Современный бэкенд для фронтенда на node.js

Андрей Мелихов

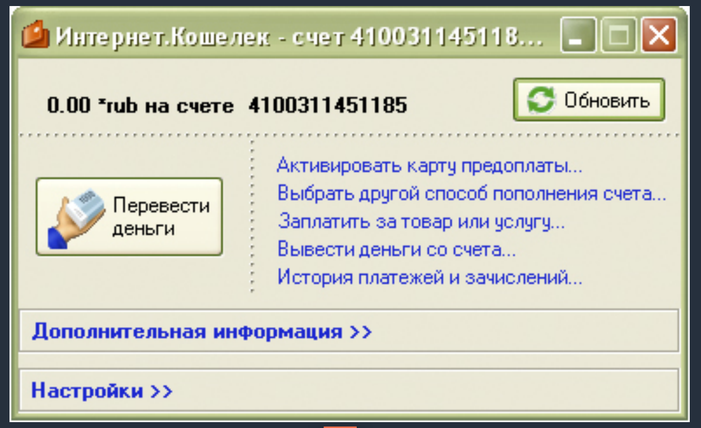
andrey.melikhov@hey.com

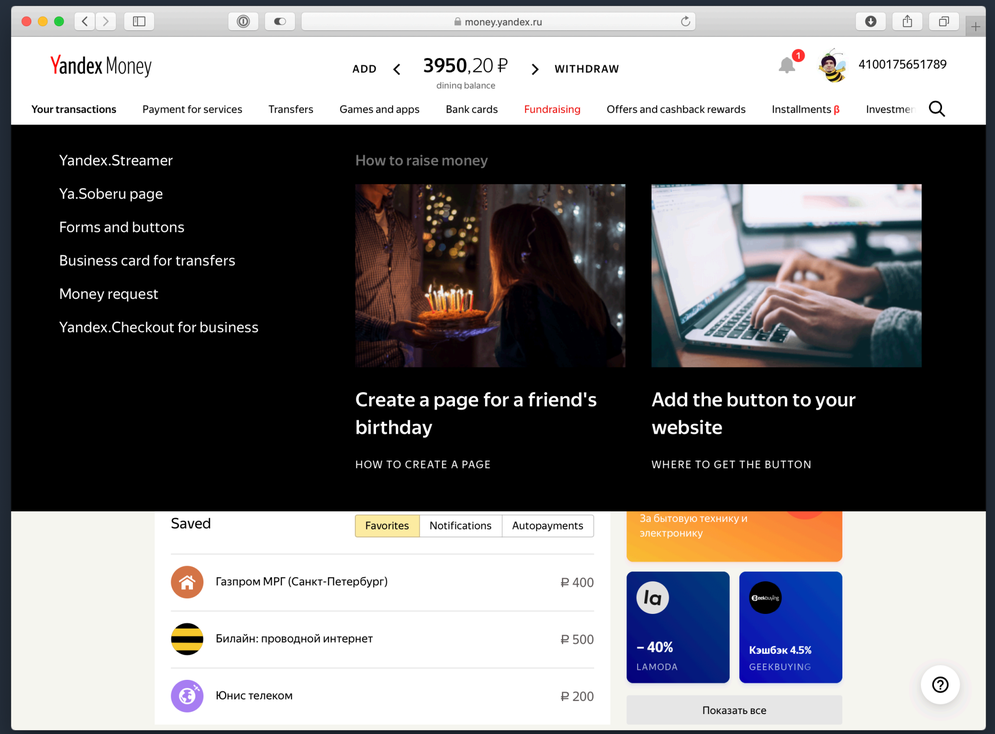
План:

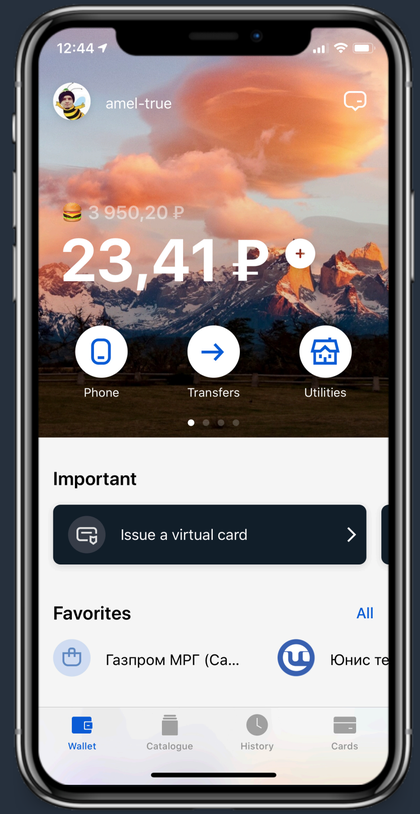
- Что такое фронтенд и зачем ему нужен промежуточный бэкенд
- Почему node.js — это лучшее из зол
- Современный взгляд на архитектуру приложения
- Как построить хорошее приложение на nest и не дать протечь слоям

Что такое front end?

Front end



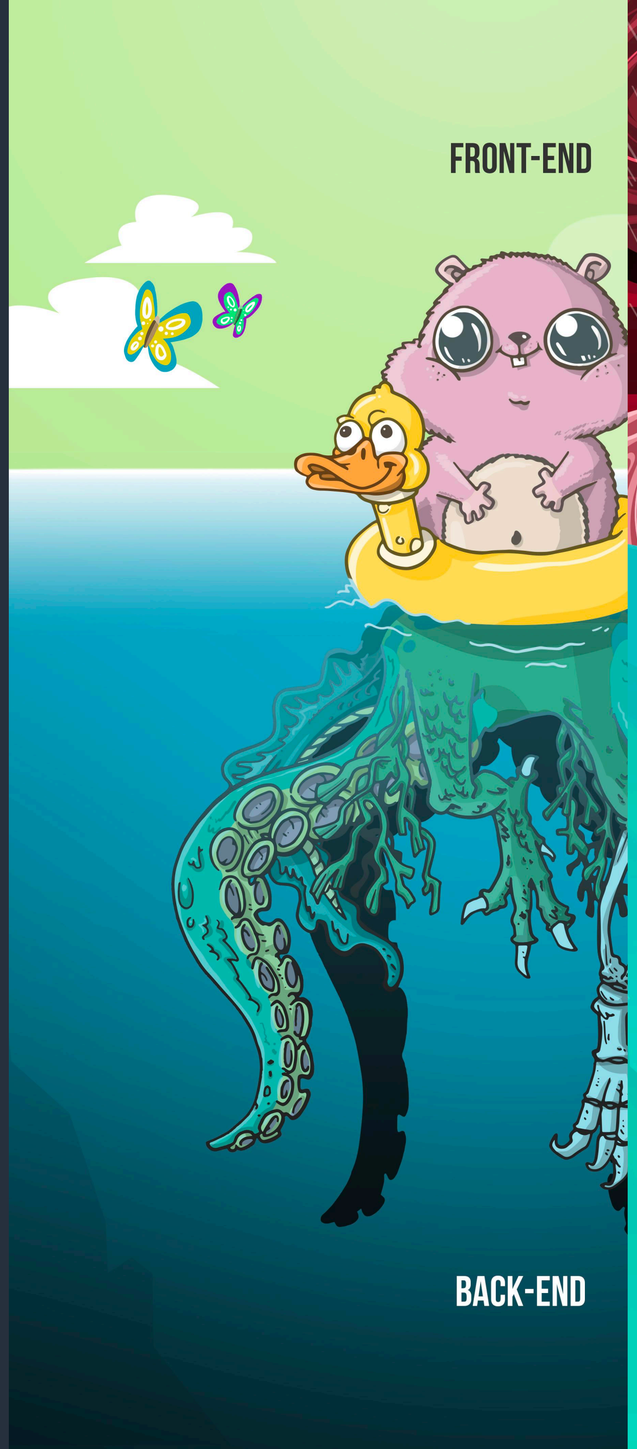


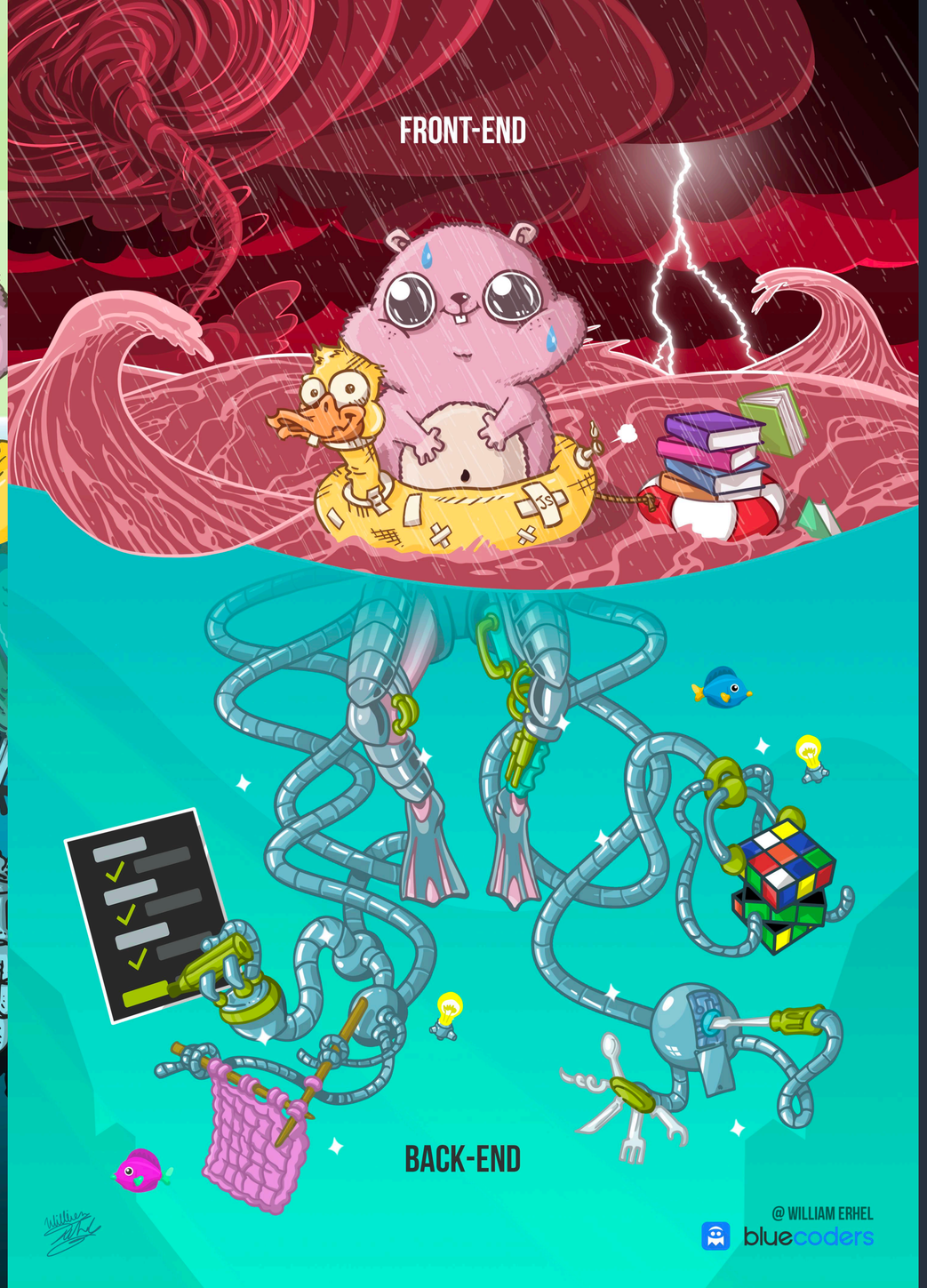


Отображение

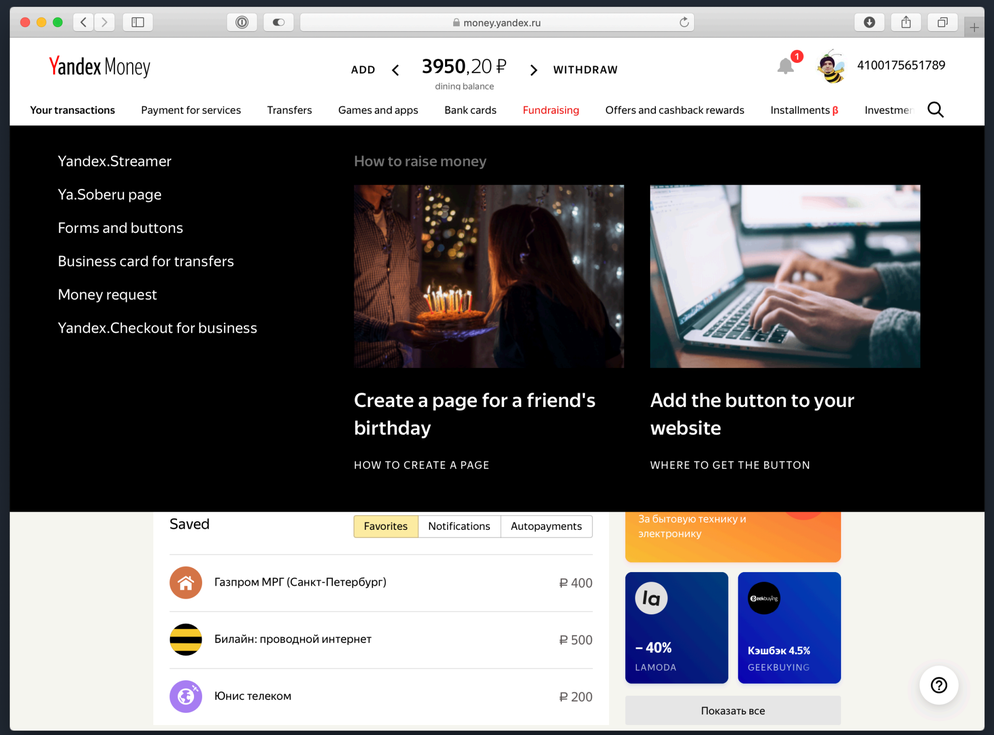
Данные

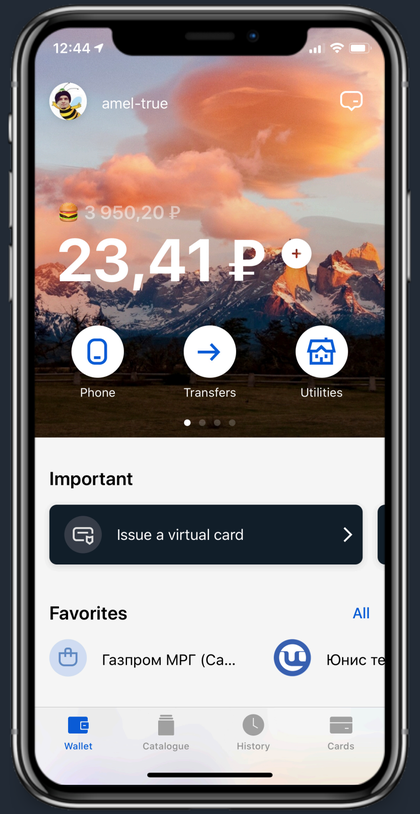
Back end


FRONT-END
BACK-END


FRONT-END
BACK-END
@ WILLIAM ERHEL
bluecoders

Front end





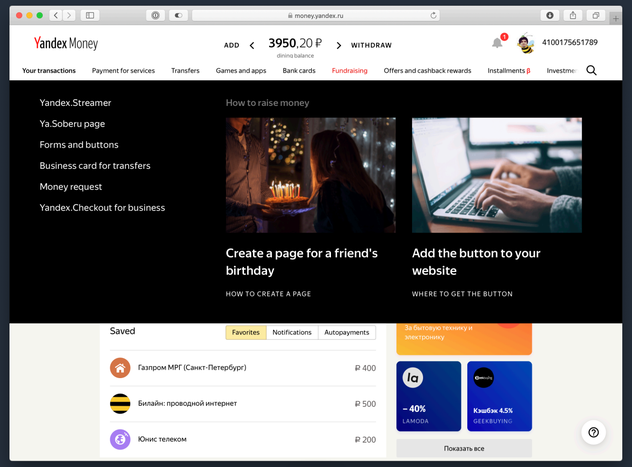
Back end

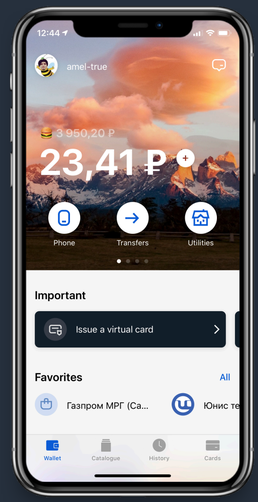
Данные

Задача №1: получение данных

Front end

Зона ответственности фронтенда





Back end

Зона ответственности бэкенда

API (One for rule them all)

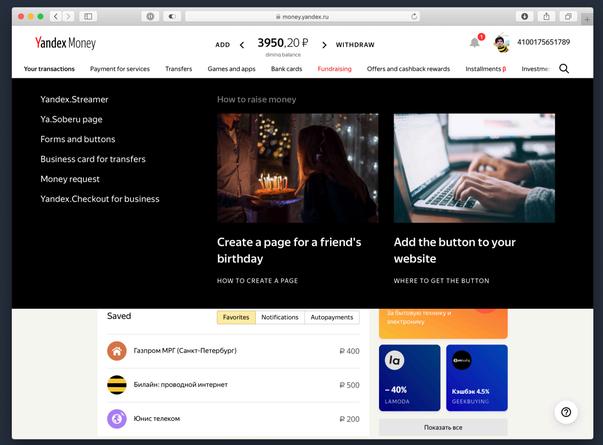
Microservice

Microservice

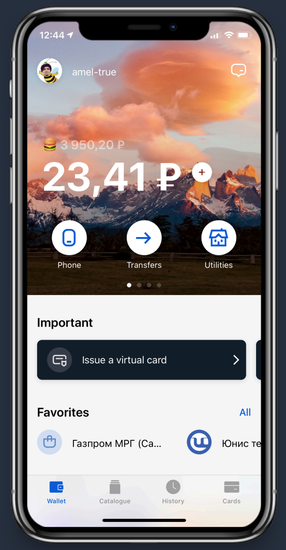
Microservice

Microservice

Проблемы?



