



# Платформа Node.js

API, архитектура и фреймворки

# Игорь Антонов

- Team Lead в Тинькофф;
- Программный комитет Podlodka Tech/Java Crew (<https://podlodka.io/techcrew>)
- Программный комитет Infostart Tech Event (<https://event.infostart.ru>)
- YouTube канал «Про JavaScript и разработку» (<https://youtube.com/@antonovjs>)
- Лектор по JS/Node.js в HTML Academy (<https://htmlacademy.ru>)

# Что вы знаете про Node.js



## Платформа

Платформа для выполнения JavaScript на стороне сервера, но есть и другие



## Странный Express

Минималистичный веб-фреймворк Express для быстрого построения REST API



## Node.js быстрая

Быстрая в определённых случаях. Есть более «быстрые» альтернативы в виде Bun.



## Без пакетов никуда

При разработке требуется много пакетов.  
Сколько весит ваша **node\_modules**?

# **Node.js и сложные проекты**

# Фреймворки

## ASP .NET Core

- Microsoft, C# и большой набор инструментов

## Spring

- Java и проверенный временем фреймворк Spring

## Laravel

- PHP и популярные решения для Enterprise-разработки: Laravel, Symphony, Yii.

# Три истории про Node.js



## API

Платформа активно развивается, появляется новый API, который часто проходит мимо.

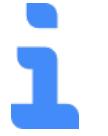
Исследования вроде State of JS подтверждают это.

Прямая замена многим пакетам.



## Архитектура приложений

Простые приложения создавать на Node.js легко, но сложно поддерживать.  
Базовые принципы построения архитектуры.



## Фреймворки

Готовые решения для разработки приложений корпоративного уровня на Node.js



ТИНЬКОФФ

# История первая Встроенный API

# Протокол node:



```
1 import { writeFile } from 'node:fs';
2 import { pathToFileURL } from 'node:url';
3 import { resolve } from 'node:path';
4
5 // VS
6
7 import { writeFile } from 'fs';
8 import { pathToFileURL } from 'url';
9 import { resolve } from 'path';
10
```

- Поддержка протокола `node:` появилась в Node.js 16. Портировали до 14-й версии. Добавляет явности. Документация использует по умолчанию.

- Как соблюдать правило? Использовать плагины для ESLint (<https://github.com/sindresorhus/eslint-plugin-unicorn>)

# crypto.hash()



```
1 import * as crypto from 'node:crypto';
2
3 const value = 'ИТМО';
4 console.log(crypto.hash('sha1', value));
```

- Появилась в версии 20.12. Быстрый способ получить hash для набора данных
- Производительность: быстрей в 1.2 — 2 раза по сравнению с createHash() на данных <= 5Mb
- Поддержка алгоритмов зависит от версии OpenSSL

# Environments



```
1 node --env-file=.my.env 12-env-file.mjs
2
3 > Привет, мир!
```



```
1 import { parseEnv } from 'node:util';
2
3 // 📁 Загрузит .env файл
4 process.loadEnvFile('./.my.env');
5 console.log(process.env.HELLO_WORLD);
6
7 // 📁 Вернёт объект { MYVAR: 'TEST' }
8 console.log(parseEnv('MYVAR=TEST'));
9
```

- Доступно с версии ~20. Стабильность: Active Development
- Загрузка переменных окружения через CLI
- В 20.12 добавили новый методы для RunTime

# Permission Model



```
1  {
2    "onerror": "log",
3    "resources": {
4      "./example-policy-app.js": {
5        "dependencies": {
6          "node:fs": "./myfs.js"
7        }
8      }
9    }
10 }
```



```
1  function readFileSync(path, encoding) {
2    console.log('Hello from myfs.js');
3  }
4
5  module.exports = {
6    readFileSync,
7  }
```



```
1  const { readFileSync } = require('node:fs');
2  console.log(readFileSync(__filename, 'utf-8'));
```

- Пока экспериментальная функциональность.  
Стадия активной разработки
- Разрешения на основе модулей: контроль доступности для файлов/URL другим модулям во время выполнения
- Разрешения на основе процесса: контроль и ограничения доступа процесса Node.js к ресурсам. Запреты, ограничения.
- Флаги для CLI: --experimental-policy, --experimental-permission, --allow-addons

# Одной строкой

- Экспериментальная поддержка WebSocket. Имплементация по стандарту от WHTWG.  
Смотри флаг `--experimental-websocket`
- Стабильный `fetch` и WebStreams

