Nest.js for Angular Developers

A Quick Start - Within an Nx Workspace

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

2025-01-03 | 19:50 | 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 Nx Enterprise: Angular Expert's Guide to Nest.js

1.1 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!

1.1.1 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.

1.1.2 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.

1.1.2.1 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. 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.*

1.1.2.2 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

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

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**.

1.1.2.3 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.

1.1.2.4 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 approach.

1.1.2.5 Field and Battle Testing

How does one properly field and battle-test?

[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.

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?

1.1.3 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 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.

1.1.3.1 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."

---

[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

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.

1.1.3.2 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/

1.1.3.3 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).

1.2 Enterprise Schema Design: Angular & Nest.js

Before diving into implementation, let's clarify some fundamental concepts that often cause confusion. Understanding these will be essential for the chapters ahead.

1.2.1 JSON Schema

JSON Schema provides a declarative vocabulary to validate JSON documents. Think of it as a strongly-typed contract that defines the expected structure, types, and constraints of your data.

Example: For a person object, a JSON Schema might specify:

— `name`: Required string, e.g. "Noah of Ancient Mesopotamia"
— `age`: Required number between 0-1000 (Don't be mad at me, think of Noah)
— `address`: Optional nested object with its own schema

In **Nx** monorepo contexts, JSON Schema validates crucial configuration files like `nx.json`. It ensures that workspace settings (`namedInputs`, `targetDefaults`, `plugins`, `generators`, etc.) maintain structural integrity and type safety.

Since `nx.json` now serves as the central configuration hub, the schema acts as a guardrail, preventing misconfigurations in project definitions, task executions, and dependency management.

1.2.1.1 JSON Schema Validation

The following diagram illustrates the principle of JSON Schema Validation (For more details, please refer to the official documentation at: **json-schema.org**):

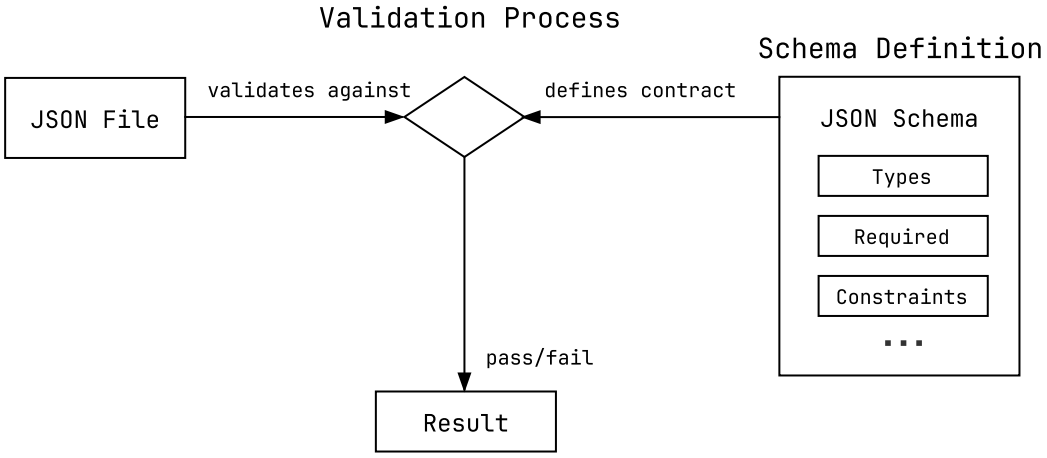


Figure 1: JSON Validation Process Overview

1.2.1.2 Why JSON Schema? Think Enterprise

Remember that awkward moment when you tried to explain to your boss *why all your apps look different?* Well, that's exactly what we're preventing here!

Nx schema isn't just another buzzword - it's your secret weapon for building enterprise-grade applications that play nicely together. Think of it as a strict but fair bouncer at an exclusive enterprise club:

— Every app built following the schema gets a VIP pass (**auto-recognition by Nx**)
— No more "but it works on my machine" drama (**consistent configurations**)
— Future-proof your apps for the ultimate enterprise party (**suite integration**)

Both Nx and Angular use JSON files for project/app configurations that are based on JSON Schema. (We'll examine these schemas in detail in another document.)

1.2.2 Schemas and Schematics

Following our deep dive into **JSON Schema**, let's connect the dots with **Schematics**: Schematics are essentially tools, **powerful tools**. They automate and standardize development workflows.

Schematics in a Nutshell

Imagine you're running a high-end restaurant. **JSON Schema** acts as your health inspector ensuring ingredients meet standards, while **Schematics** is your master robot chef who executes recipes flawlessly.

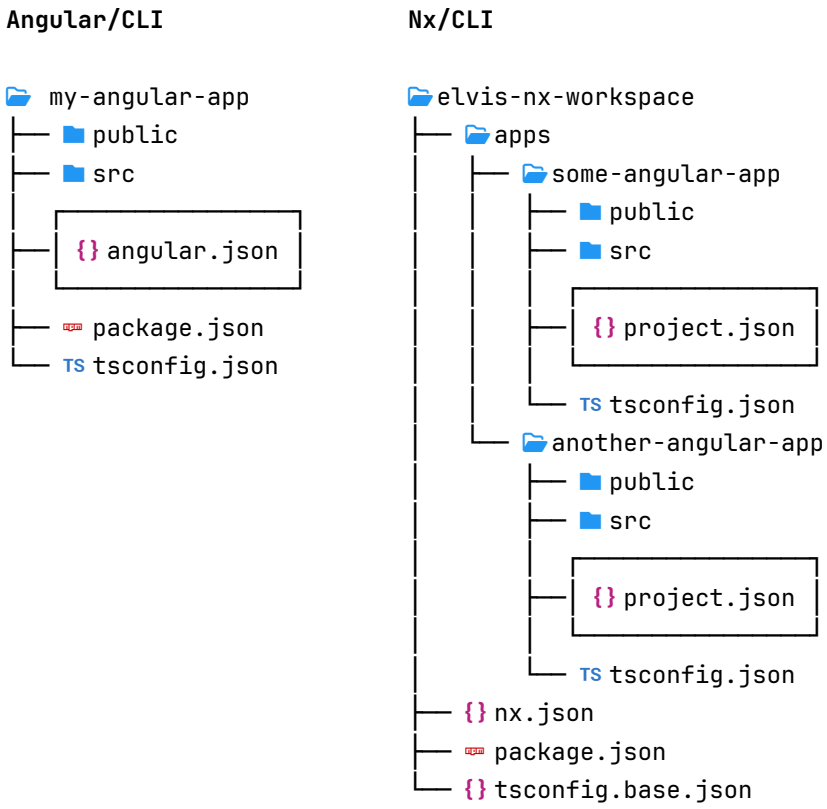
In **Nx terms**, running a generate command for a new library is like ordering your robot chef to prepare a new signature dish - it creates all necessary ingredients (**files**), updates the menu (**configuration**), and even sets up the kitchen layout (**project structure**) automatically.

One simple command, and your workspace transforms like a well-orchestrated kitchen during service time!

