Pennington

Under the Hood

Version 1.0.0

Version 1.0.0
Generated June 2026
https://usepennington.net/
Produced with Pennington 0.1.6-alpha.0.11

Contents

CHAPTER 1

Core

The Pennington mental model

Under the Hood A short map of the host, content sources, rendering pipeline, response pipeline, and DocSite/BlogSite templates before you dive into the deeper architecture pages.

Pennington is easiest to understand as a normal ASP.NET app with a content engine inside it. You run the app while writing, and you ask the same app to build static files when publishing.

That single idea explains most of the project:

- The host is ASP.NET. `Program.cs` owns dependency injection, middleware order, Razor components, and any custom endpoints.
- Content sources discover pages. Markdown folders, Razor pages, redirects, generated API reference, taxonomy pages, and custom services all report routes into the same site model.
- The content pipeline turns discovered content into rendered content. A page moves from "I know where it is" to "I parsed its front matter" to "I rendered its HTML"; failures travel through the same pipeline so the build report can name them.
- The response pipeline finishes the HTML. Cross-references, locale prefixes, base URLs, live reload, diagnostics, and other response processors run against the actual HTTP response.
- Build mode crawls the host. `dotnet run` serves through Kestrel; `dotnet run -- build` starts the same app on an in-process test server, requests every discovered route, and writes the responses to disk.

The layers

`Pennington`

`Pennington` is the lower-level engine. It gives you content discovery, markdown parsing, rendering, route resolution, response processing, diagnostics, search artifacts, feeds, and static output. You bring the site shell: layout, navigation markup, styling, and any app-specific endpoints.

Start here when you are embedding content into an existing ASP.NET app, building a custom layout, or mixing several kinds of content that do not fit a stock documentation or blog template. The first getting-started arc walks this path: Create your first Pennington site.

`Pennington.DocSite`

`Pennington.DocSite` is a pre-assembled documentation site on top of the engine. One `AddDocSite` call wires the engine, markdown content, DocSite layout, sidebar navigation, search, MonorailCSS styling, SPA navigation, feeds, and static build behavior.

Start here for a conventional documentation site. You can still customize options and add extension points, but the template owns the article-shaped layout. The scaffold tutorial starts here: Scaffold a documentation site with DocSite.

`Pennington.BlogSite`

`Pennington.BlogSite` is the same idea for a blog-first site. It is a template, not a separate engine: the underlying runtime is still Pennington content discovery plus the shared ASP.NET request pipeline.

Use it when the blog is the site. If you want a blog alongside documentation, use DocSite's built-in blog folder instead of combining DocSite and BlogSite in the same app.

`Pennington.UI`

`Pennington.UI` is the Razor component library — table-of-contents and outline navigation, breadcrumbs, pagination, cards, badges, code blocks, callouts, the search modal, and the client-side SPA navigation script. DocSite and BlogSite reference it for you and pre-register the markdown-facing components, so you rarely call it directly on a template.

You meet it the moment you go bare-engine. `AddPennington` does not assume a layout, so a custom shell that wants the same navigation chrome or wants to drop components into markdown pulls these in itself. The components and their parameters are catalogued in the UI reference: Content components, Navigation components, and Utility components. To use them inside markdown, see Drop a Razor component into a markdown page.

`Pennington.TreeSitter`

`Pennington.TreeSitter` is the optional code-fragment extractor. It powers the `:symbol` fence — addressing a member by its name path so a code block shows one method instead of a whole file — across the languages tree-sitter parses.

It is opt-in: register it with `AddTreeSitter` and point `ContentRoot` at the root that holds the source you want to fence. Nothing else changes, so you add it only when a site embeds live source. The recipes live in Embed focused code samples.

Terms you will see

```
Term Meaning     Host The ASP.NET app that registers Pennington and handles requests.
Content source A service that discovers routes and records for pages or generated artifacts.
Front matter Typed metadata parsed from the YAML block at the top of a markdown file.
Route The URL and output-file mapping for a page or endpoint.   Xref A symbolic link such as
`<xref:tutorials.docsite.scaffold>` that resolves to the current route for a page or API
member.   Response processor A hook that rewrites the HTTP response before it reaches the
browser or static output folder.   Build report The diagnostics summary printed by `dotnet
run -- build`.
```

What to read next

- Build a bare host: Create your first Pennington site
- Scaffold a documentation site: Scaffold a documentation site with DocSite
- Understand why dev and build share one app: Dev mode and build mode share one code path
- Understand the content pipeline internals: The content pipeline and union types
- See what DocSite and BlogSite wire for you: What the DocSite and BlogSite templates wire for you

The content pipeline and union types

Under the Hood Why Pennington models content as a four-case union that flows Discovered to Parsed to Rendered — with Failed as a peer case rather than an exception.

Why does Pennington model its content flow as a union of four records instead of a single `ContentItem` class with a status field or a polymorphic base class?

Context

A content engine has to move each page through at least four distinct phases — discovery, parsing, rendering, and emission — and each phase legitimately knows different things about the item. A discovered file has a source location and a route, but no parsed front matter or markdown body. A parsed item has structured metadata and text, but no HTML. A rendered item has HTML and an outline, ready to write to disk. These are not the same data at different points in time; they are genuinely different shapes.

The conventional escape routes both have a cost. A single `ContentItem` class with nullable `Metadata`, `Html`, and `Error` fields invites "is it safe to touch this yet?" checks at every call site, and the compiler has no way to enforce which combination of fields is populated at which stage. A traditional inheritance hierarchy — `ContentItem` → `ParsedItem` → `RenderedItem` — puts discovery-stage code and render-stage code in a subtyping relationship that does not reflect how they are actually used, and forces the failure case into either a parallel branch that breaks `is`-checks downstream or a nullable error field on every subclass. C# 15 discriminated unions offer a third path: the compiler tracks which case you hold, pattern matching is exhaustive, and each case carries exactly the fields that exist at that stage.

How it works

The union shape

`ContentItem` is a union of four record cases — `DiscoveredItem` (route + source), `ParsedItem` (adds metadata and raw markdown), `RenderedItem` (adds HTML and outline), and `FailedItem` (route + error). Each case is a plain record holding only the fields that make sense at its stage — no base class, no status enum, no nullable placeholders. The union itself exposes a single `Route` property that all four cases share, because "every content item, even a failed one, belongs to a route" is a genuine invariant. Call sites that need the rendered HTML must pattern-match and will get a compile error if they forget a case.

`Route` is the one projection lifted onto the union, and that narrowness is deliberate: every additional lifted property would need a sensible value for all four cases — which is exactly the nullable-field trap the union is meant to avoid. For full member lists, see Pennington.Content.IContentService.

`ContentSource` discriminates where an item came from

A `DiscoveredItem` pairs a `ContentRoute` with a second union, `ContentSource`, which records where the item came from. The reason this is a separate union rather than a field on `DiscoveredItem` is that discovery is itself a pluggable step — different sources carry different data (a file path, a Razor component type, a target URL) without forcing later stages to care. Once an item has been parsed, its source has already done its job: the parser and renderer work entirely against the resolved front matter and content text, and `ContentSource` disappears from the picture. For why this second union is shaped the way it is — its cases, and the `.Value` read that works across both target frameworks — see Why ContentSource is a union.

Stage transitions replace the item

Each stage in the pipeline works by replacing the incoming union case with the next one. `ParseAsync` pulls a stream of `ContentItem` values and, for each `DiscoveredItem`, hands its content to the registered `IContentParser`. When the parser succeeds, the `DiscoveredItem` is replaced by a `ParsedItem` carrying the resolved front matter and text. `RenderAsync` does the same thing one level further: each `ParsedItem` is handed to the `IContentRenderer`, and on success a `RenderedItem` takes its place, now carrying the HTML output and a navigation outline. The final stage, `GenerateAsync`, pattern-matches on the full union to write output files and accumulate the build report.

The replacement invariant is what gives the pipeline its composability. A `RenderedItem` flowing into `ParseAsync` is already past that stage, so `ParseAsync` passes it through unchanged. A `ParsedItem` flowing into `RenderAsync` gets rendered; a `RenderedItem` in the same stream passes through. This means you can hand the pipeline a partially-processed stream — one that mixes discovered and already-parsed items — and it will do the right thing for each. There is no need to coordinate which stage ran last; the case type carries that information.

CSHARP

```
await foreach (var item in items)
{
    // FailedItems pass through unchanged
    if (item.Value is FailedItem)
    {
        yield return item;
        continue;
    }

    if (item.Value is DiscoveredItem discovered)
    {
        // RedirectSource items are handled by PenningtonRedirectMiddleware at
        // request time (dev) and captured as 301 responses by the build crawler;
        // they don't participate in parse/render and must not reach the parser.
        // EndpointSource items (e.g., /sitemap.xml) are produced by a live HTTP
        // endpoint and GeneratedSource items by an artifact resolver — there's
        // no file to parse, same skip applies.
        if (discovered.Source.Value is RedirectSource or EndpointSource or GeneratedSource)
        {
            continue;
        }

        // No markdown parser registered (bare host): nothing can turn a DiscoveredItem
        // into a ParsedItem, so pass it through. Razor @page routes are served by Blazor
        // routing and custom sources resolve through their own endpoints.
        if (_parser is not { } parser)
        {
            yield return item;
            continue;
        }

        ContentItem result;
        try
        {
            result = await parser.ParseAsync(discovered);
        }
        catch (Exception ex)
        {
            result = new FailedItem(discovered.Route,
                new ContentError($"Parse failed: {ex.Message}", ex));
        }
        yield return result;
    }
    else
    {
        // Already parsed or rendered — pass through
        yield return item;
    }
}
```

The implementation has three explicit branches: a `FailedItem` passes through without touching the parser, a `DiscoveredItem` is handed to `IContentParser.ParseAsync` inside a try/catch that demotes any exception to a `FailedItem`, and a `ParsedItem` or `RenderedItem` passes through unchanged because the work for that stage is already done. There is also a guard for `RedirectSource` and `EndpointSource`: items whose source is a redirect or an endpoint (`sitemap.xml`, `llms.txt`) skip the parser entirely, because neither has a body to parse — a redirect is served by middleware at request time rather than written as an HTML file, and an endpoint is produced by a live HTTP endpoint.

`FailedItem` as a peer case, not an exception

Exceptions thrown inside `IContentParser.ParseAsync` or `IContentRenderer.RenderAsync` are caught at the pipeline boundary and rewritten as a `FailedItem` carrying the route and a `ContentError` that describes what went wrong. From that point on, the failed item rides the same async stream as the successful ones. Downstream stages check `is FailedItem` and short-circuit without touching the error or trying to make sense of absent fields.

This is what `GenerateAsync` relies on. Because `FailedItem` is a case in the union, the exhaustive pattern match there routes it to `BuildReportBuilder.AddError` — every parse or render exception ends up as a named entry in the build report rather than an unhandled exception that aborts the crawl. One broken markdown file does not prevent the other four hundred from rendering. Because failure is just another data case, a failed item and a successful one are the same kind of value in the stream, so the final aggregation step handles them the same way.

CSHARP

```
public record FailedItem(ContentRoute Route, ContentError Error);
```

Treating failure as a data case is what gives the exhaustive match in `GenerateAsync` something to act on. If `FailedItem` were an exception that escaped the pipeline, there would be no case to match — and no way for the compiler to confirm that every outcome was handled.

Further reading

- Reference: Content pipeline interfaces — the catalog of `IContentService`, `IContentParser`, `IContentRenderer`, and every case record with members.
- How-to: Implement a custom IContentService — how to plug a new source into the discovery stage.

Why ContentSource is a union

Under the Hood Why the case-discriminated union — FileSource, RazorPageSource, RedirectSource, EndpointSource, LlmsOnlySource — beats the polymorphic alternatives, and why every consumer goes through `.Value`.

`ContentSource` is the second of Pennington's two pipeline unions. Where `ContentItem` discriminates a page's stage in the pipeline, `ContentSource` discriminates *where the page came from*. The five cases — `FileSource`, `RazorPageSource`, `RedirectSource`, `EndpointSource`, `LlmsOnlySource` — capture every origin Pennington ships. `FileSource` carries a path *and* a format key (`"markdown"`, `"cook"`, …), so one case covers every file-backed format — the key selects the parser and renderer, which is how a custom format like Cooklang flows through the same pipeline as markdown (see Add a custom content format). The content pipeline and union types covers why the pipeline as a whole is shaped as a union; this page is about why this *particular* shape, and why the polyfill choice matters more than it looks.

Why a union and not polymorphism

A first instinct is to make `ContentSource` an interface with five implementations and dispatch through virtual methods. That solves the dispatch problem but introduces three problems the union avoids:

- **Exhaustiveness disappears.** With an interface, adding a sixth implementation later compiles silently — every existing switch keeps its `default` clause and quietly stops covering the new case. With the union, the compiler complains at every switch that no longer covers every case. A new case shows you exactly which switches need updating.
- **The "no canonical body" cases get awkward.** `RedirectSource` carries a `TargetUrl` and nothing else; `EndpointSource` carries no payload at all; `LlmsOnlySource` carries a path that explicitly never produces HTML. Modeling them as interface implementations forces them to satisfy the same shape as `FileSource`, which carries a path to read and a format key naming its parser and renderer. The union lets each case carry exactly its own data.
- **Pattern matching reads directly.** Consumers want to *branch on the case*, not call a virtual method that wraps the branch. With a union, `source.Value switch { FileSource f => …, RedirectSource r => … }` is exactly the read; with polymorphism, you'd either bake the consumer logic into each implementation (poor separation) or end up with a visitor.

