Astro

A Complete 12-Chapter Course

Topics covered:

Zero-JS-by-Default & Islands · File-Based & Dynamic Routing · Layouts & Slots

Content Collections · Styling · Partial Hydration · Cross-Framework Components

Data Fetching & API Endpoints · Rendering Modes · Markdown & MDX

Exercises: 36 hands-on challenges with sample solutions

Format: A4 · Dark-theme code examples · React/Vue/Svelte comparison tables

Table of Contents

CHAPTER 1 OF 12

What Astro Is: The Content-First, Zero-JS-by-Default Framework

CHAPTER 1

What Astro Is: The Content-First, Zero-JS-by-Default Framework

HTML first, JavaScript only where you actually ask for it

React, Vue, Svelte, Angular, and Next.js all share one assumption underneath their real differences: the browser gets a JavaScript runtime, and the page is treated as an application. Astro starts from a genuinely different assumption — most of a real page is *content*, not application state, and the framework's own default output is plain, static HTML with **no JavaScript shipped at all**, unless you deliberately ask for some.

Zero JavaScript by Default

Every Astro component renders to plain HTML at build time — including components written in React, Vue, or Svelte and used inside an Astro page. Unlike Next.js, where a React component ships its own JavaScript to the browser and hydrates automatically, an Astro page with ten components on it can ship **zero** bytes of framework JavaScript by default. Nothing hydrates unless it's told to.

The Islands Architecture

Astro's own name for this model is the **islands architecture**: picture a page as a mostly static "ocean" of plain HTML, with small, isolated "islands" of interactivity dropped in only where genuinely needed — a like button, a search box, a carousel. Each island hydrates *independently*, on its own schedule, rather than the whole page hydrating as one unit the way a client-rendered React or Vue app does.

ASTRO VS. NEXT.JS — BOTH META-FRAMEWORKS, GENUINELY DIFFERENT ASSUMPTIONS

Both ship file-based routing and both can render on the server. But Next.js is fundamentally a **React** framework — every component is a React component, and the framework assumes hydration is the point. Astro is framework-agnostic at the page level: a single Astro page can mix plain Astro components, a React component, and a Svelte component, with the page itself owning zero runtime unless specific islands opt into one — covered fully in Chapter 7.

Why "Content-First"?

Astro was built with blogs, documentation sites, marketing pages, and portfolios in mind — sites where the overwhelming majority of the page is content that never changes after it's rendered, and only a handful of small pieces are genuinely interactive. That's a real, different sweet spot from a framework built assuming the whole page is one interactive application from the start.

> **NOT "ONLY FOR STATIC BLOGS"**
> Astro's own reputation as a static-site tool is only half the picture. Astro also supports full server-side rendering and a hybrid mode mixing both — covered in Chapter 10. The zero-JS-by-default philosophy holds regardless of which rendering mode a given deployment uses.

Four Frameworks, One New Assumption

CONCEPT	REACT / VUE / SVELTE	ASTRO
Ships JS by default?	Yes — a framework runtime, always	No — plain HTML unless a component opts in
Hydration model	Whole page/app, at once	Per-component "islands," independently
Framework lock-in per page	One framework for the whole app	React, Vue, and Svelte components can coexist on one page
Sweet spot	Interactive applications	Content-heavy sites with occasional interactivity

Creating a Project

```
# scaffold a new Astro project
npm create astro@latest

cd my-astro-site
npm run dev
```

The scaffolding wizard offers a few starter templates — an empty project is the clearest one to learn from. `npm run dev` starts the dev server with live reload, the same as every other framework in this series.

- `src/pages/` — every file here becomes a real page, by convention (Chapter 3)
- `src/components/` — reusable `.astro` components
- `src/layouts/` — shared page shells (Chapter 4)
- `astro.config.mjs` — the project's own configuration file

Coding Challenges

CHALLENGE 1

Scaffold a new Astro project, then edit src/pages/index.astro so it renders a heading interpolating a name from a frontmatter variable.

View solution

CHALLENGE 2

Build a Card.astro component that accepts a title prop, use it inside index.astro, and confirm via your browser's View Source that no JavaScript was shipped for it.

View solution

CHALLENGE 3

Add a second page, src/pages/about.astro, with no route configuration written anywhere, and confirm it's served automatically at /about.

View solution

CHAPTER 1 QUICK REFERENCE

- **Zero JS by default** — every component renders to static HTML unless deliberately hydrated
- **Islands architecture** — small, independent interactive components in an otherwise static page
- **Framework-agnostic** — React, Vue, and Svelte components can all live on one Astro page
- **Content-first** — built for blogs, docs, and marketing sites, not assumed-interactive apps
- **Not static-only** — SSR and hybrid rendering exist too, covered in Chapter 10
- **npm create astro@latest** / **npm run dev** — scaffold and run
- **Next chapter:** the `.astro` file and component syntax

CHAPTER 2 OF 12

Project Setup & the .astro File

CHAPTER 2

Project Setup & the .astro File

The frontmatter fence, expression-based markup, and props — the anatomy of one Astro component

Every `.astro` file has two parts: a **frontmatter fence** at the top for plain JavaScript or TypeScript, and a **markup** section below it. Both parts behave differently from every sibling framework in this series in ways worth understanding precisely, not just by analogy.

The Frontmatter Fence: Runs Once, Not Reactive

```
---
// this is the frontmatter — plain JS/TS
const title = 'My Site';
const items = ['Alpha', 'Beta', 'Gamma'];
---

<h1>{title}</h1>
```

The code between the two `---` lines runs exactly once — at build time for a static page, or once per request in server-rendered mode (Chapter 10) — and never again after that. There's no re-run on the client, because by default there's no client-side runtime at all. This is a genuinely different mental model from Svelte's own reactive `<script>` block, which re-runs logic in response to state changes in the browser.

THE CLOSEST REAL PARALLEL: REACT SERVER COMPONENTS

Among every framework covered in this series, an Astro frontmatter's "runs once, on the server, never again" behavior is closest in spirit to a Next.js React Server Component's own async function body — both execute server-side and produce static output with no client re-execution. The genuine difference is that in Astro, this behavior is the *default* for every component, not an opt-in mode layered onto an otherwise client-rendered framework.