These clever transformations are powered by Schematics under the hood, which follows your JSON Schemas like a precise recipe book - ensuring every ingredient (file), measurement (configuration), and cooking step (code generation) meets your exacting standards, like a personal sous chef who never gets tired, never makes mistakes, and always delivers consistent results.

1.2.3 Differences in Schematics: For Angular Experts

Let's examine these two (simplified) structures:



On the left, we have an Angular app generated using Angular/CLI in a directory named "`my-angular-app`". In our case, the directory name is the app name. However, if we were to create

libraries for this app outside of `src` folder, this directory would quickly evolve into a small "workspace" in architectural terminology.

On the right, we have a workspace (established as such from the beginning) that we named `elvis-nx-workspace` (when I create an Nx workspace later, I'll simply name it `elvis`), which contains multiple Angular apps. Both apps have their own project.json, and there's no package.json in their directories since the workspace itself **has a single one**. If we were to create libraries, any Angular app could use these libraries. (The orchestration here is slightly different; we'll learn about the principle of *infinitely scalable fractals* and *true reusability* of global libraries later - libraries that can be developed *without interfering with each other*. This will probably be covered in another document or book, we'll see.)

Nx has elegantly streamlined Angular's sophisticated blueprint, optimizing it for multi-project architectures while preserving its core strengths. We'll explore this architectural evolution - particularly the transition from `angular.json` to `project.json` - when we dive into advanced workspace patterns and set up our Nx workspace together.

1.2.4 Version Clarity: Angular & Nest.js Nomenclature

When searching for "Nest.js", you'll find "NestJS" (for SEO purposes). The official website shows it as "Nest.js" and uses **nestjs.com** domain. In this document, we'll use **Nest.js**, following the official homepage nomenclature.

> **Using Synonyms**
>
> While we can use **Nest.js** and **Nest** (or **NestJS**) synonymously, we can't do the same with **Angular** and **Angular.js**[a].
>
> Because with AngularJS, we're referring to the firt initial version of the framework (version 1.x), which was released in ~2010. Angular (referred to as Angular 2+) is a complete rewrite of AngularJS, released in ~2016. **Angular.js and Angular are fundamentally different frameworks**, so it's crucial to distinguish between them.

[a] Angular experts are already familiar with the difference between Angular.js and Angular.

When we say Angular today, **we're always referring to the modern framework (Angular 2+)**, not the older Angular.js. As of this writing, the latest Angular version is 19.0.5[5].

[5] See also: **nx.dev/nx-api/angular/documents/angular-nx-version-matrix**

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

1.3.1 The First Date: Angular's Perspective

When our Angular experts establish the architecture, they know we need a configuration file that defines our `https://what-ever-it-is/api` (depending on the environment).

After all, we typically want to access an API and fetch our data from there, even though **Angular itself doesn't care where the data resides**. Since this document was written for Angular experts, intended to provide them with a quick introduction to the Nest.js world (within Nx), this context is particularly relevant.

This *"what-ever-it-is"* is typically `localhost` for local development, but only if our backend is also available locally. If the backend resides elsewhere, the targets should be configured accordingly. This mechanism is called **"Proxy Environment Configuration"** - at least that's what I call it in my documentations, and I believe it hits the nail right on the head. The allowed targets (targets we're permitted to access) are defined within this file.

You've guessed it correctly - this file should follow community best practices for naming (typically called `proxy.conf.json`, though it could technically be named anything else, but shouldn't be), should be located in a specific place (best practice: in the root directory of the respective Angular app), and must follow a specific schema.

It's crucial **not to confuse** this **"Proxy Environment Configuration"** with the localhost that Angular starts by default when running `ng serve` or `nx run <my-app>:serve`. These are two distinct configurations, though both use localhost. For the frontend, we use `localhost` because I've started my Angular app locally, and I've set `localhost` as targets in my Proxy Environment Config because, coincidentally, my backend is also available locally.

In our case, we want to set up an Enterprise Nx Mono Workspace (which we've named ELVIS), and this is where the Nx perspective comes into play.

Let's summarize what we've learned about how things look for Angular experts who have developed standalone Angular apps without Nx, versus how the world looks in an Nx Workspace. But before the brief summary, an important note: For the Angular app, it's not enough to just have the correct targets in `proxy.conf.json` - the file must also be referenced[6] properly. E.g. for local dev-servers:

For standalone Angular apps:

— Uses Angular Dev Server Proxy Configuration Schema
— Schema defined in **@angular-devkit/build-angular** package
— Configuration in angular.json references the proxy config

[6] We must know these schemas precisely because even though Nx typically adopts Angular schemas one-to-one, it makes modifications or omits certain elements. (I'll revisit this in another document/book; currently, I have the example of `vendorChunk`, which was marked for removal in Angular source code, but Nx removed it even before Angular did). This means if you're reading an older Angular book and want to simplify/optimise debugging on a Development Server, Nx might follow different guidelines. This type of information rarely appears in books - you need to read the developer comments on GitHub yourself, in this case under **build-system-migration.md**.

For Angular apps in Nx mono repo:

— Uses Nx Dev Server Proxy Configuration Schema
— Schema defined in **@nx/angular** package
— Configuration in project.json references the proxy config
— Same schema structure (almost) but different implementation under the hood

Key insight: The configuration approach remains conceptually similar, but the implementation details and file locations differ between standalone Angular and Nx environments. Fortunately, we don't need to create many of these configuration files ourselves - they're generated by the Nx CLI while creating our apps. No worries, I'll also guide you through the few manual configurations required, explaining each step in detail.

1.3.2 The Perfect Match: Nest.js Joins the Party

Angular experts know this all too well: Angular has done everything right, but somehow it's still not working, and it's not even our fault. Yes, you guessed it - CORS errors it's like Angular and Nest.js are at a fancy developer party, but they can't dance together because the bouncer (the browser) is extremely picky about cross-origin protocols.

Imagine Angular, dressed in its best HTTP requests, trying to approach Nest.js:

