퀴즈 웹앱 보고서

컴퓨터소프트웨어공학과
20243516 이승희
파이썬프로그래밍 성낙준교수님

들어가기에 앞서

코드의 실행 방법을 서술해두겠습니다.

app폴더 하위가 nextjs의 코드입니다.
먼저 터미널을 app으로 두고 모듈을 받아주세요.

$ npm install

그 다음 웹 서버 실행을 위해 프로젝트를 build해야 합니다.

$ npm run build

이제 실행 가능합니다.

$ npm start

프론트는 단지 프론트 일뿐 기능을 가지고 있지 않습니다. 벡엔드를 켜주셔야 기능이 동작합니ㅏㄷ.

server 폴더 하위는 Nestjs입니다. 벡엔드.
동일하게 모듈을 받아주세요

$ npm install

다음으로 build과정을 해주셔야 합니다.

$ npm run build

이제 실행

$ npm start

본 프로젝트는 프론트서버와 벡서버 둘다 켜져있어야 정상 작동하게 됩니다.

만약 모종의 이유로 데이터베이스에 문제가 생기면 마이그레이트와 db를 지워주시고 아래의 명령어를 실행해 주시면 자동으로 스키마 설정부터 재 생성됩니다.

$ npx prisma migrate dev --name init

감사합니다.

계획

요구사항에 맞게 구현하는 것은 어렵게 느껴지지 않았지만, 추가 기능으로 어떠한 기능을 구현할지 고민하였습니다. 난이도에 따라 점수를 배분한다 하셨기에 제가 생각하는 가장 다능한 기능을 넣으려 하고 있습니다.

자신이 원하는 퀴즈를 선택하여 풀 수 있게 할 계획이고, 댓글 기능과 통계 기능, 원하는 갯수 만큼 과제의 조건을 충족하기 위해 원하는 갯수 선택해 랜덤하게 문제를 출제하는 기능을 구현할 계획입니다.

구현에 들어가기에 앞서 가장 걱정되는 부분은 통계 기능과 퀴즈 갯수 선택 기능입니다. 퀴즈의 갯수를 선택할 수 있다면 갯수에 따라 통계가 합산되기 어렵다는 점이 존제합니다. 현재로서 생각되는 해결 방법은 갯수를 선택해서 문제를 풀시 통계 합산은 되지 않는 다는것을 공지하고 시작하는것이 좋다고 생각됩니다.

HTML과 Nest.js를 이용해 풀스택단을 구현할 생각이였지만 React를 사용해도 된다는 허락을 받았기에 Nextjs로 통합하기로 하였습니다... 라고 방금 생각하였지만, Next.js 프레임워크는 html js css로 build 절차를 실행하긴 하지만, 백엔드와 통합되어 있는 만큼 일반적인 html js css로 추출에는 한계가 있어 백엔드를 분리해줄 필요가 있었습니다. 이를 위해 Next.js의 static export 기능과 Nest.js를 사용해 프론트와 백엔드를 완벽히 분리할 계획입니다.

Nextjs는 버전이 많습니다. 버전이 업데이트 될때마다 문법이 완전 달라지는게 Next.js의 단점이라고 생각됩니다. 현재 최신버전인 14버전은 앞서 말한듯 많은 문법이 달라졌고, 제가 주로 사용하는 버전은 12임으로 편의성을 위해 익숙한 12버전을 사용하기로 하였습니다.

구현 시작

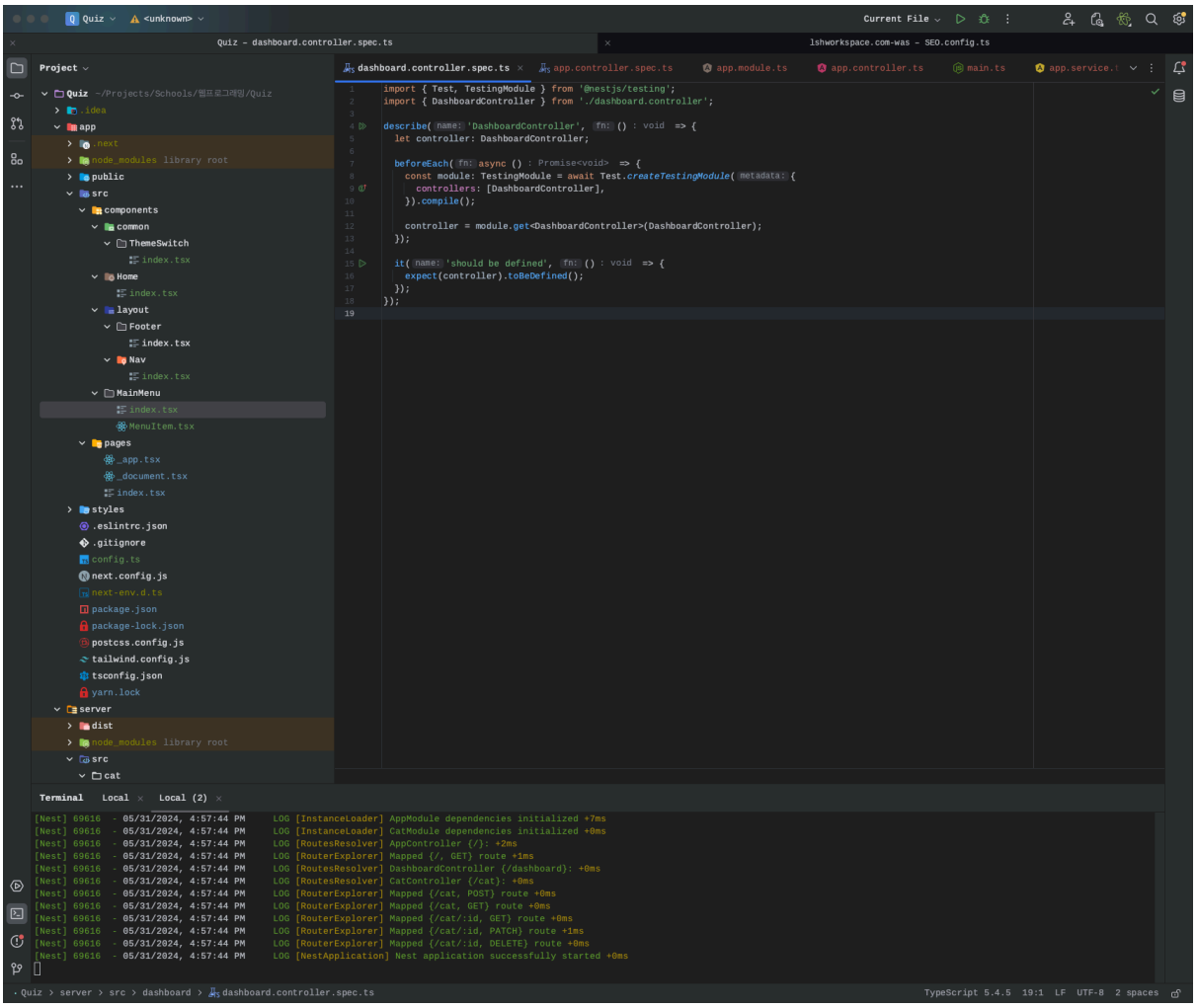
먼저 프로젝트 파일을 만들고 쉬운 버전관리를 위해 프로젝트를 git 감시하에 두었습니다.

```
> git init
```

다음 Nextjs 프로젝트 파일을 쉽게 생성해 주는 스크립트를 실행하여 12 버전의 프로젝트 파일을 만들어 주었습니다.

```
> npx create-next-app@12.2.6
```

스크립트를 통해 프로젝트를 만들게 되면 기본적으로 파일 구성을 생성해 주게 됩니다. 저는 개인적으로 파일 구성 스타일이 존재해, 저에게 맞게 구성 작업을 먼저 해주었습니다.



