Pennington

Guides

Version 1.0.0

Version 1.0.0
Generated July 2026
https://usepennington.net/
Produced with Pennington 0.1.6-alpha.0.14

Contents

CHAPTER 1

Pages

Define custom front-matter keys

Guides Declare a record implementing IFrontMatter with extra YAML keys and register it through AddMarkdownContent so a markdown source deserializes into the custom type.

To parse YAML keys the shipped front-matter records do not expose — a `namespace`, a `stability` badge, a `productName` — declare a custom `record` implementing `IFrontMatter` and the capability interfaces relevant to the keys, then register it as a markdown source with `AddMarkdownContent<T>`. For the full catalog of built-in keys, see Front matter key reference; for the design rationale behind the capability interfaces, see The front-matter capability system.

The recipe declares and registers the record in `examples/DocSiteKitchenSinkExample`, which adds `namespace` and `stability` keys on top of the built-in front-matter records, then reads those keys from a regular Razor page in `examples/CustomFrontMatterRazorPageExample`.

Before you begin

- An existing Pennington site with markdown content under a `Content/` folder (see Create your first Pennington site if not).
- A bare `AddPennington` host, or an existing `AddDocSite` / `AddBlogSite` host with room for an additional markdown source. `AddBlogSite` registers one source against `BlogSiteFrontMatter`; `AddDocSite` registers two sources (`DocSiteFrontMatter` and `BlogPostFrontMatter`). Adding a third custom-record source on top of the template is done by chaining another `AddMarkdownContent<T>()` call after `AddDocSite` / `AddBlogSite`, or by falling back to bare `AddPennington` (see Serve docs and a blog from separate content roots).

Declare the record

Implement Pennington.FrontMatter.IFrontMatter as a `record` and add only the capability interfaces the new keys need — `ITaggable`, `IOrderable`, `ISectionable`, `IRedirectable`. Unimplemented capabilities pick up their default-member values, so a minimal record is short.

CSHARP

```
namespace DocSiteKitchenSinkExample;

using Pennington.FrontMatter;

/// <summary>
/// Custom front-matter record used by the "multiple content sources" how-to.
/// Implements the same capability interfaces as <c>DocSiteFrontMatter</c>
/// plus an API-specific <see cref="Namespace"/> and <see cref="Stability"/>
/// pair so reference pages can expose a per-API namespace and stability
/// badge.
/// </summary>
/// <remarks>
/// Kept as a standalone record so tutorials can target it with
/// <c>T:DocSiteKitchenSinkExample.ApiFrontMatter</c>. Declaring a record
/// that implements <see cref="IFrontMatter"/> with a small handful of
/// capability interfaces is the canonical "write your own front matter"
/// pattern referenced by the front-matter how-to.
/// </remarks>
public record ApiFrontMatter : IFrontMatter, ITaggable, ISectionable, IOrderable,
IRedirectable
{
    public string Title { get; init; } = "";
    public string? Description { get; init; }
    public bool IsDraft { get; init; }
    public string[] Tags { get; init; } = [];
    public int Order { get; init; } = int.MaxValue;
    public string? RedirectUrl { get; init; }
    public string? SectionLabel { get; init; }
    public string? Uid { get; init; }
    public bool Search { get; init; } = true;
    public bool Llms { get; init; } = true;

    /// <summary>API namespace (e.g. <c>Pennington.Highlighting</c>).</summary>
    public string? Namespace { get; init; }

    /// <summary>Stability classification — <c>stable</c>, <c>preview</c>, or
<c>experimental</c>.</summary>
    public string Stability { get; init; } = "stable";
}
```

Property names map to YAML keys under `CamelCaseNamingConvention` — `Namespace` reads `namespace:`; `Stability` reads `stability:`. Unknown keys are dropped with a warning in lenient mode (dev) and rejected in strict mode (the build default), so a typo on a custom key is flagged — as a dev warning or a build failure — rather than silently taking effect as a default.

Register the record

Pass the record type to `AddMarkdownContent<T>` so the pipeline deserializes the YAML into that type. The configure delegate selects the content root the source reads from and the URL prefix its pages serve under. On a bare host this call goes inside the `AddPennington` lambda; on a DocSite or BlogSite host,

chain it through the `ConfigurePennington` escape hatch so the extra source sits alongside the template's own. The kitchen-sink example registers its `ApiFrontMatter` source this way:

CSHARP

```
// DocSite's default source serves all of Content/ at /. Carve out the
// symbols subtree so the custom-typed source below owns it without an
// overlap warning.
penn.MarkdownSources[0].ExcludePaths = ["symbols"];

penn.AddMarkdownContent<ApiFrontMatter>(o =>
{
    o.ContentPath = "Content/symbols";
    o.BasePageUrl = "/symbols";
    o.SectionLabel = "Symbols";
});
```

`ExcludePaths` on the template's own doc source carves the subtree out so exactly one source owns those pages — drop that line on a bare `AddPennington` host where no template source claims the folder.

Read the key in a Razor page

The lede promised a `stability` value — here is what consumes it. The `ApiFrontMatter` record is portable across hosts; only its registration differs, so this section uses a bare `AddPennington` host where a Razor page owns rendering. A page under the registered source authors the custom keys at the top of its YAML block:

YAML

```
---
title: "Highlighting service"
namespace: "Pennington.Highlighting"
stability: "preview"
---
```

A regular Razor `@page` reads those keys by injecting `IPageResolver` and calling the generic `ResolveAsync<ApiFrontMatter>`. That overload resolves the requested URL and hands back the front matter already typed as the custom record, so `Stability` and `Namespace` are plain property reads — no cast. It returns `null` when nothing matches or the page is served by a source registered against a different front-matter type:

RAZOR

```
@* Catch-all that resolves a URL to a rendered page and reads the custom front
   matter directly. ResolveAsync<ApiFrontMatter> hands back the front matter
   already typed as the custom record, so Stability and Namespace are plain
   property reads — no cast, no MdazorContext dictionary. *@

@page "/{*Path}"
@inject IPageResolver Resolver

@if (_html is not null)
{
    <PageTitle>@_title</PageTitle>
    <article>
        <h1>@_title</h1>
        @if (_stability is not null)
        {
            <p>Stability: <strong>@_stability</strong> · Namespace: <code>@_namespace</code>
</p>
        }
        @((MarkupString)_html)
    </article>
}
else
{
    <PageTitle>Not found</PageTitle>
    <p>No content matches @Path.</p>
}

@code {
    [Parameter] public string? Path { get; set; }

    private string? _title;
    private string? _html;
    private string? _stability;
    private string? _namespace;

    protected override async Task OnInitializedAsync()
    {
        var requested = new UrlPath(Path ?? string.Empty).EnsureLeadingSlash();

        // The generic overload resolves the page and narrows its front matter to
        // ApiFrontMatter in one step. It returns null when nothing matches or the
        // page is served by a source registered against a different type.
        if (await Resolver.ResolveAsync<ApiFrontMatter>(requested) is { } page)
        {
            _title = page.Metadata.Title;
            _stability = page.Metadata.Stability;
            _namespace = page.Metadata.Namespace;
            _html = page.Content.Html;
        }
    }
}
```

Any page served by the `ApiFrontMatter` source now surfaces its typed keys — the round-trip from YAML to a strongly-typed Razor page.

Verify

- In `examples/CustomFrontMatterRazorPageExample`, run `dotnet run` and visit `/symbols/highlighting-service/`. The page renders `Stability: preview` from that page's `stability:` key — proof the YAML deserialized into the typed `ApiFrontMatter.Stability` property and `ResolveAsync<ApiFrontMatter>` returned the typed page.
- The build report contains no `FrontMatterParseError` diagnostics for pages under the new source.

Related

- Reference: Front matter key reference — every built-in key, type, and default
- Reference: Built-in front-matter types — `DocFrontMatter`, `BlogFrontMatter`, `DocSiteFrontMatter`, `BlogSiteFrontMatter`
- Reference: IFrontMatter and capability defaults — the capability interfaces available to a custom record
- Background: The front-matter capability system — why the design collapsed ten interfaces into default members
- How-to: Use multiple content sources — chain a second `AddMarkdownContent<T>` against a custom record

Mark drafts, schedule posts, tag pages, and control sort order

Guides Hide unfinished pages, embargo posts until a release date, attach grouping keywords, and choose where a page lands in its sidebar section using front-matter keys.

To keep an unfinished page out of navigation, embargo a post until a release date, attach grouping keywords to a page, or change where a page appears within its sidebar section, set one of four front-matter keys.

A draft is excluded from build output. `isDraft: true` (and an unreached future `date:`) keeps the page visible under `dotnet run` for preview, but `dotnet run -- build` drops it from the static site entirely — no HTML file, and any `xref:` link targeting it dangles in the published build. Drafts are a "not yet shipping" flag, not a "shipped but unlisted" one. For the canonical key catalog and parse rules, see Front matter key reference.

Before you begin

- A working Pennington site has markdown under `Content/` (see Create your first Pennington site if not).

- Each key needs a front-matter record that implements its backing capability — `tags:` requires `ITaggable`, `order:` requires `IOrderable`; `isDraft:` and `date:` are universal. The shipped records implement different subsets, so check the per-record matrix in Front matter key reference before reaching for a key.
- The sidebar currently renders in file-order; `TableOfContentsNavigation` has not been customized.

Setting `order:` on a `BlogSiteFrontMatter` or `BlogFrontMatter` page has no effect — blog posts sort newest-first by `date:`. To reorder posts, adjust the date; to hide a post, use `isDraft: true`.

Options

Hide an unfinished page with `isDraft: true`

Setting `isDraft: true` drops the page from navigation, search, and `llms.txt`. Under `dotnet run` it stays served and `xref:` links to it resolve, so you can preview it; under `dotnet run -- build` it is excluded from output entirely.

YAML

```
---
title: Coming soon
isDraft: true
---
```

The default is `false`. For the full key catalog, see Front matter key reference.

Schedule a post for the future with `date:`

A page whose `date:` is later than the build-machine wall clock is treated the same as a draft: visible in `dotnet run` so you can preview, excluded from `dotnet run -- build` output, feeds, and search. As soon as the clock crosses the date, the next build picks the page up — no flag flip required.

YAML

```
---
title: New feature announcement
date: 2030-11-14T09:00:00
---
```

The comparison uses the build server's local wall clock, so a date without a time component (`date: 2030-11-14`) releases at local midnight. CI that runs hourly will pick the post up on the first build after that boundary.

To override the wall clock — for tests, for re-running yesterday's build, or for previewing tomorrow's release — replace the registered `TimeProvider` in DI with a fixed one (the `Microsoft.Extensions.TimeProvider.Testing` package ships `FakeTimeProvider` for exactly this).

Tag a page for grouping

`tags:` accepts a string array read through `ITaggable` into `RenderedContent.Tags`, making it available to client-side filtering widgets. Tags do not produce `/tags/<name>` index pages on their own; to generate browse-by-tag pages from a tagged front-matter record, register a taxonomy — see Build browse-by-pages with AddTaxonomy.

YAML

```
---
title: Deep dive
tags: [advanced, performance, pipeline]
---
```

Order a page inside its section

Lower `order:` values sort earlier within a section. Spacing like 10/20/30 leaves room for later inserts between existing siblings.

YAML

```
---
title: Install
order: 20
---
```

`order:` positions a page among its siblings; it does not position the *folder* that holds it. To set a section's own sort key — rather than letting it inherit the min-of-children value — drop a `_meta.yml` sidecar in the folder with its own `order` key; see Folder sidecar (`_meta.yml`). For how the navigation tree assembles those values, see Why the sidebar mirrors your folders.

Verify

- Run `dotnet run` — the drafted page's URL still responds 200 but is absent from the sidebar and from the per-locale search index under `/search/{locale}/` (for example `/search/en/index.json`)
- A page with a future `date:` behaves the same way: URL responds in dev, absent from sidebar/search/RSS; `dotnet run -- build` lists it under "Skipped"
- The tagged page's HTML carries the tag strings in its rendered output (inspect `RenderedContent.Tags` or the page body)
- Sidebar entries within the section appear in ascending `order:` — swap two values and the order flips on next reload

Related

- Reference: Front matter key reference
- How-to: Reorder, rename, or hide entries in the sidebar

- How-to: Build browse-by- pages with AddTaxonomy
- Background: The front-matter capability system

Add images and shared assets to a page

Guides Colocate page-specific images next to their markdown, and put shared images in wwwroot for one canonical URL.

To keep an image next to the markdown that references it, drop it into a folder alongside the page under `Content/`. When the same file is needed from multiple pages — a logo, a cover photo, a shared diagram — put it in `wwwroot/` instead so it has one canonical URL.

Before you begin

- An existing Pennington site with at least one markdown page (see Serve markdown through Blazor Pages if not).
- The target markdown file is known.
- The project has a `wwwroot/` folder for shared static files (the default for Web SDK projects).

Where to put images

Colocated next to the markdown file

Drop the image into a folder alongside the page — typically an `assets/` subfolder. Pennington copies every non-markdown file under `Content/` to the same relative path in the output, so the image ships with the page automatically. Reference it with a relative path:

MARKDOWN

```
![Alt text](./assets/colocated.png)
```

The link resolves correctly whether the page is served at `/main/images-and-assets/` or under a locale prefix.

Shared in `wwwroot/`

When the same image is referenced from multiple pages, drop it into `wwwroot/` so it has one canonical URL. The ASP.NET host serves `wwwroot/` as static web root by default, so `wwwroot/shared.png` is served at `/shared.png`. Reference it with a leading slash:

MARKDOWN

```
![Alt text](/shared.png)
```

When deploying under a sub-path, `BaseUrlHtmlRewriter` prepends the base URL at response time — avoid hard-coding the sub-path. See Host under a sub-path (base URL).

Verify

- Run `dotnet run` and open the page that references the colocated image — the image renders inline.
- Visit the shared-asset URL directly (for example, `/shared.png`) — the file loads.
- Run `dotnet run -- build` and confirm both files appear in `output/` at their original relative paths.

Related

- Pennington.Infrastructure.MarkdownContentOptions — `ContentPath`, `ExcludePaths`, `FilePattern`. Use `ExcludePaths` to keep a subtree of `Content/` out of the output.
- Front matter key reference — when an image is set as a social card or cover image via front matter.
- Dev mode and build mode share one code path — how `GetContentToCopyAsync` fits the content pipeline.

Forward visitors from a renamed page

Guides Set redirectUrl in front matter to forward visitors from the old path to the new one — an HTTP 301 on the live server, plus a meta-refresh stub in the static build.

When a published page is renamed or deleted, set `redirectUrl:` in its front matter to forward visitors to the new URL. On the dev and self-hosted server, Pennington issues a real HTTP 301 from the old path; the page body is not rendered or indexed. The static build also writes a `<meta http-equiv="refresh">` stub at the old path, since a static host can't issue a server-side 301 without its own redirect config.

When the page is *deleted* rather than renamed, keep the file as a front-matter-only stub at the old path — `title:` and `redirectUrl:` with no body — so the old URL still resolves to a redirect. Delete the file and the old URL 404s.

`redirectUrl:` accepts an external absolute URL (`https://…`) as well as an internal path; the value is emitted verbatim as the 301 `Location` and the meta-refresh target, so a cross-site redirect works the same way.

For batch redirects across many paths, configure them at the hosting layer instead — Nginx or IIS rules (see Self-host behind Nginx or IIS) or a Netlify/Cloudflare/Azure rules file (see Adapt the deploy workflow for other hosts).

Before you begin

- An existing Pennington site using `AddDocSite` (see Scaffold a documentation site with DocSite if not) or another host whose front-matter type implements `IRedirectable`. `DocSiteFrontMatter` and `BlogSiteFrontMatter` both do. For a custom record, add the interface — see The front-matter capability system.
- Both the old URL (the page being retired) and the new URL (the canonical destination) are known.

To copy a working setup, see examples/DocSiteKitchenSinkExample, whose `Content/main/redirect-source.md` is a complete redirect page.

Add `redirectUrl:` to the old page

Open the markdown file at the old URL and set `redirectUrl:` to the new absolute path. Keep `title:` so diagnostics stay readable; the body does not render.

MARKDOWN

```
---
title: Old page URL
description: Redirects away to the new location.
redirectUrl: /main/front-matter/
order: 200
uid: kitchen-sink.main.redirect-source
---

A visit to this URL is intercepted by the Pennington redirect middleware
and returns HTTP 301 with a meta-refresh body pointing at `redirectUrl`.
The static build captures the same 301 and writes the meta-refresh file
to disk at this page's output path — one code path for dev and publish.
```

Verify

- Run `dotnet run` and visit the old URL: the page redirects immediately to the target set in `redirectUrl`.
- View source on the old URL: the markup contains `<meta http-equiv="refresh" content="0;url=...">` and a `<link rel="canonical" href="...">` pointing at the redirect target.
- Check `/sitemap.xml` and `/llms.txt`: the old URL does not appear. A redirect has no canonical HTML page, so it is excluded from both.

Related

- Reference: Front matter key reference — the row for `redirectUrl` (type, default, which records support it).
- Reference: IFrontMatter and capability defaults — how `IRedirectable` fits alongside the other capability interfaces.
- Background: The front-matter capability system — why `IRedirectable` stayed a separate capability instead of collapsing into `IFrontMatter`.

Reuse one snippet across many pages

Guides Pull a shared Markdown partial into a page with the DocFX-style [!INCLUDE] directive — block or inline — instead of copy-pasting.

When the same prerequisite note, install snippet, or boilerplate paragraph belongs on several pages, write it once and pull it in with an `[!INCLUDE]` directive. Pennington resolves the directive while parsing — the host page renders as if the partial's text were typed inline.

Before you begin

- An existing Pennington site renders Markdown (see Create your first Pennington site if not).
- A folder for partials that sits **outside** any content root, so the partials are not discovered as standalone pages. This site keeps them in `_includes/` next to `Content/`.
- The pipeline was built through `AddPennington` / `AddDocSite` / `AddBlogSite`; include expansion runs in the Markdown parser with no extra wiring.

Block include

To drop a whole partial in as its own block — paragraphs, lists, callouts, code — put the directive on its own line. The path is relative to the page that references it.

MARKDOWN

```
[!INCLUDE [.NET prerequisite](../../../_includes/dotnet-preview-note.md)]
```

The partial `_includes/dotnet-preview-note.md` holds a single `[!NOTE]` callout. It expands here, and the alert syntax inside it passes through the normal pipeline — note that the partial's leading YAML front matter, if any, is stripped on include:

> **Note**
>
> Pennington targets .NET 10. Confirm your SDK with `dotnet --list-sdks` — at least one `10.0.*` entry must be present before the host builds.

The same partial expands the same way on every page that references it — edit `_includes/dotnet-preview-note.md` once and every host page picks up the change on the next build.

Inline include

To splice a partial into the middle of a sentence, put the directive inline. The partial should hold a single run of text with no trailing newline.

MARKDOWN

```
The current target SDK is [!INCLUDE [version](../../../_includes/sdk-version.md)], the
stable release.
```

The current target SDK is 10.0.100, the stable release.

How paths resolve

The path in the directive is resolved relative to the **referencing file**, not the content root — `../` climbs out of the page's folder. Includes nest: a partial may itself contain `[!INCLUDE]` directives, resolved relative to that partial in turn.

Expansion is plain text substitution: the partial's content replaces the directive in place, then the combined Markdown parses as one document. Two consequences follow from that, plus two rules that keep the feature predictable:

- **A block partial may hold many blocks** — paragraphs, lists, callouts, fences. Because the directive sits on its own line with blank lines around it, each block splices in as its own block. A trailing newline at the end of the partial is harmless here.
- **An inline partial must be a single run of text with no trailing newline.** A trailing newline splices a line break into the middle of the host sentence, ending the paragraph early; `_includes/sdk-version.md` is a single line (`10.0.100`) with no terminating newline for exactly this reason.
- A directive inside a fenced code block is left verbatim, so this page can show the syntax without expanding it.
- Relative links and images inside a partial are **not** rebased — they resolve as if written in the host page. Keep partials free of relative links, or use site-absolute paths.

When a partial is missing

A target that does not exist collapses to an HTML comment instead of failing the build:

HTML

```
<!-- Pennington: include not found: ../../../_includes/typo.md -->
```

View source on a page during `dotnet run` to spot a mistyped path — the comment names the directive that could not be resolved. Only local relative paths are spliced: an `[!INCLUDE …](https://…)` directive is also skipped, emitting `<!-- Pennington: include skipped (not a local file): … -->`. An include cycle is broken the same way, emitting `<!-- Pennington: include cycle broken: … -->`.

A missing include is the one failure mode you have to catch yourself. Neither the build report nor `dotnet run -- diag warnings` flags an unresolved directive — expansion happens inside the Markdown parser, which swallows the failure into the HTML comment and moves on. A typo'd path drops its content from the published site silently, so inspect the rendered output rather than trusting a clean build.

Verify

- Run `dotnet run`, open a host page, and confirm the partial's text appears where the directive sat — block partials as their own blocks, an inline partial spliced into the sentence with no stray line break.
- Edit the partial (for example, change `_includes/sdk-version.md`) and reload every host page — each one reflects the new text on the next build.

- Mistype a path on purpose and view source: the body carries `<!-- Pennington: include not found: ... -->` and that block is empty. Search the rendered HTML for `Pennington: include` to sweep a page for skipped, missing, or cyclic directives — the build itself reports nothing.

Related

- Reference: Markdown extensions catalog — include directives alongside every other non-CommonMark feature
- How-to: Tab platform or language variants together — tab a partial's worth of content per platform or language
- How-to: Define custom front-matter keys — the front-matter keys a host page still needs (a partial's own front matter is stripped on include)

Provide a 404 page

Guides Author the not-found body with a content-root 404.md (or a NotFound.razor component); the static build writes it to output/404.html for your host to serve.

A static host serves a single `404.html` for any URL it can't find. You supply that page's body with one file — no server-side code, no error-handling route. Pennington's build renders it and writes `output/404.html` for you.

Add a `404.md`

Drop a `404.md` at your content root. Give it a title and a short body, and point readers somewhere useful.

MARKDOWN

```
---
title: Page not found
description: The page you were looking for doesn't exist.
---

We couldn't find that page. It may have moved, or the link that brought you here is out of
date.

Head back to the [home page](/) to pick up where you left off.
```

Run `dotnet run -- build` and the file lands at `output/404.html`. The file is reserved: there is no `/404/` route, and it never appears in navigation, the sitemap, search, or `llms.txt`. BlogSite works the same way — put `404.md` at the content root (outside the blog folder) and it becomes the site's not-found body.

> **Tip**
>
> Don't gold-plate the 404. On a static host a reader reaches it only by following a dead link or mistyping a URL, and your host serves the same `404.html` for every miss. A title, a sentence, and a link home is plenty. You'll see it often in development; your readers almost never will.

Why there's no `/404/` route

A routable `/404/` would be a valid route whose job is to announce an invalid destination, and nothing runs on a static host to choose it; instead the body renders at the catch-all and reaches readers only through your host's `404.html` mapping. For how the build materializes that file, see Dev mode and build mode share one code path.

Use a Razor component instead

When you want components or richer markup, add a `NotFound.razor` (no `@page` directive). The catch-all finds it by name and renders it for any unmatched URL.

RAZOR

```
@namespace DocSiteChromeOverridesExample.Components
@using Microsoft.AspNetCore.Components.Web

@* A component named NotFound with no @page directive. DocSite's catch-all finds it by
   reflection and renders it for any unmatched URL when no Content/404.md is present —
   so it is the not-found body, never a route. Unlike ExtraPage.razor it needs no
   AdditionalRoutingAssemblies wiring; the reflection scan walks every loaded assembly. *@

<PageTitle>Page not found</PageTitle>

<article class="prose mx-auto py-12">
    <h1>This page wandered off</h1>
    <p>
        We couldn't find what you were looking for. Try the
        <a href="/">home page</a> or one of the guides in the sidebar.
    </p>
</article>
```

If both a `404.md` and a `NotFound.razor` exist, the markdown file wins. With neither, Pennington renders a built-in localized message, so every site still produces a valid `404.html`.

Make your host serve it

Producing `404.html` is half the job — your host has to return it for unknown URLs. The mapping differs per host:

- GitHub Pages serves a root `404.html` automatically.

- Other managed hosts (Netlify, Cloudflare Pages, Azure Static Web Apps) need a fallback rule in their config.
- Nginx or IIS need an `error_page` / fallback directive.

Verify

- Run `dotnet run -- build` and confirm `output/404.html` contains your content.
- Run `dotnet run`, visit a URL that doesn't exist, and confirm you see the body with an HTTP 404 status (`curl -I http://localhost:5000/nope`).

Related

- How-to: Build a static site — where `output/404.html` comes from in the build.
- Background: Dev mode and build mode share one code path — how the crawler materializes `404.html`.
- Background: The response pipeline — how the rendered 404 body keeps an HTTP 404 status.

CHAPTER 2

Code Samples

Annotate specific lines in a code block

Guides Apply highlight, diff, focus, and error/warning line classes to fenced code with trailing `[!code ...]` comment directives.

To call out specific lines in a fenced code block — emphasizing a change, diffing before/after, focusing attention, or surfacing a diagnostic — append a trailing `[!code ...]` comment directive to the line. Pennington promotes the directive onto the enclosing line and strips the comment, so the rendered code stays clean and the called-out line picks up a CSS class you can style.

Before you begin

- An existing Pennington site renders markdown with highlighted code fences (see Create your first Pennington site if not).
- The fenced language supports one of the recognized comment markers — `//`, `#`, `--`, `<!-- -->`, `*`, `%`, `'`, `REM`, `;`, or `/* */`.

Annotation directives

Each H3 below shows the source markdown above the rendered fence. Swap the comment marker to match the fenced language (`//` for C#/JS, `#` for YAML/Python, `--` for SQL, `<!-- -->` for HTML).

Highlight a single line

Append `// [!code highlight]` to the line.

MARKDOWN

````
```csharp
public int Add(int a, int b)
{
    return a + b; // [!code highlight]
}
```
````

CSHARP

```csharp
public int Add(int a, int b)
{
    return a + b;
}
```

Mark added or removed lines

Use `[!code ++]` for added lines and `[!code --]` for removed lines.

MARKDOWN

````
```csharp
public int Multiply(int a, int b) // [!code ++]
{
    return a * b; // [!code ++]
}
public int OldWay(int a, int b) // [!code --]
{
    return a + b; // [!code --]
}
```
````

CSHARP

```csharp
public int Multiply(int a, int b)
{
    return a * b;
}
public int OldWay(int a, int b)
{
    return a + b;
}
```

Focus one line and blur the rest

Add `[!code focus]` to the line (or lines) worth zeroing in on. Every other line in the fence blurs back so the focused line stands out.

MARKDOWN

````
```csharp
var config = new Config(); // [!code focus]
config.Apply();
config.Save();
```
````

CSHARP

```csharp
var config = new Config();
config.Apply();
config.Save();
```

Flag errors and warnings

Use `[!code error]` and `[!code warning]` to show diagnostics inline. The rendered lines read like compiler output.

MARKDOWN

```
```csharp
var path = null; // [!code error]
var length = path.Length; // [!code warning]
```
```

CSHARP

```
var path = null;
var length = path.Length;
```

Style the annotation classes

Each directive promotes a CSS class onto the enclosing `.line` span — `highlight`, `diff-add`, `diff-remove`, `focused`, `blurred`, `error`, or `warning` — and a fence with any focused line also carries `has-focused` on its `<pre>`. On a site that uses `Pennington.MonorailCss`, these classes are styled out of the box: the highlight tint, diff gutter signs, blur effect, and diagnostic backgrounds all render with no extra work. A host that ships its own CSS instead must define each class itself — without those rules the directives still resolve and strip the comments, but the lines render unstyled.

Verify

- Run `dotnet run` and load the page with the annotated fence. The highlighted line renders with a tinted background, diff lines show `+` / `-` gutters, the focused line stays sharp while the rest blur, and the `[!code …]` comments are gone.
- View source: each annotated `.line` span carries the matching class, and a fence with any focused line carries `has-focused` on its `<pre>`.

Related

- Reference: Highlighting interfaces — `ICodeHighlighter`, `ICodeBlockPreprocessor`, `HighlightingService`, and `TextMateLanguageRegistry`
- How-to: Register a code-block preprocessor — when a trailing-comment directive isn't enough and fence bodies need transformation
- Background: The syntax-highlighting cascade — why directive promotion runs after the highlighter and where custom highlighters plug in

Group adjacent code fences into a tabbed sample

Guides Collapse adjacent fenced code blocks into one tabbed widget and customize the rendered CSS class names.

When two or more code variants show the same operation — bash vs. PowerShell, a `csproj` property vs. its CLI equivalent, C# vs. F# — a tabbed group lets the audience pick one without scrolling past the others. Author each variant as a normal fenced code block with `tabs=true title="..."` in the info

string and Pennington collapses adjacent matches into a single ARIA tablist. For the info-string grammar, see Code-block argument reference.

Before you begin

- An existing Pennington site rendering markdown (see Create your first Pennington site if not).
- The host wires the default Pennington markdown pipeline, which already enables `UseTabbedCodeBlocks` under `AddDocSite`, `AddBlogSite`, or bare `AddPennington`.
- Familiarity with the fence info-string shape (language token plus key/value attributes) — the reference page above covers the grammar.

Tabs and labels

Each H3 below shows the source markdown above its rendered result. In the first, adjacent fences collapse into one tablist — the first tab is active by default, and switching tabs reveals the matching panel. In the second, intervening prose splits the fences into two separate widgets.

Adjacent fences become tabs

Author two or more fenced blocks back-to-back, each with `tabs=true title="..."` in the info string. Consecutive matches collapse into one tablist; the `title` value becomes the tab label.

MARKDOWN

````
```bash tabs=true title="bash"
dotnet add package Pennington
```

```powershell tabs=true title="PowerShell"
Install-Package Pennington
```
````

bash

BASH

```
dotnet add package Pennington
```

PowerShell

POWERSHELL

```
Install-Package Pennington
```

Prose between fences splits the group

The grouping logic only collapses fences that sit next to each other in the block stream. A paragraph, heading, or blank-lined HTML element between two fences splits the group into two separate tablists. To keep one widget, remove the intervening block.

MARKDOWN

````
````bash tabs=true title="bash"
echo "first group"
````

A paragraph here ends the first tablist.

````bash tabs=true title="bash"
echo "second group"
````
````

BASH

```
echo "first group"
```

A paragraph here ends the first tablist.

BASH

```
echo "second group"
```

Verify

- Run `dotnet run` and load the page with the back-to-back fences. They render as one widget with a tab per `title`, and clicking a tab swaps the visible panel.
- View source: the group is a single `<div>` carrying the tablist classes, with one `<button role="tab">` per fence and one panel each. A fence split by intervening prose produces a second, separate `<div>`.

Customize the tab CSS classes

The rendered HTML draws its CSS class names from `TabbedCodeBlockRenderOptions`. The `Default` instance ships with `not-prose` on the outer wrapper plus `tab-container`, `tab-list`, `tab-button`, and `tab-panel` on the nested elements — enough for the MonorailCSS preset to style them without extra work.

To override the class names, set `PenningtonOptions.TabbedCodeBlockOptions` to a `Func<TabbedCodeBlockRenderOptions>` returning a modified `with` expression.

CSHARP

```
penn.TabbedCodeBlockOptions = () => TabbedCodeBlockRenderOptions.Default with
{
    OuterWrapperCss = "not-prose",
    ContainerCss = "lab-tabs",
    TabListCss = "lab-tabs-list",
    TabButtonCss = "lab-tabs-button",
    TabPanelCss = "lab-tabs-panel",
};
```

See Markdown extensions catalog for the full `TabbedCodeBlockRenderOptions` surface.

Related

- Reference: Markdown extensions catalog — the tabs extension alongside every other non-CommonMark feature
- Reference: Code-block argument reference — the info-string grammar that carries `tabs=true` `title="..."`
- Reference: Content components — the Pennington.UI `<Tabs>` component for non-code tabsets
- Background: Dev mode and build mode share one code path — why the render options flow through one pipeline in both modes

Embed focused code samples

Guides Scope :symbol fences to one member, strip declaration noise with bodyonly, and refactor long methods into named helpers so each section of a walkthrough shows one idea.

To limit a code fence to the one member a walkthrough discusses — rather than dumping the whole enclosing file with every sibling member — use the `:symbol` preprocessor's member-scoped form. The recipes below scope a fence to one member, strip declaration noise with `,bodyonly`, carry the file's imports with `,imports`, walk a multi-phase method through named helpers, outline a type's shape with `,signatures`, diff two implementations with `symbol-diff`, and embed a whole file with a bare path. Address a member by its **name path** (`Type.Member`) rather than a hard-coded line range — a name path survives the line shifts that silently break a range. For the fence grammar itself, see Code-block argument reference.

Before you begin

- An existing Pennington site (see Create your first Pennington site if not), with `Pennington.TreeSitter` wired through `AddTreeSitter` and `ContentRoot` pointing at the root that holds the source to fence.
- Comfort authoring markdown code fences — the techniques on this page are all info-string changes on a `csharp:symbol` fence.

For a working setup, see examples/FocusedCodeSamplesExample. `MonolithWordCounter` carries one long `CountWords` method; `ModularWordCounter` splits the same logic into `Tokenize` , `Tally` , and `Format` . Both are referenced by the fences below.

Fence one member, not the whole type

When the surrounding prose is about one method, reach for `Type.Method` instead of a bare `Type` . A member path shrinks the fence to the member the reader cares about; a `Type` reference (or a bare file path with no `>`) pulls the full type or file.

The wide form, which lands on a page that only discusses `CountWords`:

MARKDOWN

````
```csharp:symbol
examples/FocusedCodeSamplesExample/MonolithWordCounter.cs > MonolithWordCounter
```
````

The narrow form, scoped to the method under discussion:

MARKDOWN

````
```csharp:symbol
examples/FocusedCodeSamplesExample/MonolithWordCounter.cs > MonolithWordCounter.CountWords
```
````

Which renders as:

CSHARP

```
public static string CountWords(string text, int topN)
{
    // Tokenize: split on whitespace, lowercase, strip surrounding punctuation.
    var words = new List<string>();
    foreach (var raw in text.Split([' ', '\t', '\n', '\r'],
StringSplitOptions.RemoveEmptyEntries))
    {
        var word = raw.Trim('.', ',', '!', '?', ';', ':', '"', '\'').ToLowerInvariant();
        if (word.Length > 0)
        {
            words.Add(word);
        }
    }

    // Tally: count occurrences and rank by frequency desc, then alphabetically.
    var counts = new Dictionary<string, int>();
    foreach (var w in words)
    {
        counts[w] = counts.GetValueOrDefault(w, 0) + 1;
    }
    var ranked = counts
        .OrderByDescending(kv => kv.Value)
        .ThenBy(kv => kv.Key)
        .Take(topN)
        .ToList();

    // Format: header line plus one word-count row per entry.
    var sb = new StringBuilder();
    sb.AppendLine($"Top {ranked.Count} words:");
    foreach (var kv in ranked)
    {
        sb.Append(kv.Key.PadRight(12));
        sb.Append(' ');
        sb.AppendLine(kv.Value.ToString());
    }
    return sb.ToString();
}
```

The name-path grammar — `Type`, `Type.Member`, nested `Type.Inner.Member` — is listed in Code-block argument reference.

Strip declaration noise with `,bodyonly`

Even a member-scoped fence still carries the signature and any leading doc comment. When the prose has already named the method and summarized what it does, both are redundant. Appending `,bodyonly` renders only the body between the braces.

MARKDOWN

````
```csharp:symbol,bodyonly
examples/FocusedCodeSamplesExample/MonolithWordCounter.cs > MonolithWordCounter.CountWords
```
````

Which renders as:

CSHARP

```csharp
// Tokenize: split on whitespace, lowercase, strip surrounding punctuation.
var words = new List<string>();
foreach (var raw in text.Split([' ', '\t', '\n', '\r'],
StringSplitOptions.RemoveEmptyEntries))
{
    var word = raw.Trim('.', ',', '!', '?', ';', ':', '"', '\'').ToLowerInvariant();
    if (word.Length > 0)
    {
        words.Add(word);
    }
}

// Tally: count occurrences and rank by frequency desc, then alphabetically.
var counts = new Dictionary<string, int>();
foreach (var w in words)
{
    counts[w] = counts.GetValueOrDefault(w, 0) + 1;
}
var ranked = counts
    .OrderByDescending(kv => kv.Value)
    .ThenBy(kv => kv.Key)
    .Take(topN)
    .ToList();

// Format: header line plus one word-count row per entry.
var sb = new StringBuilder();
sb.AppendLine($"Top {ranked.Count} words:");
foreach (var kv in ranked)
{
    sb.Append(kv.Key.PadRight(12));
    sb.Append(' ');
    sb.AppendLine(kv.Value.ToString());
}
return sb.ToString();
```

`,bodyonly` also works on types (members between the braces, skipping the type header) and properties (the `get` / `set` accessors).

Carry the imports with `,imports`

A member-scoped fence drops the file's `using` / `import` / `require` lines, so a reader can't see where the types resolve from. Append `,imports` to prepend the file's top-of-file imports above the snippet, separated by a blank line.

MARKDOWN

````
```csharp:symbol,imports
examples/FocusedCodeSamplesExample/MonolithWordCounter.cs > MonolithWordCounter.CountWords
```
````

Which renders as:

CSHARP

```csharp
using System.Text;

public static string CountWords(string text, int topN)
{
    // Tokenize: split on whitespace, lowercase, strip surrounding punctuation.
    var words = new List<string>();
    foreach (var raw in text.Split([' ', '\t', '\n', '\r'],
StringSplitOptions.RemoveEmptyEntries))
    {
        var word = raw.Trim('.', ',', '!', '?', ';', ':', '"', '\'').ToLowerInvariant();
        if (word.Length > 0)
        {
            words.Add(word);
        }
    }

    // Tally: count occurrences and rank by frequency desc, then alphabetically.
    var counts = new Dictionary<string, int>();
    foreach (var w in words)
    {
        counts[w] = counts.GetValueOrDefault(w, 0) + 1;
    }
    var ranked = counts
        .OrderByDescending(kv => kv.Value)
        .ThenBy(kv => kv.Key)
        .Take(topN)
        .ToList();

    // Format: header line plus one word-count row per entry.
    var sb = new StringBuilder();
    sb.AppendLine($"Top {ranked.Count} words:");
    foreach (var kv in ranked)
    {
        sb.Append(kv.Key.PadRight(12));
        sb.Append(' ');
        sb.AppendLine(kv.Value.ToString());
    }
    return sb.ToString();
}
```

`,imports` prepends every import at the top of the file, not only the ones the member references. It composes with `,bodyonly` — the imports lead, the body follows — and is a no-op for whole-file embeds and for languages with no syntactic import statement (Ruby's `require` is a method call, not a

declaration).

Walk a multi-phase method through named helpers

When the target method runs 25+ lines across distinct phases, fence each phase as its own helper instead of fencing the monolith. `ModularWordCounter` is the same logic as `MonolithWordCounter` split into three helpers — `Tokenize`, `Tally`, and `Format` — orchestrated by a short `CountWords`. A whole-type fence gives the reader the full picture in one place:

MARKDOWN

````
```csharp:symbol
examples/FocusedCodeSamplesExample/ModularWordCounter.cs > ModularWordCounter
```
````

Which renders as:

CSHARP

```
public static class ModularWordCounter
{
    /// <summary>
    /// Returns a column-aligned report of the <paramref name="topN"/> most
    /// frequent words in <paramref name="text"/> by orchestrating the three
    /// helpers below.
    /// </summary>
    /// <param name="text">Free-form text to analyse.</param>
    /// <param name="topN">Number of top-frequency words to include.</param>
    /// <returns>A multi-line string suitable for console output.</returns>
    public static string CountWords(string text, int topN)
    {
        var words = Tokenize(text);
        var ranked = Tally(words, topN);
        return Format(ranked);
    }

    /// <summary>
    /// Splits <paramref name="text"/> on whitespace, lower-cases every token,
    /// and strips surrounding punctuation. Empty tokens are dropped.
    /// </summary>
    public static List<string> Tokenize(string text)
    {
        var words = new List<string>();
        foreach (var raw in text.Split([' ', '\t', '\n', '\r'],
StringSplitOptions.RemoveEmptyEntries))
        {
            var word = raw.Trim('.', ',', '!', '?', ';', ':', '"', '\'').ToLowerInvariant();
            if (word.Length > 0)
            {
                words.Add(word);
            }
        }
        return words;
    }

    /// <summary>
    /// Groups <paramref name="words"/>, counts occurrences, and returns the
    /// top <paramref name="topN"/> ranked by frequency descending then
    /// alphabetically.
    /// </summary>
    public static List<KeyValuePair<string, int>> Tally(List<string> words, int topN)
    {
        var counts = new Dictionary<string, int>();
        foreach (var w in words)
        {
            counts[w] = counts.GetValueOrDefault(w, 0) + 1;
        }
        return counts
            .OrderByDescending(kv => kv.Value)
            .ThenBy(kv => kv.Key)
```

```
      /// <summary>    /// Renders <paramref name="ranked"/> as a header line plus one
/// column-aligned row per entry.    /// </summary>    public static string
Format(List<KeyValuePair<string, int>> ranked)    {        var sb = new StringBuilder();
sb.AppendLine($"Top {ranked.Count} words:");        foreach (var kv in ranked)        {
sb.Append(kv.Key.PadRight(12));            sb.Append(' ');
sb.AppendLine(kv.Value.ToString());        }        return sb.ToString();    }      ///
<summary>    /// Same output as <see cref="Format"/>, but rents its    /// <see
cref="StringBuilder"/> from <see cref="StringBuilderPool"/>    /// instead of allocating a
fresh one each call. Exists to pair with    /// <see cref="Format"/> inside an <c>xmldocid-
diff</c> fence so the    /// delta is small and focused on one mechanical change.    ///
</summary>    public static string FormatV2(List<KeyValuePair<string, int>> ranked)    {
var sb = StringBuilderPool.Get();        sb.AppendLine($"Top {ranked.Count} words:");
foreach (var kv in ranked)        {            sb.Append(kv.Key.PadRight(12));
sb.Append(' ');            sb.AppendLine(kv.Value.ToString());        }        var result =
sb.ToString();        StringBuilderPool.Return(sb);        return result;    }}
```

In a walkthrough, fence each helper separately so each section carries one idea:

MARKDOWN

````
```csharp:symbol,bodyonly
examples/FocusedCodeSamplesExample/ModularWordCounter.cs > ModularWordCounter.CountWords
```

```csharp:symbol,bodyonly
examples/FocusedCodeSamplesExample/ModularWordCounter.cs > ModularWordCounter.Tokenize
```

```csharp:symbol,bodyonly
examples/FocusedCodeSamplesExample/ModularWordCounter.cs > ModularWordCounter.Tally
```

```csharp:symbol,bodyonly
examples/FocusedCodeSamplesExample/ModularWordCounter.cs > ModularWordCounter.Format
```
````

The orchestrator renders as a three-liner that reads top-to-bottom as the outline for the walkthrough:

CSHARP

```
var words = Tokenize(text);
var ranked = Tally(words, topN);
return Format(ranked);
```

`Tokenize`:

CSHARP

```
var words = new List<string>();
foreach (var raw in text.Split([' ', '\t', '\n', '\r'],
StringSplitOptions.RemoveEmptyEntries))
{
    var word = raw.Trim('.', ',', '!', '?', ';', ':', '"', '\'').ToLowerInvariant();
    if (word.Length > 0)
    {
        words.Add(word);
    }
}
return words;
```

`Tally`:

CSHARP

```
var counts = new Dictionary<string, int>();
foreach (var w in words)
{
    counts[w] = counts.GetValueOrDefault(w, 0) + 1;
}
return counts
    .OrderByDescending(kv => kv.Value)
    .ThenBy(kv => kv.Key)
    .Take(topN)
    .ToList();
```

`Format`:

CSHARP

```
var sb = new StringBuilder();
sb.AppendLine($"Top {ranked.Count} words:");
foreach (var kv in ranked)
{
    sb.Append(kv.Key.PadRight(12));
    sb.Append(' ');
    sb.AppendLine(kv.Value.ToString());
}
return sb.ToString();
```

`:symbol` resolves a member by name path within the file, so give each helper a distinct name — overloads resolve to the first declaration and can't be told apart.

Show a type's shape with `,signatures`

When the point is a type's public members — what it exposes, not how each member works — append `,signatures`. Every member body collapses to `{ … }`, leaving the declarations, signatures, doc comments, and member order intact for an at-a-glance outline.

