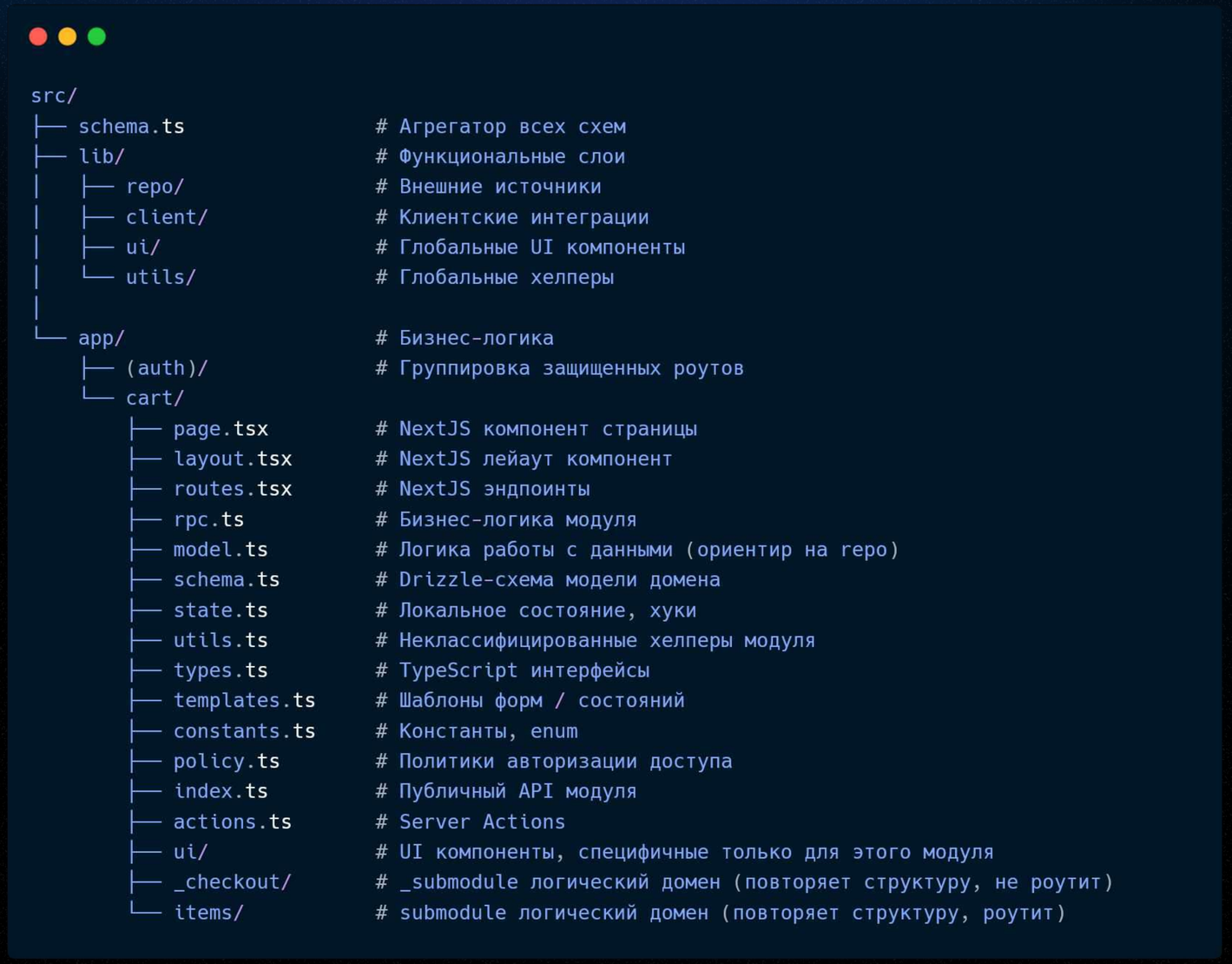
```
src/
├── app.html                  # SvelteKit render template
├── app.css                   # SvelteKit глобальные стили
├── app.d.ts                  # SvelteKit глобальные типы
├── hooks.server.ts           # SvelteKit хуки всех запросов
├── schema.ts                 # Агрегатор схем
│
├── lib/                      # Инфрастуктура без бизнес-логики
│   ├── server/
│   │   └── repo/             # Внешние источники: db, redis, vault, keto
│   ├── client/               # Клиентские интеграции
│   ├── ui/                   # Атомарные переиспользуемые компоненты
│   └── utils/                # Глобальные хелперы для типов и др
│
└── routes/                   # Бизнес-логика
    ├── (auth)/               # Группировка защищённых роутов
    └── cart/
        ├── +page.server.ts   # SvelteKit серверные хуки страницы
        ├── +server.ts        # SvelteKit эндпоинты
        ├── +layout.server.ts # SvelteKit обертка серверных хуков
        ├── +layout.svelte    # SvelteKit лейаут компонент
        ├── +page.svelte      # SvelteKit компонент страницы
        ├── rpc.ts            # Chord RPC классы вместо эндпоинтов
        ├── model.server.ts   # Логика работы с данными (ориентир на repo)
        ├── schema.ts         # Drizzle-схема модели домена
        ├── stores.ts         # Локальное реактивное состояние
        ├── utils.ts          # Неклассифицированные хелперы модуля
        ├── types.ts          # Интерфейсы и типы домена
        ├── templates.ts      # Шаблоны форм / состояний
        ├── constants.ts      # Константы, enum
        ├── policy.ts         # Политики авторизации / доступа
        ├── index.ts          # Публичный API для соседних модулей
        ├── ui/               # UI компоненты, специфичные только для этого модуля
        ├── _checkout/        # _submodule: логический блок (повторяет структуру, не роутит)
        └── items/            # submodule: страничный блок (повторяет структуру, роутит)
```