!==

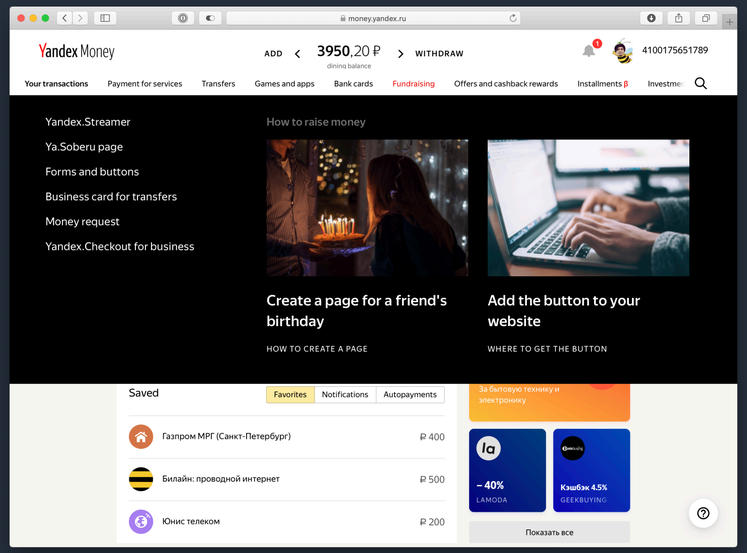


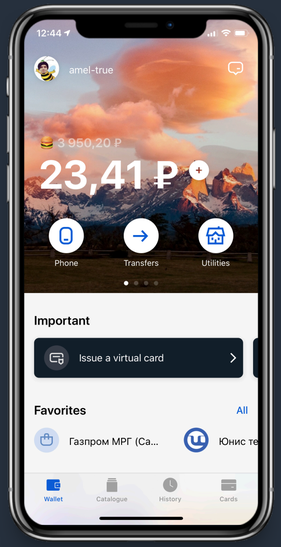
Front end

Back end

API

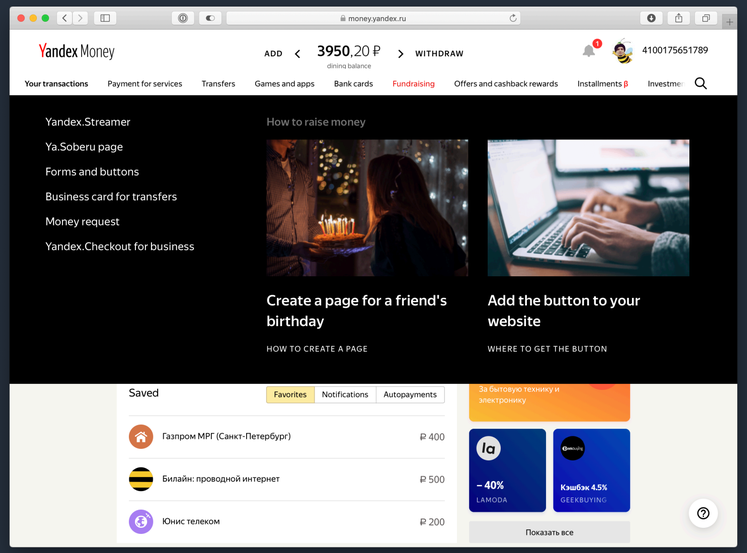
Данные



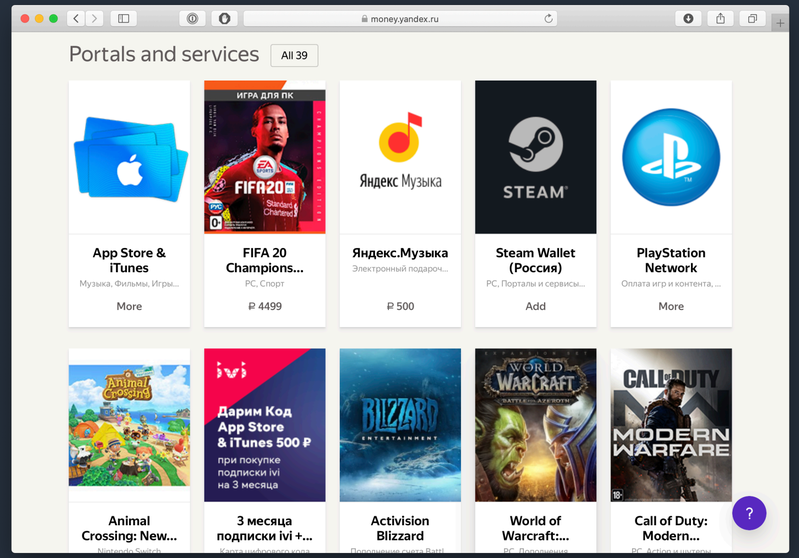


- Ежедневные релизы (Continues Deployment)
- Вечнозелёный клиент

- Редкие релизы
- Зоопарк клиентов — обратная совместимость



!==



- Разный набор данных (множество раздельных запросов)
- Избыточные данные
- Недостаточные данные

Хьюстон, у нас есть другая проблема

Типичная большая IT-компания в прошлом

SysOps

Developers

QA

MONOLITH

Типичная большая IT-компания. Наши дни

Team 1

Доменная область 1

Microservice 1

Team 2

Доменная область 2

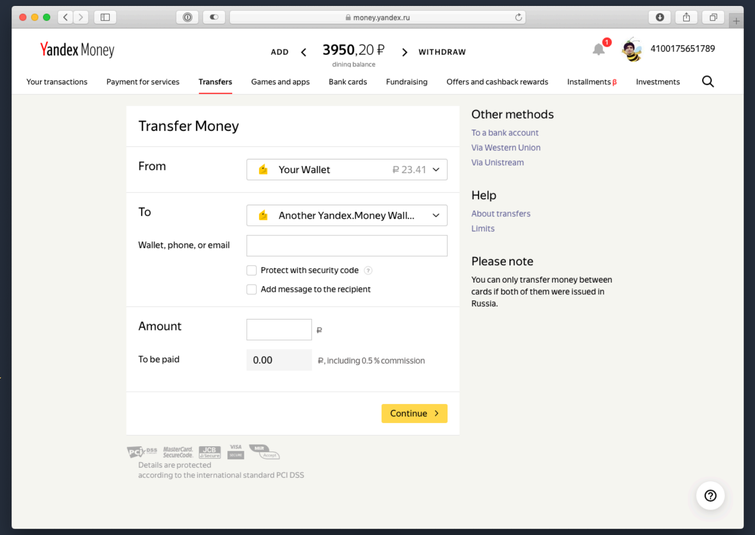
Microservice 2

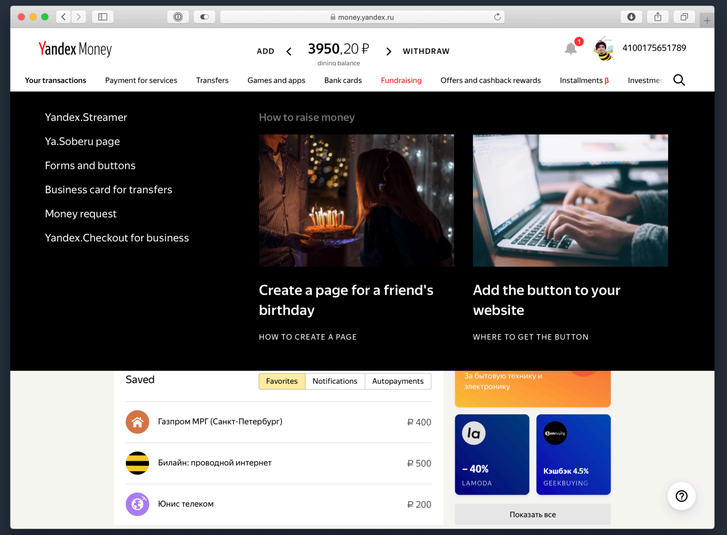
Team 3

Доменная область 3

Microservice 3

Transfers view





Front end

Back end

Generic API

Transfers

Payments

Messages

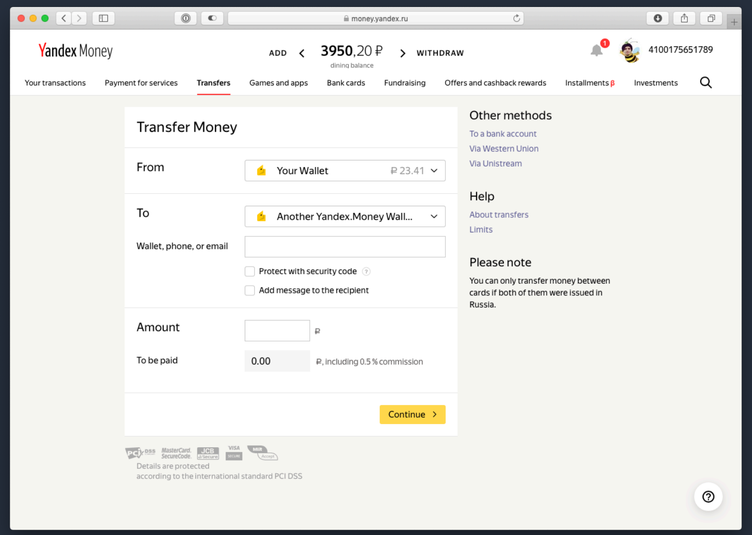
Offers

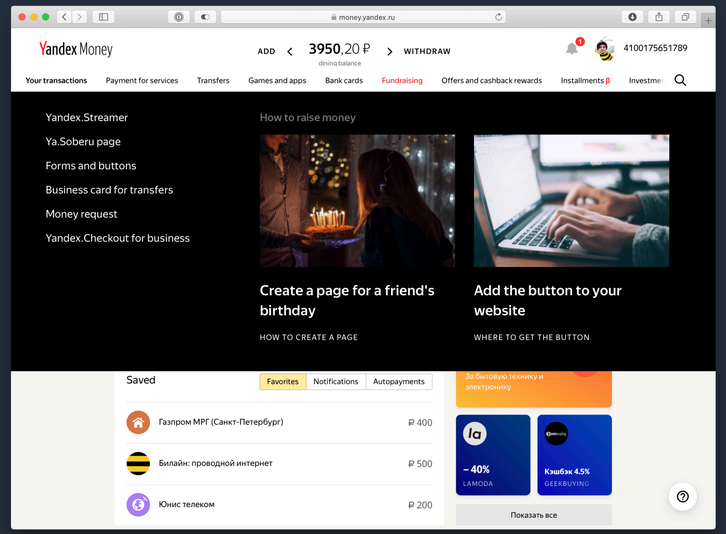
Чья это зона ответственности?


It's the A—Team.


