目錄

> Modern, powerful web application framework for Node.js.

Nest是一套现代化的基于Node.js的强大的Web应用框架

> Nest is a powerful web framework for Node.js which helps you effortlessly build efficient, scalable applications. Nest uses modern JavaScript, is built with TypeScript, and combines best concepts of both OOP (Object Oriented Progamming) and FP (Functional Programming).

Nest是一套基于Node.js的强大的Web框架，可帮助你轻松构建出高效的、可扩展的应用程序。它是通过结合OOP（面向对象编程）和FP（函数式编程）的最佳理念，采用现代化JavaScript，使用TypeScript构建的。

> Nest is not just a framework. You don't have to wait for a large community because Nest is built with awesome, popular, well-known libraries—Express and socket.io (you can use any other library if you want to)! It means, that you could quickly start using framework without worrying about a third party plugins.

Nest不仅仅只是一套框架，因为它是基于绝妙的，著名的流行库Express和Socket.io构建的（你也可以根据自己的需求选择任何其他库），所以无需等待大型社区，可以直接使用，无需担心第三方库的缺失。

安装

Git:

```
$ git clone https://github.com/kamilmysliwiec/nest-typescript-starter.git project
$ cd project
$ npm install
$ npm run start
```

NPM:

```
$ npm i --save @nestjs/core @nestjs/common @nestjs/microservices @nestjs/websockets @n
estjs/testing reflect-metadata rxjs
```

理念

> JavaScript is awesome. This language is no longer just trash to create simple animations in the browser. Now, the front end world sports a rich variety of tools. We have a lot of amazing frameworks / libraries such as Angular, React or Vue, which improve our development process and make our applications fast and flexible.

Javascript是一门非常神奇的计算机语言。它不再是一门只能在浏览器上创建简单动画的语言。现在，前端领域已经开发出了多种绝妙的高性能的框架/库，例如Angular、React 和 Vue, 这些工具大大地提高了我们的开发进程，并且使我们的应用程序变得快速而灵活。

> Node.js enabled us to use JavaScript also on the server side. There are a lot of superb libraries, helpers and tools for node, but none of them solves the main problem—the architecture.

通过Node.Js，我们可以在服务器端使用JavaScript。虽然现在有很多基于Node的库、助手和工具，但是没有任何一个可以解决主要问题-结构体系问题。

> We want to create scalable, loosely-coupled, easy-to-maintain applications. Let’s show the entire world the potential of node.js together!

我们希望创建出可扩展的、松散耦合的、易于维护的应用程序。让我们一起来看看Node.js的潜能吧！

特点

> 1. Easy to learn - syntax is similar to Angular
> 2. Built on top of TypeScript, but also compatible with plain ES6 (I strongly recommend to use TypeScript)
> 3. Based on well-known libraries (Express / socket.io) so you could share your experience
> 4. Supremely useful Dependency Injection, built-in Inversion of Control container
> 5. Hierarchical injector - increase abstraction in your application by creating reusable, loosely coupled modules with type injection
> 6. WebSockets module (based on socket.io, although you can use any other library using adapter)
> 7. Own modularity system (split your system into reusable modules)
> 8. Reactive microservices support with messages patterns (built-in transport via TCP / Redis, but you can use any other type of communication using CustomTransportStrategy)
> 9. Exceptions handler layer, exception filters, sync & async pipes layer
> 10. Testing utilities

1. 便于学习-语法结构类似Angular。
2. 基于TypeScript构建，同时兼容普通的ES6（强烈建议使用TypeScript).
3. 基于著名的（Express/Scoket.io)库，所以可以分享经验。
4. 非常有用的依赖注入，内置控制反转容器。
5. 分层注入器器—通过使用类型注入创建可重用、松耦合的模块，从而在应用程序中增加抽象性。
6. WebScokets模块（基于scoket.io，虽然你可以使用任何其他使用适配器的库。
7. 独特的模块化系统（将你的系统分割成克重用的模块）。
8. 消息类型支持的反应微服务（内置transport属性，决定使用TCP或者Redis，但是你可以选择使用任何其他使用CustomTransportStrategy的交流形式）。
9. 异常处理layer，异常过滤器，同步和异步pipes layer。
10. 测试工具

文档 & 快速开始

Documentation & Tutorial
Starter repos

TypeScript
Babel
Useful references

Modules
Examples
API Reference
People

Author - Kamil Myśliwiec
Website - http://nestjs.com
License

核心概念

> The core concept of Nest is to provide an architecture, which helps developers to accomplish maximum separation of layers and increase abstraction in their applications.

Nest的核心概念是提供结构系统，帮助开发只实现最大分层，并提高应用程序的抽象性。

> It has enormous potential, but does not solve problems for you. It is not just a set of prepared classes, with some behavior. It is an idea, which shows you how to organize modern application and enable you to focus on application logic.

它潜力无限，但是无法解决你的问题。它不仅仅是一套事先准备好的有行为的类，它提供了一套组织现代化应用程序的理念，这种理念使你能够专注与应用程序逻辑。

> Application building blocks We have three basic application building blocks in Nest: Modules Controllers Components

应用程序构建块 Nest有三种基本的应用程序构建块

1. 模块。
2. 控制器。
3. 组件。

模块

> Module is a class with @Module({}) decorator. This decorator provides metadata, which framework uses to organize application structure. Simple module with all available properties:

模块是一个带有 `@Module({})` 装饰器的类。该装饰器提供元数据，框架使用该元数据组织应用程序结构。 以下是一个简单模块的所有可用属性：

```
@Module({
    modules [ TestModule ],
    controllers: [ TestController ],
    components: [ TestComponent ],
    exports: [ TestComponent ]
})
export class ApplicationModule {}
```

> By default, modules encapsulate each dependency. It means, that it is not possible to use its components / controllers from another module. To allow different modules to share same instance of component (only components can be exported), we could simply exports it.

默认情况下，模块封装每一个依赖，也就是说模块只能在其内部使用组件/控制器。 我们可以将组件实例导出（只有组件可以被导出），这样模块之间就可以共享组件实例了。

控制器

> The Controllers layer is responsible for handling incoming HTTP requests. Controller is a simple class with @Controller() decorator.

控制层负责处理传入的HTTP请求。控制器是一个带有 `@Controller()` 装饰器的类。

```
@Controller()
class UsersController {
    @Get('users')
    getAllUsers(@Res() response) {
        res.status(201).json({});
    }
 }
```

组件

> Almost everything is a component - Service, Repository, Provider etc. and they might be injected to controllers or to another component by constructor (as in Angular).

几乎所有的事物都可以被看作一个组件-- `Service, Repository, Provider` 等。可以通过构造函数将组件注入到控制器或者另一个组件中。

```
@Component()
class UsersService {
    getAllUsers() {
        return [];
    }
}
```

> Learn more about those building blocks in next sections.

First Chapter

GitBook allows you to organize your book into chapters, each chapter is stored in a separate file like this one.

设置应用程序

> Nest is built with features from both ES6 and ES7 (decorators, async / await). It means, that the easiest way to start adventure with it is to use Babel or TypeScript. In this tutorial I will use TypeScript (it is not required) and I recommend everyone to choose this way too. Sample tsconfig.json file:

Nest 采用 ES6 和 ES7 （decorators， async / await）功能构建。也就是说使用Babel和TypeScript是开始Nest的最简单的方式。在这个教程中，我将使用TypeScript（非必需），我推荐大家也使用TypeScript。以下是一个简单的tsconfig.json文件样本：

```
{
  "compilerOptions": {
    "module": "commonjs",
    "declaration": false,
    "noImplicitAny": false,
    "noLib": false,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "target": "es6"
  },
  "exclude": [
    "node_modules"
  ]
}
```

> Remember that emitDecoratorMetadata and experimentalDecorators properties have to be set to true. So let's start from scratch. Firstly, we have to create entry module of our application:

请记住将 `emitDecoratorMetadata` 和 `experimentalDecorators` 属性设置为 `true` 。 现在，我们开始吧，首先，我们必须先创建好应用程序的整个模块：

```
import { Module } from '@nestjs/common';

@Module({})
export class ApplicationModule {}
```

> At this moment module metadata is empty, because we only want to run application (we don't have any controlles or components right now). Second step - make file (e.g. index.ts) and use NestFactory to create Nest application instance based on our module class.

```
import { NestFactory } from '@nestjs/core';
import { ApplicationModule } from './app.module';

const app = NestFactory.create(ApplicationModule);
app.listen(3000, () => console.log('Application is listening on port 3000'));
```

That's all.

Express实例

> If you want to have a full control of express instance lifecycle, you can simply pass already created object as a second argument of NestFactory.create() method, just like that:

如果你想完全控制express实例的生命周期，你可以将创建好的对象作为第二个参数传递给 `NestFactory.create()` 方法。如下所示：

```
import * as express from 'express';
import { NestFactory } from '@nestjs/core';
import { ApplicationModule } from './modules/app.module';

const instance = express();
const app = NestFactory.create(ApplicationModule, instance);
app.listen(3000, () => console.log('Application is listening on port 3000'));
```

> It means, that you can directly add some custom configuration (e.g. setup some plugins such as morgan or body-parser).

也就是说你可以直接添加自定义配置（比如：设置插件，如 `morgan` 或者 `body-parser` ）

第一个控制器

> The Controllers layer is responsible for handling incoming HTTP requests. In Nest, Controller is a simple class with @Controller() decorator.

