

Backend-NestJs 04

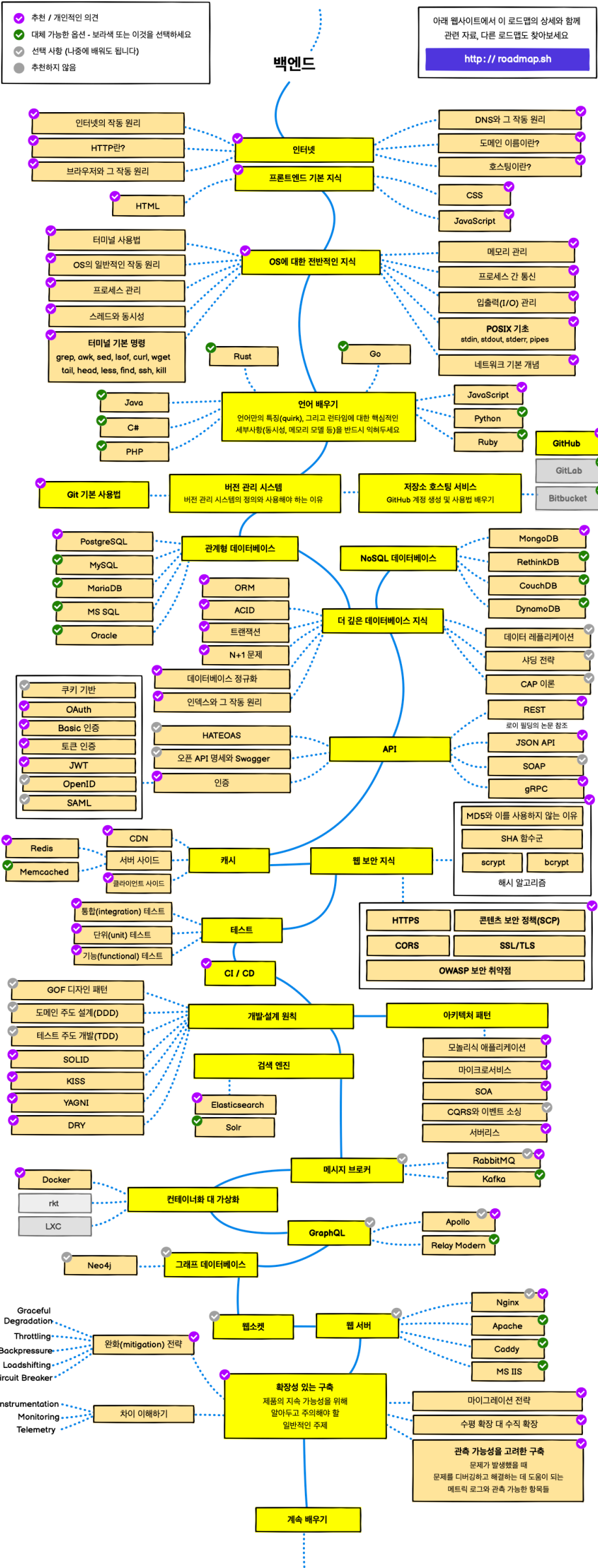
**NestJS 서버 개발 :: ERD / DB 정규화, ORM, Postgres, PgAdmin /
Configuration**