MARKDOWN

````
```csharp:symbol,signatures
examples/FocusedCodeSamplesExample/ModularWordCounter.cs > ModularWordCounter
```
````

Which renders as:

CSHARP

```csharp
public static class ModularWordCounter
{
    /// <summary>
    /// Returns a column-aligned report of the <paramref name="topN"/> most
    /// frequent words in <paramref name="text"/> by orchestrating the three
    /// helpers below.
    /// </summary>
    /// <param name="text">Free-form text to analyse.</param>
    /// <param name="topN">Number of top-frequency words to include.</param>
    /// <returns>A multi-line string suitable for console output.</returns>
    public static string CountWords(string text, int topN)
    { … }

    /// <summary>
    /// Splits <paramref name="text"/> on whitespace, lower-cases every token,
    /// and strips surrounding punctuation. Empty tokens are dropped.
    /// </summary>
    public static List<string> Tokenize(string text)
    { … }

    /// <summary>
    /// Groups <paramref name="words"/>, counts occurrences, and returns the
    /// top <paramref name="topN"/> ranked by frequency descending then
    /// alphabetically.
    /// </summary>
    public static List<KeyValuePair<string, int>> Tally(List<string> words, int topN)
    { … }

    /// <summary>
    /// Renders <paramref name="ranked"/> as a header line plus one
    /// column-aligned row per entry.
    /// </summary>
    public static string Format(List<KeyValuePair<string, int>> ranked)
    { … }

    /// <summary>
    /// Same output as <see cref="Format"/>, but rents its
    /// <see cref="StringBuilder"/> from <see cref="StringBuilderPool"/>
    /// instead of allocating a fresh one each call. Exists to pair with
    /// <see cref="Format"/> inside an <c>xmldocid-diff</c> fence so the
    /// delta is small and focused on one mechanical change.
    /// </summary>
    public static string FormatV2(List<KeyValuePair<string, int>> ranked)
    { … }
}
```

`,signatures` works on a single member too, rendering only its signature over an elided body. It targets brace-delimited languages (C#, Java, TypeScript, Go, Rust); Python and Ruby suites collapse to a best-effort `…`. As the inverse of `,bodyonly`, the two don't combine — `,signatures` wins when both are set.

Show a delta with `symbol-diff`

When the article's point is that one version replaces another — a small refactor, a migration, a perf tweak — fence both versions with `symbol-diff`. The preprocessor emits a unified diff so the reader sees the two or three lines that moved rather than comparing two fences by eye. The form works best when the delta is small; whole-method rewrites render every line as changed and bury the point.

`ModularWordCounter.FormatV2` is deliberately a one-change variant of `Format`. It rents its `StringBuilder` from a pool instead of constructing a fresh one, and returns the builder at the end. Everything else is identical, so the diff collapses to those lines.

MARKDOWN

````
```csharp:symbol-diff,bodyonly
examples/FocusedCodeSamplesExample/ModularWordCounter.cs > ModularWordCounter.Format
examples/FocusedCodeSamplesExample/ModularWordCounter.cs > ModularWordCounter.FormatV2
```
````

Which renders as:

CSHARP

```csharp
var sb = new StringBuilder();
var sb = StringBuilderPool.Get();
sb.AppendLine($"Top {ranked.Count} words:");
foreach (var kv in ranked)
{
    sb.Append(kv.Key.PadRight(12));
    sb.Append(' ');
    sb.AppendLine(kv.Value.ToString());
}
return sb.ToString();
var result = sb.ToString();
StringBuilderPool.Return(sb);
return result;
```

The fence body must hold exactly two references, one per line, in before → after order. `,bodyonly` applies to both sides, so the diff compares implementations without declaration boilerplate drowning out the change.

Embed a whole file with a bare path

Top-level-statement `Program.cs` files, `.razor` components, markdown or YAML fixtures, and JSON / TOML / config files have no member to scope to. A bare `<file>` reference with no `> member` embeds the entire file:

MARKDOWN

````
```csharp:symbol
examples/FocusedCodeSamplesExample/Program.cs
```
````

Which renders as:

CSHARP

```csharp
using FocusedCodeSamplesExample;

const string sample = """
    the quick brown fox jumps over the lazy dog
    the fox was quick and the dog was lazy
    """;

Console.WriteLine("=== MonolithWordCounter ===");
Console.WriteLine(MonolithWordCounter.CountWords(sample, topN: 3));

Console.WriteLine("=== ModularWordCounter ===");
Console.WriteLine(ModularWordCounter.CountWords(sample, topN: 3));
```

Verify

- Rebuild the site with `dotnet run --project docs/Pennington.Docs -- build` and reload the page — each fence renders at the scope its info string declares, with no carry-over of enclosing-type members. A member-scoped fence (`> Type.Member`) shows only that member; a `,bodyonly` fence drops the signature; the whole-file fence shows every line of `Program.cs`.
- Rename `Tokenize` to `Split` in `examples/FocusedCodeSamplesExample/ModularWordCounter.cs` and rebuild — the build report surfaces an unresolved `ModularWordCounter.Tokenize` reference rather than silently rendering nothing.

Related

- Reference: Markdown extensions catalog — the full fence grammar including `symbol`, `symbol,bodyonly`, and `symbol-diff`.
- Reference: Code-block argument reference — info-string parser details and the full list of suffix forms.
- How-to: Annotate code blocks — per-line `[!code highlight]` / `[!code ++]` directives that compose with the fence forms on this page.

CHAPTER 3

Rich Content

Add a colored callout for a note, tip, warning, or caution

Guides GitHub-style alerts: open a blockquote whose first line is `[!KIND]` in uppercase. Pennington recognizes five kinds and paints each one differently.

To surface a note, tip, or warning inline without reaching for a Razor component, open a standard blockquote whose first line is `[!KIND]` in uppercase. The five built-in kinds — `NOTE`, `TIP`, `IMPORTANT`, `WARNING`, `CAUTION` — fix the visual treatment; pick the one whose signal strength matches the message. The marker must be the first inline of the first paragraph — no leading text.

Before you begin

- An existing Pennington site renders markdown (see Create your first Pennington site if not).
- The pipeline was built through `AddPennington` / `AddDocSite` / `AddBlogSite`, so `UseCustomAlerts()` is already wired into the default `MarkdownPipelineFactory`.
- The default MonorailCSS integration or a stylesheet targets the `markdown-alert` / `markdown-alert-{kind}` classes.

The five alert kinds

Each kind below shows the source markdown above the rendered output. Every line after the marker is regular markdown — inline formatting, links, lists, and code spans all work because the rest of the blockquote passes through the standard Markdig pipeline unchanged.

Note

MARKDOWN

```
> [!NOTE]
> Notes carry side information worth a glance before continuing.
```

Note

Notes carry side information worth a glance before continuing.

Tip

MARKDOWN

```
> [!TIP]
> Tips point at a smart default or a pattern that keeps the common case simple.
```

> **Tip**
>
> Tips point at a smart default or a pattern that keeps the common case simple.

Important

MARKDOWN

```
> [!IMPORTANT]
> Important callouts flag content that is required for the rest of the page.
```

> **Important**
>
> Important callouts flag content that is required for the rest of the page.

Warning

MARKDOWN

```
> [!WARNING]
> Warnings flag output that is likely incorrect if the advice is ignored.
```

> **Warning**
>
> Warnings flag output that is likely incorrect if the advice is ignored.

Caution

MARKDOWN

```
> [!CAUTION]
> Cautions surface destructive operations — wire-format breaks, security risks.
```

> **Caution**
>
> Cautions surface destructive operations — wire-format breaks, security risks.

What the renderer emits

Each alert wraps in three CSS classes: `markdown-alert` (always present), `markdown-alert-{kind}` where `{kind}` is the lower-cased token, and `not-prose` (which isolates the alert from the surrounding page-prose typography rules). Stylesheets target the first two classes for the color treatment. An unrecognized token falls back to a plain `<blockquote>` with no alert styling, so the marker stays visible instead of turning into a misleading callout.

MARKDOWN

```
> [!INFO]
> Unknown kind: this falls back to a plain blockquote.
```

> [!INFO] Unknown kind: this falls back to a plain blockquote.

See Markdown extensions catalog for the full kind-to-class table.

Verify

- Each alert renders as a colored callout with no `[!KIND]` text in the body.
- View source — the outer element carries `class="markdown-alert markdown-alert-note"` (or the matching kind).
- An unrecognized kind like `[!INFO]` falls back to a plain `<blockquote>`, signaling that the parser rejected the marker.

Related

- Reference: Markdown extensions catalog — the full list of non-CommonMark features including alerts and their emitted CSS classes
- How-to: Embed a Mermaid diagram in a markdown page — Mermaid fences for when a callout is not the right shape
- How-to: Drop a Razor component into a markdown page — `<Card>` or a custom component covers cases beyond the five built-in kinds
- Background: The content pipeline and union types — where the Markdig pipeline (and the alert extension) sits in the render chain

Embed a Mermaid diagram in a markdown page

Guides Author Mermaid diagrams in markdown with a fenced `mermaid` block and let the DocSite render them client-side with theme awareness.

To drop a flowchart, sequence diagram, or other visual into a markdown article without authoring SVG by hand, fence the diagram with `mermaid` as the language. The DocSite renders the fence body verbatim, then a client script swaps each block for an SVG in the browser. Sites that build offline or behind a firewall must vendor Mermaid themselves — see Vendor the library for offline builds below.

Before you begin

- An existing Pennington site renders markdown (see Create your first Pennington site if not).
- The host uses `AddDocSite` or `AddBlogSite`, or — on a bare `AddPennington` host — references `Pennington.UI` and emits its script bundle from the layout (`<script type="module" src="/_content/Pennington.UI/scripts.js" defer></script>`).

- Familiarity with Mermaid syntax — this page covers the fence wiring, not Mermaid itself. See the upstream Mermaid docs for the grammar.

Diagram syntaxes

Pennington does not preprocess the fence body — anything valid in Mermaid renders as-is. The two most common shapes are below.

Flowchart

Fence a block with `mermaid` as the language and write a `flowchart` body. The client script swaps the `<code>` element for an SVG at page load.

MARKDOWN

````
```mermaid
flowchart LR
    A[Markdown file] --> B[MarkdownContentParser]
    B --> C[ContentPipeline]
    C --> D[MarkdownContentRenderer]
    D --> E[Response processors]
    E --> F[Rendered HTML]
```
````

MERMAID

```
flowchart LR
    A[Markdown file] --> B[MarkdownContentParser]
    B --> C[ContentPipeline]
    C --> D[MarkdownContentRenderer]
    D --> E[Response processors]
    E --> F[Rendered HTML]
```

Sequence diagram

Sequence diagrams use the same `mermaid` fence with a `sequenceDiagram` body.

MARKDOWN

````
```mermaid
sequenceDiagram
    Alice->>Bob: Hello Bob, how are you?
    Bob-->>Alice: I'm good, thanks!
    Alice->>Bob: Want to grab lunch?
    Bob-->>Alice: Sounds great.
```
````

MERMAID

```
sequenceDiagram
    Alice->>Bob: Hello Bob, how are you?
    Bob-->>Alice: I'm good, thanks!
    Alice->>Bob: Want to grab lunch?
    Bob-->>Alice: Sounds great.
```

What the renderer emits

Each fence renders as `<pre><code class="language-mermaid">…</code></pre>` with the body verbatim — Pennington does not transform it server-side. The browser script then loads Mermaid from `cdn.jsdelivr.net` and replaces each block with an inline SVG. The theme toggle re-renders every diagram with the matching built-in Mermaid theme, so diagrams track light and dark mode. Diagrams render on both the live dev server and the static build output.

For per-diagram theme overrides, use Mermaid's inline `%%{init: { 'theme': '…' } }%%` directive at the top of the fence body — Mermaid syntax, not Pennington syntax.

Vendor the library for offline builds

The bundled support loads Mermaid from `cdn.jsdelivr.net` at first render. A site that builds offline or behind a firewall must serve the library itself: vendor the Mermaid module into `wwwroot` and load it from your own layout. This is the same pattern any CDN-backed widget follows — see Load the library and your script for the vendoring recipe.

Verify

- Open a page with a diagram in the browser. The fence renders as an SVG, not as a raw code block. A diagram still showing its `flowchart` / `sequenceDiagram` text means the script never replaced it.
- On a failure, open the browser network tab and confirm the `import` from `cdn.jsdelivr.net` succeeds. A blocked or 404'd jsdelivr request is the silent-failure signature — Mermaid never loads and the original code block stays in place. Vendor the library to fix it.

Related

- Reference: Markdown extensions catalog — the full list of non-CommonMark features, for context on what Pennington does and does not preprocess
- Reference: Code-block argument reference — the info-string grammar (`mermaid` is a bare language token, no arguments needed)
- How-to: Add a colored callout for a note, tip, warning, or caution — the neighboring visual-element authoring surface, for comparison
- Background: MonorailCSS integration — how the DocSite's theme tokens (the same ones Mermaid tracks) are generated

Drop a Razor component into a markdown page

Guides Embed Pennington.UI components (and your own Razor components) inside a `.md` file through Mdazor-backed rendering.

To place a Razor component tag — `<Badge>`, `<Card>`, or one of your own — directly inside a `.md` file instead of authoring raw HTML, write the tag where CommonMark allows an HTML block. Mdazor matches the tag against registered component types and binds attribute values to `[Parameter]` properties. To author a brand-new component from scratch, see Author a custom Razor component for markdown.

Before you begin

- A working Pennington site that renders markdown (see Create your first Pennington site if not).
- The host calls `AddDocSite`, `AddBlogSite`, or `AddPennington`. The first two pre-register the Pennington.UI components meant for markdown use; bare `AddPennington` requires the registration shown under "Register components on a bare host" below.
- Component tag names start with an uppercase letter and match the Razor component type name — case-sensitive on the leading character (`<Card>`, not `<card>`).

Authoring shapes

`AddDocSite` pre-registers nine components — `<Badge>`, `<BigTable>`, `<Card>`, `<CardGrid>`, `<Checkpoint>`, `<LinkCard>`, `<RenderedFixture>`, `<Step>`, and `<Steps>`. `AddBlogSite` pre-registers the same set minus `<RenderedFixture>` (eight). The H3s below cover the three most common authoring patterns; for the full parameters of each component, see Content components.

Inline a built-in tag

Place the tag anywhere CommonMark allows an HTML block. Attribute values bind to `[Parameter]` properties by case-insensitive name match.

MARKDOWN

```
<Badge Text="Preview" />
```

Preview

Pass markdown as `ChildContent`

Whatever appears between the open and close tags becomes the component's `ChildContent` render fragment and is parsed as markdown — `**bold**`, links, and nested components all work inside the body.

MARKDOWN

```
<Card Title="New in v2">
The **v2 pipeline** ships with [unified dev and build](xref:explanation.core.dev-vs-build).
</Card>
```

New in v2

The **v2 pipeline** ships with unified dev and build.

Bind primitive attributes

Only primitive parameter types (strings, numbers, booleans) bind from markdown attributes — the value arrives as a raw string and Mdazor converts it via reflection.

MARKDOWN

```
<Card Title="Fast" Color="accent">
Pages render in a single SSR pass.
</Card>
```

Fast

Pages render in a single SSR pass.

For complex data, pack it into a delimited string and parse inside the component, or use `ChildContent` for rich content. Content components shows the same pattern in the parameter tables.

Register components on a bare host

`AddPennington` wires the component registry via `AddMdazor()` but does not register any components. Chain one `AddMdazorComponent<T>()` call per component that should be available in markdown — see Content components for the registration block DocSite and BlogSite use.

CSHARP

```
builder.Services.AddMdazorComponent<Badge>()
                .AddMdazorComponent<Card>()
                // ... one line per component
                ;
```

What the renderer emits

Mdazor parses the component tag out of the HTML block, instantiates the matching Razor component, binds attribute values to `[Parameter]` properties, and renders the result inline. The original tag literal disappears from the output. An unregistered tag on a bare `AddPennington` host falls through unchanged and renders as literal text — the fastest way to confirm whether the registration is what activates a tag.

See Content components for the parameters of each built-in component.

Read page context in a component

Attributes carry what the author types on the tag. For facts about the *page* — the source file, the canonical URL, the front matter — a component reads the ambient `MdazorContext` that Pennington supplies for every rendered page. Declare a `[CascadingParameter]` of type `MdazorContext`; nothing goes on the tag.

RAZOR

```
@using Mdazor
@using Pennington.FrontMatter

<p>Rendered from <code>@Context?["FileName"]</code> at <code>@Context?["Url"]</code>.</p>

@code {
    [CascadingParameter] public MdazorContext? Context { get; set; }
}
```

`MdazorContext` exposes the bag through `Values`, an indexer (`Context["FileName"]`), `TryGet`, and `Get<T>`. Pennington fills it with these keys, matched case-insensitively:

```
Key Value     `SourceFile` Source path on disk for the page   `FileName` /
`FileNameWithoutExtension` The source file name, with and without extension   `Url` /
`CanonicalPath` Canonical URL path for the page   `OutputFile` Static output path written
during `build`   `Locale` Locale code; empty for the default locale   `Metadata` The page's
front matter as an [IFrontMatter](https://usepennington.net/reference/front-matter/keys/)
(`Title`, `Description`, `Uid`, …)   `Derived` Enricher-contributed values (reading time,
git last-modified, …) keyed by enricher name
```

The context is delivered as a cascading value, so it reaches the component and any components nested inside it. It does **not** cross into an interactive (WebAssembly/Server) island, so read it from the statically rendered components that make up the page body. The BeyondCustomRazorComponentExample `PageFacts` component shows the full pattern.

Verify

- The `<Badge Text="Preview" />` example renders as a styled pill — a rounded, ring-bordered chip — not the literal text `<Badge Text="Preview" />`. Seeing the raw tag means the component is not registered on the host.
- View source — the badge is a `<span class="not-prose inline-flex ...">`, and no `<Badge>` literal survives in the HTML.
- On a bare `AddPennington` host, an unregistered tag passes through unchanged and shows as literal text. Add the `AddMdazorComponent<Badge>()` call and the pill appears, confirming the registration is what activates the tag.

Related

- Reference: Content components — parameters and render behavior for `Badge`, `BigTable`, `Card`, `CardGrid`, `Checkpoint`, `LinkCard`, `RenderedFixture` (DocSite only), `Step`, and `Steps`.
- Tutorial: Author a custom Razor component for markdown — write your own `<PricingCard>`-style component and wire it through `AddMdazorComponent<T>()`.
- How-to: Customize DocSite layouts and components — when you need to change the surrounding page, not only embed a tag inside markdown.

Add a custom schema.org JSON-LD type

Guides Define a record that subclasses JsonLdEntity, attribute its properties for System.Text.Json, and either let the front matter own it via IHasStructuredData or render it inline from a Razor page.

Pennington's `<StructuredData>` component takes any `JsonLdEntity` and emits it as a `<script type="application/ld+json">` in the page head. To support a schema.org type the framework doesn't ship — `Recipe`, `Product`, `ScholarlyArticle`, `Event`, or anything else — write a record in your own assembly.

There are two ways to wire it in: implement `IHasStructuredData` on your front matter so the template emits it automatically, or build the entity inline from a Razor page. The capability-interface path is the default; the inline path is the fallback when the page doesn't have a front matter (a hand-routed Razor page) or when the entity depends on something other than front-matter values.

Before you begin

- A working Pennington site with `CanonicalBaseUrl` set on `PenningtonOptions` or `DocSiteOptions`. The shipped templates skip JSON-LD when this is empty so URLs don't end up relative.

1. Define the record

Subclass `JsonLdEntity`, override `Type` with the schema.org type literal, and attribute every field with `[JsonPropertyName]`. Repeat the `[JsonPropertyName("@type")]` attribute on the override — `System.Text.Json` doesn't inherit attributes through `override`.

CSHARP

```
public sealed record JsonLdRecipe : JsonLdEntity
{
    /// <inheritdoc />
    [JsonPropertyName("@type")]
    public override string Type => "Recipe";

    /// <summary>Recipe name.</summary>
    [JsonPropertyName("name")]
    public required string Name { get; init; }

    /// <summary>Canonical URL of the recipe page.</summary>
    [JsonPropertyName("url")]
    public string? Url { get; init; }

    /// <summary>Short description of the dish.</summary>
    [JsonPropertyName("description")]
    public string? Description { get; init; }

    /// <summary>Servings count, e.g. "4 servings".</summary>
    [JsonPropertyName("recipeYield")]
    public string? RecipeYield { get; init; }

    /// <summary>Prep duration as an ISO 8601 duration, e.g. "PT15M".</summary>
    [JsonPropertyName("prepTime")]
    public string? PrepTime { get; init; }

    /// <summary>Cook duration as an ISO 8601 duration, e.g. "PT30M".</summary>
    [JsonPropertyName("cookTime")]
    public string? CookTime { get; init; }

    /// <summary>One-line ingredient strings, with amount and unit baked in.</summary>
    [JsonPropertyName("recipeIngredient")]
    public required IReadOnlyList<string> Ingredients { get; init; }

    /// <summary>Step text, one entry per instruction.</summary>
    [JsonPropertyName("recipeInstructions")]
    public required IReadOnlyList<string> Instructions { get; init; }
}
```

This example defines `JsonLdRecipe`, a `Recipe` entity record. It is not a framework type — you own it in your own assembly — and it is the record the wiring snippets in steps 3a and 3b instantiate.

The base `JsonLdEntity` already supplies `@context` (defaulted to `https://schema.org`). Override the `Context` initializer if you need a different vocabulary.

Optional fields stay nullable; `JsonLdSerializer` is configured with `JsonIgnoreCondition.WhenWritingNull`, so unset fields drop out of the JSON.

2. Apply the date converter when you have dates

For schema.org dates, attribute the property with `[JsonConverter(typeof(JsonLdDateConverter))]`. The converter emits `yyyy-MM-ddTHH:mm:ssZ` regardless of `DateTimeKind`, matching the wire format Google's rich-results validator expects.

CSHARP

```
[JsonPropertyName("datePublished")]
[JsonConverter(typeof(JsonLdDateConverter))]
public DateTime? DatePublished { get; init; }
```

3a. Wire it through the front matter (capability path)

When the entity's data lives in front matter, implement `IHasStructuredData` on your front-matter record. The DocSite and BlogSite templates check for the capability and emit whatever entities the front matter yields — no Razor code required.

CSHARP

```
public record RecipeFrontMatter : IFrontMatter, IHasStructuredData
{
    public string Title { get; init; } = "";
    public string? Description { get; init; }
    public IReadOnlyList<string> Ingredients { get; init; } = [];
    public IReadOnlyList<string> Steps { get; init; } = [];

    public IEnumerable<JsonLdEntity> GetStructuredData(StructuredDataContext context)
    {
        yield return new JsonLdRecipe
        {
            Name = Title,
            Description = Description,
            Url = context.CanonicalUrl,
            Ingredients = Ingredients,
            Instructions = Steps,
        };
    }
}
```

`StructuredDataContext.CanonicalUrl` is the absolute URL the template has already resolved (canonical base plus the page's path). `StructuredDataContext.FallbackAuthorName` is honored by BlogSite when the front matter's `Author` is empty.

A page can yield multiple entities — pair a `Recipe` with a `BreadcrumbList`, or emit a `HowTo` alongside a `Recipe` for instruction-heavy pages.

3b. Render inline from a Razor page (escape hatch)

When the entity isn't a function of front matter — a hand-routed landing page, a page that pulls from a data file, a page that wraps a third-party feed — pass the entities directly into `<StructuredData>`:

RAZOR

```
@using Pennington.StructuredData
@inject PenningtonOptions Options

@if (!string.IsNullOrEmpty(Options.CanonicalBaseUrl))
{
    <StructuredData Entities="BuildEntities()" />
}

@code {
    private IEnumerable<JsonLdEntity> BuildEntities()
    {
        yield return new JsonLdRecipe
        {
            Name = "Weeknight pasta with garlic and oil",
            Ingredients = ["1 lb spaghetti", "6 cloves garlic, thinly sliced"],
            Instructions = ["Boil the pasta", "Toast the garlic", "Toss and serve"],
        };
    }
}
```

Verify

1. Visit the page in dev mode and view source. Look for `<script type="application/ld+json">` with your `@type`.
2. Copy the rendered HTML into Google's Rich Results test and confirm the type validates.
3. If a field is missing from the JSON, check that the property is non-null and that it carries a `[JsonPropertyName]` attribute — properties without one use the C# member name verbatim.

Related

- Reference: Utility components — `<StructuredData>` parameters.
- How-to: Render a Razor component as a page on a bare host — wires the page that emits the JSON-LD.
- The schema.org vocabulary at schema.org for available types and field names.

Tab platform or language variants together

Guides Wrap whole sections of prose, code, and lists in DocFX-style content tabs — synced page-wide, with dependent tabs for a second axis.

When one instruction has per-platform or per-language variants — install steps for macOS, Linux, and Windows; the same API in C# and F# — content tabs let the reader pick one and read a self-contained walk-through. Unlike a tabbed code block, a content tab holds a whole section: paragraphs, lists, callouts, and code together.

Before you begin

- An existing Pennington site renders Markdown (see Create your first Pennington site if not).
- The pipeline was built through `AddPennington` / `AddDocSite` / `AddBlogSite`, so `UseContentTabs()` is already wired into the default `MarkdownPipelineFactory`.
- The default MonorailCSS integration, or a stylesheet, targets the `ctabs` / `ctab-btn` / `ctab-panel` classes.

Author a tab group

Each tab opens with a level-1 heading whose link points at `#tab/<id>`. The link text becomes the button label; the `<id>` identifies the tab. A horizontal rule (`---`) ends the group. Everything between two tab headings is that tab's panel.

MARKDOWN

````
# [macOS](#tab/macos)

Install the SDK with the official `.pkg` installer or Homebrew's cask.

- Confirm the install with `dotnet --list-sdks`.
- On Apple Silicon, grab the `arm64` build.

# [Linux](#tab/linux)

Install `dotnet-sdk-11.0` from your distribution's package feed.

```bash
sudo apt-get install dotnet-sdk-11.0
```

# [Windows](#tab/windows)

Install the SDK with the Windows installer or `winget`.

```powershell
winget install Microsoft.DotNet.SDK.11
```

---
````

macOS Linux Windows

Install the SDK with the official `.pkg` installer or Homebrew's cask.

- Confirm the install with `dotnet --list-sdks`.
- On Apple Silicon, grab the `arm64` build.

Install `dotnet-sdk-11.0` from your distribution's package feed.

BASH

```
sudo apt-get install dotnet-sdk-11.0
```

Install the SDK with the Windows installer or `winget`.

POWERSHELL

```
winget install Microsoft.DotNet.SDK.11
```

The first tab is active by default. Each panel is ordinary Markdown — the bullet list and fenced code above both render with the page's normal prose styling.

Tabs sync across the page

Tab `<id>`s are page-wide. Every group that shares an id selects together: pick **Linux** once and every `#tab/linux` on the page follows. The client unions id sets across all co-occurring groups, and the choice persists in `localStorage` per set, so the selection carries across pages with the same tab groupings as well as within the current page. Switch a tab in the group above and watch this one match:

macOS Linux Windows

You picked macOS — paths use `/Users/you`.

You picked Linux — paths use `/home/you`.

You picked Windows — paths use `C:\Users\you`.

Dependent tabs

A third path segment — `#tab/<id>/<condition>` — makes a tab depend on another group's selection. The `<condition>` is itself a tab id selected elsewhere. Use it when one choice (the OS) should drive a second (the tool), without making the reader pick twice.

MARKDOWN

```
# [.NET CLI](#tab/tool/linux)

Run `dotnet run` from a bash shell.

# [.NET CLI](#tab/tool/windows)

Run `dotnet run` from PowerShell.

# [Editor](#tab/editor/linux)

Open the folder with `code .`.

# [Editor](#tab/editor/windows)

Open the folder with `code .` or Visual Studio.

---
```

The group below shows **.NET CLI** and **Editor** as its two buttons; each has a Linux and a Windows variant. It follows the platform you picked above — switch **macOS / Linux / Windows** there and the panel here re-resolves.

.NET CLI Editor

Run `dotnet run` from a bash shell — the dev host binds `http://localhost:5000`.

Run `dotnet run` from PowerShell — the dev host binds `http://localhost:5000`.

Open the project with `code .` and use the C# Dev Kit extension.

Open the project with `code .`, or load the `.slnx` in Visual Studio.

A condition that names an id with no selector of its own never resolves, so give every condition a plain tab group somewhere on the page.

What the renderer emits

A group renders as a `ctabs` container holding a `ctabs-bar` tab strip and one `ctab-panel` per tab. The tab strip carries `not-prose` so the buttons stay out of page typography; the panels deliberately do **not**, so panel content renders with full prose styling.

See Markdown extensions catalog for the full element, attribute, and class reference.

Verify

- The group renders as a tab strip with one button per heading, not as stacked headings — the first panel shows and the rest are hidden.
- Clicking a button switches the visible panel; two groups that share ids switch together when you pick one.

- View source — the container is a `<div class="ctabs" data-content-tabs>` holding a `ctabs-bar` strip and one `ctab-panel` per tab, with `data-active` on the first.

Related

- Reference: Markdown extensions catalog — content tabs alongside every other non-CommonMark feature
- How-to: Group adjacent code fences into a tabbed sample — tab code-only variants inside a single code block
- How-to: Reuse one snippet across many pages — pull a shared partial into a tab panel
- How-to: Add a colored callout for a note, tip, warning, or caution — callouts, which sit naturally inside a tab panel

Ship a custom client-side widget

Guides Add browser behavior to a static Pennington site by composing a server-rendered Mdazor component, your own script, and the head seam that loads a CDN library — built here as an image-gallery lightbox.

Pennington renders every page on the server in a single pass — there is no client-side hydration. To add interactive browser behavior (a lightbox, a chart, a copy-to-clipboard button), you ship your own script and attach it to the server-rendered HTML.

This guide builds an image-gallery lightbox from three parts: a server-rendered component that emits the markup, a browser script that enhances it, and the head content option that loads both your script and the third-party library. The worked library is GLightbox (MIT-licensed, dependency-free), but the pattern is the same for any library that scans the DOM and upgrades matching elements — the bundled Mermaid support (Embed a Mermaid diagram in a markdown page) follows it too.

Before you begin

- A DocSite (`AddDocSite`) or BlogSite host — this example is a DocSite. On a bare `AddPennington` host the only difference is the head content: inject the tags through your own layout's `<head>` or a response processor that inserts before `</body>` (see Transform the response body on every page).
- Familiarity with the library you are wrapping. This page covers the wiring, not GLightbox itself.
- For a complete, running setup, see `examples/BeyondClientWidgetExample` ; the sections below embed each of its files where they apply.

Render the markup on the server

Write an Mdazor component that emits the HTML the script will later find and upgrade. The component is plain server-side Razor — it renders thumbnails wrapped in `<a class="glightbox">` anchors and nothing more. The lightbox behavior is added entirely by the script in the next step.

RAZOR

```
@* ImageGallery — a server-rendered Mdazor component that emits a thumbnail grid
   of <a class="glightbox"> / <img> pairs. It runs entirely on the server; the
   client-side script (wwwroot/gallery.js) finds the `.glightbox` anchors at page
   load and upgrades them into a lightbox. This is the "SSR component plus your
   own script" seam the how-to /how-to/rich-content/client-side-widget walks
   through.

   Consumed from markdown as:

     <ImageGallery Images="merry-mixer.png, peppermint-express.png" Group="trains" />

   Only primitive parameters bind from markdown attributes, so the image list
   arrives as one comma-separated string and captions are derived from the file
   names. *@
<div class="not-prose my-8 grid grid-cols-2 gap-4 sm:grid-cols-3">
    @foreach (var image in ParsedImages)
    {
        @* target="_blank" opts the link out of Pennington's SPA navigation (it
           skips links marked target/download/data-spa-reload) so the engine does
           not hijack the click; GLightbox calls preventDefault, so the new tab
           only opens as a graceful fallback when scripting is off. *@
        <a href="@($"{BasePath}/{image.File}")"
           target="_blank" rel="noopener"
           class="glightbox group block overflow-hidden rounded-xl border border-base-200
dark:border-base-800"
           data-gallery="@Group"
           data-title="@image.Caption">
            <img src="@($"{BasePath}/{image.File}")"
                 alt="@image.Caption"
                 loading="lazy"
                 class="aspect-video w-full object-cover transition-transform duration-200
group-hover:scale-105" />
        </a>
    }
</div>

@code {
    /// <summary>Comma-separated image file names served under <see cref="BasePath"/>.
</summary>
    [Parameter] public string Images { get; set; } = "";

    /// <summary>GLightbox gallery group — anchors sharing a group page through one
lightbox.</summary>
    [Parameter] public string Group { get; set; } = "gallery";

    /// <summary>URL prefix the image files are served from.</summary>
    [Parameter] public string BasePath { get; set; } = "/guides/assets";

    private IEnumerable<(string File, string Caption)> ParsedImages =>
        Images.Split(',', StringSplitOptions.RemoveEmptyEntries |
StringSplitOptions.TrimEntries)
```

```
    private static string ToCaption(string file)    {        var name =
Path.GetFileNameWithoutExtension(file).Replace('-', ' ');        return
System.Globalization.CultureInfo.InvariantCulture.TextInfo.ToTitleCase(name);    }}
```

Register the component so it is usable as a tag in markdown. `AddMdazorComponent<T>()` is the only DI line needed; the registry resolves the tag at render time.

CSHARP

```csharp
using BeyondClientWidgetExample;
using BeyondClientWidgetExample.Components;
using Mdazor;
using Pennington.DocSite;

var builder = WebApplication.CreateBuilder(args);

// A DocSite whose only customization is one client-side widget: an image-gallery
// lightbox. GalleryWidget.BuildDocSiteOptions injects the GLightbox CDN assets
// and the local init script into <head>; AddMdazorComponent<ImageGallery>()
// registers the server-rendered tag the script enhances. Backs the how-to
// /how-to/rich-content/client-side-widget.
builder.Services.AddDocSite(GalleryWidget.BuildDocSiteOptions);
builder.Services.AddMdazorComponent<ImageGallery>();

var app = builder.Build();

app.UseDocSite();

await app.RunDocSiteAsync(args);
```

Only primitive attributes bind from markdown, so the image list arrives as one comma-separated string and the component derives a caption from each file name. For the binding rules and the structured-data workarounds, see Drop a Razor component into a markdown page.

Write the browser script

The script runs in the browser, finds the server-rendered anchors, and hands them to the library. Keep the initializer idempotent — it runs once on the first load and again after every in-site navigation (covered under Survive SPA navigation below).

JAVASCRIPT

```
// gallery.js — the client half of the image-gallery widget.
//
// The server renders <a class="glightbox"> thumbnails (Components/ImageGallery.razor);
// this script finds them in the browser and upgrades them into a lightbox.
// GLightbox itself loads from a CDN <script> in <head> (see GalleryWidget.cs), so
// the global GLightbox function is available by the time this deferred script runs.
let lightbox = null;

function initGallery() {
    if (typeof GLightbox !== 'function') return;
    // When re-running after an in-site navigation, tear down the previous
    // instance first so its event listeners don't accumulate.
    lightbox?.destroy();
    lightbox = GLightbox({ selector: '.glightbox' });
}

// First full page load. A deferred script runs after the DOM is parsed, so the
// gallery markup is already present.
if (document.readyState === 'loading') {
    document.addEventListener('DOMContentLoaded', initGallery);
} else {
    initGallery();
}

// Pennington swaps page content on in-site navigation without a full reload, so
// re-scan for galleries after each SPA commit. No-op if spa-engine.js is absent.
document.addEventListener('spa:commit', initGallery);
```

Put the script in `wwwroot`, where the host serves it at `/gallery.js` and the static build copies it to the output.

Load the library and your script

`DocSiteOptions.AdditionalHtmlHeadContent` is a raw HTML string rendered inside every page's `<head>` — the place for the library's stylesheet and script plus your own. Load the library first, then your script. Both `<script>` tags use `defer`, so they execute in document order: the library defines its global before your script calls it.

CSHARP

```
=> """
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/glightbox@3.3.1/dist/css/glightbox.min.css">
<script src="https://cdn.jsdelivr.net/npm/glightbox@3.3.1/dist/js/glightbox.min.js" defer>
</script>
<script src="/gallery.js" defer></script>
"""
```

Pin the library to a version so the build is reproducible, and assign the fragment to the head content option on the options record:

CSHARP

```
public static DocSiteOptions BuildDocSiteOptions() => new()
{
    SiteTitle = "Client Widget Example",
    SiteDescription = "Ships an image-gallery lightbox by composing a CDN script, the head
seam, and a server-rendered Mdazor component.",
    AdditionalHtmlHeadContent = BuildGalleryHeadContent(),
    Areas =
    [
        new ContentArea("Guides", "guides"),
    ],
};
```

For a site that builds offline or behind a firewall, vendor the two library files into `wwwroot` and point the tags at the local copies — a CDN load fails silently otherwise.

Display the media the widget references

Colocate the images the gallery displays under `Content`, next to the page that uses them — here, `Content/guides/assets/`. Pennington copies colocated assets to the output and the build's link checker recognizes them, so referencing them from the rendered markup keeps the build clean. See Add images and shared assets to a page.

Use it in a page

Drop the tag into any markdown page. The `Images` attribute is the comma-separated file list, `Group` ties the thumbnails into one lightbox so the arrow keys move between them, and `BasePath` (optional) is the URL prefix the files are served from.

MARKDOWN

```
<ImageGallery Images="peppermint-express.png, merry-mixer.png, indigo-inchworm.png"
Group="trains" />
```

Survive SPA navigation

Pennington's SPA engine swaps page content on in-site navigation without a full reload, which affects a client widget two ways.

Re-run your initializer. `DOMContentLoaded` fires only on the first full load. After an in-site navigation the new page's markup arrives through a region swap, so re-bind from the `spa:commit` event — `gallery.js` above adds one listener for exactly this. See the SPA lifecycle events.

Opt link-triggered widgets out of navigation. The engine treats same-origin `<a>` clicks as navigation. Because the gallery thumbnails are links to the full image, an un-marked click would be intercepted by the engine instead of opening the lightbox. The engine automatically skips links marked `target="_blank"` or

`download` (see anchor attributes), so the component sets `target="_blank"` — GLightbox calls `preventDefault`, making the new tab a graceful fallback only when scripting is off.

Verify

- Run `dotnet run --project examples/BeyondClientWidgetExample` and open `/guides/image-gallery`. Click a thumbnail — the lightbox opens. Navigate to the page through an in-site link and click again — it still opens, confirming the `spa:commit` re-init.
- Run `dotnet run --project examples/BeyondClientWidgetExample -- build output`. Confirm `output/gallery.js`, the images under `output/guides/assets/`, and the `<a class="glightbox">` markup in `output/guides/image-gallery/index.html` are all present, and that the build reports no broken links.

Related

- How-to: Drop a Razor component into a markdown page — the attribute-binding rules for the Mdazor tag this widget renders.
- How-to: Transform the response body on every page — inject the head/`</body>` tags on a bare `AddPennington` host instead of through `DocSiteOptions`.
- How-to: Add images and shared assets to a page — where page-referenced images and shared assets live.
- Reference: SPA engine attributes and events — the SPA engine's anchor opt-outs and `spa:commit` lifecycle event.
- Background: SPA navigation through region swaps — why the SPA model is server-rendered with no client hydration.

CHAPTER 4

Navigation

Reorder, rename, or hide entries in the sidebar

Guides Use front-matter keys, a folder _meta.yml sidecar, and folder layout to control the auto-built sidebar — reorder pages and sections, promote a section landing, override the section header, and hide drafts.

The sidebar is generated from folder layout, a per-folder `_meta.yml` sidecar, and per-page front matter. Each adjustment below is independent — pick the one that matches the change you need. To replace the sidebar component itself, see Customize the DocSite chrome through DocSiteOptions.

Before you begin

- A Pennington DocSite has markdown under `Content/<area>/` with at least one subfolder (the subfolder is what creates a sidebar group — see Work with front matter if not)
- Pages use `DocSiteFrontMatter` or another type that implements `IOrderable` + `ISectionable`
- The basics of `order:` and `isDraft:` are familiar — if not, start with Manage drafts, tags, and ordering

For a working reference, see `examples/DocSiteKitchenSinkExample` — `Content/main/customize-sidebar.md` exercises the same keys.

Options

Reorder pages within a folder

Lower `order:` values sort earlier inside a folder; ties break alphabetically on `Title`. Numbers are local to the folder — start at `1` and count up. Leave gaps (`10, 20, 30`) only if you anticipate frequent inserts between siblings.

YAML

```
---
title: Install
order: 2
---
```

Reorder sections (folders) within the sidebar

Drop a `_meta.yml` sidecar into the folder you want to reposition. Its `order:` sets where the folder lands among its siblings — independent of any `order:` on the pages inside.

YAML

```
# Content/main/widgets/_meta.yml
order: 3
```

Without a sidecar, the folder's position falls back to the lowest `order:` of any descendant (the min-of-children rule). Mixing modes is fine — folders with a sidecar sort by the explicit value, folders without sort by the emergent value. See Folder sidecar (`_meta.yml`) for the full sidecar schema.

Promote a page to be the section landing

Name the file `index.md` inside the section subfolder (for example `Content/main/widgets/index.md`). Pennington routes it at the subfolder URL and surfaces it as the section's lead entry rather than a separate child.

YAML

```
---
title: Widgets
order: 1
---
```

When the folder also has a `_meta.yml`, the sidecar's `order:` overrides the index.md's `order:` for the **section's** position in its parent. The index.md's own `order:` then has no effect — set it to `1` for clarity or omit it.

Override the displayed section title

There are two ways to change the printed section header. The folder name converts from kebab-case to title case by default (`getting-started` → "Getting Started").

Option A — rename the folder. The header follows the new name.

Option B — set `title:` in `_meta.yml`. Wins over both the auto-formatted folder name and any `title:` on a sibling `index.md`.

YAML

```
# Content/main/widgets/_meta.yml
title: "Widget Catalog"
order: 3
```

The front-matter `sectionLabel:` key is separate from both — it sets the page-context label surfaced for breadcrumbs and prev/next navigation, not the sidebar group header.

Hide an unfinished page from the sidebar

For a page that isn't ready to publish, set `isDraft: true`. It drops out of the sidebar, the search index, and `llms.txt`, and `dotnet run` still serves it so you can preview your work.

YAML

```
---
title: Work in progress
isDraft: true
---
```

Under `dotnet run -- build` a draft is excluded from the static output entirely — the page file is never written, so any `xref:` link to it fails to resolve. This is the canonical draft rule documented in the front-matter key reference; `isDraft` is for pages that aren't meant to ship, not for hiding pages you still want published.

Keep a page published but out of the sidebar

To ship a page at its URL while keeping it off the sidebar — a "published but unlisted" page — leave its `title:` empty instead of drafting it. A page with no title produces no sidebar entry (and no search or `llms.txt` entry), but the route still renders and builds, so `xref:` links to it resolve.

YAML

```
---
title: ""
---
```

A page with `redirectUrl:` is also omitted from the sidebar.

Verify

- Run `dotnet run`; reordered pages appear in ascending `order:` inside their folder
- A folder with a `_meta.yml` lands at the position its `order:` specifies, even when its descendants' values would have placed it elsewhere
- The section subfolder's `index.md` lands at `/<area>/<section>/` and renders as the section's lead entry in the sidebar
- The drafted page's URL still serves the page in `dotnet run`; the entry is absent from the sidebar on reload. Under `dotnet run -- build` the page is excluded from the static output.
- The empty-`title:` page has no sidebar entry but its URL serves in both `dotnet run` and `dotnet run -- build`

Related

- Reference: Folder sidecar (_meta.yml)
- Reference: Front matter key reference
- Reference: Navigation UI components
- Background: How the sidebar is built

Link between pages without hardcoding URLs

Guides Pick the right link form for sibling pages, cross-area targets, anchors, assets, and external sites — and let Pennington rewrite for sub-path deployments.

To link from one page to another without hardcoding a URL that may break on rename or sub-path deploy, pick the link form that matches the target's relationship to the source. When rename safety matters most, give the target a stable `uid:` and cross-reference it — the last form below.

Before you begin

- An existing Pennington site with at least two markdown pages (see Serve markdown through Blazor Pages if not).
- The URL of the target page or asset is known (the sidebar or the rendered page's address bar are the quickest sources).

Link forms

Relative path to a sibling page

Write a standard markdown link with a relative target such as `[Customize the sidebar](./customize-sidebar)`. The target resolves against the current page's URL. Relative links survive section moves as long as both files stay in the same folder, which makes them the right choice for tightly coupled pages.

Absolute path to a page in another area

When the target lives in a different section or area, use a site-absolute path: `[API index](/api/)`. Absolute paths are stable across folder moves of the source page but break if the target URL changes. Reach for `uid:` cross-references when rename safety matters.

Anchor fragment to a heading

Append `#slug` to any link target to scroll to a specific heading. Markdig's auto-identifier pass slugifies headings, so `## Relative links to sibling pages` becomes `#relative-links-to-sibling-pages`. The same fragment syntax applies to relative links, absolute paths, and uid-based xrefs.

