ChatGPT

Unified Brand and Technical Specification for Monynha.com

Table of Contents

1. **Brand Identity and Narrative**
   1.1 Linguistic Meaning of "Monynha"
   1.2 Community Values: LGBTQIA+ Inclusion and Class Solidarity
   1.3 Mission Statement: Democratizing Access to Technology

2. **Technical Architecture**
   2.1 Frontend Stack (Next.js 14, Tailwind CSS, shadcn/UI)
   2.2 Backend and Database (Supabase PostgreSQL)
   2.3 Content Management Integration (Payload CMS)
   2.4 Architecture Principles (Clean Architecture, DRY, API-First)
   2.5 Internationalization (Multilingual Support EN/PT-BR/ES/FR)
   2.6 Authentication Strategy (Single Sign-On via Monynha.me)

3. **Site Requirements and Structure**
   3.1 Primary and Secondary Routes Overview
   3.2 Dynamic Content Models (Products, Blog Posts, Translations)
   3.3 Navigation and Hub Interface (Mega-Menu & Domain Links)
   3.4 Database Schema and Security (Tables and RLS Policies)

4. **User Experience (UX) and User Interface (UI) Identity**
   4.1 Interaction Patterns and Playful Micro-Interactions
   4.2 Visual Feedback, Scroll Effects, and Transitions
   4.3 Accessibility, Theming, and Responsive Design (WCAG 2.1 AA, Dark Mode)
   4.4 Design Tokens and Consistent Branding

5. **AI-Assisted MVP Generation (Lovable Prompt)**
   5.1 Full-Stack Generation Approach
   5.2 Crafting the Technical Prompt (Stack, Packages, and Schema)
   5.3 MVP Feature Priorities and Scope
   5.4 Project Structure and Next Steps

---

1. Brand Identity and Narrative

1.1 Linguistic Meaning of "Monynha"

The brand name **Monynha** carries a playful and culturally rich connotation. In Portuguese, adding the suffix "-inha" is a diminutive form often used to convey affection or endearment [1]. The name "Monynha" thus suggests something cherished or beloved, setting a warm and approachable tone for the brand. It evokes familiarity – much like calling someone by a nickname – and signals that Monynha

Softwares is friendly, non-corporate, and community-oriented. By adopting a diminutive, the brand positions itself as humble yet personable, distinguishing it from more impersonal tech companies. This linguistic choice is intentional: it aligns with Monynha's goal of making technology feel more human and accessible. The **Monynha** name is meant to be easy to remember and say, inviting users from different cultures to engage with it without intimidation.

Beyond the affectionate tone, the name may also carry personal significance within the company's culture. It could be a homage to a community nickname or simply a creative moniker chosen to stand out. Regardless of origin, **Monynha** sets the stage for a brand identity that is welcoming and inclusive. It hints at Latin roots (with the Portuguese/Spanish-sounding "-inha"), reflecting the company's connection to Lusophone (Portuguese-speaking) culture. This resonates especially with Brazilian Portuguese speakers, where **"moninha/monynha" might be interpreted as an endearing term for someone or something cute and loved**. The use of such a name underscores the brand's commitment to warmth and relatability – values that carry through to its design and messaging.

1.2 Community Values: LGBTQIA+ Inclusion and Class Solidarity

Monynha Softwares is not just a tech brand; it is built on a strong foundation of social and political values. Chief among these is a deep **commitment to the LGBTQIA+ community**. The brand embraces diversity and strives to be a safe, empowering presence for queer individuals in the tech space. This is reflected in internal policies (inclusive hiring, representation) and external messaging (support for Pride events, inclusive imagery and language). Monynha draws inspiration from the resilience and creativity of LGBTQIA+ culture – emphasizing that technology should uplift marginalized voices rather than exclude them. In line with this ethos, Monynha's narrative echoes missions of organizations like LGBT Tech, aiming to *"empower and uplift LGBTQ+ individuals, ensuring [they have] the tools, resources, and access they need to thrive"* [2]. By aligning itself with LGBTQIA+ empowerment, Monynha positions its platform as one where everyone, regardless of gender or sexual identity, can participate in and benefit from technology.

Alongside queer inclusion, **class solidarity** is a core pillar of Monynha's brand narrative. The company is outspoken about the need to bridge the digital divide and ensure that working-class and disadvantaged communities have equal access to technological opportunities. In practice, this means Monynha Softwares supports open-source projects, offers affordable (or free) services, and provides educational resources to those who historically have been left behind by the tech industry. The name "Monynha" itself, with its approachable vibe, signifies a break from elitist tech culture – it's technology by the people, for the people. The brand often highlights that technology should **not** be a luxury or privilege, but a common good. This philosophy echoes the broader concept of *digital inclusion*, defined as providing access and use of digital technologies to all people regardless of socio-economic background [3]. Monynha actively works to *"help everyone, especially those who are disadvantaged or historically excluded, to have the access and skills to fully participate in the digital world."* [3] In practical terms, this could involve community tech workshops, sliding-scale pricing, or collaboration with non-profits to distribute hardware and knowledge.

By fusing LGBTQIA+ advocacy with class consciousness, Monynha Softwares crafts a unique narrative of solidarity. The company culture celebrates intersectionality – understanding that factors like class, race, gender identity, and sexual orientation can compound to create barriers in tech. Monynha's story is about tearing down those barriers. Whether it's sponsoring a hackathon for underprivileged queer youth or implementing policies that ensure its software runs on low-cost hardware, the brand consistently infuses its social values into its business decisions. **Inclusivity** is not just a buzzword for Monynha; it's an operational principle. The brand's visual identity (rainbow accents or other pride symbolism, alongside imagery of diverse users) and content (blog posts on feminist and anti-racist tech perspectives, for example) reinforce this stance. In summary, Monynha Softwares stands as a politically aware tech entity – one that believes empowering marginalized communities and fostering class solidarity will lead to richer innovation and a fairer digital future.

1.3 Mission Statement: Democratizing Access to Technology

All the above elements – the affectionate name, the inclusive values – coalesce into Monynha's overarching mission: **to democratize access to technology**. This mission statement is the guiding star for the company's projects and initiatives. Democratizing access means making technology available and understandable to as many people as possible, removing traditional barriers such as high cost, steep learning curves, or exclusive gatekeeping of knowledge. The company often states that high-quality software and digital tools should not be the realm of only large corporations or wealthy individuals, but should be within reach for the broader populace. This philosophy is in harmony with the open-source movement, where sharing and collaboration *"have not only democratized access to technology but also fostered a culture of transparency, inclusivity, and community-building."* [4] Monynha leverages open-source frameworks and contributes back to the community, aligning its mission with the idea that freely shared knowledge in tech leads to empowerment for all.

In practice, the mission of democratizing tech access manifests in several ways. First, Monynha.com itself is envisioned as an educational and welcoming portal – featuring multilingual content (to break language barriers) and straightforward explanations of technical concepts (to break knowledge barriers). The platform might provide free resources or guides to help newcomers learn about software development or digital literacy. There is a clear parallel to many global efforts where providing *"tools, resources, and access [for communities] to thrive"* is key [2] . For Monynha, that could mean offering a free tier for its products, or building software specifically tailored to non-profit and community uses.

Secondly, the mission shows up in the product design: Monynha's software prioritizes user-friendliness and accessibility (as detailed in later sections of this document). The rationale is that complex, unintuitive tools can alienate people who are new to technology. To truly democratize tech, Monynha Softwares designs interfaces that are intuitive and inclusive for a wide range of users – from seasoned developers to first-time internet users. This includes adhering to accessibility standards and providing support in multiple languages, as well as ensuring performance on low-end devices. It's about meeting users where they are.

Finally, *democratizing access* also means advocacy. Monynha uses its platform to advocate for policies and standards that keep technology open and fair. For instance, the company vocally supports **net neutrality, open internet initiatives, and digital rights**, understanding that systemic issues can impact equal access. Internally, Monynha's diverse team and collaborative culture reflect the belief that *innovation flourishes when everyone has a seat at the table*. By emphasizing class solidarity, the company also situates itself as an ally to labor and community movements, possibly structuring as a cooperative or engaging in profit-sharing to live its values.

In summary, Monynha Softwares' mission of democratizing technology is not an empty slogan – it is evidenced by concrete commitments: **open-source ethos, multilingual and accessible design, community outreach, and advocacy for an inclusive tech ecosystem**. The brand identity and narrative all serve this mission. Monynha wants to be known as *"the people's tech company,"* where empowerment of the many outweighs the profit of the few. Every chapter of the technical specification that follows, from architecture choices to UX design, is influenced by this mission, ensuring that Monynha.com is built not just as a corporate website, but as a manifestation of these deeply held ideals.

---

2. Technical Architecture

Monynha.com's technical architecture is designed to support the brand's ambitious mission with a modern, robust stack. The architecture follows best practices for scalability, maintainability, and performance, while also prioritizing developer experience and long-term adaptability. At a high level, the system can be described as a **full-stack web application** with a Next.js frontend, a PostgreSQL-based backend (via Supabase), and a headless CMS for content. It embraces a **clean architecture** approach where concerns are separated, and it adheres to principles like DRY (Don't Repeat Yourself) and API-first design for a well-organized codebase. This section provides a breakdown of each major component and how they integrate:

2.1 Frontend Stack (Next.js 14, Tailwind CSS, shadcn/UI)

The frontend of monynha.com is built with **Next.js 14**, leveraging the latest features of the Next.js App Router and React. Next.js was chosen for its hybrid rendering capabilities (Server-Side Rendering, Static Site Generation, and Server Components), built-in routing and internationalization support, and seamless developer experience. Version 14 of Next.js, in particular, offers significant improvements such as stable **Server Actions**, enhanced performance with the Turbopack compiler, and partial prerendering for dynamic content streaming. The App Router architecture of Next.js allows for a modular structure of pages and layouts. It improves project organization and code reuse by enabling nested layouts and nested routes by default. This yields a clear, maintainable structure where features like shared headers or footers can be defined once and inherited across pages – reducing redundancy and making updates easier. The benefits of Next.js App Router include *"improved organization"* of code, *"enhanced reusability"* through shared components, and *"better performance"* via optimized loading and caching [5] [6]. In short, the App Router is *"a significant enhancement for developers looking to build scalable, organized, and high-performance applications."* [7]

For styling, **Tailwind CSS** is utilized, which is a utility-first CSS framework. Tailwind provides a comprehensive set of small, composable classes (for example, classes like `flex`, `pt-4` for padding, `text-center`, `bg-blue-500`, etc.) that can be applied directly in JSX to rapidly build custom designs without writing traditional CSS. This approach keeps styling co-located with components and significantly speeds up development while ensuring a consistent design language. As the official Tailwind documentation describes, it's *"packed with classes like flex, pt-4, text-center and rotate-90 that can be composed to build any design, directly in your markup."* [8] Tailwind's utility classes map to theme variables, ensuring that the design is cohesive. The framework also supports *responsive design* out of the box by allowing prefixing classes with breakpoints (e.g., `md:w-1/2` for medium screens) [9], and it handles dark mode theming via prefix (`dark:`) as well [10] – which aligns with our requirements for theming and accessibility. Tailwind's popularity and large ecosystem of plugins also mean we can easily extend it for custom needs (such as forms, typography, etc.). By using Tailwind, Monynha's developers can avoid context-switching into separate CSS files and can *"design web pages with custom styles with a minimal amount of code,"* as one guide noted [11]. This contributes to adhering to DRY principles, since common spacing, color, and typography decisions are abstracted into re-usable classes rather than repeated in multiple CSS definitions.

On top of Tailwind, we incorporate **shadcn/UI**, which is a collection of accessible React components styled with Tailwind and powered by Radix UI primitives. shadcn/UI provides a set of beautifully designed, pre-built components (like buttons, modals, dropdowns, etc.) that are easy to integrate and customize. One key advantage of shadcn's approach is that it's not a conventional npm library – instead, one can pull the component code directly into the project (via a CLI) and own that code. This avoids heavy dependencies and allows full control over styling and customization. As a description notes, *“Shadcn UI provides re-usable components built on top of Radix UI and Tailwind CSS. It’s highly customizable… It does not add another dependency – you fetch only the components you need, which then become part of your codebase.”* [12] [13] . By using Radix UI under the hood, all components come with baked-in accessibility and correct structure (Radix provides unstyled accessible primitives like dialogs, dropdowns, etc., and shadcn adds Tailwind styling to them). This means we get the best of both worlds: a consistent design (thanks to Tailwind) and robust, accessible interactions (thanks to Radix). The use of shadcn/UI accelerates development; instead of building common UI components from scratch, developers can scaffold them and then tweak as needed – again aligning with the DRY principle by not reinventing the wheel for standard UI patterns.

The frontend stack also includes standard **React 18+** features (with which Next.js is compatible), so we make heavy use of React hooks, context for global state (though much of state is server-managed via Next’s data fetching or client-side via React Query if needed), and possibly upcoming features like React Server Components (RSC) through the App Router. The combination of Next.js + Tailwind + shadcn/UI creates a highly productive environment. For example, building a new page template might involve using a pre-built shadcn card component, utility classes from Tailwind to adjust spacing, and Next’s file-system routing to create the page – a process that is fast and yields consistent results. Moreover, it ensures that the frontend remains **responsive and mobile-friendly** by default (Tailwind’s mobile-first approach encourages designing for small screens up front). All static assets and images will be optimized via Next’s built-in optimization (Next.js Image component or equivalent in v14), ensuring performance best practices.

In summary, the **Monynha.com frontend** stands on a modern, efficient stack: **Next.js 14 (App Router)** for structure and performance, **Tailwind CSS** for rapid and uniform styling, and **shadcn/UI (with Radix)** for accessible, pre-styled components. This choice of stack ensures a developer-friendly workflow and results in a fast, interactive UI that aligns with Monynha’s brand (playful and responsive) while being robust under the hood.

2.2 Backend and Database (Supabase PostgreSQL)

On the backend, Monynha.com utilizes **Supabase**, which serves as a scalable PostgreSQL database with additional backend as a service features. Supabase was selected for its ability to provide a reliable SQL database (Postgres) along with convenient APIs and authentication, all while being open-source and self-hostable. At its core, Supabase gives us a **hosted Postgres** database that we can interact with either directly via SQL or through Supabase’s auto-generated APIs (RESTful endpoints and client libraries) for our data. This database is the single source of truth for structured content such as product information, blog posts metadata, user submissions (e.g., contact form entries), and any other data models defined in the site requirements. One of the strengths of Supabase is that it pairs Postgres with features like built-in **Row Level Security (RLS)**, which is critical for enforcing fine-grained access control to our data directly at the database level (we discuss RLS policies in section 3.4). Supabase’s documentation emphasizes that enabling RLS ensures secure data access from the client side: *“Supabase allows convenient and secure data access from the browser, as long as you enable RLS. RLS must always be enabled on any tables stored in an exposed schema.”* [14] . This aligns well with our API-first approach, where the frontend might communicate directly with the database through Supabase’s API for certain read operations – we can safely do so because RLS policies will guard the data.

The database schema itself will be detailed in section 3.4, but to summarize: there are tables for **Products**, **BlogPosts**, **Translations**, **Careers (job listings)**, etc., each with carefully designed relationships and constraints. PostgreSQL’s relational integrity and ability to use SQL for complex queries allow us to implement robust data logic (for example, we can query all products with their translated fields in one go via SQL JOINs on a Translation table). We also leverage **PostgreSQL’s JSONB**

columns where appropriate for flexible data (if some content has varying schema or for storing localization JSON, etc.). Supabase does not hide the fact that it's essentially Postgres – we have full access to write stored procedures, use full-text search, and even PostGIS if needed. This power and familiarity of Postgres make it an excellent choice for long-term maintainability.

