Pennington

Getting Started

Version 1.0.0

Version 1.0.0
Generated June 2026
https://usepennington.net/
Produced with Pennington 0.1.2

Contents

CHAPTER 1

Getting Started

Create your first Pennington site

Getting Started Stand up a minimal ASP.NET host that serves a single markdown page through the Pennington content pipeline.

By the end of this tutorial a runnable ASP.NET project — `MyFirstPenningtonSite` — serves `Content/index.md` as HTML at `http://localhost:5000/`, with the front-matter `title` appearing in both the `<title>` tag and the page's `<h1>`. The next tutorial swaps the bare `MapGet` for a Blazor Server catch-all.

Prerequisites

Pennington's published packages target .NET 10, so the stable .NET 10 SDK is all you need to build a site — no preview language flag. The .NET 11 beta SDK is an opt-in that only matters when you extend the pipeline yourself; the SDK and the union shim explains when that pays off.

- .NET 10 SDK installed
- A terminal and a text editor or IDE

The finished code for this tutorial lives in examples/GettingStartedMinimalSiteExample.

---

1. Scaffold a bare ASP.NET host

First, let's create the project shell Pennington will plug into — no Pennington code yet, a plain web app that returns a string, so the changes in step 2 stand out.

Create the web project

Run these two commands in a working folder. The `web` template produces a minimal top-level-statement `Program.cs` — no MVC, no Razor Pages — which is the starting shape we'll edit in the steps ahead.

BASH

```
dotnet new web -n MyFirstPenningtonSite
cd MyFirstPenningtonSite
```

Add the Pennington package reference

Add the Pennington package so the `AddPennington` extension method resolves. The backing example in this repo uses a `ProjectReference`, but for a new project this one command is enough.

BASH

```
dotnet add package Pennington
```

> **Important**
>
> Pennington is in alpha — check NuGet for the current prerelease and pin every `Pennington.*` package to that same version.

Run the bare host

The `dotnet new web` template produces a `Program.cs` with a single `MapGet` returning `"Hello World!"`. Change that string to `"Hello from ASP.NET."` — a value the template never writes, so seeing it in the browser proves you're running your own edited code and not a cached default — then run the host to confirm the shell works before Pennington takes over.

BASH

```
dotnet run --urls http://localhost:5000
```

> **Checkpoint**
>
> - `http://localhost:5000/` returns the literal text `Hello from ASP.NET.`
> - Stop the process with `Ctrl+C` before continuing.

---

2. Register Pennington and point it at markdown

Now let's swap the pass-through string endpoint for the Pennington content pipeline: `AddPennington` registers the core services, `AddMarkdownContent<DocFrontMatter>` names the markdown folder (see DocFrontMatter), and the host gains a `ContentRootPath` it will watch for changes.

Create the Content folder and an index page

Create a `Content/` folder beside `Program.cs`, then add `index.md` with the contents below. Two things are required: a YAML front-matter block with a `title:` key, and a markdown body.

MARKDOWN

```
---
title: Welcome to your first Pennington site
description: The smallest Pennington host that renders a markdown page with front matter.
---

This page is a single markdown file in `Content/index.md`. Its `title` in the
front matter above is what the host reads out when it renders the page.

## What just happened

1. `AddPennington` registered the content pipeline.
2. `AddMarkdownContent<DocFrontMatter>` pointed Pennington at this folder.
3. `UsePennington` wired the middleware into the request pipeline.
4. A tiny `MapGet` endpoint walks the content service, renders this file, and
   returns the HTML.

Everything else you see in later tutorials builds on top of these four moves.
```

Wire `AddPennington` in `Program.cs`

Replace the body of `Program.cs` with the service-registration block below, which walks through `WebApplication.CreateBuilder` → `AddPennington` → `AddMarkdownContent<DocFrontMatter>` → `app.Build()`. The two `using` directives at the top bring in `DocFrontMatter` and the `AddPennington` extension — keep them, or the file won't compile.

CSHARP

```csharp
using Pennington.FrontMatter;
using Pennington.Infrastructure;

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();

await app.RunAsync();
```

`ContentRootPath` sets the host's base for static files; the `ContentPath` passed to `AddMarkdownContent` is where this particular markdown source reads from — both point at `"Content"` here.

Don't run it yet — the services are registered but nothing serves them. The middleware and the rendering endpoint go in next, and you'll run the finished host then.

3. Wire the middleware and render the page

Now we mount the middleware chain with `app.UsePennington()`, add a `MapGet` that hands each request to `IPageResolver`, and hand control to RunOrBuildAsync — which uses the same app for development and static output with no code change. `IPageResolver` is the one service you need to turn a URL into a rendered page: it walks the registered content sources, parses the markdown that matches, and renders it. A Razor page would normally call it, but a `MapGet` keeps the wiring visible in one place for this tutorial.

Add `UsePennington`, `RunOrBuildAsync`, and the rendering endpoint

Update `Program.cs` to match the complete file below. `UsePennington` installs static files, the response-processing middleware, live reload, and auto-registered endpoints like `/sitemap.xml`; `RunOrBuildAsync` serves live when called with no args and generates static HTML when passed `--build`; the `MapGet` asks `IPageResolver` to resolve the request to a rendered page, then returns its HTML (or a 404 when nothing matches).

CSHARP

```
using Pennington.Content;
using Pennington.FrontMatter;
using Pennington.Infrastructure;
using Pennington.Routing;

var builder = WebApplication.CreateBuilder(args);

// 1. Register the Pennington content pipeline. Point ContentRootPath at the
//    folder of markdown files and declare one markdown source.
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();

// 2. Wire the Pennington middleware (static files for Content/, live-reload,
//    response processing, and auto-registered endpoints like /sitemap.xml).
app.UsePennington();

// 3. Serve any URL by asking IPageResolver to find the matching markdown file,
//    parse it, and render it through the pipeline. The resolver collapses the
//    discover->parse->render loop into one call; this host only decides what to
//    do with the result. In later tutorials the DocSite template provides its
//    own Razor layout and routing.
app.MapGet("/{*path}", async (string? path, IPageResolver resolver) =>
{
    var requested = new UrlPath(path ?? string.Empty).EnsureLeadingSlash();

    if (await resolver.ResolveAsync(requested) is not { } page)
    {
        return Results.NotFound();
    }

    var html = $"""
        <!DOCTYPE html>
        <html lang="en">
        <head>
          <meta charset="utf-8" />
          <title>{page.Metadata.Title}</title>
        </head>
        <body>
          <article>
            <h1>{page.Metadata.Title}</h1>
```

```
        </html>        """;    return Results.Content(html, "text/html");});  // 4. Dev mode (`dotnet run`) serves live; build mode//    (`dotnet run -- build <baseUrl> <outputDir>`) crawls the running app and//    writes static HTML. Both args are optional; defaults are `/` and `output`.await app.RunOrBuildAsync(args);
```

This `MapGet` is deliberately minimal. `IPageResolver` collapses the discover → parse → render flow into one call; if you want to see the four-stage union pipeline it runs underneath, read The content pipeline and union types. The next tutorial replaces this `MapGet` with a Blazor Server `@page` catch-all, the form a real Pennington app stays in.

Checkpoint

That's the working site. `dotnet run --urls http://localhost:5000` serves live, and `http://localhost:5000/` returns HTML whose `<title>` element and top-level `<h1>` both read `Welcome to your first Pennington site`, pulled straight from `Content/index.md`'s front matter.

- Run `dotnet run --urls http://localhost:5000` from the project folder.
- Open `http://localhost:5000/` and confirm the page title in the browser tab reads `Welcome to your first Pennington site`.
- View source and confirm the same string appears inside the `<title>` tag and the article's `<h1>`.

The rendered page is plain unstyled HTML — Times-New-Roman serif, default browser margins, blue underlined links. That is on purpose: this host wires only the content pipeline, not the CSS layer. Replacing the bare `MapGet` with a Blazor Server `@page` catch-all is the next tutorial: Serve markdown through Blazor Pages.

---

4. Verify dev-mode hot reload

Let's confirm that `UsePennington`'s file-watcher and live-reload WebSocket are working: with `dotnet run --urls http://localhost:5000` still serving, edit the markdown file and watch the browser reload without touching the terminal.

Edit the front-matter title

Leave `dotnet run --urls http://localhost:5000` serving and keep `http://localhost:5000/` open in the browser. Open `Content/index.md` and change the `title:` value to something recognizable — for example `title: Hello, Pennington` — then save. The browser tab updates on its own within a second. If it doesn't, hard-refresh once; stale HTML may be cached from before the edit.

Checkpoint

Without any terminal input, the browser tab updates to show the new title in both the `<h1>` and the tab title. The running console logs a file-change line naming `Content/index.md`.

- Edit `Content/index.md`'s `title:` field and save.
- The browser tab title and page heading update to match — no manual refresh needed.
- The terminal logs the change.

Note

Wiring the host yourself is the normal path, and the next tutorials build straight on it. If your site is a plain documentation site, the DocSite template pre-wires this same shape — sidebar, search, Razor catch-all — in one `AddDocSite` call; Scaffold a documentation site with DocSite picks up there.

Summary

- An ASP.NET host now serves a markdown page end-to-end through `AddPennington` and `UsePennington`.
- The content pipeline reads from a folder of markdown through `ContentRootPath` plus `AddMarkdownContent<DocFrontMatter>`.
- `RunOrBuildAsync` means the same host generates a static site on `dotnet run -- build` with no code change.
- A front-matter `title:` flows from YAML into the rendered `<h1>`, and dev-mode hot reload re-renders on save.

Serve markdown through Blazor Pages

Getting Started Stand up a Pennington site whose markdown is served through a Blazor Server `@page` catch-all — the natural shape for a real app.

By the end of this tutorial a runnable ASP.NET project — `MyBlazorPenningtonSite` — serves markdown from `Content/` through a Blazor Server `@page "/{*Path}"` catch-all at `http://localhost:5000/`. The previous tutorial used a hand-rolled `MapGet`; this one swaps it for the production-shape Blazor catch-all a real app stays in.

Prerequisites

- .NET 10 SDK installed
- Create your first Pennington site — this tutorial builds on its `IPageResolver` walkthrough. It does repeat the `dotnet new web` + Pennington package bootstrap from scratch, so you can also start here cold if you prefer.

The finished code for this tutorial lives in examples/GettingStartedBlazorPagesExample. The DocSite template pre-wires this same Blazor shape for documentation sites — see Scaffold a documentation site with DocSite if that is exactly what you are building.

---

1. Set up the project shell

Start from an empty ASP.NET web project and add the Pennington package. No Pennington code yet — the shell `Program.cs` stays untouched until section 2.

Create the web project

Run these two commands in a working folder. The `web` template produces a minimal top-level-statement `Program.cs` that returns `Hello World!` — the starting shape we'll replace in the next section.

BASH

```
dotnet new web -n MyBlazorPenningtonSite
cd MyBlazorPenningtonSite
```

Add the Pennington package

Add the Pennington package so the `AddPennington` extension method resolves. The command writes the `<PackageReference>` into the project file:

BASH

```
dotnet add package Pennington
```

XML

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net10.0</TargetFramework>
    <Nullable>enable</Nullable>
    <ImplicitUsings>enable</ImplicitUsings>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Pennington" Version="0.1.2" />
  </ItemGroup>
</Project>
```

> **Important**
>
> Pennington is in alpha — check NuGet for the current prerelease and pin every `Pennington.*` package to that same version.

Create `Content/index.md`

Create a `Content/` folder beside `Program.cs` and add `index.md`. The catch-all you wire up in the next section serves anything under `Content/` — this is the file `/` will resolve to.

MARKDOWN

```
---
title: Welcome
description: The home page of the Blazor-pages tutorial site.
---

This page is `Content/index.md`. The browser asked for `/`; the Blazor catch-all
in `Components/Pages/MarkdownPage.razor` matched, walked the configured
`IContentService` instances to find this file, ran it through the parser and
renderer, and dropped the rendered HTML into the page's `<article>` element.

Add a second markdown file under `Content/` and its file path becomes its URL —
no router-table edit required.
```

Checkpoint

- `dotnet build` succeeds with no errors
- `dotnet run --urls http://localhost:5000` followed by visiting `http://localhost:5000/` returns the literal text `Hello World!` — the bare web template's response. Pennington takes over in the next section
- Stop the process with `Ctrl+C` before continuing

---

2. Wire Pennington, Blazor, and the markdown page

Replace the `Program.cs` body with the host below, then add three Razor files: a `_Imports.razor` for shared `@using` lines, an `App.razor` root component that owns the document shell, and a `MarkdownPage.razor` catch-all that renders any URL to a markdown file. Two service registrations (`AddPennington` for the content pipeline, `AddRazorComponents` for Blazor SSR) and three middleware calls (`UsePennington`, `UseAntiforgery`, `MapRazorComponents<App>()`) are all the host needs.

Important

`app.UsePennington()` must run before `app.MapRazorComponents<App>()`. The Blazor catch-all `@page "/{*Path}"` would otherwise swallow Pennington's redirect, sitemap, and llms.txt routes.

Note

The embeds come from the finished example, so they use its root namespace `GettingStartedBlazorPagesExample` (in `Program.cs` and `_Imports.razor`). Swap in your own — here, `MyBlazorPenningtonSite`.

Replace `Program.cs`

CSHARP

```csharp
using GettingStartedBlazorPagesExample.Components;
using Pennington.FrontMatter;
using Pennington.Infrastructure;

var builder = WebApplication.CreateBuilder(args);

// 1. Same Pennington wiring as the minimal-site tutorial: register the content
//    pipeline and point one markdown source at Content/.
builder.Services.AddPennington(penn =>
{
    penn.SiteTitle = "My First Pennington Site";
    penn.ContentRootPath = "Content";

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

// 2. Add Blazor Server's static-rendering services. This is what unlocks
//    `MapRazorComponents<App>()` below.
builder.Services.AddRazorComponents();

var app = builder.Build();

// 3. Order matters: UsePennington registers redirect routes, llms.txt, and
//    sitemap endpoints. The Blazor catch-all `@page "/{*Path}"` would swallow
//    those routes if MapRazorComponents ran first.
app.UsePennington();

// 4. Antiforgery middleware is required by MapRazorComponents — Blazor's
//    routed components opt into the [RequireAntiforgeryToken] metadata even
//    when no form ships in the page.
app.UseAntiforgery();

// 5. Hand routing to Blazor. Components/App.razor's <Router> finds the
//    matching @page component (in this project: Components/Pages/MarkdownPage.razor).
app.MapRazorComponents<App>();

await app.RunOrBuildAsync(args);
```

Add `_Imports.razor` at the project root

`_Imports.razor` provides the `@using` set every `.razor` file in the project sees.

RAZOR