# Test Runner



```
1 import { strictEqual } from 'node:assert';
2 import { afterEach, beforeEach, describe, it, mock } from 'node:test';
3 import { sum } from './sum.mjs';
4
5 describe('Test: sum()', () => {
6   beforeEach(() => {
7     console.log('Before each test...');
8   });
9
10  afterEach(() => {
11    console.log('After each...')
12  })
13
14  it('Adding numbers', () => {
15    strictEqual(sum(2, 1), 3);
16  });
17
18  it('Mock setTimeout', () => {
19    const fn = mock.fn();
20
21    mock.timers.enable({ apis: ['setTimeout'] });
22    setTimeout(fn, 4000);
23    strictEqual(fn.mock.callCount(), 0);
24
25    mock.timers.tick(2000);
26    strictEqual(fn.mock.callCount(), 0);
27
28    mock.timers.tick(2000);
29    strictEqual(fn.mock.callCount(), 1);
30  })
31});
```

- Встроенный Test Runner. Node 18. Активная разработка, пока экспериментально.
- Встроенная Assertion Library
- Дополнительные функции для организации тестов из `node:test`
- Формирование отчётов по покрытию/тестированию
- Дополнительный флаг для запуска: --test

# Symbol.dispose



```
1 import { open } from 'node:fs/promises';
2
3 async function openFile(path: string) {
4   const handle = await open(path, 'r');
5
6   return {
7     handle,
8     [Symbol.asyncDispose]: async () => {
9       console.log('disposed!');
10      await handle?.close();
11    }
12  }
13 }
14
15 await using file = await openFile('./using.mts');
16 // disposed
17
```

- Новый глобальный символ `Symbol.dispose`
- Чтобы определить ресурс (объект с определённым сроком службы)
- В TypeScript 5.2 завезли новое ключевое слово `using`



ТИНЬКОФФ

# История вторая. Архитектура

# Express || fastify



```
1 const express = require('express');
2 const router = express.Router();
3
4 router.get('/users', (req, res) => {
5   res.send('List of users');
6 });
7
8 router.get('/users/:id', (req, res) => {
9   const userId = req.params.id;
10  res.send(`Details of user ${userId}`);
11 });
12
13 router.post('/users', (req, res) => {
14   res.send('Create a new user');
15 });
16
17 router.put('/use
```



```
18   const userId =
19   res.send(`Upda
20 });
21
22 router.delete('/
23   const userId =
24   res.send(`Dele
25 });
26
27 module.exports =
```

```
1 const express = require('express');
2 const app = express();
3 const port = 3000;
4
5 const routes = require('./routes');
6
7 app.use('/api', routes);
8
9 app.listen(port, () => {
10   console.log(`Server running at http://localhost:${port}`);
11});
```

- Node.js позиционируется для быстрой разработки REST API
- Всевозможные backend for frontend, API Gateway и так далее
- Полно пошаговых статей как начать разработку API

# Многоуровневая архитектура

## Разделяй и властвуй

«N-уровневая архитектура приложения позволяет разработчикам создавать гибкие и повторно-используемые компоненты приложения.

Приложение разделяется на уровни абстракции. Появляется возможность вносить изменения в определённый слой, не затрагивая остальные.