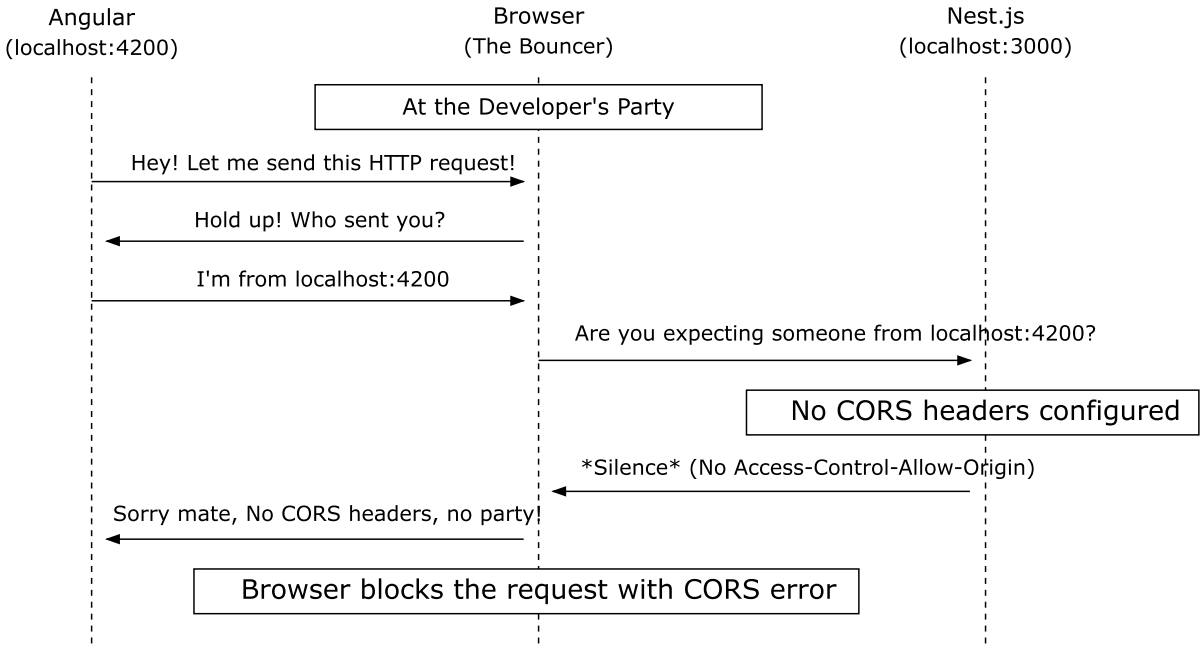


Figure 2: CORS Party Sequence Diagram

This is the infamous Cross-Origin Resource Sharing (CORS) error - where the browser plays an overprotective guardian, ensuring domains can't freely share data unless explicitly allowed. It's a crucial security feature, preventing malicious websites from making unauthorized requests to your API.

"*Don't worry, my dear Angular,*" says Nest.js, "*I'll add the proper CORS headers to my responses. Just tell me where you're from, and I'll put you on my VIP list. After all, security is important, but so is true love!*"

And here comes the voice-over you know from the movies - the Architect. He must ensure that Nest.js gets to know its Origin. Alas, nobody knows him - this unsung hero who tirelessly orchestrates things in the background, firing the right arrows to ensure this love story reaches its happy ending.

> **Hold On: Setup Details Follow**
> Don't worry about the earlier mention of manual configuration - we'll cover Angular and Nest.js app setup in detail when we create them.

And so our diligent Architect performs the time-honored ritual of matchmaking. First, he creates a special list in a file called `localDevOrigins.ts` under Nest.js's src folder - think of it as Nest.js's dating preferences:

```
export const LocalDevelopmentOrigins = [
  'https://localhost:4200',   // Formal address with a secure connection
  'http://localhost:4200',    // And the casual one
];
```

Then, in Nest.js's main.ts file, just before the grand opening (`await app.listen(port);`), the Architect whispers the sweet configuration of acceptance[7]:

```
import { LocalDevelopmentOrigins } from './localDevOrigins';
// ...

app.enableCors({
  origin: [
    // other potential suitors
    ...LocalDevelopmentOrigins,
  ],
});
```

[7] Naming is always crucial - the file should be descriptive and follow a clear naming convention. Sometimes we also call it "local-dev-origins.ts" or "local-development-origins.ts" - this naming decision is made by the team in a shared ritual known as "Coding Style Guidelines".

1.3.3 Nx Workspace: Angular and Nest.js

Now Angular and Nest.js stand face to face for the first time. Let's get their relationship status straight. To Nest.js, Angular is the great love, the irreplaceable wife, the so-called Origin. For Angular, however, Nest.js is simply the Proxy.

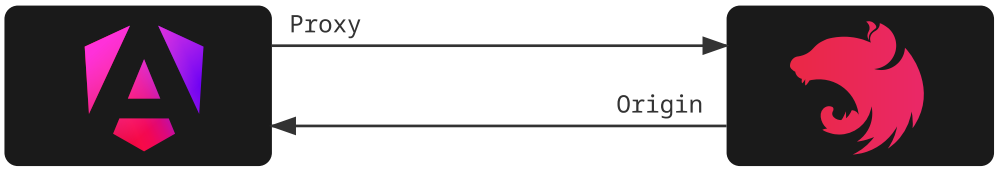


Figure 3: Relation

Why Proxy?
Well, it's kind of like a stand-in until the next one comes along, or maybe because it can be swapped out quickly and easily? Or perhaps because each environment has a different one? I'd better leave these philosophical questions to the philosophers, but I've a theory.

Both Angular and, naturally, Nx use the term `proxyConfig` in their schema. If you want to know exactly where, you'll need to look in the file: `@angular/cli/lib/config/schema.json`. In Nx, the schema is defined in `@nx/angular/src/builders/dev-server/schema.json` package. Please note that files, structures, and schemas may change in newer versions. This is just a snapshot in time.

```
Angular Schema
└── definitions
    └── AngularBuildBuildersDevServerSchema
        └── properties
            └── proxyConfig
```

```
Nx Schema
└── properties
    └── proxyConfig
```

Chronicles of Schema-Fu
As Master Oogway whispered: *Yesterday is history, tomorrow is a mystery, but your git history* 👀. *The schema you're looking for might not be the schema you need.*

Master Shifu answered: *True validation comes not from regex patterns, but from accepting that users will always find a way to break your schema. Yet you must still* ***find ways to protect the workspace****. A schema that bends will survive, but one that validates will prevail.*

Production Environments are Different

Keep in mind that serving an app locally can be entirely different from running it in a production environment (such as in the cloud where a web server might be running, or when your app is embedded in a content management system like Adobe Experience Manager)[a].

[a] In my twenty years of software architecture experience, I thought I'd seen every possible variation imaginable - at least that's what I believed two years ago. Now I find myself in agreement with Socrates, albeit in a slightly different context: *The more I learn, the more I realise there's no limit to creative implementation approaches.*

1.4 Install Recommended Software

1.4.1 VSCode

1.4.1.1 Download and Install

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/.

Here's how the VSCode interface roughly looks like (simplified), Please remember names of these individual areas (that's the VSCode Interface terminology), as we'll refer to them later:

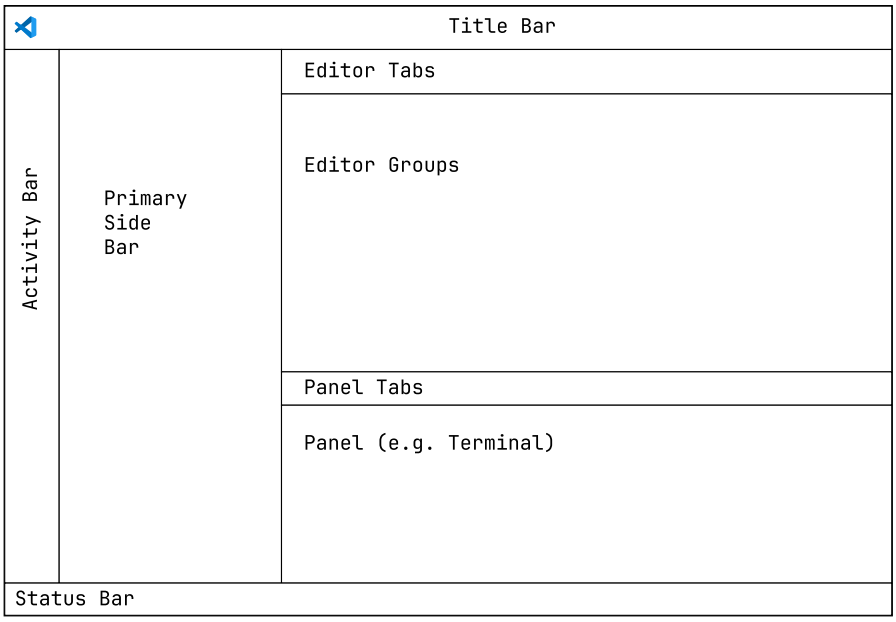


Figure 4: VCode Interface Overview

1.4.1.2 Register `code` as Shell Command

To seamlessly work with our development environment, we need to be **able to launch VSCode from the terminal**[8]. 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.