API-Team

Давайте разделим API





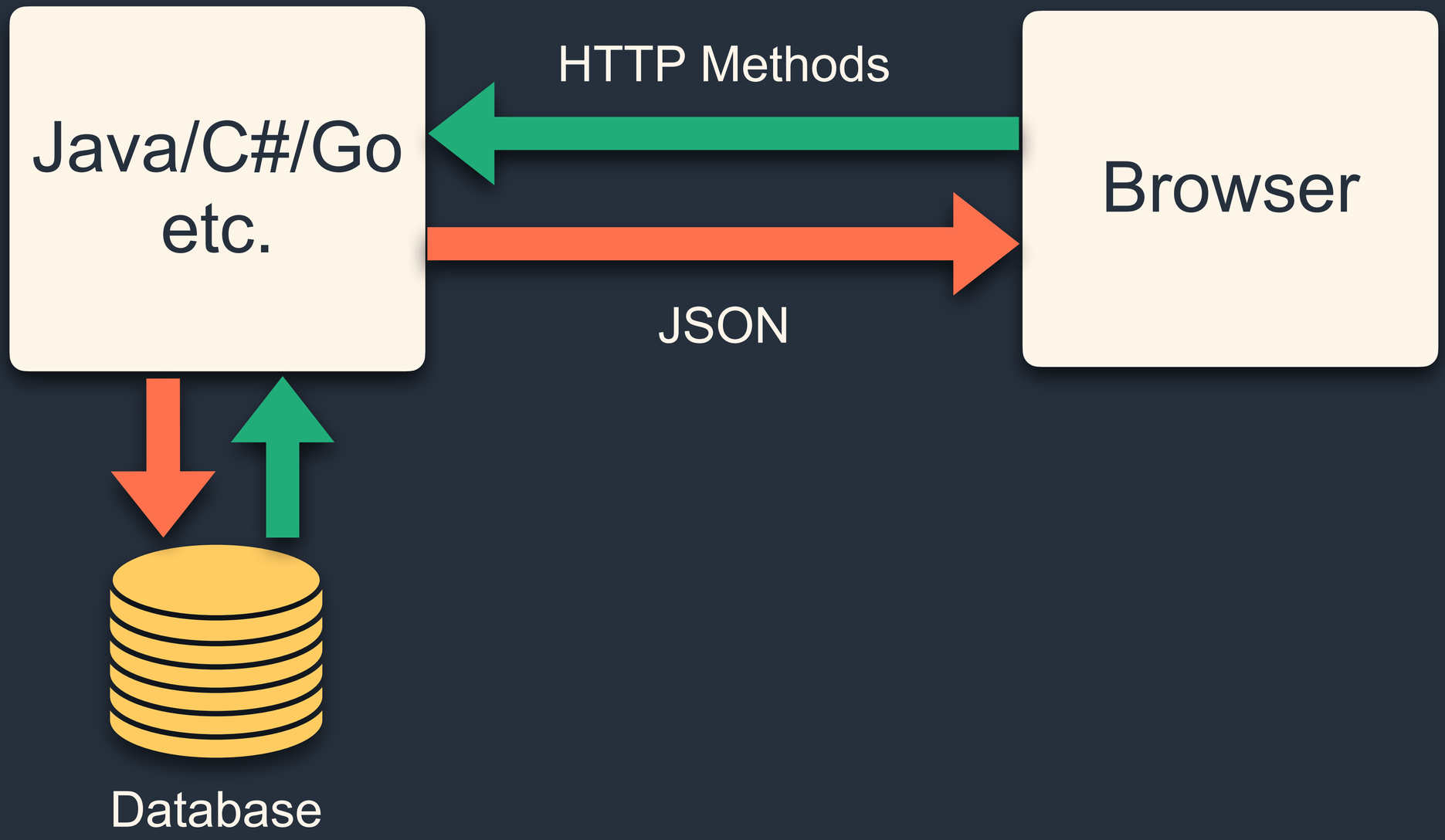
Transfers API

Dashboard API

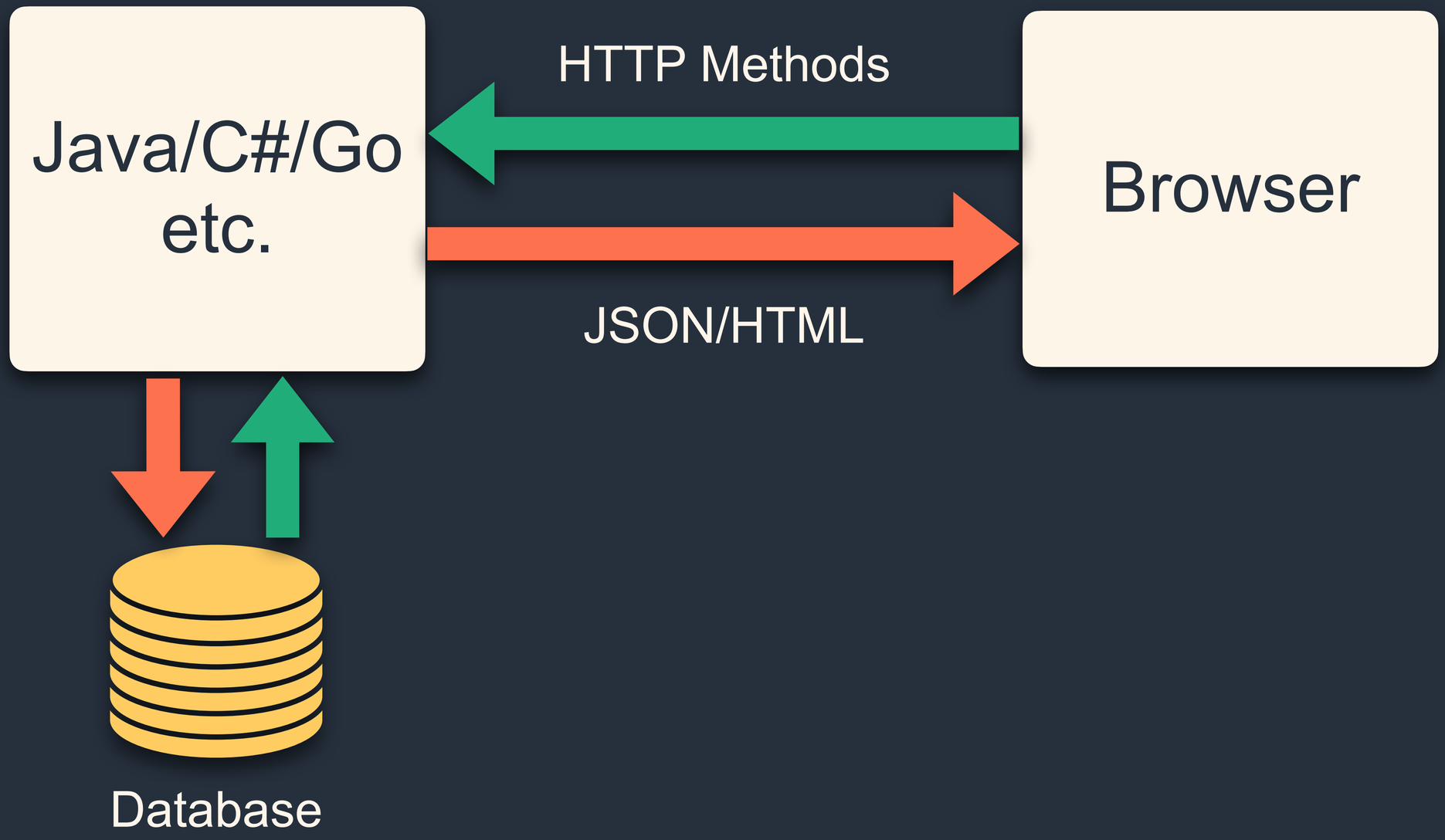
Данные

Задача №2: серверный рендеринг

Идеальный мир с точки зрения бэкенда

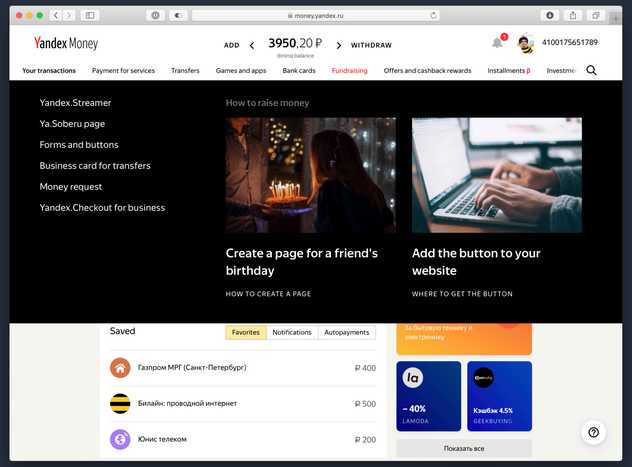


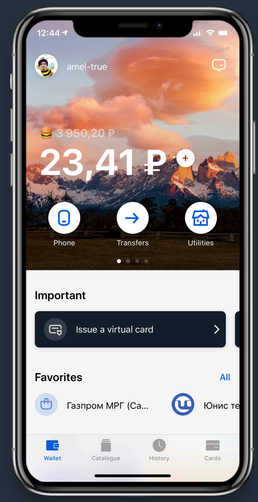
Реальность



Front end

Зона ответственности фронтенда





Server Side Rendering (SSR)

Back end

Зона ответственности бэкенда

API

Microservice

Microservice

Microservice

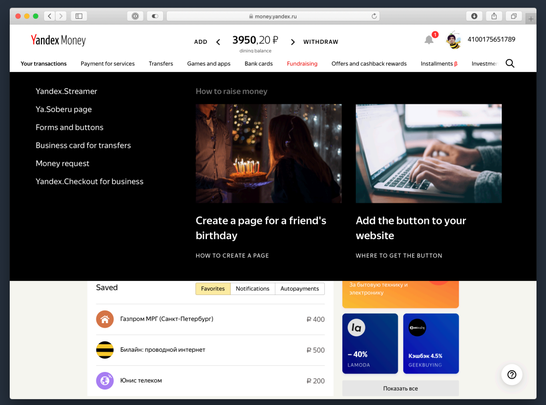
Работало до появления SPA

Что мы хотим в SPA SSR

- Общие шаблоны между клиентом и сервером
- Подготовка Preloaded State (данные, загружаемые со страницей)
- Общие роуты

Давайте заберём SSR у бэкенда

Зона ответственности фронтенда



Python, Java, etc.

JavaScript Runtime


Кто мы?
Фронтендеры!
Чего мы хотим?
Отобрать власть у бэкенда!
Когда мы это хотим?
Сейчас!
troll-face.ru

BFF

Backend For Frontend

Реальность

REST

REST, GraphQL, etc.

Java/C#/Go
etc.

BFF

Browser

JSON

JSON/HTML

Database

Требования

- Контроль над набором данных
- Общие шаблоны на клиенте и сервере
- Скорость ответа под нагрузкой
- Единый язык на клиенте и сервере

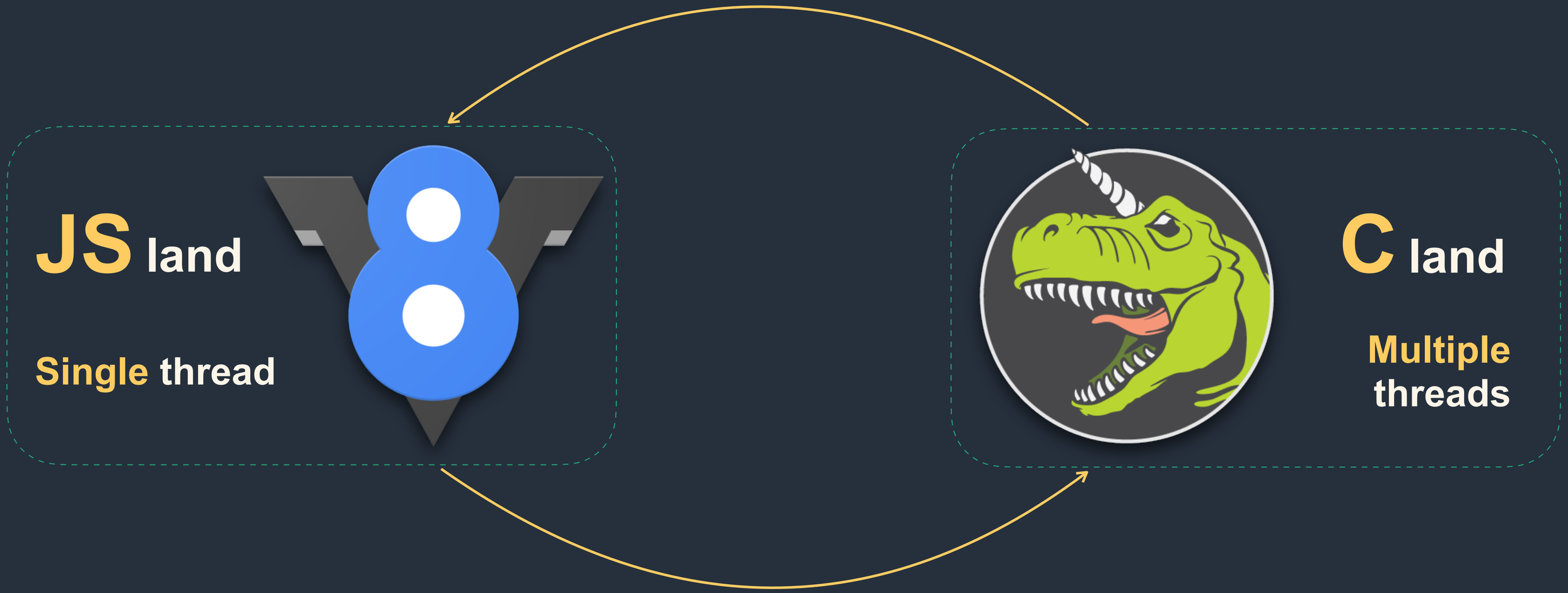
Требования

- Контроль над набором данных
- Общие шаблоны на клиенте и сервере
- Скорость ответа под нагрузкой
- Единый язык на клиенте и сервере

- API Gateway
- JSX, Template Literals, etc.
- Async I/O
- JavaScript runtime

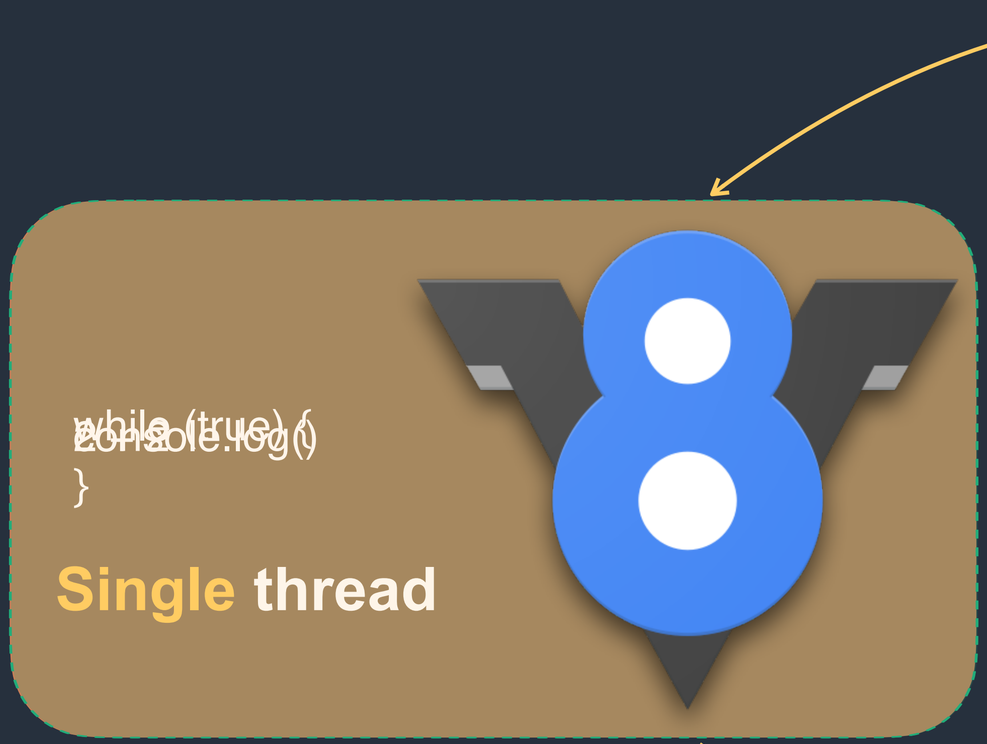
GraalVM™

Вызываем Event Handler



Регистрируем I/O задачу

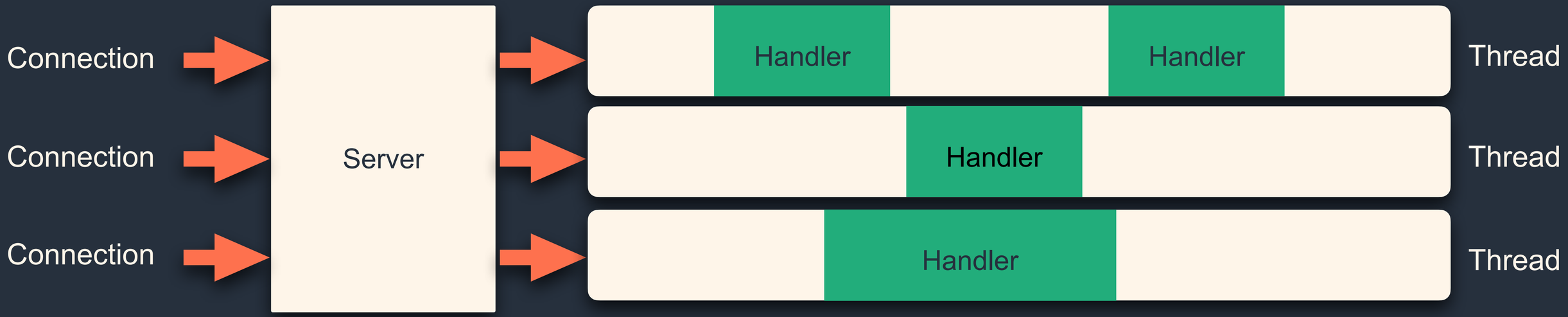
Вызываем Event Handler



Регистрируем I/O задачу

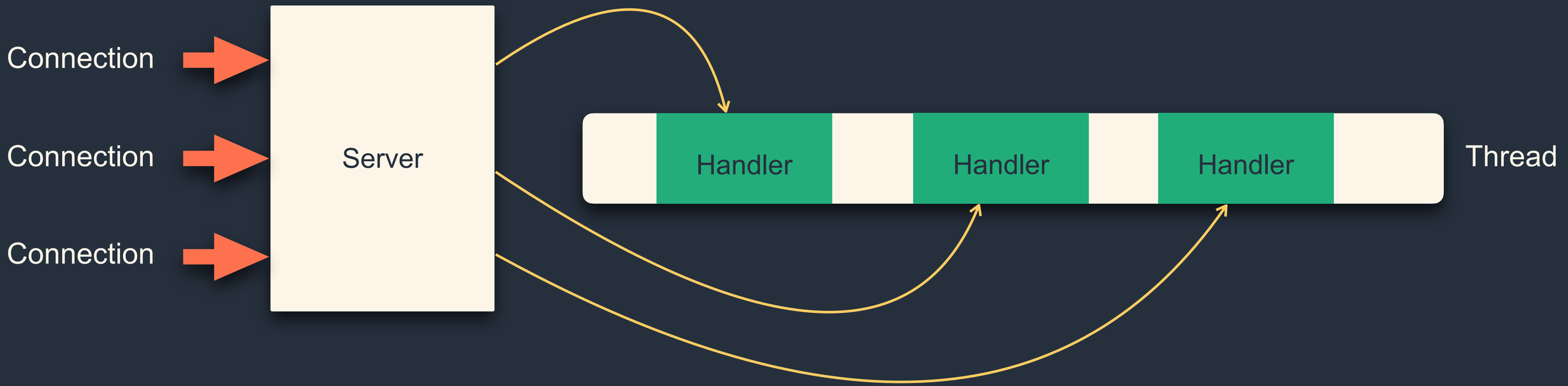
Блокирующий I/O

```
var result = db.query("select x from table_Y")
doSomethingWith(result) // waiting...
doSomethingWithOutResult() // blocking!
```



Неблокирующий I/O

```
db.query(“select x from table_Y”, function(result){
   doSomethingWith(result) //waiting...
})
doSomethingWithOutResult() //non-blocking!
```



Рантайм Node.js хорош для интенсивного I/O,
но плох для операций, нагружающих CPU