(깜빡하고 보고서 쓰면서 하는걸 잊어서 파일 구성이 백엔드까지 어느정도 되버림)

src 내부에 개발할때 주로 만지는 리소스를 담아 두고, 하위로 components에 컴포넌트를 넣습니다. 레이아웃에 관여하는 파일은 compoents/layout에 지정하고 주 페이지가 될 부분은 페이지 역할과 함께 componets/[페이지역할]로 지정합니다. 모든 컴포넌트 하위 페이지의 코드는 index가 컨트롤 역할을 하여 모듈을 분리합니다. 만약 분리될 모듈이 없다면 index내부에 넣게 됩니다. 이러한 디자인을 사용하는 이유는 비슷한 역할을 구성할때 파일 구성없이 모듈형식으로 구성할 수 있게 하기 위함입니다. 단점으로는 첫 구성에 많은 계획을 해야하지만, 이후 얻는 유지보수와 같은 이점이 크기에 저는 이러한 디자인을 추구하는 편 입니다. 재사용이 빈번한 디자인은 components/common에서 관리합니다. button이나 여러 재사용이 빈번한 디자인을 분리해서 빠르게 찾을 수 있다는 장점이 있습니다.

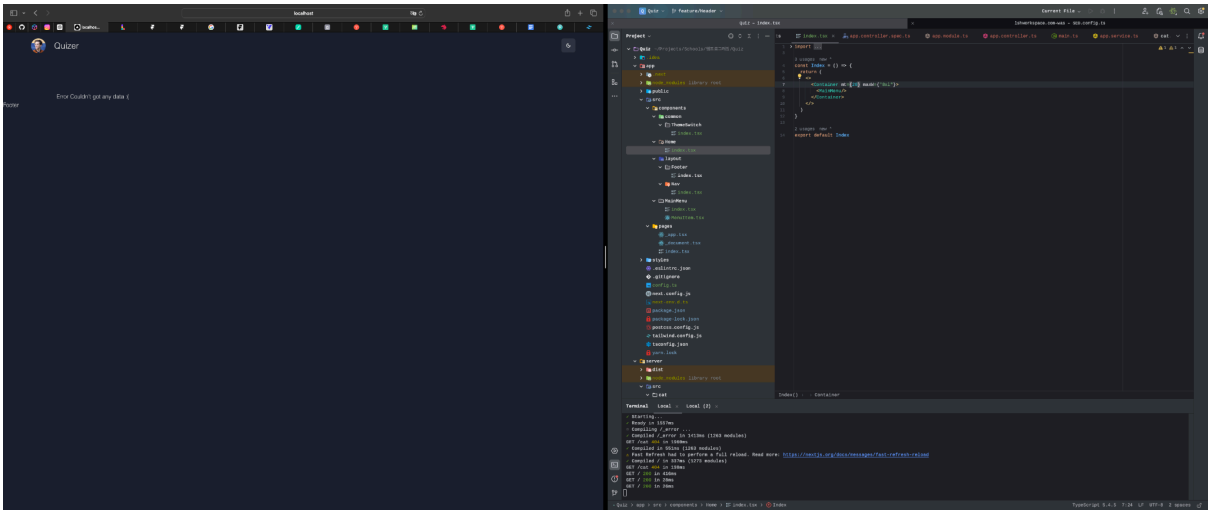
웹 디자인을 어떤 것으로 할까 고민했습니다. 현 프로젝트를 진행할때 고려 해야 하는 점은 물리적인 시간을 많이 쓰지 않는 것이였습니다. 프로젝트 시작 기점과 끝 기점에 시험기간이 겹쳐 있기 때문에 시간 분배에 신중해야 했습니다. 그렇기에 디자인에

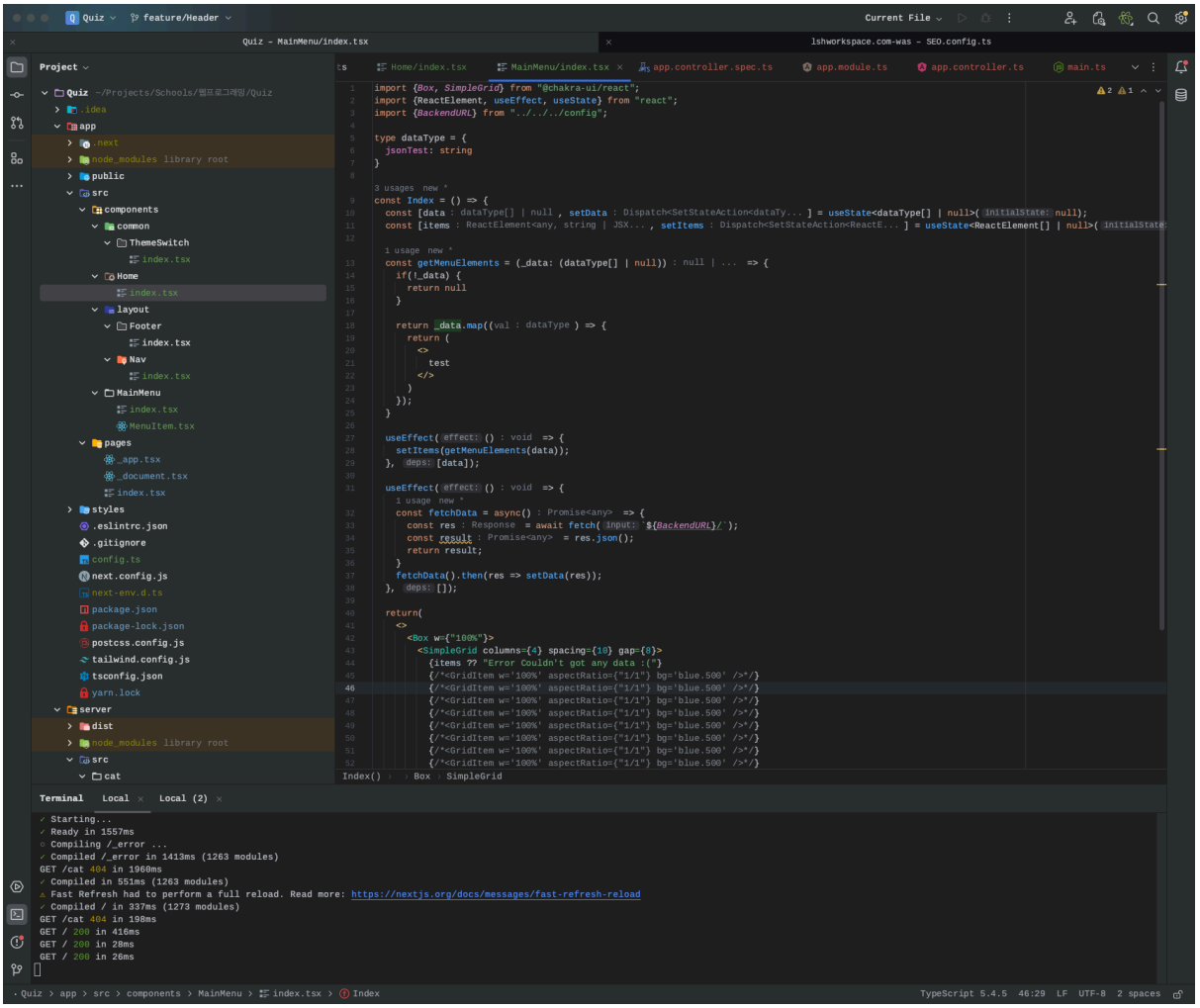
시간을 많이 쏟으면 오버될 것을 예상하여 Chakra-ui + Tailwindcss를 사용하기로 하였습니다. Chakra-ui는 ui라이브러리로 구성되어 있는 여러 컴포넌트를 사용할 수 있고 여러 디자인이 미리 정의되어 있는 편한 라이브러리입니다. 단점으로는 framer-motion기반이라, 느리다. 라는 단점이 하나 존재합니다. 하지만 단점을 고려하고도 시간 절약에 탁월하다고 생각되기에 Chakra-ui를 이용하기로 하였습니다.