控制层负责处理传入的HTTP请求。在Nest中，控制器是一个带有 `@Controller()` 装饰器的类。

> In previuos section we set up entry point for an application. Now, let's build our first endpoint /users:

上一章节中，我们已经设置好了入口点。现在，让我们开始构建我们的第一个路径 `/users` ：

```
import { Controller, Get, Post, HttpStatus } from '@nestjs/common';

@Controller()
export class UsersController {
    @Get('users')
    getAllUsers() {}

    @Get('users/:id')
    getUser() {}

    @Post('users')
    addUser() {}
}
```

> As you can guess, we created an endpoint with 3 different paths:

如你所想，我们已经创建好了三种不同的路径：

```
GET: users
GET: users/:id
POST: users
```

> It is not necessary to repeat 'users' in each path. Nest allows us to pass additional metadata to @Controller() decorator. The path, which is a prefix for each route. Let's rewrite our controller:

不需要在每个路径中都重复 `'users'` 。Nest允许我们向 `@Controller()` 装饰器传递额外的元数据。路径就是每个路由的前缀。让我们重写我们的控制器。

```
import { Controller, Get, Post, HttpStatus } from '@nestjs/common';

@Controller('users')
export class UsersController {
    @Get()
    getAllUsers() {}

    @Get('/:id')
    getUser() {}

    @Post()
    addUser() {}
}
```

路由

> Let's stop for a while. We setup routes, but they are not doing anything. We don't have any persistence, so we also do not have any user. What's now? For explanation purposes, I will use a fake data:

我们暂停一会会。我们设置路由，但是路由并没有起到任何作用。我们没有持久层，也没有用户，那么我们接下来要做什么呢？

```
@Get()
getAllUsers(req, res, next) {
    res.status(HttpStatus.OK).json([{
        id: 1, name: 'Test'
    }]);
}
```

> As you can see, methods in Nest controllers have the same behaviour as a simple routes in Express.

如你所见，Nest控制器的方法和Express的简单路由有着相同的行为。

> If you want to learn more about req (request), res (response) and next you should read short Express Routing - Documentation. In Nest, they work equivalently. Furthermore, you can use @types/express package:

如果你想了解更多关于 `req(request)` ， `res(response)` 和 `next` 的信息，你可以阅读Express Routing - 文档。在Nest中，他们是等价的。此外，你还可以使用 `@types/express` 包。

```
import { Request, Response, NextFunction } from 'express';

@Controller('users')
export class UsersController {
    @Get()
    getAllUsers(req: Request, res: Response, next: NextFunction) {}
}
```

> Moreover, Nest provides a set of custom decorators, which you can use to mark arguments.

Nest还提供了一套可以用来标记参数的自定义装饰器。

Nest	Express
@Request() / @Req()	req
@Response() / @Res()	res
@Next()	next
@Session()	req.session
@Param(param?: string)	req.params[param]
@Body(param?: string)	req.body[param]
@Query(param?: string)	req.query[param]
@Headers(param?: string)	req.headers[param]

> This is how you can use them:

以下是使用自定义装饰器的方法：

```
@Get('/:id')
public async getUser(@Response() res, @Param('id') id) {
    const user = await this.usersService.getUser(id);
    res.status(HttpStatus.OK).json(user);
}
```

> Remember to import decorators at the beginning of a file:

记住在文件开始位置导入装饰器：

```
import { Response, Param } from '@nestjs/common';
Important! If you want to use:
@Session() - install express-session
@Body() - install body-parser
```

Last Step

> UsersController is ready to use, but our module doesn't know about it yet. Let's open ApplicationModule and add some metadata.

现在已经可以使用 `UsersController` 了，但是我们的模块还没意识到 `UsersController` 可以使用。让我们打开 `ApplicationModule` 并添加一些元数据。

```
import { Module } from '@nestjs/common';
import { UsersController } from "./users.controller";

@Module({
    controllers: [ UsersController ]
})
export class ApplicationModule {}
```

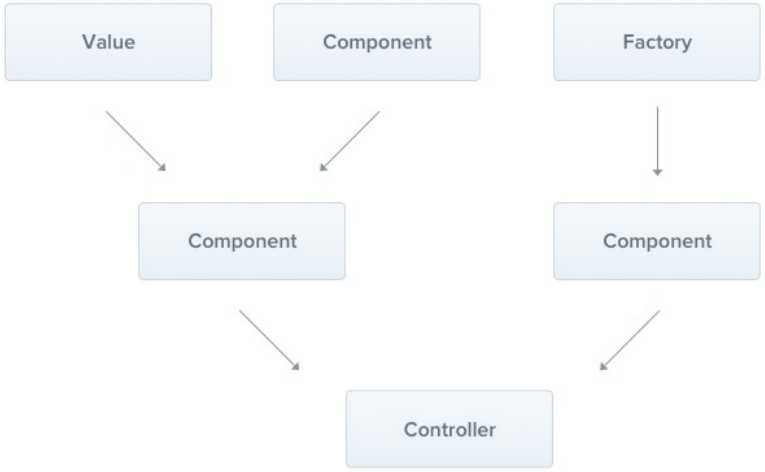
> As you can see - we only have to insert our controller into controllers array. It's everything.

如你所见--我们只需将控制器插入控制器数组中。

组件

> Almost everything is a component - Service, Repository, Provider etc. and they might be injected to controllers or to another component by constructor.

几乎所有的事物都是组件--`Service, Repository, Provider`等。可以通过构造函数将一个组件注入控制器或者另一个组件中。



> In previous section, we built a simple controller - UsersController. This controller has an access to our data (I know, it's a fake data, but it doesn't really matter here). It's not a good solution. Our controllers should only handle HTTP requests and delegate more complex tasks to services. This is why we are going to create UsersService component.

在上一章节，我们构建了一个简单的控制器--`UsersController`。该控制器可以访问我们的数据（我知道，它所能访问的数据是假数据，但是没关系）。但是这并不是一个好的方案，我们的控制器应该只处理HTTP请求并将更复杂的任务委托给服务。所以我们要创建`usersservice`组件。

> In real world, UsersService should call appropriate method from persistence layer e.g. UsersRepository component. We don't have any kind of database, so again - we will use fake data.

实际上，`usersService`应该从持久层，例如`UsersRepository`组件，调用合适的方法。我们没有数据库，所以我们还得继续使用假数据。

```
import { Component } from '@nestjs/common';
import { HttpException } from '@nestjs/core';

@Component()
export class UsersService {
    private users = [
        { id: 1, name: "John Doe" },
        { id: 2, name: "Alice Caeiro" },
        { id: 3, name: "Who Knows" },
    ];
    getAllUsers() {
        return Promise.resolve(this.users);
    }
    getUser(id: number) {
        const user = this.users.find((user) => user.id === id);
        if (!user) {
            throw new HttpException("User not found", 404);
        }
        return Promise.resolve(user);
    }
    addUser(user) {
        this.users.push(user);
        return Promise.resolve();
    }
}
```

> Nest Component is a simple class, with @Component() annotation. As might be seen in getUser() method we used HttpException. It is a Nest built-in Exception, which takes two parameters - error message (or full object) and status code. It is a good practice to create domain exceptions, which should extend HttpException (more about it in "Advanced/Error Handling" section).

Nest组件是一个带有 `@Component()` 注释的简单的类。 在 `getUser()` 中我们使用了 `HttpException` 。 `HttpException` 是Nest内置异常，该异常包含两个参数--错误消息和状态代码。创建局域异常可以扩展 `HttpException` （详见"Advanced/Error Handling"章节）。

> Our service is prepared, let's use it in UsersController from previous article.

我们的服务已经在待命了，让我们在之前创建好的 `UsersController` 中使用它。

```
@Controller('users')
export class UsersController {
    constructor(private usersService: UsersService) {}

    @Get()
    getAllUsers(@Response() res) {
        this.usersService.getAllUsers()
            .then((users) => res.status(HttpStatus.OK).json(users));
    }

    @Get('/:id')
    getUser(@Response() res, @Param('id') id) {
        this.usersService.getUser(+id)
            .then((user) => res.status(HttpStatus.OK).json(user));
    }

    @Post()
    addUser(@Response() res, @Body('user') user) {
        this.usersService.addUser(user)
            .then((msg) => res.status(HttpStatus.CREATED).json(msg));
    }
}
```

> As shown, UsersService will be injected into constructor. It is incredibly easy to manage dependencies with TypeScript, because Nest will recognize your dependencies just by type! So this:constructor(private usersService: UsersService)

通过以上方法可以将`UsersController`注入到构造函数中。 通过TypeScript管理依赖非常方便，因为Nest会根据`type`识别依赖。

> Is everything what you have to do. There is one important thing to know - you must have emitDecoratorMetadata option set to true in your tsconfig.json.