Colocated and shared assets

Reference assets stored under `Content/` with a relative path (`./assets/diagram.png`) and assets under `wwwroot/` with a site-absolute path (`/shared.png`). The content copy pass and the static-file pipeline place the files at matching URLs, so the two rules map directly to where the file lives on disk. For more on asset placement, see Add images and shared assets to a page.

External site

Write the full URL directly: `[Markdig](https://github.com/xoofx/markdig)`. Pennington leaves the `href` untouched. For sites that need `rel="noopener"` or `target="_blank"` injected uniformly, write an `IHtmlResponseRewriter` (see Rewrite HTML attributes after parsing).

Sub-path deployment

Build with `--base-url /docs` so every rendered response has its `href`, `src`, and `action` attributes prefixed at response time. Write root-relative links like `/api/` in markdown — the rewriter turns them into `/docs/api/` on the way out.

MARKDOWN

```
[API index](/api/)
```

Hardcoding the prefix in markdown defeats the rewriter. See Host under a sub-path (base URL) for the build invocation.

Cross-reference by uid

When the target may move to a different folder or get renamed, link by `uid:` instead of by path. Give the target page a stable, dot-separated `uid:` in its front matter — every shipped front-matter type already exposes the field through `IFrontMatter`, so filling it opts the page in.

YAML

```
---
title: Configure the build pipeline
uid: how-to.build.configure
---
```

Then link to it with either form. The inline `<xref:uid>` defaults its link text to the target's `Title`:

MARKDOWN

```
See <xref:kitchen-sink.main.cross-references-b> for the other half of this pairing.
```

The anchor-style `[text](xref:uid)` form takes a custom label:

MARKDOWN

```
See the [cross-reference target page](xref:kitchen-sink.main.cross-references-b) for
details.
```

Both resolve to an ordinary `<a href="/canonical/path">` at request and build time, so moving or renaming the target file leaves the link intact as long as its `uid:` does not change.

Verify

- Run `dotnet run` and click each link shape on a page that uses them — relative, absolute, anchor, asset, and external links all navigate to the right target.
- Build for a sub-path with `dotnet run -- build --base-url /docs` and open any output page: every internal `href` starts with `/docs/` and the `<body>` carries a `data-base-url="/docs"` attribute (no trailing slash). Markdown that still hardcodes the prefix shows up as `/docs/docs/…`.
- Break a `uid:` on purpose — the `xref:` link renders with `data-xref-error="Reference not found"`, a warning appears in the dev diagnostic overlay, and `dotnet run -- build` surfaces it in the `BuildReport`.
- Run `dotnet run -- build` — the build report lists zero broken-link diagnostics.

Related

- Reference: Pennington.Generation.OutputOptions — `BaseUrl` and the rest of the build-output surface.
- Reference: Pennington.Infrastructure.IResponseProcessor — the response-stage rewriters and how they compose.
- Background: Cross-reference resolution — the two-phase uid resolver, ordering, and diagnostics.

CHAPTER 5

Theming

Recolor the site

Guides Swap palettes, override syntax-highlight colors, append site-wide rules, and tweak prose through MonorailCSS without leaving DocSite or BlogSite.

When the site needs a different palette, recolored code blocks, a change to prose rules, or a chunk of site-wide CSS, the options below live on `MonorailCssOptions`, the options record for Pennington's MonorailCSS integration. `DocSiteOptions` forwards `ColorScheme`, `SyntaxTheme`, `ExtraStyles`, and `CustomCssFrameworkSettings` to it directly, so most reskins set these on `DocSiteOptions` and never leave the template.

Before you begin

- A running Pennington site (see Create your first Pennington site if not).
- An `AddDocSite` or `AddBlogSite` host (both call `AddMonorailCss` internally); bare `AddPennington` requires a separate `AddMonorailCss` call.
- Familiarity with `MonorailCssOptions` and the named-palette defaults — see Pennington.MonorailCss.MonorailCssOptions.

The `ServiceConfiguration` helpers referenced below come from `examples/DocSiteKitchenSinkExample`.

Options

Pick `NamedColorScheme` for a Tailwind-named palette

Map the three palette slots — primary, accent, base — to named palettes from `MonorailCss.Theme.ColorNames`.

CSHARP

```
ColorScheme = new NamedColorScheme
{
    PrimaryColorName = ColorName.Indigo,
    AccentColorName = ColorName.Pink,
    BaseColorName = ColorName.Slate,
}
```

Pick `AlgorithmicColorScheme` for hue-driven palettes

`AlgorithmicColorScheme` synthesises primary, base, and accent palettes from one `PrimaryHue` plus a `Chroma` and a `Scheme` (of type `CoordinatingScheme`: Complementary, SplitComplementary, Triadic, Analogous). The whole site repigments by changing one number.

CSHARP

```
=>
new()
{
PrimaryHue = 220,
Chroma = 0.18,
Scheme = CoordinatingScheme.SplitComplementary,
}
```

Assign whichever scheme to `DocSiteOptions.ColorScheme`.

Override syntax-highlight colors with `SyntaxTheme`

`SyntaxTheme` holds the five palettes used by `.hljs-*` token classes — `Keyword`, `String`, `Variable`, `Function`, and `Comment`. It is independent of the brand `ColorScheme`. `SyntaxTheme.Default` ships Sky / Emerald / Rose / Amber / Slate; build a new record to substitute your own (every slot is required). Assign it to `DocSiteOptions.SyntaxTheme`, which forwards to `MonorailCssOptions.SyntaxTheme`.

CSHARP

```
SyntaxTheme = new SyntaxTheme
{
    Keyword = ColorName.Violet,
    String = ColorName.Teal,
    Variable = ColorName.Orange,
    Function = ColorName.Cyan,
    Comment = ColorName.Gray,
}
```

Append site-wide rules with `ExtraStyles`

`ExtraStyles` is a CSS string emitted verbatim above the generated utility stylesheet — `@font-face` declarations, utility overrides, or one-off selectors. Assign to `DocSiteOptions.ExtraStyles`.

CSHARP

```
=> """
@font-face {
font-family: 'DocSiteKitchenSinkDisplay';
font-style: normal;
font-weight: 100 900;
font-display: swap;
src: url(/fonts/display.woff2) format('woff2');
}
@font-face {
font-family: 'DocSiteKitchenSinkBody';
font-style: normal;
font-weight: 100 900;
font-display: swap;
src: url(/fonts/body.woff2) format('woff2');
}
article .feature-callout-demo { letter-spacing: 0.01em; }
"""
```

Tweak prose rules with `CustomCssFrameworkSettings`

`DocSiteOptions.CustomCssFrameworkSettings` post-processes the `CssFrameworkSettings` after the DocSite theme is applied — prose adjustments, color maps, or `apply` directives without leaving DocSite. The delegate receives the fully baked settings and returns the ones the framework is built from; use a `with` expression and `AddRange` / `SetItem` so DocSite's defaults (scrollbar utilities, prose rules) survive — a plain assignment clobbers them.

CSHARP

```
CustomCssFrameworkSettings = settings => settings with
{
    Applies = settings.Applies
        .SetItem(".callout", "rounded-lg border border-base-300 bg-base-50 px-4 py-3"),
}
```

For customizations DocSite does not cover, see What the DocSite and BlogSite templates wire for you. On a bare `AddPennington` host the same delegate sits on `MonorailCssOptions`, alongside `ColorScheme`, in the options you pass to `AddMonorailCss`:

CSHARP

```
=> new()
{
ColorScheme = new NamedColorScheme
{
PrimaryColorName = ColorName.Sky,
AccentColorName = ColorName.Emerald,
BaseColorName = ColorName.Slate,
},
CustomCssFrameworkSettings = settings => settings with
{
Applies = settings.Applies
.SetItem(".lab-tabs", "flex flex-col bg-base-50 border border-base-300 rounded-lg")
.SetItem(".lab-tabs-list", "flex flex-row gap-2 border-b border-base-200 px-3 pt-2")
.SetItem(".lab-tabs-button", "py-1.5 text-sm text-base-700 data-[selected=true]:text-
primary-700 data-[selected=true]:border-b data-[selected=true]:border-primary-600")
.SetItem(".lab-tabs-panel", "hidden data-[selected=true]:block px-3 py-3"),
},
}
```

Result

Every `bg-primary-*`, `text-accent-*`, `border-base-*` utility on the site resolves to the new palette on the next page load. Code-block tokens recolor independently when `SyntaxTheme` is set, and any rules from `ExtraStyles` appear at the top of `/styles.css` ahead of the generated utilities.

Verify

- Run `dotnet run` and visit any page. Inspect a `bg-primary-500` element; the rendered color matches the palette set above.
- Open a page with a fenced code block and inspect a keyword token (`.hljs-keyword`); its color matches the `SyntaxTheme.Keyword` palette, not the brand `ColorScheme`.
- Fetch `/styles.css` and confirm the rule added through `CustomCssFrameworkSettings` (the `.callout` selector above) is present, and that the scrollbar utilities (`scrollbar-thin`) are still there.
- Confirm the `ExtraStyles` block appears at the top of `/styles.css`, above the generated utility rules.

Related

- Reference: Pennington.MonorailCss.MonorailCssOptions
- Background: What the DocSite and BlogSite templates wire for you
- Background: MonorailCSS integration

Switch the body and heading typeface

Guides Drop self-hosted woff2 files into wwwroot, register @font-face rules, declare preload hints, and point DisplayFontFamily and BodyFontFamily at the new faces — or load the faces from an external provider instead.

Swap a DocSite's default display and body typefaces for custom faces, and prime them with preload hints so they're ready for first paint.

Before you begin

- A running DocSite built with `AddDocSite` / `UseDocSite`.
- A chosen font delivery strategy — self-hosted `.woff2` files or an external provider — with files or URLs ready.
- The CSS `font-family` name each face registers under.

For a working setup, see examples/DocSiteKitchenSinkExample. The example does not ship font binaries — supply your own.

Self-host the font files

Self-hosting keeps the faces on your origin — no third-party request, no external dependency on first paint. The four steps below run in order: each one builds on the file paths and `@font-face` names the previous step established.

Drop font files into `wwwroot/fonts/`

Place each `.woff2` file under `wwwroot/fonts/`. `UsePennington` wires `UseStaticFiles`, so each file becomes available at `/fonts/<file>.woff2`. The example references `/fonts/display.woff2` and `/fonts/body.woff2`.

Register `@font-face` rules via `ExtraStyles`

Emit the `@font-face` declarations into the generated stylesheet by returning them from an `ExtraStyles` helper. MonorailCSS prepends this content verbatim above its utility output, with each `src:` pointing at the `/fonts/...` path you exposed in step 1.

CSHARP

```
=> """
@font-face {
font-family: 'DocSiteKitchenSinkDisplay';
font-style: normal;
font-weight: 100 900;
font-display: swap;
src: url(/fonts/display.woff2) format('woff2');
}
@font-face {
font-family: 'DocSiteKitchenSinkBody';
font-style: normal;
font-weight: 100 900;
font-display: swap;
src: url(/fonts/body.woff2) format('woff2');
}
article .feature-callout-demo { letter-spacing: 0.01em; }
"""
```

Declare preload hints with `FontPreloads`

Pass a `FontPreload[]` to `DocSiteOptions.FontPreloads`. DocSite then emits a `<link rel="preload" as="font" crossorigin>` tag for each entry in the document head, which prevents the flash of fallback text on first paint.

CSHARP

```
=>
[
new FontPreload("/fonts/display.woff2"),
new FontPreload("/fonts/body.woff2"),
]
```

Point `DisplayFontFamily` and `BodyFontFamily` at the new faces

Set `DisplayFontFamily` on `DocSiteOptions` to the CSS stack led by the display face, and set `BodyFontFamily` to the stack led by the body face. The stack name must match the `font-family` declared in step 2. Include a `system-ui` or `sans-serif` fallback so pages still render gracefully if a file fails to load.

CSHARP

```
=> new()
{
SiteTitle = "Kitchen Sink Docs",
SiteDescription = "A wide-surface DocSite example that backs eighteen how-to pages.",
GitHubUrl = "https://github.com/usepennington/pennington",
CanonicalBaseUrl = "https://example.com/",
HeaderContent = """<a href="/" class="font-bold">Kitchen Sink Docs</a>""",
FooterContent = BuildFooter(),
ColorScheme = BuildColorScheme(),
DisplayFontFamily = "'DocSiteKitchenSinkDisplay', system-ui, sans-serif",
BodyFontFamily = "'DocSiteKitchenSinkBody', system-ui, sans-serif",
FontPreloads = BuildFontPreloads(),
ExtraStyles = BuildExtraStyles(),
ConfigureLocalization = ConfigureLocalization,
ConfigurePennington = RegisterApiSource,
Areas = BuildAreas(),
}
```

Load the faces from an external provider instead

To pull the faces from a hosted service (Google Fonts, Fontsource, a corporate CDN) rather than self-host, the `<link>` or `@import` goes in the document head through `DocSiteOptions.AdditionalHtmlHeadContent`, a raw-HTML string appended to `<head>`. This replaces steps 1 and 2 — the provider serves both the files and the `@font-face` rules.

CSHARP

```
new DocSiteOptions
{
    AdditionalHtmlHeadContent =
        """<link rel="stylesheet" href="https://fonts.example.com/css?
family=Display+Body">""",
    DisplayFontFamily = "'Display', system-ui, sans-serif",
    BodyFontFamily = "'Body', system-ui, sans-serif",
}
```

Steps 3 and 4 still apply: set `DisplayFontFamily` / `BodyFontFamily` to the family names the provider's CSS registers, and add `FontPreloads` entries pointing at the provider's `.woff2` URLs if you want the same first-paint priming. Provider-hosted preloads need the absolute font URL, not a `/fonts/...` path.

Match MonorailCSS utilities to your stacks

`DisplayFontFamily` and `BodyFontFamily` flow into the layout's `<body>` / heading styles directly. They do not feed the MonorailCSS theme, so utility classes like `font-sans` and `font-display` still resolve to whatever theme tokens MonorailCSS was configured with. When prose uses those utilities, also update the theme via `CustomCssFrameworkSettings` (or add overrides through `ExtraStyles`) so the utility-driven text agrees with the layout chrome. See Recolor the site for how to pass `CustomCssFrameworkSettings`.

Result

Body copy renders in the new body face and headings render in the new display face. The preload hints prime the browser cache before the stylesheet is parsed, so the first paint lands with the real faces in place — no fallback flash.

Verify

- Run `dotnet run` and open any page with the DevTools **Network** panel open. Filter to **Font**: `/fonts/display.woff2` and `/fonts/body.woff2` each show **Highest** in the **Priority** column and `preload` (rather than `link` or `script`) in the **Initiator** column, confirming the preload hint fired before the stylesheet pulled the face in.
- In the **Elements** panel, the **Computed** styles on the `<body>` resolve `font-family` to the body family; a heading (`<h1>`) resolves to the display family.
- Run `dotnet run -- build`. The generated `index.html` contains a `<link rel="preload" as="font" ...>` tag per `FontPreload`, and `/fonts/*.woff2` lands in `output/fonts/`.

Related

- Reference: DocSiteOptions — the full property list including `DisplayFontFamily`, `BodyFontFamily`, `FontPreloads`, `AdditionalHtmlHeadContent`, and `ExtraStyles`.
- Reference: FontPreload — the `Href` / `Type` record shape (defaults to `font/woff2`).
- How-to: Customize MonorailCSS colors, syntax theme, and prose styles — for the broader `ExtraStyles` story and for aligning utility-class font stacks with your new families.

Populate the blog homepage

Guides Populate the BlogSite homepage — hero block, My Work card, social-icon row, and top-nav links — from the four init-only properties on BlogSiteOptions.

When a BlogSite homepage needs its hero block, "My Work" card, social-icon row, and top-nav links populated in one pass, four init-only properties on `BlogSiteOptions` cover it — `HeroContent`, `MyWork`, `Socials`, and `MainSiteLinks`. Their record types (`HeroContent`, `Project`, `SocialLink`, `HeaderLink`) are catalogued in the BlogSiteOptions reference; this page shows how to fill them. For the hand-held walkthrough, see Add a hero, projects, and social links.

Before you begin

- A running BlogSite built with `AddBlogSite` / `UseBlogSite` (see Scaffold a blog with BlogSite if not).
- At least one post under `BlogContentPath` so the recent-posts slot is not empty (see Author your first post with BlogSiteFrontMatter).
- A single `AddBlogSite(() => new BlogSiteOptions { ... })` call to edit — the four sections are init-only properties on that same record literal.

For a working setup, see examples/BlogSiteHeroProjectsSocialsExample.

Options

Set `HeroContent` for the headline block

`HeroContent` is a two-field positional record (`Title`, `Description`) rendered at the top of `/`. `Description` is emitted as a `MarkupString` in `Home.razor`, so light HTML is permitted; plain prose works for most sites.

CSHARP

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddBlogSite(() => new BlogSiteOptions
{
    SiteTitle = "Hero Blog",
    SiteDescription = "A BlogSite tutorial app demonstrating hero, projects, and social
links.",
    CanonicalBaseUrl = "https://example.com",

    AuthorName = "Author Name",
    AuthorBio = "Writing about software, tools, and the occasional side project.",

    HeroContent = new HeroContent(
        Title: "Field notes from a weekend content engine",
        Description: "I build small tools for small problems. This is where I write about
them."),
});

var app = builder.Build();
app.UseBlogSite();
app.RunBlogSiteAsync(args).GetAwaiter().GetResult();
```

Fill `MyWork` with `Project` entries

`MyWork` takes a `Project[]`, where each `Project(Title, Description, Url)` renders as a linked entry in the "My Work" sidebar card. The array is rendered verbatim, so ordering entries in the initializer controls their display order.

CSHARP

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddBlogSite(() => new BlogSiteOptions
{
    SiteTitle = "Hero Blog",
    SiteDescription = "A BlogSite tutorial app demonstrating hero, projects, and social
links.",
    CanonicalBaseUrl = "https://example.com",

    AuthorName = "Author Name",
    AuthorBio = "Writing about software, tools, and the occasional side project.",

    HeroContent = new HeroContent(
        Title: "Field notes from a weekend content engine",
        Description: "I build small tools for small problems. This is where I write about
them."),

    MyWork =
    [
        new Project(
            Title: "Pennington",
            Description: "A tiny .NET content engine for docs and blogs.",
            Url: "https://github.com/example/pennington"),
        new Project(
            Title: "MonorailCSS",
            Description: "Utility-first CSS generation for Razor.",
            Url: "https://github.com/example/monorailcss"),
        new Project(
            Title: "Mdazor",
            Description: "Inline Razor components inside Markdown.",
            Url: "https://github.com/example/mdazor"),
    ],
});

var app = builder.Build();
app.UseBlogSite();
app.RunBlogSiteAsync(args).GetAwaiter().GetResult();
```

Wire `Socials` with the built-in icon fragments

`Socials` takes a `SocialLink[]`, where `SocialLink(Icon, Url)` pairs a `RenderFragment` with an `<a href>` target. The four built-in fragments — `GithubIcon`, `BlueskyIcon`, `LinkedInIcon`, `MastodonIcon` — are `static readonly` fields on `Pennington.BlogSite.Components.SocialIcons` and are passed directly without any wrapper type or component registration. Add a `using Pennington.BlogSite.Components;` directive so the field names resolve. The same block below also fills `MainSiteLinks` (the next section).

CSHARP

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddBlogSite(() => new BlogSiteOptions
{
    SiteTitle = "Hero Blog",
    SiteDescription = "A BlogSite tutorial app demonstrating hero, projects, and social
links.",
    CanonicalBaseUrl = "https://example.com",

    AuthorName = "Author Name",
    AuthorBio = "Writing about software, tools, and the occasional side project.",

    HeroContent = new HeroContent(
        Title: "Field notes from a weekend content engine",
        Description: "I build small tools for small problems. This is where I write about
them."),

    MyWork =
    [
        new Project(
            Title: "Pennington",
            Description: "A tiny .NET content engine for docs and blogs.",
            Url: "https://github.com/example/pennington"),
        new Project(
            Title: "MonorailCSS",
            Description: "Utility-first CSS generation for Razor.",
            Url: "https://github.com/example/monorailcss"),
        new Project(
            Title: "Mdazor",
            Description: "Inline Razor components inside Markdown.",
            Url: "https://github.com/example/mdazor"),
    ],

    Socials =
    [
        new SocialLink(SocialIcons.GithubIcon, "https://github.com/example"),
        new SocialLink(SocialIcons.BlueskyIcon,
"https://bsky.app/profile/example.bsky.social"),
        new SocialLink(SocialIcons.LinkedInIcon, "https://www.linkedin.com/in/example"),
        new SocialLink(SocialIcons.MastodonIcon, "https://hachyderm.io/@example"),
    ],

    MainSiteLinks =
    [
        new HeaderLink("Home", "/"),
        new HeaderLink("Archive", "/archive"),
        new HeaderLink("Tags", "/tags"),
    ],
});

var app = builder.Build();
app.UseBlogSite();
app.RunBlogSiteAsync(args).GetAwaiter().GetResult();
```

Populate `MainSiteLinks` for the top nav

`MainSiteLinks` takes a `HeaderLink[]`, where each `HeaderLink(Title, Url)` appears in both the site header and footer via `MainLayout.razor`. Use relative URLs (`/`, `/archive`, `/tags`) so `BaseUrlHtmlRewriter` can prefix them correctly on sub-path deployments. The `MainSiteLinks = [...]` block sits alongside `Socials` in the snippet above.

Result

The homepage at `/` renders the hero block above the post list, the "My Work" card and social-icon row in the right rail, and every `HeaderLink` in both the top nav and the footer. The four sections are independent; populating any one renders that section and leaves the rest at their template defaults.

Verify

- Run `dotnet run` and open `/`. The hero title and description appear at the top, and the "My Work" card lists each `Project` entry with a working link.
- The social-icon row under the card renders one icon per `SocialLink`, each linking to its `Url`. The top-nav and footer list every `HeaderLink`.
- Run `dotnet run -- build`. The generated `index.html` contains every hero/project/socials/nav string, and the build report shows no 500s.

Related

- Tutorial: Add a hero, projects, and social links — the hand-held walkthrough of the same four sections.
- Reference: BlogSiteOptions — the full property catalog (site metadata, content paths, feeds, fonts).
- Background: What the DocSite and BlogSite templates wire for you — what each template assembles and where the wiring stops.

CHAPTER 6

Versioning

Version a DocSite

Guides Ship /v1/ and /v2/ URL trees from one DocSite host, each with its own content area and its own reflection-based API reference.

To serve `/v1/` and `/v2/` URL trees from one DocSite host, give each version its own `ContentArea`. One area per version is the whole mechanism for prose-only docs — the Lay out content by version section below is all you need.

The rest of this page layers a per-version API reference on top, which is where the only real friction lives: NuGet allows one version of an assembly per project, so the off-version DLL needs a `<PackageDownload>` workaround. If you don't need a reflected API tree, stop after the areas section.

The recipe references `examples/VersionedDocSiteExample/`, which documents `Humanizer.Core` 2.8.26 alongside 2.14.1. For how `AddApiMetadataFromCompiledAssembly` and `AddApiReference` work on a single version, see Auto-generate an API reference tree for a class library.

Before you begin

- A DocSite host already wired with `AddDocSite` (see Scaffold a documentation site with DocSite).
- A decision about which version is the *active* `PackageReference`. That version resolves via `FromPackageReference("AssemblyName")`. Every other version is staged via `<PackageDownload>` and an explicit `AssemblyFiles` path.

Lay out content by version

Use one `ContentArea` per version. The `Slug` is both the URL prefix and the folder name under `Content/`, so files at `Content/v1/foo.md` route to `/v1/foo` and the sidebar renders an area selector that doubles as a version switcher.

CSHARP

```
public static void AddVersionedAreas(WebApplicationBuilder builder)
{
    builder.Services.AddDocSite(() => new DocSiteOptions
    {
        SiteTitle = "Humanizer Docs",
        SiteDescription = "Side-by-side documentation for two versions of Humanizer.Core,
with version-scoped content and a sidebar version selector.",
        GitHubUrl = "https://github.com/Humanizr/Humanizer",
        HeaderContent = """<a href="/" class="font-bold text-lg">Humanizer Docs</a>""",
        FooterContent = """<footer class="mt-16 py-8 text-center text-sm text-base-
500">Humanizer © Mehdi Khalili & contributors. Rendered by Pennington.</footer>""",
        Areas =
        [
            new ContentArea("v1", "v1"),
            new ContentArea("v2", "v2"),
        ],
    });
}
```

The `Areas` declaration is the only place the version names appear in the host wiring. Adding a `v3` later is two lines plus a `Content/v3/` folder.

A bare `/` request — anyone landing on the site root with no version prefix — falls through to the DocSite not-found page unless you give the root a page; add a `Content/index.md` (or a routed landing component) that redirects to or links the version you treat as current. Marking one version "latest" and showing a deprecation banner on older trees are content-level conventions, not host wiring: drop a shared `[!INCLUDE]` partial into each old version's pages for the banner, and point the root and header link at the current slug. See Forward visitors from a renamed page for the root-redirect mechanics.

Reference two versions of the same NuGet package

NuGet allows only one `<PackageReference>` per assembly per project. To document a second version, add a `<PackageDownload>` element pinned with square-bracket exact-version syntax. `<PackageDownload>` fetches the package into the NuGet cache without adding it to the compile graph, leaving the `<PackageReference>` version as the one resolved through the default load context.

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include="..\..\src\Pennington.DocSite\Pennington.DocSite.csproj" />
    <ProjectReference
Include="..\..\src\Pennington.DocSite.Api\Pennington.DocSite.Api.csproj" />
    <ProjectReference
Include="..\..\src\Pennington.ApiMetadata.Reflection\Pennington.ApiMetadata.Reflection.cspro
j" />
  </ItemGroup>
  <ItemGroup>
    <!-- v2 is the active PackageReference. FromPackageReference("Humanizer")
         resolves this through the default load context. -->
    <PackageReference Include="Humanizer.Core" />
    <!-- v1 is staged into the NuGet cache without being compiled against.
         PackageDownload is how NuGet expresses "fetch this version, but do not
         add it to the compile graph" — necessary because a single project
         cannot PackageReference two versions of the same assembly. The exact
         version pin (square brackets) is required by PackageDownload. -->
    <PackageDownload Include="Humanizer.Core" Version="[2.8.26]" />
  </ItemGroup>
  <ItemGroup>
    <Watch Include="Content\**\*.*" />
  </ItemGroup>
</Project>
```

In `Program.cs`, register one named provider per version, then pair each with an `AddApiReference` registration whose `RoutePrefix` nests under the matching area slug:

CSHARP

```
public static void AddVersionedApiReferences(WebApplicationBuilder builder)
{
    var nuGetPackages = Environment.GetEnvironmentVariable("NUGET_PACKAGES")
        ?? Path.Combine(Environment.GetFolderPath(Environment.SpecialFolder.UserProfile),
".nuget", "packages");
    var humanizerV1Dll = Path.Combine(nuGetPackages, "humanizer.core", "2.8.26", "lib",
"netstandard2.0", "Humanizer.dll");

    builder.Services.AddApiMetadataFromCompiledAssembly("humanizer-v1", opts =>
        opts.AssemblyFiles.Add(humanizerV1Dll));
    builder.Services.AddApiMetadataFromCompiledAssembly("humanizer-v2", opts =>
        opts.FromPackageReference("Humanizer"));

    builder.Services.AddApiReference("humanizer-v1", opts =>
    {
        opts.RoutePrefix = "/v1/api/";
        opts.TocTitle = "API reference";
    });
    builder.Services.AddApiReference("humanizer-v2", opts =>
    {
        opts.RoutePrefix = "/v2/api/";
        opts.TocTitle = "API reference";
    });
}
```

- The active reference uses `FromPackageReference("Humanizer")` — `Assembly.Load` finds the v2 DLL via the project's `deps.json`.
- The off-version uses `AssemblyFiles.Add(path)` with an explicit path under the NuGet global-packages folder. Read that folder from the `NUGET_PACKAGES` environment variable and fall back to the per-user default (`~/.nuget/packages` on Linux and macOS, `%USERPROFILE%\.nuget\packages` on Windows) — `Environment.GetFolderPath(SpecialFolder.UserProfile)` resolves the home directory on every platform. Inside it, the simple-name folder is lowercased; the version is the literal `<PackageDownload>` value; the TFM is whichever `lib/<tfm>/` the package ships.

The two registrations resolve as follows:

```
Provider name `RoutePrefix` Resolves     `humanizer-v1` `/v1/api/` `Humanizer.dll` 2.8.26
from the NuGet cache   `humanizer-v2` `/v2/api/` `Humanizer.dll` 2.14.1 via the active
`PackageReference`
```

The Mdazor components (`<ApiMemberTable>`, `<ApiSummary>`, …) are registered once and resolve metadata per page via the keyed provider, so two trees coexist with no further wiring.

Cross-link between versions

Each named registration emits xref uids under `reference.api.{name}.{slug}` — for example, `<xref:reference.api.humanizer-v1.string-humanize-extensions>` and `<xref:reference.api.humanizer-v2.string-humanize-extensions>`. Use the qualified form when a v2 content page links to a v1 type to show what changed, or vice versa.

Verify

- Run `dotnet run --project examples/VersionedDocSiteExample`.
- Visit `/v1/`, `/v2/`, `/v1/api/`, and `/v2/api/` — each renders independently.
- The startup log prints one `ApiReferenceIndex({name}): published N auto-discovered type pages` line per registration. `N` differs between versions when the two assemblies differ.
- Confirm the sidebar area selector switches between `v1` and `v2` while staying on the same page kind.

Related

- How-to: Auto-generate an API reference tree for a class library — the single-source backend setup this recipe builds on.
- Tutorial: Organize content with sections and areas — the area-driven URL prefix mechanism reused for version slugs.
- Reference: DI and middleware extension methods — `AddDocSite` and `DocSiteOptions` surface.

CHAPTER 7

Discovery

Serve docs and a blog from separate content roots

Guides Register more than one markdown root — either as DocSite areas or as chained AddMarkdownContent calls on a bare Pennington host — and keep them from overlapping.

When one markdown tree needs more than one content root — a `/docs/` section alongside a separate `/blog/` section, or a catch-all root paired with a specialized subtree — registering multiple content sources is the answer. The right recipe depends on the host: `AddDocSite` supports multiple folder-scoped sub-trees through `ContentArea` entries on a single `DocSiteFrontMatter` pipeline; bare `AddPennington` allows any number of chained `AddMarkdownContent<T>` calls with independent front-matter types. For a first site, start with Serve markdown through Blazor Pages.

Before you begin

- A working Pennington site (see Your first Pennington site if not).
- The chosen host extension — `AddDocSite` versus bare `AddPennington` — and the reason for that choice (What the DocSite and BlogSite templates wire for you).
- Familiarity with `IFrontMatter` basics (Define custom front-matter keys).

For a working DocSite multi-area setup, see examples/DocSiteKitchenSinkExample. For the bare `AddPennington` chained-sources recipe, see examples/MultipleSourcesExample.

Split a DocSite into areas

`AddDocSite` owns exactly one markdown pipeline keyed on `DocSiteFrontMatter`. To split that pipeline into folder-scoped sub-trees, populate `DocSiteOptions.Areas` with one `ContentArea` per slug — each slug becomes both the URL prefix and the top-level folder under `ContentRootPath`.

Declare the areas

Build the `ContentArea` list — one entry per slug, where the slug is both the URL prefix and the top-level content folder.

CSHARP

```
=>
[
new ContentArea("Main", "main"),
new ContentArea("API", "api"),
]
```

Wire the areas onto `DocSiteOptions`

Assign that list to `DocSiteOptions.Areas` so the single `DocSiteFrontMatter` pipeline discovers each folder as its own sub-tree.

CSHARP

```
=> new()
{
SiteTitle = "Kitchen Sink Docs",
SiteDescription = "A wide-surface DocSite example that backs eighteen how-to pages.",
GitHubUrl = "https://github.com/usepennington/pennington",
CanonicalBaseUrl = "https://example.com/",
HeaderContent = """<a href="/" class="font-bold">Kitchen Sink Docs</a>""",
FooterContent = BuildFooter(),
ColorScheme = BuildColorScheme(),
DisplayFontFamily = "'DocSiteKitchenSinkDisplay', system-ui, sans-serif",
BodyFontFamily = "'DocSiteKitchenSinkBody', system-ui, sans-serif",
FontPreloads = BuildFontPreloads(),
ExtraStyles = BuildExtraStyles(),
ConfigureLocalization = ConfigureLocalization,
ConfigurePennington = RegisterApiSource,
Areas = BuildAreas(),
}
```

Chain `AddMarkdownContent` on a bare host

On bare `AddPennington`, call `AddMarkdownContent<TFrontMatter>` once per source. Each call accepts its own `ContentPath`, `BasePageUrl`, and optional `SectionLabel`. Front-matter types can differ between sources.

Register the first source

CSHARP

```
md.ContentPath = "Content/docs";
md.BasePageUrl = "/docs";
md.SectionLabel = "Documentation";
```

Register a second source with a different front-matter type

CSHARP

```
md.ContentPath = "Content/blog";
md.BasePageUrl = "/blog";
md.SectionLabel = "Blog";
```

Carve out an overlapping subtree with `ExcludePaths`

When one source's `ContentPath` is a parent of another's, Pennington emits an overlap warning at startup because both pipelines would discover the inner tree and produce conflicting outputs. Adding `ExcludePaths` on the broader source gives the specialized source exclusive ownership of that subtree.

CSHARP

```
md.ContentPath = "Content";
md.BasePageUrl = "/";
md.ExcludePaths = ["blog"];
```

Verify

- Run `dotnet run` and visit each source's `BasePageUrl`. Pages render under both prefixes.
- Startup logs contain no `Markdown content source rooted at '…' overlaps…` warnings, or — when an overlap is intentional — the warning text names the subtree set aside for exclusion.
- Each source's pages appear under the correct `SectionLabel` / `ContentArea.Title` in the generated navigation.

Related

- Reference: PenningtonOptions.AddMarkdownContent
- Reference: DocSiteOptions.Areas and ContentArea
- Background: What the DocSite and BlogSite templates wire for you
- Extensibility: Source content from outside the file system

Tune what the search box returns

Guides Exclude pages from the index, weight document priority, and scope the indexed HTML region without replacing the search backend.

When the search index is already live but results contain nav or footer noise, a page appears that should be hidden, or relative document weight needs adjusting, the options below tune the index without touching the search client. For how the index is built, sharded, and queried, see How the search index is built and queried.

Before you begin

- A working Pennington site that serves `/search/en/index.json` (or the default locale code) — the search index entrypoint
- Pages using `DocSiteFrontMatter` or another `IFrontMatter` implementation (which carries the `Search` default member)
- The default locale code (from `LocalizationOptions`) — it is the suffix in the index filename

examples/DocSiteKitchenSinkExample ships with the DocSite-pinned `#main-content` selector and a `Content/main/hidden.md` fixture demonstrating `search: false`.

Options

Exclude a markdown page with `search: false`

Add `search: false` to the page's front matter. The index builder skips the page entirely while it continues to render at its URL and appear in the sidebar.

YAML

```
---
title: Internal draft
search: false
---
```

MARKDOWN

```
---
title: Not in search
description: This page is intentionally excluded from the search index.
sectionLabel: authoring
order: 220
search: false
uid: kitchen-sink.main.hidden
---

This page carries `search: false` in its front matter. It still renders
at its URL and still appears in the sidebar, but the search index
JSON does **not** contain it. Open `/search-index-en.json` to verify
— this title and body are absent from the `documents` array.

Pair this with `llms: false` on a separate page to carve the opposite
hole in `/llms.txt`.
```

Exclude a Razor `@page` with a metadata sidecar

Razor components do not carry YAML front matter, so `RazorPageContentService` loads a sibling `Foo.razor.metadata.yml` file. Place the sidecar next to the component; `search: false` there has the same effect as in a markdown page's front matter.

YAML

```
title: Internal Tools
search: false
```

Set the default document priority

`SearchIndexOptions.DefaultPriority` (default `5`) is the baseline weight every document starts from — `p` is a linear multiplier on a result's score, so a document with twice the priority needs only half the term relevance to tie. Raise it for sites whose content should outrank neighbors; lower it for auxiliary content. `AreaPriorities` and `PrefixPriorities` (below) override it per content area and per URL prefix; see Pennington.Search.SearchIndexOptions for the shipped defaults.

Under `AddDocSite` this property is reachable via the `ConfigurePennington` escape hatch (`ConfigurePennington = penn => penn.SearchIndex.DefaultPriority = …`), so this adjustment does not require dropping down to bare `AddPennington`.

Drop a URL territory below prose with a prefix priority

To rank a whole URL subtree below comparable prose — generated API reference is the usual case, where dozens of type pages repeat a term and bury the article that explains it — register the prefix in `SearchIndexOptions.PrefixPriorities`. The longest matching prefix wins, and its value *replaces* the area priority rather than stacking, so the subtree drops to an absolute low `p` even inside a boosted area.

CSHARP

```
services.AddDocSite(() => new DocSiteOptions
{
    SiteTitle = "My Docs",
    SiteDescription = "Project documentation",
    ConfigurePennington = penn =>
        penn.SearchIndex.PrefixPriorities["/imagesharp/api/"] = 3,
});
```

`AddApiReference` registers each reference tree's route prefix here automatically at priority `3`; set `ApiReferenceRegistrationOptions.SearchPriority` to choose a different value (`AddApiReference(o => o.SearchPriority = 2)`).

Override the content selector on DocSite

The selector scopes which HTML element's text becomes the search body — and the same element drives llms.txt sidecars and the build-time link audit, so chrome is stripped once. `DocSiteOptions.ContentSelector` defaults to `#main-content` to match the stock `MainLayout.razor`; set it after replacing the layout or to widen the indexed region. See What the DocSite and BlogSite templates wire for you for the cases that require dropping to bare `AddPennington`.

CSHARP

```
services.AddDocSite(() => new DocSiteOptions
{
    SiteTitle = "My Docs",
    SiteDescription = "Project documentation",
    ContentSelector = "article.prose",
});
```

Add query synonyms

To make a term also match alternates, set `SearchIndexOptions.Synonyms`. Keys and values are stemmed at build time and shipped in the entrypoint, so authors write natural words; the client expands query terms as it searches.

CSHARP

```
services.AddDocSite(() => new DocSiteOptions
{
    SiteTitle = "My Docs",
    SiteDescription = "Project documentation",
    ConfigurePennington = penn =>
        penn.SearchIndex.Synonyms = new Dictionary<string, string[]>
        {
            ["config"] = ["configuration", "settings"],
        },
});
```

Choose which facets the client can filter by

`SearchIndexOptions.Facets` selects the dimensions surfaced as filter chips: content area (the first URL segment after any locale prefix), section, and tags. Only area is on by default — it stays a short, stable list that reads well as chips. Section and tag vocabularies grow large enough to bury the filter bar, so opt into them when the extra filtering is worth the chips.

CSHARP

```
services.AddDocSite(() => new DocSiteOptions
{
    SiteTitle = "My Docs",
    SiteDescription = "Project documentation",
    ConfigurePennington = penn =>
        penn.SearchIndex.Facets = SearchFacetField.Area | SearchFacetField.Section |
SearchFacetField.Tags,
});
```

Result

The build emits the index under `/search/{locale}/`: an `index.json` entrypoint with the document table, facet labels, and ranking stats, plus `t-*.json` term shards and `f-*.json` per-page fragments. A document-table row in the entrypoint, after the knobs above are applied:

JSON

```
{
  "u": "/how-to/configuration/search/",
  "t": "Tune what the search box returns",
  "l": 142,
  "p": 10,
  "f": { "section": [0], "tag": [2, 5], "area": [1] }
}
```

The page body lives in its fragment (`f-{docId}.json`), fetched only when the page appears in results. Pages with `search: false` are absent from the table; each row's `p` is `DefaultPriority`, overridden by any matching `AreaPriorities` or `PrefixPriorities` entry.

Verify

- Run `dotnet run` and fetch `/search/{locale}/index.json`. The excluded page is absent from the `docs` table
- Add a second locale and observe one index tree per locale (`/search/en/index.json`, `/search/fr/index.json`). Registered-but-empty locales return a valid entrypoint with an empty `docs` array
- Fetch the matching `/search/{locale}/f-{docId}.json` and confirm its `body` contains only the scoped element's text (no header / sidebar / footer noise)
- After raising `DefaultPriority` (or registering an `AreaPriorities` / `PrefixPriorities` entry), fetch `index.json` and confirm the affected rows carry the new value in their `p` field
- After adding a synonym, fetch `index.json` and confirm the stemmed synonym map carries the entry; in the modal, query the key term and confirm pages that mention only the alternate now appear
- After enabling the section and tag facets, fetch `index.json` and confirm rows carry `section` and `tag` ids in their `f` object; in the modal, confirm the matching filter chips appear above the results

Related

- How-to: Add the search modal to a non-DocSite site — surface this index in a search UI on a bare host
- Reference: Front matter key reference
- Reference: SearchIndexOptions — the knobs this how-to touches; see also HighlightingOptions, LlmsTxtOptions, and OutputOptions
- Background: How the search index is built and queried
- Background: What the DocSite and BlogSite templates wire for you
- How-to: Make the site discoverable to LLM crawlers

Add the search modal to a non-DocSite site

Guides Light up the Pennington.UI search modal on a bare AddPennington host: reference the UI library, serve its scripts, and add a trigger element.

`AddDocSite` ships a search modal wired up for you. On a bare `AddPennington` host you wire it yourself — but the index and the modal already exist, so the work is three pieces of markup, not a search UI. `AddPennington` emits the index at `/search/{locale}/index.json`; `Pennington.UI` carries the modal in `scripts.js`; and `Pennington.MonorailCss` already safelists the modal's styles. This guide connects them. For how that index is built and queried, see How the search index is built and queried.

Before you begin

- A bare `AddPennington` host styled with MonorailCSS — see Style the site with MonorailCSS
- The host already serves `/search/{locale}/index.json` (it does, on every `AddPennington` host). To shape what that index contains, see Tune what the search box returns

The `BareHostSearchExample` mounts the shared Bramble corpus and lights up the modal with the wiring below.

Steps

Reference `Pennington.UI` and `Pennington.MonorailCss`.

`Pennington.UI` serves `scripts.js` (the modal) and, transitively, the `DeweySearch.Web` browser client as static web assets under `/_content/`. `Pennington.MonorailCss` carries the modal's styles.