Markup Expressions: JSX-Style, Not a Directive Language

```
---
const items = ['Alpha', 'Beta', 'Gamma'];
const showList = true;
---

<ul>
  {items.map((item) => <li>{item}</li>)}
</ul>

{showList && <p>The list is visible.</p>}
```

Astro's own markup expressions — `{ }` — contain real JavaScript expressions, evaluated directly: `.map()` for lists, `&&` or a ternary for conditionals. This is genuinely closer to React's JSX than to Vue's `v-for` / `v-if` directives or Svelte's own `{#each}` / `{#if}` block syntax — Astro doesn't introduce a separate template directive language at all; the markup expressions *are* JavaScript.

Props via `Astro.props`

```
---
// src/components/Badge.astro
interface Props {
  label: string;
  featured?: boolean;
}

const { label, featured = false } = Astro.props;
---

<span>{label}</span>
{featured && <strong> ★ Featured</strong>}
```

An optional `interface Props` gives typed, self-documenting props — genuinely useful in TypeScript projects, and entirely optional. Destructuring `Astro.props` in the frontmatter is the direct equivalent of a React function component's own props parameter.

Multiple Root Elements — No Wrapping Fragment Needed

```
<h2>Section Title</h2>
<p>Some content.</p>
```

Astro components can have multiple top-level elements natively — there's no historical single-root-element rule to work around the way early React needed `<Fragment>` / `<></>` for. Vue 3 and Svelte both allow this

too; it's React's own older constraint (long since solved by Fragments) that made this worth calling out explicitly.

Frontmatter and Markup, Compared Across Frameworks

CONCEPT	REACT (JSX)	VUE / SVELTE	ASTRO
Template expressions	Real JS, embedded directly	A separate directive language (`v-for`, `{#each}`)	Real JS, embedded directly — same style as JSX
Script re-runs?	Every render (client)	Reactively, on state change	Once, at build/request time — never again by default
Multiple root elements	Needs a Fragment	Allowed natively	Allowed natively

Coding Challenges

CHALLENGE 1

Build a List.astro component that accepts an items array prop and renders it as a <ul> using a .map() expression directly in the markup.

View solution

CHALLENGE 2

Add an optional featured boolean prop with a typed Props interface, and conditionally render a badge using a {condition && ...} expression.

View solution

CHALLENGE 3

Build a component with two sibling top-level elements (no wrapping div), and confirm it renders correctly with no Fragment needed.

View solution

CHAPTER 2 QUICK REFERENCE

- **Frontmatter fence (`---`)** — plain JS/TS, runs once, never again on the client
- **Markup expressions (`{ }`)** — real JavaScript, JSX-style, not a separate directive language
- `Astro.props` — destructure props in the frontmatter, optionally typed via `interface Props`
- **No wrapping Fragment needed** — multiple top-level elements are allowed natively
- **Closest real parallel** — a Next.js React Server Component's own server-only execution model
- **Next chapter:** file-based routing and dynamic routes

CHAPTER 3 OF 12

File-Based Routing & Dynamic Routes

CHAPTER 3

File-Based Routing & Dynamic Routes

src/pages/, [slug].astro, [...path].astro, and getStaticPaths()

Chapter 1 briefly showed a plain static file in `src/pages/` becoming a real route. This chapter covers the rest of the picture: dynamic segments, catch-all routes for arbitrary-depth paths, and the one real constraint every dynamic route runs into by default — `getStaticPaths()`.

Static Routes, Recapped

- `src/pages/index.astro` → `/`
- `src/pages/about.astro` → `/about`
- `src/pages/blog/index.astro` → `/blog`

Folders nest naturally into path segments — no route configuration file exists anywhere in an Astro project.

Dynamic Single-Segment Routes

```
// src/pages/blog/[slug].astro
---
export async function getStaticPaths() {
  return [
    { params: { slug: 'first-post' } },
    { params: { slug: 'second-post' } },
  ];
}

const { slug } = Astro.params;
---

<h1>{slug}</h1>
```

`[slug].astro` matches any single path segment at that position — `/blog/first-post`, `/blog/second-post` — captured into `Astro.params.slug`.

WHY GETSTATICPATHS() IS REQUIRED HERE — A REAL CONSTRAINT, NOT A FORMALITY

Astro's default output mode is fully static: the entire site is pre-rendered to plain HTML files at build time, with no server running afterward to resolve a route on demand. That means Astro has to know, at build time, *every single value* the dynamic segment could ever take — `getStaticPaths()` is exactly that declaration. Visit a URL whose value wasn't returned by `getStaticPaths()`, and there's simply no pre-built file for it — a real 404, not a route that resolves lazily. Chapter 10's server output mode removes this constraint entirely, resolving dynamic routes per-request instead.

Catch-All Routes: Arbitrary-Depth Paths

```
// src/pages/docs/[...path].astro
---
export async function getStaticPaths() {
  return [
    { params: { path: 'guides/getting-started' } },
    { params: { path: 'reference/config' } },
  ];
}

const { path } = Astro.params;
---

<h1>{path}</h1>
```

`[...path].astro` is Astro's own catch-all segment — the three dots match zero or more path segments, slashes included, all captured into a single string on `Astro.params.path`. Same requirement applies: every real deep path this route should serve has to appear in `getStaticPaths()`'s own returned list.

DELIBERATE GROUNDWORK FOR A FUTURE REBUILD COURSE

This exact pattern — a catch-all segment resolving an arbitrary-depth path — is the same problem the site's own Website Rebuild series solved once per framework: Next.js's own `[...path]` folder convention, Django's `<path:full_path>` converter, Laravel's `{path?}` plus a regex constraint, and Rails' own `*path` glob. Astro's `[...path].astro` is this course's own version of the identical idea — the piece a future Website Rebuild with Astro course would build directly on.

