Dependency Injection в TypeScript

Владимир Санников

банк Точка

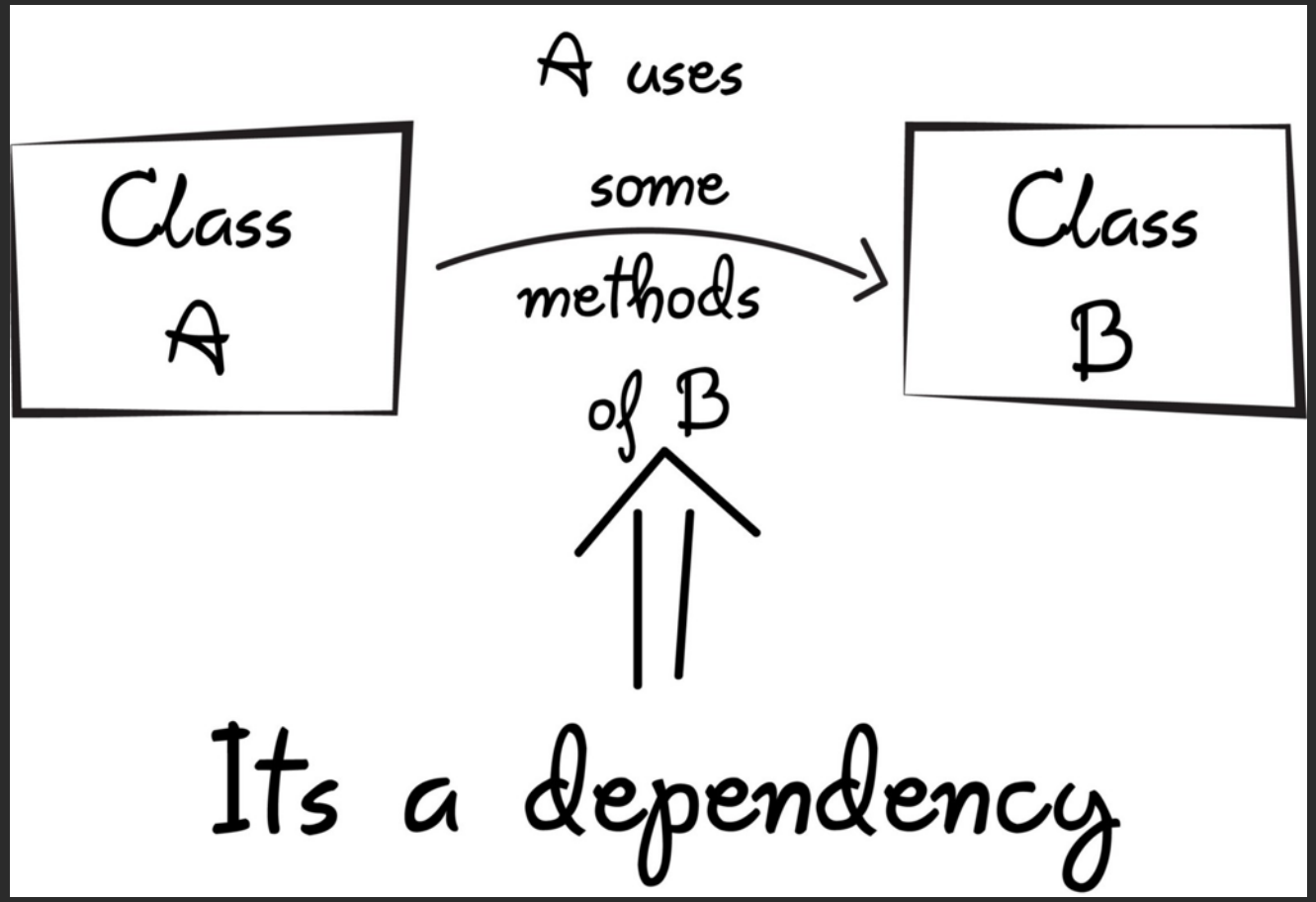
Powered by Quokka

https://quokkajs.com/

Что такое зависимость

- Зависимость - это реализация
- Зависимость при изменении потенциально ломает другой код
- Зависимость повышает coupling

Что такое зависимость



А в чем проблема зависимости?

А в чем проблема зависимости?

Зависимость надо создавать

А в чем проблема зависимости?

Зависимость надо создавать

```
const store = new Store();
const api = new API('http://api.endpoint.com');
store.loadItems(api);
```

DI в двух словах

Переложить ответственность за создание зависимостей на кого-то другого и просто использовать готовые экземпляры

Зачем управлять зависимостями?