Why `.Value` and not the case type directly

Pennington multi-targets `net10.0;net11.0`. On `net11.0+` the C# 15 `union` keyword synthesizes a discriminated union with an inner `object? Value` field; on `net10.0` a hand-written polyfill struct provides the same shape. Going through `.Value` is the one read that compiles unchanged on both TFMs and matches what every consumer in the codebase already does.

Why `RedirectSource` and `LlmsOnlySource` exclude themselves from `sitemap.xml`

Both name a route with no canonical HTML page to advertise. `RedirectSource` has no body at all — the response is a 30x to another URL. `LlmsOnlySource` has no HTML page anywhere — it only contributes to the llms.txt index and its sidecar markdown. Neither belongs in a crawler's list of indexable pages, so the sitemap filters them out in one expression:

CSHARP

```
if (discovered.Source.Value is RedirectSource or LlmsOnlySource) continue;
```

`EndpointSource` is the case that *looks* like it should join them but doesn't. It defers rendering to a sibling `MapGet`, but that endpoint returns real, canonical HTML at a stable URL — a custom content service's pages, an `AddTaxonomy` term page. Those are exactly what a sitemap is for, so they stay in, and the on-site search index and the sitemap stay consistent about which pages exist. Transport endpoints that happen to use `EndpointSource` — a JSON data route, a generated feed — emit a non-`.html` output file and are dropped earlier by `SitemapService`'s output-extension check, so the source filter only has to name the two cases that never produce an HTML page.

See also

- Background: The content pipeline and union types
- How-to: Source content from outside the file system — the recipe for constructing each case and pattern-matching it.
- How-to: Sitemap configuration

Dev mode and build mode share one code path

Under the Hood Why the static build is a crawler against the same ASP.NET pipeline as dev, not a second renderer — keeping dev fidelity and publish output in lockstep.

Why doesn't Pennington have a separate offline build step — one that reads markdown and writes HTML without starting a web server — when `dotnet run -- build` boots the entire ASP.NET host first?

Context

Most static site generators are built as compilers: read content files, transform them, write HTML. That shape is intuitive, and it was on the table for Pennington too. The problem with a separate publish renderer surfaces not at the first feature but at the second. Locale middleware runs in dev, so it needs a second implementation in the offline path; response processors run in dev, so they need it too; Blazor SSR for islands, the xref rewriter, the CSS discovery pipeline — each one accrues a corresponding "also do this in build" edit. The two implementations then diverge over time, invisibly, until a feature that works in development produces different output in publish.

Pennington keeps one host. Dev mode is that host serving requests over Kestrel; build mode is a crawler that drives the same host's request pipeline in process. There is exactly one ASP.NET pipeline, and the static build is a consumer of it. The rest of this page works through what that buys.

How it works

Dev serve: the ASP.NET host is the renderer

Running `dotnet run` causes `RunOrBuildAsync` to detect the absence of a `build` argument and call `app.RunAsync()`. Every request that lands at `localhost:5000` flows through the full middleware stack: locale routing, live reload, `ResponseProcessingMiddleware` capturing and rewriting the body, Blazor SSR for any island components, and the Markdig extensions inside `MarkdownContentRenderer`. The rendered HTML that arrives in the browser is the pipeline output, unchanged.

Nothing in this path is marked "dev-only." The diagnostic overlay and live-reload script injection are response processors ordered behind environment gates — not separate code paths. The renderer behind `localhost:5000` is the same renderer the build uses.

Build mode: a crawler driving the same pipeline

When the first argument is `build`, Pennington replaces Kestrel with an in-memory test host at service-registration time, then starts that host without a socket bind, a dev-cert prompt, or a port. The crawler dispatches requests straight into the same `RequestDelegate` Kestrel would have invoked in dev — there is no network round-trip, but every request runs the full pipeline.

URL discovery comes from two sources. Every registered `IContentService` exposes `DiscoverAsync`, which returns the set of content routes it knows about. The live `EndpointDataSource` covers `MapGet` handlers — `/styles.css`, `/sitemap.xml`, the per-locale `/search/{locale}/...` artifacts, and anything else the host has wired up explicitly. Each response is written to `OutputOptions.OutputDirectory` using the route's `OutputFile` mapping.

The 404 page is a small special case: the crawler fetches a URL that no route matches, so the catch-all fallback fires and its output is written as `404.html`. The mechanism remains a GET against the same pipeline.

The shared pipeline

Because the build drives requests through the same pipeline, every cross-cutting system runs identically in both modes. `ResponseProcessingMiddleware` captures and rewrites bodies. `IHtmlResponseRewriter` resolves xref links and applies locale prefixes and the base URL. The MonorailCSS discovery pipeline scans loaded assemblies and watched source files at startup, so the class registry is already populated before the crawler starts; content-page GETs run first and `MapGet` GETs last as a separate ordering rule, ensuring `/styles.css` and other generated endpoints see a fully-warm system.

The consequence is that dev and build cannot drift apart. The pipeline that produced `localhost:5000/foo` is the pipeline that produced `output/foo/index.html`. A feature that works in dev works in build, and one that breaks in build would have broken in dev first.

Why not a separate renderer?

The alternative — a pure in-process renderer that drives Markdig directly, writes files, skips the request pipeline entirely — is faster for small sites and simpler to maintain if the feature set never grows. The tradeoff is that every capability built on top of ASP.NET would have to be reimplemented for the offline path. Locale middleware, response processors, Blazor SSR for islands, the per-locale search artifacts, the diagnostic-header transport — each would require a second implementation. Each new feature becomes two edits and two chances for the implementations to diverge.

Build mode does add a fixed cost per page — routing and the middleware stack run on every URL rather than being bypassed — but with no socket round-trip that cost is a thin slice of each page's render time and stays flat as the site grows. The in-memory `BuildHtmlCache` further collapses the disk-write, search-index, and llms.txt passes to one render per URL. The cost of maintaining a second renderer, by contrast, grows with every feature added. Pennington accepts the per-request overhead to avoid it.

Further reading

- Reference: Build report fields
- Reference: CLI and build arguments
- How-to: Build a static site
- How-to: Host under a sub-path (base URL)

The front-matter capability system

Under the Hood How IFrontMatter distinguishes universal capabilities (as default members) from selective ones (as separate interfaces) — and why presence of a capability interface is a meaningful signal.

Where does the line fall between "every page needs this" and "only some pages need this" when those two categories share a base interface?

Context

A content engine asks a lot of questions about each page. Is it a draft? Does it belong in search? In the LLM index? Does it carry a cross-reference uid, a description, or a date? These questions apply to every page, even when the answer is trivially "no." A smaller set of questions — does it have tags? does it participate in ordered navigation? does it carry a section label? is it a redirect? — applies only to some content types.

Pennington models this split directly on the type system. The universal questions live on `IFrontMatter` itself, with sensible defaults, so every record answers them without having to opt in. The selective questions live on separate capability interfaces, so a content type that does not implement `ITaggable` is not tagged — and the engine can tell at compile time.

How it works

`IFrontMatter` : universal capabilities with defaults

`IFrontMatter` has one abstract member (`Title`) and seven default-implemented ones. Every front-matter record inherits `IsDraft => false` , `Search => true` , `Llms => true` , `SearchOnly => false` , `Uid => null` , `Description => null` , and `Date => null` without declaring them.

CSHARP

```
public interface IFrontMatter
{
    /// <summary>Page title rendered in the browser tab, navigation, and OpenGraph tags.
</summary>
    string Title { get; }

    /// <summary>True when the page is a draft and should be excluded from builds.</summary>
    bool IsDraft => false;

    /// <summary>True when the page should be included in the search index.</summary>
    bool Search => true;

    /// <summary>True when the page should be included in llms.txt output.</summary>
    bool Llms => true;

    /// <summary>
    /// When true, the page is included in indexing channels (search, llms.txt) but excluded
    /// from the rendered navigation tree. Useful for FAQ entries, glossary terms, or other
    /// content that should be discoverable by search but should not clutter the sidebar.
    /// </summary>
    bool SearchOnly => false;

    /// <summary>Stable cross-reference identifier used by xref links.</summary>
    string? Uid => null;

    /// <summary>Short summary used in meta descriptions, OpenGraph tags, and listings.
</summary>
    string? Description => null;

    /// <summary>
    /// Publication date surfaced in feeds and sitemaps. Also drives scheduled publishing:
    /// when this is set to a moment after the build clock, the page is excluded from build
    /// output (same dev-vs-build behavior as <see cref="IsDraft"/>) until the clock catches
up.
    /// </summary>
    DateTime? Date => null;
}
```

The contract gives every record common defaults it can override. A minimal record exposes a single required `Title` property and the engine handles drafts, search indexing, LLM indexing, cross-references, descriptions, and dates gracefully. Engine code uses the members directly — `if (page.IsDraft)` works on every `IFrontMatter` without checking for each interface first.

The capability interfaces

Tags, order, section labels, redirects, and Standard Site document keys live on separate interfaces because the interface's *presence* is itself a signal. Seeing `IOrderable` on a record says the content type consciously participates in ordered navigation; folding it into `IFrontMatter` would erase that distinction, since every record would then carry the member whether it meant anything or not. The selectivity is real, too: a blog post has tags but no meaningful order among siblings; a doc page has an order but no redirect target; a redirect stub carries a destination URL and little else. Folding these into `IFrontMatter` would force every record to carry empty tag arrays and meaningless sort keys.

CSHARP

```csharp
public interface IOrderable
{
    /// <summary>Sort order for this page within its section (lower sorts first).</summary>
    int Order { get; }
}
```

`NavigationBuilder` keys off the `IOrderable` interface itself, not a sentinel value in the `Order` property. A content type either implements the interface and participates in ordered navigation, or it does not; there is no "this page has no meaningful order" case to handle. The same applies to `ITaggable` (tag cloud participation), `ISectionable` (section-label breadcrumbs), `IRedirectable` (redirect-stub semantics), and `IStandardSiteDocument` (the AT Protocol record key for Standard Site syndication).

The rule of thumb is simple: if adoption is universal, the member lives on `IFrontMatter` with a sensible default. If adoption is selective, it lives on a capability interface so that pattern-matching on the interface remains meaningful.

Custom front-matter records

A custom record buys typed access to extra keys (an `apiVersion` or `gitHubUrl` field becomes a strongly-typed property) plus the same set of capability interfaces to opt into. The defaults give what the shipped records would give; the custom record only declares what it adds. See Define custom front-matter keys for the recipe.

Further reading

- Reference: IFrontMatter and capability defaults
- Reference: Front matter key reference
- How-to: Work with front matter

The response-processing pipeline

Under the Hood Why Pennington splits response rewriting into generic body processors and HTML-DOM rewriters that share one AngleSharp pass.

Why are there two extension points for rewriting the response body — a general `IResponseProcessor` and an HTML-specific `IHtmlResponseRewriter` — instead of a single chain of string-to-string processors?

Context

Two kinds of work want to touch the response body. Some concerns care about HTML structure — rewriting `href` attributes, inserting elements at specific selectors, normalizing document shape. Others treat the body as an opaque string — appending a script before `</body>`, prepending a cache buster, injecting a dev overlay into HTML responses only.

A single chain of string-to-string processors forces the structure-aware concerns to either reparse the body themselves or rewrite HTML with regex, neither of which composes. A single HTML-shaped rewriter forces the opaque-string concerns to take an AngleSharp dependency they do not need. Pennington splits the work along that line: a generic string-in/string-out contract for body-level concerns that have no need for a DOM, and one shared parse that all DOM concerns participate in together.

How it works

Pennington uses two extension points. The first is the generic body pipeline (`IResponseProcessor`); every body-touching concern, HTML or not, registers here. The second is one specific `IResponseProcessor` — `HtmlResponseRewritingProcessor` — that hosts a shared AngleSharp pass for HTML-DOM concerns (`IHtmlResponseRewriter`).

Several body processors ship built in: the rewriter host above, a CSS URL rewriter, the dev-only live-reload and diagnostic-overlay injectors, and `NotFoundStatusProcessor`. They are all body-level concerns that share the same generic contract; the response-processing interfaces reference catalogs each one.

`NotFoundStatusProcessor` is the one that needs a word, because it is doing something the others are not. A content page that resolves to a missing route does not set `StatusCode = 404` itself — it sets a marker on `HttpContext.Items` and renders the 404 body normally. Every other processor in the chain gates on a 2xx status, so flipping the status early would short-circuit them. `NotFoundStatusProcessor` runs last and flips the status to 404 only after the body is fully composed, which keeps the rendered 404 page — localized chrome, layout, structured data — intact while still surfacing a real 404 to crawlers and link checkers.

Tier A: `IResponseProcessor` (generic body capture)

`ResponseProcessingMiddleware` wraps the response body stream, captures it into memory, runs every registered `IResponseProcessor` whose `ShouldProcess` returns `true` in ascending `Order`, then flushes the final string back to the socket. The contract is intentionally narrow — an ordering integer, a predicate over `HttpContext`, and an async body transform — so anything that wants to touch the response body can participate without taking an AngleSharp dependency.