Рантайм Node.js хорош для BFF, но плох для «настоящего» бэкенда

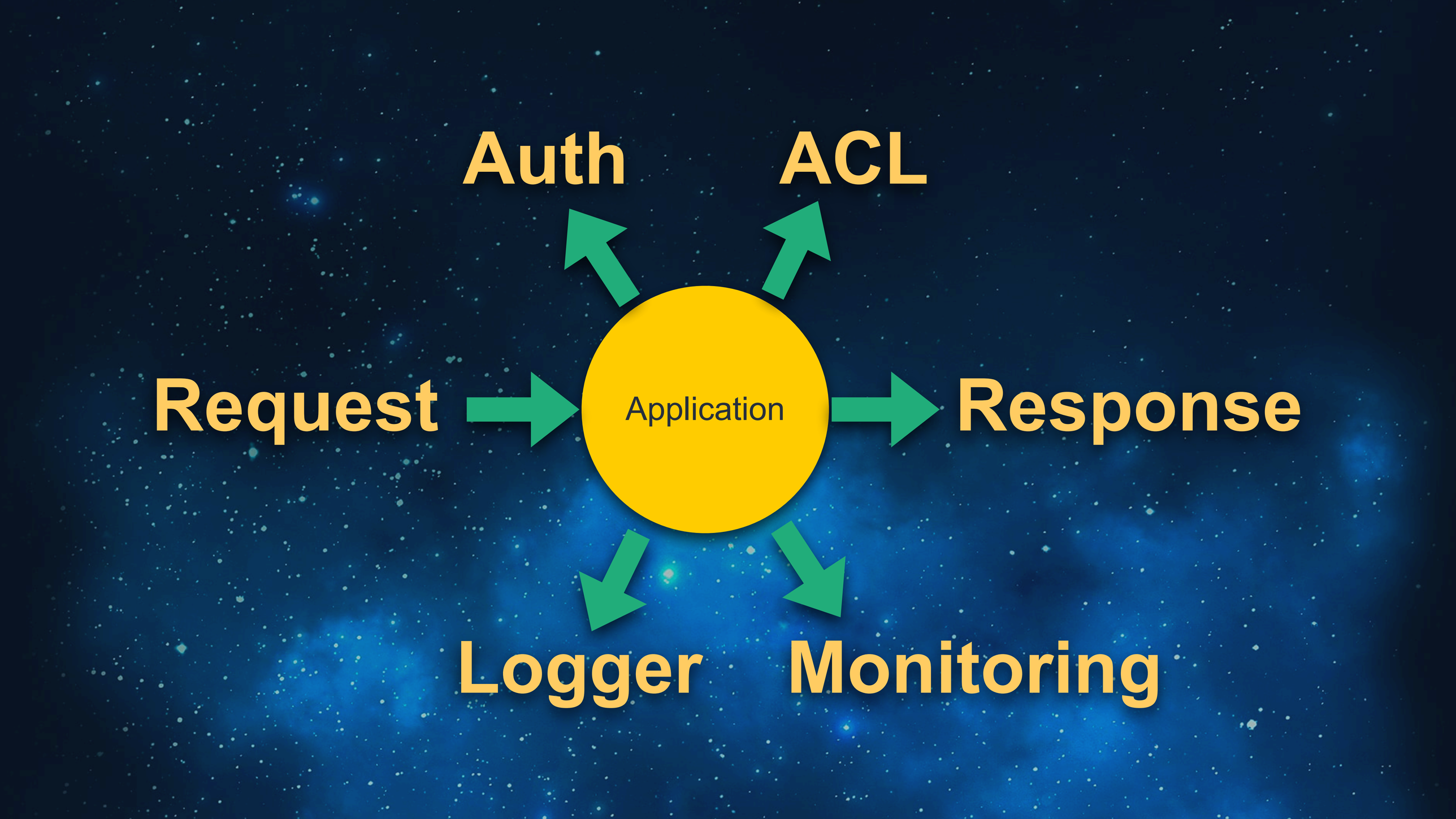
+ node JS =

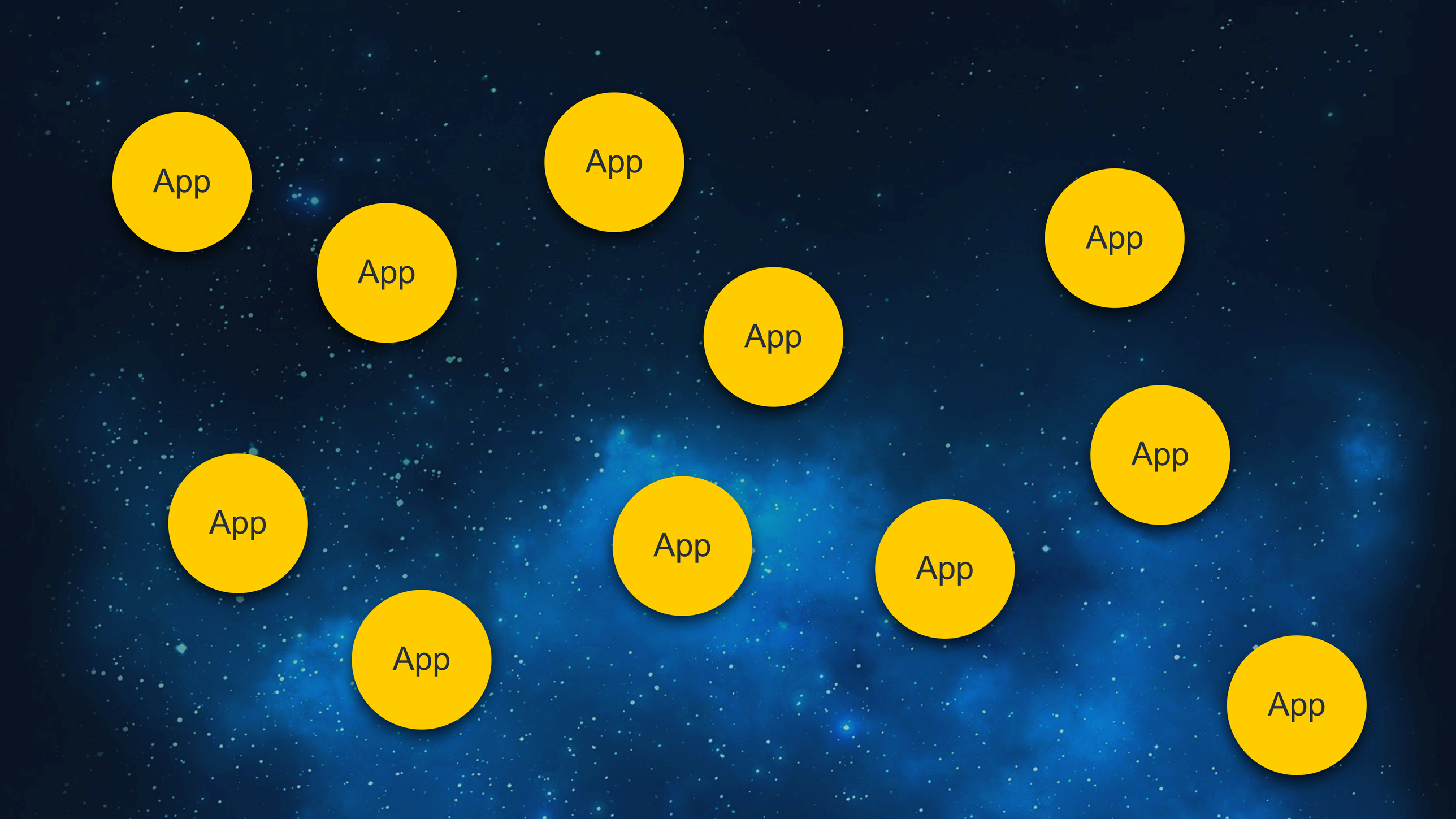
Типичный Express

```
const express = require('express');
const app = express();
const {createReadStream} = require('fs');
const path = require('path');
const Joi = require('joi');
app.use(express.json());
const schema = {id: Joi.number().required() };

app.get('/example/:id', (req, res) => {
    const result = Joi.validate(req.params, schema);
    if (result.error) {
        res.status(400).send(result.error.toString()).end();
        return;
    }
    const stream = createReadStream( path.join('..', path.sep, `example${req.params.id}.js` ));
    stream
        .on('open', () => {stream.pipe(res)})
        .on('error', (error) => {res.end(error.toString())})
});
```


Req
Middleware
Middleware
Middleware
Res


Auth
ACL
Request
Application
Response
Logger
Monitoring


App
App
App
App
App
App
App
App
App
App
App
App

Пора наводить порядок

Идеальный BFF

- Легко поддерживается
- Легко развивается

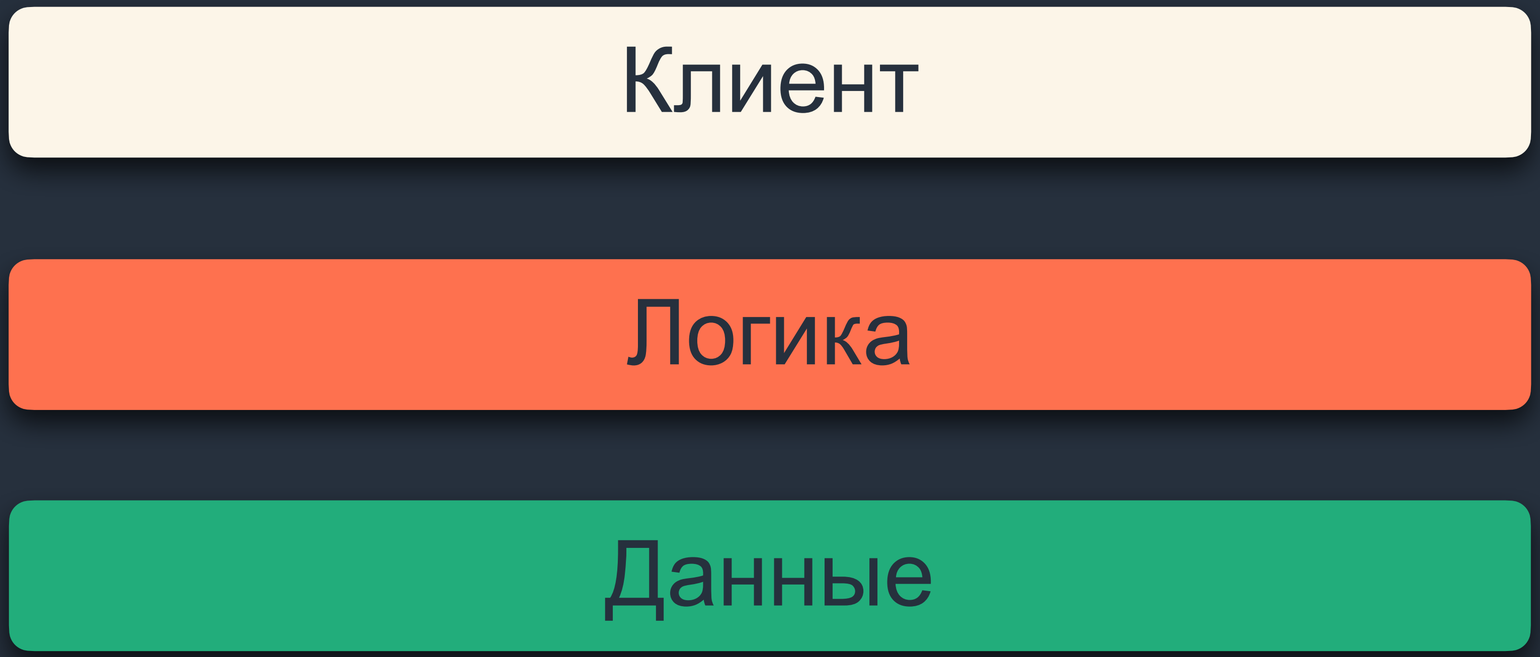
«Цель архитектуры программного обеспечения — уменьшить человеческие трудозатраты на создание и сопровождение системы.»

Из чего состоит архитектура

- Слои и
- связи между ними

Слои

Трёхуровневая архитектура



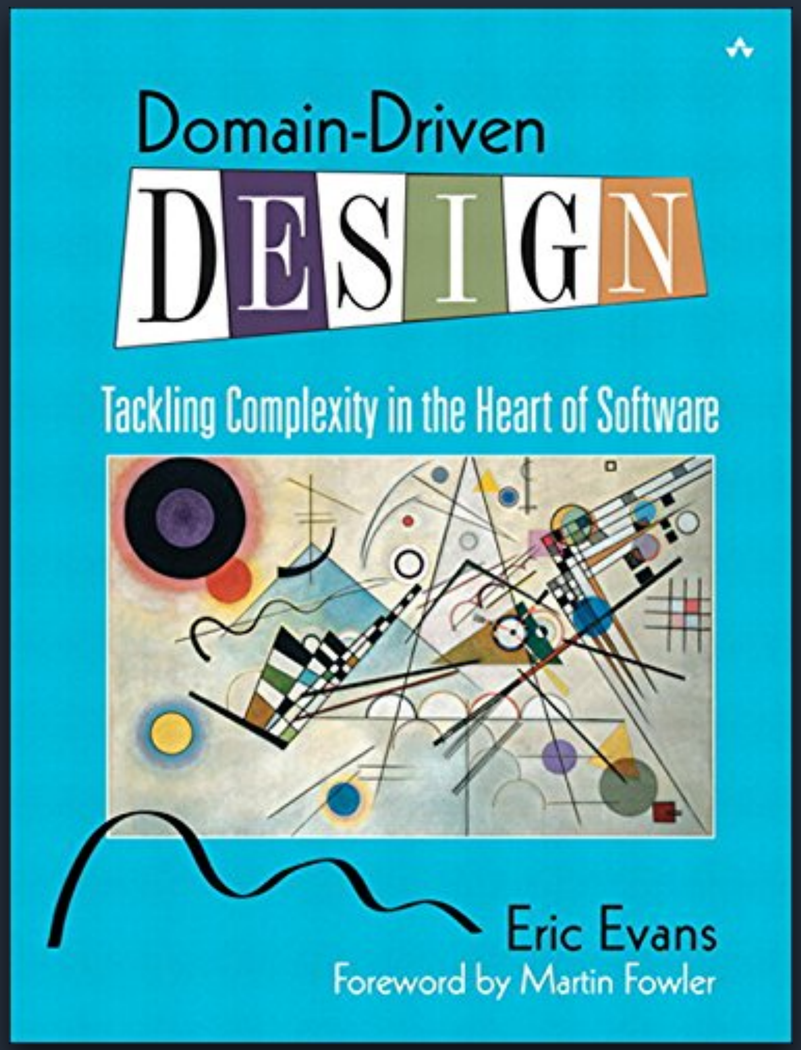
Слои по DDD

UI

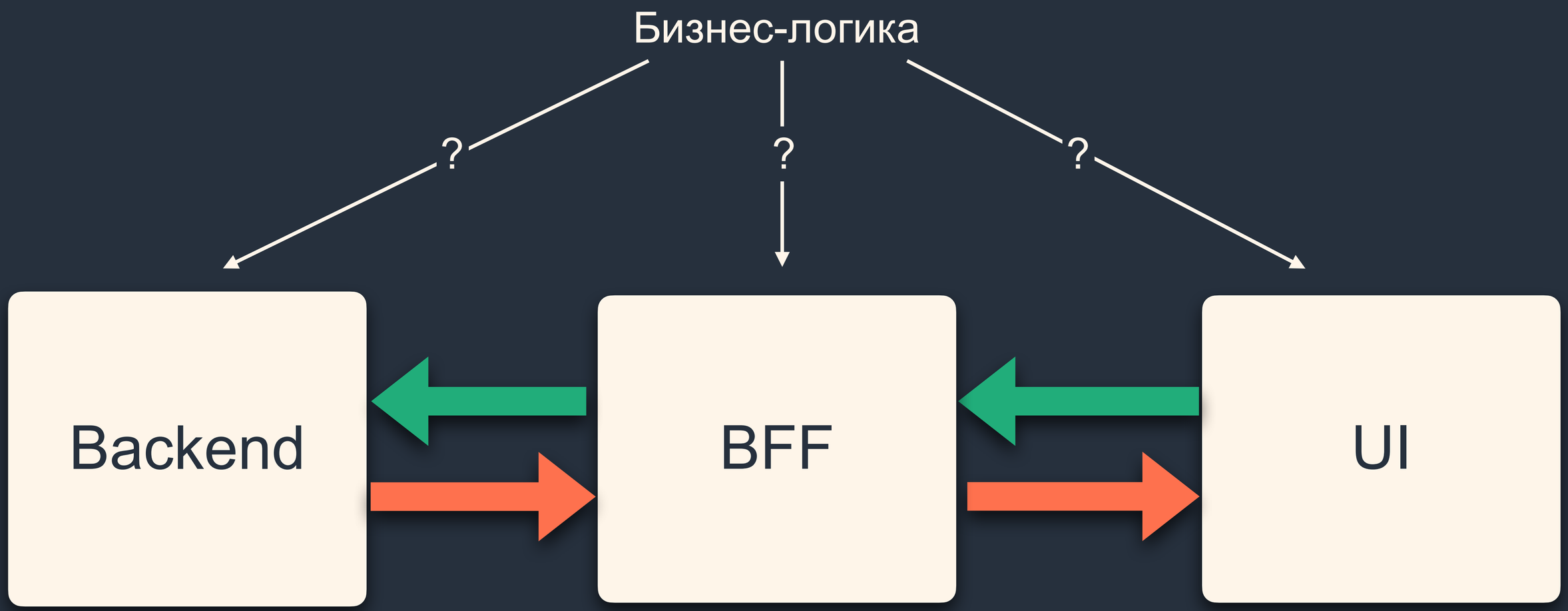
Application Layer

Domain

Infrastructure



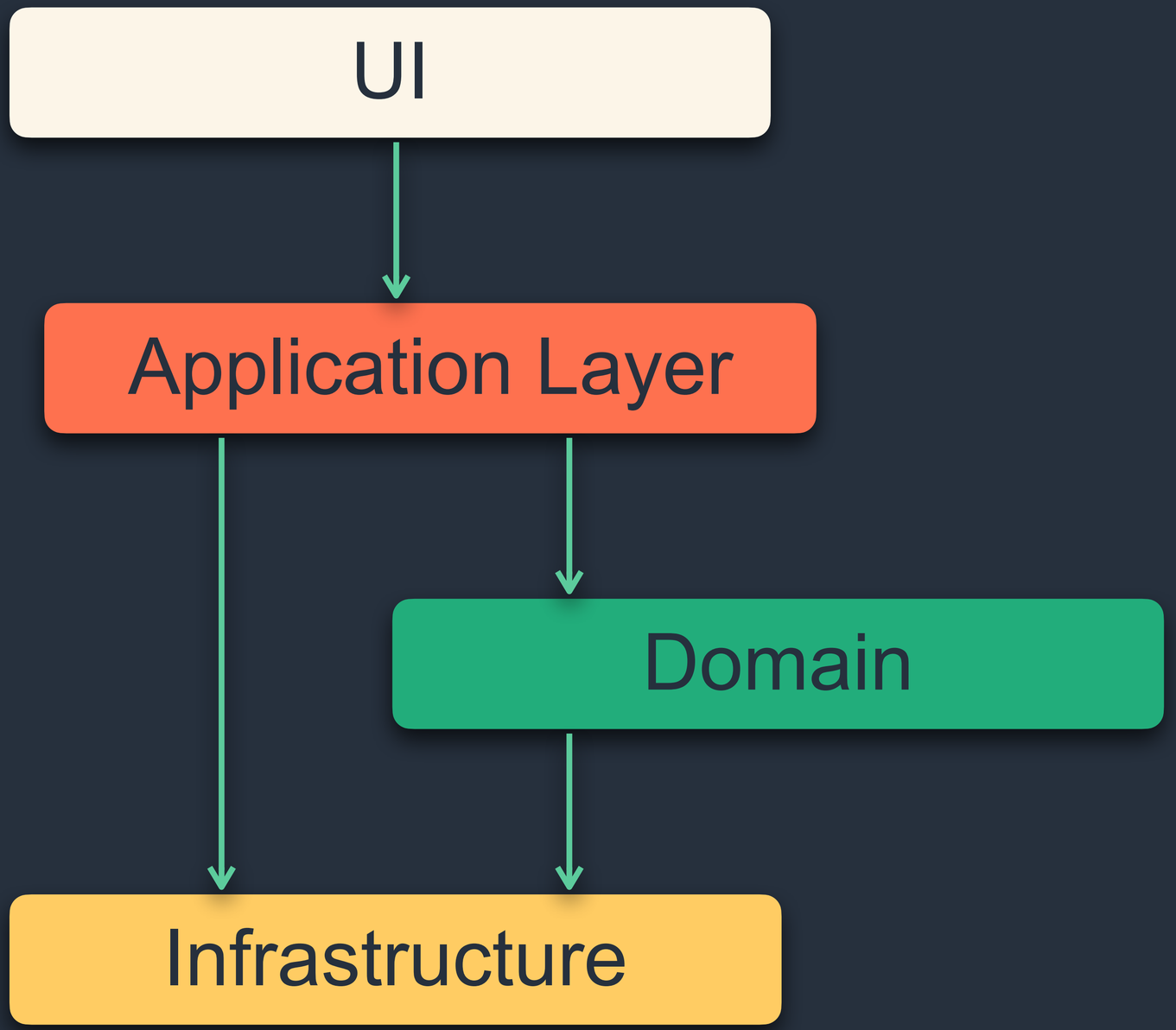
— А DDD вообще имеет какое-то отношение к фронтенду?