还有一件非常重要的事情--在`tsconfig.json`文件中必须将`emitDecoratorMetadata`选项设置为`true`。

> If you are not TypeScript enthusiast and you work with plain JavaScript, you have to do it in this way:

如果你不是一个TypeScript狂热者，使用纯JavaScript,你得操作以下步骤：

```
import { Dependencies, Controller, Get, Post, Response, Param, Body, HttpStatus } from
'@nestjs/common';

@Controller('users')
@Dependencies(UsersService)
export class UsersController {
    constructor(usersService) {
        this.usersService = usersService;
    }

    @Get()
    getAllUsers(@Response() res) {
        this.usersService.getAllUsers()
            .then((users) => res.status(HttpStatus.OK).json(users));
    }

    @Get('/:id')
    getUser(@Response() res, @Param('id') id) {
        this.usersService.getUser(+id)
            .then((user) => res.status(HttpStatus.OK).json(user));
    }

    @Post()
    addUser(@Response() res, @Body('user') user) {
        this.usersService.addUser(user)
            .then((msg) => res.status(HttpStatus.CREATED).json(msg));
    }
}
```

> Simple, right? In this moment, application will not even start working. Why? Because Nest doesn't know anything about UsersService. This component is not a part of ApplicationModule yet. We have to add it there:

非常简单，对吗？ 现在，应用程序还没有开始运行。

```
import { Module } from '@nestjs/common';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
    controllers: [ UsersController ],
    components: [ UsersService ],
})
export class ApplicationModule {}
```

> That's it! Now, our application will run, but still one of routes doesn't work properly - addUser (POST /users). Why? Because we are trying to extract request body (req.body.user) without body-parser express middleware. As you should already know, it is possible to pass express instance as a second argument of NestFactory.create() method.

完成以上步骤以后，我们的程序就开始运行了，但是仍然有一个路由无法正常运行--- `addUser (POST /users)` 。这是为什么呢？这是因为我们在没有使用 `body-parser` express中间件的情况下尝试解析请求体 `(req.body.user)` 。你应该已经了解到，我们可以将express实体作为第二个参数传递给 `NestFactory.create()` 方法。

> Let's install plugin:

那么接下来我们安装插件吧：

```
$ npm install --save body-parser
Then setup it in our express instance.
import * as express from 'express';
import * as bodyParser from 'body-parser';
import { NestFactory } from '@nestjs/core';
import { ApplicationModule } from './modules/app.module';

const instance = express();
instance.use(bodyParser.json());

const app = NestFactory.create(ApplicationModule, instance);
app.listen(3000, () => console.log('Application is listening on port 3000'));
```

完!

Async / await

> Nest is compatible with async / await feature from ES7, so we can quickly rewrite our UsersController:

Nest兼容ES7async / await功能，所以我们可以快速重写 `UsersController`

```
@Controller('users')
export class UsersController {
    constructor(private usersService: UsersService) {}

    @Get()
    async getAllUsers(@Response() res) {
        const users = await this.usersService.getAllUsers();
        res.status(HttpStatus.OK).json(users);
    }

    @Get('/:id')
    async getUser(@Response() res, @Param('id') id) {
        const user = await this.usersService.getUser(+id);
        res.status(HttpStatus.OK).json(user);
    }

    @Post()
    async addUser(@Response() res, @Body('user') user) {
        const msg = await this.usersService.getUser(user);
        res.status(HttpStatus.CREATED).json(msg);
    }
}
```

> Looks better right? There you can read more about async / await.

现在看着好多了吧？

模块

> Module is a class with @Module({}) decorator. This decorator provides metadata, which framework uses to organize application structure.

模块是一个带有 `@Module({})` 装饰器的类。该装饰器为框架提供组织应用程序结构的元数据。

> Right now, it is our ApplicationModule:

现在，我们来准备 `ApplicationModule`

```
import { Module } from '@nestjs/common';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
    controllers: [ UsersController ],
    components: [ UsersService ],
})
export class ApplicationModule {}
```

> By default, modules encapsulate each dependency. It means, that it is not possible to use its components / controllers outside module. Each module can also import another modules. In fact, you should think about Nest Modules as a tree of modules. Let's move UsersController and UsersService to UsersModule. Simply create new file e.g. users.module.ts with below content:

默认情况下，模块封装每一个依赖。也就是说，只能在模块内部使用模块的组件/控制器。 在每一个模块中都可以导入其他模块。实际上，你可以将Nest模块看作是一颗模块树。 将 `UsersController` 和 `UsersService` 移动至 `UsersModule` ，只需要简单创建一个文件，例如 `users.module.ts` 。以下是文件内容：

```
import { Module } from '@nestjs/common';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
    controllers: [ UsersController ],
    components: [ UsersService ],
})
```

> export class UsersModule {} Then import UsersModule into ApplicationModule (our main application module):

```
import { Module } from '@nestjs/common';
import { UsersModule } from './users/users.module';

@Module({
    modules: [ UsersModule ]
})
export class ApplicationModule {}
```

> It's everything. As might be seen, with Nest you can naturally split your code into separated and reusable modules!

接下来，你就可以使用Nest将你的代码拆分成一个个可重复使用的模块了。

分享实例

> You already know that Module encapsulates its components. What if you want to share instance between two or more modules? With Nest - it is pretty easy. You only have to use@Shared() decorator and add exports array, just like that:

我们都知道每个模块封装它们的组件。那么要想在模块之间分享组件该怎么办呢？有了Nest，你只需要使用 `@Shared()` 装饰器并且添加输出数组，就可以轻松分享组件了。以下是一个示例：

```
import { Module, Shared } from '@nestjs/common';

@Shared()
@Module({
    controllers: [ UsersController ],
    components: [ UsersService ],
    exports: [ UsersService ],
})
export class UsersModule {}
```

> That's all. It is especially powerful feature. You can read more about it in Advanced / SharedModule section.

这是一个很强大的功能。更多关于分享组件的信息请阅读Advanced / SharedModule章节。

依赖注入

> Module can easily inject components, which belongs to itself:

每个模块可以轻松注入它自己的组件：

```
@Module({
    controllers: [ UsersController ],
    components: [ UsersService, ChatGateway ],
})
export class UsersModule implements NestModule {
    constructor(private usersService: UsersService) {}
}
```

> Furthermore, components also can inject modules:

此外，组件也可以注入模块：

```
export class UsersController {
    constructor(private module: UsersModule) {}
}
```

中间件

> Middleware is a function, which is called before route handler. Middleware functions have access to request and response objects, so they can modify them. They can also be something like a barrier - if middleware function does not call next(), the request will never be handled by route handler.

中间件是在路由处理器工作之前被调用的函数。中间件函数可以访问 `request` 和 `response` 对象，所以中间件函数也就可以修改 `request` 和 `response` 对象。中间件也可以被看作是挡在中间的屏障--如果中间件没有调用 `next()` ，路由处理器也就没法处理 `request` 。

> The simplest example:

请看一个简单的例子：

```
import { Middleware, NestMiddleware } from '@nestjs/common';

@Middleware()
export class LoggingMiddleware implements NestMiddleware {
    resolve(): (req, res, next) => void {
        return (req, res, next) => {
            console.log('Request...');
            next();
        }
    }
}
```

> Let's build a dummy authorization middleware (for explanation purposes - just by username). We will use X-Access-Token HTTP header to provide username (weird idea, but it doesn't matter here).

让我们构建一个虚拟授权中间件（用于解释目的——只需通过用户名）。我们将用 `x-access-token HTTP header` 提供用户名（奇怪的想法，但是这不重要）。

```
import { HttpException } from '@nestjs/core';
import { Middleware, NestMiddleware } from '@nestjs/common';
import { UsersService } from './users.service';

@Middleware()
export class AuthMiddleware implements NestMiddleware {
    constructor(private usersService: UsersService) {}

    resolve(): (req, res, next) => void {
        return async (req, res, next) => {
            const userName = req.headers["x-access-token"];
            const users = await this.usersService.getAllUsers();

            const user = users.find((user) => user.name === userName);
            if (!user) {
                throw new HttpException('User not found.', 401);
            }
            req.user = user;
            next();
        }
    }
}
```

> Some facts about middlewares:
>
> - you should use @Middleware() annotation to tell Nest, that this class is a middleware,
> - you can use NestMiddleware interface, which forces on you to implement resolve() method,
> - middlewares (same as components) can inject dependencies through their constructor (dependencies have to be a part of module),
> - middlewares must have resolve() method, which must return another function (higher order function). Why? Because there is a lot of third-party, ready to use Express Middlewares (and more), which you could simply - thanks to this solution - use in Nest.
>
> Okey, we already have prepared middleware, but we are not using it anywhere. Let's set it up:

关于中间件的一些实际情况：

- 使用 `@Middleware()` 注释告诉Nest，这个类是一个中间件。
- 可以使用 `NestMiddleware` 接口，这会迫使你执行 `resolve()` 方法。
- 中间件（如组件一样）可以通过构造函数注入依赖（依赖必须是模块的一部分）。
- 中间件必须有 `resolve()` 方法，该方法必须返回另一个函数（高阶函数）。这是为什么呢？因为这样的话，你可以轻松使用将要使用Express中间件的第三方插件。 中间件准备

就绪，先设置好，待用。

```
import { Module, MiddlewaresConsumer } from '@nestjs/common';

@Module({
    controllers: [ UsersController ],
    components: [ UsersService ],
    exports: [ UsersService ],
})
export class UsersModule {
    configure(consumer: MiddlewaresConsumer) {
        consumer.apply(AuthMiddleware).forRoutes(UsersController);
    }
}
```

> As shown, Modules can have additional method - configure(). This method receives as a parameter MiddlewaresConsumer, an object, which helps us to configure middlewares.

如上所示，模块还有其他方法-`configure()`。该方法接收`MiddlewaresConsumer`对象作为一个参数，该对象可以帮助我们设置中间件。

> This object has apply() method, which receives infinite count of middlewares (method uses spread operator, so it is possible to pass multiple classes separated by comma). apply() method returns object, which has two methods:
>
> - forRoutes() - we use this method to pass infinite count of Controllers or objects (with path and method properties) separated by comma,
> - with() - we use this method to pass custom arguments to resolve() method of middleware.