`LiveReloadScriptProcessor` and `DiagnosticOverlayProcessor` illustrate why the tier exists. Both are pure string operations — a targeted insert and a before-`</body>` append. Routing them through AngleSharp would parse and serialize the document for no benefit. The tier boundary exists because "touches the body" is a broader category than "cares about HTML structure."

Tier B: `IHtmlResponseRewriter` (shared AngleSharp pass)

The HTML rewriters live entirely inside `HtmlResponseRewritingProcessor`. The orchestrator calls each rewriter's `ShouldApply(HttpContext)` first. If none return `true`, the body comes back untouched and AngleSharp never fires — non-HTML content types, error pages, and opted-out endpoints skip the parse entirely. If at least one applies, the orchestrator runs all of them through a two-phase pipeline.

The first phase, `PreParseAsync`, operates on the raw HTML string. This handles constructs AngleSharp cannot represent cleanly — the `<xref:uid>` tag syntax is not valid HTML, so it needs to be rewritten into something the parser can consume before the document is built. The second phase, `ApplyAsync`, receives a single shared `IDocument` that every rewriter mutates in turn. The document is serialized once at the end.

The result is the invariant that matters here: N rewriters, one parse, one serialize, one DOM. Adding a new DOM-shaped concern — a heading-anchor normalizer, an image lazy-loader, a table classifier — costs a method call, not another parse/serialize round trip. See Pennington.Infrastructure.IResponseProcessor for the `IResponseProcessor` and `IHtmlResponseRewriter` contracts.

Why two tiers, not one

Collapsing everything into `IHtmlResponseRewriter` would be wrong in both directions. Making the string processors HTML rewriters forces an AngleSharp parse on operations that do not benefit from a DOM, and it also means the parser tries to fix partially-valid or framework-generated HTML that those processors are content to treat as an opaque string. Conversely, letting every DOM concern be its own `IResponseProcessor` means each one parses, mutates, and serializes independently — N parses and N DOM copies instead of one of each — and it hides the cross-concern ordering assumptions behind DI registration sequence.

The split follows one question: does this concern care about HTML structure? Body-level injection and scraping live on one side; link rewriting and attribute manipulation live on the other. Each contract stays narrow to its side of that line.

Why the order matters

This is the page that owns the rewriter ordering; the other explanation pages that depend on one slot of it link here rather than restate the whole chain. Six rewriters ship, and they run lowest `Order` first:

```
`Order` Rewriter What it does     10 `XrefHtmlRewriter` Resolves `<xref:uid>` tags and
`href="xref:uid"` into canonical paths   20 `LocaleLinkHtmlRewriter` Prefixes internal links
with the active locale segment   25 `HeadCompositionHtmlRewriter` Composes the
`IHeadContributor` output into the head   30 `BaseUrlHtmlRewriter` Prepends the deployment
base URL and stamps `data-base-url`   40 `FallbackLangHtmlRewriter` Marks links served from
default-locale fallback   60 `WordBreakHtmlRewriter` Inserts soft word breaks into long
identifiers
```

The first four form a dependency chain because each produces the link shape the next consumes. `XrefHtmlRewriter` runs first so everything downstream sees real URLs rather than symbolic cross-reference handles. `LocaleLinkHtmlRewriter` runs after it because an unresolved `xref:uid` is not yet a path, and before base-URL rewriting so the locale segment ends up inside the base URL rather than outside it. `HeadCompositionHtmlRewriter` slots in between locale and base-URL rewriting so that any root-relative `href` a head contributor emits gets sub-path prefixed exactly as literal head markup would — the head subsystem explains this dependency from its own side. `BaseUrlHtmlRewriter` runs last of the four as the outermost transport layer: everyone upstream works with clean `/`-rooted paths and never has to strip a base URL before operating.

`FallbackLangHtmlRewriter` and `WordBreakHtmlRewriter` slot in afterward without joining that chain — they read the finished URLs but no later rewriter depends on their output. Reversing any two of the first four breaks one of the others' invariants. Keeping the ordering explicit in the `Order` property — rather than implicit in DI registration sequence — is what makes the dependency between rewriters visible at the call site.

Further reading

- Reference: Response processing interfaces — the member-by-member catalog of `IResponseProcessor`, `IHtmlResponseRewriter`, and the three built-in rewriters.
- How-to: Write a response processor — for touching the raw body.
- How-to: Write an HTML rewriter — for working inside the shared DOM pass.
- Related explanation: Dev mode and build mode share one code path — why the same processor chain runs against both live requests and the static-build crawler.

The head subsystem

Under the Hood Why everything that writes to the document head — title, canonical, JSON-LD, OpenGraph, alternates, Standard Site links — funnels through one typed model, one rewriter that finalizes it, and a shared `data-head` attribute.

A surprising number of concerns want to write into the document `<head>`: the title and description, the canonical link, schema.org JSON-LD, OpenGraph and Twitter card meta, RSS and llms.txt alternates, `hreflang` locale alternates, the dev-host meta that live reload reads, and the Standard Site verification links. Why does Pennington route all of them through a single `IHeadContributor` extension point instead of letting each one emit its own markup where it already lives?

Context

Before the head subsystem, those writers were spread across four unrelated mechanisms. Some were literal markup in each template's `App.razor` head. Two were head-scoped `IHtmlResponseRewriter`s — one for the canonical link, one for JSON-LD. Per-page meta was authored in Razor `<HeadContent>` blocks. The dev-host meta was a raw string insert that searched the response for `</head>`.

Four mechanisms meant four ways to reason about ordering, four places a duplicate `og:image` or a second `<title>` could slip through, and no shared notion of "this tag is already present, leave it alone." The ordering was the worst of it: the rewriters carried hand-picked integers that unrelated writers silently collided on, and nothing connected a rewriter's order to the literal markup it had to interleave with.

The worst of it showed up on the client. Pennington's SPA navigation swaps page regions rather than reloading, so the head has to be reconciled in JavaScript on every soft navigation. The old client carried a hand-maintained allowlist naming exactly which head tags to carry across a swap. Every new head tag had to be added to that list by hand, and forgetting meant the tag silently vanished the moment a visitor clicked a link — a failure invisible on first paint and easy to ship.

How it works

The subsystem has four parts: a typed data model, the `IHeadContributor` extension point, a single rewriter that finalizes the head, and a `data-head` attribute that ties the server and client halves together.

The typed model

Everything that can land in the head is a typed `HeadTag` — a union of `TitleTag`, `MetaNameTag`, `MetaPropertyTag`, `LinkTag`, `ScriptTag`, and a `RawTag` case for markup the engine does not model. Each tag that should appear at most once carries a stable `HeadTagKey`: `title`, `meta:prop:og:image`, `link:rel:canonical`, and so on. Repeatable tags — `hreflang` alternates, JSON-LD blocks, preloads, the Standard Site links — carry no key and always append.

The key is what makes dedup work. A `HeadBuilder` keeps a keyed map where the first add at a key wins and later same-key adds are dropped, alongside a keyless list for repeatables.

CSHARP

```
public sealed class HeadBuilder
{
    private readonly Dictionary<string, HeadTag> _keyed = new(StringComparer.Ordinal);
    private readonly List<string> _keyOrder = [];
    private readonly List<HeadTag> _keyless = [];

    /// <summary>Adds a tag under an explicit dedup key; the first add at a key wins.
</summary>
    public HeadBuilder Add(HeadTagKey key, HeadTag tag)
    { … }

    /// <summary>Adds a repeatable tag (hreflang, JSON-LD, preload) with no deduplication.
</summary>
    public HeadBuilder AddRepeatable(HeadTag tag)
    { … }

    /// <summary>Sets the document title (deduplicated to one).</summary>
    public HeadBuilder Title(string text) …;

    /// <summary>Sets a named meta tag, deduplicated on its <paramref name="name"/>.
</summary>
    public HeadBuilder Meta(string name, string content)
        …;

    /// <summary>Sets an OpenGraph/property meta tag, deduplicated on its <paramref
name="property"/>.</summary>
    public HeadBuilder Property(string property, string content)
        …;

    /// <summary>Sets a singleton link (e.g. canonical), deduplicated on its <paramref
name="rel"/>.</summary>
    public HeadBuilder Link(string rel, string href)
        …;

    /// <summary>The composed entries: keyed tags first (first-seen order), then keyless
tags (append order).</summary>
    public IReadOnlyList<HeadEntry> Build()
    { … }
}
```

The contributor extension point

A contributor is an ordered, gated unit — the same form as the `IHtmlResponseRewriter` it often replaces.

CSHARP

```
public interface IHeadContributor
{
    /// <summary>
    /// Ascending priority (use the <see cref="HeadOrder"/> bands). Contributors run lowest-
first,
    /// and on a <see cref="HeadTagKey"/> collision the lowest order wins.
    /// </summary>
    int Order { get; }

    /// <summary>Cheap gate. Return <c>false</c> to skip <see cref="ContributeAsync"/>
entirely.</summary>
    bool ShouldContribute(HeadContext context);

    /// <summary>Pushes tags into the builder for this request.</summary>
    Task ContributeAsync(HeadContext context, HeadBuilder head);
}
```

`Order` does double duty. Contributors run lowest-first, and on a key collision the lowest order wins — so a page-level tag beats a site-level default for the same key without either side knowing about the other. To keep those numbers from drifting back into ad-hoc collisions, `Order` is chosen from named bands rather than raw integers:

CSHARP

```
/// <summary>Page-authored or page-computed tags: title, description, per-page OpenGraph.
Wins ties against site defaults.</summary>
public const int Page = 40;

/// <summary>Site-wide defaults: canonical, <c>og:site_name</c>, RSS/llms alternates,
hreflang.</summary>
public const int Site = 60;

/// <summary>Discovery payloads: JSON-LD structured data and Standard Site verification
links.</summary>
public const int Discovery = 80;
```

A page-OpenGraph contributor at `Page` (40) and a site-default contributor at `Site` (60) can both try to set `og:image`; the page wins because it ran first, and the site default steps aside through the same dedup that would have collapsed two identical tags. The bands encode the precedence relationship that the old hand-picked integers only implied.

The composition rewriter

A single rewriter, `HeadCompositionHtmlRewriter`, is the only place head tags are finalized. It runs inside the shared AngleSharp pass described in the response-processing explanation — so composing the entire head costs no extra parse or serialize, just another mutation of the already-parsed document.

Its order matters here for a subtle reason. It sits between locale rewriting and base-URL prefixing in the shared rewriter chain, so any root-relative `href` a contributor emits — an asset, an alternate link — gets sub-path prefixed by the base-URL rewriter exactly as literal head markup would. A contributor never has

to know the deployment base URL; it emits `/rss.xml` and the downstream rewriter handles the rest. That slot is the rewriter's own `Order`:

CSHARP

```
public int Order => 25;
```

The rewriter does two things in sequence. First it composes: it runs every registered contributor whose `ShouldContribute` returns true, lowest order first, into one `HeadBuilder`, then reconciles the built tags into the document head. Tags whose keys a page already authored are left untouched — contributors fill gaps, they do not overwrite page intent. Second, it normalizes what the page authored itself.

The data-head attribute

Every finalized head element — whether a contributor emitted it or a page authored it — carries a `data-head` attribute. That one attribute drives both halves of the system. Server-side, it marks which elements the reconciler owns. Client-side, it collapses the old allowlist into one generic sweep: on a soft navigation, remove every `[data-head]` element and clone every `[data-head]` element from the fetched document. Every future head tag survives navigation automatically, because surviving is now a property of the attribute, not of a list someone has to remember to update.

This is also why page authoring keeps working. A page that writes `<PageTitle>` or a `<HeadContent>` block still renders through Blazor's `HeadOutlet`; the rewriter's normalization step pulls those rendered tags into the same model, stamps them, and dedups them against contributor output — with page authorship winning on a key collision. Markup the engine does not recognize passes through as a `RawTag`, untouched. So `<HeadContent>` does not compete with the contributors; it is one more input feeding the same model.

The deliberate exception

Two things stay as literal markup on purpose: the theme-bootstrap inline `<script>` and the stylesheet link. Both are about avoiding a flash. The theme script must run before first paint to apply dark mode, so it cannot wait for a rewriter that runs after the document is built; the stylesheet stays put because the SPA's stylesheet sync deliberately never removes an existing sheet (removing and re-adding it flashes the unstyled page). The subsystem owns meta and discovery tags — the things whose ordering and dedup were the actual problem — and leaves the two pre-paint assets where they have to be.

Why one subsystem rather than leaving writers where they lived

The alternative — every concern emits its own head markup at the site it already occupies — is what the codebase had, and it failed in three specific ways the consolidation fixes. Dedup had no home, so two writers targeting `og:image` produced two tags. Ordering lived in scattered integers and template line numbers with no shared scale, so precedence between a page tag and a site default was implicit and fragile. And client reconciliation depended on a hand-maintained list, so the cost of adding a head tag included a second, easy-to-forget edit in JavaScript.