```
@using Microsoft.AspNetCore.Components
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Pennington.Content
@using Pennington.Pipeline
@using Pennington.Routing
@using GettingStartedBlazorPagesExample.Components
```

Add `Components/App.razor`

`App.razor` is the root component `MapRazorComponents<App>()` mounts. It owns the entire HTML document — `<!DOCTYPE>`, `<html>`, `<head>` (with `<HeadOutlet>` so each routed page's `<PageTitle>` flows in), and `<body>`. The `<Router>` inside `<body>` scans the assembly for `@page` components and routes each request to the matching one.

RAZOR

```
@* Root component. Owns the entire HTML document — no MainLayout in this
   tutorial. The Router scans this assembly for [@page] components and routes
   each request to the matching one. <PageTitle> from each routed page flows
   into <head> via <HeadOutlet>. *@

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <HeadOutlet />
</head>
<body>
    <Router AppAssembly="@typeof(App).Assembly">
        <Found Context="routeData">
            <RouteView RouteData="@routeData" />
        </Found>
        <NotFound>
            <p>Not found.</p>
        </NotFound>
    </Router>
</body>
</html>
```

Add `Components/Pages/MarkdownPage.razor`

`MarkdownPage.razor` is the `@page "/{*Path}"` catch-all. Blazor binds the request path to the `Path` parameter; the component asks `IPageResolver` to resolve that URL to a rendered page and injects the HTML via `(MarkupString)`. It's the same `IPageResolver` the `MapGet` host used in the previous tutorial — only the call site has moved into a component.

RAZOR

```
@* Catch-all that matches every URL and asks IPageResolver to resolve it to a
   markdown file. It's the same IPageResolver the MapGet host injected in the
   previous tutorial — the only thing that's changed is where it's called from. *@

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

@if (_html is not null)
{
    <PageTitle>@_title</PageTitle>
    <article>
        <h1>@_title</h1>
        @((MarkupString)_html)
    </article>
}
else
{
    <PageTitle>Not found</PageTitle>
    <p>No content matches @Path.</p>
}

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

    private string? _title;
    private string? _html;

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

        if (await Resolver.ResolveAsync(requested) is { } page)
        {
            _title = page.Metadata.Title;
            _html = page.Content.Html;
        }
    }
}
```

Checkpoint

- `dotnet run --urls http://localhost:5000` and visit `http://localhost:5000/` — the page renders `Content/index.md`.
- View source. The `<title>` and `<h1>` both pull from `index.md`'s front-matter `title:`.

---

3. Add a second markdown file

The file-path-to-URL convention is unchanged by routing through Blazor. Pennington's file watcher picks up new files in `Content/` while the host runs — no restart, no router-table edit.

Add `Content/about.md`

Leave `dotnet run` going from the previous section and drop this file in.

MARKDOWN

```
---
title: About
description: Proves that adding a markdown file is enough to expose a new URL.
---

This file is `Content/about.md` and the catch-all serves it at `/about`. The
Blazor router didn't gain a new entry — `MarkdownPage.razor` matches every URL
through `@page "/{*Path}"` and asks the content pipeline whether anything on
disk corresponds to the requested path.

Rename this file to `reach-out.md` and `/reach-out` works on the next request.
The only thing routing the URL is the file's name.
```

Navigate to `/about`

Open `http://localhost:5000/about` in the browser. The catch-all serves the new file on the first request — no restart needed.

Checkpoint

- Visit `/about` — the page renders, served through the same catch-all as `/`.

Summary

- A Pennington host plus a Blazor Server router is two service registrations (`AddPennington`, `AddRazorComponents`) and three middleware calls (`UsePennington`, `UseAntiforgery`, `MapRazorComponents<App>()`).
- `app.UsePennington()` must run before `app.MapRazorComponents<App>()` — the catch-all would otherwise swallow Pennington's redirect, sitemap, and llms.txt routes.
- A single `@page "/{*Path}"` component (`MarkdownPage.razor`) handles every URL, resolves it through `IPageResolver`, and injects the rendered HTML via `(MarkupString)`.
- The file-path-to-URL convention from the markdown pipeline still holds — adding or renaming a `.md` file under `Content/` is enough.

Style the site with MonorailCSS

Getting Started Layer MonorailCSS onto the Blazor-pages site through a routed `MainLayout.razor` and watch the stylesheet regenerate as new utility classes appear in the source.

By the end of this tutorial the Blazor-pages site from Serve markdown through a Blazor catch-all is styled with MonorailCSS — a Tailwind-compatible JIT compiler in pure .NET. Every routed `@page` renders through a `MainLayout.razor` that carries the utility classes. The MonorailCSS Discovery pipeline turns those classes into real CSS rules, served at `/styles.css`. The stylesheet regenerates whenever a new class appears in the source.

Prerequisites

- .NET 10 SDK installed
- Completed Serve markdown through a Blazor catch-all (or a Pennington project with `MapRazorComponents<App>()` wired and a catch-all `MarkdownPage.razor`)

The finished code for this tutorial lives in examples/GettingStartedStylingExample. For a documentation site, the DocSite template ships this MonorailCSS-plus-`MainLayout` stack with a sidebar, search, and theme toggle already assembled — Scaffold a documentation site with DocSite covers it.

---

1. Wrap pages in a styled `MainLayout.razor`

Before MonorailCSS can do anything, the layout needs to carry the utility classes that will turn into CSS rules. The document shell moves out of `App.razor` and into a `MainLayout.razor` that holds those classes; `App.razor` shrinks to a bare router that wraps every routed page in the new layout.

Create `Components/Layout/MainLayout.razor`

Drop this file at `Components/Layout/MainLayout.razor`. Inheriting `LayoutComponentBase` makes it a Blazor layout — every routed page renders into the `@Body` placeholder. This component now owns the whole document shell — `<!DOCTYPE>`, `<html>`, `<head>` (with `<HeadOutlet>`), and `<body>` — moved here from `App.razor`. The `<link rel="stylesheet" href="/styles.css">` tag points at an endpoint section 2 will mount.

RAZOR

```
@* Styled shell. Lives once and wraps every routed @page via App.razor's
   DefaultLayout. The class strings (text-primary-700, bg-base-50, …) become
   literal IL strings after Razor compiles, and Discovery's startup IL scan
   picks them up to populate the class registry behind /styles.css. *@

@inherits LayoutComponentBase

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="stylesheet" href="/styles.css" />
    <HeadOutlet />
</head>
<body class="bg-base-50 text-base-900 min-h-screen">
    <div class="max-w-3xl mx-auto px-6 py-10">
        <header class="mb-8 border-b border-base-200 pb-4">
            <a class="text-lg font-bold text-primary-700" href="/">My Styled Pennington
Site</a>
        </header>
        <article class="prose">
            @Body
        </article>
        <footer class="mt-12 pt-4 border-t border-base-200 text-xs text-base-500">
            Styled with MonorailCSS.
        </footer>
    </div>
</body>
</html>
```

The classes — `bg-base-50`, `text-primary-700`, `border-base-200`, and so on — come from the named color palette configured in the next section.

Reference the layout namespace from `_Imports.razor`

`App.razor` refers to `MainLayout` by its bare name, so the project's `_Imports.razor` needs an `@using` for the layout's namespace — without it `typeof(MainLayout)` fails to compile. Add the `Components.Layout` line to the `_Imports.razor` at the project root:

RAZOR

```
@using Microsoft.AspNetCore.Components
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Pennington.Content
@using Pennington.Pipeline
@using Pennington.Routing
@using GettingStartedStylingExample.Components
@using GettingStartedStylingExample.Components.Layout
```

> **Note**
>
> The embeds use the finished example's root namespace, `GettingStartedStylingExample` (in `_Imports.razor` and the section 2 `Program.cs` snippet). Swap in your own.

Replace `Components/App.razor`

With the shell now in `MainLayout.razor`, `App.razor` is replaced wholesale: it drops the `<!DOCTYPE>`, `<html>`, `<head>`, and `<HeadOutlet>` it used to own and becomes just the `<Router>`. `RouteView` names `MainLayout` as the `DefaultLayout` for every matched page, and the `LayoutView` for the not-found case lets the same shell wrap the 404 message.

RAZOR

```
@* Root component. The Router scans this assembly for [@page] components and
   wraps each match in MainLayout (the styled shell). <PageTitle> from each
   routed page flows into <head> via <HeadOutlet>. *@

<Router AppAssembly="@typeof(App).Assembly">
    <Found Context="routeData">
        <RouteView RouteData="@routeData" DefaultLayout="@typeof(MainLayout)" />
    </Found>
    <NotFound>
        <LayoutView Layout="@typeof(MainLayout)">
            <p>Not found.</p>
        </LayoutView>
    </NotFound>
</Router>
```

2. Register MonorailCSS and mount `/styles.css`

MonorailCSS ships in its own package, separate from the core `Pennington` package the previous tutorial added. Pull it in:

BASH

```
dotnet add package Pennington.MonorailCss
```

With the package referenced, wire MonorailCSS into the service container, pick a color scheme, and mount the JIT stylesheet endpoint. `AddMonorailCss` registers the services; each of `PrimaryColorName`, `AccentColorName`, and `BaseColorName` takes a `ColorName` constant (indigo/pink/slate here — any combination works). `app.UseMonorailCss()` mounts `/styles.css` as a real endpoint that regenerates on every request, matching the `<link>` tag in `MainLayout.razor`. Both highlighted blocks are new in this section; the `using Pennington.MonorailCss;` at the top is what makes `AddMonorailCss`, `MonorailCssOptions`, `NamedColorScheme`, and `ColorName` resolve.

CSHARP

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

var builder = WebApplication.CreateBuilder(args);

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

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

// New in this stage: register MonorailCSS. Pick which named palettes
// back the `primary`, `accent`, and `base` utility prefixes. Any
// ColorName constant works — swap freely.
builder.Services.AddMonorailCss(_ => new MonorailCssOptions
{
    ColorScheme = new NamedColorScheme
    {
        PrimaryColorName = ColorName.Indigo,
        AccentColorName = ColorName.Pink,
        BaseColorName = ColorName.Slate,
    },
});

builder.Services.AddRazorComponents();

var app = builder.Build();

app.UsePennington();

// New in this stage: mount /styles.css. The default path matches the
// <link> tag in MainLayout.razor.
app.UseMonorailCss();

app.UseAntiforgery();
app.MapRazorComponents<App>();

await app.RunOrBuildAsync(args);
```

Checkpoint

- Run `dotnet run --urls http://localhost:5000` and visit `http://localhost:5000/` — the header, article, and footer now render with indigo accents, slate neutrals, and the layout spacing the utility classes describe
- Visit `http://localhost:5000/styles.css` directly and a populated stylesheet appears, containing rules for every utility class the layout emits

---

3. Watch the stylesheet regenerate

Under `dotnet run`, MonorailCSS rescans your project for new utility classes on the next `/styles.css` request, so classes you add in source (Razor components and other compiled C#) appear without a restart. Markdown bodies are out of scope: a utility token added to a `.md` file will not produce a CSS rule. The MonorailCSS integration explanation covers why.

Add a new utility class to `MainLayout.razor`

Open `Components/Layout/MainLayout.razor` and wrap the footer's "MonorailCSS" word in an accented span:

RAZOR

```
<footer class="mt-12 pt-4 border-t border-base-200 text-xs text-base-500">
    Styled with <span class="text-accent-600 italic">MonorailCSS</span>.
</footer>
```

The class `text-accent-600` wasn't in the layout, so it doesn't yet exist in the stylesheet.

Reload and confirm the new rule

Reload any page in the browser. The footer's "MonorailCSS" word renders in pink italic because the `.razor` edit refreshed the class set, and the next `/styles.css` request picked up the new token. Reload `/styles.css` directly and the `text-accent-600` rule is present.

Checkpoint

- The footer's "MonorailCSS" word renders in pink italic on every page
- `http://localhost:5000/styles.css` now contains a rule for `text-accent-600` that wasn't there before the edit
- No server restart was required — the MonorailCSS file watcher refreshed the stylesheet under the running `dotnet run`

Summary

- `MainLayout.razor` (a Blazor `LayoutComponentBase`) holds the utility-class scaffold every routed `@page` renders into via `App.razor`'s `DefaultLayout`.
- `AddMonorailCss(...)` registers the service container; `UseMonorailCss()` mounts the `/styles.css` endpoint that regenerates on every request.
- A `NamedColorScheme` of three `ColorName` constants drives every `primary-*`, `accent-*`, and `base-*` utility prefix.
- Under `dotnet run`, adding a new utility class to a `.razor` or `.cs` file regenerates the stylesheet on the next request without a restart — markdown edits do not participate.

The site is styled, but every page is an island with no way to reach the next. The final getting-started tutorial adds a navigation menu that links them.

Add navigation across your pages

Getting Started Give the styled bare host a header menu that builds itself from the content pipeline and highlights the current page.

By the end of this tutorial the styled site from Style the site with MonorailCSS has a navigation menu in its header. The menu links every page on the site, builds itself from the `Content/` folder — so adding a markdown file adds a menu entry — and renders the current page in bold.

This is the last step of the getting-started arc. After it, a bare `AddPennington` host serves a complete, styled, navigable multi-page site — no template involved.

Prerequisites

- .NET 10 SDK installed
- Completed Style the site with MonorailCSS — this tutorial extends that project's `MainLayout.razor` and `Content/` folder

The finished code for this tutorial lives in examples/GettingStartedNavigationExample.

---

1. Add pages to navigate to

The styling site has a single home page. A menu needs somewhere to point, so let's add three more pages — two inside a `guides/` folder, one at the top level. A folder with no `index.md` becomes a section node in the tree, so `guides/` will render as a labeled **Guides** group with its two pages nested under it.

Create the `guides/` folder with two pages

Add `Content/guides/installation.md` and `Content/guides/deployment.md`. The `order:` front-matter key sets each page's position in its section — lower sorts first.

MARKDOWN

```
---
title: Installation
description: A page inside the guides section.
order: 10
---

This page lives at `Content/guides/installation.md`. Its URL is `/guides/installation/`,
so `NavigationBuilder` places it under a **Guides** section in the menu.

Its `order:` of `10` sorts it ahead of the Deployment page within that section.
```

MARKDOWN

```
---
title: Deployment
description: Another page inside the guides section.
order: 20
---

This page also lives under `Content/guides/`, so it joins the Installation page
under the **Guides** section. Its `order:` of `20` places it second.

Open the menu and notice that the entry for the page you are on renders bold —
`NavMenu.razor` reads the `IsSelected` flag `NavigationBuilder` stamps onto the
node matching the current URL.
```

Create a top-level page

Add `Content/about.md` directly under the content root — not in a folder — so it becomes a top-level menu entry rather than part of a section.

