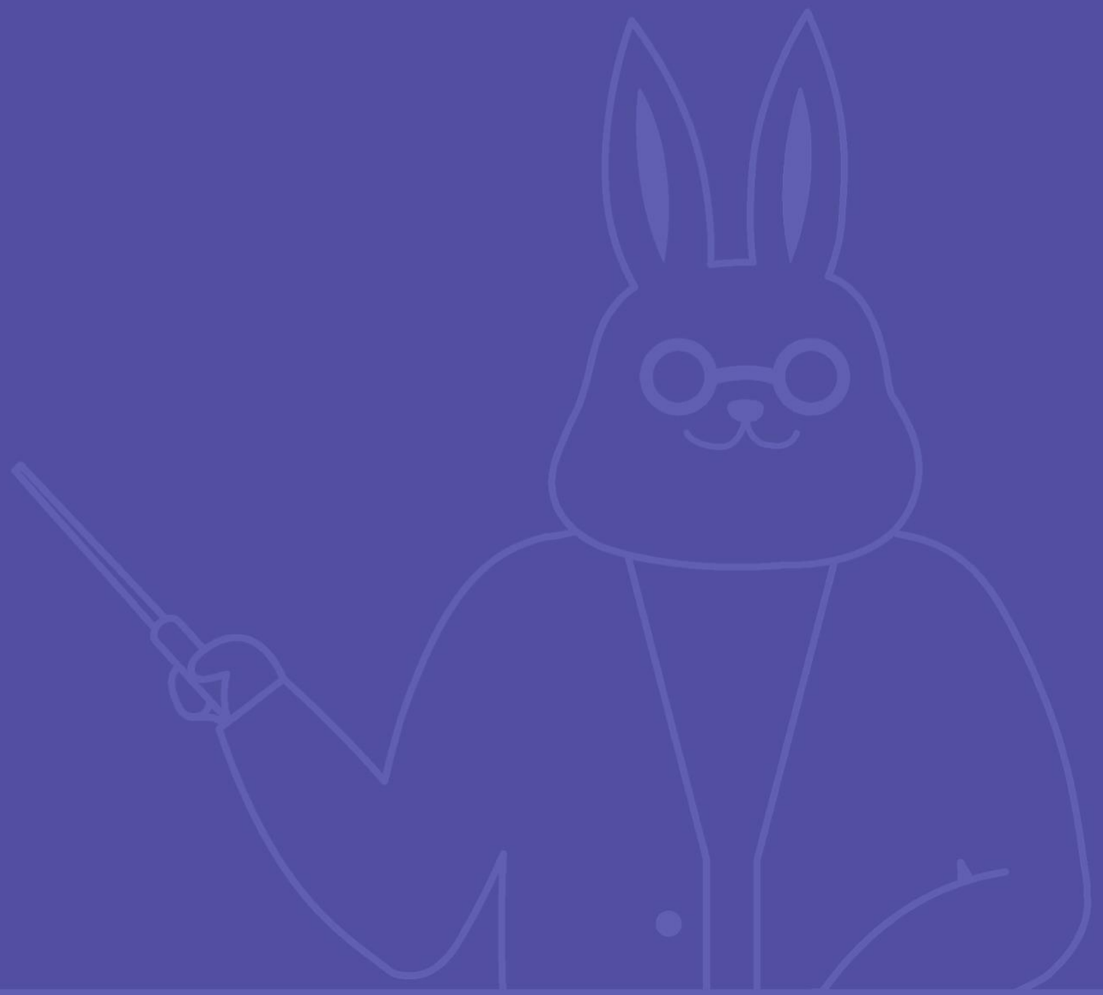

# 타입스크립트 II

## 03 Nodejs에서의 타입스크립트





/* elice */

# 목차

# 🎯 수강목표

## 1. 자신의 개발 환경에서 타입스크립트 세팅하기

실습을 통해 각자의 vscode 에디터에서 직접 타입스크립트를 세팅해보면서 타입스크립트를 사용할 수 있는 개발 환경을 직접 구축해봅니다.

## 2. Node에서 자바스크립트와 타입스크립트의 차이

express 앱을 자바스크립트로도 개발해보고, 타입스크립트로 마이그레이션 해보면서 둘의 차이를 알 수 있습니다.

## 3. 백엔드 프레임워크 Nestjs 이해하기

요즘 실무에서 많이 쓰이고 있는 Nestjs 프레임워크에 대해 이해하고, 간단한 앱을 만들어보면서 사용성을 익힙니다.

