초격차 패키지 Online.

Nest.js 기본 개념

Nest.js 기본개념

1 Nest.js 의 기본 구조

Nest.js 의 기본구조

Request

Controller

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

Module

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Service

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

@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

app.controller.ts

```ts
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

NestJS에서는 HTTP 요청을 처리하기 위해 컨트롤러를 사용

컨트롤러는 특정 URI 엔드포인트와 HTTP 요청 메서드를 처리하는 메서드를 정의

app.service.ts

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

@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

NestJS에서는 서비스를 사용하여 컨트롤러에서 사용할 비즈니스 로직을 구현

서비스는 컨트롤러와 같은 클래스이며, Injectable 데코레이터를 사용하여 주입

app.module.ts

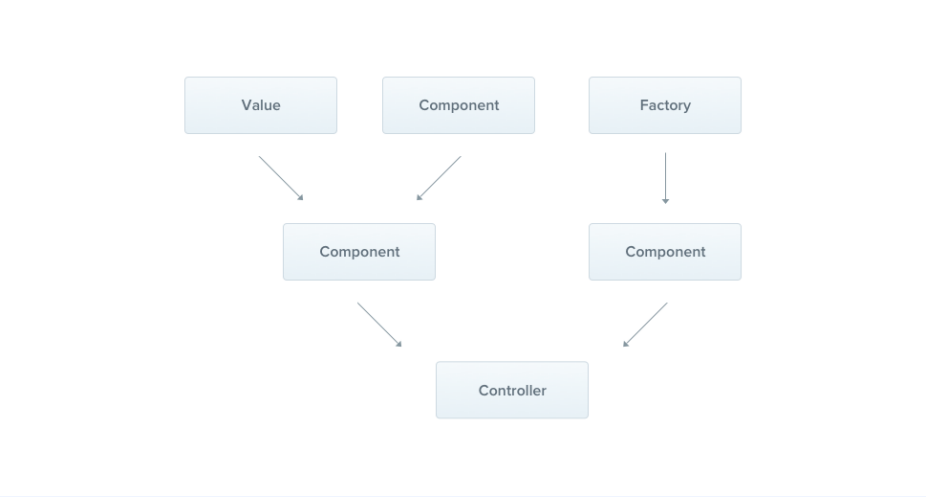
```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

NestJS에서는 모듈을 사용하여 애플리케이션을 구성

모듈은 특정 기능 또는 비즈니스 로직을 담당하는 컴포넌트 집합

애플리케이션에 필요한 모든 컨트롤러, 서비스, 프로바이더 및 미들웨어 등을 모듈에 등록

프로바이더(Providers)