MARKDOWN

```
---
title: About
description: A top-level page outside any section.
order: 30
---

This page lives at `Content/about.md` — directly under the content root, not in
a folder — so it appears as a top-level menu entry rather than inside a section.

Its `order:` of `30` is the highest on the site, so it sorts last.
```

Checkpoint

- `dotnet run --urls http://localhost:5000`, then visit `http://localhost:5000/guides/installation/` and `http://localhost:5000/about/`
- Both pages render through the styled layout — but there is still no menu, so the only way to reach them is by typing the URL

2. Build the navigation menu

`AddPennington` already registers `NavigationBuilder` — the service that turns content into a navigation tree — so the menu needs no new wiring in `Program.cs`, only a component to render it.

Add the `Pennington.Navigation` namespace to `_Imports.razor`

`NavMenu.razor` uses types from `Pennington.Navigation`, so add that namespace to the project's `_Imports.razor`:

RAZOR

```
@using Pennington.Navigation
```

`<NavMenu />` is referenced by its short tag name in `MainLayout.razor`, which resolves only when the layout folder's namespace (`<RootNamespace>.Components.Layout`) is in scope. The styling tutorial already added that line when it moved the shell into `MainLayout.razor`, so it is in place — an unresolved component tag is a build warning, not an error, and `<NavMenu />` would silently render as raw markup without it.

Create `Components/Layout/NavMenu.razor`

This component renders the menu from the content pipeline:

RAZOR

```
@* NavMenu.razor — the site's navigation, built from the content pipeline.
   Every IContentService exposes its pages as flat table-of-contents entries.
   NavigationBuilder sorts those entries by their `order:` front matter and
   nests them by folder into a tree of NavigationTreeItem. This component
   renders that tree and marks the entry matching the current URL. *@

@inject IEnumerable<IContentService> ContentServices
@inject NavigationBuilder Navigation
@inject NavigationManager NavManager

<nav class="mt-3 flex flex-wrap items-center gap-x-5 gap-y-1 text-sm">
    @foreach (var item in _tree)
    {
        if (item.Children.Count == 0)
        {
            <a class="@LinkClass(item)"
href="@item.Route.CanonicalPath.Value">@item.Title</a>
        }
        else
        {
            @* A folder with no index page becomes a section: a label, then its pages. *@
            <span class="font-semibold text-base-400">@item.Title</span>
            foreach (var child in item.Children)
            {
                <a class="@LinkClass(child)"
href="@child.Route.CanonicalPath.Value">@child.Title</a>
            }
        }
    }
</nav>

@code {
    private IReadOnlyList<NavigationTreeItem> _tree = [];

    protected override async Task OnInitializedAsync()
    {
        // Collect the table-of-contents entries from every content source.
        var entries = await ContentServices.CollectTocEntriesAsync();

        // Passing the current URL lets NavigationBuilder return the matching
        // node with IsSelected already set.
        var currentPath = new UrlPath(NavManager.ToBaseRelativePath(NavManager.Uri))
            .EnsureLeadingSlash();

        _tree = await Navigation.BuildTreeAsync(entries, currentPath);
    }

    // The current page renders bold; the rest are muted links.
    private static string LinkClass(NavigationTreeItem item) => item.IsSelected
        ? "font-semibold text-primary-700"
        : "text-base-600 hover:text-primary-700";
}
```

`CollectTocEntriesAsync` gathers a flat list — one `ContentTocItem` per page — from every content source. `BuildTreeAsync` is what gives it shape: it sorts entries by their `order:` value and nests them by folder, so `Content/guides/` becomes a **Guides** section. Passing the current URL makes the matching node come back with `IsSelected` set. For the full picture of how the tree is folded together, see Navigation-tree construction.

3. Wire the menu into the layout

Drop `<NavMenu />` into the header of `MainLayout.razor`. Because the layout wraps every routed page, the menu then appears site-wide.

RAZOR

```
@* Styled shell from the styling tutorial, now with a navigation menu in the
   header. <NavMenu /> builds its links from the content pipeline, so adding a
   markdown file to Content/ adds a menu entry — no edit to this file. *@

@inherits LayoutComponentBase

<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="stylesheet" href="/styles.css" />
    <HeadOutlet />
</head>
<body class="bg-base-50 text-base-900 min-h-screen">
    <div class="max-w-3xl mx-auto px-6 py-10">
        <header class="mb-8 border-b border-base-200 pb-4">
            <a class="text-lg font-bold text-primary-700" href="/">My Pennington Site</a>
            <NavMenu />
        </header>
        <article class="prose">
            @Body
        </article>
        <footer class="mt-12 pt-4 border-t border-base-200 text-xs text-base-500">
            Built on a bare Pennington host.
        </footer>
    </div>
</body>
</html>
```

Checkpoint

Run `dotnet run --urls http://localhost:5000` and open `http://localhost:5000/`.

- The header shows a menu: **Welcome**, a **Guides** group containing **Installation** and **Deployment**, then **About**
- The entry for the page you are viewing renders bold — click into `/guides/deployment/` and the highlight follows
- Add a new markdown file under `Content/` and reload — a menu entry appears for it with no edit to `NavMenu.razor` or `MainLayout.razor`

Summary

- `NavigationBuilder` ships with `AddPennington`; `NavMenu.razor` is the only new code, and `Program.cs` did not change.
- `NavMenu.razor` collects each source's table-of-contents entries and `NavigationBuilder.BuildTreeAsync` turns that flat list into a sorted, folder-nested tree.
- The bare host now serves a complete site: a content pipeline, a styled layout, and navigation — all on `AddPennington`.

That is the whole getting-started arc. `AddPennington` gives you the lower-level host: you wire the pipeline, the layout, and the navigation yourself, and you have now done each part. The DocSite and BlogSite templates package this wiring for documentation and blog sites. The beyond-basics tutorials build on the host you just finished.

CHAPTER 2

Docsite

Scaffold a documentation site with DocSite

Getting Started Stand up the DocSite template on an empty ASP.NET project and let a folder of markdown light up the sidebar.

By the end of this tutorial the DocSite host runs with a "Scaffold Docs" title, GitHub icon, and header/footer chrome — and a folder of markdown pages that show up in the sidebar on their own, with a landing page at the root.

`AddDocSite` wires a Divio-style documentation site — host, layout, navigation, styling — into one call; for what the template wires, where the wiring stops, and why DocSite and BlogSite can't share an app, read what the templates wire for you first.

Prerequisites

The DocSite template ships as a NuGet package built for .NET 10, so the stable .NET 10 SDK is all you need — no preview language flag. The .NET 11 beta SDK only matters if you later extend the pipeline by hand; see the SDK and the union shim.

- .NET 10 SDK installed
- A terminal and a text editor or IDE

The finished code for this tutorial lives in examples/DocSiteScaffoldExample.

---

1. Scaffold a new ASP.NET project

Start from an empty ASP.NET web project. DocSite ships everything from routing to the Razor layout, so the project shell is the only scaffolding needed before `AddDocSite`.

Create the web project

BASH

```
dotnet new web -n DocSiteScaffold
cd DocSiteScaffold
```

Add the Pennington DocSite package

BASH

```
dotnet add package Pennington.DocSite
```

> **Important**
>
> Pennington is in alpha — check NuGet for the current prerelease and pin every `Pennington.*` package to that same version.

> **Checkpoint**
>
> - Run `dotnet build` — the empty project with the DocSite package added compiles before you touch `Program.cs`

---

2. Wire `AddDocSite`, `UseDocSite`, and `RunDocSiteAsync`

`AddDocSite` is a single DI call that registers Pennington core, MonorailCSS, SPA navigation, the content resolver, and the component that renders the page body into the layout — all driven from one options object. `UseDocSite` is its middleware counterpart; `RunDocSiteAsync` dispatches the host between dev-serve and static-build modes (see CLI and build arguments for the args contract).

Replace `Program.cs` with the three DocSite calls

`AddDocSite` takes a function that returns a fresh `DocSiteOptions`, and the template registers the markdown content reader for you — no separate `AddMarkdownContent` call. `RunDocSiteAsync` serves pages live in development and generates static HTML when invoked as `dotnet run -- build`.

CSHARP

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

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();

app.UseDocSite();

await app.RunDocSiteAsync(args);
```

The five fields populated here — `SiteTitle`, `SiteDescription`, `GitHubUrl`, `HeaderContent`, `FooterContent` — each surface in the rendered chrome as soon as they're set. `DocSiteOptions` carries many more fields; see Pennington.DocSite.DocSiteOptions for the full surface, and What the DocSite and BlogSite templates wire for you for what the template hard-codes.

Checkpoint

- Run `dotnet run` and visit `http://localhost:5000/`
- The DocSite layout renders: left sidebar, header with site title, search affordance, dark-mode toggle, GitHub icon linking to `GitHubUrl`, and the footer HTML from `FooterContent`

---

3. Add pages and watch the sidebar fill in

DocSite builds the sidebar from the shape of `Content/`. Drop a markdown file into a folder and it becomes a sidebar entry — no routing table, no registration call. A subfolder turns into a navigation group named after the folder.

Create a `guides` folder with a landing page

Under `Content/`, create a `guides/` folder and add an `index.md`. The folder name becomes a **Guides** group in the sidebar, and `index.md` is the page it links to at `/guides/`.

MARKDOWN

```
---
title: Guides
description: Task walkthroughs and onboarding.
---

Walkthroughs and how-tos live in this folder. Every markdown file under
`Content/guides/` shows up as a page in the sidebar — the folder name becomes
the **Guides** group around them.
```

Add two pages inside the folder

Add two more files next to `index.md`. Each carries a `title`, a `description`, and an `order:` — the `order:` decides which sorts first. Dropping the files in the folder is the whole wiring.

MARKDOWN

```
---
title: Getting started
description: Install the package and run the site.
order: 1
---

Every markdown file under `Content/guides/` becomes a sidebar entry — no
routing table, no registration call. This page answers to
`/guides/getting-started`.

The folder name becomes the sidebar group; `order:` decides where the page
sorts inside it.
```

MARKDOWN

```
---
title: Configuration
description: Where DocSite reads its settings.
order: 2
---

A second page beside the first. Because it carries `order: 2`, it sorts below
*Getting started* (`order: 1`) under the **Guides** group in the sidebar.
```

Checkpoint

- Run `dotnet run` and visit `http://localhost:5000/guides/getting-started`
- The sidebar shows a **Guides** group with *Getting started* above *Configuration* — sorted by `order:`
- The page you're viewing is highlighted in the sidebar; clicking *Configuration* swaps to it instantly

---

4. Give the root `/` a landing page

The pages under `guides/` answer to `/guides/...`, but the root `/` has no page of its own yet — a request to `/` returns a 404. To serve the root, drop a markdown file at `Content/index.md`, next to the `guides/` folder.

Author `Content/index.md`

Use the same DocSiteFrontMatter shape as any other page. What makes this page the root is its location — `Content/index.md` maps to `/`.

MARKDOWN

```
---
title: Welcome
description: Start here, then pick a guide.
---

Welcome to Scaffold Docs.

- [Getting started](/guides/getting-started) — install and run.
- [Configuration](/guides/configuration) — where settings live.
```

Checkpoint

- `http://localhost:5000/` returns the rendered `Content/index.md` page inside the DocSite chrome
- Both links navigate into the `guides/` pages and highlight them in the sidebar

---

Summary

- An empty ASP.NET project picked up `AddDocSite` + `UseDocSite` + `RunDocSiteAsync`, and the full Razor chrome renders.
- `DocSiteOptions` carries `SiteTitle`, `SiteDescription`, `GitHubUrl`, `HeaderContent`, and `FooterContent`, and each field appears in the rendered layout.
- Markdown files under `Content/` become sidebar entries with no extra wiring — a subfolder turns into a navigation group named after the folder, and `order:` sorts the pages inside it.
- The root `/` is served by `Content/index.md`; without it, `/` returns a 404.
- To split the sidebar into switchable areas and labeled sections, see Organize content with sections and areas. For what the template hard-codes, see What the DocSite and BlogSite templates wire for you.

Add doc pages and link between them

Getting Started Build out a Guides area with two content pages, wire sibling navigation, hub-style absolute links, and rename-safe uid cross-references.

By the end of this tutorial the Guides area has two new pages — `install.md` and `configure.md` — wired into the sidebar in `order:` sequence, cross-linked with relative paths, and reachable from a hub `index.md` that uses both absolute paths and a `uid:`-based `xref:` link.

Prerequisites

- .NET 10 SDK installed
- Completed Scaffold a DocSite (or have a DocSite project with a `Content/guides/` folder ready — this tutorial adds pages to it)

The finished code for this tutorial lives in examples/DocSitePagesAndLinksExample.

---

1. Add two content pages

Let's drop two markdown files into the Guides area and watch them slot into the sidebar. The pages stand alone for now — linking arrives in the next unit.

Create `Content/guides/install.md`

Add a new file at `Content/guides/install.md` with the markdown below. The four front-matter keys are the ones DocSiteFrontMatter reads for sidebar wiring: `title` is the link label, `description` becomes the meta tag, `sectionLabel` carries through to breadcrumbs and prev/next chrome, and `order` decides where the page sorts among siblings.

MARKDOWN

````
---
title: Install Pennington
description: Add the Pennington DocSite package and wire AddDocSite + UseDocSite into a
fresh ASP.NET host.
sectionLabel: Guides
order: 20
---

# Install Pennington

Install Pennington into an ASP.NET project with one NuGet package and three lines of DI
wiring.

## 1. Add the package

```bash
dotnet add package Pennington.DocSite
```

## 2. Wire DocSite in `Program.cs`

Three calls — register the services, mount the middleware, hand control to the host:

```csharp
builder.Services.AddDocSite(() => new DocSiteOptions { /* ... */ });
app.UseDocSite();
await app.RunDocSiteAsync(args);
```

The host is now ready for content. Drop markdown files under `Content/guides/` and they
appear in the sidebar on the next request.
````

Create `Content/guides/configure.md`

Add a second file next to the first with the markdown below. Note `order: 30` — a larger number than install's `order: 20`, so configure sorts after install in the sidebar.

MARKDOWN

```
---
title: Configure the site
description: Pick a site title, set the GitHub link, and decide on a single area or
multiple.
sectionLabel: Guides
order: 30
---

# Configure the site

`DocSiteOptions` is the one options record `AddDocSite` reads. Set the fields that surface
in the rendered chrome and the rest of the template falls into place.

## Fields worth setting first

- `SiteTitle` — appears in the header and the `<title>` tag.
- `Description` — meta description used in search snippets and social cards.
- `GitHubUrl` — surfaces the GitHub icon in the header.
- `HeaderContent` / `FooterContent` — raw HTML slots, useful for a logo and a copyright
line.