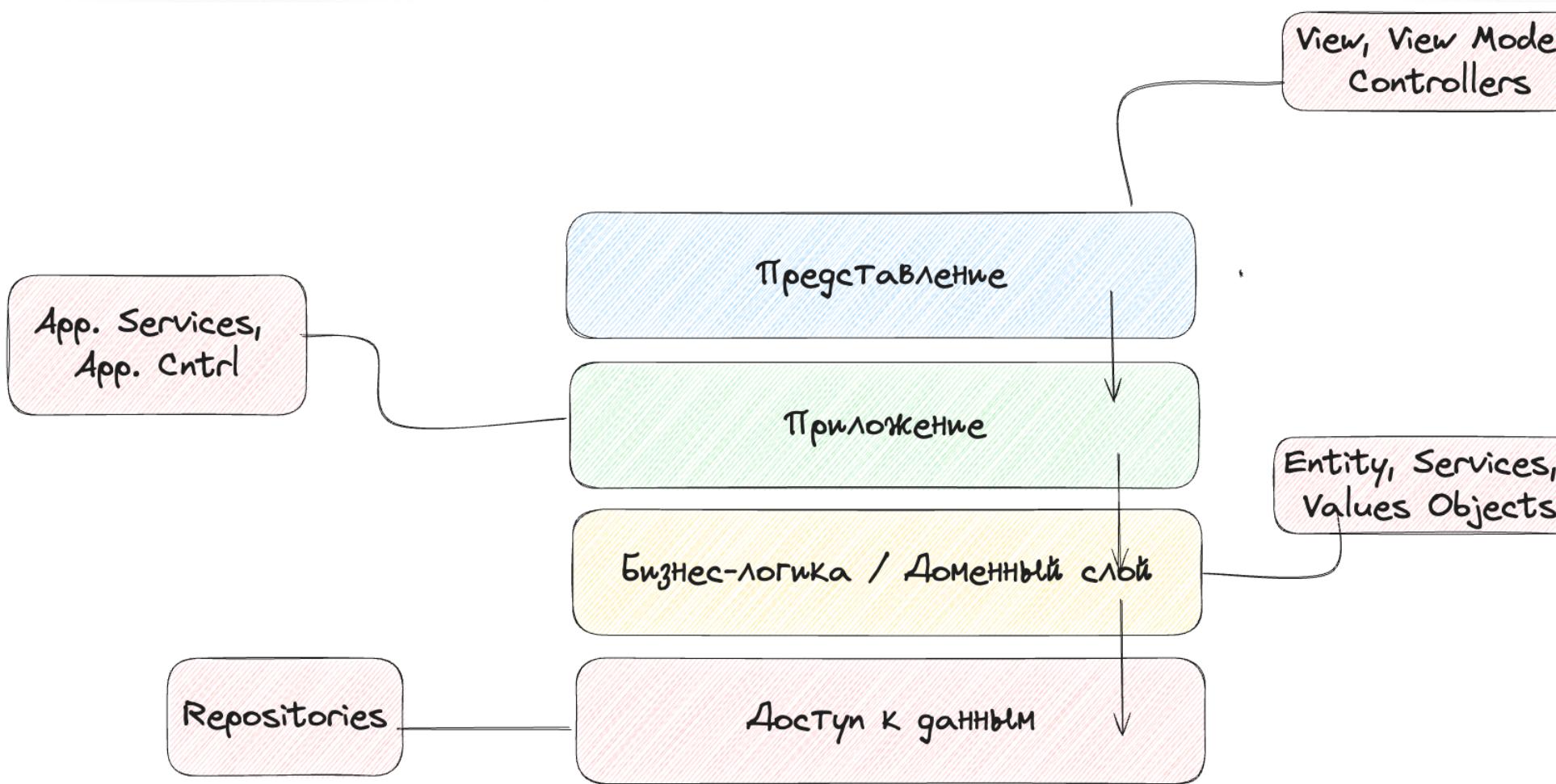
Представление

Приложение

Бизнес-логика

Доступ к данным

# Уровни / слои



- Представление. View, View Models, Controller
- Приложение. Сервисы приложения, контроллеры приложения
- Бизнес-логика / доменный слой. Entity, Services, Value Objects и так далее
- Доступ к данным. Repositories

# Файловая структура



```
1   └── MyAwesomeApplication/
2     ├── apps/
3     |   └── rest.app.ts
4     |   └── cli.app.ts
5     ├── modules/
6     |   ├── user
7     |   ├── category
8     |   ├── offer
9     |   └── comment
10    ├── types
11    ├── libs/
12    |   ├── logger
13    |   ├── config
14    |   └── rest/
15    |       ├── controllers
16    |       ├── errors
17    |       ├── middleware
18    |       └── exception-filters
19    └── static
20      └── main.ts
```

- Приземляем архитектуру с несколькими уровнями на файловую структуру приложения
- Модуль — разделяем приложение на модули
- Каждый модуль самодостаточен: контроллеры, сервисы, репозитории и так далее
- Выделяем слой для переиспользуемых компонентов
- При необходимости прибегаем к дополнительным инструментам (например, Nx)

# Модуль



```
1 .
2   └── MyAwesomeApplication/
3     └── modules/
4       └── user/
5         ├── dto/
6         │   ├── create-user.dto.ts
7         │   ├── login-user.dto.ts
8         │   └── update-user.dto.ts
9         ├── rdo/
10        │   ├── user.rdo.ts
11        │   └── logged-user.rdo.ts
12        ├── user.controller.ts
13        ├── user.entity.ts
14        ├── user.service.ts
15        └── user.constant.ts
```

- DTO, RDO
- Модуль — способ организации кода.  
Инкапсулирует логику
- Функциональность приложения разделяется на  
МОДУЛИ

# **Принципы SOLID**

# Route Interface

- Единый интерфейс для регистрации маршрутов в приложении
- Ограничен определёнными http-методами
- Поддерживает middleware



```
1 export interface Route {  
2   path: string;  
3   method: HttpMethod;  
4   handler: (req: Request, res: Response, next: NextFunction) => void;  
5   middlewares?: Middleware[];  
6 }  
7
```