고려대학교 컴퓨터학과
2017320218 홍석민

Backend Roadmap

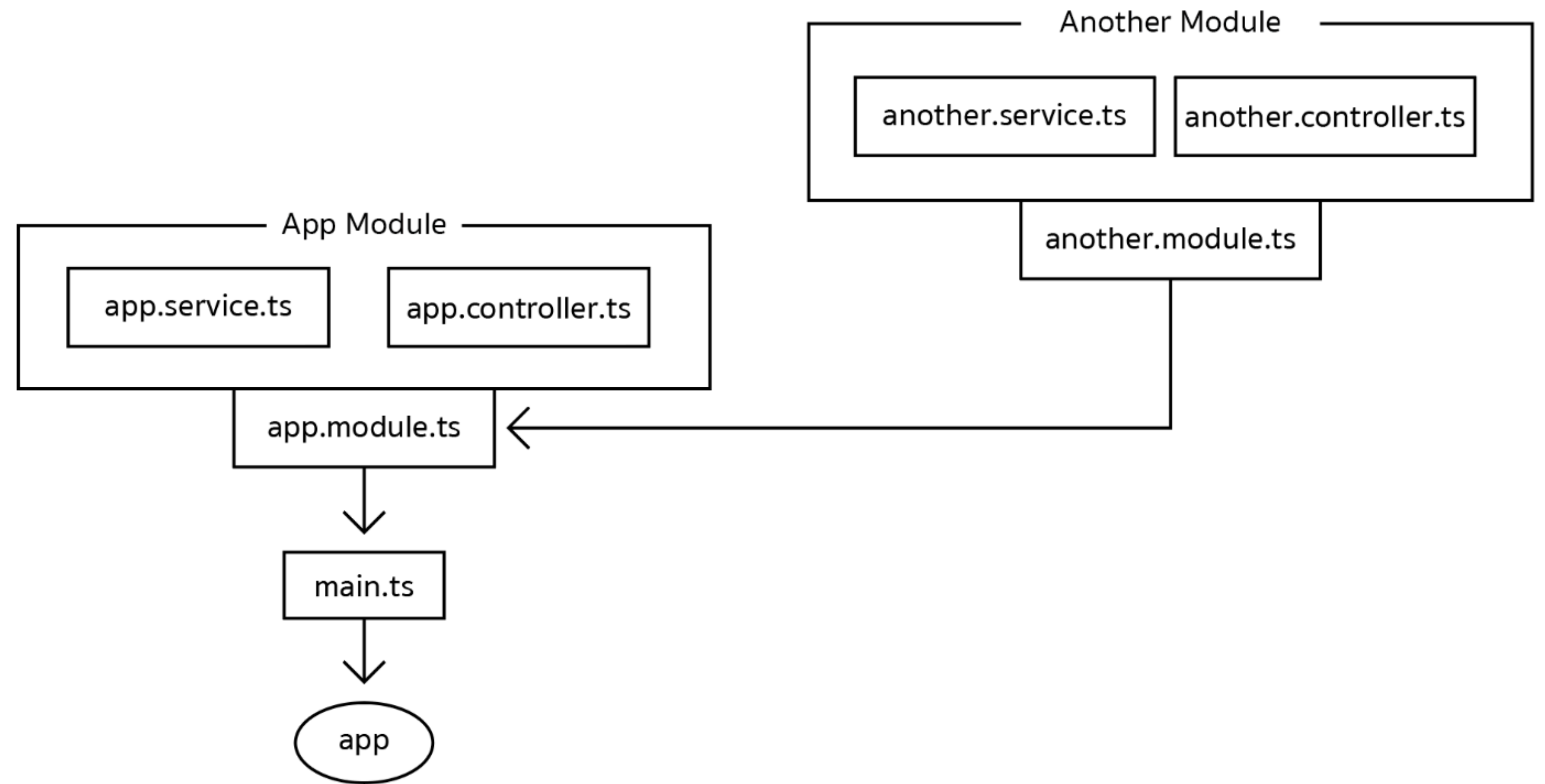
언제 다하지

- <https://roadmap.sh/backend>



NestJS 의 구조 ::

- Module
- Provider
- Controller
- Dependency Injection



Nest :: TypeScript

Typing, Interface, Generic, Decorator

- <https://wikidocs.net/158474>
- <https://wikidocs.net/158481>
- <https://typescript-kr.github.io/>

Nest :: AOP

관점 지향 프로그래밍(AOP : Aspect Oriented Programming)

- 횡단 관심사 (Cross - Cutting Concern) 의 분리를 허용하여 모듈성을 증가시키는 것이 목적인 프로그래밍 패러다임
- **횡단 관심사** ? : 애플리케이션 전반에 걸쳐 제공해야하는 공통 요소
- NestJs는 이러한 AOP 를 데코레이터를 활용해서 적용한다.
- 백엔드 애플리케이션의 요구사항
 - 사용자 요구사항
 - **유효성 검사**
 - **로깅**
 - **보안**
 - **트랜잭션**

Nest :: Controller

Router, MVC Pattern 의 Controller

```
TS users.controller.ts U X
src > users > TS users.controller.ts > ...
1  import { Controller } from '@nestjs/common';
2
3  @Controller('users')
4  export class UsersController {}
5
```

- Generate Command : nest g controller
- 서버로 들어오는 요청을 처리하고, 응답을 가공하는 역할
- 서버에서 제공하는 기능을 어떻게 클라이언트와 주고받을 지에 대한 인터페이스를 정의
- Decorator : @Controller(<ENDPOINT_PREFIX>)
- Endpoint 라우팅 : e.g.) (“/users”, “/posts”, “/lectures”)
 - 각각의 요청을 분류하여 알맞은 로직을 수행해주는 Provider로 라우팅
 - @Get(), @Post(), @Put(), @Delete() : Controller의 prefix와 함께 Router Path 설정
 - 존재하지않는 Route Path : 404 Not Found
 - Admin 페이지의 경우 ?
 - 보안적 관점에서 보면...

Nest :: Controller / Request Object

@Req / Express

- 클라이언트의 Request 를 직접 핸들링할 때 사용
- 나중에 클라이언트에서 인증을 위해 쿠키에 토큰을 담아오거나 하는 경우 사용

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

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

  @Get()
  getHello(@Req() req: Request): string {
    console.log(req);
    return this.appService.getHello();
  }
}
```

Nest :: Controller / Response

Status Code / Serialize / @Res() / @HttpCode

- Status Code

- 200
- 201
- 301 ...

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

@HttpCode(202)
@Patch('/:id')
update(@Param('id') id: string, @Body() updateUserDto: UpdateUserDto) {
  return this.userService.update(+id, updateUserDto);
}
```

- Serialize

- Return “Hello World”
- Return {id:1, name: “Seokmin”, age:26, role:0 }

```
@Get()
findAll(@Res() res) {
  const users = this.userService.findAll()

  return res.status(200).send(users);
}
```


Nest :: Controller

@Param(key ?: string)

- 동적인 경로 핸들링
- 특정 리소스를 지정하여 접근할 때 사용
 - e.g.) userId = 1 에 대한 update, delete, get 요청
- Get('/param/:paramId) 로 받아서,
- 해당 요청을 받는 함수의 매개변수에서 @Param('paramId')로 받는다.
- Validation 적용 가능