Open the `Command Palette`

> **F1 Key Might Help too**
> If not mapped differently, the F1 key works across Windows, macOS and Linux.If not, then use the following shortcuts:
>
> — Windows/Linux: `Ctrl+Shift+P`
> — macOS: (`Cmd+Shift+P`)

Then type: `Shell Command:`

You should see in the `Title Bar` of the VSCode window a dropdown menu; select `Shell Command: Install 'code' command in PATH` and press `Enter`.

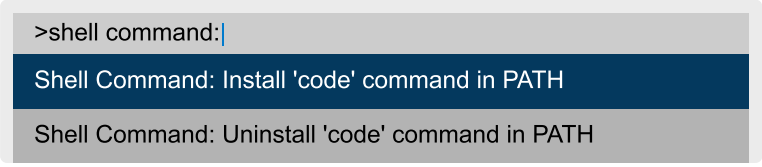


Figure 5: Registering `code` as Shell Command

Restart your terminal & verify installation:

```
code --version
1.96.2 # or higher
```

1.4.1.3 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:**.

— Zero licensing costs

[8] The `code` command is usually auto-registered during VSCode installation on Windows. However, certain corporate policies may override this registration (don't ask me how & why).

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

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.

And at this point, our **Neovim** users get a special hug from me - I fear they won't get it anywhere else - and I better stop here briefly to wipe away my tears.

1.4.2 Node.js

Ensure you have Node.js installed. If not, download it from https://nodejs.org/. Always use the latest LTS (Long Term Support) version.

At the time of writing this document, that's version `v22.11.0`.

Even better would be **installing a Node Version Manager** and switching between versions as needed[9].

MacOS and Linux Users
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**

Windows Users
For Windows users, use `nvm-windows` - it's specifically designed for Windows environments. You can download it from **https://github.com/coreybutler/nvm-windows**

[9] I can imagine that in larger enterprises, you can't access the internet from Windows Terminal and proxy servers need to be configured. There's probably a 2-page document on Confluence that's outdated and links to 10 other pages. In banks and insurance companies, getting this access can take up to a month (not joking). Until all approvals come through, we work with memory sticks (if they're not blocked too).

Corporate Reality

Only in huge enterprises (especially in Germany and Switzerland) you might find node-package in the internal software package system, but no `nvm-windows`[a].

If we can't install nvm on Windows Machines (and Azure AKS is still pending approval - which would allow us to work in containers), then everyone must install exactly the same version that also runs on the deployment server. And **just like that**, the architect has made themselves unpopular at the very beginning of the project[b].

---

[a] Don't be surprised - it's not uncommon for developers to lack admin rights on their own machines in German speaking EU. In these cases, I usually recommend using your vacation bonus to buy an Apple notebook for development. I've probably already written a VPN driver for your company's Mac setup anyway. You can then use the Windows laptop for time tracking, reading emails, and responding to MS Teams messages. *BTW: How can anyone expect developers to work without admin rights on their own machines? I find that horrifying.*

[b] But beware: If the company's internal cloud (like Azure) isn't configured with the latest `LTS` version (*I don't always have admin rights on internal cloud servers and need to ask the CTO to request one for me, assuming I don't feel like getting stoned to death*), we have to align with whatever version is there. It's not uncommon that I need to call India, South Korea, New Zealand, or Hungary to change my password. God help you if it's India - you'll definitely need a **phonetic alphabet chart**, and if the password contains special characters (*and it must*), you'll spend an afternoon on it and probably need to continue the next day. If we need to switch to an earlier LTS that doesn't support root async/await, I might even need to downgrade my ELVIS import generators. (ELVIS, our Nx Mono Repo, needs to be able to import all apps regardless of framework - more on that later).

2 Create a New Nx Workspace (Angular Setup)

This documentation is primarily designed for Angular experts[10] seeking a fast-track introduction to the NestJS ecosystem.

> **Impatient Minds**
> For immediate setup, jump to `Final Confirmation Chapter`. For detailed explanations and rationale behind this setup, continue with the following chapters.

2.1 Give Your Workspace a Name

After all this talk about relationships and configurations, it's time to get hands-on and create something new. Let's bring our first **Nx workspace** to life - we'll call it **Elvis**, because it's about to shake things up! Open your terminal and run the following command:

```
❯ npx create-nx-workspace@latest elvis # or your preferred name
```

> **Déjà Vu**
> Well, well, well... After the very first command, it looks like our Nx buddies might have spent some time in the React.js world before. While in classic Angular we'd go with `npm i -g @angular/cli@latest` followed by `ng new <my-angular-app>`, our Nx team embraces the `npx` approach (suspiciously similar to `npx create-react-app`).

At this point, Junior Developers often ask if `npx` has something to do with `nx`, and when we installed `npx`. Let's clear this up:

> **What is npx?**
> `npx` is a package runner tool that comes bundled with npm (since version 5.2.0). It's automatically installed when you install Node.js. Its main purpose is to execute npm package binaries without having to install them globally. Think of it as a way to "*try before you globally install*" - perfect for one-off commands like creating new projects.

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

Does npx install packages locally?

No, `npx` doesn't install packages permanently. It works like this:

- First checks if the package is already installed locally
- If not, downloads the package to a temporary cache
- Executes it from there
- After execution, removes the downloaded files

So for commands like `npx create-nx-workspace`, the package itself isn't installed permanently - it's temporarily cached, used, and then cleaned up. However, the `workspace` it creates is permanent and includes all necessary dependencies in your project's `node_modules`.

2.2 Confirm Download & Workspace Installation Start

Alright, let's continue with the CLI interaction. At this point, **Nx** asks for permission to install the workspace:

```
Need to install the following packages:
create-nx-workspace@20.3.0
Ok to proceed? (y)   # press Enter
```

Makes perfect sense to answer with **(y)** here - just hit Enter and let's get this party started.

Why CLI Isn't a Lottery

Pro tip: Throughout the setup, use your **up/down arrow keys** to browse through the options and press **Enter** to confirm. And no, randomly mashing arrow keys while blindly hitting Enter isn't a valid selection strategy - though I've seen people try! Let's make our choices deliberately, shall we?

