Backend Bootcamp: NestJS + PostgreSQL + Prisma (Hands-On, 2025)

Welcome! This is a **step-by-step, project-based** course designed for a frontend developer (Angular/Next.js/WordPress) to become **full-stack** using **NestJS + PostgreSQL + Prisma**.

You'll build a production-grade REST API with **Auth (JWT + Refresh)**, **CRUD**, **Validation**, **Caching**, **Docs (Swagger)**, **Testing**, **Docker**, and **CI/CD**. Each module includes tasks, code, and checkpoints.

> Repo name suggestion: `nest-postgres-prisma-starter`

---

0) Prerequisites & Tools

- **Node.js LTS** (>= 20.x) and **pnpm** or **npm** (we'll use **pnpm**)
- **Docker Desktop** (or Podman) for local Postgres/Redis
- **Git** & GitHub account
- Editor: VS Code + extensions: *Prisma*, *ESLint*, *Prettier*

Global CLIs

```
corepack enable   # enables pnpm
pnpm -v
npm i -g @nestjs/cli
nest --version
```

---

1) Project Setup & Scaffolding

1.1 Create Nest app

```
nest new nest-api --package-manager pnpm
cd nest-api
pnpm add -D prettier eslint-config-prettier eslint-plugin-simple-import-sort
```

1.2 Base structure

```
src/
  main.ts
  app.module.ts
```

```
  common/
    guards/
    interceptors/
    filters/
  prisma/
    prisma.module.ts
    prisma.service.ts
  auth/
  users/
  posts/
```

1.3 main.ts

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

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.setGlobalPrefix('api');
  app.enableCors();
  app.useGlobalPipes(new ValidationPipe({ whitelist: true,
forbidNonWhitelisted: true }));
  await app.listen(process.env.PORT || 3000);
}
bootstrap();
```

---

2) Database with Docker (PostgreSQL)

2.1 docker-compose.yml (root)

```
version: '3.9'
services:
  db:
    image: postgres:16
    restart: unless-stopped
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: app
    ports:
      - '5432:5432'
    volumes:
```

```
      - db_data:/var/lib/postgresql/data
volumes:
  db_data:
```

Start DB:

```
docker compose up -d
```

---

3) Prisma Setup

3.1 Install & init

```
pnpm add @prisma/client
pnpm add -D prisma
pnpm prisma init
```

3.2 .env

```
DATABASE_URL="postgresql://postgres:postgres@localhost:5432/app?schema=public"
JWT_ACCESS_SECRET="dev_access_secret_change"
JWT_REFRESH_SECRET="dev_refresh_secret_change"
```

3.3 prisma/schema.prisma

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

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id        String   @id @default(cuid())
  email     String   @unique
  password  String
  name      String?
  role      Role     @default(USER)
  posts     Post[]
```

```
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

enum Role {
  USER
  ADMIN
}

model Post {
  id        String   @id @default(cuid())
  title     String
  content   String?
  published Boolean  @default(false)
  authorId  String
  author    User     @relation(fields: [authorId], references: [id])
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
}

model RefreshToken {
  id         String   @id @default(cuid())
  token      String   @unique
  userId     String
  user       User     @relation(fields: [userId], references: [id])
  expiresAt  DateTime
  createdAt  DateTime @default(now())
}
```

3.4 Generate & Migrate

```
pnpm prisma generate
pnpm prisma migrate dev --name init
```

3.5 Seed (optional) Create `prisma/seed.ts` and add a couple of users/posts. Run with `pnpm ts-node prisma/seed.ts` (after adding `ts-node`).

---

4) PrismaModule (Nest <-> Prisma)

4.1 prisma/prisma.module.ts

```ts
import { Global, Module } from '@nestjs/common';
import { PrismaService } from './prisma.service';

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

4.2 prisma/prisma.service.ts

```ts
import { Injectable, OnModuleInit, OnModuleDestroy } from '@nestjs/common';
import { PrismaClient } from '@prisma/client';

@Injectable()
export class PrismaService extends PrismaClient implements OnModuleInit,
OnModuleDestroy {
  async onModuleInit() {
    await this.$connect();
  }
  async onModuleDestroy() {
    await this.$disconnect();
  }
}
```

4.3 app.module.ts

```ts
import { Module } from '@nestjs/common';
import { PrismaModule } from './prisma/prisma.module';
import { UsersModule } from './users/users.module';
import { AuthModule } from './auth/auth.module';
import { PostsModule } from './posts/posts.module';

@Module({
  imports: [PrismaModule, UsersModule, AuthModule, PostsModule],
})
export class AppModule {}
```

---

5) Users Module

5.1 DTOs `users/dto/create-user.dto.ts`