구성은 간단합니다.

리엑트 기반이 되는 거의 대부분의 대형 프레임워크와 라이브러리를 지원하고 공식 Doc이 있기 때문에 쉽게 구성할 수 있었습니다.
(Chakra-ui installation)[https://v2.chakra-ui.com/getting-started]

Tailwind또한 install script가 존재하여 이를 활용해 주었습니다.
(Tailwindcss installation)[https://tailwindcss.com/docs/installation]





어느정도 디자인과 코드를 작성하고 API 구성을 위해 서버단으로 넘어갔습니다.

Nestjs로 서버를 구성하였고, 데이터베이스와 ORM을 고민하게 되었습니다.
json 형식으로 데이터를 쓰고 읽을 예정이라 JSON에 강한 mongoDB를 사용하려 했습니다.
하지만 mongoDB를 사용하게 되면 과제 책점에서 데이터베이스를 만들어 주거나 컨테이너 환경을 실행해야 하는데, 이 부분에서 어려움이 생길 것 같아, Nodejs과 로컬 환경으로 구동이 가능한 sqlite 사용으로 변경하였습니다. sqlite는 로컬에서 디비를 생성하여 사용할 수 있고 빠르다는 장점이 있는 데이터베이스 입니다.
데이터베이스는 이렇게 sqlite를 사용하기로 하였고, 다음은 ORM 결정입니다. Nestjs에서 흔이 쓰는 ORM은 TypeORM인 것처럼 보였습니다. 대부분의 자료를 확인해 보면 대부분이 TypeORM을 사용함을 알 수 있었고, 다른 ORM을 쓰는 자료는 많지 않은 것처럼 보였습니다. 개인적으로 저는 TypeORM을 좋아하지 않습니다. 타입체크를 재대로 하지 않아 컴파일단에서 에러를 표시 하지 않고 런타임 환경에서 타입 오류로 뻗어버리는 경우가 비일비제 하기 때문입니다. 저는 그렇기에 Node.js단에서는 Prisma라는 차세대 ORM을 주로 사용합니다. 스키마와 여러 설정을 다양하고 능동적으로 지원하며, 신입 주제에 인지도가 있어 자료또한 충분하여 주로 애용하는 ORM 입니다.

Nestjs에서 prismaORM를 사용하기 위한 구성

공식 문서를 찾아본 결과, Prisma를 지원하는 것을 알 수 있었습니다.
https://docs.nestjs.com/recipes/prisma

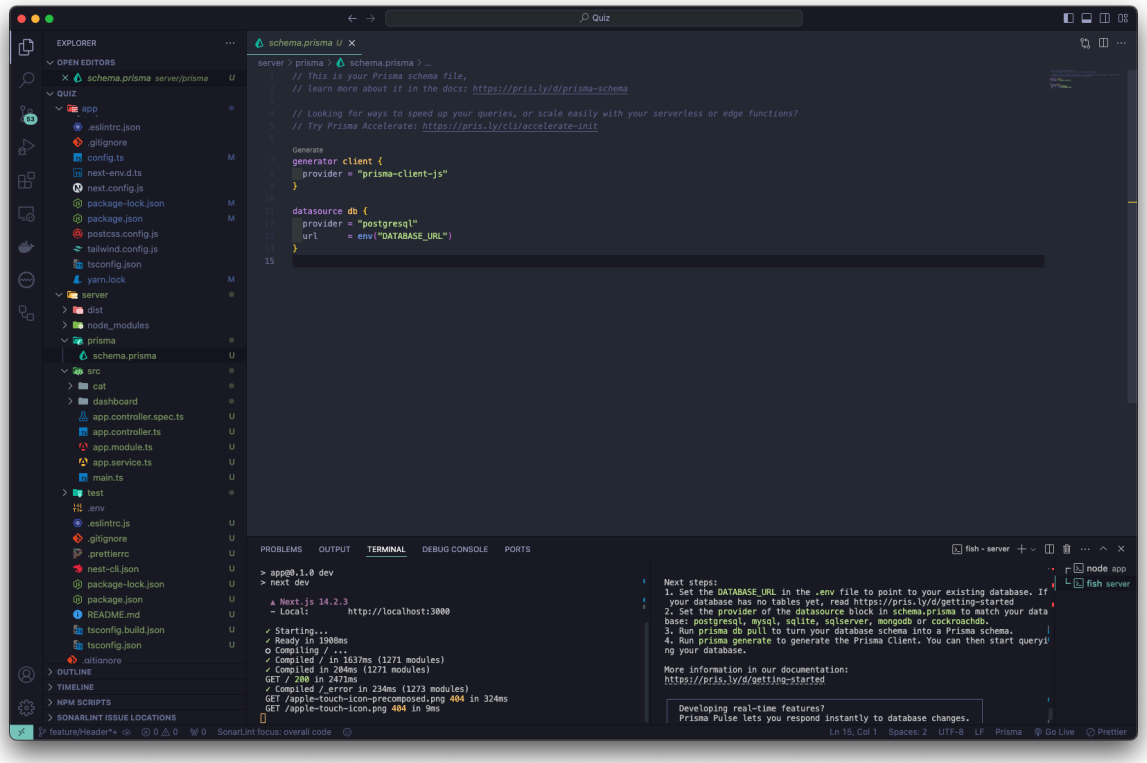
공식 문서를 천천히 읽으며 제 프로젝트와 병합 및 테스트해보겠습니다.

Prisma Example을 위한 프로젝트 생성은 하지 않았습니다.
먼저 Prisma를 인스톨해주겠습니다.

$ npm install prisma --save-dev

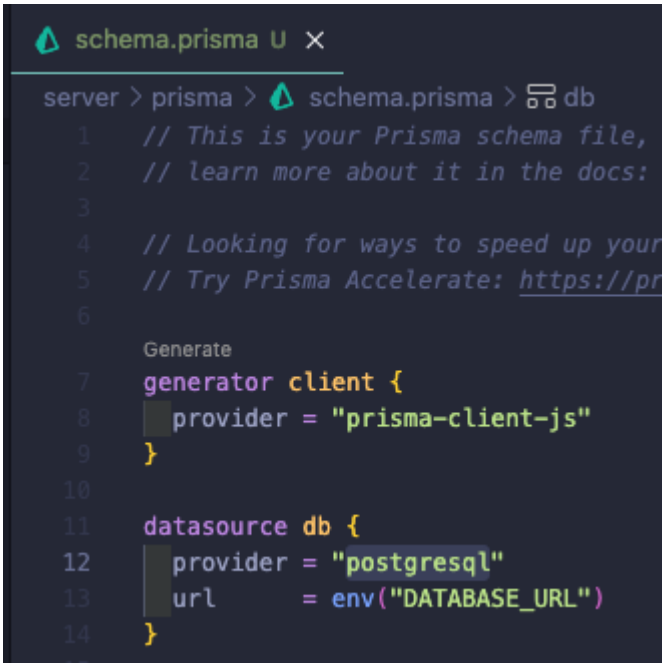
설치가 끝난후 Prisma를 설정해줄 차례입니다.
Prisma는 여러 설정을 Prisma CLI를 통해 자동으로 설정합니다. 이를 위한 기초 init명령어를 입력해주겠습니다.

