Month 4, Week 3: The Unabridged Learning Guide

Introduction: From Structure to Professional Patterns

In our previous lessons, we built a foundational understanding of NestJS. We learned how its core architectural pillars—Modules, Controllers, and Providers—provide a robust, opinionated structure that brings order and scalability to our backend applications. We have built a functioning CRUD API that is organized and connected to a database.

However, a professional application is more than just its CRUD logic. It is a resilient, secure, and consistent system. This week, we elevate our application by implementing the advanced patterns that define production-grade software. We will learn the "NestJS way" of handling the critical, cross-cutting concerns that apply to every part of our application.

First, we will master **Guards**. We will move beyond simple Express middleware and re-implement our JWT authentication strategy using this powerful feature, creating a declarative and tightly-integrated security layer that determines who can access our endpoints.

Next, we will take a deeper look at **Pipes**, focusing on their dual roles of **transformation** and **validation**. We will see how NestJS uses them to sanitize and check incoming data before it ever touches our business logic.

Finally, we will explore **Interceptors**, one of the most powerful features in the NestJS toolkit. We will learn how they allow us to "intercept" the request-response cycle to add functionality both *before* and *after* our route handlers run. We will use them to implement crucial features like consistent response formatting and performance logging.

By the end of this lesson, you will have the tools to build an API that is not just functional, but truly professional—secure, consistent, and architected for the real world.

Table of Contents

Module 1: Authentication the "NestJS Way" with Guards

What is a Guard? A Guard is a class that implements the `CanActivate` interface. Its sole purpose is to determine if a request should be processed by the route handler. It does this by implementing a single method, `canActivate()`, which must return a boolean (or a Promise/Observable that resolves to a boolean). * If it returns `true`, the request proceeds. * If it returns `false` (or throws an exception), NestJS immediately denies the request.

Guards are the idiomatic, NestJS way to handle **authentication** (is the user logged in?) and **authorization** (is this logged-in user allowed to do this specific action?).

The Request Lifecycle & Execution Context When a request comes into a NestJS application, it flows through a specific sequence: 1. Middleware (Global, then Module-specific) 2. **Guards** 3. Interceptors (the "before" part) 4. Pipes 5. Controller Method (Route Handler) 6. Interceptors (the "after" part - on the way out) 7. Exception Filters (if an error was thrown)

The `canActivate(context: ExecutionContext)` method receives an `ExecutionContext` object. This is a powerful wrapper around the incoming request. It allows Guards to be protocol-agnostic. To get the standard HTTP request object, you use `context.switchToHttp().getRequest()`.

Building a JWT `AuthGuard` Our goal is to create a reusable guard that checks for a valid JWT in the `Authorization` header.

1. **Generate the Guard:** Use the CLI: `nest g guard auth/auth`
2. **Inject Dependencies:** Our guard will need the `JwtService` (from `@nestjs/jwt`, which you'd set up in an `AuthModule`) to verify the token. We inject this into the constructor.
3. **Implement `canActivate` Logic:**
   - Get the request object from the `ExecutionContext`.
   - Write a private helper method to extract the token string from the `Authorization: Bearer <token>` header.
   - If no token is found, `throw new UnauthorizedException()`.
   - Wrap the token verification in a `try...catch` block.
   - Inside the `try`, call `await this.jwtService.verifyAsync(token, { secret: ... })`. This function will throw an error if the token is invalid or expired.
   - If verification is successful, attach the resulting payload to the request object (e.g., `request['user'] = payload;`). This is a crucial step that makes the user's identity available to the rest of the application for this request.
   - `return true;`.
   - In the `catch` block, `throw new UnauthorizedException()`.

Applying Guards You apply guards declaratively using the `@UseGuards()` decorator. This is cleaner and more integrated than the Express `app.use()` pattern.

- **On a specific route:** `@UseGuards(AuthGuard) @Get(':id') findOne(...)`

- **On an entire controller:** `@UseGuards(AuthGuard) @Controller('users') export class UsersController { ... }`

Module 2: Pipes for Transformation and Validation

The Dual Role of Pipes A Pipe is a class that implements the `PipeTransform` interface. It has a `transform()` method that is executed just before a route handler is called. Pipes operate on the arguments that will be passed to the handler (e.g., `@Body()`, `@Param()`, `@Query()`).

1. **Transformation:** A pipe can transform the input data. The `ParseIntPipe` is a perfect example: it takes a string from the URL (`'123'`) and transforms it into a number (`123`).
2. **Validation:** A pipe can validate data. If the data is invalid, the pipe should throw an exception. The `ValidationPipe` does this by checking DTOs. If the incoming body doesn't match the `class-validator` rules, it throws a `BadRequestException`.