Nuxt



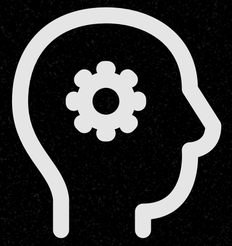
```
src/
├── schema.ts                # Агрегатор всех схем
├── lib/                     # Функциональные слои
│   ├── repo/                # Внешние источники
│   ├── client/              # Клиентские интеграции
│   ├── ui/                  # Глобальные UI компоненты
│   └── utils/               # Глобальные хелперы
│
├── server/
│   └── api/
│       └── cart.ts          # Nuxt Nitro API эндпоинты
│
└── pages/                   # Бизнес-логика
    ├── auth/                # Группировка защищенных роутов
    │   └── index.vue        # Nuxt page component
    └── cart/                # <your_module> — пример домена
        ├── index.vue        # Nuxt компонент страницы
        ├── layout.vue       # Nuxt компонент лейаута
        ├── actions.ts       # Nuxt экшены
        ├── rpc.ts           # Бизнес-логика модуля
        ├── model.ts         # Логика работы с данными (ориентир на repo)
        ├── schema.ts        # Drizzle-схема модели домена
        ├── composables.ts   # Локальное состояние и композаблы
        ├── utils.ts         # Неклассифицированные хелперы модуля
        ├── types.ts         # TypeScript интерфейсы
        ├── templates.ts     # Шаблоны форм / состояний
        ├── constants.ts     # Константы, enum
        ├── policy.ts        # Политики авторизации доступа
        ├── index.ts         # Публичный API модуля
        ├── ui/              # UI компоненты, специфичные только для этого модуля
        ├── _checkout/       # _submodule логический домен (повторяет структуру, не роутит)
        └── items/           # submodule логический домен (повторяет структуру, роутит)
```

Next



```
src/
├── schema.ts                # Агрегатор всех схем
├── lib/                     # Функциональные слои
│   ├── repo/                # Внешние источники
│   ├── client/              # Клиентские интеграции
│   ├── ui/                  # Глобальные UI компоненты
│   └── utils/               # Глобальные хелперы
│
└── app/                     # Бизнес-логика
    ├── (auth)/              # Группировка защищенных роутов
    └── cart/
        ├── page.tsx         # NextJS компонент страницы
        ├── layout.tsx       # NextJS лейаут компонент
        ├── routes.tsx       # NextJS эндпоинты
        ├── rpc.ts           # Бизнес-логика модуля
        ├── model.ts         # Логика работы с данными (ориентир на repo)
        ├── schema.ts        # Drizzle-схема модели домена
        ├── state.ts         # Локальное состояние, хуки
        ├── utils.ts         # Неклассифицированные хелперы модуля
        ├── types.ts         # TypeScript интерфейсы
        ├── templates.ts     # Шаблоны форм / состояний
        ├── constants.ts     # Константы, enum
        ├── policy.ts        # Политики авторизации доступа
        ├── index.ts         # Публичный API модуля
        ├── actions.ts       # Server Actions
        ├── ui/              # UI компоненты, специфичные только для этого модуля
        ├── _checkout/       # _submodule логический домен (повторяет структуру, не роутит)
        └── items/           # submodule логический домен (повторяет структуру, роутит)
```