REST API 명세

METHOD	URI	Params	Body	Description
GET	/users			전체 사용자 조회
GET	/users/{id}	id		사용자의 ID를 이용해서 특정 사용자 조회
PUT	/users/{id}	id	age, role	특정한 ID를 가진 사용자의 정보를 body에 전달한 age와 role에 따라서 업데이트
DELETE	/users/{id}	id		특정한 ID를 가진 사용자의 회원 탈퇴

```
/**
 * Req :: Param(key?:string)
 * Route Parameter, Pass Parameter
 * 동적인 경로 핸들링
 * http://localhost:3000/param1/1/param2/2
 */
@Get('/param1/:param1/param2/:param2')
requestParam(
  @Param('param1') param1: number,
  @Param('param2') param2: number,
) {
  return `this is request param ${param1},${param2} `;
}
```

You, 6 seconds ago • Uncommitted changes

```
/**
```

Nest :: Controller

@Body()

- 클라이언트에서 보낸 데이터 덩어리(Payload)를 처리할 때 사용
- Data handling
 - e.g.) PUT /users/1 | {age:24, role:1} update 정보
 - e.g.) POST /users | { name : “Seokmin, age: 27, role : 0 }
- DTO 의 필요성
- Validation 적용 가능

REST API 명세

METHOD	URI	Params	Body	Description
GET	/users			전체 사용자 조회
GET	/users/{id}	id		사용자의 ID를 이용해서 특정 사용자 조회
PUT	/users/{id}	id	age, role	특정한 ID를 가진 사용자의 정보를 body에 전달한 age와 role에 따라서 업데이트
DELETE	/users/{id}	id		특정한 ID를 가진 사용자의 회원 탈퇴

Nest :: Controller

DTO 란 ?

- <https://docs.nestjs.com/techniques/validation>
- Npm I —save class-validator class-transformer
- DTO
 - 확장성 있는 코드
 - 코드의 재 사용성을 높여준다.
 - 필요한 데이터 형식에 대한 추상화가 가능
- Validation
 - 해당 요청에 대해서 검증을 할 수 있다.
 - 적절한 Error Handling / Customizing 가능

REST API 명세

METHOD	URI	Params	Body	Description
GET	/users			전체 사용자 조회
GET	/users/{id}	id		사용자의 ID를 이용해서 특정 사용자 조회
PUT	/users/{id}	id	age, role	특정한 ID를 가진 사용자의 정보를 body에 전달한 age와 role에 따라서 업데이트
DELETE	/users/{id}	id		특정한 ID를 가진 사용자의 회원 탈퇴

```
27
28   @Post()
29   createUser(@Body() newUser) {
30     return this.userService.createUser(newUser);
31   }
32
33   @Delete('/:id')
34   deleteUser(@Param('id') id: number): boolean {
35     return this.userService.deleteUser(id);
36   }
37
38   @Put('/:id')
39   updateUser(
40     @Param('id') id: number,
41     @Body() updateInfo: { role: number; age: number },
42   ): boolean {
43     return this.userService.updateUser(id, updateInfo);
44   }
45 }
46
```

Nest :: Controller

@Query()

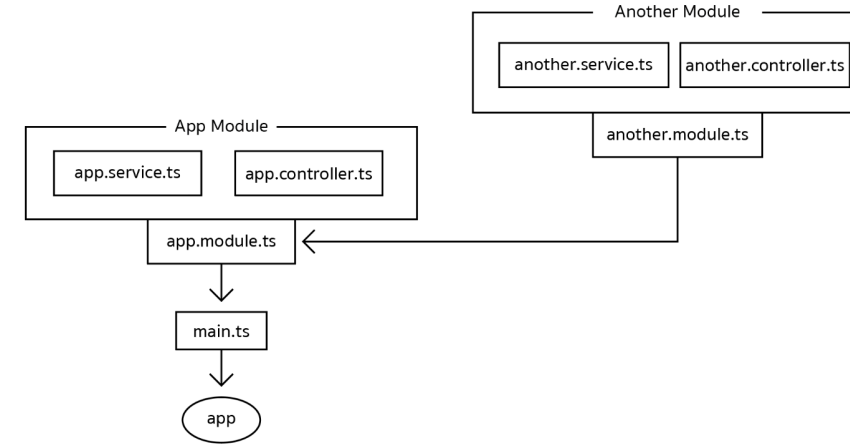
- 검색을 위한 데이터
- Paging Option :: Offset, limit
- 특정 리소스를 가져올때 Query를 이용하여 Filtering 가능
- DTO 의 필요성

```
fetchUsersByQuery(offset: number, limit: number) {  
  const fetchUsers = [];  
  for (let i = offset; i < offset + limit; i++) {  
    fetchUsers.push(this.users[i]);  
  }  
  return fetchUsers;  
}
```