$ npx prisma init

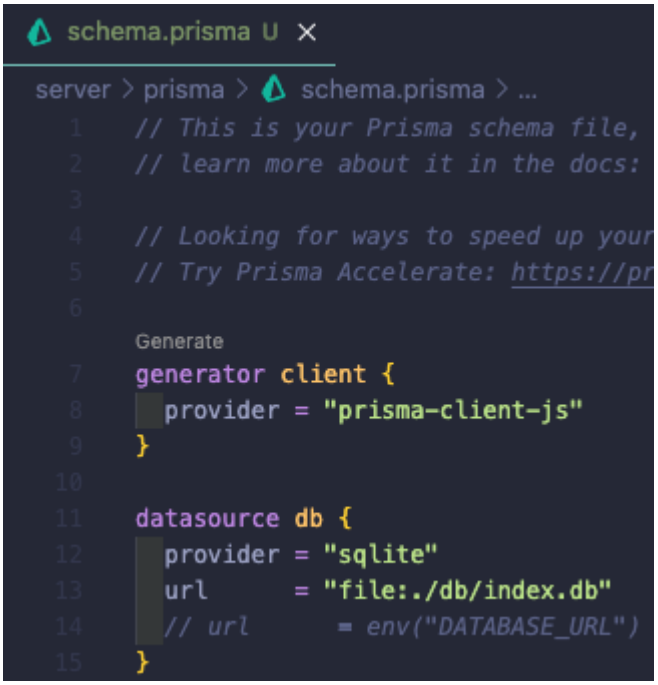


잘 적용되었습니다.

다음으로 Prisma의 기본 정보 설정을 해주겠습니다.



기본적으로 postgresql을 사용하고, 데이터베이스 URL을 env를 통해 받아오는 것을 알 수 있었습니다. 저희는 Sqlite를 사용할 예정임으로 데이터 베이스를 sqlite로 바꿔주었고, url 또한 로컬 sqlite를 사용할 예정이기에 그냥 변수에 박아주겠습니다.



다음으로 데이터 모델을 만들어 주겠습니다.
여기서 생각해야 할게 생기게 됩니다. 데이터베이스에서 저의가 담게될 것을 기획해야 되는 것이죠.

데이터베이스 모델 기획

먼저 데이터베이스에 어떤것을 담을지 생각해 봐야 합니다.

먼저 주 기능인 퀴즈를 담아야 합니다.

퀴즈
- 타이틀
- 퀴즈 설명
- 퀴즈 임배드
- 요청 수
- 퀴즈 문제 [
    - 퀴즈
    - 이미지
    - 보기 [string]
    - 정답자 수

정도가 될 수 있을 것 같습니다.
요청 수와 정답자 수는 추후 그 문제의 오답률을 계산하기 위해 수집하게 됩니다.
문제 풀이 도중 다 완료하지 않고 종료하면 통계를 집계하지 않을 생각입니다.

여기서 문제 입력때 필수 파라미터로 받지 않아도 되는 것은 이미지 정도가 될 수 있고, 받지 말아야 할것은 정답자 수, 요청수 가 되게 됩니다. 이는 서버단에서 자동적으로 처리가 되야 하고 언급하지 않은 부분에 대한 것은 필수 파라미터가 되게 됩니다. 이를 토대로 모델을 작성해 봅시다.

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = "file:./db/index.db"
  // url      = env("DATABASE_URL")
}

model post {
  id        Int      @default(autoincrement()) @id
  title     String
  des       String
  emb       String
  quiz      quiz[]   @relation("PostQuizRelation")
  createdAt DateTime @default(now())
  // author    User?    @relation(fields: [authorId], references: [id])
  // authorId  Int?
}

model quiz {
  id        Int      @default(autoincrement()) @id
  ques      String
  img       String
  ans       Json
  postId    Int
  post      post     @relation(fields: [postId], references: [id], name: "PostQuizRelation")
}
```

모델을 post와 quiz로 나눴고 모델간 관계를 설정하기 위해 역방향관계 필드를 설정하였습니다.

이제 데이터베이스를 설계하고 만들어야합니다. 이부분은 Prisma의 스크립트의 힘을 빌릴 생각입니다. Prisma는 제가 짠 스키마 코드를 보고 자동으로 데이터베이스를 구성하는 기능이 있습니다.

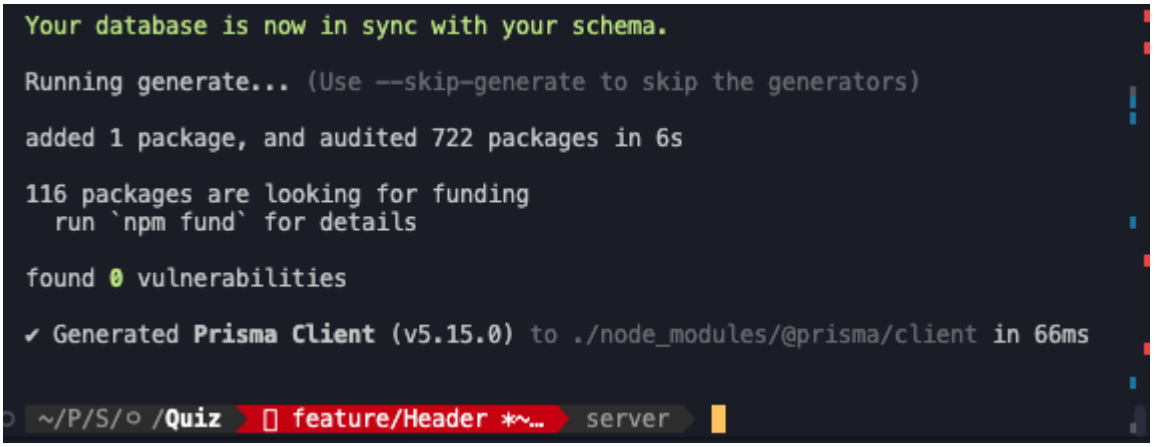
$ npx prisma migrate dev --name init

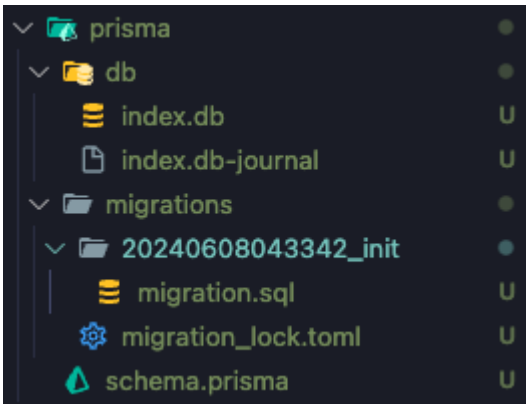
코드를 실행해 보니 문제가 하나 발생했습니다.