```ts
import { IsEmail, IsString, MinLength } from 'class-validator';

export class CreateUserDto {
  @IsEmail() email: string;
  @IsString() @MinLength(6) password: string;
  @IsString() name?: string;
}
```

5.2 Service & Controller (sketch)

```ts
// users.service.ts
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../prisma/prisma.service';
import { CreateUserDto } from './dto/create-user.dto';
import * as argon2 from 'argon2';

@Injectable()
export class UsersService {
  constructor(private prisma: PrismaService) {}
  async create(dto: CreateUserDto) {
    const hash = await argon2.hash(dto.password);
    return this.prisma.user.create({ data: { email: dto.email, password: hash,
name: dto.name } });
  }
  findByEmail(email: string) {
    return this.prisma.user.findUnique({ where: { email } });
  }
  findById(id: string) {
    return this.prisma.user.findUnique({ where: { id } });
  }
}
```

```ts
// users.controller.ts
import { Body, Controller, Get, Param, Post } from '@nestjs/common';
import { UsersService } from './users.service';
import { CreateUserDto } from './dto/create-user.dto';

@Controller('users')
export class UsersController {
  constructor(private users: UsersService) {}
  @Post()
  create(@Body() dto: CreateUserDto) { return this.users.create(dto); }
  @Get(':id')
```

```
  getOne(@Param('id') id: string) { return this.users.findById(id); }
}
```

---

6) Auth Module (JWT + Refresh)

6.1 Install

```
pnpm add @nestjs/jwt @nestjs/passport passport passport-jwt argon2
pnpm add -D @types/passport-jwt
```

6.2 auth/types.ts

```
export type JwtPayload = { sub: string; email: string; role: 'USER'|'ADMIN' };
```

6.3 auth/jwt.strategy.ts

```
import { Injectable } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { JwtPayload } from './types';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy, 'jwt') {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: process.env.JWT_ACCESS_SECRET,
    });
  }
  validate(payload: JwtPayload) { return payload; }
}
```

6.4 auth/auth.service.ts

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { PrismaService } from '../prisma/prisma.service';
import { JwtService } from '@nestjs/jwt';
import * as argon2 from 'argon2';
```

```ts
@Injectable()
export class AuthService {
  constructor(private prisma: PrismaService, private jwt: JwtService) {}

  async register(email: string, password: string, name?: string) {
    const hash = await argon2.hash(password);
    const user = await this.prisma.user.create({ data: { email, password: hash,
name } });
    const tokens = await this.issueTokens(user.id, user.email, user.role);
    return { user: { id: user.id, email: user.email, name:
user.name }, ...tokens };
  }

  async login(email: string, password: string) {
    const user = await this.prisma.user.findUnique({ where: { email } });
    if (!user || !(await argon2.verify(user.password, password))) {
      throw new UnauthorizedException('Invalid credentials');
    }
    const tokens = await this.issueTokens(user.id, user.email, user.role);
    return { user: { id: user.id, email: user.email, name:
user.name }, ...tokens };
  }

  async issueTokens(sub: string, email: string, role: 'USER'|'ADMIN') {
    const accessToken = await this.jwt.signAsync({ sub, email, role }, {
secret: process.env.JWT_ACCESS_SECRET, expiresIn: '15m' });
    const refreshToken = await this.jwt.signAsync({ sub }, { secret:
process.env.JWT_REFRESH_SECRET, expiresIn: '7d' });
    await this.prisma.refreshToken.create({ data: { token: refreshToken,
userId: sub, expiresAt: new Date(Date.now()+7*24*3600*1000) } });
    return { accessToken, refreshToken };
  }

  async rotateRefreshToken(token: string) {
    const found = await this.prisma.refreshToken.findUnique({ where: {
token } });
    if (!found || found.expiresAt < new Date()) throw new
UnauthorizedException('Invalid refresh');
    const user = await this.prisma.user.findUnique({ where: { id:
found.userId } });
    const tokens = await this.issueTokens(user!.id, user!.email, user!.role);
    // optionally revoke old token
    await this.prisma.refreshToken.delete({ where: { token } });
    return tokens;
  }
}
```

6.5 auth/auth.controller.ts

```
import { Body, Controller, Post } from '@nestjs/common';
import { AuthService } from './auth.service';

@Controller('auth')
export class AuthController {
  constructor(private auth: AuthService) {}

  @Post('register')
  register(@Body() body: { email: string; password: string; name?: string }) {
    return this.auth.register(body.email, body.password, body.name);
  }

  @Post('login')
  login(@Body() body: { email: string; password: string }) {
    return this.auth.login(body.email, body.password);
  }

  @Post('refresh')
  refresh(@Body() body: { refreshToken: string }) {
    return this.auth.rotateRefreshToken(body.refreshToken);
  }
}
```