You, 2 seconds ago • Uncommitt

Nest :: Provider

핵심 로직을 구현하는 부분



- Controller 는 클라이언트(OR Frontend)의 요청에 대해서 응답을 가공하고 처리하는 역할
- “서버가 제공하는 핵심기능 == 전달받은 요청을 어떻게 비즈니스 로직으로 해결하는가”
 - 간단한 인터페이스에 대한 클라이언트의 요청 (REST API ==> Controller)
 - 하지만, 내부적으로는 핵심기능의 복잡한 비즈니스 로직을 수행 (Business Logic ==> Provider)
 - MSA 아키텍처를 통한, AI Prediction 이 추가될 수도 있겠고, 결제 시스템이 추가될 수도 있고...
 - 이후 로직이 완료되면 적절한 응답 (Response ==> Controller)

Nest :: Provider 속 Design Pattern

SRP 와 Dependency Injection

- SOLID 원칙 : 객체 지향 프로그래밍 설계 원칙
 - SRP (Single Responsibility Principle)
 - DIP (Dependence Inversion Principle)
- Dependency Injection
 - @Injectable()

```
class Dog {  
  
    int leg = 4;  
    Person master;  
  
    public Dog(Person person) {  
        this.master = person;  
    }  
  
    public void run() {  
        person.runWith(this);  
    }  
  
    public void bark() {  
        person.makeBark(this);  
    }  
  
}
```


Nest :: Provider

Dependency Injection

You, 35 minutes ago | 1 author (You)

```
@Controller('users')
export class UsersController {
  constructor(private readonly userService: UserService) {}
```

- Service, Repository (DB), Factory (Factory Design Pattern), Helper ...
- 의존성 주입
 - 객체를 생성하고 사용할 때, 관심사 분리 ==> 코드 가독성과 재사용성을 높인다.
- Spring Bean
- 등록과 주입
 - 생성자 기반 주입
 - 속성 기반 주입

You, 2 hours ago | 1 author (You)

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

Nest :: Provider

Provider 처리 과정

Middleware -> Guards -> Pipes -> Handler -> Exception Filters

↑
Interceptor

↑
Interceptor

- Guard : 인증
- Pipes : 데이터 필터링, 전처리, Validation
- Handler(Service) : 비즈니스 로직
- Exception Filters : 예외처리
- Interceptor : Service 접근 전후 Request, Response 처리
- Middleware == Express middle ware
 - Logging, Cookie Parser...

Nest :: Module

중심부

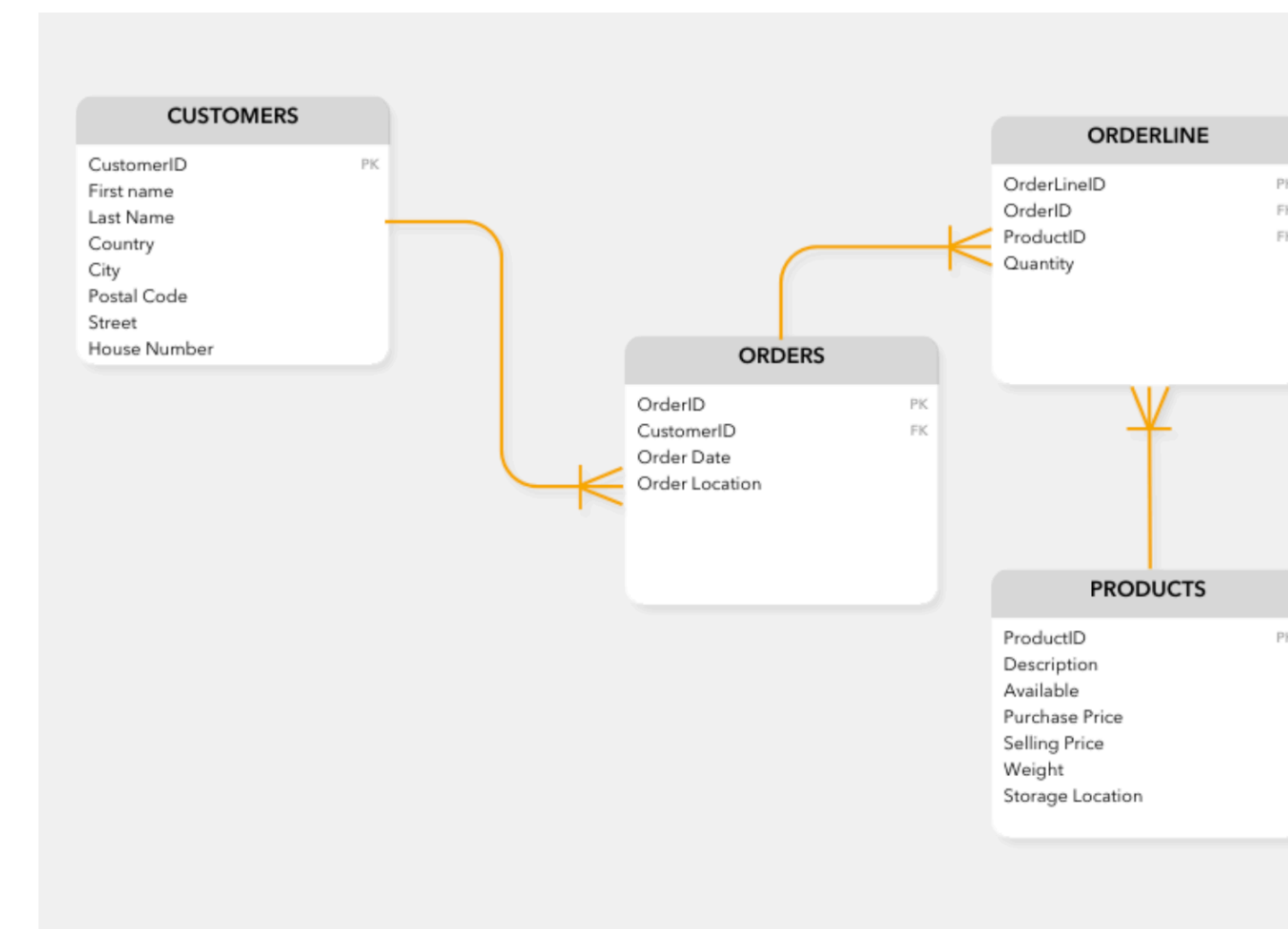
You, 3 hours ago | 1 author (You)