- Ожидаемые изменения реализаций под абстракциями
- Изолированное тестирование
- Динамическое изменение абстракций

Зачем управлять зависимостями?

- Ожидаемые изменения реализаций под абстракциями
- Изолированное тестирование
- Динамическое изменение абстракций

- Собрать все это условно в одном месте

Паттерны DI


SO SOFTWARE DESIGN
MUCH OOP
VERY GRASP
SUCH SOLID
WOW И БЕЗЫСХОДНОСТЬ
imgflip.com

Простые паттерны времени создания

- Constructor Injection
- Property/Setter Injection

Constructor Injection

Зависимость передается как аргумент конструктора

Constructor Injection

Зависимость передается как аргумент конструктора

```
class AwesomeService {
  private transportService: ITransportService;

  constructor(transportServiceImpl: ITransportService) {
    this.transportService = transportServiceImpl || new DefaultTransportService();
  }
}
```

Property/Setter Injection

Зависимость устанавливается как свойство экземпляра

Property/Setter Injection

```
class AwesomeService {
  private _transportService: ITransportService;

  get transportService(): ITransportService {
    if (!this._transportService) {
      throw new Error( message: 'No dependency provided');
    }
    return this._transportService;
  }

  set transportService(dependency: ITransportService) {
    this._transportService = dependency;
  }
}
```

Простые паттерны времени использования

Простые паттерны времени использования

- Method Injection
- ServiceLocator

Method Injection

Передаем зависимость конкретному потребителю

Method Injection

Передаем зависимость конкретному потребителю

```
class AwesomeService {
  action(transport?: ITransportService) {
    transport = transport || new DefaultTransportService();
    return transport.send().then(/* process response */);
  }
}
```

ServiceLocator

Предоставляет зависимость по требованию

```
class AwesomeService {
  loadData() {
    const transport = ServiceLocator.resolveDependency(TransportService);
    return transport.send().then(/* process response */);
  }
}
```

ServiceLocator - реализация

```
class ServiceLocator {
  private static store = new Map();

  static registerDependency<T>(
    target: new( ... args: Array<any>) ⇒ T,
    singletonImplementation: T
  ): void {
    ServiceLocator.store.set(target, singletonImplementation);
  }

  static resolveDependency<T>(target: new( ... args: Array<any>) ⇒ T): T {
    return this.store.get(target);
  }
}
```

Hardcore

Добавим немного рефлексии

Рефлексия

Процесс, во время которого программа может отслеживать и модифицировать собственную структуру и поведение во время выполнения.

Википедия

Особенности рефлексии в TS

- Есть декораторы
- Декораторы имеют доступ к типам объектов
- Декораторы модифицируют поля, методы, классы и аргументы

Декоратор класса

Имеет доступ к конструктору класса

```
function ClassDecorator(target: any) {
  console.log(new target());
}
```

Декоратор класса

Имеет доступ к конструктору класса

```
@ClassDecorator
class A {
  foo: number = 1;
}
```

Декоратор класса

Имеет доступ к конструктору класса

```
function ClassDecorator(target: any) {
  console.log(new target());  A { foo: 1 }
}
```

Декоратор свойства

- Имеет доступ к прототипу класса
- Имеет доступ к названию свойства

```
function PropertyDecorator(target, propertyKey) {
  console.log(target, target.constructor, propertyKey);
}
```