01

# Nodejs 웹앱 프레임워크 Express 소개



/* elice */

## 웹 어플리케이션 프레임워크 express

**Nodejs 기반의 웹 어플리케이션 프레임워크**

웹앱 프레임워크는 **웹 어플리케이션을 만들기 쉽게 해주는 기능과 구조를 제공**해주는 것

이외에도 fastify나 koa 등 다양한 웹 어플리케이션 프레임워크가 존재

퍼포먼스는 fastify(벤치마크)가 가장 우수하지만, express가 가장 널리 쓰이고 있음

## express: hello-world 예제 앱 만들기

Shell

```
yarn add express
```

package.json의 scripts에 추가

```
"start": "node src/app.js",
```

src/app.js

```
const express = require('express');
const app = express();
const port = 3000;

app.get('/', (req, res) => {
  res.send('Hello World!');
});

app.listen(port, () => {
  console.log('Example app listening
at http://localhost:${port}');
});
```

## express 개념 설명: Router

REST API에서의 URI: 웹 리소스 또는 인터페이스의 경로 표현

express에서의 라우터는 **애플리케이션 엔드 포인트(URI)를 정의**하고,

클라이언트가 **어떤 경로로 요청했느냐에 따라 다른 코드를 실행**

## express 개념 설명: Router

Routing

```
// GET method route
app.get('/', function (req, res) {
  res.send('GET request to the
homepage');
});

// POST method route
app.post('/', function (req, res) {
  res.send('POST request to the
homepage');
});
```

도메인이 http://locahost:3000 이면
http://locahost:3000으로 요청했을 때

get 요청이면 'GET request to the homepage'로 응답

Post 요청이면 'POST request to the homepage'로 응답

## express 개념 설명: Middleware

express의 미들웨어는 함수

- **요청과 응답 사이에서** 요청이나 응답을 변경
- 에러가 났을 때 에러 응답을 보내거나 응답을 종료

## express 개념 설명: Middleware

Routing

```
function errorMiddleware(err, req,
res, next) {
  res.status(500);
  res.render("error", { error: err });
}

function errorMiddleware2(err, req,
res, next) => {
   res.status(500).send(error.message)

}
```

