Nest.js for Angular Developers

A Quick Start - Within an Nx Workspace

@ebuelbuel - https://x.com/ebuelbuel

2024-12-23 | 23:00 | Proceeding...

Contents

Disclaimer and Professional Notice

The concepts, architectures, and implementations described in this documentation are intended for use under the guidance of experienced software architects and professionals. While the presented approaches and patterns have been thoroughly tested in production environments, their successful implementation requires:

- Oversight by experienced software architects
- Proper evaluation of specific use cases and requirements
- Thorough understanding of architectural implications
- Appropriate risk assessment and mitigation strategies
- Consideration of organizational context and constraints

THIS DOCUMENTATION AND THE ACCOMPANYING CODE ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO:

- The documentation and code examples are provided for educational and informational purposes only
- No warranty or guarantee is made regarding the accuracy, reliability, or completeness of the content
- The author(s) assume no responsibility for errors or omissions in the content
- The author(s) are not liable for any damages arising from the use of this documentation or implementations derived from it. Implementation decisions and their consequences remain the sole responsibility of the implementing organization
- Success in one context does not guarantee success in another; proper evaluation is required

To minimize risks and ensure successful implementation:

- Conduct thorough architecture reviews before implementation
- Engage experienced software architects throughout the project lifecycle
- Implement comprehensive testing strategies
- Maintain proper documentation of architectural decisions
- Consider scalability, maintenance, and long-term implications
- Establish proper governance and review processes
- Ensure adequate team training and knowledge transfer

Implementation of the described architectures and patterns requires substantial expertise in:

- Enterprise software architecture
- Monorepo management and tooling
- Modern development practices and toolchains
- System design and integration
- Performance optimization and scaling
- Security best practices
- DevOps and CI/CD processes
- Azure, AWS, Google Cloud, Hetzner Cloud, or other cloud platforms

1 The Holy Trinity of Enterprise Development

Having worked in enterprise software architecture, **you'll often encounter data structures that reflect their historical evolution** - shaped by numerous stakeholders, each contributing their unique terminology.

> **Let's just say it's an interesting archaeological expedition**

Now, imagine the scenario: A new app needs to be developed, integrating with the existing API infrastructure. The deadline? Yesterday, of course! It needs to showcase cutting-edge tools, programming languages, and modern interfaces. Why? Perhaps a new CEO or government official is eager to demonstrate innovation in their next public appearance. After all, staying ahead of the competition is the name of the game!

1.1 Angular: The First Apostle of Type-Safe Frontend Salvation

Speaking of frontends; **What could possibly outshine Angular?** Especially when operating in German-speaking Europe? **Surprisingly, Angular is the framework that gets executive approval fastest** as executives recognise it as a **"modern interface solution"** - it's familiar to them, they've heard of it repeatedly, and chances are there are already several Angular applications running somewhere in the organisation.

1.2 NestJS: Modernizing Legacy Data Without Breaking Faith

> **Now that we've settled the frontend question, what about the dreadful data that has evolved organically over years?**

If Murphy's Law applies (and it always does), you might find database tables and columns in Low German dialect, leaving international team members completely lost - or as we say in German, they "understand only train station" (*"verstehen nur Bahnhof"* - an idiom for understanding absolutely nothing). Even as a native German speaker, I sometimes feel lost too - Low German can be as cryptic as hieroglyphics.

Understanding individual elements isn't always sufficient, as context can dramatically alter their meaning. Mastering domain-specific terminology (including considerations like semantic color schemes) is essential for successful modernization.

Now that we've "diplomatically acknowledged" that the existing data structure needs improvement, let's focus on modernizing this legacy system.

1.2.1 How to Pitch NestJS to Your CTO?

Best case scenario: You have a visionary CTO. Worst case scenario: You have an accountant in CTO's clothing[1].

- If we position it as a backend, the company's backend team (our API providers) might either feel threatened or suddenly develop "concerns" that get us ejected from the project.
- If we try selling it as a "frontend backend," the CTO's disciples will immediately interrupt with "*Why not use Angular's server rendering features?*" (which, admittedly, becomes a valid question from Angular 19+ onward).
- And if we dare to mention "middleware" at an unfavourable moment, the CTO will likely ask, "*What's that?*" - and we'll be forced to explain it as "a layer between the frontend and backend that handles business logic, data transformation, and validation." If he asks "Do we really need that?" - the answer is "Yes, my Lord," especially if the company size is at least 1999 employees or larger.