该对象有`apply()`方法，它可以接收无数个中间件（该方法使用扩展运算符，所以可以传递无数个用逗号隔开的类）。`apply()`方法可以通过以下两种方式返回对象：

- `forRoutes()`使用此方法传递无数个用逗号隔开的控制器或对象（有路径和方法属性）。
- `with()`使用此方法将自定义参数传递给中间件的`apply()`方法。

运行原理

When you pass UsersController in forRoutes method, Nest will setup middleware for each route in controller:

使用`forRoutes`方法传递`UsersController`时，Nest会为控制器中的每个路由设置中间件。

```
GET: users
GET: users/:id
POST: users
```

> But it is also possible to directly define for which path middleware should be used, just like that:

但是我们也可以通过以下方式直接指定中间件将要使用的路由。

```
consumer.apply(AuthMiddleware).forRoutes({
    path: '*', method: RequestMethod.ALL
});
```

传递参数（给中间件）

> Sometimes the behaviour of the middleware depends on custom values - e.g. array of users roles. We can pass additional arguments to middleware using with() method.

有时中间件的行为取决于自定义值-如用户角色数组。我们可以通过with()方法向中间件传递附加参数。

示例:

```
const roles = ['admin', 'user'];
const options = {};
consumer.apply(AuthMiddleware)
    .with(roles, options)
    .forRoutes(UsersController);
@Middleware()
export class AuthMiddleware implements NestMiddleware
    resolve(roles: string[], options: object) {
        return async (req, res, next) => {
            console.log(roles); // ['admin', 'user'] in this case
            next();
        }
    }
}
```

链

> You can simply chain apply() calls:

你可以简单地调用 apply()链：

```
consumer.apply(AuthMiddleware, PassportMidleware)
       .forRoutes(UsersController, OrdersController, ClientController);
       .apply(...)
       .forRoutes(...);
```

调用顺序

> Middlewares are called in the same order as they are placed in array. Middlewares configured in sub-module are called after the parent module configuration.

中间件是放在数组中的，所以他们被同时调用。`sub-module`（子模块）中配置的的中间件会在父模块配置的中间件之后被调用。。

Pipes

> Pipes are really useful feature. You should think about them as a streams of data. They're called immediately before route handlers.

Pipes是非常有用的功能。可以将pipes看作是数据流。在路由控制器处理完程序之后立即调用pipes。

Pipes是什么？

> Pipe is a simple class, which is decorated by @Pipe() and implements PipeTransform interface.

Pipe是一个简单的带有 `@Pipe` 装饰器的类，它可以实现 `PipeTransform` 界面。

```
import { PipeTransform, Pipe, ArgumentMetadata } from '@nestjs/common';

@Pipe()
export class CustomPipe implements PipeTransform {
    public transform(value, metadata: ArgumentMetadata) {
        return value;
    }
}
```

> It has a one method, which receives 2 arguments:
>
> - value (any)
> - metadata (ArgumentMetadata) metadata is the metadata of an argument for which the pipe is being processed, and the value is... just its value. The metadata holds few properties:

它有一个方法，该方法接收两个参数

- `value (any)`
- `metadata (ArgumentMetadata)`

  `metadata` 指的是pipe正在处理的参数的元数据，元数据有以下属性：

```
export interface ArgumentMetadata {
    type: 'body' | 'query' | 'param';
    metatype?: any;
    data?: any;
}
```

> I'll tell you about them later.

稍后再详细讲解。

运行原理

> Let's imagine we have a route handler:

让我们想象一下我们有如下路由处理程序：

```
@Post()
public async addUser(@Res() res: Response, @Body('user') user: UserDto) {
    const msg = await this.usersService.addUser(user);
    res.status(HttpStatus.CREATED).json(msg);
}
```

> There is a request body parameter user. The type is UserDto:

有一个请求体参数用户。属性为 `UserDto`

```
export class UserDto {
    public readonly name: string;
    public readonly age: number;
}
```

> This object always has to be correct, so we have to validate those two fields. We could do it in the route handler method, but we will break the single responsibility rule. The second idea is to create validator class and delegate the task there, but we will have to call this validator at the beginning of the method every time. So maybe should we create a validation middleware? It's good idea, but it's almost impossible to create a generic middleware, which might by used along entire application. This is the first use-case, when you should consider to use a Pipe.

该对象必须校正，所以我们需要验证这两个字段。我们可以在路由处理程序方法中进行验证，但是这违反了单一职责原则。

第二种方法是我们可以创建一个验证类，由这个验证类去执行验证，但是这样的话，我们每次在执行方法的时候都必须调用该验证器。

那么我们是不是可以创建一个验证中间件呢？但是要创建一个在整个应用程序中通用的中间件几乎是不可能的。

所以这个时候我们就可以考虑使用 `Pipe` 了。

ValidatorPipe with Joi

> There is an amazing library, which name is Joi. It's commonly used with hapi. Let's create a schema:

有一个超级好用的库，叫做 `Joi` ，它通常跟 `hapi` 搭配使用。我们来创建一个模式：

```
const userSchema = Joi.object().keys({
    name: Joi.string().alphanum().min(3).max(30).required(),
    age: Joi.number().min(1).required(),
}).required();
And the appropriate Pipe:
@Pipe()
export class JoiValidatorPipe implements PipeTransform {
    constructor(private readonly schema: Joi.ObjectSchema) {}

    public transform(value, metadata: ArgumentMetadata) {
        const { error } = Joi.validate(value, this.schema);
        if (error) {
            throw new HttpException('Validation failed', HttpStatus.BAD_REQUEST);
        }
        return value;
    }
}
```

> Notice! If you want to use your domain exception instead of built-in HttpException, you have to set-up Exception Filter. Now, we only have to bind JoiValidatorPipe to our method.

注意！如果你想使用局域异常来代替 `built-in HttpException` ，必须设置异常过滤器。 现在，我们只需要将将 `JoiValidatorPipe` 与我们的方法相结合，就可以进行验证了。

```
@Post()
@UsePipes(new JoiValidatorPipe(userSchema))
public async addUser(@Res() res: Response, @Body('user') user: UserDto) {
    const msg = await this.usersService.addUser(user);
    res.status(HttpStatus.CREATED).json(msg);
}
```

> The transform() function will be evaluated for each @Body(), @Param() and @Query() argument of the route handler. If you want to run validation only for @Body() arguments - use metadata properties. Let's create pipe with predicate:

路由处理程序的每个 `@Body(), @Param() and @Query()` 参数，都要进行 `transform()` 功能评估。如果你只需要针对 `@Body()` 参数执行验证，那么你可以使用元数据属性。让我们使用谓词创建Pipe：

```
@Pipe()
export class JoiValidatorPipe implements PipeTransform {
    constructor(
        private readonly schema: Joi.ObjectSchema,
        private readonly toValidate = (metadata: ArgumentMetadata) => true) {}

    public transform(value, metadata: ArgumentMetadata) {
        if (!this.toValidate(metadata)) {
            return value;
        }
        const { error } = Joi.validate(value, this.schema);
        if (error) {
            throw new HttpException('Validation failed', HttpStatus.BAD_REQUEST);
        }
        return value;
    }
}
```

> And the usage example:

看一个有用的示例：

```
@Post()
@UsePipes(new JoiValidatorPipe(userSchema, ({ type }) => type === 'body'))
public async addUser(@Res() res: Response, @Body('user') user: UserDto) {
    const msg = await this.usersService.addUser(user);
    res.status(HttpStatus.CREATED).json(msg);
}
Also, you can directly bind pipe to chosen argument:
@Post()
public async addUser(
    @Res() res: Response,
    @Body('user', new JoiValidatorPipe(userSchema)) user: UserDto) {

    const msg = await this.usersService.addUser(user);
    res.status(HttpStatus.CREATED).json(msg);
}
```

That's all.

> Joi is a great library, but this solution is not generic. We always have to create a schema, and the pipe instance. Can we make it better? Sure, we can.

Joi是一个强大的库。但是这个解决办法并不能解决通用问题。我们总是要创建一个模式和pipe实例。

ValidatorPipe with class-validator

> There is an amazing library, which name is class-validator (great library @pleerock!). It allows to use decorator-based validation. Let's create a generic validation pipe:

还有一个名为 `class-validator` 的强大的库。使用这个库可帮助我们进行基于装饰器的验证。

```
import { validate } from 'class-validator';

@Pipe()
export class ValidatorPipe implements PipeTransform {
    public async transform(value, metadata: ArgumentMetadata) {
        const { metatype } = metadata;
        if (!this.toValidate(metatype)) {
            return value;
        }
        const object = Object.assign(new metatype(), value);
        const errors = await validate(object);
        if (errors.length > 0) {
            throw new HttpException('Validation failed', HttpStatus.BAD_REQUEST);
        }
        return value;
    }

    private toValidate(metatype = null): boolean {
        const types = [String, Boolean, Number, Array, Object];
        return !types.find((type) => metatype === type);
    }
}
```

> IMPORTANT! It works only with TypeScript.