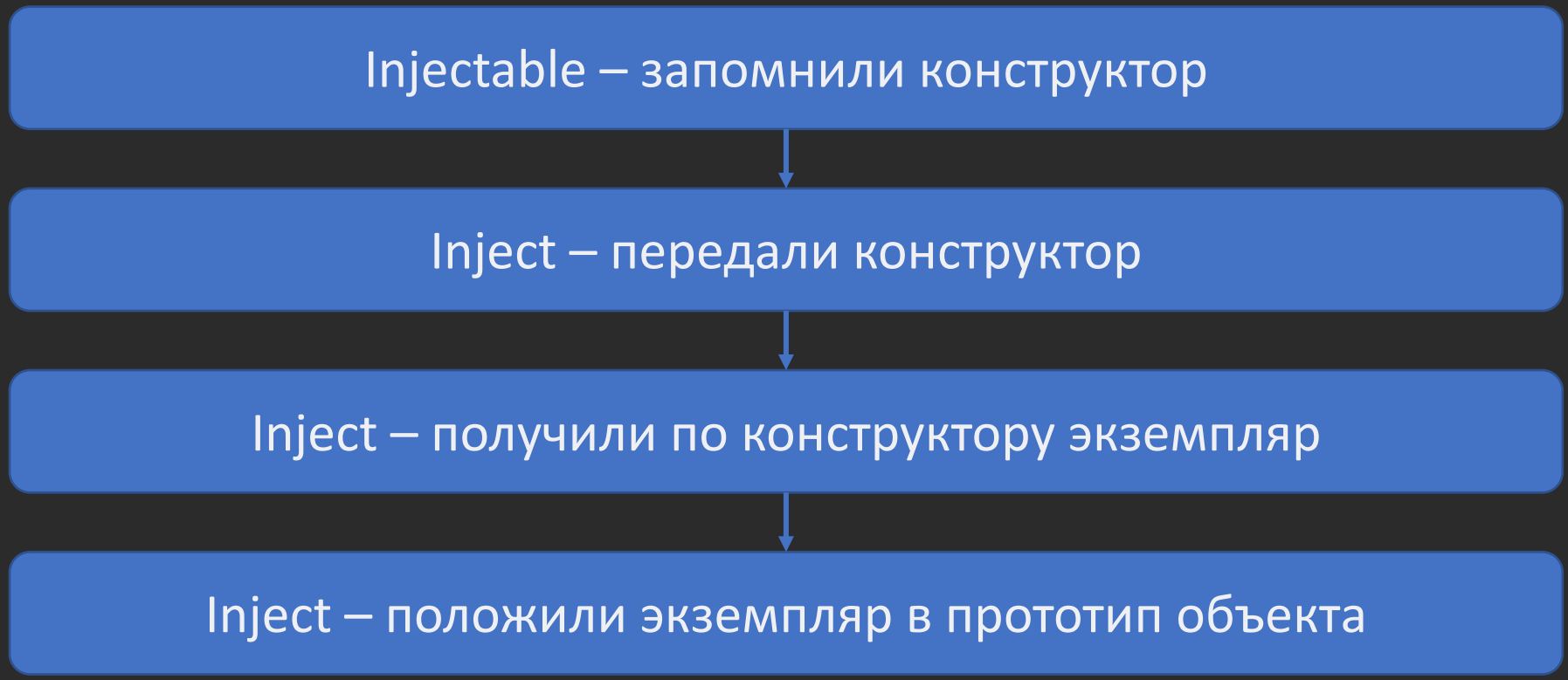
Простенький DI

```
@Injectable
class TransportService {
  send(): Promise<any> {
    return Promise.resolve( pathSegments: Math.random() > 0.5);
  }
}

class AwesomeService {
  @Inject(TransportService) transport: TransportService;

  action(): void {
    this.transport.send().then(v ⇒ console.log(v));
  }
}
```

Простенький DI - концепция



Простенький DI - реализация

```
// Декоратор чтобы пометить класс как внедряемый
export function Injectable(target: any): void {
  ServiceLocator.registerDependency(target, new target());
}
```

Простенький DI - реализация

```
// Декоратор чтобы внедрить экземпляр нужного класса в поле
export function Inject(ServiceConstructor: any): MethodDecorator {
  // tslint:disable-next-line
  return function(target, propertyKey) {
    const descriptor = {
      get(): any {
        return ServiceLocator.resolveDependency(ServiceConstructor);
      }
    };
    Object.defineProperty(target, propertyKey, descriptor);
  };
}
```

Простенький DI – в двух словах

- ServiceLocator – просто место для хранения ключ-значение
- Ключ – нужный тип, значение – нужный экземпляр
- Декоратор Injectable – аннотация чтобы положить в хранилище
- Декоратор Inject – способ достать из хранилища