The typed model gives dedup a key to work on, the single rewriter gives ordering one scale and one finalization point, and the `data-head` attribute gives the client a rule instead of a list. The price is indirection: a tag that used to be three lines of Razor is now a small class registered in DI. For a one-off tag on a single page that price is real, which is why `<HeadContent>` still works and is still the right tool for genuinely page-local markup. For anything cross-cutting — anything that needs to dedup, order against other writers, or survive navigation — a contributor is worth the indirection.

Further reading

- How-to: Add tags to the document head — write and register an `IHeadContributor`.
- Related explanation: The response-processing pipeline — the shared AngleSharp pass the rewriter runs inside.
- Related explanation: SPA navigation through region swaps — why the head needs client-side reconciliation at all.

CHAPTER 2

Rendering

MonorailCSS integration

Under the Hood Why Pennington discovers CSS classes by scanning compiled assemblies and watched source files instead of pre-building a static stylesheet.

Utility-first CSS normally needs a build step that scans source files and regenerates a stylesheet — so how does Pennington emit a correct stylesheet when there is no `npm run build` in the loop and new classes can appear the instant someone edits a markdown file?

Context

Utility-first CSS frameworks like Tailwind and MonorailCSS (the Tailwind-compatible .NET JIT compiler Pennington integrates) ship a vast class surface and rely on a scanner to collect only the classes in use, keeping the final stylesheet small. Traditional setups solve this with a pre-build step that globs source files. That model fights a runtime-rendering content engine in two ways: markdown is rendered at runtime through Markdig extensions, so classes do not exist on disk until a request renders them, and adding a Razor component or a new page would require rerunning a separate tool.

Pennington uses the `MonorailCss.Discovery` package. Discovery force-loads every non-BCL referenced assembly at startup, walks the IL for string literals that parse as utility candidates, and watches source files in development for live updates. The discovered set is exposed through an `IClassRegistry`. The `/styles.css` endpoint reads the current class set and runs it through a fresh `CssFramework` on every request, so an option change or a newly observed class shows up on the next fetch without a process restart.

How it works

Classes are discovered by scanning compiled output

`AddMonorailCss` calls `services.AddMonorailClassDiscovery()`, which registers the runtime scanner. At startup the scanner enumerates every assembly the entry app references (skipping the BCL), force-loads each one if needed, and walks IL string literals through Pennington's configured `CssFramework` to keep only the candidates the framework actually recognizes. The same theme drives both halves of the pipeline: the framework that validates discovery candidates and the one that generates the stylesheet are built from the same options, so a class survives discovery only if it would render.

Because the scan reads compiled IL rather than source text, every `class="bg-primary-500"` literal in a Razor component, every string constant in a C# helper, and every utility token in `Pennington.UI`'s shipped components participates without any per-project glob configuration. In development, Discovery

also watches the source files behind the loaded assemblies and re-scans on edits, so a new utility added to a `.razor` or `.cs` file shows up on the next `/styles.css` fetch. If a `wwwroot/app.css` is present, Discovery treats it as the source CSS prefix.

The stylesheet generates on demand, every request

`UseMonorailCss` maps a `GET /styles.css` endpoint that calls `MonorailCssService.GetStyleSheet()`. Each hit builds a fresh `CssFramework` from the current `MonorailCssOptions`, runs it over `IClassRegistry.GetClasses()`, and prepends Pennington's content-visibility preamble plus any configured `ExtraStyles`. The per-call rebuild is what lets hot-reload edits to `CustomCssFrameworkSettings` or theme tokens flow into the next stylesheet without restarting the process.

Pennington is a static content engine: the build is one-shot and the dev server is the only other consumer, so per-call regeneration is cheap enough to make caching unnecessary. The first page load primes the registry with whatever classes that page emits; the browser then fetches `/styles.css` and gets a stylesheet generated from the current class set. A subsequent navigation that introduces a new class is reflected on the very next stylesheet fetch.

A static build leans on the same ordering. The build fetches every HTML page first — priming the registry with every class the rendered site actually emits — and fetches `/styles.css` last, after all that markup has run through the pipeline. So the `styles.css` written to the output directory is a single tree-shaken file: exactly the utilities the site uses, nothing more, generated once and served as a plain static asset with no runtime regeneration in production.

Color schemes: named vs algorithmic

`ColorScheme` on `MonorailCssOptions` ships in two flavors. `NamedColorScheme` is the choice when a designer says "I want Tailwind Purple for primary": it maps `primary`, `accent`, and `base` onto built-in palettes by name. `AlgorithmicColorScheme` is the choice when the starting point is a brand hue expressed in degrees and the whole palette needs to be derived from it: it synthesizes everything from a single `PrimaryHue`. See Pennington.MonorailCss.MonorailCssOptions for the full parameter surface.

Syntax-highlight colors are deliberately kept off the brand scheme. `SyntaxTheme` on `MonorailCssOptions` holds the five roles `.hljs-*` token classes consume — keyword, string, variable, function, and comment — each mapped to its own Tailwind palette. The default picks a tuned combination (Sky / Emerald / Rose / Amber / Slate) that reads well against either a light or dark code background, so a site can pick primary and accent purely for brand reasons without constraining how code renders.

OKLCH palette generation

From a single `PrimaryHue`, the algorithmic scheme synthesizes a full palette — each color as the familiar 11-stop ramp keyed by `50` through `950` shade names, derived in the OKLCH color space rather than HSL.

OKLCH is the right choice here because of perceptual uniformity. It is a cylindrical coordinate system over the OK-Lab color space — lightness, chroma, and hue tuned so equal numeric steps look equal to the eye. That is not true of HSL, where a 500-weight green at HSL lightness 40% looks brighter than a 500-weight blue at the same value. OKLCH makes the generated scheme feel visually coherent without per-hue handwork, which is what makes "give me a palette from hue 214" a reasonable thing to ask.

Further reading

- Reference: MonorailCssOptions — the full option surface with defaults.
- How-to: Customize MonorailCSS — swapping schemes, injecting `CustomCssFrameworkSettings`, and authoring extra styles.

The syntax-highlighting cascade

Under the Hood Why Pennington dispatches code fences through a priority-ordered chain of highlighters with a guaranteed plain-text fallback instead of a single parser.

A content engine that renders Markdown through Markdig could reasonably pick one syntax highlighter and ship it — so why does Pennington dispatch every fenced code block through a priority-ordered chain of highlighters that falls through to a plain-text fallback, instead of binding the pipeline to a single parser?

Context

Pennington renders code in very different shapes: shell sessions that want command-and-flag styling but no formal grammar, and roughly eighty mainstream languages that need real tokenization. A single-parser design forces one of those shapes to lose. A shell-only build gives up nearly the full language surface the first time someone pastes a Python snippet; a TextMate-only build styles a bash command no differently from its flags. So the design layers highlighters instead of picking one: every shape is a highlighter, the ones that care most about a given language win, and a plain-text fallback catches anything no highlighter claims.

How it works

The priority chain

The cascade is specificity-ordered, not quality-ordered. Shell wins over TextMate for bash not because it produces better HTML in the abstract, but because it knows the one thing worth styling in a command fence — the command itself versus its flags. Priority encodes "which highlighter cares most about this language," not "which highlighter is best."

`HighlightingService` takes every registered `ICodeHighlighter` at construction, sorts once by descending `Priority`, and for each code block walks that list checking whether the language is supported (or the highlighter declared `"*"` as a catch-all). The first hit wins. The shipped chain is a 50/75 ladder: `TextMateHighlighter` at 50 with `"*"` so it claims any language it can find a grammar for, and `ShellHighlighter` at 75 for `bash` / `shell` / `sh` specifically. Those two integers are illustrative of the

shipped chain, not a stable contract — a custom highlighter should pick a priority *relative* to whatever currently handles its languages (above 50 to beat TextMate's catch-all, above 75 to displace shell), not hard-code `76`. The stable guarantee is the ordering rule, not the literal numbers.

When no chain entry matches and no `"*"` catch-all is registered, the service reaches for a hardcoded `PlainTextHighlighter` fallback, which HTML-encodes the code, hands it back, and emits a once-per-language `Info` diagnostic so authors notice a missing grammar without the build failing. The fallback is what keeps `HighlightingService` total: every input gets some output. It never appears in the priority chain, so it cannot be displaced by a misconfigured priority.

The `HighlightingService` dispatcher is stateless past construction, so adding a highlighter via `HighlightingOptions.AddHighlighter` in DI is enough — no registry mutation, no re-sorting at runtime, no ordering surprise that depends on registration order. Priority is the only tiebreaker that matters.

The `ICodeHighlighter` contract is three members — `SupportedLanguages`, `Priority`, and `Highlight(code, language)`. That is the narrowest shape that still lets the dispatcher choose a highlighter without having to run one first to find out whether it can handle the language. See Pennington.Highlighting.ICodeHighlighter for the interface surface.

Why TextMateSharp

Most of the chain — every language except the shell family (`bash`, `shell`, `sh`) — runs through `TextMateHighlighter`, which loads TextMate grammars through TextMateSharp and tokenizes line by line. TextMate grammars are the same regex-state-machine format VS Code uses for its default highlighting, which gives Pennington roughly eighty mainstream languages in a single dependency, without compiling a parser, without building an AST, and without pulling a language service per language. The highlighter keeps a scope-to-hljs-class mapping table so the emitted HTML uses the familiar `hljs-keyword` / `hljs-string` / `hljs-type` class names, meaning the same CSS theme highlights Python, Rust, Go, and JSON uniformly.

The alternatives that were considered and rejected make the choice clearer. A single-language semantic parser covers a language or two out of eighty and ships a heavy dependency for zero value on the rest. A Prism or highlight.js port would require either a JavaScript runtime at build time or a reimplementation of dozens of grammars in C#; TextMateSharp inherits VS Code's grammar corpus directly. A hand-rolled regex-per-language table scales linearly with language count and loses the "paste a new fence, it works" property the first time someone wants Kotlin. TextMate's cost is real — it is a regex state machine, so it does not know that `Foo` on line 40 refers to the `class Foo` on line 2 — but that ceiling is exactly what the cascade lets a more capable highlighter rise above for the one language that needs it.

The `"*"` entry in `TextMateHighlighter.SupportedLanguages` matters because it is how TextMate claims every language it can find a grammar for without enumerating the list at registration time, and it is what lets a new grammar added to the registry start working without any further configuration.

Slotting in a higher-priority highlighter

The cascade is the extension mechanism. A highlighter that wants to claim a language TextMate already handles — say a semantic C# highlighter that can tell a type name apart from a method name, resolve generic arguments, and annotate references to types in other files — registers at a priority above 50 for `csharp` / `cs` / `c#`. When it is present, the only change to the cascade is that C# rises from "TextMate handles it" to "the new highlighter handles it"; every other language keeps its previous highlighter.

Nothing in the core has to change to allow that. The base package ships the shell and TextMate tokenizers plus a plain-text fallback that together cover every site that does not need language-specific semantic treatment, and a higher-priority highlighter is purely additive — declare the relevant languages, pick a priority that beats whatever is currently handling them, register. The cascade does not know or care where a highlighter came from.

Further reading

- Reference: Highlighting interfaces — `ICodeHighlighter`, `HighlightingService`, `TextMateLanguageRegistry`, and `ICodeBlockPreprocessor` with full member tables.
- How-to: Add a custom syntax highlighter — the step-by-step for implementing `ICodeHighlighter`, picking a priority, and registering via `HighlightingOptions.AddHighlighter`.
- External: TextMateSharp — the upstream library that provides the grammar corpus; authoring new grammars follows its documentation, not Pennington's.

CHAPTER 3

Routing

URL paths and content routes

Under the Hood Why Pennington models URLs and filesystem paths as value-type records and why ContentRoute separates canonical identity from output location.

Why does Pennington wrap paths in `UrlPath` and `FilePath` records and track a separate canonical path and output file on every route, rather than passing strings around the way most static site generators do?

Context

The hard bugs in a content engine are almost never in the Markdown parser. They cluster at the points where one kind of path turns into another: where a URL becomes a file path, where a file path becomes a URL, where one URL is rewritten into another — locale prefix applied, base URL prepended, canonical link emitted, sitemap entry assembled. Each of those points is a place where the wrong kind of string silently passes through and the error surfaces somewhere completely unrelated.

When every path is a `string`, a method receives `string path` and callers squint at the parameter name to guess whether it carries a leading slash, a trailing slash, backslashes on Windows, a `.html` extension, a locale prefix, or the deployment base URL already folded in. The answer is almost never written down in the signature. It lives in a comment, a convention document, or the hard-won knowledge of whoever wrote the method. None of those survive a refactor.

The alternatives Pennington considered were a single `string` convention with documented normalization rules (document the expected shape and hope everyone remembers), a `System.Uri`-based shape (too broad — absolute URLs carry scheme and host concerns that belong to the transport layer, not the content model), and typed records per concern. The typed-record approach won, and the rest of this page explains why the shape holds up in practice — paths as value types, composition as an operator, and canonical identity separated from output location at every level.

How it works

`UrlPath` and `FilePath` as value types

`UrlPath` is a `readonly record struct` wrapping a single string. The implicit conversion from `string` keeps call sites readable — you can pass a string literal where a `UrlPath` is expected — but every parameter that means "URL" is typed that way rather than typed as `string`. That distinction is what lets the compiler catch the class of bugs that unit tests used to catch: a file path going into a URL slot, or a URL going into a file slot.

