Month 4, Week 2: The Unabridged Learning Guide

Introduction: From Manual Labor to an Automated Assembly Line

Last week, we took our first step into the world of professional, opinionated frameworks by scaffolding a basic NestJS application. We learned about the three core architectural pillars—Modules, Controllers, and Providers—that form the blueprint of every NestJS project. We now have a structure, but it is an empty one.

This week, we bring that structure to life. We will move from a simple scaffold to a robust, production-ready API by mastering the patterns that make NestJS so powerful.

First, we will take a **deep dive into Dependency Injection (DI)**. We will move beyond the basic concept to understand provider scopes and the immense power of custom providers, learning how NestJS manages the lifecycle of our services and dependencies.

Next, we will tackle one of the most critical aspects of API development: **input validation**. We will learn how to define the public "contract" of our API using **Data Transfer Objects (DTOs)** and leverage the `class-validator` and `class-transformer` libraries to create a robust, automatic validation layer. This is how we protect our application from bad data and ensure stability.

Finally, we will put all this theory into practice. We will embark on a complete **rebuild of the CRUD API** we have been working on, this time using our ORM (`TypeORM`) within the NestJS framework. By comparing the "before" (our manual Express/TypeORM implementation) and the "after" (the clean, DI-powered NestJS implementation), you will gain a profound appreciation for the power, safety, and productivity that a professional-grade framework provides.

Table of Contents

Module 1: Dependency Injection Deep Dive

Recap: Inversion of Control (IoC) Inversion of Control is the design principle where the control of object creation and linking is passed from the application code to the framework. Dependency Injection is the *pattern* used to implement this principle. Instead of `new UserService()` inside our controller, the NestJS IoC container reads the constructor's type hints and provides (injects) the necessary instance.

Provider Scopes: Singleton, Request, and Transient

- **`Scope.DEFAULT` (Singleton):** This is the default. NestJS creates **one** instance of your `UserService` and shares that exact same instance every time it's injected anywhere in your application. This is highly memory-efficient and fast.
- **`Scope.REQUEST`:** When a provider is request-scoped, a **brand new instance** is created for every single incoming HTTP request, and it is destroyed once the request is complete. This is useful for storing request-specific state (like a user's identity from a token or a unique request ID for tracing) without that state leaking between different users' requests. It has a higher performance cost due to the constant object creation.
- **`Scope.TRANSIENT`:** A transient provider is never shared. Every single class that injects a transient provider gets its own brand new, private instance.

Custom Providers for Configuration and More The DI system is not limited to class instances. You can inject any value you want by creating a custom provider. This is commonly used for configuration objects.

```
// A file for constants
// src/constants.ts
export const API_CONFIG = 'API_CONFIG';

// In your module file
// src/config/config.module.ts
@Module({
  providers: [
    {
      provide: API_CONFIG, // Use the constant as a safe token
      useValue: { // The value to be injected
        apiKey: process.env.API_KEY,
        timeout: 5000,
      },
    },
  ],
  exports: [API_CONFIG], // Export the provider so other modules can use it
})
export class ConfigModule {}

// In a service in another module that imports ConfigModule
import { Inject } from '@nestjs/common';
import { API_CONFIG } from '../constants';
```

```
// ...
constructor(@Inject(API_CONFIG) private config: { apiKey: string, timeout: number })
    // this.config.apiKey is now available
}
```

Module 2: DTOs & Robust Validation

The API Contract: Why DTOs are Essential A Data Transfer Object is a class whose sole purpose is to define the shape of data being sent over the network. It acts as a formal contract. * **For `POST` / `PATCH` requests:** It defines what the `req.body` should look like. * **For `GET` responses:** It can define the shape of the data being sent back, allowing you to separate your public API response from your internal database entity. For example, your `User` entity has a `passwordHash` property, but your `UserDto` would not, preventing you from ever accidentally leaking it.

Decorator-Based Validation with `class-validator` This library provides a huge set of decorators you can place on the properties of your DTO classes.

```
// src/users/dto/create-user.dto.ts
import { IsEmail, IsNotEmpty, IsString, MinLength, MaxLength, IsOptional, IsEnum } fr

enum UserRole {
  ADMIN = 'admin',
  EDITOR = 'editor',
  VIEWER = 'viewer',
}

export class CreateUserDto {
    @IsString()
    @IsNotEmpty()
    @MinLength(3)
    @MaxLength(50)
    name: string;

    @IsEmail()
    @IsNotEmpty()
    email: string;

    @IsEnum(UserRole)
    @IsOptional() // This field can be omitted
    role?: UserRole = UserRole.VIEWER; // We can even provide a default
}
```

Automating Validation with the `ValidationPipe` A Pipe in NestJS is a class decorated with `@Injectable()` that implements the `PipeTransform` interface. It sits between the incoming request and your controller method. The built-in `ValidationPipe` is a powerful pipe that: 1. Takes the plain JavaScript object from the request body.