```
Error code: P1012
error: Error validating field `ans` in model `quiz`: Field `ans` in model `qu
iz` can't be of type Json. The current connector does not support the Json ty
pe.
  -->  prisma/schema.prisma:32
   |
31 |   img       String
32 |   ans       Json
33 |   postId    Int
   |

Validation Error Count: 1
[Context: validate]

Prisma CLI Version : 5.15.0
! ~/P/S/○ /Quiz  feature/Header *~… server
```

읽어본 바 Sqlite는 Json을 지원하지 않는 모양입니다. 덕분에 Prisma가 자동적으로 이를 인식하고 Json타입이 안된다는 것을 알려주고 있는 모습입니다. 참 좋네요. 런타임 에러만 안보는 것으로도 기적입니다.. 해결하기 위해 데이터베이스에 배열을 저장할 수 없기 때문에 배열 자체를 String으로 때려 박고 파싱하는 방법으로 우회해보겠습니다.

문제가 되는 Json타입 부분을 String으로 변경하고 명령어를 다시 실행하였습니다.



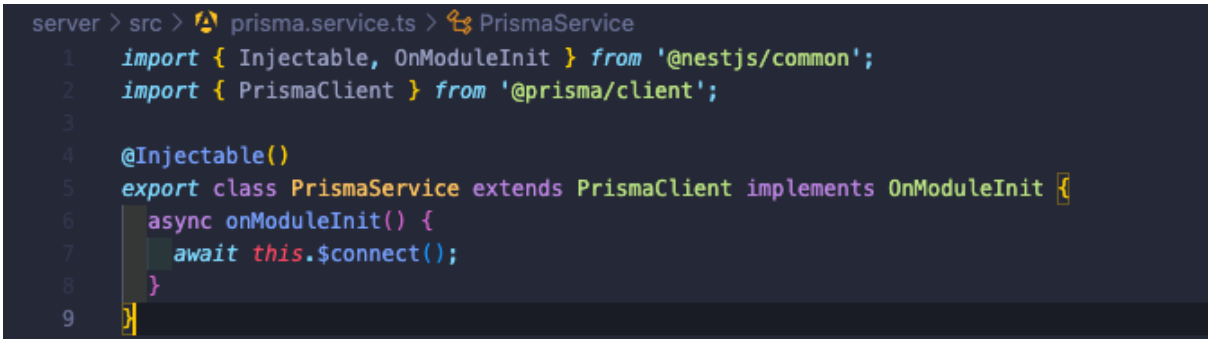


이전에는 없던 db 파일과 migrations 파일이 경로에 생성되었고 그 내부에 db가 동기화 되어있음을 알 수 있습니다.

비교적 간단하게 데이터베이스를 만들었습니다. 이제 Nestjs에 연동코드를 작성해주도록 하겠습니다.

데이터베이스 연동

먼저 연결 코드를 작성해 주기 위해 src 하위에 `user.service.ts` 파일을 하나 만들어 주겠습니다.



이제 API를 만들어 보겠습니다.

API 제작

요청을 필요로 하는 기능을 먼저 생각하고 구현해 봅시다.

먼저 대시보드 기능이 필요합니다. 어떤 주제의 문제가 있는지 확인하는 기능이 있어야 겠지요.

대시보드는 기본정보만 불러오면 될것 같습니다. post단과 퀴즈 단을 나눠둔 이유가 이 때문이죠
퀴즈를 보내는 것도 필요합니다. URL의 주소에 그 퀴즈에 대한 정보도 있어야 합니다. ID를 보내줘야 하죠. 대시보드를 구현해 봅시다.

먼저 새로운 퀴즈를 만들 수 있는 Post 요청을 만들어 봅시다.

```ts
@Controller('dashboard')
export class DashboardController {
  constructor(private readonly dashboardService: DashboardService) {}

  @Post()
  // async createPost(
  //   @Body() postData: { title: string, des: string, emb: string, quiz: { que
  // ): Promise<PostModel> {
  //   return this.dashboardService.createPost(postData)
  // }
  async createPost(@Body() postData: CreateDashboardDto): Promise<PostModel> {
    return this.dashboardService.createPost(postData);
  }

  @Get()
  findOne() {
    return "어쩔"
  }
```

컨트롤러를 작성해 주었고

```ts
  async createPost(data: CreateDashboardDto): Promise<Post> {
    const quizzes = data.quiz.map((quiz) => ({
      ques: quiz.ques,
      img: quiz.img,
      ans: JSON.stringify(quiz.ans), // JSON 문자열로 변환
    }));

    return this.prisma.post.create({
      data: {
        title: data.title,
        des: data.des,
        emb: data.emb,
        quiz: {
          create: quizzes,
        },
      },
      include: {
        quiz: true,
      },
    });
  }
}
```

타입을 지정해 서비스를 작성해 주었습니다.

```ts
import { IsString, IsArray, ValidateNested } from 'class-validator';
import { Type } from 'class-transformer';

class CreateQuizDto {
  @IsString()
  ques: string;

  @IsString()
  img: string;

  @IsString()
  ans: string; // JSON 문자열로 저장
}

export class CreateDashboardDto {
  @IsString()
  title: string;

  @IsString()
  des: string;

  @IsString()
  emb: string;

  @IsArray()
  @ValidateNested({ each: true })
  @Type(() => CreateQuizDto)
  quiz: CreateQuizDto[];
}
```

데이터 변환을 위해 DTO를 작성해 주었습니다.

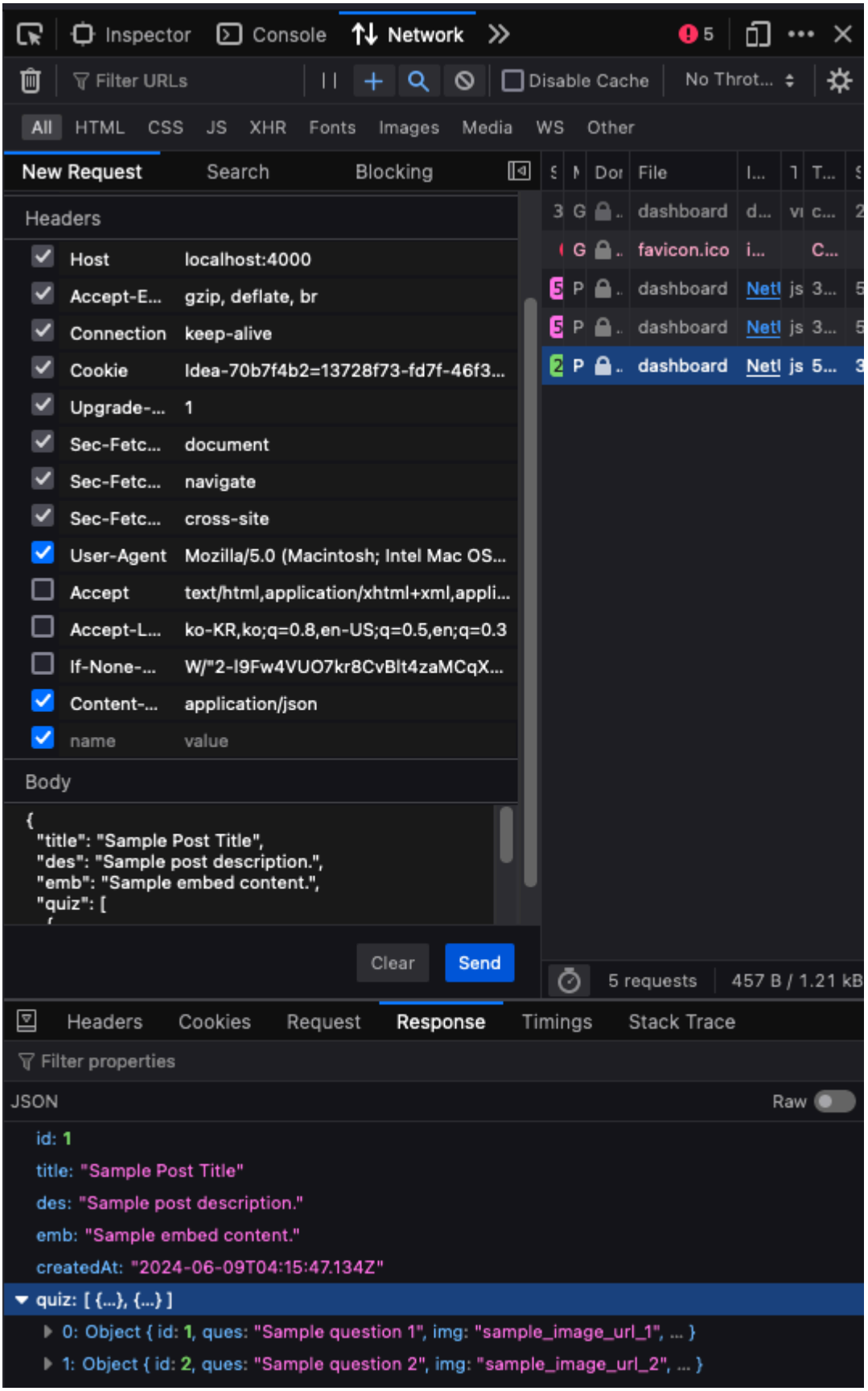
이제 Get 요청을 작성해 봅시다. 생각해보면 대시보드를 통해 모든 정보를 전달 받는것도 괜찮다고 생각됩니다. 한번 작성해 보겠습니다.