The `/` operator handles path composition. It trims a trailing slash from the left operand and a leading slash from the right, then joins them. The reason that operator exists, rather than a helper method named `Combine` or `Join`, is that composition is the dominant operation on paths in this codebase, and infix notation makes the intent read naturally at call sites. The normalization methods — `EnsureLeadingSlash`, `EnsureTrailingSlash`, `RemoveTrailingSlash`, `RemoveLeadingSlash` — share vocabulary across every call site so that "does this path have a leading slash?" is never a judgment call at the use site; it is answered once by the type.

The `Matches` method does real work for the resolver and link checker: it treats `/foo/`, `/foo/index.html`, and `/foo` as the same directory page. That behavior centralizes a subtle normalization rule that would otherwise have to be replicated, slightly differently each time, wherever route matching happens.

`FilePath` is the filesystem-shaped peer — same value-record shape, same `/` composition operator, with `Extension`, `FileName`, and `FileNameWithoutExtension` standing in for the URL normalization helpers. The two types are deliberately not interchangeable: a URL is a logical address, a file path is a disk location, and the boundary between them is crossed explicitly through `ContentRoute` rather than accidentally through an untyped string. See Pennington.Routing.ContentRoute for the full member surface of both.

`ContentRoute`: canonical versus output

Every page has two different identities that happen to look similar. The canonical identity is the URL the reader bookmarks, the URL the xref resolver writes into cross-links, the URL the sitemap lists. The output location is where the static build writes HTML on disk. For a page served at `/docs/getting-started/`, canonical identity is `UrlPath("/docs/getting-started/")` and output location is `FilePath("docs/getting-started/index.html")`. Those differ by a trailing slash and a filename, and the difference matters: conflating them is how sitemaps end up listing on-disk paths or xrefs emit `/index.html` into bookmarked URLs.

`ContentRoute` holds several fields alongside the canonical and output paths. `SourceFile` points back at the Markdown file on disk, or is absent for programmatic routes. `Locale` annotates which translation this route serves. `IsFallback` flags routes that serve default-locale content where a translation is missing. The `AbsoluteUrl` composition method is a deliberate one-line operation rather than an automatic transform: it produces a fully qualified URL for feeds and structured data. Locale prefixing, base-URL application, and absolute-URL composition are all separate call sites, each composing the canonical path rather than quietly mutating it.

Why this matters for locale and base-URL rewriting

This is where the canonical-versus-output separation earns its keep. Several `IHtmlResponseRewriter`s transform rendered HTML before it reaches the wire — xref resolution emits canonical paths from uids, locale rewriting prefixes the active locale, base-URL rewriting prepends the deployment prefix as the transport concern (the response-processing explanation owns the full chain and its ordering). The

invariant they all depend on is the one this page is about: the canonical path stays stable as transforms layer on. None of these rewriters compose cleanly if the canonical URL is conflated with the output URL — the output file needs to be computed once at route construction and then left alone.

The same separation drives the build crawler. `OutputGenerationService` fetches `CanonicalPath` over HTTP and writes the response body to `OutputOptions.OutputDirectory / OutputFile`. If canonical and output were a single string, every rewriter in the chain would need to know whether it was rewriting "for display" or "for disk" — and those two modes would have to stay in sync across every contributor and every future extension. Because they are two distinct fields on one record, rewriters only ever touch canonical paths and the crawler only ever writes output files. The two worlds do not need to negotiate.

Further reading

- Reference: Routing types
- How-to: Host under a sub-path (base URL)
- Explanation: Response processing and rewriters
- External: Parse, don't validate (Alexis King)

Why the sidebar mirrors your folders

Under the Hood Why Pennington derives the sidebar tree from folder structure and front-matter order instead of a hand-written nav file, and what that costs when sections reorder.

Why does the Pennington sidebar reflect the folder layout even for folders that were never named in front matter, and where does the ordering come from?

Context

Every content service produces a flat list of `ContentTocItem` records. Each item carries a route, a title, an authored order value, a section label, a locale, and a `HierarchyParts` array — the canonical path segmented on `/`. The sidebar, by contrast, is a tree: folders group pages, sections expand and collapse, and the currently-viewed page is highlighted with its ancestors open. The gap between "flat list" and "navigable tree" has to be bridged somewhere.

Some documentation tools require a hand-written `nav.yml` that explicitly names every section, sets every order, and assigns every label. The maintenance cost is real — the config file drifts whenever authors move pages, and it duplicates information that the filesystem already encodes. Pennington's design target was to treat the filesystem as the primary outline and derive everything else from it, with `order:` in front matter as the only setting authors need to reach for.

The result uses two signals: **folder structure** supplies the tree, and **front matter** supplies leaf ordering and breadcrumb labels. There is no third configuration file sitting between them. The sections below trace how `NavigationBuilder` performs that fold — from flat list to tree, including what happens when a folder has no index page and how locale prefixes are removed before the recursion starts.

How it works

HierarchyParts folds the flat TOC into a tree

Each `ContentTocItem` carries a `HierarchyParts` array — for example, the item at `/how-to/configuration/search` arrives with `["how-to", "configuration", "search"]`. `NavigationBuilder.BuildTreeAsync` recurses level by level rather than item by item. At each depth it selects items whose `HierarchyParts.Length` equals `depth + 1` and whose prefix matches the current parent path, orders them by `Order` then case-insensitive title, and deduplicates by canonical path. That last step guards against two content sources registering overlapping subtrees — a situation that would otherwise produce duplicate sidebar entries.

Recursing level-by-level rather than item-by-item is what makes sibling ordering work across content sources that have no knowledge of each other. The algorithm sees all siblings at once before it descends, so the relative ordering between a page from a Razor source and a page from a Markdown source is resolved in the same pass.

There is one special case at depth 0: a `ContentTocItem` whose `HierarchyParts.Length` is 0 is treated as the area's landing page. Its hierarchy was already stripped by the content service before the list was handed to the builder, so the builder injects it at the top of the tree with `Order = int.MinValue`. That anchors it above every other root entry regardless of what `order:` value was authored.

Each field on `ContentTocItem` plays a distinct role in the algorithm: `HierarchyParts` shapes the tree, `Order` and `Title` sort siblings, `SectionLabel` surfaces only in prev/next and breadcrumbs, and `Locale` feeds the filter described below (see Pennington.Content.ContentTocItem for the type).

The `currentPath` parameter passed to `BuildTreeAsync` marks items `IsSelected` and propagates `IsExpanded` up the ancestor chain. The same tree therefore powers both the "where am I" highlight and the collapsed or expanded state of every surrounding folder. The method returns an `ImmutableList<NavigationTreeItem>`, so the entire tree is a value rather than a mutable model the rendering layer binds to directly.

Sections without a direct content file

When `BuildLevel` finds deeper descendants under a hierarchy segment that has no direct item at the current depth — a folder like `/how-to/configuration/` with children but no `configuration/index.md` — it synthesizes a non-navigable section node on the fly. The title comes from `FormatSectionTitle`, which kebab-to-title-cases the folder segment: `getting-started` becomes "Getting Started". The node is given an empty `ContentRoute` so the rendering component treats it as a section header rather than a link, and `IsExpanded` is set by whether any descendant is currently selected.

This is the mechanism that lets an author drop markdown files into `/how-to/deployment/` without creating a `deployment/index.md` and still see "Deployment" appear as a collapsible sidebar heading. The folder itself is sufficient.

The important distinction is between folder-derived grouping and the per-page `sectionLabel:` front-matter key. Grouping comes entirely from subfolder; `sectionLabel:` controls only the label shown in breadcrumbs and prev/next. Two files carrying identical `sectionLabel: "Advanced"` values in different folders render under two different sidebar headers — each named after its own folder — rather than merging. Merging by label would let two unrelated folders collide under a single heading, reintroducing the configuration conflict the filesystem-driven approach was designed to eliminate.

The synthesized section node and a real leaf page share the same `NavigationTreeItem` record shape (see Pennington.Navigation.NavigationTreeItem); a section node carries an empty route, which is how the rendering component tells the two apart.

Ordering: front matter for leaves, `_meta.yml` for folders

Leaf pages at any given level sort first by their authored `Order` value, then by title using a case-insensitive ordinal comparison as a stable fallback. Folders, having no front matter of their own, take their order from a different source: by default a synthesized section node gets `Order = children.Min(c => c.Order)`, so it sorts as if it were whichever of its children would sort first. A folder with an `index.md` but no sidecar takes its order and title from that page instead.

That default — min-of-children — has an awkward consequence: sibling sections interleave by the smallest `order:` value found anywhere inside each. If "Getting Started" contains a page with `order: 10` and "Deployment" contains a page with `order: 20`, the sidebar places Getting Started above Deployment. If someone later adds a page with `order: 5` to Deployment — perhaps because they want it first within that section — the whole Deployment group jumps above Getting Started. Sites that grow past a handful of sections end up choosing globally-unique numeric prefixes (`301010, 301020, 302010, …`) to keep folder ordering stable while still allowing in-folder inserts.

The escape hatch is a `_meta.yml` sidecar: a folder declaring its own `order: 1` decouples its position from its children, so each folder's pages can restart at `1` without disturbing where the folder lands in its parent. The sidecar can also override the folder's display title and opt the subtree into a dedicated `llms.txt` split. The full schema and precedence rules live in the Folder sidecar (`_meta.yml`) reference.

Locale prefix stripping

Non-default locales are stored on disk under a locale folder (`Content/fr/...`), so a French page at `/fr/how-to/configuration/search` arrives in the flat list with `HierarchyParts` reading `["fr", "how-to", "configuration", "search"]`. If `BuildTreeAsync` recursed over those items without any preprocessing, every French page would nest under a `/fr/` root while English pages sat at the top level — two unrelated sibling trees rather than one coherent per-locale outline. The min-of-children ordering would also produce incorrect results, because "the first page in my folder" would mean something different in each language subtree.

`FilterByLocale` runs before the level-by-level recursion begins. It keeps items whose `Locale` matches the requested locale or is `null` (for locale-agnostic content), and — for non-default locales only — strips `HierarchyParts[0]` when it equals the locale code. The recursion then sees a shape identical to what

the default locale sees, with the language prefix removed. The min-of-children ordering and the section-node synthesis therefore work the same way regardless of which locale is being rendered. Items carrying `Locale == null` pass through every filter unchanged, which is why redirects and feeds appear in every locale's sidebar without requiring duplicate files on disk.

Further reading

- Reference: Folder sidecar (_meta.yml) — the full schema and precedence rules for the folder-level overrides this page motivates.
- Reference: Navigation components (TableOfContentsNavigation, OutlineNavigation) — the UI that consumes the tree `NavigationBuilder` returns.
- How-to: Customize the sidebar — the recipe that leans on the ordering rules this page explains.
- Tutorial: Organize content with sections and areas — the tutorial that introduces folder-driven grouping for new authors.

Cross-reference resolution

Under the Hood Why Pennington links pages by symbolic uid rather than filesystem path, and how the two-phase resolver turns those uids into canonical URLs without the authoring cost of hand-coded links.

Why does Pennington resolve links through a symbolic `uid` indirection rather than letting authors write the URL of the target page directly?

Context

The filesystem location of a markdown file is an unstable coordinate. Renames, reorganizations, and section moves all change the URL, and every hand-written link across the site then has to be found and updated. Relative links make the problem slightly smaller — they only break when the source or the target moves — but they do not eliminate it, and they fold poorly across locales where the same logical target has a different URL per language.

A `uid:` declared in front matter is a coordinate the author controls. Moving the file does not move the uid, and translated copies can share one uid across locales. The cost of this stability is indirection: at render time the engine must look the uid up and substitute the real URL. The rest of this page describes the shape of that lookup and why it runs in two passes.

How it works

Collection phase: uid → URL map

When the host starts, `XrefResolver` asks every registered `IContentService` for its `GetCrossReferencesAsync()` and folds the results into one case-insensitive uid → URL map. Because `XrefResolver` is registered with `AddFileWatched<T>`, any change under a watched content path tears down the cached resolver instance and the next request reconstructs it. Renaming a file or editing a front-matter `uid:` is therefore visible on the very next response without an explicit cache bust.

Uid collection is a content-service concern, not a markdown concern. `MarkdownContentService<T>` contributes uids by reading the optional `Uid` member on `IFrontMatter` for each discovered page; `RazorPageContentService` contributes none by default; a custom `IContentService` can synthesize `CrossReference` records for rows in a JSON feed or entries in a database. The resolver does not care where the uid came from — it only sees `(uid, title, route)` triples and picks the first one it encounters for each uid. That first-write-wins rule is what lets a default-locale route shadow later fallback duplicates without any special handling.

Pre-parse tag pass (`<xref:uid>`)

`XrefHtmlRewriter.PreParseAsync` runs before AngleSharp sees the response body, because `<xref:uid>` is not valid HTML. An HTML parser would try to interpret it as an unknown element, and the `:uid` portion would be lost or coerced into an attribute during error recovery. A regex substitution instead walks every match on the raw string and replaces it with a real `<a>` element whose `href` is the resolved canonical path and whose text content is the resolver's stored title. Downstream DOM-based rewriters — locale prefixing, base-URL stamping — receive ordinary HTML they can parse normally.