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

- Provider와 Controller 결합
- 다른 모듈이 내부 Provider를 사용할 수 있도록 export
- 필요한 Provider를 import 해서 모듈 내부에서 사용할 수 있도록 함
- 여러 모듈을 엮어 하나의 서비스를 만드는 것이 NestJS에서 추구하는 아키텍처

DB :: ERD

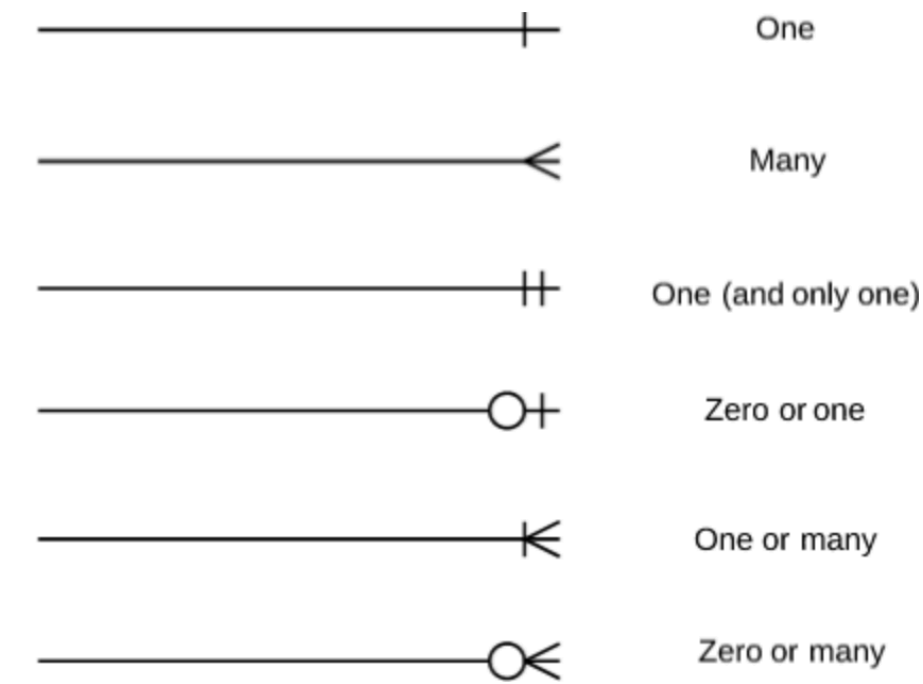
Entity Relationship Diagram



- 테이블(Relation) 간의 관계를 설명해주는 다이어그램
- Backend 를 개발하기에 앞서 필요한 도메인이 어떤 것인지 먼저 제대로 설계해야 이후에 Database Migration, 데이터 중복, 순환 참조, Relation Attribute 수정 등을 방지할 수 있다.
 - 도메인 : 장바구니, 유저, 주문정보 ...
 - draw.io 사용
 - 꺾어보면 알겠지만, 머리 박다가 문제 상황에 의해서 ERD 를 수정하는 것은 꽤나 리소스가 많이 들어가는 곤혹스러운 일
 - 프로젝트에 사용되는 DB의 구조를 한 눈에 볼 수 있다.
 - API 를 효과적으로 설계하기 위한 모델 구조도

DB :: ERD

Entity Relation Diagram



- Entity -> Class , Attribute -> Column
- User Entry 설계
 - { id, name, age, role, posts}
- Post Entity 설계
 - { id(PK), title, description, author, tag, createDate}
 - 이것을 기반으로 TypeORM 을 이용해서 Object Relation Mapping 을 하게될 것이다.
- Relation 은 어떻게 해야할까 ?
 - One to one? One to many? Many to many ?

DB :: Database Normalization

Database Normalization

- 관계형 데이터베이스에서 중복을 최소화하기 위해 데이터를 구조화하는 작업
 - 나쁜 Relation 을 좋은 Relation 으로
 - 테이블 간의 데이터 중복을 방지
 - 데이터 무결성 유지
 - DB 의 저장 공간 낭비 방지
- Insertion Anomalies : 원하지 않는 자료 삽입, 자료부족으로 삽입 불가
- Deletion Anomalies : 하나의 자료를 삭제했지만 연관된 자료 삭제
- Modification Anomalies : 연관된 정보가 한 곳에서만 수정되어 데이터 일관성 파괴

DB :: Database Normalization

Database Normalization

- 함수 종속성 (Functional Dependency)
 - Attribute 간의 함수적 종속성
 - X의 값이 Y의 값을 유일하게 결정한다면, X는 Y를 함수적으로 결정한다.
 - X : 결정자, Y : 의존자
- 정규화
 - 분해 후 무손실 조인을 보장해야한다
 - 분해 후 함수적 종속성을 보존해야한다.