미들웨어 안에서 응답하거나
next를 호출해서
다음 미들웨어로 요청을 전달

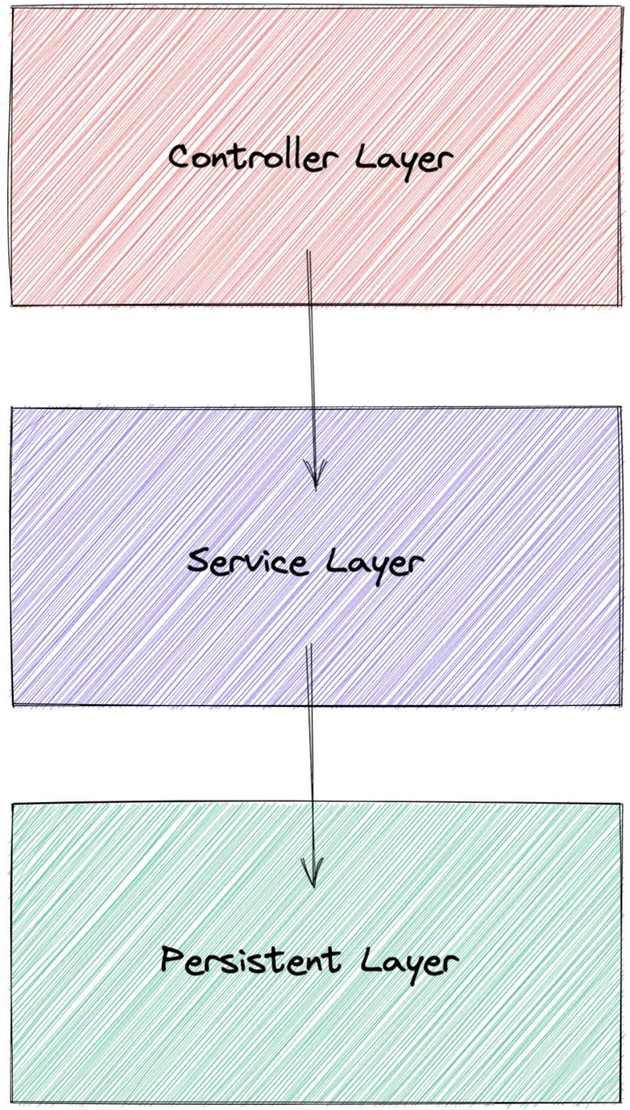
그렇지 않으면 응답이 오지 않은 상태로
실행이 정지됨

## Express 아키텍처: 3계층 설계

들어오는 요청을 service 클래스에 라우팅

비즈니스 로직을 갖고 있는 서비스 클래스
CLI와 같은 스탠드얼론(독립형) 어플리케이션에서는
독자적으로 쓰일 수 있음

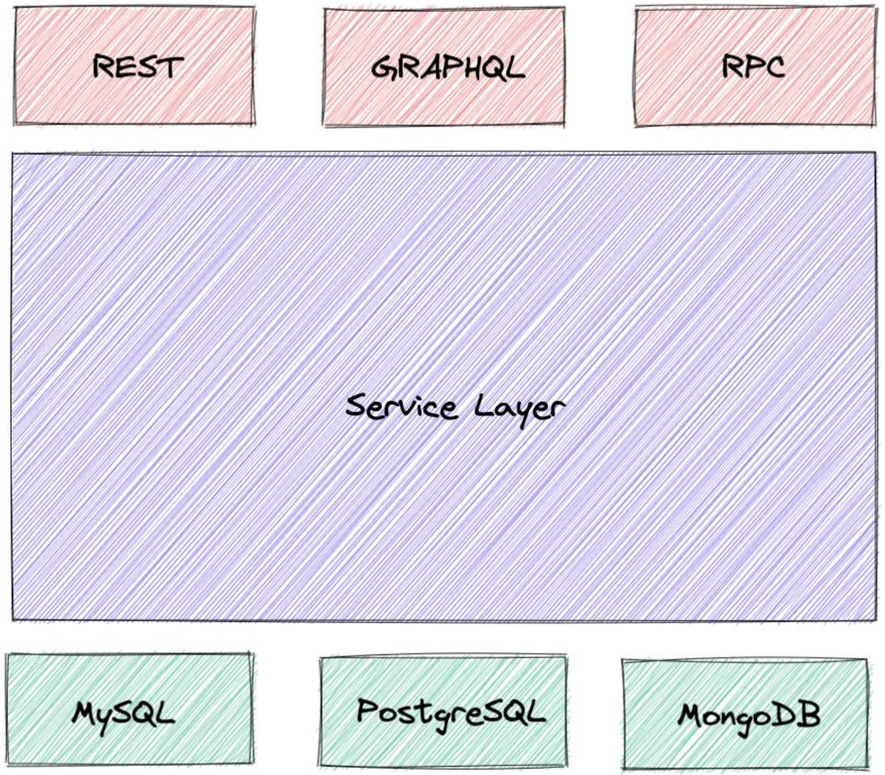
DB나 웹/앱 스토리지 같이
영구히 데이터를 저장할 수 있는 계층



## Express 아키텍처: 3계층 설계

비즈니스 로직을 서비스 레이어에 담아야 하는 이유:

Controller layer나 persistent layer가 다른 것으로
변경되어도 서비스 레이어의 비즈니스 로직이
그대로라면 변경된 레이어만 교체해주면 되기 때문



02

# 자바스크립트 + Express로 커피 주문 앱 개발하기

## Cafe Express Application

**원하는 앱의 기능**

1. 커피 메뉴 생성 및 조회 기능

2. 커피 주문 및 조회 기능. 모든 주문 조회 기능

3. 라우터에서 에러가 날 경우 미들웨어에서 처리

## 프로젝트 구조 및 파일 설명

프로젝트 구조

```
src
├── app.js
├── middlewares
│   └── errorMiddleware.js
├── routers
│   ├── menuRouter.js
│   └── orderRouter.js
└── services
    ├── CoffeeMenuService.js
    └── OrderService.js
```

- errorMiddleware.js: 에러를 핸들링하는 미들웨어
- app.js: 서버를 시작하는 메인 파일
- routers/*:app으로 들어오는 요청을 라우팅해주는 파일
- services/*: 라우팅되는 요청들을 핸들링하는 코드들. 비즈니스로직이 들어가는 코드들

03

# 자바스크립트에서 타입스크립트로 마이그레이션

## 타입스크립트 프로젝트로 변경하는 법

1. 트랜스파일링할 디렉토리 정하기
2. tsconfig.json 작성하기
3. babel을 사용한 es6 이상의 문법 지원
4. 확장자명 변경: *.js -> *.ts로 변경
5. 오류 제거: 타입 추가하기

## 트랜스파일링할 디렉토리 정하기

디렉토리 구조

```
.
├─── src
|    ├─── app.controller.spec.ts
|    ├─── app.controller.ts
|    ├─── app.module.ts
|    ├─── app.service.ts
|    └─── main.ts
└─── tsconfig.json
```

트랜스파일링할 디렉토리에 tsconfig.json 위치

만약, src 외에 다른 폴더가 있다면 그리고 서로 다른 트랜스파일 결과가 필요하다면 각 폴더에 tsconfig.json 위치

## tsconfig.json 작성하기

디렉토리 구조

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "strict": true,
    "esModuleInterop": true,
    "skipLibCheck": true,
  },
  "include": ["./src/**/*"]
}
```

tsconfig.json 파일 생성

```
npm i -g typescript
```

```
tsc --init
```

Refer: https://www.typescriptlang.org/tsconfig 또는 https://typescript-kr.github.io/pages/compiler-options.html

## Babel과 함께 사용하기

### Babel vs tsc

**tsc**

Typescript 파일과 트랜스파일링될 자바스크립트 파일이 **타이핑되는 것 빼곤 동일**
(ECMAScript 버전 신경 안 쓰고, commonjs 방식의 require/export 방식 사용 등)

**Babel**

자바스크립트 파일을 **원하는 형태로 트랜스파일링**
(esm 방식의 import/export, 개발 시에는 es6 이상의 자바스크립트 버전 사용하고 결과물은 es5 버전을 원할 때 등)

## Babel과 함께 사용하기: 패키지 설치

package.json