Beyond the raw database, Supabase provides additional services that we plan to utilize: - **Auth**: Supabase has a built-in authentication module (email/password, OAuth providers, etc.). However, in Monynha's architecture, primary auth is handled by an external service (monynha.me SSO, see section 2.6). We may still use Supabase Auth for server-to-server or internal needs, but user-facing login goes through monynha.me. If needed, Supabase Auth could serve as a fallback or for service accounts. - **Storage**: Supabase offers S3-like object storage. This is useful for any media assets (images, PDFs, etc.) that might be uploaded via the CMS or other parts of the site. For example, if blog posts include images, those could be stored in Supabase Storage buckets. The advantage is that Supabase ties the permissions of these files to the same auth and RLS system – simplifying access control. - **Realtime**: If we require real-time features (perhaps showing live data feeds or user comments in future), Supabase's realtime (built on Postgres WAL) could be used. At MVP, this might not be necessary, but the infrastructure allows it without needing an external Pub/Sub system. - **Edge Functions**: Supabase can host serverless functions if we need custom server logic that shouldn't run on the client. This might be useful for webhooks (e.g., a form submission triggering an email or Slack notification) or integrating with external APIs securely.

The backend is essentially **"serverless"** from our perspective in that we are not maintaining a separate Node.js server for Monynha.com. Next.js will use its server capabilities (via API routes or server actions) for any dynamic logic, but heavy data lifting is done directly by querying Supabase. This makes the architecture lean – fewer moving parts – and scalable, because Supabase can handle a large number of concurrent connections and can be scaled (it's built on cloud Postgres). Notably, Supabase allows direct from-browser data fetching (with auth) which fits our needs for an API-first approach where our front-end can directly pull data securely. For example, on a static generated page, we might use Next.js API routes to fetch data, but on a client side filter component, we could call Supabase's JavaScript client to query more items on the fly, protected by RLS.

One of the key reasons for choosing Supabase is the ease of integration with Payload CMS (section 2.3). Payload can use an external Postgres database – in our case, the **Supabase Postgres acts as the storage for CMS content**. A guide from Payload notes that *"Supabase offers both a PostgreSQL database and multiple S3 storage buckets, making it an excellent choice for your project"* [15] and indeed demonstrates steps to connect Payload to Supabase. By doing this, we have a single database for both CMS-managed content and application-specific data, which avoids data silos and duplication.

In summary, the **backend architecture** revolves around **Supabase's PostgreSQL**. It provides the reliability of an ACID-compliant relational database, the convenience of managed services (Auth, Storage), and aligns with our open ethos (Supabase is open source, which means we're not locked into a proprietary system – we could self-host if needed). Monynha.com's backend can scale vertically (bigger DB instance) or horizontally (read replicas, caching) as traffic grows. Combined with Next.js frontend and edge deployments, this backend forms a strong foundation that can handle our content, user data, and interactions securely and efficiently.

2.3 Content Management Integration (Payload CMS)

For managing content on the site (such as blog articles, product pages, static page sections, etc.), we integrate **Payload CMS** as a headless content manager. Payload CMS is a self-hosted, TypeScript-based CMS that is often lauded as a great choice for Next.js projects due to its flexibility and "code-first" approach. Unlike traditional monolithic CMS (WordPress, etc.), Payload operates purely as an API and admin UI; it does not render the site itself. This fits perfectly with Monynha.com's architecture, where Next.js will fetch content from Payload via API and render it in the frontend.

Why Payload CMS? Several reasons led to its selection: - *Seamless Next.js compatibility*: Payload is designed to work smoothly with Next.js. It supports Next-specific features like Incremental Static Regeneration (ISR), Server-Side Rendering, etc. in the sense that its content API can be used to generate static pages or provide on-demand revalidation. In fact, Payload's documentation highlights compatibility with Next's rendering modes: it's *"fully compatible with Next.js's key features like ISR, SSG, and SSR"*, enabling high-performing dynamic websites [16] . - *API-First and Self-Hosted*: Payload provides REST and GraphQL APIs out-of-the-box for all content types. This aligns with our API-first principle – all content (e.g., a blog post's title, body, etc.) can be fetched through an API, which Next can call during build time or runtime. Payload being self-hosted means we can run it on our own server or as a serverless function, giving us control over data. It stores content in our database (Postgres/Supabase), so the content lives alongside other site data. This is efficient and ensures consistency (for instance, we could join CMS data with other custom data via SQL if needed). - *Code-Based Schema*: Content types in Payload are defined in code (JavaScript/TypeScript schemas). This means we version control the content schema as part of our repository – any changes to a collection (content type) are done via code changes, not via clicking around in an interface. This encourages a disciplined, predictable development process and fits with our "infrastructure as code" mentality. We can review and audit schema changes easily. - *Extensibility*: Because Payload runs as a Node.js application, we can extend its functionality with hooks, access control functions, and even custom Express routes if needed. It's basically a Node app using Express and Mongo/PG under the hood, giving us the flexibility to write custom logic in the CMS if our use case demands it (for example, auto-generating slugs, sending notifications on new content, etc.). - *Admin UI/UX*: Payload provides a robust admin panel out of the box for content editors. This admin panel will be accessible (likely at `monynha.com/admin` or a subdomain) and allows non-developers to enter and manage content. It's a single-page React app that is intuitive and can be customized or themed to match our brand.

Monynha.com uses Payload to manage key collections such as: - **Products**: Each product entry (with fields like name, description, features, perhaps pricing info, and links). These might correspond to individual pages under `/product/[slug]`. - **Blog Posts**: Each post with fields (title, content, author, date, etc.). These feed the `/blog` listing and individual post pages. - **Pages/Sections**: Possibly for static pages like About, Careers, etc., we can have a collection or global object in Payload to manage their content (text, images). Payload supports "globals" which are single documents, ideal for something like a global site footer content or an About page content. - **Navigation/Menu Items**: Could even be managed via CMS if we want non-developers to edit menu links. - **Translations**: If we manage multi-language content through the CMS (one strategy is to have fields for each language, or a separate collection for translations of certain strings).

Payload stores this data in the connected PostgreSQL (Supabase) database. The integration with Supabase is straightforward because Payload just needs a Postgres connection string – it doesn't know it's Supabase specifically, it just sees a Postgres. A step-by-step guide from Payload's site even demonstrates using Supabase as the Postgres and S3 storage for Payload [17] [18] . We have followed that approach: once Payload is set up with Supabase, any media files uploaded in the CMS (like images for blog posts) can be stored in the Supabase storage bucket, and any new content is inserted into the Supabase database tables that Payload auto-generates.

One notable aspect is **clean separation of concerns**: Next.js will not directly connect to the database for content, but rather go through Payload's API. This ensures that any business logic or rich text formatting that Payload applies is preserved. For example, if a blog post content is written in rich text

(Payload uses a rich text editor that can produce JSON or HTML), the Next front-end will retrieve that and render appropriately (perhaps using a React renderer for the JSON if needed). Because Payload's API is well-structured, developers can easily fetch exactly what's needed. In many cases, we can use **Static Generation**: Next could pre-fetch all blog posts at build time via Payload API (using an API key or public read token) to generate static pages for maximum performance. For frequently updated content, we might use ISR (so content updates reflect after a revalidation window) or even on-demand revalidation triggered by Payload's webhook (Payload can send a webhook to our Next app on content publish, prompting re-build of that page).

From a principles standpoint, integrating Payload reinforces DRY and clean architecture: content management (creation, editing) is handled by the CMS, and content presentation is handled by Next.js. They communicate via a well-defined interface (HTTP APIs), which is essentially the implementation of an **API-first design** – the content can be accessed through REST or GraphQL without coupling the front-end to CMS internals [19]. This means, theoretically, other services (say a mobile app in the future) could also consume the same content API without scraping the website.

Security: Payload has its own user roles and access control. We will configure it such that only authorized staff can log into the admin panel to make changes. The content that is exposed via API can be mostly public (e.g., blog posts), and for any sensitive content, Payload's access control functions can restrict who can fetch it. Since our site is mostly public-facing content, we might make many Payload API routes publicly readable (for easier integration). Regardless, those API endpoints are separate from the front-end, maybe hosted under `/api` routes or a subdomain, and can be protected as needed (for instance, using API keys on server-side requests).

Performance: Payload's content API is fast (especially when backed by Postgres). We will likely deploy Payload as a separate process (maybe on the same Vercel deployment via a Serverless Function, or on a separate serverless container on AWS). Considering Next.js is the one serving the user, Payload can be treated as an internal service. The overhead is minimal as long as responses are cached or pages are pre-rendered.

In sum, **Payload CMS** provides Monynha with a powerful content backbone. It ensures non-developers can easily maintain the website content through a friendly UI, while developers appreciate that everything is strongly typed (we get TypeScript types for our collections), version-controlled, and integrated with our stack (Next and Supabase) smoothly. By using Payload, we achieve a balance of **flexibility** (we're not constrained by a SaaS CMS schema; we define exactly what we need) and **structure** (there's a central source for content, and it's delivered via robust APIs). This clean separation of content concerns will make Monynha.com easier to maintain as the content grows, all the while keeping the actual website code free of hardcoded content.

2.4 Architecture Principles (Clean Architecture, DRY, API-First)

Monynha.com's codebase is structured following **clean architecture** principles and other software best practices to ensure that the system is modular, easy to understand, and easy to extend. The essence of clean architecture (as popularized by Robert C. Martin/Uncle Bob) is to separate the software into layers that have clear responsibilities, with the rule that higher-level policies do not depend on lower-level details. In our context, we interpret this as separating: - **Framework/Platform-specific code** (like Next.js pages, UI components) from **domain logic** (for instance, decisions about what content to show or who can access what). - **Presentation layer** (the React components, pages) from **data layer** (services that fetch from the CMS or database).

Concretely, the project will have a structure where, for example, there might be a directory like `services/` or `lib/` that contains code to communicate with Supabase or Payload (APIs). The Next.js page components will call these service functions rather than embedding fetch calls everywhere. This makes it easier to change implementations (say, switch CMS or change a query) without touching all the components. It also facilitates testing – one could test the service logic in isolation.

We also implement the **Dependency Inversion principle** in a way by possibly defining interfaces for certain operations (for example, an interface for a ContentRepository that could be implemented by calls to Payload now, but could be swapped later). This might be a bit heavy for an initial build, but even at a simpler level, we ensure no hard-coded dependency paths that would make a refactor difficult.

A key principle we embrace is **DRY – Don't Repeat Yourself**. This means we actively refactor to eliminate duplicate code. Any piece of logic or content that appears in more than one place will be abstracted, either as a reusable component, a utility function, or a configuration stored centrally. According to the DRY principle definition, *"every piece of knowledge must have a single, unambiguous, authoritative representation within a system"* [20]. For example, if the primary navigation menu appears on every page, we will not manually write the HTML for it on each page; instead, we'll create a `<NavMenu>` component that is included where needed. If the contact email address is needed in multiple places (footer, contact page text), we'll avoid hardcoding it twice – perhaps storing it in a config or pulling from CMS global settings, so that one change updates everywhere. This not only prevents inconsistencies but also makes future changes quicker and less error-prone (change one place instead of many). DRY extends to backend as well – for instance, if both the front-end and a backend script need a list of supported languages, we'll define it once (maybe in a shared config file) rather than duplicating the list.

Monynha's architecture is also **API-First**. By API-first, we mean that all functionalities are designed and exposed through APIs, making the web interface just one of the possible consumers of these APIs. We treat the API as a first-class citizen in the development process [21]. In practice, this is seen in how we use Payload CMS and Supabase – any piece of data that the site needs is fetched via an API (be it Payload's REST API or Supabase's client). We could imagine down the line an external developer might want to retrieve Monynha's product list for a mashup – our commitment is that such data is accessible through documented endpoints, not only by scraping HTML. We document the endpoints (for internal use at least) and make sure every new feature considers how it can be offered as an API. For example, when building a contact form submission, instead of the form directly emailing someone via a traditional server rendered page, we might implement it as an API endpoint (`POST /api/contact`) that accepts form data and returns a success or error. The front-end form calls this API via AJAX. This way, the submission logic is cleanly separated and could be reused or called from different front-ends (imagine a mobile app calling the same contact API).

An **API-first design** also influences how we think about the data models: we design the schema and endpoints first (defining what a "Product" contains and how one can query products, etc.), possibly even using tools like OpenAPI/Swagger for internal planning. This approach ensures we've thought through the contracts between front-end and back-end before getting lost in UI details. It's noted that *"API-first design is a strategic approach where the API is treated as the first-class citizen in the development process"* [21], which sums up our philosophy: any feature or content on Monynha.com is available through a stable API call. This also means our Next.js frontend is somewhat decoupled – if in the future we rebuilt the front-end in another framework, the underlying content and functionality could remain accessible via the same APIs.

We also follow general **separation of concerns**: For instance, environment-specific configurations (API keys, base URLs, etc.) are kept in environment variables or config files, not scattered in code.

Presentational components are separated from container components that handle data fetching (e.g., a `BlogList` component that only knows how to display a list of posts, versus a page component that fetches the posts then passes to `BlogList`). This makes it easier to test and to reuse components.

Additionally, to maintain code quality, we enforce **TypeScript** throughout (given Next.js and Payload both work well with TS). This provides type safety – ensuring that if we change the shape of data from an API, the TypeScript types will alert us of any mismatch across the app. It's part of maintainability to catch errors at build time rather than runtime.

Another facet of clean architecture is being **framework-agnostic in core logic**. While we are using Next.js, if there are pure functions or critical logic (say, a function that calculates something domain-specific like eligibility for a certain offer), we implement it in a plain TypeScript module that doesn't rely on Next or React. That way, it could be reused in a script or tested via Jest without needing a browser environment.

In summary, the architectural approach is to keep the system **modular, cohesive, and decoupled**: - **Modular**: Organized into folders/modules (e.g., `components`, `pages`, `lib/services`, `cms`, etc.) with clear roles. - **Cohesive**: Functions and classes do one thing well (Single Responsibility principle). - **Decoupled**: Minimizing tight coupling between different parts. For example, the UI should not directly depend on how data is stored; it just calls a service. The service is the only part that knows about Supabase or Payload specifics.

By adhering to Clean Architecture and DRY, we ensure that as Monynha.com grows (more pages, more features), we won't end up in a tangle of spaghetti code. Instead, we will have a system where changes are localized: e.g., a change in a data field might only require updating the Payload schema and adjusting one service function and one component, rather than editing dozens of files. This disciplined approach might add a bit of upfront overhead (in creating abstractions or extra layers), but it pays off in resilience and agility. It's about being **future-proof** – making the codebase easier to onboard new developers and to adapt to new requirements (like adding a mobile app or switching a provider) with minimal pain.

2.5 Internationalization (Multilingual Support EN/PT-BR/ES/FR)

Monynha.com is designed as a **multilingual** website, catering to an international audience by offering content in multiple languages. English (EN) will serve as the default language, with Portuguese (Brazilian Portuguese, PT-BR), Spanish (ES), and French (FR) as additional supported locales. The internationalization (i18n) strategy ensures that users can navigate and consume content in their preferred language and that the site's structure and content are adaptable to localization needs.

Locale Routing: We leverage Next.js's built-in i18n routing capabilities to handle locale-based routes. Next.js allows us to configure a list of supported locales and a default locale in `next.config.js`. Once configured, Next will automatically prefix routes with the locale (except for the default locale if configured not to) and handle domain or subpath routing. For example, our homepage will be accessible as `/` (which might serve English by default) and also as `/pt-BR`, `/es`, `/fr` for the localized versions. If a user with a browser locale setting of French visits, Next.js can automatically redirect to the French version of the site (unless overridden), thanks to its locale detection. As the Next.js guide notes, *"you can provide a list of locales, the default locale, and domain-specific locales and Next.js will automatically handle the routing."* [22] . This simplifies our work – we don't need to manually create logic for locale subpaths; Next handles it in the routing layer.