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <RootNamespace>BareHostSearchExample</RootNamespace>
  </PropertyGroup>
  <ItemGroup>
    <ProjectReference Include="..\..\src\Pennington\Pennington.csproj" />
    <ProjectReference
Include="..\..\src\Pennington.MonorailCss\Pennington.MonorailCss.csproj" />
    <!-- Brings scripts.js (the search modal) and, transitively, the DeweySearch.Web
         browser client as static web assets under /_content. -->
    <ProjectReference Include="..\..\src\Pennington.UI\Pennington.UI.csproj" />
  </ItemGroup>
  <ItemGroup>
    <!-- No local Content/ — this example mounts the shared Bramble corpus.
         Watch that folder so dev-time live reload sees edits to it. -->
    <Watch Include="..\_shared\Bramble\Content\**\*.*" />
    <Watch Include="Components\**\*.razor" />
  </ItemGroup>
</Project>
```

Serve the `/_content/` static assets.

`UsePennington` mounts your content folders, not the RCL assets. Call `app.MapStaticAssets()` so `/_content/Pennington.UI/scripts.js` and `/_content/DeweySearch.Web/dewey-search.js` are served — `scripts.js` fetches the latter on demand when search first opens.

CSHARP

```
using BareHostSearchExample.Components;
using Pennington.FrontMatter;
using Pennington.Infrastructure;
using Pennington.MonorailCss;

var builder = WebApplication.CreateBuilder(args);

// A bare AddPennington host — no DocSite. AddPennington already emits the search
// index at /search/{locale}/index.json (term shards + per-page fragments); the
// only thing this example adds is the Pennington.UI search modal on top of it,
// wired up in MainLayout.razor. Content is the shared Bramble corpus mounted at
// the root, with the blog subtree excluded because its date/author front matter
// is not part of DocFrontMatter.
builder.Services.AddPennington(penn =>
{
    penn.SiteTitle = "Bramble";
    penn.ContentRootPath = "../_shared/Bramble/Content";

    penn.AddMarkdownContent<DocFrontMatter>(md =>
    {
        md.ContentPath = "../_shared/Bramble/Content";
        md.BasePageUrl = "/";
        md.ExcludePaths = ["blog"];
    });
});

builder.Services.AddMonorailCss(_ => new MonorailCssOptions
{
    ColorScheme = new NamedColorScheme
    {
        PrimaryColorName = ColorName.Emerald,
        AccentColorName = ColorName.Amber,
        BaseColorName = ColorName.Slate,
    },
});

builder.Services.AddRazorComponents();

var app = builder.Build();

app.UsePennington();
app.UseMonorailCss();
app.UseAntiforgery();

// Serve the static web assets Pennington.UI and DeweySearch.Web ship under /_content
// (scripts.js, dewey-search.js). UsePennington only mounts the content folders.
app.MapStaticAssets();
app.MapRazorComponents<App>();

await app.RunOrBuildAsync(args);
```

Load the script and add the trigger.

In your layout, load `scripts.js` (`defer`), set `data-default-locale` on `<body>`, and add a trigger element with `id="search-input"`. `scripts.js` self-initializes on load: it binds the click and the Ctrl/Cmd-K shortcut to that element, reads the locale attribute to locate the index, and pulls in `dewey-search.js` on demand the first time the modal opens — so you don't reference that script yourself.

RAZOR

```
@* The search modal ships in Pennington.UI/scripts.js and styles itself from the
   @apply blocks Pennington.MonorailCss safelists, so lighting it up on a bare
   (non-DocSite) host is three pieces of markup:
     1. scripts.js in <head> — it pulls in DeweySearch.Web's client on demand the
        first time search opens, so there's no separate dewey-search.js tag,
     2. data-default-locale on <body> — the client reads it to find the index, and
     3. a trigger element with id="search-input".
   scripts.js self-initializes on load and binds the click + Ctrl/Cmd-K shortcut. *@

@inherits LayoutComponentBase

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="stylesheet" href="/styles.css" />
    <script src="/_content/Pennington.UI/scripts.js" defer></script>
    <HeadOutlet />
</head>
<body class="bg-base-50 text-base-900 min-h-screen" data-default-locale="en">
    <div class="max-w-3xl mx-auto px-6 py-10">
        <header class="mb-8 flex items-center gap-4 border-b border-base-200 pb-4">
            <a class="text-lg font-bold text-primary-700" href="/">Bramble</a>
            <button type="button"
                    id="search-input"
                    class="ml-auto flex h-9 w-full max-w-xs items-center gap-2 rounded-lg
border border-base-200 bg-base-100 px-3 text-sm text-base-500 transition-colors
hover:border-base-300 hover:bg-base-200">
                <svg viewBox="0 0 24 24" fill="none" aria-hidden="true" class="h-4 w-4
shrink-0 stroke-current" stroke-width="2">
                    <circle cx="11" cy="11" r="7" />
                    <path stroke-linecap="round" d="m21 21-4.3-4.3" />
                </svg>
                <span class="flex-1 text-left">Search</span>
                <kbd class="ml-auto inline-flex items-center gap-0.5 rounded border border-
base-300 px-1.5 py-0.5 font-mono text-[11px] text-base-500">
                    <span>Ctrl</span><span>K</span>
                </kbd>
            </button>
        </header>
        <article class="prose">
            @Body
        </article>
        <footer class="mt-12 border-t border-base-200 pt-4 text-xs text-base-500">
            Bare Pennington host — the search modal comes from Pennington.UI.
        </footer>
    </div>
</body>
</html>
```

A single-locale site deployed at the domain root needs only `data-default-locale`. Two more `<body>` attributes cover the other cases:

- **`data-default-locale`** — the default locale code (for example `en`). The client falls back to this when no per-locale prefix matches, so it must always be present.
- **`data-locales`** — a comma-separated list of every locale code, in any order (for example `en,fr,de`). The client matches the first URL path segment against this list to pick which `/search/{locale}/` tree to query. Leave it empty or omit it on a single-locale site.
- **`data-base-url`** — the deploy sub-path prefix, with a leading slash and no trailing slash (for example `/docs`). The client prepends it when it fetches `dewey-search.js`, because a runtime-injected `<script>` does not pass through the base-url rewriter that fixes server-rendered links. Omit it for a domain-root deployment.

A multi-locale site deployed under `/docs` sets all three:

RAZOR

```
<body data-default-locale="en" data-locales="en,fr,de" data-base-url="/docs">
```

> **Note**
>
> No search CSS to write. The modal builds its DOM with class names (`.search-modal`, `.search-result`, …) that live only in `scripts.js`, so the MonorailCSS source scan never sees them. `AddMonorailCss` ships their styles anyway, so the modal is styled the moment it appears. A host that brings its own (non-MonorailCSS) stylesheet defines those class names there instead.

Verify

- Run the host and fetch `/_content/Pennington.UI/scripts.js` — it returns JavaScript (`text/javascript`), not the not-found page. If it returns HTML, `MapStaticAssets` is missing or the request is reaching the catch-all route
- Press Ctrl+K (or click the trigger). The modal opens **styled** — a centered dialog over a dimmed backdrop
- Type a query. Results deep-link to headings (`/page/#heading`) with a page breadcrumb, and the area chips filter by content area

Related

- How-to: Tune what the search box returns — configure the same index (exclude pages, weight priority, scope the indexed HTML)
- Background: How the search index is built and queried
- Background: What the DocSite and BlogSite templates wire for you
- Reference: SearchIndexOptions

Serve the site in multiple languages

Guides Populate LocalizationOptions, lay out translated content in locale subdirectories, register UI translations, and wire the locale-routing middleware.

When the site needs to ship in more than one language, the options below cover the wiring, content layout on disk, routing middleware, and UI translations — everything needed to take a single-locale site multilingual. For a first locale, the tutorial Add a second locale to your site walks through the same moving parts at a teaching pace.

Before you begin

- An existing Pennington site with at least one markdown page (see Create your first Pennington site if not).
- Either an `AddDocSite` host (which accepts a `ConfigureLocalization` callback) or a bare `AddPennington` host (which exposes `LocalizationOptions` directly on `PenningtonOptions.Localization`).
- Default-locale content already directly under `ContentRootPath` (not in a locale subfolder). Pennington treats the default locale as URL-root and every other locale as URL-prefixed.

For a complete reference setup, examples/BeyondLocaleExample has English under `Content/` and Spanish under `Content/es/`, wired with a single `ConfigureLocalization` action.

Wire localization end to end

Each subsection below is one part of the same setup, not an alternative — work through all five to take a single-locale site multilingual.

Populate `LocalizationOptions` with the default locale and every additional locale

On a DocSite host, set `DefaultLocale` and call `AddLocale` once per additional language inside `ConfigureLocalization`. On a bare `AddPennington` host, configure `PenningtonOptions.Localization` the same way. The default locale owns the URL root; each additional locale gets a URL prefix matching its code, so choose codes that read well in URLs.

CSHARP

```
ConfigureLocalization = loc =>
{
    loc.DefaultLocale = "en";
    loc.AddLocale("en", new LocaleInfo("English"));
    loc.AddLocale("es", new LocaleInfo("Español", HtmlLang: "es"));
},
```

See Pennington.Localization.LocalizationOptions for the `LocalizationOptions` members (`DefaultLocale`, `Locales`, `AddLocale`, `LocaleInfo`).

Mirror your content tree under `Content/<locale>/` for every non-default locale

Default-locale files stay directly under `ContentRootPath` with no prefix. For each additional locale, create a sibling folder named after the locale code and place translated files there, mirroring the default-locale filenames so the two are paired. Pages without a translation fall back to the default locale automatically, so shipping does not require a full translation pass.

MARKDOWN

```
---
title: Acerca de
description: Acerca de este ejemplo de DocSite localizado.
order: 20
---

Este es un DocSite mínimo que demuestra **URLs conscientes del idioma**.
Cada archivo markdown bajo `Content/` es la versión en inglés (el idioma
predeterminado). Cada archivo correspondiente bajo `Content/es/` es la
traducción al español.

Cuando un visitante navega a `/es/about`, el middleware
`LocaleDetectionMiddleware` elimina el prefijo `/es`, guarda `"es"` en
`LocaleContext`, y el `DocSiteContentResolver` del DocSite busca el markdown
en `Content/es/about.md`. Si falta un archivo en español, el resolvedor
recurre a la copia en inglés y marca la página como una traducción de
reserva para que el lector lo sepa.
```

Confirm `UseLocaleRouting` is in the pipeline

`UseDocSite` and `UseBlogSite` already register `UseLocaleRouting` as the first middleware — template hosts need no extra call. On a bare `AddPennington` host, call it before mapping endpoints; it handles locale detection and `UseRouting` together, so `LocaleDetectionMiddleware` can strip the locale prefix into `PathBase` ahead of endpoint matching without a separate `UseRouting` call.

CSHARP

```
app.UseLocaleRouting();
```

Add UI string translations through `TranslationOptions`

UI strings rendered by Razor components flow through `IStringLocalizer`, which Pennington backs with the in-memory `TranslationOptions` on `PenningtonOptions.Translations`. Register one entry per locale/key pair inside the `AddPennington` or `AddDocSite` configuration. Keys are free-form, and missing keys fall back to the default locale automatically.

CSHARP

```
builder.Services.AddPennington(options =>
{
    options.Translations.Add("en", "nav.home", "Home");
    options.Translations.Add("es", "nav.home", "Inicio");
});
```

See Pennington.Localization.TranslationOptions for the full `TranslationOptions` surface.

Surface the language switcher

On DocSite, the `LanguageSwitcher` component is already wired into `MainLayout.razor` and activates automatically when `LocalizationOptions.IsMultiLocale` is true; no extra markup required. On a bare host, drop `<LanguageSwitcher />` into the layout wherever the locale picker should appear:

RAZOR

```
<LanguageSwitcher />
```

See Utility components for the `LanguageSwitcher` parameter surface.

Result

The default locale owns the URL root; each additional locale gets a prefix that matches its code. For a site with English (default) and Spanish:

TEXT

```
/                       English home
/about/                 English about page
/es/                    Spanish home
/es/about/              Spanish about page
/es/missing-page/       falls back to the English page with a fallback banner
```

The language switcher in the layout lists one entry per registered locale, and each `IStringLocalizer["nav.home"]` resolves to the locale-specific value from `TranslationOptions`.

Verify

- Run `dotnet run` and visit `/`. The default-locale page renders at the URL root with no prefix.
- Visit `/{locale}/` (for example `/es/`) and confirm the translated home renders. Remove one translated file and verify the same URL falls back with a fallback banner.
- The site header (DocSite) or the layout (bare host) shows `LanguageSwitcher` with one entry per registered locale.
- UI strings registered through `TranslationOptions` resolve to the locale-appropriate value; missing keys fall back to the default locale.

Related

- Tutorial: Add a second locale to your site
- How-to: Flag missing and outdated translations in the build report and dev overlay
- Reference: LocalizationOptions
- Reference: TranslationOptions
- Background: Locale-aware URLs and content fallback

Paginate archive and tag listings

Guides Split long blog archives and tag pages into numbered pages, and apply the same pattern to a custom IContentService.

Long listings — a five-year archive, a popular tag with hundreds of posts — get unwieldy past a few dozen entries. BlogSite includes pagination for archives and tag pages; custom content services can reuse the shared `Pagination` component to do the same.

Before you begin

- A working Pennington site (see Your first Pennington site if not).
- For the custom-service half: a bare `AddPennington` host that renders Razor `@page` components — the same Blazor wiring the first-page tutorial sets up (`AddRazorComponents` + `MapRazorComponents`).
- Familiarity with custom content services — the recipe below adds one.

The custom-service recipe is implemented end to end in examples/PaginatedListingExample.

In BlogSite

Set `PostsPerPage` on `BlogSiteOptions`. Paginated URLs appear automatically.

CSHARP

```
builder.Services.AddBlogSite(() => new BlogSiteOptions
{
    SiteTitle = "My Blog",
    SiteDescription = "Posts and notes.",
    PostsPerPage = 10,
});
```

Resulting routes:

- `/archive` — canonical, page 1 (unchanged).
- `/archive/page/2/` , `/archive/page/3/` , … — emitted only when the post count exceeds `PostsPerPage` .
- `/tags/{tag}/` — canonical per-tag page (unchanged).
- `/tags/{tag}/page/2/` , … — emitted only for tags that exceed `PostsPerPage` .

The default is `10`. A non-positive value disables pagination entirely (all posts on one page). The home page is intentionally not paginated — it stays curated with the recent-post slot and a link to the archive.

In a custom content service

The pattern is three pieces: core's `PagedList<T>` record, a Razor page with two `@page` directives, and an `IContentService` that yields the paginated routes during discovery.

The PagedList record

Core ships `Pennington.Content.PagedList<T>` — a page slice plus the metadata the `Pagination` component needs to render prev/next and numbered links:

CSHARP

```
public sealed record PagedList<T>(
    IReadOnlyList<T> Items,
    int Page,
    int PageSize,
    int TotalItems)
{
    /// <summary>Total page count. At least <c>1</c> even when <see cref="TotalItems"/> is
zero.</summary>
    public int TotalPages => TotalItems <= 0 || PageSize <= 0
        ? 1
        : (int)Math.Ceiling(TotalItems / (double)PageSize);

    /// <summary>True when a page exists before <see cref="Page"/>.</summary>
    public bool HasPrevious => Page > 1;

    /// <summary>True when a page exists after <see cref="Page"/>.</summary>
    public bool HasNext => Page < TotalPages;
}
```

The Razor page

Two `@page` directives keep the canonical URL clean and add the paginated variant. Read the optional `Page` parameter, slice the source list through `ArticleResolver`, and render the shared `Pagination` component. `PageUrl` maps page 1 back to the canonical `/articles` URL.

RAZOR

```
@* Two @page directives: the canonical /articles URL plus the numbered /articles/page/N/
   variant. ArticleResolver slices the article list; the shared Pagination component renders
   the prev/numbered/next controls, with PageUrl mapping page 1 back to the canonical URL.
*@

@page "/articles"
@page "/articles/page/{Page:int}"
@inject ArticleResolver Resolver

<PageTitle>Articles@(_page is { Page: > 1 } p ? $" (page {p.Page})" : "")</PageTitle>

@if (_page is null)
{
    <p>No articles.</p>
    return;
}

<h1>Articles</h1>
<ul>
    @foreach (var article in _page.Items)
    {
        <li><a href="@article.Url">@article.Title</a></li>
    }
</ul>

<Pagination CurrentPage="@_page.Page" TotalPages="@_page.TotalPages" UrlFor="@PageUrl" />

@code {
    /// <summary>1-based page index from the route. Null on the canonical /articles URL
(page 1).</summary>
    [Parameter] public int? Page { get; set; }

    private PagedList<Article>? _page;

    protected override async Task OnInitializedAsync()
    {
        _page = await Resolver.GetPagedAsync(Page ?? 1, pageSize: 20);
    }

    private static string PageUrl(int page) => page <= 1
        ? "/articles"
        : $"/articles/page/{page}/";
}
```

`ArticleResolver` collects the markdown articles from every content source and slices them into pages. It is a plain service — not an `IContentService` — so it can inject `IEnumerable<IContentService>` directly with no risk of a cycle.

CSHARP

```
namespace PaginatedListingExample;

using Pennington.Content;
using Pennington.Pipeline;

/// <summary>One entry in the article listing.</summary>
/// <param name="Url">Canonical URL of the article.</param>
/// <param name="Title">Display title.</param>
public sealed record Article(string Url, string Title);

/// <summary>
/// Collects the markdown articles under <c>/articles/</c> from every registered
/// <see cref="IContentService"/> and serves them one page at a time. Injected by the
/// <c>ArticlesPage</c> Razor component. It is not registered as an <see
cref="IContentService"/>,
/// so the plain <see cref="IEnumerable{T}"/> injection here is safe — only the discovery
service
/// (which is in that set) has to resolve siblings lazily to avoid a cycle.
/// </summary>
public sealed class ArticleResolver(IEnumerable<IContentService> services)
{
    /// <summary>Returns the requested 1-based page of articles, ordered by URL.</summary>
    public async Task<PagedList<Article>> GetPagedAsync(int page, int pageSize)
    {
        var all = await CollectAsync();
        var skip = Math.Max(0, (page - 1) * pageSize);
        var items = all.Skip(skip).Take(pageSize).ToList();
        return new PagedList<Article>(items, page, pageSize, all.Count);
    }

    private async Task<List<Article>> CollectAsync()
    {
        var articles = new List<Article>();
        await foreach (var item in services.DiscoverAllAsync())
        {
            if (item.Source.Value is FileSource { IsMarkdown: true } &&
                item.Route.CanonicalPath.Value.StartsWith("/articles/"))
            {
                var url = item.Route.CanonicalPath.Value;
                articles.Add(new Article(url, item.Metadata?.Title ?? url));
            }
        }

        return articles.OrderBy(a => a.Url, StringComparer.Ordinal).ToList();
    }
}
```

The content service

A parameterized `@page` template (`{Page:int}`) is skipped by Pennington's automatic Razor route discovery. Emit each paginated route explicitly so the static build crawls them.

The service is itself one of the registered `IContentService` instances, so it must **not** constructor-inject `IEnumerable<IContentService>` — that forms a dependency cycle and throws at startup. Inject `IServiceProvider` instead and resolve the siblings on demand inside `DiscoverAsync`, excluding self with `!ReferenceEquals(s, this)`. This is the same pattern the library's own `SocialCardContentService` uses.

CSHARP

```
public sealed class ArticleListingContentService(IServiceProvider serviceProvider) :
IContentService, IMetaContentService
{
    private const int PageSize = 20;

    /// <inheritdoc/>
    public string DefaultSectionLabel => "";

    /// <inheritdoc/>
    public int SearchPriority => 0;

    /// <inheritdoc/>
    public async IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
    {
        // Resolve siblings on demand rather than via a ctor IEnumerable<IContentService>:
this
        // service is itself in that set, so the ctor injection would be a DI cycle. Filter
out every
        // meta-service (this one included) so the sibling walk can't recurse back into this
discovery
        // — reference-equality self-exclusion would miss the fresh transient copies
GetServices hands back.
        var siblings = serviceProvider.GetServices<IContentService>()
            .SourceServices()
            .ToList();

        var count = 0;
        await foreach (var item in siblings.DiscoverAllAsync())
        {
            if (item.Source.Value is FileSource { IsMarkdown: true } &&
                item.Route.CanonicalPath.Value.StartsWith("/articles/"))
            {
                count++;
            }
        }

        ContentSource source = new
RazorPageSource(typeof(ArticlesPage).AssemblyQualifiedName!);
        var totalPages = (int)Math.Ceiling(count / (double)PageSize);
        for (var page = 2; page <= totalPages; page++)
        {
            yield return new DiscoveredItem(
                new ContentRoute
                {
                    CanonicalPath = new UrlPath($"/articles/page/{page}/"),
                    OutputFile = new FilePath($"articles/page/{page}/index.html"),
                },
                source);
        }
    }
```

```
      /// <inheritdoc/>    public Task<ImmutableList<ContentTocItem>>
GetContentTocEntriesAsync()        => Task.FromResult(ImmutableList<ContentTocItem>.Empty);
/// <inheritdoc/>    public Task<ImmutableList<CrossReference>> GetCrossReferencesAsync()
=> Task.FromResult(ImmutableList<CrossReference>.Empty);}
```

Register the resolver and the service alongside the markdown source. `ArticleResolver` is a plain transient; `ArticleListingContentService` joins the `IContentService` set the same way any markdown source does.

CSHARP

```
builder.Services.AddTransient<ArticleResolver>();
builder.Services.AddTransient<IContentService, ArticleListingContentService>();
```

What ends up where

- **Sitemap.** Paginated routes appear in `sitemap.xml` automatically — they come from `DiscoverAsync` as HTML routes and `SitemapService` includes everything that isn't a redirect or llms-only sidecar.
- **Search index and llms.txt.** Excluded by default: `BlogSiteContentService` and the custom service above return empty `GetIndexableEntriesAsync()` (the default forwards to `GetContentTocEntriesAsync()`), so their routes never enter the search or llms paths. If a custom service does emit indexable entries for paginated routes, set `ExcludeFromSearch = true` and `ExcludeFromLlms = true` on those entries.
- **Navigation tree.** Same — paginated routes have no TOC entry, so they don't show in the sidebar or breadcrumbs.

Verify

- Run `dotnet run` and visit `/articles`. The first 20 articles render, with the `Pagination` controls below them.
- Visit `/articles/page/2/`. The remaining articles render and the control highlights page 2 — confirming `ArticleListingContentService.DiscoverAsync` emitted the overflow route.
- Run `dotnet run -- build` and open `sitemap.xml` in the output directory. It lists `/articles/page/2/` alongside the individual article URLs, because the route flows through `DiscoverAsync` as an HTML route.

Related

- Reference: BlogSiteOptions.PostsPerPage
- Background: Content pipeline overview
- Extensibility: Source content from outside the file system

Flag missing and outdated translations in the build report and dev overlay

Guides Register AddTranslationAudit so missing and stale per-locale translations surface in the build report and the dev overlay, gated by git commit history.

Once a site ships in more than one language, translations drift — a page gets reworded in the default locale and its counterpart silently falls behind, or a new page never gets translated at all. `AddTranslationAudit` registers an `IBuildAuditor` that pairs every default-locale page with its translations and reports the gaps, both per-page in the dev overlay and site-wide in the build report. Set the locales up first with Serve the site in multiple languages.

Before you begin

- A multi-locale site with default-locale content and at least one locale subfolder (see Serve the site in multiple languages).
- A git repository. The auditor reads commit dates to decide whether a translation is *outdated*; without git it still reports *missing* files but skips staleness checks.

Install the package

`AddTranslationAudit` ships in its own package, separate from the core library and the site templates. Add it to the host project:

BASH

```
dotnet add package Pennington.TranslationAudit
```

Register the auditor

`AddTranslationAudit` needs no other wiring — the auditor flows through the same audit cache the dev overlay reads, so one call next to the rest of your service registration is enough. The repository auto-discovers from the current working directory.

CSHARP

```
builder.Services.AddTranslationAudit();
```

Configure what gets audited

`AddTranslationAudit` takes an optional configuration action exposing `TranslationAuditOptions`:

CSHARP

```
builder.Services.AddTranslationAudit(options =>
{
    options.IncludedLocales = ["es"];
    options.OutdatedSeverity = DiagnosticSeverity.Error;
});
```

- `RepositoryPath` — absolute path to the git repository root. Defaults to `null`, which auto-discovers by walking up from the current directory.
- `IncludedLocales` — the locale codes to audit. Defaults to `null`, which reports every non-default locale registered in `LocalizationOptions`.
- `MissingSeverity` — severity for "no translation file exists". Defaults to `DiagnosticSeverity.Warning`.
- `OutdatedSeverity` — severity for "translation predates the source's last commit". Defaults to `DiagnosticSeverity.Warning`.
- `ReportMissing` — set to `false` to suppress the missing-file diagnostics and audit only staleness. Defaults to `true`.

Result

The audit surfaces in two places from one registration. During `dotnet run -- build`, each gap is a line in the build report's diagnostics:

TEXT

```
Build Complete — 11 pages in 0.5s
  11 pages generated
  1 warnings
WARNINGS
  /getting-started/: Missing Español (es) translation for "Getting Started" (/getting-
started/).
```

During `dotnet run`, the same diagnostics attach per-page: visiting a page with a missing or outdated translation lights up the dev overlay badge (`#penn-diag-root`, bottom-right) with the count.

When no git repository is found, the auditor logs `no git repository found at or above '<path>'. Translation status will treat every file as untracked.` once at startup. The *missing* check still runs because it only touches the filesystem; every *outdated* check is skipped because both commit lookups resolve to `null`.

Verify

- Run `dotnet run --project examples/BeyondTranslationAuditExample -- build` and confirm the build report lists the missing `es` translation.
- Run `dotnet run --project examples/BeyondTranslationAuditExample`, visit `/es/getting-started/`, and confirm the overlay badge shows one warning.

Related

- Tutorial: Add a second locale to your site
- How-to: Serve the site in multiple languages
- Background: Locale-aware URLs and content fallback

CHAPTER 8

Feeds

Make the site discoverable to LLM crawlers

Guides Expose a stripped-markdown /llms.txt index plus per-page sidecars so LLM crawlers and agents can ingest your site without scraping HTML.

Expose a `/llms.txt` index plus per-page stripped-markdown sidecars so LLM crawlers and agents can read the site without scraping HTML. On `AddDocSite` hosts the wiring is automatic; on bare `AddPennington` hosts a single `AddLlmsTxt(...)` call enables it.

Before you begin

- A working Pennington site (see Serve markdown through Blazor Pages if not).
- An `AddDocSite` host (LLM wiring is automatic) or a bare `AddPennington` host (needs an explicit `AddLlmsTxt(...)` call — see the "Bare Pennington" section below). The choice rationale is covered in What the DocSite and BlogSite templates wire for you.

Options

(Bare Pennington) Enable `LlmsTxtOptions` with `AddLlmsTxt`

On a bare host nothing is wired until you ask for it: call `penn.AddLlmsTxt(...)` once to turn on the index and per-page markdown. `AddDocSite` hosts skip this — the wiring is automatic. The options surface (`GenerateFullFile`) is documented at Pennington.LlmsTxt.LlmsTxtOptions. The chrome-stripping selector lives one layer up at `penn.SiteProjection.ContentSelector` — it is shared with the search index so both channels see the same body element.

CSHARP

```
opts.GenerateFullFile = false;
```

Set `GenerateFullFile = true` to also emit `/llms-full.txt` — every sidecar concatenated into one file, useful for one-shot ingest by agents that cannot follow per-page links. Off by default because the file can be large.

Report the documented version with `SiteVersion`

The `/llms.txt` front door stamps a version line so a crawler can tell which release the content describes. By default that is Pennington's own package version, emitted as `penningtonVersion:` — which says what generated the file, not what it documents. When your site documents a specific library or product, set `PenningtonOptions.SiteVersion` to that subject's version; the front door then emits `version:` and drops `penningtonVersion:`.

CSHARP

```
penn.SiteTitle = "Spectre.Console";
penn.SiteVersion = SpectreVersion.FromReferencedAssembly().Display; // "0.57"
```

Resolve the value however suits the site — a constant, a build property, or the informational version of a referenced assembly.

(DocSite) Opt a page out with `llms: false`

Every non-draft page is included in the index by default (`Llms = true`). Setting `llms: false` in a page's front matter causes `LlmsTxtService` to skip it when assembling `/llms.txt` and its sidecar markdown. The page still renders, appears in the sidebar, and participates in search unless `search: false` is also set. `Content/main/llms-hidden.md` in `examples/DocSiteKitchenSinkExample` is a working opted-out page:

MARKDOWN

```
---
title: Not in llms.txt
description: This page is intentionally excluded from llms.txt.
sectionLabel: authoring
order: 230
llms: false
uid: kitchen-sink.main.llms-hidden
---

This page carries `llms: false` in its front matter. It still appears
in the sidebar and in search results, but `/llms.txt` does **not**
list it. The content-stripping llms generator skips pages whose
`Llms` flag is `false` when assembling its index of documents.
```

CSHARP

```
public bool Llms { get; init; } = true;
```

Point the chrome-stripping selector at a custom wrapper

The selector that picks the body element out of the rendered page — for a different article wrapper or a non-DocSite layout — is `DocSiteOptions.ContentSelector`. It defaults to `#main-content` and is overridable without leaving DocSite. The same selector drives the search index, llms.txt sidecars, and the build-time link audit, so chrome is stripped once. (On a bare host the equivalent knob is `penn.SiteProjection.ContentSelector`, shown above.) See What the DocSite and BlogSite templates wire for you for cases that do require bare `AddPennington`.

Split content per-fragment with `humans-only` / `robots-only`

For finer control than page-level opt-out, two paired classes mark a fragment as intended for one audience. Both ship in the MonorailCSS base styles — no registration needed.

- `humans-only` — visible in the browser, stripped from the llms.txt extraction.
- `robots-only` — hidden in the browser via `display: none`, kept in the llms.txt extraction.

RAZOR

```
<div class="humans-only">
  <InteractiveTour />
</div>

<div class="robots-only">
  <p>Full method signature: <code>Task&lt;Result&gt; ProcessAsync(Options options,
CancellationToken ct = default)</code>.</p>
</div>
```

The classes work anywhere in the rendered page — markdown bodies, Razor components, auto-generated reference pages.

Result

`/llms.txt` lists each indexed page as a markdown link grouped by section, and each page gets a co-located markdown copy at `<page>.md` — the page's own URL with `.md` appended, beside its output folder (the root page lands at `/index.md`). Links are fully qualified when `PenningtonOptions.CanonicalBaseUrl` is set (or `build --base-url https://…` is passed); otherwise they fall back to root-relative paths so an agent that fetched `/llms.txt` can still resolve them against the origin.

A typical front door — a metadata block, a `## Map` of any subtrees split into their own `{prefix}llms.txt`, then the nav-grouped links:

MARKDOWN

```
# Pennington Docs

> Content engine library for .NET.

site: https://docs.example.com/
canonical: https://docs.example.com/llms.txt
generated: 2026-06-09 14:02 UTC
penningtonVersion: 0.1.0

## Map

- [Reference](https://docs.example.com/reference/llms.txt) (96 entries, ~120k tokens) — API surface, host extensions, front-matter keys, Markdig extensions, UI components, diagnostics codes.

## Tutorials

- [Your first Pennington site](https://docs.example.com/tutorials/getting-started/first-site.md): Build a static site from a single markdown file.
- [Add a second locale](https://docs.example.com/tutorials/beyond-basics/add-a-locale.md): Ship the same content in a second language.

## How-to

- [Switch the body and heading typeface](https://docs.example.com/how-to/configuration/fonts.md): Self-host woff2, declare preloads, point the family options at the new faces.
```

The `## Map` block appears only when a subtree is declared (a folder's `_meta.yml` carries an `llms:` block, or a content service registers one) — those leaves move to `/reference/llms.txt` and the front door keeps just the see-also line above. A site with no subtrees jumps straight from the metadata block to the nav-grouped links.

Verify

- Run `dotnet run` and fetch `/llms.txt`. Expect a metadata block, a `## Map` section listing per-subtree splits with token estimates, then nav-grouped links — and no line for any page marked `llms: false`
- Fetch one per-page markdown copy (for example, `/<page>.md`). Expect a YAML header with `canonical_url`, `content_hash`, `tokens`, and the body stripped to clean markdown
- With `GenerateFullFile = true`, fetch `/llms-full.txt`. Expect every sidecar concatenated in one response

Related

- Reference: LlmsTxtOptions
- How-to: Tune what the search box returns
- Background: What the DocSite and BlogSite templates wire for you

Publish an RSS feed

Guides Confirm /rss.xml is on, give every post a date so it appears in the channel, and point CanonicalBaseUrl at your production origin so links resolve.

`/rss.xml` is wired by `UseBlogSite` and enabled by default — `BlogSiteOptions.EnableRss` defaults to `true`. Two things break a working feed: a post missing `date:` (silently dropped from the channel), and an unset `CanonicalBaseUrl` (feed links emit relative URLs that aggregators cannot follow). The feed endpoint ships with the BlogSite template; to emit a feed from a bare `AddPennington` host or any non-blog content type, see Publish a custom feed from a content service.

Before you begin

- A working BlogSite (see Scaffold a blog with BlogSite if not)
- Posts under `Content/Blog/` that parse as `BlogSiteFrontMatter`
- A known production origin (such as `https://blog.example.com`). `CanonicalBaseUrl` needs the scheme and no trailing slash

---

Options

Give every post a `date:`

The `/rss.xml` feed builds the channel from posts where `Date` is non-null, ordered by descending date. A post without `date:` renders normally at its URL but does not appear in the feed. Use ISO-8601 (`2024-01-15`) so YAML parses the value as a `DateTime`.

Minimal front matter for a post that appears in the feed:

YAML

```
title: Getting started with Pennington
description: A first-post walkthrough.
date: 2024-01-15
author: Jamie Rivers
tags: [pennington, getting-started]
```

Set `CanonicalBaseUrl` to your production origin

The feed prefixes every `<link>` and `<guid>` with the canonical base. Use the production scheme and host with no trailing slash:

CSHARP

```
new BlogSiteOptions
{
    CanonicalBaseUrl = "https://blog.example.com",
    // ...
}
```

Result

`/rss.xml` returns an RSS 2.0 channel listing every dated post, newest first, with absolute URLs:

XML

```
<rss version="2.0">
  <channel>
    <title>Jamie's Blog</title>
    <link>https://blog.example.com/</link>
    <description>Notes on shipping software.</description>
    <item>
      <title>Getting started with Pennington</title>
      <link>https://blog.example.com/blog/getting-started-with-pennington/</link>
      <guid>https://blog.example.com/blog/getting-started-with-pennington/</guid>
      <pubDate>Mon, 15 Jan 2024 00:00:00 +0000</pubDate>
      <description>A first-post walkthrough.</description>
    </item>
  </channel>
</rss>
```

Verify

- Run `dotnet run` and fetch `/rss.xml`. Expect a `<rss version="2.0">` document with one `<item>` per dated post, newest first
- Inspect one `<item>`. `<link>` and `<guid>` start with the `CanonicalBaseUrl`, not a relative `/blog/...` path
- Posts without a `date:` field are absent from the channel. When an expected post is missing, add `date:` to its front matter
- After a static build (see Build a static site), `output/rss.xml` exists and carries the same dated items with absolute `CanonicalBaseUrl` links

Related

- How-to: Publish a custom feed from a content service
- Reference: Pennington.BlogSite.BlogSiteOptions
- Reference: Built-in BlogSite routes
- Reference: Front matter key reference

Publish a custom feed from a content service

Guides Build the same RSS pattern BlogSite uses for /rss.xml — a content service that caches records, an XML builder method, and a MapGet endpoint — for podcast episodes, conference sessions, changelogs, or any non-blog content type.

`BlogSiteOptions.EnableRss` only applies to `BlogSiteFrontMatter` records. For any other content type — podcast episodes, conference sessions, a changelog — reuse the pattern `BlogSite` builds on: a content service caches the records, a `Task<string>` builder turns them into feed XML, and a `MapGet` endpoint serves that XML. Every `MapGet` endpoint is fetched and baked during the static build, so the feed file lands in `output/` next to every other page with no extra registration.

The reference implementation is core's `RssFeedWriter.WriteXml` — called from `BlogPostQuery.GetRssXmlAsync` and the `MapGet` in `UseBlogSite`. This guide walks the three points you adapt in that pair, so the same shape can carry a podcast feed (with the iTunes namespace), an events feed (with iCalendar enclosures), or any custom format.

Before you begin

- A bare `AddPennington` host (see Create your first Pennington site) or any host where the records come from a custom `IContentService` — see Source content from outside the markdown pipeline for the discovery shape.
- `CanonicalBaseUrl` set on `PenningtonOptions`, `DocSiteOptions`, or `BlogSiteOptions`. Without it, `<link>` and `<guid>` emit relative URLs that aggregators cannot follow.
- For the BlogSite-shipped `/rss.xml`, use Publish an RSS feed instead — this guide is for the other shapes.

Build the feed XML on the content service

Order the cached records and emit XML with `System.Xml.Linq`. Core's `RssFeedWriter.WriteXml` — which the BlogSite feed reuses — is the reference body:

CSHARP

```
public static string WriteXml(
    string siteTitle,
    string siteDescription,
    string? canonicalBaseUrl,
    IEnumerable<RssFeedItem> items)
{
    var canonicalBase = canonicalBaseUrl?.TrimEnd('/') ?? string.Empty;
    XNamespace atom = "http://www.w3.org/2005/Atom";

    var channel = new XElement("channel",
        new XElement("title", siteTitle),
        new XElement("link", string.IsNullOrEmpty(canonicalBase) ? "/" : canonicalBase +
"/"),
        new XElement("description", siteDescription));

    if (!string.IsNullOrEmpty(canonicalBase))
    {
        channel.Add(new XElement(atom + "link",
            new XAttribute("href", canonicalBase + "/rss.xml"),
            new XAttribute("rel", "self"),
            new XAttribute("type", "application/rss+xml")));
    }

    var ordered = items
        .Where(i => i.PublishDate.HasValue)
        .OrderByDescending(i => i.PublishDate!.Value);

    foreach (var item in ordered)
    {
        var url = string.IsNullOrEmpty(canonicalBase)
            ? item.Url.Value
            : canonicalBase + item.Url.Value;

        var entry = new XElement("item",
            new XElement("title", item.Title),
            new XElement("link", url),
            new XElement("guid", new XAttribute("isPermaLink", "true"), url));

        if (!string.IsNullOrEmpty(item.Description))
        {
            entry.Add(new XElement("description", item.Description));
        }

        if (item.PublishDate.HasValue)
        {
            entry.Add(new XElement("pubDate",
item.PublishDate.Value.ToUniversalTime().ToString("r")));
        }

        if (!string.IsNullOrEmpty(item.Author))
        {
```

```
        channel.Add(entry);    }      var rss = new XElement("rss",        new
XAttribute("version", "2.0"),        new XAttribute(XNamespace.Xmlns + "atom",
atom.NamespaceName),        channel);      var doc = new XDocument(new XDeclaration("1.0",
"utf-8", null), rss);    return doc.Declaration + Environment.NewLine + doc;}
```

The pieces to adapt for your records:

- **The cache.** `DiscoverAsync` and the feed builder read from one cached list loaded once per generation, so the source files are parsed once and both paths see the same records. The `Lazy<T>` cache that already backs `DiscoverAsync` works here without changes.
- **The filter.** BlogSite drops posts without a `Date`. Replace this with whatever predicate keeps an entry in the feed (`IsPublished`, `Status == Released`, future-date skip via `TimeProvider`).
- **The ordering.** Newest-first is conventional for RSS; podcast aggregators expect it.
- **Absolute URLs.** Prefix every `<link>` and `<guid>` with `canonicalBase`. Relative paths break in feed readers.
- **The atom self-link.** `<atom:link rel="self" .../>` tells readers where the feed canonically lives. Match the URL you map below.
- **Per-item elements.** Keep `<title>`, `<link>`, `<guid>`. Add what your content type needs: `<category>` per tag, `<enclosure>` for media attachments, namespaced elements for iTunes/Atom/Dublin Core.

Wire DI so the endpoint and the discovery list share one instance

Register the concrete service, then forward `IContentService` to the same instance with a transient indirection. Two separate registrations would let the container hand the endpoint a fresh copy with a cold cache:

CSHARP

```
// File-watched when the service reads from disk; AddSingleton<T>() when the
// data source is in-process. The transient IContentService wrapper resolves
// against the current factory-managed instance so file-change recreates flow
// through to both the endpoint and the pipeline.
services.AddFileWatched<PodcastContentService>();
services.AddTransient<IContentService>(sp =>
    sp.GetRequiredService<PodcastContentService>());
```

`AddSingleton<IContentService>` here would cache the first file-watched copy and never refresh — the transient wrapper avoids that trap. The full lifetime contract for `AddFileWatched<T>` and the stale-data failure mode is in Register the service.

Map the endpoint

Inject the concrete service into a `MapGet` handler that returns the XML with the right MIME type:

CSHARP

```
app.MapGet("/feed.xml", async (PodcastContentService service) =>
    Results.Content(await service.GetRssXmlAsync(), "application/rss+xml"));
```

Two reasons this single line carries both dev and build:

- **Dev mode** serves `/feed.xml` straight from the handler.
- **Static build** fetches every `MapGet` endpoint over HTTP through the live pipeline and writes each body to `output/` — so `output/feed.xml` is baked from the same handler. No artifact-service registration is needed.

Reach for `IArtifactContentService` instead when the URL set is dynamic or derived from the rendered corpus — search shards and llms.txt files ship that way. See Emit generated output artifacts for that shape.

Adapt for podcast feeds (iTunes namespace)

A podcast RSS feed extends the same XML with the iTunes namespace plus per-item duration, episode number, and enclosure elements. Declare the namespace on the root and add the children inside the per-item loop:

CSHARP

```
XNamespace atom = "http://www.w3.org/2005/Atom";
XNamespace itunes = "http://www.itunes.com/dtds/podcast-1.0.dtd";

var rss = new XElement("rss",
    new XAttribute("version", "2.0"),
    new XAttribute(XNamespace.Xmlns + "atom", atom.NamespaceName),
    new XAttribute(XNamespace.Xmlns + "itunes", itunes.NamespaceName),
    channel);

// Per-item additions inside the feed builder's item loop:
entry.Add(
    new XElement(itunes + "duration", episode.Duration.ToString(@"hh\:mm\:ss")),
    new XElement(itunes + "episode", episode.EpisodeNumber),
    new XElement(itunes + "season", episode.SeasonNumber),
    new XElement("enclosure",
        new XAttribute("url", absoluteAudioUrl),
        new XAttribute("length", episode.AudioBytes),
        new XAttribute("type", "audio/mpeg")));
```

Channel-level iTunes elements (`<itunes:image>`, `<itunes:category>`, `<itunes:owner>`) sit alongside the existing `<title>` / `<link>` / `<description>` block. Apple's Podcasters Connect page is the authoritative list.

Adapt for Atom feeds

Atom 1.0 uses a different root and element vocabulary. The shape is identical — same cache, same builder method, same `MapGet` — only the XML changes. The sketch below shows the element structure; `canonicalBase`, `ordered`, and `absoluteUrl` are the same locals the RSS builder above sets up, dropped here for focus:

CSHARP

```
XNamespace atom = "http://www.w3.org/2005/Atom";

var feed = new XElement(atom + "feed",
    new XElement(atom + "title", _options.SiteTitle),
    new XElement(atom + "id", canonicalBase + "/"),
    new XElement(atom + "updated", DateTime.UtcNow.ToString("o")));

foreach (var entry in ordered)
{
    feed.Add(new XElement(atom + "entry",
        new XElement(atom + "title", entry.Title),
        new XElement(atom + "id", absoluteUrl),
        new XElement(atom + "updated", entry.Date.ToString("o")),
        new XElement(atom + "link", new XAttribute("href", absoluteUrl))));
}
```

Serve at a separate path (`/atom.xml`) with `application/atom+xml`. Nothing stops a site from publishing both RSS and Atom — register two endpoints against the same service.

Verify

- Run `dotnet run` and fetch `/feed.xml`. The response is the right MIME type with one item per record.
- Run `dotnet run -- build output` and confirm `output/feed.xml` exists with the same body. The build crawler reuses the live endpoint.
- Validate the XML externally — `xmllint --noout feed.xml` catches well-formedness errors. For podcasts, run the file through Apple's podcast validator before submitting to directories.
- Edit a source record and refetch `/feed.xml` in dev. The file-watched cache rebuilds and the change appears without a restart.

Related

- How-to: Publish an RSS feed (BlogSite)
- How-to: Source content from outside the file system
- How-to: Source content from a remote API
- How-to: Emit generated output artifacts
- Background: The content pipeline and union types

Publish a sitemap

Guides Expose an auto-built /sitemap.xml that enumerates every canonical URL, skips drafts and redirects, and uses front-matter dates for lastmod.

`/sitemap.xml` is registered and served automatically on every `AddPennington`-based host; a working site already emits one. The options below tune what crawlers see — absolute `<loc>` values, draft/redirect exclusion, or turning the sitemap off on BlogSite. For a first site, start with Create your first Pennington site.

Before you begin

- A working Pennington site (see Create your first Pennington site if not)
- Pages using an `IFrontMatter` implementation — `DocFrontMatter`, `BlogFrontMatter`, or a custom one — so `IsDraft` and (optionally) `Date` flow through to the sitemap builder
- A known publishing target: either a fully-qualified URL (set `CanonicalBaseUrl`) or a sub-path via `dotnet run -- build --base-url /sub/` (the sitemap falls back to `OutputOptions.BaseUrl`)

Options

Set `CanonicalBaseUrl` so `<loc>` values resolve

When `CanonicalBaseUrl` is set on `PenningtonOptions`, `DocSiteOptions`, or `BlogSiteOptions`, the sitemap builder prefixes every URL with it — typically `https://your-domain.com/` — producing the absolute `<loc>` entries crawlers require. Without it, entries fall back to the build's `--base-url` value or to `/`.

CSHARP

```
new BlogSiteOptions
{
    CanonicalBaseUrl = "https://example.com",
    // ...
}
```

Exclude drafts and redirects with front matter

The sitemap drops any page whose front matter has `isDraft: true` or sets `redirectUrl:`. `search: false` and `llms: false` are not honored — those are client-side UX preferences, not SEO directives, so opting a page out of search does not remove it from the sitemap.

Every other discovered HTML route is included, regardless of how it is sourced — markdown, Razor pages, and the routes that custom content services (Source content from outside the markdown pipeline, Source content from a remote API) and `AddTaxonomy` term pages emit all appear. The only routes left out

are those with no canonical HTML page: redirects (which serve a 30x) and llms.txt-only sidecars. Non-HTML outputs such as JSON feeds and generated data files are skipped because their output file is not `.html`.

(BlogSite only) Suppress the endpoint with `EnableSitemap = false`

On an `AddBlogSite` host, set `BlogSiteOptions.EnableSitemap = false` to skip the `/sitemap.xml` MapGet entirely — useful when the host environment owns its own sitemap. The flag forwards into `PenningtonOptions.MapSitemap`; on bare `AddPennington` or `AddDocSite`, set that property directly to opt out.

CSHARP

```
new BlogSiteOptions
{
    EnableSitemap = false,
    // ...
}
```

Result

`/sitemap.xml` returns a `<urlset>` with one `<url>` per non-draft, non-redirect page, with absolute `<loc>` values when `CanonicalBaseUrl` is set:

XML