注意！只用该库时只能使用TypeScript。

> As you can see, the metatype property of ArgumentMetadata holds type of the value. Furthermore, the pipes can by asynchronous. When we would use ValidatorPipe there:

`ArgumentMetadata` 的 `metatype` 属性包含值类型。此外，Pipes可进行异步操作。我们可以这样使用 `ValidatorPipe` ：

```
@Post()
@UsePipes(new ValidatorPipe())
public async addUser(@Res() res: Response, @Body('user') user: UserDto) {
    const msg = await this.usersService.addUser(user);
    res.status(HttpStatus.CREATED).json(msg);
}
```

> The metatype will be UserDto. That's it. We have a generic solution now. Notice! It doesn't work with TypeScript interfaces - you have to use classes instead.

metatype为UserDto。 就是如此，我们现在找到了一个方案可以解决通用性问题。 注意！它无法在TypeScript接口中运行 - 所以必须使用类代替。

作用域

> You already know that pipes can be argument-scoped and method-scoped. It's not everything! We can set-up pipe for each route handler in the Controller (controller-scoped pipes):

你已经了解到pipes可以是 `argument-scoped` 和 `method-scoped` 。不仅仅如此！我们也可以在控制器(controller-scoped pipes)中为每个路由处理程序设置pipe。

```
@Controller('users')
@UsePipes(new ValidatorPipe())
export class UsersController {}
```

> Moreover, you can set-up global pipe for each route handler in the Nest application!

此外，你还可以在Nest应用程序中为每个路由处理程序设置 `global pipe`

```
const app = NestFactory.create(ApplicationModule);
app.useGlobalPipes(new ValidatorPipe());
```

> This is how you can e.g. enable request properties auto-validation.

网关

> There are special components in Nest called Gateways. Gateways help us to create real-time web apps. They are some kind of encapsulated socket.io features adjusted to framework architecture.

Nest中有一个特殊的组件叫网关。他可以帮着我们创建实时的web应用程序。网关是适用于框架结构的封装的 `scoket.io` 功能。

```
import { WebSocketGateway } from '@nestjs/websockets';

@WebSocketGateway()
export class UsersGateway {}
```

> By default - server runs on port 80 and with default namespace. We can easily change those settings:

默认情况下--服务器使用默认命名空间在80端口上运行。我们可以很容易地更改这些设置：

```
@WebSocketGateway({ port: 81, namespace: 'users' })
```

> Of course - server will run only if UsersGateway is listed in module components array, so we have place it there:

当然--只有 `UsersGateway` 在模块组件数组中时，服务器才能运行，所以我们需要按照如下的方式将 `UsersGateway` 放置在模块组件数组中。

```
@Module({
    controllers: [ UsersController ],
    components: [ UsersService, UsersGateway ],
    exports: [ UsersService ],
})
```

> There are three useful events of Gateway:
>
> - afterInit, which gets as an argument native server socket.io object,
> - handleConnection and handleDisconnect, which gets as an argument native client socket.io object.

网关有三种有用的事件：

- `afterInit`，获取本地服务器scoket.io对象座位参数。
- `handleConnection` 和 `handleDisconnect`，获取本地客户端socket.io对象作为参数。

> There are special interfaces, which helps to manage lifecycle hooks:
>
> - `OnGatewayInit`
> - `OnGatewayConnection`
> - `OnGatewayDisconnect`

有如下三个接口可以帮助我们管理生命周期：

- `OnGatewayInit`
- `OnGatewayConnection`
- `OnGatewayDisconnect`

消息

> In Gateway, we can simply subscribe to emitted messages:

在网关中我们可以轻松订阅发出的消息：

```
import { WebSocketGateway, SubscribeMessage } from '@nestjs/websockets';

@WebSocketGateway({ port: 81, namespace: 'users' })
export class UsersGateway {
    @SubscribeMessage('drop')
    handleDropMessage(sender, data) {
        // sender is a native socket.io client object
    }
}
```

> And from client side:

从客户端接收如下：

```
import * as io from 'socket.io-client';
const socket = io('http://URL:PORT/');
socket.emit('drop', { msg: 'test' });
```

@WebSocketServer()

> If you want to assign to chosen property socket.io native server instance, you could simply decorate it with @WebSocketServer() decorator.

如果要分配选定的 socket.io 本地服务器实例属性，你可以使用 @WebSocketServer() 装饰器来简单地对属性进行装饰。

```
import { WebSocketGateway, WebSocketServer, SubscribeMessage } from '@nestjs/websocket
s';

@WebSocketGateway({ port: 81, namespace: 'users' })
export class UsersGateway {
    @WebSocketServer()
    private server: object;

    @SubscribeMessage('drop')
    handleDropMessage(sender, data) {
        // sender is a native socket.io client object
    }
}
```

> Value will be assigned after server initialization.

服务器初始化完成后将分配值。

依赖注入

> Gateway is a Component, so it can inject dependencies through constructor. Gateway also can be injected into another component.

网关是一个组件，所以可以通过构造函数注入依赖。网关也可以被注入到另一个组件中。

网关中间件

> Gateway middlewares works almost same as route middlewares. Middleware is a function, which is called before gateway message subscriber. Gateways middleware functions have access to native socket object. They can be something like a barrier - if middleware function does not call next(), the message will never be handled by subscriber.

网关中间件和路由中间件的工作原理几乎一致。中间件是一个函数，该函数在网关消息被订户处理之前调用。网关中间件函数可以访问本地 `scoket` 对象。所以它可以被看作是消息和订户中间的屏障---如果中间件函数不调用 `next()` 方法，消息将不会送达到订户那里。

> Example:

```
@Middleware()
export class AuthMiddleware implements GatewayMiddleware {
    public resolve(): (socket, next) => void {
        return (socket, next) => {
            console.log('Authorization...');
            next();
        };
    }
}
```

> Some facts about gateway middlewares:
>
> - you should use @Middleware() annotation to tell Nest, that this class is a middleware,
> - you can use GatewayMiddleware interface, which forces on you to implement resolve() method,
> - middlewares (same as components) can inject dependencies through their constructor (dependencies have to be a part of module),
> - middlewares must have resolve() method, which must return another function (higher order function).

网关中间件的实际情况：

- 使用 `@Middleware()` 注释告诉Nest，这个类是一个中间件。
- 可以使用 `NestMiddleware` 界面，这会迫使你执行 `resolve()` 方法。
- 中间件（如组件一样）可以通过构造函数注入依赖（依赖必须是模块的一部分）。
- 中间件必须有 `resolve()` 方法，该方法必须返回另一个函数（高阶函数）。

> Okey, we already have prepared middleware, but we are not using it anywhere. Let's set it up:

好了，中间件已经准备就绪，但是还没有投入到使用中，不过我们还是按照如下方式先设置好中间件待用：

```
@WebSocketGateway({
    port: 2000,
    middlewares: [ ChatMiddleware ],
})
export class ChatGateway {}
```

> As shown, @WebSocketGateway() accepts additional metadata property - middlewares, which is an array of middlewares. Those middlewares will be called before message handlers.

如上所示，`@WebSocketGateway()` 接受额外的元数据属性--`middlewares`，`middlewares` 是一个中间件数组。这个中间件数组在消息处理程序之前被调用。

适配器

> In some cases you might not want to use socket.io. It's not a problem. Nest allows you to use any other websockets library, you only have to create an adapter. Let's imagine that you want to use ws. We have to create a class, which implements WebSocketAdapter (@nestjs/common) interface:

在某些情况下你可能不想使用 `socket.io` 。没问题，Nest允许你使用任何其他的 `websockets` 库，使用其他库时，你只需要创建一个适配器就可以了。

我们来以 `ws` 作为一个例子。 我们需要创建一个可以实现 `websocketadapter` （@ nestjs /普通）接口的类：

```
export interface WebSocketAdapter {
    create(port: number);
    createWithNamespace?(port: number, namespace: string);
    bindClientConnect(server, callback: (...args) => void);
    bindClientDisconnect?(client, callback: (...args) => void);
    bindMessageHandlers(client, handlers: MessageMappingProperties[]);
    bindMiddleware?(server, middleware: (socket, next) => void);
}
```

> Three methods are obligatory - create, bindClientConnect and bindMessageHandlers. The rest are optional. Take a look at the example:

`create` , `bindClientConnect` 和 `bindMessageHandlers` 这三种方法是必须的，其他都是可选的。请看如下示例：

```
import * as WebSocket from 'ws';
import { WebSocketAdapter } from '@nestjs/common';
import { MessageMappingProperties } from '@nestjs/websockets';

class WsAdapter implements WebSocketAdapter {
    public create(port: number) {
        return new WebSocket.Server({ port });
    }
    public bindClientConnect(server, callback: (...args: any[]) => void) {
        server.on('connection', callback);
    }
    public bindMessageHandlers(client, handlers: MessageMappingProperties[]) {
        client.on('message', (buffer) => {
            const data = JSON.parse(buffer);
            const { type } = data;
            const messageHandler = handlers.find((handler) => handler.message === type
);
            messageHandler && messageHandler.callback(data);
        });
    }
}
```

> The most interesting is bindMessageHandlers function, where we have to bind messages to appropriate handlers. Handler is an object, with message property (passed in @SubscribeMessage() decorator) and callback (function to execute). I decided to recognize messages by type property, but it's your decision how you want to map them. The last step - we must set-up our adapter:

最有趣的是 `bindmessagehandlers` 功能，在该功能中我们必须将消息和对应的处理程序结合起来。 `Handler` 是一个有消息属性（ `@SubscribeMessage()` 装饰器中传递）和回调函数（执行函数）的对象。我决定通过 `type` 属性识别消息，你可以选择使用其他映射方式。 最后一步--我们必须设置我们的适配器：

```
const app = NestFactory.create(ApplicationModule);
app.useWebSocketAdapter(new WsAdapter());
```

That's it!