```
"scripts": {
    "build": "babel src --out-dir dist --copy-
files --extensions '.ts,.js'",
    "start": "nodemon dist/app.js",
  },
"devDependencies": {
    "@babel/cli": "^7.16.0",
    "@babel/core": "^7.16.0",
    "@babel/plugin-transform-runtime":
"^7.16.4",
    "@babel/preset-env": "^7.16.4",
    "@babel/preset-typescript": "^7.16.0",
    // …
  }
```

프로그램이 실행되기 전 컴파일 단계에서 사용될 라이브러리들이므로 devDependencies로 설치

- @babel/cli: cli에서 babel 명령어 사용할 때 필요
- @babel/core: 트랜스파일링하는 데 사용되는 주요 코드들이 담긴 라이브러리
- @babel/plugin-transform-runtime: 폴리필이 필요할 때 바벨 내부의 헬퍼 함수 사용하게 해주는 라이브러리
- @babel/preset-env: babel 플러그인들을 합쳐놓은 번들 파일
- @babel/preset-env-typescript: 타입스크립트 환경을 위한 플러그인들을 합친 번들 파일

## Babel과 함께 사용하기: babel.config.js 파일 작성

babel.config.js

```
module.exports = {
  presets: [
    [
      "@babel/preset-env",
      {
        targets: {
          node: "current",
        },
      },
    ],
    "@babel/preset-typescript",
  ],
  plugins: ["@babel/plugin-transform-runtime"],
};
```

tsconfig.json의 compilerOptions에서
**'isolatedModules' 옵션을 true로 설정**

타입을 export해서 값처럼 쓰는 등, babel이나 swc등의 트랜스파일러를 사용했을 때 트랜스파일러가 잘못 사용되는 것을 방지

## 확장자명 변경: *.js -> *.ts / 오류 제거하기

*.js

```
export function
errorMiddleware(error, req,
res, next) {
  res.status(500);
  res.render("error",
{ error });
}
```

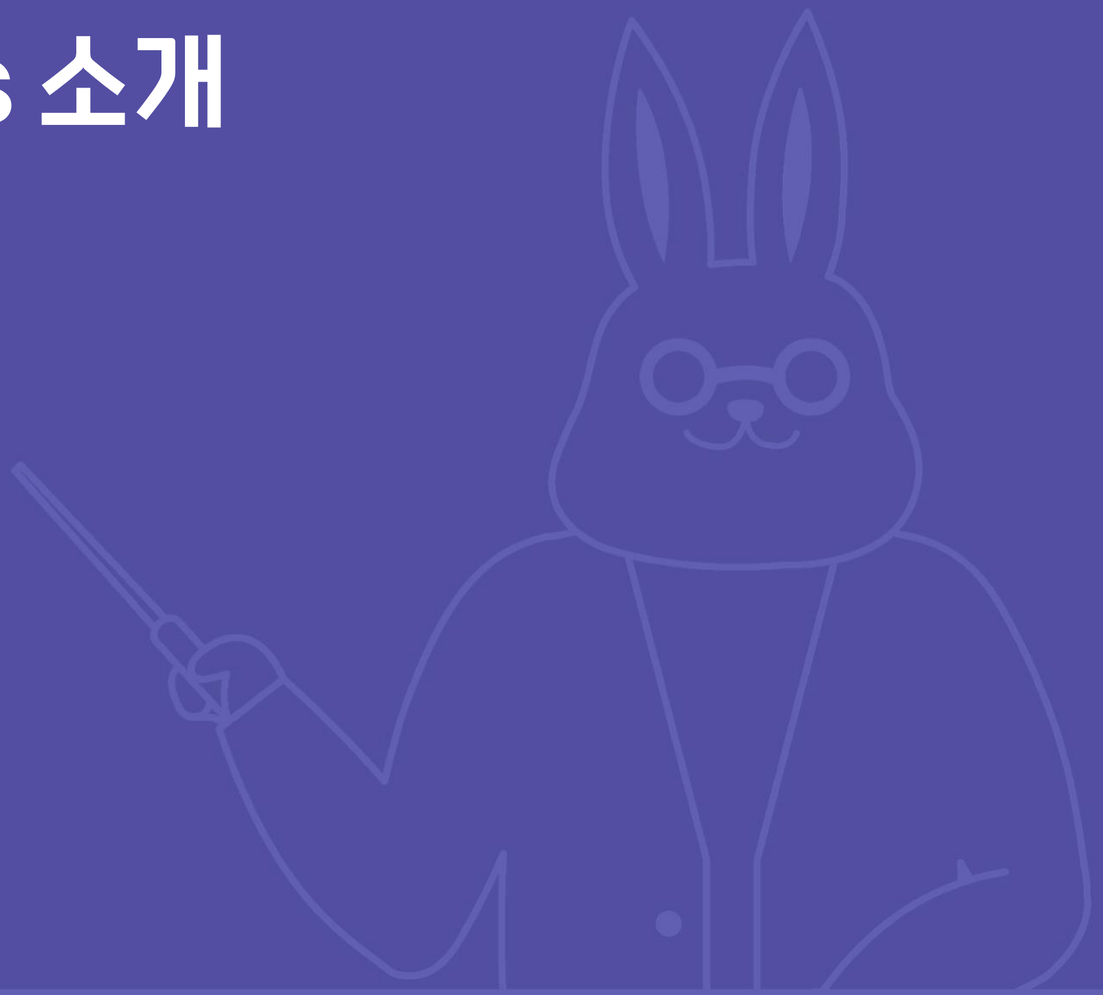
*.ts