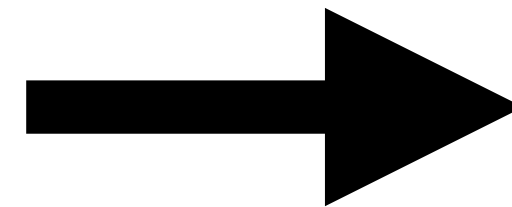
DB :: Database Normalization

Database Normalization

- 제 1 정규화 (Atomic Value) : 1NF
 - Attribute가 도메인의 오직 하나의 값만을 갖도록, 즉 복합 attribute 제거

고객취미들(이름, 취미들)

이름	취미들
김연아	인터넷
추신수	영화, 음악
박세리	음악, 쇼핑
장미란	음악
박지성	게임



고객취미(이름, 취미)

이름	취미
김연아	인터넷
추신수	영화
추신수	음악
박세리	음악
박세리	쇼핑
장미란	음악
박지성	게임

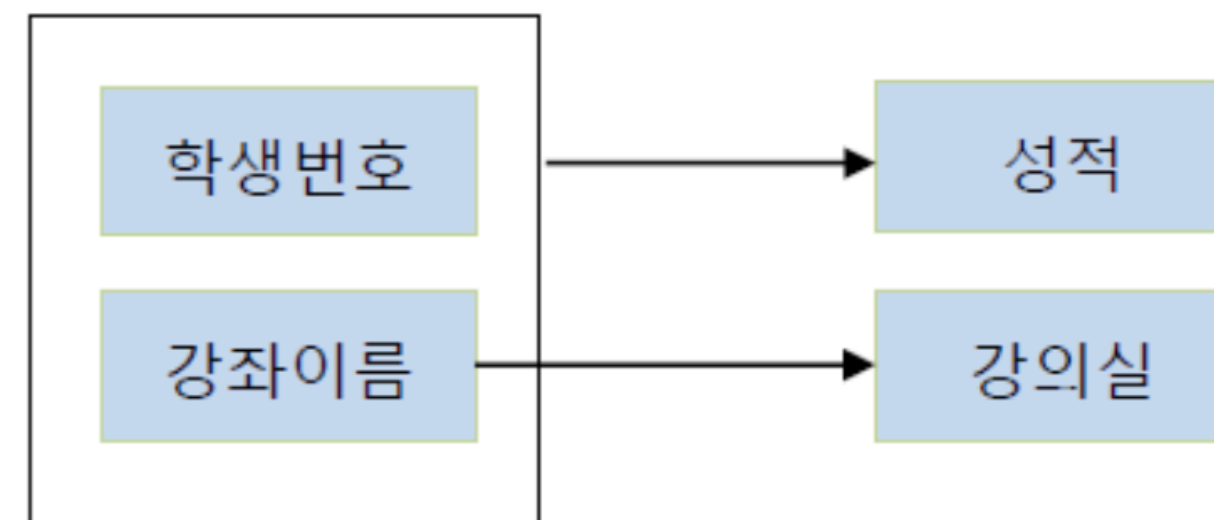
DB :: Database Normalization

Database Normalization

- 제 2 정규화(2NF) = 제 1 정규화 + **완전 함수 종속**을 만족하도록 분해
 - **완전 함수 종속** : 기본키의 부분집합이 결정자가 되어선 안된다.
 - Attribute가 도메인의 오직 하나의 값만을 갖도록, 즉 복합 attribute 제거

수강강좌

학생번호	강좌이름	강의실	성적
501	데이터베이스	공학관 110	3.5
401	데이터베이스	공학관 110	4.0
402	스포츠경영학	체육관 103	3.5
502	자료구조	공학관 111	4.0
501	자료구조	공학관 111	3.5



