

Next.js

REST API 서버 구축

React 프론트엔드 앱을 지원하는 Next.js 백엔드 API 구현

<https://nextjs.org/>

<https://nextjs.org/showcase>



실습 - 1. 프로젝트 생성 및 환경 설정

1. Next.js 프로젝트 생성

```
# 프로젝트 생성 (pnpm 사용)
```

```
pnpm create next-app backend-nextjs --typescript --eslint --app --no-src-dir --no-tailwind
```

2. 필요 패키지 설치

```
# Prisma (ORM)
```

```
pnpm add @prisma/client
```

```
pnpm add -D prisma
```

```
# 인증 관련
```

```
pnpm add bcryptjs jsonwebtoken
```

```
pnpm add -D @types/bcryptjs @types/jsonwebtoken
```

```
# OpenAI (영수증 분석용, 선택사항)
```

```
pnpm add openai
```

실습 - 1. 프로젝트 생성 및 환경 설정

3. 환경 변수 설정

```
# .env 파일 생성
touch .env

# .env
DATABASE_URL="file:/data/dev.db"
JWT_SECRET="your-super-secret-jwt-key-change-in-production"
OPENAI_API_KEY="your-openai-api-key"
ALLOWED_ORIGINS="http://localhost:3000,http://localhost:3001"
```

💡 JWT_SECRET 생성 방법 : 안전한 JWT 비밀키는 **최소 32자 이상의 랜덤 문자열** 이어야 합니다.

```
# Node.js 사용
node -e "console.log(require('crypto').randomBytes(64).toString('hex'))"
```

실습 - 1. 프로젝트 생성 및 환경 설정

4. next.config.js 설정

```
// next.config.js
/** @type {import('next').NextConfig} */
const nextConfig = {
  reactStrictMode: true,
  env: {
    DATABASE_URL: process.env.DATABASE_URL,
    JWT_SECRET: process.env.JWT_SECRET,
    OPENAI_API_KEY: process.env.OPENAI_API_KEY,
    ALLOWED_ORIGINS: process.env.ALLOWED_ORIGINS,
  },
};

module.exports = nextConfig;
```

5. 기본 타입 정의 src/types/index.ts

실습 - 2. 데이터베이스 설정

1. ORM (Object-Relational Mapping)이란?

- 정의: 객체(Object)와 관계형 데이터베이스(Relational DB)를 연결(Mapping)해주는 기술
- 핵심: 복잡한 SQL 쿼리 대신 프로그래밍 언어의 메서드를 사용하여 데이터베이스를 조작

구분	SQL 직접 작성	Prisma ORM 사용
코드	SELECT * FROM users WHERE id = 1	prisma.user.findUnique({ where: { id: 1 } })
특징	문자열 기반, 오타 발생 가능	메서드 기반, 자동완성 지원

2. Prisma의 장점

- 생산성 향상: 직관적인 데이터 모델링 및 자동 완성
- 타입 안전성(Type Safety): TypeScript와 완벽 호환, 런타임 에러 방지
- 유지보수 용이: 스키마 변경 시 클라이언트 코드 자동 업데이트

실습 - 2. 데이터베이스 설정

3. Prisma 초기화

```
# pnpm
pnpm prisma init --datasource-provider sqlite
```

4. Prisma Client 설정 (Singleton Pattern) src/lib/prisma.ts

Next.js의 **Hot Reloading** 기능으로 인해 DB 연결이 무수히 늘어나는 것을 방지하기 위해 **싱글톤 패턴**을 사용합니다

```
export const prisma =
  globalForPrisma.prisma ?? // 이미 있으면 재사용
  new PrismaClient(); // 없으면 새로 생성

// 운영에서는:
// 1. 서버 시작
// 2. globalForPrisma.prisma = undefined (처음)
// 3. new PrismaClient() 생성
// 4. export const prisma에 할당
// 5. 다른 파일에서 import 시 같은 인스턴스 사용
```