I usually call it **"middleware"** - fully aware that NestJS has its own different interpretation and implementation of this term (details coming later). *My sincere apologies to the NestJS team, but perhaps they might consider renaming their middleware concept… for my sake?*

From an architectural perspective, I prefer using NestJS to implement **"Separation of Concerns"**. This pattern helps maintain clean code boundaries by keeping business logic and data transformations in the middleware layer rather than the frontend.

Security best practices recommend implementing validation on both frontend and backend layers. While frontend validation improves user experience by **providing immediate feedback**, it can be circumvented using browser developer tools. Therefore, robust middleware validation through NestJS serves as a critical security measure (I'll demonstrate these security implications in detail later).

When **deploying Angular and NestJS**, I prefer using two different domains within the same cluster. If that's not feasible (*your accountant-turned-CTO might consider €12 per year an "unnecessary infrastructure expense"*), I'll use subdomains instead.

Under no circumstances would I advocate for hosting everything on the same domain with different ports (unless, of course, I'm developing locally on my own machine). This distinction is crucial, and I'd be delighted to elaborate on its importance at a later time.

This middleware is only accessible by authorized frontends that possess valid tokens (more on that later). *I've even offered some CTOs to personally sponsor these €12 per year - though that suggestion wasn't particularly well-received.*

---

[1] This chapter is dedicated to **@kammysliwiec**, the creator of NestJS. His dedication to the project and the community is truly admirable.

1.2.2 Making Backend Developers Question Their Life Choices

Nest.js stands as a comprehensive backend framework capable of replacing most conventional backend solutions. While its TypeScript foundations might raise eyebrows among traditional backend developers[2], its capabilities are undeniable.

This is precisely why we present **Nest.js** to **"traditional" backend developers** as an intermediate layer between our frontend and the **"core" backend**. This approach excels in scenarios requiring data modeling, internationalization, and modernized business logic handling.

While Nest.js often proves indispensable in projects dealing with historically (and sometimes hysterically) evolved data structures, positioning it as a middleware layer not only makes tactical sense but frequently becomes **a technical necessity**.

1.2.3 Why not use Angular Server Rendering?

The legitimate question "Why not use Angular Server-Side Rendering (SSR)?" deserves an answer: As Software Architects, we need to extensively field-test and validate "Angular SSR".

While Angular v19+ brings some stability (please don't stone me, but I personally had suboptimal experiences with earlier versions), we already have NestJS as middleware (in my terminology) thoroughly field and battle-tested.

It's simply not acceptable for an architect to build a modern, daring stadium that collapses during a live game. Nobody wants to be quoted in the tabloid headlines (especially not in "Bild"). That would be career suicide - you'd be relegated from designing architectures to carrying sandbags.

1.2.4 From Angular to Nest.js: Same But Different

When transitioning **from Angular to Nest.js**, you'll notice several similarities. **Both** frameworks are built on **TypeScript**, and both use **decorators** to define classes and methods.

The module system in Nest.js will feel familiar to Angular developers, though with some key differences. Controllers in Nest.js might remind you of the old Angular.js days, and while they serve a similar routing and request handling purpose as Angular components, they're fundamentally different. Think of them as the entry points for your HTTP requests rather than UI elements.

Currently, Nest.js doesn't have an equivalent to Angular's standalone components - though who knows what Kamil (Nest.js creator) has planned for the future.

While these architectural parallels appear complex at first, we'll explore all these details hands-on as we create a Nest.js application within our Nx monorepo. After all, learning by doing is the best

---

[2] The debate around "proper" backend languages often sees JavaScript/TypeScript facing skepticism from developers who favor languages like Go, Rust, Java, or C. Interestingly, even Python, despite its AI dominance, hasn't fully established itself in traditional backend development. While frameworks like Django (currently the de facto standard), Flask, CherryPy, and TurboGears have made notable attempts, Python's backend adoption remains primarily in specialized domains like data science and machine learning. Its widespread backend adoption would likely require significant hardware optimizations at the processor level.

approach.

1.2.5 Field and Battle Testing

How does one properly field and battle-test?