6.6 auth/auth.module.ts

```
import { Module } from '@nestjs/common';
import { JwtModule } from '@nestjs/jwt';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { JwtStrategy } from './jwt.strategy';

@Module({
  imports: [JwtModule.register({})],
  providers: [AuthService, JwtStrategy],
  controllers: [AuthController],
})
export class AuthModule {}
```

6.7 Route protection

```
// common/guards/jwt-auth.guard.ts
import { AuthGuard } from '@nestjs/passport';
import { Injectable } from '@nestjs/common';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}
```

Use in controllers:

```
@UseGuards(JwtAuthGuard)
@Get('me')
me(@Req() req) { return req.user; }
```

---

7) Posts Module (CRUD + Ownership)

7.1 DTOs

```
export class CreatePostDto { title: string; content?: string }
export class UpdatePostDto { title?: string; content?: string; published?:
boolean }
```

7.2 Service

```
// posts.service.ts
@Injectable()
export class PostsService {
  constructor(private prisma: PrismaService) {}
  create(authorId: string, dto: CreatePostDto) {
    return this.prisma.post.create({ data: { ...dto, authorId } });
  }
  findAll() { return this.prisma.post.findMany({ where: { published:
true } }); }
  update(id: string, userId: string, dto: UpdatePostDto) {
    return this.prisma.post.update({ where: { id, authorId: userId }, data:
dto });
  }
  delete(id: string, userId: string) {
    return this.prisma.post.delete({ where: { id, authorId: userId } });
  }
}
```

7.3 Controller

```
// posts.controller.ts
@UseGuards(JwtAuthGuard)
@Post()
create(@Req() req, @Body() dto: CreatePostDto) { return
this.posts.create(req.user.sub, dto); }

@Get()
findAll() { return this.posts.findAll(); }

@Patch(':id')
update(@Param('id') id: string, @Req() req, @Body() dto: UpdatePostDto) {
return this.posts.update(id, req.user.sub, dto); }

@Delete(':id')
remove(@Param('id') id: string, @Req() req) { return this.posts.delete(id,
req.user.sub); }
```

---

8) Validation, Error Handling, Serializers

- We enabled `ValidationPipe` with `whitelist` and `forbidNonWhitelisted`.
- Add custom exceptions or filters in `common/filters` as needed.
- Use `class-transformer` to hide sensitive fields when returning entities.

```
pnpm add class-validator class-transformer
```

Example: remove `password` before returning user (already removed in service via DTO).

---

9) API Docs (Swagger/OpenAPI)

```
pnpm add @nestjs/swagger swagger-ui-express
```

`main.ts`:

```
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';

const config = new DocumentBuilder()
  .setTitle('Nest API')
  .setDescription('REST API docs')
```

```
  .setVersion('1.0')
  .addBearerAuth()
  .build();
const document = SwaggerModule.createDocument(app, config);
SwaggerModule.setup('docs', app, document);
```

Visit: `http://localhost:3000/docs`

---

10) Caching & Rate Limiting (Optional but Recommended)

10.1 Redis for cache

```
# add to docker-compose.yml
  redis:
    image: redis:7
    ports:
      - '6379:6379'
```

```
pnpm add cache-manager ioredis @nestjs/cache-manager
```

10.2 Rate limiting

```
pnpm add @nestjs/throttler
```

Setup in `app.module.ts` with `ThrottlerModule.forRoot({ ttl: 60, limit: 60 })`.

---

11) Testing (Unit + E2E)

```
pnpm add -D @types/supertest supertest ts-node
```

- **Unit tests** for services using `Jest` (preconfigured by Nest). - **E2E tests** spin up app and hit endpoints with Supertest; use a separate test DB URL.

Create `.env.test`:

```
DATABASE_URL="postgresql://postgres:postgres@localhost:5432/app_test?
schema=public"
```

Run migrations against test DB before E2E.