```ts
  async findAll(): Promise<Post[]> {
    return this.prisma.post.findMany({
      include: {
        quiz: true,
      },
    });
  }
}
```

간단하게 모든 정보를 가져오게 했습니다. 어디까지나 과제이기 때문에 많은 퀴즈가 생길것 이라는 것을 배제하고 캐싱이나 다른 방법을 사용하지 않았습니다.

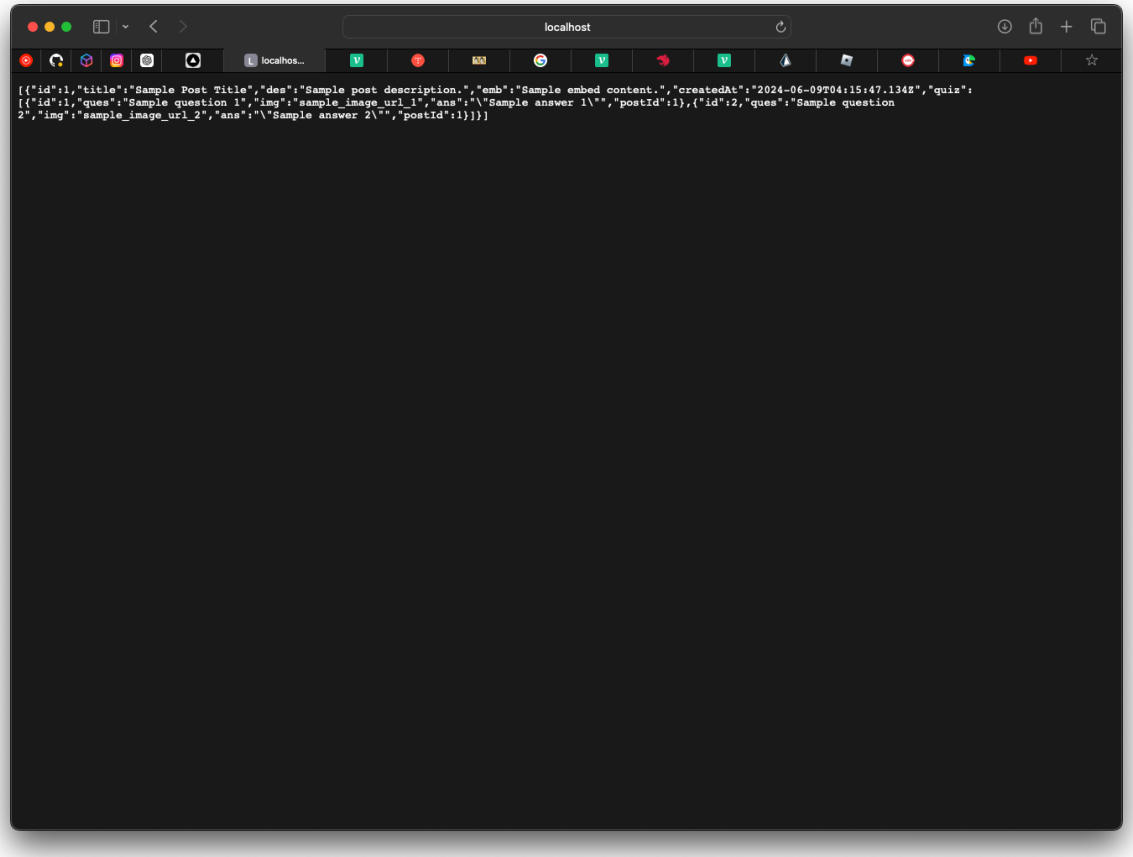
자 끝과 시작이 보이기 시작했습니다. 테스트를 위해 post 요청과 get 요청을 보내 봅시다.

백엔드 테스트



일단 post요청을 Json으로 보내본 결과 잘 응답하는 모습을 보입니다. 완벽하네요 기분이 좋습니다. 에러가 없어요 불안하고도 좋네요...

다음으로 잘 DB에 저장되고 불러와지는지 Get 요청을 보내보겠습니다.



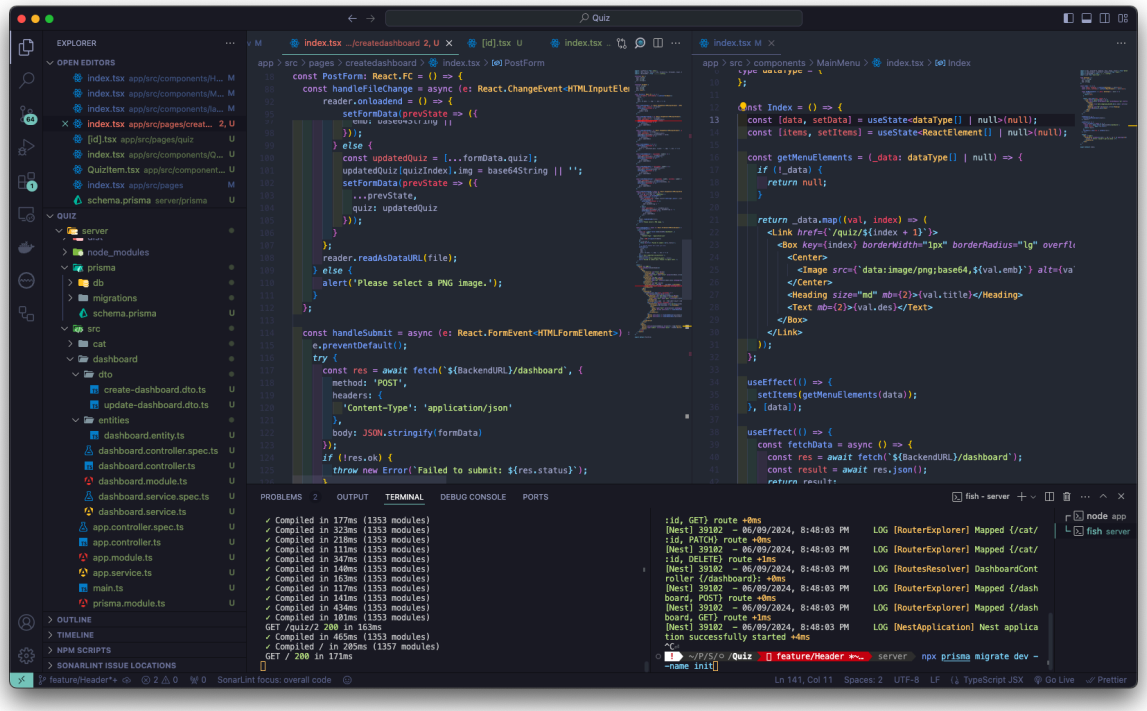
잘 불러오는 모습입니다. 이제 프론트 단을 작성해 봅시다.

먼저 Cross 문제를 해결하기 위해 한가지 설정을 더 해주겠습니다.

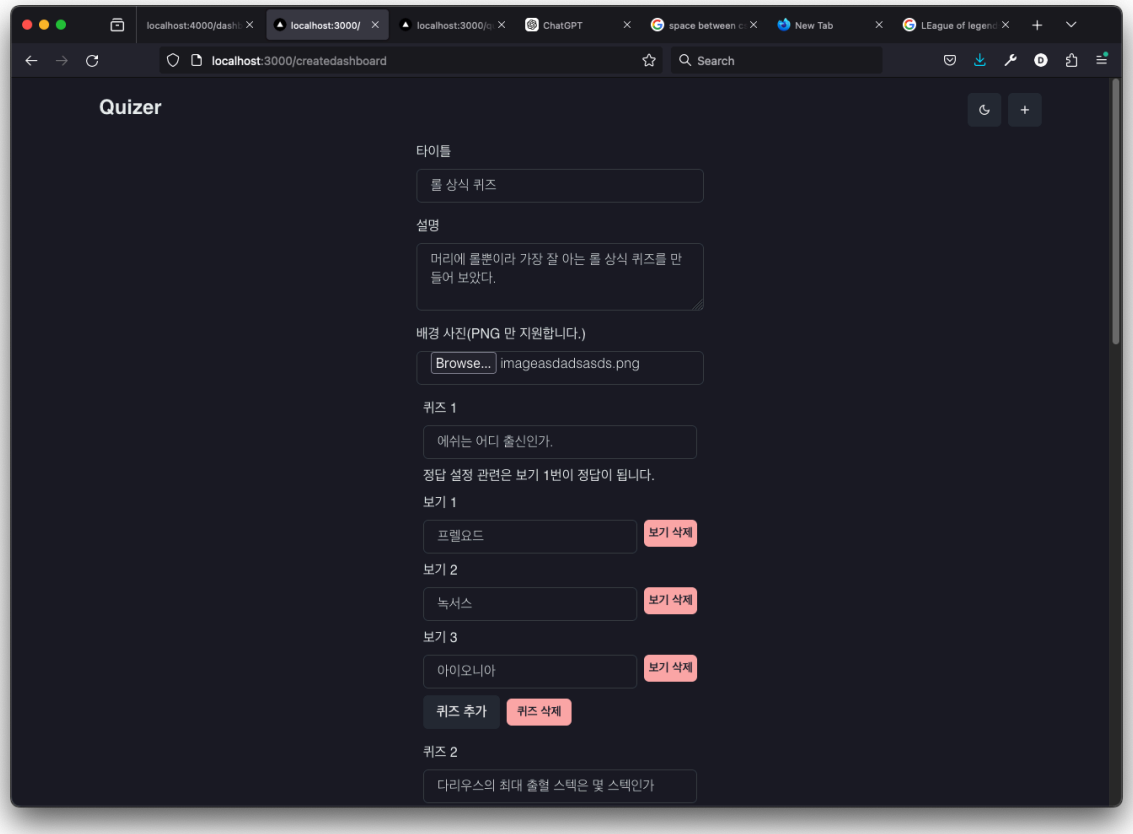
```
async function bootstrap() {
  const server = express();
  const app = await NestFactory.create(AppModule, new ExpressAdapter(server));