Content Localization: There are two types of content to consider – static and dynamic. - *Static interface text*: This includes navigation labels, form labels, headings, etc., which are part of the UI chrome. For these, we will use a JSON-based translation approach or one of Next.js compatible libraries (like `next-intl` or `react-intl`). For instance, there will be translation files for each locale (e.g., `en.json`, `pt-BR.json`, etc.) containing key-value pairs for interface strings. When rendering a page, we'll load the appropriate locale's file and use a translation hook or component to output the correct text. We ensure to follow the practice that if a translation for a key is missing in a target language, it falls back gracefully to English (or another fallback) rather than breaking. Indeed, Next.js i18n can redirect to default locale if an unsupported locale is visited [23], and libraries like i18next handle key fallback to a base language if a specific translation is not present. - *Dynamic content (CMS content)*: For content managed in Payload CMS, we have a couple of strategies. One, we can have separate localized fields in each collection (e.g., for a product, fields like `title_en`, `title_pt`, etc.), or use a nested localization system. Payload CMS actually has a localization feature built-in that can manage locales for you, storing each locale's content. Alternatively, we treat each locale as a separate entry or link entries via a relation field. For simplicity, we might use Payload's localization config: it allows defining `locales: ['en', 'pt-BR', 'es', 'fr']` and it will then allow content editors to input each field in each language tab in the admin UI. Under the hood, it might store JSON or separate tables, but it abstracts that for us. The front-end then can query specifically for a locale's content via the API. For example, a blog post entry can be fetched with `locale=pt-BR` to get the Portuguese text. This approach keeps one record per content item with multiple localizations, which is convenient.

We will need to decide on fallback behavior. Likely, if a piece of content isn't translated into a non-default language yet, we'll fall back to English content rather than show nothing. Next.js's i18n routing documentation advises that if a locale isn't available it should fallback to default [24] – meaning we should ensure our content API returns either the English content or we programmatically substitute. For instance, if a blog post isn't translated to French, when building the French page we might still show the English text but perhaps denote it's untranslated. Or we might filter out content that isn't localized. This is a content strategy decision. Initially, we may not have full translations for everything, so fallback is crucial. The Next.js docs note: *"If user locale is `nl-BE` and it is not listed in your configuration, they will be redirected to `nl` if available, or to the default locale otherwise."* [23]. In our case, `pt-BR` is listed (not just `pt`), but the idea is similar: we include country-specific to handle variants (like Brazilian Portuguese vs. potentially Portuguese of Portugal differences). We'll list `pt-BR` specifically since our content will cater to Brazilian context in language.

Language Switch UI: We will provide a language switcher on the site (likely in the navigation or footer) allowing users to manually switch languages. This will use Next.js's `<Link locale>` feature or a custom mechanism to navigate to the corresponding page in another locale. Because of identical routing structure under each locale, we can relatively easily switch (for example, if on `/about` in English, switching to French would go to `/fr/about`). We'll ensure that for dynamic routes (like `/product/[slug]`), we can map slugs properly if needed – possibly slugs themselves might be translated or we maintain a mapping. We might just keep slugs in one language (e.g., English slugs) globally to avoid confusion, meaning URLs might not be localized – or use an ID-based route. This is a detail to iron out: one approach is to have language prefixes but the slug remains constant (so the URL structure is `/{locale}/product/{english-slug}`). Alternatively, create localized slugs (which is more user-friendly for each language but harder to implement). For MVP, using a single slug is fine, focusing on translating the content, not the path.

Fallback and Default: English is the default. That means if a user hits the site root or an undefined locale, they get English. Also, if any translation is missing, we use English. Our configuration will set `defaultLocale: 'en'`. Next.js's i18n ensures consistency that way – and we likely won't prefix English paths (so English pages are at `/about`, Portuguese at `/pt-BR/about`, etc.).

Internationalization of UI components: We will observe best practices such as avoiding concatenating strings that need translation, handling pluralization, gender, etc., if needed (though for our site content it may be straightforward sentences). We will also ensure that any dates or numbers displayed are formatted according to locale. For instance, using `Intl.DateTimeFormat` to show dates in a locale-specific format (e.g., day/month order, month names in the right language), and using `Intl.NumberFormat` for any numbers or currency that may appear. Since our stack is modern JS, these internationalization APIs are readily available. If needed, libraries like `react-intl` or `formatjs` could be integrated to simplify this.

Testing and QA: We will thoroughly test the site in all four languages to ensure layout still looks good (some languages like French or Portuguese can have longer words that might break layouts if not considered, or right-to-left if we had that – but we currently have only LTR languages). We also verify that navigation goes to the correct locale and that search engines can index the multilingual content properly (Next.js will output hreflang tags to help search engines know the page exists in other languages).

By implementing multilingual support from the start, Monynha.com aligns with its inclusive mission – reaching users in Brazil, Latin America, France, etc. on launch. The infrastructure overhead is manageable thanks to Next.js's built-in routing. Our site's content strategy will ensure we maintain translations. The combination of Next.js i18n routing and Payload's localization gives us a strong pipeline: content editors can input translations in one interface, and the front-end can fetch and display it seamlessly.

In summary, **internationalization in Monynha.com** is characterized by: - Four supported locales (EN default, plus PT-BR, ES, FR) and scalable to more if needed. - Locale-specific URLs with automated routing courtesy of Next.js. - Both static UI text and dynamic content localized. - Fallbacks to English content when a translation is missing, to avoid blank spots [24] . - A good user experience for switching languages, maintaining context across locales. - Proper formatting and cultural considerations in content.

This setup ensures that Monynha's message of democratizing tech can truly cross language barriers and resonate with a global audience.

2.6 Authentication Strategy (Single Sign-On via Monynha.me)

Monynha.com itself does not handle primary user authentication internally; instead, it integrates with an external authentication service located at **Monynha.me** to provide a unified Single Sign-On (SSO) experience across the Monynha ecosystem. The rationale is to centralize user identity and session management in one domain (monynha.me), so that users who log in (or sign up) there can seamlessly access all Monynha platforms (the main site, potential future apps like monynha.tech, monynha.store, etc.) without separate logins for each. This fosters convenience and consistency in security.

SSO Overview: The SSO works conceptually similar to how logging in with Google or another identity provider on various sites might work. When a user chooses to log in on monynha.com (for example, clicking a "Login" or "Dashboard" link), they will be redirected to the **monynha.me** authentication portal (likely a subdomain or dedicated site that could be running an OAuth2/OIDC server or a custom auth app). There, they will authenticate (enter credentials, or perhaps use social login if provided). Upon successful login, monynha.me will issue an authorization token or cookie and redirect the user back to monynha.com along with proof of authentication (for instance, a JWT token in a query param or a code that monynha.com can exchange for a token).

This pattern is essentially how cross-domain SSO works: *"users log in to one domain and thereby be provided automatic authentication to another domain without further interaction."* [25] . In our case, monynha.me is the central domain that issues the login session, and monynha.com trusts that domain's session. Typically, implementing this might involve **OAuth 2.0 Authorization Code Flow with PKCE** if we treat monynha.me as an OAuth server and monynha.com as a client. Alternatively, if it's simpler, it could be a cookie set on a parent domain (if monynha.com and monynha.me shared a common top-level domain and we set a wildcard cookie). However, since .com and .me are different TLDs, cookies can't be shared, so an OAuth-like redirect flow is appropriate.

Monynha.me Auth Service: We can imagine monynha.me runs an auth service (perhaps using a library like Auth0, or a self-hosted solution like Keycloak, or even Supabase Auth if configured on that domain). This service stores user accounts (emails, passwords hashed, etc.) and perhaps profile info. It might also handle password resets, multi-factor auth, etc., all outside of monynha.com's codebase. This separation means Monynha.com doesn't have to worry about securely handling passwords or the UI of login forms – it delegates to monynha.me.

Integration on Monynha.com: From the perspective of Monynha.com, the integration steps are: - Provide UI links to login (and possibly logout or profile) that direct to monynha.me. For example, clicking "Login" could navigate to `https://auth.monynha.me/login?redirect=monynha.com`. The `redirect` query tells the auth system where to send the user after logging in. - After authentication at monynha.me, the user is sent back to a predetermined callback URL on Monynha.com (e.g., `https://monynha.com/auth/callback`) along with an auth token or code. - Monynha.com then validates this response. If using OAuth/OIDC, monynha.com (Next.js backend) would use a client secret to exchange a code for an ID token + access token from monynha.me's token endpoint. The ID token (typically a JWT) would contain user identity (user id, email, roles, etc.). Monynha.com would then set a session cookie for the user based on this (or store the token in a secure httpOnly cookie). - From that point, Monynha.com knows the user is authenticated and who they are, and can personalize content or allow access to gated areas. But it doesn't manage the actual credentials – that's all at monynha.me. - If the user logs out, Monynha.com can clear its session and optionally redirect to monynha.me to fully log out (so that the SSO session at the identity provider is also terminated).

The benefit of this approach is akin to classic SSO definitions: *"users or clients log in to one domain and are automatically authenticated to another domain without further interaction"* [25] , meaning if the user is already logged in at monynha.me (say they logged in via another Monynha application earlier), coming to monynha.com could even recognize that and log them in silently by checking with monynha.me. This requires perhaps an iframe check or a prompt – or simply the user clicking login will find they don't need to re-enter credentials.

Security Considerations: We will use a secure protocol (likely OAuth2/OIDC as mentioned). The communication will all be over HTTPS to prevent eavesdropping. We will validate tokens carefully (checking signature, issuer, expiration). For domain trust, we might have monynha.com configured as a trusted OAuth client on the monynha.me auth server, with redirect URIs locked down.

Additionally, **Supabase RLS** integration: If our site uses Supabase for certain user-specific data, we can integrate the auth such that the Supabase JWT includes the same user ID. Supabase Auth could be configured to accept external JWTs (Supabase has a concept of 3rd party auth or at least one could upsert user info). Alternatively, we bypass Supabase Auth and simply use service role keys in Next to fetch data and filter by user id manually. If we want to use RLS, we might need to propagate the user's UUID to Supabase's JWT. This might be a later optimization.

User Data and Profiles: Monynha.com may have a "My Account" or similar page after login, where it shows user profile details or their content (maybe a list of their form submissions or comments, if any). That data might come from monynha.me (profile details via an API) or from Supabase if user info is stored there. We likely let monynha.me be the source of truth for user profile (like display name, avatar). If needed, monynha.me could have an API for that, or embed a widget. But to keep things straightforward, monynha.com could store minimal user data on its side once authenticated (like store user id and email in its own database for quick reference or personalization). We will plan these details in the database schema if needed (e.g., a `users` table in Supabase synced with monynha.me accounts).

Session Management: On monynha.com, once the user is logged in, we maintain a session. In Next.js (especially if using the App Router and maybe NextAuth or a custom solution), we can use cookies or JWTs. A likely approach: after the OAuth handshake, we set a cookie `session=JWT` where the JWT might be the same one from monynha.me or a new one we sign that just indicates logged-in status. The Next.js application (via middleware or API route checks) will read that to verify the user on each request (for server props or for API calls). Alternatively, use NextAuth.js library configured for custom provider (monynha.me as the provider). NextAuth could simplify some of the token storage and rotation aspects.

Monynha Domains Hub: Given Monynha.com will act as a hub linking to other Monynha domains (e.g., monynha.tech, monynha.store, etc.), SSO makes user transitions between these smooth. For instance, if monynha.store requires login to purchase something, having a monynha.me SSO means the user uses one account across all. The hub (monynha.com) could even show a unified profile status (like "Hello, [Name]" if logged in, which might cover all sub-sites). This cross-domain trust is precisely the point of SSO. It *"allows for continuous use of multiple related systems... the user identity is maintained across the systems"* [26] . Our implementation ensures that once identity is verified at monynha.me, it's recognized by monynha.com and vice versa.

Anonymous Access: It's important to note most of Monynha.com's content is public. Authentication is only needed for specific interactions (e.g., maybe commenting on blog posts in the future, or accessing a user-specific dashboard or submitting certain forms that require login, etc.). At MVP, the site might not have heavy auth-required sections. But we lay this groundwork for future expansion (like if there's a community forum, or OSS contributor login, etc.). So the site will function fully for anonymous users, with login being an optional layer for added features.

In summary, the **authentication architecture** for Monynha.com is: - **External IdP**: Monynha.me acts as the Identity Provider (IdP) handling login UI and credential storage. - **SSO Mechanism**: OAuth2/OpenID Connect used to let Monynha.com delegate auth to Monynha.me. This provides a secure token that Monynha.com trusts. - **Session on Monynha.com**: Maintained via secure cookies or token storage, representing the logged-in user (with an ID, etc.). - **Single Logout**: If needed, logging out from one could log out from all (we might implement that later, e.g., if user logs out from monynha.com, we also hit an endpoint at monynha.me to destroy that session). - **Benefit**: One account for all Monynha platforms, better user experience, centralized security updates (if we need to enforce 2FA or monitor logins, it's all in one place).

This approach follows modern best practices for web SSO, akin to how large organizations centralize login. It is also scalable – if Monynha adds more services, they all hook into Monynha.me. It keeps

Monynha.com's code focused on content and functionality rather than authentication logic, thus adhering to separation of concerns (auth as its own service). As a user story, *"Log in once, access everything"* is realized, aligning with our mission of easy and democratic access – not locking users into multiple sign-ups. And technically, it aligns with how *"SSO allows multiple independent systems to trust one login, improving user experience and security."* [27] .

---

3. Site Requirements and Structure

This section outlines the specific requirements for the Monynha.com site, including its navigational structure, dynamic content models, and underlying data schema. Essentially, it's a blueprint of what pages and features the site will have, how they interconnect, and what data entities support them. The site aims to serve as an institutional portal for Monynha Softwares – providing information about the company, its solutions and products, community initiatives (like open-source projects), career opportunities, and governance. It also functions as a hub, pointing to other Monynha domains as needed.

3.1 Primary and Secondary Routes Overview

Monynha.com will consist of several primary pages (top-level routes) and secondary pages (sub-routes or dynamic routes). Below is a list of all major planned routes, each corresponding to a section of the site. The routes are organized in a way to be intuitive for users and SEO-friendly, with clean URLs for each content section:

- **Home Page** – `/`: The main landing page. This will introduce Monynha Softwares, highlight key messages (brand narrative, mission), feature perhaps a hero section and quick links to important sections (products, solutions, latest blog posts, etc.). It's the summary and gateway to deeper pages.
- **About** – `/about`: A page describing the company's story, values, team, and possibly press mentions or history. It reinforces the brand identity and narrative (the LGBTQIA+ and class solidarity ethos, mission to democratize tech). It might be broken into subsections (mission, story, team bios, etc.) on the same page or as sub-routes if content is extensive (e.g., `/about/team` if needed).
- **Solutions** – `/solutions`: This page outlines the services or solutions Monynha offers (if Monynha Softwares provides consulting, or specific software solutions for clients). It could list domains of expertise or industry solutions. It's more of a marketing page for potential clients, focusing on what problems Monynha can solve and how.
- **Product Pages** – `/product/[slug]`: These are dynamic routes, one for each product Monynha has. For example, if Monynha Softwares has developed specific software products or platforms, each would have a detail page. The `[slug]` represents the product identifier (e.g., `/product/monynha-cloud` or `/product/learning-portal`). These pages display product-specific information: features, screenshots, documentation links, etc. The content for these comes from the Products collection in the CMS. The route is singular "product" (not plural) to indicate a single item; we might also have an index page listing all products (e.g., `/product` showing an overview list).
- **Contact** – `/contact`: A page providing ways to get in touch with Monynha – likely a contact form that visitors can fill (which might send an email or store in a database), as well as contact info like email addresses, social media, or physical office address. It should include a form with fields: name, email, message, etc., possibly integrating with our backend to handle submissions.