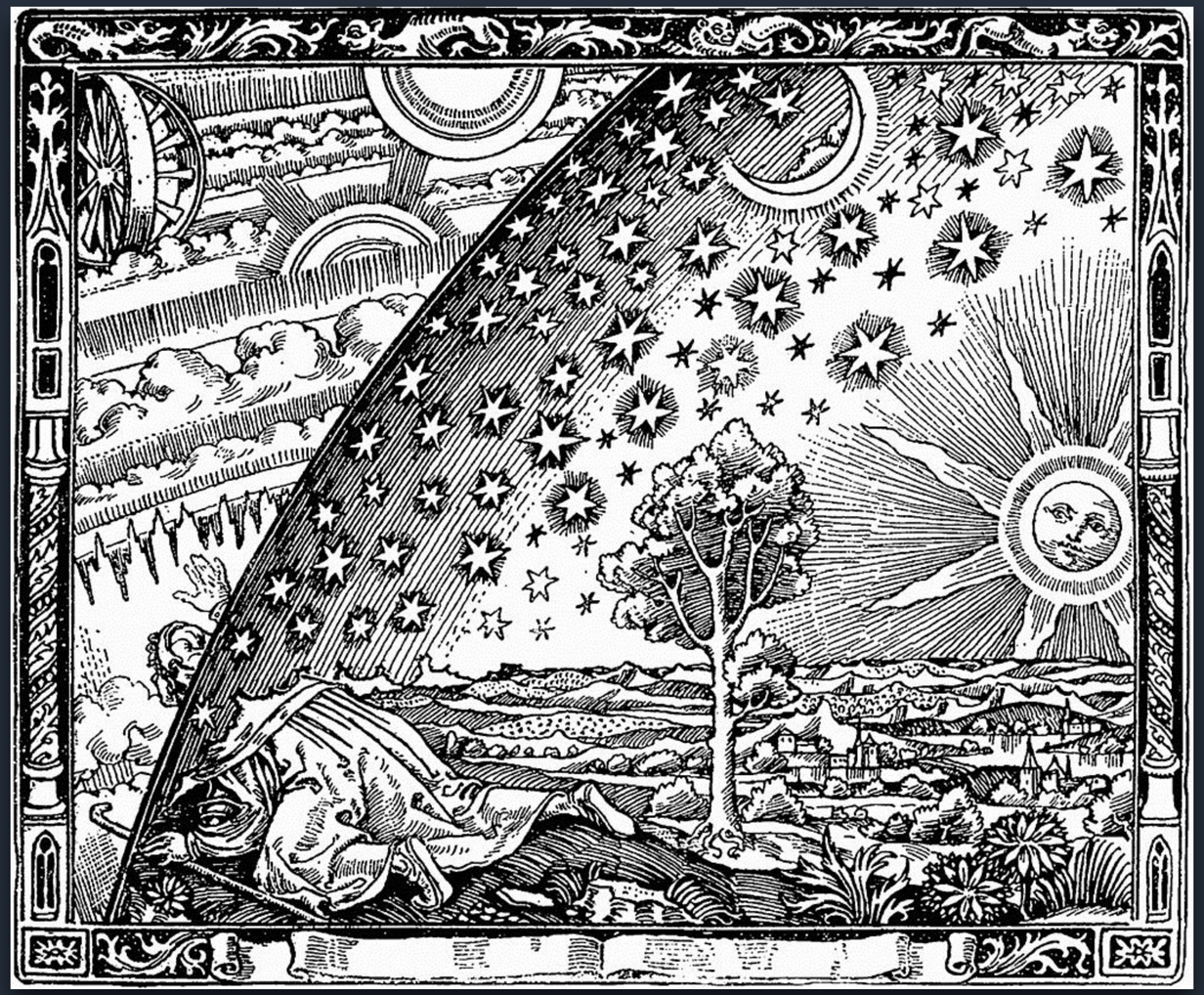

Бизнес-логика
?
?
?
Backend
BFF
UI

Примем за аксиомы

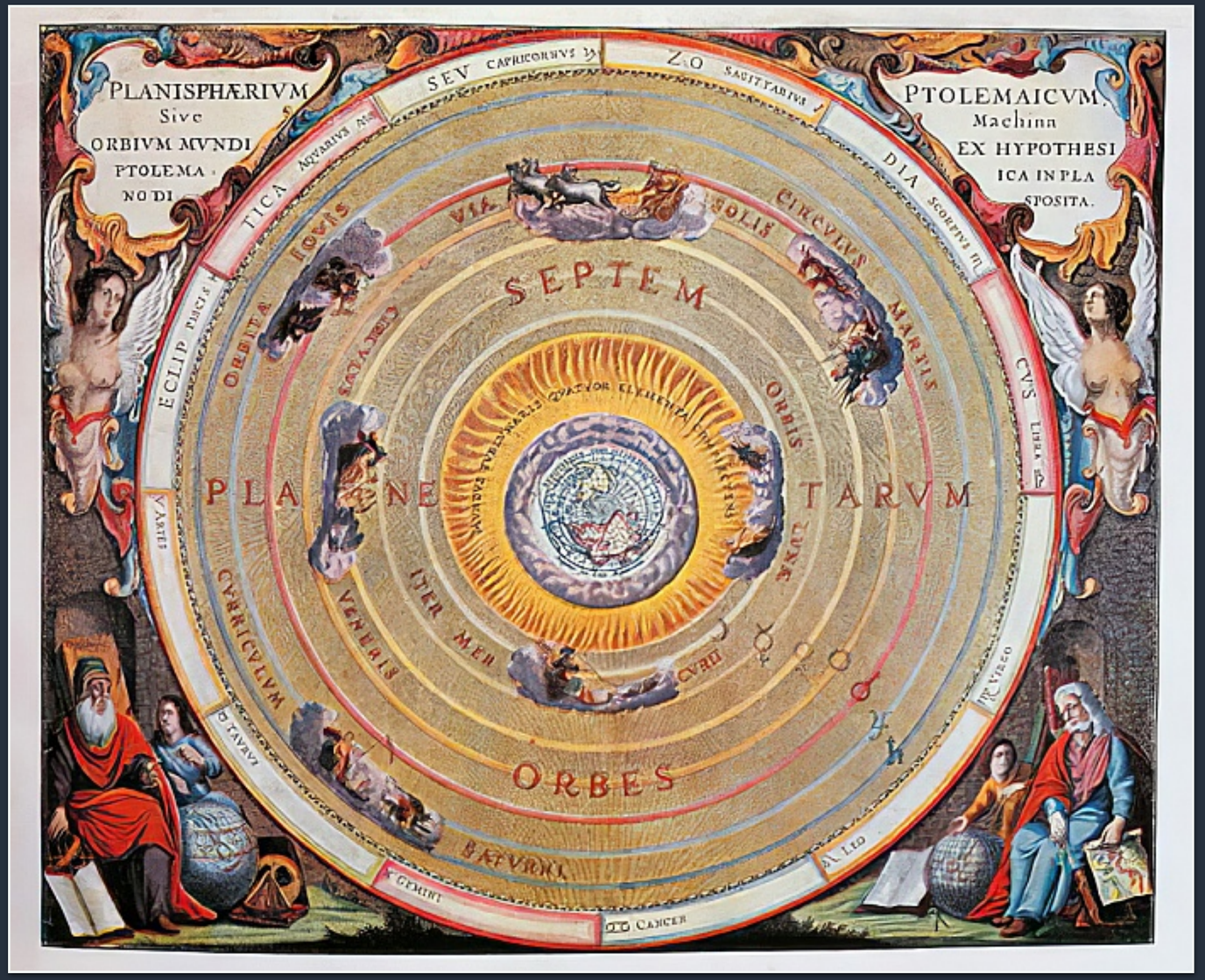
- Единый источник правды (доменная логика) — в бэкенде
- У нас всё равно всегда останется пласт логики

Слои по DDD



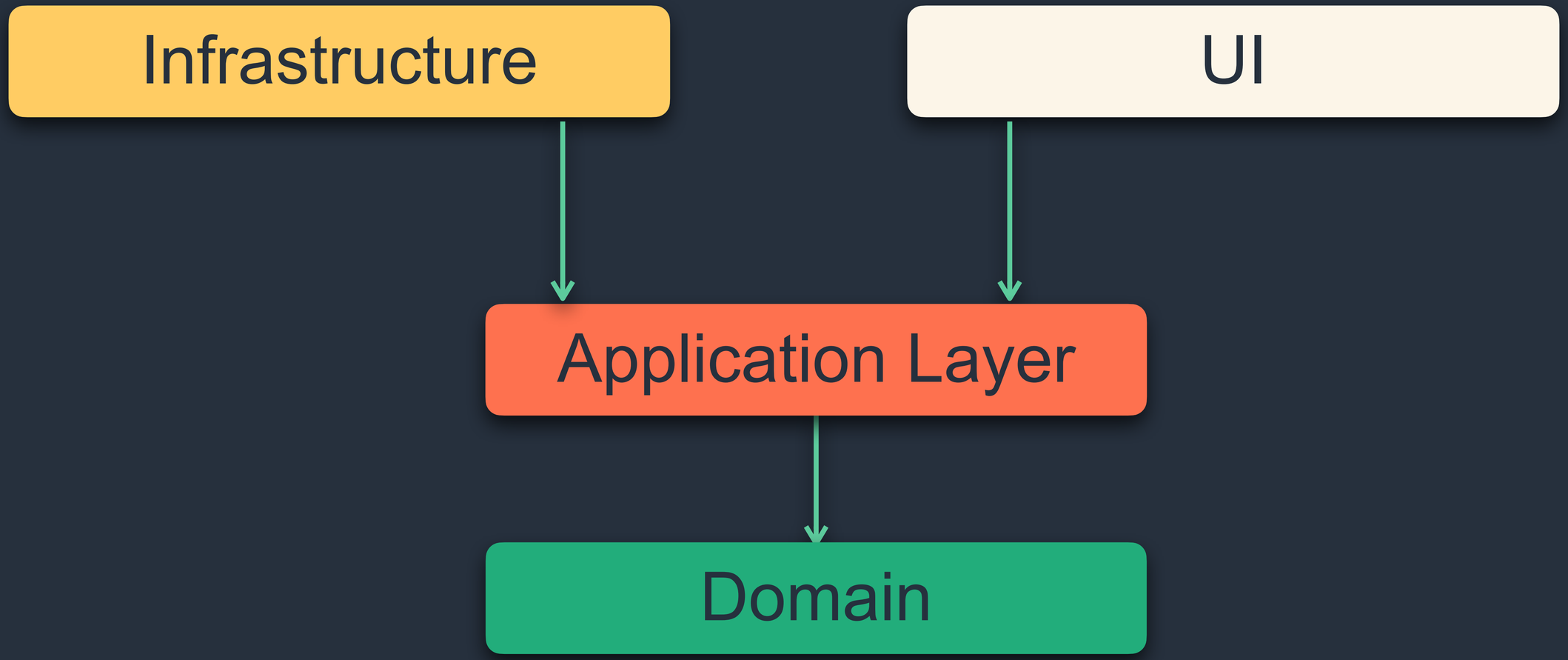


Гравюра Фламмариона «Протечка абстракций» 1554 г.

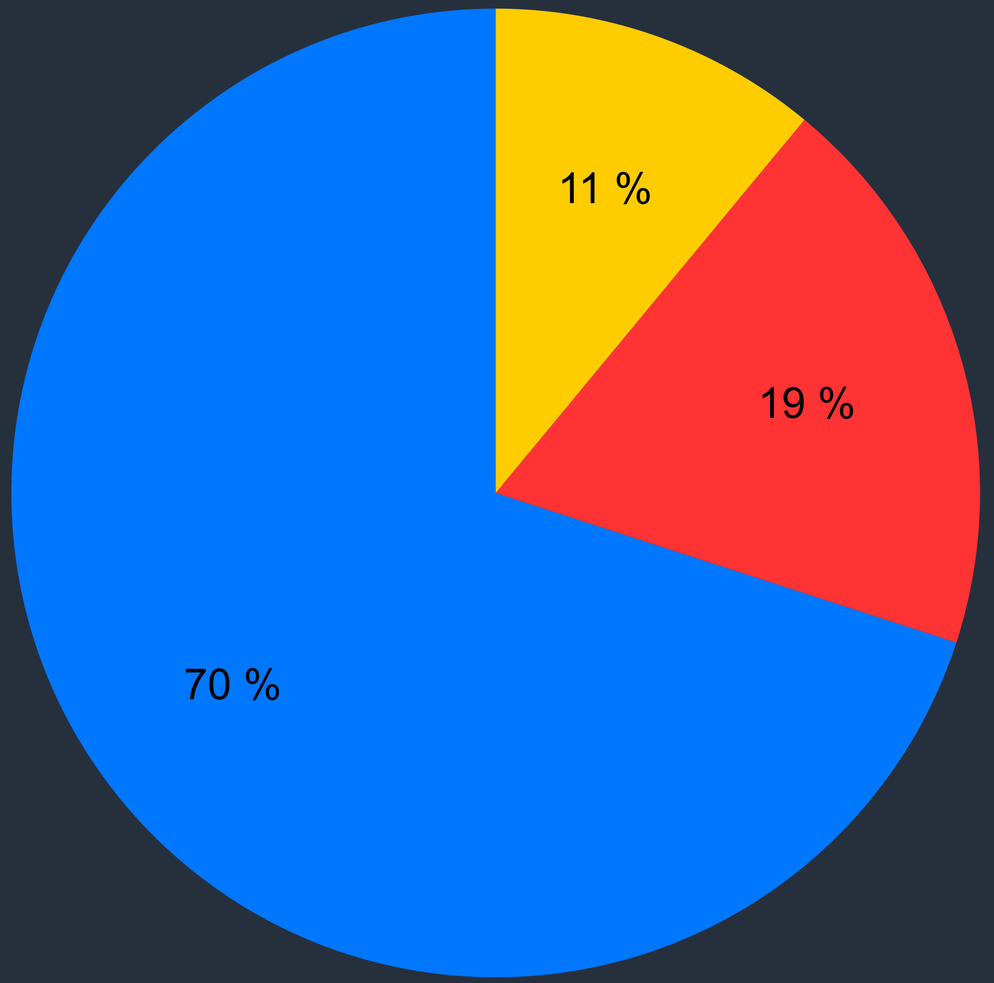


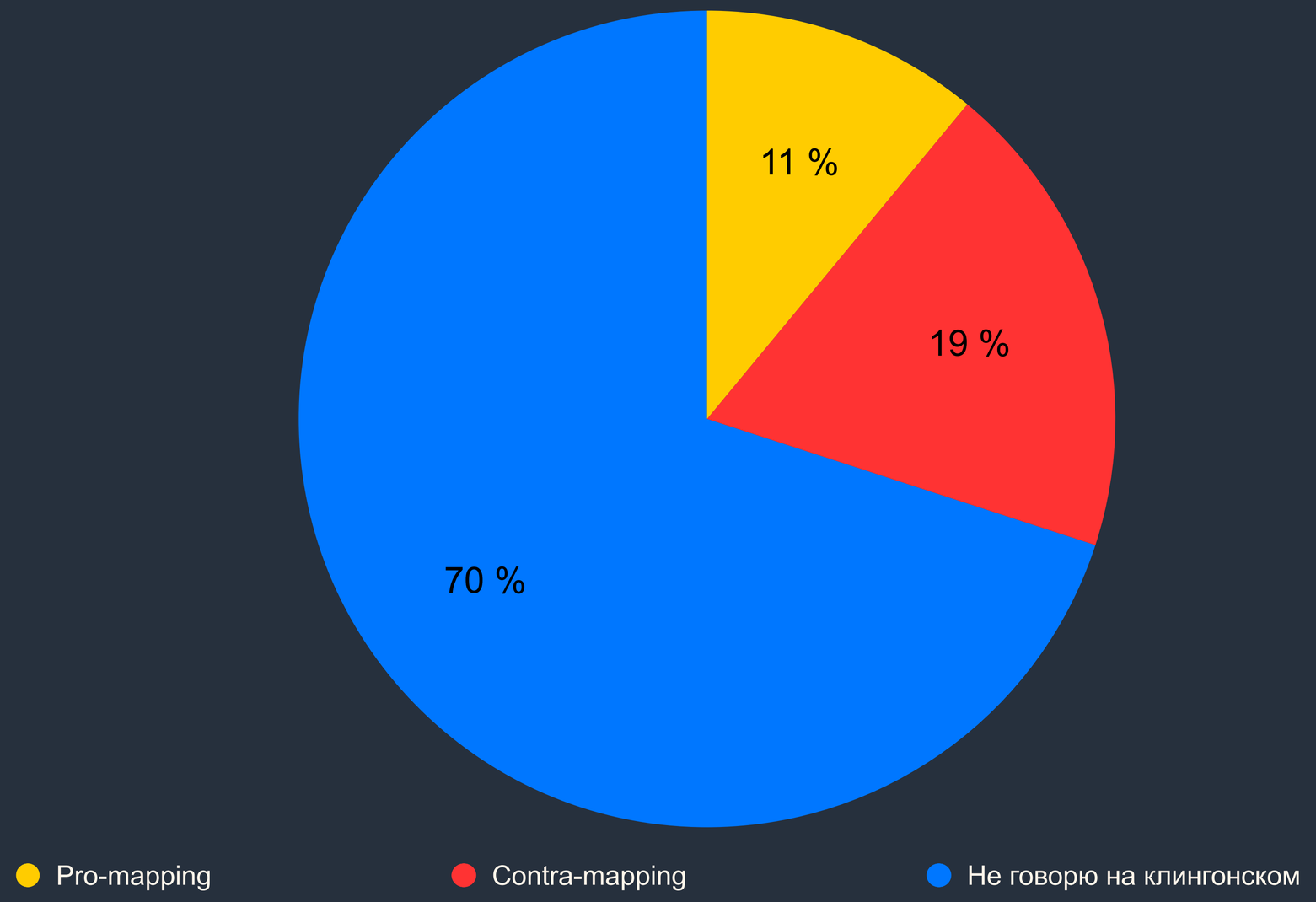
Андреас Целлариус «Чистая архитектура» 1660 г.