Using Built-in Pipes (`ParseIntPipe`) Using built-in pipes makes your controller code incredibly clean and robust.

Before (Manual, Error-Prone):

```
@Get(':id')
findOne(@Param('id') id: string) {
    const numericId = parseInt(id, 10);
    if (isNaN(numericId)) {
        throw new BadRequestException('Validation failed: ID must be a numeric string
    }
    return this.usersService.findOne(numericId);
}
```

After (Declarative, Robust):

```
@Get(':id')
findOne(@Param('id', ParseIntPipe) id: number) {
    // We can trust that `id` is a valid number at this point.
    // If it wasn't, the pipe would have already thrown a 400 error.
    return this.usersService.findOne(id);
}
```

Module 3: Interceptors for Cross-Cutting Concerns

What is an Interceptor? An Interceptor is a class that implements the `NestInterceptor` interface. It has a single method, `intercept()`, which gives you the ability to "intercept" the request-response stream. This is incredibly powerful for implementing logic that needs to wrap around the main route handler.

Use cases: * **Logging:** Log incoming requests and their outgoing responses. * **Performance Monitoring:** Time how long a request takes to process. * **Caching:** Return a

cached response instead of running the handler. * **Response Transformation:** Modify the data returned from a route handler to fit a consistent structure.

Implementing a Response Transformation Interceptor A professional API always returns data in a consistent envelope. Let's enforce that all our responses look like `{ statusCode: 200, data: ... }`.

The `intercept()` method receives the `ExecutionContext` and a `CallHandler`. The `CallHandler`'s `handle()` method returns an `Observable` from the RxJS library. We use the `.pipe()` method on this observable to apply operators. The `map` operator will take the data emitted by the route handler and transform it.

```
import { map } from 'rxjs/operators';
// ...
@Injectable()
export class TransformInterceptor<T> implements NestInterceptor<T, { statusCode: numb
  intercept(context: ExecutionContext, next: CallHandler): Observable<{ statusCode: n
    const statusCode = context.switchToHttp().getResponse().statusCode;

    return next.handle().pipe(
        map(data => ({
            statusCode,
            data
        }))
    );
  }
}
```

Applying Interceptors Globally While you can apply an interceptor with `@UseInterceptors()`, for something like response transformation, you want it to apply everywhere. You do this in your `main.ts` file.

```
// main.ts
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalInterceptors(new TransformInterceptor());
  await app.listen(3000);
}
```

Now, every successful response from your entire application will automatically be wrapped in the standardized format, without you ever having to think about it in your controllers or services. This is the power of an architectural framework.

Take-Home Assessment: Advanced Authorization and Data Sanitization

Objective: To demonstrate mastery of Guards and Interceptors by implementing a role-based authorization system and a data sanitization layer.

The Task: You will build upon the JWT-secured API from the in-class exercise.

1. Add Roles to Your System: * In your user data/entity, add a `roles` property (e.g., an array of strings like `['user', 'admin']`). * When you sign a JWT upon login, include this `roles` array in the token payload.

2. Create a `RolesGuard`: * Generate a new guard: `nest g guard auth/roles`. * This guard will be used for **authorization**. It should run *after* the `AuthGuard`. * Create a `Roles` decorator to set metadata: `export const Roles = (...roles: string[]) => SetMetadata('roles', roles);`. * In the `RolesGuard`, inject the `Reflector` service. * In the `canActivate` method: * Use the `Reflector` to get the required roles from the `@Roles()` decorator on the route handler. * Get the user object (with its roles) from `request['user']`, which was attached by the `AuthGuard`. * Compare the user's roles with the required roles. If the user has at least one of the required roles, return `true`. * If not, throw a `ForbiddenException` (which results in a `403 Forbidden` response).

3. Create a `SanitizeUserInterceptor`: * Generate a new interceptor: `nest g interceptor users/sanitize`. * This interceptor's job is to remove the `passwordHash` property from any `User` object or array of `User` objects before it is sent to the client. * In the `intercept()` method, use the `map` operator from RxJS. * Inside the `map`, check if the `data` is an array or a single object. Recursively go through the data and delete the `passwordHash` property from any user object you find. * Return the sanitized data.

4. Apply the New Features: * Apply the `SanitizeUserInterceptor` to the `UsersController` class so it affects all user routes. * On the `DELETE /users/:id` route, apply the guards and decorators to ensure only an authenticated admin can access it: `typescript @Roles('admin') @UseGuards(AuthGuard, RolesGuard) @Delete(':id') // ...`

Submission: Submit your entire updated project via a Pull Request to your personal assignments repository. Test your endpoints with different user roles to verify that your authorization logic is working correctly. ‘