The contact page reinforces accessibility by providing multiple channels (maybe also links to community forums or chat if any).

- **Open Source (OSS)** – `/oss`: A page dedicated to Monynha's open-source projects and community contributions. It might list repositories or projects that Monynha Softwares has released or is heavily involved in. Each project could have a brief description and link to GitHub (or a detail page if needed). The OSS page aligns with the mission of democratizing tech – highlighting the free contributions to the community. We can dynamically fetch project info via GitHub API or store details in CMS. If each OSS project gets its own page, we could have dynamic routes like `/oss/[project]`, but likely a single page with a list is enough for MVP.
- **Blog** – `/blog`: The main blog listing page, showing recent posts or categories. This page will fetch a list of blog posts from the CMS and display them in reverse chronological order (maybe paginated or infinite scroll if many). It should have excerpts and maybe tags or categories (if we implement them). The blog serves to share insights, updates, tutorials aligned with Monynha's values.
- **Blog Post** – `/blog/[slug]`: Dynamic route for individual blog articles. Each blog post page will show the full content of the article (rich text, images, etc.), the author name, publish date, and possibly related posts. The `[slug]` corresponds to the post's slug. These pages are generated from the BlogPosts collection in the CMS. We plan to generate them as static pages (with ISR) for performance, updating when new posts are added.
- **Careers** – `/careers`: A page listing job openings at Monynha Softwares. If the company is hiring, this will be crucial. It can list positions by department, with short blurbs and link to detailed job descriptions. If many jobs, we could have dynamic pages for each job (e.g., `/careers/[job-id]`), but initially might just list them with expand/collapse or a modal for details. The page also talks about company culture, benefits, etc. Careers page content might be managed via CMS (job posts could be a collection).
- **Governance** – `/governance`: This page outlines how Monynha Softwares is structured or governed. Given the political slant (class solidarity, etc.), Monynha might be a cooperative or have transparent governance practices. This page would detail that: e.g., "Monynha Softwares is an employee-owned cooperative" or info about its bylaws, code of conduct, community council, etc. It may have sub-sections like leadership, advisory board, or open governance policy documents. If needed, additional routes could be, for example, `/governance/reports` if annual reports or transparency reports are shared. But for now, a single page suffices.
- **Hub Links** – `/hub` (or possibly just the presence of a dropdown in nav): While not necessarily a page, the requirement is to have a hub interface linking to other Monynha domains. This might be implemented as a mega-menu or a top-bar element, not a separate page. But if we did, a `/hub` page could simply be a portal listing and describing each domain (monynha.me, monynha.tech, monynha.store, monynha.online, monynha.pt, etc.) with links. However, a more elegant solution is to incorporate this in the navigation UI (see 3.3).
- **Utility Pages**: Privacy policy (`/privacy`), Terms of Service (`/terms`), etc. These are standard pages needed for compliance. They might be simple static pages (possibly managed in CMS or just markdown in code). Not explicitly listed by user, but we should include them for completeness.
- **404 Page** – `/404`: A custom 404 page for not-found routes, in multiple languages, guiding users back to a valid section.

The above covers the primary structure. We see a combination of static routes (`/about`, `/contact`, etc.), dynamic routes under specific segments (`/product/[slug]`, `/blog/[slug]`), and possibly some groupings.

Navigation: The primary navigation menu will likely feature: Home, Solutions, Products, OSS, Blog, Careers, Contact (and maybe split About or Governance to a secondary menu or footer). We have to balance what goes in top nav vs footer. Possibly: - Top Nav: Solutions, Products, Open Source, Blog, Careers, About, Contact (though that's many – maybe some as dropdown). - Footer: could reiterate those plus Governance, Privacy, etc.

We will ensure routing consistency and that each has correct locale versions.

Each route will be implemented as a Next.js page (or dynamic page). For dynamic sections like product and blog, we use Next's file-system routing with `[slug]` file. The slugs will be fetched from CMS for static generation.

Cross-links: Relevant cross-linking will be set up. E.g., the home page highlights might link to a particular product or blog post. The Solutions page might link to relevant product pages. Blog posts might link to OSS projects or external resources. This internal linking is good for UX and SEO.

3.2 Dynamic Content Models (Products, Blog Posts, Translations)

To support the site routes, we need structured content models that correspond to the dynamic entities on the site. These content models will be managed in the CMS (Payload) and/or stored in the database (Supabase). Let's enumerate and describe each major model:

- **Product** model: Represents an individual software product or offering by Monynha. Each Product has fields such as:
- `title` (e.g., product name)
- `slug` (unique identifier for URL, e.g., "learning-portal")
- `description` (rich text or markdown providing an overview)
- `features` (maybe a list of key features or benefits)
- `image` or `logo` (for display on the product page or listing)
- `links` (like a link to documentation or a demo)
- Possibly `category` or `type` if multiple product lines.
- If multilingual, fields either repeated per locale or stored via localization.

This model populates the **Products** collection in Payload. It feeds both the product listing (if we have one) and the product detail pages (`/product/[slug]`). We likely want a way to order products or mark some as featured (could have a numeric sort order or a boolean "featured").

- **Blog Post** model: Represents a blog article.
- `title`
- `slug`
- `author` (could be a reference to an author profile or just a text name; initially maybe text or selection of preset authors)
- `date` (publish date)
- `content` (rich text or markdown for the full blog content)
- `excerpt` (short snippet for listing, or we can derive from content)
- `coverImage` (optional, an image for the top of post or thumbnail for listing)
- `tags` or `categories` (to group posts; optional for MVP)
- Possibly `language` if we decide to have separate posts per language vs one post with many translations. If using Payload localization, one post entry will have multiple locales inside; otherwise, we could have separate collections per locale or a field for language. Using localization is simpler.

Blog Post entries fill the **Blog** (or Posts) collection. They show up on `/blog` listing and on individual pages. We should also consider pagination if there are many posts – e.g., maybe 10 per page.

- **Translation Strings** model: This is an optional model to manage site-wide UI text in CMS. For instance, if non-developers need to tweak some wording that is not in content but in interface (like the tagline on home page, or menu labels), we can create a Translations collection or a Global. For instance:
- Key (identifier, e.g., "home.tagline")
- Text_en, Text_pt, Text_es, Text_fr (or a localized field if payload supports a locales array here).

Alternatively, we might manage these via JSON files in code as typical, but if we want absolutely everything editable via CMS, this is possible. It could be overkill for MVP to have CMS manage interface translations; we may rely on developers for that initially and use code files, and only manage large content via CMS. But if content editors are non-technical and likely to want to edit any text, a translation model might be handy.

If implemented, the site on load could fetch all translation entries via an API and use them, or more simply, we generate JSON from the CMS and commit it (some automation) – this might be too complex now. I lean that we handle UI strings in code and content in CMS.

- **Careers/Job model**: If we have job postings:
- `title` (e.g., "Senior Developer")
- `location` (or remote, etc.)
- `description` (full job description, maybe as rich text)
- `department` or `team`
- `applyLink` (if using an external application form or email).

These would be in a Jobs collection and shown on Careers page. If integrated with something like Greenhouse or Lever in future, we might skip a custom model. But for now, a simple model works.

- **Open Source Project model**: For listing OSS projects:
- `name`
- `description`
- `repoUrl`
- `logo` or `icon`
- Possibly a `status` (active, archived).
- We could fetch directly from GitHub org via API instead of CMS, but CMS allows adding commentary and selection of which projects to feature.

We can manage a list in CMS to avoid relying on external API on each page load.

- **Page content models**: For About, Governance, etc. We can either:
- Use *globals* in Payload for each such page (like a global named "AboutPage" with fields for each section on the about page, such as banner text, history text, team list etc.).
- Or treat them as a collection (like a collection "Pages" with a field "pageName" and content fields). Since these are one-off, globals are a good fit (Payload supports a few global content singletons).

For example, a **Global** called "AboutPageContent" might have: - `heading` - `body` (rich text) - maybe `teamMembers` (a repeatable group of name, bio, photo).

Similarly, a "HomePageContent" global could manage things like the hero tagline, feature highlights (with references to product entries to pull their info dynamically).

This approach means an editor can change home page text via CMS easily.

Relationships: - Products might relate to Solutions if applicable (e.g., product X is under solution Y category). - Blog posts could have an `author` reference if we have an Author collection (or we keep author as simple text field and not complicate). - If multi-locale, and if not using built-in localization, we might have relationships linking translations. But using built-in is easier.

Content Fallback: - If a blog post is not translated to a certain language, do we hide it or show English? Perhaps show English with a notice. We can use the locale fallback mechanism when querying (like if using payload locales, it might allow fallback to default locale content if none in requested).

Content Delivery: - Next.js will fetch these via Payload's REST/GraphQL. For static pages (like about), we may fetch at build time (since these change rarely). - For dynamic, we use getStaticPaths/getStaticProps for products and blog posts to pre-generate. - Supabase could also be directly queried for some things if needed (though we likely stick to Payload API for content; Supabase direct might be used for user-related data or form submissions).

Form Submissions: - Not exactly a content model but relevant: The Contact form likely should have a mechanism to capture submissions. We can: - Use a table in Supabase called `ContactMessages` with fields name, email, message, timestamp. This table can have RLS such that only service role can insert (or open to insert with anon if we trust front-end? Better via serverless function). - Or have Payload handle it via its built-in forms (Payload can have a collection for messages and allow creating entries without auth – but that needs careful open access).

Possibly simpler: create a Next.js API route for contact that inserts into Supabase (using service key). This way we avoid exposing any direct insert rights to client.

- Newsletter sign-up (if any) could be similar, a table of emails or an integrated service like Mailchimp.

Meta and SEO: - Each model should include fields for SEO meta tags (like meta title, meta description). We can derive them (e.g., use title and an excerpt by default), but giving control can be nice. Payload could include a reusable component for SEO in collections (some CMS do this). - Social sharing image (could default to some template or specific field per page).

Summarily, the dynamic content models align with the site map: - **Products**: dynamic pages under /product - **BlogPosts**: dynamic pages under /blog - **OpenSourceProjects**: listed on /oss - **Jobs**: listed on /careers - **PageGlobals**: content for /about, /governance, possibly /home. - **Translations** (if used) or just using built-in localization for all above.

We'll ensure the models are flexible for future expansion (for example, adding a new language or a new content section won't break things).

3.3 Navigation and Hub Interface (Mega-Menu & Domain Links)

Navigation is critical for usability. Monynha.com will feature a well-structured navigation system that helps users find content easily and also serves as a "hub" interface linking to Monynha's other domain sites. Two main aspects are considered: the primary navigation menu (possibly with dropdowns/mega-menu) for internal pages, and a set of hub links or a menu to access external Monynha domains.

Primary Navigation Menu: This is typically a top bar menu present on all pages (desktop view likely as a horizontal menu, mobile view likely a hamburger menu). The primary menu items will correspond to the main sections: - Home (could be the logo linking to home, rather than an explicit "Home" label). - Solutions - Products (if multiple, might need a dropdown if there are many product pages, or link to a landing listing). - Open Source (OSS) - Blog - Careers - About (which could include Governance as a sub-item, or Governance could be separate) - Contact

That's a lot of top-level items; to avoid crowding, we might group some: For example, an "About" dropdown that includes "About Us", "Governance", possibly "Careers" (though careers often stands alone). Or a "Community" dropdown that includes Open Source and maybe Blog. We have options: - **Mega-menu / Dropdown**: We could implement a mega-menu that opens an expanded panel when hovering "About" or "Company", showing columns for About, Governance, Team, Careers, etc., and another for "Resources" (Blog, Open Source). Alternatively, a simpler approach: - Top-level "Company" menu with sub-links: About, Careers, Governance, Contact. - Top-level "Resources" with sub-links: Blog, Open Source. - Top-level "Products & Solutions": if they want to combine, though “Products” might remain top-level and lead to an overview if multiple products.

Given the content: - It might be better to have **About** (with sub: Governance maybe), - **Products** (listing products or first product), - **Solutions** (if multiple offerings), - **Open Source**, **Blog**, **Careers**, **Contact** as standalone.

We will decide based on design preference and number of items. A mega-menu is often useful if many sub-links are needed. For example, if Monynha Softwares has multiple product categories, the Products menu could be a mega-menu listing each product with a description. If Solutions have multiple, similarly.

Hub Interface (Monynha domains): The user has multiple Monynha sites (monynha.me, monynha.tech, monynha.store, monynha.online, monynha.pt, etc.). Monynha.com should clearly direct users to those sister sites as needed. This could be done in a few ways: - A dedicated section or page listing them (like a “Monynha Network” page). - More elegantly, a special menu or dropdown accessible from every page, something like a grid icon or a label "Monynha Network" that opens a menu of all domains.

A common pattern for organizations with multiple domains is to have a header bar (maybe above main nav) listing the sibling sites, or a single icon (like nine-dot grid indicating a product family) that when clicked shows other sites. For example, Google has the app grid menu; some companies put “Other sites” in header.

We might implement a **“mega-menu” style hub**: For instance, on the far right of the nav, an element (maybe the Monynha heart logo or something) that on click/tap reveals a panel with all Monynha domains, each with an icon and name: - Monynha.me – “Account & Authentication” (or tagline) - Monynha.tech – possibly a tech blog or documentation portal? Not sure, but presumably a site (maybe for a developer portal or technical content). - Monynha.store – an online store (maybe merchandise or selling products). - Monynha.online – not sure of its purpose, but likely something community or events. - Monynha.pt – likely a Portuguese language site (though we have multi-language on .com, .pt might be for a specific region or project). - Monynha.website (saw in the screenshot) might be in portfolio.

Since we have them, listing them clearly is good. Each link should open in a new tab (since different domain) or same depending on integration (monynha.me might be same session usage). Perhaps monynha.me would not be directly linked except as “Login”, since it’s an auth domain not for general browsing.

We’ll need descriptive labels so users know what each is: For example, a hub dropdown might say: - **Monynha.me** – User Accounts (login & profile) - **Monynha.tech** – Developer Portal (if that’s what it is, maybe documentation or blog) - **Monynha.store** – Store (for products or merchandise) - **Monynha.online** – Online Services (just guessing, could be like a SaaS offering?) - **Monynha.website** – Possibly something else (maybe a test or older site, not sure if needed to expose). - **Monynha.pt** – Portuguese Site (though if we fully localize .com, maybe .pt is not used? Perhaps monynha.pt is meant to be a localized marketing or legal entity site).

If uncertain, at least list what we know: The screenshot [5] suggests these domains exist likely to cover different aspects. If unclear, maybe just list them by name for now.

Technical Implementation: The navigation will be built using our UI stack (likely a `<nav>` with Tailwind styling, and the dropdown triggered via headless UI or Radix components for accessibility). Shadcn/UI includes some Menu or NavigationMenu component which can implement a mega-menu style. Actually, Radix has a `NavigationMenu` component that can do multi-column dropdowns – Shadcn’s library likely has an example for a complex menu.

We'll ensure keyboard accessibility: users can tab through nav links and open submenus with Enter/ Space, and arrow through items, etc. This likely comes out of the box if using Radix NavigationMenu which is WCAG compliant.

Responsive behavior: On mobile, instead of hover dropdown, we might implement a drawer or accordion. Clicking a top-level would expand to show sub-links. Or just list top-level in a burger menu and some are clickable to next page. Possibly a separate “Sites” section for hub links.

Sticky header: Likely the header stays top when scrolling (maybe shrinks), for ease of navigation.