No Client-Side Router by Default

Unlike a single-page application framework, Astro doesn't ship a client-side router at all by default — navigating between pages is a genuine full page load, since there's no persistent client-side app to route within. An optional View Transitions API exists for smoother navigation animations, but it's a later, opt-in refinement, not a default routing mechanism the way React Router or Vue Router are.

Coding Challenges

CHALLENGE 1

Build a dynamic route at src/pages/products/[id].astro, with getStaticPaths() returning at least 3 hard-coded IDs, rendering the current ID via Astro.params.

View solution

CHALLENGE 2

Build a catch-all route at src/pages/docs/[...path].astro, with getStaticPaths() covering at least one multi-segment path, rendering the full captured path.

View solution

CHALLENGE 3

Build the site (npm run build) and confirm a URL not covered by getStaticPaths() has no corresponding output file — demonstrating why every real path has to be declared up front in static mode.

View solution

CHAPTER 3 QUICK REFERENCE

- `src/pages/` — every file becomes a real route, folders nest into path segments
- `[slug].astro` — matches one dynamic path segment, captured via `Astro.params`
- `[...path].astro` — catch-all, matches an arbitrary-depth path in one segment string
- `getStaticPaths()` — required in static output mode; every real path must be declared up front, or it's a genuine 404
- **No client-side router by default** — navigation is a real page load unless View Transitions is added
- **Next chapter:** Layouts, Slots & Component Composition

CHAPTER 4 OF 12

Layouts, Slots & Component Composition

CHAPTER 4

Layouts, Slots & Component Composition

Shared page shells, the default and named <slot />, and nesting layouts

Every page written so far has repeated its own `<html>` / `<head>` / `<body>` boilerplate. A **layout** is just an ordinary Astro component, conventionally kept in `src/layouts/`, that wraps a page's own content instead.

A Basic Layout

```
// src/layouts/BaseLayout.astro
---
interface Props {
  title: string;
}

const { title } = Astro.props;
---

<html lang="en">
  <head>
    <title>{title}</title>
  </head>
  <body>
    <slot />
  </body>
</html>
```

```
// src/pages/index.astro
---
import BaseLayout from '../layouts/BaseLayout.astro';
---

<BaseLayout title="Home">
  <p>Page content goes here.</p>
</BaseLayout>
```

Whatever's placed between `<BaseLayout>` and `</BaseLayout>` in the page renders wherever `<slot />` appears inside the layout itself. `title` is passed as an ordinary prop, same as any other component — a layout isn't a special kind of file, just a component used in this particular role by convention.

> **ASTRO'S SLOTS ARE THE REAL, NATIVE MECHANISM — NOT A FRAMEWORK INVENTION**
>
> Astro's `<slot />` is modeled directly on the native browser Web Component slotting API — the same underlying concept the HTML platform itself provides. React's `children` is just a plain prop with no dedicated syntax; Vue's own `<slot>` element is closer in spirit to Astro's own, both drawing from the same native idea.

Named Slots

```
// src/layouts/BaseLayout.astro (body)
<body>
  <aside>
    <slot name="sidebar" />
  </aside>
  <main>
    <slot />
  </main>
</body>
```

```
<BaseLayout title="Home">
  <p slot="sidebar">Sidebar content</p>
  <p>Main content — goes into the default slot</p>
</BaseLayout>
```

An element with a `slot="name"` attribute is routed into the matching named `<slot name="..." />`; everything else falls into the unnamed default slot.

Nesting Layouts

```astro
// src/layouts/BlogPostLayout.astro
---
import BaseLayout from './BaseLayout.astro';

interface Props {
  title: string;
  publishDate: string;
}

const { title, publishDate } = Astro.props;
---

<BaseLayout title={title}>
  <p class="date">{publishDate}</p>
  <slot />
</BaseLayout>
```

A layout can wrap another layout — `BlogPostLayout` adds blog-specific chrome (a publish date) around its own `<slot />`, while still passing through to `BaseLayout` for the shared `<html>` shell. Layouts compose the same way any other Astro components do.

Slots, Compared Across Frameworks

FRAMEWORK	MECHANISM
React	The `children` prop — a plain JavaScript value, no dedicated syntax
Vue	`<slot>`, named via `<template #name>`
Svelte 5	Snippet props rendered via `{@render children()}` — replaced the older `<slot>` mechanism used in Svelte 3/4
Astro	`<slot />` and named `<slot name="..." />`, modeled on the native Web Component API

Coding Challenges

CHALLENGE 1

Build BaseLayout.astro with a full HTML shell, a title prop for the <title> tag, and a default <slot />, then use it from index.astro.

View solution

CHALLENGE 2

Add a named "sidebar" slot to BaseLayout, and pass sidebar content into it from a page using slot="sidebar", alongside default-slot main content.

View solution

CHALLENGE 3

Build a BlogPostLayout.astro that wraps BaseLayout and adds a publishDate prop rendered above its own <slot />, then use it from a blog post page.

View solution

CHAPTER 4 QUICK REFERENCE

- `src/layouts/` — an ordinary component used by convention to wrap a page's own content
- `<slot />` — the default insertion point, modeled on the native Web Component slotting API
- `<slot name="..." />` / `slot="..."` — named slots for multiple distinct insertion points
- **Layouts are just components** — they take props and can wrap other layouts
- **Genuine cross-framework difference** — React's plain `children` prop vs. Vue/Astro's native-style `<slot>` vs. Svelte 5's newer snippet-based `{@render}`
- **Next chapter:** Content Collections

CHAPTER 5 OF 12

Content Collections: Structured Content with Type Safety

CHAPTER 5

Content Collections: Structured Content with Type Safety

defineCollection, Zod schemas, getCollection(), and an honest look at what this doesn't solve

Astro's own answer to "structured, validated content" is **Content Collections** — a real, first-class feature for organizing Markdown/MDX content into typed, schema-validated groups, kept in `src/content/`.