DI посложнее

Дважды пишем тип зависимости – от этого надо избавляться

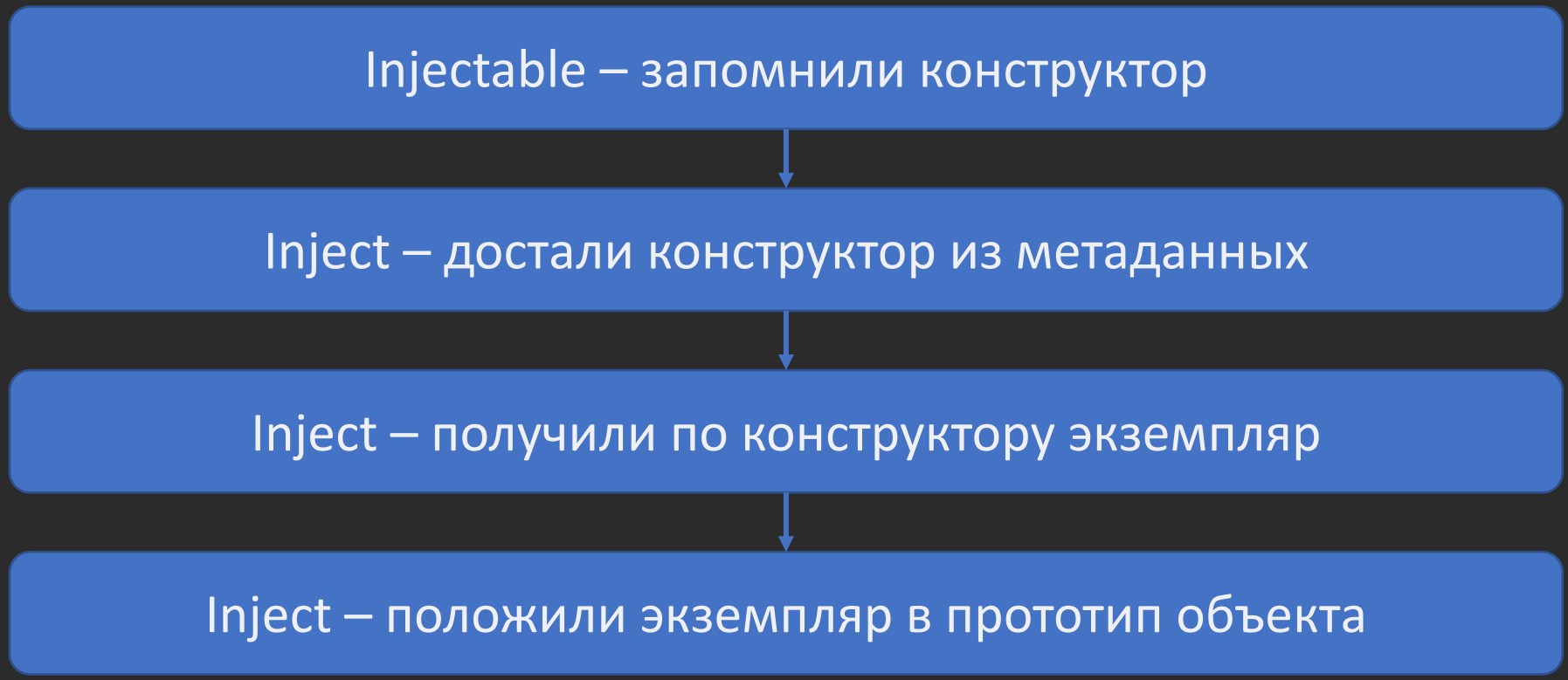
DI посложнее

```
@Injectable
class TransportService {
  send(): Promise<any> {
    return Promise.resolve( pathSegments: Math.random() > 0.5);
  }
}

class AwesomeService {
  @Inject transport: TransportService;

  action(): void {
    this.transport.send().then(v ⇒ console.log(v));
  }
}
```

DI посложнее



Метаданные

Библиотека reflect-metadata позволяет доставать метаданные:

- design:type – тип поля, над которым декоратор
- design:paramtypes – типы аргументов метода/конструктора
- design:returntype – возвращаемый тип метода