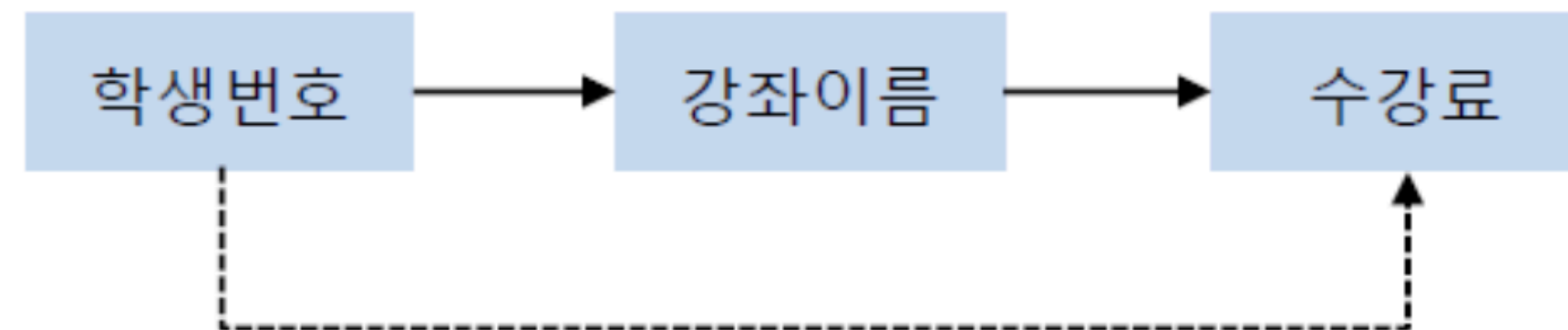
DB :: Database Normalization

Database Normalization

- 제 3 정규화(3NF) = 제 2 정규화 + **이행적 종속** 제거
- 이행적 종속** : A-> B 와 B->C 가 성립 할때, A->C 가 성립하는 것

계절학기

학생번호	강좌이름	수강료
501	데이터베이스	20000
401	데이터베이스	20000
402	스포츠경영학	15000
502	자료구조	25000



- 결과는 ?

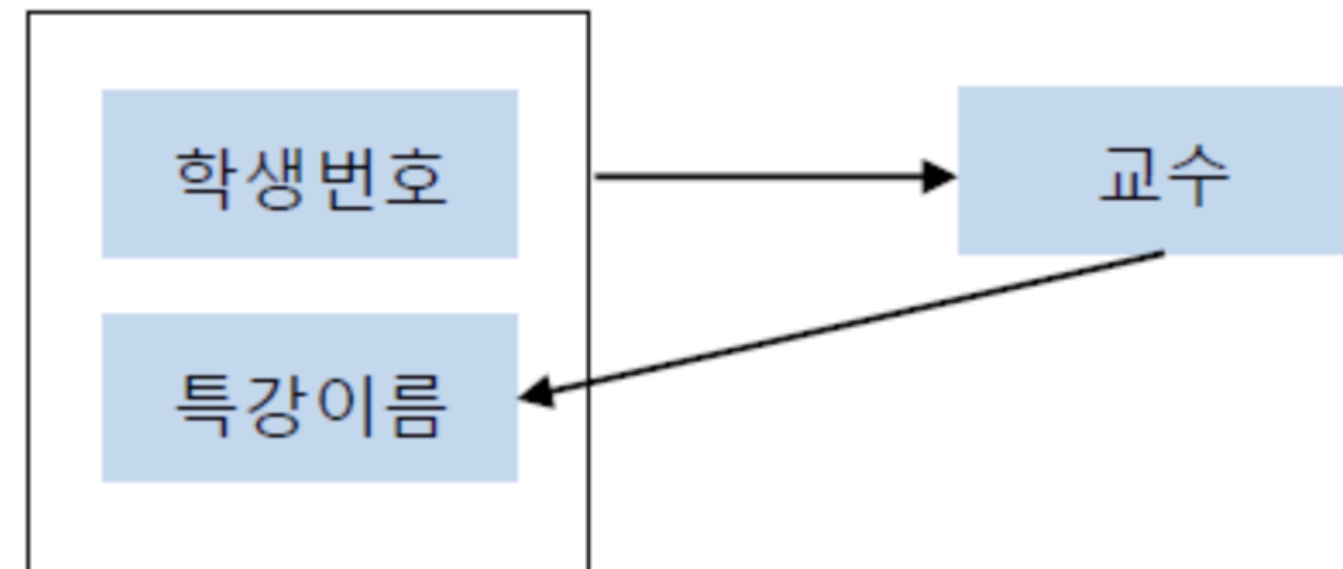
DB :: Database Normalization

Database Normalization

- BCNF 정규화 : 제 3 정규화 + 모든 결정자가 후보키가 되도록 테이블 분해

특강수강

학생번호	특강이름	교수
501	소셜네트워크	김교수
401	소셜네트워크	김교수
402	인간과 동물	승교수
502	창업전략	박교수
501	창업전략	홍교수



- 결과는 ?

DB :: Database Normalization

Database Normalization

- 정규화의 장점
 - 이상 현상 방지
 - 데이터베이스 구조 확장성 증가
- 정규화의 단점
 - Relation의 분해로 조인 연산이 많아질 수 있다.
 - 따라서 Query에 대한 응답 시간이 느려질 수 있다.
 - 반정규화 ?:

DB :: ORM

Object Relational Mapping

- 객체와 관계형 데이터 베이스의 데이터를 자동으로 매핑해주는 것을 말한다.
- 객체 지향 프로그래밍의 경우
 - 클래스와 관계형 데이터베이스의 테이블이 매핑된다.
 - ORM 을 통해서 객체 간의 관계를 바탕으로 적절한 Query를 생성하여 RDBMS에 접근한다.
 - Spring :: JPA
 - Spring :: Hibernate
 - Nest :: TypeORM