2.3 Choose Your Stack: Angular

```
NX  Let's create a new workspace [https://nx.dev/getting-started/intro]

? Which stack do you want to use? …
None:           Configures a TypeScript/JavaScript monorepo.
React:          Configures a React application with your framework of choice.
Vue:            Configures a Vue application with your framework of choice.
Angular:        Configures a Angular application with modern tooling.
Node:           Configures a Node API application with your framework of choice.
```

What do you think, dear Angular experts, which option should we choose here? Angular, of course. Let's select it and continue by pressing Enter.

2.4 Multiple Projects Setup: Integrated Monorepo

```
✓ Which stack do you want to use? · angular
? Integrated monorepo, or standalone project? …
Integrated Monorepo:  Nx creates a monorepo that contains multiple projects.
Standalone:           Nx creates a single project and makes it fast.
```

Here we'll select "Integrated Monorepo" since we want to create multiple projects (apps) - at least one Angular app (which Nx will set up for us shortly) and later we'll add a Nest.js project (app). Press Enter to confirm.

2.5 Set Your Initial Application's Name

```
✓ Which stack do you want to use? · angular
✓ Integrated monorepo, or standalone project? · integrated
✓ Application name · fe-ng-base  # or your preferred name
```

Next, Nx asks for the application name. Let's name it `fe-ng-base` (short for Frontend Angular Base) and press Enter (You can choose any meaningful name here).

2.5.1 Choose Your Bundler

```
✓ Which stack do you want to use? · angular
✓ Integrated monorepo, or standalone project? · integrated
✓ Application name · fe-ng-base
? Which bundler would you like to use? …
esbuild [ https://esbuild.github.io/ ]
Webpack [ https://webpack.js.org/ ]
```

We'll answer the question "Which bundler would you like to use?" with "`esbuild`". Why? because it's faster. (Written in GoLang, optimised for JS/TS, lower memory footprint, native ESM support, well integrated with Nx,… etc. In the context of our monorepo, esbuild's speed and efficiency will become increasingly valuable as our project grows.)

Press Enter to confirm.

> **The Architect's Wisdom**
> While esbuild is very fast, it's worth noting that Rust-based **`Rolldown`** **github.com/rolldown/rolldown** is emerging as a potentially even faster alternative. For React apps, you would have **`Vite`**, **`Webpack`**, and **`Rspack`** as alternatives. While it would be possible to adapt Rolldown for Nx Angular apps with some effort, the community will likely implement this soon, so I'd wait for that. Currently, we know that Vite plans to replace both esbuild and Rspack with Rolldown. **rolldown.rs/guide/#why-rolldown**

2.6 Set Default Stylesheet Format

```
✓ Which stack do you want to use? · angular
✓ Integrated monorepo, or standalone project? · integrated
✓ Application name · elvis-deletable
✓ Which bundler would you like to use? · esbuild
? Default stylesheet format …
CSS
SASS(.scss)       [ https://sass-lang.com   ]
LESS              [ https://lesscss.org     ]
```

At this point, we'll select **`SASS(.scss)`** as our default stylesheet format. This choice is strategic because SASS offers several key advantages for enterprise applications:

— Powerful nesting capabilities and variables
— Reusable mixins and functions
— Better code organization through partials
— Built-in color manipulation
— Advanced math operations

We'll later provide global SCSS mixins, accessibility features, and responsive design patterns out of the box (I'm planning to write a dedicated ELVIS book about these topics).

In an enterprise workspace, developers should focus on implementing critical features without getting bogged down by UI design, responsiveness, and accessibility concerns. However, these crucial and complex aspects need to be professionally implemented from the very beginning.

While product managers prioritize features for implementation, design should come from designers and usability experts who naturally maintain close contact with product management (domain experts, or "Fachleute, Fachabteilung" as we say in German-speaking EU). The architect serves as the crucial bridge between these groups.

A Wannabe Mathematician's Design Awakening

In the projects I support, I advocate for using **Figma** as our design platform. While I'm not a Figma expert myself I know enough about Figma to demonstrate its value to business departments and designers (*despite my background in Theoretical Computer Science with focus on Mathematics and later Media Informatics (UI/UX) - and yes, until that second degree, I thought I could design! Reality was quite the eye-opener*).

Often though, this isn't even necessary as they're already familiar with Figma. BTW, we've developed efficient methods to seamlessly transfer these designs into Angular applications.

This is where our SCSS mixins prove their true value. The automation capabilities that SCSS offers are remarkable. I can't emphasize enough how underrated SCSS is - it doesn't receive the recognition it deserves in my opinion.

As the connecting element, the architect orchestrates the collaboration between product management (business departments), designers, and the development team.

Trust Me, I've Seen Things...

Allowing designers to work directly with developers, or letting developers handle design, would lead to disastrous results in 99% of cases[a].

[a] Dear Niklaus, I'm thinking of you in this very moment...

Alright, let's continue with the CLI interaction. When prompted for the stylesheet format, select **`SASS(.scss)`** and press Enter.

2.7 SSR and SSG/Prerendering Setup

The next question appears:

Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)?

```
✓ Which stack do you want to use? · angular
✓ Integrated monorepo, or standalone project? · integrated
✓ Application name · elvis-deletable
✓ Which bundler would you like to use? · esbuild
✓ Default stylesheet format · scss
? Do you want to enable (SSR) and (SSG/Prerendering)? …
Yes
No
```

Let's take a brief detour to explore these two topics and then return to this point.

2.7.1 Server-Side Rendering (SSR)

First, let's clarify Server-Side Rendering (SSR)[11]. For Angular experts reading this document - you're already familiar with all this. I'm just briefly recapping for completeness. Detailed information can be found on Angular's new website: **angular.dev/guide/ssr**

SSR is a technique where Angular renders the initial HTML on the server for each request. This HTML is then sent to the client along with the necessary JavaScript code. Once loaded, the JavaScript hydrates the application, adding interactivity to the pre-rendered HTML.

My journey with SSR has been interesting. In my early Angular implementations I couldn't optimize performance effectively and eventually abandoned it.

However, Angular now offers impressive improvements, particularly in Performance Metrics like **Time to First Byte (TTFB)**, thanks to optimizations in the rendering process.

Early Angular SSR hydration attempts were "destructive". This meant that the loaded JavaScript code didn't properly register event handlers with already rendered DOM elements. Instead, the framework simply deleted all DOM elements and started rendering from scratch. This is why older Angular applications show a brief "flash" during initialisation where the already rendered page momentarily disappeared.

Regarding hydration, there has been a focus on making the hydration process **non-destructive**, meaning the client-side JavaScript doesn't need to rebuild the DOM but instead enhances it, improving initial load time and user experience. New hybrid rendering APIs have been introduced, allowing developers to mix SSR with client-side rendering for better performance where needed.

[11] While I haven't extensively tested the newest features yet I've been following the development closely. I'm planning to write a dedicated document/book about this topic with other Angular experts.

The Art of Web Hydration 🍜

Ever wondered why your grandma's old chicken soup recipe took hours to make, while instant soup is ready in minutes? Well, web hydration is kind of like that instant soup magic, but for websites!