Footer: The footer will contain secondary navigation and important links: - Repeat critical links like About, Careers, Contact, maybe Privacy/Terms, maybe smaller links to external stuff (like social media icons). - Possibly also link the other Monynha domains in footer as well (like "Other Monynha Sites: [list]" for redundancy). - A language switcher might be in footer too (or in header via an icon, but often a small language dropdown is in footer to not clutter header).

Since we have a dedicated mechanism in header for languages (maybe a globe icon if not spelled out), either is fine.

In sum, the navigation will allow users to quickly jump to any section of monynha.com or out to any related Monynha domain. The **mega-menu/hub design** will make the site feel like part of a larger suite but still keep monynha.com as the center of gravity for institutional info.

3.4 Database Schema and Security (Tables and RLS Policies)

The underlying database (PostgreSQL via Supabase) stores the content and data for the site. Here we outline the main tables in the schema, their fields, and importantly, the Row Level Security (RLS) policies applied to protect data. This ensures that the right users (or roles) can access the appropriate data and that potentially sensitive data (like user info or form submissions) isn't exposed to unauthorized parties.

Schema and Tables: We will use the default `public` schema for simplicity (Supabase by default uses `public` for user tables, and has RLS toggled per table). Key tables include:

- **products**: Stores product information.
- Columns: `id` (UUID or bigserial primary key), `slug` (text, unique), `title` (text), `description` (text or JSON for rich content), `features` (maybe JSON array of text), `image_url` (text or maybe use storage and just store path), `created_at`, `updated_at`.
- If using Payload, note: Payload will create its own tables (e.g., if collection named "products", likely a `collections_products` table or similar). But since we can orchestrate, let's conceptually define them. With Payload, the schema might be slightly different, but we can align it.
- Indexes: unique index on slug.
- RLS: For public content like products, we likely allow read access to anon (no login needed) since this is public info. So an RLS policy can allow `select` for role `anon` on products. Alternatively, we might not even enable RLS on products if we want them world-readable. But Supabase best practice says enable RLS on all tables in exposed schema [14], then add a policy for anon read. For write, no anonymous writes (edit only via admin service or via Payload server which likely uses service role).

- Policy example: `policy "Public can read products" on products for select using ( true ) to public, anon;` (if making accessible to both anon and authenticated roles) [28] [29].

- **posts** (blog posts):

- Columns: `id`, `slug`, `title`, `content` (maybe long text or JSON), `excerpt`, `author_id` (if linking to an authors table), `publish_date`, `cover_image` etc., plus timestamps.
- RLS: blog posts are also public. So similar to products, allow read for anon. Write only via admin.

- Possibly an authors table: `authors` (id, name, etc.) and posts.author_id references it. If so, authors might be public as well (just names).

- **contact_messages**:

- Columns: `id`, `name`, `email`, `message`, `created_at`, maybe `ip` or something if we store.
- This is sensitive user-submitted data (though not extremely sensitive, but it's private communication). We definitely do not want anon or any front-end to list these. Only admins should read them.
- RLS: We enable RLS and do NOT allow `select` for anon or authenticated by default. Instead, perhaps only a service role can read them (the backend with service key) or we integrate with Payload to view them in admin.
- For insertion, we want the site (even if not logged in) to be able to insert new messages. With RLS enabled, we need a policy that allows `insert` for anon (so that the public site can insert messages without auth). That needs caution: we should ensure it only allows certain fields and perhaps check that, but usually you can allow insert of those fields straightforwardly. Maybe use Supabase's function to strictly allow it.

- Example policy: `create policy "Allow public to submit contact" on contact_messages for insert with check ( true ) to anon, authenticated;`. This would let anyone insert.
- We also might have `authenticated` role synonyms. In Supabase, `anon` and `authenticated` are roles used by clients by default [28].

- Another angle is to not open insert and instead route contact form through a Next API which uses Supabase service role to insert (bypassing RLS). This is more secure (no direct exposure). We might do that to avoid having to open direct DB insertion to the world, which could be spam vector (though we can still get spam via API route too, but at least we can add captcha or rate limit in API).

- **users**:

- If we create a users table to mirror monynha.me accounts (for any site-specific data or just to have references in Supabase), it might include `id`, `auth_id` (id from monynha.me), `email`, `name`, etc. This might be populated when a user logs in the first time via an upsert.
- RLS: If present, each user should only select/update their own row. We can use the Supabase `auth.uid()` function inside RLS policies if we integrate Supabase Auth. But since we use external auth, we might not use Supabase's `authenticated` JWT except if we pass a custom JWT. If we skip the Supabase auth, we might not even use this table now. Perhaps not needed unless storing user-specific data like favorites, etc. At MVP, not necessary.

- If needed: policy such as `user_id = auth.uid()` to allow users to see their own data [30] [31].

- **jobs** (if careers):

- Columns: `id`, `title`, `description`, `location`, etc., `open` (bool if still accepting).

- RLS: public read, admin write. So similar to products, allow select for anon, restrict insert/update.

- **open_source_projects**:

- Columns: `id`, `name`, `description`, `repo_url`, etc.

- RLS: public read.

- **translations** (if used):

- If we had a translations table (like key, locale, text), reading it could be public (since it's just UI strings). But we might not implement this as said.

- **Payload CMS**:

- Note: If using Payload, it will create its own tables. Possibly naming like `payload_Products` etc., depending on config. If we go headless only via Payload, we may not interact with these tables directly, only via Payload API. Payload itself does auth for its admin, but if its data is accessible via its API publicly, we must ensure to configure that properly (like it might not expose everything unless allowed).
- If using Payload with Supabase, Supabase RLS still applies to those tables if accessed outside Payload. But Payload server likely connects with full DB rights (we would likely use the service

role for it). So in effect, payload bypasses RLS by connecting as a privileged user. Therefore, to allow direct Supabase client access to say blog posts, we might either:
    - Query through Payload (which would use payload's own access control).
    - Or allow direct selects (which requires RLS policies for anon).

Because we plan to allow the Next front-end to fetch content (like blog posts) via Supabase or Payload: - Via Payload REST: Payload can have its own access control (like you can configure collections to be publicly readable). We likely will configure Payload to allow public read for collections like blog, products, etc., through its REST API without auth token. Or provide a read-only API key to Next. - If we bypass Payload and use Supabase directly for content reading (less likely if we leverage Payload's features, but maybe for simple lists we could use Supabase RPC or views), we need RLS for anon as described.

Row Level Security (RLS): Supabase RLS ensures that even if someone got hold of our anon service, they can only do what we explicitly allow. It's a second layer beyond our app logic. As Supabase docs say, RLS is *"incredibly powerful and flexible, allowing you to write complex SQL rules … providing defense in depth"* [32] . We will indeed enable RLS on all tables and then carefully craft policies: - **Public content tables** (products, posts, etc.): Policy to allow `SELECT` for role `anon` (and `authenticated`). No insert/update for anon. Possibly allow insert/update for `authenticated` if we had some user-submitted content in those tables, but we don't; content edits will be through the admin backend (Payload or our internal usage) which can use the service role (bypassing RLS entirely using our server key). - **Protected tables** (contact_messages, any user-specific data): - `SELECT`: no anon, no authenticated by default. Only allow if `auth.role() = 'service_role'` or a specific check for staff user if we implement such roles. But in Supabase, when using the client libraries with an API key, we either use anon or service key. We likely do not expose service key to client; that's only for our server. For admin UI, since we have Payload, viewing messages can be via a Payload collection (so contact_messages could be a Payload collection with its own admin UI). - `INSERT`: allow anon as discussed to let site submit contact. - Option: We might create a PG function to handle insert (for data validation or spam check) and allow calling that function by anon instead of direct insert. But that's complexity likely not needed initially.

- **Example Policy**:
- For `products`: `ALLOW SELECT on products TO anon USING (true)` effectively (meaning any row can be seen by anon) [29] . No need for a USING condition since all are viewable.
- For `posts`: similar allow select.
- For `contact_messages`:
    - Insert policy: `USING (true)` for insert (or with check true) to allow any insert.
    - No select policy (so default deny).
    - Alternatively, if we wanted logged-in users to see their own messages, we'd have a user_id field and policy like `user_id = auth.uid()`. But since we don't have user login for contact, and these are inquiries to company, only internal should see them.
- For `users` (if any): `ALLOW SELECT TO authenticated USING (id = auth.uid())` to let users see their own data [30] . `ALLOW UPDATE` similarly for own.

We also set up **Role-based Access**: - Supabase has by default two roles for API: `anon` (not logged in) and `authenticated` (when a Supabase JWT is sent after login). If we not using Supabase Auth, all our client calls would be anon (unless we manually create a JWT for monynha.me user and somehow tell Supabase to treat it). Possibly not doing that now. - The Payload server will connect with the database using the environment connection string (likely as the `postgres` user or a service user). That connection is outside of RLS (since RLS only applies to connections that have it enabled; typically RLS policies apply to the roles used by the API with row security on). We might have to explicitly enable row security on each table, which is done by `ALTER TABLE ... ENABLE ROW LEVEL SECURITY;` [33]. If we create tables via Supabase GUI or Payload migrations, we need to ensure RLS toggled on. In Supabase web UI, by default a new table created from their interface has RLS off until toggled. But the Supabase docs state if table created via their Dashboard, RLS is enabled by default [34]. We will verify and enable where needed.

Relationships & Constraints: - Foreign key constraints for relations (e.g., post.author_id -> authors.id). - Unique constraints (slug unique). - Not null constraints where appropriate (title not null, etc.). - Indexes on important query fields (slug is unique index (which also serves as index), maybe index on publish_date for sorting, etc.).

Migrations: - If using Payload's migration system, some of this is handled by Payload's schema definitions. Or we may manually manage the DB via Supabase migration tools (they have some SQL dump or their internal migration). - We should treat schema as code: probably storing create table statements in a SQL or using Payload's JSON schema. Supabase's CLI could be used to manage migrations or the web studio for initial dev.

Defense in Depth: - Even if our API endpoints accidentally exposed a wrong query, RLS would prevent private data from leaking [35]. For example, if someone tries to query contact_messages via Supabase client as anon, RLS will block it (since no select policy). - If a malicious actor tries to alter a product via client, without an update policy, it fails. - We only give the minimal rights needed to anon and (if used) to authenticated.

Service Role Usage: - The service role (supabase's secret) bypasses RLS by default (as it has `role = postgres` or an admin role). We will use it on our server-side operations like builds or admin tasks if needed. We must keep that secret safe (only in backend environment, not in client). - E.g., if we create a Next API route to list contact messages (for an admin dashboard), we'd use service key and thus not worry about RLS. But more likely, our admin view is through Payload's admin panel reading from same DB (with the DB user likely being a privileged one).

Supabase vs Payload responsibilities: - If using Payload for all content, one could argue we might not need to manually set RLS for content if we rely solely on Payload's API for it. But if we foresee using Supabase direct queries (maybe for performance or convenience), it's good to have RLS. - Also, Supabase's REST API (which auto generates endpoints for tables) could be another route to data if we enabled it. We likely won't actively use those given we have Payload's nicer structure. But RLS covers those too.

Example snippet: Say we want to illustrate a typical RLS policy, we could cite an example: Supabase docs provide one: *"create policy "Individuals can view their own todos." on todos for select using ((auth.uid()) = user_id);"* [36]. In our case, an example: "Public profiles are viewable by everyone" – if we had profiles table:

~~~
create policy "Public profiles viewable by everyone"
on profiles for select
to authenticated, anon
using ( true );
``` [37]

We will implement similar policies as needed for each table.
~~~

In conclusion, the database layer is carefully structured to match our content models and is fortified with RLS to enforce security at the lowest level. This prevents unintended data leaks and ensures that our **API-first** approach remains secure, since *“each policy is attached to a table and executed every time… adding a WHERE clause to every query”* [38] [39] . The combination of a thoughtful schema and robust RLS allows Monynha.com to be both flexible in serving content and strict in protecting data, aligning with our commitment to both openness (for public content) and privacy (for user communications and accounts).

---

4. User Experience (UX) and User Interface (UI) Identity

Monynha.com’s UX/UI is designed to be engaging, inclusive, and delightful, reflecting the brand’s personality and values. The visual identity and interactive behavior of the site aim to set Monynha apart as a vibrant, user-friendly platform. This chapter details the key aspects of UX and UI: from the micro-interactions that add playfulness, to the accessibility practices ensuring everyone can use the site, to the theming and responsiveness that adapt the design to various contexts.

4.1 Interaction Patterns and Playful Micro-Interactions

Interaction patterns on Monynha.com are crafted to guide and reward the user as they navigate the site. Rather than static pages, the site feels lively and responsive to user actions. We incorporate **micro-interactions** – small, subtle animations or responses to user input – throughout the UI to create a sense of delight and to provide intuitive feedback. As one UX expert put it, *“Micro-interactions are small but crucial elements that enhance user experience. They offer intuitive cues and turn routine tasks into enjoyable moments.”* [40] .

Examples of micro-interactions on our site:
- **Hover effects on buttons and links**: When a user hovers over a button or a navigation link, it could gently change color, elevate (using a shadow), or display a slight scaling animation. This immediate visual feedback confirms that the element is interactive and responds to the user's presence. A button might slightly grow and change to a brighter shade to signal readiness to be clicked. These are quick (perhaps 150-200ms transitions) and use easing (like ease-out) to feel smooth.
- **Animated icons**: Icons could have playful states – for instance, a menu icon might morph into a close icon with a smooth transformation when toggled, or an icon next to "Contact us" might do a little shake or bounce to draw attention after a few seconds on the page.
- **Loading spinners or skeletons**: When content is loading (for example, after clicking to another page or submitting a form), instead of a plain wait, we provide either a spinner with brand colors or a skeleton UI. We can use a fun custom loader – maybe the Monynha logo morphing or a simple progress bar that slides across the top of the page. These not only inform

the user that something is happening but can also incorporate the brand personality (playful, friendly).
- **Form field interactions**: Form inputs will have clear focus states (glow or underline on focus) and when a user successfully submits, the form might show a checkmark icon that animates in as confirmation. If there’s validation error, perhaps the field gently shakes or an error message fades in – these micro-interactions draw the eye to the issue without solely relying on static text or color.
- **Interactive feedback icons**: For example, on the blog post page, a “like” or “share” icon might animate (a heart might burst into a little pop of confetti when clicked to like a post, as a fun touch, if such a feature exists).

These micro-interactions make the site feel **“intuitive and human-centered”**, turning *“ordinary tasks into moments of delight”* [41] [42] . Importantly, we apply them with restraint and purpose: they should always either provide feedback or enhance enjoyment, and not distract or annoy. For instance, a very subtle parallax effect on the header image as you move the mouse can add depth, but it shouldn’t be so large as to be disorienting.

Playful Animation Styles: The brand identity encourages some playfulness – perhaps using slightly bouncy easing for certain elements to avoid a sterile feel. We might use CSS animations or small JavaScript-powered animations (maybe leveraging the Web Animations API or a library like Framer Motion for React to orchestrate more complex sequences). For example:
- The hero text on the homepage could animate in (each word fading and sliding up slightly in sequence) when page loads, giving a spirited introduction.
- SVG illustrations might have subtle looping animations – e.g., an SVG of a person using a computer might have a blinking cursor or a moving progress bar on the screen.
- When navigating between pages, instead of an abrupt cut, we can use Next.js transition hooks or simply CSS to fade out the old content and fade in the new, smoothing the experience.

Guiding the User: Micro-interactions also serve functional roles like guiding focus. E.g., if a user adds a comment (if that feature exists) successfully, a small notification could slide in saying “Thank you!” and then slide out. Or if a section comes into view on scroll, we can animate its appearance (like a slight fade-in and upward motion) to catch attention – but doing so in a way that’s subtle and not heavy on performance.

We are also mindful of not overwhelming users. The design principle is *“less is more”* for micro-interactions [43] . We choose a handful of signature interactions that users might even come to recognize as Monynha’s style (for example, maybe using a particular springy animation curve on key elements consistently). Overusing animations can be counterproductive, so we maintain consistency and only animate where it adds value (feedback, focus, fun).

To give users a sense of cohesion, animations follow a theme:
- Duration: short interactions ~200ms, larger transitions ~300-500ms.