Defining a Collection

```
// src/content/config.ts
import { defineCollection, z } from 'astro:content';

const blog = defineCollection({
  type: 'content',
  schema: z.object({
    title: z.string(),
    publishDate: z.date(),
    tags: z.array(z.string()).optional(),
  }),
});

export const collections = { blog };
```

The `schema` is a real **Zod** object — Zod is a popular TypeScript-first schema validation library, used here to declare exactly what fields every entry's own frontmatter must have, and of what type. Every Markdown file in `src/content/blog/` gets validated against this schema.

TYPE SAFETY IS A BUILD-TIME GUARANTEE, NOT A SUGGESTION

If a blog post's frontmatter is missing `title`, or writes `publishDate` as plain text instead of a real date, Astro reports a real, build-time type error — not a silent runtime bug discovered later. TypeScript also infers the exact shape of every entry's own `data` object directly from this schema, so autocomplete and type-checking work correctly anywhere a collection entry is used in code.

Reading a Collection: `getCollection()`

```
// src/pages/blog/index.astro
---
import { getCollection } from 'astro:content';

const posts = await getCollection('blog');
---

<ul>
  {posts.map((post) => <li>{post.data.title}</li>)}
</ul>
```

`getCollection('blog')` returns every entry in the collection, each with a validated, typed `data` object matching the Zod schema above.

Rendering a Single Entry — Connecting Back to `getStaticPaths()`

```
// src/pages/blog/[slug].astro
---
import { getCollection } from 'astro:content';

export async function getStaticPaths() {
  const posts = await getCollection('blog');
  return posts.map((post) => ({
    params: { slug: post.slug },
    props: { post },
  }));
}

const { post } = Astro.props;
const { Content } = await post.render();
---

<h1>{post.data.title}</h1>
<Content />
```

Chapter 3's own `getStaticPaths()` can return a `props` object alongside `params` — whatever's passed here becomes directly available as `Astro.props` in the page, avoiding a second lookup. `post.render()` returns a real `<Content />` component rendering that entry's own Markdown body.

HONEST LIMIT: THIS DOESN'T SOLVE ARBITRARY-DEPTH TREES ON ITS OWN

Content Collections model flat or shallow grouped content well — a `blog` collection, a `docs` collection — but there's no built-in mechanism for a genuinely self-referencing, arbitrary-depth tree the way a real database and ORM provide. Zod's own `reference()` helper can link one entry to another (a `parent` field referencing a sibling entry), but that's a schema-level cross-reference, not the same thing as the adjacency-list-plus-materialized-path hybrid every course in the Website Rebuild series independently arrived at. A future Website Rebuild with Astro course would

genuinely need to choose between building that reference pattern by hand within Content Collections, or reaching for database-backed content in server output mode (Chapter 10) instead — not assume Content Collections alone already solve the same problem.

Coding Challenges

CHALLENGE 1

Define a docs collection with a Zod schema (title: string, order: number), add at least three content files, and use getCollection() to list all entries sorted by order.

View solution

CHALLENGE 2

Build a dynamic route rendering a single docs entry's own Content, passing the entry through getStaticPaths()'s props rather than looking it up a second time.

View solution

CHALLENGE 3

Deliberately break one content file's frontmatter (e.g. write order as text instead of a number), run the dev server, and report the exact build-time type error Astro produces.

View solution

CHAPTER 5 QUICK REFERENCE

- `src/content/config.ts` — defines every collection and its own Zod schema
- `defineCollection({ schema: z.object({...}) })` — validated, typed frontmatter
- `getCollection('name')` — returns every validated, typed entry in a collection
- `post.render()` — returns a `<Content />` component for that entry's Markdown body
- `getStaticPaths()` **'s own** `props` — passes data straight through, avoiding a second lookup
- **Honest limit** — no built-in self-referencing arbitrary-depth tree; a future rebuild needs `reference()` or database-backed content instead
- **Next chapter:** Styling in Astro

CHAPTER 6 OF 12

Styling in Astro

CHAPTER 6

Styling in Astro

Scoped by default, is:global to opt out, define:vars, and Sass support

A `<style>` block inside an `.astro` component is scoped to that component automatically — no explicit keyword needed, matching Svelte's own zero-config default rather than Vue's opt-in `scoped` attribute.

Scoped by Default

```
// src/components/Card.astro
<div class="card">Card content</div>

<style>
  .card { background: #161b22; padding: 1rem; }
</style>
```

Astro adds a unique attribute to this component's own elements at build time, so a different component's own `.card` rule never collides with this one — the identical technique, and the identical zero-runtime-cost result, Svelte's own scoped styles already used.

Opting Out: `is:global`

```
<style is:global>
  h1 { color: #FF5D01; }
</style>
```

`is:global` on a specific `<style>` block disables scoping for just that block, letting its rules apply site-wide from wherever the component happens to be used.

A Real Site-Wide Stylesheet

```
// src/layouts/BaseLayout.astro
---
import '../styles/global.css';
---
```

For genuinely site-wide styling — this project's own established dark theme, for instance — a plain imported `.css` file in a shared layout's frontmatter is the natural choice, applying unscoped to the whole page.

`define:vars`: Frontmatter Values Inside CSS

```
---
const accentColor = '#FF5D01';
---

<h2>Styled Heading</h2>

<style define:vars={{ accentColor }}>
  h2 { color: var(--accentColor); }
</style>
```

> **A GENUINE ASTRO-SPECIFIC CONVENIENCE**
>
> `define:vars` binds a frontmatter value directly into a real CSS custom property, usable anywhere inside that component's own scoped `<style>` block — no inline `style` attribute and no separate CSS-in-JS library required to bridge computed values into CSS.

Sass Support, Out of the Box

```
npm install sass
```

```
<style lang="scss">
  $accent: #FF5D01;
  .card {
    border: 1px solid $accent;
  }
</style>
```

Installing the `sass` package is the only setup required — no bundler configuration to write. `lang="scss"` on any `<style>` block enables real Sass syntax, still scoped by default the same as plain CSS.