  // Enable CORS with specific configuration
  app.enableCors({
    origin: 'http://localhost:3000', // Next.js 클라이언트 URL로 변경
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE,OPTIONS',
    credentials: true,
  });

  await app.listen(4000);
}
```

프론트 엔드

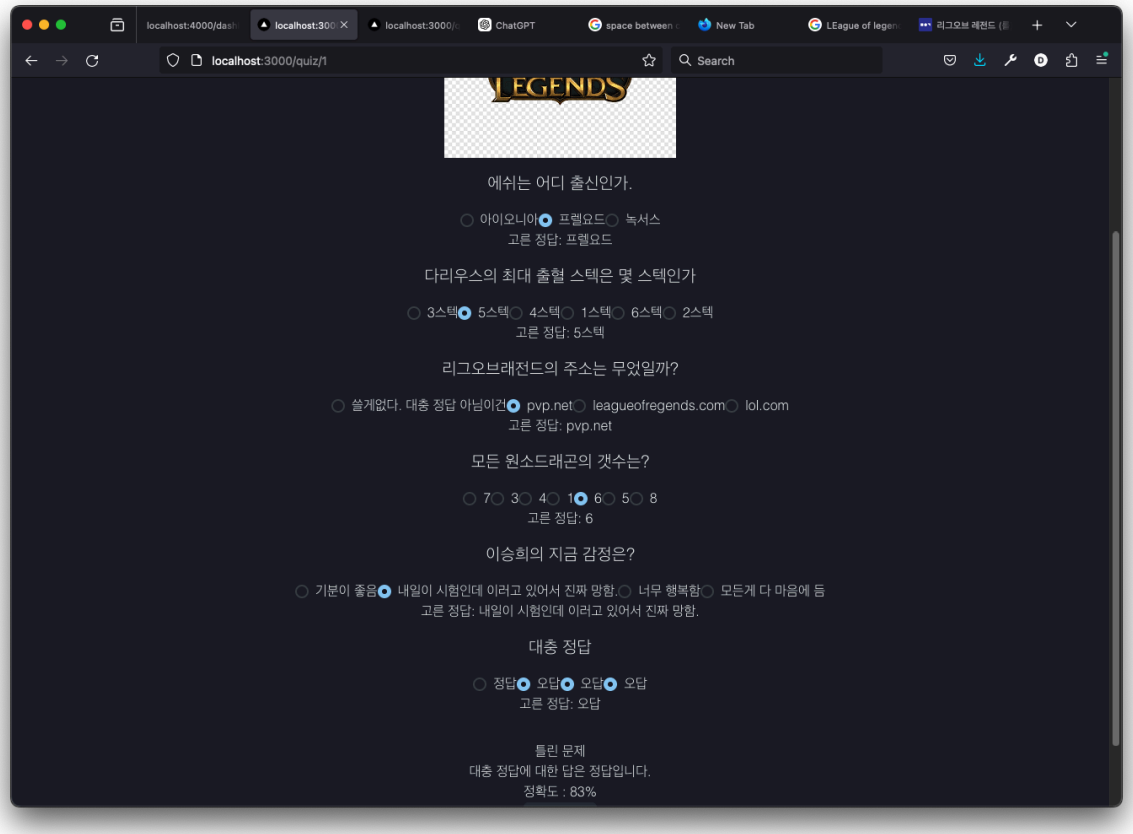
프론트에 대한 부분은 리엑트로 처리하였기에 코드가 난해합니다.



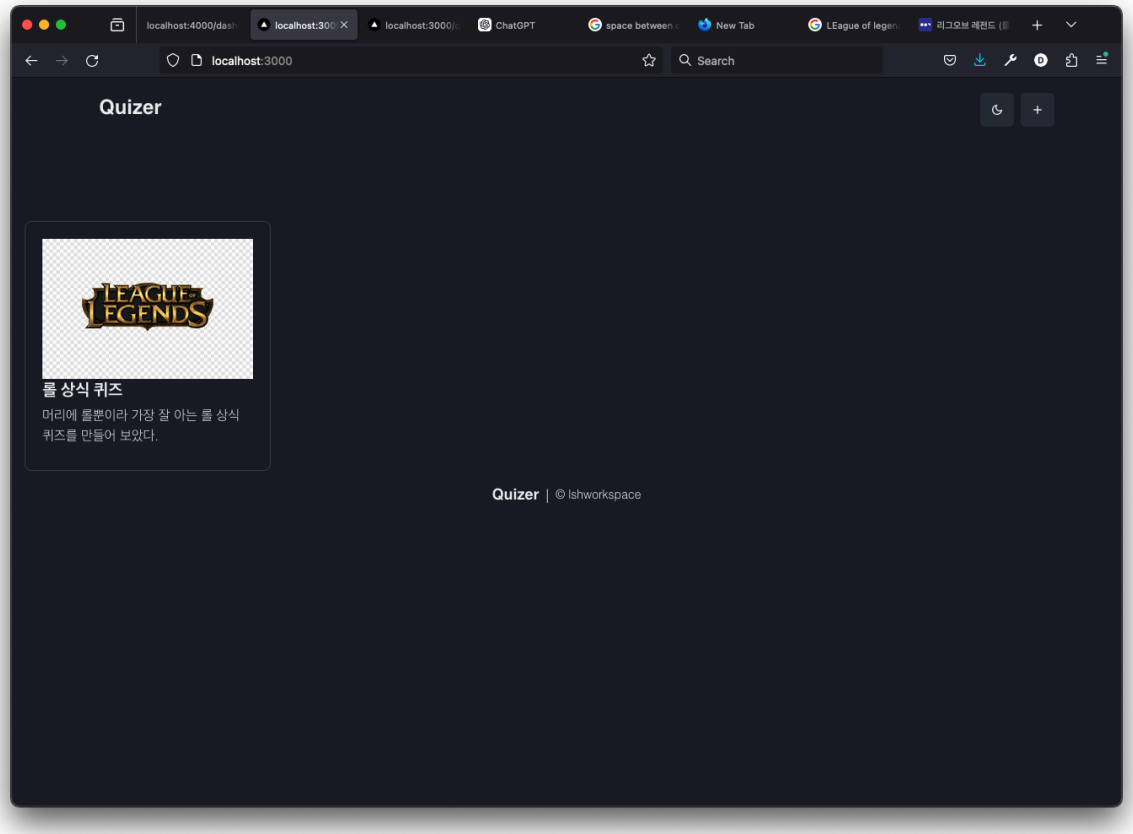
간단하게 설명하면 **Post**와 **Get** 방식에서 **Json**으로 데이터를 파싱하는 함수를 모듈로 분리해 두었고, 타입을 철저히 검사해 오류가 최대한 일어나지 않도록 하였습니다. 모든 데이터를 불러와 프론트 단에서 데이터를 분리, 디자인을 입혀 **Lazy**하게 표현하도록 하였습니다. 프론트의 로딩이 서버의 전송보다 빠르면 안되기에 프론트에서는 서버의 통신 이후 **useEffect**를 통해 데이터 변동을 감지하고 변경하게 해 두었습니다. 또한 퀴즈마다의 페이지를 다르게 하기 위해 파라미터 값을 가져와서 그에 맞게 데이터베이스의 데이터를 가져와 처리할 수 있도록 하였습니다.



등록페이지 입니다. 퀴즈를 추가 삭제할 수 있고, 보기를 추가 삭제 할 수 있습니다. 사진도 넣을 수 있게 되어있습니다. 사진은 base64로 인코딩 해서 서버에 전송하게 됩니다. 서버에서의 사진 정보는 base64코드로 저장되어 있게 되는 거죠, 사용하게 될때는 프론트에서 png파일로 디코딩하여 사용하게 하였습니다. 본래 확장자에 따라 코드를 지정하여 어떤 사진이든 지원하려 하였지만, 데이터베이스의 스키마를 이미 정해두었고, 시간적 여유가 많지 않아 png의 무손실을 노리고 Png만 지원하기로 하였습니다.



퀴즈 단 입니다.