# Base Controller

```
● ● ●  
1  export abstract class BaseController implements Controller {  
2    private readonly _router: Router;  
3  
4    constructor(  
5      protected readonly logger: Logger  
6    ) {  
7      this._router = Router();  
8    }  
9  
10   get router() {  
11     return this._router;  
12   }  
13  
14   public addRoute(route: Route) {  
15     const wrapperAsyncHandler = asyncHandler(route.handler.bind(this));  
16     const middlewareHandlers = route.middlewares?.map(  
17       (item) => asyncHandler(item.execute.bind(item))  
18     );  
19     const allHandlers = middlewareHandlers ? [...middlewareHandlers, wrapperAsyncHandler] : wrapperAsyncHandler;  
20  
21     this._router[route.method](route.path, allHandlers);  
22     this.logger.info(`Route registered: ${route.method.toUpperCase()} ${route.path}`);  
23   }  
24  
25   public send<T>(res: Response, statusCode: number, data: T): void {  
26     res  
27       .type(DEFAULT_CONTENT_TYPE)  
28       .status(statusCode)  
29       .json(data);  
30   }  
31  
32   public created<T>(res: Response, data: T): void {  
33     this.send(res, StatusCodes.CREATED, data);  
34   }  
35  
36   public noContent<T>(res: Response, data: T): void {  
37     this.send(res, StatusCodes.NO_CONTENT, data);  
38   }  
39  
40   public ok<T>(res: Response, data: T): void {  
41     this.send(res, StatusCodes.OK, data);  
42   }  
43 }
```

- Функциональность контроллеров повторяется
- Абстрактный контроллер
- Обёртка над Express Router (или любой другой)
- Единая точка регистрации middleware

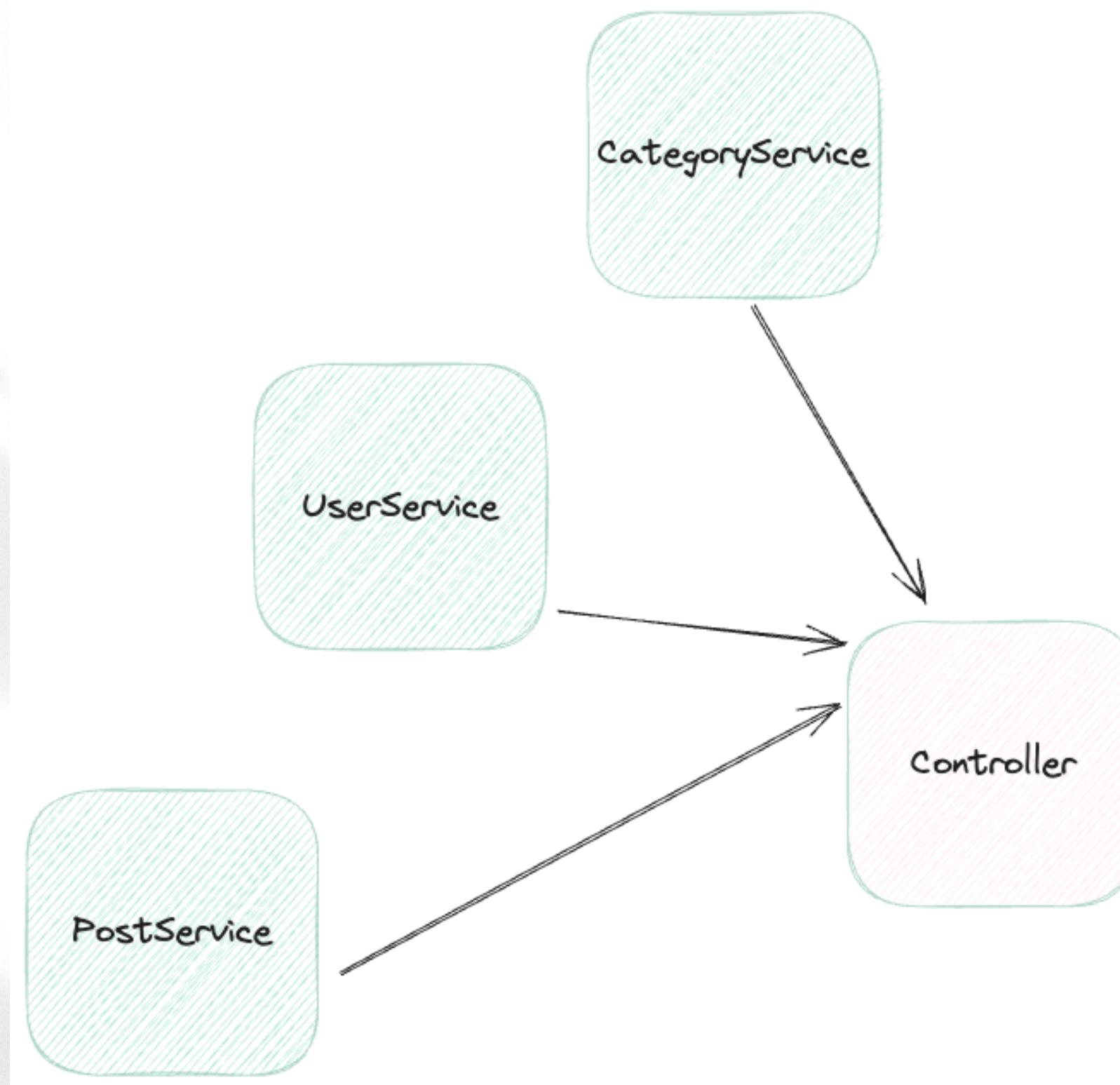
# Category Controller



```
1  export class CategoryController extends BaseController {
2    constructor(
3      protected readonly logger: Logger,
4      private readonly categoryService: CategoryService,
5    ) {
6      super(logger);
7
8      this.logger.info('Register routes for CategoryController...');
9
10     this.addRoute({ path: '/', method: HttpMethod.Get, handler: this.index });
11     this.addRoute({ path: '/', method: HttpMethod.Post, handler: this.create });
12   }
13
14   public async index(_req: Request, res: Response): Promise<void> {
15     const categories = await this.categoryService.find();
16     const responseData = fillDTO(CategoryRdo, categories);
17     this.ok(res, responseData);
18   }
19
20   public async create(
21     { body }: Request<CategoryRequest, CreateCategoryDto>,
22     res: Response
23   ): Promise<void> {
24     const result = await this.categoryService.create(body);
25     this.created(res, fillDTO(CategoryRdo, result));
26   }
27 }
```