Scoped Styles, Compared

FRAMEWORK	SCOPED BY DEFAULT?
React	No — needs CSS Modules or a separate styling library
Vue	Opt-in via `<style scoped>`
Svelte	Yes — automatic, zero configuration
Astro	Yes — automatic, zero configuration, same mechanism as Svelte

Coding Challenges

CHALLENGE 1

Build two components that each style an h3 tag differently in their own scoped <style> block, and confirm both render with their own distinct styling when placed on the same page.

View solution

CHALLENGE 2

Use define:vars to bind a frontmatter color variable into a CSS custom property, applied inside that component's own scoped style block.

View solution

CHALLENGE 3

Add a global.css stylesheet imported from a shared layout applying a site-wide dark background, and confirm a component's own scoped styles still layer correctly on top of it.

View solution

CHAPTER 6 QUICK REFERENCE

- **Scoped by default** — `<style>` is automatically scoped, same mechanism as Svelte, unlike Vue's opt-in `scoped`
- `is:global` — opts a specific style block out of scoping
- **Imported global stylesheet** — the right tool for genuinely site-wide styling
- `define:vars` — binds a frontmatter value into a real CSS custom property, no CSS-in-JS library needed

- **Sass** — `npm install sass` plus `lang="scss"`, no bundler config required
- **Next chapter:** Islands Architecture In Depth

CHAPTER 7 OF 12

Islands Architecture In Depth: Partial Hydration

CHAPTER 7

Islands Architecture In Depth: Partial Hydration

client:load, client:idle, client:visible, client:media, client:only

Chapter 1 claimed Astro ships zero JavaScript by default. Here's where that claim gets paid off concretely: **client directives** are the only mechanism that opts an individual component into hydration — without one, even a React or Svelte component renders to static HTML and nothing more.

No Directive: Static HTML, No Exceptions

```
---
import Counter from '../components/Counter.jsx';
---

<!-- renders the button's markup, but it's inert — no JS shipped -->
<Counter />
```

Even though `Counter` is a real React component with its own `useState`, without a client directive it's treated exactly like any Astro component: rendered once to HTML, then discarded. Clicking the button does nothing.

The Five Client Directives

```
<Counter client:load />
<Counter client:idle />
<Counter client:visible />
<Counter client:media="(max-width: 768px)" />
<Counter client:only="react" />
```

- `client:load` — hydrates immediately once the page loads. For genuinely critical, above-the-fold interactivity.
- `client:idle` — hydrates once the browser's main thread goes idle (via `requestIdleCallback`). For interactivity that matters but isn't urgent.

- `client:visible` — hydrates only when the component scrolls into the viewport (via `IntersectionObserver`). Ideal for anything below the fold.
- `client:media="(query)"` — hydrates only when a CSS media query matches — a mobile-only menu, for instance, that never hydrates at all on desktop.
- `client:only="react"` — skips server rendering entirely, rendering only in the browser. For components that depend on browser-only APIs that would error during the build.

THIS ISN'T JUST DEFERRED EXECUTION — THE JS NEVER EVEN DOWNLOADS UNTIL NEEDED

`client:visible` and `client:media` aren't only deferring *when* a component's JavaScript runs — they defer *whether its bundle is even downloaded at all*. An island using `client:visible` placed far down a long page genuinely doesn't fetch its own JS bundle until the user scrolls near it; a `client:media`-gated mobile menu never downloads its JS on a desktop visit at all. This is a real, measurable difference from a typical single-page application, which downloads its entire framework runtime and every component's code upfront regardless of whether the visitor ever scrolls that far or is on that device.

`CLIENT:MEDIA` IS GENUINELY DISTINCTIVE

Among every framework covered in this series, a conditional-on-a-media-query hydration directive is unique to Astro. React, Vue, and Svelte all have ways to conditionally render markup based on viewport size, but none has a first-class mechanism for conditionally shipping and hydrating a component's own JavaScript based on a media query — that's specifically an islands-architecture idea, not something a monolithic-runtime framework needs or offers.

Choosing the Right Directive

DIRECTIVE	HYDRATES WHEN	BEST FOR
`client:load`	Immediately	A critical, above-the-fold widget
`client:idle`	Main thread is idle	Important but non-urgent interactivity
`client:visible`	Scrolled into view	Anything below the fold
`client:media`	A media query matches	Device- or viewport-specific components
`client:only`	Client only, no SSR at all	Components using browser-only APIs

Coding Challenges

CHALLENGE 1

Add a React (or Vue/Svelte) counter component to an Astro page with client:load, and confirm the button actually increments when clicked.

View solution

CHALLENGE 2

Place the same counter far down a long page using client:visible instead, and confirm via your browser's Network tab that its JS bundle only loads once it's scrolled into view.

View solution

CHALLENGE 3

Build a mobile-only interactive component (e.g. a hamburger menu) using client:media, and confirm by resizing your browser that it only hydrates below the breakpoint.

View solution

CHAPTER 7 QUICK REFERENCE

- **No directive** — static HTML only, even for a React/Vue/Svelte component; nothing hydrates
- `client:load` — hydrates immediately
- `client:idle` — hydrates when the main thread is idle
- `client:visible` — hydrates on scroll into view; defers the JS *download*, not just execution
- `client:media="(query)"` — hydrates only when a media query matches; unique among this series' frameworks
- `client:only="react"` — client-only rendering, skips SSR entirely
- **Next chapter:** Using React, Vue, or Svelte Components Inside Astro

CHAPTER 8 OF 12

Using React, Vue, or Svelte Components Inside Astro

CHAPTER 8

Using React, Vue, or Svelte Components Inside Astro

Integrations, the serializable-props boundary, and how cross-framework composition actually works

This chapter builds directly on this site's own existing `react1`–`react4`, `vue1`, and `svelte1` courses — it covers the *integration boundary* between Astro and those frameworks, not the frameworks' own syntax, which those courses already teach in full.

Adding a Framework Integration

```
npx astro add react
npx astro add vue
npx astro add svelte
```

```
// astro.config.mjs
import { defineConfig } from 'astro/config';
import react from '@astrojs/react';
import vue from '@astrojs/vue';
import svelte from '@astrojs/svelte';

export default defineConfig({
  integrations: [react(), vue(), svelte()],
});
```

`npx astro add [framework]` installs the needed packages and wires up `astro.config.mjs` automatically.