Слои в чистой архитектуре



Пересечение границ


11 %
19 %
70 %


11 %
19 %
70 %
Pro-mapping
Contra-mapping
Не говорю на клингонском

Аргументы

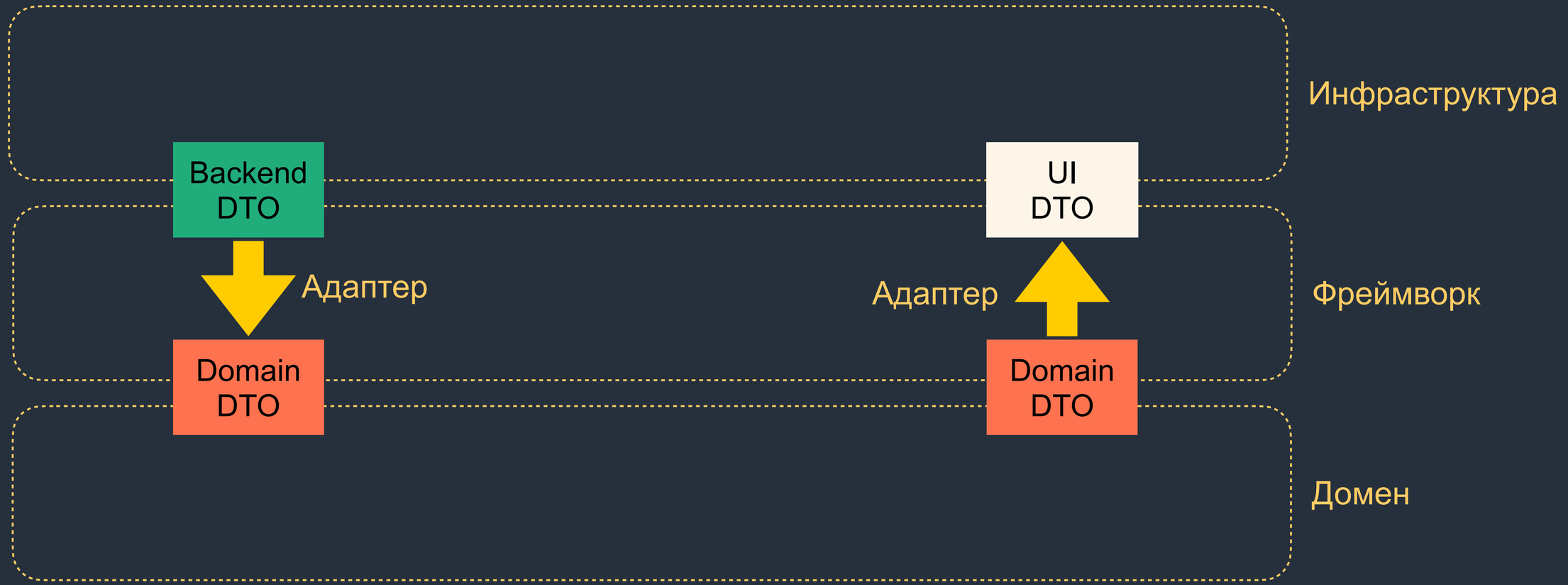
Pro

- Использование одной модели между слоями приведёт к высочайшей связности приложения!

Contra

- Использование множества DTO приведёт к тоннам боллерплейта и усложнит разработку простых CRUD

Данные и DTO


Инфраструктура
Backend DTO
UI DTO
Адаптер
Адаптер
Фреймворк
Domain DTO
Domain DTO
Домен

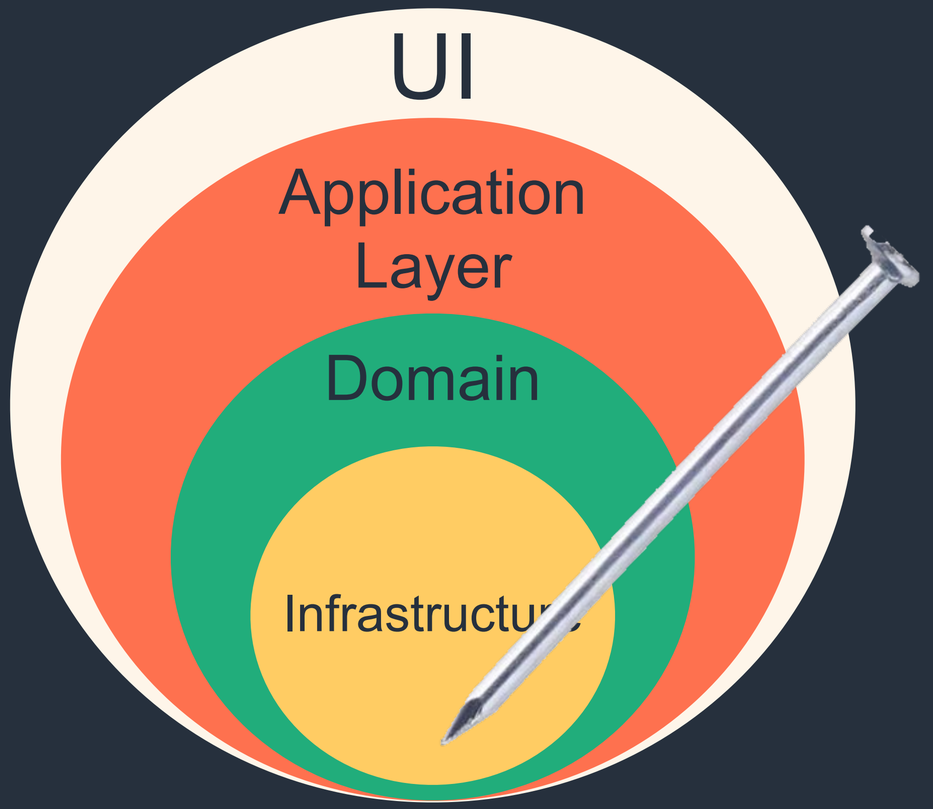
Какую стратегию выбрать?

Зависит от

Связи

Зависимости

Инфрастуктура в центре

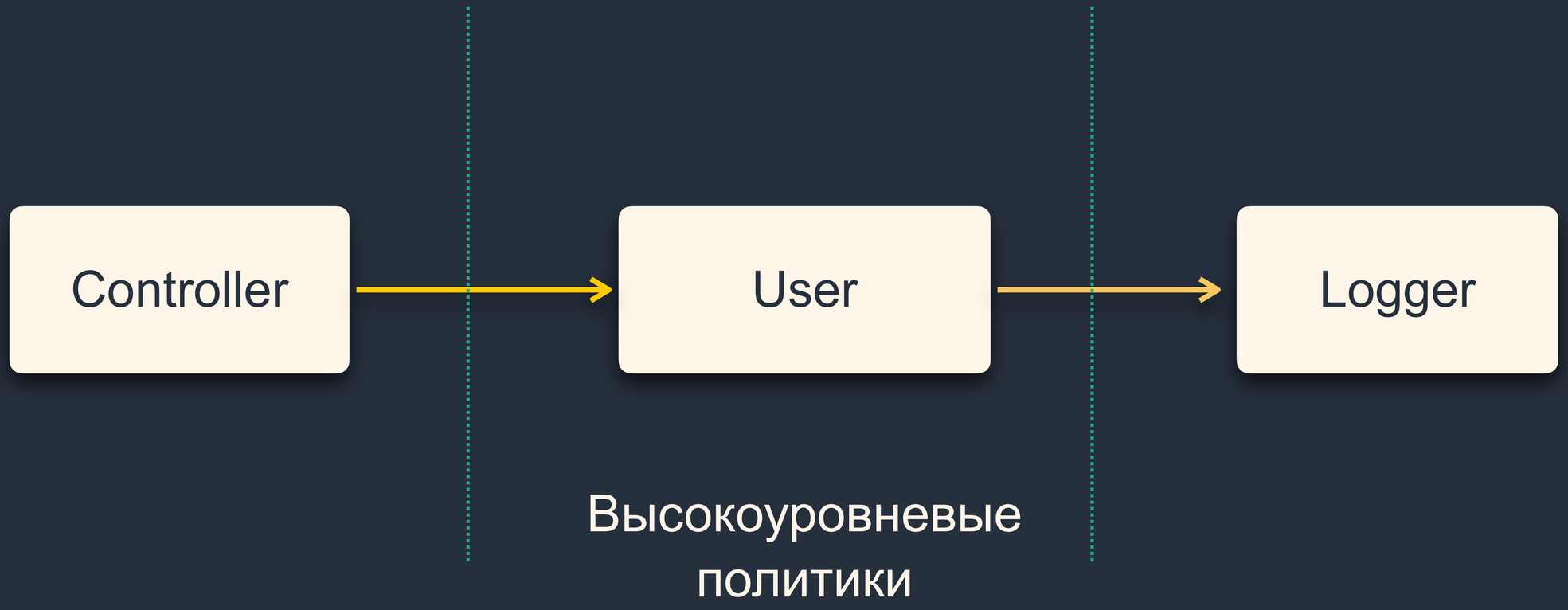


Упс

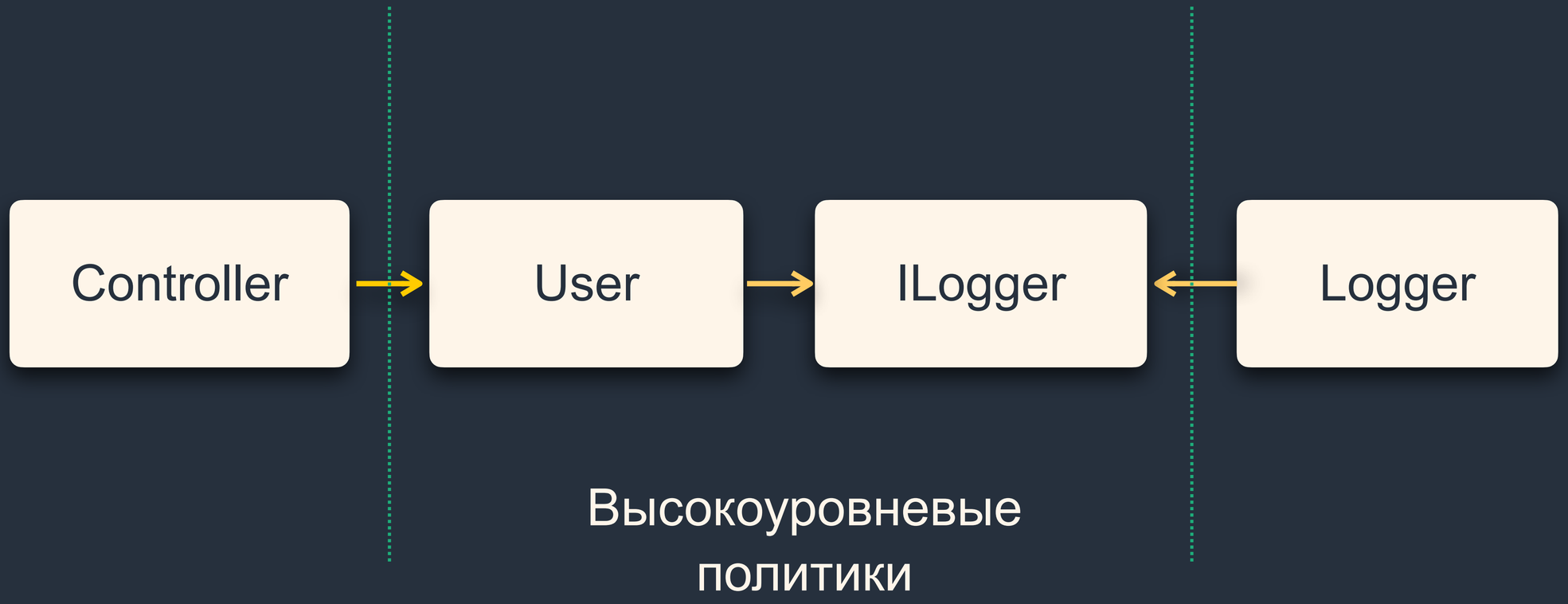
Вывернуть наизнанку

Отделение инфраструктуры

Развёрнутые зависимости



Развёрнутые зависимости



Прямые зависимости

```
export class UserService {
  private _logger: Logger;
  constructor() {
    this._logger = new Logger();
  }
  ...
}
...
const userService = new UserService();
```

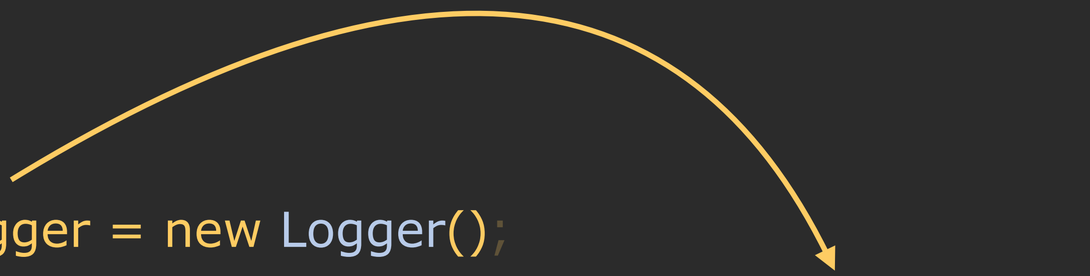
SOLID

Dependency Inversion

С помощью Dependency Injection

Простейший DI

```
export class UserService {
  constructor(private _logger: ILogger) { }
  ...
}
...
const logger = new Logger();
const userService = new UserService(logger);
```

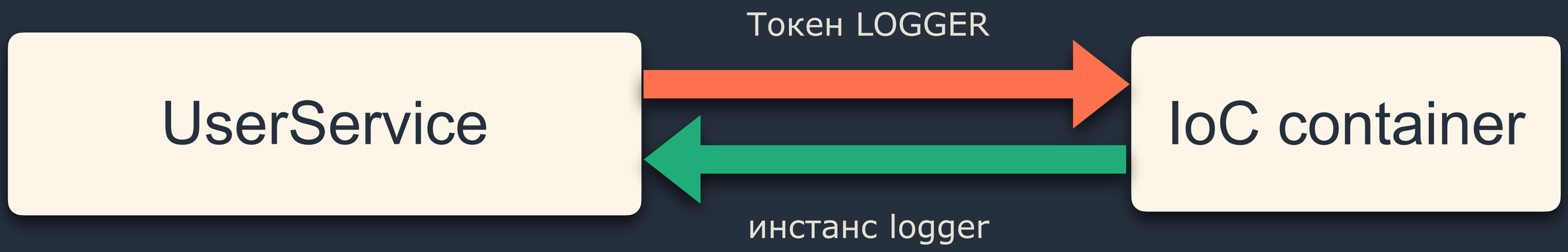


Очень много боллерплейта


C++
Вы собрали лошадь
Она адски уродлива и некоторые части болтаются, но она работает
JAVA
Вы очень хотите собрать лошадь
Но сперва вам нужно собрать лошадиную фабрику

DI на декораторах, да с контейнером

```
export class UserService {
  constructor(@Inject(LOGGER) private readonly _logger: ILogger){ }
}
```



Фреймворки нас спасут?

