Quick Start: Nx, Angular & Nest.js

@ebuelbuel

2024-12-19 | started

Contents

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.

Even when I understand the individual elements, context can completely transform their meaning. Mastering terminology (and yes, even the juxtaposition of certain color schemes, which can lead to rather unfortunate combinations) is crucial.

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.

- 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).

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?*

Personally, I always use NestJS as a form of **"Separation of Concerns"**. I simply don't want business logic or data transformation in the frontend. And validation must be done on both sides, as I can bypass the validations on the frontend via developer tools (more on that 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 think it's expensive at €12 per year), I'll use subdomains instead.

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.*

1.2.2 Why not use Angular Server Rendering?

The legitimate question "Why not use Angular Server Rendering?" deserves an answer: As Software Architects, we need to thoroughly field and battle-test "Angular Server Rendering."

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.

1.2.3 Field and Battle Testing

How does one properly field and battle-test?

Simple: Rent a cloud server from Hetzner, 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 develop-

ment 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[1]. 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 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.

The only condition I set for frontend apps is that they must be written in TypeScript. (Those who want to be represented in ELVIS must migrate their JavaScript apps to TypeScript.). Why TypeScript? That's a topic for another day (But I've valid reasons, I promise!).

---

[1] 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

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

If you wish to pursue a more academic approach, you can achieve cross-framework library compatibility when any of these conditions are met (Note: In our holy ELVIS, Core Logic could also reside in NestJS, though NestJS isn't mandatory for any app within ELVIS):

- 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).