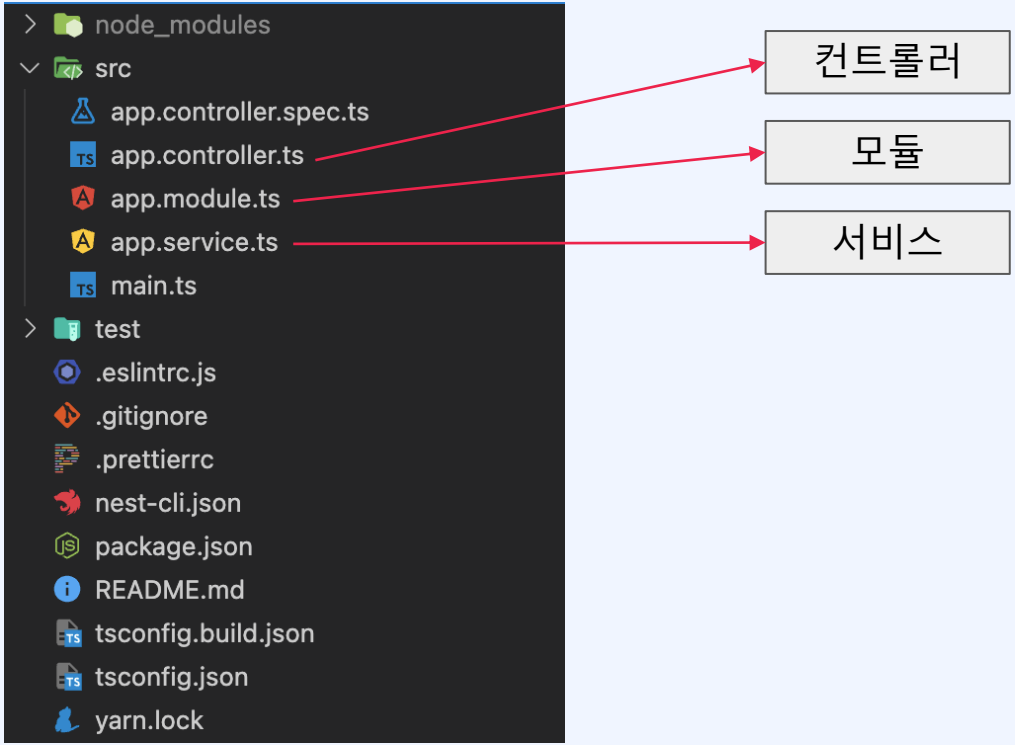
NestJS에서는 프로바이더를 사용하여 의존성 주입을 관리

프로바이더는 컨트롤러나 서비스에서 사용하는 객체, 함수 등을 제공

Nest.js 기본개념

2 프로젝트 구성

프로젝트 구성



Main.ts

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
bootstrap();
```

NestJS 애플리케이션의 진입점

NestFactory 클래스를 사용하여 NestJS 애플리케이션을 생성

생성된 애플리케이션에 필요한 미들웨어 및 모듈을 등록

HTTP 서버를 시작

app.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

NestJS에서는 모듈을 사용하여 애플리케이션을 구성

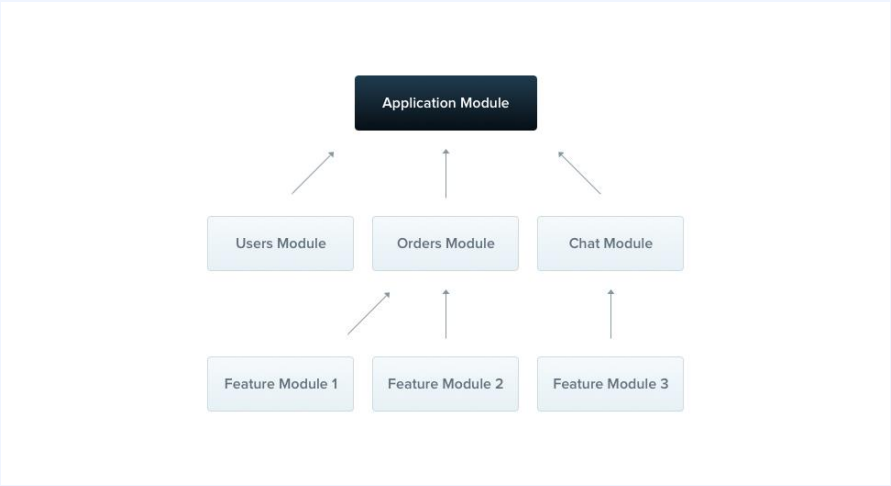
모듈은 특정 기능 또는 비즈니스 로직을 담당하는 컴포넌트 집합

애플리케이션에 필요한 모든 컨트롤러, 서비스, 프로바이더 및 미들웨어 등을 모듈에 등록

Nest.js 기본개념

2 Module

모듈(Modules)



모듈(Modules)

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';
import { CommonModule } from '../common/common.module';

@Module({
  imports: [CommonModule],
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService],
})
export class CatsModule {}
```

모듈(Modules)

providers

모듈이 생성하고, 의존성 주입 컨테이너에 추가할 클래스 인스턴스 또는 값의 배열

주로 서비스와 리포지토리 등이 여기에 포함됨

controllers

모듈이 정의하는 컨트롤러의 배열, 컨트롤러는 클라이언트의 요청을 처리하고, 적절한 응답을 반환하는 역할

imports

모듈이 의존하는 다른 모듈의 배열, NestJS는 이러한 모듈들을 현재 모듈의 providers와 controllers가 사용할 수 있도록 제공

exports

모듈에서 제공하며, 다른 모듈에서 import하여 사용할 수 있는 providers의 배열

기능 모듈 (Feature Modules)

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

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

애플리케이션의 특정 기능을 캡슐화

예) 사용자관리, 상품관리, 주문처리 등 특정 기능에 대해 컨트롤러, 서비스, 리포지토리 등을 그룹화

공유 모듈 (Shared Modules)

```
import { Module } from '@nestjs/common';
import { DatabaseService } from './database.service';

@Module({
  providers: [DatabaseService],
  exports: [DatabaseService],
})
export class DatabaseModule {}
```

애플리케이션 전반에 공유되는 기능을 제공

예를 들면 데이터베이스 접속, 로깅 인증 등 공통적인 작업을 수행하는 기능들을 Shared 모듈로 구성할 수 있음

공유 모듈의 적용 예)

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

@Module({
  imports: [DatabaseModule],
  controllers: [UsersController],
  providers: [UsersService],
})
export class UsersModule {}
```

@Global()

```
import { Module, Global } from '@nestjs/common';
import { DatabaseService } from './database.service';