Picture this: You're staring at a bowl of dried soup powder (that's your static HTML from the server). Sure, it looks like soup, but try eating it.Then add hot water, and suddenly you've got yourself a proper soup. That's exactly what happens with web hydration, minus the actual soup.

When a server sends your browser a webpage, it's like getting that packet of dried soup. It looks like a website, but try clicking those buttons and… nothing happens. That's because it's just a static snapshot, as interactive as a printed screenshot. But then JavaScript swoops in like hot water to our rescue, bringing everything to life!

Suddenly those lifeless buttons spring into action, forms start working, and your website transforms from a digital statue into a living, breathing application.

Could we explain Angular's approach to SSR to a web expert without Angular knowledge? Yes, we can. No matter which library or framework we use, they all need a root anchor, an id where everything begins. In a React.js world, it would look something like this (highly simplified), assuming we have:

```
<div id="root">
  <!-- this is where the life begins -->
</div>
```

In Angular, it would look approximately like this (Angular makes it appear even more magical):

```
<app-root>
   <!-- this is where the life begins -->
</app-root>
```

A static content (without SSR) inside could look like this (now general and not framework-specific):

```
<div id="root">
  <div class="loading">Loading...</div>
</div>
```

And when dynamic parts (JavaScript, Data, …) are loaded, the DOM is modified. Here, the loading part disappears and is replaced by another element, that need at least 2 DOM operations: Delete and Insert, afterwards we would have:

```
<!-- Then after JavaScript loads and executes: -->
<div id="root">
    <div class="user-profile">
        <h2>Welcome, Inspector Clouseau</h2>
        <button class="likes-button">
            Likes: 0.5
        </button>
    </div>
</div>
```

What's the difference with SSR? Through SSR, the client would receive something like this - a piece of HTML and some JavaScript code, and this Code will soon bring the already loaded parts to life:

```
<!-- Static Part: What to show -->
<div class="user-profile">
    <h2>Welcome, Inspector Clouseau</h2>
    <button class="likes-button">
        Likes: 0
    </button>
</div>
<!-- Dynamic Part: What to do -->
<script>
document
  .querySelector(".likes-button")
  .addEventListener("click", () => {
    const currentLikes = parseInt(
      this.textContent.split(":")[1],
    );
    this.textContent = `Likes: ${currentLikes + 1}`;
      /**
      * Real implementation might include:
      * - API calls, State management,
      * - Real-time updates (incl. User Interaction)
      * - Error handling, Loading states
      * - Analytics tracking, and much more
      * */
  });
</script>
```

This means there's a mechanism (called hydration) that (when activated) always expects two components: a static part and a dynamic part. The static part provides immediate visual content, while the dynamic part adds interactivity through JavaScript. Unlike client-side rendering, hydration avoids expensive DOM replacement operations - the view is already there and only needs to be 'brought to life' with event handlers and state management. This is the essence of web hydration.

Now some readers might wonder why such a seemingly simple requirement can be so enormously complex? The answer is simple: **Web Hydration** is a complex process that involves multiple layers of abstraction, intricate browser APIs, and a delicate dance between server and client. It's a bit like a magic trick - it looks simple on the surface, but behind the scenes, there's a lot of sleight of hand going on.

This complexity arises from several intertwined challenges which we can principally categorize into 3 groups:

- Component Lifecycle
  - Initial HTML from server must match exactly what the client would render
  - Event handlers need precise reattachment without rebuilding the DOM
  - Component states must synchronise seamlessly
  - User interactions during hydration need careful handling
- Route Management
  - Server must handle the same routes as the client
  - Deep links must work immediately
  - Client-side navigation needs to take over smoothly
  - Route guards and resolvers must function correctly
- State & Data Flow
  - Server state must transfer to client without loss
  - API calls need deduplication
  - Client cache needs proper initialization
  - Real-time updates must integrate smoothly

While the audience of this document consists of Angular experts who are already familiar with most of these details, we won't delve into the depths here (although that's where it gets more interesting). Instead, we'll focus on setting up an Nx workspace.

But at this point, we should briefly mention Static Site Generation (SSG/Prerendering), as our Nx CLI asked: 'Do you want to enable Server-Side Rendering (SSR) and Static Site Generation (SSG/Prerendering)?' We've covered SSR, now let's look at SSG.

2.7.2 Static Site Generation (SSG/Prerendering)

SSG is like SSR's efficient cousin - instead of rendering pages on each request, it pre-renders them during build time. Imagine you're running an e-commerce site:

- SSR: Renders pages when users request them
- SSG: Renders all pages once during deployment

Key differences:

— Build vs Runtime: SSG generates HTML files during build, not per request
— Performance: SSG serves pre-built static files (extremely fast)
— Resource Usage: No server computation needed for each visit
— Perfect for: Documentation, blogs, landing pages, product catalogs
— Less ideal for: Highly dynamic content, real-time data

Think of SSG as a printing press (print once, distribute many) while SSR is more like a real-time translator (translates as needed). In Angular's implementation, you can use both strategies where they make the most sense.

2.7.3 Challenges

For Angular developers new to **SSR** and **SSG**, the journey involves several important adaptations to their development approach.

The first noticeable change comes with **build** and **deployment** processes. Instead of a single browser-targeted build, you'll work with multiple build outputs targeting different platforms. Your application needs to run both in the browser and on a Node.js server, which introduces new complexity to the deployment pipeline.

Code architecture requires a more thoughtful approach. Things that were once taken for granted, like direct access to browser APIs (window, document, localStorage), now need careful consideration. Your code must gracefully handle **both server and browser environments**, often requiring **platform-specific checks and conditional logic**.

Data handling becomes more sophisticated. You'll need to manage the flow of data between server and client, handle API calls appropriately in different environments, and implement effective caching strategies. The challenge lies in maintaining a **consistent state** across these different **execution contexts**.

The **development workflow** itself evolves. **Builds** take longer as they target **multiple platforms**. **Debugging** becomes more complex as issues might manifest differently on server versus client. **Testing** needs to account for both environments, and error handling must be environment-aware.

Common challenges include **avoiding infinite loops in lifecycle hooks**, **preventing memory leaks on the server**, and **handling browser-only third-party libraries**. These aren't insurmountable obstacles, but they require a broader understanding of the **application's execution context**.

The key is understanding that your application now lives in two worlds - **server** and **client** - and needs to transition smoothly between them. This dual-nature brings powerful benefits but requires a more nuanced development approach.

2.7.4 Architect's Notes

When do architects change their proven principles?

Most architects have a portfolio of diverse requirements and choose their proven tools according to the situation and goals. For me, **NestJS** is a proven tool, and I've had exclusively good experiences with it. However, even NestJS must sometimes make way for alternatives.

Consider scenarios where **raw performance** becomes critical - here, switching to **Go** or **Rust** can provide significantly better runtime performance. Enterprise environments often mandate **Java** for better integration with existing systems, especially when API teams develop exclusively in Java and need to maintain technological consistency across their stack.

The decision to switch can also be driven by specific technical demands: **microservices** requiring extreme **efficiency in resource usage**, platform requirements demanding **native capabilities**, or when **SSR/SSG** mechanisms simply offer more elegant solutions for particular use cases. **Real-time processing** with **strict latency requirements** might also necessitate a different technological approach.