- Пример контроллера для ресурса Category
- Используем API базового контроллера

# Зависимости



- Управление зависимостями
- Ресурсы для управления зависимостями  
(создание экземпляров, передача между слоями / компонентами системы и так далее)

# Инверсия управления

- Один из принципов разработки программного обеспечения. Заключается в передаче контроля над выполнением программы к внешней зависимости (библиотеке, фреймворку, контейнеру)
- Проще: внешний механизм отвечает за создание и управление жизненным циклом

# Inversify



```
1 @injectable()
2 export class CategoryController extends BaseController {
3   constructor(
4     @inject(Component.Logger) protected readonly logger: Logger,
5     @inject(Component.CategoryService) private readonly categoryService: CategoryService,
6   ) {
7     super(logger);
8
9     this.logger.info('Register routes for CategoryController...');
10
11    this.addRoute({ path: '/', method: HttpMethod.Get, handler: this.index });
12    this.addRoute({ path: '/', method: HttpMethod.Post, handler: this.create });
13  }
14
15  public async index(_req: Request, res: Response): Promise<void> {
16    const categories = await this.categoryService.find();
17    const responseData = fillDTO(CategoryRdo, categories);
18    this.ok(res, responseData);
19  }
20
21  public async create(
22    { body }: Request<CategoryRequest, CreateCategoryDto>,
23    res: Response
24  ): Promise<void> {
25    const result = await this.categoryService.create(body);
26    this.created(res, fillDTO(CategoryRdo, result));
27  }
28}
29
```



```
1 export function createCategoryContainer() {
2   const categoryContainer = new Container();
3
4   categoryContainer.bind<CategoryService>(Component.CategoryService).to(DefaultCategoryService);
5   categoryContainer.bind<types.ModelType<CategoryEntity>>(Component.CategoryModel).toConstantValue(CategoryModel);
6   categoryContainer.bind<Controller>(Component.CategoryController).to(CategoryController).inSingletonScope();
7
8   return categoryContainer;
9 }
```

- Простая и легковесная библиотека IoC
- Основана на декораторах
- Встроенная поддержка DI

# N-tier

Представление

Приложение

Бизнес-логика

Доступ к данным

- Упрощается поддержка и развитие
- Больше думаем об API компонентов
- Упрощается тестирование



ТИНЬКОФФ

# История третья. Фреймворки

# фреймворки





**Спасибо!**