```
- Easing: use easing that feels friendly (perhaps `cubic-bezier(0.4, 0, 0.2,
1)` which is standard ease in-out for material design).
- Trigger: primarily on user actions (hover, click, scroll into view) so they
are prompted, not random.

All these small touches contribute to making the site feel **interactive and
engaging**. As research shows, micro-interactions *“guide users with
intuitive cues… and make digital platforms more engaging and intuitive.”* [42]
[44] . And at an emotional level, they *“infuse emotional value to create a
connection between the user and the product”* [45] – which is exactly what we
want for Monynha, to feel personable.

### 4.2 Visual Feedback, Scroll Effects, and Transitions

To maintain a dynamic and polished user experience, Monynha.com employs
various **visual feedback mechanisms** and **scroll-based effects**. These
not only enhance aesthetics but also improve usability by giving users clear
signals as they interact or move through content.

**Visual Feedback for Actions**:
- We ensure that every interactive element (links, buttons, toggles) provides
immediate visual feedback on interaction. For example, a button press will
have an active state (perhaps a slight depression effect or color darkening)
so the user knows the click has been registered. This adheres to the
principle that *users should never be left wondering if their action was
registered* [46] .
- Form input validation: If a user enters invalid data and blurs the field,
we show a red outline and maybe an icon or message. If valid, maybe a subtle
green check icon appears. Real-time feedback while typing (like an email
field showing a red X until a properly formatted email is entered, then
turning to a green check) can gently guide the user.
- Menu interactions: a dropdown menu item on hover might highlight with a
background fill, and a sub-menu arrow might rotate to indicate expansion. If
using Radix UI, these patterns are built in accessibly.

**Scroll Effects**:
- **Sticky headers**: As the user scrolls down long pages (like a blog post
or documentation), the header might condense and stick to top (if we
implement that), ensuring navigation is always handy. The transition might
involve shrinking the logo or changing background to a solid color for
readability.
- **Parallax sections**: Certain hero sections or images might use a slight
parallax effect – where the background image moves slower than foreground
text as you scroll, creating a depth illusion. We’ll keep it subtle to not
distract or affect reading.
- **Reveal on scroll**: Content elements can animate into view as the user
scrolls down. For instance, each feature item in a list could fade-up when it
enters the viewport (controlled via intersection observer or CSS animations).
This technique, if timed well, can maintain user interest as they scroll,
each new piece appearing *like a guided tour*.
- **Back-to-top button**: After scrolling a significant amount, a back-to-top
```

```
arrow may appear at bottom-right. We can make this fade in with a slight
slide when triggered, and hide when near top. Clicking it scrolls smoothly to
top.

**Page Transitions**:
- Between pages (especially if Next.js does client-side transitions for same
locale navigation), we can use a loading bar (like an ultra-thin progress bar
at top, ala YouTube or GitHub) that quickly slides while new content loads.
Next.js can support route change events to trigger such a bar. This provides
continuity and shows progress during any fetch delay.
- Alternatively or in addition, fading content: The old page content can fade
out and new fades in, or slide. However, given our site uses mostly static
generation and instant loads, heavy transitions might not be necessary. But
if we have areas that fetch client-side data (like filtering on a page
without full reload), we incorporate transitions there too.
- If we have a multi-step form or an interactive wizard somewhere,
transitions between steps should be smooth (like sliding to the next step, or
flipping a card).

**Loaders**:
- We touched on spinners for content loads. We prefer not to leave any action
with no feedback beyond 0.1-0.2s. For small network actions (like sending a
form), we can disable the submit button and show a spinner in it or next to
it so user knows it's processing.
- If an entire page content is loading (client side route), we show either a
full-screen spinner or skeleton. Skeletons (gray blocks representing text
lines and image boxes) are great if the structure is known. For example, when
navigating to a blog post page, before content arrives we could show a
skeleton title bar and paragraph lines. It sets user expectation of content
shape and is more visually pleasing than a blank screen.

**Responsiveness**:
- Our transitions and effects will be responsive too – on mobile, heavy
parallax might be turned off or simplified (to save performance and because
small screen might not handle it well). Hover effects are replaced or
supplemented by touch feedback (like a button on touch could ripple or
highlight).
- Animation durations might be slightly adjusted for mobile if needed
(sometimes shorter if mobile hardware is slower, to not feel laggy).

**Consistent Design Language**:
- The animations and feedback share a consistent style (e.g., using the same
easing curves and similar durations across the site). This consistency means
users subconsciously learn the interface's responses. If one button
highlights on hover in a certain way, all should, so users feel in control –
nothing is unpredictable.
- This consistent feedback fosters trust: interactions have *“predictable and
uniform”* results, akin to how DRY was about consistency in code, we maintain
consistency in UX behavior [47] .

**Delight without frustration**:
```

```
- We ensure that none of the effects hinder usability: for example, content
that slides in should do so quickly and not push content in jarring ways.
- We also provide ways to disable or reduce animations for those who prefer
(respecting `prefers-reduced-motion` CSS media query to reduce motion for
users who set that in their OS, which is an accessibility consideration). So
if someone has reduced motion preference, we will turn off parallax and large
animations, using simple fade or immediate appearance, to avoid dizziness or
discomfort.

By implementing these feedback and transition patterns, Monynha.com will feel
modern and polished. Users will experience a smooth journey where actions
have immediate, understandable reactions and moving through content is
intuitive. As one design article noted, such polished transitions *“make
interactions feel more responsive and interactive”*, improving perceived
performance and satisfaction [48] . We want users to sense the quality and care
in the design – which in turn reflects Monynha’s attention to user needs and
joyful experience.

### 4.3 Accessibility, Theming, and Responsive Design (WCAG 2.1 AA, Dark
Mode)

Accessibility is a first-class concern for Monynha.com. We strive to meet or
exceed **WCAG 2.1 AA** guidelines so that our site can be used by people of
all abilities. In tandem, we implement a **dark mode** for comfort in low-
light environments and ensure the design is fully **responsive** to various
screen sizes and devices. Additionally, our design system employs tokens and
careful color choices to accommodate these theming needs.

**Accessibility (WCAG 2.1 AA compliance)**:
We adhere to the four main POUR principles – our content will be
*Perceivable, Operable, Understandable, and Robust* [49] . Key practices
include:
- **Semantic HTML**: Using proper elements for content (headings `<h1>-<h6>`
in hierarchical order, lists `<ul>` for menus or grouped links, `<nav>`
landmarks for navigation, `<main>` for main content, `<form>` with labels for
inputs, etc.). This ensures screen readers can navigate content meaningfully.
We avoid divs or spans where a native element would convey semantics.
- **Keyboard Navigation**: The site will be fully navigable via keyboard
alone (for those who cannot use a mouse). This means:
  - All interactive elements (links, buttons, form fields) are reachable via
Tab order in a logical sequence. We manage focus order if needed when modals
or menus open (focus trap inside, return focus when closed).
  - Visible focus indicators: we do not remove outlines without replacement.
In fact, we style focus rings to be highly visible (maybe a 2px solid outline
in a color that meets contrast over the background). Example: a dark button
might get a bright outline when focused so it's obvious.
  - Skip Navigation link: a "Skip to content" link as the first focusable
item (possibly visually hidden until focused) allows keyboard or screen
reader users to jump to main content easily, bypassing repetitive nav links
 [50] .
- **Color Contrast**: All text meets at least AA contrast ratios (4.5:1 for
```

normal text, 3:1 for large text) against its background [51]. We have tested our color palette for this. For instance, if our brand colors are vibrant, we ensure that e.g. white text on brand color or brand color text on white meets contrast. If not, we adjust shades. Also for dark mode, ensure contrast similarly.
- **No reliance on color alone**: We do not convey information solely by color differences. For example, required form fields are not only indicated by red outline but also have an asterisk or text; links are not distinguished solely by color but also perhaps underline (so colorblind users or monochrome view can still tell).
- **Resizable Text**: The design will not break if text is zoomed in (up to 200%). Layouts (especially responsive ones) account for reflow with larger fonts. We use relative units (em/rem) for type where possible so that if a user has a browser default font size larger, our text scales accordingly.
- **Screen Reader Attributes**: Use ARIA labels and roles sparingly but effectively. For example:
  - Icon-only buttons (like a close “X” or a search lens icon) have `aria-label="Close"` or appropriate.
  - Landmark roles like `role="navigation"` aside from `<nav>` to denote nav, `role="main"` if needed, but HTML5 tags cover most.
  - Ensure dynamic content has ARIA live regions if needed (like form error announcements).
- **Media and Animations**:
  - All images have descriptive `alt` text (or empty alt if purely decorative) to support screen reader users.
  - Videos (if any on site, e.g., a demo reel) have captions or transcripts available.
  - Animations: as mentioned, honor `prefers-reduced-motion`. This means in our CSS/JS we check this preference; if true, we disable non-essential animations (parallax, large fades, etc.). This prevents issues for users with motion sensitivity.
  - Also, no animations that flash intensely (nothing flashes >3 times per second to avoid seizure risk) [52].
- **Forms**: Labels are explicitly associated with inputs (using `<label for>` or wrapping input). We also provide additional instructions or error messages as needed. We consider using ARIA `aria-describedby` to tie error text to the field so screen readers read it.
- **Testing**: We will test with accessibility tools (like Lighthouse, axe) and with keyboard and screen reader (NVDA/VoiceOver) to catch issues. We aim to fix any item that would fail WCAG AA success criteria (like missing alt, insufficient contrast, missing form label, etc.).

By doing all above, we aim to ensure that **everyone** – including users with visual, auditory, motor, or cognitive disabilities – can access content and use features. This aligns with Monynha’s inclusive mission. As an inclusive design example: ensuring our website can be *“navigable without using a mouse – the keyboard ‘tab’ button alone should enable users to navigate any page”* [53], which we've implemented.

Dark Mode:
Monynha.com will offer a dark theme toggle (and/or auto based on user OS

preference using `prefers-color-scheme`). Dark mode provides a dimmer color scheme that is easier on the eyes in low light and preferred by many users generally. Implementation:
- We define a color palette for dark mode – often it’s invert of light with adjustments. For example, light backgrounds become dark grey/black, text that was black becomes light grey or white. Accent colors might be slightly desaturated for darkness. We ensure contrast in dark mode too (light text on dark backgrounds).
- Using Tailwind, we can leverage its dark: variant classes or CSS variables:
  - E.g., `bg-white dark:bg-gray-900`, `text-gray-900 dark:text-gray-100` on elements to have dual themes.
  - Or we set up design tokens (in CSS custom properties) that swap on `html.dark` class or media query. Possibly easier: Tailwind’s `dark:` is triggered by adding `class="dark"` on html (we control this with a script or CSS preference).
- Automatic detection: If user OS is in dark mode and they haven’t overridden on site, we can default to dark on first load. But also give a toggle (like a moon/sun icon) to switch. That toggle can store preference in localStorage and add/remove `dark` class on <html>.
- We test images/logos on dark (if our logo is dark text, we might use an alternate white logo in dark mode).
- Focus states and other feedback must also be visible in dark mode (e.g., an outline might need a different color on dark).
- The dark mode respects accessibility too – e.g., color contrast must still meet guidelines. We likely use slightly lighter dark backgrounds (like not pure black but dark grey) because pure white on pure black can be high contrast but harsh; we’ll find a balance that meets AA and is comfortable.

Responsive Design:
We employ a **mobile-first** approach (Tailwind is mobile-first by design). The site will gracefully adapt to:
- **Small screens (mobile)**: The layout will stack elements vertically, navigation collapses to a mobile menu. Font sizes might adjust slightly (maybe same base, maybe slightly smaller if needed to fit headings). Interactive elements remain easily tappable (we ensure buttons/links have adequate height, ideally 44px or more in clickable area per Apple guidelines).
- **Medium screens (tablet)**: Possibly show nav menu if space allows, or still use mobile menu. Multi-column layouts (like a 3-column feature grid) might become 2-column.
- **Large screens (desktop)**: We might enhance with multi-column layouts, sidebars for additional navigation (like on blog, a sidebar for recent posts could appear on desktop, which on mobile becomes a section below content).
- **Very large screens**: We can limit max width of content for readability (~1200-1400px for text content) and center it, so it doesn't stretch too wide. Perhaps the header and footers expand full width with background but main content stays at a comfortable reading width.

We test on actual devices or emulators to ensure no clipping or overflow issues. Images use responsive techniques (like the Next Image component with `sizes` attribute) to load appropriate sizes per device pixel ratio and

```
width.

**Design Tokens**:
We use design tokens for consistent values across light/dark and to aid
maintainability. For instance:
- Colors: define CSS variables like `--color-bg`, `--color-text`, `--color-
primary` etc. On light vs dark we swap their values [54] . We can do this via
Tailwind config or in CSS.
- Spacing, font sizes: likewise, consistent scale to ensure spacing remains
balanced on all device sizes.
- Having tokens means if brand tweak happens (like change primary blue to
slightly different blue), we update in one place.
- It also helps ensure consistent UI: e.g., all shadows are using a token (--
shadow-elevation-1 vs 2 for different depths) so they look uniform.

Design tokens approach *“allows a single source of truth for design values”*
 [55] , which is great for theming and scaling.

**Testing**:
We will verify compliance by running through a checklist (many of which are
enumerated in WCAG) and using tooling:
- Use screen reader (VoiceOver/NVDA) to navigate major pages: does it read
logically, do images announce properly, can forms be filled?
- Use high contrast mode (or simulate color blindness with tools) to ensure
differentiation still exists.
- Validate HTML (ensuring proper nesting and no duplicate IDs etc. to meet
4.1.1 parsing rules).
- Check contrast via tools for all text/background pairs.
- Simulate low bandwidth to see if any UI elements break without images or if
alt text is present.

By treating accessibility and responsive design as core requirements, not
afterthoughts, we uphold Monynha’s values of inclusion. The site will be
*usable for everyone and in every context*. And by including dark mode, we
also cater to user personalization/preferences, which many modern users
appreciate.

In essence, Monynha.com will exemplify good citizen web design: it *“ensures
perceivable information, operable UI by all (keyboard, AT), understandable
content, and robust compatibility with various tools/agents”* [56] . Combined
with theming and responsiveness, the interface flexes to user needs and
conditions, making for a truly user-centric experience.

### 4.4 Design Tokens and Consistent Branding

Monynha.com’s UI design is governed by a system of **design tokens** that
ensure consistency across all components and facilitate easy theming. Design
tokens are essentially a repository of **reusable, atomic design values** –
things like colors, typography styles, spacing units, and more – which form
the foundation of the design system. By using tokens, we replace hardcoded
values with semantic, self-explanatory names that can be managed in one place
```

```
[57] [55]. This approach yields a cohesive look and feel, and makes updating the
branding or themes straightforward.