These examples illustrate a broader principle: while we should value our proven tools, we must remain flexible enough to embrace better solutions when they **truly serve our specific needs**. The key lies not in changing technologies for change's sake, but in recognizing when a different approach genuinely better serves our architecture's goals.

When it comes to technology choices, I'm always ready to pivot when **quantum leaps** in **performance** or **complexity management** emerge. It's like upgrading from a horse and buggy to a Tesla - sometimes the jump is just that dramatic. However, the reverse can also be true - sometimes you **might trade in your Tesla** because what you actually need right now are some cows, sheep, or chickens. It's about **choosing the right tool for the current need**, not always about having the most advanced technology.

Our development landscape has seen **numerous transformative shifts** that reshaped how we work. **JQuery** gracefully bowed out as native **DOM APIs** evolved, while **SOAP** messages transformed into sleek **REST APIs** before **GraphQL** emerged to change the game again.

We witnessed the shift from **monolithic PHP applications** to modern microservices, and saw the evolution **from Angular.js to Angular 2+** - a revolution disguised as an upgrade. Even **Flash**, once ubiquitous, gracefully retired in favor of **HTML5**.

Today's quantum leaps are equally compelling: **Rust** offers tenfold performance improvements for **CPU-intensive tasks**, while **Go** reduces **memory footprint** by orders of magnitude in **high-concurrency scenarios**. **Bun** demonstrates significantly faster npm install times compared to **Node.js**, and **edge computing platforms** achieve **microsecond responses** versus traditional cloud deployments.

Meanwhile, the rapid evolution of **AI**, from **narrow AI** to the pursuit of **AGI** (Artificial General Intelligence) and theoretical **ASI** (Artificial Superintelligence), is reshaping how we think about software architecture and system design. The integration of AI capabilities has become **a game-changer**, transforming traditional programming paradigms and opening new possibilities in **automated coding**, **optimization**, and **decision-making processes**.

Sometimes, the factors that drive these changes **extend beyond pure performance metrics**. We're seeing fundamental shifts in how systems are designed, developed, and maintained.

While narrow AI is already enhancing our development workflows through tools like copilots and code analyzers, the potential impact of more advanced AI systems could revolutionize our entire approach to software architecture.

PHP: Permanently Heartwarming Platform

PHP has shown remarkable resilience, initially thanks to frameworks like Yii, Symfony, Laravel, and WordPress, and later through its community's incredible adaptability. It remains the go-to language for catalogs, blogs, and static CMS systems. PHP has become something of the COBOL of our time - steadfastly reliable and surprisingly enduring.

Even its name evolution tells a story of growth - from "Personal Home Page" to "PHP: Hypertext Preprocessor", following the clever recursive naming pattern popularized by GNU ("GNU's Not Unix"). Like many developers, I built my first websites with PHP and financed my university studies through PHP projects. It's fascinating - I've yet to meet anyone who has worked with PHP and doesn't maintain a fond appreciation for it.

This staying power demonstrates an important principle in technology: sometimes the most practical solution, even if not the most elegant, becomes a lasting one. PHP's pragmatic approach to web development has earned it a permanent place in our technological landscape.

That's why I simply had to write something for PHP. And now I need a tissue to wipe away my tears.

2.7.5 SSR & SSG in Nx: Decision Time

The focus of this document is Nest.js for Angular experts. We've made a conscious decision to maintain a clear separation: Angular for the client, Nest.js for the server. (While we could explore combining these technologies - and there's nothing technically preventing us from doing so - that discussion would exceed the scope of this document. We'll save that fascinating topic for the upcoming ELVIS book.)

```
✓ Which stack do you want to use? · angular
✓ Integrated monorepo, or standalone project? · integrated
✓ Application name · elvis-deletable
✓ Which bundler would you like to use? · esbuild
✓ Default stylesheet format · scss
? Do you want to enable (SSR) and (SSG/Prerendering)? …
Yes
No
```

Please select **No** here and press Enter.

2.8 Choose Your Test Runner (E2E)

```
✓ Which stack do you want to use? · angular
✓ Integrated monorepo, or standalone project? · integrated
✓ Application name · elvis-deletable
✓ Which bundler would you like to use? · esbuild
✓ Default stylesheet format · scss
✓ Do you want to enable (SSR) and (SSG/Prerendering)? · No
? Test runner to use for end to end (E2E) tests …
Playwright [ https://playwright.dev/ ]
Cypress [ https://www.cypress.io/ ]
None
```

Testing is an extraordinarily critical subject - one that could easily warrant its own book. While this topic isn't strictly part of this document, I'd like to take this opportunity to highlight some **key principles**.

2.8.1 Key Principles

You have complete flexibility in choosing any of these options for your E2E testing needs.

While I lean toward **Playwright** for **Azure Cloud** environments, **Cypress** shines brilliantly in its own right. Its intuitive interface is a masterpiece of design, offering an exceptional developer experience. The selector playground is particularly impressive - it makes element selection effortless and speeds up test creation significantly.

Corporate Policies Matter

As enterprise architects and developers, we understand a fundamental reality: tools must be pre-approved before they can be implemented. Having introduced numerous new tools, languages, and frameworks into traditionally conservative enterprise environments, I'm intimately familiar **with the challenges** this process presents.

When starting a new project, if the CTO indicates that technology X is off-limits and I know of a viable alternative Y, I'll pragmatically choose Y. However, when X is truly indispensable, we must strategically make our case.

This involves creating a **comprehensive**, yet **accessible comparison** of advantages and disadvantages, **presented visually** in a way that **resonates with board-level executives**.

Success in these situations requires more than technical merit - it demands clear communication of business value in non-technical terms. While **there's never a 100% guarantee of approval**, a well-structured, business-focused presentation significantly improves our chances of gaining support for essential technical innovations.

Remember: The key is choosing our battles wisely and presenting our case in a language that aligns with business objectives.

Newcomers to **E2E** testing often find **Cypress's learning curve** remarkably gentle, thanks to its **well-thought-out design and outstanding documentation**. The vibrant, knowledgeable **community** consistently provides top-tier support and valuable insights.

For Azure Cloud environments, I prefer Playwright due to its compatibility with **enterprise policies** regarding **binary downloads and pipeline permissions**. Its broader feature set proves particularly valuable in complex enterprise scenarios. (While obtaining special approvals for Cypress is possible, the process can be time-consuming and potentially delay project timelines.)

As an aside, **Puppeteer** would be a wonderful addition to this toolkit, especially for **AI development** where its capabilities extend beyond traditional testing. (A note to our **Nx Core Team** - considering Puppeteer would be appreciated.)

2.8.2 Challenges

Even brilliant architects can find themselves in challenging or seemingly absurd situations when navigating German-speaking EU enterprise companies without prior experience. The fundamental truth remains: software isn't properly validated without thorough testing.