## Areas — one or many?

`Areas` is an `IReadOnlyList<ContentArea>`. One entry is enough to ship; more entries turn
on the area selector and split the sidebar by top-level folder. Stick with a single area
until the content outgrows it.
```

For deeper coverage of section grouping and `order:` strategy, see Organize content with sections and areas.

Checkpoint

- Run `dotnet run` and visit `http://localhost:5000/guides/install` — the Install Pennington page renders.
- Visit `http://localhost:5000/guides/configure` — the Configure the site page renders.
- The Guides sidebar lists both new pages, with **Install Pennington** above **Configure the site** (sorted by `order:`).

---

2. Link siblings with relative paths

Both pages exist but neither knows about the other. Adding a relative-path link at the bottom of each one creates a natural "previous / next" flow without hardcoding the area slug.

Add a "Next" footer to `install.md`

Append a `## Next` heading and a relative-path link to the bottom of `install.md`. `./configure` resolves against the current page's URL (`/guides/install`), so it points at `/guides/configure` no matter where the area sits.

MARKDOWN

````
---
title: Install Pennington
description: Add the Pennington DocSite package and wire AddDocSite + UseDocSite into a
fresh ASP.NET host.
sectionLabel: Guides
order: 20
---

# Install Pennington

Install Pennington into an ASP.NET project with one NuGet package and three lines of DI
wiring.

## 1. Add the package

```bash
dotnet add package Pennington.DocSite
```

## 2. Wire DocSite in `Program.cs`

Three calls — register the services, mount the middleware, hand control to the host:

```csharp
builder.Services.AddDocSite(() => new DocSiteOptions { /* ... */ });
app.UseDocSite();
await app.RunDocSiteAsync(args);
```

The host is now ready for content. Drop markdown files under `Content/guides/` and they
appear in the sidebar on the next request.

## Next

Pick a site title and decide on areas in [Configure the site](./configure).
````

Add a "Previously" footer to `configure.md`

Mirror the same shape on `configure.md` with a `./install` link back to the first page. Both pages now link to their sibling with a path that survives any move of the `Content/guides/` folder.

MARKDOWN

```
---
title: Configure the site
description: Pick a site title, set the GitHub link, and decide on a single area or
multiple.
sectionLabel: Guides
order: 30
---

`DocSiteOptions` is the one options record `AddDocSite` reads. Set the fields that surface
in the rendered chrome and the rest of the template falls into place.

## Fields worth setting first

- `SiteTitle` — appears in the header and the `<title>` tag.
- `Description` — meta description used in search snippets and social cards.
- `GitHubUrl` — surfaces the GitHub icon in the header.
- `HeaderContent` / `FooterContent` — raw HTML slots, useful for a logo and a copyright
line.

## Areas — one or many?

`Areas` is an `IReadOnlyList<ContentArea>`. One entry is enough to ship; more entries turn
on the area selector and split the sidebar by top-level folder. Stick with a single area
until the content outgrows it.

## Previously

[Install Pennington](./install) covered getting the package and wiring in place.
```

Checkpoint

- Reload `http://localhost:5000/guides/install` — a **Configure the site** link sits at the bottom of the page. Click it.
- The browser lands on `/guides/configure`. A **Install Pennington** link at the bottom of that page returns to the first.
- View source on either page: the relative `./configure` and `./install` markdown links rendered as `href="/guides/configure"` and `href="/guides/install"`.

---

3. Turn the index into a hub with absolute paths

The `Content/guides/index.md` page from the scaffold still says "Walkthroughs and how-tos live in this folder" — a placeholder that no longer matches the content under it. Let's rewrite it as a hub that links to both pages with absolute paths.

Replace `index.md` with the hub markdown below

Absolute paths (`/guides/install`) survive folder moves of the source page. For the full link-form rundown, see Link between pages without hardcoding URLs.

MARKDOWN

```
---
title: Guides
description: Onboarding walkthroughs for a fresh Pennington DocSite.
sectionLabel: Guides
order: 10
---

# Guides

Two short walkthroughs cover the path from empty project to running site.

- [Install Pennington](/guides/install) — add the package and wire `AddDocSite`.
- [Configure the site](/guides/configure) — set the site title, GitHub link, and areas.
```

Checkpoint

- Visit `http://localhost:5000/guides/` — the Guides landing page now lists the two walkthroughs.
- Click **Install Pennington** — the browser navigates to `/guides/install`.
- Click **Configure the site** — the browser navigates to `/guides/configure`.

---

4. Make one link rename-safe with `uid:` + `xref:`

Absolute paths break the moment the target file moves or gets renamed. A `uid:` declared in the page's front matter gives the page a stable identifier; `xref:` links resolve through it, so the link survives the file moving or even the URL changing.

Add `uid: guides.install` to `install.md`'s front matter

Open `install.md` and add one front-matter key — the page is now reachable by `xref:guides.install` no matter where the file lives.

MARKDOWN

```
---
title: Install Pennington
description: Add the Pennington DocSite package and wire AddDocSite + UseDocSite into a
fresh ASP.NET host.
uid: guides.install
sectionLabel: Guides
order: 20
---

Install Pennington into an ASP.NET project with one NuGet package and three lines of DI
wiring.

## 1. Add the package

```bash
dotnet add package Pennington.DocSite
```

2. Wire DocSite in `Program.cs`

Three calls — register the services, mount the middleware, hand control to the host:

```csharp
builder.Services.AddDocSite(() => new DocSiteOptions { /* ... */ });
app.UseDocSite();
await app.RunDocSiteAsync(args);
```

The host is now ready for content. Drop markdown files under `Content/guides/` and they
appear in the sidebar on the next request.

Next

Pick a site title and decide on areas in [Configure the site](./configure).
```

Swap the install link in `index.md` to use `xref:`

Open `index.md` and replace `/guides/install` with `xref:guides.install`. The configure link stays as an absolute path — handy for seeing both forms side by side in the rendered output.

MARKDOWN

```
---
title: Guides
description: Onboarding walkthroughs for a fresh Pennington DocSite.
sectionLabel: Guides
order: 10
---

Two short walkthroughs cover the path from empty project to running site.

- [Install Pennington](xref:guides.install) — add the package and wire `AddDocSite`.
- [Configure the site](/guides/configure) — set the site title, GitHub link, and areas.
```

Checkpoint

- Reload `http://localhost:5000/guides/` — both links in the hub still work. The rendered `<a>` for **Install Pennington** points at `/guides/install` just as the absolute-path version did.
- View source: the `xref:guides.install` href has been rewritten to the canonical URL. The xref form is the same shape an editor would have produced — but the source markdown now survives any rename of `install.md`.

Summary

- Two markdown files under `Content/guides/` showed up in the sidebar without any extra wiring, sorted by `order:` from front matter.
- Relative paths (`./configure`) link tightly coupled sibling pages — the form that survives area-folder renames.
- Absolute paths (`/guides/configure`) link from a hub where the source page may move but the target's location is stable.
- `uid:` plus `xref:` — the rename-safe form — turns the page identifier itself into the link target.
- For the full link-form reference (anchors, assets, sub-path deployments), see Link between pages without hardcoding URLs. For deeper `uid:` semantics, see Cross-reference resolution.

Organize content with sections and areas

Getting Started Split a DocSite's Content/ folder into two top-level areas with subfolder-driven sections, and use staggered order: values so the sidebar groups in the order you expect.

By the end of this tutorial the DocSite runs at `http://localhost:5000` with an area selector showing **Guides** and **Reference**. Each area renders its own grouped sidebar: *Getting Started* and *Advanced* under Guides, *Core API* and *Extensions* under Reference, with pages sorted by `order:` inside each group. For the algorithm behind the sidebar, see Why the sidebar mirrors your folders.

Prerequisites

- .NET 10 SDK installed
- Completed Scaffold a documentation site with DocSite (provides the area-free `Content/guides/` host this tutorial turns into areas)
- Completed Add doc pages and link between them (so `DocSiteFrontMatter` keys like `sectionLabel:` and `order:` are already familiar)

The finished code for this tutorial lives in examples/DocSiteSectionsExample.

---

1. Register two areas and start from a flat page

So far `Content/` is area-free — every page shares one sidebar tree. This tutorial splits that into two switchable tabs, **Guides** and **Reference**, then fills each with grouped sections. To watch the grouping build up from nothing, reset the Guides area to a single flat page first: delete the `configure.md` and hub `index.md` you added in Add doc pages and link between them, leaving just `install.md`. Then register the areas and strip `install.md` back to minimal front matter, so the sidebar starts as a single ungrouped entry before any sections appear.

Register two content areas in `Program.cs`

Add two `ContentArea` entries — `Guides` bound to `guides/` and `Reference` bound to `reference/`. Each binds a top-level folder under `Content/` to its own sidebar tab, and the selector appears once more than one area is configured. Every other change in this tutorial is a filesystem change under `Content/`.

CSHARP

```
using Pennington.DocSite;

var builder = WebApplication.CreateBuilder(args);

// Same DocSite host shape as apps #4 and #5 — the focus here is on the
// *structure* of `Content/`. Two areas, each broken into two subfolder-backed
// sections. `NavigationBuilder` (inside `MainLayout`) turns the discovered
// flat TOC list into a grouped sidebar: each subfolder under an area becomes
// a non-navigable section header, and the pages inside sort by their front
// matter `order:` (tiebreaker: title).
builder.Services.AddDocSite(() => new DocSiteOptions
{
    SiteTitle = "Sections Docs",
    SiteDescription = "Structure Content/ into areas and sections using subfolders, section,
and order front matter.",
    GitHubUrl = "https://github.com/usepennington/pennington",
    HeaderContent = """<a href="/">Sections Docs</a>""",
    FooterContent = """<footer class="mt-16 py-8 text-center text-sm text-base-500">Built
with Pennington DocSite.</footer>""",

    // Two areas bound to two top-level content folders. The sidebar renders
    // an area selector above the per-area TOC; each area's TOC is grouped
    // by subfolder (sections "Getting Started" / "Advanced" under guides,
    // "Core Api" / "Extensions" under reference).
    Areas =
    [
        new ContentArea("Guides", "guides"),
        new ContentArea("Reference", "reference"),
    ],
});

var app = builder.Build();

app.UseDocSite();

await app.RunDocSiteAsync(args);
```

Strip `Content/guides/install.md` back to minimal front matter

Leave `Content/guides/install.md` with just a `title:` and a `description:` — drop the `order:`, `uid:`, and the Next footer from the earlier tutorial so the page starts as a bare entry.

MARKDOWN

```
---
title: Install Pennington
description: Add the Pennington package to a new or existing ASP.NET project.
---

The first thing every new Pennington site needs is the package itself.
```

With no subfolder, the page is a single ungrouped entry directly under the Guides area — there is no section header because there is no folder to title-case into one. A missing `order:` defaults to `int.MaxValue`, so an un-ordered page sorts *after* any page with an explicit `order:` — but here it is the only page, so it just appears on its own.

Checkpoint

The sidebar shows the page directly, with no section header above it.

- Run `dotnet run` from your project and visit `http://localhost:5000/guides/install`
- The Guides sidebar shows the **Install Pennington** link directly under the area with no section header above it

---

2. Move the page into a subfolder to create a section

Now let's move the same page under a `getting-started/` subfolder and add `sectionLabel:` plus `order:` to the front matter. The sidebar gains its first grouped section header.

Move `install.md` under `Content/guides/getting-started/`

Delete `Content/guides/install.md` and create `Content/guides/getting-started/installation.md` in its place. The subfolder name is what creates the sidebar section header — Pennington title-cases the folder (`getting-started` → *Getting Started*) and renders it as a non-navigable group label.

Moving the file changes its URL. Routes preserve subfolders, so the page no longer serves at `/guides/install` — it now serves at `/guides/getting-started/installation/`, mirroring its path under `Content/`. The folder you added for grouping also became a URL segment.

Add `sectionLabel: Getting Started` and `order: 10` to the front matter

MARKDOWN

```
---
title: Install Pennington
description: Add the Pennington package to a new or existing ASP.NET project.
sectionLabel: Getting Started
order: 10
---

The first thing every new Pennington site needs is the package itself.
```

`order:` sorts pages within the section (smaller first). `sectionLabel:` surfaces in breadcrumbs and prev/next chrome.

Checkpoint

- The old `http://localhost:5000/guides/install` URL now 404s — the page moved. Visit `http://localhost:5000/guides/getting-started/installation/` instead
- The Guides sidebar shows a non-navigable **Getting Started** header with the **Install Pennington** link indented under it
- The breadcrumb at the top of the article reads *Guides › Getting Started › Install Pennington*

---

3. Fill in the rest of the Guides area

Let's add the remaining pages to `getting-started/` and `advanced/` so Guides has two sibling sections with staggered `order:` values.

Add two more pages to `getting-started/` with `order: 20` and `order: 30`

Add the Guides landing page and two more pages to the `getting-started/` subfolder. Give `first-project.md` an `order:` of `20` and `configuration.md` an `order:` of `30`. Each page also carries `sectionLabel: Getting Started`.

MARKDOWN

```
---
title: Guides
description: Walk-throughs for getting productive with Pennington, grouped by skill level.
sectionLabel: Guides
order: 0
---

Welcome to the **Guides** area. The sidebar groups every page in this area by
the subfolder it lives in — *Getting Started* collects the first three pages
you want to read, *Advanced* holds the deeper material. Within each group,
pages are ordered by the `order:` front-matter key.

Pick a page from the sidebar to jump in.
```

MARKDOWN

````
---
title: Create your first project
description: Wire AddPennington and UsePennington into Program.cs and drop in a markdown
page.
sectionLabel: Getting Started
order: 20
---

With the package installed, the smallest useful site is two lines of
registration and one markdown file on disk.

## Program.cs

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

var app = builder.Build();
app.UsePennington();
await app.RunOrBuildAsync(args);
```

## Drop in a page

Create `Content/index.md` with a `title:` front-matter key and a body. Run
`dotnet run` and the page is served from `/` with hot reload on file change.

The next guide covers the handful of options you will reach for first.
````

MARKDOWN

```
---
title: Configure your site
description: Tour the PenningtonOptions knobs you will reach for on day one.
sectionLabel: Getting Started
order: 30
---

`AddPennington` accepts a configuration callback that lets you tune the
content root, URL style, and a handful of feature toggles without leaving
`Program.cs`.

## Point at a content folder

By default Pennington reads from `Content/` next to your project. Override
the path when your content lives elsewhere in the repo or ships from an
embedded resource.

## Change the URL style

`LowercaseUrls` and `AppendTrailingSlash` control the shape of every
generated link. Pick a style early — changing it later invalidates existing
inbound links unless you pair the change with redirect front matter.

The next area — *Advanced* — covers layout overrides and the response
pipeline.
```

The 10/20/30 spacing leaves room to drop pages in later without renumbering. The minimum `order:` value in the section is `10` — that matters in the next step.

Add the `advanced/` section with `order: 40` and `order: 50`

Create `Content/guides/advanced/` and add two pages with `sectionLabel: Advanced` and `order:` values of `40` and `50`.