Итоги



Easy to learn

FDA предлагает простую ментальную модель

Фреймворки

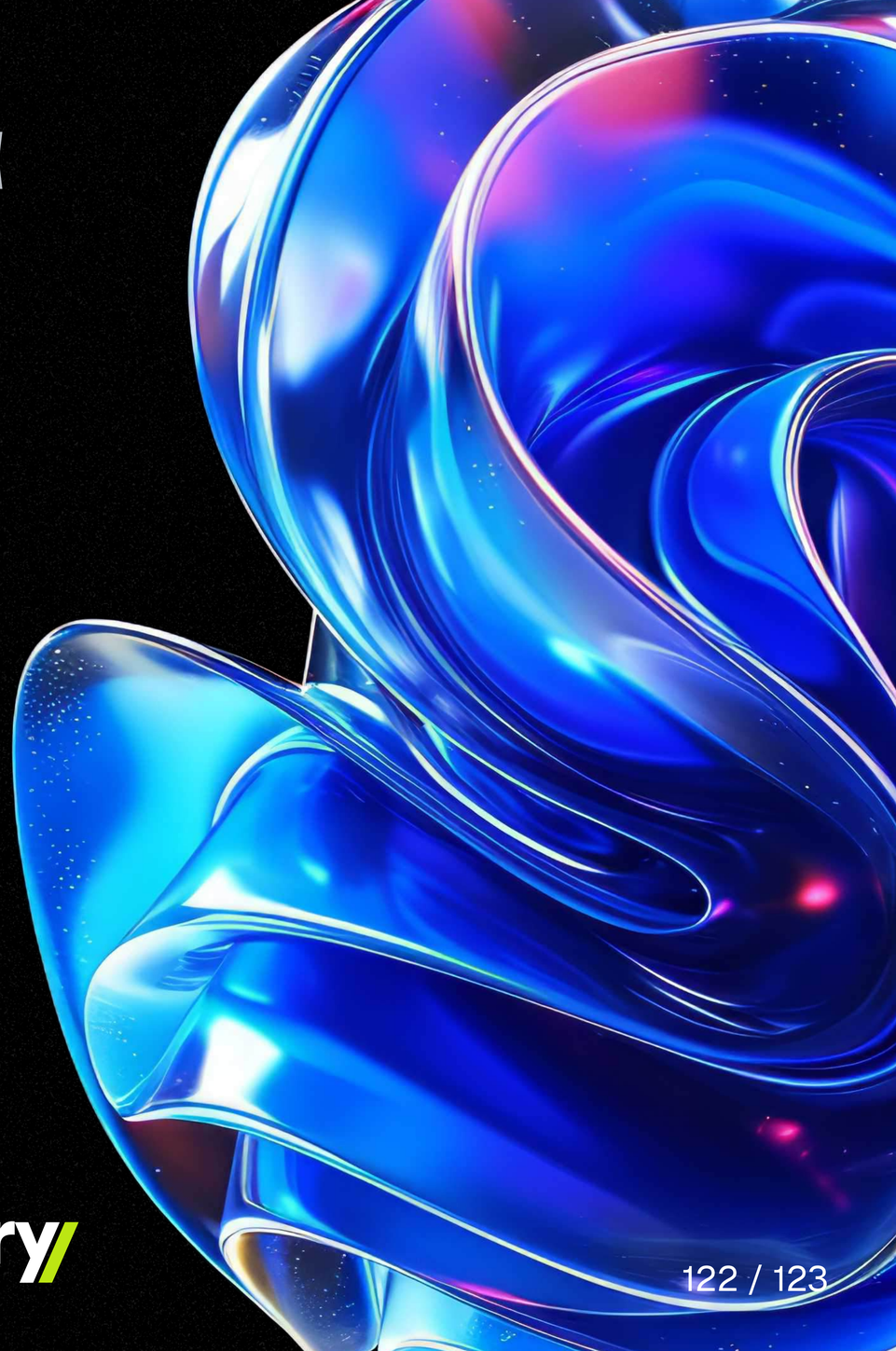
Эффективно встраивается и работает на мета-фреймворках

Масштабирование

Декомпозирует приложение и инкапсулирует сложность

Архитектура хоть куда
Потому что это FDA!

Спасибо за внимание!



15.05

Дмитрий Дин

HOLY JS

Далее × Subquery

QR-CODES

Документация FDA

Репозиторий FDA