A GENUINELY UNUSUAL FACT: ALL THREE CAN COEXIST IN ONE PROJECT

Every other framework covered in this series requires committing to exactly one — a React app is React, a Vue app is Vue. Astro genuinely doesn't: `react()`, `vue()`, and `svelte()` can all be active integrations in the same project simultaneously, with a single page importing a `.jsx` React component, a `.vue` Vue component, and a `.svelte` Svelte component side by side, each hydrating independently as its own island (Chapter 7).

Using a Framework Component

```
---
import ReactCounter from '../components/ReactCounter.jsx';
---

<ReactCounter initialCount={5} client:load />
```

Props are passed as ordinary attributes, the same as any Astro component. Each framework's own file extension (`.jsx` / `.tsx` , `.vue` , `.svelte`) is imported directly — no wrapping needed.

A Real Constraint: Props Must Be Serializable

> **NOT EXACTLY "JUST LIKE NORMAL PROPS"**
>
> Props crossing from Astro's own frontmatter into a framework island are serialized to cross the server-render boundary — plain data (strings, numbers, arrays, plain objects) survives correctly, but a function or a class instance passed as a prop does not transfer the way it would inside a pure React or Vue app. Any event handling logic needs to live *inside* the framework component itself, not be passed in from the Astro side as a callback prop.

A Second Real Constraint: No Direct Cross-Framework Nesting

A React component cannot directly contain a Vue component as a JSX child, and vice versa — the two framework runtimes don't compose that way. Composition across frameworks happens at the **Astro layer** instead: an Astro component can wrap a framework island and use `<slot />` (Chapter 4) to let Astro-level content — including other framework islands — surround it.

```
---
// src/components/Panel.astro
---

<div class="panel">
  <slot />
</div>
```

```
---
import Panel from '../components/Panel.astro';
import ReactCounter from '../components/ReactCounter.jsx';
import VueToggle from '../components/VueToggle.vue';
---

<Panel>
  <ReactCounter client:load />
  <VueToggle client:load />
</Panel>
```

Both islands sit side by side inside `Panel`'s own `<slot />` — real, valid composition, achieved entirely at the Astro layer rather than one framework component containing the other directly.

> **FOR THE FRAMEWORK SYNTAX ITSELF**
>
> This chapter deliberately stops at the integration boundary. For everything about how React's own hooks, Vue's own reactivity, or Svelte's own runes actually work, this site's existing `react1` – `react4`, `vue1`, and `svelte1` courses already cover that ground in full.

Coding Challenges

CHALLENGE 1

Add the React integration, then use a React component in an Astro page passing a plain numeric prop (e.g. initialCount={5}), confirming the value is received correctly.

View solution

CHALLENGE 2

Demonstrate the correct way to give a React island its own click handler — defined inside the component itself, not passed in as a function prop from Astro's frontmatter.

View solution

CHALLENGE 3

Build an Astro wrapper component with a <slot />, and use it to compose a React island and a Svelte (or Vue) island side by side inside it.

View solution

CHAPTER 8 QUICK REFERENCE

- `npx astro add [react|vue|svelte]` — installs and configures a framework integration
- **Multiple integrations can coexist** — a genuinely unusual trait among frameworks in this series
- **Props are serialized** — plain data survives; functions/class instances don't cross the boundary as props
- **No direct cross-framework nesting** — composition happens via an Astro wrapper's own `<slot />`
- **This chapter stops at the boundary** — see `react1`–`react4` / `vue1` / `svelte1` for the frameworks themselves
- **Next chapter:** Data Fetching & API Endpoints

CHAPTER 9 OF 12

Data Fetching & API Endpoints

CHAPTER 9

Data Fetching & API Endpoints

Build-time fetch(), src/pages/api/ endpoints, and client-side fetching in an island

There are three genuinely different ways to get data into an Astro site, each with its own timing and tradeoffs.

Fetching in the Frontmatter: Build Time, Not Request Time

```
---
const response = await fetch('https://api.example.com/posts');
const posts = await response.json();
---

<ul>
  {posts.map((post) => <li>{post.title}</li>)}
</ul>
```

A DIRECT CONSEQUENCE OF CHAPTER 2'S OWN FINDING

Chapter 2 established that the frontmatter fence runs once, not reactively. That's exactly why this `fetch()` call happens at **build time** in Astro's default static output mode — the response gets baked directly into the pre-rendered HTML. If the API's own data changes afterward, the site won't reflect it until the next build and deploy. This isn't a bug to work around; it's the same static-by-default model already established, just applied to a network request instead of a plain variable.

API Endpoints

```
// src/pages/api/posts.json.ts
export async function GET() {
  const posts = [
    { id: 1, title: 'First Post' },
    { id: 2, title: 'Second Post' },
  ];

  return new Response(JSON.stringify(posts), {
    headers: { 'Content-Type': 'application/json' },
  });
}
```

Any file in `src/pages/api/` exporting a named `GET` (or `POST`, `PUT`, etc.) function returning a real `Response` becomes a working API endpoint — here, reachable at `/api/posts.json`. In static output mode, this endpoint is also generated once at build time, the same constraint as the frontmatter fetch above; Chapter 10's server/hybrid modes are what let an endpoint like this run fresh on every request instead.

Client-Side Fetching Inside an Island

```
// src/components/LivePosts.jsx
import { useEffect, useState } from 'react';

export default function LivePosts() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    fetch('/api/posts.json')
      .then((r) => r.json())
      .then(setPosts);
  }, []);

  return (
    <ul>{posts.map((p) => <li key={p.id}>{p.title}</li>)}</ul>
  );
}
```

A hydrated island (Chapter 7) can fetch data itself, in the browser, at runtime — genuinely fresh every time the component loads, regardless of when the site was last built. The real cost is exactly what Chapter 7 already established: this requires shipping and hydrating real client-side JavaScript, unlike the frontmatter fetch's zero-JS static output.