DB :: ORM

Object Relational Mapping

- 장점

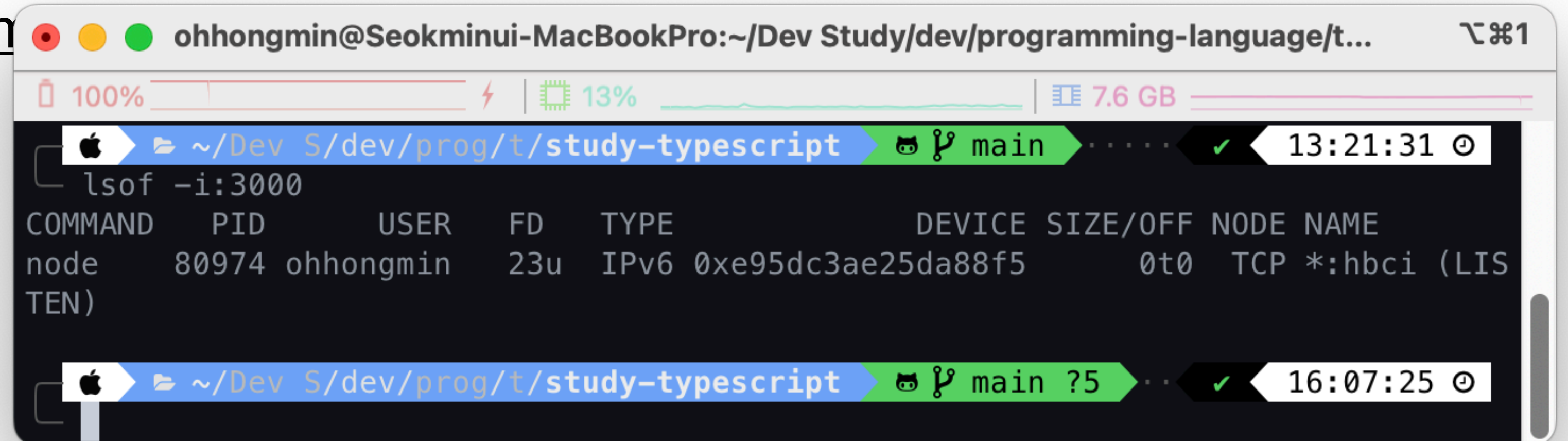
- 객체 지향적인 코드로 비즈니스 로직 구현에 더 집중할 수 있다.
- 메서드로 SQL Query를 이용할 수 있다.
- 커넥션 할당, close 등 부수적인 코드가 급격히 줄어든다.
- 객체 지향적으로 SQL을 접근하므로 생산성이 증가한다.

- 단점

- ORM으로만 완벽하게 SQL 을 다룰수는 없다.
- 잘못 구현하면 데이터의 일관성을 해치는 문제점이 생길 수 있다.
- 대형 쿼리는 속도를 위해서 SP를 쓰는 등 별도의 튜닝이 필요하다.

DB :: Postgres ORM

- Brew Install
 - <https://someone-life.tistory.com/90>
- Brew services start postgresql
- PgAdmin4
 - <https://mseagle.tistory.com>
- \du
- \dt
- 사용 중인 포트 확인



ohhongmin@Seokminui-MacBookPro:~/Dev Study/dev/programming-language/t... ⌵⌘1

100% | 13% | 7.6 GB

```
~/Dev S/dev/prog/t/study-typescript main 13:21:31
lsof -i:3000
COMMAND  PID    USER   FD   TYPE    DEVICE  SIZE/OFF NODE NAME
node     80974 ohhongmin 23u  IPv6  0xe95dc3ae25da88f5      0t0  TCP *:hbc (LISTEN)

~/Dev S/dev/prog/t/study-typescript main ?5 16:07:25
```

Assignment 3 : 본격적으로 개발 시작 TypeORM 연결 ORM

- Reference
 - Main : <https://docs.nestjs.com/techniques/database>
 - Sub : <https://typeorm.io/>
- NestJs에서 TypeORM을 이용해서 Postgres를 연결해보기
- In-memory 배열로 다뤘던 User 정보를 이제 Postgres DB로 넣어본다.
 - HINT : postgres connection string url
- User Entity 를 생성하고 기존의 In-memory User 배열을 database 로 옮긴다.
- User Entity(Relation)을 제어하는 Repository 를 이용해서 Get, Post, Put, Delete 를 업그레이드 해보자.
- 꽤 어려울 수 있으니 질문은 환영

Assignment 3 : 본격적으로 개발 시작 TypeORM 연결 ORM

- 최종 목표 :: 유저 서비스 + 유저별 게시물 생성 및 관리 + **a + b + c + d + e + f ...**
- **GraphQL, Socket, Redis, MongoDB, ...**
 - 회원 가입
 - 회원 가입화면을 통해 유저 정보를 입력받아 유저 생성 요청을 받습니다.
 - 우리는 프론트엔드에서 유저 생성과 관련한 데이터를 전달해 준다고 가정
 - 생성된 유저 정보는 가입 대기
 - 이메일 인증
 - 서버 응답에는 다시 이메일 검증을 위한 요청으로의 링크가 포함되어 있습니다.
 - 사용자가 이 링크를 누르면 이메일 승인 요청이 들어오게 되고 회원 가입 준비 단계에 있는 유저를 승인합니다.
 - 로그인
 - 회원 정보 조회 기능

Reference

- ERD : <https://velog.io/@kjhxxxx/DataBase-ERD%EB%9E%80>
- NestJS : <https://wikidocs.net/book/7059>
- Database 정규화 : <https://mangkyu.tistory.com/110>
- TypeORM : <https://typeorm.io/>

Reference

- 다음주는... ?