```
import { Request, Response } from
"express";

export function errorMiddleware(
  err: Error,
  req: Request,
  res: Response,
) {
  res.status(500);
  res.render("error", { err });
}
```

04

# Nodejs 프레임워크 Nestjs 소개



## Nodejs 프레임워크 Nestjs의 특징

Nodejs Framework

유연한 Platform
(express, fastify)

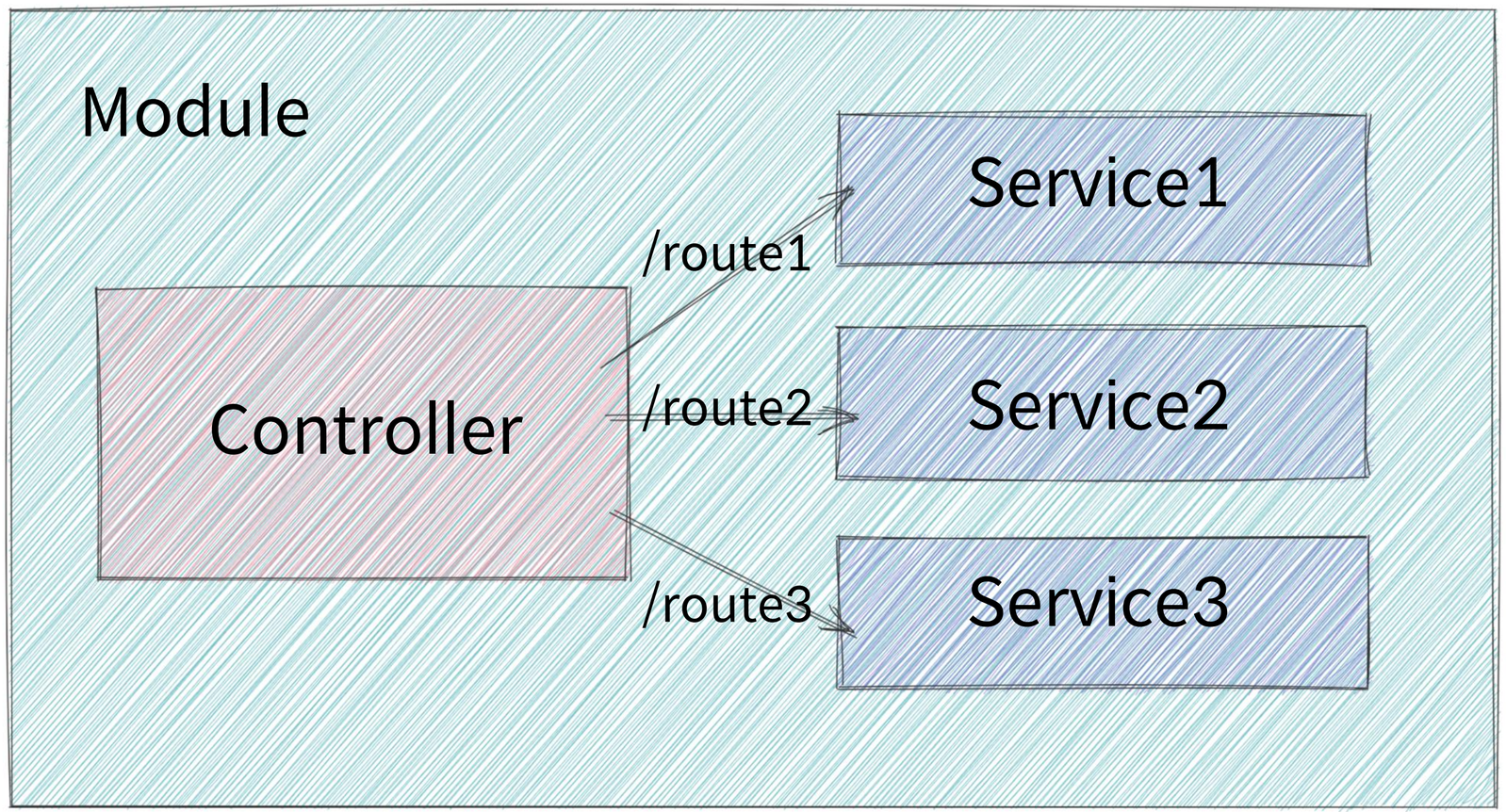
다양한 서버 개발 가능
(GraphQL, REST, CLI, MVC)

Angular에 영향을 받은
깔끔하고 견고한 아키텍처

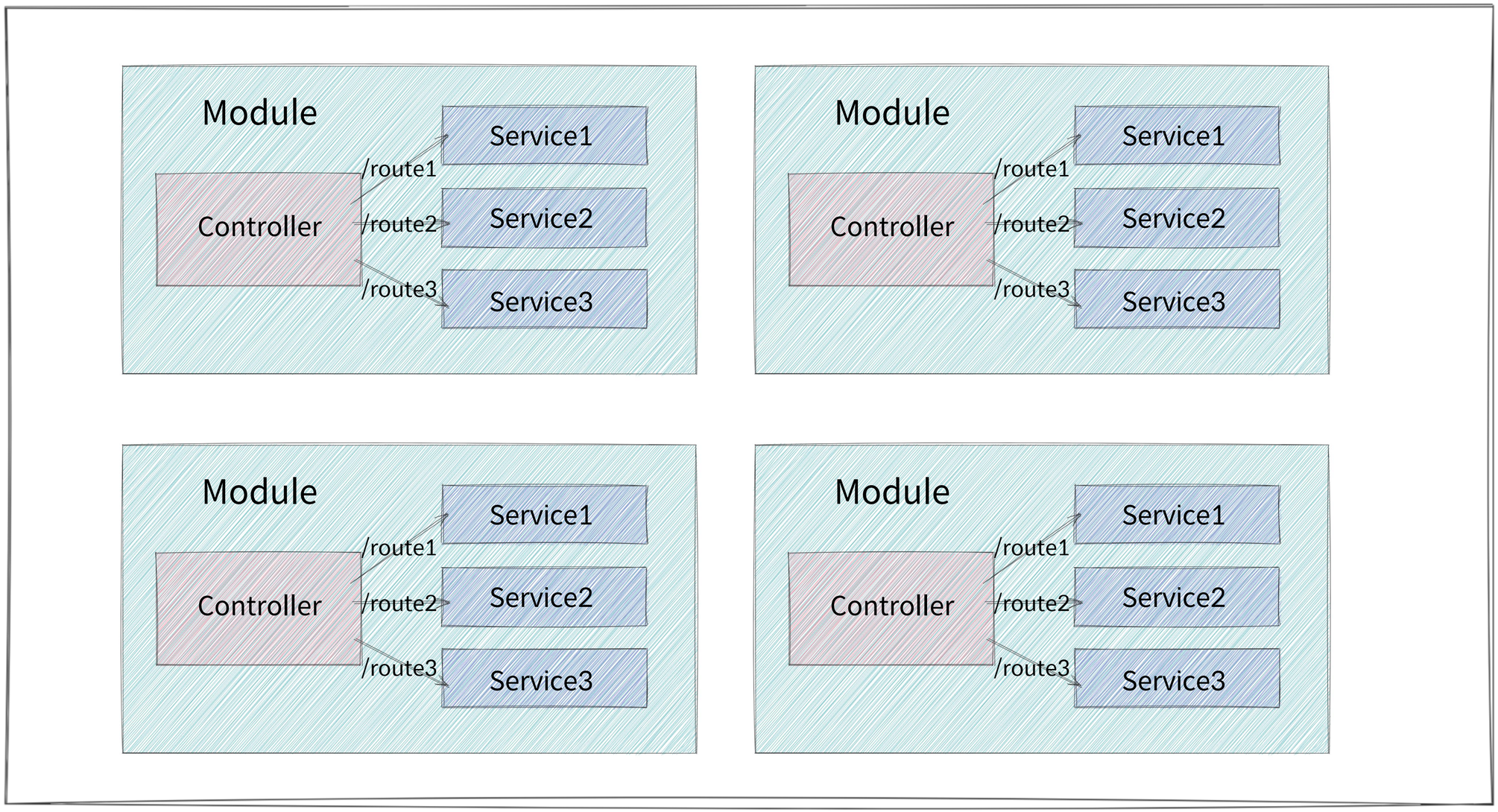
Typescript, jest, lint,
Code generator cli 지원

다양한 플러그인
(Testing, Swagger, etc)

## Nestjs 구조


Module
Controller
/route1
/route2
/route3
Service1
Service2
Service3

## Nestjs 구조


Module
Module
Controller
/route1
/route2
/route3
Service1
Service2
Service3
Module
Controller
/route1
/route2
/route3
Service1
Service2
Service3
Module
Controller
/route1
/route2
/route3
Service1
Service2
Service3
Module
Controller
/route1
/route2
/route3
Service1
Service2
Service3

## Nestjs의 설계 패턴: Dependency Injection

### 생성자 함수 내에서 생성