Basics

> It is unbelievably simple to transform Nest application into Nest microservice. Take a look - this is how you create web application:

将Nest应用程序转换为Nest微服务非常简单。 让我们来看看如何创建web应用程序。

```
const app = NestFactory.create(ApplicationModule);
app.listen(3000, () => console.log('Application is listening on port 3000'));
```

> Now, switch it to a microservice:

现在，将应用程序转换为微服务：

```
const app = NestFactory.createMicroservice(ApplicationModule, { port: 3000 });
app.listen(() => console.log('Microservice is listening on port 3000'));
```

> It's everything!

就是如此！

TCP通信

> By default Nest microservice is listening for messages via TCP protocol. It means that right now e.g. @Get() will not be useful, because it is mapping HTTP requests. So, how microservice will recognize messages? Just by patterns. What is pattern? It is nothing special. It could be an object, string or even number (but it is not a good idea). Let's create MathController:

默认情况下， Nest微服务是通过 `TCP协议` 监听消息的。也就是说现在@Get()将用武之地，因为它映射 `HTTP` 请求。所以微服务是怎么识别消息的呢？--通过模式识别。

模式是什么呢？模式不是什么特殊的东西，它可以是对象、字符串或者数字（这不是一个好想法）。

我们来创建一个 `MathController` ：

```
import { MessagePattern } from '@nestjs/microservices';

@Controller()
export class MathController {
    @MessagePattern({ cmd: 'add' })
    public add(data: number[]): Observable<number> {
        const numbers = data || [];
        return Observable.of(numbers.reduce((a, b) => a + b));
    }
}
```

> As you might seen - if you want to create message handler, you have to decorate it wih @MessagePattern(pattern). In this example, I chose { cmd: 'add' } as a pattern. The handler method receives single argument - data, which is a variable with data sent from another microservice (or just web application). Also, the handler returns Observable from Rxjs package, so it's possible to return multiple values. IMPORTANT! You have to complete Observable if you want to terminate the data stream.

如上所示，如果你想创建一个消息 `handler` ，你必须用 `@MessagePattern(模式)` 来装饰这个 `handler` 。在上面的例子中，我是用的是 `{ cmd: 'add' }` 模式。

`handler` 接受由另一个微服务（或者web应用程序）发送的单个参数--数据变量。

同时， `handler` 从 `Rxjs package` 返回监控属性，所以可能会返回多个值。注意！如果想种植数据流必须完成监控属性。

客户端

> You already know how to listen for messages. Now, let's check how to send them from another microservice or web application. Before you can start, Nest has to know, where you're exactly going to send messages. It's easy - you only have to create @Client object.

你已经了解了如何监听消息。现在我们来看看如何从另一个微服务或者web应用程序发送消息。 在开始之前，你必须告诉Nest你要将消息发送到哪里。很简单，你只需要创建一个 `@Client` 对象即可。

```
import { Controller } from '@nestjs/common';
import { Client, ClientProxy, Transport } from '@nestjs/microservices';

@Controller()
export class ClientController {
    @Client({ transport: Transport.TCP, port: 5667 })
    client: ClientProxy;
}
```

> @Client() decorator receives object as a parameter. This object can have 3 properties:
>
> - transport - with this you can decide which method you're going to use - TCP or Redis (TCP by default),
> - url - only for Redis purposes (default - redis://localhost:6379),
> - port (default 3000).

@Client装饰器接收对象作为参数。该对象有以下三个属性：

- `transport` - 这个属性帮助你决定通信方式--- `TCP` 还是 `Redis` （默认使用 `TCP`).
- `url` - 仅用于Redis（默认-- `redis://localhost:6379` ）。
- `port` （默认为 `3000` ）。

使用客户端

> Let's create custom endpoint to test our communication.

让我们来创建端点来测试通信。

```
import { Controller, Get } from '@nestjs/common';
import { Client, ClientProxy, Transport } from '@nestjs/microservices';

@Controller()
export class ClientController {
    @Client({ transport: Transport.TCP, port: 5667 })
    client: ClientProxy;

    @Get('client')
    public sendMessage(@Res() res: Response) {
        const pattern = { cmd: 'add' };
        const data = [ 1, 2, 3, 4, 5 ];

        this.client.send(pattern, data)
            .catch((err) => Observable.empty())
            .subscribe((result) => res.status(200).json({ result }));
    }
}
```

> As you might seen, in order to send message you have to use send method, which receives message pattern and data as an arguments. This method returns an Observable. It is very important feature, because reactive Observables provide set of amazing operators to deal with, e.g. combine, zip, retryWhen, timeout and more... Of course, if you want to use Promises instead of Observables, you could simply use toPromise() method. That's all.

如上所示，要发送消息就必须使用 `send` 方法，该方法接收消息模式和数据作为参数，并返回一个监控属性。

这是一个非常重要的功能，因为反应监控属性提供了一组强大的操作来处理，例如 `combine`，`zip`，`retryWhen`，`timeout` 等等。

当然，如果你想使用 `Promises` 代替 `Observables`，只需使用 `toPromise()` 属性即可。

> Now, when someone will make /test request (GET), that's how response should looks like (if both microservice and web app are available):

现在，当有人发送 `/test request`（GET）时，`response` 将会显示如下（如果微服务和 web 应用都可用）：

```
{
  "result": 15
}
```

Redis

> There is another way to work with Nest microservices. Instead of direct TCP communication, we could use amazing Redis feature - publish / subscribe.

还有另外一种与Nest 微服务的通信方式。我们可以使用强大的Redis功能 `publish` / `subscribe` 代替直接TCP通信。

> Of course before you can use it, it is obligatory to install Redis.

当然，必须安装Redis之后才能使用。

创建微服务

> To create Redis Microservice, you have to pass additional configuration in NestFactory.createMicroservice() method.

要创建Redis微服务就必须在 `NestFactory.createMicroservice()` 方法中传递其他配置。

```
const app = NestFactory.createMicroservice(
    MicroserviceModule,
    {
        transport: Transport.REDIS,
        url: 'redis://localhost:6379'
    }
);
app.listen(() => console.log('Microservice listen on port:', 5667 ));
```

> And that's all. Now your microservice will subscribe to messages published via Redis. The rest works same - patterns, error handling, etc.

好了，现在你创建的微服务可以订阅通过Redis发布的消息了。其他的步骤与TCP相同--模式，异常处理等。

客户端

> Now, let's see how to create client. Previously, your client instance configuration looks like that:

现在，让我们来创建客户端。在使用TCP时，你的客户端实例配置如下：

```
@Client({ transport: Transport.TCP, port: 5667 })
client: ClientProxy;
```

> We want to use Redis instead of TCP, so we have to change those settings:

我们使用Redis代替TCP，所以我们应该更改这些设置：

```
@Client({ transport: Transport.REDIS, url: 'redis://localhost:6379' })
client: ClientProxy;
```

> Easy, right? That's all. Other functionalities works same as in TCP communication.

是不是很简单？其他功能与TCP通信相同。

自定义传输

> Nest provides TCP and Redis as a built-in transport methods. It makes prototyping incredibly fast & easy, but sometimes you might want to use another type of transport, e.g. RabbitMQ messaging. Is it possible? Yes, sure.

Nest提供 `TCP` 和 `Redis` 作为内置传输方法。这些方法使得原型化特别容易，特别迅速。但是，有时候你可能想使用其他类型的传输类型，比如 `RabbitMQ messaging` 。

> You can port any transport strategy to Nest. You only have to create a class, which extends Server and implements CustomTransportStrategy interface.

你只需要创建一个类，就可以将任何传输方式应用到Nest中。因为这个类可以扩展服务器并实现 `CustomTransportStrategy` 接口。

> The Server class provides getHandlers() method, which returns MessagePattern mappings (object, where key is a pattern and value is a callback), while CustomTransportStrategy forces on you to implement both listen() and close() methods.

该服务器类提供 `getHandlers()` 方法，该方法返回 `MessagePattern` 映射（对象，该对象中，key是模式，值是回调函数），`CustomTransportStrategy` 迫使你实现 `listen()` and `close()` 方法。

> Let's create a simple RabbitMQServer class. We will use ampqlib library.

让我们使用 `amprlib` 库创建一个简单的 `RabbitMQServer` 类。

```
import * as amqp from 'amqplib';
import { Server, CustomTransportStrategy } from '@nestjs/microservices';
import { Observable } from 'rxjs/Observable';

export class RabbitMQServer extends Server implements CustomTransportStrategy {
    private server = null;
    private channel = null;

    constructor(
        private readonly host: string,
        private readonly queue: string) {
            super();
        }

    public async listen(callback: () => void) {
        await this.init();
        this.channel.consume(`${this.queue}_sub`, this.handleMessage.bind(this), { noA
ck: true });
    }
```

```
    public close() {
        this.channel && this.channel.close();
        this.server && this.server.close();
    }

    private handleMessage(message) {
        const { content } = message;
        const msg = JSON.parse(content.toString());

        const handlers = this.getHandlers();
        const pattern = JSON.stringify(msg.pattern);
        if (!this.messageHandlers[pattern]) {
            return;
        }

        const handler = this.messageHandlers[pattern];
        const response$ = handler(msg.data) as Observable<any>;
        response$ && this.send(response$, (data) => this.sendMessage(data));
    }

    private sendMessage(message) {
        this.channel.sendToQueue(`${this.queue}_pub`, Buffer.from(JSON.stringify(messa
ge)));
    }

    private async init() {
        this.server = await amqp.connect(this.host);
        this.channel = await this.server.createChannel();
        this.channel.assertQueue(`${this.queue}_sub`, { durable: false });
        this.channel.assertQueue(`${this.queue}_pub`, { durable: false });
    }
}
```

> The most interesting method is handleMessage(). Its resposibility is to match pattern with appropriate handler and call it with received data. Also, notice that I used send() method inherited from Server class. You should use it too if you want to avoid e.g. sending disposed message when Observable is completed.