MARKDOWN

```
---
title: Swap in a custom layout
description: Override the default DocSite layout with your own Razor component.
sectionLabel: Advanced
order: 40
---

DocSite ships with a sensible default layout, but every surface is a plain
Razor component. When you need a different header, footer, or body grid,
register your layout after `AddDocSite` and it takes precedence.

## Replace the main layout

Pass `AdditionalRoutingAssemblies` a reference to the assembly that holds
your `MyLayout.razor`, then mark it with the same `@layout`/`@inherits`
shape DocSite's `MainLayout` uses.

## Keep the sidebar or write your own

The easiest path is to keep `AreaNavigation` and `TableOfContentsNavigation`
inside your custom layout so sidebar grouping still works exactly as this
tutorial describes. The harder path — writing a bespoke nav — is covered
in a later how-to.
```

MARKDOWN

````
---
title: Hook into the response pipeline
description: Intercept rendered HTML before it reaches the browser with IResponseProcessor.
sectionLabel: Advanced
order: 50
---

Every rendered page flows through a response pipeline before hitting the
wire. Register an `IResponseProcessor` to mutate the HTML — add a feedback
widget, rewrite anchor IDs, or inject analytics — without touching the
markdown.

## Register a processor

```csharp
builder.Services.AddSingleton<IResponseProcessor, MyProcessor>();
```

Processors run in `Order` ascending. Override `ShouldProcess` to scope the
work to particular routes, content types, or request metadata.

## Where this fits vs islands

Response processors mutate the already-rendered HTML server-side, before it
reaches the browser. Islands — the SPA engine's `data-spa-region` blocks —
mark which server-rendered regions swap in on in-site navigation; Pennington
renders them on the server with no client hydration. Reach for a processor to
change the HTML every page ships; for interactive client behavior, ship your
own client-side script that enhances the rendered HTML.
````

Section headers inherit the minimum `order:` of their pages. Leaving gaps between section order ranges — `getting-started/` at 10/20/30, `advanced/` at 40/50 — keeps *Getting Started* above *Advanced* without relying on alphabetical tie-breaks.

Checkpoint

- Revisit `http://localhost:5000/guides/getting-started/installation/`
- The Guides sidebar shows, top to bottom: **Getting Started** (with *Install Pennington*, *Create your first project*, *Configure your site*) then **Advanced** (with *Custom layouts*, *The response pipeline*)
- Click around — breadcrumbs and prev/next labels reflect the `sectionLabel:` on each page

---

4. Populate the Reference area to confirm it repeats the pattern

The same subfolder-plus-staggered-order pattern applies to the `Reference` area. Switching between both areas through the sidebar's area selector confirms each gets its own independent sidebar tree.

Fill in `Content/reference/core-api/` with `order: 10` and `order: 20`

Create the `core-api/` subfolder under `Content/reference/` and add two pages, each with `sectionLabel: Core API` and `order:` values of `10` and `20`. The folder creates the section, the key labels it, and the staggered numbers keep sibling sections predictable.

MARKDOWN

```
---
title: Reference
description: API surface for the Pennington library, split into core and extension packages.
sectionLabel: Reference
order: 0
---

The **Reference** area mirrors the package structure: *Core API* covers the
types in `Pennington` itself, *Extensions* covers the optional surface
(Markdig extensions, content services). Page order inside each section
matches the order you are likely to discover the types while building a
site.

Pick a page from the sidebar to inspect a specific surface.
```

MARKDOWN

```
---
title: PenningtonOptions
description: Core configuration surface handed to AddPennington.
sectionLabel: Core API
order: 10
---

`PenningtonOptions` is the record the configuration callback on
`AddPennington` mutates. It holds a handful of knobs that apply to every
page in the site regardless of content source.

## Keys worth knowing

- `ContentRootPath` — folder (or embedded-resource root) content is
  discovered from.
- `LowercaseUrls` — whether to force every generated URL to lowercase.
- `AppendTrailingSlash` — pick slash vs no-slash for clean URLs.
- `UrlStyle` — `Clean` (folder/index.html) or `Extension` (filename.html).

Every option has a sensible default; populate only the ones you need to
deviate from.
```

MARKDOWN

```
---
title: ContentPipeline
description: The discovery/parse/render pipeline every content source flows through.
sectionLabel: Core API
order: 20
---

`ContentPipeline` is the assembly line `IContentService` implementations
feed into. Each source yields `DiscoveredItem`s, the pipeline parses them
into `ParsedItem`s via `IContentParser`, and finally renders them into
`RenderedItem`s via `IContentRenderer`.

## The three stages

1. **Discover** — each registered `IContentService` walks its source (disk,
   Razor pages, a JSON feed, whatever) and emits `DiscoveredItem` unions.
2. **Parse** — the matching `IContentParser` reads the item, separates
   front matter from body, and produces a `ParsedItem`.
3. **Render** — `IContentRenderer` turns the parsed body into HTML (plus
   an outline of headings and any diagnostics).

Custom parsers and renderers plug into the same pipeline — see the
*Extensions* section in the sidebar for how to write one.
```

Add `Content/reference/extensions/` with `order: 30` and `order: 40`

Create `extensions/` and drop two pages in it with `sectionLabel: Extensions` and `order:` values of `30` and `40`. Continuing the count rather than restarting at `10` keeps the section order ranges separated — *Core API* at 10/20, *Extensions* at 30/40 — so *Core API* sorts above *Extensions* by the same minimum-`order:` rule from unit 3.

MARKDOWN

```
---
title: Markdown extensions
description: The Markdig extensions Pennington ships with — alerts, tabbed code,
highlighting.
sectionLabel: Extensions
order: 30
---

Pennington configures Markdig with a curated set of extensions that light
up the authoring syntax the tutorials lean on.

## What ships in the box

- **Alerts** — GitHub-flavoured block quotes (`> [!NOTE]`, `TIP`,
  `IMPORTANT`, `WARNING`, `CAUTION`).
- **Tabbed code groups** — two or more adjacent fenced blocks with
  `tabs=true title="…"`.
- **Syntax highlighting** — TextMate grammars and ANSI shell output.
- **Code annotations** — trailing-comment `[!code highlight]` markers.

Registering your own extension is covered in the *Hook into the response
pipeline* guide's companion how-to.
```

MARKDOWN

```
---
title: Custom content services
description: Teach Pennington a new content source by implementing IContentService.
sectionLabel: Extensions
order: 40
---

`IContentService` is the extension point for loading content from anything
that isn't plain markdown — a JSON feed, a database, a remote API, an
embedded resource. Register an implementation and the pipeline treats its
items exactly like every other source.

## The four methods

- `DiscoverAsync()` — yield a `DiscoveredItem` per logical page.
- `GetContentTocEntriesAsync()` — flat list of TOC entries with title,
  order, and hierarchy parts.
- `GetCrossReferencesAsync()` — any `uid`-to-route mappings you want the
  xref resolver to see.
- `GetContentToCopyAsync()` — assets the static-build step should copy
  alongside the rendered HTML.

Implementations live next to `MarkdownContentService` in the DI container
and are iterated in registration order.
```

Switch areas with the sidebar's area selector

Click the area selector pill at the top of the sidebar — the control that toggles between *Guides* and *Reference*. Each area has its own independent sidebar tree. The `ContentArea` bindings from `Program.cs` plus the subfolder layout are what make this work, with no extra code.

Checkpoint

- With the host running, visit `http://localhost:5000/reference/core-api/pennington-options`
- The sidebar shows **Core API** above **Extensions**, with two pages under each in `order:` sequence
- Click the area selector to **Guides** — the sidebar replaces itself with the *Getting Started* / *Advanced* groups from unit 3
- The area selector tracks the current area as navigation moves between pages

---

Summary

- A DocSite's `Content/` folder splits into multiple `ContentArea` entries, and each one gets its own sidebar tree.
- **The subfolder name creates the sidebar section** — `sectionLabel:` is metadata for breadcrumbs and prev/next labels, not a grouper.
- Staggered `order:` values across sibling sections (10/20/30 for one, 40/50 for the next) sort section headers in the intended order, without relying on alphabetical tie-breaks between folder names.
- The shape of the generated sidebar is predictable from the shape of the `Content/` folder before running the site.

Add a Razor landing page at the site root

Getting Started Route a Razor component at / so a DocSite opens on a hand-built landing page, and swap the doc-page chrome for the sidebar-free FullWidthLayout.

By the end of this tutorial the DocSite at `http://localhost:5000/` opens on a Razor landing page — a hero heading, a call to action, and two cards linking into the Guides area — laid out with the sidebar-free `FullWidthLayout` instead of the default doc-page chrome.

Along the way the tutorial covers routing a Razor component at `/`, why a `@page "/"` route wins over DocSite's catch-all, and swapping the layout a routed page renders inside.

Prerequisites

- .NET 10 SDK installed
- Completed Add doc pages and link between them — it provides the single-area host and the `install` / `configure` guide pages this landing page links to

The finished code for this tutorial lives in examples/DocSitePagesAndLinksExample.

1. Clear the root so a Razor component can claim it

The host from Add doc pages and link between them binds one content area, `guides`, to the `/guides/` tab. If you also followed Scaffold a documentation site with DocSite, a `Content/index.md` is serving the root — that's the markdown landing page this tutorial replaces with a routed Razor component. Delete it so the root is free, then confirm `/` returns a 404 before you route a component at it.

Delete `Content/index.md`

Remove the scaffold's root markdown page. With nothing bound to the root — no `Content/index.md` and no routed component pointed at `/` — a request to `/` returns a 404.

Run the host and visit the root

BASH

```
dotnet run
```

Open `http://localhost:5000/` in a browser.

Checkpoint

- `http://localhost:5000/` returns a 404 — nothing serves the root.
- `http://localhost:5000/guides/` still renders the Guides hub from Add doc pages and link between them.

2. Route a Razor component at the root

A routed Razor component whose `@page` template is `/` owns the root URL. `AddDocSite` adds your project's assembly to the routing assemblies it hands both the live Blazor router and the static build's page scanner, so a `@page` component in your project is picked up by both with no extra wiring. And a literal `/` route is more specific than DocSite's own catch-all, so it wins the match.

Create `Components/Index.razor`

Create a `Components/` folder at the project root and add `Index.razor` with a `@page "/"` directive and minimal markup.

RAZOR

```
@page "/"

<h1>Pages &amp; Links</h1>
<p>The site root now renders a Razor component.</p>
```

The `@page "/"` directive is the whole wiring — no `Program.cs` change, no registration call.

Restart the host

A `.razor` edit is a compile change, so stop the host and run `dotnet run` again to pick up the new component.

Checkpoint

- `http://localhost:5000/` no longer 404s — it renders the **Pages & Links** heading.
- The page is wrapped in the default doc-page chrome: a sidebar on the left and an outline rail on the right. The next unit replaces that layout.

---

3. Switch to the full-width layout

A routed component with no `@layout` directive renders inside DocSite's default, `MainLayout` — the three-column doc-page chrome with the sidebar and outline rail. A landing page wants the header and footer but not the navigation columns. `FullWidthLayout` is exactly that shape.

Add a `@layout` directive to `Index.razor`

Add one line under `@page` naming the layout by its full type name.

RAZOR

```
@page "/"
@layout Pennington.DocSite.Components.Layout.FullWidthLayout

<h1>Pages &amp;amp; Links</h1>
<p>The site root now renders a Razor component.</p>
```

`FullWidthLayout` keeps the DocSite header and footer and gives the page the full content width — no sidebar, no outline rail.

Restart the host

The `@layout` directive is another `.razor` edit, so stop the host and run `dotnet run` again to recompile.

Checkpoint

- `http://localhost:5000/` renders the heading across the full content width.
- The sidebar and outline rail are gone; the DocSite header and footer remain.

---

4. Build out the landing page

With routing and layout settled, the component is plain Razor markup. Fill it with a hero, a call to action, and two cards linking into the Guides area. Styling is MonorailCSS utility classes using the semantic palette — `primary`, `accent`, `base` — with a `dark:` variant on every color-bearing utility.

Replace `Index.razor` with the finished landing page

RAZOR

```
@page "/"
@layout Pennington.DocSite.Components.Layout.FullWidthLayout
@using Microsoft.AspNetCore.Components.Web

@* A Razor component named Index with @page "/" owns the root URL. DocSite
   registers this project's assembly in its routing assemblies, so the literal
   "/" route is picked up by both the live Blazor router and the static build —
   and it beats the catch-all in Pages.razor. @layout swaps the sidebar layout
   for FullWidthLayout, which keeps the header and footer but drops the nav. *@

<PageTitle>Pages &amp;amp; Links</PageTitle>

<section class="py-12 lg:py-20">
    <p class="text-xs font-display font-semibold tracking-widest uppercase text-primary-600
dark:text-primary-400 mb-4">
        Pennington DocSite
    </p>
    <h1 class="font-display text-4xl lg:text-6xl font-bold tracking-tight text-base-900
dark:text-base-50 leading-tight mb-6 text-balance">
        A docs site with a front door.
    </h1>
    <p class="text-base lg:text-lg text-base-600 dark:text-base-400 max-w-2xl mb-8 leading-
relaxed">
        Every guide in one place — installing Pennington, configuring the host,
        and the linking patterns that hold a documentation site together.
    </p>
    <div class="flex flex-wrap items-center gap-3">
        <a href="/guides/"
           class="inline-flex items-center font-display font-semibold bg-primary-600
hover:bg-primary-500 text-white rounded-xl px-6 py-3 transition-colors">
            Read the guides
        </a>
        <a href="https://github.com/usepennington/pennington"
           class="inline-flex items-center font-display font-semibold text-base-700
dark:text-base-200 hover:text-primary-700 dark:hover:text-primary-400 rounded-xl px-5 py-3
ring-1 ring-base-300 dark:ring-base-700 transition-colors">
            View on GitHub
        </a>
    </div>
</section>

<section class="pb-16 lg:pb-24">
    <div class="grid grid-cols-1 md:grid-cols-2 gap-4">
        <a href="/guides/install"
           class="rounded-2xl border border-base-200 dark:border-base-800 bg-base-50
dark:bg-base-900/50 p-6 transition-colors hover:border-primary-400 dark:hover:border-
primary-500">
            <h2 class="font-display text-xl font-semibold text-base-900 dark:text-base-50
mb-2">
                Install Pennington
            </h2>
```

```
        </a>        <a href="/guides/configure"           class="rounded-2xl border border-
base-200 dark:border-base-800 bg-base-50 dark:bg-base-900/50 p-6 transition-colors
hover:border-primary-400 dark:hover:border-primary-500">            <h2 class="font-display
text-xl font-semibold text-base-900 dark:text-base-50 mb-2">                Configure the
site            </h2>            <p class="text-sm text-base-600 dark:text-base-400 leading-
relaxed">                Set the title, footer, and area routing.            </p>
</a>    </div></section>
```

The two cards link to `/guides/install` and `/guides/configure` — the pages built in Add doc pages and link between them. `<PageTitle>` sets the browser tab text, the same component DocSite uses on doc pages.