```
class PostgresDB {
  getOne(id: string) {
    return {};
  }
  getList(ids: string[]) {
    return {};
  }
}

class PostService {
  db: PostgresDB;

  constructor() {
    this.db = new PostgresDB();
  }

  getPost(id: string) {
    return this.db.getOne(id);
  }
}
```

### 무수히 많은 클래스에서 반복될 때

```
class PostService {
  db: PostgresDB;

  constructor() {
    this.db = new PostgresDB();
  }
}

class CommentService {
  db: PostgresDB;

  constructor() {
    this.db = new PostgresDB();
  }
}

class UserService {
  db: PostgresDB;

  constructor() {
    this.db = new PostgresDB();
  }
}
```

## Nestjs의 설계 패턴: Dependency Injection

### 생성자 함수 내에서 생성

```
class Logger {}

class PostgresDB {
  constructor(private logger: Logger) {}
}

class PostService {
  db: PostgresDB;

  constructor() {
    this.db = new PostgresDB(new Logger());
  }
}

class CommentService {
  db: PostgresDB;

  constructor() {
    this.db = new PostgresDB(new Logger());
  }
}

class UserService {
  db: PostgresDB;

  constructor() {
    this.db = new PostgresDB(new Logger());
  }
}
```

의존성을 갖고 있는 클래스의 생성자 함수에 변경이 생기는 경우 그 클래스를 의존하고 있는 모든 클래스에서 생성 로직을 변경해줘야 함

## Nestjs의 설계 패턴: Dependency Injection

생성자 함수 내에서 생성

```
class Logger {}

class PostgresDB {
  constructor(private logger: Logger) {}
}

class PostService {
  constructor(private db: PostgresDB) {}
}

class CommentService {
  constructor(private db: PostgresDB) {}
}

class UserService {
  constructor(private db: PostgresDB) {}
}

const db = new PostgresDB(new Logger());
new PostService(db);
new CommentService(db);
new UserService(db);
```

### Dependency Injection

따라서 의존하는 클래스를 자신의 생성자 함수에서 생성하지 않고 외부에서 인스턴스를 주입받는 형태로 변경

여러 인스턴스가 생기지도 않고, 주입하기 전 인스턴스화할 때만 변경된 생성 로직을 적용하면 됨

## Nestjs의 Dependency Injection: 데코레이터

Nestjs에서는 Dependency Injection을 데코레이터를 통해서도 가능

데코레이터를 사용함으로써 생성 로직을 작성하는 데 드는 비용을 절감

## Nestjs의 Dependency Injection 데코레이터: Injectable, Inject

### Injectable: 의존 클래스(Provider)

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

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

### 주입받는 클래스

```
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();
  }
}
```

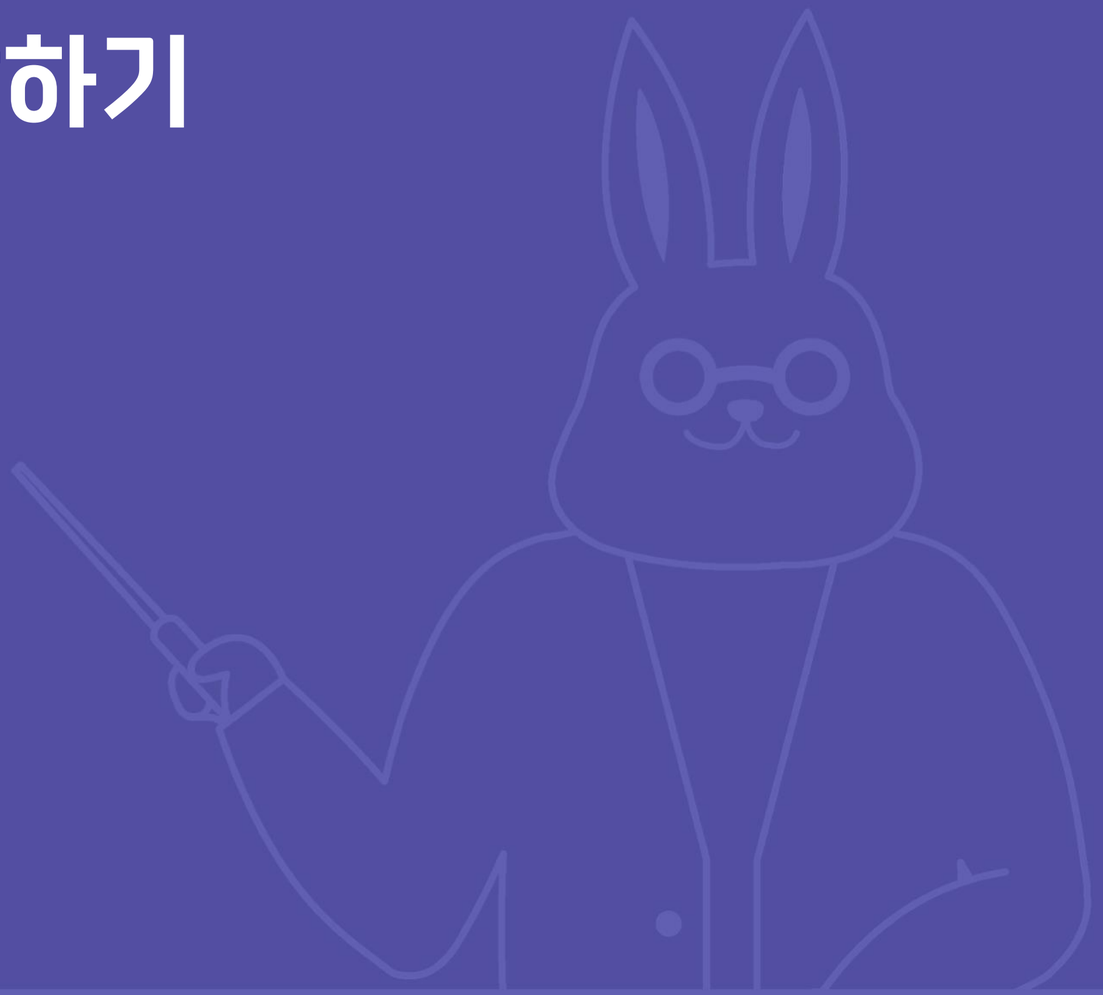
### Module

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

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

05

# Nestjs로 커피 주문 앱 개발하기





/* elice */

## 프로젝트 생성과 실행

### 1. Nestjs cli 설치