Варианты

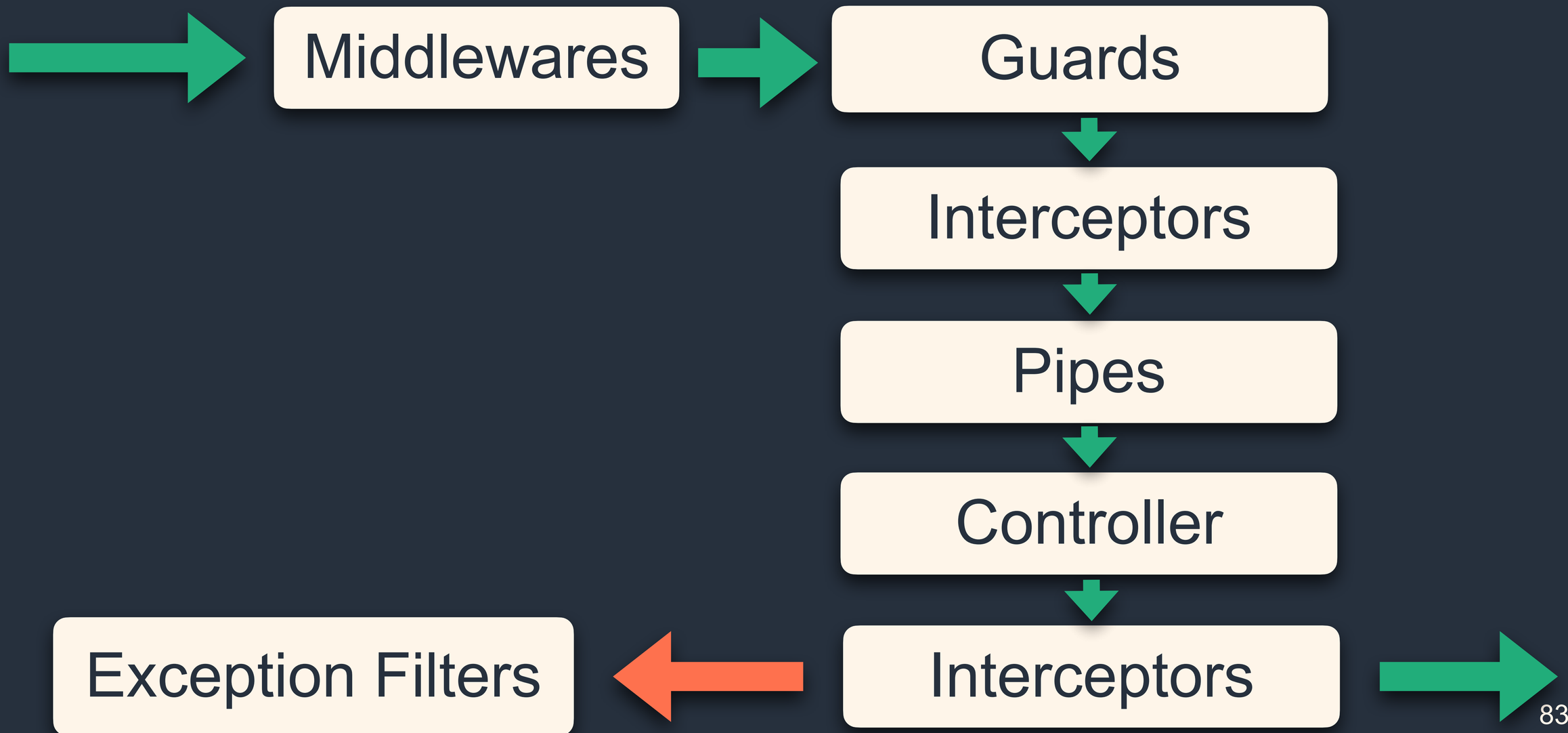
- Express + TS + Inversify
- Loopback 4
- Nest

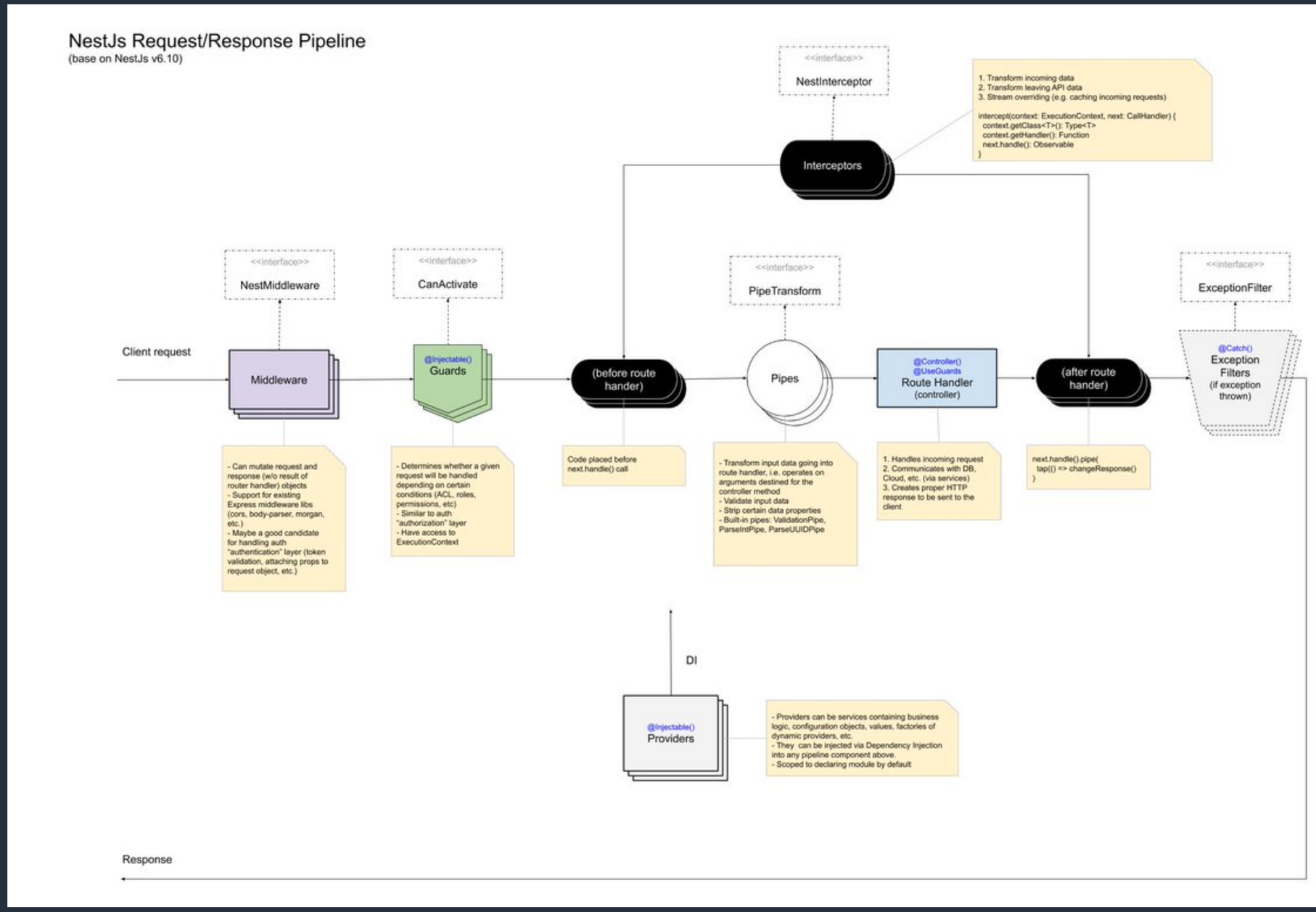
Nest

- Написан на TypeScript
- Построен поверх Express/Fastify, совместимость в middleware
- Декларирует модульность логики
- Предоставляет IoC-контейнер

Жизненный цикл Express

Жизненный цикл Nest

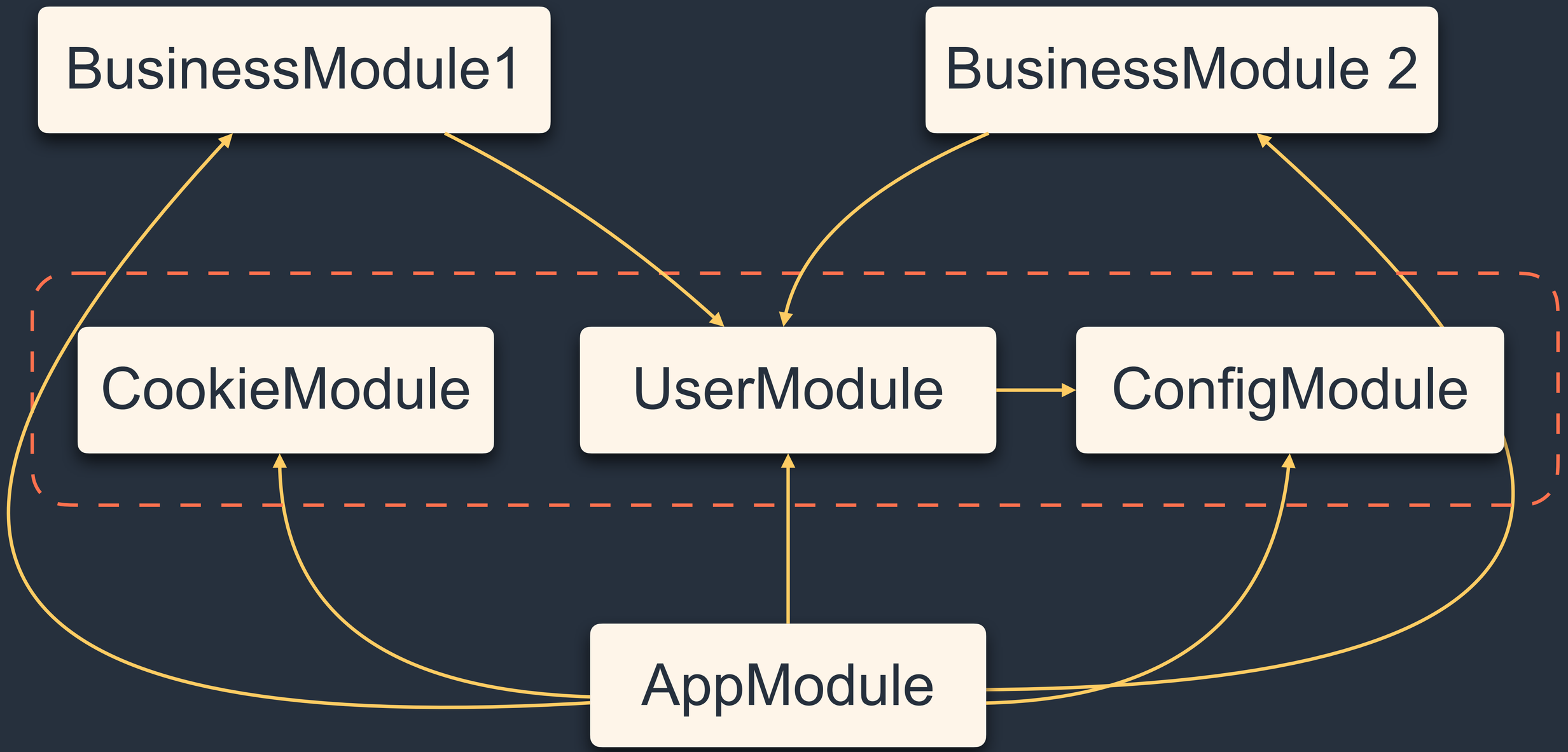




https://twitter.com/yjaaidi/status/1219930242564939776

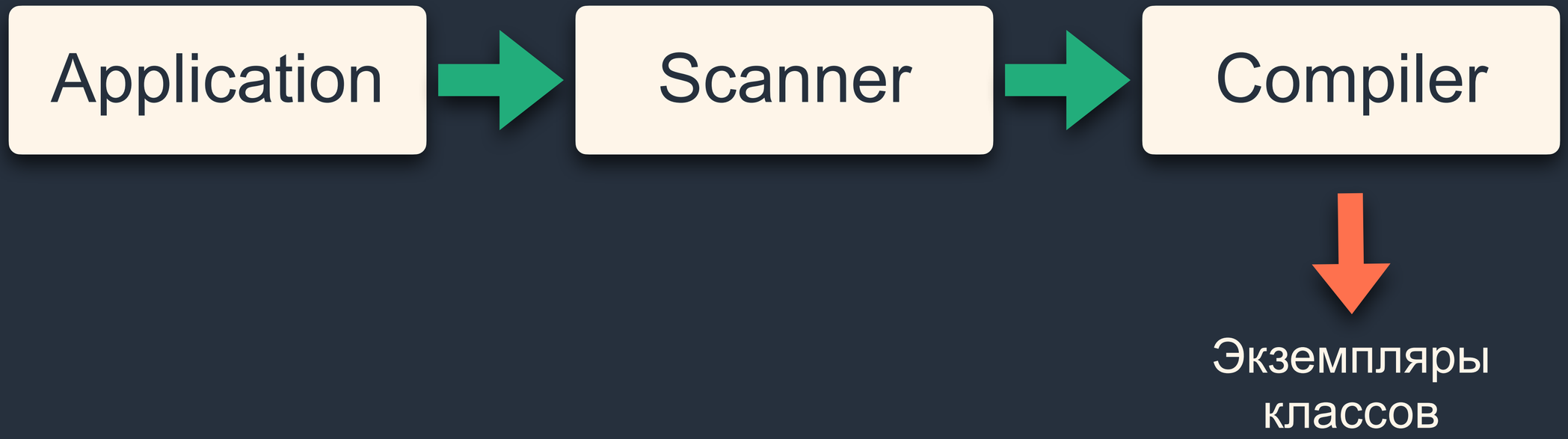
Module

это базовая сущность


BusinessModule1
BusinessModule 2
CookieModule
UserModule
ConfigModule
AppModule

Корневой модуль приложения

```
@Module({
    providers: [/*...*/],
    controllers: [/*...*/],
    imports: [/*...*/],
    exports: [/*...*/],
})
export class AppModule implements NestModule {}
```


Application
Scanner
Compiler
Экземпляры
классов


Ну и что у нас плохого?
Всё это добром не кончится.

Проблемы nest

Декораторы

Dependency Injection

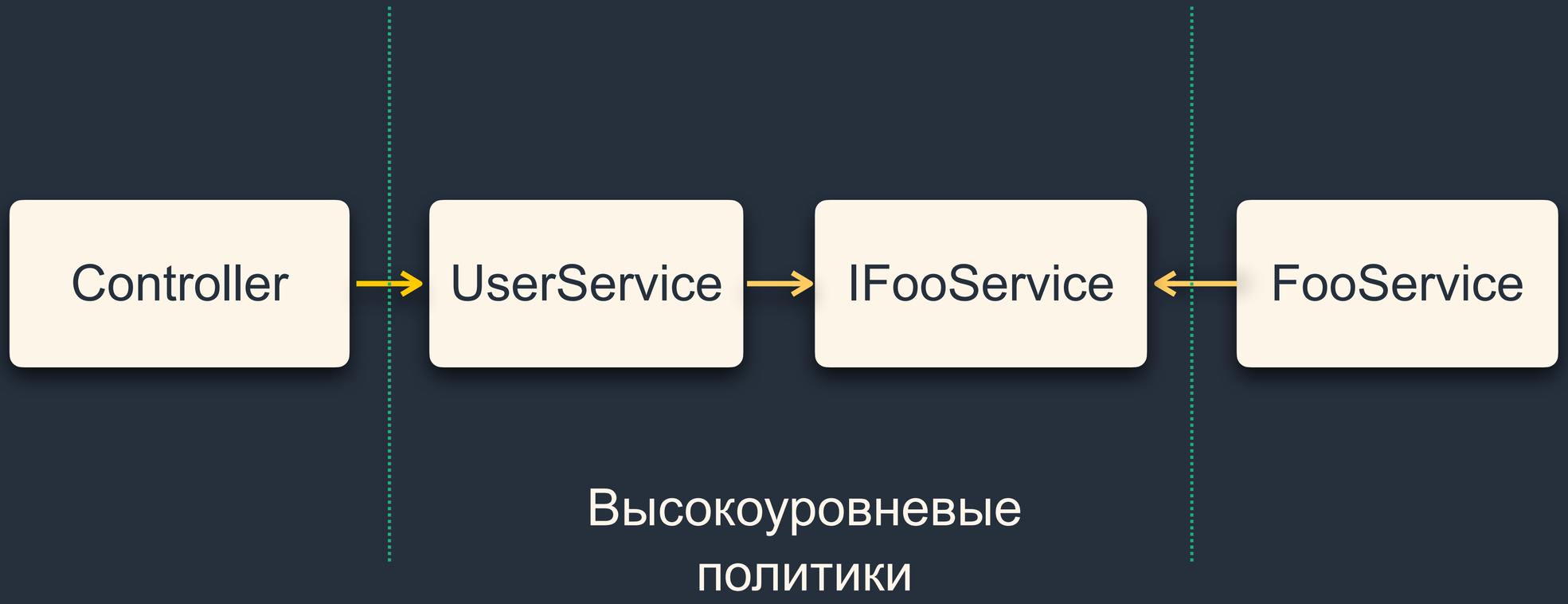
B nest

Проблемы IoC-контейнера nest

```
import {FooService} from 'FooService';
@Injectable()
export class UserService {
  constructor(
        private readonly _fooService: FooService
  ){ }
}
```


Инверсии
нет

Развёрнутые зависимости



Как достигнуть инверсии

```
@Injectable()
export class UserService {
  constructor(
        private _fooService: AbstractFooService
  ){ } }
export {AbstractFooService};
```

```
{ provide: AbstractFooService, useClass: FooService }
// FooService extends AbstractFooService
```

Как достигнуть инверсии

```
@Injectable()
export class UserService {
  constructor(
        @Inject(FOO_SERVICE) private readonly _fooService: IFooService
  ){ } }
export {IFooService};
```

```
{ provide: FOO_SERVICE, useClass: FooService }
// FooService implements IFooService
```

Exceptions

```
{
  "statusCode": 403,
  "message": "Forbidden resource"
}
```

Note that behind the scenes, when a guard returns `false`, the framework throws a `ForbiddenException`. If you want to return a different error response, you should throw your own specific exception. For example:

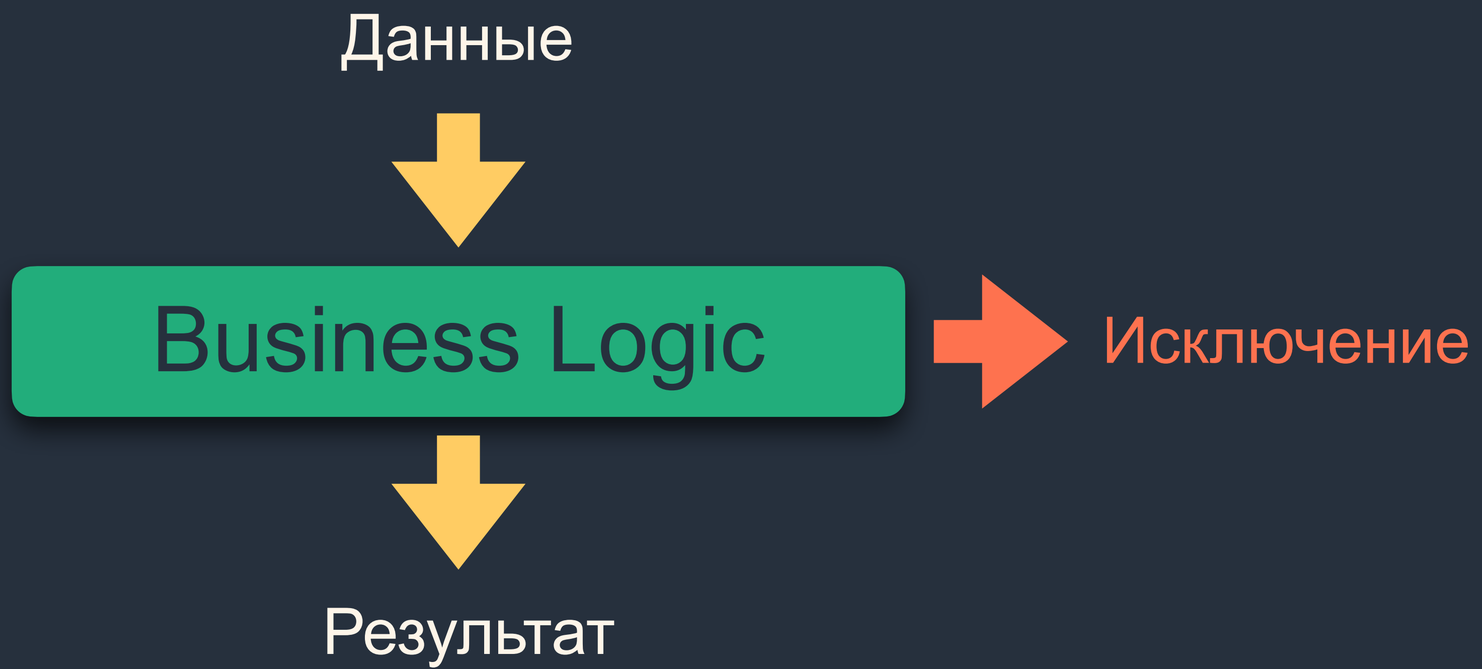
```
throw new UnauthorizedException();
```

Any exception thrown by a guard will be handled by the **exceptions layer** (global exceptions filter and any exceptions filters that are applied to the current context).