Restart the host and open the root

Stop the host and run `dotnet run` again to recompile the component, then open `http://localhost:5000/`.

Checkpoint

- `http://localhost:5000/` renders the hero heading, the **Read the guides** button, and two guide cards.
- Clicking a card navigates to the matching guide page; the **Read the guides** button lands on `/guides/`.
- Run `dotnet run -- build` — the static build's page scanner picks up the same `@page "/"` route and writes the landing page to `output/index.html`.

Summary

- A Razor component with `@page "/"` owns the site root — `AddDocSite` already routes your project's assembly, so the directive is the whole wiring.
- A literal `/` route beats DocSite's catch-all, and the same route is honored by both the live host and the static build.
- A routed component defaults to `MainLayout`; a `@layout` directive naming `FullWidthLayout` drops the sidebar for a landing-page shape.
- The component body is ordinary Razor styled with MonorailCSS — semantic palette utilities, `dark:` variants, and links straight into the content areas.

Add a blog to your documentation site

Getting Started Drop a Content/blog folder into a DocSite and watch the blog index, post pages, browse-by-tag pages, and RSS feed light up — no Program.cs changes.

By the end of this tutorial the DocSite at `http://localhost:5000/` carries a **Blog** link in its header, a `/blog` index listing two posts newest-first, post pages with a date and byline, browse-by-tag pages, and an RSS feed at `/rss.xml`.

The blog activates from content alone — a folder named `blog` under `Content/`. There is no `Program.cs` change anywhere in this tutorial; every step is a markdown file.

Prerequisites

- .NET 10 SDK installed
- Completed Scaffold a documentation site with DocSite — or any DocSite host ready to run

The finished code for this tutorial lives in examples/DocSiteBlogExample.

---

1. Write your first post

A DocSite turns on its blog when it finds a folder named `blog` under `Content/`. Create that folder, drop one post into it, and the blog appears.

Create `Content/blog/launching-the-docs.md`

Add a `blog` folder under `Content/`, alongside your area folders, and create `launching-the-docs.md` inside it. The filename minus `.md` becomes the post's URL slug, so this post serves at `/blog/launching-the-docs`.

> **Note**
>
> The folder must be named exactly `blog`, in lowercase. On a case-sensitive filesystem `Blog` or `blogs` is not detected.

Paste in the post

Paste the markdown below. The four front-matter fields are the ones every post uses: `title` is the post heading and link label, `description` is the summary shown on the blog index, `author` is the byline, and `date` is the publication date.

MARKDOWN

```
---
title: Launching the Harbor docs
description: The Harbor documentation site is live, and it now has a blog.
author: Dana Reyes
date: 2026-05-08
---

Harbor's documentation has a new home. Alongside the guides, this blog will
carry release notes, design notes, and the occasional field report.

## Why a blog

Guides explain how Harbor works today. A blog is the place for what changed
and why — the running story behind the tool.
```

Run the host

BASH

```
dotnet run
```

Open `http://localhost:5000/`.

Checkpoint

- A **Blog** link appears in the site header, beside the theme toggle.
- `http://localhost:5000/blog` lists one post — **Launching the Harbor docs** — with its date and description.
- `http://localhost:5000/blog/launching-the-docs` renders the post body under its title, with the **By Dana Reyes** byline.

---

2. Add a second post

The blog index orders posts by `date`, newest first. Add a second, more recent post and watch it take the top slot.

Create `Content/blog/whats-next.md`

Add a second file in the `blog` folder with the markdown below. Its `date` — `2026-05-15` — is a week after the first post.

MARKDOWN

```
---
title: What's next for Harbor
description: A look at the features on the Harbor roadmap for the next release.
author: Dana Reyes
date: 2026-05-15
---

The docs are live, so here is where Harbor goes next.

## On the roadmap

- Faster incremental builds.
- A watch mode that reloads on content changes.
- A plugin hook for custom output formats.
```

Reload the blog index

A markdown edit needs no restart — the host watches `Content/`. Save the file and reload `http://localhost:5000/blog`.

Checkpoint

- `http://localhost:5000/blog` now lists two posts.
- **What's next for Harbor** sits above **Launching the Harbor docs** — the newer `date` puts it first.

---

3. Tag your posts and find the feed

A `tags` list on a post adds it to browse-by-tag pages. Tag both posts, then look at the feed the blog has been publishing all along.

Add tags to the first post

Add a `tags` block to `launching-the-docs.md`. The file now reads:

MARKDOWN

```
---
title: Launching the Harbor docs
description: The Harbor documentation site is live, and it now has a blog.
author: Dana Reyes
date: 2026-05-08
tags:
  - announcements
  - docs
---

Harbor's documentation has a new home. Alongside the guides, this blog will
carry release notes, design notes, and the occasional field report.

## Why a blog

Guides explain how Harbor works today. A blog is the place for what changed
and why — the running story behind the tool.
```

Add tags to the second post

Give `whats-next.md` its own `tags` block. One tag, `announcements`, is shared with the first post.

MARKDOWN

```
---
title: What's next for Harbor
description: A look at the features on the Harbor roadmap for the next release.
author: Dana Reyes
date: 2026-05-15
tags:
  - announcements
  - roadmap
---

The docs are live, so here is where Harbor goes next.

## On the roadmap

- Faster incremental builds.
- A watch mode that reloads on content changes.
- A plugin hook for custom output formats.
```

Reload and follow the tags

Save both files and reload `http://localhost:5000/blog/launching-the-docs`.

Checkpoint

- The post page now lists its tags beneath the body — **launching-the-docs** shows **announcements** and **docs** as links.
- Following a tag opens its page — `http://localhost:5000/blog/tags/announcements` lists both posts; `/blog/tags/docs` lists only the first.
- The blog index carries a **Browse by tag** link to `http://localhost:5000/blog/tags`, which lists all three tags with their post counts — **announcements** (2), **docs** (1), and **roadmap** (1).

Now look at the feed the blog has been publishing since your first post.

Checkpoint

- `http://localhost:5000/rss.xml` is a valid RSS feed — a `<channel>` with the site title and two `<item>` elements whose `<title>`, `<description>`, `<pubDate>`, and `<author>` come from the front matter. The `<link>` URLs are relative to your site root, since no `CanonicalBaseUrl` is set; set `DocSiteOptions.CanonicalBaseUrl` to make them absolute.
- Run `dotnet run -- build` — the static build writes `blog/index.html`, the two post pages, the `blog/tags/` pages, and `rss.xml` into `output/`.

Every front-matter field you wrote into those posts is parsed into a BlogPostFrontMatter record, the type a DocSite binds for content under the `blog` folder.

Summary

- A folder named `blog` under `Content/` is the whole switch — `AddDocSite` finds it at startup and turns on the blog index, post pages, tag pages, the RSS feed, and the header link.
- Each `BlogPostFrontMatter` field drives a surface: `title`, `description`, and `date` for the index; `author` for the byline and RSS; `tags` for the `/blog/tags/` pages.
- Posts sort newest-first by `date`.
- `/rss.xml` is generated automatically; its `<link>` URLs are relative until you set `DocSiteOptions.CanonicalBaseUrl`.
- None of it needed a `Program.cs` change — the blog is pure content.

That rounds out the Getting Started with DocSite series. To go further, the **Beyond the Basics** tutorials build on this same host: add a second locale to your site and author a custom Razor component for markdown.

CHAPTER 3

Blogsite

Scaffold a blog with BlogSite

Getting Started Swap the bare Pennington host for the BlogSite template and configure the core options that drive the home, archive, post, tag, and RSS routes.

By the end of this tutorial, a running BlogSite host titled "Scaffold Blog" serves a home listing, `/archive`, `/blog/<slug>/`, `/tags/`, `/tags/<name>/`, and `/rss.xml` — all from a single placeholder post under `Content/Blog/`.

`AddBlogSite` folds the host, layout, navigation, and styling into one call, configured for a site where the blog *is* the site; for what the template wires, where the wiring stops, and why DocSite and BlogSite can't share an app, read what the templates wire for you first.

Prerequisites

- .NET 10 SDK installed
- Completed Create your first Pennington site
- Completed Serve markdown through a Blazor catch-all (so `Content/` already has at least one markdown file)

The stable .NET 10 SDK is all BlogSite needs — its package targets .NET 10, and you never write the `union` keyword that would call for a preview SDK. See the SDK and the union shim for when the .NET 11 beta is worth opting into.

The finished code for this tutorial lives in examples/BlogSiteScaffoldExample.

---

1. Start from the bare Pennington host

The host you built in the getting-started tutorials calls `AddPennington`, registers content with `AddMarkdownContent<DocFrontMatter>`, mounts `UsePennington`, and routes through a Blazor `@page "/{*Path}"` catch-all (`MarkdownPage.razor`) that resolves each URL with `IPageResolver` to serve individual pages.

That host serves individual pages but nothing else: no home listing, no `/archive`, no `/blog/<slug>` pages, no `/tags` listings, no `/rss.xml` feed, and no MonorailCSS chrome. The next section brings all of that in with a single `AddBlogSite` call.

Add the first post

Posts live under `{ContentRootPath}/{BlogContentPath}` — `Content/Blog/` with the defaults. Add one placeholder post so the host has something to serve. It uses four front-matter keys — `title`, `description`, `date`, and `author` — the minimum the home listing and RSS feed will need once the template is wired:

MARKDOWN

```
---
title: Hello world
description: A placeholder post so the scaffold has something to render. Tutorial 1.3.20
teaches the real BlogSiteFrontMatter fields.
date: 2026-04-13
author: Author Name
tags:
  - scaffold
---

This post exists so the bare BlogSite scaffold has at least one entry on the
home page and in the RSS feed. The next tutorial, **Author your first post
with `BlogSiteFrontMatter`**, walks through the full set of post front-matter
fields (tags, series, repository, section, redirectUrl, and more).
```

Checkpoint

- Run `dotnet run` and visit `/blog/hello-world` at the URL the console prints — the page shows unstyled HTML for the markdown. None of the BlogSite chrome (home listing, archive, tag pages, RSS) exists yet.

---

2. Wire `AddBlogSite`, `UseBlogSite`, and `RunBlogSiteAsync`

`AddBlogSite` is the BlogSite template's single registration call; it stands in for `AddPennington` plus the `AddMarkdownContent<DocFrontMatter>` line, wiring Pennington core, MonorailCSS, the Razor chrome, and the blog content services in one call. `UseBlogSite` mounts the middleware stack and Razor component routes; `RunBlogSiteAsync` dispatches between dev-serve and static-build.

Replace `Program.cs` with the BlogSite calls

CSHARP

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

builder.Services.AddBlogSite(() => new BlogSiteOptions
{
    SiteTitle = "Scaffold Blog",
    SiteDescription = "A minimal BlogSite scaffold showing AddBlogSite, UseBlogSite, and
RunBlogSiteAsync.",
    CanonicalBaseUrl = "https://example.com",

    ContentRootPath = "Content",
    BlogContentPath = "Blog",
    BlogBaseUrl = "/blog",

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

var app = builder.Build();

app.UseBlogSite();

await app.RunBlogSiteAsync(args);
```

The options populated here cover site identity (`SiteTitle`, `SiteDescription`, `CanonicalBaseUrl`), content paths shown at their defaults (`ContentRootPath`, `BlogContentPath`, `BlogBaseUrl`), and author fallbacks (`AuthorName`, `AuthorBio`). The full surface lives in Pennington.BlogSite.BlogSiteOptions.

Checkpoint

- Run `dotnet run` and visit `/` at the URL the console prints
- The BlogSite home layout appears: site title "Scaffold Blog", a recent-posts list with one entry, header chrome, and MonorailCSS styling

---

3. Verify every built-in route

With the post from section 1 in place and `AddBlogSite` wired, every route the template ships now responds. The placeholder post carries `tags: [scaffold]`, so the tag routes have one entry to list.

Walk the page routes

With the host running, visit each of these at the URL the console prints. Every one returns 200:

- `/` — home listing with `hello-world` as the only recent post.
- `/archive` — full archive, same single post in reverse-chronological order.
- `/blog/hello-world` — the post itself, now rendered with BlogSite chrome.
- `/tags` — the tag index, showing `scaffold` with a count of one.

- `/tags/scaffold` — the per-tag listing with the one post.

Check the RSS feed

Visit `/rss.xml`. It returns `application/rss+xml` with one `<item>` carrying the post title, link, description, pub date, and author.

The full route surface, including the paginated `/archive/page/{n}` and per-tag pages, is cataloged in Built-in BlogSite routes. The next tutorial expands the post to the full `BlogSiteFrontMatter` surface, adding `tags`, `series`, `repository`, `sectionLabel`, and `redirectUrl`.

Checkpoint

- Each page route above returns 200 and renders the placeholder post's metadata
- `/rss.xml` returns `application/rss+xml` content with one item whose `<guid>` matches the canonical post URL

Summary

- The bare `AddPennington` host gave way to `AddBlogSite` + `UseBlogSite` + `RunBlogSiteAsync`, and the full BlogSite chrome now renders.
- The core `BlogSiteOptions` surface — `SiteTitle`, `SiteDescription`, `CanonicalBaseUrl`, `ContentRootPath`, `BlogContentPath`, `BlogBaseUrl`, `AuthorName`, `AuthorBio` — is populated, and each field flows through to the rendered output.
- BlogSite binds posts through `AddMarkdownContent<BlogSiteFrontMatter>` (introduced in the next tutorial) and defaults content paths to `Content/Blog` served at `/blog`, which distinguishes it from the `DocSite` template's area-driven layout.
- Every built-in route the template ships responds: `/`, `/archive`, `/blog/<slug>`, `/tags`, `/tags/<name>`, and `/rss.xml`.

Publish your first post and light up the RSS feed

Getting Started Replace the scaffold's placeholder post with a fully-populated BlogSiteFrontMatter block and watch it flow into the blog index, per-tag pages, and the built-in RSS feed.

By the end of this tutorial, a running BlogSite at `http://localhost:5000` surfaces a single, fully-populated post on the home page, the archive, the per-tag index, and `/rss.xml` — the placeholder from the scaffold tutorial swapped for a post of your own. Each `BlogSiteFrontMatter` field maps to a specific surface; the steps walk through them in turn.

Prerequisites

- .NET 10 SDK installed
- Completed Scaffold a blog with BlogSite (or have that example's `Program.cs` and a single placeholder post ready to reuse)

The finished code for this tutorial lives in examples/BlogSiteFirstPostExample.

---

1. Start from a bare-minimum front-matter block

This section replaces the scaffold's placeholder post with a new file `Content/Blog/my-first-post.md` that carries only the three fields every post truly needs — `title`, `description`, and `date` — then confirms it renders on the home listing, the archive, and the RSS feed even in this minimal state.

Delete `Content/Blog/hello-world.md` and create `my-first-post.md`

The scaffold tutorial left a placeholder post named `hello-world.md` in `Content/Blog/`. Delete it, then create a new file `my-first-post.md` in the same folder. The filename (minus `.md`) becomes the URL slug, so the post serves at `/blog/my-first-post/`.

Paste in title, description, and date only

Paste the markdown body below into the new file. These three fields are the smallest front matter that lets a BlogSite post render cleanly: `title` is the post's heading and link label; `description` is what the home card, archive card, and RSS `<description>` element all pull from; and `date` drives both the archive sort order and the RSS `<pubDate>` element.

MARKDOWN

```
---
title: Shipping a tiny content engine for weekend projects
description: Notes from the first month of building Pennington.
date: 2026-04-10
---

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 it.