2. Uses `class-transformer` to convert it into an instance of your DTO class. 3. Uses `class-validator` to run all the validation decorators you defined on that class. 4. If validation fails, it automatically throws an exception, which NestJS formats into a `400 Bad Request` response with a detailed list of the validation errors. 5. If validation succeeds, it passes the now-validated and transformed DTO instance to your controller method.

Module 3: Rebuilding the CRUD API the NestJS Way

Injecting a TypeORM Repository To connect our NestJS module with our TypeORM entities, we use the `TypeOrmModule`. 1. **In the root `AppModule`**, you import `TypeOrmModule.forRoot({...})` with your main database connection options. 2. **In each feature module** (e.g., `UsersModule`), you import `TypeOrmModule.forFeature([User, ...])`. This tells TypeORM, "The repositories for these specific entities should be made available for injection within this module." 3. **In your service** (`UsersService`), you inject the repository using the `@InjectRepository(User)` decorator in the constructor.

Implementing the Service Logic The service is where the database interaction happens. The code becomes a clean translation of our intent into TypeORM repository methods.

```
@Injectable()
export class UsersService {
  constructor(
    @InjectRepository(User)
    private usersRepository: Repository<User>,
  ) {}

  create(createUserDto: CreateUserDto): Promise<User> {
    const newUser = this.usersRepository.create(createUserDto);
    return this.usersRepository.save(newUser);
  }

  findAll(): Promise<User[]> {
    return this.usersRepository.find();
  }

  async findOne(id: number): Promise<User> {
    const user = await this.usersRepository.findOneBy({ id });
    if (!user) {
      throw new NotFoundException(`User with ID ${id} not found`);
    }
    return user;
  }
  // ... and so on for update and remove
}
```

Notice how the service now throws specific NestJS exceptions (`NotFoundException`),

which the framework will automatically translate into the correct HTTP status codes (404).

Connecting the Controller The controller becomes incredibly lean. Its only job is to define the route, apply parameter decorators to extract data, and call the corresponding service method. It contains no business logic itself.

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

  @Post()
  create(@Body() createUserDto: CreateUserDto) {
    return this.usersService.create(createUserDto);
  }

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

  @Get(':id')
  findOne(@Param('id') id: string) {
    return this.usersService.findOne(+id);
  }
  // ...
}
```

Take-Home Assessment: The Fully-Featured `books` API

Objective: To demonstrate mastery of NestJS architecture by building a new, fully-validated CRUD resource from scratch and connecting it to the database via TypeORM.

The Task: You will be given the NestJS project with the `users` resource already built. Your task is to add a complete, production-ready `books` API.

1. Generate the Resource: * Use the NestJS CLI to generate a `books` resource: `nest g resource books`.

2. Create the `Book` Entity (`src/books/entities/book.entity.ts`): * Define a `Book` entity that maps to a `books` table. * It should have properties for `id` (primary key), `title` (string), `author` (string), and `publicationYear` (number). Use the appropriate TypeORM decorators.

3. Update the `BooksModule`: * Import `TypeOrmModule.forFeature([Book])` to make the `Book` repository available for injection.

4. Create the DTOs: * In `src/books/dto/create-book.dto.ts`, add properties for `title`, `author`, and `publicationYear`. Add `class-validator` decorators to ensure `title` and `author` are non-empty strings and `publicationYear` is a positive integer. *

In `src/books/dto/update-book.dto.ts`, do the same, but make all properties optional using `@IsOptional()`.

5. Implement the `BooksService` (`src/books/books.service.ts`): * Inject the `BookRepository`. * Implement all five business logic methods (`create`, `findAll`, `findOne`, `update`, `remove`) using the repository to interact with the database. Ensure you handle the "not found" case gracefully by throwing a `NotFoundException`.

6. Update the `BooksController`: * Ensure all five controller methods are correctly implemented, calling their corresponding service methods. * Make sure the parameter decorators (`@Param`, `@Body`) are correctly typed with your DTOs.

7. Test Thoroughly: * With the `ValidationPipe` enabled globally, use Postman to test every endpoint. * Test creating a book with valid data. * Test creating a book with invalid data (e.g., missing title, non-numeric year) and verify that you receive a `400 Bad Request` with detailed error messages. * Test all other GET, PATCH, and DELETE endpoints.

Submission: Submit your entire updated project via a Pull Request to your personal assignments repository. ‘