동일한 정답은 같이 선택하게 해두었고 문제는 랜덤하게 섞이게 됩니다. 이 이유가 정답을 1번 보기로 받게한 이유입니다. 정보를 섞어서 저장하는 것보다 프론트단에서 섞어서 표현하는게 더 편하다고 생각되어 채택하였습니다. 정답을 표출할때 섞어둔 인덱스를 다시 제 조합하고 원본과 대조하여 정답의 여부를 검사합니다. 검사에 실패한 틀린 문항은 문제와 문답의 0번째 인덱스를 가져와 표출하게 해두었습니다. 정답 버튼을 누르면 다시 문답을 수정할 수 없게 코드를 작성해 두었습니다. 혹시 햇갈릴까봐 아래에 어떤 문답을 선택했는지 도 나오게 하였습니다. 절때로 선택 코드 테스트 하다가 그냥 남겨둔게 맞습니다.



대쉬보드에서의 디자인 입니다. 한번 직접 추가 해 보시는 것도 좋을 것 같습니다. 버튼은 상단 위 +

결론 및 제언

구현이 끝났습니다. 개인적으로 프론트 엔드 코드를 자세하게 풀고 싶지만, 저도 재정신인 상태로 한게 아니라 스파게티로 짜서 하나하나 다시 생각하기에 너무 오래 걸릴 것 같아서 간략하게 기능을 요약하였습니다. 개을리 하지는 않았지만, 이 프로젝트를 완성하기 위해 Nestjs와 Nextjs, Typescript, Sqlite, Prisma, Express등 여러개를 같이 공부하며 코드를 작성하다 보니 시간이 오래 걸려 코드가 띔띔히 작성되어 저도 과거의 제 코드를 다시 공부하며 천천히 작성하였습니다...공식문서를 읽다보니 영어 실력도 조금 좋아진 것 같습니다.

재미있었던 프로젝트 인것 같습니다. 제 개인적인 사이트에 Oauth 2.0을 탑제할 예정인데, 이와 관련된 자료가 많이 없습니다. 아무래도 토큰 발행, 여러 사이트간의 연결등 다른 사이트의 Oauth로 연결해서 하는게 아니라 제가 모두 관리해야 하기 때문에 대부분의 사람들이 이에 관해 자료를 안만들어 둔 것 같습니다. 그나마 보였던게 Spring Sec를 써서 구현하는 방법인데 개인적으로 Spring의 무거움을 좋아하지 않아 Nest의 자료가 있어 이왕에 Nest공부도 같이 할겸 서버와 백엔드로 모든 기능을 구현해 보았습니다. 분명 미래에 도움이 될것 이라고 믿어 의심치 않습니다.

PDF도 마크다운을 지원했으면 하는바가 있습니다... 너무 오래 걸리는 것 같아서 다음에 만들어볼까 생각중입니다.

시험이 내일인데 당일에 끝냈습니다.. 화이팅 하겠습니다..

긴 글 읽어주셔서 감사합니다.