## What's here so far

Just title, description, and date. That is enough for the home
listing, the archive page, and the RSS feed to light up.
```

The two `---` fences delimit the YAML front matter block. The `date:` value parses as an ISO-8601 date; any format that round-trips as a date string works. For the full list of recognized front-matter keys, see the Pennington.BlogSite.BlogSiteFrontMatter reference page.

Checkpoint

- Run `dotnet run` from the example project
- Visit `http://localhost:5000/` — a single recent-posts card titled **Shipping a tiny content engine for weekend projects** appears with the description from the front matter
- Visit `http://localhost:5000/archive` — the same post appears as the only archive entry, dated **2026-04-10**
- Visit `http://localhost:5000/blog/my-first-post/` — the post body renders with its H1 and paragraph text

---

2. Add the author, tags, series, and repository fields

Next, the front-matter block expands with the four `BlogSiteFrontMatter` fields that each light up a distinct blog surface — `author`, `tags`, `series`, and `repository`.

Replace the front matter with the fully-populated block

Replace the existing YAML block with the fully-populated block below. Each new key drives a different surface:

- `author:` — byline on the post page and the `<author>` element in the RSS feed
- `tags:` — `/tags/<tag>/` index pages plus chips on the post page
- `series:` — shared-banner threading on the post chrome
- `repository:` — "Source Code" link card on the post page

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
---

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 it with something the BlogSite template
actually has opinions about.

## What the front matter is doing

Each field in the 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.
```

`BlogSiteFrontMatter` carries more keys than this — `sectionLabel` for navigation grouping and `redirectUrl` for migrated posts among them. For the full record definition, see Pennington.BlogSite.BlogSiteFrontMatter.

Reload and confirm every surface updated

With the file saved, reload the running site and verify each new field in turn. No code changes are needed — the host from the scaffold tutorial stays untouched.

The RSS feed is on because `EnableRss` defaults to `true`; to turn it off, set it on `BlogSiteOptions` (see Pennington.BlogSite.BlogSiteOptions).

Checkpoint

- Visit `http://localhost:5000/blog/my-first-post/` — the post header shows the byline **Author Name**, the series banner **Pennington Field Notes**, three tag chips (**pennington**, **dotnet**, **blogging**), and a **Source Code** link card pointing at the `repository:` URL.
- Visit `http://localhost:5000/tags/pennington/` — the post appears on the per-tag index; repeat for `/tags/dotnet/` and `/tags/blogging/`.
- Visit `http://localhost:5000/archive` — the archive card carries the longer description from the expanded front matter.
- Visit `http://localhost:5000/rss.xml` — the feed `<channel>` carries the site title and one `<item>` whose `<title>`, `<description>`, `<pubDate>`, `<author>`, and `<link>` all map back to the front matter.

Summary

- A Pennington blog post backed by `BlogSiteFrontMatter` maps predictably onto each blog surface — title/description/date for listings, author for byline and RSS, tags for `/tags/<tag>/` indexes, series for the shared banner, repository for the source-code link card.
- Dropping a new `Content/Blog/*.md` file brings it straight to the home page, the archive, every tag it claims, and `/rss.xml` — no `Program.cs` changes needed.

Add a hero, projects, and social links

Getting Started Populate the four BlogSite homepage surfaces — hero block, My Work card, social-icon row, and top-nav links — on BlogSiteOptions.

By the end of this tutorial, the BlogSite host displays a hero headline on the home page, a "My Work" sidebar card listing three projects, a row of four social-media icons beneath it, and a top-nav bar populated from `MainSiteLinks` — all driven by `BlogSiteOptions` without a line of Razor.

Prerequisites

- .NET 10 SDK installed
- Completed Scaffold a blog with BlogSite
- Completed Publish your first post and light up the RSS feed

The finished code for this tutorial lives in examples/BlogSiteHeroProjectsSocialsExample.

---

1. Populate the hero block

The BlogSite home page renders a headline block at the very top, driven entirely by `BlogSiteOptions.HeroContent`. `HeroContent` is a two-field positional record — `Title` and `Description` — so a single constructor call is all it takes.

Add `HeroContent` to the `AddBlogSite` call

Open the `AddBlogSite` call from the previous tutorial and add one property. The `HeroContent = new HeroContent(Title: …, Description: …)` assignment is the only addition — no new DI registrations, no new Razor files, no front matter changes. The rest of the options block carries forward unchanged from the scaffold tutorial.

CSHARP

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

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

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

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

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

Checkpoint

- Run `dotnet run` and visit `http://localhost:5000/`
- The hero title "Field notes from a weekend content engine" and the description paragraph stack above the recent-posts list from the first-post tutorial

---

2. Add a "My Work" projects section

`BlogSiteOptions.MyWork` accepts a `Project[]` that the home page renders as a sidebar card titled "My Work". `Project` is a three-field positional record — `Title`, `Description`, `Url`. The `Url` becomes the `<a href>` around each rendered entry, so it can point at a GitHub repo, a product page, or

any other URL.

Populate `MyWork` with a `Project[]` collection expression

Add the `MyWork` property right below `HeroContent`, populated with a C# collection expression. The property is typed as `Project[]` on `BlogSiteOptions`; its default is an empty list, so the "My Work" card stays invisible in the UI until populated here.

CSHARP

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

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

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

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

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

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

Checkpoint

- Run `dotnet run` and visit `http://localhost:5000/`
- A "My Work" sidebar card appears with three linked entries — Pennington, MonorailCSS, Mdazor — each clickable

---

3. Wire social links and top-nav header links

The final two surfaces land in the same `AddBlogSite` call. `Socials` is a `SocialLink(RenderFragment Icon, string Url)[]`; `MainSiteLinks` is a `HeaderLink(string Title, string Url)[]` that BlogSite renders in both the top-nav of `MainLayout.razor` and the footer. The listing below adds both.

Reference the built-in social icons

The four built-in icons ship as `static readonly RenderFragment` fields on SocialIcons — pass the field directly, not as a component tag (`SocialIcons.GithubIcon`, not `<SocialIcons.GithubIcon />`). Add a `using Pennington.BlogSite.Components;` directive at the top of `Program.cs` so `SocialIcons.GithubIcon` resolves.

Add the `Socials` and `MainSiteLinks` blocks

Add a `Socials = [...]` block with four entries covering all four built-ins, then a `MainSiteLinks = [...]` block with three entries — `Home` pointing to `/`, `Archive` to `/archive`, and `Tags` to `/tags`. Those URLs match the routes BlogSite includes, so the top-nav populates with no additional code.

CSHARP

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

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

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

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

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

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

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

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

Checkpoint

- Run `dotnet run` and visit `http://localhost:5000/`
- A horizontal row of four SVG icons — GitHub, Bluesky, LinkedIn, Mastodon — sits below the "My Work" sidebar card, each linking out to its `Url`
- A "Home / Archive / Tags" link row appears in the site header, and the same three links repeat in the footer nav
- Click **Archive** — the archive page lists the first post from the previous tutorial

---

Summary

- `HeroContent` now drives the home-page headline block.
- A `Project[]` on `MyWork` brought the "My Work" sidebar card to life with three linked entries.
- Four `SocialLink` entries wire the built-in `SocialIcons.GithubIcon`, `BlueskyIcon`, `LinkedInIcon`, and `MastodonIcon` `RenderFragment` fields — no Razor required.
- `MainSiteLinks` holds three `HeaderLink` entries, and they render in both the top-nav and the footer.
- All four homepage surfaces on `BlogSiteOptions` (hero, work, socials, header links) populate from one options block.

That completes the BlogSite getting-started arc — you have a scaffolded host, a fully-populated post, and a configured home page. From here, populate the blog homepage is the how-to that revisits these same options when you need them on their own, generate social card images adds OpenGraph cards to every post, and Built-in BlogSite routes catalogs the full route surface the template serves.

CHAPTER 4

Beyond Basics

Add a second locale to your site

Getting Started Turn a single-language DocSite into a bilingual one by registering a second locale, translating three pages, and letting the built-in LanguageSwitcher appear in the header.

By the end of this tutorial you'll have a running DocSite at `http://localhost:5000` that serves three English pages at `/`, `/about`, and `/getting-started`, plus three Spanish translations at `/es/`, `/es/about`, and `/es/getting-started`. A LanguageSwitcher component appears in the header and toggles between the two languages without any manual layout edits.

A single `ConfigureLocalization` action on `DocSiteOptions` enables multi-locale behavior. The default locale lives at the URL root; every other locale gets a folder prefix equal to its code. The LanguageSwitcher is already wired into DocSite chrome and stays hidden until a second locale is registered.

Prerequisites

- .NET 10 SDK installed
- Completed Scaffold a documentation site with DocSite (provides the single-locale DocSite host this tutorial extends)
- Completed Add doc pages and link between them (so the front-matter shape of each page is already familiar)

You'll work in the DocSite project from Scaffold a documentation site with DocSite. The finished version of every change lives in examples/BeyondLocaleExample — including the Spanish translations you'll author in section 3 — as a reference to check against.

---

1. Confirm the single-locale baseline

Your scaffold host serves markdown from `Content/` with no localization and no switcher. A clear baseline makes the contrast obvious when localization arrives in section 2.

There is no `ConfigureLocalization` action on `DocSiteOptions` yet, so `LocalizationOptions.IsMultiLocale` is false and the built-in `LanguageSwitcher` in `MainLayout.razor` renders nothing. The host you carry forward looks like this — the same `AddDocSite` shape from the scaffold tutorial:

CSHARP

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

builder.Services.AddDocSite(() => new DocSiteOptions
{
    SiteTitle = "Beyond Locale",
    SiteDescription = "Adding a second locale to a Pennington DocSite.",
    GitHubUrl = "https://github.com/usepennington/pennington",
    HeaderContent = """<a href="/">Beyond Locale</a>""",
    FooterContent = """<footer class="mt-16 py-8 text-center text-sm text-base-500">Built
with Pennington DocSite.</footer>""",
});

var app = builder.Build();

app.UseDocSite();

await app.RunDocSiteAsync(args);
```

Place three English pages directly under `Content/`

Add `index.md`, `about.md`, and `getting-started.md` directly under `Content/` — not in any locale subfolder. These are the default-locale pages, and they own the URL root.

MARKDOWN

```
---
title: Welcome
description: A DocSite homepage teaching Pennington localization.
order: 10
---

This site is written in two languages. The English version you're reading
lives under `Content/` — the default locale owns the URL root so its pages
serve from `/`, `/about`, and `/getting-started`.

Use the language switcher in the site header to jump to the Spanish version.
Every URL on this site has an equivalent in each configured locale, and the
`LanguageSwitcher` component in `MainLayout.razor` builds those links
automatically from the current request path.
```

MARKDOWN

```
---
title: About
description: About this localized DocSite example.
order: 20
---

This is a minimal DocSite that demonstrates **locale-aware URLs**. Every
markdown file under `Content/` is the English (default) version. Every
matching file under `Content/es/` is the Spanish translation.

When a visitor navigates to `/es/about`, `LocaleDetectionMiddleware` strips
the `/es` prefix, stores `"es"` in `LocaleContext`, and the DocSite's
`DocSiteContentResolver` picks up the Spanish markdown from `Content/es/about.md`.
If a Spanish file is missing, the resolver falls back to the English copy
and marks the page as a translation-fallback so the reader knows.
```

MARKDOWN

```
---
title: Getting Started
description: Get started with the localized DocSite example.
order: 30
---

To add a new locale to your own Pennington site:

1. Open `Program.cs` and call `loc.AddLocale(code, new LocaleInfo(displayName))`
   inside the `ConfigureLocalization` action on `DocSiteOptions`.
2. Create `Content/<code>/` and copy each page you want translated from the
   default-locale tree, translating the front matter `title:` and the body.
3. Run `dotnet run` — `LanguageSwitcher` appears in the site header as soon
   as `LocalizationOptions.Locales.Count > 1`.

There is no other wiring. The default locale keeps its URLs unchanged; every
additional locale gets a URL prefix equal to its code.
```

Checkpoint

- Run `dotnet run` from your project folder
- Visit `http://localhost:5000/`, `http://localhost:5000/about`, and `http://localhost:5000/getting-started` — each English page renders
- The DocSite header shows the site title and GitHub link but **no language switcher pill** — because only one locale is registered

---

2. Register a second locale with `ConfigureLocalization`

Add a `ConfigureLocalization` action that names `"en"` as the default and registers `"es"` as a second locale. Once LocalizationOptions.IsMultiLocale is `true`, the switcher, the locale detection middleware, and the per-locale search index all activate. `UseDocSite` already wires the locale-routing middleware internally — no extra `app.Use…` call.

Add the `ConfigureLocalization` action to your existing `DocSiteOptions`

The snippet below is your host with the change applied — the highlighted lines are the only additions. Add the `ConfigureLocalization` property inside the `DocSiteOptions` you already pass to `AddDocSite`, alongside the `SiteTitle`, `GitHubUrl`, and the rest, not replacing them. The `using Pennington.Localization;` directive at the top brings `LocaleInfo` into scope.

CSHARP

```
using Pennington.DocSite;
using Pennington.Localization;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDocSite(() => new DocSiteOptions
{
    SiteTitle = "Beyond Locale",
    SiteDescription = "Adding a second locale to a Pennington DocSite.",
    GitHubUrl = "https://github.com/usepennington/pennington",
    HeaderContent = """<a href="/">Beyond Locale</a>""",
    FooterContent = """<footer class="mt-16 py-8 text-center text-sm text-base-500">Built
with Pennington DocSite.</footer>""",

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

var app = builder.Build();

app.UseDocSite();

await app.RunDocSiteAsync(args);
```

The new action has three pieces:

- `DefaultLocale = "en"` — English owns the URL root with no prefix.
- `AddLocale("en", new LocaleInfo("English"))` — registers English with the display name the switcher shows.

- `AddLocale("es", new LocaleInfo("Español", HtmlLang: "es"))` — registers Spanish. `HtmlLang` is what Pennington emits on the `<html>` element for that locale's pages.

`AddLocale` is overloaded with a string-only display-name shorthand; the localization how-to surveys the full `LocalizationOptions` surface.

Checkpoint

- Rebuild and run the site (or let hot reload pick up the change)
- Refresh `http://localhost:5000/` — the DocSite header now shows a **language switcher pill** offering *English* and *Español*
- Click *Español* — the URL becomes `http://localhost:5000/es/` and you see a DocSite fallback notice explaining that Spanish content is missing, because no `Content/es/` files exist yet

---

3. Add translated markdown under `Content/es/`

Now let's give Spanish its content. Mirror the three English pages under a `Content/es/` subfolder — same file names, same front-matter keys, translated body copy. The content resolver matches each Spanish URL to the corresponding Spanish file.

Create `Content/es/` and translate `index.md`

Create the `Content/es/` subfolder and add `index.md` with Spanish front-matter and Spanish body copy. The rule that matters: **the subfolder name matches the locale code passed to `AddLocale`** — `es` here, because that is the code registered in section 2. Files under `Content/es/` serve from `/es/*`; files directly under `Content/` serve from `/*`.

MARKDOWN

```
---
title: Bienvenido
description: Página de inicio de un DocSite que enseña la localización de Pennington.
order: 10
---

Este sitio está escrito en dos idiomas. La versión en español que estás
leyendo ahora vive en `Content/es/` — cada idioma adicional tiene su propia
subcarpeta en el árbol de contenido y un prefijo de URL igual a su código
(`/es/`, `/es/about`, `/es/getting-started`).