실습 - 2. 데이터베이스 설정

3. Prisma 스키마 작성 : **src/prisma/schema.prisma**

4. seed 파일 작성 : **src/prisma/seed.ts**

5. package.json에 스크립트 추가 "prisma": {"seed": "pnpm ts-node --esm prisma/seed.ts"}

6. 스키마 DB에 적용 : pnpm prisma db push

7. Prisma Client 생성 : pnpm prisma generate # Prisma Client 생성 (타입 정의 생성)

8. 데이터베이스 리셋
데이터베이스 완전 초기화 (모든 데이터 삭제!)
pnpm prisma db push --force-reset

또는 프로젝트의 pnpm 스크립트 사용
pnpm db:reset

실습 - 3. 인증 시스템 구현

토큰 기반 인증 (JWT 방식) : 사용자가 로그인하면 서버가 'JWT(JSON Web Token)'라는 암호화된 토큰을 발급하고, 클라이언트는 이 토큰을 저장했다가 API 요청 시 헤더에 담아 보내 인증을 처리하는 방식

JWT의 구조 (Header . Payload . Signature)

- 헤더(Header): 토큰의 타입(typ)과 서명 알고리즘(alg) 정보가 담겨 있습니다.
- 페이로드(Payload): 사용자 정보(ID, 권한 등)와 토큰 만료 시간(exp) 같은 '클레임(Claim)'이 Key/Value 형태로 담겨 있습니다.
- 시그니처(Signature): 헤더, 페이로드, 서버의 비밀 키를 조합해 암호화한 것으로, 토큰의 위변조 방지 및 유효성 검증에 사용됩니다.

동작 과정

1. 로그인: 사용자가 아이디/비밀번호로 로그인 요청 → 서버 DB 확인 후 유효하면 JWT 생성하여 클라이언트에 발급.
2. 요청: 클라이언트는 발급받은 JWT를 Authorization 헤더에 담아 서버로 요청.
3. 검증: 서버는 받은 JWT의 시그니처를 검증하여 변조 여부 및 유효성을 확인.
4. 응답: 검증 성공 시 요청 처리

실습 - 3. 인증 시스템 구현

1. 인증 유틸리티 : src/lib/auth.ts

```
import jwt from 'jsonwebtoken';
import bcrypt from 'bcryptjs';
import { NextRequest } from 'next/server';
const JWT_SECRET = process.env.JWT_SECRET || 'fallback-secret-key';
export interface JwtPayload { // JWT 페이로드 타입
  userId: number;
  email: string;
  iat?: number;
  exp?: number;
}
// JWT 토큰 생성
export function generateToken(userId: number, email: string): string {
  return jwt.sign( { userId, email }, JWT_SECRET, { expiresIn: '24h' } );}
```

실습 - 4. 미들웨어

미들웨어 = “중간에서 처리하는 layer” 요청과 응답 사이에서 공통 로직을 처리하는 계층

미들웨어가 하는 일: 인증/인가 확인, CORS 헤더 추가, 로깅, 요청 리다이렉트, 요청/응답 수정, Rate Limiting

- **Middleware**: 요청이 완료되기 전에 실행되는 코드 (Gatekeeper 역할)
- **Route Handler**: 실제 요청을 처리하고 응답을 반환하는 코드 (Backend Logic)

구분	미들웨어 (Middleware)	API Route Handler
파일 위치	프로젝트 루트. middleware.ts	앱 라우터 폴더 내 app/api/.../route.ts
실행 시점	요청이 들어오는 즉시 (페이지/API 렌더링 전)	특정 URL 경로와 매칭되었을 때
적용 범위	전역 또는 특정 패턴 (matcher 설정 가능)	해당 파일이 위치한 특정 라우트
주요 용도	공통 로직 처리 (보안, 리다이렉트)	비즈니스 로직 처리 (CRUD, 데이터 가공)
실행 환경	Edge Runtime (가볍고 빠름, 기능 제한)	Node.js Runtime (모든 Node.js API 사용 가능)

실습 - 4. 미들웨어 - CORS 헤더 추가

CORS = "교차 출처 리소스 공유"

웹 브라우저가 다른 출처(Origin)의 리소스에 접근할 때 적용되는 보안 정책입니다.

출처(Origin)란?

`https://example.com:3000/path`

→ 프로토콜 + 호스트 + 포트 = Origin

CORS 필요성 :  보안을 위한 브라우저 정책:

브라우저는 기본적으로 다른 출처로의 요청을 차단합니다. 이를 "동일 출처 정책 (Same-Origin Policy)"이라고 합니다.

- React 앱 (localhost:3000) → API 서버 (localhost:8080)
- 프론트엔드 (app.com) → 백엔드 API (api.app.com)

이런 정당한 요청은 CORS 설정으로 허용해야 함!