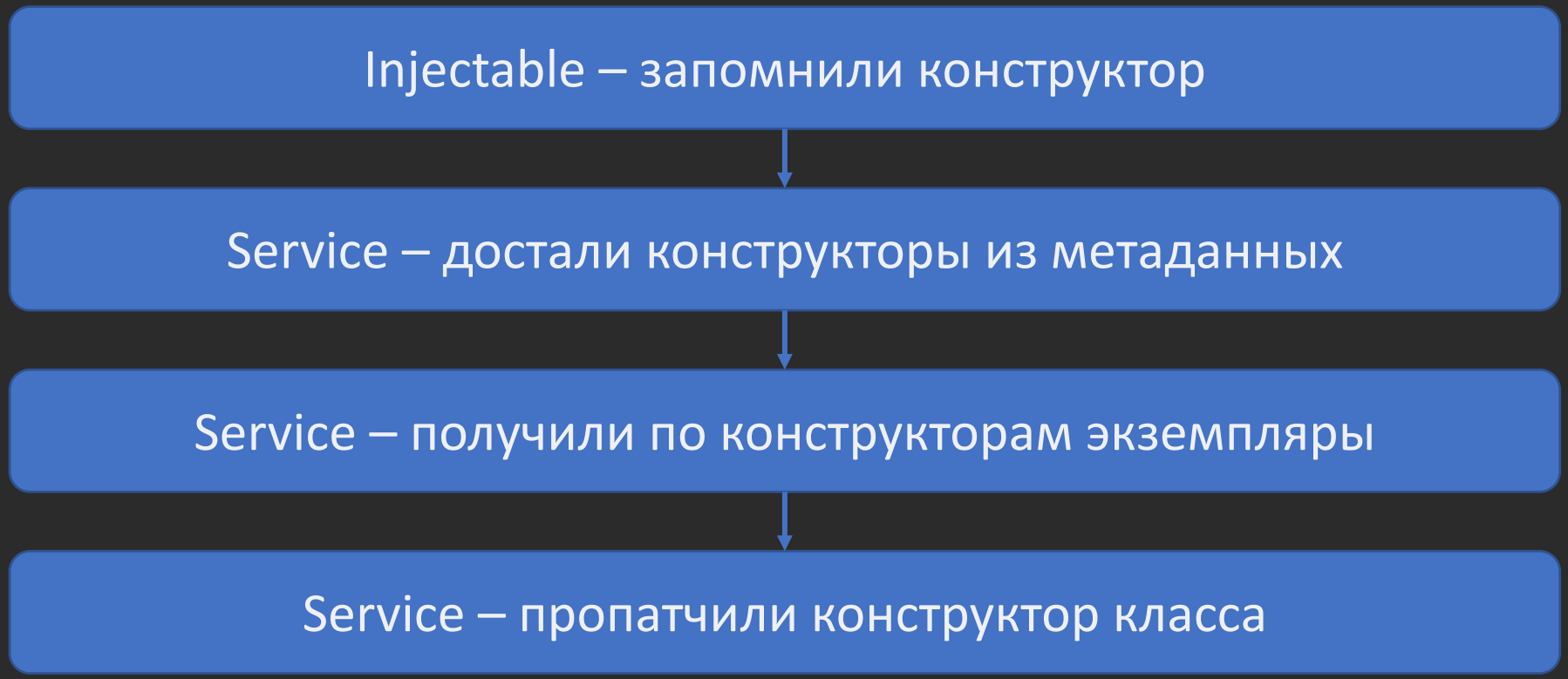
Переносим зависимость в конструктор

```
@Injectable
class TransportService {
  send(): Promise<any> {
    return Promise.resolve( pathSegments: Math.random() > 0.5);
  }
}

@Service
class AwesomeService {
  constructor(private transport?: TransportService) {}

  action(): void {
    this.transport.send().then(v ⇒ console.log(v));
  }
}
```

Зависимость в конструкторе - концепция



Простая перегрузка реализаций

```
@Injectable
class TransportService {
  send(): Promise<any> {
    console.log('Im never execute');
    return Promise.resolve( pathSegments: Math.random() > 0.5);
  }
}

@Override(TransportService)
class TestTransportService {
  send(): Promise<any> {
    console.log('Im always return true');
    return Promise.resolve(true);
  }
}
```

Перегрузка реализаций

- Вместо реализации зависимости в ServiceLocator можно передать функцию, которая будет возвращать экземпляр зависимости
- В этой функции можно реализовать бизнес-логику
- Функция отрывается от концепции декораторов

Перегрузка реализаций

- Вместо реализации зависимости в ServiceLocator можно передать функцию, которая будет возвращать экземпляр зависимости
- В этой функции можно реализовать бизнес-логику
- Функция отрывается от концепции декораторов

- Проще говоря – используем ServiceLocator напрямую

Перегрузка реализаций

А если в функции нужны другие зависимости?

Перегрузка реализаций

А если в функции нужны другие зависимости?

Можно сделать её методом класса, и внедрять зависимости в него

Модуль

Превращаем обычный DI в микрофреймворк

Модуль

```
@Module({
  entry: AwesomeService,
  providers: [
    {
      provide: TransportService,
      useClass: TestTransportService
    }
  ]
})
class AwesomeModule {}
```

Модуль

```
@Module({
  entry: AwesomeService,
  providers: [
    {
      provide: TransportService,
      useExist: TestTransportService
    }
  ]
})
class AwesomeModule {}
```

Модуль

```
@Module({
  entry: AwesomeService,
  providers: [
    {
      provide: TransportService,
      useValue: new TestTransportService()
    }
  ]
})
class AwesomeModule {}
```

Модуль

```
@Module({
  entry: AwesomeService,
  providers: [
    {
      provide: TransportService,
      useFactory: (envService: EnvService) ⇒ {
        return envService.isDevEnv() ? new TestTransportService() : new TransportService();
      },
      deps: [EnvService]
    }
  ]
})
class AwesomeModule {}
```

Спасибо за внимание!

https://github.com/SoEasy/di_presentation

@kbddh

SoEasy