```
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://example.com/</loc>
    <lastmod>2024-01-15</lastmod>
  </url>
  <url>
    <loc>https://example.com/how-to/feeds/sitemap/</loc>
    <lastmod>2024-02-03</lastmod>
  </url>
</urlset>
```

Verify

- Run `dotnet run` and fetch `/sitemap.xml`. Expect a `<urlset>` document with one `<url><loc>…</loc></url>` per non-draft, non-redirect page
- Mark a page `isDraft: true` or set `redirectUrl:` on it and refetch. That URL is absent from the `<urlset>`
- Publish with `CanonicalBaseUrl = "https://example.com"` and confirm every `<loc>` starts with `https://example.com/`. Omit it and run `dotnet run -- build /sub/` to see `<loc>` values start with `/sub/`

Related

- Reference: SitemapService
- How-to: Publish an RSS feed
- How-to: Configure redirects

Generate social card images

Guides Configure SocialCards so every page ships a generated OpenGraph/Twitter image: Pennington discovers, serves, bakes, and meta-tags one card per page; you supply the drawing code.

When a page is shared on a social network or chat app, the preview image comes from its `og:image` / `twitter:image` meta tags. Setting `SocialCards` turns on generated cards: Pennington discovers one card route per page (`/social-cards/{page-path}.png`), renders each on demand during development, bakes them all in the static build, and points every page's meta tags at its own card. You own only the drawing, through a single `Render` hook — bring whatever image library fits (ImageSharp, SkiaSharp, or a headless browser screenshotting an HTML template).

The same option works on every host shape: `DocSiteOptions` and `BlogSiteOptions` forward a `SocialCards` property; a bare host sets it on `PenningtonOptions` directly.

Before you begin

- A working Pennington site (see Create your first Pennington site if not)
- A production origin for `CanonicalBaseUrl` — OpenGraph crawlers require an absolute `og:image` URL, so without it the tags emit root-relative paths that work in dev but do not unfurl when shared

---

Turn on generated cards

Set `SocialCards` with a `Render` hook. This is the complete `BlogSiteSocialCardsExample` host:

CSHARP

```
using Pennington.BlogSite;
using Pennington.SocialCards;

var builder = WebApplication.CreateBuilder(args);

// Generated social cards. Pennington owns the integration — it discovers one card
// route per content page (so the static build bakes them), serves each on demand at
// `/social-cards/<page>.png`, and points every post's `og:image`/`twitter:image` at it.
// The host owns only the drawing, via `SocialCardOptions.Render`: it receives the page's
// resolved metadata (title, description, date, tags, the card's own absolute URL) and
// returns PNG bytes — or null to skip a page.
builder.Services.AddBlogSite(() => new BlogSiteOptions
{
    SiteTitle = "Social Cards Blog",
    SiteDescription = "A BlogSite host demonstrating generated OpenGraph social cards.",
    CanonicalBaseUrl = "https://example.com",
    AuthorName = "Author Name",

    SocialCards = new SocialCardOptions
    {
        // This sample paints a solid placeholder card with a dependency-free PNG encoder so
the
        // example needs no image library. In a real app, draw `request.Title` /
        // `request.Description` onto the canvas with an image library (ImageSharp,
SkiaSharp) or
        // screenshot an HTML template with Playwright — `request` carries everything you
need,
        // and the IServiceProvider lets the renderer resolve registered services (font
caches,
        // theme options, ...).
        Render = (request, services, _) =>
        {
            // The home page gets its own card too — BlogSite projects a site-identity
record
            // for `/` (title/description from the options), rendered at /social-
cards/index.png.
            // CanonicalPath is how a renderer varies the design per page: purple for the
home
            // card here, slate blue for posts.
            var png = request.CanonicalPath.Value == "/"
                ? SocialCardPainter.SolidCard(request.Width, request.Height, 0x6D, 0x28,
0xD9)
                : SocialCardPainter.SolidCard(request.Width, request.Height);
            return Task.FromResult<byte[]?>(png);
        },
    },
});

var app = builder.Build();

app.UseBlogSite();

await app.RunBlogSiteAsync(args);
```

`Render` runs whenever a card route is requested: once per page during a static build, and on demand in development each time the route is hit. It receives the page's resolved metadata, the request's `IServiceProvider` (resolve anything registered — a font cache, theme options), and a cancellation token. Return the image bytes, or `null` to skip the page: its card route serves 404 and is omitted from the build.

Everything the hook receives arrives on the request:

CSHARP

```
public sealed record SocialCardRequest(
    string Title,
    string? Description,
    DateTime? Date,
    UrlPath CanonicalPath,
    string CardUrl,
    string? Locale,
    string SiteTitle,
    string? SiteDescription,
    IFrontMatter Metadata,
    int Width,
    int Height);
```

`Metadata` is the page's full typed front matter, so a renderer can pattern-match capability interfaces — tags, author, series — beyond the common fields. The remaining `SocialCardOptions` members have working defaults: cards publish under `/social-cards/`, at the 1200x630 OpenGraph standard, as `image/png`.

Which pages get cards

A card exists for every page that projects a content record — the discovery seam that carries a page's typed front matter.

Markdown pages

Automatic on every host shape: DocSite pages, DocSite and BlogSite blog posts, and markdown registered with `AddMarkdownContent<T>` on a bare host all project records, so each gets a card and the matching meta tags.

The home page

BlogSite projects a site-identity record for `/` — title and description from `BlogSiteOptions` — so the root URL gets a card at `/social-cards/index.png` with no extra configuration. To give it a distinct design, branch on `request.CanonicalPath`, as the example above does.

Razor pages

A routed `@page` component gets a card when it has sidecar metadata — a `{Component}.razor.metadata.yml` file next to the component:

YAML

```
title: About us
description: Who we are and why we build this.
```

A Razor page without a sidecar projects no record, so it gets no card and no card meta tags.

Pages to skip

Return `null` from `Render`. The card route 404s, the build omits the file, and the page keeps any site-wide default image instead.

Use your own image for some pages

A page that authors its own `og:image` wins — the generated card's tags only fill gaps, through the same head reconciliation every contributor goes through (see the head subsystem). How a page declares that image depends on the host shape.

BlogSite

`SocialMediaImageUrlFactory` is the per-post hook: return an image URL to use it for that post, or `null` to fall back to the generated card.

CSHARP

```
new BlogSiteOptions
{
    SocialMediaImageUrlFactory = post =>
        post.FrontMatter.Tags.Contains("announcement") ? "/img/announcement-card.png" :
null,
    SocialCards = new SocialCardOptions { Render = ... },
}
```

DocSite or a bare host

There is no per-page factory option here, so override the card the way any tag overrides a built-in one: a head contributor that emits `og:image` from a band below the card's. The card contributor sits at `HeadOrder.Page`, so a lower `Order` wins the slot through the lowest-order-wins rule, and the card's tag steps aside for those pages. Read the per-page image off the resolved record's front matter (give your front-matter type an image field, or branch on tags as below) and skip pages that should keep the generated card.

CSHARP

```
internal sealed class CardOverrideHeadContributor : IHeadContributor
{
    // Below HeadOrder.Page so this wins the og:image slot against the generated card.
    public int Order => HeadOrder.Page - 1;

    public bool ShouldContribute(HeadContext context) =>
        context.Record?.Metadata is ITaggable { Tags: var tags } &&
tags.Contains("announcement");

    public Task ContributeAsync(HeadContext context, HeadBuilder head)
    {
        head.Property("og:image", "/img/announcement-card.png");
        head.Meta("twitter:image", "/img/announcement-card.png");
        return Task.CompletedTask;
    }
}
```

Register it after the host wiring (see Add tags to the document head for the full contributor surface):

CSHARP

```
builder.Services.AddHeadContributor<CardOverrideHeadContributor>();
```

Result

Every recorded page carries meta tags pointing at its card, absolute when `CanonicalBaseUrl` is set:

HTML

```
<meta property="og:image" content="https://example.com/social-cards/blog/hello-card.png"
data-head="meta:prop:og:image">
<meta name="twitter:image" content="https://example.com/social-cards/blog/hello-card.png"
data-head="meta:name:twitter:image">
<meta name="twitter:card" content="summary_large_image" data-head="meta:name:twitter:card">
```

The static build bakes one PNG per page under `output/social-cards/`, mirroring the page tree, with the home page reserved as `index.png`.

Verify

- Run `dotnet run` and open `/social-cards/<page-path>.png` — expect your rendered card; a 404 means the page projects no record (or `Render` returned `null`)
- View-source any post and confirm the three meta tags above point at the page's own card
- Run `dotnet run -- build output` and confirm `output/social-cards/` contains one `.png` per page, including `index.png` on BlogSite

Related

- Explanation: The head subsystem — how card meta tags compose with page-authored tags

- How-to: Add tags to the document head — write your own head contributor
- Reference: Pennington.BlogSite.BlogSiteOptions
- How-to: Publish an RSS feed

CHAPTER 9

Content Services

Source content from outside the markdown pipeline

Guides Implement IContentService to surface JSON files, a database table, or a remote API as routed pages, navigation entries, search documents, and xref targets.

Implement `IContentService` directly to give content the markdown pipeline can't reach — a folder of JSON release notes, a SQL table, a remote API, generated API reference — its own routed pages, navigation entries, cross-references, and search documents, exactly the way markdown pages get them.

Two narrower needs have shorter answers. For a second markdown tree with a different front-matter type, use chained `AddMarkdownContent<T>` instead — see Serve docs and a blog from separate content roots. When a dataset feeds existing pages but needs no routes of its own, register it with `AddDataFile<T>` rather than a content service — see Use a YAML or JSON data file in pages.

The recipe references `examples/ExtensibilityLabExample/ReleaseNotesContentService.cs`, which turns `Content/releases/*.json` into `/releases/{version}/` routes.

Before you begin

- A working Pennington site on bare `AddPennington` (see Create your first Pennington site if not). `AddDocSite` pins its own markdown service, so adding a second content service on top works, but the concepts below assume the unwrapped host.
- Familiarity with the four-stage pipeline at a conceptual level (The content pipeline and union types).
- Source data that can be enumerated synchronously or asynchronously on startup — `DiscoverAsync` runs both at build time and on demand for live requests.

Write the service

Implement Pennington.Content.IContentService as a sealed class. Cache the parsed records in a `Lazy<ImmutableList<T>>` so discovery and the TOC share one pass over the source. That cache holds for the service's lifetime, which has consequences for live edits — see the stale-data warning under Register the service.

CSHARP

```
namespace ExtensibilityLabExample;

using System.Collections.Immutable;
using System.Globalization;
using System.Text.Json;
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Hosting;
using Pennington.Content;
using Pennington.FrontMatter;
using Pennington.Pipeline;
using Pennington.Routing;
using Pennington.Search;
using Pennington.StructuredData;

/// <summary>
/// Demonstrates <see cref="IContentService"/> by turning a folder of
/// <c>Content/releases/*.json</c> files into site pages, a navigation
/// section, and cross-reference entries.
/// <para>
/// Emits one <see cref="DiscoveredItem"/> per JSON file plus an index
/// route. The static-build crawler fetches each one over HTTP from the
/// running host, so a sibling <c>MapGet("/releases/{version}/")</c>
/// endpoint in <c>Program.cs</c> does the actual HTML rendering — the
/// same code path dev-mode uses. That keeps the service focused on
/// discovery, TOC, and cross-references and leaves presentation to the
/// endpoint.
/// </para>
/// <para>
/// Backs how-to 2.3.10 <c>/how-to/extensibility/custom-content-service</c>.
/// </para>
/// </summary>
public sealed class ReleaseNotesContentService : IContentService
{
    private readonly string _releasesDirectory;
    private readonly Lazy<ImmutableList<ReleaseEntry>> _entriesLazy;

    public ReleaseNotesContentService(IWebHostEnvironment environment)
    {
        _releasesDirectory = Path.Combine(environment.ContentRootPath, "Content",
"releases");
        _entriesLazy = new Lazy<ImmutableList<ReleaseEntry>>(LoadEntries);
    }

    public string DefaultSectionLabel => "Releases";
    public int SearchPriority => 20;

    /// <summary>The full set of release entries this service knows about.</summary>
    public IReadOnlyList<ReleaseEntry> Entries => _entriesLazy.Value;

    /// <summary>Find a single release by version string, or null if no match.</summary>
    public ReleaseEntry? TryGet(string version) =>
```

```
    /// One discovered item for the index plus one per JSON file. Each route is    ///
paired with <see cref="EndpointSource"/> — the build crawler discovers    /// the URL and
fetches it through the live pipeline, where the sibling    /// <c>MapGet</c> endpoint in
<c>Program.cs</c> produces the HTML. Because the    /// endpoint serves real canonical HTML,
these routes are included in    /// <c>sitemap.xml</c> like any other page.    /// <para>
/// Each release item carries its <see cref="ReleaseEntry"/> as    /// <see
cref="DiscoveredItem.Metadata"/>. That single assignment surfaces the records to    ///
discovery: the default <c>GetRecordsAsync</c> bridge picks them up, so the browse-by-channel
/// taxonomy, the custom <c>channel</c> search facet, and the per-page JSON-LD all light up
from    /// the one record — no separate index page, the same treatment markdown gets.
/// The index item carries no metadata, so it is not itself a record.    /// </para>    ///
</summary>    public async IAsyncEnumerable<DiscoveredItem> DiscoverAsync()    {
yield return new DiscoveredItem(            ContentRouteFactory.FromUrl(new
UrlPath("/releases/")),            new EndpointSource());          foreach (var entry in
_entriesLazy.Value)        {            var route = ContentRouteFactory.FromCustom(
url: new UrlPath($"/releases/{entry.Version}/"),               sourceFile: new
FilePath(entry.SourcePath));            yield return new DiscoveredItem(route, new
EndpointSource()) { Metadata = entry };        }          await Task.CompletedTask;    }
/// <summary>No static files to copy — JSON sources are transformed, not republished.
</summary>    public Task<ImmutableList<ContentToCopy>> GetContentToCopyAsync()        =>
Task.FromResult(ImmutableList<ContentToCopy>.Empty);      /// <summary>TOC entries so the
pages show up in navigation and the search index.</summary>    public
Task<ImmutableList<ContentTocItem>> GetContentTocEntriesAsync()    {        var entries =
_entriesLazy.Value;        var builder = ImmutableList.CreateBuilder<ContentTocItem>();
builder.Add(new ContentTocItem(            Title: "Releases",            Route:
ContentRouteFactory.FromUrl(new UrlPath("/releases/")),            Order: 100,
HierarchyParts: ["releases"],            SectionLabel: DefaultSectionLabel,
Locale: null));          var order = 110;        foreach (var entry in entries)        {
builder.Add(new ContentTocItem(               Title: entry.Title,               Route:
ContentRouteFactory.FromUrl(new UrlPath($"/releases/{entry.Version}/")),
Order: order,               HierarchyParts: ["releases", entry.Version],
SectionLabel: DefaultSectionLabel,               Locale: null));            order += 10;
}          return Task.FromResult(builder.ToImmutable());    }      /// <summary>One cross-
reference per release so <c>&lt;xref:release-1.0.0&gt;</c> resolves.</summary>    public
Task<ImmutableList<CrossReference>> GetCrossReferencesAsync()    {        var entries =
_entriesLazy.Value;        var builder = ImmutableList.CreateBuilder<CrossReference>();
foreach (var entry in entries)        {            var route =
ContentRouteFactory.FromUrl(new UrlPath($"/releases/{entry.Version}/"));
builder.Add(new CrossReference($"release-{entry.Version}", entry.Title, route));        }
return Task.FromResult(builder.ToImmutable());    }      private ImmutableList<ReleaseEntry>
LoadEntries()    {        if (!Directory.Exists(_releasesDirectory))        {
return [];        }          var builder = ImmutableList.CreateBuilder<ReleaseEntry>();
foreach (var file in Directory.EnumerateFiles(_releasesDirectory, "*.json"))        {
var json = File.ReadAllText(file);            var dto =
JsonSerializer.Deserialize<ReleaseJson>(json, JsonOptions);            if (dto is null)
{                continue;            }              builder.Add(new ReleaseEntry
{                Version = dto.Version,                Title = dto.Title,
Date = DateTime.TryParse(dto.Date, CultureInfo.InvariantCulture,
DateTimeStyles.AssumeUniversal | DateTimeStyles.AdjustToUniversal, out var parsed)
? parsed                        : null,                Channel =
string.IsNullOrWhiteSpace(dto.Channel) ? "stable" : dto.Channel!,               Tags =
dto.Tags?.ToArray() ?? [],               Highlights = dto.Highlights ?? [],
```

```
SourcePath=file,            });        }          return [.. builder.OrderBy(e => e.Version,
StringComparer.OrdinalIgnoreCase)];    }      private static readonly JsonSerializerOptions
JsonOptions = new()    {        PropertyNameCaseInsensitive = true,    };      private
sealed record ReleaseJson(        string Version,        string Title,        string Date,
string? Channel,        List<string>? Tags,        List<string>? Highlights);}  ///
<summary>/// One parsed release record. Implements <see cref="IFrontMatter"/> plus the
discovery capability/// mixins, so a single record feeds the browse-by-channel taxonomy
(<see cref="ITaggable"/> //// the <c>Channel</c> key), the custom <c>channel</c> search
facet (<see cref="IHasSearchFacets"/>),/// and the per-page JSON-LD (<see
cref="IHasStructuredData"/>) — the same treatment markdown front/// matter gets, with no
extra wiring beyond attaching it to the discovered item./// </summary>public sealed record
ReleaseEntry : IFrontMatter, ITaggable, IHasSearchFacets, IHasStructuredData{    ///
<summary>Semantic version, used as the route slug (e.g. <c>1.1.0</c>).</summary>    public
string Version { get; init; } = "";      /// <inheritdoc/>    public string Title { get;
init; } = "";      /// <inheritdoc/>    public DateTime? Date { get; init; }      ///
<summary>Release channel (<c>stable</c>, <c>beta</c>, ...) — the taxonomy key and custom
search facet.</summary>    public string Channel { get; init; } = "stable";      ///
<inheritdoc/>    public string[] Tags { get; init; } = [];      /// <summary>Bullet
highlights rendered on the detail page.</summary>    public IReadOnlyList<string> Highlights
{ get; init; } = [];      /// <summary>Absolute path of the JSON source file (drives file-
watching).</summary>    public string SourcePath { get; init; } = "";      /// <inheritdoc/>
public IReadOnlyDictionary<string, string[]> SearchFacets => new Dictionary<string,
string[]>    {        ["channel"] = [Channel],    };      /// <inheritdoc/>    public
IEnumerable<JsonLdEntity> GetStructuredData(StructuredDataContext context) =>    [
new ReleaseJsonLd        {            Name = Title,            SoftwareVersion = Version,
DatePublished = Date?.ToString("yyyy-MM-dd", CultureInfo.InvariantCulture),            Url =
context.CanonicalUrl,        },    ];}  /// <summary>A schema.org <c>SoftwareApplication</c>
describing one release, emitted as JSON-LD.</summary>public sealed record ReleaseJsonLd :
JsonLdEntity{    /// <inheritdoc/>    [JsonPropertyName("@type")]    public override string
Type => "SoftwareApplication";      /// <summary>Release title.</summary>
[JsonPropertyName("name")]    public required string Name { get; init; }      ///
<summary>Semantic version string.</summary>    [JsonPropertyName("softwareVersion")]
public string? SoftwareVersion { get; init; }      /// <summary>ISO publication date.
</summary>    [JsonPropertyName("datePublished")]    public string? DatePublished { get;
init; }      /// <summary>Canonical URL of the release page.</summary>
[JsonPropertyName("url")]    public string? Url { get; init; }}
```

Three non-obvious moves carry this service:

- **`DiscoverAsync` pairs a route with a `ContentSource` case.** Build the route with `ContentRouteFactory.FromUrl` (synthetic URL, no backing file) or `ContentRouteFactory.FromCustom` (URL plus an on-disk `FilePath` so file-watching picks up edits). The example uses `EndpointSource` so the build crawler fetches each URL through a sibling `MapGet` endpoint; those routes serve real canonical HTML, so they appear in `sitemap.xml` like any other page.
- **`GetContentTocEntriesAsync` feeds the sidebar and the search index.** Set `Title`, `Route`, `Order` (10/20/30 spacing), `HierarchyParts` (sidebar nesting), and `SectionLabel` (group header). The same items power search ranking.
- **`GetCrossReferencesAsync` publishes one `CrossReference(uid, title, route)` per record** so authors can deep-link entries with `<xref:uid>`. Pick a stable prefix (`release-1.0.0` here) so the uid does not depend on a URL that may move.

`GetContentToCopyAsync` returns `ImmutableList.Empty` when HTML served by an endpoint is the only output. Byte artifacts (a `robots.txt`, a JSON sidecar) are not a content-service concern — they belong on `IArtifactContentService` (Emit generated output artifacts).

Register the service

`AddPennington` does not auto-discover `IContentService` implementations — register directly on `IServiceCollection`. When an endpoint in `Program.cs` needs the concrete type to render detail pages, register it once by concrete type and forward `IContentService` to the same instance so the container does not create a second copy.

CSHARP

```
builder.Services.AddSingleton<ReleaseNotesContentService>();
builder.Services.AddSingleton<IContentService>(sp =>
    sp.GetRequiredService<ReleaseNotesContentService>());
```

This service reads its JSON into a `Lazy<T>` once and lives as a singleton, so an edit to a release file during a dev session is not picked up until restart. When live-reload on source edits matters, register the service file-watched instead — `AddFileWatched<T>` plus a *transient* `IContentService` wrapper, as Publish a custom feed from a content service shows. `AddSingleton<IContentService>` over a file-watched type silently caches the first copy and serves stale data. To source from a remote API rather than disk, see Source content from a remote API.

Feed your records to taxonomy, search, and JSON-LD

The service above produces routes, navigation, and cross-references. To also let your records drive browse-by-field pages, custom search facets, and JSON-LD — the same way markdown records do — give each record a typed front matter that implements the capability mixins, and **attach it to the discovered item**. The example's `ReleaseEntry` does exactly this:

CSHARP

```
public sealed record ReleaseEntry : IFrontMatter, ITaggable, IHasSearchFacets,
IHasStructuredData
{
    /// <summary>Semantic version, used as the route slug (e.g. <c>1.1.0</c>).</summary>
    public string Version { get; init; } = "";

    /// <inheritdoc/>
    public string Title { get; init; } = "";

    /// <inheritdoc/>
    public DateTime? Date { get; init; }

    /// <summary>Release channel (<c>stable</c>, <c>beta</c>, ...) — the taxonomy key and
custom search facet.</summary>
    public string Channel { get; init; } = "stable";

    /// <inheritdoc/>
    public string[] Tags { get; init; } = [];

    /// <summary>Bullet highlights rendered on the detail page.</summary>
    public IReadOnlyList<string> Highlights { get; init; } = [];

    /// <summary>Absolute path of the JSON source file (drives file-watching).</summary>
    public string SourcePath { get; init; } = "";

    /// <inheritdoc/>
    public IReadOnlyDictionary<string, string[]> SearchFacets => new Dictionary<string,
string[]>
    {
        ["channel"] = [Channel],
    };

    /// <inheritdoc/>
    public IEnumerable<JsonLdEntity> GetStructuredData(StructuredDataContext context) =>
    [
        new ReleaseJsonLd
        {
            Name = Title,
            SoftwareVersion = Version,
            DatePublished = Date?.ToString("yyyy-MM-dd", CultureInfo.InvariantCulture),
            Url = context.CanonicalUrl,
        },
    ];
}
```

CSHARP

```
yield return new DiscoveredItem(route, new EndpointSource()) { Metadata = entry };
```

That `Metadata` assignment lets taxonomy, search, and JSON-LD read the record: the engine reads it through `GetRecordsAsync` (the default bridges from `DiscoverAsync`, so attaching metadata is all it takes — no override needed):

- **Taxonomy** — `AddTaxonomy<ReleaseEntry, string>(opts => opts.SelectKey = fm => fm.Channel)` gives you `/channel/` browse pages with no `FileSource` required (see Build browse-by- pages with AddTaxonomy).
- **Search facets** — the IHasSearchFacets `channel` axis emits alongside the built-in `section` / `tag` / `area` dimensions.
- **JSON-LD** — the IHasStructuredData entity is injected into each release page's `<head>` automatically when `CanonicalBaseUrl` is set; no `<script>` to hand-write.

A record participates in a taxonomy axis only when its metadata *is* that axis's `TFrontMatter`, so type your records as the front matter you intend to browse. If you project records that don't flow through `DiscoverAsync` (or want to filter which ones do), override `GetRecordsAsync` directly instead of attaching `Metadata`.

Result

The discovered records produce a "Releases" section in the sidebar, one route per entry, and one xref id per entry:

TEXT

```
/releases/                  -> Releases (index)
/releases/1.0.0/            -> uid: release-1.0.0
/releases/1.1.0/            -> uid: release-1.1.0
```

Each `/releases/{version}/` URL renders through the sibling `MapGet` endpoint, the entries appear in the search index under the "Releases" section, and `<xref:release-1.0.0>` in any markdown page resolves to `/releases/1.0.0/`.

Verify

- Run `dotnet run --project examples/ExtensibilityLabExample` and visit `/releases/` — the index lists every entry and each `/releases/{version}/` renders.
- The "Releases" section shows up in navigation with one child per discovered record.
- Authoring `<xref:release-1.0.0>` inside a markdown page resolves to the right URL, and the static build (`dotnet run -- build`) writes one HTML file per route under `output/releases/`.

Related

- Reference: Content pipeline interfaces
- Reference: Routing types
- How-to: Source content from a remote API
- How-to: Use a YAML or JSON data file in pages
- Background: The content pipeline and union types
- Background: What the DocSite and BlogSite templates wire for you

Source content from a remote API

Guides Implement IContentService over a typed HttpClient to turn a remote JSON API — here the GitHub Releases API — into routed pages, navigation, search, and xref targets, with one cached fetch per build, rendered markdown bodies, and a fixture fallback when the API is down.

To build pages from a remote HTTP API instead of local files — a release feed, a CMS, a product catalog behind a JSON endpoint — implement `IContentService` over a typed `HttpClient`. This guide is the remote counterpart to Source content from outside the markdown pipeline: that page covers the discovery, TOC, and cross-reference work every content service shares. This one adds the four things a *network* source needs:

- awaiting HTTP in `DiscoverAsync`,
- caching one fetch across every pipeline pass,
- rendering markdown bodies that arrive over the wire,
- and surviving a slow or unreachable API at build time.

The recipe references `examples/BeyondRemoteContentExample`, which turns the GitHub Releases API into `/releases/{version}/` pages.

Before you begin

- A working Pennington site on bare `AddPennington` (see Create your first Pennington site).
- The discovery shape from Source content from outside the markdown pipeline — this guide assumes you know how `DiscoverAsync`, `GetContentTocEntriesAsync`, and `GetCrossReferencesAsync` fit together, and focuses on what changes when the source is remote.
- An API reachable over HTTP that returns JSON. The example uses an unauthenticated public endpoint; for an authenticated API, add the token to the typed client's default headers.

Fetch the data with a typed `HttpClient`

Register a typed `HttpClient` with `AddHttpClient<T>`. Set a `User-Agent` — the GitHub API answers `403` without one — and a `Timeout`, so a stalled API cannot hang the build:

CSHARP

```
builder.Services.AddHttpClient<GitHubReleasesClient>(client =>
{
    client.BaseAddress = new Uri("https://api.github.com/");
    client.DefaultRequestHeaders.UserAgent.ParseAdd("Pennington-Remote-Content-Example");
    client.Timeout = TimeSpan.FromSeconds(10);
});
```

The client itself is a thin wrapper that fetches and deserializes. `GetFromJsonAsync` with a snake-case naming policy maps GitHub's `tag_name` / `html_url` / `published_at` onto a PascalCase record:

CSHARP

```
public async Task<ImmutableList<GitHubRelease>> GetReleasesAsync()
{
    try
    {
        // The User-Agent header (set in Program.cs) is required — GitHub answers
        // 403 without one. per_page caps the page; a busy repo paginates via the
        // response Link header.
        var releases = await _http.GetFromJsonAsync<List<GitHubRelease>>(
            $"repos/{Owner}/{Repo}/releases?per_page=20", JsonOptions);

        if (releases is { Count: > 0 })
        {
            return [.. releases.Where(r => !r.Draft)];
        }

        _logger.LogWarning("GitHub returned no releases; using the bundled fixture.");
    }
    catch (Exception ex) when (ex is HttpRequestException or TaskCanceledException or
JsonException)
    {
        // Fail open: a slow, unreachable, or rate-limited API must not break the
        // build. To fail the build instead, rethrow here and let it propagate.
        _logger.LogWarning(ex, "GitHub releases fetch failed; using the bundled fixture.");
    }

    return await LoadFixtureAsync();
}
```

The `try` / `catch` is the build-time failure boundary — covered under Handle a slow or unreachable API below.

Cache the fetch across every pass

`DiscoverAsync`, the TOC pass, the cross-reference pass, and the rendering endpoint each need the data. Fetching in each one would hit the API four-plus times per build. Cache the result in an `AsyncLazy<T>` (from `Pennington.Infrastructure`) created once in the constructor, and `await` it everywhere:

CSHARP

```
private readonly AsyncLazy<ImmutableList<ReleaseEntry>> _entriesLazy;

public GitHubReleasesContentService(GitHubReleasesClient client)
    => _entriesLazy = new AsyncLazy<ImmutableList<ReleaseEntry>>(() => LoadAsync(client));
```

`AsyncLazy<T>` runs its factory once on first access and replays the same task to every later caller; a faulted fetch is evicted so the next access retries. `DiscoverAsync` awaits it like every other pass, pairing each route with `EndpointSource` so the build crawler fetches the URL through the endpoint below:

CSHARP

```
public async IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
{
    yield return new DiscoveredItem(
        ContentRouteFactory.FromUrl(new UrlPath("/releases/")),
        new EndpointSource());

    foreach (var entry in await _entriesLazy)
    {
        yield return new DiscoveredItem(
            ContentRouteFactory.FromUrl(new UrlPath($"/releases/{entry.Version}/")),
            new EndpointSource());
    }
}
```

Because this service reads no local files, there is nothing to file-watch — register it as a process-lifetime singleton (see Register the service). A service backed by files that change during a dev session needs the file-watched lifetimes described in Publish a custom feed from a content service, or it serves stale data.

Render the API's markdown body yourself

GitHub returns each release's notes as markdown. `EndpointSource` routes are *not* run through Markdig automatically — the framework hands the route to your endpoint and renders nothing — so call `IContentRenderer` yourself. Build a `ParsedItem` from the markdown and render it; the result's `Content.Html` is the same output a markdown file would produce:

CSHARP

```
public static async Task<string> RenderDetailAsync(IContentRenderer renderer, ReleaseEntry
entry)
{
    var route = ContentRouteFactory.FromUrl(new UrlPath($"/releases/{entry.Version}/"));
    var parsed = new ParsedItem(route, new ReleaseFrontMatter(entry.Title),
entry.BodyMarkdown ?? "");
    var rendered = await renderer.RenderAsync(parsed);

    var body = rendered.Value is RenderedItem item
        ? item.Content.Html
        : "<p>Release notes are unavailable.</p>";

    return Page(entry.Title, $"""
        <p class="meta">
          <time datetime="{entry.Date:yyyy-MM-dd}">{entry.Date:yyyy-MM-dd}</time>
          &middot; <a href="{entry.HtmlUrl}">View on GitHub</a>
        </p>
        {body}
        """);
}
```

Wrap that HTML in the element named by `SiteProjection.ContentSelector`. Set the selector to match that element in the `AddPennington` callback — here both are `<article>`:

CSHARP

```
builder.Services.AddPennington(penn =>
{
    // The endpoint below wraps each release body in <article>; point the
    // projection selector at that element.
    penn.SiteProjection.ContentSelector = "article";
});
```

At build time the projection self-fetches every TOC-listed route through the live pipeline and splits the selected element into heading sections — so an `EndpointSource` page rendered this way **is indexed at heading level for search and llms.txt**, exactly like a markdown page. Without the selector match, the page chrome leaks into the index instead of the release body.

> **Note**
>
> Because each release page serves real canonical HTML at a stable URL, it **is** included in `sitemap.xml` — `EndpointSource` routes are crawled like any other page. (Only `RedirectSource` and `LlmsOnlySource` routes, which have no canonical HTML, are left out.) See Publish a sitemap.

Register the service

Register the concrete type, then forward `IContentService` to the same instance so the endpoint and the pipeline share one cache:

CSHARP

```
builder.Services.AddSingleton<GitHubReleasesContentService>();
builder.Services.AddSingleton<IContentService>(sp =>
    sp.GetRequiredService<GitHubReleasesContentService>());
```

A singleton holding a typed `HttpClient` is normally discouraged — the pooled handler stops rotating, so long-lived processes can pin stale DNS. Here it is fine: the client is used for a single fetch at startup and never again; the *cache*, not the client, is what lives for the process. For a long-running host that re-fetches periodically, inject `IHttpClientFactory` and create a client per fetch instead.

Handle a slow or unreachable API at build time

A build that fetches from the network inherits the network's failure modes: the API can be down, rate-limited, or slow. The `Timeout` on the typed client bounds the slow case. For the rest, `GetReleasesAsync` **fails open** — on `HttpRequestException`, a timeout, or malformed JSON it logs a warning and falls back to a committed fixture (`fixtures/github-releases.json`), so an offline or rate-limited build still produces a complete site. The fixture also makes the build deterministic in CI.

To **fail closed** instead — stop the build when the data is unavailable rather than ship a snapshot that may be stale or empty — rethrow from the `catch` and let it propagate out of `DiscoverAsync`. Choose per site: a marketing build might prefer last-known-good data; a compliance build might prefer to fail loudly.

Keep the published data fresh

> ⚠ **Warning**
>
> A static build is a point-in-time snapshot. The published site shows the releases that existed *when you built it* and will not change until you build again — a new release on GitHub does not appear on its own.

For content that updates on its own schedule, rebuild on a schedule. The only thing a scheduled rebuild adds to an ordinary deploy workflow is a `schedule` trigger — a `cron` entry that fires the same build-and-deploy job on a timer instead of (or alongside) a push:

YAML

```yaml
on:
  schedule:
    - cron: "0 6 * * *"   # rebuild nightly at 06:00 UTC
  workflow_dispatch:       # plus a manual trigger
```

The job that this trigger runs — checkout, `setup-dotnet`, `dotnet run -- build`, and the deploy steps — is the same one a push build uses. Build it from Deploy to GitHub Pages and add the `schedule` trigger above.

Verify

- Run `dotnet run --project examples/BeyondRemoteContentExample` and open `/releases/`. The index lists every release and each `/releases/{version}/` renders its notes as formatted HTML (headings, lists, links), not raw markdown.
- Run `dotnet run --project examples/BeyondRemoteContentExample -- build output`. Confirm `output/releases/` has one folder per release, the rendered markdown carries `<h2>` headings, `output/search/` indexes those heading texts (proving the `EndpointSource` bodies reach search via the self-fetch), and `output/sitemap.xml` lists every `/releases/{version}/` URL.
- Disconnect from the network and rebuild. The build still succeeds, serving the fixture snapshot.

Related

- How-to: Source content from outside the file system — the base `IContentService` shape this guide builds on.
- How-to: Publish a custom feed from a content service — the DI lifetimes for a *file-backed* service, and the stale-cache trap.
- How-to: Publish a sitemap — what the sitemap includes and excludes.
- Background: Why ContentSource is a union — what `EndpointSource` means and why its pages are still listed in the sitemap.

Auto-generate an API reference tree for a class library

Guides Wire the reflection metadata backend (a compiled .dll + .xml pair), call AddApiReference, and get one /reference/api// page per public type plus inline Mdazor components for member tables, summaries, and extension-method catalogs.

To ship a DocSite whose reference section stays in sync with a class library's public API, register a metadata backend and call `AddApiReference()`. One Razor template renders every public type, and a handful of Mdazor components (`<ApiMemberTable>`, `<ApiSummary>`, `<ExtensionMethods>`, `<ApiParameterTable>`) are available inline in markdown for hand-authored reference pages. Every generated page, search entry, and xref keys off a single pass over the configured backend.

`Pennington.ApiMetadata.Reflection.AddApiMetadataFromCompiledAssembly()` is the metadata backend: it reflects over a compiled `.dll` and parses the companion xmldoc `.xml` file. It documents any assembly — one you build alongside the docs host, or a third-party NuGet package — without needing its source.

Before you begin

- `AddApiReference` runs after `AddDocSite`. It appends its own assembly to `DocSiteOptions.AdditionalRoutingAssemblies` at registration time, so `AddDocSite` must already be wired.
- One metadata backend is registered before `AddApiReference`. Without one, the content service has nothing to publish.

Wire the reflection backend

Add a project reference to `Pennington.ApiMetadata.Reflection`, add a `<PackageReference>` to the library you want to document, and have Pennington resolve the assembly by simple name. A complete single-package DocSite host:

CSHARP

```
using Pennington.ApiMetadata.Reflection;
using Pennington.DocSite;
using Pennington.DocSite.Api;

var builder = WebApplication.CreateBuilder(args);

// Standard DocSite wiring. No Areas — single default TOC.
builder.Services.AddDocSite(() => new DocSiteOptions
{
    SiteTitle = "FusionCache Docs",
    SiteDescription = "Demo of Pennington's reflection-based API metadata backend,
documenting ZiggyCreatures.FusionCache straight from its NuGet package.",
    GitHubUrl = "https://github.com/ZiggyCreatures/FusionCache",
    HeaderContent = """<a href="/" class="font-bold text-lg">FusionCache Docs</a>""",
    FooterContent = """<footer class="mt-16 py-8 text-center text-sm text-base-
500">FusionCache © ZiggyCreatures. Rendered by Pennington.</footer>""",
});

// Reflection-backed API metadata sourced from the ZiggyCreatures.FusionCache
// NuGet package reference in the .csproj — no live compilation, no staged
// dll/xml, no vendored source.
builder.Services.AddApiMetadataFromCompiledAssembly(opts =>
    opts.FromPackageReference("ZiggyCreatures.FusionCache"));

// Auto-publishes /api/{slug}/ pages off the metadata provider and registers
// the <ApiSummary>, <ApiMemberTable>, <ApiParameterTable>, ... Mdazor
// components. Sits in the Guides section of the sidebar so readers drop into
// the full type index from the same nav they read the stampede/fail-safe
// guides in.
builder.Services.AddApiReference(configure: opts =>
{
    opts.RoutePrefix = "/api/";
    opts.TocSectionLabel = "Guides";
});

var app = builder.Build();
app.UseDocSite();
await app.RunDocSiteAsync(args);
```

`FromPackageReference` resolves the `.dll` (and its companion xmldoc `.xml`) from a matching `<PackageReference>` by simple name — no staged dll, no committed binary, and bumping the documented version is a `<PackageReference Version=…>` change.

When the target isn't a normal NuGet reference — a locally-built assembly, a single-file bundle, or something else without a file location — fall back to the explicit form:

CSHARP

```
builder.Services.AddApiMetadataFromCompiledAssembly(opts =>
    opts.AssemblyFiles.Add(Path.Combine(builder.Environment.ContentRootPath, "lib",
"net9.0", "Foo.dll")));
```

The reflection backend loads each `.dll` into a `MetadataLoadContext` — it inspects metadata without running the assembly's code, so it needs no MSBuild workspace and no source. It resolves `<inheritdoc/>` and union-case xmldoc from that metadata. Because it reads metadata rather than source text, this backend can populate the `<ApiSummary>` and `<ApiMemberTable>` components below, but it cannot back a fence that embeds a member's *source* — the kind of resolver you would write with an ICodeBlockPreprocessor has nothing to read.

Customize the route prefix

The default prefix is `/reference/api/`. Override it per registration via `AddApiReference`'s `RoutePrefix` option when the shorter `/api/` (or any other prefix) is a better fit:

CSHARP

```
builder.Services.AddApiMetadataFromCompiledAssembly(opts => { /* ... */ });
builder.Services.AddApiReference(configure: opts => opts.RoutePrefix = "/api/");
```

Type pages land at `/api/{slug}/`, and xref uids stay as `reference.api.{slug}` for back-compat with the default registration.

Document multiple libraries on one site

Pair each library with its own named `AddApiMetadataFrom*` call and a matching `AddApiReference` call. Names are the key that wires a reference tree to its provider, and every tree gets its own URL prefix:

CSHARP

```
builder.Services.AddApiMetadataFromCompiledAssembly("spectre-console", opts =>
    opts.FromPackageReference("Spectre.Console"));
builder.Services.AddApiMetadataFromCompiledAssembly("spectre-console-cli", opts =>
    opts.FromPackageReference("Spectre.Console.Cli"));

builder.Services.AddApiReference("spectre-console", opts =>
    opts.RoutePrefix = "/api/spectre/");
builder.Services.AddApiReference("spectre-console-cli", opts =>
    opts.RoutePrefix = "/api/spectre-cli/");
```

Each `FromPackageReference` call resolves one DLL from its matching `<PackageReference>`. Cross-package type references resolve automatically across the NuGet cache root.

Cross-references between named trees: uids pick up a qualifier. Default-named registrations emit `reference.api.{slug}` (unchanged). Named registrations emit `reference.api.{name}.{slug}` — for example, `<xref:reference.api.spectre-console.ansi-console>` and `<xref:reference.api.spectre-console-cli.command-app>`.

Hand-authored markdown: components like `<ApiSummary>` auto-pick up the source from the enclosing generated page. For markdown pages outside the generated tree that reach into a specific named registration, add an explicit `Source` attribute:

MARKDOWN

```
<ApiSummary XmlDocId="T:Spectre.Console.Cli.CommandApp" Source="spectre-console-cli" />
```

Narrow what gets published

The reflection backend documents the assemblies you point it at, so narrowing is a matter of which assemblies you add. The built-in rules already exclude types that are not public, are delegates or attributes, derive from `ComponentBase`, or carry no xmldoc `<summary>`.

Use `AssemblyFiles` to document an explicit list of `.dll` paths when a folder holds more assemblies than you want documented — for example, dependencies copied alongside the target only so `MetadataLoadContext` can resolve them:

CSHARP

```
builder.Services.AddApiMetadataFromCompiledAssembly(opts =>
{
    opts.AssemblyFiles.Add(Path.Combine(libDir, "MyLibrary.dll"));
    opts.AssemblyFiles.Add(Path.Combine(libDir, "MyLibrary.Extensions.dll"));
});
```

Use `AssemblyDirectories` instead to document every `.dll` / `.xml` pair in a folder — the typical NuGet `lib/<tfm>/` layout.

Render reference fragments inline

Summarize one symbol with `<ApiSummary>`

Pulls the `<summary>` tag off a type or member and renders it as prose. Pass an xmldocid as `XmlDocId`.

MARKDOWN

```
<ApiSummary XmlDocId="T:Pennington.ApiMetadata.ApiTypeSummary" />
```

Lightweight header describing a documented type, used for listings, slug disambiguation, and cross-link display names.

Enumerate type members with `<ApiMemberTable>`

`Kind="All"` groups members by category (Properties, Constructors, Fields, Methods, Events) with headings between; narrow it with `Kind="Properties"` or `Kind="Methods"` for a single bucket.

MARKDOWN

```
<ApiMemberTable XmlDocId="T:Pennington.ApiMetadata.ApiTypeSummary" Kind="Properties" />
```

`Assembly` `string` Declaring assembly name without extension. `FullTypeName` `string` Fully-qualified type name (namespace + dot + type name). `Kind` `ApiTypeKind` Category of the type. `Name` `string` Short type name without namespace (e.g. `ContentPipeline`). `Namespace` `string` Fully-qualified containing namespace, empty for the global namespace. `Summary` `string` First-sentence plain-text summary, or `null` when no xmldoc summary is available. `Uid` `string` Canonical xmldocid (e.g. `T:Namespace.TypeName`). Normalized to xmldocid form regardless of source backend.

List a method's parameters with `<ApiParameterTable>`

Pass a method xmldocid (`M:...`). The table pulls parameter names and types from the provider's pre-formatted `ApiMember` and descriptions from each `<param>` tag.

MARKDOWN

```
<ApiParameterTable
XmlDocId="M:Pennington.ApiMetadata.IApiMetadataProvider.GetMembersAsync(System.String,Pennin
gton.ApiMetadata.MemberKind,Pennington.ApiMetadata.AccessFilter,Pennington.ApiMetadata.Membe
rOrder)" />
```

`typeUid` `string` Uid of the type whose members are returned. `kind` `MemberKind` Member categories to include (properties, methods, and so on). `access` `AccessFilter` Accessibility levels to include. `order` `MemberOrder` Sort order applied to the returned members.

Catalog extension methods by receiver with `<ExtensionMethods>`

Groups every public extension method in the assembly by the unqualified short name of its first (receiver) parameter. `Receiver="IServiceCollection"` gathers every `services.AddX()` helper the library ships.

MARKDOWN