Simple: Rent a cloud server from Hetzner[3], create an app with Angular Server-Side Rendering (without a firewall), then post on Reddit or the dark web claiming, "*I've built the most secure frontend web app - who dares to challenge it?*"

If it survives six months without firewalls (but with OS updates), it might be production-ready.

But seriously, **security testing requires academically recognized metrics**. I haven't had the chance to thoroughly evaluate Angular v19+ yet - I'm still learning myself. The Angular team has done a masterful job here, and I tip my hat to their achievement.

And **testing**, whilst not quite a science, is certainly an art form: Unit testing, Integration testing, End-to-End testing (E2E), Security and Penetration testing, Usability and Accessibility testing, Responsiveness testing, Local testing, Pipeline testing, Background offline testing (during development as pre-commit and pre-push rules), Test-Driven Development (TDD), and Behaviour-Driven Development (BDD - which is essentially TDD done right).

I realise I might be turning this quick start guide into a doctoral thesis. But I say, when it comes to testing, it's rather easy to fall down the rabbit hole.

And thus, we have discussed two Apostles of our Holy Trinity. But what of the third one? What divine powers does Nx bring to our enterprise salvation?

1.3 Nx: The Holy Spirit of Enterprise Scalability

I've been using Nx since its inception[4]. When we want to create enterprise workspaces that offer a proven schema from day one (and through my own enhancements, provide infinite scalability and true re-usability - more on that later), Nx is ideal.

You know naming is hard, especially with multiple participants pulling in different directions. Well, surprisingly, I have a secret formula here: **ELVIS**. When I propose this name, no one rejects it. The business departments and domain experts are extremely creative, quickly transforming ELVIS into "electronic files", "super builder", "enterprise live information system", and so on.

I've been building enterprise workspaces named ELVIS since around 2008, and since 2018, I've been

---

[3] **Hetzner.com** is a German cloud provider known for its affordable prices and reliable services. It's a popular choice among European developers and it is also my preferred choice - they're an environmentally conscious company where I train my models cost-effectively. **Full disclosure:** I have no business relationship with Hetzner and am not affiliated with them.

[4] I just checked - I actually started using Nx a year after its creation, which proves my cautious approach to adopting new technologies. See for yourself: https://api.github.com/repos/nrwl/nx

doing it with great joy using Nx. So, if you're in a company with a workspace/framework/solution/product/… named ELVIS, there's a high probability that was my doing.

1.3.1 Why not use Angular CLI?

The reason is simple: the strong, omnipresent React community. But jokes aside, when a smarty-pants department head jumps up during my ELVIS presentation declaring "I don't like Angular", the response is immediate: "I love React too (*well, actually I just like her - it's not true love*) which is why you're welcome to develop your apps within ELVIS using React."

Usually, they sit down with a smile, but then another one springs up: "Can we use the libraries you've developed for Angular in React?" And if you know how, the answer is an immediate yes.

One of the main reasons we don't use Angular CLI is that large enterprises (typically operating internationally) want to bundle their applications and products for **specific markets** and offer them as a **unified software suite**. There are countless **existing apps**, and **they're not all necessarily Angular applications** - there's **everything** in the mix.

When **establishing our Nx Mono Repo approach** (*which we've named ELVIS*), existing applications (Angular, Vue, React, whatever framework you prefer) need to **become ELVIS-compatible**.

We typically prepare **automated import mechanisms** to manage this process. For these mechanisms to work effectively, TypeScript is a technical prerequisite. (*The comprehensive benefits and reasoning behind this choice will be explored in a dedicated chapter.*)

This necessity often leads to the **delicate task** of requesting all participating departments, products, and sub-products **to migrate their JavaScript applications to TypeScript**. While this typically triggers **an avalanche of reactions** across the organization, it's a **necessary transition** for achieving our architectural goals.

If the applications are already written in TypeScript, the integration becomes significantly simpler. And if teams have developed their apps in a current or previous Nx Mono Repo setup, the process becomes even more straightforward.

1.3.2 Why Nx?

While I won't market Nx - and frankly, it doesn't need marketing - its practical value for architects is clear: The workspace schema naturally supports fractal structures (as Mandelbrot would appreciate), making the internal logic instantly recognizable to all stakeholders.

The library management enables genuine re-usability (more on real versus fake re-usability later), while the CLI and plugin architecture allow for extensive automation and customization.