Key categories of design tokens and how we apply them:
- **Color Tokens**: We define a palette of colors as tokens, e.g.,
  - `--color-primary`, `--color-primary-light`, `--color-primary-dark` for
brand primary color in different contexts,
  - `--color-bg` and `--color-text` for standard background and text,
  - `--color-accent` for secondary highlights,
  - `--color-success`, `--color-error` for feedback states.
  Each token holds a color value (in HEX or CSS `rgb()` etc.), and these are
used throughout CSS (either directly or via Tailwind if we integrate them
into Tailwind’s config as theme colors).
  For instance, instead of using `#3456FF` in multiple places, we use `var(--
color-primary)`. If the brand color changes or if dark mode needs a different
primary, we adjust the token.
  We ensure the naming is meaningful, representing role/purpose, not just
appearance (so "primary" vs "blue", because maybe in dark mode primary might
not actually be a pure blue but a lighter variant, etc.).
- **Typography Tokens**: These include font families and size/line-height
scales.
  - e.g., `--font-family-sans` (e.g., `"Inter", sans-serif` if that's our
chosen font),
  - `--font-size-base`, `--font-size-lg`, `--font-size-sm`, etc.,
  - `--line-height-base`, etc.
  Using tokens for type means consistency in text sizing and easier
adjustments for responsiveness. Tailwind typically covers responsive
typography with modifiers, but we can incorporate tokens by customizing
Tailwind theme or by CSS variables for font sizing.
  Also weight tokens if needed (like `--font-weight-bold: 700`, etc., though
usually using classes is fine).
- **Spacing Tokens**: Consistent spacing scale (e.g., spacing units 4px, 8px,
16px etc. could be tokens or tailwind’s default spacing scale, which is
essentially design tokens).
  - e.g., `--spacing-xs: 4px`, `--spacing-sm: 8px`, `--spacing-md: 16px`, `--
spacing-lg: 32px`, etc.
  We then use these in margins/paddings (like `margin: var(--spacing-md)`).
  This ensures vertical rhythm and whitespace is uniform. If we find overall
spacing needs adjusting (maybe more whitespace for a cleaner look), we tweak
a couple of these values.
- **Sizing Tokens**: e.g., for common component sizes or container widths.
Tailwind’s container and width classes might suffice, but if we have
something like `--content-max-width: 1200px` as a token, we can use that for
the main container max width.
- **Border Radius and Shadow Tokens**:
  - e.g., `--radius-sm: 4px`, `--radius-md: 8px` for rounding corners of
buttons, cards, etc., so they all have same curvature.
  - `--shadow-sm`, `--shadow-lg` as values for box-shadow styles. By using
tokens, if we adjust the elevation intensity or color (like maybe adding
subtle brand color to shadows), we do it uniformly.
- **Animation Tokens**: Duration and easing curves can be tokens:
  - e.g., `--duration-fast: 150ms`, `--duration-medium: 300ms`,
  - `--easing-default: cubic-bezier(0.4, 0, 0.2, 1)` (standard ease in-out),
`--easing-spring: cubic-bezier(...)` if we have a springy one.
  Then all animations refer to those, which ensures the timing is consistent
(so nothing oddly slow or too fast relative to rest).
- **Breakpoints**: If customizing, though Tailwind’s sm, md, lg are
essentially tokens (sm = 640px etc.). Could treat them as tokens
conceptually.

By leveraging design tokens in this way, our development process gets a
boost:
- Designers and developers share a common language – tokens names – which
reduces confusion.
- Updating branding (like changing the primary color’s shade or global font)
is straightforward as it's changing the token’s value and all UI reflecting
that automatically.
- It prevents "drift" where two similar colors or sizes might accidentally
get used; tokens enforce a constraint to predefined values, which fosters
unity in design.

We ensure that design tokens are integrated into our tools:
- In Tailwind, we can customize theme with our tokens (for example, set
Tailwind `colors: { primary: 'var(--color-primary)' }` so that our classes
use tokens under the hood).
- Alternatively, we apply CSS variables to `:root` and then in our CSS (or
Tailwind via plugin) use them. Tailwind now has JIT which may not directly
parse CSS var in config but we can do a workaround or simply use classes for
each token value.

**Consistent Branding**:
Beyond tokens, consistency is also about overall brand presence:
- We use the same logo (in correct colors) in header and footer. The brand
iconography style remains consistent (if we use a certain style of
illustration or icon set, we stick to it site-wide).
- Tone of voice in content is consistent, but that’s content side. Visually,
every page looks like part of the same site:
  - Use of colors, shapes (for instance, if we use rounded cards with a
slight shadow as a motif on one page, we use them for similar content
grouping elsewhere).
  - Use of imagery: If brand guidelines say use abstract shapes in the
background, we incorporate them in various pages subtly so it ties together.
  - Buttons styles are uniform (primary button style vs secondary style are
defined and reused).
  - Spacing scale usage is consistent (no random huge gap on one page that’s
out of line with rest).

To maintain this, we might create a Storybook or design reference for
components so that we reuse those rather than creating new styling from
scratch each time. The tokens approach is akin to what Material Design or
others do – *“small, reusable design decisions that make up a design system's
visual style”* [57] , exactly how we treat them.
```

Example: Suppose the brand primary color is a teal. We set `--color-primary: #14B8A6` (teal500). All buttons use `background-color: var(--color-primary)`. Links might use it for hover or underline. If one day branding says "we prefer a slightly more bluish teal", changing that hex in one place updates all those elements at once. Because tokens *“provide a centralized way to manage and update design properties across the design system”* [55] .

Also, by having tokens for states (like success, warning), if we decide the success green in light mode should be different in dark mode for better contrast, we simply define `--color-success` differently under `.dark`.

Theming:
If in future Monynha had multiple theme variants (like maybe a Pride month theme with rainbow accent?), tokens allow that by swapping out values for the period or via toggling a class. It's forward-looking.

Overall, the heavy use of design tokens underpins our commitment to a professional, scalable design system. It avoids arbitrary, ad-hoc styling and ensures that as the site grows, it won't devolve into inconsistent styles. It also ties into the Clean Architecture/DRY approach but for design: a single source of truth for each style decision, avoiding repetition of magic numbers or colors. As the user browses, everything feels harmonious and intentional – which builds trust and brand recognition. This consistency yields a UX where nothing is jarringly off-brand; instead, every page and element reinforces Monynha’s identity through a unified aesthetic.

---

5. AI-Assisted MVP Generation (Lovable Prompt)

In order to efficiently build out the Monynha.com platform as envisioned, we plan to leverage an AI-assisted development approach using a generative AI tool (such as **Lovable AI** or a similar full-stack code generation platform). This involves crafting a comprehensive and precise prompt that instructs the AI to generate the initial codebase – covering both frontend and backend – aligned with our specifications. By doing so, we aim to bootstrap the Minimum Viable Product (MVP) quickly, while ensuring the output adheres to our technical stack and requirements. This section details the approach and content of the prompt, the generation scope and priorities, and how the project will be structured for further development.

5.1 Full-Stack Generation Approach

The chosen AI tool (Lovable AI, as an example) is capable of **generating a complete application** given a natural language prompt that describes the desired outcome [58] . Our strategy is to describe the Monynha.com site in terms that the AI can translate into code for both the Next.js frontend and the Supabase (or Node/Payload) backend. The prompt will serve as a blueprint, much like the content of this document, but tailored to instruct an AI on what to build.

```
Key points of the approach:
- **Front and Back Separation**: We will explicitly instruct the AI to create
a Next.js 14 project (with TypeScript) for the frontend and to set up a
Supabase PostgreSQL database (with relevant schema) for the backend. If the
AI environment supports it, it might also generate serverless functions or
Node.js code for Payload CMS or any needed server logic. Essentially, *“full-
stack generation: frontend + backend + database”* is a core feature of
Lovable [58] , and we intend to use it.
- **Technical Stack Guidance**: The prompt will clearly state the tech stack:
Next.js 14 with App Router structure, using Tailwind CSS and shadcn/UI (which
implies including Radix UI and Tailwind), integration with Supabase (maybe
using Supabase JS client in the Next app for any dynamic calls), and using
Payload CMS for content. We'll instruct how these pieces connect (e.g., "use
Supabase connection URL for Postgres in Payload config" etc.). The more
specific we are, the better the AI can include the correct dependencies and
file structure.
- **File/Directory structure**: We mention high-level structure expected:
  - An Next.js `app/` directory with subfolders for pages (like `app/(main)/
about/page.tsx`, etc., maybe grouping if needed). If we use Next 14, we can
specify using the new /app router with layout and page components.
  - Components directory for reusable components (NavBar, Footer, etc.).
  - A styles setup (Tailwind config, global.css).
  - Integration files for Tailwind (postcss config, tailwind.config).
  - For Supabase, maybe a `schema.sql` or using Supabase CLI the AI might not
do, but it could output table definitions.
  - If generating Payload code: that implies a separate `cms/` directory or
even a separate server. Possibly out-of-scope for the AI in one go, but we
can at least have it outline the models.
- **API routes**: In Next, define needed API routes or server actions. For
example, an API route for contact form submission (if not using direct
supabase client in front). We'll instruct something like: "Implement Next.js
API route `/api/contact` to handle form submission, storing data in Supabase
(contact_messages table) and returning success." The AI should then produce
that code (using Supabase Node client).
- **State management**: If needed for any interactive parts (maybe minimal),
the prompt might not need to mention it explicitly, but if there's something
like a search or filter on the front-end (maybe for blog posts by tag), we
can instruct to use React hooks or context as necessary.

A well-structured prompt ensures that the AI's output is coherent and close
to our target. For instance, Lovable expects a description of the app:
*“describe the app you want... Lovable generates UI, database, and backend
logic ready.”* [59] . We'll do exactly that:
We will break the prompt into sections so the AI understands each part:
- Project Overview (what Monynha.com is, its purpose and audience).
- Pages and Routes (list out each page, what should be on it).
- Tech stack specifics (versions, libraries, e.g., "Use Next.js 14 with the
new App Router structure. Use Tailwind CSS for styling. Integrate shadcn/UI
components for accessible UI elements (like modal, dropdown). Setup Supabase
as the primary database and interface via Supabase JS client. Use Payload CMS
to manage content with the following collections...").
- Data models (outline each table or Payload collection with fields and
types).
- API and logic (e.g., "Provide a contact form submission handler that writes
to database and sends an email (if we wanted email sending, but maybe skip
for code gen MVP or simulate it)").
- Auth integration ("Implement authentication via monynha.me SSO by providing
an OAuth callback route and using a placeholder JWT verification, but for now
it can be stubbed as this is external." We might not get actual SSO code from
AI if it's complex, but we can request a dummy flow or mention "monynha.me
provides JWT, assume it's available and focus on verifying and protecting
routes x and y").
- Multilingual feature ("Configure Next.js i18n for locales en, pt-BR, es,
fr. Provide example of loading locale-specific text, possibly with a simple
JSON or using next-i18next if known. Use fallback to en for missing.
Implement a language switch in the navbar.").
- Accessibility and SEO: instruct the AI to use semantic HTML, include alt
tags, etc., and add `<Head>` metadata for pages.
- The AI may not perfectly enforce all that, but including it increases the
chance. Tools like Lovable likely already consider some best practices, but
explicit ask helps.
- Testing or misc: "Ensure the build runs without errors" might not need
stating, but we can say "Project should run with `npm run dev` without
errors."

### 5.2 Crafting the Technical Prompt (Stack, Packages, and Schema)

The prompt we give to the AI will be essentially a condensed yet specific
version of this specification. We will phrase it in imperative or descriptive
language that the AI parser can understand, structured likely as bullet
points or paragraphs covering each aspect.

Key elements to mention:
- **Stack and Packages**:
  - "Create a Next.js 14 application using TypeScript and the App Router. Use
Tailwind CSS for styling and include `@shadcn/ui` components (which depends
on Radix UI) for pre-built accessible UI."
  - "Set up Tailwind with the default configuration and any necessary plugins
(e.g., typography for rich text styling, forms for better form styles)."
  - "Use Supabase (PostgreSQL) as the database. Use the Supabase JavaScript
client to interact with the database from the Next.js app (for any dynamic
operations, like form submissions or server-side data fetching)."
  - "Integrate Payload CMS: scaffold a basic Payload CMS configuration in a
separate directory or as part of the Next.js API routes. Define collections
as per content models (Products, BlogPosts, etc.). Alternatively, you can
simulate CMS content as static JSON if direct integration is complex; focus
on the schema design."
  - If Lovable doesn’t directly support generating a separate CMS server, we
may instruct the AI to produce content fetching via dummy JSON or supabase
tables for now. Possibly better to not confuse it too much; maybe treat CMS
content as just coming from Supabase tables for generation.
```

```
  - "Implement authentication placeholder hooking to monynha.me (e.g., stub
an OAuth callback route)."
- **Schema**:
  - We will list the tables and fields. For example:
    - "Create a table `products` with columns: id (uuid primary key), slug
(text unique), title (text), description (text), features (json or text),
image_url (text), created_at, updated_at timestamps."
    - "Create a table `posts` with columns: id, slug, title, content (text),
author (text for now), published_at, etc."
    - "Create a table `contact_messages` with id, name, email, message,
created_at."
    - If authors separate: an authors table with id, name, etc.
    - We should mention relations if any (like foreign key from posts to
authors).
    - We might instruct to use Prisma or Supabase migrations if needed, but
Lovable likely uses an internal method. Maybe just specify in plain terms and
the tool does it.
    - If using Payload, define collections similarly, but that might be too
advanced for a generic tool unless it knows Payload specifically.
  - "Apply Row Level Security policies accordingly (Public read on products/
posts, insert-only on contact messages)" – but this might be too detailed for
codegen at MVP. Perhaps skip deep RLS detail; we can manually add that. We
could mention "enable RLS on all tables by default and only allow appropriate
operations."
- **Functional prompt details**:
  - We describe what each route should do:
    - "/" – show overview, possibly recent blog posts (so AI might code
fetching posts and rendering titles).
    - "/about" – static content, can be hard-coded or from a JSON.
    - "/solutions" – static content.
    - "/product/[slug]" – dynamic page. The AI needs to generate routing for
dynamic segments in Next (in App Router, that means a folder [slug] with
page.tsx). We tell it how to fetch the product by slug (maybe using a
Supabase query in `generateStaticParams` or `getStaticProps`).
    - Same for "/blog/[slug]".
    - We instruct to use Next's data fetching: e.g., "Use Next.js dynamic
routes for products and blog. Fetch data in `getStaticProps` (or the new
`generateStaticParams` and `getStaticProps` in App Router context, i.e., use
fetch in a Server component or use fetch from supabase restful endpoint on
server side). Ensure pages are pre-rendered when possible."
    - "Implement a navigation bar with routes to Home, Solutions, Products,
Open Source, Blog, Careers, About, Contact, and a language switcher. The nav
bar should collapse into a mobile menu on small screens."
    - "Implement a footer with basic info and external domain links."
    - "Implement form handling: the Contact page includes a form with fields
Name, Email, Message. On submit, an API route `/api/contact` should insert
into the database and return JSON success. Display a success alert on front-
end on completion."
    - "Implement language support: For simplicity, create a dictionary object
or JSON files for each of the four languages with a few key strings (like
site title, nav labels). Provide a mechanism to switch language which re-
```

```
renders the page in the chosen language (this might be a challenge to fully
implement in code generation; but we can instruct next-intl integration or
just show an example multi-language text)."
    - Possibly simpler: "Include i18n routing in next.config and demonstrate
one translated page to illustrate (like /fr/about showing French text vs /
about English)."
  - "Ensure all pages are mobile-responsive (use Tailwind utility classes for
grid/ flex that adapt)."
  - "Use shadcn/UI components for any complex component like modal if needed
(maybe none needed actually except nav menu could use headless UI popover).
But we can ask to use shadcn's Alert component for success message, etc."

The prompt might become lengthy, but Lovable is said to handle detailed
prompts well. In fact, *“using clear and structured prompts”* is recommended
[60]. We'll likely format it with bullet points or numbered lists for clarity,
which can be helpful for the AI to break down tasks.