```
npm i -g @nestjs/cli
```

### 2. 프로젝트 생성

```
nest new word-count
```

### 3. 프로젝트 실행

```
yarn start
```

## 프로젝트 구조 및 파일 설명

프로젝트 구조

```
.
├── README.md
├── nest-cli.json
├── package.json
├── src
│   ├── app.controller.spec.ts
│   ├── app.controller.ts
│   ├── app.module.ts
│   ├── app.service.ts
│   └── main.ts
├── test
│   ├── app.e2e-spec.ts
│   └── jest-e2e.json
├── tsconfig.build.json
├── tsconfig.json
└── yarn.lock
```

- nest-cli.json: nest-cli에 대한 설정 파일
- main.ts: 서버를 시작하는 메인 파일
- app.module.ts: 가장 최상위 모듈 AppModule이 있는 파일
- app.controller.ts: AppModule에 들어오는 요청을 라우팅
- app.service.ts: 라우팅되는 요청을 핸들링하는 코드들. 비즈니스로직이 들어가는 코드들
- tsconfig.build.json: nest에서 코드를 build 할 때 사용하는 tsconfig 설정 파일. 없으면 nest에서 build 할 때 오류 발생

## express와의 차이점

- 아키텍처에 대한 고민을 하지 않아도 된다
- 의존성 주입이 간편하다
- 타입스크립트 지원, jest, eslint, prettier 등 기본 설정이 다 되어있다
- express 또는 fastify를 유연하게 선택할 수 있다

06

# Nestjs에서 타입스크립트를 유용하게 사용하기

## 타입스크립트에서 Parameter Decorator로 validation

Parameter + Method Decorator + reflect-metadata

```
import "reflect-metadata";
const requiredMetadataKey = Symbol("required");

function required(target: Object, propertyKey: string | symbol, parameterIndex:
number) {
 let existingRequiredParameters: number[] =
Reflect.getOwnMetadata(requiredMetadataKey, target, propertyKey) || [];
 existingRequiredParameters.push(parameterIndex);
 Reflect.defineMetadata( requiredMetadataKey, existingRequiredParameters, target,
propertyKey);
}

function validate(target: any, propertyName: string, descriptor:
TypedPropertyDescriptor<Function>) {
 let method = descriptor.value!;

 descriptor.value = function () {
  let requiredParameters: number[] = Reflect.getOwnMetadata(requiredMetadataKey,
target, propertyName);
  if (requiredParameters) {
   for (let parameterIndex of requiredParameters) {
    if (parameterIndex >= arguments.length || arguments[parameterIndex] ===
undefined) {
     throw new Error("Missing required argument.");
    }
   }
  }
  return method.apply(this, arguments);
 };
}
```

Code: https://www.typescriptlang.org/docs/handbook/decorators.html

타입스크립트에서 parameter validation 하려면

**reflect-data 라이브러리를 사용하여 parameter decorator와 method decorator를 정의**하여 함께 사용했어야 함

## Nestjs의 Custom Decorator

### Parameter Decorator

```
import { createParamDecorator,
ExecutionContext } from '@nestjs/common';

export const validateParamString =
createParamDecorator(
  (paramKey: string, ctx: ExecutionContext)
=> {
    const req =
ctx.switchToHttp().getRequest();
    isNotNumberString(req.params[paramKey]);
    isNotObjectString(req.params[paramKey]);
  },
);
```

### Controller

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

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

  @Get('/:keyword')
  getKeyword(@validateParamString('keyword')
keyword: string): string {
    return this.appService.getDoc(keyword);
  }
}
```

## Custom Provider: Non-class-based provider tokens

interface를 만들어서 class에서 implements하고
해당 interface를 주입하고 싶은 경우 사용

interface는 자바스크립트 파일로 트랜스파일링 되면 없어지는 타입
따라서 **값으로 쓰일 수 있는 토큰(문자열, 심볼)을 사용해 대신 주입**

## Custom Provider: Non-class-based provider tokens

animal.service.ts

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

const ANIMAL =
Symbol('ANIMAL');

interface AnimalService {
  eat: (food: string) => void;
}

@Injectable()
export class AnimalServiceImpl
implements AnimalService {
  eat(food: string) {}
}
```

animal.controller.ts

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

class AnimalController
{
  constructor(@Inject(A
NIMAL) animal:
AnimalService) {}
}
```

animal.module.ts

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

@Module({
  imports: [],
  controllers:
[AnimalController],
  providers: [{ provide:
ANIMAL, useClass:
AnimalServiceImpl }],
})
export class AppModule {}
```

# 크레딧

/* elice */

**코스 매니저**
이재성

**콘텐츠 제작자**
송현지

**강사**
송현지

**감수자**
이재성

**디자이너**
김루미

# 연락처


**TEL**
070-4633-2015

**WEB**
https://elice.io

**E-MAIL**
contact@elice.io