---

12) Scripts (package.json)

```
{
  "scripts": {
    "dev": "nest start --watch",
    "build": "nest build",
    "start": "node dist/main.js",
    "start:prod": "NODE_ENV=production node dist/main.js",
    "prisma:migrate": "prisma migrate dev",
    "prisma:deploy": "prisma migrate deploy",
    "prisma:studio": "prisma studio",
    "test": "jest",
    "test:e2e": "jest --config ./test/jest-e2e.json"
  }
}
```

---

13) Security Checklist

- Store secrets in `.env` (never commit)
- Hash passwords with **argon2**
- Use **JWT** with short access token (15m) + long refresh (7d)
- Enable **CORS** and **Helmet**
- Add **rate limiting**, **input validation**
- Enforce **RBAC** (role guard) for admin routes

---

14) Deployment (Docker + Render/Railway/Fly/AWS)

14.1 Dockerfile

```
FROM node:20-alpine AS deps
WORKDIR /app
COPY package.json pnpm-lock.yaml ./
RUN corepack enable && pnpm i --frozen-lockfile

FROM node:20-alpine AS builder
WORKDIR /app
COPY --from=deps /app/node_modules ./node_modules
COPY . .
```

```
RUN pnpm build

FROM node:20-alpine AS runner
WORKDIR /app
ENV NODE_ENV=production
COPY --from=builder /app/node_modules ./node_modules
COPY --from=builder /app/dist ./dist
COPY prisma ./prisma
CMD ["node", "dist/main.js"]
```

14.2 Deploy notes - Set env vars in cloud: - `DATABASE_URL` - `JWT_ACCESS_SECRET`, `JWT_REFRESH_SECRET` - On deploy, run `prisma migrate deploy` before starting app.

---

15) CI/CD (GitHub Actions)

`.github/workflows/ci.yml`

```
name: CI
on: [push, pull_request]
jobs:
  build-and-test:
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres:16
        env:
          POSTGRES_USER: postgres
          POSTGRES_PASSWORD: postgres
          POSTGRES_DB: app_test
        ports: ['5432:5432']
        options: >-
          --health-cmd="pg_isready -U postgres" --health-interval=10s --health-
timeout=5s --health-retries=5
    steps:
      - uses: actions/checkout@v4
      - uses: pnpm/action-setup@v4
        with: { version: 9 }
      - uses: actions/setup-node@v4
        with:
          node-version: 20
          cache: 'pnpm'
      - run: pnpm i
      - run: cp .env.example .env || true
      - run: pnpm prisma generate
```

```
      - run:
DATABASE_URL=postgresql://postgres:postgres@localhost:5432/app_test pnpm prisma
migrate deploy
      - run: pnpm test
```

---

16) API Quick Test (cURL)

```
# Register
curl -X POST http://localhost:3000/api/auth/register \
 -H 'Content-Type: application/json' \
 -d '{"email":"a@b.com","password":"secret123","name":"Abhi"}'

# Login
curl -X POST http://localhost:3000/api/auth/login \
 -H 'Content-Type: application/json' \
 -d '{"email":"a@b.com","password":"secret123"}'

# Create Post (replace TOKEN)
curl -X POST http://localhost:3000/api/posts \
 -H 'Authorization: Bearer TOKEN' -H 'Content-Type: application/json' \
 -d '{"title":"Hello","content":"First post"}'
```

---

17) Stretch Goals

- **GraphQL API** with `@nestjs/graphql`
- **WebSockets** (Gateway) for real-time updates
- **File uploads** (S3/GCS) + presigned URLs
- **RBAC Guard** and **Policy-based access control**
- **Audit logs** with Prisma middleware

---

18) Milestones & Checkpoints

1. **M1 – Bootstrapped**: Nest app runs, CORS+Validation on
2. **M2 – DB Ready**: Postgres up, Prisma schema migrated
3. **M3 – Users**: create user, no password leaks
4. **M4 – Auth**: register/login/refresh works, guarded routes
5. **M5 – Posts**: Create/Update/Delete with ownership
6. **M6 – Docs**: Swagger reachable, bearer auth works
7. **M7 – Tests**: at least 3 unit + 1 E2E
8. **M8 – Deploy**: API live with DB + migrations

---

19) What You'll Build (Deliverables)

- `nest-api/` repository with:
- `src/` modules (auth, users, posts, prisma, common)
- `prisma/` schema + migrations (+ optional seed)
- `docker-compose.yml` (postgres, redis optional)
- Dockerfile
- Swagger docs at `/docs`
- CI workflow

---

20) Next Steps (Do Now)

1. Install Docker & Node 20 LTS.
2. Run `nest new nest-api` and paste the provided code files.
3. Bring up Postgres: `docker compose up -d`.
4. Configure `.env`, run `pnpm prisma migrate dev`.
5. Start API: `pnpm dev` → test `http://localhost:3000/docs`.

When you're ready, ping me: **"Module 1 done"** and I'll drop **Module 2 labs** (GraphQL, WebSockets, and RBAC), or we can integrate this with your **Angular** app as a client.