Usa el selector de idioma en la cabecera del sitio para volver al inglés.
El componente `LanguageSwitcher` en `MainLayout.razor` construye los
enlaces automáticamente a partir de la ruta de la solicitud actual.
```

Translate `about.md` and `getting-started.md`

Repeat the move for the two remaining pages. Each Spanish file keeps the same filename as its English sibling; URL routing derives the path from the filename, not from any front-matter key.

Skipping a translation is fine. The content resolver falls back to the default-locale copy and renders a FallbackNotice banner naming the requested and default locales.

MARKDOWN

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

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

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

MARKDOWN

```
---
title: Primeros Pasos
description: Primeros pasos con el ejemplo de DocSite localizado.
order: 30
---

Para añadir un nuevo idioma a tu propio sitio Pennington:

1. Abre `Program.cs` y llama a `loc.AddLocale(code, new LocaleInfo(displayName))`
   dentro de la acción `ConfigureLocalization` en `DocSiteOptions`.
2. Crea `Content/<code>/` y copia cada página que quieras traducir del
   árbol del idioma predeterminado, traduciendo el `title:` del front
   matter y el cuerpo.
3. Ejecuta `dotnet run` — `LanguageSwitcher` aparece en la cabecera del
   sitio tan pronto como `LocalizationOptions.Locales.Count > 1`.

No hay más cableado. El idioma predeterminado mantiene sus URLs sin cambios;
cada idioma adicional obtiene un prefijo de URL igual a su código.
```

Checkpoint

- With the host still running, visit `http://localhost:5000/es/` — the page renders in Spanish with no fallback banner
- Visit `http://localhost:5000/es/about` and `http://localhost:5000/es/getting-started` — both serve Spanish translations
- Inspect the `<html>` element in dev tools on a Spanish page — `lang="es"` (from the `LocaleInfo.HtmlLang` set in section 2)

---

4. Use the built-in `LanguageSwitcher` to move between locales

The `LanguageSwitcher` component is already included in DocSite's `MainLayout.razor`. Now let's verify that it swaps locales in place by rewriting the current URL, landing on the same page in the other language rather than bouncing back to the home page.

Navigate to `http://localhost:5000/es/about`, open the language switcher in the header, and click *English*. The URL becomes `http://localhost:5000/about`. The switcher strips the `/es` prefix because English is the default locale and preserves the rest of the path, so the About page stays in view.

Checkpoint

- From `http://localhost:5000/es/about`, click *English* — the URL becomes `http://localhost:5000/about`
- From `http://localhost:5000/getting-started`, click *Español* — the URL becomes `http://localhost:5000/es/getting-started`
- From `http://localhost:5000/`, click *Español* — the URL becomes `http://localhost:5000/es/` (the default locale's root maps to the secondary locale's prefix root)

Summary

- A single-locale DocSite becomes multi-locale by adding one `ConfigureLocalization` action to `DocSiteOptions` — no explicit middleware call, no layout edits.
- The default locale owns the URL root and every other locale gets a code prefix equal to the string passed to `AddLocale`, with the matching `Content/<code>/` subfolder providing the translations.
- The `LanguageSwitcher` appears automatically once `LocalizationOptions.IsMultiLocale` is true, and it rewrites the current URL in place rather than redirecting to the home page.
- When a translation is missing, the content resolver falls back to the default-locale copy and renders a `FallbackNotice` banner naming the requested and default locales.

Author a custom Razor component for markdown

Getting Started Author a PricingCard Razor component inside a DocSite, register it with AddMdazorComponent(), and render it from a markdown page with two parameter sets.

By the end of this tutorial you'll have a running DocSite at `http://localhost:5000/pricing` that renders two styled `<PricingCard />` cards — a standard "Basic" tier and a highlighted "Pro" tier — both driven by tag attributes inside a plain markdown file.

Along the way, the tutorial covers authoring a Razor component with `[Parameter]`-decorated properties, wiring it into Mdazor's component registry with one `AddMdazorComponent<T>()` line, and consuming it from markdown with self-closing tag syntax whose attribute values bind case-insensitively to the component's parameters.

Prerequisites

- .NET 10 SDK installed
- Completed Scaffold a documentation site with DocSite (provides the `AddDocSite` / `UseDocSite` / `RunDocSiteAsync` host shape this tutorial extends)
- Basic Razor familiarity — a `.razor` file with `@code {}` and `[Parameter]` properties should feel routine

The finished code for this tutorial lives in examples/BeyondCustomRazorComponentExample.

---

1. Author the PricingCard component

Before Mdazor can render a custom tag from markdown, a real Razor component has to exist in your project. This unit adds `Components/PricingCard.razor` and a top-level `_Imports.razor` so `[Parameter]` is in scope without per-file `@using` lines.

Add a project-wide `_Imports.razor`

Drop an `_Imports.razor` file at your project root so every `.razor` file in the project gets the Blazor component namespaces. This is the same file a Blazor template ships with — the `@using` lines are what make `[Parameter]` resolve inside the component file in the next step, and the last line brings your `Components/` folder into scope so markdown can reference `PricingCard` by name.

RAZOR

```
@using Microsoft.AspNetCore.Components
@using Microsoft.AspNetCore.Components.Web
@using <your root namespace>.Components
```

> **Use your own root namespace.** Replace `<your root namespace>` with your project's default

> namespace (the `.csproj` name).

Create `Components/PricingCard.razor`

Create a `Components/` folder and add `PricingCard.razor` with four `[Parameter]` properties — `Tier`, `Price`, `Features`, and `Highlighted` — and markup that renders a pricing card, switching to a thicker accent border when `Highlighted` is set. The `Features` parameter is a pipe-delimited string because Mdazor binds only primitive parameter types from markdown attributes; lists arrive as strings and are split inside the component.

RAZOR

```
<div class="not-prose my-6">
    <div class="@CardClasses">
        <h3 class="text-xl font-bold">@Tier</h3>
        <div class="mt-2 flex items-baseline gap-1">
            <span class="text-4xl font-extrabold">$@Price</span>
            <span class="text-sm">/ month</span>
        </div>
        <ul class="mt-4 space-y-2 text-sm">
            @foreach (var feature in ParsedFeatures)
            {
                <li>@feature</li>
            }
        </ul>
    </div>
</div>

@code {
    [Parameter] public string Tier { get; set; } = "Basic";
    [Parameter] public string Price { get; set; } = "0";
    [Parameter] public string Features { get; set; } = "";
    [Parameter] public bool Highlighted { get; set; }

    private IEnumerable<string> ParsedFeatures =>
        (Features ?? "").Split('|', StringSplitOptions.RemoveEmptyEntries |
StringSplitOptions.TrimEntries);

    private string CardClasses => Highlighted
        ? "rounded-xl border-2 border-primary-500 p-6"
        : "rounded-xl border border-base-200 p-6";
}
```

The file is a regular Blazor component — nothing Pennington-specific yet.

Checkpoint

Run `dotnet build` from your project root. The build succeeds. The `PricingCard` type now exists at `<your root namespace>.Components.PricingCard`, but it is not yet wired to Mdazor, so a `<PricingCard />` tag in markdown would still render as a literal custom element.

2. Register the component with Mdazor

DocSite already calls `AddMdazor()` and registers the built-in Pennington.UI components. The only remaining step is one `AddMdazorComponent<PricingCard>()` line so Mdazor's registry knows about the new type.

Add `AddMdazorComponent<PricingCard>()` to `Program.cs`

Open `Program.cs` and add a single `builder.Services.AddMdazorComponent<PricingCard>()` line after the `AddDocSite` block. The extension lives in the `Mdazor` namespace and ships from the `Mdazor` NuGet package, already transitively referenced through `Pennington.DocSite` — no package add required.

CSHARP

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

builder.Services.AddDocSite(() => new DocSiteOptions
{
    SiteTitle = "Beyond Custom Razor Component",
    SiteDescription = "Authoring a Razor component and rendering it inline from markdown.",
    GitHubUrl = "https://github.com/usepennington/pennington",
    HeaderContent = """<a href="/">Beyond Custom Razor Component</a>""",
    FooterContent = """<footer class="mt-16 py-8 text-center text-sm text-base-500">Built
with Pennington DocSite.</footer>""",
});

// The one new line vs stage 1: tell Mdazor about the PricingCard type.
// AddMdazorComponent<T>() is an IServiceCollection extension in the
// Mdazor namespace (from the Mdazor NuGet package, transitively
// referenced through Pennington.DocSite). It returns the same
// IServiceCollection so it chains with further registrations.
builder.Services.AddMdazorComponent<PricingCard>();

var app = builder.Build();

app.UseDocSite();

await app.RunDocSiteAsync(args);
```

Checkpoint

Run `dotnet build` from your project root again. The build still succeeds, and Mdazor's registry now contains `PricingCard`. Nothing renders differently yet — the next section adds a markdown page that uses the tag.

3. Consume the component from markdown

Now let's add a markdown page that uses `<PricingCard />` twice with different attribute values, exercising both the default and highlighted visual states of the component.

Create `Content/pricing.md`

Add a new markdown page under `Content/` with front matter (`title: Pricing`, `description:`, `order: 20`) and two `<PricingCard ... />` tags between headings. The first card uses `Tier="Basic" Price="9"`; the second adds `Highlighted="true"` and richer feature text.

MARKDOWN

```
---
title: Pricing
description: Two PricingCard components rendered from markdown with distinct parameter
values.
order: 20
---

Pick a plan that fits your team. Both tiers below are rendered from a single
Razor component, `PricingCard`, authored in this example's `Components/`
folder and registered via `AddMdazorComponent<PricingCard>()` in
`Program.cs`. The markdown below consumes the component by name — Mdazor
intercepts tags that look like registered components, binds their
attributes as parameters, and hands the resulting HTML back to the Markdig
pipeline.

## Plans

<PricingCard Tier="Basic" Price="9" Features="1 project|5 GB storage|Community support" />

<PricingCard Tier="Pro" Price="49" Features="Unlimited projects|100 GB storage|Priority
email support|Team seats included" Highlighted="true" />

## Why two cards?

Rendering the component twice with different attribute values proves that
Mdazor resolves `<PricingCard />` tags on every occurrence, not just the
first. The second card passes `Highlighted="true"`, which flips the
component into its emphasised visual state — a thicker accent border.

## How the wiring works

1. The component is a regular Razor component with `[Parameter]`-decorated
   properties for `Tier`, `Price`, `Features`, and `Highlighted`.
2. `services.AddMdazorComponent<PricingCard>()` adds the type to Mdazor's
   component registry.
3. When the markdown renderer encounters `<PricingCard ... />`, it looks up
   the registered type, instantiates it, assigns parameters via
   case-insensitive reflection, renders the component through Blazor's
   server-side `HtmlRenderer`, and inlines the resulting HTML into the page.

Self-closing (`<PricingCard ... />`) and open/close (`<PricingCard ...></PricingCard>`)
forms are both supported; the open/close form lets the component receive
`ChildContent` populated by any markdown between the tags.
```

Mdazor matches tag names case-sensitively on the leading character — `<PricingCard>` must start with a capital letter — and binds attribute values to parameters case-insensitively. See Content components for the full binding rules.

Run the site

Start the host with `dotnet run` from your project root, then open `http://localhost:5000/pricing`. Mdazor intercepts each `<PricingCard ... />` tag, looks up the registered type, instantiates it, binds the attributes to its parameters, and inlines the rendered HTML in place of the tag.

Checkpoint

Visit `http://localhost:5000/pricing`. Two pricing cards appear: a **Basic** card at `$9 / month` with a thin border, and a **Pro** card at `$49 / month` with a thicker accent border (its `Highlighted="true"` attribute). View the page source — `<PricingCard>` has been replaced by real HTML (a `<div>` tree with the card classes), not left as a literal custom element.

---

4. Pass more parameters and verify binding

Now let's confirm the markdown-to-parameter binding is real by editing attribute values in the markdown and watching the rendered output change — this is the whole authoring loop.

Edit the Pro card to change `Price` and `Features`

In `Content/pricing.md`, change `Price="49"` to `Price="99"` and extend the `Features=""` string with an extra pipe-separated entry (for example, `"...|24/7 chat support"`). Save the file.

Flip `Highlighted` on the Basic card

Add `Highlighted="true"` to the first `<PricingCard Tier="Basic" ... />` tag. Boolean attribute values from markdown bind with case-insensitive `true` / `false` — `Highlighted="True"` and `Highlighted="true"` both flip the card into its emphasized state.

Checkpoint

Reload `http://localhost:5000/pricing`. The dev host picks up markdown changes as you save, so no rebuild is required.

- The Pro card now reads **$99 / month** and lists the extra feature bullet
- The Basic card now renders with the thicker accent border instead of its plain one
- Open the browser's dev tools — the generated HTML under each `<PricingCard>` has changed to match

---

Summary

- A Razor component lives under `Components/` with `[Parameter]`-decorated properties and is consumed from markdown by name.
- Any component type registers with Mdazor in one line: `services.AddMdazorComponent<T>()` after `AddDocSite` (or after `AddPennington` on a custom host).

- Two binding rules govern markdown-driven consumption: tag names start with a capital letter, and attribute values bind case-insensitively to parameter properties of primitive types (`string`, `bool`, numbers).
- Built-in Pennington.UI components and custom components mix freely in the same markdown page — both go through the same Mdazor registry.