```
<ExtensionMethods Receiver="IServiceCollection" />
```

Result

Every public type with an xmldoc comment gets a route under `/reference/api/{slug}/`:

TEXT

```
/reference/api/                       -> uid: reference.api
/reference/api/api-type-summary/      -> uid: reference.api.api-type-summary
/reference/api/api-member/            -> uid: reference.api.api-member
```

Xref links like `<xref:reference.api.api-type-summary>` resolve, the pages flow through search and llms.txt, and the index page at `/reference/api/` lists every type grouped by namespace. One TOC entry — the index page, titled "API reference" by default — appears in the sidebar; per-type pages stay out of the sidebar and are reached via type-name search, xref links, and the index. Override `TocTitle` and `TocSectionLabel` on `ApiReferenceRegistrationOptions` to customize, or set `TocTitle = null` to suppress the index entry entirely.

Verify

- Run `dotnet run` and visit `/reference/api/` — expect one `<li>` per public documented type, grouped by namespace.
- Visit `/reference/api/{some-type-slug}/` for a type you know has an xmldoc `<summary>` — expect the summary prose and a member table grouped by kind.
- Add `<xref:reference.api.{slug}>` to any markdown page and confirm it resolves to the generated page after a rebuild.

Related

- How-to: Source content from outside the markdown pipeline — hand-write an `IContentService` when `AddApiReference`'s discovery rules are not the right fit.
- Reference: Pennington.ApiMetadata.ApiTypeSummary, Pennington.ApiMetadata.ApiMember.

Emit generated output artifacts

Guides Implement an IArtifactContentService that owns a URL territory and produces byte artifacts — robots.txt, JSON sidecars, generated images — served live in dev and written into the static build.

To emit a byte artifact — `robots.txt`, a sitemap variant, a social-image `.png`, a sidecar `.json` index — that is not a routed page, not in navigation, and not an xref target, implement `IArtifactContentService`. The interface has three members and one rule: the same resolver produces the bytes for a live dev request and for the static build, so the two surfaces can never drift.

- `Claims` declares the URL territory the service owns (an exact path, a prefix, or a path suffix). Claims derive from options alone — they are consulted on every request and must never trigger expensive work.
- `ResolveAsync` turns one claimed path into bytes plus a content type, or returns null to decline so the request falls through to content routing.
- `DiscoverAsync` enumerates the routes the static build writes — each one resolved through `ResolveAsync` and written to its output file.

Pennington's own search shards (`/search/**.json`), llms.txt files, and book PDFs ship through this interface; `RobotsTxtContentService` below is the smallest possible example.

For the opposite case — a service that contributes routed pages, TOC entries, and xrefs from a non-markdown source — see Source content from outside the markdown pipeline.

Before you begin

- A working Pennington site on bare `AddPennington` (see Create your first Pennington site if not).
- Familiarity with the four-stage pipeline at a conceptual level (The content pipeline and union types).

Write the service

CSHARP

```
namespace ExtensibilityLabExample;

using System.Collections.Immutable;
using System.Text;
using Pennington.Artifacts;
using Pennington.Pipeline;
using Pennington.Routing;

/// <summary>
/// Demonstrates the artifact tier — <see cref="IArtifactContentService"/> for byte outputs
/// (robots, search-index sidecars, social-image generators) that are not routed pages, not
in
/// navigation, and not xref targets. <see cref="Claims"/> declares the URL territory,
/// <see cref="ResolveAsync"/> produces the bytes (served live in dev by the artifact
router),
/// and <see cref="DiscoverAsync"/> enumerates the routes the static build writes — one byte
/// path for both surfaces.
/// <para>
/// Backs how-to <c>/how-to/extensibility/emit-generated-artifacts</c>.
/// </para>
/// </summary>
public sealed class RobotsTxtContentService : IArtifactContentService
{
    private const string Body = """
        User-agent: *
        Allow: /
        Sitemap: /sitemap.xml
        """;

    /// <summary>The one URL this service owns.</summary>
    public ImmutableList<ArtifactClaim> Claims { get; } =
        [new ArtifactClaim("robots", new ExactClaim(new UrlPath("/robots.txt")),
"robots.txt")];

    /// <summary>
    /// Produces the robots.txt bytes — for a live dev request and for the static build
alike.
    /// Returning null declines the request so it falls through to content routing.
    /// </summary>
    public Task<ArtifactContent?> ResolveAsync(string relativePath, CancellationToken
cancellationToken)
        => Task.FromResult(relativePath.Equals("robots.txt",
StringComparison.OrdinalIgnoreCase)
            ? new ArtifactContent(Encoding.UTF8.GetBytes(Body), "text/plain; charset=utf-8")
            : null);

    /// <summary>Enumerates the single robots.txt route for the static build.</summary>
    public async IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
    {
        await Task.CompletedTask;
        yield return new DiscoveredItem(
```

```
                OutputFile = new FilePath("robots.txt"),            },            new
GeneratedSource("text/plain"));    }}
```

The pieces:

- `ArtifactClaim` carries an owner name, a shape, and a description. The shape is a union: `ExactClaim` (one path), `PrefixClaim` (everything under a prefix, optionally narrowed by extension — `/search/` + `.json`), or `SuffixClaim` (a path ending at any depth — this is how `{section}/llms.txt` works, a territory no endpoint route template can express). `diag routes` lists every registered claim.
- `ResolveAsync` receives the request path without its leading slash (`robots.txt`, `search/en/index.json`). Returning null declines: the request continues into content routing, so a real page under a claimed prefix keeps working.
- `DiscoverAsync` yields `DiscoveredItem`s with a `GeneratedSource`. Routes that should exist only in dev are resolvable without being enumerated — the book package serves its live `/book-preview/` this way while enumerating only the PDFs.
- The resolver may do expensive work on demand (build an index, fold over `ISiteProjection`, run a headless browser). The claims must not.

Register the service

Register on the artifact tier — never as `IContentService`, which would put the service in every request-path discovery walk:

CSHARP

```
builder.Services.AddTransient<IArtifactContentService, RobotsTxtContentService>();
```

Result

The dev server answers `/robots.txt` live, and the static build writes the same bytes to the output root:

TEXT

```
User-agent: *
Allow: /
Sitemap: /sitemap.xml
```

Verify

- Fetch `/robots.txt` from the dev server and expect the body above — same bytes both surfaces.
- Run `dotnet run --project examples/ExtensibilityLabExample -- build output` and confirm `output/robots.txt` exists with the expected body.
- Run `dotnet run -- diag routes` and confirm the claim appears under "Artifact territories".

Related

- Reference: Content pipeline interfaces
- How-to: Source content from outside the file system
- Background: The content pipeline and union types

Use a YAML or JSON data file in pages

Guides Register a YAML or JSON file as a typed value any Razor page or component can read through IDataFiles. The file hot-reloads when you edit it.

When a piece of site content is structured data — sponsors, navigation, schedule, feature flags, footer links — authoring it as Markdown front-matter or hand-rolling an `IContentService` is overkill. Register the file with `AddDataFile<T>` and read it through `IDataFiles`; the file hot-reloads on edit.

Register the file

`AddDataFile<T>(name, path)` deserializes `path` into `T` on first access and tracks the file for changes. Format is chosen from the extension: `.yml` and `.yaml` go through SharpYaml, `.json` through `System.Text.Json`. Both deserializers use camelCase property naming with case-insensitive matching, mirroring how front matter is parsed.

CSHARP

```
builder.Services.AddDataFile<List<Sponsor>>("sponsors", "data/sponsors.yml");
builder.Services.AddDataFile<Schedule>     ("schedule", "data/schedule.yml");
builder.Services.AddDataFile<List<NavLink>>("nav",      "data/nav.json");
```

The lookup key (`"sponsors"`) is case-insensitive and must be unique across registrations. Paths are resolved against the current working directory if relative.

The target type needs a parameterless constructor — use a record with init-only properties so SharpYaml can populate it:

CSHARP

```
public record Sponsor
{
    public string Name { get; init; } = "";
    public string Tier { get; init; } = "";
    public string Url { get; init; } = "";
}
```

Read the data from a Razor page

Inject `IDataFiles` and call `Get<T>` with the registered name. The lookup is keyed by name *and* type — request the same `T` you registered with.

RAZOR

```razor
@inject IDataFiles Data

<section>
    <h2>Sponsors</h2>
    <ul>
        @foreach (var sponsor in Data.Get<List<Sponsor>>("sponsors").OrderBy(s => s.Tier))
        {
            <li><a href="@sponsor.Url">@sponsor.Name</a> &mdash; @sponsor.Tier</li>
        }
    </ul>
</section>
```

For lookups that may not exist (optional configuration, feature-flag style toggles), `TryGet<T>` returns `false` when the name is missing or the type does not match:

CSHARP

```csharp
if (Data.TryGet<FooterConfig>("footer", out var footer))
{
    // render footer
}
```

Load a directory of records

When each record is its own file — `data/maintainers/agc93.yml`, `data/maintainers/devlead.yml` — register the directory instead of every file by hand. `AddDataDirectory<TItem>(name, path)` deserializes every `.yml`, `.yaml`, and `.json` file in `path` and aggregates them into one list:

CSHARP

```csharp
builder.Services.AddDataDirectory<Maintainer>("maintainers", "data/maintainers");
```

Each file contributes one `TItem`. A file whose root is an array contributes every element, so a directory can mix single-record and multi-record files. Files are ordered by name; anything that is not `.yml`, `.yaml`, or `.json` is ignored, and subdirectories are not scanned.

Read it back as an `IReadOnlyList<TItem>` — that is the type the directory is registered under:

RAZOR

```razor
@inject IDataFiles Data

<ul>
    @foreach (var m in Data.Get<IReadOnlyList<Maintainer>>("maintainers").OrderBy(m =>
m.Name))
    {
        <li>@m.Name</li>
    }
</ul>
```

Adding, editing, or removing a file in the directory invalidates the cached list, the same way editing a single data file does.

Hot reload

When a data file changes on disk, the cached value is invalidated and the next `Get<T>` call reloads and re-deserializes it. Pages that read the data through `IDataFiles` see the fresh value on the next request — no app restart needed.

This is the same lifetime model that `MarkdownContentService<T>` uses for content files. Edits during `dotnet run` propagate immediately.

Verify

- Run `dotnet run` and open a page that reads the data file — confirm the records render (the sponsors list, the nav links, whatever you registered).
- With the server still running, edit a value in `data/sponsors.yml` and save. Refresh the page — the new value appears without a restart, confirming hot reload.
- Request the data under a name you never registered (`Data.Get<List<Sponsor>>("sponsorz")`) and confirm the `KeyNotFoundException` message lists the registered names — proof the registration took.

Errors

`AddDataFile` itself never reads the file; the read happens on the first `Get<T>`. The three you will actually hit while authoring:

- **`FileNotFoundException`** — the registered path does not exist (usually a typo). For `AddDataDirectory`, a missing directory raises `DirectoryNotFoundException`.
- **`InvalidDataException`** — the file content failed to deserialize. The message includes the absolute path and the underlying serializer error, so a bad indent or stray comma points straight at the file.
- **`KeyNotFoundException`** — `Get<T>` was called with a name that was never registered. The message lists every registered name.

The wrong-extension (`NotSupportedException`) and type-mismatch (`InvalidCastException`) cases are catalogued in Pennington.Data.IDataFiles.

What this is not

`AddDataFile` is for data that *decorates* pages (sponsors strip on the homepage, footer links, a feature flag). It does not produce routes — a `data/speakers.yml` registered this way will not give you `/speakers/jane-doe/`. For one-page-per-record needs, write a custom Source content from outside the markdown pipeline.

Related

- Reference: Pennington.Data.IDataFiles — `Get<T>`, `TryGet<T>`, and `Names` with the full exception list.
- How-to: Source content from outside the file system — when each record needs its own route.
- Background: The content pipeline and union types — why data files sit outside the route-producing pipeline.

Build browse-by-{field} pages with AddTaxonomy

Guides Group your content by any front-matter field (cuisine, tag, audience, series) and render the resulting term pages from a Razor component. Hot-reloads when source files change.

To make the same content reachable through more than one browse axis — recipes by cuisine *and* by dietary tag, docs by audience, posts by series — wire each axis with `AddTaxonomy<TFrontMatter, TKey>`. Each call emits a `/{base}/` index plus one `/{base}/{slug}/` term page per distinct key, each rendered from a Razor component you supply.

`AddTaxonomy` groups the records every other registered `IContentService` already projects — it does not re-parse files. Markdown is one such source, but so is any custom content service whose records carry `TFrontMatter` (see Source content from outside the markdown pipeline).

Define your front matter

Add a property for the field you want to group on. Implement Pennington.FrontMatter.ITaggable when one of your axes is multi-valued.

CSHARP

```csharp
public record RecipeFrontMatter : IFrontMatter, ITaggable
{
    public string Title { get; init; } = "";
    public string Cuisine { get; init; } = "";
    public string[] Tags { get; init; } = [];
}
```

A recipe page then carries:

YAML

```yaml
---
title: Carbonara
cuisine: italian
tags: [pasta, eggs, weeknight]
---
```

Register the axis

Each `AddTaxonomy<TFrontMatter, TKey>` call is one axis. Use `SelectKey` for single-valued projections, `SelectKeys` for multi-valued — exactly one of the two is required.

CSHARP

```
builder.Services.AddTaxonomy<RecipeFrontMatter, string>(opts =>
{
    opts.BaseUrl    = "/cuisine";
    opts.SelectKey  = fm => fm.Cuisine;
    opts.IndexPage  = typeof(Pages.CuisineIndex);
    opts.TermPage   = typeof(Pages.CuisineTerm);
});

builder.Services.AddTaxonomy<RecipeFrontMatter, string>(opts =>
{
    opts.BaseUrl    = "/tag";
    opts.SelectKeys = fm => fm.Tags;
    opts.IndexPage  = typeof(Pages.TagIndex);
    opts.TermPage   = typeof(Pages.TagTerm);
});
```

A `Pasta` recipe tagged `[pasta, eggs, weeknight]` ends up under `/tag/pasta/`, `/tag/eggs/`, and `/tag/weeknight/`. A `Sushi` recipe with `cuisine: japanese` ends up under `/cuisine/japanese/`. The two registrations coexist on the same `RecipeFrontMatter` because they target different `BaseUrl`s.

Mount the endpoints

`AddTaxonomy` registers an `IContentService` so the build crawler discovers the routes; the live HTTP handlers are mounted by `MapTaxonomy`:

CSHARP

```
app.MapTaxonomy<RecipeFrontMatter, string>();
```

Call `MapTaxonomy` once per `<TFrontMatter, TKey>` pair — it walks every `AddTaxonomy` registration of that pair and mounts both index and term endpoints for each.

`HtmlRenderer` is required to render the components — wire it the same way the bare-host Razor recipe does:

CSHARP

```
builder.Services.AddRazorComponents();
builder.Services.AddHttpContextAccessor();
```

See Render a Razor component as a page on a bare host for the full bare-host setup.

Author the term page

The Razor component receives the matching `TaxonomyTerm<TFrontMatter, TKey>` as a `Term` parameter:

RAZOR

```
@using Pennington.Taxonomy

<h1>@Term.Label</h1>
<p>@Term.Items.Count recipes</p>

<ul>
    @foreach (var item in Term.Items)
    {
        <li><a href="@item.Url">@item.FrontMatter.Title</a></li>
    }
</ul>

@code {
    [Parameter] public TaxonomyTerm<RecipeFrontMatter, string> Term { get; set; } = null!;
}
```

The index page receives the full term list as `Terms`:

RAZOR

```
@using Pennington.Taxonomy
@using System.Collections.Immutable

<h1>Browse by cuisine</h1>
<ul>
    @foreach (var term in Terms)
    {
        <li><a href="@term.Url">@term.Label (@term.Items.Count)</a></li>
    }
</ul>

@code {
    [Parameter] public ImmutableList<TaxonomyTerm<RecipeFrontMatter, string>> Terms { get;
set; } = [];
}
```

The snippets above are deliberately minimal — bare fragments that get the term data onto the page. Each component backs a route, so wrap its markup in your site layout the same way you would any bare-host Razor page.

Customize slugs and labels

Default slug encoding lowercases the key, replaces whitespace with hyphens, and URL-encodes the rest. Override either:

CSHARP

```
opts.SlugFor  = key => key.ToLowerInvariant();                                  // skip
the URL-encode for plain ASCII
opts.LabelFor = key => CultureInfo.CurrentCulture.TextInfo.ToTitleCase(key);    //
pretty-print on the term page
```

Hot reload

When a markdown file the taxonomy reads changes, the cached term list is invalidated and the next request rebuilds it.

Edits during `dotnet run` propagate immediately.

Verify

- Run `dotnet run` and visit `/cuisine/` — the index lists every cuisine, and `/cuisine/japanese/` renders the term page with the sushi recipe in it.
- Visit `/tag/pasta/` — the same carbonara recipe appears under its tag axis, confirming both registrations coexist.
- Run `dotnet run -- build` and confirm the static build writes `output/cuisine/japanese/index.html` (and one folder per term under `output/tag/`).

Related

- How-to: Source content from outside the file system
- How-to: Paginate a long listing
- How-to: Render a Razor page on a bare host
- Background: The content pipeline and union types

Caveats

- **Listed in the sitemap.** Taxonomy routes use `EndpointSource` (the canonical HTML lives behind `MapTaxonomy`'s endpoints), but they serve real HTML, so they appear in navigation, search, cross-references, *and* `/sitemap.xml` — same as a Source content from outside the markdown pipeline page.
- **Records of `TFrontMatter`, from any source.** An axis collects only records whose metadata is a `TFrontMatter`; everything else is ignored. To feed it from something other than markdown, project that type from a custom service (see Source content from outside the markdown pipeline).
- **Drafts and future-dated posts are skipped.** Items whose `IsHiddenFromBuild` is `true` — `IsDraft` set, or a `Date` in the future — are excluded from every term, same convention as the rest of the pipeline.
- **One Razor component per axis.** Different cuisines can't render with different templates; switch on `Term.Key` inside `TermPage` if some terms need a custom layout.

Add a custom content format

Guides Register a non-markdown file format — here Cooklang .cook recipes — so its files are discovered, parsed, and rendered as pages through the same pipeline as markdown: supply a front-matter type, an IContentParser, an IContentRenderer, and one AddContentFormat call.

To serve a file format Pennington doesn't parse natively — a recipe format, an org-mode file, a bespoke DSL — register it as a *content format*. Pennington discovers the files and tracks them for navigation, search, and resolution; a parser and a renderer you supply turn each file into a page. The pipeline dispatches by the format's key, so your format and markdown coexist on one site.

Reach for this when your content is files with a consistent front-matter-plus-body shape. To source content that isn't file-backed — a remote API, a database — implement `IContentService` directly instead (see Source content from outside the markdown pipeline).

This guide follows `examples/BeyondCookFormatExample`, which registers Cooklang `.cook` recipes at `/recipes/{slug}/` next to a markdown landing page.

Before you begin

- A bare `AddPennington` host with a catch-all that resolves through `IPageResolver` (see Create your first Pennington site).
- A file format with a `---` YAML front-matter block and a body your code can turn into HTML. The example parses recipe bodies with the `CooklangSharp` NuGet package.

Define the front-matter type

Every page carries typed front matter implementing `IFrontMatter` (the contract guarantees a `Title`). Add capability interfaces — `ITaggable`, `IOrderable`, `ISectionable` — for the fields navigation and search should pick up.

CSHARP

```
namespace BeyondCookFormatExample;

using Pennington.FrontMatter;

/// <summary>
/// Front matter for a <c>.cook</c> recipe. Implements <see cref="IFrontMatter"/> (every
page needs a
/// <c>Title</c>) and <see cref="ITaggable"/> (recipe tags feed navigation and the search
facet). The
/// property names are camelCase-matched to the YAML keys (<c>prepTime</c>, <c>cookTime</c>,
…), so they
/// bind without extra attributes and don't trip the build's strict unknown-key check.
/// </summary>
public sealed record CookFrontMatter : IFrontMatter, ITaggable
{
    /// <inheritdoc/>
    public string Title { get; init; } = "";

    /// <summary>Short recipe description, shown under the title.</summary>
    public string? Description { get; init; }

    /// <summary>Number of servings the recipe yields.</summary>
    public string? Servings { get; init; }

    /// <summary>Preparation time (e.g. "15 minutes").</summary>
    public string? PrepTime { get; init; }

    /// <summary>Active cooking time (e.g. "25 minutes").</summary>
    public string? CookTime { get; init; }

    /// <summary>Total time from start to finish.</summary>
    public string? TotalTime { get; init; }

    /// <summary>Resting time, when the recipe calls for it.</summary>
    public string? RestTime { get; init; }

    /// <inheritdoc/>
    public string[] Tags { get; init; } = [];
}
```

Important

Front-matter keys bind to camelCased property names, and a build (`-- build`) throws on any key no property matches. Name your YAML keys to match — `prepTime` , not `prep time` — or author the property to a single word. A multi-word key with a space never binds and fails the build.

Write the parser

An `IContentParser` reads a discovered file and returns a `ParsedItem` — the typed front matter plus the raw body. Inject the framework's `FrontMatterParser` to split the YAML exactly as the markdown parser does, and hand the body on untouched for the renderer.

CSHARP

```
namespace BeyondCookFormatExample;

using System.IO.Abstractions;
using Pennington.FrontMatter;
using Pennington.Pipeline;

/// <summary>
/// Parses a discovered <c>.cook</c> file into a <see cref="ParsedItem"/>: it reads the
file, splits the
/// YAML front matter into a typed <see cref="CookFrontMatter"/>, and hands the Cooklang
body on as the
/// parsed body. The Cooklang markup itself is parsed later by <see
cref="CookContentRenderer"/>. The
/// dispatching pipeline stamps the <c>"cook"</c> format onto the returned item so the
matching renderer
/// is selected.
/// </summary>
public sealed class CookContentParser : IContentParser
{
    private readonly FrontMatterParser _frontMatter;
    private readonly IFileSystem _fileSystem;

    /// <summary>Creates the parser. Both dependencies are registered by
<c>AddPennington</c>.</summary>
    public CookContentParser(FrontMatterParser frontMatter, IFileSystem fileSystem)
    {
        _frontMatter = frontMatter;
        _fileSystem = fileSystem;
    }

    /// <inheritdoc/>
    public async Task<ContentItem> ParseAsync(DiscoveredItem item)
    {
        if (item.Source.Value is not FileSource file)
        {
            return new FailedItem(item.Route, new ContentError("CookContentParser:
unsupported content source."));
        }

        try
        {
            var content = await _fileSystem.File.ReadAllTextAsync(file.Path.Value);
            var result = _frontMatter.Parse<CookFrontMatter>(content, file.Path.Value);
            var metadata = result.Metadata ?? new CookFrontMatter();
            return new ParsedItem(item.Route, metadata, result.Body);
        }
        catch (Exception ex)
        {
            return new FailedItem(item.Route, new ContentError($"Failed to parse
{file.Path}: {ex.Message}", ex));
        }
    }
}
```

The discovered item's source is a `FileSource` carrying the file path and the format key. You don't stamp the format onto the `ParsedItem` yourself — the dispatcher does that from the source, so the matching renderer is selected downstream.

Write the renderer

Markdown renders through a text pipeline; a structured format like a recipe renders through a **Razor component**. Subclass `RazorContentRenderer<TComponent>` (in `Pennington.Pipeline`): the base owns the Blazor `HtmlRenderer` dispatch, heading anchors, and outline extraction, so you write only a component and a `BuildParameters` that projects the parsed body into the component's parameters.

The component binds the parsed model and emits the markup. The page structure is Razor; the tight inline token run within a step is built as a string (inline HTML is whitespace-sensitive — a stray space would land before a `.` or `(`):

RAZOR

```
@namespace BeyondCookFormatExample
@using System.Net
@using System.Text
@using CooklangSharp.Models

@* Renders a parsed Cooklang recipe. The page structure is Razor markup; only the tight
inline
   token run within a step is built as an HTML string (the PhilsRecipes Method.razor
pattern),
   because inline flow is whitespace-sensitive. *@

<article class="recipe">
    @if (!string.IsNullOrWhiteSpace(FrontMatter.Title))
    {
        <h1>@FrontMatter.Title</h1>
    }
    @if (!string.IsNullOrWhiteSpace(FrontMatter.Description))
    {
        <p class="description">@FrontMatter.Description</p>
    }
    @if (_meta.Count > 0)
    {
        <p class="meta">@string.Join(" · ", _meta)</p>
    }

    @if (_ingredients.Count > 0)
    {
        <h2>Ingredients</h2>
        <ul class="ingredients">
            @foreach (var ingredient in _ingredients)
            {
                <li>@if (ingredient.Qty.Length > 0){<span class="qty">@ingredient.Qty</span>
}@ingredient.Name</li>
            }
        </ul>
    }

    <h2>Method</h2>
    @foreach (var section in Recipe.Sections)
    {
        <section class="step-section">
            @if (!string.IsNullOrWhiteSpace(section.Name))
            {
                <h3>@section.Name</h3>
            }
            <ol class="steps">
                @foreach (var block in section.Content)
                {
                    if (block is StepContent step)
                    {
                        <li class="step">@((MarkupString)StepHtml(step.Step.Items))</li>
```

```
                        <li class="note">@note.Value.Trim()</li>                    }
}            </ol>        </section>    }</article>  @code {    /// <summary>The parsed
Cooklang recipe to render.</summary>    [Parameter] public required Recipe Recipe { get;
set; }      /// <summary>The recipe's typed front matter.</summary>    [Parameter] public
required CookFrontMatter FrontMatter { get; set; }      private readonly List<string> _meta
= [];    private readonly List<(string Name, string Qty)> _ingredients = [];      protected
override void OnParametersSet()    {        _meta.Clear();        AddMeta("Serves",
FrontMatter.Servings);        AddMeta("Prep", FrontMatter.PrepTime);        AddMeta("Cook",
FrontMatter.CookTime);        AddMeta("Total", FrontMatter.TotalTime);
AddMeta("Rest", FrontMatter.RestTime);          _ingredients.Clear();        var seen = new
HashSet<string>(StringComparer.OrdinalIgnoreCase);        foreach (var section in
Recipe.Sections)        {            foreach (var block in section.Content)            {
if (block is not StepContent step)                {                    continue;
}                foreach (var item in step.Step.Items)                {
if (item is IngredientItem ingredient && seen.Add(ingredient.Name))                    {
_ingredients.Add((ingredient.Name, Qty(ingredient.Quantity, ingredient.Units)));
}                }            }        }    }      private void AddMeta(string label,
string? value)    {        if (!string.IsNullOrWhiteSpace(value))        {
_meta.Add($"{label} {value}");        }    }      // The inline token run: text interleaved
with ingredient/cookware/timer spans, built as a    // string so adjacent tokens carry no
stray whitespace (a space before a '.' or '(' would show).    private static string
StepHtml(IReadOnlyList<Item> items)    {        var sb = new StringBuilder();        foreach
(var item in items)        {            switch (item)            {                case
TextItem text:                    sb.Append(Enc(text.Value));                    break;
case IngredientItem ingredient:                    var quantity = Qty(ingredient.Quantity,
ingredient.Units);                    sb.Append($"<span class=\"ingredient\">
{Enc(ingredient.Name)}");                    if (quantity.Length > 0)                    {
sb.Append($" <span class=\"qty\">{Enc(quantity)}</span>");                    }
sb.Append("</span>");                    break;                case CookwareItem cookware:
sb.Append($"<span class=\"cookware\">{Enc(cookware.Name)}</span>");
break;                case TimerItem timer:                    sb.Append($"<span
class=\"timer\">{Enc(Qty(timer.Quantity, timer.Units))}</span>");                    break;
}        }        return sb.ToString();    }      private static string Qty(QuantityValue?
quantity, string units)        => string.Join(" ", new[] { quantity?.ToString() ?? "", units
}.Where(part => !string.IsNullOrWhiteSpace(part)));      private static string Enc(string
value) => WebUtility.HtmlEncode(value);}
```

The renderer parses the Cooklang body and hands the model to the component:

CSHARP

```
namespace BeyondCookFormatExample;

using CooklangSharp;
using Microsoft.AspNetCore.Components.Web;
using Pennington.Pipeline;

/// <summary>
/// Renders a parsed <c>.cook</c> recipe by binding the CooklangSharp model to the <see
cref="RecipeView"/> Razor
/// component. All markup lives in the component; this renderer only parses the body and
supplies the parameters.
/// The <see cref="RazorContentRenderer{TComponent}"/> base owns the Blazor
<c>HtmlRenderer</c> dispatch, heading
/// anchors, and outline extraction.
/// </summary>
public sealed class CookContentRenderer : RazorContentRenderer<RecipeView>
{
    /// <summary>Creates the renderer over the Blazor <c>HtmlRenderer</c> resolved from DI.
</summary>
    public CookContentRenderer(HtmlRenderer renderer) : base(renderer)
    {
    }

    /// <inheritdoc/>
    protected override IReadOnlyDictionary<string, object?> BuildParameters(ParsedItem item)
    {
        var parsed = CooklangParser.Parse(item.RawMarkdown);
        if (parsed.Recipe is not { } recipe)
        {
            throw new InvalidOperationException(
                $"Cooklang parse failed: {string.Join("; ", parsed.Diagnostics.Select(d =>
d.Message))}");
        }

        return new Dictionary<string, object?>
        {
            [nameof(RecipeView.Recipe)] = recipe,
            [nameof(RecipeView.FrontMatter)] = (CookFrontMatter)item.Metadata,
        };
    }
}
```

Throwing from `BuildParameters` (here, when the body won't parse) is captured as a `FailedItem` — it lands in the build report and the dev overlay like any markdown failure. The base produces the page-body HTML and its outline; the host or layout supplies the surrounding chrome, the same way it wraps a rendered markdown body.

Register the format

`AddContentFormat` ties the pieces together — a content directory, a file glob, the format key, and the parser and renderer types (resolved from DI). Call it on the `penn` options inside your `AddPennington` callback, alongside `AddMarkdownContent`, so prose and recipes share the host. Because the renderer dispatches through Blazor's `HtmlRenderer`, the host also needs Razor's component services — register `AddRazorComponents()` on the service collection first:

CSHARP

```
builder.Services.AddRazorComponents();

builder.Services.AddPennington(penn =>
{
    penn.AddMarkdownContent<DocFrontMatter>(md => md.BasePageUrl = "/");

    penn.AddContentFormat<CookFrontMatter>("cook", cook =>
    {
        cook.ContentPath = "recipes";
        cook.FilePattern = "*.cook";
        cook.BasePageUrl = "/recipes";
        cook.SectionLabel = "Recipes";
    })
    .UseParser<CookContentParser>()
    .UseRenderer<CookContentRenderer>();
});
```

That's the whole wiring. The pipeline routes each URL to the parser and renderer registered for its format, so `IPageResolver`, the build crawler, navigation, search, and the sitemap treat cook pages exactly like markdown ones — the catch-all `MapGet("/{*path}", IPageResolver resolver)` resolves both without changes.

Verify

- Run `dotnet run --project examples/BeyondCookFormatExample` and open `/` (the markdown landing page) and `/recipes/chicken-piccata/` (a recipe rendered to HTML — title, ingredient list, and method steps).
- Run `dotnet run --project examples/BeyondCookFormatExample -- diag routes`. Each `/recipes/{slug}/` is listed with the `cook` kind next to the markdown `/`.
- Run `dotnet run --project examples/BeyondCookFormatExample -- build output`. Confirm `output/recipes/{slug}/index.html` exists for every recipe and that `output/sitemap.xml` lists each `/recipes/{slug}/` URL.

Related

- How-to: Source content from outside the file system — implement `IContentService` directly when content isn't file-backed.

- Background: Why ContentSource is a union — what `FileSource` is and how the dispatcher routes a format to its parser and renderer.
- Background: The content pipeline — the discover → parse → render path your format plugs into.

CHAPTER 10

Markdown Pipeline

Add a Markdig extension or inline parser

Guides Register any Markdig extension, inline parser, or block parser through ConfigureMarkdownPipeline — wiki-links as the worked example — and check what the default pipeline already enables before you add.

To add syntax Markdig doesn't parse out of the box — `[[wiki-links]]`, a definition shortcut, a custom container — register a Markdig extension or a raw inline/block parser through `PenningtonOptions.ConfigureMarkdownPipeline`. The hook runs after every built-in extension with the resolved `IServiceProvider`, so you extend the same pipeline that renders the rest of the site rather than replacing it.

This is the hook to reach for instead of writing your own renderer. For directives that expand to a string before parsing, a shortcode is lighter; to claim a whole fenced block, use a code-block preprocessor. Everything else — new inline tokens, new block syntax, swapping a renderer — goes through `ConfigureMarkdownPipeline`.

The recipe references `examples/ExtensibilityLabExample/WikiLinkExtension.cs`, which adds a `[[…]]` inline parser to a bare `AddPennington` host.

Before you begin

- An existing Pennington site with markdown rendering wired (see Create your first Pennington site if not).
- A look at what the pipeline already enables, below — several "missing" features are already on.

Check what's already enabled

Pennington's pipeline is Markdig's `UseAdvancedExtensions()` plus its own renderers. Before you register anything, confirm the feature isn't already parsed — re-adding an extension that's present is a double-register that, depending on the extension, duplicates parsers, reorders them, or shadows the built-in.

`UseAdvancedExtensions()` already turns on the usual advanced set — auto-identifiers, footnotes, grid and pipe tables, task lists, **mathematics**, and the rest — and Pennington layers its own front matter, syntax highlighting, tabbed code, content tabs, custom alerts, and Mdazor rendering on top. The extensions catalog is the full list to check against.

Math already works — don't re-register it

`UseAdvancedExtensions()` includes the mathematics extension, so math is parsed today with no configuration. Inline `$E = mc^2$` renders to `<span class="math">\(E = mc^2\)</span>` and a `$$…$$` block to `<div class="math">\[…\]</div>` — already in the `\(…\)` / `\[…\]` delimiters KaTeX and MathJax expect. Registering `UseMathematics()` again is the double-register to avoid. Turning that markup into typeset math is a client-side step, not a Markdig one: load KaTeX (or MathJax) through the head content option and re-run its auto-render on `spa:commit`, exactly the head-content-plus-script pattern in Ship a custom client-side widget.

Write a custom inline parser

A genuinely new token needs a parser. `WikiLinkExtension` is an `IMarkdownExtension` whose `Setup` inserts one inline parser; the parser claims `[[Target]]` / `[[Target|Label]]` and emits an ordinary `LinkInline` so Markdig's own anchor renderer writes the `<a>`.

CSHARP

```
namespace ExtensibilityLabExample;

using System.Text;
using Markdig;
using Markdig.Helpers;
using Markdig.Parsers;
using Markdig.Parsers.Inlines;
using Markdig.Renderers;
using Markdig.Renderers.Html;
using Markdig.Syntax;
using Markdig.Syntax.Inlines;

/// <summary>
/// A Markdig extension that teaches the pipeline a new inline token: the
/// <c>[[Target]]</c> / <c>[[Target|Label]]</c> wiki-link. Each match renders as an
/// internal anchor — <c>&lt;a class="wikilink"
href="/notes/&lt;slug&gt;/"&gt;Label&lt;/a&gt;</c>
/// — so digital-garden cross-references resolve like ordinary links.
/// <para>
/// Registered through <see
cref="Pennington.Infrastructure.PenningtonOptions.ConfigureMarkdownPipeline"/>
/// in <c>Program.cs</c>; that hook runs after Pennington's built-in extensions, so the
/// extension only adds the one parser the built-ins don't already supply.
/// </para>
/// <para>
/// Backs how-to 2.2.65 <c>/how-to/markdown-pipeline/markdig-extension</c>.
/// </para>
/// </summary>
public sealed class WikiLinkExtension : IMarkdownExtension
{
    /// <summary>Inserts the wiki-link inline parser ahead of the built-in link parser.
</summary>
    public void Setup(MarkdownPipelineBuilder pipeline)
    {
        // Run before the CommonMark link parser so a leading "[[" is claimed as a
        // wiki-link instead of being read as the start of two nested "[...]" links.
        if (!pipeline.InlineParsers.Contains<WikiLinkInlineParser>())
        {
            pipeline.InlineParsers.InsertBefore<LinkInlineParser>(new
WikiLinkInlineParser());
        }
    }

    /// <summary>No renderer wiring needed — the emitted <see cref="LinkInline"/> uses
Markdig's own anchor renderer.</summary>
    public void Setup(MarkdownPipeline pipeline, IMarkdownRenderer renderer)
    {
    }
}

/// <summary>
```

```
/// </summary>public sealed class WikiLinkInlineParser : InlineParser{    ///
<summary>Registers <c>[</c> as the trigger; <see cref="Match"/> bails unless it doubles.
</summary>    public WikiLinkInlineParser()    {        OpeningCharacters = ['['];    }
/// <summary>Matches <c>[[Target]]</c> / <c>[[Target|Label]]</c> and emits the anchor
inline.</summary>    public override bool Match(InlineProcessor processor, ref StringSlice
slice)    {        // Only a doubled opener is a wiki-link; "[" alone belongs to the link
parser.        if (slice.PeekChar() != '[')        {            return false;        }
var saved = slice;        slice.SkipChar(); // first [        slice.SkipChar(); // second [
// Scan the inner text up to the closing "]]". Wiki-links never span lines        // or
nest, so a newline or a fresh "[" aborts the match.        var contentStart = slice.Start;
var contentEnd = -1;        var c = slice.CurrentChar;        while (c != '\0')        {
if (c == ']' && slice.PeekChar() == ']')            {                contentEnd =
slice.Start - 1; // last char before "]]"                slice.SkipChar(); // first ]
slice.SkipChar(); // second ]                break;            }            if (c is '\n'
or '\r' or '[')            {                break;            }            c =
slice.NextChar();        }        // Unterminated or empty ("[[]]"): restore the slice and
let other parsers try.        if (contentEnd < contentStart)        {            slice =
saved;            return false;        }        var inner = new StringSlice(slice.Text,
contentStart, contentEnd).AsSpan().ToString();        var (target, label) =
SplitTargetAndLabel(inner);        if (target.Length == 0)        {            slice =
saved;            return false;        }        var spanStart =
processor.GetSourcePosition(saved.Start, out var line, out var column);        var spanEnd =
processor.GetSourcePosition(slice.Start - 1);        var link = new LinkInline        {
Url = $"/notes/{Slugify(target)}/",            IsClosed = true,            Span = new
SourceSpan(spanStart, spanEnd),            Line = line,            Column = column,
};        // The class is the contract downstream consumers key on. Internal hrefs like
// the one above are still rewritten by the response pipeline (locale prefixing,        //
base-URL prefixing), so wiki-links stay portable across deploys and locales.
link.GetAttributes().AddClass("wikilink");        link.AppendChild(new
LiteralInline(label));        processor.Inline = link;        return true;    }      //
"Target|Label" → (Target, Label); "Target" → (Target, Target).    private static (string
Target, string Label) SplitTargetAndLabel(string inner)    {        var pipe =
inner.IndexOf('|');        if (pipe < 0)        {            var only = inner.Trim();
return (only, only);        }        var target = inner[..pipe].Trim();        var label =
inner[(pipe + 1)..].Trim();        return (target, label.Length == 0 ? target : label);    }
// Lowercase, collapse runs of non-alphanumerics to single dashes, trim trailing dashes.
private static string Slugify(string value)    {        var sb = new
StringBuilder(value.Length);        var prevDash = false;        foreach (var ch in
value.Trim().ToLowerInvariant())        {            if (char.IsLetterOrDigit(ch))
{                sb.Append(ch);                prevDash = false;            }
else if (!prevDash && sb.Length > 0)            {                sb.Append('-');
prevDash = true;            }        }        return sb.ToString().TrimEnd('-');    }}
```

Three details carry the parser:

- **`OpeningCharacters` triggers `Match`.** The parser registers `[`, then returns `false` immediately unless the next character is also `[`, leaving single-bracket `[text](url)` links to the built-in parser.
- **Insert before the link parser.** `InsertBefore<LinkInlineParser>` gives the wiki-link parser first claim on `[[`; otherwise the CommonMark link parser reads the brackets as nested links.

- **Emit a `LinkInline` and tag it.** Setting `Url`, appending a `LiteralInline` label, and calling `GetAttributes().AddClass("wikilink")` produces a normal anchor with a class downstream code can target. Restore the saved `StringSlice` and return `false` on any malformed input so other parsers get their turn.

A block-level construct follows the same shape with a `BlockParser` inserted into `pipeline.BlockParsers`; to change how an existing node renders, swap its renderer in the second `Setup(MarkdownPipeline, IMarkdownRenderer)` overload the way Pennington's own syntax-highlighting and scrollable-tables extensions replace Markdig's default code-block and table renderers.

Register it through ConfigureMarkdownPipeline

`ConfigureMarkdownPipeline` is `Action<MarkdownPipelineBuilder, IServiceProvider>`, set inside the `AddPennington` lambda. It runs after the built-ins, and the second argument is the resolved service provider — resolve dependencies from it when a parser needs them (an `HttpClient`, options, a file-watched index). The lab discards it with `_`:

CSHARP

```
penn.ConfigureMarkdownPipeline = (pipeline, _) =>
    pipeline.Extensions.AddIfNotAlready(new WikiLinkExtension());
```

Result

The demo page uses both wiki-link forms and a math block:

MARKDOWN

```
---
title: Wiki-links and math markup
description: A custom [[wiki-link]] inline parser registered via ConfigureMarkdownPipeline,
plus the math markup the built-in pipeline already emits.
---

The `WikiLinkExtension` registered through `penn.ConfigureMarkdownPipeline`
teaches Markdig one new inline token. A bare target links to its slug — see
the [[Glossary]] — and the piped form sets its own label, like
[[content-pipeline|how rendering works]]. Each renders as
`<a class="wikilink" href="/notes/<slug>/">`, so the response pipeline prefixes
the internal href the same way it prefixes any other link.

A single bracket is untouched: an ordinary [link](https://example.com) still
parses through the built-in CommonMark link parser.