From an architect's perspective, Nx brings three critical capabilities:

- Enforced structural consistency across large-scale projects
- Granular control over module boundaries and dependencies
- Scalable build and test orchestration

For detailed technical reference and implementation guidance, consult the comprehensive Nx documentation: https://nx.dev/

1.3.3 Cross-Framework Library Re-Usability

When introducing Nx with Angular, we occasionally encounter raised eyebrows. React developers - as previously mentioned - often inquire about Angular libraries' compatibility with React. The answer is an unequivocal "yes."

Beyond this documentation repository, I plan to establish an ELVIS repository that will evolve step-by-step into an enterprise-ready Nx monorepo workspace. Detailed examples will follow as we progress through:

- Starting with a bare Nx framework
- Integrating Angular and Nest.js
- Developing our reusable libraries (is in another documentation)

For those eager to dive in, let's clarify the path to **cross-framework harmony**. You can achieve cross-framework library compatibility when any of these conditions are met.

- Core logic in TypeScript without framework dependencies
- Thin framework-specific wrappers for Angular, React, or others
- Shared interfaces for consistent data contracts
- Framework-agnostic state management
- Common utility functions and services

And I've developed a sixth approach: a special kind of Proxy Pattern. Not the traditional OOP (Object-Oriented Programming) proxy, but rather a translator between apps that makes the whole interaction transparent and straightforward (more on this later).

2 Setup Your Nx Workspace

This documentation is primarily designed for Angular experts[5] seeking a fast-track introduction to the NestJS ecosystem.

[5] This chapter is dedicated to **@DanielGlejzner** and **@Armandotrue**, two talented young colleagues who tirelessly share their Angular expertise with the community.

2.1 Create a New Nx Workspace

2.1.1 Pre-Requisites

2.1.1.1 VSCode

We will use VSCode as IDE (Integrated Development Environment). If you don't have it installed, Download and install VSCode from https://code.visualstudio.com/.

To seamlessly work with our development environment, we need to be **able to launch VSCode from the terminal**. This enables powerful workflows like:

- Opening project directories directly: code my-project
- Opening files at specific lines: code file.ts:42
- Managing VSCode from CI/CD pipelines
- Using VSCode as default Git editor etc.

Let's set this up now. Here's how to enable command line access:

1. Open Command Palette:
   - Windows/Linux: `Ctrl+Shift+P`
   - macOS: ⇧ ⌘ P (Command+Shift+P)
2. Type: `Shell Command: Install 'code' command in PATH`
3. Restart your terminal
4. Verify installation: `code --version` (should show `1.96.2` or higher)

2.1.1.1.1 Why VSCode for Initial Development?

As an architect, I fully support diverse development environments. However, when establishing a new enterprise-wide development initiative, it's strategic to start with a standardized, zero-cost solution that minimizes initial friction. **VSCode is an excellent starting point:** [6]

- Zero licensing costs
- Wide enterprise acceptance (particularly in Microsoft-dominated environments)
- Robust extension ecosystem
- Minimal learning curve
- Strong community support

---

[6] While my heart belongs to neovim, and I deeply empathize with developers who prefer their specific IDEs, VSCode represents a practical compromise that helps us move forward efficiently as a team. As you can see, even I as an architect need to make compromises - we're all in this together!

While tools like **IntelliJ** are excellent (and yes, I hear you, IntelliJ enthusiasts!), we'll defer discussions about premium IDE licenses until we've demonstrated concrete business value - typically after our first production deployment shows measurable ROI. This pragmatic approach helps us maintain focus on delivering value before optimizing developer preferences.

This standardization is temporary - once we've established our foundation and demonstrated success, we can revisit IDE choices and accommodate team preferences.

2.1.1.2 Node.js

Ensure you have Node.js installed. If not, download it from https://nodejs.org/. Use always the last LTS (Long Term Supported) version.

For macOS or linux users, I recommend using `n` as your Node.js version manager. Install it globally via:

```
1 npm install -g n
```

`n` provides seamless Node.js version management with minimal overhead. For comprehensive documentation and usage examples, refer to the official repository: https://github.com/tj/n

2.2 When Angular Met Nest.js: A Developer's Love Story

!!! note Proceeding,… Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla et euismod nulla. Curabitur feugiat, tortor non consequat finibus, justo purus auctor massa, nec semper lorem quam in massa.