The tag form is the one to reach for when you want the engine to supply both the URL and the link text. `<xref:explanation.routing.url-paths>` renders as an anchor whose visible text is the target page's title with no additional markup from the author. If the uid is missing, the substitution still happens but the anchor carries `data-xref-error` and `data-xref-uid` attributes, making the broken link visible in dev-tools and stylable in CSS. A `DiagnosticContext` warning is accumulated on the request in parallel so the problem surfaces in both the dev overlay and the build report.

DOM attribute pass (`href="xref:uid"`)

The second form is the markdown-native `[text](xref:uid)`. Markdig renders it as an ordinary `<a href="xref:uid">text</a>`, which AngleSharp parses without complaint. `XrefHtmlRewriter.ApplyAsync` selects `a[href^='xref:']` on the shared document, resolves each uid, and rewrites only the `href` — the anchor's existing text content is preserved because the author chose it. The one exception is the rare case where the text and the href are the same literal `xref:uid` string, which happens when `<xref:foo>` is used as a bare markdown link with no `[text]` wrapper; in that case the resolver substitutes the stored title so the anchor renders meaningfully.

This pass operates on the same `IDocument` that the locale and base-URL rewriters mutate next — they all share one AngleSharp parse/serialize round trip, and xref resolution runs first so the canonical paths it emits are visible to those downstream rewriters (why that order holds). The two-phase split exists because the two link forms are genuinely different syntactic problems. One is not parseable HTML and has to be rewritten as a string; the other is parseable HTML and is cleaner to rewrite on the DOM. Merging them into a single pass would require parsing the document first, which would defeat the purpose of the pre-parse stage.

Broken xrefs as diagnostics

An unresolved uid is never a hard failure. Both resolver phases emit a `Warning` to the scoped `DiagnosticContext`, and the response still renders — the anchor carries the `data-xref-error="Reference not found"` attribute so the reader sees a link that does not work rather than a crashed page. In dev serve, `DiagnosticOverlayProcessor` collects the context and renders the warning into the corner panel of the running site.

During a static build, the unified dev-and-build code path carries this through. `OutputGenerationService` crawls the live host and folds the accumulated diagnostics into the `BuildReport` alongside the broken-link, missing-trailing-slash, and render-failure entries. The result is that every unresolved uid emitted during the crawl shows up in the printed build report and, if it reaches `Error` severity, sets the process exit code — without Pennington needing a separate "verify xrefs" build step.

Further reading

- Reference: Response processing interfaces — the member catalog for `IHtmlResponseRewriter`, `XrefHtmlRewriter`, and execution order.
- Reference: Markdown extensions catalog — cross-reference tag and attribute syntax alongside the other non-CommonMark features.
- How-to: Link to a page without breakage — the authoring recipe for setting `uid:` and linking with `<xref:uid>` or `[text](xref:uid)`.
- Related explanation: The response-processing pipeline — why xref resolution lives inside the shared AngleSharp pass and runs first in the rewriter order.
- Related explanation: URL paths and content routes — the `UrlPath` / `ContentRoute` value types that `CrossReference` carries into the resolver.

CHAPTER 4

Spa

SPA navigation through region swaps

Under the Hood Why Pennington's SPA fetches the canonical HTML page and swaps marked regions — reusing the server render and its rewriters instead of a parallel JSON envelope.

DocSite ships a small SPA navigation engine — in-site clicks fetch the canonical URL, parse the response, swap marked regions, and merge head metadata. The design question this page answers: why fetch the same URL the address bar shows and parse it client-side, instead of round-tripping a small JSON envelope or letting the browser do a full reload?

Context

Classic server-rendered sites swap the whole document on every click — simple, full-fidelity, but heavy and visually jarring when the shared chrome redrawing is indistinguishable from the content changing. Full SPAs hydrate the entire app in the browser, make every navigation instant, and pay for it with a multi-megabyte runtime on first load, a separate SEO story, and a rendering path that diverges from whatever the server would have produced. Documentation sites sit awkwardly between those two extremes. Most of each page is static prose that does not benefit from client rendering, a couple of areas — the article body, the top bar with the language switcher — genuinely want to update on each navigation, and a third category of chrome (the sidebar's active link, the page outline) should not re-render at all but does need its state nudged when the page changes around it.

Pennington takes a third position. Every page is fully server-rendered on first load. When the visitor clicks an in-site link, the browser fetches the same URL — the canonical HTML page — parses it with `DOMParser`, swaps regions tagged `data-spa-region` from the new document into the current one, and merges head metadata. The server render and its HTML rewriters are reused as-is, so there is no second pipeline to keep in sync.

How it works

One render, many slices

The first request to any URL returns complete server-rendered HTML, exactly as it would without SPA support — good for cold loads, good for crawlers, functional when JavaScript is disabled. Once the browser has that page and the `spa-engine.js` script from `Pennington.UI` is active, the client intercepts same-origin link clicks and re-fetches the destination URL. The response is parsed into a `Document` that supplies both the regions that change and the head deltas that follow them.

Every server-side rewriter — xref resolution, locale-aware link rewriting, base-URL prefixing, anything else registered as `IHtmlResponseRewriter` — applies to that response by default, because it travelled through the same `ResponseProcessingMiddleware` as a fresh-tab visit. There is no second pipeline to

mirror.

The `data-spa-region` contract

Anywhere in the layout that should update on navigation gets a `data-spa-region="name"` attribute. The DocSite layout marks two regions out of the box:

- `content` — the article body, including breadcrumbs and prev/next links.
- `outline` — the right-rail page outline, populated client-side from the article headings on every commit.

Anything outside a marked region — the top bar, the sidebar, the outer page chrome, the mobile menu's expanded state, scroll position — stays put. The header and sidebar are intentionally outside the region system: the search button keeps its event handlers across navigations, and the sidebar keeps its scroll position while its active-state flags are patched in place from the destination's HTML. Both follow the persistent-chrome pattern covered in the next section.

The client picks the regions in the current document, finds elements with the same name in the parsed response, and swaps `innerHTML`. If the set of regions does not match — for example, navigating from a `MainLayout` page (`content` plus `outline`) to a `FullWidthLayout` page (only `content`) — the engine triggers a full page load. Crossing a layout boundary reloads rather than half-updating the page.

Persistent chrome

Some chrome has the same shape on every page. The DocSite sidebar is the canonical example — across every doc page within a layout, the table of contents is structurally identical, and the only thing that varies is which link carries `data-current="true"`. Marking that as a swap region works, but it throws away DOM nodes the engine is about to rebuild to the same shape, and along with them: the user's scroll position in a long sidebar, focus on the link they tabbed to, any expand/collapse state a reader interacted with, and any iframe or animation state inside the region.

Leaving the element outside `data-spa-region` is the answer. The engine never queries it, never swaps its `innerHTML`, never re-runs scripts inside it. The same DOM nodes survive every navigation — `scrollTop`, focus, and live state are preserved by the browser automatically, because nothing relocates them. The cost is that the active-state attributes no longer change automatically; the server-rendered destination has the right `data-current` flags, but they live in HTML the engine no longer looks at.

The `spa:commit` event is the extension point. It fires after each navigation with `detail.doc` — the parsed `Document` of the destination, the same HTML the server would have rendered for that URL. Consumers read the destination's chrome out of `doc`, copy whatever state actually changed onto the live nodes, and let the server-rendered destination stay authoritative. Active-state flags, the active-area pill, anything the server already computes — gets patched in place rather than re-derived on the client. The DocSite uses this to keep the sidebar's `data-current` flags in sync without ever rebuilding the tree.

Head merging

The `<head>` of the parsed response is authoritative for everything page-specific: the client overwrites the title and a fixed set of managed tags — the page metadata the server already computes per URL — from the destination's head. Stylesheet `<link>` elements are merged by href — any new ones append to the head before the region swap so the browser has the rules ready when the new content paints. A stylesheet tagged `data-spa-reload` re-fetches with a cache buster on every navigation, the opt-in workaround for JIT stylesheets like MonorailCSS in dev where the URL stays constant but the contents diverge per page; the attribute is documented in SPA engine attributes and events.

Synchronous swap, no animation

The round trip is small but not instant. View-transition wrappers (`document.startViewTransition` with a short cross-fade) and per-region skeleton placeholders were both on the table during the design and rejected. Both layers introduce more visible motion than they mask: the cross-fade is a flash for the eye to notice, and a skeleton replaces real content with shimmer the moment the network takes longer than a tick. The engine waits instead — old content stays on screen while the fetch runs — and the swap, scroll reset, and head update all execute in one synchronous block so the browser paints the new page as a single frame. Hover-prefetch hides the wait for the cases where it would otherwise be felt.

A top-of-viewport progress bar handles the unusual case where the response takes longer than the engine's silent threshold — a cold cache, a slow CDN edge. It only shows after the threshold elapses, so fast navigations never see it.

Why one render path

A second rendering path — the JSON-envelope approach an earlier version of this engine used — looked appealing at first: a small payload, a typed metadata header, no head parsing. In practice it carved a permanent fork down the middle of the codebase. Locale rewriting did not apply to JSON responses unless re-implemented. Per-island parameter dictionaries duplicated whatever the page's Razor render already built. Active-state nav, breadcrumbs in the sidebar, language-switcher hrefs — anything outside the swapped region went stale, and consumers patched it back up with one-off client-side JavaScript. Each new HTML rewriter had to be applied twice, or quietly skipped on SPA navigation.

The single-path approach trades some payload size for the elimination of all of that. The full HTML response gzips to within a few KB of the JSON envelope it replaced, and the prefetch-on-hover path hides whatever cost remains.

Further reading

- Reference: SPA engine attributes and events — the `data-spa-*` attribute contract and the `spa:commit` / `spa:before-navigate` events this page describes.
- How-to: Ship a custom client-side widget — attach your own browser behavior to the server-rendered HTML and re-bind it from `spa:commit` after each navigation.
- Reference: Pennington.DocSite.DocSiteOptions

- External: Islands Architecture (Jason Miller) — the term "island" originates here; Pennington's regions are a degenerate case where the server renders every "island" itself.

CHAPTER 5

Localization

Locale-aware URLs and content fallback

Under the Hood Why Pennington treats the URL path prefix as the authoritative locale signal, how DocSiteContentResolver strips it and falls back to the default locale, and why the search index is split per locale.

When a reader hits `/fr/guides/intro`, how does Pennington know which language to serve, which markdown file to load, and what to do if `guides/intro.md` only exists in English?

Why the URL is the locale signal

A multilingual site has to pick a locale signal early enough that routing, content resolution, HTML rewriters, and the search index all agree on it. The obvious candidate is the browser's `Accept-Language` header, but that signal is noisy: users travel, share links, and sit behind corporate proxies, and a locale chosen from a header cannot be bookmarked, cached by a CDN, or indexed distinctly by search engines. Pennington uses the URL instead. The path prefix (`/fr/…`, `/es/…`) is the single source of truth, and the default locale owns the unprefixed root. That same prefix drives both request-time content resolution in development and build-time output placement during the static crawl — one code path, one answer. Because translation coverage is rarely complete, the question "what happens when `/fr/foo` has no French source?" needs a principled answer, and that is what the rest of this page explores.

How it works

URL prefix drives detection

`LocaleDetectionMiddleware` reads `HttpContext.Request.Path`, asks `LocalizationOptions.GetLocaleFromUrl` which locale the first path segment maps to, and populates the scoped `LocaleContext`. When the detected locale is not the default, the middleware rewrites the request path to strip the prefix and pushes that prefix onto `PathBase` so URL generation stays correct. The net effect is that Blazor routing and every downstream consumer see `/guides/intro` rather than `/fr/guides/intro`, which means a single `@page "/guides/intro"` directive serves every locale without duplication.

The middleware never consults `Accept-Language`. The culture provider (`PenningtonUrlRequestCultureProvider`) also derives its answer from the URL, so the entire request pipeline agrees on the same locale code without any negotiation step. Prefix-first means the same link produces the same page for everyone who clicks it, regardless of their browser settings or location.

Resolving content: precomputed routes, then a runtime miss

`DocSiteContentResolver.GetContentByUrlAsync` takes the full URL — locale prefix intact — and resolves it against the registered content services. The first attempt asks every `IContentService` for a route that matches the exact URL, prefix and all. This is where two distinct flavors of fallback get conflated if you are not careful, so it is worth separating them.

The first is *startup-precomputed* fallback. When a multi-locale `MarkdownContentService` discovers a default-locale file, it registers not only that file's own route but also an extra route at each non-default locale prefix that lacks its own copy — `/fr/guides/intro` pointing at the English `guides/intro.md`, with `ContentRoute.IsFallback` already set to `true`. That route exists in the table before any request arrives. So the exact-URL match for `/fr/guides/intro` can succeed outright, and the resolver reads the fallback flag off the route it found (`rendered.Route.IsFallback`) rather than computing anything. This is the common path: most missing translations are known at startup.

The second is *runtime* fallback, and it only runs when the first attempt finds nothing at all. If no route matched, the site is multi-locale, and the active locale is not the default, the resolver strips the prefix via `LocalizationOptions.StripLocalePrefix` and resolves again against the content-relative path (`guides/intro`). If that second resolve succeeds, the resolver sets `IsFallback` itself. This path covers content sources that don't precompute fallback routes the way `MarkdownContentService` does, so the resolver still degrades gracefully to the default locale.

Either way — flag read off a precomputed route, or set after a runtime miss — the resolver records `RequestedLocale` so the view layer can render a "this page has not been translated yet" notice via `FallbackNotice`. The resolver never rewrites URLs — the URL the reader typed stays in the address bar, only the content source changes.

Fallback: default locale stands in

There is exactly one fallback step: the default locale. A missing `/es/guides/intro` falls through to `guides/intro` — the unprefixed default-locale source — not to `/fr/guides/intro`. There is no cascade across non-default locales, no per-locale fallback chain, and no "similar language" matcher. The reason is that a cascade hides coverage gaps. Readers end up seeing a page in a language they did not request and cannot explain why, and translators cannot tell which pages still need coverage because every miss silently inherits from a neighbor. A single fallback step keeps the rule easy to reason about and makes missing translations easy to find.

The default locale is not a special kind of locale — it is the locale that owns the unprefixed URL space. `StripLocalePrefix` is a no-op for the default locale, `BuildLocaleUrl` emits unprefixed URLs for it, and content authored at `Content/guides/intro.md` is default-locale content by virtue of sitting outside every locale subdirectory. The fallback rule is a consequence of those URL-math rules, not a separate "fallback locale" setting.

`StripLocalePrefix` (see Pennington.Localization.LocalizationOptions) is pure URL math with no file-system knowledge. `DocSiteContentResolver` decides whether the stripped path resolves to a file on disk, and that separation is what lets the same helper serve both request-time fallback and build-time URL

generation.

Per-locale search indices

`SearchArtifactService` emits one set of sharded artifacts per configured locale, keyed by each TOC item's route locale (or `DefaultLocale` when the route has none). `UsePennington` mounts `SearchArtifactMiddleware` under `/search/`, which serves the per-locale shards (`/search/{locale}/index.json` plus its segment files), and the DeweySearch client fetches only the bundle for the active locale. This keeps the client payload small on multilingual sites: a reader browsing `/es/` never downloads French or Japanese documents. More importantly, it keeps search semantics scoped. A query typed into the Spanish UI ranks against Spanish content, not against a mixed-language pool where term frequencies across languages distort each other's results.

A page that exists only in the default locale appears once, in the default-locale index. Fallback happens when a page renders; it does not add that page to other locales' indices. That asymmetry is deliberate. When a French reader searches for a term that only exists in English content, the right answer is "no results in French" rather than silently returning English hits the reader cannot read. The same per-locale split applies to sitemap construction and `hreflang` alternate-language tags, so search, sitemap, and alternates report the same set of pages for each locale.

Further reading

- Reference: LocalizationOptions — `DefaultLocale`, `AddLocale`, and the URL helpers that back this mechanism.
- How-to: Enable multiple locales — the recipe for populating `LocalizationOptions`, organizing content subdirectories, and wiring `UseLocaleRouting`.
- External: W3C — Language tags in HTML and XML — background on BCP 47 locale codes, which the URL prefix scheme surfaces one-to-one.

CHAPTER 6

Dev Experience

Hot reload and file watching

Under the Hood Why Pennington ships its own file watcher and WebSocket reload channel, and how the dev-only script is kept out of published builds.

Content files — `.md` sources, front matter, images, assets tracked under a source's `ContentPath` — are not part of the .NET compilation. Restarting the host for every markdown typo would tear down Kestrel and throw away the expensive in-memory caches that make the second request fast. Pennington instead watches content in process and reloads only the browser: an in-process file watcher, services that discard and rebuild their derived caches on change, and a debounced WebSocket channel through which the browser is told to reload. WebSocket also makes server restarts easy to detect on the client — the socket closes and reconnects, and the browser reloads — without the careful `onerror` plumbing SSE would need for the same signal. Polling, the third alternative, adds latency under load and noise under idle.

How it works

The mechanism is a single chain: files change, cached services drop their state, a debounce window elapses, and the browser reloads.

Watching content directories

A service tells the engine which directories to watch by declaring `WatchScopes` — the public contract through which every file-reactive service registers its content roots. Creates, deletes, and renames count as changes, not just edits in place, so a new markdown file or a deleted asset reloads the same way a save does. The watcher behaves consistently across Windows and WSL, the two platforms most contributors run, so the dev loop feels the same on either.

The `IFileWatchAware` contract

Several services build expensive lookup tables from disk on startup: link resolvers, cross-reference uid maps, search indexes, sitemaps, and blog content resolvers. They register through `AddFileWatched<T>`, which is constrained to types implementing `IFileWatchAware` — the single contract every file-reactive service shares. A service declares the directories it needs watched and an `OnFileChanged` method whose return value says how the change should be handled: `Ignore` it, report it `Refreshed` its own state, or ask to be `Recreate`d.

The services above return `Recreate`, so on the next request the stale instance is dropped — disposed if it implements `IDisposable` — and a fresh one is rebuilt through normal constructor injection. The approach is to discard and rebuild the whole instance rather than bust individual caches: no service needs to know when to flush itself, because the engine replaces the entire instance when the underlying content moves.

This is the extension point you reach for when your own service caches something derived from content files. The how-tos that lean on it: write a custom content service, publish a custom feed from a content service, and use a YAML or JSON data file in pages.

Debouncing and broadcasting over WebSocket

`LiveReloadServer` resets a 300ms debounce timer on every change notification, so it broadcasts a single reload only after 300ms of quiet — coalescing rapid saves (editor auto-save, multi-file renames) into one browser reload. It listens on the WebSocket endpoint `/__pennington/reload`, which is worth knowing if a reverse proxy or a Content-Security-Policy sits in front of the dev server and needs to allow the upgrade.

Script injection and reconnection

`LiveReloadScriptProcessor` is an `IResponseProcessor` at `Order = 20`, positioned between the HTML rewriting pipeline at `Order = 10` and the diagnostic overlay at `Order = 30`. When active it finds the last `</body>` tag and inserts an inline script that opens a WebSocket to `/__pennington/reload`. The script includes three refinements over a naive `location.reload()` approach: a `beforeunload` guard that suppresses reconnect attempts during normal page navigation, a 150ms delay before reload so the response pipeline has time to settle, and a reload on reconnect so that a host restart refreshes the browser without waiting for a file-change message.

Build-mode gating

Both `LiveReloadScriptProcessor` and `UseLiveReload` check `PenningtonCli.Current.IsHeadlessOneShot` — true for any headless one-shot run, which covers `build` and `diag` alike, not just the `build` verb. When it is true, the processor's `ShouldProcess` returns `false` and the middleware skips endpoint registration entirely. This means the `OutputGenerationService` crawler sees clean HTML with no script and no WebSocket endpoint: no publish-time stripping step, no build configuration to set, and no dev-only flag to forget.

Relation to `dotnet watch` and .NET Hot Reload

This is a different layer from .NET Hot Reload and `dotnet watch`, and the two are complementary. .NET Hot Reload patches running CLR code — your `.cs` and `.razor` source — and `dotnet watch` restarts or re-applies edits when compiled code changes. Pennington's watcher covers the half they don't: content files (`.md`, front matter, images, `_meta.yml`, data files) that never enter the compilation. Run the host under `dotnet watch` and you get both — code edits patched by the runtime, content edits reloaded by Pennington — without either stepping on the other.

Disabling reload or tuning the debounce

In serve mode live reload is always on, and the 300ms debounce and reconnection behavior are fixed; there is no option to tune them or switch reload off while still serving. The single off-switch is build mode: a headless one-shot run (`build` or `diag`) gates the whole subsystem out, which is exactly what you want

for published output. If you need a dev server with no reload at all, host the engine without `UsePennington`'s dev path rather than reaching for a flag.

Further reading

- Reference: DI and middleware extension methods
- Reference: Response processing interfaces
- Explanation: The response-processing pipeline
- Explanation: Dev mode and build mode share one code path

CHAPTER 7

Positioning

What the DocSite and BlogSite templates wire for you

Under the Hood DocSite and BlogSite are pre-assembled shortcuts on top of the AddPennington engine — what each template wires, and where you have to drop down to the engine itself.

`AddDocSite` and `AddBlogSite` are shortcuts. Each one pre-assembles a host that `AddPennington` would otherwise have you wire by hand. This page is about what they assemble — and where that assembly stops.

Context

`AddPennington` is the engine, and building on it is how Pennington sites are made: content discovery, the rendering pipeline, response processing, and diagnostics. It expects the host to bring its own layout, routing, and CSS wiring — which the getting-started tutorials walk through end to end.

`AddDocSite` and `AddBlogSite` are templates layered on that engine. Each composes `AddPennington`, `AddMonorailCss` (Pennington's integration of MonorailCSS, the Tailwind-compatible .NET JIT compiler), and a Razor `App` component into a single call with a small options surface — `DocSiteOptions` or `BlogSiteOptions`. They exist to skip the wiring for two shapes that come up often: a Divio-style documentation site, and a site where the blog *is* the site. When a site is exactly one of those, the template is a real head start. When it is not, the host is built on `AddPennington` directly, which is a normal way to build a Pennington site.

The distinction that matters is that neither options record is a mirror of `PenningtonOptions`. Each is a curated subset plus a handful of template-specific options, and the things it deliberately does not expose mark the edge of what the template covers.

One template per host

`AddDocSite` and `AddBlogSite` cannot run in the same app — a host wires one or the other, never both. The constraint is structural: each call registers its own Razor `App` component through `MapRazorComponents<App>`, claims the root route `/`, and composes its own MonorailCSS theme. Two templates in one host means two `App` components and two pages fighting for `/`, which is not a configuration the engine resolves. Pick the template that matches the site's primary shape.

The split is rarely a hard choice, because the overlap has a built-in answer. A documentation site that also wants a blog stays on DocSite: it has a native blog you switch on by adding a `Content/blog/` folder, with no `Program.cs` change. BlogSite is the right choice only when the blog *is* the site — its home page, archive, and tag routes are the whole front end. When a host genuinely needs both a doc-shaped and a blog-shaped surface that the native blog cannot express, the answer is not two templates but `AddPennington` directly.

The rest of this page examines DocSite in detail, then returns to where BlogSite differs. BlogSite is the same kind of shortcut with a narrower options surface, and most of the reasoning carries over.

How it works

What DocSite gives you for free

A single `AddDocSite` call wires quite a bit:

- The Pennington engine, with site title, description, canonical URL, and content root forwarded from `DocSiteOptions`.
- One `AddMarkdownContent<DocSiteFrontMatter>` registration rooted at the content folder.
- A pre-scoped `llms.txt` and search index, both defaulting to `#main-content`, the wrapper around the stock article.
- The Razor `App` component and `DocSiteArticle` rendering shell.
- Mdazor inline components — `Badge`, `Card`, `Step`, and others — registered so markdown can embed UI without per-site plumbing.
- MonorailCSS with the DocSite theme.
- SPA navigation through the `data-spa-region` markup the layout emits, with no extra DI registration — the client script lives in `Pennington.UI`.

The value is less about the feature count than about ordering: every one of those registrations lands in a compatible order with the others. Getting that ordering right, especially between the pipeline, the response processor, and the search index scoping, is most of what trips up a hand-rolled host on the first attempt.

What DocSite caps

DocSite owns exactly one `AddMarkdownContent<DocSiteFrontMatter>` registration. Wiring a second front-matter type — say, a blog post shape alongside docs — is not reachable by setting a `DocSiteOptions` property. It takes either the ConfigurePennington callback (a `Action<PenningtonOptions>` that runs after DocSite's defaults land) or dropping to bare `AddPennington` outright. The callback is enough for adding one more source. It falls short when the extra source needs different theming, a different layout, or a different slot renderer.

`DocSiteOptions.ColorScheme`, `DisplayFontFamily`, `BodyFontFamily`, `ExtraStyles`, and `CustomCssFrameworkSettings` offer tweak points against MonorailCSS, but the theme composition itself — `AddMonorailCss` plus the DocSite `App` component plus the `DocSiteArticle` shell — is fixed. Replacing the `App` component or introducing a non-article layout means registering extra routing assemblies via `AdditionalRoutingAssemblies` and accepting that custom components ride alongside DocSite's, not in place of them.

`ContentSelector` on `DocSiteOptions` defaults to `#main-content` — the wrapper the stock layout places around the article — and accepts any CSS selector, including the empty string to index the full body when the layout has been replaced. That one selector picks the body element that the search index, the llms.txt sidecars, and the build-time link audit all consume, so chrome is stripped once and all three read the same element.

The escape hatch — DocSite's source as reference

When a site outgrows what the options expose, the productive move is not to fight `DocSiteOptions` but to copy the pattern out of `DocSiteServiceExtensions.AddDocSite` and paste what is needed into a bare `AddPennington` host. That method is a single composition with no hidden state: the service registrations, the option forwarding, and the middleware order are all visible. Reading it gives a checklist of what a Pennington-shaped host needs, one that can be freely subsetted or extended.

The ExtensibilityLabExample project serves as the canonical bare-host reference. It wires extension points directly against `AddPennington`, uses its own markdown rendering via `MapGet`, and demonstrates that the engine runs happily without the DocSite template. For hosts whose shape the template does not fit, `examples/ExtensibilityLabExample/Program.cs` is a better starting skeleton than a blank `WebApplication.CreateBuilder`.

The example host is intentionally minimal — no Razor layout, no slot renderer, hand-written HTML strings. It demonstrates the engine surface, not a production layout pattern. The DocSite extension method is the right place to look for the layout recipe.

For a documented walkthrough rather than source to read, the first-site tutorial builds a bare `AddPennington` host step by step — the same path the templates are a shortcut for — and is the recipe to follow when a host needs its own layout and routing from the start.

The shape the template assumes

A template fits a particular kind of site. DocSite fits an article-centric documentation site rendered through a fixed Razor layout. A handful of site types fall outside it — and there the host is built on the engine, not the template:

- Multiple markdown front-matter types served from the same host with different themes or different layouts. `ConfigurePennington` can register a second source, but it cannot give that source a separate layout shell.
- Replacing the `App` component or the `DocSiteArticle` shell with a layout that is not article-shaped — a dashboard, a directory, a storefront.
- A non-Razor rendering story: custom `MapGet` handlers, Minimal API endpoints that emit HTML strings, or a reverse-proxy shape where the engine feeds a different front-end entirely.
- Embedding Pennington inside an existing ASP.NET app that already owns its routing, authentication, or layout conventions. Adding `AddDocSite` on top tends to fight those choices rather than cooperate with them.
- Shipping the engine as a library into another product where only the pipeline is needed, not the layout.

None of these is exotic. They are the common reason a host is built on `AddPennington` directly. The limits are narrow because the template is opinionated, and those opinions are about documentation sites specifically.

Where BlogSite differs

BlogSite is the same kind of shortcut, aimed at a different shape: a site where the blog *is* the site. A single `AddBlogSite` call composes `AddPennington`, forwards site identity and content paths from `BlogSiteOptions`, registers one `AddMarkdownContent<BlogSiteFrontMatter>` source rooted at `Content/Blog/`, wires MonorailCSS with the BlogSite theme, and registers the same Mdazor inline components DocSite does. The same ordering value holds — every registration lands compatible with the others, which is the part a hand-rolled host gets wrong first.

What BlogSite adds beyond DocSite is its route surface. The template ships Home, Archive, Tag, Tags, and Blog Razor pages inside `Pennington.BlogSite.dll`, so the home listing, `/archive`, `/blog/<slug>/`, the tag pages, and the `/rss.xml` feed exist without the host authoring a single `@page`. That is the inverse of DocSite, whose pages all come from markdown under `Content/`; BlogSite's structural pages are compiled into the template and its content is the posts you drop in.

The caps are tighter than DocSite's in two ways. First, `BlogSiteOptions` has no `ConfigurePennington` callback — the post-defaults `Action<PenningtonOptions>` hook DocSite exposes. Reaching engine surface the options do not forward means dropping to bare `AddPennington` rather than threading a callback. Second, there is no `ContentSelector`: the body element the search index and llms.txt consume is fixed by the BlogSite layout rather than selectable. The tweak points it does share with DocSite — `ColorScheme`, `ExtraStyles`, `DisplayFontFamily`, `BodyFontFamily` — adjust the theme without swapping the composition, exactly as on DocSite. What `BlogSiteOptions` adds instead are blog-shaped knobs: author chrome (`AuthorName`, `AuthorBio`), homepage composition (`HeroContent`, `MyWork`, `Socials`), and feed toggles (`EnableRss`, `EnableSitemap`). The full surface is in the BlogSiteOptions reference.

The escape hatch is identical in spirit. `BlogSiteServiceExtensions.AddBlogSite` is a single visible composition, and when a site outgrows the options the move is again to copy what is needed into a bare `AddPennington` host rather than fight the template.

Further reading

- Tutorial: Create your first Pennington site — building a host on `AddPennington` directly, the path the templates are a shortcut for.
- Reference: DocSiteOptions reference — the full list of tweak points with defaults and forwarding semantics.
- Reference: BlogSiteOptions reference — the BlogSite option surface, including the homepage and feed knobs DocSite has no equivalent for.
- How-to: Use multiple content sources — the canonical guide for the "two front-matter types" case, covering both the `ConfigurePennington` path and the drop to bare `AddPennington`.
- How-to: Override DocSite components — the template-compatible extension path for layout tweaks before dropping a level.

The SDK you need and the union shim

Under the Hood Why the published packages need only the stable .NET 10 SDK, when the .NET 11 preview SDK is worth opting into, and what the union shim does underneath.

Pennington's source is written in C# 15 — it uses the `union` keyword for its pipeline types. A reasonable question follows: does building a site on Pennington need the C# 15 compiler and a preview .NET SDK? For the published packages, no. The stable .NET 10 SDK is enough.

Which SDK each audience needs

Every consumer path — the DocSite template, the BlogSite template, and a host wired directly on `AddPennington` — builds against the stable .NET 10 SDK. The project file needs nothing more than `<TargetFramework>net10.0</TargetFramework>`; there is no `<LangVersion>preview</LangVersion>` to set.

The reason is that a consumer never writes the `union` keyword. Pennington's pipeline types — `ContentItem`, `ContentSource`, and the rest — are unions, but you only ever *call* methods that return them and read the case through `.Value`. Reading `.Value` is ordinary C# that compiles on .NET 10 unchanged. The preview language feature lives entirely inside Pennington's own source, not at your call sites. The first-site tutorial wires a host this way, on plain `net10.0`.

Building from source versus consuming the packages

The one place the preview SDK is still required is building Pennington itself. The library multi-targets `net10.0;net11.0`, and the `net11.0` build compiles the real `union` keyword, so the repository pins the .NET 11 preview SDK in `global.json`. That requirement belongs to the library's build, not to yours: when you reference the published `Pennington.*` packages, NuGet hands your `net10.0` project the `net10.0` build, and the preview SDK never enters the picture.

What the union shim is

Multi-targeting is what lets one source tree serve both SDKs. On `net11.0` the C# 15 `union` keyword synthesizes each pipeline union. On `net10.0`, where that keyword does not exist, a hand-written shim struct stands in — same cases, same `.Value` field, same shape at every call site. The shim is why a `net10.0` consumer sees an API identical to a `net11.0` one. Why ContentSource is a union covers why every read goes through `.Value`, and why the shim is shaped to match the keyword exactly rather than take a shortcut that would diverge between the two builds.

What the .NET 11 preview SDK buys you

Opting into the .NET 11 preview SDK and targeting `net11.0` changes one thing, and it matters to one audience: people extending the pipeline. When you branch on the `ContentItem` or `ContentSource` cases in your own code, the `net11.0` build lets you switch over the union directly, and the compiler enforces exhaustiveness — a `switch` that stops covering every case becomes a compile error that

points at exactly the code a new case broke. On `net10.0` that direct match is the preview feature you are avoiding, so you read the case through `.Value` and switch over that instead — `item.Value switch { ParsedItem p => … }` — which compiles cleanly but gives up the exhaustiveness check. Reading `.Value` is the portable form, and it is the one the templates and examples use so they build on either SDK.

Opting in is three concrete changes. Install the .NET 11 preview SDK (the version the library builds against; the repository's `global.json` records the exact preview it pins). Pin that SDK for your own project with a `global.json` of your own — `"version"` set to the installed preview, `"allowPrerelease": true` — so the build does not silently fall back to the stable SDK. Then move the project's `<TargetFramework>` from `net10.0` to `net11.0` so the `union` keyword and its exhaustive `switch` are in scope. Nothing about a host's content, layout, or wiring changes; only the SDK, the `global.json`, and the TFM do.

That safety net is the whole of the upgrade. It is invisible to anyone who is not extending the pipeline, which is why stable .NET 10 is the default the templates and tutorials assume. The tradeoff for a pipeline author — a preview SDK in exchange for a compiler guarantee — is the one worth weighing; see The content pipeline and union types for why the unions are exhaustive in the first place, and Source content from outside the markdown pipeline for the extension recipe.

Further reading

- Explanation: Why ContentSource is a union — why every consumer goes through `.Value`, and why the shim matches the keyword.
- Explanation: The content pipeline and union types — the pipeline unions and why exhaustiveness matters when you extend them.
- Tutorial: Create your first Pennington site — a consumer host wired on stable .NET 10.

CHAPTER 8

Discovery

How the search index is built and queried

Under the Hood Why Pennington ships a sharded, heading-level search index built at render time and queried entirely in the browser — and what that shape buys the reader.

Pennington has no search server. The index is a set of static JSON files generated alongside the rest of the site, and the query runs in the visitor's browser. That single constraint — no backend at query time — shapes every other decision: how the index is split, what a record represents, and how a multi-locale site keeps its indexes apart.

Context

The search engine itself is not Pennington's. The tokenizer, stemmer, inverted index, and ranking live in the external **DeweySearch** package; the browser client ships from `DeweySearch.Web` as `dewey-search.js`. Pennington's job is the adapter layer: turn rendered pages into `DeweySearch.SearchDocument` records, hand them to DeweySearch's index builder, and lay the resulting artifacts out on disk where the client can fetch them.

Keeping the engine external means Pennington never re-implements ranking, and an upgrade to DeweySearch's relevance model arrives without a Pennington change. What Pennington owns is everything domain-specific: what counts as a record, what URL a result links to, and which dimensions become filter facets.

Records are heading-level, not page-level

A naive index has one record per page. Search a thousand-word reference page and the whole page matches; the result drops the reader at the top and leaves them to scroll.

Pennington indexes at the heading instead. After a page renders, `HeadingSectionExtractor` walks the post-pipeline HTML and splits it into one section per `h2`–`h6` heading, plus a *lead* section for the text before the first heading. `SearchIndexBuilder` maps each section onto its own `SearchDocument`: the lead section carries the page's title, description, and URL; every heading section carries the heading text as its title and an anchored URL (`/page/#heading`). Each non-lead record also carries a page→heading breadcrumb trail, so the client can group results by their source page and a result deep-links to the exact section that matched.

The tradeoff is record count — a page with twelve headings produces thirteen records instead of one. That cost is paid in the index, which the visitor never downloads whole, and bought back as precision: the result is the section, not the page.

The index is sharded

A site of any size produces an index too large to ship as one file and download on the first keystroke. So the build splits it.

Each locale gets a tree under `/search/{locale}/`:

- `index.json` — the entrypoint: the document table (one row per record: URL, title, length, priority, facet ids), the facet label vocabularies, ranking statistics, and the stemmed synonym map.
- `t-*.json` — term shards. Terms are bucketed by the first few characters of their stemmed form, so a query fetches only the shards for the terms it contains.
- `f-*.json` — per-page fragments holding the indexed body text, fetched only when a page surfaces in results.

The client downloads `index.json` once, then pulls term shards and fragments on demand. Typing a query fetches a handful of small files rather than one large one; opening a result fetches that page's fragment and nothing else. The shard granularity is tunable, but the default keeps shards small enough that no single fetch dominates.

The build is a fold over the render

The index is not a second pass over the content. `SearchArtifactService` folds over the same site projection that produced every page's HTML, so each page's rendered body and heading split already exist by the time search sees them — building the index is a pure mapping from rendered page to records, not a re-render.

The same service feeds two consumers: the build-time emitter that writes the JSON files into the static output, and the dev-time middleware that serves them live. One source of truth means the index a developer queries locally is the index that ships. Because the service derives its state from content files, it is file-watched: edit a page and the index it holds is dropped and rebuilt.

Two kinds of page never reach a record. Pages marked `search: false` are excluded upstream — the content service's table-of-contents builder sets `ExcludeFromSearch`, and the fold skips them — while still rendering at their URL and appearing in the sidebar. Pages with no HTML body (endpoint and llms-only sources) have nothing to index and are skipped too.

What becomes a facet

DeweySearch's facet model is an open dictionary — any axis the host emits becomes a filterable dimension. Pennington maps three built-in axes onto it: the content *area* (the first URL segment after any locale prefix), the *section* label, and the page *tags*. A record carries an axis only when the page actually has a value for it, which is exactly how DeweySearch decides a facet exists.

Area is the only facet on by default. Areas are few and stable, so they read well as a short row of filter chips; section and tag vocabularies grow large enough to bury the filter bar, so they are opt-in. A front-matter record can also declare custom facet axes by implementing `IHasSearchFacets`; those ride

alongside the built-ins but can never overwrite `area`, `section`, or `tag`, which stay authoritative.

One index tree per locale

A multi-locale site is really several sites sharing a host, and a French query should not match English bodies. So the fold groups records by the page's locale and builds a separate DeweySearch index per group, laid out under its own `/search/{locale}/` tree. Every configured locale gets a tree even when it has no content yet — a registered-but-empty locale serves a valid entrypoint with an empty document table rather than a 404, so the client's fetch always resolves.

The browser client picks which tree to query from the first URL path segment, matched against the `data-locales` list the layout emits, falling back to the default locale. The locale routing that puts `/fr/` in front of a French page is the same signal that selects the French index — the two stay aligned without a separate configuration.

Further reading

- How-to: Tune what the search box returns — exclude pages, weight priority, scope the indexed region, add synonyms, choose facets.
- How-to: Add the search modal to a non-DocSite site — surface this index in a search UI on a bare `AddPennington` host.
- Reference: SearchIndexOptions — the knobs that shape the build.