## Math is already on

No extension registration is needed for math — `UseAdvancedExtensions` (part of
Pennington's default pipeline) already parses it. Inline math like $E = mc^2$
renders to a `<span class="math">`, and a display block to a `<div class="math">`:

$$
\int_0^1 x^2 \,\mathrm{d}x = \frac{1}{3}
$$

The markup ships in KaTeX/MathJax delimiters; rendering it is a client-side step,
not a Markdig one.
```

The wiki-links render as anchors carrying `class="wikilink"`, and the math block renders as the KaTeX-ready `<div class="math">`:

HTML

```
<a href="/notes/glossary/" class="wikilink">Glossary</a>
<a href="/notes/content-pipeline/" class="wikilink">how rendering works</a>

<div class="math">
\[
\int_0^1 x^2 \,\mathrm{d}x = \frac{1}{3}
\]</div>
```

How your custom HTML survives the response pipeline

After rendering, every page passes through the response rewriters and is harvested by the site projection. Custom markup rides through the same as built-in markup, with two things to do.

Emit your HTML into the content region. Search, llms.txt, and the link audit read the element named by `PenningtonOptions.SiteProjection.ContentSelector` (the lab uses `article`); markup placed there is captured and indexed by its visible text, while anything injected into the chrome or `<head>` is not. See Tune what the search box returns and Make the site discoverable to LLM crawlers.

Watch the word-break rewriter if you emit text-bearing `<span>`s. The shipped rewriters rewrite the internal `href`s you emit — adding the locale prefix and deploy base URL, so a wiki-link to `/notes/glossary/` stays portable — but the opt-in word-break rewriter's default selector includes `span`. If a client script reads a `<span>` verbatim (a math span, say), leave word-break off for it or narrow its selector.

Verify

On your own site, register your extension through `ConfigureMarkdownPipeline`, put a `[[Target]]` (or your token) in any content page, run your site, and view source on that page: your token rendered as the markup your parser emits — `<a class="wikilink" href="/notes/…/">` for the wiki-link parser. Then run your static build and grep the page's `index.html` in the output for the same markup. The build-time link audit treats custom anchors as real internal links, so it reports any `href` with no matching page as broken — which is the audit working, not a misfire.

To check against the reference implementation, run `dotnet run --project examples/ExtensibilityLabExample`, visit `/wikilinks-demo/`, and confirm the wiki-links are `<a class="wikilink" href="/notes/…/">` and the math block is `<div class="math">`.

Related

- How-to: Expand a directive before Markdig parses — string expansion before the parser runs, for stamping values rather than new syntax.
- How-to: Add a custom fence syntax — claim a fenced block instead of an inline token.
- How-to: Ship a custom client-side widget — the head-content + `spa:commit` pattern the math example uses.
- Reference: Markdown extensions catalog — the syntax the default pipeline already provides.
- Background: The response-processing pipeline — where the rewriters and projection run.

Add a custom fence syntax

Guides Implement ICodeBlockPreprocessor to claim a fence language or :modifier suffix and return pre-rendered HTML before the default highlighter chain runs.

To intercept a fence language or `:modifier` suffix — a chart block, a plaintext wrapper, an xmldocid resolver — implement `ICodeBlockPreprocessor`. The preprocessor returns pre-rendered HTML before the default highlighter chain runs, including the rendered `<pre><code>...</code></pre>`. For line-level CSS classes on an otherwise normal code block, trailing-comment directives are the lighter-weight choice — see Annotate specific lines in a code block.

The recipe references `examples/ExtensibilityLabExample/LineCountPreprocessor.cs`, which claims the `linecount` fence.

Before you begin

- An existing Pennington site with markdown rendering wired (see Create your first Pennington site if not).
- A chosen fence identifier — either a full `languageId` (`linecount`) or a `:modifier` suffix (`csharp:symbol`).

Write the preprocessor

Implement Pennington.Markdown.Extensions.ICodeBlockPreprocessor as a sealed class. `TryProcess(code, languageId)` receives the full fence info string unchanged. Compare it case-insensitively against the claimed language id or modifier, return `null` for anything else so the next preprocessor or the default highlighter can handle it, and otherwise build the wrapper HTML around the encoded source.

CSHARP

```
namespace ExtensibilityLabExample;

using System.Net;
using Pennington.Markdown.Extensions;

/// <summary>
/// Implements <see cref="ICodeBlockPreprocessor"/>. Intercepts fenced
/// code blocks tagged <c>linecount</c> and renders them inside a
/// <c>&lt;figure class="linecount"&gt;</c> wrapper that reports how many
/// lines the snippet spans. Returns <see langword="null"/> for any
/// other language so the default highlighter chain runs.
/// <para>
/// <see cref="CodeBlockPreprocessResult.SkipTransform"/> is <c>true</c>
/// because the output already contains the line count badge we want and
/// should not be touched by <c>CodeTransformer</c>'s <c>// [!code]</c>
/// annotation pass.
/// </para>
/// <para>
/// Backs how-to 2.3.20 <c>/how-to/extensibility/code-block-preprocessor</c>.
/// </para>
/// </summary>
public sealed class LineCountPreprocessor : ICodeBlockPreprocessor
{
    /// <summary>
    /// 500 — higher than the shipped code-fragment preprocessors so
    /// <c>linecount</c> wins over any language-modifier preprocessor
    /// that might claim the same fence info string.
    /// </summary>
    public int Priority => 500;

    public CodeBlockPreprocessResult? TryProcess(string code, string languageId)
    {
        if (!string.Equals(languageId, "linecount", StringComparison.OrdinalIgnoreCase))
        {
            return null;
        }

        var lineCount = CountLines(code);
        var encoded = WebUtility.HtmlEncode(code);

        var html = $"""
            <figure class="linecount" data-extensibility-lab="line-count-preprocessor">
              <figcaption>Line count: <strong>{lineCount}</strong></figcaption>
              <pre><code>{encoded}</code></pre>
            </figure>
            """;

        return new CodeBlockPreprocessResult(
            HighlightedHtml: html,
            BaseLanguage: "linecount",
            SkipTransform: true);
```

```
    {        if (string.IsNullOrEmpty(code))        {            return 0;        }
var count = 1;        foreach (var ch in code)        {            if (ch == '\n')
{                count++;            }        }        // Trim trailing newline so
"one\ntwo\n" counts as 2.        if (code.EndsWith('\n'))        {            count--;
}          return count;    }}
```

The returned `CodeBlockPreprocessResult` carries the pre-rendered HTML, the `BaseLanguage` CSS class Pennington stamps on the block, and `SkipTransform`. Set `SkipTransform` to `true` when the output is final and the `[!code ...]` annotation pass should not re-process it.

Pick a Priority value

`CodeBlockRenderingService` sorts preprocessors by `Priority` descending and returns the first non-null result. The only shipped preprocessor is the tree-sitter one that claims `:symbol` and `:symbol-diff`, at `100`. `LineCountPreprocessor` uses `500` so its `linecount` fence runs ahead of the tree-sitter preprocessor — relevant only if both could claim the same info string. Pick above `100` to beat the shipped `:symbol` preprocessor on a contested `:modifier`, or below it to let `:symbol` resolve first.

Register the implementation

Pennington collects every `ICodeBlockPreprocessor` from DI. Register anywhere after `AddPennington` — there is no `PenningtonOptions` knob. `AddTreeSitter` performs the equivalent registration for its `:symbol` preprocessor.

CSHARP

```
builder.Services.AddSingleton<ICodeBlockPreprocessor, LineCountPreprocessor>();
```

Verify

On your own site, add a fence tagged with the language your preprocessor claims, then run `dotnet run` and view source on the page that holds it. A claimed `linecount` fence renders inside a `<figure>` with the line-count badge instead of going through the default highlighter, while adjacent fences with other languages keep flowing through the highlighter chain:

HTML

```
<figure class="linecount" data-extensibility-lab="line-count-preprocessor">
  <figcaption>Line count: <strong>3</strong></figcaption>
  <pre><code>first line
second line
third line</code></pre>
</figure>
```

The wrapper markup proves `TryProcess` returned a result rather than the default highlighter rendering the block. Confirm too that a static build picks the preprocessor up: `dotnet run -- build output`, then grep the emitted HTML for the same wrapper.

To see the shipped example instead, run `dotnet run --project examples/ExtensibilityLabExample` and visit `/line-count-demo/` — the `linecount` fence renders the figure above while the adjacent `text` fence highlights through the default chain.

Related

- Reference: Highlighting interfaces — full signatures for `ICodeHighlighter`, `ICodeBlockPreprocessor`, `HighlightingService`, and `TextMateLanguageRegistry`
- How-to: Annotate code blocks — trailing-comment directives when only line classes are needed
- Background: The syntax-highlighting cascade — why preprocessors run before the highlighter and how `CodeTransformer` interacts with `SkipTransform`

Add a custom syntax highlighter

Guides Implement ICodeHighlighter for a fence language TextMateSharp doesn't cover and register it with HighlightingOptions.AddHighlighter.

To color a fence language TextMateSharp does not cover — a DSL, config format, or domain notation — implement `ICodeHighlighter`. For line-level callouts on a language already supported, see Annotate specific lines in a code block. For transforming the fence body rather than coloring its tokens, see Add a custom fence syntax.

The recipe below references `examples/ExtensibilityLabExample/PipelineHighlighter.cs`, which stakes out a fictional `pipeline` DSL against a bare `AddPennington` host.

Before you begin

- An existing Pennington site rendering markdown fences (see Create your first Pennington site if not).
- A target language not already served by `TextMateHighlighter` (priority 50) or `ShellHighlighter` (priority 75) — render a fence and inspect the emitted HTML for built-in token spans to confirm. `PlainTextHighlighter` is the hardcoded final fallback inside `HighlightingService`, reached only when no registered highlighter matches; it is not on the priority chain.

Write the highlighter

Implement Pennington.Highlighting.ICodeHighlighter as a sealed class. `Highlight(code, language)` returns the full HTML for the block, including the outer `<pre><code>` wrapper — the implementation owns escaping. Use `WebUtility.HtmlEncode` on every literal not wrapped in a span; anything missed becomes an injection vector.

CSHARP

```
namespace ExtensibilityLabExample;

using System.Net;
using System.Text;
using System.Text.RegularExpressions;
using Pennington.Highlighting;

/// <summary>
/// Implements <see cref="ICodeHighlighter"/> for a fictional <c>pipeline</c>
/// DSL — pipelines of the form
/// <c>source "name" -&gt; filter where=paid | transform total=sum | sink "name"</c>.
/// <para>
/// Keywords (<c>source</c>, <c>filter</c>, <c>transform</c>, <c>sink</c>)
/// and arrows (<c>-&gt;</c>, <c>|</c>) get wrapped in spans with CSS
/// classes so the stylesheet can theme them. Unrecognized tokens are
/// HTML-encoded and left alone.
/// </para>
/// <para>
/// Priority 100 — above <see cref="TextMateHighlighter"/>'s default (50)
/// and below <see cref="ShellHighlighter"/>'s 75 so this highlighter only
/// owns the <c>pipeline</c> language and nothing else.
/// </para>
/// <para>
/// Backs how-to 2.3.30 <c>/how-to/extensibility/custom-highlighter</c>.
/// </para>
/// </summary>
public sealed partial class PipelineHighlighter : ICodeHighlighter
{
    private static readonly HashSet<string> _keywords =
new(StringComparer.OrdinalIgnoreCase)
        { "source", "filter", "transform", "sink", "where" };

    /// <summary>The languages this highlighter claims.</summary>
    public IReadOnlySet<string> SupportedLanguages { get; } = new HashSet<string>
(StringComparer.OrdinalIgnoreCase)
        { "pipeline" };

    /// <summary>Priority for highlighter dispatch — higher wins.</summary>
    public int Priority => 100;

    /// <summary>Produce the highlighted HTML for one fence's body.</summary>
    public string Highlight(string code, string language)
    {
        if (string.IsNullOrEmpty(code))
        {
            return string.Empty;
        }

        var sb = new StringBuilder();
        sb.Append("<pre><code data-extensibility-lab=\"pipeline-highlighter\">");
```

```
            var position = 0;              while (position < line.Length)           {
// Arrow `->`                if (position + 1 < line.Length && line[position] == '-' &&
line[position + 1] == '>')                {                    sb.Append("<span
class=\"pipeline-arrow\">-&gt;</span>");                    position += 2;
continue;                }                  // Pipe `|`                if (line[position] ==
'|')                {                    sb.Append("<span class=\"pipeline-pipe\">|
</span>");                    position++;                    continue;                }
// String literal "..."                if (line[position] == '"')                {
var end = line.IndexOf('"', position + 1);                    if (end > 0)
{                        var literal = line[position..(end + 1)];
sb.Append("<span class=\"pipeline-string\">");
sb.Append(WebUtility.HtmlEncode(literal));                        sb.Append("</span>");
position = end + 1;                        continue;                    }                }
// Identifier / keyword                var identMatch = IdentifierRegex().Match(line,
position);                if (identMatch.Success && identMatch.Index == position)
{                    var word = identMatch.Value;                    if
(_keywords.Contains(word))                    {                        sb.Append("<span
class=\"pipeline-keyword\">");
sb.Append(WebUtility.HtmlEncode(word));                        sb.Append("</span>");
}                    else                    {
sb.Append(WebUtility.HtmlEncode(word));                    }                    position +=
word.Length;                    continue;                }                // Fallback:
encode one character and continue.
sb.Append(WebUtility.HtmlEncode(line[position].ToString()));                position++;
}             sb.Append('\n');        }          sb.Append("</code></pre>");        return
sb.ToString();    }      [GeneratedRegex(@"[A-Za-z_][A-Za-z0-9_\-]*")]    private static
partial Regex IdentifierRegex();}
```

Two values shape how the highlighter slots into the chain:

- `SupportedLanguages` — every token returned here maps to a fence language (` ```pipeline `) that routes to this implementation. Use `StringComparer.OrdinalIgnoreCase` so `Pipeline` and `PIPELINE` match too.
- `Priority` — higher wins when two highlighters claim the same language. For a brand-new language like `pipeline` that nothing else touches, the value is irrelevant — any number routes the fence to your implementation. Priority matters only when you override a language a shipped highlighter already owns: pick above `75` to beat `ShellHighlighter` (`bash` / `shell` / `sh`), or above `50` to beat `TextMateHighlighter` (every grammar it can load). `PipelineHighlighter` uses `100` purely to make the intent — "this wins outright" — legible.

Register the highlighter

`PenningtonOptions.Highlighting.AddHighlighter` inserts the instance into the priority-sorted chain resolved by `HighlightingService`. Call it inside the `AddPennington` delegate so the highlighter is active for both `dotnet run` and `dotnet run -- build output`.

CSHARP

```
builder.Services.AddPennington(penn =>
{
    penn.Highlighting.AddHighlighter(new PipelineHighlighter());
});
```

A markdown fence tagged with one of the strings from `SupportedLanguages` now routes to the custom highlighter instead of the fallback chain.

MARKDOWN

```
```pipeline
source "orders" -> filter where=paid | transform total=sum | sink "warehouse"
```
```

Style the emitted classes

The highlighter only wraps tokens in spans — `pipeline-keyword`, `pipeline-arrow`, `pipeline-pipe`, `pipeline-string`. Until a stylesheet colors those classes the block renders in the surrounding body color, so the fence looks no different from the unstyled `text` fallback. Built-in languages look colored out of the box because the shipped theme already styles TextMate's `hljs-*` classes; your custom classes are new, so the theme says nothing about them. Add the rules to the stylesheet the site already serves:

CSS

```
.pipeline-keyword { color: #c678dd; font-weight: 600; }
.pipeline-arrow   { color: #56b6c2; }
.pipeline-pipe    { color: #56b6c2; }
.pipeline-string  { color: #98c379; }
```

The class names are whatever `Highlight` emits — keep the CSS and the span classes in sync. Reuse the theme's existing token colors (or its CSS custom properties) so the new language matches the rest of the site instead of introducing a fourth palette.

Verify

On your own site, render a page with a `pipeline` fence next to a `text` fence and load it in a browser:

- The `pipeline` fence shows colored keywords, arrows, and string literals; the `text` fence stays a single color. If both blocks look identical, the highlighter is running but the CSS rules above are missing or not loaded.
- View source: the `pipeline` block carries `<span class="pipeline-keyword">` tokens. If it does not, the fence never reached your highlighter — confirm the fence tag matches a string in `SupportedLanguages` and the registration runs inside the `AddPennington` delegate.
- Static build: run your build (`dotnet run -- build output`) and search the emitted HTML for `class="pipeline-keyword"` to confirm the highlighter runs during publish, not only under `dotnet run`.

For a complete worked highlighter and a demo fence that emits the `pipeline-*` spans, run `dotnet run --project examples/ExtensibilityLabExample` and visit `/pipeline-demo/`.

Related

- Reference: Highlighting interfaces
- Background: The syntax-highlighting cascade
- Related how-to: Register a code-block preprocessor

Expand a directive before Markdig parses

Guides Register an IShortcode handler so directives in markdown expand to text or HTML before the rest of the pipeline runs.

To stamp a value into a page — a version string, a repo link, a build timestamp — implement `IShortcode`. The handler runs **before** Markdig parses the page, so the string it returns becomes part of the markdown source and flows through the rest of the pipeline as if you had typed it yourself.

> **Note**
>
> The shortcode expander only runs on markdown pages. Razor templates, HTML files, and other content services bypass it.

Write a shortcode

Implement `IShortcode` as a sealed class. `ExecuteAsync` receives the parsed invocation — positional args, named args, and inline content (null for self-closing tags) — plus the host page's route and metadata. Return the string that should replace the directive in the markdown source. Return raw HTML when the output should not be re-parsed as markdown.

The lab below claims `GitHubRepo` and turns a positional repo slug into an anchor:

CSHARP

```
public sealed class GitHubRepoShortcode : IShortcode
{
    /// <inheritdoc />
    public string Name => "GitHubRepo";

    /// <inheritdoc />
    public Task<string> ExecuteAsync(
        ShortcodeInvocation invocation,
        ShortcodeContext context,
        CancellationToken cancellationToken)
    {
        // Throw idiomatic guard clauses — the expander catches and degrades to a
        // build warning + HTML comment so the page still ships.
        if (invocation.PositionalArgs.Count == 0)
        {
            throw new ArgumentException("GitHubRepo requires a repo slug as the first
positional argument.");
        }

        var slug = invocation.PositionalArgs[0];
        var encoded = WebUtility.HtmlEncode(slug);
        var html = $"""<a class="github-repo" data-extensibility-lab="github-repo-shortcode"
href="https://github.com/{encoded}">{encoded}</a>""";
        return Task.FromResult(html);
    }
}
```

A few patterns worth copying:

- **HTML-encode untrusted input.** `WebUtility.HtmlEncode` everywhere a user-supplied value touches the output. The string is spliced into the source and re-rendered, so injection vectors are real.
- **Throw idiomatic guard clauses.** The expander catches every handler exception, surfaces the message as a build warning, and emits an HTML comment in place. One bad call site never fails the build.
- **Lean on context.** `context.Route.SourceFile` tells the handler which page invoked it — useful for path-relative resolution.

Register the handler

Pennington collects every `IShortcode` from DI. Register anywhere after `AddPennington` — there is no `PenningtonOptions` knob.

CSHARP

```
builder.Services.AddSingleton<IShortcode, GitHubRepoShortcode>();
```

The expander reads `IEnumerable<IShortcode>` at construction; when two handlers share a `Name`, the last-registered wins. The built-in `Version` shortcode is registered as a singleton inside `AddPennington`, so any handler you register afterwards joins the same dispatch table.

Result

A markdown page that mixes the custom handler with a built-in:

MARKDOWN

````
---
title: Shortcodes demo
description: Pre-render shortcodes expanded before Markdig parses the page.
---

Shortcodes are textual directives the renderer expands **before** Markdig
parses the page, so handler output flows through the rest of the pipeline
as regular markdown. The framework ships one (`Version`); this lab adds
`GitHubRepo` to show how a custom handler slots into the same dispatch
table.

## Built-in: Version

Write `\<?# Version /?>` and the renderer substitutes the entry
assembly's version: <?# Version /?>.

Pass `format=major` for just the leading component: <?# Version format=major /?>.

## Custom: GitHubRepo

`GitHubRepo` takes one positional argument — the repository slug — and
emits an anchor. Source: <?# GitHubRepo "anthropic/anthropic-sdk-python" /?>.

## Inside a fenced block, shortcodes still expand

The expander runs across the whole markdown source, so install snippets
can stamp the real version into a copyable command:

```bash
dotnet add package Pennington --version <?# Version /?>
```

To show a literal directive without expanding it — say, when documenting
the syntax in a code sample — prefix the opener with a backslash. The
expander consumes the backslash and emits the directive as-is:

```markdown
Run \\<?# Version /?> to stamp the host version.
```
````

After expansion, the prose reads as if the version and link were authored inline. The fenced code block at the bottom is left alone so the demo page can document the syntax it uses.

Verify

On your own site, add the handler and call it from a page:

- Register the handler after `AddPennington`, drop `<?# GitHubRepo "owner/repo" /?>` into any markdown page, and load that page in a browser — the directive renders as a working link to the GitHub repo. View source and look for `data-extensibility-lab="github-repo-shortcode"` on the anchor (rename the attribute to your own once you copy the handler). That attribute means `ExecuteAsync` produced the HTML rather than Markdig parsing it as markdown.
- Static build: run your build (`dotnet run -- build output`) and grep the emitted HTML for the link to confirm the expander runs during publish, not only under `dotnet run`.
- If the directive renders verbatim instead of expanding, the handler's `Name` does not match the directive (names are case-insensitive but must otherwise match) or the registration ran before `AddPennington`.

The lab ships a complete worked version:

- Run `dotnet run --project examples/ExtensibilityLabExample` and visit `/shortcodes-demo/` — the page shows the host's version and a working link to the GitHub repo.
- Static build: `dotnet run --project examples/ExtensibilityLabExample -- build output` — grep the emitted HTML for the version string to confirm the expander runs during publish.

Syntax and built-ins

For the full directive grammar, the shipped `Version` and `PackageVersion` shortcodes, and the error-degradation contract, see the Shortcodes section of the Markdown extensions catalog. The essentials follow.

A shortcode call has three shapes:

MARKDOWN

```
<?# Name /?>                                ← self-closing
<?# Name positional key=value /?>           ← positional and named arguments
<?# Name ?>inline content<?#/ Name ?>      ← block form with content
```

Names are case-insensitive and match the handler's `Name` property. Values that contain whitespace must be double-quoted: `title="A Long Title"`. Shortcodes expand everywhere in the markdown source — including inside fenced code blocks — so install snippets can carry the real version string straight out of the build:

BASH

```
dotnet add package Pennington --version 0.1.6-alpha.0.14
```

To *document* the syntax rather than call into it, prefix the opener with a backslash. The expander consumes the backslash and emits the directive as-is; Markdig HTML-encodes the angle brackets downstream so the reader sees the literal text in their rendered prose or code sample.

MARKDOWN

```
Run \<?# Version /?> to stamp the host's version into a page.
```

Handler exceptions and unknown names degrade automatically: each produces an HTML comment in place of the directive (`<!-- Pennington: shortcode 'Name' failed: <message> -->`) plus a warning diagnostic, so one bad call site never fails the render. If a particular failure should fail the build, register a response processor that promotes the matching diagnostic to error severity.

Related

- Reference: Pennington.Markdown.Shortcodes.IShortcode — the `IShortcode` interface and the `ShortcodeInvocation` / `ShortcodeContext` it receives.
- Reference: Shortcodes in the Markdown extensions catalog — directive grammar, built-ins, and error semantics.
- How-to: Add a custom fence syntax — intercept fenced code blocks instead of inline directives.

Attach derived metadata to every page

Guides Implement IMetadataEnricher to merge derived values like reading time or git timestamps into ParsedItem.Derived, kept separate from authored front matter.

To compute values from a page rather than have an author type them — reading time, a git last-modified date, a word count — implement `IMetadataEnricher`. Each enricher contributes a dictionary that `MetadataEnrichmentService` merges into `ParsedItem.Derived`. Derived values land in their own bag, not in the strongly-typed `Metadata`, so authored front matter stays the single source of truth and computed values can change between builds without rewriting any page.

Before you begin

- An existing Pennington site with markdown rendering wired (see Create your first Pennington site if not).

Reading time ships built in

`AddPennington` registers `ReadingTimeEnricher` by default, so every page with body text carries an estimate under the `reading_time_minutes` key. The estimate divides the word count by 200 words per minute and rounds up, with a floor of one minute. A page with no words contributes no key, so consumers guard the read with `TryGetValue`. The shipped enricher is a pure function of `ParsedItem.RawMarkdown` — no file access:

CSHARP

```
namespace Pennington.Pipeline;

/// <summary>
/// Estimates reading time from the markdown body and contributes it as
/// <c>reading_time_minutes</c>. A pure function of <see cref="ParsedItem.RawMarkdown"/>
/// — no file access, no external dependencies.
/// </summary>
public sealed class ReadingTimeEnricher : IMetadataEnricher
{
    /// <summary>Words read per minute used to derive the estimate.</summary>
    private const int WordsPerMinute = 200;

    /// <summary>Key written into <see cref="ParsedItem.Derived"/>.</summary>
    public const string Key = "reading_time_minutes";

    /// <inheritdoc/>
    public Task<IReadOnlyDictionary<string, object?>> EnrichAsync(ParsedItem item)
    {
        var words = CountWords(item.RawMarkdown);
        if (words == 0)
        {
            return Task.FromResult<IReadOnlyDictionary<string, object?>>(
                new Dictionary<string, object?>());
        }

        var minutes = Math.Max(1, (int)Math.Ceiling(words / (double)WordsPerMinute));
        return Task.FromResult<IReadOnlyDictionary<string, object?>>(
            new Dictionary<string, object?> { [Key] = minutes });
    }

    private static int CountWords(string text)
    {
        if (string.IsNullOrWhiteSpace(text))
        {
            return 0;
        }

        var count = 0;
        var inWord = false;
        foreach (var c in text)
        {
            if (char.IsWhiteSpace(c))
            {
                inWord = false;
            }
            else if (!inWord)
            {
                inWord = true;
                count++;
            }
        }
```

```
        return count;    }}
```

Expose the key as a `const` (as `ReadingTimeEnricher.Key` does) so consumers reference it without retyping the string.

Write an enricher

Implement `IMetadataEnricher` and return the keys you contribute. `EnrichAsync` receives the parsed item and returns an `IReadOnlyDictionary<string, object?>`; return an empty dictionary to contribute nothing for a given page. `GitTimestampEnricher` reads the source file's timestamp from `ParsedItem.Route.SourceFile` and contributes a `git_last_modified` date — the value a real enricher would instead pull from `git log -1`. Pages with no file on disk (generated content) contribute nothing:

CSHARP

```csharp
public sealed class GitTimestampEnricher : IMetadataEnricher
{
    /// <summary>Key written into <see cref="ParsedItem.Derived"/>.</summary>
    public const string Key = "git_last_modified";

    /// <inheritdoc />
    public Task<IReadOnlyDictionary<string, object?>> EnrichAsync(ParsedItem item)
    {
        var path = item.Route.SourceFile?.Value;
        if (path is null || !File.Exists(path))
        {
            return Task.FromResult<IReadOnlyDictionary<string, object?>>(
                new Dictionary<string, object?>());
        }

        var modified = File.GetLastWriteTimeUtc(path).ToString("yyyy-MM-dd");
        return Task.FromResult<IReadOnlyDictionary<string, object?>>(
            new Dictionary<string, object?> { [Key] = modified });
    }
}
```

Register your enricher

Register the implementation after `AddPennington` — there is no `PenningtonOptions` knob. `MetadataEnrichmentService` runs every registered enricher in registration order and merges each contribution into `Derived`, so a later enricher overrides an earlier one on a key collision.

CSHARP

```csharp
builder.Services.AddTransient<IMetadataEnricher, GitTimestampEnricher>();
```

Read derived metadata in a component

The renderer exposes the merged `Derived` dictionary to every Mdazor component under the `Derived` context key. A component reads it through `[CascadingParameter] public MdazorContext? Context` — no tag attributes, the dictionary cascades in from the page being rendered. `LastModified.razor` reads the `git_last_modified` key and renders the date:

RAZOR

```
@* LastModified — reads the ambient MdazorContext that Pennington populates for each
   page and renders the git_last_modified date contributed by GitTimestampEnricher.
   The enricher merges into ParsedItem.Derived; the renderer exposes that dictionary
   under the "Derived" context key. Registered in Program.cs with
   services.AddMdazorComponent<LastModified>(). *@
@using Mdazor
@using Microsoft.AspNetCore.Components

@if (Date is not null)
{
    <p class="last-modified" data-extensibility-lab="last-modified">Last modified: @Date</p>
}

@code {
    // Pennington cascades the page's facts in per render — no tag attributes needed.
    [CascadingParameter] public MdazorContext? Context { get; set; }

    // The "Derived" key carries the IMetadataEnricher contributions for this page.
    private string? Date =>
        Context?["Derived"] is IReadOnlyDictionary<string, object?> derived
        && derived.TryGetValue(GitTimestampEnricher.Key, out var value)
            ? value?.ToString()
            : null;
}
```

Register the component with `AddMdazorComponent<LastModified>()`, then drop `<LastModified />` into any page body.

Verify

- Build the lab (`dotnet run --project examples/ExtensibilityLabExample -- build`) and open `/metadata-demo/index.md` in the output — its front-matter block carries both `git_last_modified` and `reading_time_minutes`, the two keys `Derived` accumulated for that page.
- Render `/metadata-demo/` and confirm the `<LastModified />` component prints the date, proving a component read `Context["Derived"]`.

Related

- Reference: IMetadataEnricher and ReadingTimeEnricher

- How-to: Generate an llms.txt index — a built-in consumer of `Derived`

CHAPTER 11

Response Pipeline

Rewrite HTML attributes after parsing

Guides Implement IHtmlResponseRewriter to mutate already-parsed HTML — lowercase anchors, normalize hrefs, stamp rel=noopener — sharing the document parse with every other rewriter.

To rewrite anchors, inject attributes, normalize URLs, or strip sentinels in already-rendered HTML, implement `IHtmlResponseRewriter`. Every rewriter shares one AngleSharp parse against the same `IDocument`. For non-HTML response types (JSON, plain text) or work that needs the final byte stream, use Transform the response body on every page instead.

The recipe references `examples/ExtensibilityLabExample/AnchorLowercaseRewriter.cs`, which exercises both phases of the contract against a bare `AddPennington` host.

Before you begin

- An existing Pennington site rendering HTML pages (see Create your first Pennington site if not).
- A clear sense of which phase fits the edit: a non-HTML token (something not valid HTML structure, like `<xref:uid>` or a sentinel comment) belongs in `PreParseAsync`; anything queryable by selectors belongs in `ApplyAsync`.

Write the rewriter

Implement Pennington.Infrastructure.IHtmlResponseRewriter as a sealed class. Three rules carry the page:

- `ShouldApply` runs per-response; return `false` to skip both phases when the content-type, path, or headers mean there is nothing to do. The example narrows to `text/html` responses so non-HTML endpoints (search index JSON, llms.txt) bypass the rewriter entirely.
- `PreParseAsync` receives the raw HTML string and returns the string to parse. Use it only when the target construct is not valid HTML structure — raw `<xref:uid>` tags are the canonical shipped example. Return the input unchanged when there is nothing to do.
- `ApplyAsync` receives the already-parsed `IDocument` shared by every rewriter — query with `QuerySelectorAll`, mutate attributes and text, and return. Do not re-serialize or reparse.

CSHARP

```
namespace ExtensibilityLabExample;

using AngleSharp.Dom;
using AngleSharp.Html.Dom;
using Microsoft.AspNetCore.Http;
using Pennington.Infrastructure;

/// <summary>
/// Implements <see cref="IHtmlResponseRewriter"/> and demonstrates both
/// halves of the contract:
/// <list type="bullet">
/// <item><description><see cref="PreParseAsync"/> runs a cheap string
///   replace over the raw HTML before AngleSharp parses it. We use it to
///   strip the <c>&lt;!--LOWERCASE-SENTINEL--&gt;</c> comment — the kind
///   of pre-parse cleanup a real rewriter does for non-HTML tokens like
///   <c>&lt;xref:uid&gt;</c>.</description></item>
/// <item><description><see cref="ApplyAsync"/> walks the parsed document
///   and lowercases the text content of every <c>&lt;a&gt;</c> tag
///   marked <c>data-lowercase</c>.</description></item>
/// </list>
/// <para>
/// <see cref="Order"/> is 500 — after the shipped xref (10), locale (20),
/// and base-URL (30) rewriters so our pass sees already-resolved hrefs.
/// </para>
/// <para>
/// Backs how-to 2.3.50 <c>/how-to/extensibility/html-rewriter</c>.
/// </para>
/// </summary>
public sealed class AnchorLowercaseRewriter : IHtmlResponseRewriter
{
    public int Order => 500;

    public bool ShouldApply(HttpContext context)
    {
        var contentType = context.Response.ContentType;
        return contentType is not null
               && contentType.StartsWith("text/html", StringComparison.OrdinalIgnoreCase);
    }

    /// <summary>
    /// Pre-parse pass. Strip the sentinel comment so it is gone before
    /// AngleSharp runs. A string replace is the right tool when the
    /// target construct is not valid HTML structure (raw <c>&lt;xref&gt;</c>
    /// tags are the canonical example shipped with Pennington).
    /// </summary>
    public Task<string> PreParseAsync(string html, HttpContext context)
    {
        if (!html.Contains("<!--LOWERCASE-SENTINEL-->", StringComparison.Ordinal))
        {
            return Task.FromResult(html);
        }
```

```
    }      /// <summary>    /// DOM pass. Walk the parsed document, find every <c>&lt;a&gt;
</c>    /// with <c>data-lowercase</c>, lowercase its text content.    /// </summary>
public Task ApplyAsync(IDocument document, HttpContext context)    {        foreach (var
element in document.QuerySelectorAll("a[data-lowercase]"))        {            if (element
is not IHtmlAnchorElement anchor)            {                continue;            }
if (string.IsNullOrEmpty(anchor.TextContent))            {                continue;
}             anchor.TextContent = anchor.TextContent.ToLowerInvariant();        }
return Task.CompletedTask;    }}
```

Pick an Order value

The shipped rewriters occupy `Order` values from 10 (xref resolution) through 60 (the last built-in transform); xref resolution, locale prefixing, and base-URL prefixing run in that relative order because each produces the link form the next one consumes. Pick above 60 to run after every shipped transform, below 10 to run before xref resolution, or between the built-ins only when that placement is deliberate. For the exact `Order` of each shipped rewriter, see Pennington.Infrastructure.IHtmlResponseRewriter. The example uses 500 so anchors are lowercased after every shipped transform has run.

Register the rewriter

Every registered `IHtmlResponseRewriter` is picked up and ordered by its `Order` value, so a single registration next to the host wiring is sufficient. Use the lifetime that matches your dependencies — `AddSingleton` for stateless rewriters, `AddTransient` (or `AddFileWatched`) when the rewriter captures file-watched state.

CSHARP

```
builder.Services.AddSingleton<IHtmlResponseRewriter, AnchorLowercaseRewriter>();
```

Configure the shipped word-break rewriter

One shipped rewriter you configure rather than implement is the word-break rewriter. `AddWordBreak` turns it on; it inserts `<wbr>` break opportunities into long identifiers so dotted namespaces and PascalCase names wrap inside narrow columns instead of overflowing.

CSHARP

```
builder.Services.AddWordBreak(options =>
{
    options.CssSelector = "h1, h2, h3, h4, h5, h6, span, .text-break";
    options.MinimumCharacters = 20;
});
```

A heading like `Pennington.Infrastructure.WordBreakOptions` then renders with breaks after each dot and before each interior case boundary:

Before:

HTML

```
<h3>Pennington.Infrastructure.WordBreakOptions</h3>
```

After:

HTML

```
<h3>Pennington.<wbr>Infrastructure.<wbr>WordBreakOptions</h3>
```

For every option and its default, see Pennington.Infrastructure.WordBreakOptions.

Result

Anchors marked `data-lowercase` have their text content lowercased, and the sentinel comment is gone from view-source.

Before:

HTML

```
<!--LOWERCASE-SENTINEL-->
<a data-lowercase href="/docs/">Read the DOCS</a>
<a data-lowercase href="/blog/">Latest POSTS</a>
```

After:

HTML

```
<a data-lowercase href="/docs/">read the docs</a>
<a data-lowercase href="/blog/">latest posts</a>
```

Anchors without `data-lowercase` and non-HTML responses pass through unchanged.

Verify

- Run `dotnet run --project examples/ExtensibilityLabExample` and visit `/lowercase-demo/`. Every `<a data-lowercase>` anchor text is lowercase in the rendered HTML and `<!--LOWERCASE-SENTINEL-->` is absent from view-source.
- Static build: `dotnet run --project examples/ExtensibilityLabExample -- build output` — grep `output/lowercase-demo/index.html` to confirm the rewriter also runs during publish.

Related

- Reference: Response processing interfaces
- Reference: WordBreakOptions — the shipped word-break rewriter's configuration
- Background: The response-processing pipeline
- Related how-to: Write a response processor

Transform the response body on every page

Guides Implement IResponseProcessor to rewrite the final response body as a string — inject HTML before , log an outgoing payload, or append a non-HTML footer.

To transform the final response body on every rendered page, implement `IResponseProcessor`. The processor receives the full body as a string and returns the replacement — use it to insert a pre-serialized HTML fragment before `</body>`, log an outgoing payload, or append a non-HTML footer. When the work is DOM-shaped (anchor rewrites, attribute additions, element injection at a CSS selector), implement `IHtmlResponseRewriter` instead so every rewriter shares one AngleSharp parse. See Rewrite HTML attributes after parsing.

The recipe references `examples/ExtensibilityLabExample/FeedbackWidgetProcessor.cs`, which injects a "Was this helpful?" aside before `</body>` against a bare `AddPennington` host.

Before you begin

- An existing Pennington site (see Create your first Pennington site if not).
- The response pipeline buffers the full response body before the processor runs. This is fine for HTML pages but unsuitable for large binary streams — gate those out in `ShouldProcess`.

Write the processor

Implement Pennington.Infrastructure.IResponseProcessor as a sealed class. Two rules carry the page:

- `ShouldProcess` runs before the body is buffered. Returning `false` skips body capture entirely, so this is where filtering by status code, content type, or request path belongs. The example accepts only 2xx HTML responses, letting static assets, JSON endpoints, and redirects pass through untouched.
- `ProcessAsync` receives the full captured body as a string and returns the replacement. The example locates the last `</body>` with `LastIndexOf` and inserts the widget HTML there, falling back to append-at-end when the tag is absent so content still reaches the browser.

CSHARP

```
namespace ExtensibilityLabExample;

using System.Text;
using Microsoft.AspNetCore.Http;
using Pennington.Infrastructure;

/// <summary>
/// Implements <see cref="IResponseProcessor"/>. Injects a
/// "Was this helpful?" footer before the closing <c>&lt;/body&gt;</c>
/// tag of every rendered HTML page.
/// <para>
/// Runs at <see cref="Order"/> 500 — after the xref/locale/base-URL HTML
/// rewriting processor (<c>HtmlResponseRewritingProcessor</c>) so the
/// injected HTML is not subject to any further pipeline passes in this
/// app, and well before the live-reload and diagnostic-overlay
/// processors at 1000+.
/// </para>
/// <para>
/// <see cref="ShouldProcess"/> gates on content type: text/html only,
/// and only for 2xx responses. Static assets and API JSON skip through.
/// </para>
/// <para>
/// Backs how-to 2.3.40 <c>/how-to/extensibility/response-processor</c>.
/// </para>
/// </summary>
public sealed class FeedbackWidgetProcessor : IResponseProcessor
{
    private const string WidgetHtml = """
        <aside class="feedback-widget" data-extensibility-lab="feedback-widget">
          <p><strong>Was this helpful?</strong>
            <button type="button" data-feedback="yes">Yes</button>
            <button type="button" data-feedback="no">No</button>
          </p>
        </aside>
        """;

    public int Order => 500;

    public bool ShouldProcess(HttpContext context)
    {
        if (context.Response.StatusCode is < 200 or >= 300)
        {
            return false;
        }

        var contentType = context.Response.ContentType;
        return contentType is not null
               && contentType.StartsWith("text/html", StringComparison.OrdinalIgnoreCase);
    }

    public Task<string> ProcessAsync(string responseBody, HttpContext context)
```

```
            return Task.FromResult(responseBody);        }          var closeBodyIndex =
responseBody.LastIndexOf("</body>", StringComparison.OrdinalIgnoreCase);        if
(closeBodyIndex < 0)        {            // No </body> — append at end. Still visible, still
verifiable.            return Task.FromResult(responseBody + WidgetHtml);        }
var sb = new StringBuilder(responseBody.Length + WidgetHtml.Length);
sb.Append(responseBody, 0, closeBodyIndex);        sb.Append(WidgetHtml);
sb.Append(responseBody, closeBodyIndex, responseBody.Length - closeBodyIndex);        return
Task.FromResult(sb.ToString());    }}
```

Pick an Order value

Slot into the `Order` sequence so the processor sees the HTML state it expects. Anything below `10` would see un-resolved `<xref:...>` placeholders that `HtmlResponseRewritingProcessor` expands. The example uses `500` so the widget is inserted after every built-in pass has run. For the full table of shipped `Order` values, see Pennington.Infrastructure.IResponseProcessor.

Register the processor

Every registered `IResponseProcessor` is picked up and ordered by its `Order` value, so a single registration is the entire wiring step. Use the lifetime that matches your dependencies — `AddSingleton` for stateless processors, `AddTransient` (or `AddFileWatched`) when the processor captures file-watched state.

CSHARP

```
builder.Services.AddSingleton<IResponseProcessor, FeedbackWidgetProcessor>();
```

Result

Every `text/html` response carries the widget aside immediately before its closing `</body>` tag:

HTML

```
<aside class="feedback-widget" data-extensibility-lab="feedback-widget">
      <p><strong>Was this helpful?</strong>
        <button type="button" data-feedback="yes">Yes</button>
        <button type="button" data-feedback="no">No</button>
      </p>
    </aside>
  </body>
</html>
```

Non-HTML endpoints (`/styles.css`, `/sitemap.xml`) are unmodified because `ShouldProcess` returns `false` for them.

Verify

- Run `dotnet run --project examples/ExtensibilityLabExample` and visit `/`. The rendered HTML contains `<aside class="feedback-widget" data-extensibility-lab="feedback-widget">` immediately before `</body>`; fetch `/styles.css` and the aside is absent.
- Static build: `dotnet run --project examples/ExtensibilityLabExample -- build output` — grep `output/index.html` for `data-extensibility-lab="feedback-widget"` to confirm the processor runs during publish as well as dev.

Related

- Reference: Response processing interfaces
- Background: The response-processing pipeline
- Related how-to: Write an HTML rewriter

Customize the DocSite chrome through DocSiteOptions

Guides Use DocSiteOptions to inject head content, append CSS, replace the header/footer HTML, and route extra @page components without forking the template.

To replace the bundled DocSite header or footer (or inject head tags, append CSS, route additional `@page` components) without forking the template, populate the four extension points — the *slot seams* — on `DocSiteOptions`. The bundled layout, content pipeline, SPA navigation, and MonorailCSS wiring keep working. To rearrange the layout shell fundamentally, read What the DocSite and BlogSite templates wire for you before deciding whether `AddDocSite` is still the right starting point.

Before you begin

- An existing Pennington site wired through `AddDocSite(...)` (see Scaffold a documentation site with DocSite if not).
- All edits go in the `DocSiteOptions` factory passed to `AddDocSite`, not the DocSite source.
- These extension points are set at host-build time — changes take effect on the next `dotnet run`, whose source watch reloads them.

For a working setup, see `examples/DocSiteChromeOverridesExample`. `SiteChromeOverrides.cs` returns a populated `DocSiteOptions` exercising all four extension points, `Components/ExtraHeadFragment.razor` backs the head-slot fragment, and `Components/ExtraPage.razor` is the routed `@page` component showing that `AdditionalRoutingAssemblies` widened the router. `Program.cs` runs the DocSite end-to-end against those overrides.

Build the populated options

All the code for this recipe lives in one factory method, so the four extension points sit together on a single record initializer. The example sets `SiteTitle` and `SiteDescription` alongside the override properties — plus a brand `ColorScheme` (covered below) — matching the options produced by `AddDocSite(() => SiteChromeOverrides.BuildDocSiteOptions())`.

CSHARP

```
public static DocSiteOptions BuildDocSiteOptions() => new()
{
    SiteTitle = "DocSite Chrome Overrides",
    SiteDescription = "Running DocSite that exercises every override seam on
DocSiteOptions.",
    // A curated ColorTheme repigments the whole site from one hue. Orchid's neutral
    // `base` is auto-selected to coordinate with its magenta brand (mauve grays, not a
    // generic neutral); the home page renders the palette via
Components/BrandPalette.razor.
    ColorScheme = ColorTheme.Orchid,
    SyntaxTheme = ColorTheme.Orchid.SyntaxTheme,
    HeaderContent = """<span class="chrome-header" data-chrome-overrides="docsite-
header">Chrome Overrides</span>""",
    FooterContent = """<span class="chrome-footer" data-chrome-overrides="docsite-footer">
(c) 2026 Pennington</span>""",
    AdditionalHtmlHeadContent = BuildHtmlHeadContent(),
    ExtraStyles = BuildExtraStyles(),
    AdditionalRoutingAssemblies = BuildAdditionalRoutingAssemblies(),
    Areas =
    [
        new ContentArea("Guides", "guides"),
    ],
};
```

Inject tags into `<head>` via `AdditionalHtmlHeadContent`

`AdditionalHtmlHeadContent` is a raw HTML string rendered inside every page's `<head>`, making it the right place for meta tags, preconnect hints, analytics snippets, and font `<link>` elements that MonorailCSS does not know about. To author the fragment as a Razor component instead, render it with `ToHtmlString()` once at startup and pass the resulting string — the example pairs `SiteChromeOverrides.BuildHtmlHeadContent` with `Components/ExtraHeadFragment.razor` so both approaches sit side by side.

Use this string for static site-wide markup you do not want to write a class for; reach for an IHeadContributor instead when the tag must deduplicate against another writer, order against site or page defaults, or be computed per-page. Both routes flow through the same head reconciler, so either way the tags get a `data-head` stamp and survive SPA navigation.

CSHARP

```
=> """
<meta name="x-chrome-overrides-head" content="extra-head-fragment">
<link rel="preconnect" href="https://example.com">
"""
```

Prepend rules to the generated stylesheet via `ExtraStyles`

`ExtraStyles` is a CSS string emitted above the MonorailCSS-generated rules inside `/styles.css`, making it the right home for `@font-face` declarations, custom-property overrides, and any selector the utility-class scanner will not discover on its own. Keep this string small — anything expressible as MonorailCSS utilities in Razor markup gets picked up automatically by the `MonorailCss.Discovery` pipeline.

CSHARP

```
=> """
.chrome-header { font-weight: 600; color: var(--color-primary-700); }
.chrome-footer { font-size: 0.875rem; color: var(--color-base-500); }
"""
```

Replace the site-title link and footer with the content slots

`HeaderContent` owns the entire header brand area: the default document icon and the `<a href="/">SiteTitle</a>` link both step aside, so you control that region outright while the rest of the header chrome (search button, theme toggle, repo link) keeps rendering around it. `FooterContent` is what the layout drops into the footer region. Both accept either a raw HTML string or a `RenderFragment` — assign a string for inline markup, or point them at a `RenderFragment` (for example a static fragment defined in a `.razor`) for a component-authored header, no `AdditionalRoutingAssemblies` entry required.

CSHARP