Consider this scenario: Your team has already invested significant effort creating comprehensive E2E tests with Cypress. Then you discover that your CI/CD pipelines won't support Cypress binaries, and the tool hasn't received corporate approval. The consequence? All tests need to be rewritten - a classic own goal for the architect. Developers will understandably be frustrated, having dedicated substantial time to crafting excellent E2E tests. (*And if we're forced to use Selenium, that's a frustration I personally share*)

This creates a complex dilemma:

— If you delay E2E testing while waiting for framework approval, developers might question your architectural judgment
— If you implement tests that later need complete rewriting, you waste valuable development resources

The solution lies in proactive communication and planning:

— Inform the development team upfront about the approval process and potential tooling constraints
— Engage with product management early to reserve time and budget for potential testing implementation or migration
— Account for framework approval timelines (which could take 1 week to 3+ months) in the project planning
— Set aside dedicated sprints (minimum 2) for implementing E2E tests once tools are approved

This approach prevents surprises and conflicts with both development teams and product owners who might otherwise push back against "unexpected" testing efforts later in the project lifecycle.

Remember: In enterprise environments, technical decisions must align with both organizational policies and project timelines from the very beginning.

2.8.3 Decision Time

Choose the tool that best fits your specific context - both options have their strengths, and either could be the perfect choice depending on your needs.

```
? Test runner to use for end to end (E2E) tests …
Playwright [ https://playwright.dev/ ]
Cypress [ https://www.cypress.io/ ]
None
```

Make your choice and press Enter.

2.9 CI Provider Section

```
✓ Which stack do you want to use? · angular
✓ Integrated monorepo, or standalone project? · integrated
✓ Application name · elvis-deletable
✓ Which bundler would you like to use? · esbuild
✓ Default stylesheet format · scss
✓ Do you want to enable (SSR) and (SSG/Prerendering)? · No
✓ Test runner to use for end to end (E2E) tests · playwright
? Which CI provider would you like to use? …

      GitHub Actions
      Gitlab
      Azure DevOps
      BitBucket Pipelines
      Circle CI

      Do it later

Remote caching, task distribution and test splitting are provided by Nx Cloud. Read
↪  more at https://nx.dev/ci
```

While CI/CD configuration is a particularly engaging aspect of development (and one I personally enjoy), we'll select "Do it later" for this setup. This topic deserves comprehensive coverage and will be addressed in detail in the upcoming ELVIS book.

Select **`Do it later`** and press Enter to proceed.

2.10 Remote Caching

```
✓ Which stack do you want to use? · angular
✓ Integrated monorepo, or standalone project? · integrated
✓ Application name · elvis-deletable
✓ Which bundler would you like to use? · esbuild
✓ Default stylesheet format · scss
✓ Do you want to enable (SSR) and (SSG/Prerendering)? · No
✓ Test runner to use for end to end (E2E) tests · playwright
✓ Which CI provider would you like to use? · skip
? Would you like remote caching to make your build faster? …
(can be disabled any time)
Yes
No - I would not like remote caching

Read more about remote caching at https://nx.dev/ci/features/remote-cache
```

While remote caching is an excellent feature that significantly improves build performance, we'll select "No" here for a critical reason: Enterprise EU corporations typically have **strict data protection policies** that prevent using external cloud services for build artifacts. The rationale behind this limitation:

— **EU data protection regulations (GDPR)** require careful handling of any data leaving corporate networks
— Build artifacts might contain **sensitive information** or **intellectual property**
— External cloud services often conflict with corporate security policies
— Internal **audit requirements** often mandate that all build processes remain within controlled environments
— Many enterprises require that development artifacts stay within their own infrastructure

Even though Nx Cloud's remote caching could dramatically improve build times, the regulatory and compliance requirements of EU enterprise environments typically preclude its use. Most enterprises prefer to implement their own internal caching solutions that comply with their security policies.

Select **No** and press Enter to continue with the setup.

Order 66: When Caches Turn to the Dark Side
Accidentally choosing "Yes" to remote caching:

While technically possible to connect to Nx Cloud from your machine when working remotely (with VPN conveniently "disabled"), this might trigger some… interesting corporate responses:

Scenario A - Working From Home

At 4:30 AM, your peaceful sleep might be interrupted by a tactical security team, complete with dramatic blue lights illuminating your neighborhood. You'll have the unique experience of being escorted in full restraints while your neighbors get an early morning light show.

Scenario B - At the Office

A security team member, armed with what appears to be a butterfly net, will swiftly apprehend you. You'll be expertly guided to the exit, while your laptop - now considered a potential security threat - is whisked away for "further investigation." **Remember:** In enterprise security, there's no such thing as "it's just a build cache."

2.11 Final Confirmation

```
✓ Which stack do you want to use? · angular
✓ Integrated monorepo, or standalone project? · integrated
✓ Application name · elvis-deletable
✓ Which bundler would you like to use? · esbuild
✓ Default stylesheet format · scss
✓ Do you want to enable (SSR) and (SSG/Prerendering)? · No
✓ Test runner to use for end to end (E2E) tests · playwright
✓ Which CI provider would you like to use? · skip
✓ Would you like remote caching to make your build faster? · No

 NX   Creating your v20.3.0 workspace.

✓ Installing dependencies with npm
⠸ Creating your workspace in elvis
✓ Successfully created the workspace: elvis.

 NX   Welcome to the Nx community! 👋

🌟 Star Nx on GitHub: https://github.com/nrwl/nx
📢 Stay up to date on X: https://x.com/nxdevtools
💬 Discuss Nx on Discord: https://go.nx.dev/community
```

3 Add NestJS to Your Workspace

3.1 Open Your Workspace in VS Code

Let's switch to our Nx workspace (in our case, it's named elvis) and launch VSCode from the terminal. For those who haven't set up the command line shortcut yet, you can refer to the "Install Recommended Software" chapter.

```
cd elvis
code .
```

When you click the **Explorer** icon in the `Activity Bar`:

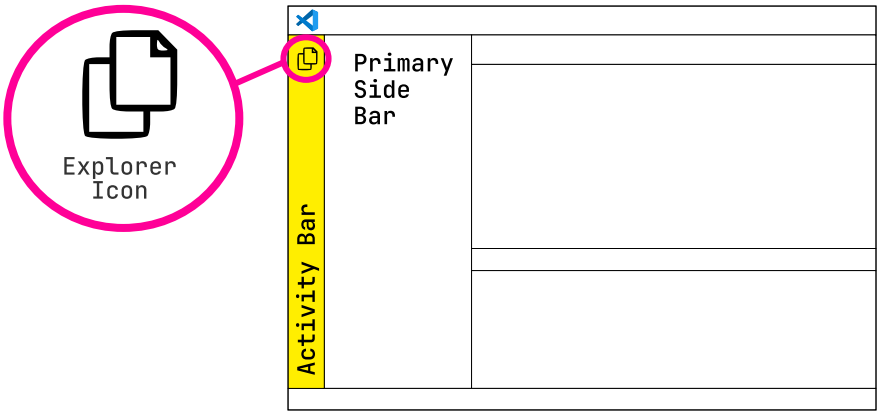


Figure 6: VSCode Explorer Icon & Primary Side Bar

(refer to the VSCode chapter for detailed interface terminology - these terms are essential as we'll use them frequently), you should see this content in the primary side bar: