Pennington

Reference

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

Front Matter

Front matter key reference

Reference Every built-in YAML front-matter key recognized by the shipped IFrontMatter implementations, with type, default, source interface, and applicable front-matter record.

YAML keys parsed into the five shipped `IFrontMatter` records — `DocFrontMatter`, `BlogFrontMatter`, `BlogPostFrontMatter`, `DocSiteFrontMatter`, `BlogSiteFrontMatter`. See Pennington.FrontMatter.IFrontMatter for the interface surface and The front-matter capability system for the design rationale.

Keys

Rows are alphabetical by YAML key. Each entry shows the records that expose the key, the declaring interface (`IFrontMatter`, one of the capability interfaces, or `record-local`), and every distinct type and default across records.

`atprotoRkey` `string?` Default: `null` Applies to: BlogPostFrontMatter, BlogSiteFrontMatter. Declared on: `record-local`.

Record key of this post's published `site.standard.document` record (Standard Site), if any. `author` `string / string?` Default: `"" / null` Applies to: BlogFrontMatter, BlogPostFrontMatter, BlogSiteFrontMatter. Declared on: `record-local`.

Author name shown in the post byline and RSS feed. `date` `DateTime?` Default: `null` Applies to: BlogFrontMatter, BlogPostFrontMatter, BlogSiteFrontMatter. Declared on: `IFrontMatter`.

Publication date. Posts are ordered by this date, newest first. `description` `string?` Default: `null` Applies to: BlogFrontMatter, BlogPostFrontMatter, BlogSiteFrontMatter, DocFrontMatter, DocSiteFrontMatter. Declared on: `IFrontMatter`.

Short description used for the meta description and post listings. `isDraft` `bool` Default: `false` Applies to: BlogFrontMatter, BlogPostFrontMatter, BlogSiteFrontMatter, DocFrontMatter, DocSiteFrontMatter. Declared on: `IFrontMatter`.

When true, the post is skipped during production builds. `llms` `bool` Default: `true` Applies to: BlogFrontMatter, BlogPostFrontMatter, BlogSiteFrontMatter, DocFrontMatter, DocSiteFrontMatter. Declared on: `IFrontMatter`.

When false, the post is excluded from the generated llms.txt output. `order` `int` Default: `2147483647` Applies to: DocFrontMatter, DocSiteFrontMatter. Declared on: `IOrderable`.

Sort order within the containing section. Lower values appear first. `redirectUrl` `string?` Default: `null` Applies to: BlogPostFrontMatter, BlogSiteFrontMatter, DocSiteFrontMatter. Declared on: `IRedirectable`.

When set, the post emits a client-side redirect to this URL instead of normal content. `repository` `string` Default: `""` Applies to: BlogSiteFrontMatter. Declared on: `record-local`.

URL to the post's source repository or related project (optional). `search` `bool` Default: `true` Applies to: BlogFrontMatter, BlogPostFrontMatter, BlogSiteFrontMatter, DocFrontMatter, DocSiteFrontMatter. Declared on: `IFrontMatter`.

When false, the post is excluded from the search index. `searchOnly` `bool` Default: `true / false` Applies to: BlogFrontMatter, BlogPostFrontMatter, BlogSiteFrontMatter, DocFrontMatter, DocSiteFrontMatter. Declared on: `IFrontMatter`.

Always true: posts are indexed for search and llms.txt but kept out of the documentation navigation sidebar. Not author-settable — the blog has its own index and tag pages. `sectionLabel` `string?` Default: `null` Applies to: BlogSiteFrontMatter, DocFrontMatter, DocSiteFrontMatter. Declared on: `ISectionable`.

Section heading this page belongs under in navigation. `series` `string / string?` Default: `"" / null` Applies to: BlogFrontMatter, BlogSiteFrontMatter. Declared on: `record-local`.

Series name grouping related posts together. `tags` `string[]` Default: `[]` Applies to: BlogFrontMatter, BlogPostFrontMatter, BlogSiteFrontMatter, DocFrontMatter, DocSiteFrontMatter. Declared on: `ITaggable`.

Tags applied to the post for the tag index and browse-by-tag pages. `title` `string` Default: `"" / "Empty title"` Applies to: BlogFrontMatter, BlogPostFrontMatter, BlogSiteFrontMatter, DocFrontMatter, DocSiteFrontMatter. Declared on: `IFrontMatter`.

Post title rendered in the browser tab and post heading. `uid` `string?` Default: `null` Applies to: BlogFrontMatter, BlogPostFrontMatter, BlogSiteFrontMatter, DocFrontMatter, DocSiteFrontMatter. Declared on: `IFrontMatter`.

Stable identifier used for cross-references (`[text](xref:uid)`).

Parse rules

- YAML keys are the camelCase form of the C# property names. Matching is case-insensitive.
- Unknown keys are dropped with a warning diagnostic in lenient mode (the default outside build); in strict mode (the build default) they throw a `YamlException` and fail the parse.
- Lenient versus strict is controlled by `PenningtonOptions.FrontMatter.StrictUnknownKeys`, settable in the `AddPennington(options => …)` callback. It defaults to `false` (lenient), and `--build` flips it to `true` unless the host has already set it. `diag frontmatter` prints the active value.
- Absent keys fall through to the record's `init` default.

Drafts and scheduled pages

`isDraft: true` excludes a page from build output entirely: `-- build` skips the route, so it is never written, never crawled, and its `uid` does not resolve in the static site. Development requests still render it so authors can preview. A `date:` set after the build clock has the same effect until the clock catches up (scheduled publishing).

This is the canonical statement of the rule. `isDraft` is a build switch, not a navigation switch — to keep a page published but out of the sidebar, use `searchOnly: true` instead, which leaves the route in the build and indexes while hiding it from the rendered navigation tree.

Example

A `DocSiteFrontMatter` page populating the most common keys:

MARKDOWN

````
---
title: Front matter
description: The YAML block at the top of every markdown page.
tags: [authoring, front-matter]
sectionLabel: authoring
order: 20
uid: kitchen-sink.main.front-matter
---

Every page in this site opens with a YAML block between `---` markers.
Those keys drive the sidebar title, description, tags, ordering, draft
state, and cross-reference `uid`. Each built-in front-matter record maps
the same keys onto a strongly-typed record.

## The built-in DocSite record

The DocSite template uses `DocSiteFrontMatter` under the hood. Its fields
cover the full capability surface — `Title`, `Description`, `IsDraft`,
`Tags`, `Order`, `RedirectUrl`, `Section`, `Uid`, `Search`, and `Llms`.

## A custom front-matter record

When you need extra fields, declare a record implementing `IFrontMatter`
(plus any capability interfaces you want). This site ships an
`ApiFrontMatter` record used by the API area to add `Namespace` and
`Stability` fields:

```yaml
---
title: Symbol reference
namespace: Pennington.Search
stability: preview
order: 30
---
```

Declare the record alongside your host project; Pennington discovers it
by type when you call `AddMarkdownContent<ApiFrontMatter>(...)`.
````

A `BlogSiteFrontMatter` page populating the blog-only keys (`author`, `series`, `repository`, `date`):

MARKDOWN

```
---
title: Shipping a tiny content engine for weekend projects
description: Notes from the first month of building Pennington — why Markdig plus Razor
components plus a little YAML beats reaching for a heavier framework.
date: 2026-04-10
author: Author Name
tags:
  - pennington
  - dotnet
  - blogging
series: Pennington Field Notes
repository: https://github.com/example/pennington-field-notes
sectionLabel: field-notes
redirectUrl:
---

Welcome to the first real post on this blog. The scaffold from the previous
tutorial gave us a running BlogSite with one placeholder post; this post
replaces that placeholder with something the BlogSite template actually
has opinions about. Every key in the front-matter block above lights up a
different surface — the archive card, the post header, the `/tags/<tag>`
listings, the RSS channel, the JSON-LD metadata — and walking through them
in order is the point of this tutorial.

## What the front matter is doing

`title`, `description`, and `date` are the three fields that drive every
listing surface: the home page card, the `/archive` page, and the RSS item.
`author` flows into the same places plus the per-post byline and the RSS
`<author>` element; when it matches `BlogSiteOptions.AuthorName` the blog
defaults to the configured author bio for the post chrome.

`tags` build the `/tags/<tag>/` index pages. The three tags above mean this
post shows up under `/tags/pennington/`, `/tags/dotnet/`, and
`/tags/blogging/`. `series` lets several posts thread together under a
shared banner — a later tutorial walks through grouping posts by series.
`repository` is a hint for "view source" links; `section` groups this post
under a named slice of the archive; `redirectUrl` is left empty because
this post has no previous home elsewhere on the web.

## Why you'd bother populating all of it

You can absolutely ship a post with only `title`, `description`, and
`date` — the BlogSite will still render it. But each field you populate
turns on one more piece of chrome: RSS readers show the author, social
previews show the description, `/tags/<tag>/` listings pick up the post,
and the series banner threads together the posts that belong together.
Populating the full set once, on the first post, is how you make the rest
of the blog's defaults work for you.

The next tutorial wires up the homepage hero, the project grid, and the
social-link icons — so the scaffolded blog starts to feel like a real
personal site rather than an archive of posts.
```

See also

- Related reference: IFrontMatter and capability defaults
- Related reference: Built-in front-matter types
- How-to: Work with front matter
- Background: The front-matter capability system

Folder sidecar (`_meta.yml`)

Reference Per-folder YAML sidecar that overrides a folder's display title, sort order in its parent, and llms.txt subtree opt-in.

A `_meta.yml` file dropped in any content folder declares folder-level metadata: an alternative display title, an explicit position in the parent navigation level, and (optionally) opt-in to a dedicated `llms.txt` subtree split. All fields are optional.

Without a sidecar, a folder's sort position is the **min-of-children** value: the smallest `order:` found on any page inside it. The `order` key below overrides that emergent value.

Schema

YAML

```
title: "Reference"
order: 5
llms:
  description: "API surface, host extensions, front-matter keys, ..."
```

```
Key Type Default Effect     `title` string `null` Overrides both `FormatSectionTitle` (auto-generated from the folder slug) and the title from a sibling `index.md`.   `order` int `null` Sets the folder's position among its parent's children. Overrides the emergent min-of-children rule and any `order:` set on a sibling `index.md`.   `llms.description` string `null` When present, opts the folder into `llms.txt` subtree generation. Requires `title` to also be set; without it the description is silently ignored (no subtree, no warning, no error).
```

Resolution rules

- A field that's set wins over every other source for that folder.
- A field that's omitted falls through to the original behavior: `FormatSectionTitle(folderSlug)` for the title, min-of-children for the order, and "not an llms subtree" for the llms split.
- Folders without `_meta.yml` are unaffected — adoption is folder-by-folder.

Discovery

`MarkdownContentService` discovers `_meta.yml` files under its content path, but it is not the only source. `FolderMetadataRegistry` aggregates rows from every registered `IContentService` that implements `IFolderMetadataProvider`, keyed by the folder's canonical URL prefix. A custom content service that surfaces its own folder metadata through that interface contributes sidecars on equal footing. Hot-reload refreshes the registry on file change.

Example

YAML

```yaml
# docs/Pennington.Docs/Content/reference/_meta.yml
title: Reference
llms:
  description: "API surface, host extensions, front-matter keys, Markdig extensions, UI components, diagnostics codes."
```

The `Reference` folder appears in the sidebar with that title, and `/reference/llms.txt` is generated as a subtree split of the main `llms.txt`.

See also

- Background: Navigation-tree construction — how `_meta.yml` interacts with `HierarchyParts` and per-page `order:`.
- Related reference: Front-matter keys — per-page YAML keys including `order:`.

CHAPTER 2

Markdown

Markdown extensions catalog

Reference Every non-CommonMark Markdown feature Pennington adds — tabs, alerts, code annotations, and cross-reference tags — with syntax, arguments, and emitted classes.

The catalog of non-CommonMark Markdown features enabled in Pennington's Markdig pipeline. Markdig's own built-in syntax (tables, footnotes, and so on) is not covered here.

```
Extension Syntax Controlled by Doc page     Tabbed code Adjacent fences with `tabs=true` `UseTabbedCodeBlocks` [Tabbed code](https://usepennington.net/how-to/code-samples/tabbed-code/)   Content tabs `# [Label](#tab/id)` headings ended by `---` `UseContentTabs` [Content tabs](https://usepennington.net/how-to/rich-content/content-tabs/)   Includes `[!INCLUDE …]` (full form under Includes) Markdown parser (always on) [Reuse shared content](https://usepennington.net/how-to/pages/include-shared-content/)   Alerts `> [!KIND]` inside blockquote `UseCustomAlerts` [Alerts](https://usepennington.net/how-to/rich-content/alerts/) Code annotations Trailing-comment `[!code …]` directive `UseSyntaxHighlighting` [Code annotations](https://usepennington.net/how-to/code-samples/code-annotations/)   Cross-reference tags `<xref:uid>` or `href="xref:uid"` Resolved in the response stage [Cross-references](https://usepennington.net/how-to/navigation/linking/)   Shortcodes `<?# Name args /?>` Registered `IShortcode` handlers (pre-parse expansion) [Shortcodes](https://usepennington.net/how-to/markdown-pipeline/shortcodes/)
```

Tabs

Renders a run of consecutive fenced code blocks (starting with one that carries `tabs=true`) as a single tabbed container with `role="tablist"`, `role="tab"`, and panel regions. The first tab is active by default.

Syntax

MARKDOWN

````
```csharp tabs=true title="C#"
// block A
```


```razor title="Razor"
@* block B *@
```
````

Each fenced block in the consecutive run becomes a tab panel; only the first block requires `tabs=true` to open the group.

Arguments

```
Name Type Default Description     `tabs` `true` (absent) Applies to the first fence in the group. Marks a fenced block as the start of a tabbed run; consecutive subsequent fences join the same group.   `title` `string` (optionally quoted) pretty language name derived from the info string Applies to each fence in the group. Overrides the label shown on the tab button.
```

Arguments are `key=value` pairs; quoted values are allowed. See Code-block argument reference for the full grammar.

Emitted CSS classes

```
Option Default class Role     `OuterWrapperCss` `not-prose` Outer `<div>` wrapper that opts out of prose styling.   `ContainerCss` `tab-container` Container wrapping tablist and panels.   `TabListCss` `tab-list` `role="tablist"` row.   `TabButtonCss` `tab-button` `role="tab"` `<button>` (carries `data-state="active"|"inactive"`).   `TabPanelCss` `tab-panel` `aria-labelledby`-bound panel wrapping the rendered code block.
```

Classes are configurable via `TabbedCodeBlockRenderOptions` passed to `UseTabbedCodeBlocks`.

Minimal example

Markdown source showing a two-fence tabbed group:

MARKDOWN

````
---
title: Authoring a doc page
description: Populate DocSiteFrontMatter, add an alert, and group code samples into tabs.
tags:
  - authoring
  - front-matter
  - markdown
sectionLabel: Guides
order: 20
---

# Authoring a doc page

## Callouts

> [!NOTE]
> Alerts render with a coloured left border and an icon matching the kind.

## Tabbed code groups

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

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

```xml title="csproj"
<PackageReference Include="Pennington" Version="*" />
```
````

Content tabs

Groups a run of DocFX-style tab headings into a single tabset whose panels hold arbitrary Markdown — prose, lists, code, callouts. Distinct from tabbed code: the tab strip sits in the reading flow rather than inside code chrome.

Syntax

A tab opens with a level-1 heading whose only inline is a link to `#tab/<id>`; the link text is the button label. Consecutive tab headings form one group, ended by a thematic break (`---`).

MARKDOWN

```
# [Bash](#tab/bash)

Use the bash variant.

# [PowerShell](#tab/pwsh)

Use the PowerShell variant.

---
```

Dependent tabs

A third path segment — `#tab/<id>/<condition>` — gates the tab on another group's selection. `<condition>` is a tab id that has its own plain group elsewhere on the page; the dependent panel shows only when its `<id>` is the active button and its `<condition>` is that other group's selected id.

MARKDOWN

```
# [.NET](#tab/lang/linux)

.NET on Linux.

# [.NET](#tab/lang/windows)

.NET on Windows.

---
```

Arguments

```
Name Type Default Description     `id` identifier — First segment after `#tab/`. Identifies the tab; ids are page-wide, so equal ids select together.   `condition` identifier (absent) Optional second segment. Gates the tab on the selected id of the condition's own group.
```

Emitted CSS classes

```
Element Class Attributes Role     Container `ctabs` `data-content-tabs` Tabset wrapper; not `not-prose`.   Tab strip `ctabs-bar not-prose` `role="tablist"` Button row; `not-prose` isolates the buttons from page typography.   Button `ctab-btn` `data-tab`, `data-active`, `role="tab"`, `aria-selected` One per distinct id.   Panel `ctab-panel` `data-tab`, `data-condition`, `data-active`, `role="tabpanel"` One per tab heading; not `not-prose`, so content keeps prose styling.
```

The first panel is active in the server-rendered HTML; the client script recomputes selection on load, syncs equal ids page-wide, and persists each choice in `localStorage`.

Minimal example

A two-tab group whose panels hold prose; the `---` thematic break closes the set:

MARKDOWN

```
# [Bash](#tab/bash)

Run the bash script.

# [PowerShell](#tab/pwsh)

Run the PowerShell script.

---
```

Includes

Splices a referenced Markdown file into the host page during parsing. The directive works as a standalone block or inline within a sentence; the referenced file is expanded recursively.

Syntax

MARKDOWN

```
[!INCLUDE [block partial](../_includes/partial.md)]

Text before [!INCLUDE [inline partial](../_includes/snippet.md)] and after.
```

Arguments

```
Name Type Default Description     `title` string — Bracketed label. Parsed but not emitted;
present for DocFX compatibility.   `path` path — File path resolved relative to the
referencing file. Absolute URLs are not fetched.
```

Behavior

```
Case Result     Target file found Content is spliced in; a leading YAML front-matter block
is stripped first.   Directive in a fenced code block Left verbatim, so the syntax can be
documented.   Target missing Replaced with `<!-- Pennington: include not found: <path> -->`.
Include cycle, or depth past 16 Replaced with `<!-- Pennington: include cycle broken: <path>
-->`.   Absolute URL path Replaced with `<!-- Pennington: include skipped (not a local
file): <path> -->`.
```

Relative links and images inside an included file are not rebased — they resolve as if written in the host page.

Minimal example

A standalone block include splicing a shared partial into the host page:

MARKDOWN

```
[!INCLUDE [install steps](../_includes/install.md)]
```

Alerts

Parses a GitHub-flavored `> [!KIND]` token as the first line of a blockquote and emits an `AlertBlock` with two CSS classes. The blockquote form is the only accepted syntax.

Syntax

MARKDOWN

```
> [!NOTE]
> Body text of the alert, CommonMark rendered.
```

`KIND` is a case-insensitive alphabetic token.

Arguments

```
Name Type Default Description     `KIND` identifier — One of `NOTE`, `TIP`, `CAUTION`,
`WARNING`, `IMPORTANT`. Case-insensitive. Unrecognized tokens still parse, emitting
`markdown-alert-<kind>` using the lowercased value.
```

Built-in kinds and emitted CSS classes

Every alert receives two classes: `markdown-alert` (constant) and `markdown-alert-<kind>`.

```
Kind Secondary class Typical use     `NOTE` `markdown-alert-note` Supplementary information.
`TIP` `markdown-alert-tip` Helpful aside.   `CAUTION` `markdown-alert-caution` Risky
operation.   `WARNING` `markdown-alert-warning` Something likely to go wrong.   `IMPORTANT`
`markdown-alert-important` Must-read information.
```

Minimal example

Markdown excerpt showing a `[!NOTE]` block in context:

MARKDOWN

```
---
title: Authoring a doc page
description: Populate DocSiteFrontMatter, add an alert, and group code samples into tabs.
tags:
  - authoring
  - front-matter
  - markdown
sectionLabel: Guides
order: 20
---

# Authoring a doc page

## Callouts

> [!NOTE]
> Alerts render with a coloured left border and an icon matching the kind.
> Supported kinds include `NOTE`, `TIP`, `IMPORTANT`, `WARNING`, and
> `CAUTION`.
```

Code annotations

After syntax highlighting, each rendered line is scanned for a `[!code …]` directive inside a language-appropriate comment. The directive is stripped and a CSS class is applied to the line (and optionally to the enclosing `<pre>`). The `word:` variant wraps a matching substring; the `include-start` / `include-end` / `exclude-start` / `exclude-end` directives remove surrounding lines from the output.

Syntax

MARKDOWN

```
```csharp
var x = 1; // [!code highlight]
var y = 2; // [!code ++]
var z = 3; // [!code word:z|renamed from q]
```
```

The directive must appear inside a comment marker recognized for the language; the marker and any now-empty comment wrapper are removed, leaving trailing content intact. Code-block argument reference is the canonical reference for the full directive set (`highlight`, `++` / `--`, `focus`, `error`, `warning`, `word:`, the `include` / `exclude` region markers) and the recognized comment markers.

Emitted CSS classes

Line-level classes (`highlight`, `diff-add`, `diff-remove`, `focused` / `blurred`, `error`, `warning`) are added to the `<span class="line">` wrapper. Block-level classes (`has-highlighted`, `has-diff`, `has-focused`, `has-errors`, `has-warnings`, `has-word-highlights`) are added to the outer `<pre>`.

The `word:` notation emits `word-highlight` or `word-highlight-with-message` on the wrapped span, plus the callout elements `word-highlight-wrapper`, `word-highlight-message`, `word-highlight-arrow-container`, `word-highlight-arrow-outer`, and `word-highlight-arrow-inner`.

Minimal example

An annotated fence exercising `[!code highlight]`, `[!code ++]`, and `[!code --]`; the enclosing `<pre>` receives `has-highlighted` and `has-diff`, and the trailing directive comments are stripped from the emitted HTML:

MARKDOWN

````
```csharp
var message = "hello";   // [!code highlight]
var added = "added";     // [!code ++]
var removed = "gone";    // [!code --]
```
````

Cross-reference tags

`xref:` links resolve after rendering against the uid-to-route map built from every page's front-matter `uid:`. Two surface forms are supported: the tag form `<xref:uid>` and the attribute form `[text](xref:uid)`. Unknown uids emit a diagnostic that surfaces in the dev overlay and in the static-build report.

Syntax

MARKDOWN

```
See <xref:reference.api.pennington-options>.

See [PenningtonOptions](xref:reference.api.pennington-options).
```

`uid` is the exact string declared in a page's front-matter `uid:` key.

Arguments

```
Name Type Default Description     `uid` identifier — Exact string declared in a page's
front-matter `uid:` key. The tag form derives link text from the target page's title; the
attribute form uses the supplied link text verbatim.
```

Emitted CSS classes

The rewriter emits a standard `<a href="…">` element with no added class; styling is delegated to the surrounding prose stylesheet.

Minimal example

Both surface forms resolving the same uid — the tag form derives its link text from the target page title, the attribute form uses the supplied label verbatim:

MARKDOWN

```
See <xref:reference.api.pennington-options> for the full options catalog.

Configure MonorailCSS through [the options record](xref:reference.api.monorail-css-options).
```

Shortcodes

Shortcodes run before Markdig parses the page rather than as a Markdig extension: each `<?# Name args /?>` directive is expanded to text or HTML first by a registered `IShortcode` handler, and the result flows through the pipeline as ordinary markdown. Names are case-insensitive.

Syntax

A call has a self-closing form and a block form with inline content:

MARKDOWN

```
<?# Name positional key="value" /?>

<?# Name ?>inline content<?#/ Name ?>
```

To document a directive without expanding it, prefix the opener with a backslash — the expander emits the directive verbatim. (Every literal directive on this page uses that escape.)

Built-ins

```
Name Arguments Description     `Version` `format=full\|major\|minor\|informational` (default
`full`) Emits the host application's assembly version.   `PackageVersion` none Emits
Pennington's own published NuGet version.
```

Unknown names and handler failures degrade to an HTML comment plus a warning diagnostic, so one bad call site never fails the render. See the shortcodes how-to for the handler contract and registration.

See also

- How-to: Add a Markdig extension or inline parser — register syntax this catalog doesn't list
- How-to: Tabbed code
- How-to: Content tabs
- How-to: Reuse shared content
- How-to: Alerts
- How-to: Code annotations
- How-to: Cross-references

- How-to: Expand a directive before Markdig parses
- Related reference: Code-block argument reference

Code-block argument reference

Reference The fence info-string grammar Pennington parses — language token, key/value attributes, quoted values — and the trailing-comment [!code …] directive grammar used for line annotations.

Pennington tokenises the fence info-string — the opening-fence text after the three backticks — left-to-right into a language (with an optional colon-suffix) followed by `key=value` attribute pairs. The `[!code …]` directive grammar runs separately, against the highlighted HTML.

Fence info-string grammar

TEXT

```
info-string   := language [ ":" suffix ] ( WS attribute )*
language      := IDENT
suffix        := "symbol" symbol-flag*
               |  "symbol-diff" [ ",bodyonly" ]
symbol-flag   := ",bodyonly" | ",imports" | ",signatures"
attribute     := key "=" value
key           := IDENT
value         := bare-value | "'" quoted-value "'" | '"' quoted-value '"'
bare-value    := any run of non-whitespace chars
quoted-value  := any chars up to the matching quote
```

`language` is typically `csharp`, `razor`, `text`, and so on. Quoting is required only when a value contains whitespace, and attribute keys are matched case-insensitively. Authors of a custom Markdig extension read the same fence through Markdig's own API: the language and colon-suffix arrive on `FencedCodeBlock.Info` and the attribute tail on `FencedCodeBlock.Arguments`.

Attributes

```
Name Values Description Example     `tabs` `true` Marks adjacent fenced blocks for grouping into a single tabbed widget. ````  ```csharp tabs=true title="C#"````   `title` any quoted string Tab label shown on the tabbed widget; falls back to the normalized language name when omitted. ````  ```csharp tabs=true title="Program.cs"````
```

Suffix forms (code-embedding)

```
Form Body shape Description     `<lang>:symbol` one `<file>` path, optionally followed by `> Member.Path`, per line Embeds the whole file, or the named member's declaration and body. Concatenated in order.    `<lang>:symbol,bodyonly` same as `:symbol` Embeds only the member body, stripping the declaration line and enclosing braces.    `<lang>:symbol,imports` same as `:symbol` Prepends the file's top-of-file import/using/require statements above the snippet. Composes with `,bodyonly`; a no-op for whole-file embeds and import-less languages.  `<lang>:symbol,signatures` same as `:symbol` Replaces member bodies with an elision marker (`{ … }`, or `…` for non-brace bodies) for an outline view. Mutually exclusive with `,bodyonly`.    `<lang>:symbol-diff` exactly two references, before then after Emits a unified diff between the two members' source text. Accepts the `,bodyonly` suffix.
```

The `symbol` flags (`,bodyonly`, `,imports`, `,signatures`) are comma-separated and order-independent. Suffix forms are resolved by an `ICodeBlockPreprocessor`; `Pennington.TreeSitter` ships the implementations for `symbol` and `symbol-diff`.

`[!code …]` directives

A directive is the literal text `[!code <notation>]` wrapped in a line-trailing comment marker recognized for the block's language (`//`, `#`, `--`, `<!--`, `*`, `%`, `'`, `REM`, `;`, `/*`); the directive pass runs against the highlighted HTML. The comment marker is stripped when the directive consumes the whole comment and preserved when trailing content remains; the directive itself is always removed from the rendered line.

```
Directive Behavior Example     `highlight` (alias `hl`) Adds class `highlight` to the line and `has-highlighted` to the `<pre>`. `var x = 1; // `    `++` Adds class `diff-add` to the line and `has-diff` to the `<pre>`. `var x = 1; // [!code ++]`    `--` Adds class `diff-remove` to the line and `has-diff` to the `<pre>`. `var x = 0; // [!code --]`    `focus` Adds class `focused` to the line, `has-focused` to the `<pre>`, and `blurred` to every non-focused line. `var x = 1; // [!code focus]`    `error` Adds class `error` to the line and `has-errors` to the `<pre>`. `throw new(); // [!code error]`    `warning` Adds class `warning` to the line and `has-warnings` to the `<pre>`. `// TODO // [!code warning]` `word:TEXT` Wraps the first occurrence of `TEXT` on the line in `<span class="word-highlight">`. `var token = Get(); // [!code word:token]`    `word:TEXT&#124;MESSAGE` As `word:`, but wraps the span in a callout carrying `MESSAGE`. `// [!code word:token&#124;The auth token]`    `include-start` / `include-end` Marks a region to retain; all lines outside paired include regions are dropped. `// [!code include-start]` ... `// [!code include-end]` `exclude-start` / `exclude-end` Marks a region to drop; lines inside are removed from the rendered output. `// [!code exclude-start]` ... `// [!code exclude-end]`
```

See also

- How-to: Annotate code blocks
- How-to: Create tabbed code groups
- Related reference: Markdown extensions catalog
- Background: The syntax-highlighting cascade

CHAPTER 3

UI

Navigation components

Reference Parameters, slots, and NavigationInfo bindings for the four Pennington.UI navigation components — TableOfContentsNavigation, OutlineNavigation, Breadcrumb, and Pagination.

The four navigation-oriented Razor components in `Pennington.UI`. `TableOfContentsNavigation` and `OutlineNavigation` render, respectively, the sidebar page tree and the floating in-page heading outline, and live in namespace `Pennington.UI.Components.Navigation`. `Breadcrumb` and `Pagination` render the article-header trail and prev/numbered/next paging controls, and live in the base namespace `Pennington.UI.Components`. All four are consumed by `Pennington.DocSite`'s `MainLayout`.

`TableOfContentsNavigation`

Declaration

RAZOR

```
@inject IServiceProvider Services
@if (TableOfContents != null)
{
    <nav>
        <ul class="@_list">
            @foreach (var tableOfContentEntry in TableOfContents.OrderBy(i => i.Order))
            {
                @TocEntry(tableOfContentEntry)
            }
        </ul>
    </nav>
}

@code {
    /// <summary>Navigation tree to render; when null the component renders nothing, and
entries are sorted by <see cref="NavigationTreeItem.Order"/> at each level.</summary>
    [Parameter] public ImmutableList<NavigationTreeItem>? TableOfContents { get; set; }

    /// <summary>Optional label forwarded from the caller's
<c>NavigationInfo.SectionName</c>; not rendered by the default template.</summary>
    [Parameter] public string? SectionLabel { get; set; }

    /// <summary>Visual archetype: <see cref="TocVariant.Rail"/> (default) or <see
cref="TocVariant.Pill"/>. A site template sets this once to pick a cohesive look.</summary>
    [Parameter] public TocVariant Variant { get; set; } = TocVariant.Rail;

    /// <summary>Classes Tailwind-merged over the variant's outer <c>&lt;ul&gt;</c> base.
</summary>
    [Parameter] public string? ListClass { get; set; }

    /// <summary>Classes Tailwind-merged over the variant's per-section <c>&lt;li&gt;</c>
base.</summary>
    [Parameter] public string? SectionClass { get; set; }

    /// <summary>Classes Tailwind-merged over the variant's section-label base (the
<c>&lt;div&gt;</c> for empty-route entries, or the <c>&lt;a&gt;</c> when a top-level entry
has children).</summary>
    [Parameter] public string? SectionTitleClass { get; set; }

    /// <summary>Classes Tailwind-merged over the variant's nested <c>&lt;ul&gt;</c> base
that holds a section's child entries.</summary>
    [Parameter] public string? SectionListClass { get; set; }

    /// <summary>Classes Tailwind-merged over the variant's child-link base; the base also
carries the <c>data-[current=true]</c> state styling.</summary>
    [Parameter] public string? LinkClass { get; set; }

    /// <summary>Classes Tailwind-merged over the variant's top-level leaf-link base; the
base also carries the <c>data-[current=true]</c> state styling.</summary>
    [Parameter] public string? TopLinkClass { get; set; }
```

```
service and
    // we append instead. Blazor's [Inject] has no optional mode, hence GetService.
private ClassMerge? _merge;      private string _list = "";    private string _section = "";
private string _sectionTitle = "";    private string _sectionList = "";    private string
_link = "";    private string _topLink = "";      protected override void OnInitialized() =>
_merge = Services.GetService<ClassMerge>();      protected override void OnParametersSet()
{        var slots = TocVariantStyles.For(Variant);        _list = Apply(slots.List,
ListClass);        _section = Apply(slots.Section, SectionClass);        _sectionTitle =
Apply(slots.SectionTitle, SectionTitleClass);        _sectionList = Apply(slots.SectionList,
SectionListClass);        _link = Apply(slots.Link, LinkClass);        _topLink =
Apply(slots.TopLink, TopLinkClass);    }      // A registered ClassMerge resolves conflicts;
without one (bare host) we append, accepting    // that conflicting base utilities are not
removed.    private string Apply(string baseClasses, string? extra)        => _merge is not
null            ? _merge.Apply(baseClasses, extra)            : string.IsNullOrEmpty(extra)
? baseClasses : $"{baseClasses} {extra}".Trim();      private RenderFragment
TocEntry(NavigationTreeItem tocEntry) =>        @<li class="@_section">            @if
(tocEntry.Route.CanonicalPath.Value == "")            {                <div
class="@_sectionTitle">@tocEntry.Title</div>            }            else            {
<a data-current="@tocEntry.IsSelected.ToString().ToLowerInvariant()"
href="@tocEntry.Route.CanonicalPath.Value" class="@(tocEntry.Children.Count == 0 ? _topLink
: _sectionTitle)">@tocEntry.Title</a>            }            @if (tocEntry.Children.Count >
0)            {                <ul class="@_sectionList">                    @foreach (var
childEntry in tocEntry.Children.OrderBy(i => i.Order).Where(i => i.Route.CanonicalPath.Value
!= ""))                    {                        <li>                            <a data-
current="@childEntry.IsSelected.ToString().ToLowerInvariant()"
href="@childEntry.Route.CanonicalPath.Value" class="@_link">@childEntry.Title</a>
</li>                    }                </ul>            }        </li>;}
```

Renders an ordered `<nav><ul>` of `NavigationTreeItem` entries, recursing one level into each entry's `Children` collection and sorting by `NavigationTreeItem.Order` at each level. Root entries with an empty `Route.CanonicalPath` render as plain section headers; entries with a path render as anchor links carrying `data-current="true"` when `IsSelected` is set.

Parameters

Every `*Class` parameter defaults to `null`, meaning the component renders its built-in default for that element. A value you pass is Tailwind-merged over that default for the instance — conflicting utilities are replaced, the rest kept. `Variant` selects which built-in look those defaults come from.

```
Name Type Default Description     `TableOfContents` `ImmutableList<NavigationTreeItem>?`
`null` Navigation tree to render; when `null` the component renders nothing.
`SectionLabel` `string?` `null` Optional label forwarded from the caller's
`NavigationInfo.SectionName`; not rendered by the default template.   `Variant` `TocVariant`
`TocVariant.Rail` Built-in look: `Rail` (a bordered left rail) or `Pill` (rounded pill
buttons with a tinted active state).   `ListClass` `string?` `null` The outer `<ul>` that
holds the top-level navigation entries.   `SectionClass` `string?` `null` Each top-level
`<li>`.   `SectionTitleClass` `string?` `null` A section's label — the plain `<div>` for
empty-route entries, or the `<a>` when a top-level entry has children.   `SectionListClass`
`string?` `null` The nested `<ul>` that holds a section's child entries.   `LinkClass`
`string?` `null` Each child-level `<a>`, including its `data-current=true` state styling.
`TopLinkClass` `string?` `null` A top-level leaf `<a>` (an entry with no children),
including its `data-current=true` state styling.
```

Binding

`TableOfContents` accepts an `ImmutableList<NavigationTreeItem>` produced by `await NavigationBuilder.BuildTreeAsync(items, currentPath, locale)`. It does not accept a `NavigationInfo`. `SectionLabel` is typically passed from `NavigationInfo.SectionName`. No `RenderFragment` slots.

Example

RAZOR

```razor
@{
    var tree = await NavigationBuilder.BuildTreeAsync(items, currentPath, locale);
}

<TableOfContentsNavigation TableOfContents="tree" SectionLabel="@navigation.SectionName" />
```

`OutlineNavigation`

Declaration

RAZOR

```
@inject IServiceProvider Services
@if (!string.IsNullOrWhiteSpace(Title))
{
    <div class="@_title">@Title</div>
}
<div data-role="page-outline" data-content-selector="@ContentSelector" class="relative
@_container">
    <div data-role="page-outline-highlighter" class="absolute opacity-0 @_marker"></div>
    <div>
        <ul class="@_list"
            data-outline-link-class="@_link"
            data-outline-nested-link-class="@_nestedLink">
            @* Outline links will be dynamically generated by JavaScript *@
        </ul>
    </div>
</div>

@code {
    /// <summary>CSS selector the client-side outline script queries to discover heading
elements; must be non-empty for the outline to populate.</summary>
    [Parameter, EditorRequired] public string ContentSelector { get; set; } = "";

    /// <summary>Optional eyebrow rendered above the outline; pass an empty string to
suppress.</summary>
    [Parameter] public string Title { get; set; } = "On this page";

    /// <summary>Classes Tailwind-merged over the eyebrow above the outline list.</summary>
    [Parameter] public string? TitleClass { get; set; }

    /// <summary>Classes Tailwind-merged over the outer <c>data-role="page-outline"</c>
container; <c>relative</c> stays hardcoded for marker positioning.</summary>
    [Parameter] public string? ContainerClass { get; set; }

    /// <summary>Classes Tailwind-merged over the moving highlight bar; <c>absolute</c> and
<c>opacity-0</c> stay hardcoded — the client script positions the bar and toggles its
opacity.</summary>
    [Parameter] public string? MarkerClass { get; set; }

    /// <summary>Classes Tailwind-merged over the outline <c>&lt;ul&gt;</c>.</summary>
    [Parameter] public string? ListClass { get; set; }

    /// <summary>Classes Tailwind-merged over the link base, emitted as <c>data-outline-
link-class</c> and applied by the client-side script to each generated <c>&lt;a&gt;</c>; the
base also carries the <c>data-[selected=true]</c> state styling.</summary>
    [Parameter] public string? LinkClass { get; set; }

    /// <summary>Classes Tailwind-merged over the nested-link base, emitted as <c>data-
outline-nested-link-class</c> and appended by the client-side script to nested (H3-level)
outline links.</summary>
    [Parameter] public string? NestedLinkClass { get; set; }
```

```
has no
    // optional mode, hence GetService.    private ClassMerge? _merge;      private string
_title = "";    private string _container = "";    private string _marker = "";    private
string _list = "";    private string _link = "";    private string _nestedLink = "";
protected override void OnInitialized() => _merge = Services.GetService<ClassMerge>();
// Defaults are inline literals (a method body, so an edit hot-reloads); the per-instance
// *Class param merges over each.    protected override void OnParametersSet()    {
_title = Apply("font-display text-[13px] font-semibold mb-3 text-base-600 dark:text-base-
300", TitleClass);        _container = Apply("border-l border-base-200 dark:border-base-
800", ContainerClass);        _marker = Apply("left-[-1px] w-[2px] rounded-sm bg-primary-600
dark:bg-primary-300 transition-all duration-500", MarkerClass);        _list = Apply("list-
none pl-4 text-base-500 dark:text-base-400", ListClass);        _link = Apply("block py-1
ml-[calc(-1*(4em-1px))] pl-[calc(4em+1px)] transition-colors duration-150 hover:text-base-
900 dark:hover:text-base-50 data-[selected=true]:text-primary-700 dark:data-
[selected=true]:text-primary-300 data-[selected=true]:font-medium", LinkClass);
_nestedLink = Apply("pl-4", NestedLinkClass);    }      // A registered ClassMerge resolves
conflicts; without one (bare host) we append, accepting    // that conflicting base
utilities are not removed.    private string Apply(string baseClasses, string? extra)
=> _merge is not null            ? _merge.Apply(baseClasses, extra)            :
string.IsNullOrEmpty(extra) ? baseClasses : $"{baseClasses} {extra}".Trim();}
```

Emits a `data-role="page-outline"` container and an empty `<ul>` whose items are populated client-side by scraping headings from the element matched by `ContentSelector`. The component performs no server-side heading extraction; the companion script in `Pennington.UI/wwwroot/` reads `data-content-selector`, `data-outline-link-class`, and `data-outline-nested-link-class` to build and highlight the outline in the browser.

Parameters

`ContentSelector` is `[EditorRequired]`. Every `*Class` parameter defaults to `null`, meaning the component renders its built-in default for that element; a value you pass is Tailwind-merged over that default for the instance.

```
Name Type Default Description     `ContentSelector` `string` `""` (required) CSS selector
the client-side outline script queries to discover heading elements; must be non-empty for
the outline to populate.   `Title` `string` `"On this page"` Eyebrow rendered above the
outline list as a `<div>`; pass an empty string to suppress.   `TitleClass` `string?` `null`
The eyebrow above the outline list.   `ContainerClass` `string?` `null` The outer `data-
role="page-outline"` container; `relative` stays hardcoded for marker positioning.
`MarkerClass` `string?` `null` The moving highlight bar that tracks the active heading;
`absolute` and `opacity-0` stay hardcoded — the script positions the bar and toggles its
opacity.   `ListClass` `string?` `null` The outline `<ul>`.   `LinkClass` `string?` `null`
Emitted as `data-outline-link-class` and applied by the client-side script to each generated
`<a>`, including its `data-selected=true` state styling.   `NestedLinkClass` `string?`
`null` Emitted as `data-outline-nested-link-class` and appended by the client-side script to
nested (H3-level) outline links.
```

Binding

The component performs no server-side heading extraction. The outline list is populated at runtime by the companion client script in `Pennington.UI/wwwroot/`, which queries the element matched by `ContentSelector` and reads `data-content-selector`, `data-outline-link-class`, and `data-outline-nested-link-class` from the container. `NavigationInfo` is not consulted. No `RenderFragment` slots.

Example

RAZOR

```
<OutlineNavigation ContentSelector="#main-content" Title="On this page" />
```

`Breadcrumb`

Declaration

RAZOR

```
@* Visible breadcrumb trail intended to live inside an article header.
   Pairs with the ImmutableList<BreadcrumbItem> exposed by
   Pennington.Navigation.NavigationInfo (built by NavigationBuilder).
   Use TrailingContent for an "Edit on GitHub" link or other right-aligned
   chrome on the same line. *@
@using System.Collections.Immutable
@using Pennington.Navigation

@if (Items.Count > 0)
{
    <nav class="flex items-center gap-2 font-display text-[12.5px] font-medium text-base-500
dark:text-base-400 min-w-0 flex-wrap" aria-label="Breadcrumb">
        @for (var i = 0; i < Items.Count; i++)
        {
            var item = Items[i];
            var isLast = i == Items.Count - 1;
            var url = item.Route?.CanonicalPath.Value;
            var hasUrl = !string.IsNullOrEmpty(url);
            if (hasUrl && !isLast)
            {
                <a href="@url" class="hover:text-base-900 dark:hover:text-base-50
transition-colors">@item.Title</a>
            }
            else if (isLast)
            {
                <span class="text-base-700 dark:text-base-200" aria-
current="page">@item.Title</span>
            }
            else
            {
                <span>@item.Title</span>
            }
            if (!isLast)
            {
                <span class="text-base-300 dark:text-base-700" aria-hidden="true">/</span>
            }
        }
        @if (TrailingContent is not null)
        {
            <span class="ml-auto flex items-center gap-3">
                @TrailingContent
            </span>
        }
    </nav>
}

@code {
    /// <summary>The breadcrumb trail to render. Empty list renders nothing.</summary>
    [Parameter] public ImmutableList<BreadcrumbItem> Items { get; set; } = [];

    /// <summary>
    /// Optional content rendered on the trailing edge of the breadcrumb row
    /// (e.g. "Edit on GitHub" link). Pushed right via <c>ml-auto</c>.
```

```
    /// </summary>    [Parameter] public RenderFragment? TrailingContent { get; set; }}
```

Renders a visible breadcrumb trail inside an article header from the `ImmutableList<BreadcrumbItem>` `NavigationBuilder` exposes via `NavigationInfo`. Each item links to its route except the last (which renders as a current-page `<span aria-current="page">`); the trail renders nothing when the list is empty. `TrailingContent` supplies optional right-aligned chrome on the same row (an "Edit on GitHub" link, repository metadata).

Parameters

```
Name Type Default Description     `Items` `ImmutableList<BreadcrumbItem>` `[]` The breadcrumb trail to render. Empty list renders nothing.   `TrailingContent` `RenderFragment?` `null` Optional content rendered on the trailing edge of the breadcrumb row; pushed right via `ml-auto`.
```

Binding

`Items` accepts the `ImmutableList<BreadcrumbItem>` exposed as `NavigationInfo.Breadcrumbs`. The last item renders as the current page; every prior item with a `Route` renders as a link. `TrailingContent` is a `RenderFragment` slot for right-aligned chrome on the same row.

Example

RAZOR

```
<Breadcrumb Items="navigation.Breadcrumbs">
    <TrailingContent>
        <a href="@editUrl">Edit on GitHub</a>
    </TrailingContent>
</Breadcrumb>
```

`Pagination`

Declaration

RAZOR

```
@* Prev / numbered / next pagination controls. Caller supplies the page-N URL via UrlFor —
   the component is URL-shape agnostic so it works for /archive/page/N/,
/tags/{tag}/page/N/,
   /docs/page-N/, or anything else. Renders nothing when TotalPages <= 1. *@

@if (TotalPages > 1)
{
    <nav class="mt-12 flex items-center justify-between gap-4 border-t border-base-200
dark:border-base-800 pt-6" aria-label="Pagination">
        <div class="flex-1">
            @if (CurrentPage > 1)
            {
                <a href="@UrlFor(CurrentPage - 1)" rel="prev" class="inline-flex items-
center gap-2 rounded-md px-3 py-2 text-sm font-medium text-base-700 dark:text-base-200
hover:bg-base-100 dark:hover:bg-base-800 transition-colors">
                    <svg viewBox="0 0 16 16" fill="none" aria-hidden="true" class="h-4 w-4">
                        <path d="M9.25 5.75 6.75 8l2.5 2.25" stroke="currentColor" stroke-
width="1.5" stroke-linecap="round" stroke-linejoin="round"></path>
                    </svg>
                    Previous
                </a>
            }
        </div>

        <ol class="hidden sm:flex items-center gap-1">
            @foreach (var slot in BuildSlots())
            {
                @if (slot.IsGap)
                {
                    <li aria-hidden="true" class="px-2 text-sm text-base-400 dark:text-base-
500">...</li>
                }
                else if (slot.Page == CurrentPage)
                {
                    <li>
                        <span aria-current="page" class="inline-flex h-8 min-w-8 items-
center justify-center rounded-md px-2 text-sm font-semibold bg-primary-600 text-white
dark:bg-primary-500">@slot.Page</span>
                    </li>
                }
                else
                {
                    <li>
                        <a href="@UrlFor(slot.Page)" class="inline-flex h-8 min-w-8 items-
center justify-center rounded-md px-2 text-sm font-medium text-base-700 dark:text-base-200
hover:bg-base-100 dark:hover:bg-base-800 transition-colors">@slot.Page</a>
                    </li>
                }
            }
        </ol>
```

```
        <div class="flex-1 flex justify-end">            @if (CurrentPage < TotalPages)
{                <a href="@UrlFor(CurrentPage + 1)" rel="next" class="inline-flex items-
center gap-2 rounded-md px-3 py-2 text-sm font-medium text-base-700 dark:text-base-200
hover:bg-base-100 dark:hover:bg-base-800 transition-colors">                    Next
<svg viewBox="0 0 16 16" fill="none" aria-hidden="true" class="h-4 w-4">
<path d="M6.75 5.75 9.25 8l-2.5 2.25" stroke="currentColor" stroke-width="1.5" stroke-
linecap="round" stroke-linejoin="round"></path>                    </svg>
</a>            }        </div>    </nav>}  @code {    /// <summary>1-based page index for
the current view. Highlighted in the numbered list.</summary>    [Parameter] public int
CurrentPage { get; set; } = 1;      /// <summary>Total number of pages. The component
renders nothing when this is 1 or less.</summary>    [Parameter] public int TotalPages {
get; set; } = 1;      /// <summary>    /// Returns the URL for a given 1-based page index.
The component never calls this with    /// the current page, but callers should be prepared
to map page <c>1</c> to the canonical    /// (non-paginated) URL of the listing.    ///
</summary>    [Parameter] public Func<int, string> UrlFor { get; set; } = page => $"?page=
{page}";      /// <summary>    /// Number of numeric page links flanking the current page in
the truncated list. The    /// first and last pages are always rendered; gaps between them
and the window collapse    /// to "...". Default of 1 yields windows like <c>1 ... 4 5 6 ...
12</c>.    /// </summary>    [Parameter] public int SiblingCount { get; set; } = 1;
private IEnumerable<Slot> BuildSlots()    {        if (TotalPages <= 1)        {
yield break;        }          var lo = Math.Max(2, CurrentPage - SiblingCount);        var
hi = Math.Min(TotalPages - 1, CurrentPage + SiblingCount);          yield return new Slot(1,
false);          if (lo > 2)        {            yield return new Slot(0, true);        }
for (var p = lo; p <= hi; p++)        {            yield return new Slot(p, false);        }
if (hi < TotalPages - 1)        {            yield return new Slot(0, true);        }
if (TotalPages > 1)        {            yield return new Slot(TotalPages, false);        }
}      private readonly record struct Slot(int Page, bool IsGap);}
```

Prev / numbered / next pagination controls. URL-pattern agnostic — the caller supplies a `Func<int, string>` that maps a 1-based page index to a URL, so the same component drives `/archive/page/N/`, `/tags/{tag}/page/N/`, or any other pattern. Renders nothing when `TotalPages` is 1 or less.

Parameters

```
Name Type Default Description     `CurrentPage` `int` `1` 1-based page index for the current
view; highlighted in the numbered list.   `TotalPages` `int` `1` Total number of pages. The
component renders nothing when this is 1 or less.   `UrlFor` `Func<int, string>` `page => "?
page={page}"` Returns the URL for a given 1-based page index. Callers should map page 1 to
the canonical (non-paginated) URL of the listing.   `SiblingCount` `int` `1` Number of
numeric page links flanking the current page in the truncated list. The first and last pages
are always rendered; gaps collapse to `...`. Default of 1 yields windows like `1 ... 4 5 6
... 12`.
```

Binding

`Pagination` does not consult `NavigationInfo`. The caller supplies `CurrentPage` and `TotalPages` as plain integers and maps each 1-based page index to a URL through the `UrlFor` delegate, so the same component drives any paging URL shape. No `RenderFragment` slots.

Example

RAZOR

```
@{
    string PageUrl(int page) => page == 1 ? "/archive/" : $"/archive/page/{page}/";
}

<Pagination CurrentPage="currentPage" TotalPages="totalPages" UrlFor="PageUrl" />
```

See also

- How-to: Customize the sidebar
- Related reference: Navigation types
- Related reference: Content components

Content components

Reference Parameter and usage reference for the ten Pennington.UI content components — Badge, BigTable, Card, CardGrid, Checkpoint, LinkCard, Step, Steps (Mdazor-registered), RenderedFixture (DocSite Mdazor only), and CodeBlock (Razor-page only).

The content-oriented subset of the `Pennington.UI.Components` Razor component library, covering callout cards, numbered steps, syntax-highlighted code, and wide-table overflow handling. Components live in namespace `Pennington.UI.Components`. All but `CodeBlock` are pre-registered with Mdazor by `DocSiteServiceExtensions.AddDocSite`, making them available as tags inside markdown without additional wiring; `CodeBlock` is Razor-page-only — markdown authors use fenced code blocks instead.

Stylesheet

The components ship as MonorailCSS utility classes; the package contributes no separate stylesheet. There is no `_content/Pennington.UI/styles.css` to load.

Overview

```
Component Purpose Razor usage Markdown (Mdazor) usage     `Badge` Inline pill rendering a
short label in one of five variants. `<Badge Text="New" Variant="tip" />` `<Badge Text="New"
Variant="tip" />`   `BigTable` Wraps a wide table in a horizontal-scroll container.
`<BigTable>@ChildContent</BigTable>` `<BigTable>` ... markdown table ... `</BigTable>`
`Card` Static callout card with optional icon and title. `<Card Title="..."
Color="primary">@ChildContent</Card>` `<Card Title="..." Color="primary">` ... `</Card>`
`CardGrid` Responsive grid container for Card / LinkCard children. `<CardGrid
Columns="3">@ChildContent</CardGrid>` `<CardGrid Columns="3">` ... `</CardGrid>`
`Checkpoint` "Verify what you should see now" callout for tutorial pages.
`<Checkpoint>@ChildContent</Checkpoint>` `<Checkpoint>` ... `</Checkpoint>`   `CodeBlock`
Razor-page entry to the shared code-block rendering pipeline. `<CodeBlock
Language="csharp">@ChildContent</CodeBlock>` Not registered — use a fenced code block.
`LinkCard` Clickable card wrapping its content in an anchor. `<LinkCard Title="..."
Href="/foo">@ChildContent</LinkCard>` `<LinkCard Title="..." Href="/foo">` ... `</LinkCard>`
`RenderedFixture` Renders a solution fixture file as a captioned `<figure>`.
`<RenderedFixture Path="examples/Foo/bar.md" />` `<RenderedFixture
Path="examples/Foo/bar.md" />` (DocSite only)   `Step` Single numbered list item inside a
`Steps` container. `<Step StepNumber="1">@ChildContent</Step>` `<Step StepNumber="1">` ...
`</Step>`   `Steps` Vertical numbered-step list container for `Step` children.
`<Steps>@ChildContent</Steps>` `<Steps>` ... `</Steps>`
```

Each component is listed alphabetically below with a one-line summary, a parameter table, and a minimal usage example.

`Badge`

Inline pill with ring and tinted background, variant-mapped to a MonorailCSS color palette; renders a `<span>` and is safe inside flowing prose and table cells.

Parameters

```
Name Type Default Description     `Size` `string` `"medium"` One of `"small"`, `"medium"`,
`"large"`; drives padding and text size.   `Text` `string` `""` Label rendered inside the
badge when `ChildContent` is not set.   `ChildContent` `RenderFragment?` `null` Markup
rendered inside the badge; takes precedence over `Text`.   `Variant` `string` `"note"` One
of `"note"`, `"success"`, `"tip"`, `"caution"`, `"danger"`; selects the color palette (base
/ emerald / sky / amber / rose).
```

Example

RAZOR

```
<Badge Text="New" Variant="tip" Size="small" />
```

`BigTable`

Overflow wrapper for tables wider than the main column; emits a `<div>` with horizontal scroll and reduced text size, with the table supplied as `ChildContent`.

Parameters

```
Name Type Default Description     `ChildContent` `RenderFragment?` `null` The table content (typically a `<table>` element or a GFM table when used from Mdazor).
```

Example

RAZOR

```
<BigTable>
    <table>...</table>
</BigTable>
```

`Card`

Static non-clickable callout card; renders a rounded, tinted panel with an optional icon region and bold heading, with `not-prose` applied inside so surrounding prose styles do not affect card body content.

Parameters

```
Name Type Default Description     `ChildContent` `RenderFragment?` `null` Body content rendered beneath the title.   `Color` `string` `"primary"` MonorailCSS color-family name used to tint borders, background, text, and icon fill.   `Icon` `RenderFragment?` `null` Optional leading icon fragment rendered to the left of the title + body stack.   `Title` `string?` `null` Bold heading rendered above `ChildContent`.
```

Example

RAZOR

```
<Card Title="Fast" Color="emerald">
    Pages render in a single SSR pass through the content pipeline.
</Card>
```

`CardGrid`

Responsive grid container for `Card` or `LinkCard` children; renders one column on small viewports and a `Columns`-wide grid from the `sm` breakpoint up, with `Columns` interpolated into a MonorailCSS class.

Parameters

```
Name Type Default Description     `ChildContent` `RenderFragment?` `null` Grid items — typically `<Card>` or `<LinkCard>` children.   `Columns` `string` `"2"` Number of columns at the `sm` breakpoint and above; passed through as a MonorailCSS class fragment.
```

Example

RAZOR

```
<CardGrid Columns="3">
    <LinkCard Title="Getting started" Href="/tutorials" />
    <LinkCard Title="How-to guides" Href="/how-to" />
    <LinkCard Title="Reference" Href="/reference" />
</CardGrid>
```

`Checkpoint`

Standalone "verify what you should see now" callout for tutorial pages; emits `<div class="markdown-alert markdown-alert-checkpoint not-prose">` with a literal **Checkpoint** label paragraph and the body content beneath it, sharing chrome with GitHub-style alerts. Renders as a `<div>`, not a heading, so the right-side outline nav skips it.

Parameters

```
Name Type Default Description     `ChildContent` `RenderFragment?` `null` Body of the checkpoint callout — usually one or two paragraphs and a verification list.
```

Example

RAZOR

```
<Checkpoint>

Run `dotnet run` and visit `http://localhost:5000/`.

- The page renders with the expected H1.
- The right-side outline lists only the real section headings.

</Checkpoint>
```

`CodeBlock`

Razor-page entry to the shared code-block rendering pipeline. A modifier-bearing `Language` (for example, `csharp:symbol`) routes through the registered `ICodeBlockPreprocessor` implementations — the public extension point that the tree-sitter `:symbol` family plugs into when `AddTreeSitter` is wired. Not registered with Mdazor — markdown authors should use a fenced code block (same pipeline, same output) instead.

Parameters

```
Name Type Default Description     `ChildContent` `RenderFragment?` `null` Code content as the component's child text; de-indented before rendering.   `Code` `string?` `null` Code content as a string attribute; takes precedence over `ChildContent` when both are set.   `IsInTabGroup` `bool` `false` When `true`, omits standalone container classes so the block composes inside a tabbed code group.   `Language` `string` `""` Required (`EditorRequired`) [fence info-string](#p--reference-markdown-code-block-args) — a bare language like `"csharp"` or a modifier-bearing form like `"csharp:symbol,bodyonly"` (the `:symbol` family ships with `Pennington.TreeSitter`).
```

Example

RAZOR

```
<CodeBlock Language="csharp">
var x = 1;
</CodeBlock>

<CodeBlock Language="csharp:symbol,bodyonly">examples/Foo/Program.cs >
Program.Main</CodeBlock>
```

`LinkCard`

Clickable variant of `Card`; wraps the entire card body in an `<a>` bound to `Href`, with hover states tinting the background using the `Color` palette.

Parameters

```
Name Type Default Description     `ChildContent` `RenderFragment?` `null` Body content rendered beneath the title.   `Color` `string` `"primary"` MonorailCSS color-family name used for borders, background, hover state, text, and icon fill.   `Href` `string?` `null` Destination URL; passed through directly to the wrapping anchor.   `Icon` `RenderFragment?` `null` Optional leading icon fragment.   `Title` `string?` `null` Bold heading rendered above `ChildContent`.
```

Example

RAZOR

```
<LinkCard Title="Getting started" Href="/tutorials/getting-started" Color="primary">
    A zero-to-running-site walkthrough.
</LinkCard>
```

`RenderedFixture` (DocSite only)

Embeds a fixture file (markdown or HTML) from anywhere in the solution as a captioned `<figure>`, rendering markdown through the standard `MarkdownPipeline`. Useful when a how-to page wants to show the actual rendered output of a complete example file (a full alert syntax, a composed configuration)

rather than authoring the same content twice. Registered with Mdazor by `AddDocSite` but not by `AddBlogSite`.

Parameters

```
Name Type Default Description     `Path` `string` `""` (required) Solution-relative path to the fixture file (for example, `examples/Foo/Content/bar.md`). Rejected when it contains `..` or is rooted.   `Caption` `string?` `null` Caption shown in the `<figcaption>` above the rendered output. When unset or blank, the figcaption renders the literal `"Rendered output"`.
```

Example

RAZOR

```
<RenderedFixture Path="examples/DocSitePagesAndLinksExample/snippets/markdown-alert-example.md"
                 Caption="Built-in alert syntax" />
```

`Step`

One step inside a `Steps` list; renders a `<section class="step">` with a numbered medallion on the left rail, an optional title, the body content, and an optional `Checkpoint` slot for a "verify the result" callout. Must be nested directly inside `<Steps>` for the rail border to align.

Parameters

```
Name Type Default Description     `ChildContent` `RenderFragment?` `null` Step body content, rendered to the right of the numbered medallion.   `Checkpoint` `RenderFragment?` `null` Optional inline checkpoint rendered after the body using `markdown-alert-checkpoint` chrome — handy for tutorials that verify a result at the end of a step.   `StepNumber` `string` `"1"` Label shown inside the circular medallion — a string so non-numeric markers (for example, `"A"`, `"i"`) are possible.   `Title` `string?` `null` Optional title rendered above the body content.
```

Example

RAZOR

```
<Steps>
    <Step StepNumber="1" Title="Install the template">Run the dotnet new install command.
</Step>
    <Step StepNumber="2">Run <code>dotnet run</code>.</Step>
</Steps>
```

`Steps`

Container for a vertical step list; emits `<div class="steps-thread not-prose">` rendering a vertical thread on the left, populated by `Step` children which punch a numbered medallion onto the thread.

Parameters

```
Name Type Default Description     `ChildContent` `RenderFragment?` `null` One or more
`<Step>` children.
```

Example

RAZOR

```razor
<Steps>
    <Step StepNumber="1">First, install.</Step>
    <Step StepNumber="2">Then, run.</Step>
    <Step StepNumber="3">Finally, deploy.</Step>
</Steps>
```

Mdazor registration

`AddDocSite` pre-registers nine components with Mdazor (eight content components plus `RenderedFixture`); `AddBlogSite` registers eight (everything but `RenderedFixture`). Sites built on either template can invoke these tags directly inside markdown without calling `AddMdazorComponent<T>()` manually.

CSHARP

```csharp
// AddDocSite
services.AddMdazorComponent<Badge>()
        .AddMdazorComponent<BigTable>()
        .AddMdazorComponent<Card>()
        .AddMdazorComponent<CardGrid>()
        .AddMdazorComponent<Checkpoint>()
        .AddMdazorComponent<LinkCard>()
        .AddMdazorComponent<RenderedFixture>() // AddDocSite only
        .AddMdazorComponent<Step>()
        .AddMdazorComponent<Steps>();
```

Hosts without `AddDocSite` register the same surface through one `AddMdazorComponent<T>()` call per component. See Drop a Razor component into a markdown page for the host-by-host recipe.

See also

- How-to: Use UI components inside markdown
- Related reference: Navigation components
- Related reference: Utility components

Utility components

Reference The three Pennington.UI utility components — LanguageSwitcher, StructuredData, and FallbackNotice — their parameters and a one-line use-when row each.

`Pennington.UI` ships three utility Razor components — `LanguageSwitcher`, `StructuredData`, and `FallbackNotice` — that handle locale selection, JSON-LD head injection, and fallback-locale notification respectively. All three live in namespace `Pennington.UI.Components` and are made available via the `@using Pennington.UI.Components` import.

`LanguageSwitcher`

Renders a `<details>`-backed dropdown of alternate-language links configured for SPA reload via `data-spa-reload`. Hides itself when fewer than two locales are available. Auto-computes the list from `LocaleContext` and `LocalizationOptions` when `AlternateLanguages` is null or empty.

Parameters

```
Name Type Default Description     `AlternateLanguages`
`IReadOnlyList<AlternateLanguageItem>?` `null` Explicit list of alternate-language items;
when null or empty, the component auto-computes the list from the injected `LocaleContext`
and `LocalizationOptions`.
```

`AlternateLanguageItem`

Nested record type that callers supply when constructing an explicit `AlternateLanguages` list; `DocSite`'s `MainLayout` builds one instance per locale per-request from `DocSiteContentResolver.GetAlternateLanguagesAsync`.

```
Name Type Description     `Locale` `string` Locale code written to the `data-locale`
attribute on the rendered `<a>`.   `DisplayName` `string` Visible label used in the dropdown
row and as the currently-selected summary text.   `Url` `string` `href` written on the
anchor; typically a locale-prefixed canonical path.   `IsCurrentLocale` `bool` When `true`,
the row renders with current-locale styling (`font-semibold` and the primary accent color).
```

Example

RAZOR

```razor
@if (LocalizationOptions.IsMultiLocale)
{
    <LanguageSwitcher AlternateLanguages="_langSwitcherItems" />
}
```

`StructuredData`

Emits one `<script type="application/ld+json">` per supplied entity into the document `<head>` via `<HeadContent>`. Accepts any sequence of `JsonLdEntity` — including user-defined subclasses — and serializes each with `JsonLdSerializer`. Null entries in the sequence are skipped.

Parameters

```
Name Type Default Description     `Entities` `IEnumerable<JsonLdEntity>?` `null` Sequence of schema.org entities to emit. Each is serialized by `JsonLdSerializer.Serialize` and rendered as its own `<script type="application/ld+json">` block.
```

Example

RAZOR

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

@code {
    private IEnumerable<JsonLdEntity> entities = [
        new JsonLdArticle { Headline = "...", Url = "..." },
        new JsonLdBreadcrumbList { Items = [...] },
    ];
}
```

To define a schema.org type Pennington doesn't ship, see Add a custom schema.org JSON-LD type.

`FallbackNotice`

Renders an inline amber notice banner above the article region when the requested locale has no translation and the page is being served from the default locale. Renders nothing when `RequestedLocale` is null or empty.

Parameters

```
Name Type Default Description     `RequestedLocale` `string?` `null` Locale code the visitor requested; when non-null and non-empty, the notice renders and displays this value as the unavailable locale.   `DefaultLocale` `string?` `null` Locale code the page is served in; displayed in the notice as the locale the visitor sees instead.
```

Example

RAZOR

```
<FallbackNotice RequestedLocale="@Article.FallbackRequestedLocale"
                DefaultLocale="@LocalizationOptions.DefaultLocale" />
```

See also

- Related reference: Navigation components — sibling `Pennington.UI` reference page for `TableOfContentsNavigation` and `OutlineNavigation`.

- Related reference: Content components — sibling `Pennington.UI` reference page for `Card`, `Badge`, `CodeBlock`, and the rest of the content-authoring API.
- How-to: Add a custom schema.org JSON-LD type — define a new `JsonLdEntity` subclass and emit it through `StructuredData`.
- How-to: Add a second locale to your site — tutorial that wires `LanguageSwitcher` and `FallbackNotice` end-to-end via `AddDocSite`.

CHAPTER 4

Spa

SPA engine attributes and events

Reference The data-spa-* attribute contract the spa-engine.js script in Pennington.UI reads — region markers, scroll keys, stylesheet handling, root tuning, and lifecycle events.

`Pennington.UI` ships `spa-engine.js`, a navigation script that fetches the destination URL, parses the HTML response, swaps marked regions, and merges head changes. This page catalogs the `data-spa-*` attribute contract the engine reads from markup, plus the three custom events it dispatches on `document`.

For the design rationale, see SPA navigation through region swaps.

Region attributes

Every attribute in this table is read on a region element — the element marked `data-spa-region`. The engine discovers regions on every navigation, so attributes can be added or removed by other code at runtime.

```
Name Values Description     `data-spa-region` identifier Marks the element as a swap target; the value is the region name and must be unique per page.   `data-spa-region-key` any string Identifies the content set inside the region; when the incoming element's key differs from the current one, the region and its closest scrollable ancestor have `scrollTop` reset to `0`. Absent keys carry scroll position across navigations.
```

Anchor and stylesheet attributes

```
Selector Attribute Description     `<a>` `data-spa-reload` Forces a full-page navigation for that link; the engine treats it as a non-SPA anchor. Used by `LanguageSwitcher` so the locale change re-runs the request pipeline.   `<a>` `target="_blank"` or `download` Excluded from SPA handling automatically; no opt-in attribute required.   `<link rel="stylesheet">` `data-spa-reload` Re-fetches the stylesheet with a `_spa=<timestamp>` cache-buster query on every navigation. Opt-in only; see [why dev needs it and production doesn't] (https://usepennington.net/explanation/core/dev-vs-build/).
```

Document-root tuning

One integer attribute on `<html>` adjusts the progress bar timing.

```
Name Type Default Description     `data-spa-progress-delay` milliseconds `100` Threshold the engine waits before showing the top-of-viewport progress bar; navigations that resolve faster never show it.
```

Parsed once on script load. Changing it after the script runs has no effect.

Lifecycle events

All three events fire on `document`. Listeners attached outside a swapped region survive every navigation; listeners attached inside a region must be re-bound from a `spa:commit` handler because the region's contents are replaced via `innerHTML`.

```
Event `detail` shape When it fires     `spa:before-navigate` `{ url, slug }` After a same-origin link click is intercepted, before the fetch is issued.   `spa:commit` `{ url, slug, doc }` After head and region swaps complete, in the same synchronous block as the DOM replacement. `doc` is the parsed `Document` of the fetched response — the read-side contract for patching elements that live outside any region. See [Persistent chrome](https://usepennington.net/explanation/spa/islands/#persistent-chrome).   `spa:diagnostics` `Diagnostic[]` After `spa:commit`, only when the response carries a `<script type="application/spa-diagnostics+json">` block.
```

`<script>` elements inside a swapped region are re-created so the parser executes them on commit; scripts of type `application/ld+json` and `application/spa-diagnostics+json` are skipped.

Boundary fallbacks

The engine falls back to `location.href = url` (full page load) in three cases.

```
Trigger Reason     No `[data-spa-region]` on the current page Nothing to swap.   Incoming document's region set differs from the current one Layout boundary; for example, `MainLayout` (`content`, `outline`) → `FullWidthLayout` (`content`).   Fetch fails Network or server error; the browser handles the navigation.
```

See also

- Background: SPA navigation through region swaps — why the engine re-fetches the canonical URL instead of a JSON envelope, and when persistent chrome beats a swap region.
- Related reference: Utility components — `LanguageSwitcher` is the canonical consumer of `data-spa-reload` on an anchor.
- Related reference: Diagnostics request context — origin of the payload `spa:diagnostics` carries.

CHAPTER 5

Host

DI and middleware extension methods

Reference Index of every AddPennington/UsePennington/Run* extension method across the referenced Pennington packages.

The list of public extension methods Pennington exposes for wiring the library into an ASP.NET Core host — `Add*` (DI registration), `Use*` (middleware and endpoints), and `Run*` (host entry points). Grouped below by receiver type; each method is declared in an `*Extensions` static class under its owning feature namespace.

`IServiceCollection` extensions

DI registration entry points. The three composition roots and the options record each configures:

- AddPennington — PenningtonOptions
- AddDocSite — DocSiteOptions
- AddBlogSite — BlogSiteOptions

The full set follows, each tagged with its owning package.

`AddApiMetadataFromCompiledAssembly` `IServiceCollection` `AddApiMetadataFromCompiledAssembly(this IServiceCollection services, Action<CompiledAssemblyApiOptions> configure)` Package `Pennington.ApiMetadata.Reflection`

Convenience overload: registers under the `"default"` name for sites documenting a single library. `AddApiMetadataFromCompiledAssembly` `IServiceCollection` `AddApiMetadataFromCompiledAssembly(this IServiceCollection services, string name, Action<CompiledAssemblyApiOptions> configure)` Package `Pennington.ApiMetadata.Reflection`

Registers `CompiledAssemblyApiMetadataProvider` as a keyed `IApiMetadataProvider` under `name`. Call once per library you want to document — each call builds its own `MetadataLoadContext` and xmldoc index scoped to the supplied `AssemblyDirectories`. The shared `IXmlDocParser` / `IXmlDocHtmlRenderer` services are registered once (idempotent). `AddApiReference` `IServiceCollection AddApiReference(this IServiceCollection services, string name, Action<ApiReferenceRegistrationOptions> configure)` Package `Pennington.DocSite.Api`

Registers one named API-reference tree. Call once per library you want to document. Each call pairs with a matching `AddApiMetadataFrom*(name, …)` provider registration and publishes its type pages at the configured `RoutePrefix`. `AddBlogSite` `IServiceCollection AddBlogSite(this IServiceCollection services, Func<BlogSiteOptions> configureOptions)` Package `Pennington.BlogSite`

Registers BlogSite services with the provided options. `AddDataDirectory<TItem>` `IServiceCollection AddDataDirectory<TItem>(this IServiceCollection services, string name, string path)` Package `Pennington`

Registers every `.yml`, `.yaml`, and `.json` file in `path` as a single aggregated `IReadOnlyList` accessible through `IDataFiles` under the lookup key `name`. Each file contributes one record, or several when its root is an array; files are ordered by name. Edits, additions, and removals in the directory invalidate the cached value so the next read returns the fresh content. `AddDataFile<T>` `IServiceCollection AddDataFile<T>(this IServiceCollection services, string name, string path)` Package `Pennington`

Registers `path` as a data file accessible through `IDataFiles` under the lookup key `name`. Format is inferred from the file extension (`.yml`, `.yaml`, `.json`). Edits to the file invalidate the cached value so the next read returns the fresh content. `AddDocSite` `IServiceCollection AddDocSite(this IServiceCollection services, Func<DocSiteOptions> configureOptions)` Package `Pennington.DocSite`

Registers DocSite services with the provided options. `AddFileWatched<T>` `IServiceCollection AddFileWatched<T>(this IServiceCollection services)` Package `Pennington`

Register a concrete service whose instance is managed by `FileWatchDependencyFactory`. `AddFileWatched<TService, TImplementation>` `IServiceCollection AddFileWatched<TService, TImplementation>(this IServiceCollection services)` Package `Pennington`

Register a service whose instance is managed by `FileWatchDependencyFactory`. The factory (singleton) recreates the instance when the implementation's `OnFileChanged` returns `Recreate`. The service (transient) always returns the current instance from the factory. `AddHead` `IServiceCollection AddHead(this IServiceCollection services)` Package `Pennington`

Registers the head composition rewriter. Inert until at least one `IHeadContributor` is also registered, so adding this on its own leaves head output byte-identical. `AddHeadContributor<T>` `IServiceCollection AddHeadContributor<T>(this IServiceCollection services)` Package `Pennington`

Registers a single head contributor. Transient so contributors capturing a file-watched dependency (e.g. the content registry) pick up the current instance per request. `AddLlmsSubtree` `IServiceCollection AddLlmsSubtree(this IServiceCollection services, LlmsSubtree subtree)` Package `Pennington`

Registers a `LlmsSubtree` so all leaves under `RoutePrefix` get split out into a dedicated `{RoutePrefix}llms.txt`. Multiple registrations are allowed; programmatic registrations override `_meta.yml`-discovered subtrees with the same prefix. `AddMonorailCss` `IServiceCollection AddMonorailCss(this IServiceCollection services, Func<IServiceProvider, MonorailCssOptions> optionFactory)` Package `Pennington.MonorailCss`

Registers MonorailCSS services and the runtime class-discovery pipeline. With no configuration, the discovery pipeline force-loads every non-BCL assembly the app references, scans each one's IL, watches the project's source files in development, and loads `wwwroot/app.css` as the source CSS prefix when present. The CSS endpoint served by `UseMonorailCss` regenerates whenever the class set changes. `AddPennington` `IServiceCollection AddPennington(this IServiceCollection services, Action<PenningtonOptions> configure)` Package `Pennington`

Register all Pennington services. `AddPenningtonBook` `IServiceCollection AddPenningtonBook(this IServiceCollection services, Action<BookOptions> configure)` Package `Pennington.Book`

Adds PDF book generation: a per-locale book per `BookDefinition` (or one whole-site book when none are configured), served on demand at `/pdf/{slug}.pdf` in dev and emitted into the static build. Registers an `IDownloadLinkProvider` a host's chrome can advertise. `AddTaxonomy<TFrontMatter, TKey>` `IServiceCollection AddTaxonomy<TFrontMatter, TKey>(this IServiceCollection services, Action<TaxonomyOptions<TFrontMatter, TKey>> configure)` Package `Pennington`

Registers a `TaxonomyContentService` configured by `configure`. Multiple `AddTaxonomy` calls with the same `TFrontMatter` / `TKey` pair coexist as long as each uses a distinct `BaseUrl`. `AddTranslationAudit` `IServiceCollection AddTranslationAudit(this IServiceCollection services, Action<TranslationAuditOptions> configure)` Package `Pennington.TranslationAudit`

Register `TranslationAuditor` as an `IBuildAuditor`. Diagnostics land in the dev overlay (per-page) and in the build report (site-wide) automatically. `AddTreeSitter` `IServiceCollection AddTreeSitter(this IServiceCollection services, Action<TreeSitterOptions> configure)` Package `Pennington.TreeSitter`

Adds tree-sitter based multi-language code-fragment extraction — the `:symbol` fence modifier. Services are registered only when `ContentRoot` is configured. `AddWordBreak` `IServiceCollection AddWordBreak(this IServiceCollection services, Action<WordBreakOptions> configure)` Package `Pennington`

Registers `WordBreakHtmlRewriter` in the shared HTML rewriting pipeline, so long identifiers in the configured elements get `<wbr>` break opportunities without an extra DOM parse. `AddYamlContext` `IServiceCollection AddYamlContext(this IServiceCollection services, YamlSerializerContext context)` Package `Pennington`

Register a source-generated `YamlSerializerContext` so the types it covers deserialize without reflection (NativeAOT/trim-friendly). Types not covered by any registered context fall back to reflection. Satellite templates call this for their own front-matter records; end users call it for theirs. `ReplaceContentRenderer<TOld, TNew>` `IServiceCollection ReplaceContentRenderer<TOld, TNew>(this IServiceCollection services)` Package `Pennington`

Replaces every registered `IContentRenderer` with `TNew`, resolved through DI as a transient. The `TOld` type parameter documents the renderer being swapped out — it is informational and unused at runtime, but lets the call site read as "replace TOld with TNew". `ReplaceContentRenderer<TOld, TNew>`

`IServiceCollection ReplaceContentRenderer<TOld, TNew>(this IServiceCollection services, Func<IServiceProvider, TNew> factory)` Package `Pennington`

Replaces every registered `IContentRenderer` with one produced by `factory`. Use this overload when the new renderer takes ctor arguments DI cannot resolve (e.g. a version string or per-site constant).

`WebApplication` extensions

Middleware and endpoint wiring. The template `Use*` methods each wrap a fixed sequence, listed below.

`RunBlogSiteAsync` `Task RunBlogSiteAsync(this WebApplication app, string[] args)` Package `Pennington.BlogSite`

Runs the BlogSite: either serves the app or performs a static build, based on command-line args. `RunDocSiteAsync` `Task RunDocSiteAsync(this WebApplication app, string[] args)` Package `Pennington.DocSite`

Runs the DocSite: either serves the app or performs a static build, based on command-line args. `RunOrBuildAsync` `Task RunOrBuildAsync(this WebApplication app, string[] args)` Package `Pennington`

Runs the host: serves live (no verb), builds the static site (`build`), or runs a diagnostic command (`diag <sub>`). Everything flows through one System.CommandLine pipeline, so `--help` / `--version` work at the root and every subcommand. Build and diag run one-shot against a started in-memory host that is disposed afterward; serve hands off to `RunAsync` . `UseBlogSite` `WebApplication UseBlogSite(this WebApplication app)` Package `Pennington.BlogSite`

Wires BlogSite middleware, Razor components, and RSS endpoint into the request pipeline. `UseDocSite` `WebApplication UseDocSite(this WebApplication app)` Package `Pennington.DocSite`

Wires DocSite middleware and Razor components into the request pipeline. `UseLiveReload` `WebApplication UseLiveReload(this WebApplication app)` Package `Pennington`

Adds live reload WebSocket support for development. Skipped during static build (see `PenningtonCli`). `UseLocaleRouting` `WebApplication UseLocaleRouting(this WebApplication app)` Package `Pennington`

Adds locale detection and URL path rewriting middleware. Must be called `MapRazorComponents` so that Blazor routing sees the locale-stripped path (e.g., `/gen-z/schedule` becomes `/schedule`). Called automatically by `UsePennington` when it hasn't been called yet, but at that point it is too late for Blazor endpoint routing. Sites that use `@page` directives with locale prefixes must call this explicitly. `UseMonorailCss` `WebApplication UseMonorailCss(this WebApplication app, string path)` Package `Pennington.MonorailCss`

Maps the MonorailCSS stylesheet endpoint. The endpoint pulls the current class set from the discovery pipeline registered in `AddMonorailCss` , generates CSS, and serves it. `UsePennington` `WebApplication UsePennington(this WebApplication app)` Package `Pennington`

Configure the Pennington middleware pipeline.

`UseDocSite` middleware order

`UseDocSite` registers this sequence before mapping the Razor component endpoint:

1. `UseLocaleRouting`
2. `UseAntiforgery`
3. `UseStaticFiles`
4. `UseMonorailCss`
5. `UsePennington`
6. `MapRazorComponents<App>()`

`UseBlogSite` middleware order

`UseBlogSite` registers the same sequence minus locale routing, which BlogSite does not wire:

1. `UseAntiforgery`
2. `UseStaticFiles`
3. `UseMonorailCss`
4. `UsePennington`
5. `MapRazorComponents<App>()`

For why each step lands where it does, see Dev mode and build mode share one code path.

`Run*` host entry points

Host entry points that run one System.CommandLine pipeline: serve live with no verb, build the static site with `build`, or run a read-only inspection with `diag <sub>`. Build and diag run one-shot against a started in-memory host that is disposed afterward; serve hands off to `RunAsync`.

- RunOrBuildAsync — the core dispatcher; call it directly on a bare `AddPennington` host.
- RunDocSiteAsync — DocSite wrapper over `RunOrBuildAsync`.
- RunBlogSiteAsync — BlogSite wrapper over `RunOrBuildAsync`.

Example

A complete DocSite host wiring all three layers — `AddDocSite`, `UseDocSite`, `RunDocSiteAsync` — in call order.

CSHARP

```
using Pennington.DocSite;

var builder = WebApplication.CreateBuilder(args);

// Swap the bare `AddPennington` host for the DocSite template. `AddDocSite`
// wires the full documentation experience on top of Pennington core — a
// Blazor-rendered layout with sidebar navigation, header, search surface,
// outline nav, dark-mode toggle — driven entirely from `DocSiteOptions`.
builder.Services.AddDocSite(() => new DocSiteOptions
{
    SiteTitle = "Scaffold Docs",
    SiteDescription = "A minimal DocSite scaffold built on AddDocSite.",
    GitHubUrl = "https://github.com/usepennington/pennington",
    HeaderContent = """<a href="/">Scaffold Docs</a>""",
    FooterContent = """<footer class="mt-16 py-8 text-center text-sm text-base-500">Built
with Pennington DocSite.</footer>""",
});

var app = builder.Build();

// `UseDocSite` mounts locale routing, antiforgery, static files, Razor
// component routing (`Pages.razor` owns `/{*fileName:nonfile}`), MonorailCSS,
// SPA navigation, and the core Pennington middleware in the right order.
app.UseDocSite();

// `RunDocSiteAsync` delegates to `RunOrBuildAsync`, so `dotnet run` serves live
// and `dotnet run -- build <baseUrl> <outputDir>` generates static HTML. Both
// positional args are optional (defaults: `/` and `output`).
await app.RunDocSiteAsync(args);
```

See also

- Reference: CLI and build arguments
- Reference: PenningtonOptions
- Reference: DocSiteOptions
- Background: Dev mode and build mode share one code path

CLI and build arguments

Reference The argument and environment-variable surface for RunOrBuildAsync — positional args, named flags, and the variables consulted when the host boots.

The command-line arguments and environment variables `RunOrBuildAsync` dispatches on. Two verbs are recognized — `build` (generate the static site) and `diag` (read-only inspection); dev-serving is the default when neither verb is present. `--help`, `-h`, `-?`, and `--version` print and exit without booting the host. Build arguments are an optional base URL and output directory, or the equivalent `--base-url` / `--output` named flags.

Commands

```
Command Arguments Effect     *(none)* — Dev-serve.   `build` `[baseUrl] [outputDirectory]` positional, or `--base-url` / `--output` named flags Static build; writes to `OutputOptions.OutputDirectory`, prints `BuildReport`, sets `Environment.ExitCode = 1` when the report has errors.   `diag <subcommand>` one of `info`, `toc`, `routes`, `warnings`, `translation`, `frontmatter`, `llms`, `standard-site` Read-only inspection. Runs the host headless (in-process, no socket bind), writes text to stdout, and exits. `diag --help` lists the subcommands.   `--help` / `--version` — Print usage (or the package version) and exit without serving. `-h` and `-?` are help aliases.   *anything else* — Dev-serve. An unrecognized `args[0]` is not interpreted as `build`/`diag` arguments.
```

Positional arguments

Two positional slots follow the `build` verb, in this order:

```
Slot Name Default Description     1 `baseUrl` `/` The URL sub-path the site will be served from; materialized as `OutputOptions.BaseUrl` (a `UrlPath`).   2 `outputDirectory` `output` The filesystem directory to write the generated site into; materialized as `OutputOptions.OutputDirectory` (a `FilePath`).
```

Named flags

```
Flag Value form Maps to     `--base-url` `--base-url /sub` or `--base-url=/sub` `OutputOptions.BaseUrl`   `--output` `--output dist` or `--output=dist` `OutputOptions.OutputDirectory`
```

Both flags match case-insensitively and accept either the space-separated or `=`-joined value form.

Flag and positional resolution

Named flags win outright. A value given by `--base-url` or `--output` always populates its slot. The remaining positional arguments then fill any slots not already claimed by a flag, in declaration order — first the unclaimed `baseUrl`, then the unclaimed `outputDirectory`. So `build --base-url=/sub dist` puts `/sub` in `baseUrl` and the lone positional `dist` in `outputDirectory`, because the flag took the first slot and the positional flows into the next free one.

Unknown flags are silently ignored, which lets stray dev-mode arguments (`--urls`, the flags `dotnet watch` injects) pass through `build` without error.

Accepted input normalization

The base-URL value is normalized so common shell shapes resolve to a usable path:

- **Bare segment.** `--base-url my-app` is accepted and promoted to `/my-app`. A leading slash is optional, which lets POSIX shells pass the value without the slash that MSYS would otherwise rewrite.

- **Windows-path recovery.** A value shaped like `C:/Program Files/Git/my-app` — the result of Git Bash translating a leading-slash argument — triggers a warning on stderr and is recovered to its last segment (`/my-app`) so the build still produces usable links.

Environment variables

```
Variable Consumer Effect when set     `ASPNETCORE_URLS` ASP.NET Core host Standard ASP.NET
URL binding for dev-serve. Inert under `build`, which [binds no port]
(https://usepennington.net/explanation/core/dev-vs-build/).   `ASPNETCORE_ENVIRONMENT`
ASP.NET Core host No Pennington-specific effect. Dev tooling (live reload, diagnostic
overlay) gates on the run mode, not on this variable.
```

Listening port

In dev mode, Pennington uses the standard ASP.NET Core host port-binding mechanisms — `--urls` , `ASPNETCORE_URLS` , or `launchSettings.json` — and the library adds middleware and endpoints on top of whatever URL Kestrel is told to listen on. Build mode binds no port: it drives the same pipeline in process. See Dev mode and build mode share one code path for the mechanism.

Exit codes

- `0` — `build` completed without errors (`BuildReport.HasErrors == false`), or dev-serve exited cleanly.
- `1` — `build` completed but the `BuildReport` contains at least one error diagnostic or failed page (`BuildReport.HasErrors == true`). Set explicitly by `RunOrBuildAsync` after writing the report.

Example

CSHARP

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

builder.Services.AddPennington(penn =>
{
    penn.SiteTitle = "My First Pennington Site";
    penn.ContentRootPath = "Content";

    penn.AddMarkdownContent<DocFrontMatter>(md =>
    {
        md.ContentPath = "Content";
        md.BasePageUrl = "/";
    });
});

var app = builder.Build();

app.UsePennington();

await app.RunOrBuildAsync(args);
```

`dotnet run` against this host serves live; `dotnet run -- build` generates to `./output/` at base URL `/`; `dotnet run -- build /sub dist` or `dotnet run -- build --base-url=/sub --output=dist` generates to `./dist/` at base URL `/sub`.

See also

- Related reference: DI and middleware extension methods
- Related reference: Build report fields
- How-to: Build a static site
- How-to: Host under a sub-path (base URL)
- Background: Dev mode and build mode share one code path

CHAPTER 6

Diagnostics

Request-scoped diagnostics

Reference The per-request diagnostic accumulator (DiagnosticContext), the Diagnostic record, the DiagnosticSeverity enum, and the dev-mode overlay plus X-Pennington-Diagnostic header that surface them.

The scoped accumulator and record types that collect per-request warnings, errors, and info messages, plus the two transports that surface them: the `X-Pennington-Diagnostic` response header (emitted on every request, including during a static build, where it feeds the build report) and the dev-only on-page overlay. The accumulator and record types live in `Pennington.Diagnostics`; the header emission and overlay processor live in `Pennington.Infrastructure`.

`DiagnosticContext`

Per-request accumulator registered as `Scoped`. Not thread-safe; resolved via DI for the lifetime of one request.

Members

```
Member Description     `Add(Diagnostic diagnostic)` Appends a pre-constructed `Diagnostic`.   `AddError(string message, string? source = null)` Appends a `Diagnostic` with `Severity = Error`.   `AddWarning(string message, string? source = null)` Appends a `Diagnostic` with `Severity = Warning`.   `AddInfo(string message, string? source = null)` Appends a `Diagnostic` with `Severity = Info`.   `Diagnostics` Read-only view of the diagnostics accumulated so far, in insertion order.   `HasAny` `true` when at least one diagnostic has been appended.   `HasErrors` `true` when at least one appended diagnostic has `Severity = Error`.
```

`Diagnostic`

Immutable record carrying one diagnostic event. Route-agnostic — `HttpContext` supplies the route context.

Parameters

```
Name Type Default Description     `Severity` `DiagnosticSeverity` — Severity band controlling overlay color and the `HasErrors` flag.   `Message` `string` — Human-readable body rendered into the overlay panel and, percent-encoded, into the second segment of the `X-Pennington-Diagnostic` header value.   `Source` `string?` `null` Optional producer label (for example, `"XrefResolver"`); rendered as the overlay pill subtitle and, percent-encoded, appended as a third header segment when non-null.
```

`DiagnosticSeverity`

Three-value enum.

Values

```
Name Value Description     `Warning` `0` Recoverable issue (for example, an unresolved xref); contributes to the overlay warning count and the amber badge color.   `Error` `1` Failure that indicates broken content or misconfiguration; flips `HasErrors` and renders with the red badge color.   `Info` `2` Informational notice about degraded but non-broken behavior; does not contribute to the error or warning counts.
```

`X-Pennington-Diagnostic` header and dev-mode overlay

Two transports surface the accumulated diagnostics, both wired inside `UsePennington` via the response-processor pipeline. The header is emitted on every request; the overlay is dev-only.

```
Transport Type Availability Shape     Response header `X-Pennington-Diagnostic` Every request where `HasAny` is `true`, including during a static build (the build pipeline parses these headers and folds them into the [build report](#p--reference-api-build-report)) One header value per diagnostic, pipe-delimited: `Severity|Message` (or `Severity|Message|Source` when `Source` is non-null). Each segment is independently percent-encoded with `Uri.EscapeDataString` so non-ASCII values (accented locale names, content titles) survive the ASCII-only header writer — a machine parser must split on `|` then `Uri.UnescapeDataString` each segment.   On-page overlay `DiagnosticOverlayProcessor` (`Order = 30`, `IResponseProcessor`) Dev-serve requests (the host was not launched with the build verb) where status is `2xx` and content type contains `text/html` Floating badge injected before `</body>` summarizing error/warning counts; clicking expands a panel listing every diagnostic. Re-renders on the `spa:diagnostics` DOM event for SPA navigations.
```

Example

CSHARP

```
public int Order => 50;

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)
{
    if (!responseBody.Contains("rel=\"canonical\"", StringComparison.OrdinalIgnoreCase))
    {
        diagnostics.AddWarning(
            "Page is missing a <link rel=\"canonical\"> tag.",
            source: context.Request.Path);
    }
    return Task.FromResult(responseBody);
}
```

See also

- Related reference: Build report fields
- Related reference: Response processing interfaces
- How-to: Write a response processor

CHAPTER 7

Blogsite

Built-in BlogSite routes

Reference Catalog of the routes the Pennington.BlogSite package ships out of the box, keyed to the BlogSiteOptions knobs that control them.

`UseBlogSite` mounts the Razor pages that serve the homepage, archive, tag index, per-tag listing, and individual posts, plus an optional `/rss.xml` endpoint. Pages are discovered via `RazorPageContentService`; `AddBlogSite` registers `Pennington.BlogSite` as an additional routing assembly so Razor picks them up.

Entry point

`UseBlogSite` calls `MapRazorComponents<App>`, which discovers every `@page`-annotated component in `Pennington.BlogSite.dll`; when `BlogSiteOptions.EnableRss` is `true` it additionally maps a `MapGet("/rss.xml", …)` endpoint that returns the RSS feed. The `/sitemap.xml` endpoint is mounted by `UsePennington` via `SitemapService` (gated on `PenningtonOptions.MapSitemap`, which `AddBlogSite` mirrors from `BlogSiteOptions.EnableSitemap`).

Routes

Every route a BlogSite host serves, ordered by the surface they serve. Most are `@page` Razor components mapped by `UseBlogSite`, including the `/{*path:nonfile}` root catch-all; `/sitemap.xml` is the exception, mounted by `UsePennington`. `UseBlogSite` runs `UsePennington` before mapping the component endpoint, so `redirectUrl:` pages short-circuit with a redirect rather than falling through to the catch-all.

```
Path Method Option controlling it Description     `/` `GET` — (fixed) Homepage Razor page (`Home.razor`); renders `BlogSiteOptions.HeroContent`, recent posts via `BlogSummary`, and sidebar modules from `MyWork`/`Socials`/`AuthorBio`.   `/archive` `GET` — (fixed) Full archive Razor page (`Archive.razor`); renders every post in reverse chronological order through `BlogSummary`.   `/archive/page/{Page:int}` `GET` `PostsPerPage` Second `@page` directive on `Archive.razor` for paginated archive views; only renders extra pages when `PostsPerPage > 0` and there are more posts than fit on one page.   `/tags` `GET` — (fixed) Tag index Razor page (`Tags.razor`); lists every tag with post counts, backed by a registered taxonomy axis (`AddTaxonomy<BlogSiteFrontMatter, string>`). `/tags/{TagEncodedName}` `GET` — (fixed) Per-tag listing Razor page (`Tag.razor`); resolves the tag slug against the taxonomy axis and renders its posts through `BlogPostsList`. `/blog/{*fileName:nonfile}` `GET` `BlogBaseUrl` (see note) Post-rendering catch-all Razor page (`Blog.razor`); the `:nonfile` route constraint excludes paths that look like static files. Looks up the post by `{BlogBaseUrl}/{fileName}` and renders it through `BlogPost` with OpenGraph and structured-data head tags.   `/{*path:nonfile}` `GET` — (fixed) Root catch-all Razor page (`Pages.razor`); the lowest-priority route that handles any request matching no other page. Renders a content-root `404.md`, then a host-provided `NotFound` component, then the built-in localized not-found message, and marks the response 404. The build's sentinel request also lands here, so whatever it renders becomes `output/404.html`. `/rss.xml` `GET` `EnableRss` `MapGet` endpoint in `UseBlogSite` that returns the RSS feed; omitted entirely when `EnableRss` is `false`.   `/sitemap.xml` `GET` `EnableSitemap` `MapGet` endpoint mounted by `UsePennington` (not `UseBlogSite`); `AddBlogSite` forwards `EnableSitemap` into `PenningtonOptions.MapSitemap` to gate it.
```

The `@page` directives on `Tags.razor`, `Tag.razor`, and `Blog.razor` are fixed string literals. `/tags` and `/tags/{slug}` are backed by a registered taxonomy axis (`AddTaxonomy<BlogSiteFrontMatter, string>(BaseUrl = "/tags")`), which supplies the term data and discovers the per-tag routes for the static build while the `@page` components render them with full site chrome. `BlogBaseUrl` only affects the post URLs `Blog.razor` resolves; the page route stays at `/blog/{*fileName:nonfile}` unless replacement Razor pages are supplied via `AdditionalRoutingAssemblies`.

Option-to-route matrix

`BlogSiteOptions` knobs that affect route registration or URL resolution, one row per option.

```
Option Default Routes it affects Effect     `BlogBaseUrl` `"/blog"` `/blog/{*fileName:nonfile}` The URL prefix `Blog.razor` strips before resolving a post; must match the `/blog/...` literal in the page route (see the fixed-string-literal note above). `EnableRss` `true` `/rss.xml` Gates the `MapGet("/rss.xml", …)` call in `UseBlogSite`; when `false` the endpoint is not registered and the static crawler does not emit `rss.xml`. `PostsPerPage` `10` `/archive/page/{Page:int}` Page size for the paginated archive routes. Set to `0` to disable pagination — all posts then render on the first page. `EnableSitemap` `true` `/sitemap.xml` (from `UsePennington`) Forwards into `PenningtonOptions.MapSitemap`; when `false`, `UsePennington` skips the `/sitemap.xml` `MapGet` and the static crawler omits it from the build output.
```

Example

A BlogSite host that mounts every route above via a single `UseBlogSite` call.

CSHARP

```
using Pennington.BlogSite;

var builder = WebApplication.CreateBuilder(args);

// Swap the bare `AddPennington` host for the BlogSite template. `AddBlogSite`
// wires the full blog experience on top of Pennington core — a Blazor
// layout with a home page that lists recent posts, an /archive page,
// /blog/<slug> post pages, /tags and /tags/<name> listings, and an
// /rss.xml feed — all driven from `BlogSiteOptions`.
builder.Services.AddBlogSite(() => new BlogSiteOptions
{
    SiteTitle = "Scaffold Blog",
    SiteDescription = "A minimal BlogSite scaffold showing AddBlogSite, UseBlogSite, and
RunBlogSiteAsync.",
    CanonicalBaseUrl = "https://example.com",

    // BlogSite defaults put posts under `{ContentRootPath}/{BlogContentPath}`
    // (Content/Blog) and serves them at `BlogBaseUrl` (/blog); tag listings live
    // at /tags. Overriding the defaults is as simple as setting the matching
    // property — shown here with the defaults for clarity.
    ContentRootPath = "Content",
    BlogContentPath = "Blog",
    BlogBaseUrl = "/blog",

    // Author identity feeds into the RSS channel, JSON-LD article markup,
    // and any post that omits its own `author:` front-matter value.
    AuthorName = "Author Name",
    AuthorBio = "Writing about software, tools, and the occasional side project.",
});

var app = builder.Build();

// `UseBlogSite` mounts antiforgery, static files, Razor component routing
// (Home/Archive/Blog/Tag/Tags live inside Pennington.BlogSite.dll), the
// MonorailCSS `/styles.css` endpoint, and the core Pennington middleware —
// all in the right order. When `EnableRss` is true (the default) it also
// maps `/rss.xml` so the static crawler picks the feed up.
app.UseBlogSite();

// `RunBlogSiteAsync` delegates to `RunOrBuildAsync`, so `dotnet run` serves the
// blog live and `dotnet run -- build <baseUrl> <outputDir>` generates static
// HTML. Both positional args are optional (defaults: `/` and `output`).
await app.RunBlogSiteAsync(args);
```

See also

- Reference: BlogSiteOptions
- Reference: Built-in SocialIcons RenderFragments
- How-to: Customize DocSite layouts and components
- How-to: Configure the BlogSite homepage

Built-in SocialIcons RenderFragments

Reference The four static RenderFragment fields on Pennington.BlogSite.Components.SocialIcons and the syntax for plugging them into SocialLink.Icon.

`SocialIcons` is a Razor component with no render body whose public API is four `public static readonly RenderFragment` fields, each a self-contained inline SVG for use as the `Icon` property on a `SocialLink`. It lives in namespace `Pennington.BlogSite.Components` and is consumed by `BlogSiteOptions.Socials` via the `SocialLink` record in namespace `Pennington.BlogSite`.

Declaration

RAZOR

```
@code {
    public static readonly RenderFragment GithubIcon = @<svg
xmlns="http://www.w3.org/2000/svg" viewBox="0 0 24 24" stroke="currentColor" fill="none">
        <path d="M6.51734 17.1132C6.91177 17.6905 8.10883 18.9228 9.74168 19.2333M9.86428
22C8.83582 21.8306 2 19.6057 2 12.0926C2 5.06329 8.0019 2 12.0008 2C15.9996 2 22 5.06329 22
12.0926C22 19.6057 15.1642 21.8306 14.1357 22C14.1357 22 13.9267 18.5826 14.0487
17.9969C14.1706 17.4113 13.7552 16.4688 13.7552 16.4688C14.7262 16.1055 16.2043 15.5847
16.7001 14.1874C17.0848 13.1032 17.3268 11.5288 16.2508 10.0489C16.2508 10.0489 16.5318
7.65809 15.9996 7.56548C15.4675 7.47287 13.8998 8.51192 13.8998 8.51192C13.4432 8.38248
12.4243 8.13476 12.0018 8.17939C11.5792 8.13476 10.5568 8.38248 10.1002 8.51192C10.1002
8.51192 8.53249 7.47287 8.00036 7.56548C7.46823 7.65809 7.74917 10.0489 7.74917
10.0489C6.67316 11.5288 6.91516 13.1032 7.2999 14.1874C7.79575 15.5847 9.27384 16.1055
10.2448 16.4688C10.2448 16.4688 9.82944 17.4113 9.95135 17.9969C10.0733 18.5826 9.86428 22
9.86428 22Z" stroke-width="1.5" stroke-linecap="round" stroke-linejoin="round"/>
    </svg>;

    public static readonly RenderFragment LinkedInIcon = @<svg
xmlns="http://www.w3.org/2000/svg" viewBox="0 0 24 24" stroke="currentColor" fill="none">
        <path d="M7 10V17" stroke-width="1.5" stroke-linecap="round" stroke-
linejoin="round"></path>
        <path d="M11 13V17M11 13C11 11.3431 12.3431 10 14 10C15.6569 10 17 11.3431 17
13V17M11 13V10" stroke-width="1.5" stroke-linecap="round" stroke-linejoin="round"></path>
        <path d="M7.00801 7L6.99902 7" stroke-width="1.5" stroke-linecap="round" stroke-
linejoin="round"></path>
        <path d="M2.5 12C2.5 7.52166 2.5 5.28249 3.89124 3.89124C5.28249 2.5 7.52166 2.5 12
2.5C16.4783 2.5 18.7175 2.5 20.1088 3.89124C21.5 5.28249 21.5 7.52166 21.5 12C21.5 16.4783
21.5 18.7175 20.1088 20.1088C18.7175 21.5 16.4783 21.5 12 21.5C7.52166 21.5 5.28249 21.5
3.89124 20.1088C2.5 18.7175 2.5 16.4783 2.5 12Z" stroke-width="1.5" stroke-linejoin="round">
</path>
    </svg>;

    public static readonly RenderFragment BlueskyIcon = @<svg
xmlns="http://www.w3.org/2000/svg" viewBox="0 0 24 24" stroke="currentColor" fill="none">
        <path d="M12 11.4963C11.8936 11.2963 7.45492 3 3.50417 3C1.33647 3 2.00456 8 2.50443
10.5C2.70653 11.5108 3.50417 14.5 8.003 14C8.003 14 4.00404 14.5 4.00404 17C4.00404 18.5
6.50339 21 8.50287 21C10.4606 21 11.9391 16.6859 12 16.5058C12.0609 16.6859 13.5394 21
15.4971 21C17.4966 21 19.996 18.5 19.996 17C19.996 14.5 15.997 14 15.997 14C20.4958 14.5
21.2935 11.5108 21.4956 10.5C21.9954 8 22.6635 3 20.4958 3C16.5451 3 12.1064 11.2963 12
11.4963Z" stroke-width="1.5" stroke-linejoin="round"></path>
    </svg>;

    public static readonly RenderFragment MastodonIcon = @<svg
xmlns="http://www.w3.org/2000/svg" viewBox="0 0 24 24" stroke="currentColor" fill="none">
        <path d="M17 13.5V8C17 6.61929 15.8807 5.5 14.5 5.5C13.1193 5.5 12 6.61929 12 8M12
8V11.5M12 8C12 6.61929 10.8807 5.5 9.5 5.5C8.11929 5.5 7 6.61929 7 8V13.5" stroke-
width="1.5" stroke-linecap="round" stroke-linejoin="round"/>
        <path d="M13 16.9954C15.0099 16.9954 16.89 16.6876 18.4949 16.1525C20.1275 15.6081
21 13.9512 21 12.2302V7.52349C21 5.34784 19.8297 3.2779 17.7281 2.715C16.0259 2.25905
14.0744 2 12 2C9.9256 2 7.97414 2.25905 6.27189 2.715C4.17033 3.2779 3 5.34785 3
7.52349V14.4961C3 22.4937 11 21.9938 11 21.9938C13.5 21.9938 15 21 15 21V20C15 20 13.5
20.4943 11 20.4943C5.68009 20.4943 7.06011 15.9957 7.06011 15.9957C8.75781 16.627 10.8012
```

```
16.9954 13 16.9954Z"stroke-width="1.5"stroke-linecap="round"stroke-linejoin="round"/>
</svg>;}
```

There is no render body, class body, or constructor — just the four fields.

Icons

All four fragments share `viewBox="0 0 24 24"`, `stroke="currentColor"`, and `fill="none"`; color and size are inherited from the surrounding anchor or container. The available fields are:

- `GithubIcon`
- `LinkedInIcon`
- `BlueskyIcon`
- `MastodonIcon`

`SocialLink.Icon` shape

CSHARP

```
public record SocialLink(RenderFragment Icon, string Url);
```

The `Icon` property accepts the static `RenderFragment` directly (`SocialIcons.GithubIcon`), not the Razor component tag form. For consumer wiring see Populate the blog homepage.

See also

- Related reference: BlogSiteOptions (see the `SocialLink` helper record and the `Socials` property)
- Related reference: Built-in BlogSite routes
- How-to: Configure the BlogSite homepage
- How-to: Customize DocSite layouts and components

CHAPTER 8

API

Pennington.ApiMetadata.AccessFilter

Pennington.ApiMetadata.AccessFilter

CSHARP

```
namespace Pennington.ApiMetadata;

/// Accessibility filter applied when enumerating members.
public enum AccessFilter
{
    /// Include only protected members.
    public static const AccessFilter Protected
;

    /// Include only public members.
    public static const AccessFilter Public
;

    /// Include public and protected members.
    public static const AccessFilter PublicAndProtected
;
}
```

Pennington.ApiMetadata.ApiMember

Pennington.ApiMetadata.ApiMember

CSHARP

```
namespace Pennington.ApiMetadata;

/// Single member (property, field, method, constructor, or event) of a documented type,
with all display strings pre-formatted by the backend provider.
public record ApiMember
{
    /// Single member (property, field, method, constructor, or event) of a documented type,
with all display strings pre-formatted by the backend provider.
    public ApiMember(string Uid, string Name, MemberKind Kind, string TypeDisplay, string
DefaultValue, bool IsRequired, bool HasInheritDocDirective, ParsedXmlDoc Xmldoc, string
SignatureHtml, ImmutableArray<ApiParameter> Parameters, string ReturnTypeDisplay, string
InheritedFromUid = null, string InheritedFromName = null)
;

    /// Formatted default value literal, or null when none is declared.
    public string DefaultValue { get; set; }

    /// Whether the source xmldoc carried an <inheritdoc/> directive; consumers use this to
suppress "missing summary" diagnostics when inheritance didn't resolve.
    public bool HasInheritDocDirective { get; set; }

    /// When the member is inherited, the short name of the declaring type for grouping
headers (e.g. IContentEmitter). null for directly-declared members.
    public string InheritedFromName { get; set; }

    /// When the member is inherited from a base interface (rather than declared on the
queried type), the xmldocid of the declaring type. null for directly-declared members.
    public string InheritedFromUid { get; set; }

    /// Whether the member carries the required modifier.
    public bool IsRequired { get; set; }

    /// The kind of member this record represents.
    public MemberKind Kind { get; set; }

    /// Display name of the member (for methods, includes type-parameter list; for
constructors, the containing type name).
    public string Name { get; set; }

    /// Formatted parameter list for methods/constructors; empty for
properties/fields/events.
    public ImmutableArray<ApiParameter> Parameters { get; set; }

    /// Formatted return type for methods that return a value; null otherwise.
    public string ReturnTypeDisplay { get; set; }

    /// Pre-highlighted declaration HTML ready to inject with @((MarkupString)…), or null
when no declaration signature is available.
    public string SignatureHtml { get; set; }

    /// Human-readable type signature (return type for methods, declared type for
```

```
    /// Canonical xmldocid (e.g. M:Namespace.Type.Method(System.Int32)).    public string
Uid { get; set; }      /// Parsed xmldoc for the member.    public ParsedXmlDoc Xmldoc {
get; set; }  }
```

Pennington.ApiMetadata.ApiParameter

Pennington.ApiMetadata.ApiParameter

CSHARP

```
namespace Pennington.ApiMetadata;

/// One parameter of a method or constructor, formatted for display.
public record ApiParameter
{
    /// One parameter of a method or constructor, formatted for display.
    public ApiParameter(string Name, string TypeDisplay, ImmutableArray<XmlDocNode>
Description)
;

    /// Parsed xmldoc nodes from the matching <param> element.
    public ImmutableArray<XmlDocNode> Description { get; set; }

    /// Parameter name as declared.
    public string Name { get; set; }

    /// Fully-formatted type display including ref/out/in prefixes and optional markers.
    public string TypeDisplay { get; set; }

}
```

Pennington.ApiMetadata.ApiTypeDetail

Pennington.ApiMetadata.ApiTypeDetail

CSHARP

```
namespace Pennington.ApiMetadata;

/// Full detail for a documented type, including parsed xmldoc and inheritance data, but not its members (those stream from GetMembersAsync).
public record ApiTypeDetail
{
    /// Full detail for a documented type, including parsed xmldoc and inheritance data, but not its members (those stream from GetMembersAsync).
    public ApiTypeDetail(ApiTypeSummary Summary, ParsedXmlDoc Xmldoc, string SignatureHtml, ImmutableArray<string> Inheritance, ImmutableArray<string> Implements)
;

    /// Implemented-interface uids.
    public ImmutableArray<string> Implements { get; set; }

    /// Base-type uids, most-derived first. Empty for interfaces and object.
    public ImmutableArray<string> Inheritance { get; set; }

    /// Pre-highlighted declaration HTML, or null when not available.
    public string SignatureHtml { get; set; }

    /// Header describing the type.
    public ApiTypeSummary Summary { get; set; }

    /// Parsed xmldoc for the type itself.
    public ParsedXmlDoc Xmldoc { get; set; }

}
```

Pennington.ApiMetadata.ApiTypeKind

Pennington.ApiMetadata.ApiTypeKind

CSHARP

```
namespace Pennington.ApiMetadata;

/// Kind of a documented type.
public enum ApiTypeKind
{
    /// A reference type declared with class.
    public static const ApiTypeKind Class
;

    /// A delegate declaration.
    public static const ApiTypeKind Delegate
;

    /// An enum declaration.
    public static const ApiTypeKind Enum
;

    /// An interface declaration.
    public static const ApiTypeKind Interface
;

    /// A record or record struct declaration.
    public static const ApiTypeKind Record
;

    /// A value type declared with struct.
    public static const ApiTypeKind Struct
;
}
```

Pennington.ApiMetadata.ApiTypeSummary

Pennington.ApiMetadata.ApiTypeSummary

CSHARP

```
namespace Pennington.ApiMetadata;

/// Lightweight header describing a documented type, used for listings, slug disambiguation, and cross-link display names.
public record ApiTypeSummary
{
    /// Lightweight header describing a documented type, used for listings, slug disambiguation, and cross-link display names.
    public ApiTypeSummary(string Uid, string Name, string Namespace, string Assembly, ApiTypeKind Kind, string Summary)
;

    /// Declaring assembly name without extension.
    public string Assembly { get; set; }

    /// Fully-qualified type name (namespace + dot + type name).
    public string FullTypeName { get; }

    /// Category of the type.
    public ApiTypeKind Kind { get; set; }

    /// Short type name without namespace (e.g. ContentPipeline).
    public string Name { get; set; }

    /// Fully-qualified containing namespace, empty for the global namespace.
    public string Namespace { get; set; }

    /// First-sentence plain-text summary, or null when no xmldoc summary is available.
    public string Summary { get; set; }

    /// Canonical xmldocid (e.g. T:Namespace.TypeName). Normalized to xmldocid form regardless of source backend.
    public string Uid { get; set; }

}
```

Pennington.ApiMetadata.CodeBlockNode

Pennington.ApiMetadata.CodeBlockNode

CSHARP

```
namespace Pennington.ApiMetadata;

/// Fenced code block from a <code> element.
public record CodeBlockNode
{
    /// Fenced code block from a <code> element.
    public CodeBlockNode(string Language, string Text)
;

    /// Language identifier used by the code fence (e.g. csharp).
    public string Language { get; set; }

    /// The code content with leading indentation stripped.
    public string Text { get; set; }

}
```

Pennington.ApiMetadata.CrefNode

Pennington.ApiMetadata.CrefNode

CSHARP

```
namespace Pennington.ApiMetadata;

/// Cross-reference from a <see cref="..."/> or <see href="..."/> element.
public record CrefNode
{
    /// The raw cref id or href value.
    public string CrefId { get; set; }

    /// Cross-reference from a <see cref="..."/> or <see href="..."/> element.
    public CrefNode(string CrefId, string DisplayText)
;

    /// Optional display text from the element body.
    public string DisplayText { get; set; }

}
```

Pennington.ApiMetadata.ExtensionMethodEntry

Pennington.ApiMetadata.ExtensionMethodEntry

CSHARP

```
namespace Pennington.ApiMetadata;

/// One public static extension method discovered in a workspace assembly, projected for reference-doc rendering.
public record ExtensionMethodEntry
{
    /// One public static extension method discovered in a workspace assembly, projected for reference-doc rendering.
    public ExtensionMethodEntry(string Name, string Signature, string Package, string Uid, string ReceiverTypeName, ParsedXmlDoc Xmldoc)
;

    /// Short method name (no parameter list, no enclosing type).
    public string Name { get; set; }

    /// Owning assembly name, used as the package label on the rendered page.
    public string Package { get; set; }

    /// Unqualified short name of the first (receiver) parameter's type, used as the grouping key.
    public string ReceiverTypeName { get; set; }

    /// Full C# signature including return type and parameter list.
    public string Signature { get; set; }

    /// Canonical xmldocid (M:...) of the method.
    public string Uid { get; set; }

    /// Parsed xmldoc for the method, with summary/remarks/returns/etc.
    public ParsedXmlDoc Xmldoc { get; set; }

}
```

Pennington.ApiMetadata.IApiMetadataProvider

Pennington.ApiMetadata.IApiMetadataProvider

CSHARP

```
namespace Pennington.ApiMetadata;

/// Backend-neutral source of API documentation metadata. Implementations adapt compiled assemblies, or other metadata sources, to a single contract consumed by the API reference UI.
public interface IApiMetadataProvider
{
    /// Returns public extension methods whose first parameter's receiver type has the unqualified name receiverTypeName. Backends that cannot enumerate extensions (e.g. DocFx YAML) return an empty array.
    public Task<ImmutableArray<ExtensionMethodEntry>> GetExtensionMethodsForAsync(string receiverTypeName)
;

    /// Returns a standalone ApiMember for the method/property/field/event identified by uid, or null when the uid is unknown or resolves to a type. Used by components that render a single member (e.g. <ApiParameterTable>).
    public Task<ApiMember> GetMemberAsync(string uid)
;

    /// Returns members of the type identified by typeUid matching the filters.
    public Task<ImmutableArray<ApiMember>> GetMembersAsync(string typeUid, MemberKind kind, AccessFilter access, MemberOrder order)
;

    /// Returns full detail for the type identified by uid, or null when the type is not known to this provider.
    public Task<ApiTypeDetail> GetTypeAsync(string uid)
;

    /// Returns every documented type the provider knows about, sorted by FullTypeName.
    public Task<ImmutableArray<ApiTypeSummary>> GetTypesAsync()
;

    /// Returns parsed xmldoc for the type or member identified by uid, or Empty when the uid is unknown. Used by inline components like <ApiSummary> that target arbitrary symbols.
    public Task<ParsedXmlDoc> GetXmldocAsync(string uid)
;
}
```

Pennington.ApiMetadata.InlineCodeNode

Pennington.ApiMetadata.InlineCodeNode

CSHARP

```
namespace Pennington.ApiMetadata;

/// Inline code span from a <c> element.
public record InlineCodeNode
{
    /// Inline code span from a <c> element.
    public InlineCodeNode(string Text)
;

    /// The code text.
    public string Text { get; set; }

}
```

Pennington.ApiMetadata.IXmlDocHtmlRenderer

Pennington.ApiMetadata.IXmlDocHtmlRenderer

CSHARP

```
namespace Pennington.ApiMetadata;

/// Renders ParsedXmlDoc nodes to HTML. Inline form avoids block wrapping for use in table cells.
public interface IXmlDocHtmlRenderer
{
    /// Render as block HTML — wraps bare text in <p>, promotes <para> to paragraphs, etc.
    public string RenderHtml(IEnumerable<XmlDocNode> nodes)
;

    /// Render as inline HTML — no paragraph wrapping. Use inside <td> or single-sentence descriptions.
    public string RenderInlineHtml(IEnumerable<XmlDocNode> nodes)
;
}
```

Pennington.ApiMetadata.IXmlDocParser

Pennington.ApiMetadata.IXmlDocParser

CSHARP

```
namespace Pennington.ApiMetadata;

/// Parses raw xmldoc XML (the string returned by ISymbol.GetDocumentationCommentXml() or the summary/remarks text in DocFx ManagedReference YAML) into a structured ParsedXmlDoc tree.
public interface IXmlDocParser
{
    /// Parses the given xmldoc XML string and returns a structured tree, or Empty when the input is null, empty, or malformed.
    public ParsedXmlDoc Parse(string xmlDocumentation)
;
}
```

Pennington.ApiMetadata.ListNode

Pennington.ApiMetadata.ListNode

CSHARP

```
namespace Pennington.ApiMetadata;

/// List block from a <list> element.
public record ListNode
{
    /// Items contained in the list.
    public ImmutableArray<XmlDocListItem> Items { get; set; }

    /// List type attribute value (e.g. bullet, number, table).
    public string Kind { get; set; }

    /// List block from a <list> element.
    public ListNode(string Kind, ImmutableArray<XmlDocListItem> Items)
;
}
```

Pennington.ApiMetadata.MemberKind

Pennington.ApiMetadata.MemberKind

CSHARP

```
namespace Pennington.ApiMetadata;

/// Kind of type member to include when enumerating members for documentation rendering.
public enum MemberKind
{
    /// All kinds of members.
    public static const MemberKind All
;

    /// Constructors declared on the type.
    public static const MemberKind Constructors
;

    /// Events declared on the type.
    public static const MemberKind Events
;

    /// Fields declared on the type.
    public static const MemberKind Fields
;

    /// Methods declared on the type (excluding constructors).
    public static const MemberKind Methods
;

    /// Properties declared on the type.
    public static const MemberKind Properties
;

    /// Cases of a discriminated union (the case types it wraps).
    public static const MemberKind UnionCases
;
}
```

Pennington.ApiMetadata.MemberOrder

Pennington.ApiMetadata.MemberOrder

CSHARP

```
namespace Pennington.ApiMetadata;

/// Ordering used when rendering a list of members.
public enum MemberOrder
{
    /// Sort members alphabetically by name.
    public static const MemberOrder Alphabetical
;

    /// Preserve the declaration order from source.
    public static const MemberOrder Declaration
;
}
```

Pennington.ApiMetadata.ParamRefNode

Pennington.ApiMetadata.ParamRefNode

CSHARP

```
namespace Pennington.ApiMetadata;

/// Parameter reference from a <paramref name="..."/> element.
public record ParamRefNode
{
    /// Name of the referenced parameter.
    public string ParamName { get; set; }

    /// Parameter reference from a <paramref name="..."/> element.
    public ParamRefNode(string ParamName)
;
}
```

Pennington.ApiMetadata.ParaNode

Pennington.ApiMetadata.ParaNode

CSHARP

```
namespace Pennington.ApiMetadata;

/// Paragraph block from a <para> element.
public record ParaNode
{
    /// Inline nodes contained within the paragraph.
    public ImmutableArray<XmlDocNode> Children { get; set; }

    /// Paragraph block from a <para> element.
    public ParaNode(ImmutableArray<XmlDocNode> Children)
;
}
```

Pennington.ApiMetadata.ParsedXmlDoc

Pennington.ApiMetadata.ParsedXmlDoc

CSHARP

```
namespace Pennington.ApiMetadata;

/// Structured representation of a parsed xmldoc comment split into its standard sections.
public record ParsedXmlDoc
{
    /// Shared empty instance used when no xmldoc is available.
    public static ParsedXmlDoc Empty { get; }

    /// Nodes from the <example> element.
    public ImmutableArray<XmlDocNode> Example { get; set; }

    /// True when Example contains any nodes.
    public bool HasExample { get; }

    /// True when Remarks contains any nodes.
    public bool HasRemarks { get; }

    /// True when Returns contains any nodes.
    public bool HasReturns { get; }

    /// True when Summary contains any nodes.
    public bool HasSummary { get; }

    /// Per-parameter nodes keyed by parameter name, from <param> elements.
    public ImmutableDictionary<string, ImmutableArray<XmlDocNode>> Params { get; set; }

    /// Structured representation of a parsed xmldoc comment split into its standard
sections.
    public ParsedXmlDoc(ImmutableArray<XmlDocNode> Summary, ImmutableArray<XmlDocNode>
Remarks, ImmutableDictionary<string, ImmutableArray<XmlDocNode>> Params,
ImmutableDictionary<string, ImmutableArray<XmlDocNode>> TypeParams,
ImmutableArray<XmlDocNode> Returns, ImmutableArray<XmlDocNode> Example,
ImmutableArray<string> SeeAlso)
;

    /// Nodes from the <remarks> element.
    public ImmutableArray<XmlDocNode> Remarks { get; set; }

    /// Nodes from the <returns> element.
    public ImmutableArray<XmlDocNode> Returns { get; set; }

    /// Cref values collected from <seealso> elements.
    public ImmutableArray<string> SeeAlso { get; set; }

    /// Nodes from the <summary> element.
    public ImmutableArray<XmlDocNode> Summary { get; set; }

    /// Per-type-parameter nodes keyed by name, from <typeparam> elements.
    public ImmutableDictionary<string, ImmutableArray<XmlDocNode>> TypeParams { get; set; }

}
```

Pennington.ApiMetadata.Reflection.CompiledAssemblyApiMetadataExt

Pennington.ApiMetadata.Reflection.CompiledAssemblyApiMetadataExt

CSHARP

```
namespace Pennington.ApiMetadata.Reflection;

/// DI extension that registers a reflection-backed IApiMetadataProvider.
public class CompiledAssemblyApiMetadataExtensions
{
    /// Registers CompiledAssemblyApiMetadataProvider as a keyed IApiMetadataProvider under name. Call once per library you want to document — each call builds its own MetadataLoadContext and xmldoc index scoped to the supplied AssemblyDirectories. The shared IXmlDocParser / IXmlDocHtmlRenderer services are registered once (idempotent).
    public static IServiceCollection AddApiMetadataFromCompiledAssembly(IServiceCollection services, string name, Action<CompiledAssemblyApiOptions> configure)
;

    /// Convenience overload: registers under the "default" name for sites documenting a single library.
    public static IServiceCollection AddApiMetadataFromCompiledAssembly(IServiceCollection services, Action<CompiledAssemblyApiOptions> configure)
;
}
```

Pennington.ApiMetadata.Reflection.CompiledAssemblyApiMetadataPro

Pennington.ApiMetadata.Reflection.CompiledAssemblyApiMetadataPro

CSHARP

```
namespace Pennington.ApiMetadata.Reflection;

/// Reflection-backed IApiMetadataProvider. Loads configured .dll files via
MetadataLoadContext, pairs each with its companion .xml xmldoc file, and materializes the
provider DTOs. No source code and no MSBuild workspace required.
public class CompiledAssemblyApiMetadataProvider
{
    /// Initializes the provider.
    public CompiledAssemblyApiMetadataProvider(CompiledAssemblyApiOptions options,
IXmlDocParser xmlDocParser, ICodeHighlighter highlighter)
;

    /// Returns public extension methods whose first parameter's receiver type has the
unqualified name receiverTypeName. Backends that cannot enumerate extensions (e.g. DocFx
YAML) return an empty array.
    public Task<ImmutableArray<ExtensionMethodEntry>> GetExtensionMethodsForAsync(string
receiverTypeName)
;

    /// Returns a standalone ApiMember for the method/property/field/event identified by
uid, or null when the uid is unknown or resolves to a type. Used by components that render a
single member (e.g. <ApiParameterTable>).
    public Task<ApiMember> GetMemberAsync(string uid)
;

    /// Returns members of the type identified by typeUid matching the filters.
    public Task<ImmutableArray<ApiMember>> GetMembersAsync(string typeUid, MemberKind kind,
AccessFilter access, MemberOrder order)
;

    /// Returns full detail for the type identified by uid, or null when the type is not
known to this provider.
    public Task<ApiTypeDetail> GetTypeAsync(string uid)
;

    /// Returns every documented type the provider knows about, sorted by FullTypeName.
    public Task<ImmutableArray<ApiTypeSummary>> GetTypesAsync()
;

    /// Returns parsed xmldoc for the type or member identified by uid, or Empty when the
uid is unknown. Used by inline components like <ApiSummary> that target arbitrary symbols.
    public Task<ParsedXmlDoc> GetXmldocAsync(string uid)
;
}
```

Pennington.ApiMetadata.Reflection.CompiledAssemblyApiOptions

Pennington.ApiMetadata.Reflection.CompiledAssemblyApiOptions

CSHARP

```
namespace Pennington.ApiMetadata.Reflection;

/// Options for CompiledAssemblyApiMetadataProvider.
public class CompiledAssemblyApiOptions
{
    /// Directories to scan for *.dll files with matching *.xml xmldoc files. Every matched pair in each directory contributes types to the provider. Use this when every .dll in the folder is intended for documentation — the typical NuGet lib/<tfm>/ layout.
    public IList<string> AssemblyDirectories { get; }

    /// Explicit .dll paths to document. Use this when a folder contains more assemblies than you want documented (e.g. dependencies copied alongside the target for MetadataLoadContext resolution). The companion .xml file is loaded from the same directory. When both AssemblyDirectories and AssemblyFiles are set, both contribute.
    public IList<string> AssemblyFiles { get; }

}
```

Pennington.ApiMetadata.Reflection.CompiledAssemblyApiOptionsExte

Pennington.ApiMetadata.Reflection.CompiledAssemblyApiOptionsExte

CSHARP

```
namespace Pennington.ApiMetadata.Reflection;

/// Sugar for CompiledAssemblyApiOptions that resolves documented assemblies via Assembly instead of filesystem paths.
public class CompiledAssemblyApiOptionsExtensions
{
    /// Resolves assemblySimpleName in the host application's default load context (populated from the project's .deps.json and the NuGet cache), then adds the resolved .dll path to AssemblyFiles. Requires a matching <PackageReference> in the docsite project; no source reference to any type in the package is needed.
    public static CompiledAssemblyApiOptions FromPackageReference(CompiledAssemblyApiOptions options, string assemblySimpleName)
;
}
```

Pennington.ApiMetadata.TextNode

Pennington.ApiMetadata.TextNode

CSHARP

```
namespace Pennington.ApiMetadata;

/// Literal text content within an xmldoc node tree.
public record TextNode
{
    /// The text content.
    public string Text { get; set; }

    /// Literal text content within an xmldoc node tree.
    public TextNode(string Text)
;
}
```

Pennington.ApiMetadata.TypeParamRefNode

Pennington.ApiMetadata.TypeParamRefNode

CSHARP

```
namespace Pennington.ApiMetadata;

/// Type parameter reference from a <typeparamref name="..."/> element.
public record TypeParamRefNode
{
    /// Name of the referenced type parameter.
    public string ParamName { get; set; }

    /// Type parameter reference from a <typeparamref name="..."/> element.
    public TypeParamRefNode(string ParamName)
;
}
```

Pennington.ApiMetadata.UidDisplay

Pennington.ApiMetadata.UidDisplay

CSHARP

```
namespace Pennington.ApiMetadata;

/// Formatting helpers for xmldocid strings (the T:, M:, P:… prefixed uids the C# compiler emits).
public class UidDisplay
{
    /// Returns the short, unqualified display name for a uid (e.g. T:System.Collections.Generic.List`1 → List), stripping the kind prefix, any parameter list, the namespace, and generic-arity markers.
    public static string Shorten(string uid)
;
}
```

Pennington.ApiMetadata.XmlDocHtmlRenderer

Pennington.ApiMetadata.XmlDocHtmlRenderer

CSHARP

```
namespace Pennington.ApiMetadata;

/// Renders XmlDocNode trees into HTML for display in the DocSite.
public class XmlDocHtmlRenderer
{
    /// Renders the nodes as block-level HTML, wrapping loose inline content in <p> and emitting <pre>/<ul>/<ol> for code blocks and lists.
    public string RenderHtml(IEnumerable<XmlDocNode> nodes)
;

    /// Renders the nodes as inline HTML without wrapping paragraphs, suitable for embedding inside an existing block element.
    public string RenderInlineHtml(IEnumerable<XmlDocNode> nodes)
;
}
```

Pennington.ApiMetadata.XmlDocListItem

Pennington.ApiMetadata.XmlDocListItem

CSHARP

```
namespace Pennington.ApiMetadata;

/// Single item within a ListNode.
public record XmlDocListItem
{
    /// Nodes from the <description> element (or the item body when no description element is present).
    public ImmutableArray<XmlDocNode> Description { get; set; }

    /// Nodes from the <term> element.
    public ImmutableArray<XmlDocNode> Term { get; set; }

    /// Single item within a ListNode.
    public XmlDocListItem(ImmutableArray<XmlDocNode> Term, ImmutableArray<XmlDocNode> Description)
;
}
```

Pennington.ApiMetadata.XmlDocNode

Pennington.ApiMetadata.XmlDocNode

CSHARP

```
namespace Pennington.ApiMetadata;

/// Discriminated union of node kinds that make up a parsed xmldoc tree.
public struct XmlDocNode
{
    /// Fenced code block from a <code> element.
    public record CodeBlockNode(string Language, string Text) : object,
IEquatable<CodeBlockNode>

    /// Cross-reference from a <see cref="..."/> or <see href="..."/> element.
    public record CrefNode(string CrefId, string DisplayText) : object, IEquatable<CrefNode>

    /// Inline code span from a <c> element.
    public record InlineCodeNode(string Text) : object, IEquatable<InlineCodeNode>

    /// List block from a <list> element.
    public record ListNode(string Kind, ImmutableArray<XmlDocListItem> Items) : object,
IEquatable<ListNode>

    /// Parameter reference from a <paramref name="..."/> element.
    public record ParamRefNode(string ParamName) : object, IEquatable<ParamRefNode>

    /// Paragraph block from a <para> element.
    public record ParaNode(ImmutableArray<XmlDocNode> Children) : object,
IEquatable<ParaNode>

    /// Literal text content within an xmldoc node tree.
    public record TextNode(string Text) : object, IEquatable<TextNode>

    /// Type parameter reference from a <typeparamref name="..."/> element.
    public record TypeParamRefNode(string ParamName) : object, IEquatable<TypeParamRefNode>

    /// Wrapped case instance; inspect via pattern matching on the case types.
    public object Value { get; }

    /// Wraps a TextNode.
    public XmlDocNode(TextNode value)
;

    /// Wraps an InlineCodeNode.
    public XmlDocNode(InlineCodeNode value)
;

    /// Wraps a CodeBlockNode.
    public XmlDocNode(CodeBlockNode value)
;

    /// Wraps a ParaNode.
    public XmlDocNode(ParaNode value)
;
```

```
      /// Wraps a ParamRefNode.    public XmlDocNode(ParamRefNode value);      /// Wraps a TypeParamRefNode.    public XmlDocNode(TypeParamRefNode value);      /// Wraps a ListNode.    public XmlDocNode(ListNode value);}
```

Pennington.ApiMetadata.XmlDocParser

Pennington.ApiMetadata.XmlDocParser

CSHARP

```
namespace Pennington.ApiMetadata;

/// Default IXmlDocParser implementation that parses xmldoc XML via XDocument.
public class XmlDocParser
{
    /// Parses the given xmldoc XML string and returns a structured tree, or Empty when the input is null, empty, or malformed.
    public ParsedXmlDoc Parse(string xmlDocumentation)
;
}
```

Pennington.Artifacts.ArtifactClaim

Pennington.Artifacts.ArtifactClaim

CSHARP

```
namespace Pennington.Artifacts;

/// A declared URL territory owned by an IArtifactContentService. Claims are derived from
options at construction — cheap, startup-stable, and consulted on every request by the
artifact router — so they must never trigger discovery, the site projection, or any other
lazy corpus work. The owning service's resolver stays authoritative: a claim match with a
null resolve falls through to content routing.
public record ArtifactClaim
{
    /// A declared URL territory owned by an IArtifactContentService. Claims are derived
from options at construction — cheap, startup-stable, and consulted on every request by the
artifact router — so they must never trigger discovery, the site projection, or any other
lazy corpus work. The owning service's resolver stays authoritative: a claim match with a
null resolve falls through to content routing.
    public ArtifactClaim(string Owner, ArtifactClaimShape Shape, string Description)
;

    /// Human label shown in diag routes and namespace-conflict warnings.
    public string Description { get; set; }

    /// True when path (leading-slash request path) falls inside this territory.
    public bool Matches(string path)
;

    /// Short stable name of the owning feature (e.g. search, book), shown in diag output
and conflict warnings.
    public string Owner { get; set; }

    /// Glob-style rendering of the territory for diagnostics (e.g. /search/**.json,
**/llms.txt).
    public string Pattern { get; }

    /// The territory this claim reserves.
    public ArtifactClaimShape Shape { get; set; }

}
```

Pennington.Artifacts.ArtifactClaimShape

Pennington.Artifacts.ArtifactClaimShape

CSHARP

```
namespace Pennington.Artifacts;

/// Union of the URL-territory shapes an ArtifactClaim can declare.
public struct ArtifactClaimShape
{
    /// Wraps a PrefixClaim.
    public ArtifactClaimShape(PrefixClaim value)
;

    /// Wraps a SuffixClaim.
    public ArtifactClaimShape(SuffixClaim value)
;

    /// Wraps an ExactClaim.
    public ArtifactClaimShape(ExactClaim value)
;

    /// Matches exactly one request path.
    public record ExactClaim(UrlPath Path) : object, IEquatable<ExactClaim>

    /// Matches any request path under Prefix, optionally narrowed to paths ending with
Suffix (e.g. /search/ + .json).
    public record PrefixClaim(UrlPath Prefix, string Suffix = null) : object,
IEquatable<PrefixClaim>

    /// Matches any request path ending with Suffix at any depth — the mid-path catch-all no
endpoint route template can express (e.g. /reference/api/llms.txt via /llms.txt). A path
that is nothing but the suffix does not match, so a root file like /llms.txt stays claimable
by an ExactClaim.
    public record SuffixClaim(string Suffix) : object, IEquatable<SuffixClaim>

    /// Wrapped case instance; inspect via pattern matching on the case types.
    public object Value { get; }

}
```

Pennington.Artifacts.ArtifactContent

Pennington.Artifacts.ArtifactContent

CSHARP

```
namespace Pennington.Artifacts;

/// The bytes and content type for one resolved artifact request.
public record ArtifactContent
{
    /// The bytes and content type for one resolved artifact request.
    public ArtifactContent(byte[] Bytes, string ContentType)
;

    /// Response body, served in dev and written verbatim by the static build.
    public byte[] Bytes { get; set; }

    /// Full content-type header value (e.g. application/json; charset=utf-8).
    public string ContentType { get; set; }

}
```

Pennington.Artifacts.ExactClaim

Pennington.Artifacts.ExactClaim

CSHARP

```
namespace Pennington.Artifacts;

/// Matches exactly one request path.
public record ExactClaim
{
    /// Matches exactly one request path.
    public ExactClaim(UrlPath Path)
;

    /// The leading-slash path (e.g. /.well-known/atproto-did).
    public UrlPath Path { get; set; }

}
```

Pennington.Artifacts.IArtifactContentService

Pennington.Artifacts.IArtifactContentService

CSHARP

```
namespace Pennington.Artifacts;

/// The corpus-derived artifact tier: a service whose URLs and bytes derive from the
rendered site (via ISiteProjection) or from configuration — search shards, llms.txt files,
book PDFs, well-known verification files. Registered ONLY under this interface, never as
IContentService, so request-path discovery walkers (PageResolver), the projection's own
input set, sitemap, and the record registry structurally cannot trigger its potentially
expensive discovery. One byte path serves both surfaces: ResolveAsync answers dev requests
through the artifact router and produces the static build's output for every route
DiscoverAsync enumerates — dev/build parity by construction. Routes that should exist only
in dev (e.g. live book previews) are resolvable via Claims without being enumerated.
public interface IArtifactContentService
{
    /// URL territories this service serves. Options-derived and consulted on every request
— must be cheap and must not trigger discovery, the projection, or any lazy corpus work.
    public ImmutableList<ArtifactClaim> Claims { get; }

    /// Enumerates every artifact route the static build should write, as GeneratedSource
items. May consume the projection — the build invokes this outside any request, after the
page crawl has primed the render cache. Never called on the request path.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Returns the bytes for relativePath (no leading slash, e.g. search/en/index.json), or
null to decline so the request falls through to content routing. May materialize the
projection, build an index, or run Chromium on demand.
    public Task<ArtifactContent> ResolveAsync(string relativePath, CancellationToken
cancellationToken)
;
}
```

Pennington.Artifacts.PrefixClaim

Pennington.Artifacts.PrefixClaim

CSHARP

```
namespace Pennington.Artifacts;

/// Matches any request path under Prefix, optionally narrowed to paths ending with Suffix (e.g. /search/ + .json).
public record PrefixClaim
{
    /// Leading-slash path prefix, including its trailing slash (e.g. /pdf/).
    public UrlPath Prefix { get; set; }

    /// Matches any request path under Prefix, optionally narrowed to paths ending with Suffix (e.g. /search/ + .json).
    public PrefixClaim(UrlPath Prefix, string Suffix = null)
;

    /// Optional required path ending (e.g. .pdf); null claims the whole prefix.
    public string Suffix { get; set; }

}
```

Pennington.Artifacts.SuffixClaim

Pennington.Artifacts.SuffixClaim

CSHARP

```
namespace Pennington.Artifacts;

/// Matches any request path ending with Suffix at any depth — the mid-path catch-all no endpoint route template can express (e.g. /reference/api/llms.txt via /llms.txt). A path that is nothing but the suffix does not match, so a root file like /llms.txt stays claimable by an ExactClaim.
public record SuffixClaim
{
    /// Required path ending, including its leading slash (e.g. /llms.txt).
    public string Suffix { get; set; }

    /// Matches any request path ending with Suffix at any depth — the mid-path catch-all no endpoint route template can express (e.g. /reference/api/llms.txt via /llms.txt). A path that is nothing but the suffix does not match, so a root file like /llms.txt stays claimable by an ExactClaim.
    public SuffixClaim(string Suffix)
;
}
```

Pennington.BlogSite.BlogSiteFrontMatter

Pennington.BlogSite.BlogSiteFrontMatter

CSHARP

```
namespace Pennington.BlogSite;

/// Front matter bound by AddBlogSite. Consolidates all post-authoring fields (Author,
Repository, Series, Date, RedirectUrl) in one contract. Implements IFrontMatter, ITaggable,
ISectionable, IRedirectable, and IHasStructuredData (emits a schema.org Article).
public record BlogSiteFrontMatter
{
    /// Record key of this post's published site.standard.document record (Standard Site),
if any.
    public string AtprotoRkey { get; set; }

    /// Author name shown in the byline and RSS feed.
    public string Author { get; set; }

    /// Publication date. Posts are ordered by this date in archives and feeds.
    public DateTime? Date { get; set; }

    /// Short post description used for meta tags, RSS summary, and archive listings.
    public string Description { get; set; }

    /// Returns the schema.org entities to emit on the page. Implementations typically yield
one Article/Recipe/Product/etc. built from front matter values, plus the CanonicalUrl the
template supplies.
    public IEnumerable<JsonLdEntity> GetStructuredData(StructuredDataContext context)
;

    /// When true, the post is skipped during production builds.
    public bool IsDraft { get; set; }

    /// When false, the post is excluded from the generated llms.txt output.
    public bool Llms { get; set; }

    /// When set, the post emits a client-side redirect to this URL instead of normal
content.
    public string RedirectUrl { get; set; }

    /// URL to the post's source repository or related project (optional).
    public string Repository { get; set; }

    /// When false, the post is excluded from the search index.
    public bool Search { get; set; }

    /// When true, the post is indexed for search/llms but hidden from the rendered
navigation tree.
    public bool SearchOnly { get; set; }

    /// Section heading the post belongs under in navigation.
    public string SectionLabel { get; set; }

    /// Series name grouping related posts together.
    public string Series { get; set; }
```

```
        /// Post title.     public string Title { get; set; }       /// Stable identifier used
for cross-references ([text](xref:uid)).     public string Uid { get; set; }  }
```

Pennington.BlogSite.BlogSiteOptions

Pennington.BlogSite.BlogSiteOptions

CSHARP

```
namespace Pennington.BlogSite;

/// Options record passed to AddBlogSite that configures the BlogSite template: site
identity, content paths, typography, author chrome, homepage composition (hero, project
cards, social links, nav), and feed toggles (RSS, sitemap).
public record BlogSiteOptions
{
    /// Raw HTML appended to the document <head> (for analytics, meta tags, etc.).
    public string AdditionalHtmlHeadContent { get; set; }

    /// Additional assemblies scanned for Razor components so out-of-project pages
participate in routing.
    public Assembly[] AdditionalRoutingAssemblies { get; set; }

    /// Short author bio displayed on the homepage and post pages.
    public string AuthorBio { get; set; }

    /// Author name displayed in the byline and RSS channel.
    public string AuthorName { get; set; }

    /// URL prefix under which blog posts are published.
    public string BlogBaseUrl { get; set; }

    /// Folder (relative to ContentRootPath) containing blog post markdown files.
    public string BlogContentPath { get; set; }

    /// Absolute base URL used to build canonical links, sitemap, and RSS entries.
    public string CanonicalBaseUrl { get; set; }

    /// Color scheme driving the MonorailCSS theme. Defaults to the built-in BlogSite
palette when null.
    public IColorScheme ColorScheme { get; set; }

    /// Root folder (relative to the content project) that holds the content tree.
    public FilePath ContentRootPath { get; set; }

    /// When true, an RSS feed is generated for blog posts.
    public bool EnableRss { get; set; }

    /// When true, a sitemap.xml is generated for the site.
    public bool EnableSitemap { get; set; }

    /// Additional CSS appended to the generated stylesheet.
    public string ExtraStyles { get; set; }

    /// Favicon / icon links. Forwarded to Favicons.
    public FaviconOptions Favicons { get; set; }

    /// Fonts to preload via <link rel="preload"> for faster first paint.
    public FontPreload[] FontPreloads { get; set; }
```

```
    /// Navigation links rendered in the site header.    public HeaderLink[] MainSiteLinks { get; set; }      /// Featured projects displayed on the homepage.    public Project[] MyWork { get; set; }      /// Number of posts per page on the archive listing. Set to a non-positive value to disable pagination.    public int PostsPerPage { get; set; }      /// Short description used for the meta description tag and RSS channel.    public string SiteDescription { get; set; }      /// Site title shown in the header, OpenGraph tags, and RSS channel.    public string SiteTitle { get; set; }      /// Enables generated per-post social cards. When set, each post's og:image/twitter:image points at an on-demand-rendered card (unless SocialMediaImageUrlFactory returns a URL for that post, which wins). The host supplies the drawing via Render.    public SocialCardOptions SocialCards { get; set; }  /// Factory producing a social-share image URL for a given post. Return null to fall back to defaults.    public Func<BlogPostRef<BlogSiteFrontMatter>, string> SocialMediaImageUrlFactory { get; set; }      /// Social media links rendered in the site chrome.    public SocialLink[] Socials { get; set; }      /// Standard Site (AT Protocol) integration. Forwarded to StandardSite.    public StandardSiteOptions StandardSite { get; set; }  }
```

Pennington.BlogSite.BlogSiteServiceExtensions

Pennington.BlogSite.BlogSiteServiceExtensions

CSHARP

```
namespace Pennington.BlogSite;

/// DI extension methods for registering and running the BlogSite template.
public class BlogSiteServiceExtensions
{
    /// Registers BlogSite services with the provided options.
    public static IServiceCollection AddBlogSite(IServiceCollection services,
Func<BlogSiteOptions> configureOptions)
;

    /// Runs the BlogSite: either serves the app or performs a static build, based on command-line args.
    public static Task RunBlogSiteAsync(WebApplication app, string[] args)
;

    /// Wires BlogSite middleware, Razor components, and RSS endpoint into the request pipeline.
    public static WebApplication UseBlogSite(WebApplication app)
;
}
```

Pennington.BlogSite.HeaderLink

Pennington.BlogSite.HeaderLink

CSHARP

```
namespace Pennington.BlogSite;

/// Link rendered in the main site navigation.
public record HeaderLink
{
    /// Link rendered in the main site navigation.
    public HeaderLink(string Title, string Url)
;

    /// Display text.
    public string Title { get; set; }

    /// Target URL.
    public string Url { get; set; }

}
```

Pennington.BlogSite.HeroContent

Pennington.BlogSite.HeroContent

CSHARP

```
namespace Pennington.BlogSite;

/// Hero block at the top of the blog homepage.
public record HeroContent
{
    /// Hero subhead/body text.
    public string Description { get; set; }

    /// Hero block at the top of the blog homepage.
    public HeroContent(string Title, string Description)
;

    /// Hero headline.
    public string Title { get; set; }

}
```

Pennington.BlogSite.Project

Pennington.BlogSite.Project

CSHARP

```
namespace Pennington.BlogSite;

/// Featured project card shown on the blog homepage.
public record Project
{
    /// Short project description.
    public string Description { get; set; }

    /// Featured project card shown on the blog homepage.
    public Project(string Title, string Description, string Url)
;

    /// Project title.
    public string Title { get; set; }

    /// Link target for the card.
    public string Url { get; set; }

}
```

Pennington.BlogSite.RenderedNotFound

Pennington.BlogSite.RenderedNotFound

CSHARP

```
namespace Pennington.BlogSite;

/// A rendered not-found body sourced from a content-root 404.md. Carries only a title and
HTML — the post chrome (date, tags, series) does not apply to an error page.
public record RenderedNotFound
{
    /// Rendered HTML body.
    public string Html { get; set; }

    /// A rendered not-found body sourced from a content-root 404.md. Carries only a title
and HTML — the post chrome (date, tags, series) does not apply to an error page.
    public RenderedNotFound(string Title, string Html)
;

    /// Title from the 404.md front matter (or a default when absent).
    public string Title { get; set; }

}
```

Pennington.BlogSite.Services.BlogContentResolver

Pennington.BlogSite.Services.BlogContentResolver

CSHARP

```
namespace Pennington.BlogSite.Services;

/// Resolves the site's not-found body from a content-root 404.md, rendered through the markdown pipeline. Post listings, single-post rendering, browse-by-tag, and RSS are served by the shared BlogPostQuery and the registered taxonomy axis.
public class BlogContentResolver
{
    /// Creates a new resolver with the supplied front-matter parser, renderer, and options.
    public BlogContentResolver(FrontMatterParser parser, IContentRenderer renderer, BlogSiteOptions options)
;

    /// Resolves the site's not-found body from a content-root 404.md, rendered through the markdown pipeline. Returns null when no 404.md exists — the catch-all then tries a NotFound component, then the built-in message. The file is reserved out of discovery (ReserveNotFoundPage), so it is never a post route.
    public Task<RenderedNotFound> GetNotFoundContentAsync()
;
}
```

Pennington.BlogSite.Services.BlogSiteContentService

Pennington.BlogSite.Services.BlogSiteContentService

CSHARP

```
namespace Pennington.BlogSite.Services;

/// Emits the paginated archive routes (/archive/page/{n}) the parameterized @page template hides from automatic discovery, and projects the site-identity record for the template-owned home page (so / participates in social-card generation). Post counts come from the shared BlogPostQuery (cached records, no disk re-read); browse-by-tag is a registered taxonomy; the canonical /archive route comes from RazorPageContentService.
public class BlogSiteContentService
{
    /// Creates the service. The provider resolves BlogPostQuery on demand to avoid a construction cycle through the content-record registry.
    public BlogSiteContentService(BlogSiteOptions options, IServiceProvider services)
;

    /// Default section label applied to discovered items that do not supply one via front matter.
    public string DefaultSectionLabel { get; }

    /// Discover all content items this service is responsible for.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Navigation entries for table of contents.
    public Task<ImmutableList<ContentTocItem>> GetContentTocEntriesAsync()
;

    /// Static files to copy to output (images, downloads, etc.)
    public Task<ImmutableList<ContentToCopy>> GetContentToCopyAsync()
;

    /// Cross-references for xref resolution.
    public Task<ImmutableList<CrossReference>> GetCrossReferencesAsync()
;

    /// Projects this service's routable content as ContentRecords — the discovery seam consumed by taxonomy, search faceting, and structured-data emission. Default: bridges from DiscoverAsync, yielding one record per discovered item that carries Metadata and is neither a RedirectSource (transport, not content) nor an LlmsOnlySource (no human-facing URL). A service that attaches typed metadata to its discovered items — as MarkdownContentService does — therefore participates with no extra code. Override only to project records that do not flow through DiscoverAsync, or to suppress records entirely.A service that emits routable content from DiscoverAsync but leaves Metadata unset projects no records, and so silently sits out of taxonomy, search faceting, and structured data. Set the metadata (or override this) to opt in.
    public IAsyncEnumerable<ContentRecord> GetRecordsAsync()
;

    /// Relative priority for ordering results in the search index (higher values rank first).
    public int SearchPriority { get; }

}
```

Pennington.BlogSite.SocialLink

Pennington.BlogSite.SocialLink

CSHARP

```
namespace Pennington.BlogSite;

/// Icon and URL for a social media link.
public record SocialLink
{
    /// Rendered icon markup.
    public RenderFragment Icon { get; set; }

    /// Icon and URL for a social media link.
    public SocialLink(RenderFragment Icon, string Url)
;

    /// Target URL.
    public string Url { get; set; }

}
```

Pennington.Book.BookArtifactContentService

Pennington.Book.BookArtifactContentService

CSHARP

```
namespace Pennington.Book;

/// Artifact-tier façade over BookArtifactService: claims /pdf/ and /book-preview/, renders
PDFs and live previews on demand in dev through core's artifact router, and enumerates the
PDFs for the static build. Preview routes are resolvable but deliberately not enumerated —
they exist for live print-CSS iteration only, so build output stays PDF-only. Transient so
each resolution captures the current file-watched service.
public class BookArtifactContentService
{
    /// Creates the façade over the given BookArtifactService.
    public BookArtifactContentService(BookArtifactService service)
;

    /// URL territories this service serves. Options-derived and consulted on every request
— must be cheap and must not trigger discovery, the projection, or any lazy corpus work.
    public ImmutableList<ArtifactClaim> Claims { get; }

    /// Enumerates every artifact route the static build should write, as GeneratedSource
items. May consume the projection — the build invokes this outside any request, after the
page crawl has primed the render cache. Never called on the request path.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Returns the bytes for relativePath (no leading slash, e.g. search/en/index.json), or
null to decline so the request falls through to content routing. May materialize the
projection, build an index, or run Chromium on demand.
    public Task<ArtifactContent> ResolveAsync(string relativePath, CancellationToken
cancellationToken)
;
}
```

Pennington.Book.BookArtifactService

Pennington.Book.BookArtifactService

CSHARP

```
namespace Pennington.Book;

/// Single source of truth for every book artifact, behind the artifact-tier façade
BookArtifactContentService (which serves dev requests and enumerates the PDFs for the static
build) — the same shape as SearchArtifactService. The projection fold (post-pipeline HTML
per page) and the static-asset map are computed once, lazily; each book's composed HTML is
cached per (book, locale); PDF bytes are rendered on demand through the singleton
ChromiumBrowserProvider and cached until a file change drops the whole service.
EnumerateArtifacts is deliberately cheap — pure options × locales, no projection, no
Chromium — so build enumeration and link verification never trigger a render.
public class BookArtifactService
{
    /// Creates the service; all artifacts are computed lazily on first request.
    public BookArtifactService(BookOptions options, PenningtonOptions penn,
LocalizationOptions localization, ISiteProjection projection, IEnumerable<IContentService>
contentServices, NavigationBuilder navigationBuilder, BookComposer composer, AssetInliner
assetInliner, ChromiumBrowserProvider chromium, TimeProvider clock,
ILogger<BookArtifactService> logger)
;

    /// Renders the PDF for the artifact at pdfPath (e.g. pdf/tutorials.pdf), or null when
no book matches.
    public Task<byte[]> GetPdfAsync(string pdfPath)
;

    /// Returns the composed HTML for the preview route at previewPath, or null when no book
matches.
    public Task<string> GetPreviewHtmlAsync(string previewPath)
;

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;
}
```

Pennington.Book.BookCatalog

Pennington.Book.BookCatalog

CSHARP

```csharp
namespace Pennington.Book;

/// Advertises each configured book as a DownloadLink. Derives the link list purely from
BookOptions, PenningtonOptions, and LocalizationOptions — it never touches the site
projection, so it is safe to resolve on the request path (the DocSite sidebar resolves it in
MainLayout.OnInitializedAsync). The link label comes from the pennington.book.download
translation when one is registered, falling back to "Download as PDF".
public class BookCatalog
{
    /// Creates the catalog over the configured book, localization, and translation options.
    public BookCatalog(BookOptions options, PenningtonOptions penn, LocalizationOptions
localization, TranslationOptions translations)
;

    /// Returns the links available for locale (or the default locale when null), each with
a locale-appropriate URL.
    public IReadOnlyList<DownloadLink> GetLinks(string locale = null)
;
}
```

Pennington.Book.BookDefinition

Pennington.Book.BookDefinition

CSHARP

```
namespace Pennington.Book;

/// One book carved out of the site's table of contents. A book covers every page whose canonical path falls under RoutePrefix; the whole TOC when the prefix is /.
public record BookDefinition
{
    /// One book carved out of the site's table of contents. A book covers every page whose canonical path falls under RoutePrefix; the whole TOC when the prefix is /.
    public BookDefinition(string Title, string RoutePrefix)
;

    /// Effective output slug — Slug when set, otherwise derived from RoutePrefix.
    public string EffectiveSlug { get; }

    /// Route prefix normalized to a leading + trailing slash (/tutorials/); / for the whole site.
    public string NormalizedRoutePrefix { get; }

    /// Canonical route prefix the book covers (for example /tutorials/, or / for the whole site).
    public string RoutePrefix { get; set; }

    /// Optional slug for the output file (pdf/{slug}.pdf); defaults to the route prefix flattened to a slug.
    public string Slug { get; set; }

    /// Optional cover subtitle; defaults to the site description for the whole-site book.
    public string Subtitle { get; set; }

    /// Book title, shown on the cover and the download link.
    public string Title { get; set; }

}
```

Pennington.Book.BookOptions

Pennington.Book.BookOptions

CSHARP

```csharp
namespace Pennington.Book;

/// Configuration for AddPenningtonBook: which books to carve out of the TOC and how to launch Chromium. When Books is empty a single book covers the whole site.
public class BookOptions
{
    /// Extra command-line arguments passed to Chromium (for example --no-sandbox in CI).
    public string[] AdditionalChromiumArgs { get; set; }

    /// Extra CSS appended after the built-in print stylesheet, for per-site print tweaks.
    public string AdditionalCss { get; set; }

    /// Explicit books. Empty (the default) means one whole-site book titled from the site title.
    public List<BookDefinition> Books { get; }

    /// Path to a Chromium/Chrome executable. When null, PuppeteerSharp downloads a private Chromium on first use.
    public string ChromiumExecutablePath { get; set; }

    /// When true, appends a grayscale override stylesheet so books print without color — neutral grays for links, alerts, code, and the syntax palette. AdditionalCss still wins, since it is appended last.
    public bool Monochrome { get; set; }

}
```

Pennington.Book.Composition.AssetInliner

Pennington.Book.Composition.AssetInliner

CSHARP

```
namespace Pennington.Book.Composition;

/// Resolves <img src> values to self-contained data: URIs so the composed book renders
offline (Chromium navigates via SetContentAsync with no server to fetch from). Internal
sources resolve first from the content-copy map (a content source's own static assets), then
from the host web root (wwwroot + RCL static web assets). External http(s) images are left
untouched — an offline Chromium simply omits them — and unresolved internal sources are
rewritten to an absolute URL so they at least point somewhere real.
public class AssetInliner
{
    /// Creates an inliner over the given file system, web-root provider, and canonical
base.
    public AssetInliner(IFileSystem fileSystem, IFileProvider webRoot, CanonicalBaseUrl
canonicalBase)
;

    /// Builds the output-path → source-file map from every content service's static-copy
declarations. First registration wins on a duplicate output path, mirroring the build's copy
pass.
    public static Task<IReadOnlyDictionary<string, string>>
BuildContentMapAsync(IEnumerable<IContentService> services)
;

    /// Resolves src to a data: URI when the bytes can be found, an absolute URL when an
internal source is unresolved, or src unchanged for external images.
    public string Resolve(string src, IReadOnlyDictionary<string, string> contentMap)
;
}
```

Pennington.Book.Composition.BookComposer

Pennington.Book.Composition.BookComposer

CSHARP

```
namespace Pennington.Book.Composition;

/// Composes one self-contained book document from a navigation tree and the projected pages
under it. Each page is normalized exactly like llms.txt — its post-pipeline HTML is
converted to markdown (HtmlToMarkdownConverter) and re-rendered through the shared
MarkdownPipeline, which re-highlights fences and normalizes alerts/tabs regardless of which
content service produced the page. The result is wrapped into chapter/page sections,
prefixed with a cover and a page-numbered table of contents, and emitted with the print
stylesheet and the paged.js polyfill inlined so the document renders offline (no server, no
external assets).
public class BookComposer
{
    /// Creates a composer that re-renders through pipeline, absolutizes out-of-book links
against canonicalBase, titles the cover from penn, and localizes book chrome strings through
translations for the locale resolved against localization.
    public BookComposer(MarkdownPipeline pipeline, CanonicalBaseUrl canonicalBase,
PenningtonOptions penn, TranslationOptions translations, LocalizationOptions localization)
;

    /// Composes the full HTML document for book from tree (already scoped to the book) and
pageByPath (the projected pages keyed by trimmed canonical path). A tree wrapped in the
book's own index node is unwrapped first (UnwrapBookRoot) so the index's children become the
chapters. When monochrome is set, a grayscale override stylesheet is appended after the
built-in one; additionalCss is appended last (so it still wins), and resolveImageSrc inlines
image sources to data: URIs. stamp supplies provenance (version, date, locale) for the cover
version line and the colophon page; when null both are omitted and the document language
falls back to the default locale.
    public string Compose(BookDefinition book, ImmutableList<NavigationTreeItem> tree,
IReadOnlyDictionary<string, RenderedPage> pageByPath, string additionalCss, Func<string,
string> resolveImageSrc, bool monochrome = false, BookStamp stamp = null)
;
}
```

Pennington.Book.Composition.BookStamp

Pennington.Book.Composition.BookStamp

CSHARP

```
namespace Pennington.Book.Composition;

/// Provenance stamped into a composed book: the cover's version line and the colophon page. When no stamp is supplied to Compose, both are omitted.
public record BookStamp
{
    /// Provenance stamped into a composed book: the cover's version line and the colophon page. When no stamp is supplied to Compose, both are omitted.
    public BookStamp(string Version, DateTimeOffset GeneratedAt, string Locale)
;

    /// Generation timestamp; the colophon prints it as month + year.
    public DateTimeOffset GeneratedAt { get; set; }

    /// Locale code driving the lang attribute, translation lookups, and date formatting; null means the default locale.
    public string Locale { get; set; }

    /// Host site version (informational version with build metadata trimmed); null omits the version lines.
    public string Version { get; set; }

}
```

Pennington.Book.PenningtonBookExtensions

Pennington.Book.PenningtonBookExtensions

CSHARP

```
namespace Pennington.Book;

/// Dependency injection extensions for the Pennington PDF book feature.
public class PenningtonBookExtensions
{
    /// Adds PDF book generation: a per-locale book per BookDefinition (or one whole-site book when none are configured), served on demand at /pdf/{slug}.pdf in dev and emitted into the static build. Registers an IDownloadLinkProvider a host's chrome can advertise.
    public static IServiceCollection AddPenningtonBook(IServiceCollection services, Action<BookOptions> configure = null)
;
}
```

Pennington.Book.Rendering.ChromiumBrowserProvider

Pennington.Book.Rendering.ChromiumBrowserProvider

CSHARP

```
namespace Pennington.Book.Rendering;

/// Owns the process-lifetime Chromium instance used to render composed book HTML to PDF.
Registered as a DI singleton — the documented "connection pool" exception to the
transient/file-watched default — because launching Chromium is expensive and the browser is
safe to reuse across renders. PDF rendering is serialized through a semaphore so a single
browser never juggles concurrent paginations.
public class ChromiumBrowserProvider
{
    /// Creates the provider; Chromium is launched lazily on the first render.
    public ChromiumBrowserProvider(BookOptions options, ILogger<ChromiumBrowserProvider>
logger)
;

    /// Closes the browser process if one was launched.
    public ValueTask DisposeAsync()
;

    /// Renders html to PDF bytes. The HTML must signal completion by setting
window.__pagedDone = true (the paged.js after hook the composer wires up); rendering waits
up to two minutes for that flag before producing the PDF.
    public Task<byte[]> RenderPdfAsync(string html, CancellationToken cancellationToken =
default)
;
}
```

Pennington.Cli.AsciiTreeWriter

Pennington.Cli.AsciiTreeWriter

CSHARP

```
namespace Pennington.Cli;

/// Renders a hierarchy as an ASCII tree (├─ └─ │) to a TextWriter.
public class AsciiTreeWriter
{
    /// Writes nodes as a tree. label formats one node to a single line; children yields a
node's children. Recurses until maxDepth (1-based; the top level is depth 1).
    public static void Write<T>(TextWriter writer, IReadOnlyList<T> nodes, Func<T, string>
label, Func<T, IReadOnlyList<T>> children, int maxDepth = 2147483647)
;
}
```

Pennington.Cli.IDiagCommand

Pennington.Cli.IDiagCommand

CSHARP

```
namespace Pennington.Cli;

/// A diag subcommand. Implementations register in DI as IDiagCommand; the CLI discovers
them and adds one System.CommandLine Command per implementation under the diag group. Each
command inspects the started app's services and writes a human-readable text report.
Optional packages (and hosts) add their own inspection verbs by registering an
implementation.
public interface IDiagCommand
{
    /// Builds the System.CommandLine Command for this subcommand, wiring its options and an
action that inspects services (a fully started host's service provider) and writes to
output. The action returns the process exit code.
    public Command Build(IServiceProvider services, TextWriter output)
;

    /// One-line description shown in diag --help.
    public string Description { get; }

    /// Subcommand verb shown under diag (e.g. toc, warnings).
    public string Name { get; }

}
```

Pennington.Content.BlogPostQuery

Pennington.Content.BlogPostQuery

CSHARP

```
namespace Pennington.Content;

/// Shared read model for blog-style content: lists posts, paginates them, renders a single post, and builds the RSS feed. Everything reads the cached ContentRecordRegistry snapshot and IPageResolver, so nothing re-reads or re-parses markdown per request. Generic over the host's front-matter type; both the DocSite and BlogSite templates consume this one service.
public class BlogPostQuery
{
    /// Creates the query over the content-record registry, page resolver, and wall clock.
    public BlogPostQuery(ContentRecordRegistry records, IPageResolver resolver, TimeProvider clock = null)
;

    /// Returns one page of posts under basePrefix, newest first. Returns null when page/pageSize are non-positive or the page is past the end; page 1 of an empty set yields an empty page rather than null.
    public Task<PagedList<BlogPostRef<TFrontMatter>>> GetPageAsync<TFrontMatter>(string basePrefix, int page, int pageSize)
;

    /// Returns every published TFrontMatter post whose route sits under basePrefix (e.g. /blog), newest first. Drafts and future-dated posts are excluded. Reads the cached record snapshot — no file I/O.
    public Task<ImmutableList<BlogPostRef<TFrontMatter>>> GetPostsAsync<TFrontMatter>(string basePrefix)
;

    /// Renders the single post at url through the cached page resolver. Returns null when nothing matches or the matched page is not a TFrontMatter.
    public Task<RenderedBlogPost<TFrontMatter>> GetRenderedPostAsync<TFrontMatter>(UrlPath url)
;

    /// Builds RSS 2.0 XML for the posts under basePrefix, newest first. author projects each post's author name (the field is template-specific, not part of IFrontMatter); pass null to omit authors.
    public Task<string> GetRssXmlAsync<TFrontMatter>(string siteTitle, string siteDescription, string canonicalBaseUrl, string basePrefix, Func<TFrontMatter, string> author = null)
;
}
```

Pennington.Content.BlogPostRef

Pennington.Content.BlogPostRef

CSHARP

```
namespace Pennington.Content;

/// A blog post's typed front matter paired with its canonical URL, for listings.
public record BlogPostRef
{
    /// A blog post's typed front matter paired with its canonical URL, for listings.
    public BlogPostRef`1(TFrontMatter FrontMatter, UrlPath Url)
;

    /// Parsed front matter for the post.
    public TFrontMatter FrontMatter { get; set; }

    /// Canonical URL of the post.
    public UrlPath Url { get; set; }

}
```

Pennington.Content.ContentChangeImpact

Pennington.Content.ContentChangeImpact

CSHARP

```
namespace Pennington.Content;

/// What an IContentService reports as affected by a single FileChangeNotification.
AffectedRoutes is null for a wildcard (consumers drop their full cache), empty for no
impact, and populated for per-route eviction.
public record ContentChangeImpact
{
    /// Routes that need re-rendering, or null to signal a wildcard.
    public ImmutableArray<ContentRoute>? AffectedRoutes { get; set; }

    /// What an IContentService reports as affected by a single FileChangeNotification.
AffectedRoutes is null for a wildcard (consumers drop their full cache), empty for no
impact, and populated for per-route eviction.
    public ContentChangeImpact(ImmutableArray<ContentRoute>? AffectedRoutes)
;

    /// No routes affected — the change is outside this service's scope.
    public static ContentChangeImpact None { get; }

    /// Affects the given routes; consumers evict per-route.
    public static ContentChangeImpact Routes(ImmutableArray<ContentRoute> routes)
;

    /// Affects an unknown / unbounded set of routes; consumers should drop their full
cache.
    public static ContentChangeImpact Wildcard { get; }

}
```

Pennington.Content.ContentRecord

Pennington.Content.ContentRecord

CSHARP

```
namespace Pennington.Content;

/// A source-agnostic, routable content record — the unit every discovery channel reads.
Pairs a canonical ContentRoute with its typed IFrontMatter, independent of whether the page
came from markdown, a Razor page, or a custom IContentService / endpoint. Taxonomy, search
faceting, and structured-data emission all consume records, so a service that projects
records through GetRecordsAsync gets the same browse-by-field, custom-facet, and JSON-LD
behavior that markdown has. The capabilities each pillar reads — ITaggable, ISectionable,
IHasStructuredData, IHasSearchFacets — ride on Metadata.
public record ContentRecord
{
    /// A source-agnostic, routable content record — the unit every discovery channel reads.
Pairs a canonical ContentRoute with its typed IFrontMatter, independent of whether the page
came from markdown, a Razor page, or a custom IContentService / endpoint. Taxonomy, search
faceting, and structured-data emission all consume records, so a service that projects
records through GetRecordsAsync gets the same browse-by-field, custom-facet, and JSON-LD
behavior that markdown has. The capabilities each pillar reads — ITaggable, ISectionable,
IHasStructuredData, IHasSearchFacets — ride on Metadata.
    public ContentRecord(ContentRoute Route, IFrontMatter Metadata)
;

    /// Typed front matter carrying the record's title, dates, indexing flags, and
capability mixins.
    public IFrontMatter Metadata { get; set; }

    /// Canonical route the record is served at.
    public ContentRoute Route { get; set; }

}
```

Pennington.Content.ContentRecordRegistry

Pennington.Content.ContentRecordRegistry

CSHARP

```
namespace Pennington.Content;

/// Aggregates the ContentRecords projected by every registered IContentService into a
snapshot keyed by canonical path, so consumers can resolve the typed front matter for a
route without re-walking the services. Registered via AddFileWatched<ContentRecordRegistry>
() so the table is dropped when any service's source changes. Search faceting and
structured-data emission both join the rendered corpus back to its records through this
registry: the route a page was served at resolves to the Metadata that carries its
capabilities (IHasSearchFacets, IHasStructuredData, ...).
public class ContentRecordRegistry
{
    /// Creates a registry that lazily aggregates records across all registered content
services.
    public ContentRecordRegistry(IEnumerable<IContentService> contentServices)
;

    /// Returns the aggregated record snapshot keyed by canonical path with slashes trimmed
— matching SiteProjection's route key so the two join cleanly. Materializes on first call;
dropped on file-watch invalidation so the next access rebuilds.
    public Task<FrozenDictionary<string, ContentRecord>> GetSnapshotAsync()
;

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;
}
```

Pennington.Content.ContentRootAssetService

Pennington.Content.ContentRootAssetService

CSHARP

```
namespace Pennington.Content;

/// Surfaces every servable file under the content root as a ContentToCopy, so the static
build mirrors the whole-content-root static mount that UsePennington serves at runtime.
Files placed in the content root but outside any markdown source's ContentPath (the
documented home for shared, absolute-URL assets) are served live but were never copied by
the build. Routing them through GetContentToCopyAsync closes that gap on one code path: the
build copy and both link auditors already consume CollectContentToCopyAsync, so dev and
build cannot diverge.Carries no routes, navigation, or search entries — it is an asset-copy
source only. Register it after the markdown sources so their (prefix-aware) outputs win the
output-path dedup in the build.
public class ContentRootAssetService
{
    /// Creates the service over contentRootPath; the content-type gate defaults to
ASP.NET's standard extension map.
    public ContentRootAssetService(string contentRootPath, IFileSystem fileSystem,
IContentTypeProvider contentTypeProvider = null)
;

    /// Default section label applied to discovered items that do not supply one via front
matter.
    public string DefaultSectionLabel { get; }

    /// Discover all content items this service is responsible for.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Navigation entries for table of contents.
    public Task<ImmutableList<ContentTocItem>> GetContentTocEntriesAsync()
;

    /// Static files to copy to output (images, downloads, etc.)
    public Task<ImmutableList<ContentToCopy>> GetContentToCopyAsync()
;

    /// Cross-references for xref resolution.
    public Task<ImmutableList<CrossReference>> GetCrossReferencesAsync()
;

    /// Relative priority for ordering results in the search index (higher values rank
first).
    public int SearchPriority { get; }

}
```

Pennington.Content.ContentServiceExtensions

Pennington.Content.ContentServiceExtensions

CSHARP

```csharp
namespace Pennington.Content;

/// Helpers that collapse the repeated foreach service { … } fan-out patterns across
consumers of IEnumerable of IContentService.
public class ContentServiceExtensions
{
    public static Task<ImmutableList<ContentToCopy>>
CollectContentToCopyAsync(IEnumerable<IContentService> services)
;

    public static Task<ImmutableList<CrossReference>>
CollectCrossReferencesAsync(IEnumerable<IContentService> services)
;

    public static Task<ImmutableList<ContentTocItem>>
CollectIndexableEntriesAsync(IEnumerable<IContentService> services)
;

    public static Task<ImmutableList<ContentTocItem>>
CollectTocEntriesAsync(IEnumerable<IContentService> services)
;

    public static IAsyncEnumerable<DiscoveredItem>
DiscoverAllAsync(IEnumerable<IContentService> services, CancellationToken cancellationToken
= default)
;

    public static IAsyncEnumerable<ContentRecord>
GetAllRecordsAsync(IEnumerable<IContentService> services, CancellationToken
cancellationToken = default)
;

    public static IAsyncEnumerable<ParsedItem>
ParseAllContentAsync(IEnumerable<IContentService> services, CancellationToken
cancellationToken = default)
;

    public static IEnumerable<IContentService> SourceServices(IEnumerable<IContentService>
services)
;
}
```

Pennington.Content.ContentTocItem

Pennington.Content.ContentTocItem

CSHARP

```
namespace Pennington.Content;

/// A table-of-contents entry for navigation.
public record ContentTocItem
{
    /// A table-of-contents entry for navigation.
    public ContentTocItem(string Title, ContentRoute Route, int Order, string[]
HierarchyParts, string SectionLabel, string Locale)
;

    /// Front-matter description, surfaced as a boosted field in the search index.
    public string Description { get; set; }

    /// When true, excluded from llms.txt.
    public bool ExcludeFromLlms { get; set; }

    /// When true, excluded from the search index.
    public bool ExcludeFromSearch { get; set; }

    /// Ancestor path segments used to place the entry in the tree.
    public string[] HierarchyParts { get; set; }

    /// Locale the entry belongs to, or null when the entry is locale-neutral.
    public string Locale { get; set; }

    /// Sort order within the entry's section.
    public int Order { get; set; }

    /// Content route the entry links to.
    public ContentRoute Route { get; set; }

    /// When true, the entry is indexed for search/llms but filtered out of the rendered
navigation tree by BuildTreeAsync. Search-index and llms.txt consumers keep the entry as
long as ExcludeFromSearch / ExcludeFromLlms allow it.
    public bool SearchOnly { get; set; }

    /// Section label used for grouping, or null for the default bucket.
    public string SectionLabel { get; set; }

    /// Front-matter tags, surfaced as a search facet. Empty when the front matter is not
ITaggable.
    public string[] Tags { get; set; }

    /// Display title shown in navigation.
    public string Title { get; set; }

}
```

Pennington.Content.ContentToCopy

Pennington.Content.ContentToCopy

CSHARP

```
namespace Pennington.Content;

/// A static file to copy to the output directory.
public record ContentToCopy
{
    /// A static file to copy to the output directory.
    public ContentToCopy(FilePath SourcePath, FilePath OutputPath)
;

    /// Destination path in the generated site.
    public FilePath OutputPath { get; set; }

    /// Source file to copy.
    public FilePath SourcePath { get; set; }

}
```

Pennington.Content.FileContentService

Pennington.Content.FileContentService

CSHARP

```
namespace Pennington.Content;

/// Generic file-format content source: globs FilePattern under the content directory,
parses each file's front matter into TFrontMatter, and emits FileSource discovered items
tagged with the format key. Registered by AddContentFormat; the format's registered
parser/renderer turn the bodies into HTML. Deliberately leaner than MarkdownContentService —
no locale fan-out, no .llms.md handling, no _meta.yml folder metadata.
public class FileContentService
{
    /// Default section label applied to discovered items that do not supply one via front
matter.
    public string DefaultSectionLabel { get; }

    /// Discover all content items this service is responsible for.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Creates the service and prepares lazy discovery of the content directory.
    public FileContentService`1(FileContentServiceOptions options, FrontMatterParser parser,
IFileSystem fileSystem, TimeProvider clock = null, ILogger<FileContentService<TFrontMatter>>
logger = null)
;

    /// Maps a file-change notification to the set of routes this service projects from that
file, without mutating any cached state. Consulted by file-watched caches (SiteProjection,
BuildHtmlCache) to invalidate only the affected entries instead of clearing wholesale.
Default: None.
    public ContentChangeImpact GetAffectedRoutes(FileChangeNotification change)
;

    /// Navigation entries for table of contents.
    public Task<ImmutableList<ContentTocItem>> GetContentTocEntriesAsync()
;

    /// Static files to copy to output (images, downloads, etc.)
    public Task<ImmutableList<ContentToCopy>> GetContentToCopyAsync()
;

    /// Cross-references for xref resolution.
    public Task<ImmutableList<CrossReference>> GetCrossReferencesAsync()
;

    /// Redirect sources this service emits (each item's Source is a RedirectSource).
Consumed by RedirectContentService to build the unified redirect map without iterating every
service's DiscoverAsync — which would force services that have no redirects to pay the full
cost of discovery just to return nothing. Default: empty. Services backed by front-matter
records that implement IRedirectable override this.
    public Task<ImmutableList<DiscoveredItem>> GetRedirectSourcesAsync()
;

    /// Called on the file-watcher thread for every watched change. Must be quick and
```

```
      /// Discovers and parses this service's content with the service's own front-matter type, yielding ParsedItems (typed metadata + body). Consumers like LlmsTxtService use this instead of re-parsing with a foreign parser, which would mis-flag valid keys from other content types. Default: empty — services whose content is sourced elsewhere (Razor/API pages fetched as rendered HTML) opt out.    public IAsyncEnumerable<ParsedItem> ParseContentAsync();      /// Relative priority for ordering results in the search index (higher values rank first).    public int SearchPriority { get; }      /// Directories needing an OS-level watcher. Empty (the default) for aggregators that ride notifications other watchers already produce.    public IReadOnlyList<FileWatchScope> WatchScopes { get; }
}
```

Pennington.Content.FileContentServiceOptions

Pennington.Content.FileContentServiceOptions

CSHARP

```
namespace Pennington.Content;

/// Configuration for a FileContentService — the discovery source for a custom file format registered via AddContentFormat.
public class FileContentServiceOptions
{
    /// URL prefix prepended to routes generated from this content directory.
    public UrlPath BasePageUrl { get; set; }

    /// Filesystem path to the directory containing the format's source files.
    public FilePath ContentPath { get; set; }

    /// Relative paths (forward-slash, from ContentPath) whose subtrees are skipped during discovery. Matching is case-insensitive and segment-based.
    public ImmutableArray<string> ExcludePaths { get; set; }

    /// Glob pattern used to enumerate source files (for example *.cook).
    public string FilePattern { get; set; }

    /// Format key stamped onto discovered items, selecting the parser and renderer.
    public string Format { get; set; }

    /// Relative ordering priority for this source's entries in the search index.
    public int SearchPriority { get; set; }

    /// Default section label applied to entries when front matter doesn't specify one.
    public string SectionLabel { get; set; }

}
```

Pennington.Content.FolderMetadata

Pennington.Content.FolderMetadata

CSHARP

```
namespace Pennington.Content;

/// Metadata for a content folder discovered from a _meta.yml sidecar. Lets a folder declare its own display title, its position in the parent's nav level, and (optionally) its participation in the llms.txt sidecar.
public record FolderMetadata
{
    /// Metadata for a content folder discovered from a _meta.yml sidecar. Lets a folder declare its own display title, its position in the parent's nav level, and (optionally) its participation in the llms.txt sidecar.
    public FolderMetadata(string FolderUrlPrefix, string Title, int? Order, string LlmsDescription)
;

    /// Canonical URL prefix in /foo/bar/ form (always leading and trailing slash) identifying the folder this metadata applies to.
    public string FolderUrlPrefix { get; set; }

    /// When non-null, opts the subtree into llms.txt generation; the value is the subtree blurb.
    public string LlmsDescription { get; set; }

    /// Sort order of the folder within its parent level; overrides emergent min(children) and any index.md order when set.
    public int? Order { get; set; }

    /// Display title for the folder; overrides FormatSectionTitle and any index.md title when set.
    public string Title { get; set; }

}
```

Pennington.Content.FolderMetadataRegistry

Pennington.Content.FolderMetadataRegistry

CSHARP

```
namespace Pennington.Content;

/// Aggregates FolderMetadata rows from every IFolderMetadataProvider registered as an
IContentService and exposes them as an async-loaded snapshot keyed by canonical folder URL
prefix. Registered via AddFileWatched<FolderMetadataRegistry>() so the aggregated table is
dropped when any provider's underlying file source changes. Callers await
registry.GetSnapshotAsync() once at the top of their build path (e.g. BuildTreeAsync) and
pass the snapshot down through sync recursion. The async load runs on a thread-pool thread;
no caller blocks on sync-over-async.
public class FolderMetadataRegistry
{
    /// Creates a registry that lazily aggregates folder metadata across all registered
content services.
    public FolderMetadataRegistry(IEnumerable<IContentService> contentServices)
;

    /// Returns the aggregated folder-metadata snapshot, keyed by canonical folder URL
prefix (/foo/bar/ form). Materializes on first call; subsequent calls return the same task.
The instance is dropped on file-watch invalidation, so the next access rebuilds.
    public Task<FrozenDictionary<string, FolderMetadata>> GetSnapshotAsync()
;

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;
}
```

Pennington.Content.IContentService

Pennington.Content.IContentService

CSHARP

```
namespace Pennington.Content;

/// Discovers and provides content for the pipeline.
public interface IContentService
{
    /// Default section label applied to discovered items that do not supply one via front
matter.
    public string DefaultSectionLabel { get; }

    /// Discover all content items this service is responsible for.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Maps a file-change notification to the set of routes this service projects from that
file, without mutating any cached state. Consulted by file-watched caches (SiteProjection,
BuildHtmlCache) to invalidate only the affected entries instead of clearing wholesale.
Default: None.
    public ContentChangeImpact GetAffectedRoutes(FileChangeNotification change)
;

    /// Navigation entries for table of contents.
    public Task<ImmutableList<ContentTocItem>> GetContentTocEntriesAsync()
;

    /// Static files to copy to output (images, downloads, etc.)
    public Task<ImmutableList<ContentToCopy>> GetContentToCopyAsync()
;

    /// Cross-references for xref resolution.
    public Task<ImmutableList<CrossReference>> GetCrossReferencesAsync()
;

    /// Entries that should appear in the search index and llms.txt. Default: returns
GetContentTocEntriesAsync. That is correct when "shown in navigation" ≡ "discoverable via
search" — the default holds for markdown, because MarkdownContentService's TOC entries
already honor search: and llms: front-matter fields via ExcludeFromSearch /
ExcludeFromLlms.Override when the two sets diverge — for example, RazorPageContentService
emits sidecar-less pages here (so users can search for them) without adding them to
navigation (which would clutter the TOC with auto-titled entries).Implementors of custom
content services that build ContentTocItems directly: set ExcludeFromSearch and
ExcludeFromLlms from your metadata's Search / Llms flags, or per-page opt-outs will be
silently ignored.
    public Task<ImmutableList<ContentTocItem>> GetIndexableEntriesAsync()
;

    /// Projects this service's routable content as ContentRecords — the discovery seam
consumed by taxonomy, search faceting, and structured-data emission. Default: bridges from
DiscoverAsync, yielding one record per discovered item that carries Metadata and is neither
a RedirectSource (transport, not content) nor an LlmsOnlySource (no human-facing URL). A
service that attaches typed metadata to its discovered items — as MarkdownContentService
does — therefore participates with no extra code. Override only to project records that do
```

```
override this) to opt in.
    public IAsyncEnumerable<ContentRecord> GetRecordsAsync();      /// Redirect sources this service emits (each item's Source is a RedirectSource). Consumed by RedirectContentService to build the unified redirect map without iterating every service's DiscoverAsync — which would force services that have no redirects to pay the full cost of discovery just to return nothing. Default: empty. Services backed by front-matter records that implement IRedirectable override this.    public Task<ImmutableList<DiscoveredItem>> GetRedirectSourcesAsync();      /// Discovers and parses this service's content with the service's own front-matter type, yielding ParsedItems (typed metadata + body). Consumers like LlmsTxtService use this instead of re-parsing with a foreign parser, which would mis-flag valid keys from other content types. Default: empty — services whose content is sourced elsewhere (Razor/API pages fetched as rendered HTML) opt out.    public IAsyncEnumerable<ParsedItem> ParseContentAsync();      /// Relative priority for ordering results in the search index (higher values rank first).    public int SearchPriority { get; }  }
```

Pennington.Content.IFolderMetadataProvider

Pennington.Content.IFolderMetadataProvider

CSHARP

```
namespace Pennington.Content;

/// Optional capability for a IContentService to surface FolderMetadata rows discovered during its own scan (for example, _meta.yml sidecars under a markdown content tree).
public interface IFolderMetadataProvider
{
    /// Returns the folder-metadata rows declared by this provider's content.
    public Task<ImmutableList<FolderMetadata>> GetFolderMetadataAsync()
;
}
```

Pennington.Content.IMarkdownContentSource

Pennington.Content.IMarkdownContentSource

CSHARP

```
namespace Pennington.Content;

/// Non-generic capability interface exposing the root directory and base URL of a markdown
content source. Used by MarkdownLinkResolver to compute URL paths for relative asset
references without having to deal with the generic MarkdownContentService type, and by the
overlap detector to diagnose misconfigurations where two markdown sources claim overlapping
subtrees.
public interface IMarkdownContentSource
{
    /// Absolute path to the content directory (default locale root).
    public string AbsoluteContentRoot { get; }

    /// Base URL under which this source's content is served.
    public UrlPath BasePageUrl { get; }

    /// Normalized (forward-slash, lowercase) relative paths excluded from discovery and
content copying. Empty for sources that don't opt out of any subtree.
    public ImmutableArray<string> ExcludePaths { get; }

}
```

Pennington.Content.IMetaContentService

Pennington.Content.IMetaContentService

CSHARP

```
namespace Pennington.Content;

/// Marks an IContentService whose output is DERIVED from the other registered content
services — taxonomy axes, paginated listings, social-card routes, and the like — rather than
from its own source files. When such a service walks its siblings it must skip every other
meta-service, itself included, or two meta-services (or a transient self-registration whose
reference-equality self-check never matches) would recurse without end. Filter the sibling
set with ContentServiceExtensions.SourceServices before walking it.
public interface IMetaContentService
{
}
```

Pennington.Content.IPageResolver

Pennington.Content.IPageResolver

CSHARP

```
namespace Pennington.Content;

/// Resolves a requested URL to a fully rendered page by walking the registered content
services, parsing the matching item, and rendering it.
public interface IPageResolver
{
    /// Returns the rendered page whose canonical route matches requested, or null when
nothing matches or the match fails to parse or render.
    public Task<RenderedItem> ResolveAsync(UrlPath requested)
;
}
```

Pennington.Content.MarkdownContentService

Pennington.Content.MarkdownContentService

CSHARP

```
namespace Pennington.Content;

/// Discovers and provides markdown content from a directory. When LocalizationOptions has
multiple locales, also discovers content from locale subdirectories (e.g., Content/fr/,
Content/de/).
public class MarkdownContentService
{
    /// Absolute filesystem path to the root of this service's content directory.
    public string AbsoluteContentRoot { get; }

    /// URL prefix prepended to routes generated from this content directory.
    public UrlPath BasePageUrl { get; }

    /// Default section label applied to discovered items that do not supply one via front
matter.
    public string DefaultSectionLabel { get; }

    /// Discover all content items this service is responsible for.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Normalized forward-slash subtree paths (relative to the content root) that are
skipped during discovery.
    public ImmutableArray<string> ExcludePaths { get; }

    /// Sidecar filename that, when dropped at any folder under the content root, declares
folder metadata (display title, sort order, llms-subtree opt-in).
    public static const string FolderMetadataSidecarFileName
;

    /// Maps a file-change notification to the set of routes this service projects from that
file, without mutating any cached state. Consulted by file-watched caches (SiteProjection,
BuildHtmlCache) to invalidate only the affected entries instead of clearing wholesale.
Default: None.
    public ContentChangeImpact GetAffectedRoutes(FileChangeNotification change)
;

    /// Navigation entries for table of contents.
    public Task<ImmutableList<ContentTocItem>> GetContentTocEntriesAsync()
;

    /// Static files to copy to output (images, downloads, etc.)
    public Task<ImmutableList<ContentToCopy>> GetContentToCopyAsync()
;

    /// Cross-references for xref resolution.
    public Task<ImmutableList<CrossReference>> GetCrossReferencesAsync()
;

    /// Returns the folder-metadata rows declared by this provider's content.
    public Task<ImmutableList<FolderMetadata>> GetFolderMetadataAsync()
```

```
GetContentTocEntriesAsync. That is correct when "shown in navigation" ≡ "discoverable via search" — the default holds for markdown, because MarkdownContentService's TOC entries already honor search: and llms: front-matter fields via ExcludeFromSearch / ExcludeFromLlms.Override when the two sets diverge — for example, RazorPageContentService emits sidecar-less pages here (so users can search for them) without adding them to navigation (which would clutter the TOC with auto-titled entries).Implementors of custom content services that build ContentTocItems directly: set ExcludeFromSearch and ExcludeFromLlms from your metadata's Search / Llms flags, or per-page opt-outs will be silently ignored.
    public Task<ImmutableList<ContentTocItem>> GetIndexableEntriesAsync();      /// Returns the subtrees declared by this provider's content.    public Task<ImmutableList<LlmsSubtree>> GetLlmsSubtreesAsync();      /// Projects this service's routable content as ContentRecords — the discovery seam consumed by taxonomy, search faceting, and structured-data emission. Default: bridges from DiscoverAsync, yielding one record per discovered item that carries Metadata and is neither a RedirectSource (transport, not content) nor an LlmsOnlySource (no human-facing URL). A service that attaches typed metadata to its discovered items — as MarkdownContentService does — therefore participates with no extra code. Override only to project records that do not flow through DiscoverAsync, or to suppress records entirely.A service that emits routable content from DiscoverAsync but leaves Metadata unset projects no records, and so silently sits out of taxonomy, search faceting, and structured data. Set the metadata (or override this) to opt in.    public IAsyncEnumerable<ContentRecord> GetRecordsAsync();      /// Redirect sources this service emits (each item's Source is a RedirectSource). Consumed by RedirectContentService to build the unified redirect map without iterating every service's DiscoverAsync — which would force services that have no redirects to pay the full cost of discovery just to return nothing. Default: empty. Services backed by front-matter records that implement IRedirectable override this.    public Task<ImmutableList<DiscoveredItem>> GetRedirectSourcesAsync();      /// File-name suffix that marks a markdown file as llms-only — emitted to the llms.txt sidecar but never as an HTML page.    public static const string LlmsOnlyFileSuffix;      /// Initializes the service and prepares lazy metadata loading. FileWatchDispatcher watches the content directory and drives cache invalidation through OnFileChanged.    public MarkdownContentService`1(MarkdownContentServiceOptions options, FrontMatterParser parser, IFileSystem fileSystem, LocalizationOptions localization, TimeProvider clock = null, ILogger<MarkdownContentService<TFrontMatter>> logger = null);      /// Content-root file name reserved as the not-found page body when ReserveNotFoundPage is set.    public static const string NotFoundPageFileName;      /// Surgically updates the discovery caches when a file under this source's content directory changes. A markdown edit re-parses just that file's front matter; a _meta.yml change re-reads just that sidecar; an asset change is a no-op. Renames (where the watcher only carries the new path) fall back to a full re-seed. Changes outside the scope are ignored.    public FileWatchResponse OnFileChanged(FileChangeNotification change);      /// Discovers and parses this service's content with the service's own front-matter type, yielding ParsedItems (typed metadata + body). Consumers like LlmsTxtService use this instead of re-parsing with a foreign parser, which would mis-flag valid keys from other content types. Default: empty — services whose content is sourced elsewhere (Razor/API pages fetched as rendered HTML) opt out.    public IAsyncEnumerable<ParsedItem> ParseContentAsync();      /// Relative priority for ordering results in the search index (higher values rank first).    public int SearchPriority { get; }      /// Directories needing an OS-level watcher. Empty (the default) for aggregators that ride notifications other watchers already produce.    public IReadOnlyList<FileWatchScope> WatchScopes { get; }  }
```

Pennington.Content.MarkdownContentServiceOptions

Pennington.Content.MarkdownContentServiceOptions

CSHARP

```
namespace Pennington.Content;

/// Configuration for a markdown content source.
public class MarkdownContentServiceOptions
{
    /// URL prefix prepended to routes generated from this content directory.
    public UrlPath BasePageUrl { get; set; }

    /// Filesystem path to the directory containing markdown files.
    public FilePath ContentPath { get; set; }

    /// Relative paths (forward-slash, from ContentPath) whose subtrees should be skipped during discovery and content copying. Use this when another content source already owns a subfolder — e.g. a default DocFrontMatter source rooted at Content can set ExcludePaths = ["changelog"] so a specialized ChangelogFrontMatter source rooted at Content/changelog is the sole owner of that subtree. Matching is case-insensitive and segment-based: "a/b" excludes a/b and anything beneath it, but not a/bcd.
    public ImmutableArray<string> ExcludePaths { get; set; }

    /// Glob pattern used to enumerate source files (defaults to *.md).
    public string FilePattern { get; set; }

    /// Dispatch key stamped on this source's discovered items so the pipeline routes them to this source's front-matter parser rather than a shared one. The host assigns a distinct key per markdown source via SourceKey; defaults to the shared Key for standalone construction.
    public string Format { get; set; }

    /// Locale code associated with single-locale content (ignored when multi-locale routing is active).
    public string Locale { get; set; }

    /// When true, a content-root-level 404.md (and its per-locale {locale}/404.md sibling) is skipped during discovery so it never becomes a routable page or appears in navigation, the sitemap, search, llms.txt, or build output. Host templates (DocSite, BlogSite) render the file's body on demand as the not-found page instead. Nested 404.md files (e.g. guides/404.md) stay ordinary pages. Bare hosts leave this false, keeping 404.md a normal route.
    public bool ReserveNotFoundPage { get; set; }

    /// Relative ordering priority for this source's entries in the search index.
    public int SearchPriority { get; set; }

    /// Default section label applied to discovered items when front matter doesn't specify one.
    public string SectionLabel { get; set; }

}
```

Pennington.Content.MarkdownSourceOverlapDetector

Pennington.Content.MarkdownSourceOverlapDetector

CSHARP

```
namespace Pennington.Content;

/// Detects overlapping markdown content sources — pairs where one source's
AbsoluteContentRoot is a strict descendant directory of another's, AND the outer source does
not opt out of that subtree via ExcludePaths.  Overlap without an explicit carve-out is
almost always a misconfiguration: the outer (catch-all) source and the inner (specialized)
source both emit routes for the same canonical URLs, producing duplicate TOC entries in
every page's sidebar and file-lock races when two pipelines write the same output file. The
engine emits warnings rather than silently deduping so users can see and fix the misconfig.
public class MarkdownSourceOverlapDetector
{
    /// Returns one warning per unresolved overlap. Empty when everything is either disjoint
or explicitly carved out.
    public static ImmutableArray<string> DetectOverlaps(IEnumerable<IMarkdownContentSource>
sources)
;
}
```

Pennington.Content.PagedList

Pennington.Content.PagedList

CSHARP

```
namespace Pennington.Content;

/// A single page of items with the bookkeeping the Pagination component needs to render
prev/next controls and numbered page links.
public record PagedList
{
    /// True when a page exists after Page.
    public bool HasNext { get; }

    /// True when a page exists before Page.
    public bool HasPrevious { get; }

    /// Items in this page slice.
    public IReadOnlyList<T> Items { get; set; }

    /// 1-based page index this slice represents.
    public int Page { get; set; }

    /// A single page of items with the bookkeeping the Pagination component needs to render
prev/next controls and numbered page links.
    public PagedList`1(IReadOnlyList<T> Items, int Page, int PageSize, int TotalItems)
;

    /// Items per page.
    public int PageSize { get; set; }

    /// Total item count across all pages.
    public int TotalItems { get; set; }

    /// Total page count. At least 1 even when TotalItems is zero.
    public int TotalPages { get; }

}
```

Pennington.Content.PageResolver

Pennington.Content.PageResolver

CSHARP

```
namespace Pennington.Content;

/// Default IPageResolver over the registered content services, parser, and renderer.
public class PageResolver
{
    /// Creates the resolver from the registered content services and renderer. The parser is optional: a bare host that registers no markdown source has no IContentParser, so ResolveAsync resolves nothing.
    public PageResolver(IEnumerable<IContentService> services, IContentRenderer renderer, IContentParser parser = null)
;

    /// Returns the rendered page whose canonical route matches requested, or null when nothing matches or the match fails to parse or render.
    public Task<RenderedItem> ResolveAsync(UrlPath requested)
;
}
```

Pennington.Content.PageResolverExtensions

Pennington.Content.PageResolverExtensions

CSHARP

```
namespace Pennington.Content;

/// Convenience extensions over IPageResolver.
public class PageResolverExtensions
{
    /// Resolves requested and narrows the rendered page's front matter to TFrontMatter in one step, so callers read typed properties without a cast. Returns null when nothing matches, the match fails to render, or the matched page's front matter is not a TFrontMatter.
    public static Task<RenderedItem<TFrontMatter>> ResolveAsync<TFrontMatter>(IPageResolver resolver, UrlPath requested)
;
}
```

Pennington.Content.RazorPageContentService

Pennington.Content.RazorPageContentService

CSHARP

```
namespace Pennington.Content;

/// Discovers @page Razor components for the content pipeline. Scans configured assemblies
for types inheriting ComponentBase with non-parameterized [RouteAttribute] routes.
Optionally loads metadata from sidecar .razor.metadata.yml files placed alongside the
component.
public class RazorPageContentService
{
    /// Default section label applied to discovered items that do not supply one via front
matter.
    public string DefaultSectionLabel { get; }

    /// Discover all content items this service is responsible for.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Navigation entries for table of contents.
    public Task<ImmutableList<ContentTocItem>> GetContentTocEntriesAsync()
;

    /// Static files to copy to output (images, downloads, etc.)
    public Task<ImmutableList<ContentToCopy>> GetContentToCopyAsync()
;

    /// Cross-references for xref resolution.
    public Task<ImmutableList<CrossReference>> GetCrossReferencesAsync()
;

    /// Entries that should appear in the search index and llms.txt. Default: returns
GetContentTocEntriesAsync. That is correct when "shown in navigation" ≡ "discoverable via
search" — the default holds for markdown, because MarkdownContentService's TOC entries
already honor search: and llms: front-matter fields via ExcludeFromSearch /
ExcludeFromLlms.Override when the two sets diverge — for example, RazorPageContentService
emits sidecar-less pages here (so users can search for them) without adding them to
navigation (which would clutter the TOC with auto-titled entries).Implementors of custom
content services that build ContentTocItems directly: set ExcludeFromSearch and
ExcludeFromLlms from your metadata's Search / Llms flags, or per-page opt-outs will be
silently ignored.
    public Task<ImmutableList<ContentTocItem>> GetIndexableEntriesAsync()
;

    /// Razor @page directives that were missing a trailing slash. Populated after
DiscoverAsync runs.
    public IReadOnlyList<ValueTuple<string, string>> MissingTrailingSlashPages { get; }

    /// Initializes the service with the assemblies to scan for routable Razor components.
contentRootPath (the host content root) seeds the sidecar file scan alongside the assembly-
location walkup — required under a centralized artifacts output, where no .csproj lives
above the compiled binary.
    public RazorPageContentService(Assembly[] assemblies, IFileSystem fileSystem,
FrontMatterParser frontMatterParser, ILogger<RazorPageContentService> logger, string
```

```
    /// Relative priority for ordering results in the search index (higher values rank first).    public int SearchPriority { get; }  }
```

Pennington.Content.RedirectContentService

Pennington.Content.RedirectContentService

CSHARP

```
namespace Pennington.Content;

/// Holds the unified redirect map used by PenningtonRedirectMiddleware at dev time and by
the static build crawler's 301 handling at publish time. The map is the union of two
sources: A _redirects.yml file at the top of ContentRootPath.Every page whose front matter
sets redirectUrl: (see IRedirectable). DiscoverAsync emits those as RedirectSource items;
this service collects them by scanning registered IContentServices on first access. Both
sources flow through one middleware so dev and static build behave identically — a request
to the redirect source URL returns HTTP 301 with a meta-refresh body, and
OutputGenerationService writes that body to disk.
public class RedirectContentService
{
    /// Default section label applied to discovered items that do not supply one via front
matter.
    public string DefaultSectionLabel { get; }

    /// Yields one DiscoveredItem per _redirects.yml entry so the static build crawler hits
the source URL, gets the middleware's 301 response, and writes a meta-refresh HTML file at
that path. Per-page redirects already appear via their owning content service's
DiscoverAsync, so they are not re-emitted here.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Navigation entries for table of contents.
    public Task<ImmutableList<ContentTocItem>> GetContentTocEntriesAsync()
;

    /// Static files to copy to output (images, downloads, etc.)
    public Task<ImmutableList<ContentToCopy>> GetContentToCopyAsync()
;

    /// Cross-references for xref resolution.
    public Task<ImmutableList<CrossReference>> GetCrossReferencesAsync()
;

    /// Returns the merged source-URL → target-URL map (_redirects.yml entries plus front-
matter redirects). First call is eager — it enumerates every registered content service's
DiscoverAsync() to pick up RedirectSource items.
    public Task<ImmutableDictionary<string, string>> GetRedirectMappingsAsync()
;

    /// Initializes the service and prepares lazy loading of the unified redirect map.
    public RedirectContentService(IServiceProvider serviceProvider, PenningtonOptions
pennOptions, IFileSystem fileSystem, IFileWatcher fileWatcher)
;

    /// Relative priority for ordering results in the search index (higher values rank
first).
    public int SearchPriority { get; }

}
```

Pennington.Content.RenderedBlogPost

Pennington.Content.RenderedBlogPost

CSHARP

```
namespace Pennington.Content;

/// A blog post's front matter, canonical URL, and rendered HTML body.
public record RenderedBlogPost
{
    /// Parsed front matter for the post.
    public TFrontMatter FrontMatter { get; set; }

    /// Rendered HTML body.
    public string Html { get; set; }

    /// A blog post's front matter, canonical URL, and rendered HTML body.
    public RenderedBlogPost`1(TFrontMatter FrontMatter, UrlPath Url, string Html)
;

    /// Canonical URL of the post.
    public UrlPath Url { get; set; }

}
```

Pennington.Data.DataDirectoryEntry

Pennington.Data.DataDirectoryEntry

CSHARP

```
namespace Pennington.Data;

/// Singleton holder for a directory of data files, aggregating every .yml, .yaml, and .json
file into one list of TItem. The first call to GetValue enumerates the directory and
deserializes each file via DataFileLoader; subsequent calls return the cached list until
FileWatchDispatcher reports a change in the directory, at which point the next access
reloads it.
public class DataDirectoryEntry
{
    /// Creates the entry; the directory itself is not read until GetValue is called.
    public DataDirectoryEntry`1(string name, string path, IFileSystem fileSystem)
;

    /// Returns the current loaded value, refreshed if the underlying file has changed since
last access.
    public object GetValue()
;

    /// Logical name supplied at registration; lookup key for Get.
    public string Name { get; }

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;

    /// The closed generic type the entry was registered with.
    public Type ValueType { get; }

    /// Directories needing an OS-level watcher. Empty (the default) for aggregators that
ride notifications other watchers already produce.
    public IReadOnlyList<FileWatchScope> WatchScopes { get; }

}
```

Pennington.Data.DataFileEntry

Pennington.Data.DataFileEntry

CSHARP

```
namespace Pennington.Data;

/// Singleton holder for a single registered data file. The first call to GetValue loads and
deserializes the file via DataFileLoader; subsequent calls return the cached value until
FileWatchDispatcher reports the file has changed, at which point the next access reloads it.
public class DataFileEntry
{
    /// Creates the entry; the file itself is not read until GetValue is called.
    public DataFileEntry`1(string name, string path, IFileSystem fileSystem)
;

    /// Returns the current loaded value, refreshed if the underlying file has changed since
last access.
    public object GetValue()
;

    /// Logical name supplied at registration; lookup key for Get.
    public string Name { get; }

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;

    /// The closed generic type the entry was registered with.
    public Type ValueType { get; }

    /// Directories needing an OS-level watcher. Empty (the default) for aggregators that
ride notifications other watchers already produce.
    public IReadOnlyList<FileWatchScope> WatchScopes { get; }

}
```

Pennington.Data.DataFileLoader

Pennington.Data.DataFileLoader

CSHARP

```
namespace Pennington.Data;

/// Deserializes a data file's bytes into a typed value. Format is chosen from the extension
(.yml / .yaml use SharpYaml, .json uses System.Text.Json), both configured with camelCase
naming and case-insensitive property matching to mirror FrontMatterParser's behavior.
Arbitrary data types deserialize via reflection (no source-gen context covers them).
public class DataFileLoader
{
    /// Reads absolutePath through fileSystem and deserializes it into T.
    public static T Load<T>(string absolutePath, IFileSystem fileSystem)
;

    /// Reads absolutePath through fileSystem and deserializes it into a list of TItem. A
file whose root is an array yields its elements; a file whose root is a single record yields
a one-element list.
    public static IReadOnlyList<TItem> LoadMany<TItem>(string absolutePath, IFileSystem
fileSystem)
;
}
```

Pennington.Data.DataFiles

Pennington.Data.DataFiles

CSHARP

```
namespace Pennington.Data;

/// Default IDataFiles implementation that aggregates every registered IDataFile by name.
Also the single IFileWatchAware for the data-file subsystem: it fans changes out to its
entries.
public class DataFiles
{
    /// Creates the registry from every IDataFile registered in DI.
    public DataFiles(IEnumerable<IDataFile> entries)
;

    /// Retrieves the loaded value for the data file registered under name.
    public T Get<T>(string name)
;

    /// The names of all registered data files.
    public IEnumerable<string> Names { get; }

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;

    /// Tries to retrieve the loaded value for the data file registered under name. Returns
false when no data file is registered with that name OR when the registered type does not
match T.
    public bool TryGet<T>(string name, out T value)
;

    /// Directories needing an OS-level watcher. Empty (the default) for aggregators that
ride notifications other watchers already produce.
    public IReadOnlyList<FileWatchScope> WatchScopes { get; }

}
```

Pennington.Data.DataFileServiceExtensions

Pennington.Data.DataFileServiceExtensions

CSHARP

```
namespace Pennington.Data;

/// DI helpers for registering YAML/JSON data files that hot-reload when the underlying file changes on disk.
public class DataFileServiceExtensions
{
    /// Registers every .yml, .yaml, and .json file in path as a single aggregated IReadOnlyList accessible through IDataFiles under the lookup key name. Each file contributes one record, or several when its root is an array; files are ordered by name. Edits, additions, and removals in the directory invalidate the cached value so the next read returns the fresh content.
    public static IServiceCollection AddDataDirectory<TItem>(IServiceCollection services, string name, string path)
;

    /// Registers path as a data file accessible through IDataFiles under the lookup key name. Format is inferred from the file extension (.yml, .yaml, .json). Edits to the file invalidate the cached value so the next read returns the fresh content.
    public static IServiceCollection AddDataFile<T>(IServiceCollection services, string name, string path)
;
}
```

Pennington.Data.IDataFile

Pennington.Data.IDataFile

CSHARP

```
namespace Pennington.Data;

/// Non-generic facade over a single registered data file. Used by DataFiles to enumerate
every DataFileEntry the container has registered without reflecting over the closed generic
types. Reloads through IFileWatchAware.
public interface IDataFile
{
    /// Returns the current loaded value, refreshed if the underlying file has changed since
last access.
    public object GetValue()
;

    /// Logical name supplied at registration; lookup key for Get.
    public string Name { get; }

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;

    /// The closed generic type the entry was registered with.
    public Type ValueType { get; }

    /// Directories needing an OS-level watcher. Empty (the default) for aggregators that
ride notifications other watchers already produce.
    public IReadOnlyList<FileWatchScope> WatchScopes { get; }

}
```

Pennington.Data.IDataFiles

Pennington.Data.IDataFiles

CSHARP

```
namespace Pennington.Data;

/// Typed, name-keyed access to data files registered with AddDataFile. Data files reload automatically when the underlying file changes on disk.
public interface IDataFiles
{
    /// Retrieves the loaded value for the data file registered under name.
    public T Get<T>(string name)
;

    /// The names of all registered data files.
    public IEnumerable<string> Names { get; }

    /// Tries to retrieve the loaded value for the data file registered under name. Returns false when no data file is registered with that name OR when the registered type does not match T.
    public bool TryGet<T>(string name, out T value)
;
}
```

Pennington.Diagnostics.Diagnostic

Pennington.Diagnostics.Diagnostic

CSHARP

```
namespace Pennington.Diagnostics;

/// A diagnostic produced during HTTP request handling. Route-agnostic — the route is known by the request context, not the diagnostic.
public record Diagnostic
{
    /// A diagnostic produced during HTTP request handling. Route-agnostic — the route is known by the request context, not the diagnostic.
    public Diagnostic(DiagnosticSeverity Severity, string Message, string Source = null)
;

    /// Human-readable description of the problem.
    public string Message { get; set; }

    /// Severity of the diagnostic.
    public DiagnosticSeverity Severity { get; set; }

    /// Optional identifier for the component that raised the diagnostic.
    public string Source { get; set; }

}
```

Pennington.Diagnostics.DiagnosticContext

Pennington.Diagnostics.DiagnosticContext

CSHARP

```
namespace Pennington.Diagnostics;

/// Scoped service that accumulates diagnostics for a single HTTP request. Registered as
scoped in DI — fresh instance per request, no thread-safety needed.
public class DiagnosticContext
{
    /// Appends a pre-built diagnostic to the context.
    public void Add(Diagnostic diagnostic)
;

    /// Records an error-severity diagnostic with the given message and optional source
label.
    public void AddError(string message, string source = null)
;

    /// Records an info-severity diagnostic with the given message and optional source
label.
    public void AddInfo(string message, string source = null)
;

    /// Records a warning-severity diagnostic with the given message and optional source
label.
    public void AddWarning(string message, string source = null)
;

    /// Diagnostics accumulated for the current request, in insertion order.
    public IReadOnlyList<Diagnostic> Diagnostics { get; }

    /// True when at least one diagnostic has been recorded.
    public bool HasAny { get; }

    /// True when at least one recorded diagnostic has Error severity.
    public bool HasErrors { get; }

}
```

Pennington.Diagnostics.DiagnosticSeverity

Pennington.Diagnostics.DiagnosticSeverity

CSHARP

```
namespace Pennington.Diagnostics;

/// Severity levels for a request-scoped Diagnostic.
public enum DiagnosticSeverity
{
    /// Failure that indicates broken content or misconfiguration.
    public static const DiagnosticSeverity Error
;

    /// Informational notice about degraded but non-broken behavior.
    public static const DiagnosticSeverity Info
;

    /// Potential issue that does not block rendering.
    public static const DiagnosticSeverity Warning
;
}
```

Pennington.DocSite.Api.ApiReferenceRegistration

Pennington.DocSite.Api.ApiReferenceRegistration

CSHARP

```
namespace Pennington.DocSite.Api;

/// One named API-reference tree: a provider key and the public URL prefix under which its pages are served.
public record ApiReferenceRegistration
{
    /// One named API-reference tree: a provider key and the public URL prefix under which its pages are served.
    public ApiReferenceRegistration(string Name, string RoutePrefix, string TocTitle, string TocSectionLabel, int SearchPriority)
;

    /// Registration name. Matches the name argument passed to the metadata backend's AddApiMetadataFrom* call.
    public string Name { get; set; }

    /// Normalized URL prefix with leading and trailing slashes, e.g. "/api/" or "/reference/api/spectre/".
    public string RoutePrefix { get; set; }

    /// Search-index priority applied to every page under RoutePrefix; lower ranks later.
    public int SearchPriority { get; set; }

    /// Section label the TOC entry groups under, or null to keep it unsectioned.
    public string TocSectionLabel { get; set; }

    /// Sidebar title for the index page's TOC entry, or null to suppress the entry.
    public string TocTitle { get; set; }

}
```

Pennington.DocSite.Api.ApiReferenceRegistrationOptions

Pennington.DocSite.Api.ApiReferenceRegistrationOptions

CSHARP

```
namespace Pennington.DocSite.Api;

/// Per-registration options passed to AddApiReference.
public class ApiReferenceRegistrationOptions
{
    /// URL prefix under which this reference tree's pages are served. Leading and trailing slashes are normalized automatically. Examples: "/reference/api/", "/api/", "/api/spectre-cli/". Defaults to "/reference/api/".
    public string RoutePrefix { get; set; }

    /// Search priority for every page under this tree's RoutePrefix, registered into SearchIndexOptions.PrefixPriorities so it populates the index p field. Lower ranks later. Defaults to 3 — below typical prose so generated reference pages don't bury conceptual articles that match the same term.
    public int SearchPriority { get; set; }

    /// Section label the TOC entry is grouped under in the sidebar. Leave null to keep the entry unsectioned. Set to a string that matches another page's sectionLabel to join that section.
    public string TocSectionLabel { get; set; }

    /// Sidebar title for the single TOC entry pointing at this registration's index page. Set to null to suppress the TOC entry entirely (the pages still publish). Defaults to "API reference".
    public string TocTitle { get; set; }

}
```

Pennington.DocSite.Api.ApiReferenceServiceExtensions

Pennington.DocSite.Api.ApiReferenceServiceExtensions

CSHARP

```
namespace Pennington.DocSite.Api;

/// DI extension methods for registering the API reference package.
public class ApiReferenceServiceExtensions
{
    /// Registers one named API-reference tree. Call once per library you want to document. Each call pairs with a matching AddApiMetadataFrom*(name, …) provider registration and publishes its type pages at the configured RoutePrefix.
    public static IServiceCollection AddApiReference(IServiceCollection services, string name = "default", Action<ApiReferenceRegistrationOptions> configure = null)
;
}
```

Pennington.DocSite.Api.Components.Reference.ApiDefinitionRow

Pennington.DocSite.Api.Components.Reference.ApiDefinitionRow

CSHARP

```
namespace Pennington.DocSite.Api.Components.Reference;

/// One row in an ApiDefinitionList: a named definition with a type, an optional default
value, an optional "required" flag, and a description.
public record ApiDefinitionRow
{
    /// One row in an ApiDefinitionList: a named definition with a type, an optional default
value, an optional "required" flag, and a description.
    public ApiDefinitionRow(string Name, string TypeDisplay, string DefaultValue, bool
Required, string DescriptionHtml)
;

    /// Default value as it would appear in source, or null when no default applies.
    public string DefaultValue { get; set; }

    /// Pre-rendered HTML for the description column (already escaped).
    public string DescriptionHtml { get; set; }

    /// Identifier shown in the Name column.
    public string Name { get; set; }

    /// When true, the row renders a "required" badge.
    public bool Required { get; set; }

    /// Human-readable type, shown in the Type column.
    public string TypeDisplay { get; set; }

}
```

Pennington.DocSite.Api.Components.Reference.ApiFetch

Pennington.DocSite.Api.Components.Reference.ApiFetch

CSHARP

```
namespace Pennington.DocSite.Api.Components.Reference;

/// Result of Fetch: a success flag with either the value or inline error HTML.
public struct ApiFetch
{
    /// Result of Fetch: a success flag with either the value or inline error HTML.
    public ApiFetch`1(bool Ok, T Value, string ErrorHtml)
;

    /// Inline error HTML when Ok is false; otherwise null.
    public string ErrorHtml { get; set; }

    /// Whether the fetch succeeded.
    public bool Ok { get; set; }

    /// The fetched value when Ok is true; otherwise the default.
    public T Value { get; set; }

}
```

Pennington.DocSite.BlogFeature

Pennington.DocSite.BlogFeature

CSHARP

```
namespace Pennington.DocSite;

/// Marker resolved by the DocSite chrome and blog pages to know whether the blog is active.
The blog activates only when markdown articles exist under the content project's blog folder
at startup.
public record BlogFeature
{
    /// Marker resolved by the DocSite chrome and blog pages to know whether the blog is
active. The blog activates only when markdown articles exist under the content project's
blog folder at startup.
    public BlogFeature(bool Enabled)
;

    /// True when the blog content folder contains at least one post.
    public bool Enabled { get; set; }

}
```

Pennington.DocSite.BlogPostFrontMatter

Pennington.DocSite.BlogPostFrontMatter

CSHARP

```
namespace Pennington.DocSite;

/// Front matter for blog posts under the content project's blog folder. Bound by AddDocSite when the blog is active. Implements IFrontMatter, ITaggable, IRedirectable, IStandardSiteDocument, and IHasStructuredData (emits a schema.org Article with the post's date and author).
public record BlogPostFrontMatter
{
    /// Record key of this post's published site.standard.document record (Standard Site), if any.
    public string AtprotoRkey { get; set; }

    /// Author name shown in the post byline and RSS feed.
    public string Author { get; set; }

    /// Publication date. Posts are ordered by this date, newest first.
    public DateTime? Date { get; set; }

    /// Short description used for the meta description and post listings.
    public string Description { get; set; }

    /// Returns the schema.org entities to emit on the page. Implementations typically yield one Article/Recipe/Product/etc. built from front matter values, plus the CanonicalUrl the template supplies.
    public IEnumerable<JsonLdEntity> GetStructuredData(StructuredDataContext context)
;

    /// When true, the post is skipped during production builds.
    public bool IsDraft { get; set; }

    /// When false, the post is excluded from the generated llms.txt output.
    public bool Llms { get; set; }

    /// When set, the post emits a client-side redirect to this URL instead of normal content.
    public string RedirectUrl { get; set; }

    /// When false, the post is excluded from the search index.
    public bool Search { get; set; }

    /// Always true: posts are indexed for search and llms.txt but kept out of the documentation navigation sidebar. Not author-settable — the blog has its own index and tag pages.
    public bool SearchOnly { get; }

    /// Tags applied to the post for the tag index and browse-by-tag pages.
    public string[] Tags { get; set; }

    /// Post title rendered in the browser tab and post heading.
    public string Title { get; set; }

    /// Stable identifier used for cross-references ([text](xref:uid)).
```

```
    public string Uid { get; set; }  }
```

Pennington.DocSite.ContentArea

Pennington.DocSite.ContentArea

CSHARP

```
namespace Pennington.DocSite;

/// Defines a content area within a documentation site. Each area maps to a top-level
directory and URL prefix, and gets its own TOC section.
public record ContentArea
{
    /// Defines a content area within a documentation site. Each area maps to a top-level
directory and URL prefix, and gets its own TOC section.
    public ContentArea(string Title, string Slug, string Icon = null, int? SearchBoost =
default)
;

    /// Optional SVG or markup for an icon beside the title.
    public string Icon { get; set; }

    /// Optional search ranking boost for this area, relative to
SearchIndexOptions.DefaultPriority (positive promotes, negative demotes). When null, the
area is auto-boosted by its position in the list — earlier areas weigh more — so task-
oriented docs lead over reference when matches are comparable. Set it to override that
default for a specific area.
    public int? SearchBoost { get; set; }

    /// URL path prefix / top-level directory name (e.g., "getting-started").
    public string Slug { get; set; }

    /// Display name shown in the area selector (e.g., "Getting Started").
    public string Title { get; set; }

}
```

Pennington.DocSite.DocSiteFrontMatter

Pennington.DocSite.DocSiteFrontMatter

CSHARP

```
namespace Pennington.DocSite;

/// Front matter bound by AddDocSite. Extends the DocFrontMatter shape with RedirectUrl via
IRedirectable. Implements IFrontMatter, ITaggable, ISectionable, IOrderable, IRedirectable,
and IHasStructuredData (emits a schema.org Article).
public record DocSiteFrontMatter
{
    /// Short description used for the meta description and social cards.
    public string Description { get; set; }

    /// Returns the schema.org entities to emit on the page. Implementations typically yield
one Article/Recipe/Product/etc. built from front matter values, plus the CanonicalUrl the
template supplies.
    public IEnumerable<JsonLdEntity> GetStructuredData(StructuredDataContext context)
;

    /// When true, the page is skipped during production builds.
    public bool IsDraft { get; set; }

    /// When false, the page is excluded from the generated llms.txt output.
    public bool Llms { get; set; }

    /// Sort order within the containing section. Lower values appear first.
    public int Order { get; set; }

    /// When set, the page emits a client-side redirect to this URL instead of normal
content.
    public string RedirectUrl { get; set; }

    /// When false, the page is excluded from the search index.
    public bool Search { get; set; }

    /// When true, the page is indexed for search/llms but hidden from the rendered
navigation tree.
    public bool SearchOnly { get; set; }

    /// Section heading this page belongs under in navigation.
    public string SectionLabel { get; set; }

    /// Tags applied to this page for filtering and the tag index.
    public string[] Tags { get; set; }

    /// Page title rendered in the browser tab and page heading.
    public string Title { get; set; }

    /// Stable identifier used for cross-references ([text](xref:uid)).
    public string Uid { get; set; }

}
```

Pennington.DocSite.DocSiteHttpContextKeys

Pennington.DocSite.DocSiteHttpContextKeys

CSHARP

```
namespace Pennington.DocSite;

/// Well-known keys for Items shared across DocSite and its integrations.
public class DocSiteHttpContextKeys
{
    /// Pre-rewrite public request path. Middleware that rewrites Path for internal routing stashes the caller-visible path here so the layout can resolve the active area and TOC selection against the URL the user actually sees.
    public static const string OriginalPath
;
}
```

Pennington.DocSite.DocSiteOptions

Pennington.DocSite.DocSiteOptions

CSHARP

```
namespace Pennington.DocSite;

/// Options record passed to AddDocSite that configures the DocSite template: site chrome,
typography, color scheme, content areas, and escape-hatch callbacks for the underlying
PenningtonOptions and MonorailCssOptions.
public record DocSiteOptions
{
    /// Additional raw HTML appended to the document <head> (for analytics, meta tags,
etc.).
    public string AdditionalHtmlHeadContent { get; set; }

    /// Additional assemblies scanned for Razor components so out-of-project pages
participate in routing.
    public Assembly[] AdditionalRoutingAssemblies { get; set; }

    /// Content areas for the documentation site. When empty or containing a single area, no
area selector is shown. Each area's slug must match a top-level directory name under
ContentRootPath.
    public IReadOnlyList<ContentArea> Areas { get; set; }

    /// CSS font-family stack used for body copy.
    public string BodyFontFamily { get; set; }

    /// Absolute base URL used when emitting canonical links, sitemap entries, and absolute
feed URLs.
    public string CanonicalBaseUrl { get; set; }

    /// Color scheme driving the MonorailCSS theme. Defaults to the built-in DocSite palette
when null.
    public IColorScheme ColorScheme { get; set; }

    /// Configure localization options (locales, default locale).
    public Action<LocalizationOptions> ConfigureLocalization { get; set; }

    /// Escape hatch for additional content wiring: callback invoked against the underlying
PenningtonOptions after DocSite's own defaults are applied. Use to register extra
AddMarkdownContent sources, add highlighters, register islands, etc., without dropping to
bare AddPennington.
    public Action<PenningtonOptions> ConfigurePennington { get; set; }

    /// Root folder (relative to the content project) that holds the markdown and razor
content tree.
    public FilePath ContentRootPath { get; set; }

    /// Override the CSS selector used to scope the shared site projection to a page region.
The same selector drives the search index, llms.txt sidecars, and build-time link audit, so
chrome (navigation, footers) is stripped once. Default is #main-content — the element
wrapping the article in the stock DocSite layout. Set to an empty string to project the
whole page body, or to a custom selector when you've replaced the layout.
    public string ContentSelector { get; set; }
```

```
get;set;}      /// CSS font-family stack used for display type (headings and hero copy).
public string DisplayFontFamily { get; set; }      /// Wraps the baseline
ProseCustomization. Forwarded to ExtendProseCustomization.    public
Func<ProseCustomization, ProseCustomization> ExtendProseCustomization { get; set; }      ///
Additional CSS appended to the generated stylesheet.    public string ExtraStyles { get;
set; }      /// Favicon / icon links. Forwarded to Favicons.    public FaviconOptions
Favicons { get; set; }      /// Fonts to preload via <link rel="preload"> for faster first
paint.    public FontPreload[] FontPreloads { get; set; }      /// Content rendered in the
site footer. Assign a raw HTML string or a RenderFragment — both convert implicitly. Strings
are emitted as raw HTML, not markdown.    public MarkupContent? FooterContent { get; set; }
/// URL to the project's GitHub repository. When set, a GitHub link is shown in the header.
public string GitHubUrl { get; set; }      /// The header brand area (logo + title). Assign
a raw HTML string or a RenderFragment — both convert implicitly. When set, it replaces the
default icon and SiteTitle link outright, giving total control over that region; when null,
the default document icon and title link render. Strings are emitted as raw HTML, not
markdown.    public MarkupContent? HeaderContent { get; set; }      /// CSS font-family
stack used for monospaced contexts (code blocks, inline code, kbd).    public string
MonoFontFamily { get; set; }      /// Short description used in the meta description tag and
default OpenGraph description.    public string SiteDescription { get; set; }      /// Title
displayed in the site chrome and used as a default for OpenGraph tags.    public string
SiteTitle { get; set; }      /// Enables generated per-page social cards. When set, each
page emits an og:image/ twitter:image pointing at an on-demand-rendered card (and the site-
wide SocialImageUrl default steps aside). The host supplies the drawing via Render.
public SocialCardOptions SocialCards { get; set; }      /// Default social-share image URL
used when a page does not specify its own.    public string SocialImageUrl { get; set; }
/// Standard Site (AT Protocol) integration. Forwarded to StandardSite.    public
StandardSiteOptions StandardSite { get; set; }      /// Syntax-highlight color palette used
by .hljs-* token classes. Defaults to Default when null. Values may reference custom palette
names registered via ColorScheme.    public SyntaxTheme SyntaxTheme { get; set; }  }
```

Pennington.DocSite.DocSiteServiceExtensions

Pennington.DocSite.DocSiteServiceExtensions

CSHARP

```
namespace Pennington.DocSite;

/// DI extension methods for registering and running the DocSite template.
public class DocSiteServiceExtensions
{
    /// Registers DocSite services with the provided options.
    public static IServiceCollection AddDocSite(IServiceCollection services,
Func<DocSiteOptions> configureOptions)
;

    /// Runs the DocSite: either serves the app or performs a static build, based on
command-line args.
    public static Task RunDocSiteAsync(WebApplication app, string[] args)
;

    /// Wires DocSite middleware and Razor components into the request pipeline.
    public static WebApplication UseDocSite(WebApplication app)
;
}
```

Pennington.DocSite.Services.BlogContentService

Pennington.DocSite.Services.BlogContentService

CSHARP

```
namespace Pennington.DocSite.Services;

/// Surfaces the blog index route (/blog) the static build cannot otherwise discover, and declares the /blog/ llms.txt subtree. Browse-by-tag routes come from the registered AddTaxonomy<BlogPostFrontMatter, string> axis; post pages from the markdown source; post data, pagination, and RSS from the shared BlogPostQuery. Stateless — it reads no files and holds no cache.
public class BlogContentService
{
    /// Default section label applied to discovered items that do not supply one via front matter.
    public string DefaultSectionLabel { get; }

    /// Discover all content items this service is responsible for.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Navigation entries for table of contents.
    public Task<ImmutableList<ContentTocItem>> GetContentTocEntriesAsync()
;

    /// Static files to copy to output (images, downloads, etc.)
    public Task<ImmutableList<ContentToCopy>> GetContentToCopyAsync()
;

    /// Cross-references for xref resolution.
    public Task<ImmutableList<CrossReference>> GetCrossReferencesAsync()
;

    /// Declares /blog/ as an llms.txt subtree so posts split out of the front-door llms.txt into a dedicated /blog/llms.txt. Always declared — the service is only registered when the content project has a blog folder.
    public Task<ImmutableList<LlmsSubtree>> GetLlmsSubtreesAsync()
;

    /// Relative priority for ordering results in the search index (higher values rank first).
    public int SearchPriority { get; }

}
```

Pennington.DocSite.Services.DocSiteContentResolver

Pennington.DocSite.Services.DocSiteContentResolver

CSHARP

```
namespace Pennington.DocSite.Services;

/// The DocSite's per-request content facade. Resolves a page by URL (delegating the
discover → parse → render step to the core IPageResolver) and adds the DocSite-specific
concerns around it: locale detection with fallback to the default locale, the
ResolvedContent view-model, navigation/TOC, alternate languages, and area scoping. Distinct
from IPageResolver, which is the locale-naive single-page primitive shared with bare hosts.
public class DocSiteContentResolver
{
    /// Creates a new resolver with the supplied content services, page resolver, options,
and pipeline primitives.
    public DocSiteContentResolver(IEnumerable<IContentService> services, IPageResolver
pageResolver, NavigationBuilder navBuilder, LocalizationOptions localization, DocSiteOptions
docSiteOptions, BlogFeature blog, IContentParser parser = null, IContentRenderer renderer =
null)
;

    /// Get alternate language versions for a page URL. Always includes all configured
locales — fallback resolution handles missing translations. Delegates the URL math to
GetAlternateLanguages.
    public Task<ImmutableList<AlternateLanguage>> GetAlternateLanguagesAsync(string url)
;

    /// Get rendered content for a URL. Returns null if not found. Handles locale detection
and fallback to default locale.
    public Task<ResolvedContent> GetContentByUrlAsync(string url)
;

    /// Get navigation info for a URL, filtered by locale.
    public Task<NavigationInfo> GetNavigationInfoAsync(string url)
;

    /// Get navigation info (prev/next/breadcrumbs) scoped to an area. When area is null,
falls back to the full-site navigation.
    public Task<NavigationInfo> GetNavigationInfoForAreaAsync(string url, ContentArea area)
;

    /// Resolves the site's not-found body from a content-root 404.md, rendered through the
full markdown pipeline. Returns null when no 404.md exists (the catch-all then tries a
NotFound component, then the built-in message) or when the host registered no markdown
parser. The file is reserved out of discovery (ReserveNotFoundPage), so it is never a
routable page. One body serves every locale: the static build emits a single root 404.html,
which is all any static host serves for an unknown URL.
    public Task<ResolvedContent> GetNotFoundContentAsync()
;

    /// Get all TOC items, optionally filtered by locale.
    public Task<IReadOnlyList<ContentTocItem>> GetTocItemsAsync(string locale = null)
;

    /// Get TOC items scoped to a specific area. Filters by area slug matching
```

```
ContentAreaarea);      /// Resolves which content area the given URL belongs to, based on
the first path segment matching a configured area slug. Returns null if no area matches.
public ContentArea ResolveCurrentArea(string url);}
```

Pennington.DocSite.Services.ResolvedContent

Pennington.DocSite.Services.ResolvedContent

CSHARP

```
namespace Pennington.DocSite.Services;

/// Rendered content plus the surrounding locale/fallback metadata needed to render a page.
public record ResolvedContent
{
    /// Page description from front matter.
    public string Description { get; set; }

    /// Display name for the locale actually served, shown in the fallback notice.
    public string FallbackDefaultDisplayName { get; set; }

    /// Display name for the requested locale, shown in the fallback notice.
    public string FallbackRequestedDisplayName { get; set; }

    /// Rendered HTML body.
    public string Html { get; set; }

    /// True when content from the default locale was served because the requested locale had no match.
    public bool IsFallback { get; set; }

    /// Locale used to render the page (may differ from the requested locale when falling back).
    public string Locale { get; set; }

    /// Parsed front matter for the page.
    public IFrontMatter Metadata { get; set; }

    /// Headings extracted from the body for the on-page outline.
    public OutlineEntry[] Outline { get; set; }

    /// Locale the user originally requested, when different from Locale.
    public string RequestedLocale { get; set; }

    /// Rendered content plus the surrounding locale/fallback metadata needed to render a page.
    public ResolvedContent(ContentRoute Route, string Title, string Description, string Html, OutlineEntry[] Outline, IFrontMatter Metadata, string Locale = "", bool IsFallback = false, string RequestedLocale = null, string FallbackRequestedDisplayName = null, string FallbackDefaultDisplayName = null)
    ;

    /// Canonical route for the resolved page.
    public ContentRoute Route { get; set; }

    /// Page title from front matter.
    public string Title { get; set; }

}
```

Pennington.Favicon.FaviconLink

Pennington.Favicon.FaviconLink

CSHARP

```
namespace Pennington.Favicon;

/// One icon <link>. The generic Rel/Href/Type/ Sizes/Color model covers rel="icon"
(multiple sizes/types), apple-touch-icon, mask-icon, and manifest without special-casing.
public record FaviconLink
{
    /// The color attribute used by rel="mask-icon", or null to omit.
    public string Color { get; set; }

    /// One icon <link>. The generic Rel/Href/Type/ Sizes/Color model covers rel="icon"
(multiple sizes/types), apple-touch-icon, mask-icon, and manifest without special-casing.
    public FaviconLink(string Href)
;

    /// The icon URL. A root-relative href is sub-path prefixed at serve/build time; an
absolute href is left as-is.
    public string Href { get; set; }

    /// The rel attribute; defaults to icon.
    public string Rel { get; set; }

    /// The sizes attribute (e.g. 32x32, any), or null to omit.
    public string Sizes { get; set; }

    /// The MIME type; inferred from the Href extension when null.
    public string Type { get; set; }

}
```

Pennington.Favicon.FaviconOptions

Pennington.Favicon.FaviconOptions

CSHARP

```
namespace Pennington.Favicon;

/// Host configuration for favicon / icon <link> tags emitted into the document head. The
icon files themselves are user-provided static assets served and copied by the content-root
static-files mechanism; this only emits the discovery markup. Set this on Favicons
(templates forward it from their own options) to enable the feature; leaving it null
disables it.
public record FaviconOptions
{
    /// The icon links to emit, in document order. Empty means nothing is emitted.
    public ImmutableArray<FaviconLink> Icons { get; set; }

}
```

Pennington.Feeds.RssFeedItem

Pennington.Feeds.RssFeedItem

CSHARP

```
namespace Pennington.Feeds;

/// Single entry in a generated RSS feed.
public record RssFeedItem
{
    /// Optional author name or email.
    public string Author { get; set; }

    /// Optional summary or excerpt.
    public string Description { get; set; }

    /// Publication date, when known.
    public DateTime? PublishDate { get; set; }

    /// Single entry in a generated RSS feed.
    public RssFeedItem(string Title, string Description, UrlPath Url, DateTime? PublishDate,
string Author)
;

    /// Item title shown in the feed.
    public string Title { get; set; }

    /// Site-relative canonical path of the entry; RssFeedWriter composes the absolute link.
    public UrlPath Url { get; set; }

}
```

Pennington.Feeds.RssFeedWriter

Pennington.Feeds.RssFeedWriter

CSHARP

```
namespace Pennington.Feeds;

/// Serializes a set of RssFeedItem to an RSS 2.0 document for the /rss.xml endpoint. Items without a publish date are dropped; the rest are ordered newest-first and their links composed against the feed base URL.
public class RssFeedWriter
{
    /// Builds the RSS 2.0 XML for a feed.
    public static string WriteXml(string siteTitle, string siteDescription, string canonicalBaseUrl, IEnumerable<RssFeedItem> items)
;
}
```

Pennington.Feeds.SitemapBuilder

Pennington.Feeds.SitemapBuilder

CSHARP

```
namespace Pennington.Feeds;

/// Builds sitemap entries from a set of SitemapCandidate rows.
public class SitemapBuilder
{
    /// Build sitemap entries from candidate rows. Excludes drafts and future-dated (scheduled) entries.
    public ImmutableList<SitemapEntry> Build(IReadOnlyList<SitemapCandidate> candidates)
;

    /// Canonical site base URL used when resolving absolute entry URLs.
    public UrlPath CanonicalBase { get; }

    /// Filters candidates down to the rows that actually belong in the sitemap — drafts, future-dated (scheduled), and redirect rows removed. Use this when deriving sitemap-adjacent data (e.g. hreflang alternates) so it stays in lockstep with Build.
    public IReadOnlyList<SitemapCandidate> Publishable(IReadOnlyList<SitemapCandidate> candidates)
;

    /// Initializes the builder with the canonical site base URL used to produce absolute entry URLs and the wall clock used to filter out future-dated (scheduled) entries.
    public SitemapBuilder(UrlPath canonicalBase, TimeProvider clock = null)
;
}
```

Pennington.Feeds.SitemapCandidate

Pennington.Feeds.SitemapCandidate

CSHARP

```
namespace Pennington.Feeds;

/// A candidate row for the sitemap — a route and (optionally) the front matter metadata
that was parsed for it. For markdown sources, metadata carries Date (lastmod) and IsDraft
(filter). For programmatic / Razor sources we typically have no metadata, which is fine:
those entries are emitted with their URL and no lastmod.
public record SitemapCandidate
{
    /// Front matter for the route, when available.
    public IFrontMatter Metadata { get; set; }

    /// Route to emit in the sitemap.
    public ContentRoute Route { get; set; }

    /// A candidate row for the sitemap — a route and (optionally) the front matter metadata
that was parsed for it. For markdown sources, metadata carries Date (lastmod) and IsDraft
(filter). For programmatic / Razor sources we typically have no metadata, which is fine:
those entries are emitted with their URL and no lastmod.
    public SitemapCandidate(ContentRoute Route, IFrontMatter Metadata)
;
}
```

Pennington.Feeds.SitemapEntry

Pennington.Feeds.SitemapEntry

CSHARP

```
namespace Pennington.Feeds;

/// Single <url> row written to sitemap.xml.
public record SitemapEntry
{
    /// Last-modified timestamp, when available.
    public DateTime? LastModified { get; set; }

    /// Single <url> row written to sitemap.xml.
    public SitemapEntry(UrlPath Url, DateTime? LastModified)
;

    /// Absolute URL for this sitemap entry.
    public UrlPath Url { get; set; }

}
```

Pennington.Feeds.SitemapService

Pennington.Feeds.SitemapService

CSHARP

```
namespace Pennington.Feeds;

/// Generates sitemap XML for the /sitemap.xml endpoint. Uses AsyncLazy for lazy, thread-safe computation. When managed by FileWatchDependencyFactory, the instance is recreated on file changes — trusts IContentService for fresh metadata. Enumerates every DiscoverAsync result. Markdown sources carry the front matter their content service already parsed at discovery time, so Date (lastmod) is read straight off Metadata — no re-parse, and no risk of applying one source's front-matter type to another's files. Programmatic / Razor page sources surface metadata only at render time and carry none on the discovered item, so their routes are emitted with no extra metadata. Sitemap, search index, and llms.txt each answer a different question, so they intentionally run different filtering paths:(this service) — every canonical HTML URL a crawler should index. Sourced from DiscoverAsync because Razor / programmatic routes with no TOC entry still need to appear. Per-page search: / llms: opt-outs are  honored here: those are search UX preferences, not SEO directives.— enumerates GetIndexableEntriesAsync, which carries ExcludeFromSearch / ExcludeFromLlms flags sourced from search: / llms: front-matter.Keep these two paths distinct — collapsing them would either leak search opt-outs into the sitemap (bad for SEO) or force every Razor/programmatic source to grow a TOC entry purely so it shows up in the sitemap.
public class SitemapService
{
    /// Returns the serialized sitemap XML, generating it on first access and caching the result.
    public Task<string> GetSitemapXmlAsync()
;

    /// Called on the file-watcher thread for every watched change. Must be quick and thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;

    /// Initializes the service and prepares lazy sitemap generation driven by the provided builder.
    public SitemapService(IEnumerable<IContentService> contentServices, LocalizationOptions localization, SitemapBuilder builder)
;
}
```

Pennington.FrontMatter.BlogFrontMatter

Pennington.FrontMatter.BlogFrontMatter

CSHARP

```
namespace Pennington.FrontMatter;

/// Core-library front matter for blog posts on bare AddPennington hosts. Carries Date,
Author, and Series alongside the IFrontMatter defaults, and implements IFrontMatter and
ITaggable. Not the record bound by AddBlogSite — see BlogSiteFrontMatter in the
Pennington.BlogSite package for that.
public record BlogFrontMatter
{
    /// Optional author name rendered in post bylines and feeds.
    public string Author { get; set; }

    /// Publication date surfaced in feeds and sitemaps. Also drives scheduled publishing:
when this is set to a moment after the build clock, the page is excluded from build output
(same dev-vs-build behavior as IsDraft) until the clock catches up.
    public DateTime? Date { get; set; }

    /// Short summary used in meta descriptions, OpenGraph tags, and listings.
    public string Description { get; set; }

    /// True when the page is a draft and should be excluded from builds.
    public bool IsDraft { get; set; }

    /// True when the page should be included in llms.txt output.
    public bool Llms { get; set; }

    /// True when the page should be included in the search index.
    public bool Search { get; set; }

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

    /// Optional series identifier used to group related posts together.
    public string Series { get; set; }

    /// Tags applied to this content.
    public string[] Tags { get; set; }

    /// Page title rendered in the browser tab, navigation, and OpenGraph tags.
    public string Title { get; set; }

    /// Stable cross-reference identifier used by xref links.
    public string Uid { get; set; }

}
```

Pennington.FrontMatter.DocFrontMatter

Pennington.FrontMatter.DocFrontMatter

CSHARP

```csharp
namespace Pennington.FrontMatter;

/// Core-library front matter for documentation pages on bare AddPennington hosts.
/// Implements IFrontMatter, ITaggable, ISectionable, and IOrderable — the default capability
/// shape for doc content without the DocSite template. Hosts using AddDocSite bind
/// DocSiteFrontMatter from the Pennington.DocSite package instead.
public record DocFrontMatter
{
    /// Short summary used in meta descriptions, OpenGraph tags, and listings.
    public string Description { get; set; }

    /// True when the page is a draft and should be excluded from builds.
    public bool IsDraft { get; set; }

    /// True when the page should be included in llms.txt output.
    public bool Llms { get; set; }

    /// Sort order for this page within its section (lower sorts first).
    public int Order { get; set; }

    /// True when the page should be included in the search index.
    public bool Search { get; set; }

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

    /// Section label for this page, used in breadcrumbs and prev/next navigation.
    public string SectionLabel { get; set; }

    /// Tags applied to this content.
    public string[] Tags { get; set; }

    /// Page title rendered in the browser tab, navigation, and OpenGraph tags.
    public string Title { get; set; }

    /// Stable cross-reference identifier used by xref links.
    public string Uid { get; set; }

}
```

Pennington.FrontMatter.FrontMatterExtensions

Pennington.FrontMatter.FrontMatterExtensions

CSHARP

```
namespace Pennington.FrontMatter;

/// Helpers that combine IFrontMatter capability checks into the single "skip this page during build" decision every content service and feed builder asks.
public class FrontMatterExtensions
{
    /// True when the page should be excluded from production build output — either explicitly drafted (IsDraft) or scheduled for a future Date. Mirrors the dev-vs-build split: dev-time requests still render so authors can preview, but the static crawler skips these routes.
    public static bool IsHiddenFromBuild(IFrontMatter frontMatter, TimeProvider clock)
;

    /// True when Date is set to a moment after clock's current wall-clock time. Comparison uses the wall-clock time in clock's configured local zone (without re-converting through the system zone), which is how YAML date: values — parsed as Unspecified — are meant to read.
    public static bool IsScheduled(IFrontMatter frontMatter, TimeProvider clock)
;
}
```

Pennington.FrontMatter.FrontMatterParser

Pennington.FrontMatter.FrontMatterParser

CSHARP

```
namespace Pennington.FrontMatter;

/// Parses YAML front matter from markdown content.
public class FrontMatterParser
{
    /// Deserialize raw YAML content (no --- delimiters) into a front matter type. Used for sidecar metadata files.
    public T DeserializeYaml<T>(string yaml, string sourcePath = null, DiagnosticContext diagnostics = null)
;

    /// Initializes the parser. Built-in front-matter types deserialize through the source-generated PenningtonYamlContext; other types fall back to reflection. Keys are camelCase matched case-insensitively. In lenient mode (the default outside build) unknown keys are dropped after a warning; in strict mode they additionally throw a YamlException.
    public FrontMatterParser(FrontMatterParserOptions options, IHttpContextAccessor httpContextAccessor, PenningtonYamlContextProvider yaml)
;

    /// Convenience constructor for direct instantiation (tests, scripts) that defaults to lenient mode, emits no diagnostics, and uses only the built-in serializer context. Production hosts should resolve the parser from DI so the configured FrontMatterParserOptions and any registered contexts apply.
    public FrontMatterParser()
;

    /// Parse front matter and return the metadata + remaining markdown body. Returns null metadata if no front matter block is present.
    public FrontMatterResult<T> Parse<T>(string content, string sourcePath = null, DiagnosticContext diagnostics = null)
;
}
```

Pennington.FrontMatter.FrontMatterParserOptions

Pennington.FrontMatter.FrontMatterParserOptions

CSHARP

```
namespace Pennington.FrontMatter;

/// Options that control front-matter parsing behavior.
public class FrontMatterParserOptions
{
    /// When true, unknown YAML keys cause the deserializer to throw a YamlException instead of silently dropping the value. Independently of this flag, every unknown key is reported as a Warning-severity diagnostic via DiagnosticContext so dev overlays and build reports surface typos. Defaults to false (lenient); the engine flips the default to true in build mode.
    public bool StrictUnknownKeys { get; set; }

}
```

Pennington.FrontMatter.FrontMatterResult

Pennington.FrontMatter.FrontMatterResult

CSHARP

```
namespace Pennington.FrontMatter;

/// Result of front matter parsing.
public record FrontMatterResult
{
    /// Markdown body with the front matter block stripped.
    public string Body { get; set; }

    /// Result of front matter parsing.
    public FrontMatterResult`1(T Metadata, string Body)
;

    /// Deserialized front matter, or null when the content had no front matter block.
    public T Metadata { get; set; }

}
```

Pennington.FrontMatter.IFrontMatter

Pennington.FrontMatter.IFrontMatter

CSHARP

```
namespace Pennington.FrontMatter;

/// Minimum: every content page has a title. Default members provide sensible opt-out values so implementations only declare properties they actually parse from front matter.
public interface IFrontMatter
{
    /// Publication date surfaced in feeds and sitemaps. Also drives scheduled publishing: when this is set to a moment after the build clock, the page is excluded from build output (same dev-vs-build behavior as IsDraft) until the clock catches up.
    public DateTime? Date { get; }

    /// Short summary used in meta descriptions, OpenGraph tags, and listings.
    public string Description { get; }

    /// True when the page is a draft and should be excluded from builds.
    public bool IsDraft { get; }

    /// True when the page should be included in llms.txt output.
    public bool Llms { get; }

    /// True when the page should be included in the search index.
    public bool Search { get; }

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

    /// Page title rendered in the browser tab, navigation, and OpenGraph tags.
    public string Title { get; }

    /// Stable cross-reference identifier used by xref links.
    public string Uid { get; }

}
```

Pennington.FrontMatter.IOrderable

Pennington.FrontMatter.IOrderable

CSHARP

```
namespace Pennington.FrontMatter;

/// Content that has explicit ordering.
public interface IOrderable
{
    /// Sort order for this page within its section (lower sorts first).
    public int Order { get; }

}
```

Pennington.FrontMatter.IRedirectable

Pennington.FrontMatter.IRedirectable

CSHARP

```csharp
namespace Pennington.FrontMatter;

/// Content that can redirect to another URL.
public interface IRedirectable
{
    /// Target URL when this page should redirect; null or empty means no redirect.
    public string RedirectUrl { get; }

}
```

Pennington.FrontMatter.ISectionable

Pennington.FrontMatter.ISectionable

CSHARP

```csharp
namespace Pennington.FrontMatter;

/// Content that carries a section label. The label surfaces on breadcrumbs and prev/next
navigation; it does NOT drive sidebar grouping (the subfolder under an area drives grouping
— see NavigationBuilder).
public interface ISectionable
{
    /// Section label for this page, used in breadcrumbs and prev/next navigation.
    public string SectionLabel { get; }

}
```

Pennington.FrontMatter.IStandardSiteDocument

Pennington.FrontMatter.IStandardSiteDocument

CSHARP

```
namespace Pennington.FrontMatter;

/// Content published as a Standard Site (AT Protocol) site.standard.document record.
Carries the record key the verification head tag links back to. A page opts in by setting
atprotoRkey in its front matter once the record exists; pages without it emit no document
link (graceful).
public interface IStandardSiteDocument
{
    /// Record key of this page's site.standard.document record; null when none exists yet.
    public string AtprotoRkey { get; }

}
```

Pennington.FrontMatter.ITaggable

Pennington.FrontMatter.ITaggable

CSHARP

```
namespace Pennington.FrontMatter;

/// Content that supports tags.
public interface ITaggable
{
    /// Tags applied to this content.
    public string[] Tags { get; }

}
```

Pennington.FrontMatter.PenningtonYamlContextProvider

Pennington.FrontMatter.PenningtonYamlContextProvider

CSHARP

```
namespace Pennington.FrontMatter;

/// Routes a type to the registered YamlSerializerContext that knows it — the built-in
PenningtonYamlContext, a satellite package context, or one a user added via AddYamlContext —
and falls back to reflection for everything else. A source-generated context only serves the
types it was generated for and rejects foreign options, so each type is dispatched to its
own context rather than combined into a single resolver.
public class PenningtonYamlContextProvider
{
    /// A provider seeded with only the built-in context, for non-DI use (tests, scripts).
    public static PenningtonYamlContextProvider Default { get; }

    /// Deserializes yaml into T using the source-generated context that covers T, or
reflection when none does.
    public T Deserialize<T>(string yaml)
;

    /// Initializes the provider with the serializer contexts registered in DI.
    public PenningtonYamlContextProvider(IEnumerable<YamlSerializerContext> contexts)
;
}
```

Pennington.Generation.AuditCache

Pennington.Generation.AuditCache

CSHARP

```
namespace Pennington.Generation;

/// Default IAuditCache implementation; written to by AuditRunner.
public class AuditCache
{
    /// The diagnostics produced by the most recent run, in insertion order.
    public ImmutableList<BuildDiagnostic> Diagnostics { get; }

    /// Raised after the cache is replaced. Use to log or refresh derived state.
    public event Action Updated
;
}
```

Pennington.Generation.AuditRunner

Pennington.Generation.AuditRunner

CSHARP

```
namespace Pennington.Generation;

/// Hosted service that runs every registered IBuildAuditor at startup and again whenever
IFileWatcher reports a content change. Writes the aggregated diagnostics into the shared
AuditCache. In dev mode, emits a one-line summary via ILogger after each run.
public class AuditRunner
{
    /// Wires the runner to its dependencies.
    public AuditRunner(IServiceProvider services, AuditCache cache, IFileWatcher
fileWatcher, LocalizationOptions localization, IHostApplicationLifetime lifetime,
ILogger<AuditRunner> logger)
;

    public Task StartAsync(CancellationToken cancellationToken)
;

    public Task StopAsync(CancellationToken cancellationToken)
;
}
```

Pennington.Generation.BuildAuditContext

Pennington.Generation.BuildAuditContext

CSHARP

```
namespace Pennington.Generation;

/// Inputs handed to AuditAsync.
public record BuildAuditContext
{
    /// Inputs handed to AuditAsync.
    public BuildAuditContext(ImmutableList<ContentTocItem> Pages, LocalizationOptions
Localization)
;

    /// Configured locales and the default-locale code.
    public LocalizationOptions Localization { get; set; }

    /// All TOC entries from every registered IContentService, post-discovery and locale-
aware.
    public ImmutableList<ContentTocItem> Pages { get; set; }

}
```

Pennington.Generation.BuildDiagnostic

Pennington.Generation.BuildDiagnostic

CSHARP

```
namespace Pennington.Generation;

/// Single diagnostic entry captured during a static build.
public record BuildDiagnostic
{
    /// Single diagnostic entry captured during a static build.
    public BuildDiagnostic(DiagnosticSeverity Severity, ContentRoute Route, string Message,
Exception Exception = null, string SourceFile = null)
;

    /// Optional exception captured with the diagnostic.
    public Exception Exception { get; set; }

    /// Human-readable message.
    public string Message { get; set; }

    /// Route the diagnostic relates to, if any.
    public ContentRoute Route { get; set; }

    /// Severity level of the diagnostic.
    public DiagnosticSeverity Severity { get; set; }

    /// Optional source file path associated with the diagnostic.
    public string SourceFile { get; set; }

}
```

Pennington.Generation.BuildReport

Pennington.Generation.BuildReport

CSHARP

```
namespace Pennington.Generation;

/// Aggregated result of a static build run, including diagnostics and per-page outcomes.
public class BuildReport
{
    /// Initializes a completed build report with all captured results.
    public BuildReport(ImmutableList<BuildDiagnostic> diagnostics,
ImmutableList<ContentRoute> generatedPages, ImmutableList<ContentRoute> skippedPages,
ImmutableList<ContentRoute> failedPages, TimeSpan duration)
;

    /// Diagnostics recorded during the build.
    public ImmutableList<BuildDiagnostic> Diagnostics { get; }

    /// Total wall-clock duration of the build.
    public TimeSpan Duration { get; }

    /// Routes whose generation failed.
    public ImmutableList<ContentRoute> FailedPages { get; }

    /// Routes that were successfully written to the output directory.
    public ImmutableList<ContentRoute> GeneratedPages { get; }

    /// True when the build produced any error diagnostics, broken links, or failed pages.
    public bool HasErrors { get; }

    /// Routes that were skipped (typically drafts).
    public ImmutableList<ContentRoute> SkippedPages { get; }

    /// Returns the human-readable summary as a single formatted string.
    public string ToFormattedString()
;

    /// Total number of pages considered, including generated, skipped, and failed.
    public int TotalPages { get; }

    /// Writes a human-readable summary of the report to writer.
    public void WriteTo(TextWriter writer)
;
}
```

Pennington.Generation.ClaimConflictAuditor

Pennington.Generation.ClaimConflictAuditor

CSHARP

```
namespace Pennington.Generation;

/// IBuildAuditor that polices the artifact tier's declared URL territories: warns when a
content route falls inside an ArtifactClaim (the artifact router would shadow it in dev and
the build output could collide), and when two claims from different owners overlap
(resolution order would silently decide the winner). Rides the audit pipeline, so the same
warnings reach the dev overlay, diag warnings, and the build report. Consults claims and
route discovery only — never artifact discovery, which may materialize the projection.
public class ClaimConflictAuditor
{
    /// Runs the auditor against context and returns its diagnostics.
    public Task<IReadOnlyList<BuildDiagnostic>> AuditAsync(BuildAuditContext context,
CancellationToken cancellationToken)
;

    /// Wires the auditor to the content discovery surface and the artifact tier.
    public ClaimConflictAuditor(IEnumerable<IContentService> contentServices,
IEnumerable<IArtifactContentService> artifactServices)
;

    /// Stable identifier surfaced on every diagnostic this auditor emits.
    public string Code { get; }

}
```

Pennington.Generation.IAuditCache

Pennington.Generation.IAuditCache

CSHARP

```
namespace Pennington.Generation;

/// Singleton store for the most recent audit pass. Read by the dev-mode overlay processor
(per request, filtered to the current route) and by OutputGenerationService at the end of a
static build (copied into the BuildReport).
public interface IAuditCache
{
    /// The diagnostics produced by the most recent run, in insertion order.
    public ImmutableList<BuildDiagnostic> Diagnostics { get; }

    /// Raised after the cache is replaced. Use to log or refresh derived state.
    public event Action Updated
;
}
```

Pennington.Generation.IBuildAuditor

Pennington.Generation.IBuildAuditor

CSHARP

```
namespace Pennington.Generation;

/// A build-time auditor that inspects discovered content and produces BuildDiagnostics.
Auditors run during dev-mode startup and on file changes (results feed the diagnostic
overlay), and again at build time (results land in Diagnostics). Implement and register via
services.AddTransient<IBuildAuditor, MyAuditor>(). Auditors should be cheap to invoke
repeatedly because the runner re-runs every registered auditor on every content-tree change.
public interface IBuildAuditor
{
    /// Runs the auditor against context and returns its diagnostics.
    public Task<IReadOnlyList<BuildDiagnostic>> AuditAsync(BuildAuditContext context,
CancellationToken cancellationToken)
;

    /// Stable identifier for the auditor (e.g. translation.audit). Surfaced on
BuildDiagnostics via the source label so users can filter or suppress by code in CI
dashboards.
    public string Code { get; }

}
```

Pennington.Generation.IRenderedAuditor

Pennington.Generation.IRenderedAuditor

CSHARP

```
namespace Pennington.Generation;

/// A build-time auditor that needs the post-pipeline rendered HTML for each page. Runs
after every IBuildAuditor in AuditRunner and writes into the same IAuditCache, so consumers
(the dev overlay, the build report) don't need to know which interface produced a given
diagnostic. Prefer IBuildAuditor when a structural check on Pages is enough — this seam is
for checks (broken links, accessibility passes, etc.) that genuinely need rendered markup.
Each rendered audit costs an HTTP self-dispatch per page.
public interface IRenderedAuditor
{
    /// Runs the auditor against context and returns its diagnostics.
    public Task<IReadOnlyList<BuildDiagnostic>> AuditAsync(RenderedAuditContext context,
CancellationToken cancellationToken)
;

    /// Stable identifier for the auditor (e.g. content.links).
    public string Code { get; }

}
```

Pennington.Generation.LinkAuditor

Pennington.Generation.LinkAuditor

CSHARP

```
namespace Pennington.Generation;

/// IRenderedAuditor that fetches each TOC page through the live pipeline and runs
LinkVerificationService over its rendered HTML, surfacing broken internal links in the dev
overlay (per page) and the build report.
public class LinkAuditor
{
    /// Runs the auditor against context and returns its diagnostics.
    public Task<IReadOnlyList<BuildDiagnostic>> AuditAsync(RenderedAuditContext context,
CancellationToken cancellationToken)
;

    /// Stable identifier surfaced on every diagnostic this auditor emits.
    public string Code { get; }

    /// Wires the auditor to the content discovery surface, the artifact tier, the endpoint
table, the output options, and the host environment (for wwwroot/RCL assets).
    public LinkAuditor(IEnumerable<IContentService> contentServices,
IEnumerable<IArtifactContentService> artifactServices, EndpointDataSource
endpointDataSource, OutputOptions outputOptions, IWebHostEnvironment environment)
;
}
```

Pennington.Generation.LinkType

Pennington.Generation.LinkType

CSHARP

```
namespace Pennington.Generation;

/// Classifies a broken link discovered during build-time link verification.
public enum LinkType
{
    /// In-page anchor (fragment) link.
    public static const LinkType Anchor
;

    /// Link to an external origin outside the site.
    public static const LinkType External
;

    /// Image (or other media) reference.
    public static const LinkType Image
;

    /// Link to another page within the site.
    public static const LinkType Internal
;
}
```

Pennington.Generation.OutputGenerationService

Pennington.Generation.OutputGenerationService

CSHARP

```
namespace Pennington.Generation;

/// Generates a static site by HTTP-crawling the running app. Pages are fetched in priority order: HTML content first, then MapGet routes (like /styles.css) last. MonorailCSS Discovery's IL scan populates the class registry at startup, so the stylesheet is correct regardless of fetch order; the MapGet-last rule keeps generated endpoints downstream of any other dependencies the same crawler might exercise.
public class OutputGenerationService
{
    /// Crawls the running app and writes every discovered route to the output directory.
    public Task<BuildReport> GenerateAsync()
;

    /// Crawls the running app and returns a BuildReport. When writeToDisk is false the output directory is left untouched — the HTTP crawl, diagnostic collection, and link verification still run, which makes this mode suitable for dev-time validators that want the same warnings a real build would produce.
    public Task<BuildReport> GenerateAsync(bool writeToDisk)
;

    /// Sentinel URL fetched during site generation to produce 404.html. The path is not a real content route — it exists only to trigger the catch-all fallback handler whose rendered HTML is written to disk. Other parts of the engine (e.g. LocalizationOptions.GetAlternateLanguages) must recognize this sentinel so language switchers on 404.html don't emit phantom /{locale}/__pennington-404-generator/ links.
    public static const string NotFoundGeneratorPath
;

    /// Initializes the service with the dependencies required to crawl the running app and write output.
    public OutputGenerationService(IEnumerable<IContentService> contentServices, IEnumerable<IArtifactContentService> artifactServices, OutputOptions outputOptions, IWebHostEnvironment environment, EndpointDataSource endpointDataSource, IFileSystem fileSystem, IInProcessHttpDispatcher dispatcher, IAuditCache auditCache, ILogger<OutputGenerationService> logger, AuditRunner auditRunner = null)
;
}
```

Pennington.Generation.OutputOptions

Pennington.Generation.OutputOptions

CSHARP

```
namespace Pennington.Generation;

/// Options controlling the static build output: target directory, base URL, and cleanup
behavior.
public class OutputOptions
{
    /// Base URL the site is deployed under (used to rewrite links in generated HTML).
    public UrlPath BaseUrl { get; set; }

    /// When true, the output directory is cleared before a build run.
    public bool CleanOutput { get; set; }

    /// Parses CLI arguments into OutputOptions, honoring positional and named flag forms.
    public static OutputOptions FromArgs(string[] args)
;

    /// Directory where generated output is written.
    public FilePath OutputDirectory { get; set; }

}
```

Pennington.Generation.OverlapAuditor

Pennington.Generation.OverlapAuditor

CSHARP

```
namespace Pennington.Generation;

/// IBuildAuditor that flags markdown content sources whose AbsoluteContentRoot directories
overlap without an explicit ExcludePaths carve-out. Wraps MarkdownSourceOverlapDetector so
the same warnings reach the dev overlay (via AuditCache) and the build report.
public class OverlapAuditor
{
    /// Runs the auditor against context and returns its diagnostics.
    public Task<IReadOnlyList<BuildDiagnostic>> AuditAsync(BuildAuditContext context,
CancellationToken cancellationToken)
;

    /// Stable identifier surfaced on every diagnostic this auditor emits.
    public string Code { get; }

    /// Wires the auditor to the registered content services.
    public OverlapAuditor(IEnumerable<IContentService> contentServices)
;
}
```

Pennington.Generation.RenderedAuditContext

Pennington.Generation.RenderedAuditContext

CSHARP

```
namespace Pennington.Generation;

/// Inputs handed to AuditAsync: the routes to fetch plus a delegate that returns each
route's post-pipeline rendered HTML. Unlike BuildAuditContext, the page set is the full
generated route set — every discovered HTML page — so rendered checks cover routes that
never appear in navigation (the homepage and other Razor-only pages).
public record RenderedAuditContext
{
    /// Fetches the rendered HTML for a Pages route through the live application pipeline.
Returns null when the route does not resolve (404) so the caller can filter rather than
handle exceptions.
    public Func<ContentRoute, CancellationToken, Task<string>> GetRenderedHtmlAsync { get;
set; }

    /// Configured locales and the default-locale code.
    public LocalizationOptions Localization { get; set; }

    /// Every generated HTML route, from full content discovery rather than the navigation
TOC.
    public ImmutableList<ContentRoute> Pages { get; set; }

    /// Inputs handed to AuditAsync: the routes to fetch plus a delegate that returns each
route's post-pipeline rendered HTML. Unlike BuildAuditContext, the page set is the full
generated route set — every discovered HTML page — so rendered checks cover routes that
never appear in navigation (the homepage and other Razor-only pages).
    public RenderedAuditContext(ImmutableList<ContentRoute> Pages, LocalizationOptions
Localization, Func<ContentRoute, CancellationToken, Task<string>> GetRenderedHtmlAsync)
;
}
```

Pennington.Generation.XrefAuditor

Pennington.Generation.XrefAuditor

CSHARP

```
namespace Pennington.Generation;

/// IBuildAuditor that scans markdown source files for xref: references whose UID does not resolve through the live XrefResolver. Catches typos and stale links pre-render so the dev overlay flags them on the page that contains them and the build report lists them once per route.
public class XrefAuditor
{
    /// Runs the auditor against context and returns its diagnostics.
    public Task<IReadOnlyList<BuildDiagnostic>> AuditAsync(BuildAuditContext context, CancellationToken cancellationToken)
;

    /// Stable identifier surfaced on every diagnostic this auditor emits.
    public string Code { get; }

    /// Wires the auditor to the content discovery surface, the xref resolver, and the file system.
    public XrefAuditor(IEnumerable<IContentService> contentServices, XrefResolver resolver, IFileSystem fileSystem)
;
}
```

Pennington.Head.HeadBuilder

Pennington.Head.HeadBuilder

CSHARP

```
namespace Pennington.Head;

/// Collects HeadTags from every contributor. Keyed tags deduplicate by HeadTagKey (first
add at a key wins; later same-key adds are dropped) while preserving first-seen order;
keyless tags (added via AddRepeatable) always append.
public class HeadBuilder
{
    /// Adds a tag under an explicit dedup key; the first add at a key wins.
    public HeadBuilder Add(HeadTagKey key, HeadTag tag)
;

    /// Adds a repeatable tag (hreflang, JSON-LD, preload) with no deduplication.
    public HeadBuilder AddRepeatable(HeadTag tag)
;

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

    /// Sets a singleton link (e.g. canonical), deduplicated on its rel.
    public HeadBuilder Link(string rel, string href)
;

    /// Sets a named meta tag, deduplicated on its name.
    public HeadBuilder Meta(string name, string content)
;

    /// Sets an OpenGraph/property meta tag, deduplicated on its property.
    public HeadBuilder Property(string property, string content)
;

    /// Sets the document title (deduplicated to one).
    public HeadBuilder Title(string text)
;
}
```

Pennington.Head.HeadContext

Pennington.Head.HeadContext

CSHARP

```
namespace Pennington.Head;

/// Per-request inputs handed to every IHeadContributor.
public class HeadContext
{
    /// Request path with the locale segment reattached (PathBase + Path) — the same key
ContentRecordRegistry joins on (after trimming slashes).
    public string FullPath { get; set; }

    /// The active HTTP request.
    public HttpContext HttpContext { get; set; }

    /// The content record resolved for this request, or null for endpoint/404 pages.
    public ContentRecord Record { get; set; }

}
```

Pennington.Head.HeadEntry

Pennington.Head.HeadEntry

CSHARP

```
namespace Pennington.Head;

/// A composed head tag paired with its dedup key (null for repeatable tags).
public struct HeadEntry
{
    /// A composed head tag paired with its dedup key (null for repeatable tags).
    public HeadEntry(HeadTagKey? Key, HeadTag Tag)
;

    /// Dedup key, or null when the tag is repeatable.
    public HeadTagKey? Key { get; set; }

    /// The tag to emit.
    public HeadTag Tag { get; set; }

}
```

Pennington.Head.HeadOrder

Pennington.Head.HeadOrder

CSHARP

```
namespace Pennington.Head;

/// Named priority bands for Order. Contributors run lowest-first, and on a HeadTagKey
collision the lowest order wins — so page-level contributions (lower bands) beat site-level
defaults (higher bands). Bands replace the ad-hoc integer ordering the HTML rewriters used,
where unrelated writers silently collided on the same number.
public class HeadOrder
{
    /// Discovery payloads: JSON-LD structured data and Standard Site verification links.
    public static const int Discovery
;

    /// Page-authored or page-computed tags: title, description, per-page OpenGraph. Wins
ties against site defaults.
    public static const int Page
;

    /// Site-wide defaults: canonical, og:site_name, RSS/llms alternates, hreflang.
    public static const int Site
;
}
```

Pennington.Head.HeadServiceExtensions

Pennington.Head.HeadServiceExtensions

CSHARP

```
namespace Pennington.Head;

/// Registration helpers for the head-composition pipeline.
public class HeadServiceExtensions
{
    /// Registers the head composition rewriter. Inert until at least one IHeadContributor
is also registered, so adding this on its own leaves head output byte-identical.
    public static IServiceCollection AddHead(IServiceCollection services)
;

    /// Registers a single head contributor. Transient so contributors capturing a file-
watched dependency (e.g. the content registry) pick up the current instance per request.
    public static IServiceCollection AddHeadContributor<T>(IServiceCollection services)
;
}
```

Pennington.Head.HeadTag

Pennington.Head.HeadTag

CSHARP

```
namespace Pennington.Head;

/// The set of elements a contributor can place in the document <head>.
public struct HeadTag
{
    /// Wraps a TitleTag.
    public HeadTag(TitleTag value)
;

    /// Wraps a MetaNameTag.
    public HeadTag(MetaNameTag value)
;

    /// Wraps a MetaPropertyTag.
    public HeadTag(MetaPropertyTag value)
;

    /// Wraps a LinkTag.
    public HeadTag(LinkTag value)
;

    /// Wraps a ScriptTag.
    public HeadTag(ScriptTag value)
;

    /// Wraps a RawTag.
    public HeadTag(RawTag value)
;

    /// A <link> tag (canonical, alternate, stylesheet, preload, verification).
    public sealed record LinkTag(string Rel, string Href) : object, IEquatable<LinkTag>

    /// A <meta name="..." content="..."> tag (description, twitter:*, generator).
    public sealed record MetaNameTag(string Name, string Content) : object,
IEquatable<MetaNameTag>

    /// A <meta property="..." content="..."> tag (OpenGraph og:*).
    public sealed record MetaPropertyTag(string Property, string Content) : object,
IEquatable<MetaPropertyTag>

    /// An escape hatch carrying arbitrary head markup verbatim (e.g.
AdditionalHtmlHeadContent).
    public sealed record RawTag(string Html) : object, IEquatable<RawTag>

    /// A <script> tag — JSON-LD, a deferred asset, or an inline bootstrap.
    public sealed record ScriptTag : object, IEquatable<ScriptTag>

    /// The document <title>. Deduplicated to exactly one.
    public sealed record TitleTag(string Text) : object, IEquatable<TitleTag>

    /// Wrapped case instance; inspect via pattern matching on the case types.
```

```
    public object Value { get; }  }
```

Pennington.Head.HeadTagKey

Pennington.Head.HeadTagKey

CSHARP

```
namespace Pennington.Head;

/// Identity used for head deduplication. Two tags with the same key collapse to one — the first contributor to add at a key wins (contributors run lowest-Order first, so page-level beats site-level). Repeatable tags (hreflang, JSON-LD, preloads) opt out by being added with no key via AddRepeatable.
public struct HeadTagKey
{
    /// Identity used for head deduplication. Two tags with the same key collapse to one — the first contributor to add at a key wins (contributors run lowest-Order first, so page-level beats site-level). Repeatable tags (hreflang, JSON-LD, preloads) opt out by being added with no key via AddRepeatable.
    public HeadTagKey(string Value)
;

    /// Builds the key for a singleton link rel (e.g. link:rel:canonical).
    public static HeadTagKey LinkRel(string rel)
;

    /// Builds the key for a named meta tag (e.g. meta:name:description).
    public static HeadTagKey MetaName(string name)
;

    /// Builds the key for an OpenGraph/property meta tag (e.g. meta:prop:og:image).
    public static HeadTagKey MetaProperty(string property)
;

    /// The singleton document title.
    public static readonly HeadTagKey Title
;

    /// Stable string identity (e.g. title, meta:prop:og:image, link:rel:canonical).
    public string Value { get; set; }

}
```

Pennington.Head.IHeadContributor

Pennington.Head.IHeadContributor

CSHARP

```
namespace Pennington.Head;

/// Contributes tags to the document <head>. Every contributor feeds a single HeadBuilder;
the composed result is reconciled into the DOM once by the head composition rewriter.
Contributors never touch the DOM directly — they emit typed HeadTags, and dedup/ordering is
handled centrally.
public interface IHeadContributor
{
    /// Pushes tags into the builder for this request.
    public Task ContributeAsync(HeadContext context, HeadBuilder head)
;

    /// Ascending priority (use the HeadOrder bands). Contributors run lowest-first, and on
a HeadTagKey collision the lowest order wins.
    public int Order { get; }

    /// Cheap gate. Return false to skip ContributeAsync entirely.
    public bool ShouldContribute(HeadContext context)
;
}
```

Pennington.Head.LinkTag

Pennington.Head.LinkTag

CSHARP

```
namespace Pennington.Head;

/// A <link> tag (canonical, alternate, stylesheet, preload, verification).
public record LinkTag
{
    /// Extra attributes in emission order (e.g. type, title, hreflang, as, crossorigin).
    public ImmutableArray<KeyValuePair<string, string>> Attributes { get; set; }

    /// The href attribute.
    public string Href { get; set; }

    /// A <link> tag (canonical, alternate, stylesheet, preload, verification).
    public LinkTag(string Rel, string Href)
;

    /// The rel attribute.
    public string Rel { get; set; }

}
```

Pennington.Head.MetaNameTag

Pennington.Head.MetaNameTag

CSHARP

```
namespace Pennington.Head;

/// A <meta name="..." content="..."> tag (description, twitter:*, generator).
public record MetaNameTag
{
    /// The content attribute.
    public string Content { get; set; }

    /// A <meta name="..." content="..."> tag (description, twitter:*, generator).
    public MetaNameTag(string Name, string Content)
;

    /// The name attribute.
    public string Name { get; set; }

}
```

Pennington.Head.MetaPropertyTag

Pennington.Head.MetaPropertyTag

CSHARP

```
namespace Pennington.Head;

/// A <meta property="..." content="..."> tag (OpenGraph og:*).
public record MetaPropertyTag
{
    /// The content attribute.
    public string Content { get; set; }

    /// A <meta property="..." content="..."> tag (OpenGraph og:*).
    public MetaPropertyTag(string Property, string Content)
;

    /// The property attribute.
    public string Property { get; set; }

}
```

Pennington.Head.RawTag

Pennington.Head.RawTag

CSHARP

```
namespace Pennington.Head;

/// An escape hatch carrying arbitrary head markup verbatim (e.g.
AdditionalHtmlHeadContent).
public record RawTag
{
    /// Raw HTML inserted into the head as-is.
    public string Html { get; set; }

    /// An escape hatch carrying arbitrary head markup verbatim (e.g.
AdditionalHtmlHeadContent).
    public RawTag(string Html)
;
}
```

Pennington.Head.ScriptTag

Pennington.Head.ScriptTag

CSHARP

```
namespace Pennington.Head;

/// A <script> tag — JSON-LD, a deferred asset, or an inline bootstrap.
public record ScriptTag
{
    /// Emit a defer attribute on an external script.
    public bool Defer { get; set; }

    /// Inline script body (e.g. a JSON-LD payload or a theme bootstrap).
    public string InlineBody { get; set; }

    /// External script URL; mutually exclusive with InlineBody.
    public string Src { get; set; }

    /// The type attribute (e.g. application/ld+json).
    public string Type { get; set; }

}
```

Pennington.Head.TitleTag

Pennington.Head.TitleTag

CSHARP

```
namespace Pennington.Head;

/// The document <title>. Deduplicated to exactly one.
public record TitleTag
{
    /// Title text.
    public string Text { get; set; }

    /// The document <title>. Deduplicated to exactly one.
    public TitleTag(string Text)
;
}
```

Pennington.Highlighting.HighlightingService

Pennington.Highlighting.HighlightingService

CSHARP

```
namespace Pennington.Highlighting;

/// Dispatches code highlighting to the highest-priority ICodeHighlighter that supports the
requested language. Falls back to PlainTextHighlighter and emits an Info diagnostic once per
unknown language per instance.
public class HighlightingService
{
    /// Highlight code using the best available highlighter for the language. Returns the
highlighted HTML string.
    public string Highlight(string code, string language)
;

    /// Initializes the service with the registered highlighters, ordered by descending
Priority.
    public HighlightingService(IEnumerable<ICodeHighlighter> highlighters)
;

    /// Initializes the service with the registered highlighters and an optional HTTP
context accessor used to surface unknown-language Info diagnostics to the per-request
DiagnosticContext.
    public HighlightingService(IEnumerable<ICodeHighlighter> highlighters,
IHttpContextAccessor httpContextAccessor)
;
}
```

Pennington.Highlighting.ICodeHighlighter

Pennington.Highlighting.ICodeHighlighter

CSHARP

```csharp
namespace Pennington.Highlighting;

/// Syntax highlighter that converts source code into HTML fragments.
public interface ICodeHighlighter
{
    /// Highlight code. Returns HTML with spans.
    public string Highlight(string code, string language)
;

    /// Priority — higher wins when multiple highlighters support a language.
    public int Priority { get; }

    /// Languages this highlighter handles (e.g., "csharp", "python").
    public IReadOnlySet<string> SupportedLanguages { get; }

}
```

Pennington.Highlighting.PlainTextHighlighter

Pennington.Highlighting.PlainTextHighlighter

CSHARP

```csharp
namespace Pennington.Highlighting;

/// Fallback highlighter — HTML-encodes code, no syntax highlighting.
public class PlainTextHighlighter
{
    /// Highlight code. Returns HTML with spans.
    public string Highlight(string code, string language)
;

    /// Priority — higher wins when multiple highlighters support a language.
    public int Priority { get; }

    /// Languages this highlighter handles (e.g., "csharp", "python").
    public IReadOnlySet<string> SupportedLanguages { get; }

}
```

Pennington.Highlighting.ShellHighlighter

Pennington.Highlighting.ShellHighlighter

CSHARP

```
namespace Pennington.Highlighting;

/// Provides syntax highlighting for shell/bash/batch commands. Implements ICodeHighlighter
with priority 75 (higher than TextMate for shell languages).
public class ShellHighlighter
{
    /// Highlight code. Returns HTML with spans.
    public string Highlight(string code, string language)
;

    /// Priority — higher wins when multiple highlighters support a language.
    public int Priority { get; }

    /// Languages this highlighter handles (e.g., "csharp", "python").
    public IReadOnlySet<string> SupportedLanguages { get; }

}
```

Pennington.Highlighting.TextMateHighlighter

Pennington.Highlighting.TextMateHighlighter

CSHARP

```
namespace Pennington.Highlighting;

/// Provides syntax highlighting for code blocks using TextMate grammars. Implements
ICodeHighlighter with priority 50.
public class TextMateHighlighter
{
    /// Highlight code. Returns HTML with spans.
    public string Highlight(string code, string language)
;

    /// Priority — higher wins when multiple highlighters support a language.
    public int Priority { get; }

    /// Languages this highlighter handles (e.g., "csharp", "python").
    public IReadOnlySet<string> SupportedLanguages { get; }

    /// Initializes the highlighter with a TextMate grammar registry shared across
instances.
    public TextMateHighlighter(TextMateLanguageRegistry languageRegistry)
;
}
```

Pennington.Highlighting.TextMateLanguageRegistry

Pennington.Highlighting.TextMateLanguageRegistry

CSHARP

```
namespace Pennington.Highlighting;

/// Registry over the built-in TextMate grammar set. Exposes the underlying Registry and
resolves a language identifier to its TextMate scope.
public class TextMateLanguageRegistry
{
    /// Initializes the registry over the built-in grammar set.
    public TextMateLanguageRegistry()
;
}
```

Pennington.Infrastructure.AsyncHelpers

Pennington.Infrastructure.AsyncHelpers

CSHARP

```
namespace Pennington.Infrastructure;

/// Run async code synchronously without deadlocks.
public class AsyncHelpers
{
    /// Synchronously runs an async delegate on the default task scheduler, bypassing the current synchronization context.
    public static void RunSync(Func<Task> func)
;

    /// Synchronously runs an async delegate on the default task scheduler and returns its result, bypassing the current synchronization context.
    public static TResult RunSync<TResult>(Func<Task<TResult>> func)
;
}
```

Pennington.Infrastructure.AsyncLazy

Pennington.Infrastructure.AsyncLazy

CSHARP

```
namespace Pennington.Infrastructure;

/// Thread-safe asynchronous lazy initialization. The factory is queued onto the thread pool on first access; subsequent accesses return the same task. A faulted task is evicted so the next access retries. Use GetAwaiter (i.e. await asyncLazy) from async contexts. Pennington has no sync-over-async on this type in production code; if a new consumer reaches for asyncLazy.Task.GetAwaiter().GetResult(), make its caller async instead — that's the pattern that exhausts thread-pool budget and occasionally deadlocks under hostile schedulers.
public class AsyncLazy
{
    /// Initializes the instance with a factory invoked on first access.
    public AsyncLazy`1(Func<Task<T>> factory)
;

    /// Lets the lazy be awaited directly: var value = await asyncLazy;.
    public TaskAwaiter<T> GetAwaiter()
;

    /// Returns the underlying task. Starts the factory on a thread-pool thread the first time it is accessed; subsequent accesses replay the same task. A previously-faulted task is evicted so the next access retries.
    public Task<T> Task { get; }

}
```

Pennington.Infrastructure.BaseUrlCssResponseProcessor

Pennington.Infrastructure.BaseUrlCssResponseProcessor

CSHARP

```
namespace Pennington.Infrastructure;

/// Prefixes root-relative url(...) references in CSS responses with the configured base
URL, mirroring BaseUrlHtmlRewriter for stylesheets. Without this, hand-authored @font-face
rules and other asset references fed through MonorailCssOptions.ExtraStyles resolve against
the deployment root and 404 under sub-path deployments.
public class BaseUrlCssResponseProcessor
{
    /// Initializes the processor with the base URL from OutputOptions.
    public BaseUrlCssResponseProcessor(OutputOptions outputOptions)
;

    /// Execution order; lower values run earlier in the response pipeline.
    public int Order { get; }

    /// Transforms responseBody and returns the processed body.
    public Task<string> ProcessAsync(string responseBody, HttpContext context)
;

    /// Returns true when this processor should run for the current request.
    public bool ShouldProcess(HttpContext context)
;
}
```

Pennington.Infrastructure.BrokenLinkResult

Pennington.Infrastructure.BrokenLinkResult

CSHARP

```
namespace Pennington.Infrastructure;

/// A link that failed verification.
public record BrokenLinkResult
{
    /// A link that failed verification.
    public BrokenLinkResult(ContentRoute SourcePage, string Url, LinkType Type, string
Reason)
;

    /// Human-readable reason the link is considered broken.
    public string Reason { get; set; }

    /// Page that contained the link.
    public ContentRoute SourcePage { get; set; }

    /// Classification of the broken link.
    public LinkType Type { get; set; }

    /// Link target URL.
    public string Url { get; set; }

}
```

Pennington.Infrastructure.BuildHtmlCache

Pennington.Infrastructure.BuildHtmlCache

CSHARP

```
namespace Pennington.Infrastructure;

/// Process-lifetime cache of fully rendered in-process HTTP responses, keyed by request
path. The static build crawls itself: the disk-write pass, the search index, and the
llms.txt sidecar each self-fetch the same pages, so without sharing every page renders 2–3×
through the full middleware pipeline. Installed behind HttpDispatcher via
CachingHttpHandler, this collapses that to one render per URL — every consumer replays the
first render.Eviction rides the existing FileWatchDispatcher: as an IFileWatchAware with no
scopes of its own, it is notified of every change another watcher already observes. On
notification, it consults GetAffectedRoutes on every registered content service and evicts
only the affected keys — wholesale only on a Wildcard report.
public class BuildHtmlCache
{
    /// Initializes the cache with the content services it consults for affected routes on
file change.
    public BuildHtmlCache(IEnumerable<IContentService> contentServices)
;

    /// Returns the cached response for key, invoking factory exactly once on the first
request for that key. Concurrent first-requests coalesce onto the same render; a faulted
render is evicted so a later request can retry.
    public Task<CachedResponse> GetOrAddAsync(string key, Func<Task<CachedResponse>>
factory)
;

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;
}
```

Pennington.Infrastructure.CachedResponse

Pennington.Infrastructure.CachedResponse

CSHARP

```
namespace Pennington.Infrastructure;

/// A captured HTTP response — status, body, and headers — replayable as a fresh
HttpResponseMessage any number of times. Headers are preserved verbatim so per-request
signals the build relies on (the X-Pennington-Diagnostic headers, Location on redirects,
Content-Type) survive replay.
public record CachedResponse
{
    /// The response body bytes.
    public byte[] Body { get; set; }

    /// A captured HTTP response — status, body, and headers — replayable as a fresh
HttpResponseMessage any number of times. Headers are preserved verbatim so per-request
signals the build relies on (the X-Pennington-Diagnostic headers, Location on redirects,
Content-Type) survive replay.
    public CachedResponse(HttpStatusCode Status, byte[] Body,
IReadOnlyList<KeyValuePair<string, string[]>> ResponseHeaders,
IReadOnlyList<KeyValuePair<string, string[]>> ContentHeaders)
;

    /// Reads response fully into a replayable CachedResponse.
    public static Task<CachedResponse> CaptureAsync(HttpResponseMessage response,
CancellationToken ct)
;

    /// Captured content-level headers.
    public IReadOnlyList<KeyValuePair<string, string[]>> ContentHeaders { get; set; }

    /// Captured response-level headers.
    public IReadOnlyList<KeyValuePair<string, string[]>> ResponseHeaders { get; set; }

    /// The response status code.
    public HttpStatusCode Status { get; set; }

    /// Rebuilds a fresh HttpResponseMessage from the captured state.
    public HttpResponseMessage ToHttpResponseMessage()
;
}
```

Pennington.Infrastructure.CachingHttpHandler

Pennington.Infrastructure.CachingHttpHandler

CSHARP

```
namespace Pennington.Infrastructure;

/// Caches GET responses from the in-process crawl in BuildHtmlCache so the disk-write pass
and the search/llms.txt sidecars share one render per URL. Installed by HttpDispatcher as
the outer handler over the TestServer/Kestrel client. Non-GET requests and the 404 sentinel
pass straight through uncached.
public class CachingHttpHandler
{
    /// Initializes the handler over the shared cache.
    public CachingHttpHandler(BuildHtmlCache cache)
;
}
```

Pennington.Infrastructure.ContentFormatOptions

Pennington.Infrastructure.ContentFormatOptions

CSHARP

```
namespace Pennington.Infrastructure;

/// Options for a custom content-format source registered via AddContentFormat.
public class ContentFormatOptions
{
    /// URL prefix prepended to routes generated from this source.
    public string BasePageUrl { get; set; }

    /// Filesystem path to the directory containing the format's source files.
    public string ContentPath { get; set; }

    /// Relative subpaths (from ContentPath) to skip during discovery.
    public ImmutableArray<string> ExcludePaths { get; set; }

    /// Glob pattern used to enumerate source files (for example *.cook).
    public string FilePattern { get; set; }

    /// Default section label applied when front matter does not specify one.
    public string SectionLabel { get; set; }

    /// Sets the IContentParser type that parses this format's files (resolved from DI).
    public ContentFormatOptions UseParser<TParser>()
;

    /// Sets the IContentRenderer type that renders this format's parsed items (resolved
from DI).
    public ContentFormatOptions UseRenderer<TRenderer>()
;
}
```

Pennington.Infrastructure.CorpusFetchScope

Pennington.Infrastructure.CorpusFetchScope

CSHARP

```
namespace Pennington.Infrastructure;

/// Runtime tripwire for the b719d73 deadlock class: a single-flight corpus task (the site projection) awaited from work its own materialization spawned is a task-level circular wait that hangs forever. Two flags mark the dangerous regions so SiteProjection can throw a descriptive exception instead of deadlocking: InsideCorpusFetch — this request IS a projection-issued page self-fetch. RenderedHtmlFetcher stamps HeaderName on every fetch (an AsyncLocal alone would not cross the Kestrel loopback socket in dev), and UsePennington enters the scope when the header is present.InsideMaterialization — the projection's materialization is on the current async flow; re-entering it (e.g. from a content service's discovery) would await a task that is waiting on the caller.
public class CorpusFetchScope
{
    /// Marks the current async flow as a projection-issued page fetch until disposed.
    public static IDisposable EnterCorpusFetch()
;

    /// Marks the current async flow as projection materialization until disposed.
    public static IDisposable EnterMaterialization()
;

    /// Header stamped on projection-issued self-fetches. Dev-only surface: a client sending it manually gets the fail-fast exception on any page whose render path consumes the projection — which is exactly the bug the exception describes.
    public static const string HeaderName
;

    /// True while processing a request the projection issued to render a page.
    public static bool InsideCorpusFetch { get; }

    /// True while the projection's corpus materialization is on the current async flow.
    public static bool InsideMaterialization { get; }

}
```

Pennington.Infrastructure.ExternalLink

Pennington.Infrastructure.ExternalLink

CSHARP

```
namespace Pennington.Infrastructure;

/// A link that points to an external origin and was not verified by the internal checker.
public record ExternalLink
{
    /// A link that points to an external origin and was not verified by the internal
checker.
    public ExternalLink(ContentRoute SourcePage, string Url)
;

    /// Page that contained the link.
    public ContentRoute SourcePage { get; set; }

    /// External target URL.
    public string Url { get; set; }

}
```

Pennington.Infrastructure.FileChangeNotification

Pennington.Infrastructure.FileChangeNotification

CSHARP

```
namespace Pennington.Infrastructure;

/// A single file-change notification carrying the full path and the type of change.
public struct FileChangeNotification
{
    /// Kind of change reported by FileSystemWatcher.
    public WatcherChangeTypes ChangeType { get; set; }

    /// A single file-change notification carrying the full path and the type of change.
    public FileChangeNotification(string FullPath, WatcherChangeTypes ChangeType)
;

    /// Absolute path to the file that changed.
    public string FullPath { get; set; }

}
```

Pennington.Infrastructure.FileWatchDependencyFactory

Pennington.Infrastructure.FileWatchDependencyFactory

CSHARP

```
namespace Pennington.Infrastructure;

/// Manages a cached service instance that auto-invalidates when watched files change.
public class FileWatchDependencyFactory
{
    public void Dispose()
;

    /// Initializes the factory. FileWatchDispatcher drives invalidation.
    public FileWatchDependencyFactory`1(IServiceProvider serviceProvider,
ILogger<FileWatchDependencyFactory<T>> logger)
;

    /// Returns the cached instance, constructing one via DI on first access.
    public T GetInstance()
;

    /// Consults the current instance (if one has been built) and drops it when the instance
asks to be recreated. Called by FileWatchDispatcher.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;
}
```

Pennington.Infrastructure.FileWatchDispatcher

Pennington.Infrastructure.FileWatchDispatcher

CSHARP

```
namespace Pennington.Infrastructure;

/// Owns every IFileWatcher call on behalf of the application's IFileWatchAware services. On
construction it registers an OS-level watcher for each declared FileWatchScope and
subscribes once to the change stream; every subsequent change is fanned out to all
IFileWatchAware instances.
public class FileWatchDispatcher
{
    /// Wires the watches: registers every declared scope and subscribes to the change
stream.
    public FileWatchDispatcher(IEnumerable<IFileWatchAware> aware, IFileWatcher fileWatcher,
ILogger<FileWatchDispatcher> logger = null)
;
}
```

Pennington.Infrastructure.FileWatchedServiceExtensions

Pennington.Infrastructure.FileWatchedServiceExtensions

CSHARP

```
namespace Pennington.Infrastructure;

/// DI helpers for registering services whose lifetimes are bound to file-change
invalidation.
public class FileWatchedServiceExtensions
{
    /// Register a concrete service whose instance is managed by FileWatchDependencyFactory.
    public static IServiceCollection AddFileWatched<T>(IServiceCollection services)
;

    /// Register a service whose instance is managed by FileWatchDependencyFactory. The
factory (singleton) recreates the instance when the implementation's OnFileChanged returns
Recreate. The service (transient) always returns the current instance from the factory.
    public static IServiceCollection AddFileWatched<TService, TImplementation>
(IServiceCollection services)
;
}
```

Pennington.Infrastructure.FileWatchedValue

Pennington.Infrastructure.FileWatchedValue

CSHARP

```
namespace Pennington.Infrastructure;

/// A lazily-loaded value that reloads on next access when a file in its Scope changes.
Implements IFileWatchAware so FileWatchDispatcher drives the reload — this type holds no
IFileWatcher subscription of its own.
public class FileWatchedValue
{
    /// Creates the holder; the value is not loaded until Value is first read.
    public FileWatchedValue`1(FileWatchScope scope, Func<T> load)
;

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;

    /// The directory and pattern this value reloads for.
    public FileWatchScope Scope { get; }

    /// The current value, loaded on first access and reloaded after a change in Scope.
    public T Value { get; }

    /// Directories needing an OS-level watcher. Empty (the default) for aggregators that
ride notifications other watchers already produce.
    public IReadOnlyList<FileWatchScope> WatchScopes { get; }

}
```

Pennington.Infrastructure.FileWatcher

Pennington.Infrastructure.FileWatcher

CSHARP

```
namespace Pennington.Infrastructure;

/// Manages FileSystemWatcher instances and notifies subscribers of changes.
public class FileWatcher
{
    /// Watch a path for file changes matching a pattern.
    public void AddPathWatch(string path, string filePattern, Action<string,
WatcherChangeTypes> onFileChanged, bool includeSubdirectories = true)
;

    public void Dispose()
;

    /// Initializes the watcher with a filesystem abstraction, clock, and optional logger.
    public FileWatcher(IFileSystem fileSystem, TimeProvider clock = null,
ILogger<FileWatcher> logger = null)
;

    /// Subscribe to be notified when any watched file changes.
    public void SubscribeToChanges(Action onUpdate)
;

    /// Subscribe to be notified when any watched file changes, with the changed path and
change type.
    public void SubscribeToChanges(Action<FileChangeNotification> onUpdate)
;
}
```

Pennington.Infrastructure.FileWatchResponse

Pennington.Infrastructure.FileWatchResponse

CSHARP

```
namespace Pennington.Infrastructure;

/// How a file-watched service wants FileWatchDispatcher to treat a change.
public enum FileWatchResponse
{
    /// The change is irrelevant; nothing was done.
    public static const FileWatchResponse Ignore
;

    /// Drop the instance; a fresh one is built on the next resolve.
    public static const FileWatchResponse Recreate
;

    /// The service refreshed its own state in place; keep the instance.
    public static const FileWatchResponse Refreshed
;
}
```

Pennington.Infrastructure.FileWatchScope

Pennington.Infrastructure.FileWatchScope

CSHARP

```
namespace Pennington.Infrastructure;

/// A directory and file pattern an IFileWatchAware needs OS-level watching for.
public struct FileWatchScope
{
    /// A directory and file pattern an IFileWatchAware needs OS-level watching for.
    public FileWatchScope(string Path, string Pattern, bool IncludeSubdirectories = false)
;

    /// Whether changes in nested directories also count.
    public bool IncludeSubdirectories { get; set; }

    /// Returns whether change falls within this scope.
    public bool Matches(FileChangeNotification change)
;

    /// Absolute directory to watch.
    public string Path { get; set; }

    /// File pattern within Path (for example *.yml).
    public string Pattern { get; set; }

}
```

Pennington.Infrastructure.FontPreload

Pennington.Infrastructure.FontPreload

CSHARP

```
namespace Pennington.Infrastructure;

/// Represents a font file to preload via a link rel="preload" hint in the HTML head.
public record FontPreload
{
    /// Represents a font file to preload via a link rel="preload" hint in the HTML head.
    public FontPreload(string Href, string Type = "font/woff2")
;

    /// The URL path to the font file (e.g., "fonts/lexend.woff2").
    public string Href { get; set; }

    /// The MIME type of the font file. Defaults to "font/woff2".
    public string Type { get; set; }

}
```

Pennington.Infrastructure.HighlightingOptions

Pennington.Infrastructure.HighlightingOptions

CSHARP

```
namespace Pennington.Infrastructure;

/// Options for code highlighting configuration.
public class HighlightingOptions
{
    /// Registers a pre-built highlighter instance.
    public void AddHighlighter(ICodeHighlighter highlighter)
;

    /// Registers a highlighter type, constructed with its parameterless constructor.
    public void AddHighlighter<T>()
;

    /// Highlighters registered via AddHighlighter or the generic overload.
    public IReadOnlyList<ICodeHighlighter> Highlighters { get; }

}
```

Pennington.Infrastructure.HtmlToMarkdownConverter

Pennington.Infrastructure.HtmlToMarkdownConverter

CSHARP

```
namespace Pennington.Infrastructure;

/// Tiny, hand-rolled HTML → Markdown converter used for llms.txt output. Scope is deliberately minimal: the output is consumed by LLMs and doesn't need to round-trip to the original HTML. It covers headings, paragraphs, lists, links, inline/fenced code, blockquotes, images, horizontal rules, and basic inline formatting. Everything else recurses into text content.If this grows past ~250 lines, switch to a real library (e.g. ReverseMarkdown on NuGet) rather than expanding it further.
public class HtmlToMarkdownConverter
{
    /// Converts root to markdown.
    public static string Convert(IElement root, Func<string, string> rewriteHref = null)
;
}
```

Pennington.Infrastructure.HttpDispatcher

Pennington.Infrastructure.HttpDispatcher

CSHARP

```
namespace Pennington.Infrastructure;

/// Default IInProcessHttpDispatcher. Inspects the registered IServer and returns an in-memory client when it's a TestServer, or a socket-bound client pointing at Kestrel's listening address otherwise.
public class HttpDispatcher
{
    /// Returns an HttpClient whose requests flow through the running app's pipeline.
    public HttpClient CreateClient()
;

    /// Initializes the dispatcher with the host's registered IServer and the shared render cache.
    public HttpDispatcher(IServer server, BuildHtmlCache cache)
;
}
```

Pennington.Infrastructure.IFileWatchAware

Pennington.Infrastructure.IFileWatchAware

CSHARP

```
namespace Pennington.Infrastructure;

/// The single contract for anything that reacts to file-system changes. Implementers
declare the directories they need watched and how they respond to a change;
FileWatchDispatcher owns every IFileWatcher call.
public interface IFileWatchAware
{
    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;

    /// Directories needing an OS-level watcher. Empty (the default) for aggregators that
ride notifications other watchers already produce.
    public IReadOnlyList<FileWatchScope> WatchScopes { get; }

}
```

Pennington.Infrastructure.IFileWatcher

Pennington.Infrastructure.IFileWatcher

CSHARP

```
namespace Pennington.Infrastructure;

/// Watches file system paths for changes and notifies subscribers.
public interface IFileWatcher
{
    /// Watch a path for file changes matching a pattern.
    public void AddPathWatch(string path, string filePattern, Action<string,
WatcherChangeTypes> onFileChanged, bool includeSubdirectories = true)
;

    /// Subscribe to be notified when any watched file changes.
    public void SubscribeToChanges(Action onUpdate)
;

    /// Subscribe to be notified when any watched file changes, with the changed path and
change type.
    public void SubscribeToChanges(Action<FileChangeNotification> onUpdate)
;
}
```

Pennington.Infrastructure.IHtmlResponseRewriter

Pennington.Infrastructure.IHtmlResponseRewriter

CSHARP

```
namespace Pennington.Infrastructure;

/// A participant in the unified HTML response rewriting pipeline. Multiple rewriters share a single parsed IDocument per response, so the DOM is parsed and serialized exactly once regardless of how many rewriters apply. Consumed by HtmlResponseRewritingProcessor, which is the single IResponseProcessor doing HTML rewriting.
public interface IHtmlResponseRewriter
{
    /// Mutate the shared parsed document. The orchestrator serializes it once after every rewriter has run.
    public Task ApplyAsync(IDocument document, HttpContext context)
;

    /// Sort order within the HTML rewriting pipeline. Rewriters run in ascending Order. Xref resolution at 10, locale prefixing at 20, base-URL prefixing at 30 — the outside-in order that was previously expressed across separate IResponseProcessors.
    public int Order { get; }

    /// Regex / string pre-pass over the raw HTML before AngleSharp parses it. Exists for constructs that are not valid HTML (such as raw <xref:uid> tags) and therefore must be substituted out before the parser runs. Default: pass-through. Rewriters that only touch the DOM should not override this.
    public Task<string> PreParseAsync(string html, HttpContext context)
;

    /// Cheap gate checked before parsing. Return false to skip both PreParseAsync and ApplyAsync for this response. If every rewriter returns false, the orchestrator skips parsing entirely.
    public bool ShouldApply(HttpContext context)
;
}
```

Pennington.Infrastructure.IInProcessHttpDispatcher

Pennington.Infrastructure.IInProcessHttpDispatcher

CSHARP

```
namespace Pennington.Infrastructure;

/// Dispatches HTTP requests against the running app's pipeline. Replaces the "self-fetch
via Kestrel socket" pattern with a transport that the host can satisfy in-process (via
Microsoft.AspNetCore.TestHost.TestServer) or over the wire (via Kestrel's listening address)
depending on which IServer is registered. Internal services (LlmsTxtService,
SearchArtifactService, build crawler) take this instead of IHttpClientFactory so they don't
need to know whether the host is using TestServer (build / tests) or Kestrel (dev). The
middleware pipeline runs identically either way — the only difference is who delivers the
bytes.
public interface IInProcessHttpDispatcher
{
    /// Returns an HttpClient whose requests flow through the running app's pipeline.
    public HttpClient CreateClient()
;
}
```

Pennington.Infrastructure.IResponseProcessor

Pennington.Infrastructure.IResponseProcessor

CSHARP

```
namespace Pennington.Infrastructure;

/// Processes HTTP response bodies for a specific concern.
public interface IResponseProcessor
{
    /// Execution order; lower values run earlier in the response pipeline.
    public int Order { get; }

    /// Transforms responseBody and returns the processed body.
    public Task<string> ProcessAsync(string responseBody, HttpContext context)
;

    /// Returns true when this processor should run for the current request.
    public bool ShouldProcess(HttpContext context)
;
}
```

Pennington.Infrastructure.LinkCheckResult

Pennington.Infrastructure.LinkCheckResult

CSHARP

```
namespace Pennington.Infrastructure;

/// Outcome of checking a single link during build-time verification.
public struct LinkCheckResult
{
    /// A link that failed verification.
    public record BrokenLinkResult(ContentRoute SourcePage, string Url, LinkType Type,
string Reason) : object, IEquatable<BrokenLinkResult>

    /// A link that points to an external origin and was not verified by the internal
checker.
    public record ExternalLink(ContentRoute SourcePage, string Url) : object,
IEquatable<ExternalLink>

    /// Wraps a ValidLink.
    public LinkCheckResult(ValidLink value)
;

    /// Wraps a BrokenLinkResult.
    public LinkCheckResult(BrokenLinkResult value)
;

    /// Wraps an ExternalLink.
    public LinkCheckResult(ExternalLink value)
;

    /// A link that resolved to a known internal target.
    public record ValidLink(ContentRoute SourcePage, string Url) : object,
IEquatable<ValidLink>

    /// Wrapped case instance; inspect via pattern matching on the case types.
    public object Value { get; }

}
```

Pennington.Infrastructure.LinkVerificationService

Pennington.Infrastructure.LinkVerificationService

CSHARP

```
namespace Pennington.Infrastructure;

/// Verifies internal links in rendered HTML against known routes. Extracts href and src
attributes and classifies each as valid, broken, or external. Does not make HTTP requests —
this is purely static analysis.
public class LinkVerificationService
{
    /// Find internal page links in HTML that are missing a trailing slash. Returns the list
of offending URLs.
    public static ImmutableList<string> FindLinksWithoutTrailingSlash(string html)
;

    /// Create with the set of all known page canonical paths, the static asset paths the
engine copied into the output tree, and the base URL the surrounding site was rendered with.
The base URL is used to strip a common prefix (e.g. /preview) from extracted hrefs before
comparing against the unprefixed canonical CanonicalPath, and before applying the framework-
asset prefix check for /_content/ / /_framework/ / /_blazor/. Passing "/" (the default)
disables prefix stripping. copiedAssetPaths are relative output paths produced by
GetContentToCopyAsync (e.g. media/sample.svg); they get normalized into absolute root-
relative URLs (/media/sample.svg) and added to the known-paths set so that <img src>
references to assets the engine just copied aren't flagged as broken. artifactClaims are the
artifact tier's declared URL territories: a link landing inside one is treated as valid
without per-path verification, because the exact artifact set is lazy and may not be
enumerable on the request path. Build-mode callers that enumerate exact artifact routes into
knownRoutes should pass no claims, so typos inside a claimed territory still get flagged.
    public LinkVerificationService(IEnumerable<ContentRoute> knownRoutes,
IEnumerable<string> copiedAssetPaths = null, string baseUrl = "/",
IEnumerable<ArtifactClaim> artifactClaims = null)
;

    /// Verify all links found in a page's rendered HTML.
    public ImmutableList<LinkCheckResult> VerifyLinks(ContentRoute sourcePage, string html)
;
}
```

Pennington.Infrastructure.LinkVerificationServiceBuilder

Pennington.Infrastructure.LinkVerificationServiceBuilder

CSHARP

```
namespace Pennington.Infrastructure;

/// Shared builder for LinkVerificationService instances. Both the in-pipeline per-page verifier (PageLinkVerifier) and the corpus-wide build auditor (LinkAuditor) collect the same (knownRoutes, copiedAssetPaths, MapGet) triple from the content surface. Artifact-tier URLs are folded in one of two ways, selected by enumerateArtifactRoutes: false (request path): only the cheap, options-derived Claims are folded — a link into a claimed territory is trusted. Artifact discovery fans out through the site projection, which must never run on the request path (see ISiteProjection).true (build mode): exact artifact routes are enumerated into the known set and no claims are folded, so a typo inside a claimed territory is still flagged.
public class LinkVerificationServiceBuilder
{
    /// Builds a verifier from the content services, artifact services, endpoint table, and output options. Pass enumerateArtifactRoutes=true only from build-mode callers. Pass webRootFileProvider (the host's WebRootFileProvider) so wwwroot/RCL assets — copied by the build but owned by no content service — are treated as known assets.
    public static Task<LinkVerificationService> BuildAsync(IEnumerable<IContentService> contentServices, IEnumerable<IArtifactContentService> artifactServices, EndpointDataSource endpointDataSource, OutputOptions outputOptions, bool enumerateArtifactRoutes, IFileProvider webRootFileProvider = null, CancellationToken cancellationToken = default)
;
}
```

Pennington.Infrastructure.LiveReloadExtensions

Pennington.Infrastructure.LiveReloadExtensions

CSHARP

```
namespace Pennington.Infrastructure;

/// Extensions that wire the live reload WebSocket endpoint into the request pipeline.
public class LiveReloadExtensions
{
    /// Adds live reload WebSocket support for development. Skipped during static build (see PenningtonCli).
    public static WebApplication UseLiveReload(WebApplication app)
;
}
```

Pennington.Infrastructure.LiveReloadServer

Pennington.Infrastructure.LiveReloadServer

CSHARP

```
namespace Pennington.Infrastructure;

/// Manages WebSocket connections for live reload during development. When watched files
change, debounces rapid notifications and sends a single reload message to all connected
browsers.
public class LiveReloadServer
{
    /// Disposes the debounce timer and closes any open client sockets.
    public ValueTask DisposeAsync()
;

    /// Initializes the server and subscribes to watched-file change notifications.
    public LiveReloadServer(IFileWatcher fileWatcher, IHostApplicationLifetime lifetime,
ILogger<LiveReloadServer> logger = null)
;
}
```

Pennington.Infrastructure.MarkdownContentOptions

Pennington.Infrastructure.MarkdownContentOptions

CSHARP

```
namespace Pennington.Infrastructure;

/// Options for a markdown content source.
public class MarkdownContentOptions
{
    /// URL prefix prepended to routes generated from this source.
    public string BasePageUrl { get; set; }

    /// Filesystem path to the directory containing markdown files.
    public string ContentPath { get; set; }

    /// Relative subpaths (from ContentPath) to skip during discovery and content copying.
Set this on a broad catch-all source to carve out a subtree that is owned by a more specific
markdown source registered nearby. See MarkdownContentServiceOptions.ExcludePaths for
matching semantics.
    public ImmutableArray<string> ExcludePaths { get; set; }

    /// When true, a content-root 404.md is reserved as the not-found body: skipped during
discovery so it never becomes a routable page or a nav/sitemap/search/llms entry. Host
templates render it on demand as the 404 page. See
MarkdownContentServiceOptions.ReserveNotFoundPage.
    public bool ReserveNotFoundPage { get; set; }

    /// Default section label applied when front matter does not specify one.
    public string SectionLabel { get; set; }

}
```

Pennington.Infrastructure.NotFoundResponseExtensions

Pennington.Infrastructure.NotFoundResponseExtensions

CSHARP

```
namespace Pennington.Infrastructure;

/// Lets a content-resolving page signal that the requested route is missing. The marker is read by NotFoundStatusProcessor, which flips the status to 404 after the rendered 404 body (with its layout and chrome) has been produced — so pages set the marker instead of writing StatusCode directly.
public class NotFoundResponseExtensions
{
    /// Returns true when MarkNotFound has been called for this request.
    public static bool IsMarkedNotFound(HttpContext context)
;

    /// Marks the current request as having resolved to a missing route.
    public static void MarkNotFound(HttpContext context)
;
}
```

Pennington.Infrastructure.PageLinkVerifier

Pennington.Infrastructure.PageLinkVerifier

CSHARP

```
namespace Pennington.Infrastructure;

/// Holds a LinkVerificationService built from the current corpus of known routes and copied assets. File-watched so the known-paths set refreshes when content changes. Lets PageLinkAuditProcessor verify per-page links without running a corpus-wide rendered audit.
public class PageLinkVerifier
{
    /// Returns the current verifier; rebuilds on first access after a file change.
    public Task<LinkVerificationService> GetVerifierAsync()
;

    /// Called on the file-watcher thread for every watched change. Must be quick and thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;

    /// Creates the verifier; the underlying LinkVerificationService is built lazily on first request.
    public PageLinkVerifier(IEnumerable<IContentService> contentServices, IEnumerable<IArtifactContentService> artifactServices, EndpointDataSource endpointDataSource, OutputOptions outputOptions, IWebHostEnvironment environment)
;
}
```

Pennington.Infrastructure.PenningtonExtensions

Pennington.Infrastructure.PenningtonExtensions

CSHARP

```
namespace Pennington.Infrastructure;

/// DI and pipeline extensions that wire Pennington services into an ASP.NET Core host.
public class PenningtonExtensions
{
    /// Register all Pennington services.
    public static IServiceCollection AddPennington(IServiceCollection services,
Action<PenningtonOptions> configure)
;

    /// Register a source-generated YamlSerializerContext so the types it covers deserialize
without reflection (NativeAOT/trim-friendly). Types not covered by any registered context
fall back to reflection. Satellite templates call this for their own front-matter records;
end users call it for theirs.
    public static IServiceCollection AddYamlContext(IServiceCollection services,
YamlSerializerContext context)
;

    /// Runs the host: serves live (no verb), builds the static site (build), or runs a
diagnostic command (diag <sub>). Everything flows through one System.CommandLine pipeline,
so --help / --version work at the root and every subcommand. Build and diag run one-shot
against a started in-memory host that is disposed afterward; serve hands off to RunAsync.
    public static Task RunOrBuildAsync(WebApplication app, string[] args)
;

    /// Adds locale detection and URL path rewriting middleware. Must be called
MapRazorComponents so that Blazor routing sees the locale-stripped path (e.g., /gen-
z/schedule becomes /schedule). Called automatically by UsePennington when it hasn't been
called yet, but at that point it is too late for Blazor endpoint routing. Sites that use
@page directives with locale prefixes must call this explicitly.
    public static WebApplication UseLocaleRouting(WebApplication app)
;

    /// Configure the Pennington middleware pipeline.
    public static WebApplication UsePennington(WebApplication app)
;
}
```

Pennington.Infrastructure.PenningtonOptions

Pennington.Infrastructure.PenningtonOptions

CSHARP

```
namespace Pennington.Infrastructure;

/// Main configuration options for the Pennington content engine.
public class PenningtonOptions
{
    /// Register a content source for a custom file format. The format's IContentParser and IContentRenderer are supplied via UseParser / UseRenderer and the pipeline routes to them by format. The format's files are discovered, parsed, and rendered through the same pipeline as markdown.
    public ContentFormatOptions AddContentFormat<TFrontMatter>(string format, Action<ContentFormatOptions> configure)
;

    /// Extra assemblies to scan for routable @page Razor components. The entry assembly is always scanned, so a bare host need only set this to add components defined in other assemblies.
    public Assembly[] AdditionalRoutingAssemblies { get; set; }

    /// Enable llms.txt generation for this site.
    public LlmsTxtOptions AddLlmsTxt(Action<LlmsTxtOptions> configure = null)
;

    /// Register a markdown content source with a specific front matter type.
    public MarkdownContentOptions AddMarkdownContent<TFrontMatter>
(Action<MarkdownContentOptions> configure)
;

    /// Absolute base URL used to generate canonical, OpenGraph, and feed links.
    public string CanonicalBaseUrl { get; set; }

    /// Customize the Markdig pipeline after Pennington's built-in extensions (including Mdazor) are added. Runs with the resolved IServiceProvider so extensions requiring DI can be wired up.
    public Action<MarkdownPipelineBuilder, IServiceProvider> ConfigureMarkdownPipeline { get; set; }

    /// Content formats registered via AddContentFormat.
    public IReadOnlyList<ContentFormatOptions> ContentFormats { get; }

    /// Root filesystem directory containing site content.
    public FilePath ContentRootPath { get; set; }

    /// Favicon / icon link generation. Set to emit <link rel="icon"> (and apple-touch-icon, mask-icon, manifest) tags into every page head; null disables the feature. The icon files are user-provided static assets in the content root. Templates forward their own option into this.
    public FaviconOptions Favicons { get; set; }

    /// Front-matter parser configuration (strict-mode toggle).
    public FrontMatterParserOptions FrontMatter { get; }
```

```
    /// llms.txt options registered via AddLlmsTxt, or null when not enabled.    public
LlmsTxtOptions LlmsTxt { get; }      /// Localization configuration, including locales and
defaults.    public LocalizationOptions Localization { get; }      /// When true (the
default), UsePennington maps the /sitemap.xml endpoint. Set to false to suppress the
endpoint when the host environment supplies its own sitemap. Template extensions like
AddBlogSite forward their own toggle into this flag.    public bool MapSitemap { get; set; }
/// Markdown content sources registered via AddMarkdownContent.    public
IReadOnlyList<MarkdownContentOptions> MarkdownSources { get; }      /// Configuration for
the search index.    public SearchIndexOptions SearchIndex { get; }      /// Default site
description used for meta tags when a page supplies none.    public string SiteDescription {
get; set; }      /// Configuration for the shared ISiteProjection consumed by every corpus
aggregator (search index, llms.txt, build-time link audit). Set ContentSelector here when
the layout wraps content in chrome that should be stripped before indexing or extracting
markdown.    public SiteProjectionOptions SiteProjection { get; }      /// Site title shown
in the browser tab, OpenGraph tags, and RSS feed.    public string SiteTitle { get; set; }
/// Version of the subject the site documents (a library, product, or API) — emitted as the
version: line in llms.txt. When set it replaces the generator's own penningtonVersion:,
since a reader cares which release of the documented thing the content describes, not which
Pennington build produced it. Resolve it however suits the site (for example, a referenced
assembly's informational version); left null, llms.txt falls back to reporting the
Pennington version.    public string SiteVersion { get; set; }      /// Social-card
(OpenGraph / Twitter image) generation. Set to enable per-page card discovery, the on-demand
rendering endpoint, and the meta-tag wiring; null disables the feature. Templates forward
their own option into this.    public SocialCardOptions SocialCards { get; set; }      ///
Standard Site (AT Protocol long-form publishing) integration. Set to emit the verification
well-known files and per-page site.standard.* head links; null disables the feature.
Templates forward their own option into this.    public StandardSiteOptions StandardSite {
get; set; }      /// Site-level author fallback for auto-emitted JSON-LD, surfaced as
FallbackAuthorName when a record's front matter describes structured data but names no
author of its own. BlogSite forwards its BlogSiteOptions.AuthorName here.    public string
StructuredDataAuthorName { get; set; }      /// Override the CSS class names emitted by the
tabbed-code-block renderer. When set, the returned TabbedCodeBlockRenderOptions replaces the
Default shape on the pipeline's single registration of the tabbed extension.    public
Func<TabbedCodeBlockRenderOptions> TabbedCodeBlockOptions { get; set; }      /// Translation
string configuration for localized UI.    public TranslationOptions Translations { get; }  }
```

Pennington.Infrastructure.RenderedHtmlFetcher

Pennington.Infrastructure.RenderedHtmlFetcher

CSHARP

```
namespace Pennington.Infrastructure;

/// Fetches fully rendered page HTML from the running app and exposes a section of it (via
CSS selector) as an AngleSharp IElement. SiteProjection (its sole consumer) uses this to get
post-pipeline HTML — i.e., after Markdig extensions, Razor SSR, xref resolution, locale
rewriting, and any other middleware have run. The pre-pipeline IContentRenderer path misses
Razor pages entirely and misses request-pipeline transforms for everything else.Requests are
dispatched via IInProcessHttpDispatcher, which delivers them in-memory through TestServer
(build mode + integration tests) or over Kestrel's listening socket (dev mode). The
middleware pipeline runs identically in either case. Every fetch carries HeaderName so the
served request can fail fast (instead of deadlocking) if its render path awaits the
projection — a future consumer of this fetcher outside the projection inherits that header
and its tripwire semantics.
public class RenderedHtmlFetcher
{
    /// Fetches path from the running app, parses the response body as HTML, and returns the
element matching selector. When selector is null or no match is found, returns Body. Returns
null on non-success responses.
    public Task<IElement> FetchContentAsync(string path, string selector, CancellationToken
ct = default)
;

    /// Initializes the fetcher with the in-process dispatcher and a logger.
    public RenderedHtmlFetcher(IInProcessHttpDispatcher dispatcher,
ILogger<RenderedHtmlFetcher> logger)
;
}
```

Pennington.Infrastructure.SelfFetchUnavailableException

Pennington.Infrastructure.SelfFetchUnavailableException

CSHARP

```
namespace Pennington.Infrastructure;

/// Thrown by CreateClient when the in-process transport is not ready — the host's IServer
has not started yet (a TestServer whose application is still null, or a Kestrel host that
has not bound a listening address). Distinct from a per-page content failure: site-crawling
consumers (notably SiteProjection) must let this propagate so a partially-built or empty
corpus is never cached as if the crawl had completed.
public class SelfFetchUnavailableException
{
    /// Initializes the exception with a message describing why the transport is
unavailable.
    public SelfFetchUnavailableException(string message)
;

    /// Initializes the exception with a message and the underlying cause.
    public SelfFetchUnavailableException(string message, Exception innerException)
;
}
```

Pennington.Infrastructure.ValidLink

Pennington.Infrastructure.ValidLink

CSHARP

```
namespace Pennington.Infrastructure;

/// A link that resolved to a known internal target.
public record ValidLink
{
    /// Page that contained the link.
    public ContentRoute SourcePage { get; set; }

    /// Link target URL.
    public string Url { get; set; }

    /// A link that resolved to a known internal target.
    public ValidLink(ContentRoute SourcePage, string Url)
;
}
```

Pennington.Infrastructure.WordBreakExtensions

Pennington.Infrastructure.WordBreakExtensions

CSHARP

```
namespace Pennington.Infrastructure;

/// Opt-in registration for word-break typography.
public class WordBreakExtensions
{
    /// Registers WordBreakHtmlRewriter in the shared HTML rewriting pipeline, so long identifiers in the configured elements get <wbr> break opportunities without an extra DOM parse.
    public static IServiceCollection AddWordBreak(IServiceCollection services, Action<WordBreakOptions> configure = null)
;
}
```

Pennington.Infrastructure.WordBreakOptions

Pennington.Infrastructure.WordBreakOptions

CSHARP

```
namespace Pennington.Infrastructure;

/// Configures WordBreakHtmlRewriter — which elements receive word-break opportunities and how each break is rendered.
public class WordBreakOptions
{
    /// CSS selector identifying the elements whose text gets word-break opportunities. Only elements that contain text and no child elements are rewritten; to reach text wrapped in an inline element, name that element directly (for example add span) rather than using a descendant combinator like h1 * — descendant combinators force AngleSharp to scan every element on the page. Defaults to headings, spans, and the .text-break class.
    public string CssSelector { get; set; }

    /// Minimum length a word must reach before any break is inserted. Defaults to 20.
    public int MinimumCharacters { get; set; }

    /// Markup inserted at each break opportunity. Defaults to <wbr>.
    public string WordBreakCharacters { get; set; }

}
```

Pennington.Infrastructure.XrefResolver

Pennington.Infrastructure.XrefResolver

CSHARP

```
namespace Pennington.Infrastructure;

/// Resolves cross-reference UIDs to URLs and titles. Builds a case-insensitive lookup from
all registered content services on first use. When managed by FileWatchDependencyFactory,
the instance is recreated on file changes, ensuring fresh data from content services.
public class XrefResolver
{
    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;

    /// Returns the CrossReference for uid, or null when not found.
    public Task<CrossReference> ResolveAsync(string uid)
;

    /// Initializes the resolver and prepares lazy aggregation of UID entries across content
services.
    public XrefResolver(IEnumerable<IContentService> contentServices)
;
}
```

Pennington.Infrastructure.XrefResolvingService

Pennington.Infrastructure.XrefResolvingService

CSHARP

```
namespace Pennington.Infrastructure;

/// Resolves xref: cross-reference links in HTML. Two phases:ResolveXrefTagsAsync
substitutes raw <xref:uid> tags via regex. These are not valid HTML, so they must be
rewritten before any DOM parser sees them.ResolveXrefLinksAsync walks a[href^='xref:'] on an
already-parsed document.The response pipeline calls both phases via XrefHtmlRewriter,
reusing the orchestrator's shared document. The SPA data endpoint for island HTML fragments
calls the combined ResolveAsync entrypoint, which performs its own parse/serialize because
it receives raw HTML strings with no caller-owned document.Registered as a transient so
every resolution picks up the current file-watched XrefResolver instance. The orchestrator
(and rewriter that wraps this service) are transient too; the whole chain is rebuilt per
request when the middleware resolves its processors, so no capture pins a stale
XrefResolver.
public class XrefResolvingService
{
    /// Standalone entrypoint for callers that have a raw HTML string and no document (SPA
island fragments). Returns resolved HTML.
    public Task<string> ResolveAsync(string html, DiagnosticContext diagnostics = null)
;

    /// Phase 2: DOM rewrite of a[href^='xref:'] links on an already-parsed document.
Returns true when any link was rewritten — callers that used ResolveAsync use this to decide
whether to re-serialize the document.
    public Task<bool> ResolveXrefLinksAsync(IDocument document, DiagnosticContext
diagnostics)
;

    /// Phase 1: regex substitution of raw <xref:uid> tags. Produces an <a> element that the
later DOM phase (or any downstream HTML parser) can see as normal markup. Skips content
inside <code> and <pre> blocks so authoring docs can show literal <xref:uid> samples in
fenced code without the rewriter latching onto them — the highlighter also splits long
tokens across span boundaries, which would otherwise let [^>]+ consume span markup as the
uid.
    public Task<string> ResolveXrefTagsAsync(string html, DiagnosticContext diagnostics)
;

    /// Initializes the service with the current XrefResolver from DI.
    public XrefResolvingService(XrefResolver resolver)
;
}
```

Pennington.LlmsTxt.ILlmsSubtreeProvider

Pennington.LlmsTxt.ILlmsSubtreeProvider

CSHARP

```
namespace Pennington.LlmsTxt;

/// Optional capability for a IContentService to surface subtree declarations discovered during its own scan (for example, _meta.yml sidecars with an llms block under a markdown content tree).
public interface ILlmsSubtreeProvider
{
    /// Returns the subtrees declared by this provider's content.
    public Task<ImmutableList<LlmsSubtree>> GetLlmsSubtreesAsync()
;
}
```

Pennington.LlmsTxt.LlmsArtifactContentService

Pennington.LlmsTxt.LlmsArtifactContentService

CSHARP

```
namespace Pennington.LlmsTxt;

/// Artifact-tier façade over LlmsTxtService: one service owning every llms URL — the root /llms.txt front door (and optional /llms-full.txt), the per-subtree {prefix}/llms.txt indexes (a mid-path territory only a SuffixClaim can express — endpoint routing would let a content route's {slug} segment capture them), and the per-page co-located {route}.md markdown (root page at /index.md). Serves dev requests through the artifact router and enumerates the same files for the static build. Transient so each resolution captures the current file-watched service.
public class LlmsArtifactContentService
{
    /// URL territories this service serves. Options-derived and consulted on every request — must be cheap and must not trigger discovery, the projection, or any lazy corpus work.
    public ImmutableList<ArtifactClaim> Claims { get; }

    /// Enumerates every artifact route the static build should write, as GeneratedSource items. May consume the projection — the build invokes this outside any request, after the page crawl has primed the render cache. Never called on the request path.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Creates the façade; claims derive from the options alone.
    public LlmsArtifactContentService(LlmsTxtService service, LlmsTxtOptions options)
;

    /// Returns the bytes for relativePath (no leading slash, e.g. search/en/index.json), or null to decline so the request falls through to content routing. May materialize the projection, build an index, or run Chromium on demand.
    public Task<ArtifactContent> ResolveAsync(string relativePath, CancellationToken cancellationToken)
;
}
```

Pennington.LlmsTxt.LlmsSubtree

Pennington.LlmsTxt.LlmsSubtree

CSHARP

```
namespace Pennington.LlmsTxt;

/// Declares that all leaves under RoutePrefix should be split out of the main /llms.txt
into a dedicated {RoutePrefix}llms.txt, leaving the front door with a single see-also
pointer line.
public record LlmsSubtree
{
    /// Short blurb rendered after the title and as the see-also pointer description.
    public string Description { get; }

    /// Initializes a subtree, normalizing routePrefix to /foo/bar/ form.
    public LlmsSubtree(string routePrefix, string title, string description)
;

    /// URL prefix in canonical /foo/bar/ form (always leading and trailing slash).
    public string RoutePrefix { get; }

    /// Header rendered at the top of the subtree's llms.txt.
    public string Title { get; }

}
```

Pennington.LlmsTxt.LlmsTxtEndpointExtensions

Pennington.LlmsTxt.LlmsTxtEndpointExtensions

CSHARP

```
namespace Pennington.LlmsTxt;

/// Endpoint convention extensions for opting a MapGet route into the generated llms.txt
index.
public class LlmsTxtEndpointExtensions
{
    /// Registers the endpoint as an llms.txt entry. The route's URL is the link target in
the generated index — typical use is a MapGet("/_llms/{slug}.md", () =>
Results.Text(markdown, "text/markdown")) that returns markdown directly. No HTML page is
manufactured, and no sidecar is generated; the user-supplied response IS the content
llms.txt consumers fetch.
    public static TBuilder WithLlmsTxtEntry<TBuilder>(TBuilder builder, string title, string
description = null)
;
}
```

Pennington.LlmsTxt.LlmsTxtEntryMetadata

Pennington.LlmsTxt.LlmsTxtEntryMetadata

CSHARP

```
namespace Pennington.LlmsTxt;

/// Endpoint metadata that surfaces a MapGet route in llms.txt without producing an HTML
page. Attached to a route via WithLlmsTxtEntry; consumed by LlmsTxtService to fold the
endpoint's URL into the generated index alongside discovered markdown entries.
public record LlmsTxtEntryMetadata
{
    /// Optional one-line description rendered after the link.
    public string Description { get; set; }

    /// Endpoint metadata that surfaces a MapGet route in llms.txt without producing an HTML
page. Attached to a route via WithLlmsTxtEntry; consumed by LlmsTxtService to fold the
endpoint's URL into the generated index alongside discovered markdown entries.
    public LlmsTxtEntryMetadata(string Title, string Description = null)
;

    /// Display title shown in the llms.txt front door.
    public string Title { get; set; }

}
```

Pennington.LlmsTxt.LlmsTxtOptions

Pennington.LlmsTxt.LlmsTxtOptions

CSHARP

```
namespace Pennington.LlmsTxt;

/// Configuration for llms.txt generation.
public class LlmsTxtOptions
{
    /// Whether to also generate llms-full.txt with all content concatenated.
    public bool GenerateFullFile { get; set; }

}
```

Pennington.LlmsTxt.LlmsTxtService

Pennington.LlmsTxt.LlmsTxtService

CSHARP

```
namespace Pennington.LlmsTxt;

/// Generates llms.txt index and stripped markdown files. When managed by
FileWatchDependencyFactory, the instance is recreated on file changes — no manual watcher
subscription needed. Folds over ISiteProjection: every renderable page's post-pipeline HTML
is already captured by the shared projection, so this service is a pure adapter from
RenderedPage + HtmlToMarkdownConverter to the front-door index, per-subtree index files, and
per-page sidecar markdown.
public class LlmsTxtService
{
    /// Returns the optional concatenated llms-full.txt content, or null when disabled.
    public Task<string> GetLlmsFullTxtAsync()
;

    /// Returns the generated llms.txt index content.
    public Task<string> GetLlmsTxtAsync()
;

    /// Returns the per-page stripped markdown files emitted alongside llms.txt.
    public Task<ImmutableList<MarkdownFile>> GetMarkdownFilesAsync()
;

    /// Returns per-subtree {prefix}llms.txt index files split out of the front door.
    public Task<ImmutableList<MarkdownFile>> GetSubtreeFilesAsync()
;

    /// Creates the service; data is computed lazily on first request.
    public LlmsTxtService(ISiteProjection projection, IEnumerable<IContentService>
contentServices, IEnumerable<LlmsSubtree> subtrees, IFileSystem fileSystem,
IWebHostEnvironment hostingEnvironment, PenningtonOptions pennOptions, LlmsTxtOptions
llmsTxtOptions, CanonicalBaseUrl canonicalBase, NavigationBuilder navigationBuilder,
ILogger<LlmsTxtService> logger)
;

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;
}
```

Pennington.LlmsTxt.LlmsTxtServiceExtensions

Pennington.LlmsTxt.LlmsTxtServiceExtensions

CSHARP

```
namespace Pennington.LlmsTxt;

/// DI extension methods for the llms.txt feature.
public class LlmsTxtServiceExtensions
{
    /// Registers a LlmsSubtree so all leaves under RoutePrefix get split out into a
dedicated {RoutePrefix}llms.txt. Multiple registrations are allowed; programmatic
registrations override _meta.yml-discovered subtrees with the same prefix.
    public static IServiceCollection AddLlmsSubtree(IServiceCollection services, LlmsSubtree
subtree)
;
}
```

Pennington.LlmsTxt.MarkdownFile

Pennington.LlmsTxt.MarkdownFile

CSHARP

```
namespace Pennington.LlmsTxt;

/// A stripped markdown file produced for the llms output.
public record MarkdownFile
{
    /// UTF-8 bytes of the stripped markdown body.
    public byte[] Content { get; set; }

    /// A stripped markdown file produced for the llms output.
    public MarkdownFile(FilePath OutputPath, byte[] Content)
;

    /// Relative output path for the markdown file.
    public FilePath OutputPath { get; set; }

}
```

Pennington.Localization.AlternateLanguage

Pennington.Localization.AlternateLanguage

CSHARP

```
namespace Pennington.Localization;

/// One language version of a page, used for language switchers and hreflang link tags.
Content-route-independent (pure URL math).
public record AlternateLanguage
{
    /// One language version of a page, used for language switchers and hreflang link tags.
Content-route-independent (pure URL math).
    public AlternateLanguage(string Locale, string DisplayName, string HtmlLang, string Url,
bool IsCurrentLocale = false)
;

    /// User-visible language name.
    public string DisplayName { get; set; }

    /// Value to emit in hreflang and lang attributes.
    public string HtmlLang { get; set; }

    /// True when this entry represents the current request locale.
    public bool IsCurrentLocale { get; set; }

    /// Locale code (e.g. en, fr, pt-BR).
    public string Locale { get; set; }

    /// URL of the page in this locale.
    public string Url { get; set; }

}
```

Pennington.Localization.FallbackLangHtmlRewriter

Pennington.Localization.FallbackLangHtmlRewriter

CSHARP

```
namespace Pennington.Localization;

/// When a non-default-locale request was served with content from a different locale (fallback), rewrites <html lang="..." dir="..."> to match the actual content locale so screen readers and lang-aware tooling don't misidentify the body's language. Signal carrier: pages that resolve a fallback set HttpContext.Items["Pennington.FallbackContentLocale"] to the locale code whose content was actually rendered. Absence of that key means no rewrite.
public class FallbackLangHtmlRewriter
{
    /// Mutate the shared parsed document. The orchestrator serializes it once after every rewriter has run.
    public Task ApplyAsync(IDocument document, HttpContext context)
;

    /// Key written by content-resolving pages when a fallback was used.
    public static const string FallbackContentLocaleKey
;

    /// Creates the rewriter.
    public FallbackLangHtmlRewriter(LocalizationOptions localization)
;

    /// Sort order within the HTML rewriting pipeline. Rewriters run in ascending Order. Xref resolution at 10, locale prefixing at 20, base-URL prefixing at 30 — the outside-in order that was previously expressed across separate IResponseProcessors.
    public int Order { get; }

    /// Cheap gate checked before parsing. Return false to skip both PreParseAsync and ApplyAsync for this response. If every rewriter returns false, the orchestrator skips parsing entirely.
    public bool ShouldApply(HttpContext context)
;
}
```

Pennington.Localization.LocaleContext

Pennington.Localization.LocaleContext

CSHARP

```
namespace Pennington.Localization;

/// Scoped per-request locale context, set by LocaleDetectionMiddleware. Provides the
current locale and locale-aware URL building for Razor components. Analogous to Astro's
Astro.currentLocale.
public class LocaleContext
{
    /// The request URL with locale prefix stripped (e.g., "/schedule" regardless of
locale).
    public string ContentPath { get; set; }

    /// Text direction for this locale ("ltr" or "rtl").
    public string Direction { get; }

    /// The HTML lang attribute value for this locale.
    public string HtmlLang { get; }

    /// Metadata for this locale.
    public LocaleInfo Info { get; set; }

    /// True when the current locale is the default locale.
    public bool IsDefaultLocale { get; set; }

    /// The Pennington locale code for this request (e.g., "en", "fr", "gen-z").
    public string Locale { get; set; }

    /// Creates the context, initialized to the default locale until middleware populates
the request's locale.
    public LocaleContext(LocalizationOptions localization)
;

    /// Builds a locale-aware URL from a content path. For the default locale, returns the
path as-is. For other locales, prefixes with /{locale}/.
    public string Url(string path)
;
}
```

Pennington.Localization.LocaleInfo

Pennington.Localization.LocaleInfo

CSHARP

```
namespace Pennington.Localization;

/// Metadata describing a configured locale.
public record LocaleInfo
{
    /// Text direction, "ltr" or "rtl".
    public string Direction { get; set; }

    /// Human-readable name for the locale (e.g., "English", "Français").
    public string DisplayName { get; set; }

    /// Optional HTML lang attribute value; when null, the locale code is used.
    public string HtmlLang { get; set; }

    /// Metadata describing a configured locale.
    public LocaleInfo(string DisplayName, string Direction = "ltr", string HtmlLang = null)
;
}
```

Pennington.Localization.LocalizationOptions

Pennington.Localization.LocalizationOptions

CSHARP

```
namespace Pennington.Localization;

/// Options for localization.
public class LocalizationOptions
{
    /// Registers a locale with the supplied metadata.
    public void AddLocale(string code, LocaleInfo info)
;

    /// Registers a locale with just a display name.
    public void AddLocale(string code, string displayName)
;

    /// Builds a full URL for a content path in a specific locale.
    public string BuildLocaleUrl(string contentPath, string locale)
;

    /// Locale code used when no URL locale prefix is present.
    public string DefaultLocale { get; set; }

    /// Gets alternate language versions for a page URL across all configured locales. Pure
URL math — does not check if content exists (fallback handles that).
    public IReadOnlyList<AlternateLanguage> GetAlternateLanguages(string url)
;

    /// Extracts the locale code from a URL path. Returns the default locale when the first
segment is not a known non-default locale.
    public string GetLocaleFromUrl(string url)
;

    /// True when more than one locale is configured.
    public bool IsMultiLocale { get; }

    /// Configured locales keyed by locale code.
    public IReadOnlyDictionary<string, LocaleInfo> Locales { get; }

    /// Strips the locale prefix from a URL, returning the content-relative path. For the
default locale (no prefix), returns the URL unchanged.
    public string StripLocalePrefix(string url, string locale)
;
}
```

Pennington.Localization.PenningtonStringLocalizer

Pennington.Localization.PenningtonStringLocalizer

CSHARP

```
namespace Pennington.Localization;

/// An IStringLocalizer backed by TranslationOptions. Reads the locale that
LocaleDetectionMiddleware already detected for the request from the scoped LocaleContext,
and looks up translations with fallback to the default locale, then to the key itself.
public class PenningtonStringLocalizer
{
    /// Enumerates registered strings for the active locale, optionally including default-
locale fallbacks.
    public IEnumerable<LocalizedString> GetAllStrings(bool includeParentCultures)
;

    /// Returns the translation for name, or the key itself when no translation is
registered.
    public LocalizedString Item { get; }

    /// Returns the translation for name formatted with arguments.
    public LocalizedString Item { get; }

    /// Creates the localizer. httpContextAccessor supplies the per-request LocaleContext;
when absent (outside a request) the default locale is used.
    public PenningtonStringLocalizer(TranslationOptions translations, LocalizationOptions
localization, IHttpContextAccessor httpContextAccessor = null)
;
}
```

Pennington.Localization.PenningtonStringLocalizerFactory

Pennington.Localization.PenningtonStringLocalizerFactory

CSHARP

```
namespace Pennington.Localization;

/// An IStringLocalizerFactory that returns PenningtonStringLocalizer instances backed by
TranslationOptions. All localizer instances share the same translation dictionary — the
type/location parameters are ignored.
public class PenningtonStringLocalizerFactory
{
    /// Returns the shared localizer; resourceSource is ignored.
    public IStringLocalizer Create(Type resourceSource)
;

    /// Returns the shared localizer; baseName and location are ignored.
    public IStringLocalizer Create(string baseName, string location)
;

    /// Creates the factory with the shared localizer instance.
    public PenningtonStringLocalizerFactory(TranslationOptions translations,
LocalizationOptions localization, IHttpContextAccessor httpContextAccessor)
;
}
```

Pennington.Localization.PenningtonUrlRequestCultureProvider

Pennington.Localization.PenningtonUrlRequestCultureProvider

CSHARP

```
namespace Pennington.Localization;

/// An IRequestCultureProvider that reads the locale from the URL path prefix and maps it to
the closest CultureInfo for ASP.NET's request localization pipeline.
public class PenningtonUrlRequestCultureProvider
{
    /// Derives the request culture from the URL locale prefix.
    public Task<ProviderCultureResult> DetermineProviderCultureResult(HttpContext
httpContext)
;

    /// Creates the provider.
    public PenningtonUrlRequestCultureProvider(LocalizationOptions localization)
;
}
```

Pennington.Localization.TranslationOptions

Pennington.Localization.TranslationOptions

CSHARP

```
namespace Pennington.Localization;

/// In-memory store for UI string translations, keyed by locale and string key. Configured
in Translations and consumed by PenningtonStringLocalizer.
public class TranslationOptions
{
    /// Add a single translation entry.
    public void Add(string locale, string key, string value)
;

    /// Add multiple translation entries for a locale.
    public void Add(string locale, Dictionary<string, string> entries)
;

    /// Look up a translation by locale and key. Returns null if not found.
    public string Get(string locale, string key)
;
}
```

Pennington.Markdown.Extensions.CodeBlockPreprocessResult

Pennington.Markdown.Extensions.CodeBlockPreprocessResult

CSHARP

```
namespace Pennington.Markdown.Extensions;

/// Result from a code block preprocessor.
public record CodeBlockPreprocessResult
{
    /// The base language for CSS class purposes.
    public string BaseLanguage { get; set; }

    /// Result from a code block preprocessor.
    public CodeBlockPreprocessResult(string HighlightedHtml, string BaseLanguage, bool
SkipTransform = false)
;

    /// Fully highlighted HTML (wrapped in pre/code tags).
    public string HighlightedHtml { get; set; }

    /// If true, skip CodeTransformer on the output.
    public bool SkipTransform { get; set; }

}
```

Pennington.Markdown.Extensions.CodeBlockRenderingService

Pennington.Markdown.Extensions.CodeBlockRenderingService

CSHARP

```
namespace Pennington.Markdown.Extensions;

/// Shared rendering pipeline for fenced code blocks. Runs registered
ICodeBlockPreprocessors first, falls back to HighlightingService on the base language,
applies CodeTransformer, and wraps the result via CodeBlockHtmlBuilder. Used by both the
Markdig renderer and the <CodeBlock> Razor component so markdown fences and Razor usages
produce identical HTML.
public class CodeBlockRenderingService
{
    /// Creates the service with a highlighting dispatcher and the registered preprocessors
(ordered by descending priority at construction).
    public CodeBlockRenderingService(HighlightingService highlightingService,
IEnumerable<ICodeBlockPreprocessor> preprocessors = null)
;

    /// Renders code with language tag languageId through the full pipeline.
    public string Render(string code, string languageId, CodeHighlightRenderOptions options
= null, bool isInTabGroup = false)
;
}
```

Pennington.Markdown.Extensions.CodeHighlightRenderOptions

Pennington.Markdown.Extensions.CodeHighlightRenderOptions

CSHARP

```
namespace Pennington.Markdown.Extensions;

/// Options for customizing the CSS classes used in the code highlight renderer.
public record CodeHighlightRenderOptions
{
    /// Default CSS class configuration used by the code highlight renderer.
    public static readonly CodeHighlightRenderOptions Default
;

    /// CSS class for the outer wrapper element.
    public string OuterWrapperCss { get; set; }

    /// CSS classes for the Pre element.
    public string PreBaseCss { get; set; }

    /// Additional CSS classes for the Pre element when not in a tabbed code block.
    public string PreStandaloneCss { get; set; }

    /// CSS classes for the container when not in a tabbed code block.
    public string StandaloneContainerCss { get; set; }

}
```

Pennington.Markdown.Extensions.ICodeBlockPreprocessor

Pennington.Markdown.Extensions.ICodeBlockPreprocessor

CSHARP

```
namespace Pennington.Markdown.Extensions;

/// Preprocesses fenced code blocks before normal highlighting. Implementations can
intercept blocks with specific language modifiers (e.g., "csharp:xmldocid") and provide pre-
highlighted HTML.
public interface ICodeBlockPreprocessor
{
    /// Priority — higher runs first.
    public int Priority { get; }

    /// Attempts to preprocess a code block. Returns a result if handled, or null to pass
through.
    public CodeBlockPreprocessResult TryProcess(string code, string languageId)
;
}
```

Pennington.Markdown.Extensions.Tabs.ContentTabsRenderOptions

Pennington.Markdown.Extensions.Tabs.ContentTabsRenderOptions

CSHARP

```csharp
namespace Pennington.Markdown.Extensions.Tabs;

/// Options for customizing the CSS classes used in the content-tabs renderer. The tab strip
carries not-prose while the panels do not, so panel content keeps the page's prose
typography.
public record ContentTabsRenderOptions
{
    /// CSS classes for the outer container; intentionally not not-prose.
    public string ContainerCss { get; set; }

    /// Default CSS class configuration used by the content-tabs renderer.
    public static readonly ContentTabsRenderOptions Default
;

    /// CSS classes for each tab button.
    public string TabButtonCss { get; set; }

    /// CSS classes for the tab strip; carries not-prose to isolate the buttons.
    public string TabListCss { get; set; }

    /// CSS classes for each tab panel; intentionally not not-prose.
    public string TabPanelCss { get; set; }

}
```

Pennington.Markdown.Extensions.Tabs.TabbedCodeBlockRenderOptio

Pennington.Markdown.Extensions.Tabs.TabbedCodeBlockRenderOptio

CSHARP

```
namespace Pennington.Markdown.Extensions.Tabs;

/// Options for customizing the CSS classes used in the tabbed code block renderer.
public record TabbedCodeBlockRenderOptions
{
    /// CSS classes for the container.
    public string ContainerCss { get; set; }

    /// Default CSS class configuration used by the tabbed code block renderer.
    public static readonly TabbedCodeBlockRenderOptions Default
;

    /// CSS class for the outer wrapper element.
    public string OuterWrapperCss { get; set; }

    /// CSS classes for the tab buttons.
    public string TabButtonCss { get; set; }

    /// CSS classes for the tab list.
    public string TabListCss { get; set; }

    /// CSS classes for the tab panels.
    public string TabPanelCss { get; set; }

}
```

Pennington.Markdown.IncludeExpander

Pennington.Markdown.IncludeExpander

CSHARP

```
namespace Pennington.Markdown;

/// Expands DocFX-style [!INCLUDE [title](path)] directives by splicing the referenced
Markdown file's content in place. Targets are resolved relative to the referencing file and
expanded recursively; a missing or cyclic target collapses to an HTML comment so the build
still completes. Directives inside fenced code blocks are left verbatim so syntax can be
documented.
public class IncludeExpander
{
    /// Expands every include directive in markdown. Paths resolve relative to sourceFile;
relative links inside included content are not rebased, so they resolve as if written in the
host page.
    public static string Expand(string markdown, FilePath sourceFile, IFileSystem
fileSystem)
;
}
```

Pennington.Markdown.MarkdownContentParser

Pennington.Markdown.MarkdownContentParser

CSHARP

```
namespace Pennington.Markdown;

/// Parses discovered markdown files into ParsedItems using FrontMatterParser.
public class MarkdownContentParser
{
    /// Creates the parser.
    public MarkdownContentParser`1(FrontMatterParser frontMatterParser, IFileSystem
fileSystem)
;

    /// Parse a discovered item. Returns ParsedItem on success, FailedItem on failure.
    public Task<ContentItem> ParseAsync(DiscoveredItem item)
;
}
```

Pennington.Markdown.MarkdownContentRenderer

Pennington.Markdown.MarkdownContentRenderer

CSHARP

```
namespace Pennington.Markdown;

/// Renders parsed markdown items to HTML using Markdig. After rendering, relative author-
written links (e.g. ../how-to/foo.md, sample-post, ./image.png) are rewritten to absolute
canonical URLs via MarkdownLinkResolver.
public class MarkdownContentRenderer
{
    /// Creates the renderer; the default Markdig pipeline is used when none is supplied.
    public MarkdownContentRenderer(MarkdownPipeline pipeline = null, MarkdownLinkResolver
linkResolver = null, IFileSystem fileSystem = null, ShortcodeExpander shortcodeExpander =
null)
;

    /// Render a parsed item. Returns RenderedItem on success, FailedItem on failure.
    public Task<ContentItem> RenderAsync(ParsedItem item)
;
}
```

Pennington.Markdown.MarkdownLinkResolver

Pennington.Markdown.MarkdownLinkResolver

CSHARP

```
namespace Pennington.Markdown;

/// Resolves author-written relative links inside markdown bodies to absolute canonical
URLs. Handles three cases: ../how-to/foo.md — rewrites to the canonical URL of the target
markdown file and strips the .md suffix.sample-post (no extension) — treated as a sibling
markdown reference, resolved against the source file's directory and looked up by trying
common extensions../image.png or other non-markdown relative assets — resolved against the
source file's directory and emitted as an absolute URL relative to the owning content
source's base URL. The index is built lazily from all registered IContentService instances.
When managed by FileWatchDependencyFactory, the instance is recreated on file changes so the
index stays fresh.
public class MarkdownLinkResolver
{
    /// Creates the resolver; the link index is built lazily on first resolution.
    public MarkdownLinkResolver(IEnumerable<IContentService> contentServices)
;

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;

    /// Resolve a markdown-author-written href against the source file that contains it.
Returns the rewritten href, or null if the href is external / absolute / unresolvable and
should be left untouched.
    public ValueTask<string> ResolveAsync(FilePath sourceFile, string href)
;
}
```

Pennington.Markdown.MarkdownOutlineGenerator

Pennington.Markdown.MarkdownOutlineGenerator

CSHARP

```
namespace Pennington.Markdown;

/// Extracts heading outline from a parsed Markdown document.
public class MarkdownOutlineGenerator
{
    /// Produces outline entries for every heading in the document that has an id attribute.
    public static OutlineEntry[] GenerateOutline(MarkdownDocument document)
;
}
```

Pennington.Markdown.MarkdownPipelineFactory

Pennington.Markdown.MarkdownPipelineFactory

CSHARP

```
namespace Pennington.Markdown;

/// Creates a configured Markdig MarkdownPipeline.
public class MarkdownPipelineFactory
{
    /// Creates a basic Markdig pipeline with advanced extensions and YAML front matter.
    public static MarkdownPipeline CreateDefault()
;

    /// Creates a pipeline with syntax highlighting, tabbed code blocks, custom alerts, and Mdazor component rendering. The optional configure hook runs after built-in extensions so consumers can add their own.
    public static MarkdownPipeline CreateWithExtensions(IServiceProvider serviceProvider, CodeBlockRenderingService renderingService, Func<CodeHighlightRenderOptions> codeOptions = null, Func<TabbedCodeBlockRenderOptions> tabOptions = null, Action<MarkdownPipelineBuilder, IServiceProvider> configure = null)
;
}
```

Pennington.Markdown.Shortcodes.AssemblyVersionShortcode

Pennington.Markdown.Shortcodes.AssemblyVersionShortcode

CSHARP

```
namespace Pennington.Markdown.Shortcodes;

/// Built-in shortcode that emits the running host application's version string. Resolves
GetEntryAssembly, then prefers the assembly's informational version (which captures Git SHA
/ pre-release suffixes set by MSBuild) and falls back to the file version and finally
"unknown" when no entry assembly is available (test hosts, some embedded scenarios). The
optional format named argument accepts full (default), major, minor, and informational.
public class AssemblyVersionShortcode
{
    /// Produces the replacement text for one invocation. invocation carries the parsed
arguments and inline content (null for self-closing tags); context carries the host page's
route and metadata.
    public Task<string> ExecuteAsync(ShortcodeInvocation invocation, ShortcodeContext
context, CancellationToken cancellationToken)
;

    /// Case-insensitive name used to dispatch <?# Name ... ?> invocations.
    public string Name { get; }

}
```

Pennington.Markdown.Shortcodes.IShortcode

Pennington.Markdown.Shortcodes.IShortcode

CSHARP

```
namespace Pennington.Markdown.Shortcodes;

/// Handler for a named shortcode invocation expanded before Markdig parsing.
Implementations are registered as DI services and dispatched by Name (case-insensitive). The
string returned by ExecuteAsync is spliced into the markdown source and then parsed as
markdown — return raw HTML when the output should bypass markdown processing (use HTML block
syntax so Markdig leaves it intact).
public interface IShortcode
{
    /// Produces the replacement text for one invocation. invocation carries the parsed
arguments and inline content (null for self-closing tags); context carries the host page's
route and metadata.
    public Task<string> ExecuteAsync(ShortcodeInvocation invocation, ShortcodeContext
context, CancellationToken cancellationToken)
;

    /// Case-insensitive name used to dispatch <?# Name ... ?> invocations.
    public string Name { get; }

}
```

Pennington.Markdown.Shortcodes.PackageVersionShortcode

Pennington.Markdown.Shortcodes.PackageVersionShortcode

CSHARP

```
namespace Pennington.Markdown.Shortcodes;

/// Built-in shortcode that emits Pennington's published package version, dispatched by the name PackageVersion. Resolves the Pennington core assembly's AssemblyInformationalVersionAttribute and trims MinVer's +<sha> build metadata, so it matches the NuGet version a reader installs. Unlike the Version shortcode (which reads the host's entry assembly), this always reports Pennington itself — the value to stamp into install snippets. Takes no arguments.
public class PackageVersionShortcode
{
    /// Produces the replacement text for one invocation. invocation carries the parsed arguments and inline content (null for self-closing tags); context carries the host page's route and metadata.
    public Task<string> ExecuteAsync(ShortcodeInvocation invocation, ShortcodeContext context, CancellationToken cancellationToken)
;

    /// Case-insensitive name used to dispatch <?# Name ... ?> invocations.
    public string Name { get; }

}
```

Pennington.Markdown.Shortcodes.ShortcodeContext

Pennington.Markdown.Shortcodes.ShortcodeContext

CSHARP

```
namespace Pennington.Markdown.Shortcodes;

/// Per-invocation context describing the page that hosts the shortcode call site.
public record ShortcodeContext
{
    /// Front matter of the page being rendered.
    public IFrontMatter Metadata { get; set; }

    /// Route of the page being rendered.
    public ContentRoute Route { get; set; }

    /// Per-invocation context describing the page that hosts the shortcode call site.
    public ShortcodeContext(ContentRoute Route, IFrontMatter Metadata)
;
}
```

Pennington.Markdown.Shortcodes.ShortcodeExpander

Pennington.Markdown.Shortcodes.ShortcodeExpander

CSHARP

```
namespace Pennington.Markdown.Shortcodes;

/// Expands Statiq-style pre-render shortcodes in a markdown source. Recognises <?# Name args ?>...<?#/ Name ?> blocks and <?# Name args /?> self-closing tags, dispatches each call to a matching IShortcode registered in DI, and splices the handler's output back into the source before Markdig parses it. Directives expand everywhere — including inside fenced code blocks — so install snippets and generated samples can stamp real values. To show a literal directive without expanding it, prefix the opener with a backslash (\<?# ... ?>); the expander consumes the backslash and emits the directive as-is. Unknown names and handler failures degrade to HTML comments and a diagnostic so one bad call site cannot fail the render.
public class ShortcodeExpander
{
    /// Returns markdown with every shortcode call site replaced by its handler output. Returns the input unchanged when no handlers are registered or the source contains no shortcode opener.
    public Task<string> ExpandAsync(string markdown, ShortcodeContext context, CancellationToken cancellationToken)
;

    /// Creates the expander with the DI-registered handlers and an optional accessor for per-request DiagnosticContext.
    public ShortcodeExpander(IEnumerable<IShortcode> shortcodes = null, IHttpContextAccessor httpContextAccessor = null)
;
}
```

Pennington.Markdown.Shortcodes.ShortcodeInvocation

Pennington.Markdown.Shortcodes.ShortcodeInvocation

CSHARP

```
namespace Pennington.Markdown.Shortcodes;

/// One parsed shortcode call site, supplied to ExecuteAsync.
public record ShortcodeInvocation
{
    /// Inline content between opener and closer; null for self-closing tags.
    public string Content { get; set; }

    /// Named (key=value) arguments; keys are case-insensitive.
    public IReadOnlyDictionary<string, string> NamedArgs { get; set; }

    /// Positional arguments in source order; empty when none were supplied.
    public IReadOnlyList<string> PositionalArgs { get; set; }

    /// One parsed shortcode call site, supplied to ExecuteAsync.
    public ShortcodeInvocation(IReadOnlyList<string> PositionalArgs,
IReadOnlyDictionary<string, string> NamedArgs, string Content)
;
}
```

Pennington.MonorailCss.AlgorithmicColorScheme

Pennington.MonorailCss.AlgorithmicColorScheme

CSHARP

```
namespace Pennington.MonorailCss;

/// A color scheme that generates palettes algorithmically from a seed hue and chroma. The
scheme synthesises primary, base, and one or more accent palettes; see
ApplyAlgorithmicColorScheme for the underlying mechanics.
public class AlgorithmicColorScheme
{
    /// Gets or sets additional color mappings beyond the core slots. Key is the target slot
name (e.g., "info", "warning"), value is the source color.
    public Dictionary<string, ColorName> AdditionalMappings { get; set; }

    /// Applies the color scheme to the given theme.
    public Theme ApplyToTheme(Theme theme)
;

    /// Gets or sets the seed chroma for the primary palette. Typical range is 0.05 (muted)
to 0.30 (vivid); the 500 stop of the generated primary lands on this value.
    public double Chroma { get; set; }

    /// Gets or sets the primary hue value in degrees (0-360).
    public double PrimaryHue { get; set; }

    /// Gets or sets the coordinating scheme that picks accent hues relative to PrimaryHue.
Defaults to Complementary.
    public CoordinatingScheme Scheme { get; set; }

}
```

Pennington.MonorailCss.ColorName

Pennington.MonorailCss.ColorName

CSHARP

```
namespace Pennington.MonorailCss;

/// A color reference that provides IntelliSense discoverability for known Tailwind colors
while still accepting arbitrary custom color names via implicit string conversion.
public struct ColorName
{
    /// Amber
    public static ColorName Amber { get; }

    /// Black
    public static ColorName Black { get; }

    /// Blue
    public static ColorName Blue { get; }

    /// A color reference that provides IntelliSense discoverability for known Tailwind
colors while still accepting arbitrary custom color names via implicit string conversion.
    public ColorName(string Value)
;

    /// Cyan
    public static ColorName Cyan { get; }

    /// Emerald
    public static ColorName Emerald { get; }

    /// Fuchsia
    public static ColorName Fuchsia { get; }

    /// Gray
    public static ColorName Gray { get; }

    /// Green
    public static ColorName Green { get; }

    /// Indigo
    public static ColorName Indigo { get; }

    /// Lime
    public static ColorName Lime { get; }

    /// Mauve
    public static ColorName Mauve { get; }

    /// Mist
    public static ColorName Mist { get; }

    /// Neutral
    public static ColorName Neutral { get; }

    /// Olive
```

```
    public static ColorName Orange { get; }      /// Pink    public static ColorName Pink { get; }      /// Purple    public static ColorName Purple { get; }      /// Red    public static ColorName Red { get; }      /// Rose    public static ColorName Rose { get; }      /// Sky    public static ColorName Sky { get; }      /// Slate    public static ColorName Slate { get; }      /// Stone    public static ColorName Stone { get; }      /// Taupe    public static ColorName Taupe { get; }      /// Teal    public static ColorName Teal { get; }      public string ToString();      /// Underlying color name (Tailwind palette key or custom identifier).    public string Value { get; set; }      /// Violet    public static ColorName Violet { get; }      /// White    public static ColorName White { get; }      /// Yellow    public static ColorName Yellow { get; }      /// Zinc    public static ColorName Zinc { get; }  }
```

Pennington.MonorailCss.ColorPaletteGenerator

Pennington.MonorailCss.ColorPaletteGenerator

CSHARP

```
namespace Pennington.MonorailCss;

/// Generates Tailwind-style 11-step OKLCH palettes from a seed hue and chroma, plus the coordinating accent and neutral base palettes that go with them.
public class ColorPaletteGenerator
{
    /// Wires algorithmic palettes (base + primary + accents) onto the theme. Primary is a foreground palette at hue/chroma; base is a desaturated neutral palette at the same hue; accents are foreground palettes at hues picked by scheme. Schemes with multiple accents register the first as accent and additional ones as accent-2, accent-3, ….
    public static Theme ApplyAlgorithmicColorScheme(Theme theme, double hue, double chroma, CoordinatingScheme scheme)
;

    /// Generates a vibrant foreground 11-step OKLCH palette. The 500 stop lands on chroma; other stops scale relative to it via the foreground chroma curve.
    public static ImmutableDictionary<string, string> GenerateForeground(double hue, double chroma)
;

    /// Generates a near-gray neutral 11-step OKLCH palette. Chroma scales by intensity (chroma / curve at 500) and the dark tail blends between tapered and held-high shapes via DarkChromaRetention.
    public static ImmutableDictionary<string, string> GenerateNeutral(double hue, double chroma)
;
}
```

Pennington.MonorailCss.ColorTheme

Pennington.MonorailCss.ColorTheme

CSHARP

```
namespace Pennington.MonorailCss;

/// A curated, named color theme: one seed hue grows the algorithmic primary and accent
brand palettes plus a coordinating OKLCH syntax-highlight palette, while base is the stock
MonorailCss neutral whose undertone sits nearest the hue (see NeutralForHue). Assign
ColorScheme to the theme and SyntaxTheme to its SyntaxTheme.
public record ColorTheme
{
    /// The full curated catalog, in hue-wheel order.
    public static IReadOnlyList<ColorTheme> All { get; }

    /// Applies the color scheme to the given theme.
    public Theme ApplyToTheme(Theme theme)
;

    /// Bright cyan-blue with an orange complement.
    public static ColorTheme Aqua { get; }

    /// Classic azure blue with a warm complement.
    public static ColorTheme Azure { get; }

    /// Optional override for the base neutral. Leave it null (the default) to auto-pick the
MonorailCss neutral whose undertone is nearest PrimaryHue via NeutralForHue. Set it to a
specific family (e.g. Zinc, or Neutral for crisp untinted grays) to force that one instead.
Comments — which track base — follow this choice too.
    public ColorName? BaseColorName { get; set; }

    /// Seed chroma for the brand foreground palettes. Typical range 0.05 (muted) to 0.18
(vivid); the 500 stop of the generated primary lands on this value.
    public double Chroma { get; set; }

    /// Yellow-green citrus, triadic for a punchy three-color set.
    public static ColorTheme Citron { get; }

    /// How coordinating accent hues are picked relative to PrimaryHue.
    public CoordinatingScheme Coordinating { get; set; }

    /// Warm red-orange with cyan rhythm. Split-complementary.
    public static ColorTheme Ember { get; }

    /// Leafy green with neighbouring analogous accents.
    public static ColorTheme Fern { get; }

    /// Restrained blue-gray for a quiet, professional brand; syntax stays legible via the
chroma floor.
    public static ColorTheme Graphite { get; }

    /// Deep indigo, triadic across the wheel.
    public static ColorTheme Indigo { get; }

    /// Violet iris, split-complementary toward yellow-green.
```

```
    public static ColorTheme Lagoon { get; }      /// Golden amber with a blue complement.
public static ColorTheme Marigold { get; }      /// Display name of the theme.    public
string Name { get; set; }      /// Picks the MonorailCss neutral palette whose undertone hue
sits nearest hue (measured around the color wheel), giving base grays that coordinate with
the brand. The pure, hueless Neutral is never auto-selected — request it explicitly via
BaseColorName when you want untinted grays.    public static ColorName NeutralForHue(double
hue);      /// Magenta orchid with a green complement.    public static ColorTheme Orchid {
get; }      /// Primary hue in degrees (0–360); the seed every palette is grown from.
public double PrimaryHue { get; set; }      /// Pink-red rose with analogous warmth.
public static ColorTheme Rose { get; }      /// The syntax-highlight theme paired with this
color theme. Keyword/string/variable/function map onto the generated syntax-* accent
palettes (registered by ApplyToTheme); comments track the neutral base palette.    public
SyntaxTheme SyntaxTheme { get; }  }
```

Pennington.MonorailCss.CoordinatingScheme

Pennington.MonorailCss.CoordinatingScheme

CSHARP

```
namespace Pennington.MonorailCss;

/// Selects accent hues relative to the primary hue.
public enum CoordinatingScheme
{
    /// Two accents at -30° and +30°.
    public static const CoordinatingScheme Analogous
;

    /// Single accent at +180°.
    public static const CoordinatingScheme Complementary
;

    /// Two accents at +150° and +210°.
    public static const CoordinatingScheme SplitComplementary
;

    /// Two accents at +120° and +240°.
    public static const CoordinatingScheme Triadic
;
}
```

Pennington.MonorailCss.IColorScheme

Pennington.MonorailCss.IColorScheme

CSHARP

```
namespace Pennington.MonorailCss;

/// Defines how color schemes are applied to the MonorailCSS theme.
public interface IColorScheme
{
    /// Applies the color scheme to the given theme.
    public Theme ApplyToTheme(Theme theme)
;
}
```

Pennington.MonorailCss.MonorailCssOptions

Pennington.MonorailCss.MonorailCssOptions

CSHARP

```
namespace Pennington.MonorailCss;

/// Options for configuring the Monorail CSS framework integration.
public class MonorailCssOptions
{
    /// Gets or sets the color scheme for the site. The default is a NamedColorScheme with
Blue (primary), Purple (accent), and Slate (base).
    public IColorScheme ColorScheme { get; set; }

    /// Gets or sets a function to customize the CSS framework settings before construction.
The callback receives a fully-baked CssFrameworkSettings with Pennington's defaults already
applied (theme, applies, scrollbar utilities, prose rules) and returns the settings the
framework is built from.
    public Func<CssFrameworkSettings, CssFrameworkSettings> CustomCssFrameworkSettings {
get; set; }

    /// Wraps the baseline ProseCustomization Pennington registers, letting consumers add or
override prose rules without rebuilding the customization from scratch. The callback
receives the framework's prose customization and returns the one used.
    public Func<ProseCustomization, ProseCustomization> ExtendProseCustomization { get; set;
}

    /// Gets or sets any extra CSS styles to be included in the generated stylesheet.
    public string ExtraStyles { get; set; }

    /// Gets or sets the syntax-highlight color theme. Controls the Tailwind palettes used
by .hljs-* token classes, independent of the site's brand ColorScheme.
    public SyntaxTheme SyntaxTheme { get; set; }

}
```

Pennington.MonorailCss.MonorailCssService

Pennington.MonorailCss.MonorailCssService

CSHARP

```
namespace Pennington.MonorailCss;

/// Wraps the discovery-pipeline output with Pennington's stylesheet prefix
(ContentVisibilityRules + ExtraStyles) and serves the result on every /styles.css hit.
public class MonorailCssService
{
    /// Builds a Tailwind-aware class merge delegate from Pennington's options:
(baseClasses, overrideClasses) => merged, dropping utilities in options-rendered baseClasses
that conflict with later overrideClasses. Backed by Merge over the same framework the site
renders with, so the semantic palette and custom utilities define the conflicts. Consumed by
Pennington.UI's ClassMerge so a component's per-instance *Class parameter merges over its
variant base through the same framework the site renders with.
    public static Func<string, string, string> CreateClassMerger(MonorailCssOptions options)
;

    /// Builds the stylesheet by running the current class set from GetClasses through a
cached CssFramework. This deliberately skips Css (which caches against the framework that
was baked into MonorailDiscoveryOptions.Framework at startup).
    public string GetStyleSheet()
;

    /// Wraps the discovery-pipeline output with Pennington's stylesheet prefix
(ContentVisibilityRules + ExtraStyles) and serves the result on every /styles.css hit.
    public MonorailCssService(MonorailCssOptions options, IClassRegistry classRegistry,
ILogger<MonorailCssService> logger)
;
}
```

Pennington.MonorailCss.MonorailServiceExtensions

Pennington.MonorailCss.MonorailServiceExtensions

CSHARP

```
namespace Pennington.MonorailCss;

/// Extension methods for registering and configuring MonorailCSS services.
public class MonorailServiceExtensions
{
    /// Registers MonorailCSS services and the runtime class-discovery pipeline. With no configuration, the discovery pipeline force-loads every non-BCL assembly the app references, scans each one's IL, watches the project's source files in development, and loads wwwroot/app.css as the source CSS prefix when present. The CSS endpoint served by UseMonorailCss regenerates whenever the class set changes.
    public static IServiceCollection AddMonorailCss(IServiceCollection services, Func<IServiceProvider, MonorailCssOptions> optionFactory = null)
;

    /// Maps the MonorailCSS stylesheet endpoint. The endpoint pulls the current class set from the discovery pipeline registered in AddMonorailCss, generates CSS, and serves it.
    public static WebApplication UseMonorailCss(WebApplication app, string path = "/styles.css")
;
}
```

Pennington.MonorailCss.NamedColorScheme

Pennington.MonorailCss.NamedColorScheme

CSHARP

```
namespace Pennington.MonorailCss;

/// A color scheme that uses named Tailwind colors.
public class NamedColorScheme
{
    /// Gets or sets the color name to map to "accent".
    public ColorName AccentColorName { get; set; }

    /// Gets or sets additional color mappings beyond the core slots. Key is the target slot name (e.g., "info", "warning"), value is the source color.
    public Dictionary<string, ColorName> AdditionalMappings { get; set; }

    /// Applies the color scheme to the given theme.
    public Theme ApplyToTheme(Theme theme)
;

    /// Gets or sets the color name to map to "base".
    public ColorName BaseColorName { get; set; }

    /// Gets or sets the color name to map to "primary".
    public ColorName PrimaryColorName { get; set; }

}
```

Pennington.MonorailCss.SyntaxTheme

Pennington.MonorailCss.SyntaxTheme

CSHARP

```csharp
namespace Pennington.MonorailCss;

/// Color palette used by .hljs-* syntax-highlight token classes. Each slot is a Tailwind color name whose shades (300-800) are consumed by light/dark theme rules.
public record SyntaxTheme
{
    /// Comments and quotes. Usually the site's base color.
    public ColorName Comment { get; set; }

    /// Default palette: Sky keywords, Emerald strings, Rose variables, Amber functions, Slate comments.
    public static SyntaxTheme Default { get; }

    /// Function/method titles, parameters, built-ins.
    public ColorName Function { get; set; }

    /// Keywords, class names, literals, selector tags.
    public ColorName Keyword { get; set; }

    /// String literals, numbers, regular expressions.
    public ColorName String { get; set; }

    /// Variables, attribute names, symbols.
    public ColorName Variable { get; set; }

}
```

Pennington.Navigation.BreadcrumbItem

Pennington.Navigation.BreadcrumbItem

CSHARP

```
namespace Pennington.Navigation;

/// One segment in a page's breadcrumb trail.
public record BreadcrumbItem
{
    /// One segment in a page's breadcrumb trail.
    public BreadcrumbItem(string Title, ContentRoute Route)
;

    /// Route the segment links to, or null for a non-linked label.
    public ContentRoute Route { get; set; }

    /// Display title for the segment.
    public string Title { get; set; }

}
```

Pennington.Navigation.DownloadLink

Pennington.Navigation.DownloadLink

CSHARP

```
namespace Pennington.Navigation;

/// A downloadable artifact a site advertises in its chrome (for example a sidebar "Download
as PDF" link).
public record DownloadLink
{
    /// A downloadable artifact a site advertises in its chrome (for example a sidebar
"Download as PDF" link).
    public DownloadLink(string Label, string Url, string RoutePrefix)
;

    /// Display-ready link text, already localized by the provider.
    public string Label { get; set; }

    /// Canonical route prefix the artifact covers (for example /tutorials/); / for a whole-
site artifact.
    public string RoutePrefix { get; set; }

    /// Site-relative URL of the artifact (for example pdf/tutorials.pdf).
    public string Url { get; set; }

}
```

Pennington.Navigation.IDownloadLinkProvider

Pennington.Navigation.IDownloadLinkProvider

CSHARP

```
namespace Pennington.Navigation;

/// DI-discovered provider of download links a host's chrome can advertise (the DocSite
sidebar renders every registered provider's links beneath the matching area's table of
contents). Implementations must be cheap to resolve on the request path — derive purely from
configured options, never from the site projection.
public interface IDownloadLinkProvider
{
    /// Returns the links available for locale (or the default locale when null), each with
a locale-appropriate URL.
    public IReadOnlyList<DownloadLink> GetLinks(string locale = null)
;
}
```

Pennington.Navigation.NavigationBuilder

Pennington.Navigation.NavigationBuilder

CSHARP

```
namespace Pennington.Navigation;

/// Builds hierarchical navigation trees and related navigation metadata from flat TOC entries.
public class NavigationBuilder
{
    /// Build NavigationInfo for the page at currentPath within the tree. See BuildTreeAsync for snapshot-resolution semantics.
    public Task<NavigationInfo> BuildNavigationInfoAsync(IReadOnlyList<ContentTocItem> items, UrlPath currentPath, string locale = null)
;

    /// Build a navigation tree from flat TOC items. Resolves the folder-metadata snapshot once via GetSnapshotAsync when configured; recursion stays sync once the snapshot is in hand. When locale is specified, filters to that locale and strips the locale prefix from hierarchy parts. When currentPath is specified, stamps IsSelected/IsExpanded along the path to that page.
    public Task<ImmutableList<NavigationTreeItem>> BuildTreeAsync(IReadOnlyList<ContentTocItem> items, UrlPath? currentPath = default, string locale = null)
;

    /// Creates a navigation builder that consults folderMetadata for folder-level title and order overrides discovered from _meta.yml sidecars.
    public NavigationBuilder(FolderMetadataRegistry folderMetadata)
;

    /// Called on the file-watcher thread for every watched change. Must be quick and thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;
}
```

Pennington.Navigation.NavigationInfo

Pennington.Navigation.NavigationInfo

CSHARP

```
namespace Pennington.Navigation;

/// Page-scoped navigation context exposed to layouts and components.
public record NavigationInfo
{
    /// Breadcrumb trail from the site root to the current page.
    public ImmutableList<BreadcrumbItem> Breadcrumbs { get; set; }

    /// Page-scoped navigation context exposed to layouts and components.
    public NavigationInfo(string SectionName, ImmutableList<BreadcrumbItem> Breadcrumbs,
string PageTitle, NavigationTreeItem PreviousPage, NavigationTreeItem NextPage)
;

    /// Next page in reading order, or null at the end.
    public NavigationTreeItem NextPage { get; set; }

    /// Title of the current page.
    public string PageTitle { get; set; }

    /// Previous page in reading order, or null at the start.
    public NavigationTreeItem PreviousPage { get; set; }

    /// Label of the containing top-level section, or null if none.
    public string SectionName { get; set; }

}
```

Pennington.Navigation.NavigationTreeItem

Pennington.Navigation.NavigationTreeItem

CSHARP

```
namespace Pennington.Navigation;

/// A node in the hierarchical navigation tree.
public record NavigationTreeItem
{
    /// Child nodes nested under this one.
    public ImmutableList<NavigationTreeItem> Children { get; set; }

    /// True when the node should render expanded (contains or is the current route).
    public bool IsExpanded { get; set; }

    /// True when the node matches the current route.
    public bool IsSelected { get; set; }

    /// A node in the hierarchical navigation tree.
    public NavigationTreeItem(string Title, ContentRoute Route, int Order, string
SectionLabel, bool IsSelected, bool IsExpanded, ImmutableList<NavigationTreeItem> Children)
;

    /// Sort order within its parent.
    public int Order { get; set; }

    /// Route the node links to.
    public ContentRoute Route { get; set; }

    /// Optional section grouping label.
    public string SectionLabel { get; set; }

    /// Display title for the node.
    public string Title { get; set; }

}
```

Pennington.Pipeline.ContentError

Pennington.Pipeline.ContentError

CSHARP

```
namespace Pennington.Pipeline;

/// Describes a failure encountered while processing a content item.
public record ContentError
{
    /// Describes a failure encountered while processing a content item.
    public ContentError(string Message, Exception Exception = null)
;

    /// Underlying exception, if any.
    public Exception Exception { get; set; }

    /// Human-readable error message.
    public string Message { get; set; }

}
```

Pennington.Pipeline.ContentFormatRegistry

Pennington.Pipeline.ContentFormatRegistry

CSHARP

```
namespace Pennington.Pipeline;

/// Maps content-format keys (e.g. "markdown", "cook") to the factories that build the
format's IContentParser and IContentRenderer. Populated once while AddPennington wires
services, then read by DispatchingContentParser and DispatchingContentRenderer at request
time. Format keys are matched case-insensitively.
public class ContentFormatRegistry
{
    /// Registers (overwriting) the parser and renderer factories for a format key.
    public void Register(string format, Func<IServiceProvider, IContentParser> parser,
Func<IServiceProvider, IContentRenderer> renderer)
;

    /// Gets the parser factory registered for format, if any.
    public bool TryGetParser(string format, out Func<IServiceProvider, IContentParser>
factory)
;

    /// Gets the renderer factory registered for format, if any.
    public bool TryGetRenderer(string format, out Func<IServiceProvider, IContentRenderer>
factory)
;
}
```

Pennington.Pipeline.ContentItem

Pennington.Pipeline.ContentItem

CSHARP

```
namespace Pennington.Pipeline;

/// Union of all content item states flowing through the pipeline.
public struct ContentItem
{
    /// Wraps a DiscoveredItem.
    public ContentItem(DiscoveredItem value)
;

    /// Wraps a ParsedItem.
    public ContentItem(ParsedItem value)
;

    /// Wraps a RenderedItem.
    public ContentItem(RenderedItem value)
;

    /// Wraps a FailedItem.
    public ContentItem(FailedItem value)
;

    /// A content item discovered by a content service but not yet run through the pipeline.
    public record DiscoveredItem(ContentRoute Route, ContentSource Source) : object,
IEquatable<DiscoveredItem>

    /// A content item that failed during parsing or rendering.
    public record FailedItem(ContentRoute Route, ContentError Error) : object,
IEquatable<FailedItem>

    /// A content item whose front matter and raw markdown body have been parsed.
    public record ParsedItem(ContentRoute Route, IFrontMatter Metadata, string RawMarkdown)
: object, IEquatable<ParsedItem>

    /// A content item whose body has been rendered to HTML.
    public record RenderedItem(ContentRoute Route, IFrontMatter Metadata, RenderedContent
Content) : object, IEquatable<RenderedItem>

    /// The route for the current item regardless of state.
    public ContentRoute Route { get; }

    /// Wrapped case instance; inspect via pattern matching on the case types.
    public object Value { get; }

}
```

Pennington.Pipeline.ContentPipeline

Pennington.Pipeline.ContentPipeline

CSHARP

```
namespace Pennington.Pipeline;

/// Orchestrates the content processing pipeline.
public class ContentPipeline
{
    /// Creates the pipeline from the registered content services and renderer. The parser is optional: a bare host that registers no markdown source has no IContentParser, so discovered items pass through the parse stage unchanged.
    public ContentPipeline(IEnumerable<IContentService> services, IContentRenderer renderer, IContentParser parser = null, TimeProvider clock = null)
;

    /// Entry: content services produce discovered items.
    public IAsyncEnumerable<ContentItem> DiscoverAsync()
;

    /// Exit: generate output files.
    public Task<BuildReport> GenerateAsync(IAsyncEnumerable<ContentItem> items)
;

    /// Transform: parse items (read files, extract YAML + body).
    public IAsyncEnumerable<ContentItem> ParseAsync(IAsyncEnumerable<ContentItem> items)
;

    /// Transform: render items (Markdig pipeline to HTML).
    public IAsyncEnumerable<ContentItem> RenderAsync(IAsyncEnumerable<ContentItem> items)
;

    /// Convenience: run the full pipeline end-to-end.
    public Task<BuildReport> RunAsync()
;
}
```

Pennington.Pipeline.ContentRendererServiceExtensions

Pennington.Pipeline.ContentRendererServiceExtensions

CSHARP

```
namespace Pennington.Pipeline;

/// DI helpers for swapping the registered IContentRenderer.
public class ContentRendererServiceExtensions
{
    /// Replaces every registered IContentRenderer with TNew, resolved through DI as a transient. The TOld type parameter documents the renderer being swapped out — it is informational and unused at runtime, but lets the call site read as "replace TOld with TNew".
    public static IServiceCollection ReplaceContentRenderer<TOld, TNew>(IServiceCollection services)
;

    /// Replaces every registered IContentRenderer with one produced by factory. Use this overload when the new renderer takes ctor arguments DI cannot resolve (e.g. a version string or per-site constant).
    public static IServiceCollection ReplaceContentRenderer<TOld, TNew>(IServiceCollection services, Func<IServiceProvider, TNew> factory)
;
}
```

Pennington.Pipeline.ContentSource

Pennington.Pipeline.ContentSource

CSHARP

```
namespace Pennington.Pipeline;

/// Union of all ways content can be sourced for a route.
public struct ContentSource
{
    /// Wraps a FileSource.
    public ContentSource(FileSource value)
;

    /// Wraps a RazorPageSource.
    public ContentSource(RazorPageSource value)
;

    /// Wraps a RedirectSource.
    public ContentSource(RedirectSource value)
;

    /// Wraps an EndpointSource.
    public ContentSource(EndpointSource value)
;

    /// Wraps an LlmsOnlySource.
    public ContentSource(LlmsOnlySource value)
;

    /// Wraps a GeneratedSource.
    public ContentSource(GeneratedSource value)
;

    /// Marker source for routes whose content is produced by a live HTTP endpoint (e.g., a
custom IContentService page or an AddTaxonomy term page). These items exist so the build
crawler discovers the URL and fetches it through the live pipeline — they do not participate
in parse/render and are not redirects. Because the endpoint serves real canonical HTML at a
stable URL, the route is listed in the sitemap; transport endpoints that emit a non-HTML
file are dropped earlier by SitemapService's output-extension check.
    public record EndpointSource : object, IEquatable<EndpointSource>

    /// Content sourced from a file on disk, tagged with a format key that selects its
parser/renderer.
    public record FileSource(FilePath Path, string Format) : object, IEquatable<FileSource>

    /// A route whose bytes are produced by an IArtifactContentService resolver rather than
parsed from a file or rendered by a component — search shards, llms.txt files, book PDFs.
Yielded only from the artifact tier's discovery: the static build writes each item via the
owning service's resolver, and the artifact router serves the same bytes in dev. Never
crawled, never a sitemap or record candidate.
    public record GeneratedSource(string ContentType) : object, IEquatable<GeneratedSource>

    /// A markdown file that contributes only to the llms.txt index and its sidecar markdown
— no HTML page is emitted, and the route is excluded from navigation, sitemap, RSS, and the
search index. Conventionally produced by MarkdownContentService when it sees a *.llms.md
```

```
IEquatable<LlmsOnlySource>      /// Content rendered by a Razor page/component.    public record RazorPageSource(string ComponentType) : object, IEquatable<RazorPageSource>      /// A route that redirects to another URL.    public record RedirectSource(UrlPath TargetUrl) : object, IEquatable<RedirectSource>      /// Wrapped case instance; inspect via pattern matching on the case types.    public object Value { get; }  }
```

Pennington.Pipeline.CrossReference

Pennington.Pipeline.CrossReference

CSHARP

```
namespace Pennington.Pipeline;

/// A resolvable cross-reference target identified by a unique id.
public record CrossReference
{
    /// A resolvable cross-reference target identified by a unique id.
    public CrossReference(string Uid, string Title, ContentRoute Route)
;

    /// Route the reference resolves to.
    public ContentRoute Route { get; set; }

    /// Display title for the target.
    public string Title { get; set; }

    /// Unique identifier used in xref: links.
    public string Uid { get; set; }

}
```

Pennington.Pipeline.DiscoveredItem

Pennington.Pipeline.DiscoveredItem

CSHARP

```
namespace Pennington.Pipeline;

/// A content item discovered by a content service but not yet run through the pipeline.
public record DiscoveredItem
{
    /// A content item discovered by a content service but not yet run through the pipeline.
    public DiscoveredItem(ContentRoute Route, ContentSource Source)
;

    /// Front matter the discovering service already parsed, when its source carried any;
null for sources whose metadata is not known until parse or render time.
    public IFrontMatter Metadata { get; set; }

    /// Raw markdown body the discovering service already parsed and cached, when its source
carried any; null for sources whose body is not read until parse time. When both this and
Metadata are present, parsers serve them directly instead of re-reading the file. The
discovering service keeps this cache fresh by re-reading a changed file on a trailing-edge
settle, so it reflects the finished file rather than one the writer still holds.
    public string RawBody { get; set; }

    /// Canonical route for the item.
    public ContentRoute Route { get; set; }

    /// Origin describing how the item's content is produced.
    public ContentSource Source { get; set; }

}
```

Pennington.Pipeline.DispatchingContentParser

Pennington.Pipeline.DispatchingContentParser

CSHARP

```
namespace Pennington.Pipeline;

/// The single IContentParser the pipeline resolves: routes each discovered item to the
parser registered for its format (see ContentFormatRegistry) and stamps the resolved format
onto the returned ParsedItem so DispatchingContentRenderer can dispatch on it.
public class DispatchingContentParser
{
    /// Creates the dispatcher from the format registry and the service provider it resolves
parsers from.
    public DispatchingContentParser(ContentFormatRegistry registry, IServiceProvider
services)
;

    /// Parse a discovered item. Returns ParsedItem on success, FailedItem on failure.
    public Task<ContentItem> ParseAsync(DiscoveredItem item)
;
}
```

Pennington.Pipeline.DispatchingContentRenderer

Pennington.Pipeline.DispatchingContentRenderer

CSHARP

```
namespace Pennington.Pipeline;

/// The single IContentRenderer the pipeline resolves: routes each parsed item to the
renderer registered for its Format (see ContentFormatRegistry).
public class DispatchingContentRenderer
{
    /// Creates the dispatcher from the format registry and the service provider it resolves
renderers from.
    public DispatchingContentRenderer(ContentFormatRegistry registry, IServiceProvider
services)
;

    /// Render a parsed item. Returns RenderedItem on success, FailedItem on failure.
    public Task<ContentItem> RenderAsync(ParsedItem item)
;
}
```

Pennington.Pipeline.EndpointOrigin

Pennington.Pipeline.EndpointOrigin

CSHARP

```
namespace Pennington.Pipeline;

/// Endpoint opted into llms.txt via WithLlmsTxtEntry; the direct URL is the link target and
no HTML is captured.
public record EndpointOrigin
{
    /// URL of the endpoint the llms.txt index should point at.
    public string DirectUrl { get; set; }

    /// Endpoint opted into llms.txt via WithLlmsTxtEntry; the direct URL is the link target
and no HTML is captured.
    public EndpointOrigin(string DirectUrl)
;
}
```

Pennington.Pipeline.EndpointSource

Pennington.Pipeline.EndpointSource

CSHARP

```
namespace Pennington.Pipeline;

/// Marker source for routes whose content is produced by a live HTTP endpoint (e.g., a
custom IContentService page or an AddTaxonomy term page). These items exist so the build
crawler discovers the URL and fetches it through the live pipeline — they do not participate
in parse/render and are not redirects. Because the endpoint serves real canonical HTML at a
stable URL, the route is listed in the sitemap; transport endpoints that emit a non-HTML
file are dropped earlier by SitemapService's output-extension check.
public record EndpointSource
{
    /// Marker source for routes whose content is produced by a live HTTP endpoint (e.g., a
custom IContentService page or an AddTaxonomy term page). These items exist so the build
crawler discovers the URL and fetches it through the live pipeline — they do not participate
in parse/render and are not redirects. Because the endpoint serves real canonical HTML at a
stable URL, the route is listed in the sitemap; transport endpoints that emit a non-HTML
file are dropped earlier by SitemapService's output-extension check.
    public EndpointSource()
;
}
```

Pennington.Pipeline.FailedItem

Pennington.Pipeline.FailedItem

CSHARP

```
namespace Pennington.Pipeline;

/// A content item that failed during parsing or rendering.
public record FailedItem
{
    /// Error describing the failure.
    public ContentError Error { get; set; }

    /// A content item that failed during parsing or rendering.
    public FailedItem(ContentRoute Route, ContentError Error)
;

    /// Canonical route for the item that failed.
    public ContentRoute Route { get; set; }

}
```

Pennington.Pipeline.FileSource

Pennington.Pipeline.FileSource

CSHARP

```
namespace Pennington.Pipeline;

/// Content sourced from a file on disk, tagged with a format key that selects its
parser/renderer.
public record FileSource
{
    /// Content sourced from a file on disk, tagged with a format key that selects its
parser/renderer.
    public FileSource(FilePath Path, string Format)
;

    /// Format key (e.g. "markdown", "cook") selecting the parser and renderer.
    public string Format { get; set; }

    /// True when Format is one of the built-in markdown dispatch keys (see MarkdownFormat).
    public bool IsMarkdown { get; }

    /// Absolute path to the source file.
    public FilePath Path { get; set; }

}
```

Pennington.Pipeline.GeneratedSource

Pennington.Pipeline.GeneratedSource

CSHARP

```
namespace Pennington.Pipeline;

/// A route whose bytes are produced by an IArtifactContentService resolver rather than
parsed from a file or rendered by a component — search shards, llms.txt files, book PDFs.
Yielded only from the artifact tier's discovery: the static build writes each item via the
owning service's resolver, and the artifact router serves the same bytes in dev. Never
crawled, never a sitemap or record candidate.
public record GeneratedSource
{
    /// MIME type of the generated bytes (informational; the resolver's content type is what
gets served).
    public string ContentType { get; set; }

    /// A route whose bytes are produced by an IArtifactContentService resolver rather than
parsed from a file or rendered by a component — search shards, llms.txt files, book PDFs.
Yielded only from the artifact tier's discovery: the static build writes each item via the
owning service's resolver, and the artifact router serves the same bytes in dev. Never
crawled, never a sitemap or record candidate.
    public GeneratedSource(string ContentType)
;
}
```

Pennington.Pipeline.IContentParser

Pennington.Pipeline.IContentParser

CSHARP

```
namespace Pennington.Pipeline;

/// Parses a discovered item into a parsed item (extracts front matter + markdown body).
public interface IContentParser
{
    /// Parse a discovered item. Returns ParsedItem on success, FailedItem on failure.
    public Task<ContentItem> ParseAsync(DiscoveredItem item)
;
}
```

Pennington.Pipeline.IContentPipeline

Pennington.Pipeline.IContentPipeline

CSHARP

```
namespace Pennington.Pipeline;

/// The content processing pipeline.
public interface IContentPipeline
{
    /// Entry: content services produce discovered items.
    public IAsyncEnumerable<ContentItem> DiscoverAsync()
;

    /// Exit: generate output files.
    public Task<BuildReport> GenerateAsync(IAsyncEnumerable<ContentItem> items)
;

    /// Transform: parse items (read files, extract YAML + body).
    public IAsyncEnumerable<ContentItem> ParseAsync(IAsyncEnumerable<ContentItem> items)
;

    /// Transform: render items (Markdig pipeline to HTML).
    public IAsyncEnumerable<ContentItem> RenderAsync(IAsyncEnumerable<ContentItem> items)
;
}
```

Pennington.Pipeline.IContentRenderer

Pennington.Pipeline.IContentRenderer

CSHARP

```
namespace Pennington.Pipeline;

/// Renders a parsed item into HTML.
public interface IContentRenderer
{
    /// Render a parsed item. Returns RenderedItem on success, FailedItem on failure.
    public Task<ContentItem> RenderAsync(ParsedItem item)
;
}
```

Pennington.Pipeline.IMetadataEnricher

Pennington.Pipeline.IMetadataEnricher

CSHARP

```
namespace Pennington.Pipeline;

/// Contributes derived (non-authored) metadata for a parsed content item — reading time,
git last-modified, GitHub permalinks, and the like. Contributions are merged into Derived by
MetadataEnrichmentService. Register implementations with AddTransient<IMetadataEnricher, T>
().
public interface IMetadataEnricher
{
    /// Returns key/value pairs to merge into the item's derived metadata. Return an empty
dictionary to contribute nothing. Later-registered enrichers override earlier ones on key
collision.
    public Task<IReadOnlyDictionary<string, object>> EnrichAsync(ParsedItem item)
;
}
```

Pennington.Pipeline.ISiteProjection

Pennington.Pipeline.ISiteProjection

CSHARP

```
namespace Pennington.Pipeline;

/// Single shared corpus projection: walks every indexable route once, captures its post-pipeline HTML once, parses the DOM once, and yields a stream of RenderedPage records. Every site-wide aggregator (search index, llms.txt, build-time link audit) folds over this stream instead of independently fanning out across the corpus.  this projection is for build-time and artifact-service consumers only. Materialization triggers parallel HTTP self-fetches that re-enter the request pipeline, so no component that runs during a content page's render or response processing may await it — that is the task-cycle deadlock from commit b719d73. The implementation fails fast instead of hanging: every projection-issued fetch is stamped via Infrastructure.CorpusFetchScope, and consuming the projection from such a request (or from inside its own materialization) throws a descriptive InvalidOperationException. Artifact services may consume it freely — their claimed URLs are disjoint from the page corpus the projection fetches. Request-path link verification stays on Infrastructure.PageLinkVerifier, which consults IContentService.DiscoverAllAsync and artifact claims only.
public interface ISiteProjection
{
    /// Returns the projected page at canonicalPath, or null when no page matches. Triggers full materialization on first call; cheap on subsequent calls.
    public Task<RenderedPage> GetPageAsync(UrlPath canonicalPath, CancellationToken cancellationToken = default)
;

    /// Yields every renderable page in deterministic discovery order. Materializes lazily on first enumeration; subsequent calls replay the cached array until the instance is dropped by file-watch invalidation.
    public IAsyncEnumerable<RenderedPage> GetPagesAsync(CancellationToken cancellationToken = default)
;
}
```

Pennington.Pipeline.LlmsOnlySource

Pennington.Pipeline.LlmsOnlySource

CSHARP

```
namespace Pennington.Pipeline;

/// A markdown file that contributes only to the llms.txt index and its sidecar markdown — no HTML page is emitted, and the route is excluded from navigation, sitemap, RSS, and the search index. Conventionally produced by MarkdownContentService when it sees a *.llms.md file: the discovered route uses the slug with the .llms suffix stripped, and downstream stages key off the source type to skip HTML.
public record LlmsOnlySource
{
    /// Markdown dispatch key (see MarkdownFormat) of the source that produced this file, selecting the parser for its front-matter type.
    public string Format { get; set; }

    /// A markdown file that contributes only to the llms.txt index and its sidecar markdown — no HTML page is emitted, and the route is excluded from navigation, sitemap, RSS, and the search index. Conventionally produced by MarkdownContentService when it sees a *.llms.md file: the discovered route uses the slug with the .llms suffix stripped, and downstream stages key off the source type to skip HTML.
    public LlmsOnlySource(FilePath Path, string Format)
;

    /// Absolute path to the markdown file.
    public FilePath Path { get; set; }

}
```

Pennington.Pipeline.MarkdownFormat

Pennington.Pipeline.MarkdownFormat

CSHARP

```
namespace Pennington.Pipeline;

/// Dispatch-key conventions for the built-in markdown sources. Each source registered via
AddMarkdownContent gets a distinct key (SourceKey) so the pipeline routes it to that
source's front-matter parser instead of a single shared one — they all still render through
the one markdown renderer. Matches recognises any markdown key, for consumers that ask "is
this a markdown file?" rather than which source produced it.
public class MarkdownFormat
{
    /// Key for the first markdown source, and the standalone key for single-source hosts.
    public static const string Key
;

    /// True when format is one of the markdown dispatch keys produced by SourceKey.
    public static bool Matches(string format)
;

    /// Distinct dispatch key for the markdown source at index in registration order (index
0 keeps the bare Key).
    public static string SourceKey(int index)
;
}
```

Pennington.Pipeline.MarkdownOrigin

Pennington.Pipeline.MarkdownOrigin

CSHARP

```
namespace Pennington.Pipeline;

/// Origin information for a RenderedPage: the page came from a markdown source (carrying a
ParsedItem with front matter + derived metadata) or from an endpoint opt-in
(LlmsTxtEndpointExtensions) where the link target is the endpoint URL itself and no HTML is
fetched. Razor / programmatic sources carry no origin metadata; see Origin.
public record MarkdownOrigin
{
    /// Origin information for a RenderedPage: the page came from a markdown source
(carrying a ParsedItem with front matter + derived metadata) or from an endpoint opt-in
(LlmsTxtEndpointExtensions) where the link target is the endpoint URL itself and no HTML is
fetched. Razor / programmatic sources carry no origin metadata; see Origin.
    public MarkdownOrigin(ParsedItem Parsed)
;

    /// Enriched parsed item for the page, including derived metadata.
    public ParsedItem Parsed { get; set; }

}
```

Pennington.Pipeline.MetadataEnrichmentService

Pennington.Pipeline.MetadataEnrichmentService

CSHARP

```csharp
namespace Pennington.Pipeline;

/// Runs the registered IMetadataEnricher pipeline over a ParsedItem, merging every
contribution into Derived. A no-op when no enrichers are registered.
public class MetadataEnrichmentService
{
    /// Returns item with derived metadata from every enricher merged into Derived. Returns
the item unchanged when no enricher is registered or none contributes a value.
    public Task<ParsedItem> EnrichAsync(ParsedItem item)
;

    /// Creates the service over the registered enrichers (registration order).
    public MetadataEnrichmentService(IEnumerable<IMetadataEnricher> enrichers)
;
}
```

Pennington.Pipeline.OutlineEntry

Pennington.Pipeline.OutlineEntry

CSHARP

```csharp
namespace Pennington.Pipeline;

/// A single heading entry in a page outline.
public record OutlineEntry
{
    /// Anchor id for the heading.
    public string Id { get; set; }

    /// Heading level (1–6).
    public int Level { get; set; }

    /// A single heading entry in a page outline.
    public OutlineEntry(string Id, string Text, int Level)
;

    /// Heading text.
    public string Text { get; set; }

}
```

Pennington.Pipeline.PageOrigin

Pennington.Pipeline.PageOrigin

CSHARP

```
namespace Pennington.Pipeline;

/// Where a RenderedPage originated and what metadata is available for it.
public struct PageOrigin
{
    /// Endpoint opted into llms.txt via WithLlmsTxtEntry; the direct URL is the link target and no HTML is captured.
    public record EndpointOrigin(string DirectUrl) : object, IEquatable<EndpointOrigin>

    /// Origin information for a RenderedPage: the page came from a markdown source (carrying a ParsedItem with front matter + derived metadata) or from an endpoint opt-in (LlmsTxtEndpointExtensions) where the link target is the endpoint URL itself and no HTML is fetched. Razor / programmatic sources carry no origin metadata; see Origin.
    public record MarkdownOrigin(ParsedItem Parsed) : object, IEquatable<MarkdownOrigin>

    /// Wraps a MarkdownOrigin.
    public PageOrigin(MarkdownOrigin value)
;

    /// Wraps an EndpointOrigin.
    public PageOrigin(EndpointOrigin value)
;

    /// Wrapped case instance; inspect via pattern matching on the case types.
    public object Value { get; }

}
```

Pennington.Pipeline.ParsedItem

Pennington.Pipeline.ParsedItem

CSHARP

```
namespace Pennington.Pipeline;

/// A content item whose front matter and raw markdown body have been parsed.
public record ParsedItem
{
    /// Derived, non-authored metadata contributed by IMetadataEnricher implementations (reading time, git last-modified, permalinks, …), keyed by enricher-defined names. Kept separate from the strongly-typed Metadata so authored front matter stays the single source of truth.
    public IReadOnlyDictionary<string, object> Derived { get; set; }

    /// Format key selecting the renderer for this item; defaults to "markdown".
    public string Format { get; set; }

    /// Parsed front matter metadata.
    public IFrontMatter Metadata { get; set; }

    /// A content item whose front matter and raw markdown body have been parsed.
    public ParsedItem(ContentRoute Route, IFrontMatter Metadata, string RawMarkdown)
;

    /// Markdown body text, with front matter stripped.
    public string RawMarkdown { get; set; }

    /// Canonical route for the item.
    public ContentRoute Route { get; set; }

}
```

Pennington.Pipeline.RazorContentRenderer

Pennington.Pipeline.RazorContentRenderer

CSHARP

```
namespace Pennington.Pipeline;

/// Base IContentRenderer that renders a Razor component TComponent to HTML. A subclass projects a ParsedItem into the component's parameters via BuildParameters; this base owns the Blazor HtmlRenderer dispatch, heading-anchor assignment, and outline extraction. A structured content format therefore renders through Razor markup the way markdown renders through Markdig — both produce a RenderedContent. The host must register Razor component services (AddRazorComponents()).
public class RazorContentRenderer
{
    /// Render a parsed item. Returns RenderedItem on success, FailedItem on failure.
    public Task<ContentItem> RenderAsync(ParsedItem item)
;
}
```

Pennington.Pipeline.RazorPageSource

Pennington.Pipeline.RazorPageSource

CSHARP

```
namespace Pennington.Pipeline;

/// Content rendered by a Razor page/component.
public record RazorPageSource
{
    /// Fully qualified name of the component type.
    public string ComponentType { get; set; }

    /// Content rendered by a Razor page/component.
    public RazorPageSource(string ComponentType)
;
}
```

Pennington.Pipeline.ReadingTimeEnricher

Pennington.Pipeline.ReadingTimeEnricher

CSHARP

```
namespace Pennington.Pipeline;

/// Estimates reading time from the markdown body and contributes it as
reading_time_minutes. A pure function of RawMarkdown — no file access, no external
dependencies.
public class ReadingTimeEnricher
{
    /// Returns key/value pairs to merge into the item's derived metadata. Return an empty
dictionary to contribute nothing. Later-registered enrichers override earlier ones on key
collision.
    public Task<IReadOnlyDictionary<string, object>> EnrichAsync(ParsedItem item)
;

    /// Key written into Derived.
    public static const string Key
;
}
```

Pennington.Pipeline.RedirectSource

Pennington.Pipeline.RedirectSource

CSHARP

```
namespace Pennington.Pipeline;

/// A route that redirects to another URL.
public record RedirectSource
{
    /// A route that redirects to another URL.
    public RedirectSource(UrlPath TargetUrl)
;

    /// Destination URL for the redirect.
    public UrlPath TargetUrl { get; set; }

}
```

Pennington.Pipeline.RenderedContent

Pennington.Pipeline.RenderedContent

CSHARP

```
namespace Pennington.Pipeline;

/// Output produced by the render stage for a single content item.
public record RenderedContent
{
    /// Cross-reference targets defined by the content.
    public ImmutableList<CrossReference> CrossReferences { get; set; }

    /// Rendered HTML body.
    public string Html { get; set; }

    /// Heading outline extracted from the content.
    public OutlineEntry[] Outline { get; set; }

    /// Output produced by the render stage for a single content item.
    public RenderedContent(string Html, OutlineEntry[] Outline, ImmutableList<Tag> Tags,
ImmutableList<CrossReference> CrossReferences, SocialMetadata Social)
;

    /// Optional social/Open Graph metadata for this page.
    public SocialMetadata Social { get; set; }

    /// Tags associated with the content.
    public ImmutableList<Tag> Tags { get; set; }

}
```

Pennington.Pipeline.RenderedItem

Pennington.Pipeline.RenderedItem

CSHARP

```
namespace Pennington.Pipeline;

/// A content item whose body has been rendered to HTML.
public record RenderedItem
{
    /// Rendered content and extracted page data.
    public RenderedContent Content { get; set; }

    /// Parsed front matter metadata.
    public IFrontMatter Metadata { get; set; }

    /// A content item whose body has been rendered to HTML.
    public RenderedItem(ContentRoute Route, IFrontMatter Metadata, RenderedContent Content)
;

    /// Canonical route for the item.
    public ContentRoute Route { get; set; }

}
```

Pennington.Pipeline.RenderedItem

Pennington.Pipeline.RenderedItem

CSHARP

```
namespace Pennington.Pipeline;

/// A RenderedItem whose front matter has been narrowed to a known type, returned by the
generic ResolveAsync<TFrontMatter> convenience.
public record RenderedItem
{
    /// Rendered content and extracted page data.
    public RenderedContent Content { get; set; }

    /// Front matter typed as TFrontMatter.
    public TFrontMatter Metadata { get; set; }

    /// A RenderedItem whose front matter has been narrowed to a known type, returned by the
generic ResolveAsync<TFrontMatter> convenience.
    public RenderedItem`1(ContentRoute Route, TFrontMatter Metadata, RenderedContent
Content)
;

    /// Canonical route for the item.
    public ContentRoute Route { get; set; }

}
```

Pennington.Pipeline.RenderedPage

Pennington.Pipeline.RenderedPage

CSHARP

```
namespace Pennington.Pipeline;

/// One page in the projected corpus: route + TOC entry + origin metadata, plus the post-pipeline HTML and a parsed AngleSharp element to aggregate over. Produced by ISiteProjection once per route per file-watch generation and shared across every corpus aggregator (search, llms.txt, link audit). Consumers pattern-match on Origin to recover front-matter / derived metadata where it exists. A nullOrigin means a Razor / programmatic page with no parsed item — the HTML is still available, just without front-matter context.Content is owned by a per-page AngleSharp browsing context — the projection holds the context for the page's lifetime. Treat Content as read-only: mutating it corrupts other consumers' views.
public record RenderedPage
{
    /// Parsed content element from the post-pipeline HTML; null for endpoint pages.
    public IElement Content { get; set; }

    /// Post-pipeline HTML of the selector-matched content element; empty for endpoint pages.
    public string Html { get; set; }

    /// Origin information (markdown / endpoint), or null for Razor / programmatic pages.
    public PageOrigin? Origin { get; set; }

    /// One page in the projected corpus: route + TOC entry + origin metadata, plus the post-pipeline HTML and a parsed AngleSharp element to aggregate over. Produced by ISiteProjection once per route per file-watch generation and shared across every corpus aggregator (search, llms.txt, link audit). Consumers pattern-match on Origin to recover front-matter / derived metadata where it exists. A nullOrigin means a Razor / programmatic page with no parsed item — the HTML is still available, just without front-matter context.Content is owned by a per-page AngleSharp browsing context — the projection holds the context for the page's lifetime. Treat Content as read-only: mutating it corrupts other consumers' views.
    public RenderedPage(ContentRoute Route, ContentTocItem Toc, PageOrigin? Origin, string Html, IElement Content, Lazy<IReadOnlyList<HeadingSection>> Sections)
;

    /// Canonical route for the page.
    public ContentRoute Route { get; set; }

    /// Lazy heading-section split of Content; empty for endpoint pages.
    public Lazy<IReadOnlyList<HeadingSection>> Sections { get; set; }

    /// TOC entry that drove the page's inclusion.
    public ContentTocItem Toc { get; set; }

}
```

Pennington.Pipeline.SiteProjection

Pennington.Pipeline.SiteProjection

CSHARP

```
namespace Pennington.Pipeline;

/// Default ISiteProjection. Walks CollectIndexableEntriesAsync plus any WithLlmsTxtEntry
endpoint registrations, fetches post-pipeline HTML for each route in parallel via
RenderedHtmlFetcher, and folds every result into a stable index-keyed array (deterministic
ordering for snapshot tests). Llms-only items have no HTTP route, so the projection renders
them in-process via IContentRenderer + XrefResolvingService instead. Registered as
IFileWatchAware with Refreshed — file-watch invalidation marks just the affected routes
stale (via GetAffectedRoutes) so the next access re-fetches only those, reusing the cached
HTML for everything else. A Wildcard report (rename, folder-metadata edit) falls back to a
full rebuild.
public class SiteProjection
{
    /// Returns the projected page at canonicalPath, or null when no page matches. Triggers
full materialization on first call; cheap on subsequent calls.
    public Task<RenderedPage> GetPageAsync(UrlPath canonicalPath, CancellationToken
cancellationToken = default)
;

    /// Yields every renderable page in deterministic discovery order. Materializes lazily
on first enumeration; subsequent calls replay the cached array until the instance is dropped
by file-watch invalidation.
    public IAsyncEnumerable<RenderedPage> GetPagesAsync(CancellationToken cancellationToken
= default)
;

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;

    /// Creates the projection; the corpus is materialized lazily on first access.
    public SiteProjection(IEnumerable<IContentService> contentServices,
MetadataEnrichmentService enrichment, IContentRenderer renderer, XrefResolvingService
xrefResolver, RenderedHtmlFetcher fetcher, HeadingSectionExtractor extractor,
SiteProjectionOptions options, EndpointDataSource endpointDataSource,
ILogger<SiteProjection> logger)
;
}
```

Pennington.Pipeline.SiteProjectionOptions

Pennington.Pipeline.SiteProjectionOptions

CSHARP

```
namespace Pennington.Pipeline;

/// Configuration for ISiteProjection: the shared corpus projection consumed by every site-wide aggregator (search index, llms.txt, link audit).
public class SiteProjectionOptions
{
    /// CSS selector identifying the main content element inside the rendered page HTML (e.g. #main-content, article, main). When null, the entire <body> is used. Layouts that wrap content in a navigation/footer chrome should set this so the chrome does not leak into the search index or llms.txt sidecars.
    public string ContentSelector { get; set; }

}
```

Pennington.Pipeline.SocialMetadata

Pennington.Pipeline.SocialMetadata

CSHARP

```
namespace Pennington.Pipeline;

/// Open Graph / social card metadata for a page.
public record SocialMetadata
{
    /// Author name for articles.
    public string Author { get; set; }

    /// Page description.
    public string Description { get; set; }

    /// URL of the social preview image.
    public string ImageUrl { get; set; }

    /// Publication timestamp for articles.
    public DateTime? PublishedTime { get; set; }

    /// Open Graph / social card metadata for a page.
    public SocialMetadata(string Description, string ImageUrl, string Type, DateTime? PublishedTime, string Author)
;

    /// Open Graph type (e.g., "article", "website").
    public string Type { get; set; }

}
```

Pennington.Pipeline.Tag

Pennington.Pipeline.Tag

CSHARP

```csharp
namespace Pennington.Pipeline;

/// A content tag with a display name and URL-safe slug.
public record Tag
{
    /// Human-readable tag name.
    public string Name { get; set; }

    /// URL-safe slug used in tag routes.
    public string Slug { get; set; }

    /// A content tag with a display name and URL-safe slug.
    public Tag(string Name, string Slug)
;
}
```

Pennington.Routing.CanonicalBaseUrl

Pennington.Routing.CanonicalBaseUrl

CSHARP

```csharp
namespace Pennington.Routing;

/// Singleton wrapper for the site's effective canonical base URL, used by generators
(sitemap, RSS, llms.txt) to produce absolute links. Resolved from
PenningtonOptions.CanonicalBaseUrl when set, otherwise from OutputOptions.BaseUrl.
public record CanonicalBaseUrl
{
    /// Singleton wrapper for the site's effective canonical base URL, used by generators
(sitemap, RSS, llms.txt) to produce absolute links. Resolved from
PenningtonOptions.CanonicalBaseUrl when set, otherwise from OutputOptions.BaseUrl.
    public CanonicalBaseUrl(UrlPath Value)
;

    /// Combines the base with a site-relative path; see Combine.
    public UrlPath Combine(UrlPath relative)
;

    /// The underlying canonical base (an origin like https://site.com, or a path like / or
/sub/).
    public UrlPath Value { get; set; }

}
```

Pennington.Routing.ContentRoute

Pennington.Routing.ContentRoute

CSHARP

```
namespace Pennington.Routing;

/// Describes the canonical location of a piece of content, its output file, and originating
locale.
public record ContentRoute
{
    /// Compose the canonical path with the site's canonical base URL; see Combine.
    public UrlPath AbsoluteUrl(UrlPath canonicalBase)
;

    /// Canonical URL path (leading slash, trailing slash for directories).
    public UrlPath CanonicalPath { get; set; }

    /// True when this route belongs to the default (unprefixed) locale.
    public bool IsDefaultLocale { get; }

    /// True when this route serves default-locale content as a fallback for a missing
translation.
    public bool IsFallback { get; set; }

    /// Locale code for this route; empty for the default locale.
    public string Locale { get; set; }

    /// Relative output file path written during static generation.
    public FilePath OutputFile { get; set; }

    /// Originating source file on disk, if any.
    public FilePath? SourceFile { get; set; }

}
```

Pennington.Routing.ContentRouteFactory

Pennington.Routing.ContentRouteFactory

CSHARP

```
namespace Pennington.Routing;

/// Creates ContentRoute instances from source file paths or URL paths.
public class ContentRouteFactory
{
    /// Redirect: source URL to route (output is redirect HTML).
    public static ContentRoute ForRedirect(UrlPath sourceUrl)
;

    /// Custom: for non-markdown content services.
    public static ContentRoute FromCustom(UrlPath url, FilePath? sourceFile = default,
string locale = "")
;

    /// Markdown: file path to route. Converts a markdown file path relative to contentRoot
into a URL. Example: sourceFile="Content/Docs/getting-started.md",
contentRoot="Content/Docs", basePageUrl="/docs" results in CanonicalPath="/docs/getting-
started/", OutputFile="docs/getting-started/index.html"
    public static ContentRoute FromMarkdownFile(FilePath sourceFile, FilePath contentRoot,
UrlPath basePageUrl, string locale = "")
;

    /// Razor: @page directive to route.
    public static ContentRoute FromRazorPage(string pageRoute, string locale = "")
;

    /// Programmatic: explicit URL to route.
    public static ContentRoute FromUrl(UrlPath url, string locale = "")
;
}
```

Pennington.Routing.FilePath

Pennington.Routing.FilePath

CSHARP

```
namespace Pennington.Routing;

/// A file system path value used for content source and output locations.
public struct FilePath
{
    /// The file extension including the leading dot, or empty if none.
    public string Extension { get; }

    /// The file name with extension.
    public string FileName { get; }

    /// The file name without its extension.
    public string FileNameWithoutExtension { get; }

    /// A file system path value used for content source and output locations.
    public FilePath(string Value)
;

    /// Resolves path against root: returns it unchanged when it is already rooted or when root is null or empty, otherwise combines the two.
    public static string ResolveAgainstRoot(string path, string root)
;

    /// Returns the underlying path string.
    public string ToString()
;

    /// Underlying path string.
    public string Value { get; set; }

}
```

Pennington.Routing.UrlComposer

Pennington.Routing.UrlComposer

CSHARP

```
namespace Pennington.Routing;

/// Composes a canonical base URL with a site-relative path, yielding either a fully-qualified URL (when the base has an http(s) scheme) or a root-relative path (when the base is path-only like / or /sub/).
public class UrlComposer
{
    /// Combines canonicalBase with relative to produce an absolute URL when the base has an http(s) scheme, or a normalized root-relative path otherwise.
    public static UrlPath Combine(UrlPath canonicalBase, UrlPath relative)
;
}
```

Pennington.Routing.UrlPath

Pennington.Routing.UrlPath

CSHARP

```
namespace Pennington.Routing;

/// A URL path value supporting composition and normalization.
public struct UrlPath
{
    /// Returns a path guaranteed to start with a slash.
    public UrlPath EnsureLeadingSlash()
;

    /// Returns a path guaranteed to end with a slash.
    public UrlPath EnsureTrailingSlash()
;

    /// Compares two URL paths ignoring trailing slashes, index.html suffixes, and case.
    public bool Matches(UrlPath other)
;

    /// Removes a leading slash if present.
    public UrlPath RemoveLeadingSlash()
;

    /// Removes a trailing slash (except from the root path).
    public UrlPath RemoveTrailingSlash()
;

    /// Returns the underlying URL string.
    public string ToString()
;

    /// A URL path value supporting composition and normalization.
    public UrlPath(string Value)
;

    /// Underlying path string.
    public string Value { get; set; }

}
```

Pennington.Search.HeadingSection

Pennington.Search.HeadingSection

CSHARP

```
namespace Pennington.Search;

/// One heading-delimited section of a rendered page: a heading plus the text beneath it,
/// down to the next heading of the same or higher level. The text before the first heading is
/// the IsLead section.
public record HeadingSection
{
    /// The heading's anchor id (for deep-linking), or null for the lead section.
    public string AnchorId { get; set; }

    /// Ancestor heading texts (excluding this heading and the page title), nearest-last.
    public IReadOnlyList<string> Crumbs { get; set; }

    /// One heading-delimited section of a rendered page: a heading plus the text beneath
it, down to the next heading of the same or higher level. The text before the first heading
is the IsLead section.
    public HeadingSection(string AnchorId, string Title, int Level, IReadOnlyList<string>
Crumbs, string Text, bool IsLead)
;

    /// True for the page-lead section (content before the first heading).
    public bool IsLead { get; set; }

    /// The heading level (2–6), or 1 for the lead section.
    public int Level { get; set; }

    /// Plain-text content of the section, whitespace-collapsed.
    public string Text { get; set; }

    /// The heading text, or empty for the lead section (the host supplies the page title).
    public string Title { get; set; }

}
```

Pennington.Search.HeadingSectionExtractor

Pennington.Search.HeadingSectionExtractor

CSHARP

```
namespace Pennington.Search;

/// Splits post-pipeline page HTML into one HeadingSection per heading (plus a lead section) so the search index can carry heading-level records that deep-link to anchors. Walks the rendered content element in document order; h2–h6 with an id start a new section, h1 is treated as the page title (not indexed into a section body), and <pre> subtrees are dropped when code blocks are excluded.
public class HeadingSectionExtractor
{
    /// Extracts the lead section plus one section per anchored heading from content.
    public IReadOnlyList<HeadingSection> Extract(IElement content, bool excludeCodeBlocks)
;
}
```

Pennington.Search.IHasSearchFacets

Pennington.Search.IHasSearchFacets

CSHARP

```
namespace Pennington.Search;

/// Capability interface for an IFrontMatter type that declares custom search facet axes beyond the built-in section/tag/area dimensions. Each entry becomes a facet dimension on the page's search records, so the client can filter on it.
public interface IHasSearchFacets
{
    /// Custom facet axes for this page: each key is a facet dimension (for example "company" or "language"), each value the page's membership values within that dimension. Return an empty dictionary to contribute none.
    public IReadOnlyDictionary<string, string[]> SearchFacets { get; }

}
```

Pennington.Search.SearchArtifactContentService

Pennington.Search.SearchArtifactContentService

CSHARP

```
namespace Pennington.Search;

/// Artifact-tier façade over SearchArtifactService: claims the sharded index territory
under /search/, serves shards in dev through the artifact router, and enumerates the same
files for the static build — one byte path for both surfaces. Transient so each resolution
captures the current file-watched service.
public class SearchArtifactContentService
{
    /// URL territories this service serves. Options-derived and consulted on every request
— must be cheap and must not trigger discovery, the projection, or any lazy corpus work.
    public ImmutableList<ArtifactClaim> Claims { get; }

    /// Enumerates every artifact route the static build should write, as GeneratedSource
items. May consume the projection — the build invokes this outside any request, after the
page crawl has primed the render cache. Never called on the request path.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Returns the bytes for relativePath (no leading slash, e.g. search/en/index.json), or
null to decline so the request falls through to content routing. May materialize the
projection, build an index, or run Chromium on demand.
    public Task<ArtifactContent> ResolveAsync(string relativePath, CancellationToken
cancellationToken)
;

    /// Creates the façade over the given SearchArtifactService.
    public SearchArtifactContentService(SearchArtifactService service)
;
}
```

Pennington.Search.SearchArtifactService

Pennington.Search.SearchArtifactService

CSHARP

```
namespace Pennington.Search;

/// Builds the sharded search artifacts for every configured locale and exposes them as a
single path -> bytes map — the one source of truth behind SearchArtifactContentService,
which serves them in dev and enumerates them for the static build. Folds over
ISiteProjection — every page's post-pipeline HTML and heading-split sections have already
been produced once by the shared projection, so this service is a pure mapping from
RenderedPage + HeadingSection to SearchDocument. The corpus is grouped by locale and handed
to the external DeweySearch IndexBuilder; the resulting per-locale artifacts are laid out
under search/{locale}/. Computed lazily and recreated on file changes when managed by
FileWatchDependencyFactory.
public class SearchArtifactService
{
    /// Returns the bytes for a single artifact path, or null when no artifact matches.
    public Task<byte[]> GetArtifactAsync(string relativePath)
;

    /// Returns every artifact keyed by its relative output path (e.g.
search/en/index.json).
    public Task<IReadOnlyDictionary<string, byte[]>> GetArtifactFilesAsync()
;

    /// Called on the file-watcher thread for every watched change. Must be quick and
thread-safe.
    public FileWatchResponse OnFileChanged(FileChangeNotification change)
;

    /// Creates the service; artifacts are computed lazily on first request.
    public SearchArtifactService(ISiteProjection projection, SearchIndexBuilder
corpusBuilder, IndexBuilder indexBuilder, LocalizationOptions localization,
ContentRecordRegistry recordRegistry)
;
}
```

Pennington.Search.SearchFacetField

Pennington.Search.SearchFacetField

CSHARP

```
namespace Pennington.Search;

/// Facet dimensions the search index can surface for client-side filtering. Combine with
bitwise OR to enable several at once.
public enum SearchFacetField
{
    /// Facet on the content area — the first URL segment after any locale prefix.
    public static const SearchFacetField Area
;

    /// No facets are generated.
    public static const SearchFacetField None
;

    /// Facet on SectionLabel.
    public static const SearchFacetField Section
;

    /// Facet on page tags (front matter implementing ITaggable).
    public static const SearchFacetField Tags
;
}
```

Pennington.Search.SearchIndexBuilder

Pennington.Search.SearchIndexBuilder

CSHARP

```
namespace Pennington.Search;

/// Maps a page's HeadingSections onto DeweySearch SearchDocument records — one record per
heading (plus a page-lead record) so search results are heading-level and deep-link to
anchors. This is the Pennington-specific adapter: it builds anchor URLs, the page→heading
breadcrumb trail, and maps the content model (section label, tags, content area) onto
DeweySearch's open facet dictionary. Draft filtering is handled upstream by each
IContentService's TOC builder.
public class SearchIndexBuilder
{
    /// Builds a DeweySearch SearchDocument for one section of a page. The lead section
becomes the page record (page URL, page title, page description); each heading section
becomes an anchored record carrying the page→heading breadcrumb trail.
    public SearchDocument BuildSection(ContentTocItem toc, HeadingSection section,
IFrontMatter metadata = null)
;

    /// Creates the adapter from the search options and localization (for content-area
derivation).
    public SearchIndexBuilder(SearchIndexOptions options, LocalizationOptions localization)
;
}
```

Pennington.Search.SearchIndexOptions

Pennington.Search.SearchIndexOptions

CSHARP

```
namespace Pennington.Search;

/// Configuration for search index generation.
public class SearchIndexOptions
{
    /// Per-area search priority (area slug to priority); higher ranks first. Documents whose area is absent from the map use DefaultPriority. The DocSite derives this from its configured area order so results lean toward earlier areas (e.g. tutorials/how-to over reference) when matches are otherwise comparable. Default: empty (uniform priority).
    public Dictionary<string, int> AreaPriorities { get; set; }

    /// Default priority assigned to indexed documents. Default: 5.
    public int DefaultPriority { get; set; }

    /// Facet dimensions to generate for client-side filtering. Default: Area only — content areas are few and stable, so they make clean filter chips. Section and tag faceting are opt-in because their vocabularies are large enough to bury the filter bar in chips; combine the flags to re-enable (e.g. Area | Section | Tags).
    public SearchFacetField Facets { get; set; }

    /// Upper bound on the edit distance the client applies for typo-tolerant matching. The client also scales the budget down for short terms; this caps it. Set to 0 to require exact matches. Default: 2.
    public int MaxEditDistance { get; set; }

    /// Per-route-prefix search priority (URL prefix to priority), checked before AreaPriorities; the longest matching prefix wins and its value replaces the area value rather than stacking. Drops a whole URL territory below comparable prose that keeps its area lean — generated API reference, for example, registers its route prefix here so its pages don't bury conceptual articles. Default: empty (area/default apply).
    public Dictionary<string, int> PrefixPriorities { get; set; }

    /// Number of leading characters of a stemmed term used as its shard key. Lower values produce fewer, larger shards; higher values produce more, smaller shards. Default: 2.
    public int ShardPrefixLength { get; set; }

    /// Query-time synonyms. Each entry maps a term to the additional terms it should also match. Keys and values are stemmed at build time and shipped in the index entrypoint, so authors write natural words (e.g. "config" => ["configuration"]). Default: empty.
    public Dictionary<string, string[]> Synonyms { get; set; }

}
```

Pennington.SocialCards.SocialCardContentService

Pennington.SocialCards.SocialCardContentService

CSHARP

```
namespace Pennington.SocialCards;

/// Discovers one social-card image route per content page so the static build bakes a card for every page: the crawler HTTP-fetches each discovered route and the sibling MapSocialCards endpoint renders it on demand. Mirrors TaxonomyContentService — it emits EndpointSource routes served by an endpoint and projects no records of its own.
public class SocialCardContentService
{
    /// Default section label applied to discovered items that do not supply one via front matter.
    public string DefaultSectionLabel { get; }

    /// Discover all content items this service is responsible for.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Navigation entries for table of contents.
    public Task<ImmutableList<ContentTocItem>> GetContentTocEntriesAsync()
;

    /// Static files to copy to output (images, downloads, etc.)
    public Task<ImmutableList<ContentToCopy>> GetContentToCopyAsync()
;

    /// Cross-references for xref resolution.
    public Task<ImmutableList<CrossReference>> GetCrossReferencesAsync()
;

    /// Projects this service's routable content as ContentRecords — the discovery seam consumed by taxonomy, search faceting, and structured-data emission. Default: bridges from DiscoverAsync, yielding one record per discovered item that carries Metadata and is neither a RedirectSource (transport, not content) nor an LlmsOnlySource (no human-facing URL). A service that attaches typed metadata to its discovered items — as MarkdownContentService does — therefore participates with no extra code. Override only to project records that do not flow through DiscoverAsync, or to suppress records entirely.A service that emits routable content from DiscoverAsync but leaves Metadata unset projects no records, and so silently sits out of taxonomy, search faceting, and structured data. Set the metadata (or override this) to opt in.
    public IAsyncEnumerable<ContentRecord> GetRecordsAsync()
;

    /// Relative priority for ordering results in the search index (higher values rank first).
    public int SearchPriority { get; }

    /// Creates the service. Sibling content services are resolved on demand to avoid a DI cycle.
    public SocialCardContentService(IServiceProvider serviceProvider, SocialCardOptions options)
;
}
```

Pennington.SocialCards.SocialCardEndpointExtensions

Pennington.SocialCards.SocialCardEndpointExtensions

CSHARP

```
namespace Pennington.SocialCards;

/// Endpoint helper that renders social-card images on demand.
public class SocialCardEndpointExtensions
{
    /// Maps {BaseUrl}/{**slug}.png to the host's Render hook. Serves cards live during development and is the route the static build crawler fetches to bake each card discovered by SocialCardContentService. Resolving page metadata from ContentRecordRegistry (a discovery-time join, not the request-path-forbidden ISiteProjection) keeps this safe to consume from a live request.
    public static IEndpointRouteBuilder MapSocialCards(IEndpointRouteBuilder routes)
;
}
```

Pennington.SocialCards.SocialCardOptions

Pennington.SocialCards.SocialCardOptions

CSHARP

```csharp
namespace Pennington.SocialCards;

/// Host configuration for social-card (OpenGraph / Twitter image) generation. Pennington owns the discovery, routing, and meta-tag wiring; the host owns the drawing via Render and brings its own image library (ImageSharp, SkiaSharp, Playwright, ...). Set this on SocialCards (templates forward it from their own options) to enable the feature; leaving it null disables it.
public record SocialCardOptions
{
    /// URL prefix under which card routes are published (e.g. /social-cards/blog/post.png).
    public string BaseUrl { get; set; }

    /// MIME type served for, and meta-tagged on, the generated card.
    public string ContentType { get; set; }

    /// Card height in pixels passed to Render. Defaults to the 1200x630 OpenGraph standard.
    public int Height { get; set; }

    /// Host renderer invoked once per page to produce the card image bytes. Receives the request's IServiceProvider so a renderer can resolve registered services (font caches, theming options, ...). Return null to skip a page (its card route then serves 404 and is omitted from the build).
    public Func<SocialCardRequest, IServiceProvider, CancellationToken, Task<byte[]>> Render { get; set; }

    /// Card width in pixels passed to Render. Defaults to the 1200x630 OpenGraph standard.
    public int Width { get; set; }

}
```

Pennington.SocialCards.SocialCardRequest

Pennington.SocialCards.SocialCardRequest

CSHARP

```
namespace Pennington.SocialCards;

/// Everything Render needs to draw a card for one page: the page's resolved metadata plus the site identity and the card's own absolute URL.
public record SocialCardRequest
{
    /// Canonical path of the page the card represents.
    public UrlPath CanonicalPath { get; set; }

    /// The card's own URL — absolute when a canonical base URL is configured, else root-relative.
    public string CardUrl { get; set; }

    /// Publication date, when set.
    public DateTime? Date { get; set; }

    /// Page description, when the front matter supplies one.
    public string Description { get; set; }

    /// Requested card height in pixels.
    public int Height { get; set; }

    /// Locale code of the page, or null for the default locale.
    public string Locale { get; set; }

    /// The page's full typed front matter, so a renderer can read tags, author, and other capabilities.
    public IFrontMatter Metadata { get; set; }

    /// Site description, available for fallback copy.
    public string SiteDescription { get; set; }

    /// Site title, for branding the card.
    public string SiteTitle { get; set; }

    /// Everything Render needs to draw a card for one page: the page's resolved metadata plus the site identity and the card's own absolute URL.
    public SocialCardRequest(string Title, string Description, DateTime? Date, UrlPath CanonicalPath, string CardUrl, string Locale, string SiteTitle, string SiteDescription, IFrontMatter Metadata, int Width, int Height)
    ;

    /// Page title.
    public string Title { get; set; }

    /// Requested card width in pixels.
    public int Width { get; set; }

}
```

Pennington.SocialCards.SocialCardUrl

Pennington.SocialCards.SocialCardUrl

CSHARP

```
namespace Pennington.SocialCards;

/// The single card-URL convention shared by the discovery service
(SocialCardContentService), the rendering endpoint (MapSocialCards), and the templates'
meta-tag wiring, so a page's canonical path and its card URL always agree:
{BaseUrl}/{canonical-path}.png, with the home page (empty path) reserved as
{BaseUrl}/index.png.
public class SocialCardUrl
{
    /// Card URL for a page — absolute when canonicalBaseUrl is set (OpenGraph crawlers
require an absolute og:image), otherwise the root-relative path.
    public static string For(UrlPath canonicalPath, string baseUrl, string canonicalBaseUrl)
;

    /// Reserved slug for the home page, whose canonical path trims to the empty string.
    public static const string HomeSlug
;

    /// Root-relative card path for a page, e.g. /social-cards/blog/my-post.png.
    public static string RelativePath(UrlPath canonicalPath, string baseUrl)
;

    /// Reverses RelativePath: maps the catch-all slug captured after BaseUrl (e.g. blog/my-
post.png) back to a ContentRecordRegistry key (blog/my-post); the home slug maps to the
empty key.
    public static string SlugToRecordKey(string slug)
;
}
```

Pennington.StandardSite.AtUri

Pennington.StandardSite.AtUri

CSHARP

```
namespace Pennington.StandardSite;

/// Builds and parses AT Protocol at:// URIs of the form at://{did}/{collection}/{rkey}.
Pure string work — no PDS access.
public class AtUri
{
    /// Composes an at:// URI from its parts.
    public static string Build(string did, string collection, string rkey)
;

    /// Parses an at:// URI into its DID, collection NSID, and record key, or returns null
when the input is not a well-formed three-part AT-URI.
    public static ValueTuple<string, string, string>? Parse(string uri)
;
}
```

Pennington.StandardSite.StandardSiteOptions

Pennington.StandardSite.StandardSiteOptions

CSHARP

```
namespace Pennington.StandardSite;

/// Configures the Standard Site (AT Protocol long-form publishing) integration. The
verification surface — the well-known files and per-page site.standard.* head links — needs
only Did and PublicationRkey and writes nothing to any PDS. When either is blank the feature
emits nothing (fail-safe). Records themselves are authored out-of-band (an editor such as
standard.horse); this engine only publishes the proof that points at them.
public record StandardSiteOptions
{
    /// Author/owner DID, e.g. did:plc:abc123. Required for any output.
    public string Did { get; set; }

    /// Resolves a page's site.standard.document record key from its content record.
Defaults to reading AtprotoRkey off the front matter, so hosts that don't implement that
capability can map the rkey from their own metadata instead.
    public Func<ContentRecord, string> DocumentRkeyResolver { get; set; }

    /// When true, also emit /.well-known/atproto-did so the site's domain doubles as the
author's atproto handle. Default false: this is a stronger, separate claim than publication
verification, and a mismatched DID can break an existing domain handle.
    public bool EmitAtprotoDid { get; set; }

    /// When true (the default), emit the site-wide <link rel="site.standard.publication">
discovery hint in every page head — what a Bluesky card reader looks for.
    public bool EmitPublicationLink { get; set; }

    /// True when both Did and PublicationRkey are set.
    public bool IsConfigured { get; }

    /// Path the publication is served under, with a leading slash and no trailing slash
(default "" = domain root). Appended to the publication well-known suffix.
    public string PublicationPath { get; set; }

    /// Record key of the site.standard.publication record in the author's repo. The
publication AT-URI is at://{Did}/site.standard.publication/{PublicationRkey}.
    public string PublicationRkey { get; set; }

}
```

Pennington.StandardSite.StandardSiteUriResolver

Pennington.StandardSite.StandardSiteUriResolver

CSHARP

```
namespace Pennington.StandardSite;

/// Resolves Standard Site AT-URIs: the site-wide publication URI (config only) and a per-page site.standard.document URI (joining the request path to the content registry, then reading the page's rkey via DocumentRkeyResolver). Registered transient so it captures the current file-watched ContentRecordRegistry.
public class StandardSiteUriResolver
{
    /// The site.standard.document AT-URI for a request path, or null when the page resolves to no record or declares no rkey. Keys the registry on fullPath.Trim('/') — identical to the structured-data join.
    public Task<string> DocumentUriAsync(string fullPath)
;

    /// The publication AT-URI (at://{Did}/site.standard.publication/{PublicationRkey}).
    public string PublicationUri { get; }

    /// Creates the resolver from the Standard Site options and the record registry.
    public StandardSiteUriResolver(StandardSiteOptions options, ContentRecordRegistry records)
;
}
```

Pennington.StandardSite.WellKnownArtifactService

Pennington.StandardSite.WellKnownArtifactService

CSHARP

```
namespace Pennington.StandardSite;

/// Artifact-tier service for the Standard Site verification well-known files: the
publication AT-URI at /.well-known/site.standard.publication[{path}] and, when enabled, the
bare DID at /.well-known/atproto-did. Bytes derive from options alone — the trivial proof
that the artifact tier does not require the site projection. Claims nothing when the options
are incompletely configured (fail-safe). Transient so it captures current options.
public class WellKnownArtifactService
{
    /// URL territories this service serves. Options-derived and consulted on every request
— must be cheap and must not trigger discovery, the projection, or any lazy corpus work.
    public ImmutableList<ArtifactClaim> Claims { get; }

    /// Enumerates every artifact route the static build should write, as GeneratedSource
items. May consume the projection — the build invokes this outside any request, after the
page crawl has primed the render cache. Never called on the request path.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Returns the bytes for relativePath (no leading slash, e.g. search/en/index.json), or
null to decline so the request falls through to content routing. May materialize the
projection, build an index, or run Chromium on demand.
    public Task<ArtifactContent> ResolveAsync(string relativePath, CancellationToken
cancellationToken)
;

    /// Creates the service; claims derive from the options alone.
    public WellKnownArtifactService(StandardSiteOptions options)
;
}
```

Pennington.StructuredData.IHasStructuredData

Pennington.StructuredData.IHasStructuredData

CSHARP

```
namespace Pennington.StructuredData;

/// Capability interface for an IFrontMatter type that can describe its own schema.org JSON-
LD. Templates that render content (DocSite, BlogSite, custom hosts) check for this
capability and emit the returned entities through <StructuredData>.
public interface IHasStructuredData
{
    /// Returns the schema.org entities to emit on the page. Implementations typically yield
one Article/Recipe/Product/etc. built from front matter values, plus the CanonicalUrl the
template supplies.
    public IEnumerable<JsonLdEntity> GetStructuredData(StructuredDataContext context)
;
}
```

Pennington.StructuredData.JsonLdArticle

Pennington.StructuredData.JsonLdArticle

CSHARP

```csharp
namespace Pennington.StructuredData;

/// schema.org Article — a blog post or content page, emitted in the page head.
public record JsonLdArticle
{
    /// Author entity. Omitted when null.
    public JsonLdPerson Author { get; set; }

    /// Publication date. Omitted when null; serialized as yyyy-MM-ddTHH:mm:ssZ.
    public DateTime? DatePublished { get; set; }

    /// Short description of the article. Omitted when null.
    public string Description { get; set; }

    /// Article headline.
    public string Headline { get; set; }

    /// schema.org type literal (e.g. "Article", "Recipe").
    public string Type { get; }

    /// Canonical URL of the article.
    public string Url { get; set; }

}
```

Pennington.StructuredData.JsonLdBreadcrumbItem

Pennington.StructuredData.JsonLdBreadcrumbItem

CSHARP

```
namespace Pennington.StructuredData;

/// A single rung in a JsonLdBreadcrumbList.
public record JsonLdBreadcrumbItem
{
    /// Display name for this crumb.
    public string Name { get; set; }

    /// 1-based position of the item in the crumb trail.
    public int Position { get; set; }

    /// schema.org @type literal.
    public string Type { get; }

    /// URL the crumb links to. Omitted when null (typically the current page).
    public string Url { get; set; }

}
```

Pennington.StructuredData.JsonLdBreadcrumbList

Pennington.StructuredData.JsonLdBreadcrumbList

CSHARP

```
namespace Pennington.StructuredData;

/// schema.org BreadcrumbList describing a page's place in the navigation tree.
public record JsonLdBreadcrumbList
{
    /// Ordered crumb items.
    public IReadOnlyList<JsonLdBreadcrumbItem> Items { get; set; }

    /// schema.org type literal (e.g. "Article", "Recipe").
    public string Type { get; }

}
```

Pennington.StructuredData.JsonLdDateConverter

Pennington.StructuredData.JsonLdDateConverter

CSHARP

```
namespace Pennington.StructuredData;

/// Emits DateTime values in the JSON-LD wire format yyyy-MM-ddTHH:mm:ssZ regardless of
Kind.
public class JsonLdDateConverter
{
    /// Reads a JSON-LD date string back to a UTC DateTime.
    public DateTime Read(ref Utf8JsonReader reader, Type typeToConvert,
JsonSerializerOptions options)
;

    /// Writes a DateTime as yyyy-MM-ddTHH:mm:ssZ.
    public void Write(Utf8JsonWriter writer, DateTime value, JsonSerializerOptions options)
;
}
```

Pennington.StructuredData.JsonLdEntity

Pennington.StructuredData.JsonLdEntity

CSHARP

```
namespace Pennington.StructuredData;

/// Base record for a schema.org JSON-LD entity. Subclass with JsonPropertyNameAttribute-
decorated properties and override Type to declare a new schema.org type. Repeat the
[JsonPropertyName("@type")] attribute on the override — Json does not inherit it from the
abstract base.
public record JsonLdEntity
{
    /// JSON-LD context. Defaults to schema.org; override for a different vocabulary.
    public string Context { get; set; }

    /// schema.org type literal (e.g. "Article", "Recipe").
    public string Type { get; }

}
```

Pennington.StructuredData.JsonLdPerson

Pennington.StructuredData.JsonLdPerson

CSHARP

```
namespace Pennington.StructuredData;

/// schema.org Person — author or contributor on an JsonLdArticle.
public record JsonLdPerson
{
    /// Display name of the person.
    public string Name { get; set; }

    /// schema.org @type literal.
    public string Type { get; }

}
```

Pennington.StructuredData.JsonLdSerializer

Pennington.StructuredData.JsonLdSerializer

CSHARP

```
namespace Pennington.StructuredData;

/// Serializes any JsonLdEntity subclass to a JSON string safe for embedding in a <script type="application/ld+json"> tag.
public class JsonLdSerializer
{
    /// Serializes entity to JSON-LD. The concrete runtime type is used so subclass-only properties are included.
    public static string Serialize(JsonLdEntity entity)
;
}
```

Pennington.StructuredData.JsonLdWebSite

Pennington.StructuredData.JsonLdWebSite

CSHARP

```
namespace Pennington.StructuredData;

/// schema.org WebSite — site-level identity, emitted on the home page.
public record JsonLdWebSite
{
    /// Short site description. Omitted when null.
    public string Description { get; set; }

    /// Site name.
    public string Name { get; set; }

    /// schema.org type literal (e.g. "Article", "Recipe").
    public string Type { get; }

    /// Site canonical URL.
    public string Url { get; set; }

}
```

Pennington.StructuredData.StructuredDataContext

Pennington.StructuredData.StructuredDataContext

CSHARP

```
namespace Pennington.StructuredData;

/// Per-request context passed to GetStructuredData so a front matter implementation can
build canonical URLs and apply site-level fallbacks it does not own (e.g. the BlogSite
author).
public record StructuredDataContext
{
    /// Absolute canonical URL for the page, including base URL and trailing slash.
    public string CanonicalUrl { get; set; }

    /// Site-level author fallback. Honored when the front matter has no author of its own
and the template has a default author configured (BlogSite's BlogSiteOptions.AuthorName).
May be null.
    public string FallbackAuthorName { get; set; }

}
```

Pennington.Taxonomy.ITaxonomyContentService

Pennington.Taxonomy.ITaxonomyContentService

CSHARP

```
namespace Pennington.Taxonomy;

/// Marker interface for every closed-generic TaxonomyContentService. Lets a taxonomy
service identify other taxonomies in the registered IContentService set without reflecting
on the open generic — used to break the otherwise-circular dependency when two taxonomies
sit on the same content tree.
public interface ITaxonomyContentService
{
}
```

Pennington.Taxonomy.TaxonomyAccessor

Pennington.Taxonomy.TaxonomyAccessor

CSHARP

```
namespace Pennington.Taxonomy;

/// Resolves the registered TaxonomyContentService for a given BaseUrl so a routed Razor
@page can read a taxonomy's terms directly — the alternative to mounting the bare-render
MapTaxonomy endpoints when the page wants the host's full layout, chrome, and search
indexing.
public class TaxonomyAccessor
{
    /// Returns the term list for the axis mounted at baseUrl, or empty when none matches.
    public Task<ImmutableList<TaxonomyTerm<TFrontMatter, TKey>>> GetTermsAsync<TFrontMatter,
TKey>(string baseUrl)
;

    /// Creates the accessor over the registered content services.
    public TaxonomyAccessor(IEnumerable<IContentService> services)
;

    /// Looks up a single term by slug on the axis mounted at baseUrl; null when absent.
    public Task<TaxonomyTerm<TFrontMatter, TKey>> TryGetTermAsync<TFrontMatter, TKey>(string
baseUrl, string slug)
;
}
```

Pennington.Taxonomy.TaxonomyContentService

Pennington.Taxonomy.TaxonomyContentService

CSHARP

```
namespace Pennington.Taxonomy;

/// Walks every other registered IContentService's ContentRecords, selects those whose
Metadata is a TFrontMatter, projects keys via SelectKey or SelectKeys, and exposes the
result as the taxonomy's index plus one route per term. Because it reads records rather than
re-parsing markdown files, any content service — markdown or custom — participates as long
as it projects records of the taxonomy's front-matter type. The service emits its routes
with EndpointSource — the canonical HTML is produced by the sibling MapTaxonomy endpoints,
mirroring the pattern documented in how-to/content-services/custom-content-service.md. As a
consequence the routes do not appear in sitemap.xml; they do appear in navigation, search,
and cross-references through GetContentTocEntriesAsync and GetCrossReferencesAsync.The
service caches its computed term list in an AsyncLazy and subscribes to IFileWatcher so any
change anywhere in the watched content tree drops the cache and the next request rebuilds
it.
public class TaxonomyContentService
{
    /// The configured base URL (e.g. /cuisine) — exposed so endpoint mapping can find it.
    public string BaseUrl { get; }

    /// Default section label applied to discovered items that do not supply one via front
matter.
    public string DefaultSectionLabel { get; }

    /// Discover all content items this service is responsible for.
    public IAsyncEnumerable<DiscoveredItem> DiscoverAsync()
;

    /// Navigation entries for table of contents.
    public Task<ImmutableList<ContentTocItem>> GetContentTocEntriesAsync()
;

    /// Static files to copy to output (images, downloads, etc.)
    public Task<ImmutableList<ContentToCopy>> GetContentToCopyAsync()
;

    /// Cross-references for xref resolution.
    public Task<ImmutableList<CrossReference>> GetCrossReferencesAsync()
;

    /// Projects this service's routable content as ContentRecords — the discovery seam
consumed by taxonomy, search faceting, and structured-data emission. Default: bridges from
DiscoverAsync, yielding one record per discovered item that carries Metadata and is neither
a RedirectSource (transport, not content) nor an LlmsOnlySource (no human-facing URL). A
service that attaches typed metadata to its discovered items — as MarkdownContentService
does — therefore participates with no extra code. Override only to project records that do
not flow through DiscoverAsync, or to suppress records entirely.A service that emits
routable content from DiscoverAsync but leaves Metadata unset projects no records, and so
silently sits out of taxonomy, search faceting, and structured data. Set the metadata (or
override this) to opt in.
    public IAsyncEnumerable<ContentRecord> GetRecordsAsync()
;
```

```
    public Task<ImmutableList<TaxonomyTerm<TFrontMatter, TKey>>> GetTermsAsync();      /// The Razor component to render the index page.    public Type IndexPage { get; }      /// The fully-resolved index URL with leading and trailing slashes.    public string IndexUrl { get; }      /// Relative priority for ordering results in the search index (higher values rank first).    public int SearchPriority { get; }      /// Creates the service and subscribes to file-change notifications for hot reload.    public TaxonomyContentService`2(TaxonomyOptions<TFrontMatter, TKey> options, IServiceProvider serviceProvider, IFileWatcher fileWatcher, TimeProvider clock = null);      /// The Razor component to render each per-term page.    public Type TermPage { get; }      /// Builds the per-term URL for the given slug.    public string TermUrl(string slug);      /// Looks up a single term by its slug. Returns null when not found.    public Task<TaxonomyTerm<TFrontMatter, TKey>> TryGetTermAsync(string slug);}
```

Pennington.Taxonomy.TaxonomyItem

Pennington.Taxonomy.TaxonomyItem

CSHARP

```
namespace Pennington.Taxonomy;

/// One content item discovered by the taxonomy walker, paired with the URL of the page that owns it. Surfaces the parsed front matter so term pages can render titles, dates, summaries, and any other field the consumer needs without re-reading the source file.
public record TaxonomyItem
{
    /// Parsed front matter from the source markdown file.
    public TFrontMatter FrontMatter { get; set; }

    /// One content item discovered by the taxonomy walker, paired with the URL of the page that owns it. Surfaces the parsed front matter so term pages can render titles, dates, summaries, and any other field the consumer needs without re-reading the source file.
    public TaxonomyItem`1(TFrontMatter FrontMatter, UrlPath Url)
;

    /// URL of the page the front matter belongs to.
    public UrlPath Url { get; set; }

}
```

Pennington.Taxonomy.TaxonomyOptions

Pennington.Taxonomy.TaxonomyOptions

CSHARP

```csharp
namespace Pennington.Taxonomy;

/// Configures one taxonomy axis (browse-by-cuisine, browse-by-tag, browse-by-audience, ...). Pass to AddTaxonomy; the extension validates the options and registers a TaxonomyContentService against them.
public class TaxonomyOptions
{
    /// Base URL the taxonomy mounts under. The index lives at this URL; per-term pages at {BaseUrl}/{slug}/. Required.
    public string BaseUrl { get; set; }

    /// Whether to publish a cross-reference per term (e.g. tag-csharp). Default true.
    public bool EmitCrossReferences { get; set; }

    /// Razor component that renders the index at BaseUrl. Receives the full term list as a Terms parameter. Required.
    public Type IndexPage { get; set; }

    /// Optional human-readable label override. Default returns the key's string form.
    public Func<TKey, string> LabelFor { get; set; }

    /// Search-index priority for the discovered routes. Higher ranks first. Default 10.
    public int SearchPriority { get; set; }

    /// Section label applied to the navigation entries. Defaults to the last segment of BaseUrl with the first letter uppercased.
    public string SectionLabel { get; set; }

    /// Single-valued projection. Set this OR SelectKeys — exactly one is required. Items whose projection returns null (when TKey is nullable) or the default value are skipped.
    public Func<TFrontMatter, TKey> SelectKey { get; set; }

    /// Multi-valued projection. Set this OR SelectKey — exactly one is required. Each returned key produces a (key, item) pair. Empty enumerations cause the item to be skipped.
    public Func<TFrontMatter, IEnumerable<TKey>> SelectKeys { get; set; }

    /// Slug encoder. Default lowercases the key's string form and replaces whitespace with hyphens; URL-encodes any remaining unsafe characters.
    public Func<TKey, string> SlugFor { get; set; }

    /// Razor component that renders each per-term page at {BaseUrl}/{slug}/. Receives the matching TaxonomyTerm as a Term parameter. Required.
    public Type TermPage { get; set; }

}
```

Pennington.Taxonomy.TaxonomyServiceExtensions

Pennington.Taxonomy.TaxonomyServiceExtensions

CSHARP

```
namespace Pennington.Taxonomy;

/// DI + endpoint helpers for registering and mounting taxonomy axes (browse-by-tag, browse-by-cuisine, browse-by-audience, ...).
public class TaxonomyServiceExtensions
{
    /// Registers a TaxonomyContentService configured by configure. Multiple AddTaxonomy calls with the same TFrontMatter/TKey pair coexist as long as each uses a distinct BaseUrl.
    public static IServiceCollection AddTaxonomy<TFrontMatter, TKey>(IServiceCollection services, Action<TaxonomyOptions<TFrontMatter, TKey>> configure)
;

    /// Mounts the live HTTP endpoints for every TaxonomyContentService registered for the given type pair. The index URL renders IndexPage with a Terms parameter; per-term URLs render TermPage with a Term parameter.
    public static IEndpointRouteBuilder MapTaxonomy<TFrontMatter, TKey>(IEndpointRouteBuilder routes)
;
}
```

Pennington.Taxonomy.TaxonomySlug

Pennington.Taxonomy.TaxonomySlug

CSHARP

```
namespace Pennington.Taxonomy;

/// The default taxonomy slug encoding, shared so links that point at a term page (e.g. tag chips on a post) produce the same URL that TaxonomyContentService discovers.
public class TaxonomySlug
{
    /// Lowercases value, collapses whitespace runs to single hyphens, and URL-encodes any remaining unsafe characters.
    public static string Slugify(string value)
;
}
```

Pennington.Taxonomy.TaxonomyTerm

Pennington.Taxonomy.TaxonomyTerm

CSHARP

```
namespace Pennington.Taxonomy;

/// One value of the taxonomy axis along with every content item that projects to it. Razor
pages registered with TermPage receive an instance of this type as their Term parameter;
index pages receive the full IReadOnlyList as their Terms parameter.
public record TaxonomyTerm
{
    /// Every content item that projected to this key, in the order they were discovered.
    public ImmutableList<TaxonomyItem<TFrontMatter>> Items { get; set; }

    /// The raw key value (the input to SlugFor).
    public TKey Key { get; set; }

    /// Human-readable label. Defaults to the key's string form; override via LabelFor.
    public string Label { get; set; }

    /// URL-safe slug; the last segment of Url.
    public string Slug { get; set; }

    /// One value of the taxonomy axis along with every content item that projects to it.
Razor pages registered with TermPage receive an instance of this type as their Term
parameter; index pages receive the full IReadOnlyList as their Terms parameter.
    public TaxonomyTerm`2(TKey Key, string Label, string Slug, UrlPath Url,
ImmutableList<TaxonomyItem<TFrontMatter>> Items)
;

    /// Absolute URL of this term page (e.g. /cuisine/italian/).
    public UrlPath Url { get; set; }

}
```

Pennington.TranslationAudit.CommitInfo

Pennington.TranslationAudit.CommitInfo

CSHARP

```
namespace Pennington.TranslationAudit;

/// Minimal commit metadata exposed by IGitHistoryReader.
public record CommitInfo
{
    /// Minimal commit metadata exposed by IGitHistoryReader.
    public CommitInfo(string Sha, DateTimeOffset When)
;

    /// Short commit hash (7 chars).
    public string Sha { get; set; }

    /// Commit author timestamp.
    public DateTimeOffset When { get; set; }

}
```

Pennington.TranslationAudit.IGitHistoryReader

Pennington.TranslationAudit.IGitHistoryReader

CSHARP

```
namespace Pennington.TranslationAudit;

/// Reads the most recent commit affecting a tracked file. Abstracted for testability.
public interface IGitHistoryReader
{
    /// Returns the latest commit touching absoluteFilePath, or null when the file is
untracked or the repo is missing.
    public CommitInfo GetLatestCommit(string absoluteFilePath)
;
}
```

Pennington.TranslationAudit.LibGit2GitHistoryReader

Pennington.TranslationAudit.LibGit2GitHistoryReader

CSHARP

```
namespace Pennington.TranslationAudit;

/// IGitHistoryReader backed by LibGit2Sharp. Holds the repository open for the process
lifetime and serializes access through a single instance because Repository is not thread-
safe.
public class LibGit2GitHistoryReader
{
    public void Dispose()
;

    /// Returns the latest commit touching absoluteFilePath, or null when the file is
untracked or the repo is missing.
    public CommitInfo GetLatestCommit(string absoluteFilePath)
;

    /// Opens the repository at repositoryRoot, falling back to a no-op reader when no repo
is found.
    public LibGit2GitHistoryReader(string repositoryRoot, ILogger<LibGit2GitHistoryReader>
logger)
;
}
```

Pennington.TranslationAudit.TranslationAuditExtensions

Pennington.TranslationAudit.TranslationAuditExtensions

CSHARP

```
namespace Pennington.TranslationAudit;

/// DI extensions for registering the translation auditor.
public class TranslationAuditExtensions
{
    /// Register TranslationAuditor as an IBuildAuditor. Diagnostics land in the dev overlay
(per-page) and in the build report (site-wide) automatically.
    public static IServiceCollection AddTranslationAudit(IServiceCollection services,
Action<TranslationAuditOptions> configure = null)
;
}
```

Pennington.TranslationAudit.TranslationAuditOptions

Pennington.TranslationAudit.TranslationAuditOptions

CSHARP

```
namespace Pennington.TranslationAudit;

/// Configuration for TranslationAuditor.
public class TranslationAuditOptions
{
    /// Locale codes to include. When null, every non-default locale configured in
LocalizationOptions is reported.
    public HashSet<string> IncludedLocales { get; set; }

    /// Severity for "translation file does not exist" diagnostics. Default Warning.
    public DiagnosticSeverity MissingSeverity { get; set; }

    /// Severity for "translation predates source's last commit" diagnostics. Default
Warning.
    public DiagnosticSeverity OutdatedSeverity { get; set; }

    /// When true (default), missing translations are reported. Set false to defer that
signal to a different mechanism.
    public bool ReportMissing { get; set; }

    /// Absolute path to the git repository root. When null, the repo is auto-discovered by
walking up from the current directory until a .git folder is found.
    public string RepositoryPath { get; set; }

}
```

Pennington.TranslationAudit.TranslationAuditor

Pennington.TranslationAudit.TranslationAuditor

CSHARP

```
namespace Pennington.TranslationAudit;

/// IBuildAuditor that pairs each default-locale page with its translations in every other
configured locale and classifies the pair as Up-to-date, Outdated (translation predates
source's last commit) or Missing (no translation file).
public class TranslationAuditor
{
    /// Runs the auditor against context and returns its diagnostics.
    public Task<IReadOnlyList<BuildDiagnostic>> AuditAsync(BuildAuditContext context,
CancellationToken cancellationToken)
;

    /// Stable identifier surfaced on every diagnostic this auditor emits.
    public string Code { get; }

    /// Wires the auditor to its options and git history reader.
    public TranslationAuditor(TranslationAuditOptions options, IGitHistoryReader git)
;
}
```

Pennington.TreeSitter.Fragments.FragmentOptions

Pennington.TreeSitter.Fragments.FragmentOptions

CSHARP

```csharp
namespace Pennington.TreeSitter.Fragments;

/// Per-reference extraction flags parsed from a :symbol info-string.
public record FragmentOptions
{
    /// Emit only the declaration's body, stripping the signature and enclosing braces.
    public bool BodyOnly { get; set; }

    /// The default: full declaration text, no imports, no elision.
    public static FragmentOptions Default { get; }

    /// Prepend the file's top-of-file import/using/require statements to the fragment.
    public bool IncludeImports { get; set; }

    /// Render the node with member bodies replaced by an elision marker (outline view).
    public bool SignaturesOnly { get; set; }

}
```

Pennington.TreeSitter.Fragments.FragmentResult

Pennington.TreeSitter.Fragments.FragmentResult

CSHARP

```
namespace Pennington.TreeSitter.Fragments;

/// Outcome of a fragment lookup: either the extracted Text or an Error message.
public record FragmentResult
{
    /// The failure message when unsuccessful; otherwise null.
    public string Error { get; set; }

    /// Creates a failed result with the given error message.
    public static FragmentResult Fail(string error)
;

    /// Outcome of a fragment lookup: either the extracted Text or an Error message.
    public FragmentResult(string Text, string Error)
;

    /// Creates a successful result wrapping text.
    public static FragmentResult Ok(string text)
;

    /// True when extraction succeeded.
    public bool Succeeded { get; }

    /// The extracted source text when successful; otherwise null.
    public string Text { get; set; }

}
```

Pennington.TreeSitter.Fragments.ISourceFragmentService

Pennington.TreeSitter.Fragments.ISourceFragmentService

CSHARP

```
namespace Pennington.TreeSitter.Fragments;

/// Resolves a source file plus an optional member name path to its source text using tree-sitter.
public interface ISourceFragmentService
{
    /// Returns the source text of namePath within relativeFilePath, or a failure. An empty namePath returns the whole file. options select body-only extraction, an elided-body outline, and whether the file's imports are prepended.
    public FragmentResult GetFragment(string languageId, string relativeFilePath, string namePath, FragmentOptions options)
;
}
```

Pennington.TreeSitter.Preprocessing.TreeSitterCodeBlockPreprocessor

Pennington.TreeSitter.Preprocessing.TreeSitterCodeBlockPreprocessor

CSHARP

```
namespace Pennington.TreeSitter.Preprocessing;

/// Preprocesses code blocks with a :symbol modifier (e.g. python:symbol) by extracting the referenced source via tree-sitter and rendering it with the shared highlighting service. A :symbol-diff variant emits a unified diff between two referenced fragments.
public class TreeSitterCodeBlockPreprocessor
{
    /// Priority — higher runs first.
    public int Priority { get; }

    /// Creates the preprocessor wired to the fragment service, shared highlighting service, and per-request diagnostics accessor.
    public TreeSitterCodeBlockPreprocessor(ISourceFragmentService fragmentService, HighlightingService highlightingService, IHttpContextAccessor httpContextAccessor)
;

    /// Attempts to preprocess a code block. Returns a result if handled, or null to pass through.
    public CodeBlockPreprocessResult TryProcess(string code, string languageId)
;
}
```

Pennington.TreeSitter.Resolution.LanguageDeclarationConfig

Pennington.TreeSitter.Resolution.LanguageDeclarationConfig

CSHARP

```
namespace Pennington.TreeSitter.Resolution;

/// Describes, for one language, which syntax-tree node types are named declarations and how to read a declaration's name, so a single generic walker can resolve a dotted member path across languages.
public record LanguageDeclarationConfig
{
    /// Field name holding a declaration's body, used for body-only extraction. Defaults to body.
    public string BodyFieldName { get; set; }

    /// Node types that count as named declarations (e.g. class_declaration, method_declaration).
    public IReadOnlySet<string> DeclarationNodeTypes { get; set; }

    /// Field name holding a declaration's name node. Defaults to name.
    public string DefaultNameField { get; set; }

    /// File-level node types that count as import/using/require statements, used to prepend a snippet's import context. Empty when the language has no syntactic import form the walker can collect.
    public IReadOnlySet<string> ImportNodeTypes { get; set; }

    /// Returns the field name to read for the given declaration node type's name.
    public string NameFieldFor(string nodeType)
;

    /// Per-node-type overrides for the name field (e.g. Rust impl_item is identified by its type field).
    public IReadOnlyDictionary<string, string> NameFieldOverrides { get; set; }

    /// Structural wrapper node types the walker descends through transparently (e.g. C# declaration_list, Python block), so name matching is not tripped up by grammar container nodes.
    public IReadOnlySet<string> TransparentNodeTypes { get; set; }

    /// Language identifier passed to the tree-sitter binding (e.g. C#, Python, Rust).
    public string TreeSitterLanguageName { get; set; }

}
```

Pennington.TreeSitter.Resolution.LanguageDeclarationConfigDefaults

Pennington.TreeSitter.Resolution.LanguageDeclarationConfigDefaults

CSHARP

```
namespace Pennington.TreeSitter.Resolution;

/// Built-in LanguageDeclarationConfig definitions seeded into TreeSitterOptions.
public class LanguageDeclarationConfigDefaults
{
    /// Creates the default fence-language to declaration-config map, keyed case-
insensitively and including common aliases (e.g. cs, py, ts, rs).
    public static Dictionary<string, LanguageDeclarationConfig> CreateDefaults()
;
}
```

Pennington.TreeSitter.Resolution.NamePathResolver

Pennington.TreeSitter.Resolution.NamePathResolver

CSHARP

```
namespace Pennington.TreeSitter.Resolution;

/// Resolves a dotted member path to a declaration node via a generic, config-driven tree
descent.
public class NamePathResolver
{
    /// Resolves segments (e.g. ["Calculator", "add"]) to a declaration node under root, or
null if no path matches. Each segment must name a declaration nested (through transparent
wrapper nodes) within a node matched by the previous segment.
    public Node Resolve(Node root, IReadOnlyList<string> segments, LanguageDeclarationConfig
config)
;
}
```

Pennington.TreeSitter.TreeSitterExtensions

Pennington.TreeSitter.TreeSitterExtensions

CSHARP

```
namespace Pennington.TreeSitter;

/// Dependency injection extensions for registering the Pennington tree-sitter integration.
public class TreeSitterExtensions
{
    /// Adds tree-sitter based multi-language code-fragment extraction — the :symbol fence
modifier. Services are registered only when ContentRoot is configured.
    public static IServiceCollection AddTreeSitter(IServiceCollection services,
Action<TreeSitterOptions> configure = null)
;
}
```

Pennington.TreeSitter.TreeSitterOptions

Pennington.TreeSitter.TreeSitterOptions

CSHARP

```
namespace Pennington.TreeSitter;

/// Configuration for the Pennington tree-sitter integration.
public class TreeSitterOptions
{
    /// Base directory that :symbol file references resolve against. When null or empty, AddTreeSitter registers only these options and no services.
    public string ContentRoot { get; set; }

    /// Maps a fence base-language id (e.g. python, cs) to the declaration config used to resolve member name paths. Seeded with built-in defaults; callers may add or replace entries.
    public Dictionary<string, LanguageDeclarationConfig> LanguageConfigs { get; }

    /// Returns the declaration config registered for languageId, or null when none is configured.
    public LanguageDeclarationConfig ResolveConfig(string languageId)
;

    /// File globs watched (recursively) under ContentRoot for live-reload. Defaults to the source extensions of the built-in languages plus common markup/data formats (HTML, CSS, JSON, Razor) that are embedded whole-file — those need no LanguageConfigs entry because a bare :symbol reference returns the file as-is. Watching by extension keeps most build output from triggering reloads; note *.json and *.cs can still match files under bin//obj/, so prefer a focused content root (a dedicated snippets folder) when pointing at a source tree. Add or remove globs to customize.
    public ISet<string> WatchFilePatterns { get; }

}
```

Pennington.UI.Components.Navigation.Slots

Pennington.UI.Components.Navigation.Slots

CSHARP

```
namespace Pennington.UI.Components.Navigation;

/// Base classes per rendered slot, before any per-instance merge.
public struct Slots
{
    /// A child link; also carries the data-[current=true] state styling.
    public string Link { get; set; }

    /// The outer <ul> wrapping the whole tree.
    public string List { get; set; }

    /// The per-section <li>.
    public string Section { get; set; }

    /// The nested <ul> holding a section's child entries.
    public string SectionList { get; set; }

    /// The section label — the <div> for empty-route entries, or the <a> when a top-level
entry has children.
    public string SectionTitle { get; set; }

    /// Base classes per rendered slot, before any per-instance merge.
    public Slots(string List, string Section, string SectionTitle, string SectionList,
string Link, string TopLink)
;

    /// A top-level leaf link; also carries the data-[current=true] state styling.
    public string TopLink { get; set; }

}
```

Pennington.UI.Components.Navigation.TocVariant

Pennington.UI.Components.Navigation.TocVariant

CSHARP

```
namespace Pennington.UI.Components.Navigation;

/// Visual archetype for TableOfContentsNavigation. A variant names a cohesive look; per-instance *Class parameters Tailwind-merge on top of it for one-off tweaks. The class strings live in For — inline literals in a method body, so an edit hot-reloads under dotnet watch (unlike a static dictionary, whose initializer only runs once).
public enum TocVariant
{
    /// Rounded pill buttons with a tinted active background — the DocSite sidebar look.
    public static const TocVariant Pill
;

    /// Bordered rail: a left border with the active child marked by a colored edge. The bare Pennington.UI default.
    public static const TocVariant Rail
;
}
```

Pennington.UI.Components.Navigation.TocVariantStyles

Pennington.UI.Components.Navigation.TocVariantStyles

CSHARP

```
namespace Pennington.UI.Components.Navigation;

/// Per-variant base class strings for each slot TableOfContentsNavigation renders. One For call returns every slot for a variant; the component merges any per-instance *Class parameter over these.
public class TocVariantStyles
{
    /// Resolves the slot bases for variant.
    public static Slots For(TocVariant variant)
;
}
```

Pennington.UI.MarkupContent

Pennington.UI.MarkupContent

CSHARP

```
namespace Pennington.UI;

/// A chrome content value that is either a raw HTML/markup string or a RenderFragment.
Implicitly converts from both, so consumers assign a string or a fragment directly; render
it via Content. Strings are emitted as raw markup, not markdown-processed.
public struct MarkupContent
{
    /// The fragment that emits this content; a no-op when default-constructed.
    public RenderFragment Content { get; }

    /// Wraps a raw HTML/markup string.
    public MarkupContent(string html)
;

    /// Wraps a RenderFragment.
    public MarkupContent(RenderFragment fragment)
;
}
```

Pennington.UI.Styling.ClassMerge

Pennington.UI.Styling.ClassMerge

CSHARP

```
namespace Pennington.UI.Styling;

/// Tailwind-aware class merge a component uses to fold a per-instance *Class parameter over
its variant base, so a passed utility knocks out the conflicting base utility instead of
duplicating it. Site templates register one built from
MonorailCssService.CreateClassMerger(...); bare hosts leave it unregistered and the
component falls back to appending (conflicting base utilities are not removed, but the
passed classes are at least present).
public class ClassMerge
{
    /// Folds extra over baseClasses. Returns baseClasses unchanged when extra is null or
empty.
    public string Apply(string baseClasses, string extra)
;

    /// Creates a merger over the given conflict-aware delegate, or an appending fallback
when none is supplied.
    public ClassMerge(Func<string, string, string> merge = null)
;
}
```