```
var options = new DocSiteOptions
{
    HeaderContent = """<span class="chrome-header" data-chrome-overrides="docsite-
header">Chrome Overrides</span>""",
    FooterContent = """<span class="chrome-footer" data-chrome-overrides="docsite-footer">
(c) 2026 Pennington</span>""",
    // ...
};
```

The `data-chrome-overrides` attributes are not required by `DocSiteOptions` — they are markers that make the swapped-in chrome easy to spot in page source, matching what the example renders and what the Result describes below.

Route your own `@page` components via `AdditionalRoutingAssemblies`

The DocSite shell only discovers `@page` directives in its own assembly by default; adding the host assembly to `AdditionalRoutingAssemblies` makes any `@page "/route"` component in that assembly routable alongside the bundled pages. The example returns `[typeof(SiteChromeOverrides).Assembly]` so a Razor component like `ExtraPage.razor` sitting next to `Program.cs` gets picked up without any additional DI wiring.

CSHARP

```
=>
[typeof(SiteChromeOverrides).Assembly]
```

Recolor the chrome with `ColorScheme`

The four points above inject markup; `ColorScheme` repaints it. It is the other `DocSiteOptions` property this example sets — assigned to `ColorTheme.Orchid`, one of the curated catalog schemes. A `ColorTheme` grows the whole palette from a single hue: the `primary` and `accent` brand roles algorithmically, and the neutral `base` ramp by auto-selecting the stock MonorailCSS neutral whose undertone sits nearest that hue. Orchid's magenta lands on `mauve`, so the surface grays carry a faint mauve tint instead of a generic gray. The example forwards the theme's coordinated `SyntaxTheme` too.

CSHARP

```
ColorScheme = ColorTheme.Orchid,
SyntaxTheme = ColorTheme.Orchid.SyntaxTheme,
```

The home page renders the resulting palette as swatches through a small Mdazor component, `Components/BrandPalette.razor`, registered in `Program.cs` with `AddMdazorComponent<BrandPalette>()`. See Recolor the site for the full range of color-scheme options and Author a custom Razor component for markdown for authoring Mdazor components.

Register the implementation

`AddDocSite` takes a `Func<DocSiteOptions>` factory, so the most direct wiring is to pass the helper as a method reference and keep the host file short. The example's `Program.cs` runs this exact shape end-to-end, with the one extra `AddMdazorComponent<BrandPalette>()` line that registers the home-page palette component.

CSHARP

```
using DocSiteChromeOverridesExample;
using DocSiteChromeOverridesExample.Components;
using Mdazor;
using Pennington.DocSite;

var builder = WebApplication.CreateBuilder(args);

// Live wiring referenced from step 6 of
// how-to/extensibility/override-docsite-components. The factory is a
// method reference, so the helper in SiteChromeOverrides.cs owns the
// full DocSiteOptions shape and Program.cs stays short.
builder.Services.AddDocSite(SiteChromeOverrides.BuildDocSiteOptions);

// The home page (Content/index.md) renders <BrandPalette /> to show off the
// active ColorScheme; register the component so Mdazor resolves the tag.
builder.Services.AddMdazorComponent<BrandPalette>();

var app = builder.Build();

app.UseDocSite();

await app.RunDocSiteAsync(args);
```

Result

The chrome on every page is replaced by the configured fragments, one outcome per extension point:

- **Header and footer.** The header brand area reads "Chrome Overrides" on the left, rendered as `<span class="chrome-header" data-chrome-overrides="docsite-header">` in place of the default icon and `<a href="/">…</a>` link, with the rest of the header chrome (search, theme toggle, repo link) intact; the footer carries the matching `data-chrome-overrides="docsite-footer"` copyright span.
- **Head content.** Every `<head>` gains the `<meta name="x-chrome-overrides-head">` tag and the `https://example.com` preconnect.
- **Styles.** `/styles.css` begins with the prepended `.chrome-header` / `.chrome-footer` rules, above the generated MonorailCSS utilities.
- **Routing.** Any `@page "/route"` component in the host assembly (for example `/extra`) routes alongside the bundled DocSite pages.

Verify

- Run `dotnet run` and view page source on `/` — expect the `<meta name="x-chrome-overrides-head">` tag inside `<head>`, your `HeaderContent` and `FooterContent` markup in the layout, and the `.chrome-header` rule inside `/styles.css`.
- Navigate to a route defined by a Razor component in your app assembly (for example `/extra`) and confirm it renders. A 404 here means `AdditionalRoutingAssemblies` is not including the right assembly.

- Visit `/` and confirm the `<BrandPalette />` swatches render. The `base` ramp reads mauve-tinted next to the stock `neutral` ramp — proof the `ColorScheme` auto-picked a coordinating neutral for the brand hue.
- Run `dotnet run -- build output` and search `output/index.html` for your head fragment and `output/styles.css` for your `ExtraStyles` rules to confirm the overrides survive publish.

Related

- Reference: Pennington.DocSite.DocSiteOptions — the full set of properties, including `ConfigurePennington`, `CustomCssFrameworkSettings`, and every other override point beyond the four covered here.
- How-to: Add tags to the document head — the typed alternative to `AdditionalHtmlHeadContent` when a head tag must dedup, order, or compute per-page.
- How-to: Serve docs and a blog from separate content roots — register extra markdown sources through `DocSiteOptions.ConfigurePennington`.
- How-to: Source content from outside the markdown pipeline — register a custom `IContentService` alongside DocSite's own.
- Background: What the DocSite and BlogSite templates wire for you — when forking DocSite or dropping to bare `AddPennington` becomes the right move.
- Background: SPA navigation through region swaps — `data-spa-region` semantics for SPA-aware layout components.

Render a Razor component as a page on a bare host

Guides Use HtmlRenderer.RenderComponentAsync inside a MapGet to make a Razor component the entire response body, no DocSite layout pipeline required.

To render a Razor component as the whole response body for a custom route on a bare `AddPennington` host, render it through Blazor's server-side `HtmlRenderer` from inside a `MapGet`. The component owns the document — `<html>`, `<head>`, `<body>` — so the response is a complete HTML page without DocSite or BlogSite layout machinery. Use this pattern when a custom `IContentService` discovers per-record routes (`/instructors/{slug}/`, `/status/{slug}/`) and the rendered output is too complex for inline HTML strings.

Before you begin

- A working Pennington site on bare `AddPennington` (see Create your first Pennington site if not).
- A reference to `Microsoft.AspNetCore.Components.Web` — already transitive through `Pennington`.
- Familiarity with `IContentService` for publishing the routes you'll render against (Source content from outside the markdown pipeline).

A working reference: `examples/BareHostRazorPageExample` — one Razor component plus a single `MapGet` that renders it.

Author the page component

Write a Razor component whose `[Parameter]` properties are everything the page needs — there is no ambient `HttpContext`, layout, or cascading state from a parent. The component renders the entire document so it includes `<!DOCTYPE html>` and the `<link rel="stylesheet" href="/styles.css">` tag for MonorailCSS output.

RAZOR

```
@* StatusPage — a Razor component used as the entire page body for routes like
   /status/{slug}. Program.cs renders it through HtmlRenderer.RenderComponentAsync
   inside a MapGet, so the component owns the whole document including <html>,
   <head>, and <body>. *@
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <title>@Title</title>
    <link rel="stylesheet" href="/styles.css" />
</head>
<body class="bg-base-50 text-base-900 dark:bg-base-950 dark:text-base-50">
    <main class="mx-auto max-w-2xl px-6 py-12">
        <header class="mb-8">
            <p class="text-xs font-semibold uppercase tracking-wide text-accent-
500">@Slug</p>
            <h1 class="mt-1 font-display text-3xl font-bold">@Title</h1>
        </header>

        <p class="text-base-700 dark:text-base-300">@Summary</p>

        <dl class="mt-8 grid grid-cols-1 gap-x-6 gap-y-3 sm:grid-cols-[10rem_1fr]">
            @foreach (var item in Facts)
            {
                <dt class="text-xs font-semibold uppercase tracking-wide text-base-500
dark:text-base-400">@item.Key</dt>
                <dd class="text-sm text-base-700 dark:text-base-300">@item.Value</dd>
            }
        </dl>
    </main>
</body>
</html>

@code {
    [Parameter, EditorRequired] public string Slug { get; set; } = string.Empty;
    [Parameter, EditorRequired] public string Title { get; set; } = string.Empty;
    [Parameter, EditorRequired] public string Summary { get; set; } = string.Empty;
    [Parameter] public IReadOnlyList<KeyValuePair<string, string>> Facts { get; set; } = [];
}
```

Register the Blazor renderer services

`HtmlRenderer` needs Blazor's component services and an `IHttpContextAccessor` so cascading values can resolve. Register both alongside the `AddPennington` and `AddMonorailCss` hosts:

CSHARP

```
builder.Services.AddRazorComponents();
builder.Services.AddHttpContextAccessor();
```

`AddRazorComponents` registers `HtmlRenderer` and its dispatcher; `AddHttpContextAccessor` lets a rendered component resolve cascading values. There is no `MapRazorComponents`, no `App.razor`, and no `_Host` page — the bare host never starts the Blazor router. Components reach the response only through the `MapGet` below.

Render the component inside a `MapGet`

The route handler turns a slug into the component's `[Parameter]` values and hands them to a render helper. A missing record returns null parameters, which the helper turns into a 404:

CSHARP

```
public static async Task<IResult> RenderRazorPageAsync<TComponent>(
    HtmlRenderer renderer,
    IDictionary<string, object?>? parameters)
    where TComponent : IComponent
{
    if (parameters is null)
    {
        return Results.NotFound();
    }

    var html = await renderer.Dispatcher.InvokeAsync(async () =>
    {
        var output = await renderer.RenderComponentAsync<TComponent>(
            ParameterView.FromDictionary(parameters));
        return output.ToHtmlString();
    });
    return Results.Content(html, "text/html");
}
```

`RenderRazorPageAsync<TComponent>` is the only Blazor-specific code the host needs: it dispatches the render onto the renderer's dispatcher, materializes the output with `ToHtmlString`, and hands the complete HTML string to `Results.Content`. Reuse it for any other component-as-page route. The route wiring itself is a plain minimal-API endpoint:

CSHARP

```
app.MapGet("/status/{slug}/", (string slug, StatusPagesContentService statuses, HtmlRenderer
renderer)
    => BareHostRenderer.RenderRazorPageAsync<StatusPage>(renderer, statuses.TryGet(slug) is
{ } entry
        ? new Dictionary<string, object?>
        {
            [nameof(StatusPage.Slug)] = entry.Slug,
            [nameof(StatusPage.Title)] = entry.Title,
            [nameof(StatusPage.Summary)] = entry.Summary,
            [nameof(StatusPage.Facts)] = entry.Facts,
        }
        : null));
```

Why not a Blazor `@page`?

A routed `@page` component needs the Blazor router, an `App.razor`, and `MapRazorComponents` — the machinery Serve markdown through Blazor Pages stands up. A bare `AddPennington` host runs none of that, so a `@page` directive would never be routed. Rendering through `HtmlRenderer` inside a `MapGet` keeps the host minimal: the component is a render target, not a routed endpoint, and your `IContentService` owns route discovery.

Publish the routes through `IContentService`

A custom `IContentService` yields one `EndpointSource` per route so the build crawler discovers each URL and fetches it through the live pipeline — your `MapGet` produces the HTML the same way at build time as at request time. See Source content from outside the markdown pipeline for the per-record discovery pattern, including a worked `EndpointSource` example.

Verify

- Run `dotnet run --project examples/BareHostRazorPageExample` and open `/status/intro/` and `/status/verify/` at the URL the console prints (the `Now listening on:` line). Each renders the `StatusPage` component as a full HTML page styled by `/styles.css`.
- Confirm the static build picks up both routes: `dotnet run --project examples/BareHostRazorPageExample -- build` writes `output/status/intro/index.html` and `output/status/verify/index.html`.

Related

- How-to: Source content from outside the file system
- Background: The content pipeline and union types

Add tags to the document head

Guides Implement IHeadContributor to emit title, meta, link, or script tags into with central deduplication, band-based ordering, and automatic SPA-navigation survival.

To add a head tag that deduplicates against other writers, orders predictably against site and page defaults, and survives SPA navigation, implement `IHeadContributor`. Reach for a contributor when the tag is shared across pages — emitted on many pages, or competing with another writer for the same slot. For one-off, page-local markup there are two lighter options instead: a Razor `<HeadContent>` block (see Keep authoring in Razor) or, on a DocSite, the `AdditionalHtmlHeadContent` string (see Customize the DocSite chrome). For background on why the head funnels through one extension point that every writer goes through, see The head subsystem.

Before you begin

- An existing Pennington site. `AddPennington` (and therefore `AddDocSite` / `AddBlogSite`) already registers the head composition rewriter, so contributors activate as soon as you register one.
- A sense of which slot the tag occupies: does it appear at most once (a `<title>`, a canonical link, one `og:image`), or can it repeat (alternates, JSON-LD)? That choice drives whether you add it with a dedup key or as a repeatable.

Write the contributor

Implement `IHeadContributor` as a sealed class. The interface is three members — `Order`, `ShouldContribute`, and `ContributeAsync` (see IHeadContributor for the member catalog).

Push tags through the `HeadBuilder` handed to `ContributeAsync`. Its helpers cover the common cases — `Title`, `Meta` (name/content), `Property` (OpenGraph), and `Link` (rel/href) each add under a dedup key, while `AddRepeatable` appends a tag that may occur more than once (see HeadBuilder for the full surface).

A minimal contributor that stamps a site-wide `generator` meta tag on every page:

CSHARP

```csharp
using Pennington.Head;

internal sealed class GeneratorMetaHeadContributor : IHeadContributor
{
    public int Order => HeadOrder.Site;

    public bool ShouldContribute(HeadContext context) => true;

    public Task ContributeAsync(HeadContext context, HeadBuilder head)
    {
        head.Meta("generator", "Pennington");
        return Task.CompletedTask;
    }
}
```

The page authored no generator tag, so before composition its `<head>` carries none:

HTML

```
<head>
  <title>Getting started</title>
  <!-- no generator meta -->
</head>
```

After composition the rewriter appends the contributed tag and stamps it with `data-head` — the value is the dedup key, here `meta:name:generator`:

HTML

```
<head>
  <title data-head="title">Getting started</title>
  <meta name="generator" content="Pennington" data-head="meta:name:generator">
</head>
```

That `data-head` stamp is what later same-key contributors dedup against and what the SPA engine carries across a soft navigation.

Pick an order band

`Order` is chosen from the `HeadOrder` bands, not a raw integer. Contributors run lowest-first, and on a dedup-key collision the lowest order wins — so a tag in a lower band overrides the same key in a higher one.

Use `Page` (40) for tags computed from the current page that should beat site defaults, `Site` (60) for site-wide defaults, and `Discovery` (80) for structured-data and verification payloads. The generator meta above sits at `Site` because it is a constant site default with no page-level override. See HeadOrder for the complete band list with values.

Register it

Register with `AddHeadContributor<T>()` after the host wiring. Registration is transient, which is what you want for a contributor that reads file-watched state such as the content registry.

CSHARP

```
builder.Services.AddDocSite(() => new DocSiteOptions { /* … */ });
builder.Services.AddHeadContributor<GeneratorMetaHeadContributor>();
```

Options

Emit a repeatable tag

When a tag can appear more than once — an `hreflang` alternate, a JSON-LD block, an RSS alternate — add it with `AddRepeatable` and no key, building the `LinkTag` / `ScriptTag` directly. The shipped RSS-alternate contributor shows the shape, including extra attributes in emission order:

CSHARP

```
public Task ContributeAsync(HeadContext context, HeadBuilder head)
{
    head.AddRepeatable(new HeadTag(new LinkTag("alternate", "/rss.xml")
    {
        Attributes = [new("type", "application/rss+xml"), new("title",
_pennington.SiteTitle)],
    }));
    return Task.CompletedTask;
}
```

Gate with `ShouldContribute`

Return `false` from `ShouldContribute` to skip a contributor entirely for a request. This is the cheap precondition check — a missing config value, a feature flag, a content type that should not carry the tag. The canonical-link contributor self-gates on whether a base URL is configured, so registering it unconditionally is harmless:

CSHARP

```
public bool ShouldContribute(HeadContext context) =>
!string.IsNullOrEmpty(_options.CanonicalBaseUrl);
```

Read the resolved page record

`HeadContext` carries the request and the content record resolved for it, so a contributor can compute tags from the page's front matter. Its members are `HttpContext`, `FullPath`, and the nullable `Record` (see HeadContext).

`Record` is `null` on endpoint and 404 pages, so guard it. `FullPath` is the request path with the locale segment reattached — the same key the content registry and structured-data join on.

Override a built-in tag

To replace a tag a built-in contributor emits, add the same key from a lower band. A page-level OpenGraph contributor at `Page` overrides the `Site`-band default for `meta:prop:og:image` purely through the lowest-order-wins rule — neither contributor references the other. The site default's own dedup makes it step aside.

Keep authoring in Razor

A `<HeadContent>` or `<PageTitle>` block on a page still works. The reconciler pulls whatever `HeadOutlet` rendered into the same model, stamps it, and dedups it against contributor output — with the page winning on a key collision.

There are three routes to "add a head tag", and the decision rule is which scope owns it: a contributor for anything shared across pages or competing for a slot (it dedups, orders, and survives navigation); a Razor `<HeadContent>` block for one-off markup local to a single page; and, on a DocSite,

AdditionalHtmlHeadContent for a raw site-wide string (analytics snippets, preconnect hints) you do not want to write a class for. The string route runs through the same head reconciler, so its tags also get a `data-head` stamp.

Verify

- Run `dotnet run` and view-source on any page. The contributed tag is present and carries a `data-head` attribute (the stamp that drives dedup and SPA-navigation survival) — for the generator example above, expect `<meta name="generator" content="Pennington" data-head="meta:name:generator">`.
- Navigate between pages without a full reload and confirm the tag is still present — the generic `[data-head]` sweep carries it across the region swap with no per-tag wiring.
- Static build: `dotnet run -- build output`, then grep a published page for the stamped tag to confirm it ships at publish time too — `grep 'data-head="meta:name:generator"' output/index.html`.

Related

- Background: The head subsystem — the model, ordering bands, and `data-head` invariant this how-to builds on.
- Related how-to: Rewrite HTML attributes after parsing — for whole-document edits outside the head.
- Reference: DocSiteOptions and host extensions

CHAPTER 12

Deployment

Build a static site

Guides Produce a deployable `output/` directory by running the same app in build mode and reading the `BuildReport` for failures.

To turn a working Pennington site into a folder of static HTML for a static host, run the app in build mode. For why the same `Program.cs` works in both dev and build, see Dev mode and build mode share one code path; for platform-specific upload steps, see Deploy to GitHub Pages; for sub-path sites, see Host under a sub-path (base URL).

Before you begin

- A working Pennington site that serves under `dotnet run` (see Create your first Pennington site if not).
- The host composes `RunOrBuildAsync` directly or via `RunDocSiteAsync` / `RunBlogSiteAsync` (most apps do — confirm `Program.cs` ends with one of those calls).
- A writable local directory — the build deletes and re-creates `output/` by default.

Steps

Invoke the build verb

Pass `build` as the first argument to `dotnet run`. The argument is parsed into `OutputOptions` via `FromArgs`; without it, the app starts as a dev server instead. Three argument shapes are supported:

BASH

```
# defaults: BaseUrl = "/", OutputDirectory = "output"
dotnet run -- build

# positional: base URL, then output dir
dotnet run -- build /my-site dist

# named flags (order-independent, preferred for scripts)
dotnet run -- build --base-url=/my-site --output=dist
```

See CLI and build arguments for the full grammar.

Read the `BuildReport` printed to stdout

When the crawl finishes, `RunOrBuildAsync` writes a human-readable report and exits with a non-zero code when the build failed; see Pennington.Generation.BuildReport for the fields and the exact failure conditions. Fix the routes it lists before deploying.

For custom CI presentation (a GitHub Actions summary, a Slack message), use `BuildHost.PrintBuildReport` in `examples/SubPathDeployableExample/BuildHost.cs` as a starting point.

Verify

- `dotnet run -- build` exits `0` and the stdout report opens with `Build Complete — N pages in Xs` followed by `N pages generated`, with no `ERRORS` or `WARNINGS` section. A broken internal link is reported as a warning, so an empty `WARNINGS` section means every internal link resolved.
- `output/index.html` and `output/404.html` both exist — open `index.html` in a browser to spot-check the rendered output.

Related

- Reference: CLI and build arguments
- Reference: Build report fields
- Background: Dev mode and build mode share one code path

Deploy to GitHub Pages

Guides Ship a Pennington site to GitHub Pages with a ready-to-copy Actions workflow, base-URL injection, and the `.nojekyll` marker.

This guide covers deploying a working Pennington site committed to a GitHub repo, so Pages builds and deploys it automatically on every push to `main`. When the site still only runs under `dotnet run`, complete Build a static site first — the directory structure of `output/` is easier to automate once it's familiar.

Before you begin

- A Pennington site that builds locally with `dotnet run --project <your-project> -- build` (see Build a static site if not).
- The repo is pushed to GitHub and Pages is enabled under **Settings → Pages → Build and deployment → Source: GitHub Actions**.
- The site will serve under a repository sub-path like `https://<user>.github.io/<repo>/`. Root-domain deployments are called out in Step 5.

For a working setup, see examples/SubPathDeployableExample — the `.github/workflows/deploy.yml` and `BuildHost` helper are the relevant siblings.

Steps

Enable GitHub Pages with the Actions source

In the repo settings, switch **Pages → Build and deployment → Source** to **GitHub Actions** so the deploy workflow is authorized to publish. Also confirm the three workflow permissions the deploy action needs — `contents: read`, `pages: write`, `id-token: write` — are not blocked at the organization level. The workflow declares them explicitly, but an org-wide deny overrides that.

Add the deploy workflow

Commit the YAML below to `.github/workflows/deploy.yml` at the repo root. It pins `actions/setup-dotnet@v4` to .NET 10, derives the base URL from `${{ github.event.repository.name }}` so the same file works on forks and renames, runs `dotnet run -- build "$BASE_URL"`, writes `.nojekyll`, and hands `output/` to `actions/upload-pages-artifact@v3` and `actions/deploy-pages@v4`.

YAML

```
# Canonical GitHub Pages workflow for a Pennington static site.
#
# Assumes the site is served under a repository sub-path — the typical
# project-Pages URL is `https://<user>.github.io/<repo>/`, which requires
# a matching `baseUrl` argument at build time so internal anchors, CSS,
# JS, and data URLs all resolve under `/<repo>/`.
#
# The workflow:
#   1. Derives the base URL from `${{ github.event.repository.name }}` so
#      the same file works on any fork or renamed repo.
#   2. Runs `dotnet run --project … -- build /<repo>` to emit `output/`.
#   3. Drops a `.nojekyll` marker so GitHub Pages serves `_content/*`
#      folders verbatim (Jekyll would silently strip underscore paths).
#   4. Uploads `output/` as a Pages artifact and deploys it.
#
# If your site sits at an org root or a custom domain (served from `/`),
# set `BASE_URL` to an empty string. `build "$BASE_URL"` then passes an
# empty argument and the base-URL rewriter leaves internal links untouched.
name: Deploy to GitHub Pages

on:
  push:
    branches: [main]
  workflow_dispatch:

permissions:
  contents: read
  pages: write
  id-token: write

concurrency:
  group: pages
  cancel-in-progress: false

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup .NET
        uses: actions/setup-dotnet@v4
        with:
          dotnet-version: 10.0.x

      - name: Build static site
        env:
          BASE_URL: /${{ github.event.repository.name }}
        run: |
          dotnet run \
            --project examples/SubPathDeployableExample \
```

```
      - name: Disable Jekyll processing        run: touch output/.nojekyll        - name:
Upload Pages artifact        uses: actions/upload-pages-artifact@v3        with:
path: output    deploy:    needs: build    runs-on: ubuntu-latest    environment:      name:
github-pages      url: ${{ steps.deployment.outputs.page_url }}    steps:      - name:
Deploy to GitHub Pages        id: deployment        uses: actions/deploy-pages@v4
```

> **Note**
>
> The `touch output/.nojekyll` step is load-bearing: without it GitHub Pages runs the artifact through Jekyll, which strips any path starting with an underscore — including Pennington's `_content/` static-web-asset folder. The marker disables Jekyll so `_content/*` ships verbatim.

Point the `--project` path at your site

The template targets `examples/SubPathDeployableExample`; edit the `--project` argument and any `working-directory` references so the `dotnet run` step points at the correct csproj.

Match the build `baseUrl` to the Pages URL

Project Pages sites serve at `https://<user>.github.io/<repo>/`, so the workflow passes `/<repo>` as the first positional `build` argument and `BaseUrlHtmlRewriter` prefixes every internal `href`, `src`, and `action` on the way out. For sites at an org-level root (`https://<org>.github.io/`) or a custom apex domain, the site serves from `/`: set `BASE_URL` to an empty string so `build "$BASE_URL"` passes an empty argument and the rewriter leaves links untouched. The workflow's header comment marks the same two lines to change. Sub-path wiring is covered in Host under a sub-path (base URL).

Customize the exit semantics

`RunOrBuildAsync` already sets a non-zero exit code on errors, so the workflow above fails fast on broken pages. When you need stricter or more selective behavior — failing the main-branch build on broken xrefs while letting warnings pass on feature branches — skip the `RunOrBuildAsync` extension, run the generator yourself, and inspect the `BuildReport` before setting the exit code. The `BuildHost` helper in the example does exactly that:

CSHARP

```csharp
public static void PrintBuildReport(BuildReport report)
{
    report.WriteTo(Console.Out);
    if (report.HasErrors)
    {
        Environment.ExitCode = 1;
    }
}
```

`report.HasErrors` covers broken xrefs and failed pages; branch on `report.Diagnostics` for finer-grained rules. Call `BuildHost.RunOrBuildAsync` from `Program.cs` in place of the default extension to route the build through it.

Verify

- Push to `main`; the **Deploy to GitHub Pages** workflow runs the `build` and `deploy` jobs in sequence and turns green.
- Visit `https://<user>.github.io/<repo>/` — the landing page loads, navigation links resolve under `/<repo>/`, and view-source shows `<body data-base-url="/<repo>">` (the rewriter trims the trailing slash).
- Open the **build** job log — expect the `BuildReport` summary line with zero failed pages and zero broken links; any non-zero count fails the job.

Related

- Recipe: Build a static site — what `build [baseUrl] [outputDirectory]` produces before you automate it.
- Recipe: Host under a sub-path (base URL) — how `BaseUrlHtmlRewriter` handles the `/<repo>/` prefix for non-GitHub-Pages hosts.
- Recipe: Adapt the deploy workflow for other hosts — Azure Static Web Apps, Cloudflare Pages, and Netlify deltas against this workflow.
- Reference: CLI and build arguments — the `build [baseUrl] [outputDirectory]` surface this workflow drives.
- Reference: Build report fields — the `BuildReport` surface (`HasErrors`, `FailedPages`, `Diagnostics`) the CI step above checks.

Adapt the deploy workflow for other hosts

Guides Port the GitHub Pages recipe to Azure Static Web Apps, Cloudflare Pages, or Netlify by swapping four shared values and dropping in one host-specific config file.

With Deploy to GitHub Pages already in place and the `dotnet run -- build [baseUrl]` → `output/` → artifact pipeline understood, this page covers the deltas for Azure Static Web Apps, Cloudflare Pages, and Netlify. The material assumes the GitHub Pages recipe is the starting point.

Before you begin

- The canonical GitHub Pages workflow is committed and building cleanly.
- A deploy target account exists (SWA resource, Cloudflare Pages project, or Netlify site) and the repo is connected.
- The site serves either at the host's domain root (`baseUrl = "/"`) or under a known sub-path to pass as the first positional argument to `build`. See Host under a sub-path (base URL) for the rewriter behavior.

For a working setup, see examples/SubPathDeployableExample — the `.github/workflows/deploy.yml`, `staticwebapp.config.json`, and `netlify.toml` siblings each express the same handful of settings in their host's syntax.

Host deltas

Every host restates the same four settings — build command (`dotnet run --project <your-project> -- build "<base-url>"`), publish directory (`output/` , the default from `OutputOptions`), .NET SDK pin (`10.0.x` , matching `setup-dotnet@v4` in the GitHub Pages workflow), and base URL (`/` for apex domains, `/<path>` for sub-path hosting). The table below is the diff against the GitHub Pages workflow — it shows where each setting diverges per host. See CLI and build arguments for the `OutputOptions.FromArgs` grammar.

```
Concern GitHub Pages (canonical) Azure Static Web Apps Cloudflare Pages Netlify     Config file `.github/workflows/deploy.yml` `staticwebapp.config.json` + SWA's own build action Pages dashboard (no first-party config file) `netlify.toml`   Build command `dotnet run --project … -- build "$BASE_URL"` same (invoked via `Azure/static-web-apps-deploy@v1` `app_build_command`) same (set in dashboard → **Build command**) same (declared in `[build] command`)   Publish directory `output` (via `upload-pages-artifact@v3`) `output_location: "output"` on the SWA action **Build output directory:** `output` `publish = "output"`   .NET SDK pin `actions/setup-dotnet@v4` with `10.0.x` add `actions/setup-dotnet@v4` before the SWA action dashboard env: `DOTNET_VERSION = 10.0.x` `[build.environment] DOTNET_VERSION = "10.0.x"`   Base URL strategy derived from `${{ github.event.repository.name }}` `BASE_URL` env var, passed into `app_build_command` as `"${BASE_URL:-/}"`; apex by default `BASE_URL` env var, passed into the build command as `"${BASE_URL:-/}"`; apex by default `BASE_URL` env var with `/` default; override in dashboard per site   SPA / deep-link fallback `.nojekyll` marker + `404.html` `navigationFallback.rewrite: "/404.html"` (see Azure below) Cloudflare auto-serves `404.html` from build output `[[redirects]]` with `status = 404 → /404.html` (see Netlify below)   Cache headers for `/_content/*` GitHub Pages default (short TTL) `routes[]` entry, `Cache-Control: public, max-age=31536000, immutable` `_headers` file in `output/` (same directive) `[[headers]] for = "/_content/*"` (same directive)   `.nojekyll` needed? yes no no no
```

Azure Static Web Apps

Commit `staticwebapp.config.json` at the repo root; SWA reads it during deploy and applies routes, MIME overrides, nav fallback, and 404 handling. In the SWA workflow (`.github/workflows/azure-static-web-apps-<id>.yml` , generated by the Azure portal), set `app_build_command` to `dotnet run --project <your-project> -- build "${BASE_URL:-/}"` and `output_location` to `output` . The `${BASE_URL:-/}` expansion is what carries the sub-path through — a bare `-- build` always deploys at the apex regardless of the env var. Define `BASE_URL` as a workflow `env:` entry (or leave it unset for apex hosting). Everything else from the GitHub Pages workflow applies unchanged.

JSON

```
{
  "$schema": "https://json.schemastore.org/staticwebapp.config.json",
  "trailingSlash": "auto",
  "mimeTypes": {
    ".json": "application/json",
    ".xml": "application/xml",
    ".webmanifest": "application/manifest+json"
  },
  "routes": [
    {
      "route": "/sitemap.xml",
      "headers": {
        "Cache-Control": "public, max-age=3600"
      }
    },
    {
      "route": "/llms.txt",
      "headers": {
        "Cache-Control": "public, max-age=3600"
      }
    },
    {
      "route": "/_content/*",
      "headers": {
        "Cache-Control": "public, max-age=31536000, immutable"
      }
    }
  ],
  "navigationFallback": {
    "rewrite": "/404.html",
    "exclude": [
      "/_content/*",
      "/*.{css,js,json,png,jpg,jpeg,gif,svg,webp,ico,woff,woff2,ttf,xml,txt,webmanifest}"
    ]
  },
  "responseOverrides": {
    "404": {
      "rewrite": "/404.html"
    }
  },
  "globalHeaders": {
    "X-Content-Type-Options": "nosniff",
    "Referrer-Policy": "strict-origin-when-cross-origin"
  }
}
```

Netlify

Commit `netlify.toml` at the repo root; Netlify autodetects it and no dashboard build-setting changes are needed beyond linking the repo. `BASE_URL` defaults to `/` — override it in **Site configuration → Environment variables** for sub-path hosting. The `[[redirects]]` block with `status = 404` routes deep-link misses to the generated `output/404.html`.

TOML

```toml
# Netlify configuration for a Pennington static site.
#
# Netlify serves `publish = "output"` verbatim. Set `BASE_URL` in the
# Netlify dashboard (Site configuration → Environment variables) if you
# need a sub-path; for a root-served site leave it as the default `/`.
#
# The 404 fallback uses Netlify's conditional `status = 404` rewrite so
# deep-link misses return the generated `output/404.html` page body
# instead of Netlify's default 404 shell.

[build]
  command = "dotnet run --project examples/SubPathDeployableExample -- build ${BASE_URL:-/}"
  publish = "output"

[build.environment]
  DOTNET_VERSION = "10.0.x"
  BASE_URL = "/"

[[headers]]
  for = "/_content/*"
  [headers.values]
    Cache-Control = "public, max-age=31536000, immutable"

[[headers]]
  for = "/*"
  [headers.values]
    X-Content-Type-Options = "nosniff"
    Referrer-Policy = "strict-origin-when-cross-origin"

# Pretty-URL fallback: Pennington's DocSite emits `<slug>/index.html`,
# which Netlify already serves at `/<slug>/`. The explicit 404 rule
# below only fires when nothing else matches.
[[redirects]]
  from = "/*"
  to = "/404.html"
  status = 404
```

Cloudflare Pages

Cloudflare Pages has no first-party config file equivalent to SWA or Netlify, so the four shared values live in the project dashboard under **Settings → Builds & deployments**:

- **Build command:** `dotnet run --project <your-project> -- build "${BASE_URL:-/}"` — the `${BASE_URL:-/}` expansion passes the env var through as the base-url argument, defaulting to `/` when it is unset. A bare `-- build` would ignore `BASE_URL` entirely and always deploy at the apex.
- **Build output directory:** `output`
- **Environment variables:** `DOTNET_VERSION=10.0.x`, plus `BASE_URL=/<path>` when serving under a sub-path (leave `BASE_URL` unset for apex hosting).

For custom cache headers on `/_content/*`, drop a `_headers` file into `wwwroot/` so it ships as part of `output/` — the directive format matches the Netlify and Azure snippets above.

Verify

- Trigger a deploy on the target host. The build log shows `setup-dotnet` (or equivalent) picking up `10.0.x`, `dotnet run -- build` exiting zero, and the host uploading `output/` as the publish directory.
- Open the deployed URL — the landing page loads, nested links resolve, and view-source shows the expected `<body data-base-url="...">` (either absent for root deployments, or `/<path>` with no trailing slash for sub-path hosts).
- Visit a non-existent path like `/does-not-exist/` — the response body is the generated `output/404.html` rather than the host's default 404 shell.

Related

- Recipe: Deploy to GitHub Pages — the canonical workflow this page diffs against.
- Recipe: Self-host behind Nginx or IIS — for hosts where you own the web server config instead of a managed platform.
- Recipe: Host under a sub-path (base URL) — how `BaseUrlHtmlRewriter` prefixes internal URLs when the host serves under `/<path>/`.
- Reference: CLI and build arguments — the `build [baseUrl] [outputDirectory]` surface every host command above invokes.

Self-host behind Nginx or IIS

Guides Serve the generated `output/` directory from Nginx or IIS with pretty-URL rewrites and the generated `404.html` as the fallback.

Serve an `output/` directory produced by `dotnet run -- build` from a server you control — a VPS running Nginx or a Windows host running IIS. When a managed static host is an option, Deploy to GitHub Pages is simpler.

Before you begin

- A built `output/` directory (see Build a static site), ready to copy onto the target server.
- Root or administrator access to install a config file and reload the web server.
- The site serves from the domain root. Sub-path deployments (`https://host/docs/`) require building with `dotnet run -- build /docs` — see Host under a sub-path (base URL).

Steps

Upload `output/` to the web root

Copy the full contents of `output/` to the directory the web server will serve — `/var/www/pennington/output/` for Nginx (the path the snippet's `root` points at) or the IIS site's **Physical path** for IIS. Keep the `_content/` folder intact; fingerprinted static-web-asset bundles (Razor library CSS and JS) live under that underscore-prefixed path and ship verbatim.

Install the server config

Drop the snippet for your server into its config location and reload. Both snippets cover trailing-slash directory indexes, the generated `404.html` as the miss fallback (served with a real 404 status), and `public, immutable` cache headers on `/_content/` fingerprinted assets. The IIS snippet also declares MIME types for `.webmanifest` and `.woff2`, which IIS does not know by default; Nginx serves those from its global `mime.types` include.

Nginx

Drop into `/etc/nginx/sites-enabled/` (or `conf.d/`), then `nginx -s reload`.

NGINX

```
# Self-host a Pennington static site behind Nginx.
#
# `root` points at the directory you uploaded from your CI (the
# contents of `output/`). `try_files $uri $uri/ =404` lets the directory
# index serve `<slug>/index.html` for every trailing-slash URL the
# DocSite layout emits, and returns a real 404 status when nothing
# matches; `error_page 404 /404.html` then serves the generated
# `404.html` body for that status.
#
# If you are serving under a sub-path (e.g. `https://host/docs/`), build
# the site with `dotnet run -- build /docs` *and* mount the `output`
# directory at that same sub-path using `location /docs/ { alias … }`.

server {
    listen 80;
    server_name _;

    root /var/www/pennington/output;
    index index.html;

    # Immutable fingerprinted assets.
    location /_content/ {
        expires 1y;
        add_header Cache-Control "public, immutable";
        try_files $uri =404;
    }

    # Pennington writes every content page as `<slug>/index.html`, so
    # the directory-index fallback covers every canonical URL. A miss
    # returns `=404` (a real 404 status) rather than rewriting to
    # `/404.html`, which would serve the body with a 200; `error_page`
    # below then renders the generated `404.html` for that status.
    location / {
        try_files $uri $uri/ =404;
    }

    # DocSite serves `sitemap.xml` and `llms.txt` as top-level files.
    location = /sitemap.xml { default_type application/xml; }
    location = /llms.txt    { default_type text/plain;      }

    error_page 404 /404.html;

    # Security headers (not strictly required for static content but
    # worth having everywhere).
    add_header X-Content-Type-Options nosniff;
    add_header Referrer-Policy strict-origin-when-cross-origin;
}
```

IIS

Drop `web.config` into the site root alongside `index.html`, then run `iisreset` or recycle the app pool.

XML

```
<?xml version="1.0" encoding="utf-8"?>
<!--
  Self-host a Pennington static site behind IIS.

  Drop the contents of `output/` into the IIS site's physical path and
  this `web.config` alongside. The rewrite rule mirrors the Nginx
  `try_files` fallback: serve any directory index, otherwise serve the
  generated `404.html` with an HTTP 404 status.

  IIS does not know about `.webmanifest` or the `application/manifest+json`
  MIME type by default, so those are declared explicitly.
-->
<configuration>
  <system.webServer>
    <staticContent>
      <remove fileExtension=".json" />
      <mimeMap fileExtension=".json" mimeType="application/json" />
      <remove fileExtension=".webmanifest" />
      <mimeMap fileExtension=".webmanifest" mimeType="application/manifest+json" />
      <remove fileExtension=".woff2" />
      <mimeMap fileExtension=".woff2" mimeType="font/woff2" />
    </staticContent>
    <defaultDocument>
      <files>
        <clear />
        <add value="index.html" />
      </files>
    </defaultDocument>
    <httpErrors errorMode="Custom" existingResponse="Replace">
      <remove statusCode="404" />
      <error statusCode="404" path="/404.html" responseMode="File" />
    </httpErrors>
    <rewrite>
      <rules>
        <rule name="Pretty URL -> directory index" stopProcessing="true">
          <match url="^(.*[^/])$" />
          <conditions>
            <add input="{REQUEST_FILENAME}" matchType="IsDirectory" />
          </conditions>
          <action type="Redirect" url="{R:1}/" redirectType="Permanent" />
        </rule>
      </rules>
    </rewrite>
    <httpProtocol>
      <customHeaders>
        <add name="X-Content-Type-Options" value="nosniff" />
        <add name="Referrer-Policy" value="strict-origin-when-cross-origin" />
      </customHeaders>
    </httpProtocol>
  </system.webServer>
</configuration>
```

Serve under a sub-path

When the site does not own the domain root — it lives at `https://host/docs/` — build with the prefix (`dotnet run -- build --base-url=/docs`, see Host under a sub-path (base URL)) so every internal link carries it, then point the server at `output/` under that same path.

For Nginx, mount the directory with `alias` (not `root`) inside a `location` block named for the prefix:

NGINX

```
location /docs/ {
    alias /var/www/pennington/output/;
    try_files $uri $uri/ =404;
}
```

For IIS, host the site as an application or virtual directory named `docs` and drop the same `web.config` into its physical path — the rewrite and `404.html` rules apply relative to the application root, so no changes are needed.

Verify

- Reload the server, then `curl -I https://<host>/` returns `200 OK` with `content-type: text/html; charset=utf-8` and the landing page renders in a browser.
- `curl -I https://<host>/guides/first-page/` returns 200; dropping the trailing slash still resolves (301 → 200 on IIS, 200 directly on Nginx via `try_files $uri/`).
- `curl -I https://<host>/definitely-not-a-page` returns `404 Not Found` and the body is the generated `404.html` rather than the server's default error page.

Related

- Recipe: Build a static site — what `build [baseUrl] [outputDirectory]` produces before you copy `output/` onto the server.
- Recipe: Host under a sub-path (base URL) — how `BaseUrlHtmlRewriter` handles a `/docs/` prefix when your Nginx or IIS site does not own the domain root.
- Reference: CLI and build arguments — the `build [baseUrl] [outputDirectory]` surface that produces the `output/` directory this page serves.
- Background: Dev mode and build mode share one code path — motivates why `404.html` is generated as a real HTTP response rather than a static template.

Host under a sub-path (base URL)

Guides Serve a Pennington site from a non-root URL by passing `[baseUrl]` to the build and letting `BaseUrlHtmlRewriter` prefix every internal href, src, and action.

To serve under a sub-path, pass it as the first argument to `build`. `BaseUrlHtmlRewriter` prefixes every root-relative `href`, `src`, and `action` on the way out; the same `RunOrBuildAsync` call handles root and sub-path identically.

Before you begin

- A working Pennington site that builds locally with `dotnet run -- build` (see Build a static site if not).
- **The sub-path the host will serve from** — for example `/docs` for `https://example.com/docs/` or `/<repo>` for a GitHub Pages project site.
- Internal links authored as root-relative (`/guides/first-page/`). The rewriter only matches the leading `/`; protocol-relative (`//cdn.example.com/x.js`), absolute (`https://…`), hash (`#section`), and page-relative (`./neighbor/`) links pass through untouched.
- The host is already configured to serve `output/` at that sub-path (see Deploy to GitHub Pages or Self-host behind Nginx or IIS).

For a working setup, see examples/SubPathDeployableExample.

Build with the prefix

Pass the sub-path as the `--base-url` flag. Include the leading slash and omit the trailing slash — the rewriter normalizes either way.

BASH

```
dotnet run -- build --base-url=/docs
```

For the positional form and the rest of the argument grammar, see CLI and build arguments.

Reproduce the prefix from client-side code

When an island, Blazor component, or custom script builds URLs at runtime, read the prefix from `document.body.dataset.baseUrl` (stamped by the rewriter) instead of hard-coding `/docs`. The same `output/` then runs under `/docs` in staging and `/` in preview with only a different `--base-url`.

JAVASCRIPT

```
const base = document.body.dataset.baseUrl ?? "";
const href = `${base}/guides/first-page/`;
```

Verify

- Run `dotnet run -- build --base-url=/docs` and open `output/index.html` — every internal `href`, `src`, and `action` now starts with `/docs/`, and `<body>` carries `data-base-url="/docs"`.

- Serve the build so the prefix is part of the path. A static server roots at `/`, so place `output/` inside a folder named for the prefix and serve the parent: `mkdir -p site/docs && cp -r output/* site/docs/ && npx http-server site -p 5000`. Open `http://localhost:5000/docs/` — deep links like `/docs/guides/first-page/` resolve and their in-page links stay under the prefix.
- Re-run with no `--base-url` — the generated HTML reverts to root-relative paths with no `data-base-url` attribute, confirming the rewriter short-circuits when the prefix is empty or `/`.

Related

- Reference: CLI and build arguments — the `build [baseUrl] [outputDirectory]` surface this page drives.
- Background: The response-processing pipeline — why base-URL rewriting runs at `Order => 30`, after xref and locale rewriters.
- Background: Dev mode and build mode share one code path — why the same rewriter runs identically in `dotnet run` and `dotnet run -- build`.