Three Approaches, Compared

APPROACH	RUNS WHEN	SHIPS JS?	DATA FRESHNESS
Frontmatter `fetch()`	Build time, once	No	Frozen until next build
API endpoint (static mode)	Build time, once	No (unless called from an island)	Frozen until next build
Client-side fetch in an island	Every time the component loads, in the browser	Yes	Always current

Coding Challenges

CHALLENGE 1

Fetch data from a public placeholder API inside a page's frontmatter, render it as a list, then rebuild the site and confirm the rendered output only changes on a fresh build, not on every page load.

View solution

CHALLENGE 2

Build a src/pages/api/posts.json.ts endpoint returning a hard-coded JSON array via a GET function, and confirm it's reachable directly by visiting /api/posts.json.

View solution

CHALLENGE 3

Build a client-side-fetching island (client:load) that fetches from your own /api/posts.json endpoint at runtime, and explain how this differs from Challenge 1's frontmatter-fetch approach.

View solution

CHAPTER 9 QUICK REFERENCE

- **Frontmatter** `fetch()` — runs once at build time, result frozen into static HTML
- `src/pages/api/*.ts` — a named `GET` / `POST` export returning a `Response` becomes a real endpoint
- **Both are static in default mode** — Chapter 10's server/hybrid modes remove this constraint
- **Client-side fetch in an island** — always fresh, but requires shipping real JavaScript
- **Next chapter:** Rendering Modes: Static, Server & Hybrid

CHAPTER 10 OF 12

Rendering Modes: Static, Server & Hybrid

CHAPTER 10

Rendering Modes: Static, Server & Hybrid

output: 'static' vs 'server', adapters, and per-page opt-out with prerender

Every earlier chapter used Astro's default: `output: 'static'`. This chapter covers the other side — `output: 'server'`, and mixing both per page — which removes several constraints already flagged along the way, including Chapter 1's own promise that Astro "isn't only for static blogs."

Static Output — the Default, Recapped

Every page pre-rendered to plain HTML files at build time. `getStaticPaths()` (Chapter 3) is required for any dynamic route, since every possible path has to be known up front — there's no server running afterward to resolve one on demand.

Server Output: Per-Request Rendering

```
// astro.config.mjs
import { defineConfig } from 'astro/config';
import node from '@astrojs/node';

export default defineConfig({
  output: 'server',
  adapter: node({ mode: 'standalone' }),
});
```

`output: 'server'` renders every page fresh, per request. Astro itself doesn't ship a production server runtime — an **adapter** (`@astrojs/node`, `@astrojs/vercel`, `@astrojs/netlify`, and others) is what actually runs the server on a given platform, installed via `npx astro add node`.

CHAPTER 3'S GETSTATICPATHS() CONSTRAINT IS GONE

In server mode, a dynamic route like `[slug].astro` reads `Astro.params.slug` directly and resolves per request — no `getStaticPaths()` function needed at all. This is exactly the change Chapter 3's own finding-box promised: the "every path must be known at build time" constraint was specific to static output, not to Astro itself.

Mixing Both: Per-Page Opt-Out

```
// src/pages/dashboard.astro — even in a mostly static project
---
export const prerender = false; // opt this one page into server rendering
---
```

A single page can opt out of the project's own default mode via `export const prerender` — a mostly-static project can mark one genuinely dynamic page (a live dashboard, an admin panel) for server rendering, while everything else stays pre-built and fast. The reverse works too: in a server-output project, a page that's genuinely static (a privacy policy, an about page) can be marked `prerender = true` to skip per-request rendering for content that never changes.

GROUNDWORK FOR A FUTURE REBUILD COURSE'S OWN DEPLOYMENT CHAPTER

Every framework in the Website Rebuild series needed a real, dynamic admin CRUD interface — none of them could have built one with a purely static site generator. A future Website Rebuild with Astro course would need `output: 'server'` (or the per-page opt-out shown above) plus a real adapter, for exactly the same reason: an admin interface reading and writing live data has no meaningful static-build equivalent.

Three Rendering Approaches, Compared

	STATIC (DEFAULT)	SERVER	PER-PAGE OPT-OUT
When it renders	Once, at build time	Every request	Mixed, page by page
Needs an adapter?	No	Yes	Yes, for the dynamic pages
`getStaticPaths()` needed?	Yes, for dynamic routes	No	Only for the pages still using static rendering
Good fit for	Blogs, docs, marketing pages	An admin panel, live data	A content site with one genuinely dynamic section

Coding Challenges

CHALLENGE 1

Switch a project to output: 'server' with the Node adapter, remove getStaticPaths() from a dynamic route built in Chapter 3, and confirm it still resolves correctly per-request.

View solution

CHALLENGE 2

In that same server-mode project, add export const prerender = true to one specific static page, and confirm it's still pre-rendered at build time while the rest of the site renders per-request.

View solution

CHALLENGE 3

Build a POST API endpoint that only works correctly in server output mode, and explain concretely why a purely static build couldn't support it.

View solution

CHAPTER 10 QUICK REFERENCE

- `output: 'static'` — the default; pre-rendered once, requires `getStaticPaths()` for dynamic routes
- `output: 'server'` — per-request rendering; requires an adapter (`@astrojs/node`, `@astrojs/vercel`, etc.)
- `export const prerender` — opts a single page into or out of the project's own default mode
- **Server mode removes** `getStaticPaths()` **entirely** — dynamic routes resolve fresh, per request
- **Rebuild groundwork** — a future rebuild course's own admin interface needs server output, the same conclusion every sibling rebuild course already reached
- **Next chapter:** Markdown & MDX Content Authoring

CHAPTER 11 OF 12

Markdown & MDX Content Authoring

CHAPTER 11

Markdown & MDX Content Authoring

Plain Markdown as the default, MDX for the genuinely rare case that needs a live component mid-content

Chapter 5 covered Content Collections as a structure. This chapter covers what actually goes *inside* them — plain Markdown for the vast majority of content, and MDX for the real, specific case where a piece of content needs a live, interactive component embedded directly in the prose.