Ошибка !== Исключение

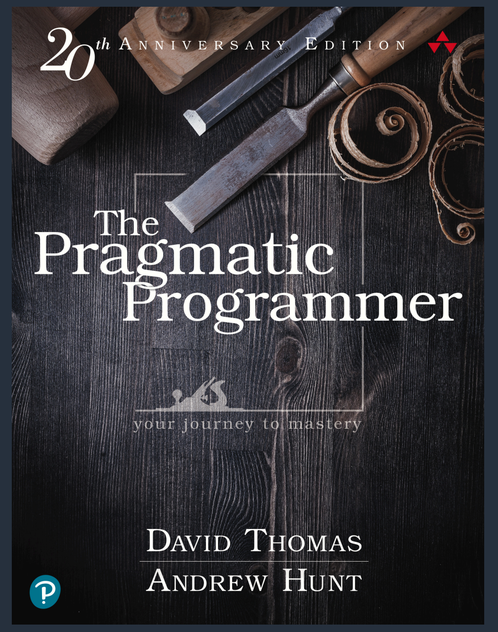
Error !== Exception

Поток данных



«Если ошибка — это ожидаемое поведение, то вы не должны использовать исключения.»

«Будет ли этот код работать, если я удалю все обработчики исключений?» Если ответ «нет», то, возможно, исключения используются в неисключительных обстоятельствах.

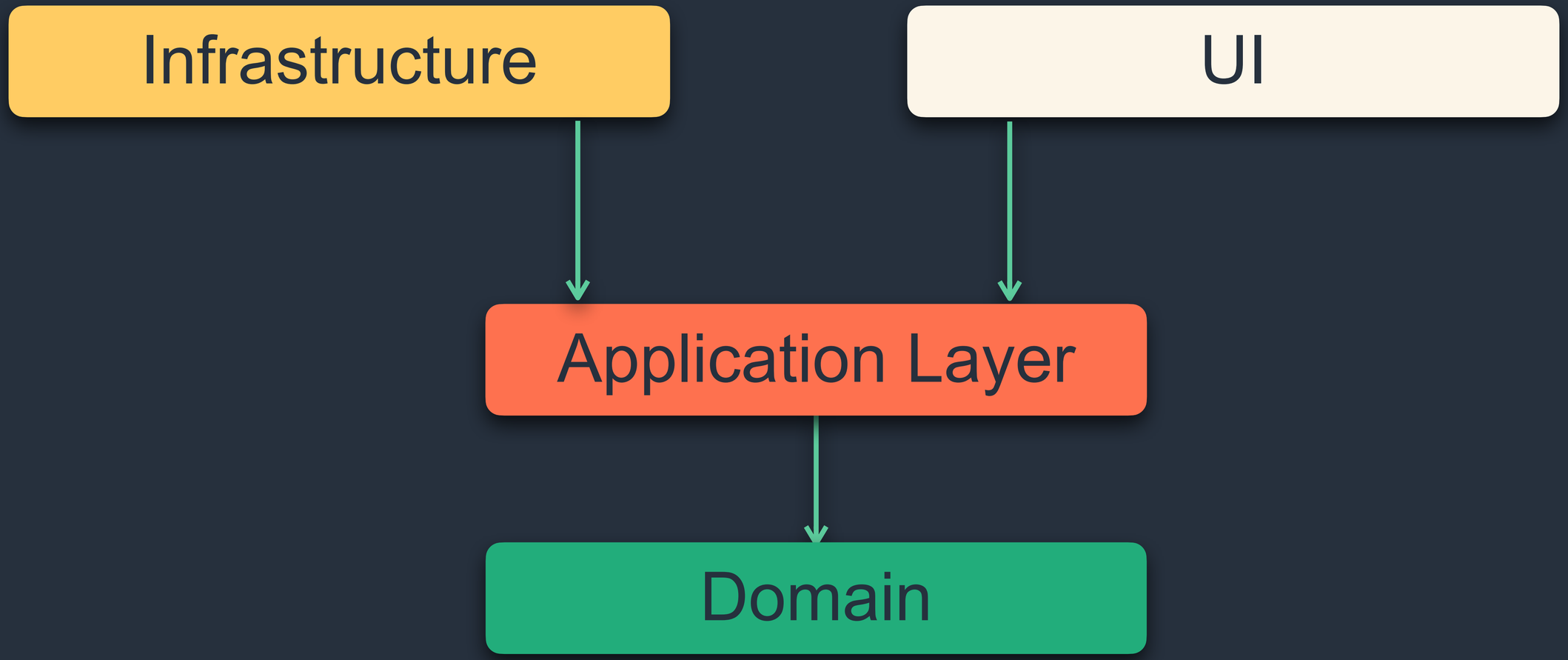


Контейнер Result

```
const successResult = Result.ok(false);
const failResult = Result.fail(new ConnectionError())
```

Разложим nest по слоям

Домен в центре



Сущности nest и DDD

Controllers

Pipes

Guards

Interceptors

Exceptions

ExceptionFilters

Middlewares

Services

Сущности приложения

UI	Infrastructure	Application Layer	Domain
React	Gateways (DAL)	Controllers	Entities
	Logger	Pipes	Use Cases
	Config	Guards	
	Monitoring	Interceptors	
		Exceptions	
		ExceptionFilters	
		Middlewares	
		Services	

Сущности приложения

Infrastructure

Gateways (DAL)

Logger

Config

Monitoring

UI

React

Framework

Controllers

Pipes

Guards

Interceptors

Exceptions

ExceptionFilters

Middlewares

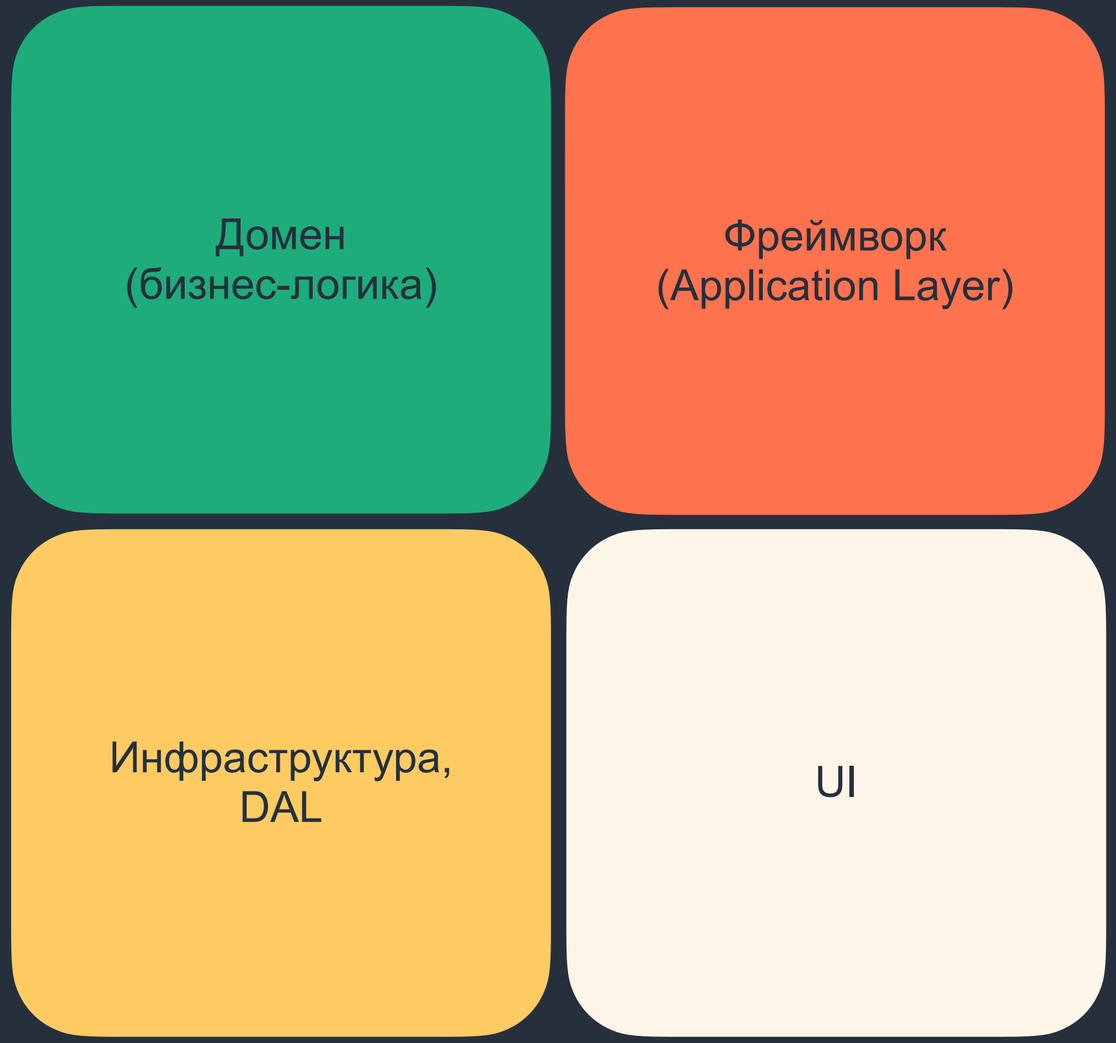
Services

Domain

Entities

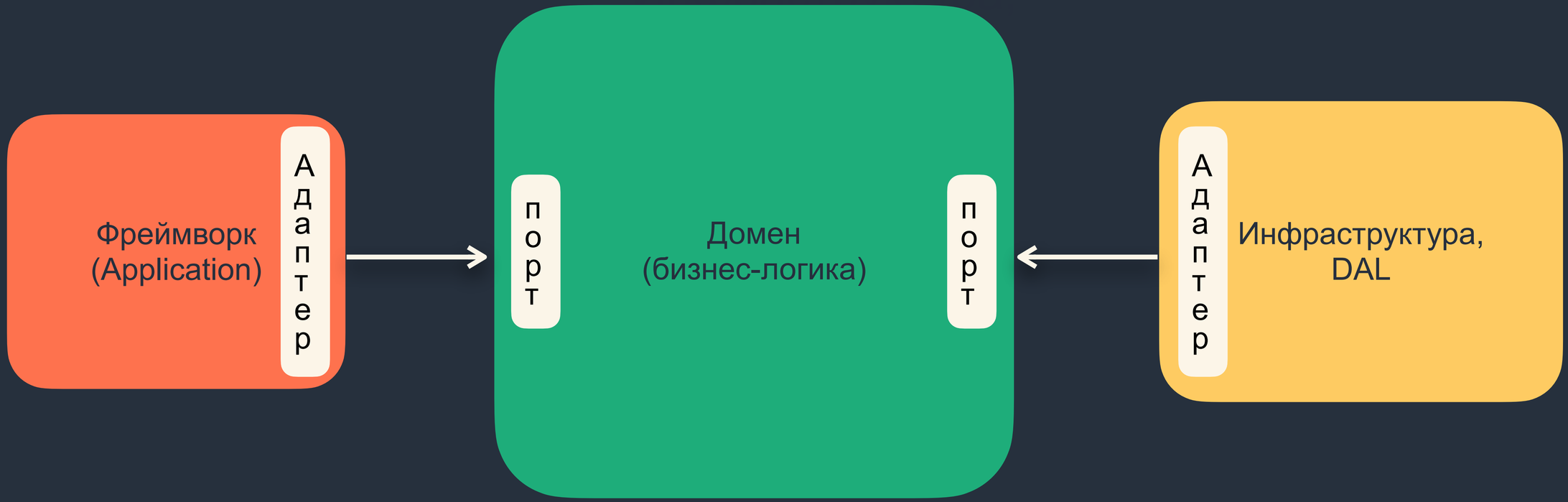
Use Cases

Независимые части приложения



«Фреймворк — это деталь.»

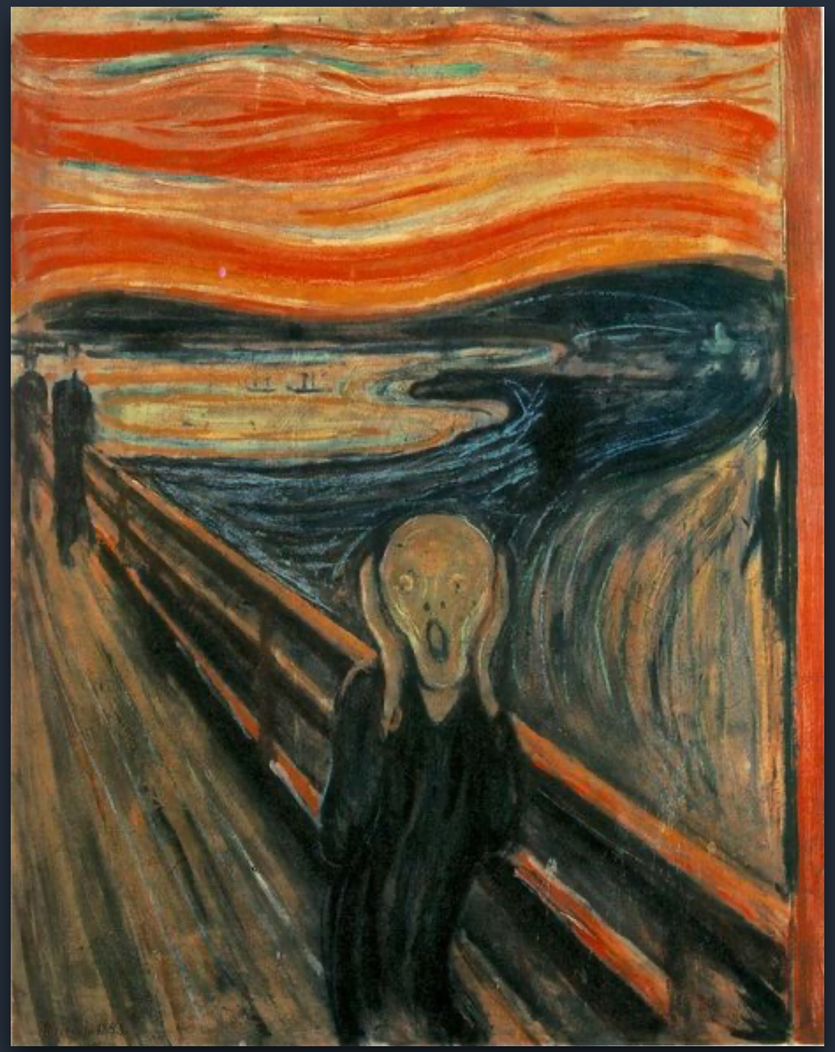
Порты и адаптеры



Реализация

«Не позволяйте фреймворку отравить бизнес логику.»

Файловая структура



Эдвард Мунк «Кричащая архитектура» 1893 г.

```
src
    client
    common
    server
        domains
            entities
            ports
                in
                out
            services
        modules
            adapter-in
                adapter-in.module.ts
                adapter-in.controller.ts
            adapter-out
                adapter-out.module.ts
                adapter-out.orm-entity.ts
        app.module.ts
    main.ts
```

Итого

Уроки, которые мы вынесли

- Node.js — это хорошо. Для BFF. С ней можно жить
- Готовых решений нет
- Фрейморки не важны
- Если ваша архитектура становится слишком сложной, если вы упираетесь в типизацию — возможно вы выбрали не тот инструмент

Спасибо,
что не уснули!

Андрей Мелихов

andrey.melikhov@hey.com