最有趣的方法是 `handleMessage()` 。该方法负责用合适的 `handler` 匹配模式，并且用接收的数据调用该模式。请注意，我使用的是从服务器类继承的 `send()` 方法。你也应该使用该方法避免 `Observable` 完成后发送设置信息。

> Last step is to set-up our RabbitMQ strategy:

最后一个步骤是设置我们的 `RabbitMQ` 方法：

```
const app = NestFactory.createMicroservice(ApplicationModule, {
    strategy: new RabbitMQServer('amqp://localhost', 'example'),
});
```

> It's everything!

客户端

> The RabbitMQ server is listening for messages. Now, we must create a client class, which should extends built-in ClientProxy. We only have to override abstract sendSingleMessage() method.

`RabbitMQ`服务器监听消息。现在，我们必须创建一个可以扩展内置`ClientProxy`的客户端类。

> Let's create RabbitMQClient class:

让我们来创建一个`RabbitMQClient`类。

```
import * as amqp from 'amqplib';
import { ClientProxy } from '@nestjs/microservices';

export class RabbitMQClient extends ClientProxy {
    constructor(
        private readonly host: string,
        private readonly queue: string) {
            super();
        }

    protected async sendSingleMessage(msg, callback: (err, result, disposed?: boolean)
 => void) {
        const server = await amqp.connect(this.host);
        const channel = await server.createChannel();
        const sub = this.getSubscriberQueue();
        const pub = this.getPublisherQueue();

        channel.assertQueue(sub, { durable: false });
        channel.assertQueue(pub, { durable: false });

        channel.consume(pub, (message) => this.handleMessage(message, server, callback
), { noAck: true });
        channel.sendToQueue(sub, Buffer.from(JSON.stringify(msg)));
    }

    private handleMessage(message, server, callback: (err, result, disposed?: boolean)
 => void) {
        const { content } = message;
        const { err, response, disposed } = JSON.parse(content.toString());
        if (disposed) {
            server.close();
        }
        callback(err, response, disposed);
    }

    private getPublisherQueue(): string {
        return `${this.queue}_pub`;
    }

    private getSubscriberQueue(): string {
        return `${this.queue}_sub`;
    }
}
```

> How to use it? There is nothing special, just create an instance:

该怎么使用它呢？只需要创建一个实例即可。

```
export class ClientController {
    private readonly client = new RabbitMQClient('amqp://localhost', 'example');
}
```

> The rest work equivalently (use send() method).

剩余的跟(`use send()` 方法)相同。

测试

> Nest gives you a set of test utilities, which boost application testing process. There are two different approaches to test your components and controllers - isolated tests or with dedicated Nest test utilities.

Nest为你提供了一些列的测试工具，这些测试工具可以强化应用程序测试过程。有两种测试组件控制器的方式----隔离测试和专用嵌套测试。

分离测试

> Both Nest controllers and components are a simple JavaScript classes. It means, that you could easily create them by yourself:

Nest控制器和组件都是JavaScript的简单的类。也就是说，你可以轻松创建组件和控制器：

```
const component = new SimpleComponent();
```

> If your class has any dependency, you could use test doubles, for example from such libraries as Jasmine or Sinon.

Nest测试工具

> The another way to test your applications building block is to use dedicated Nest Test Utilities.

测试应用程序构建模块的另一个方法是使用专有Nest测试工具。

> Those Test Utilities are placed in static Test class (@nestjs/testing module).

这些测试工具放置在静态测试类中（ `@nestjs/testing module` ）

```
import { Test } from '@nestjs/testing';
```

> This class provide two methods:
>
> - createTestingModule(metadata: ModuleMetadata), which receives as an parameter simple module metadata (the same as Module() class). This method creates a Test Module (the same as in real Nest Application) and stores it in memory.
> - get(metatype: Metatype), which returns instance of chosen (metatype passed as parameter) controller / component (or null if it is not a part of module).

该测试类提供两种方法：

- `createTestingModule(metadata:ModuleMetadata)`，该方法接收简单模块元数据作为一个参数（与`Module()`类相同）该方法创建一个测试模块（与实际嵌套Nest应用程序相同），并将该模块存储在内存中。
- `get<T>(metatype: Metatype<T>)`，该方法返回选定的控制器（传递`metatype`作为参数）/组件的实例（如果该实例不是模块的一部分，则返回`null`）

> Example:

示例

```
Test.createTestingModule({
    controllers: [ UsersController ],
    components: [ UsersService ]
});
const usersService = Test.get<UsersService>(UsersService);
```

Mocks, spies, stubs

> Sometimes you might not want to use a real instance of component / controller. Instead of this - you can use test doubles or custom values / objects.

有时候你可能不行使用实际组件/控制机实例，你可以使用测试替身或者（`test doubles`）自定义值/对象

```
const fakeService = {};
Test.createTestingModule({
    controllers: [ UsersController ],
    components: [
        { provide: UsersService, useValue: fakeService }
    ]
});
const usersService = Test.get<UsersService>(UsersService); // mockService
```

依赖注入

> Dependency Injection is a strong mechanism, which helps us easily manage dependencies of our classes. It is very popular pattern in strongly typed languages like C# and Java.

依赖注入是一个很强大的机制，该机制可以帮助我们轻松管理各个类的依赖。它在类型非常强大的语言如 `C#` 和 `Java` 中是非常流行的模式。

> In Node.js it is not such important to use those kind of features, because we already have amazing module loading system and e.g. sharing instance between files is effortless.

在Node.js中，这些功能不是很重要。因为我们已经有了很强大的模块加载系统。比如说，我们可以很轻易地文件之间分享实例。

> The module loading system is sufficient for small and medium size applications. When amount of code grows, it is harder and harder to smoothly organize dependencies between layers. It is also less intuitive than DI by constructor.

对于小中型应用程序来说，模块加载系统已经完全够用了。当代码量增加时，组织各个层之间的依赖会变得越来越困难。它还不如构造函数的DI直观。

> This is the reason, why Nest has its own DI system.

所以Nest有一套自己的DI系统。

自定义组件

> You have already learnt, that it is incredibly easy to add component to chosen module:

你应该已经了解到向选定模块添加组件是非常简单的：

```
@Module({
    controllers: [ UsersController ],
    components: [ UsersService ]
})
```

> But there is some other scenarios, which Nest allows you to take advantages of.

但是Nest允许你使用其他场景。

When:

> - you want to use specific value. Now, in this module Nest will associate value with UsersService metatype,
> - you want to use test doubles (unit testing).

当

- 你想使用具体的值时。在这个模块中，Nest会将值与 `UsersService` 元数据相结合。
- 你想使用测试替身时（单元测试）。

使用值:

```
const  value = {};
@Module({
    controllers: [ UsersController ],
    components: [
        { provide: UsersService, useValue: value }
    ],
})
```

When:

> you want to use chosen, more specific class only in this module.

当

- 你想仅在此模块箱使用更精确的选定的类时

使用类:

```
@Component()
class CustomUsersService {}

@Module({
    controllers: [ UsersController ],
    components: [
        { provide: UsersService, useClass: CustomUsersService }
    ],
})
```

When:

> - you want to provide a value, which has to be calculated using other components (or custom packages features),
> - you want to provide async value (just return Observable or Promise), e.g. database connection.

当

- 你想提供一个必须使用其他组件（或其他自定义包功能）进行计算的值时。
- 你想提供异步值（只返回 `Observable` 或者 `Promise` ），例如数据库连接。

使用 **factory:**

```
@Module({
    controllers: [ UsersController ],
    components: [
        ChatService,
        {
            provide: UsersService,
            useFactory: (chatService) => {
                return Observable.of('customValue');
            },
            inject: [ ChatService ]
        }
    ],
})
```

记住:

> - if you want to use components from module, you have to pass them in inject array. Nest will pass instances as a arguments of factory in the same order.

如果你想使用模块内的组件，必须在 `inject` 数组中传递组件。Nest会以相同的顺序传递实例作为 `factory` 的参数。

When:

> - you want to provide component with a chosen key.

当

- 你想提供有选定 `key` 的组件时。

自定义 providers

```
@Module({
    controllers: [ UsersController ],
    components: [
        { provide: 'isProductionMode', useValue: false }
    ],
})
```

Remember:

记住

> - it is possible to use each types useValue, useClass and useFactory. How to use? To inject custom provided component with chosen key, you have to tell Nest about it, just like that:

可以使用每一个类型的 `useValue` ， `useClass` 和 `useFactory` . 如何使用呢？注入带有选定 `key` 的自定义提供的组件，并以如下方式告知Nest:

```
import { Inject } from '@nestjs/common';

@Component()
class SampleComponent {
    constructor(@Inject('isProductionMode') isProductionMode: boolean) {
        console.log(isProductionMode);
    }
}
```

异常过滤器

> With Nest you can move exception handling logic to special classes called Exception Filters.

你可以使用Nest将异常处理逻辑移动到异常过滤器的特殊类中。

How it works?