### 5.3 MVP Feature Priorities and Scope

The MVP (Minimum Viable Product) for Monynha.com will focus on implementing
the core features and pages with baseline functionality, while some of the
more advanced or polish items can be slated for post-MVP enhancements. It's
important that the prompt (and thus the AI-generated output) concentrates on
these priorities so that the initial deliverable is coherent and usable.

**MVP Inclusions (Scope)**:
1. **All main pages and routing** as identified:
   - Home, About, Solutions, Contact, OSS, Blog (listing + posts), Careers,
Governance.
   - Product pages dynamic.
   These should all be present, even if some are stubbed with placeholder
content (the structure matters, content can be refined).
2. **Basic Content Rendering**:
   - Display content for each page either from static source or from
database. For MVP, static content for About/Solutions/Governance is fine
(just hardcode in JSX or use a simple JSON).
   - Dynamic content (Products, Blog posts) should be fetched from some data
store (to demonstrate that functionality). Could use Supabase calls to a pre-
filled table, or static JSON objects in code for now if connecting live is
heavy.
   - List pages (like blog listing or product listing if needed on main
products page) should loop through sample data and render multiple items with
links.
3. **Navigation & Linking**:
   - A working nav menu and footer on all pages (so layout component with
header/footer).
   - The links navigate correctly (client-side transitions working).
   - The hub links to other domains can just be static hrefs in footer or
top, not much logic.
4. **Contact Form working**:
   - The form collects input and triggers an API route which stores data and
```

```
responds (the "happy path"). We may not integrate email sending at MVP (that
can be a next step via a service).
   - Just confirm it writes to DB or logs on server and displays success
message to user.
5. **Styling and Responsiveness**:
   - The site should be styled nicely (Tailwind default styles plus any
custom as needed to match brand colors).
   - On mobile view, ensure menu works and content stacks, etc.
   - Doesn’t need to be pixel-perfect to some custom design (we might not
have one in prompt), but should look modern and clean.
   - If using shadcn/UI, ensure to import the global CSS for it (since shadcn
uses tailwind classes but might need certain setup).
6. **Multilingual infrastructure**:
   - At least have the Next.js i18n config in place and maybe demonstrate one
or two strings or pages translated. Full content translation for all is not
needed for MVP, but structure should allow it.
   - We can for MVP just show a language switch toggling a state that changes
a text string as proof of concept.
7. **Accessibility basics**:
   - Proper labels on form fields, alt text on any images used (we can
instruct alt = "" on decorative or fill with descriptive on meaningful).
   - Keyboard nav and focus states should be naturally okay with HTML +
Tailwind (we will ensure no outline-none usage except maybe on custom focus
styling).
   - Not expecting AI to handle advanced ARIA but basic semantics should be
fine.
8. **Project setup**:
   - Make sure the AI includes a `package.json` with needed dependencies
(Next, React, Tailwind, maybe `@supabase/supabase-js`, etc., and for shadcn
maybe needed Radix packages or headlessui).
   - Provide instructions or script for database setup if possible (maybe
just mention tables creation in a README).
   - The output might include some readme notes. We should prompt it to
"include necessary instructions for running the project (like environment
variables needed: Supabase URL, anon key, etc.)".

**MVP Exclusions (what can be deferred)**:
- Pixel-perfect fine-tuning of layout and design beyond a reasonable
baseline. Once AI gives code, human can refine spacing or styling
differences.
- Complex SSO integration. We'll stub login maybe by simulating a user (like
assume user id "demo" logged in for any gated content if needed). Or simply
have a login button that links to monynha.me without actual token handling.
Real SSO integration is beyond MVP and can be done manually later or with a
specialized library.
- Comprehensive CMS integration. Instead of fully hooking Payload CMS UI, we
might accept that content is loaded from static files or Supabase directly.
The AI might not spin up a separate CMS app easily. We can integrate Payload
after MVP manually if needed. For MVP, just having the data models and maybe
a simple seeding approach is fine.
- Advanced error handling or form validation (maybe just basic required check
on contact form).
- Testing (AI likely won't write tests unless asked; we skip tests for
initial MVP, focusing on features).
- Performance optimizations like image optimization with next/image can be
included, but if not we can add later. The AI may include next/image by
default for static images if any.
- Security aspects like RLS or rate limiting on contact form – these can be
done after MVP.

**Project Structure after Generation**:
We expect a structure something like:
```

monynha-portal/ ├── app/ │ ├── layout.tsx (with header & footer import) │ ├── page.tsx (Home page) │ ├── about/page.tsx │ ├── solutions/page.tsx │ ├── contact/page.tsx (with form) │ ├── product/ │ │ ├── page.tsx (maybe list of products if needed) │ │ └── [slug]/page.tsx (product detail) │ ├── blog/ │ │ ├── page.tsx (list posts) │ │ └── [slug]/page.tsx (post detail) │ ├── careers/ page.tsx │ ├── governance/page.tsx │ └── oss/page.tsx ├── components/ (NavBar.tsx, Footer.tsx, maybe others like LanguageSwitcher.tsx) ├── lib/ (maybe supabaseClient.ts init file) ├── pages/api/ contact.ts (API route for form handling; in App Router, we could also use route handlers under app, e.g., app/api/contact/route.ts) ├── public/ (assets like logo images if any) ├── styles/ (globals.css with Tailwind base imports) ├── supabase/ (maybe sql schema or config if any) ├── package.json ├── next.config.js (with i18n settings) └── tailwind.config.js

```
This is what we'll aim for from the AI.

We will instruct some of that (like "organize pages under Next.js App Router
with appropriate folder structure as above").

**Prompt Example Outline**:
```

[Project Overview] Monynha.com is an institutional multi-language website for Monynha Softwares, built with Next.js 14, Tailwind CSS, and Supabase (PostgreSQL) for data. It includes static informational pages and dynamic content (products and blog posts). The site should be accessible (WCAG 2.1 AA) and responsive, with a dark mode toggle.

[Tech Stack Requirements] - Use Next.js 14 (App Router, TypeScript) for the frontend. - Use Tailwind CSS for styling. Configure a base Tailwind setup (including dark mode using class strategy). - Use shadcn/ui (Radix UI) for accessible components where appropriate (e.g., modal or alert). - Use Supabase (Postgres) as the backend database. Set up tables as described and use the Supabase JS client or API routes to interact with the data. - Include any necessary packages: `@supabase/supabase-js` for database, etc.

[Data Models / Database Schema] Define database schema: - Table `products`: columns = id (UUID, PK), slug (text, unique), title (text), description (text), features (text or JSON), image_url (text), created_at (timestamp). - Table `posts`: columns = id (UUID), slug (text unique), title (text), content (text), author (text), published_at (timestamp). - Table `contact_messages`: columns = id (serial PK), name (text), email (text), message (text), created_at (timestamp). - Add sample data for products and posts for demonstration. (You can assume the database is pre-populated or provide a seed script.) Enable Row Level Security on tables (but allow anon read on products/posts, and anon insert on contact_messages).

[Pages and Features] - **Layout**: Create a common layout with a header and footer used on all pages. - Header: shows site name/logo and a navigation menu linking to Home "/", Solutions, Products, Open Source, Blog, Careers, About, Contact. Also include a language switcher (just a button or dropdown for "EN, PT, ES, FR" that switches site language). - Footer: repeat key links and list other Monynha domains (monynha.me, monynha.tech, monynha.store, etc.) as external links. - **Home Page ("/")**: A welcome section summarizing Monynha Softwares' mission (use placeholder text). Include maybe a brief highlight of 1) a product, 2) a latest blog post, and 3) a call-to-action to contact. (For example, display the title and summary of the first product and first blog post from the database). - **About Page ("/about")**: Static content describing the brand (placeholder text about the company, its values like LGBTQ+ inclusion and class solidarity). - **Solutions Page ("/solutions")**: Static content describing what services/solutions Monynha offers. - **Products Page ("/product")**: List all products from the `products` table. For each product, show its title and short description and a link to its detail page. - **Product Detail ("/product/[slug]")**: Dynamic route. Fetch the product by slug from the database. Display its title, full description, and list of features. If image_url is provided, display the product image. If product not found, return 404. - **Open Source Page ("/oss")**: Static page listing open-source projects (you can hardcode 2-3 project names with links to GitHub as examples). - **Blog Listing ("/blog")**: Fetch list of posts from `posts` table (order by published date). Display title and maybe date for each, linking to their page. - **Blog Post ("/blog/[slug]")**: Dynamic route. Fetch post by slug. Display title, author, date, and content (allow content to be HTML or markdown rendered). If not found, 404. - **Careers Page ("/careers")**: Static page listing job openings (you can list 1-2 example positions with title and short description). - **Governance Page ("/governance")**: Static content about the company's governance or community (placeholder text). - **Contact Page ("/contact")**: A contact form with fields: Name, Email, Message. Use proper labels. On submit, call an API route to insert a new record into `contact_messages`. Provide user feedback: - If successful, display a success message "Thank you for your message!" (you can use a shadcn/ui Alert or a simple div). - If error, show an error message. The API route (`/api/contact` or app/api/contact/route.ts) should validate input (basic, e.g., email not empty) and then use Supabase client to insert into `contact_messages` table. Return JSON response.

[Internationalization] - Configure Next.js internationalized routing for locales: "en" (default), "pt-BR", "es", "fr". - Provide example translated text for one or two pieces of content (for instance, have a JSON or object mapping "Welcome to Monynha" in those languages). Implement the language switcher button in the header that, when clicked, changes the locale (you can use `next/link` to the same page with the locale or Next.js router). - For MVP, it's acceptable to only partially translate pages (just to demonstrate the mechanism). - Ensure all pages use `<html lang={currentLocale}>` and that navigation links respect locale (e.g., link to "/es/about" for Spanish).

[Theming and Dark Mode] - Implement dark mode support using Tailwind's dark variant. Define a dark color scheme (e.g., dark background, light text). Allow toggling dark mode with a button (sun/moon icon in header). You can manage dark mode by adding a `dark` class on <html>. - Use CSS variables or Tailwind config to ensure consistent colors in light vs dark. - Example: body background is white in light, #121212 in dark; text is near-black in light, near-white in dark; primary brand color may remain same or adjust slightly for dark contrast.

[Accessibility] - Use semantic HTML elements (nav, main, footer, etc.). - Ensure each form field has a <label>. - Include alt attributes for images (placeholder alt text is fine for now). - Ensure focus styles are visible (Tailwind by default shows outline; do not disable it). - Include a "Skip to main content" link at top for screen readers. - All interactive elements (buttons, links) should be accessible via keyboard (native HTML behavior is fine).

[Responsiveness] - The layout must be responsive: - On small screens, use a mobile menu (e.g., a burger icon that toggles the nav links). You can implement a simple state in the Header component to show/

hide the menu. - Sections and images should stack or scale for mobile (use Tailwind grid/flex utilities). - Test main pages in mobile width and ensure no content overflows.

[Project Structure and Setup] - Organize pages using Next.js App Router (create folders and page.tsx files as needed for routes above). - Create a `_app` or global layout that includes the header and footer on every page. - Add a Tailwind CSS global stylesheet (with @tailwind directives for base, components, utilities). - Create a supabase client instance (with URL and anon key from environment variables). Use environment variables for any keys (e.g., `NEXT_PUBLIC_SUPABASE_URL`, `NEXT_PUBLIC_SUPABASE_ANON_KEY`). - Ensure to import and configure shadcn/ui styles (if needed, e.g., import tailwind base for Radix). - Provide any necessary configuration files (tailwind.config.js with dark mode 'class'). - After generation, the app should run (e.g., `npm install` then `npm run dev`) and the pages should render with the described functionality.

```

The above could be the essence of the prompt (though it may be long; we trust the AI to parse it).

Finally, instruct the AI: "Please output the project code including all relevant files and a brief README on how to run it."

Given Lovable's capabilities described [61] [58], it should create a working full-stack code. It might even handle DB schema in a Prisma or SQL file.

We'll likely need to review and refine the AI output, but this gets us a big head start, which is the goal.

In summary, by using this detailed prompt with an AI app builder, we aim to rapidly generate the scaffolding and much of the implementation for Monynha.com’s MVP. This allows the team to then focus on fine-tuning, adding content, and any complex custom logic that the AI might not handle. It's an acceleration strategy in line with Monynha’s innovative spirit – *going "from prompt to product in minutes"* [62], thereby saving development time and quickly bringing our institutional site to life.

---

[1] Diminutives for family & others : r/Portuguese
https://www.reddit.com/r/Portuguese/comments/1ejh7mh/diminutives_for_family_others/

[2] LGBT Tech Statement on The Next Four Years
https://www.lgbttech.org/post/lgbt-tech-statement-on-the-next-four-years

[3] What is Digital Inclusion? | IxDF
https://www.interaction-design.org/literature/topics/digital-inclusion?srsltid=AfmBOooL_BnXwyR7EfXiEB9SWZ-zUxIR4jvYpaXkkNMtB9qkqyLrErTA

[4] The Importance of Open Source | argile
https://argile.agency/insights/fr/insight-2/

[5] [6] [7] Understanding some of the routing mechanisms using App Router in Next.js 14
https://www.qed42.com/insights/understanding-some-routing-mechanisms-app-router-next-js-14

[8] [9] [10] [54] Tailwind CSS - Rapidly build modern websites without ever leaving your HTML.
https://tailwindcss.com/

[11] [12] [13] My Tiny Guide to Shadcn, Radix, and Tailwind | by Mairaj Pirzada | Medium
https://medium.com/@immairaj/my-tiny-guide-to-shadcn-radix-and-tailwind-da50fce3140a
```

14 28 29 30 31 32 33 34 35 36 37 38 39 Row Level Security | Supabase Docs
https://supabase.com/docs/guides/database/postgres/row-level-security

15 17 18 Setting Up Payload with Supabase for your Next.js App: A Step-by-Step Guide
https://payloadcms.com/posts/guides/setting-up-payload-with-supabase-for-your-nextjs-app-a-step-by-step-guide

16 19 Why Payload CMS is the Best CMS for NextJS
https://focusreactive.com/payload-best-nextjs-headless-cms/

20 47 Don't repeat yourself - Wikipedia
https://en.wikipedia.org/wiki/Don%27t_repeat_yourself

21 API-First Design: Key Principles Explained
https://www.linkedin.com/pulse/api-first-design-key-principles-explained-ashutosh-shashi-rvxbe

22 23 24 Guides: Internationalization | Next.js
https://nextjs.org/docs/pages/guides/internationalization

25 26 27 An Introduction to Single Sign-On | All You Need to Know
https://curity.io/resources/learn/single-sign-on-introduction/

40 42 43 44 45 The Role of Micro-interactions in Modern UX | IxDF
https://www.interaction-design.org/literature/article/micro-interactions-ux?
srsltid=AfmBOoquDsoCzNrvSoJ-9pft7F4nD1SvfaBv_BDhtvtOqNbkCJr0fxA9

41 46 48 The Power of Micro Interactions: Boosting User Engagement and Usability - Lantern
https://lanternstudios.com/insights/blog/the-power-of-micro-interactions-boosting-user-engagement-and-usability/

49 50 51 52 53 56 A Guide to WCAG | Web Accessibility Guidelines Overview
https://www.wcag.com/resource/what-is-wcag/

55 What are Design Tokens? - by UXPin
https://www.uxpin.com/studio/blog/what-are-design-tokens/

57 Design tokens – Material Design 3
https://m3.material.io/foundations/design-tokens/overview

58 59 61 V0 vs Lovable: Which AI App Builder Should You Choose? | UI Bakery Blog
https://uibakery.io/blog/v0-vs-lovable

60 62 Lovable AI Features Deep Dive: From Prompt to Product in Minutes
https://www.sidetool.co/post/lovable-ai-features-deep-dive-from-prompt-to-product-in-minutes