@Global()
@Module({
  providers: [DatabaseService],
  exports: [DatabaseService],
})
export class DatabaseModule {}
```

애플리케이션 전역적로 사용되는 모듈이라면 Global 데코레이터를 통해 전역적으로 설정 가능

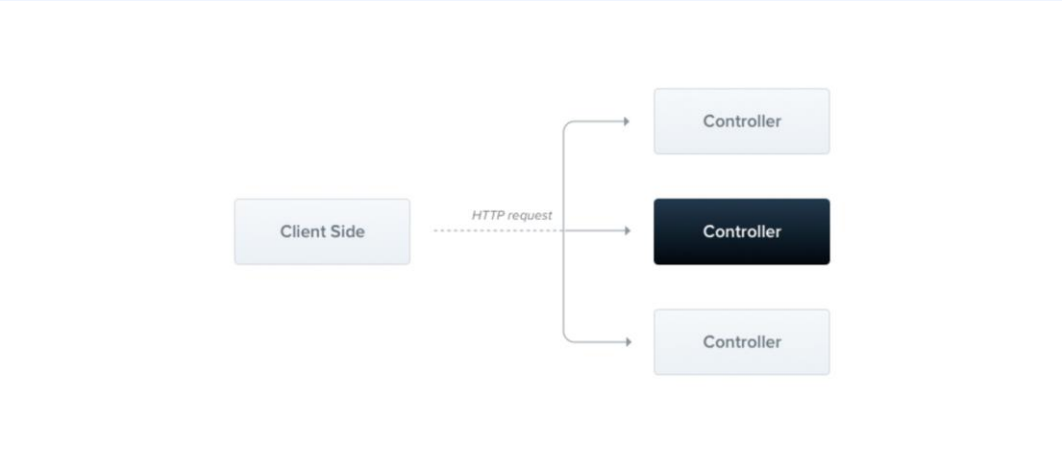
Global 데코레이터가 명시되어 있는 모듈은 imports 없이 사용가능

Global 데코레이터는 보통 애플리케이션의 루트나 코어 부분에 구현

Nest.js 기본개념

3 Controller

컨트롤러(Controller)



클라이언트의 요청을 받아 처리하고 응답을 반환하는 역할

REST API 엔드포인트를 노출하는데 사용

Routing

```
@Controller('hello')
export class HelloController {
    @Get()
    get(): string {
        return 'get';
    }

    @Post()
    create(): string {
        return 'create';
    }

    @Put()
    update(): string {
      return 'update';
    }

    @Delete()
    remove(): string {
        return 'remove';
    }
}
```

@Controller 데코레이터 사용

모든 표준 HTTP 메서드를 데코레이터 제공

매개변수 와 쿼리스트링

```
// 예시 /hello/gildong?country=korea
@Get(':name')
get(
    @Param('name') name: string,
    @Query('country') country: string
) {
    return `my name is ${name} from ${country}`;
}
```

@Param: 매개변수

@Query: 쿼리스트링

Nest.js 기본개념

4 Service

서비스(Service)

일반적인 비즈니스 로직을 담당

컨트롤러가 클라이언트의 요청을 처리하는데 필요한 작업을 처리

데이터베이스의 데이터를 가져오거나 외부 API 호출 등의 데이터 처리

서비스(Service)

```
@Injectable()
export class AppService {
  getHello(): string {
    return 'Hello World!';
  }
}
```

```
@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

의존성 주입

@Injectable 데코레이터 사용 > 클래스가 주입가능한 상태로 변환

Nest.js 기본개념

6 DI 이해하기

DI(Dependency Injection, 의존성주입)

소프트웨어 엔지니어링 디자인 패턴 중 하나

특정 클래스가 의존하고 있는 다른 클래스나 컴포넌트를 직접 만들지 않고, 외부에서 주입받아 사용하는 방식

모듈간의 높은 결합도를 줄이고, 유연성과 재사용성을 높이고자 나온 패턴

DI 동작 방식

1. 클래스는 필요한 의존성을 명시적으로 정의

2. DI 컨테이너 또는 IoC(Inversion of Control) 컨테이너는 이러한 의존성을 관리합니다. 이 컨테이너는 필요한 의존성을 찾아서 인스턴스를 생성하고, 이를 요청한 클래스에 주입합니다.

3. 클래스는 직접적으로 의존성을 생성하거나 관리할 필요 없이 해당 의존성을 사용할 수 있게 됩니다.

NestJS DI

```ts
// cats.controller.ts
import { Controller, Get, Post, Body } from '@nestjs/common';
import { CatsService } from './cats.service';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
  constructor(private catsService: CatsService) {}

  @Post()
  async create(@Body() cat: Cat) {
    this.catsService.create(cat);
  }

  @Get()
  async findAll(): Promise<Cat[]> {
    return this.catsService.findAll();
  }
}
```

CatService 의 인스턴스는 NestJS 프레임워크에서 생성하여 CatController 주입

의존성주입은 단위테스트에 용이

단, module 에 providers 에 명시되지 않는 service 는 의존성주입이 되지 않음