怎么运行呢？

> Let's take a look at the following code:

让我们来看看下面的代码：

```
@Get('/:id')
public async getUser(@Response() res, @Param('id') id) {
    const user = await this.usersService.getUser(id);
    res.status(HttpStatus.OK).json(user);
}
```

> Imagine that usersService.getUser(id) method could throws UserNotFoundException. What's now? We have to catch an exception in route handler:

想象以下 `usersService.getUser(id)` 方法可能抛出一个异常-- `UserNotFoundException` 。那么怎么办呢？我们应该在路由处理器中捕捉该异常。

```
@Get('/:id')
public async getUser(@Response() res, @Param('id') id) {
    try {
        const user = await this.usersService.getUser(id);
        res.status(HttpStatus.OK).json(user);
    }
    catch(exception) {
        res.status(HttpStatus.NOT_FOUND).send();
    }
}
```

> To sum up, we have to add try...catch blocks to each route handler, where an exception may occur. Is there another way? Yes - Exception Filters. Let's create NotFoundExceptionFilter:

总的来说，我们应该为每个可能出现异常路由处理程序添加 `try...catch` 快。还有其他方式么？有--我们可以使用异常过滤器。 让我们来创建一个过滤器-- `NotFoundExceptionFilter` 。

```
import { Catch, ExceptionFilter, HttpStatus } from '@nestjs/common';

export class UserNotFoundException {}
export class OrderNotFoundException {}

@Catch(UserNotFoundException, OrderNotFoundException)
export class NotFoundExceptionFilter implements ExceptionFilter {
    public catch(exception, response) {
        response.status(HttpStatus.NOT_FOUND).send();
    }
}
```

> Now, we only have to tell our method to use this filter:

现在我们应该通知我们的方法使用该过滤器：

```
@Get('/:id')
@UseFilters(new CustomExceptionFilter())
public async getUser(@Res() res: Response, @Param('id') id: string) {
    const user = await this.usersService.getUser(id);
    res.status(HttpStatus.OK).json(user);
}
```

> So if usersService.getUser(id) throws UserNotFoundException, NotFoundExceptionFilter will catch it.

所以如果 `usersService.getUser(id)` 抛出 `UserNotFoundException` 时，可以使用 `NotFoundExceptionFilter` 捕捉 `UserNotFoundException` 。

Scope

> The exception filters can be method-scoped, controller-scoped and global-scoped. There are no contraindications to set-up exception filter to each route handler in the Controller:

范围

异常过滤器可以时 `method-scoped` , `controller-scoped` 和 `global-scoped` 的，所以在控制器中为每个路由处理程序设置异常处理过滤器时是没有任何限制的：

```
@UseFilters(new CustomExceptionFilter())
export class UsersController {}
```

> Or even to set-up it globally:

甚至可以直接将它设置为 `globally` 。

```
const app = NestFactory.create(ApplicationModule);
app.useGlobalFilters(new CustomExceptionFilter());
```

ModuleRef

> Sometimes you might want to directly get component instance from module reference.
> It not a big thing with Nest - just inject ModuleRef in your class:

有时候你可能想直接从模块引用中获取组件实例。这对于Nest来说很容易操作，你只需要在你的类中注入 `ModuleRef` 即可：

```
import { ModuleRef } from '@nestjs/core';
import { Controller } from '@nestjs/common';

@Controller()
export class UsersController {
    constructor(
        private usersService: UsersService,
        private moduleRef: ModuleRef) {}
}
```

> ModuleRef provides one method:
>
> - get(key), which returns instance for equivalent token.

`ModuleRef` 提供一个方法：

- `get<T>(key)` ，该方法返回一个等价 `token` 的实例：

> Example:

示例

```
moduleRef.get<UsersService>(UsersService)
```

> It returns instance of UsersService component from current module.

返回一个当前模块中的 `UsersService` 组件。

共享模块

> Nest Modules can export their components. It means that we can easily share component instance between them. The best way to share an instance between two or more modules is to create **Shared Module**.

Nest模块可以导出其组件。也就是说我们可以轻松在模块之间共享组件。 模块间共享组件的最好方式是创建共享模块。

> For example - if we want to share ChatGateway component across entire application, we could do it in this way:

例如---如果我们想在整个应用程序中共享 `ChatGateway` 组件，我们可以按照如下方式进行：

```
import { Module, Shared } from '@nestjs/common';

@Shared()
@Module({
    components: [ ChatGateway ],
    exports: [ ChatGateway ]
})
export class SharedModule {}
```

> Then, we only have to import this module into another modules, which should share component instance:

接下来，我们只需要将这个要分享组件实例的模块导入到其他模块中即可：

```
@Module({
    modules: [ SharedModule ]
})
export class FeatureModule {}
```

第三方模块

> If you want to share 3rd-party module, you can use Shared as a function:

如果你想共享第三方模块，你可以将 `Shared` 看作一个函数来使用：

```
@Module({
    modules: [ Shared()(ThirdPartyModule) ]
})
export class FeatureModule {}
```

范围

> If you don't want to share components across entire application, but only within smaller, defined scope, you can use a **namespace-scoped** @Shared('namespace') module.

如果你不想在整个应用程序中共享组件，只想在比较小的指定的范围内分享组件的话，你可以使用 `namespace-scoped*``@Shared('namespace')` 模块。

```
@Module({
    modules: [ Shared('namespace')(ThirdPartyModule) ]
})
export class FeatureModule {}
```

Http异常

> Notice: It is mainly for REST applications. Nest has error handling layer, which catches all unhandled exceptions. If - somewhere - in your application, you will throw an Exception, which is not HttpException (or inherited one), Nest will handle it and return to user below json object (500 status code):

注意：主要对REST应用程序适用。 Nest有错误处理层，该层可以捕获所有未被处理的异常。 当你的应用程序的某个地方抛出异常，如果该异常不是 `HttpException` 异常（或继承异常），Nest将会处理该异常并将该异常以下面的 `json` 对象(`500 status code`)形式返回给用户：

```
{
  "message": "Unkown exception"
}
```

异常结构

> In your application, you should create your own Exceptions Hierarchy. All 'HTTP exceptions' should inherit from built-in HttpException. For example, you can create NotFoundException and UserNotFoundException classes:

你应该在你的应用程序中创建你自己的异常结构。所有的HTTP异常都应该继承自内置的 `HttpException` 。

```
import { HttpException } from '@nestjs/core';

export class NotFoundException extends HttpException {
    constructor(msg: string | object) {
        super(msg, 404);
    }
}

export class UserNotFoundException extends NotFoundException {
    constructor() {
        super('User not found.');
    }
}
```

> Then - if you somewhere in your application throw UserNotFoundException, Nest will response to user with status code 404 and above json object:

接下类，如果你的应用程序的某个地方抛出 `UserNotFoundException` ,Nest将上示 `json` 对象以及 `404` 状态码返回给用户。

```
{
  "message": "User not found."
}
```

> It allows you to focus on logic and make your code much easier to read.

它允许你重视逻辑，并使你的代码简单易懂。

公共路由前缀

> To set default path (route prefix), use setGlobalPrefix() method of INestApplication object. Example:

使用 `INestApplication` 对象的 `setGlobalPrefix()` 方法可以设置默认路径 `(route prefix)` 。

```
const app = NestFactory.create(ApplicationModule);
app.setGlobalPrefix('api');
```

生命周期事件

> There are two module lifecycle events OnModuleInit and OnModuleDestroy. You should use them for all the initialization stuff and avoid to work in the constructor. The constructor should only be used to initialize class members but nothing more. Example:

有两个生命周期事件 `OnModuleInit` 和 `OnModuleDestroy` 。你应该使用它们来初始化所有的内容，从而避免在构造函数中初始化。因为构造函数只用于初始化类成员。

示例：

```
import { OnModuleInit, OnModuleDestroy } from '@nestjs/common';

@Component()
export class UsersService implements OnModuleInit, OnModuleDestroy {
    onModuleInit() {
        console.log('Module initialized...');
    }
    onModuleDestroy() {
       console.log('Module destroyed...');
    }
}
```

混合应用

> It's possible to connect infinite count of microservices to your existing Nest web application.

可能会在你现有的Nest web应用程序中上连接无数微服务。

> Example:

示例：

```
const app = NestFactory.create(ApplicationModule);
const microservice = app.connectMicroservice({
    transport: Transport.TCP,
});

app.startAllMicroservices(() => console.log('All microservices are listening...'));
app.listen(port, () => console.log('Application listen on port:', port));
```

Lazy 微服务客户端

> Sometimes you have to load initial data before you can create your @Client(). In this case, you can use ClientProxyFactory, which provides create() method.

有时候在创建 `@Client()` 之前你需要加载原始数据。这时，你可以使用 `ClientProxyFactory`，它提供 `create()` 方法。

```
import { ClientProxyFactory } from '@nestjs/microservices';
const client = ClientProxyFactory.create({
    transport: Transport.TCP,
});
```

多个同步服务器

> Since you have full control of express instance lifecycle, it's not a problem to create a few multiple simultaneous servers (e.g. both HTTP & HTTPS). Example:

因为你已经可以完全控制 `express` 实例的生命周期了，所以创建多个同步同步服务器（例如 `HTTP & HTTPS`)也是很简单的。

示例：

```
let httpsOptions = {
    key: fs.readFileSync("./secrets/private-key.pem"),
    cert: fs.readFileSync("./secrets/public-certificate.pem")
};

const server = express();
const app = NestFactory.create(ApplicationModule, server);
app.init();

http.createServer(server).listen(3000);
https.createServer(httpsOptions, server).listen(443);
```