Plain Markdown

```
---
title: 'Getting Started with Islands'
publishDate: 2026-08-06
---

## Why Islands Matter

Astro ships zero JavaScript by default. Here's a code example:

```js
const count = 0;
```

Key Points

- No runtime by default
- Interactivity is opt-in
- Each island hydrates independently
```

Standard Markdown syntax — headings, lists, fenced code blocks — renders exactly as expected. Fenced code blocks get real syntax highlighting **automatically**, via Astro's own built-in Shiki integration — no separate highlighting library or configuration needed, unlike many other static site tools.

MDX: Markdown with Embedded Components

```
---
title: 'A Post With a Live Chart'
publishDate: 2026-08-06
---

import SalesChart from '../../components/SalesChart.jsx';

Here's some regular Markdown text, same as always.

<SalesChart client:visible />

More regular Markdown text continues right after the component.
```

`.mdx` files allow real `import` statements and real component tags directly inside otherwise-plain Markdown prose — requires the `@astrojs/mdx` integration (`npx astro add mdx`).

STILL NEEDS ITS OWN CLIENT DIRECTIVE

An embedded component inside MDX content doesn't hydrate automatically just because it's inside content — `SalesChart` above still needs `client:visible` (or another directive from Chapter 7) exactly the same as anywhere else in an Astro project. MDX changes *where* a component can be placed, not the rules governing whether it ships JavaScript.

NOT THE DEFAULT CHOICE

MDX is genuinely the exception, not the rule — the overwhelming majority of blog posts, docs pages, and articles need nothing beyond plain Markdown. Reach for MDX specifically when a piece of content needs a real, live component embedded mid-prose — not as a default file extension for every post.

Mixing `.md` and `.mdx` in One Collection

A single Content Collection can contain both `.md` and `.mdx` files side by side, as long as every entry — regardless of extension — satisfies the same Zod schema defined in `src/content/config.ts` (Chapter 5). `getCollection()` returns both kinds of entries together, with no special handling needed to distinguish them.

Markdown vs. MDX, Compared

	MARKDOWN (`.MD`)	MDX (`.MDX`)
Can embed live components?	No	Yes

	MARKDOWN (`.md`)	MDX (`.mdx`)
Requires an integration?	No, built in	Yes — `@astrojs/mdx`
Right for	The vast majority of prose content	The genuine, specific case needing a live component mid-content

Coding Challenges

CHALLENGE 1

Write a full Markdown blog post in a content collection with multiple headings, a list, and a fenced code block, and confirm it renders with proper syntax highlighting with zero extra configuration.

View solution

CHALLENGE 2

Convert that post to MDX and embed a real interactive island component (with a client directive) directly inside the prose, confirming both the text and the component render and work correctly.

View solution

CHALLENGE 3

Confirm a collection can mix .md and .mdx entries by adding one of each satisfying the same schema, and list both together via a single getCollection() call.

View solution

CHAPTER 11 QUICK REFERENCE

- **Plain Markdown** — the right default for the vast majority of content, built-in Shiki syntax highlighting
- **MDX (`.mdx`)** — real imports and component tags inside Markdown prose, via `@astrojs/mdx`
- **Embedded components still need a client directive** — MDX changes placement, not Chapter 7's own hydration rules
- **Not the default** — MDX is for the specific case, not every post
- **Mixed collections** — `.md` and `.mdx` can coexist in one collection, same schema, same `getCollection()` call
- **Next chapter:** Capstone — Building a Content-Driven Site

CHAPTER 12 OF 12

Capstone: Building a Content-Driven Site

CHAPTER 12

Capstone: Building a Content-Driven Site

One real blog, tying together every prior chapter

This capstone builds one real, working blog — every earlier chapter contributes a real, working piece of it, not just a concept in isolation.

```
// The finished project structure
src/
├── content/
│   ├── config.ts                // Ch5 — the blog collection's Zod schema
│   └── blog/
│       ├── zero-js-by-default.md    // Ch11 — plain Markdown
│       └── live-demo-post.mdx       // Ch11 — MDX with an embedded island
├── layouts/
│   └── BaseLayout.astro         // Ch4/Ch6 — shared shell, global styles
├── components/
│   └── NewsletterSignup.jsx     // Ch7/Ch8 — a React island
├── pages/
│   ├── blog/
│   │   ├── index.astro          // Ch5 — getCollection() listing
│   │   └── [slug].astro         // Ch3/Ch5 — getStaticPaths() + render()
│   └── api/
│       └── subscribe.ts         // Ch9/Ch10 — a real POST endpoint
└── astro.config.mjs             // Ch8/Ch10 — mdx, react, and server output
```

The Content Collection

```ts
// src/content/config.ts
import { defineCollection, z } from 'astro:content';

const blog = defineCollection({
  type: 'content',
  schema: z.object({
    title: z.string(),
    publishDate: z.date(),
  }),
});

export const collections = { blog };
```

Both `zero-js-by-default.md` and `live-demo-post.mdx` satisfy this same schema (Chapters 5 and 11).

Listing & Rendering Posts

```astro
// src/pages/blog/index.astro
---
import BaseLayout from '../../layouts/BaseLayout.astro';
import { getCollection } from 'astro:content';

const posts = await getCollection('blog');
---

<BaseLayout title="Blog">
  <ul>
    {posts.map((post) => (
      <li><a href={`/blog/${post.slug}`}>{post.data.title}</a></li>
    ))}
  </ul>
</BaseLayout>
```

```astro
// src/pages/blog/[slug].astro
---
import BaseLayout from '../../layouts/BaseLayout.astro';
import { getCollection } from 'astro:content';

export async function getStaticPaths() {
  const posts = await getCollection('blog');
  return posts.map((post) => ({
    params: { slug: post.slug },
    props: { post },
  }));
}

const { post } = Astro.props;
const { Content } = await post.render();
---

<BaseLayout title={post.data.title}>
  <h1>{post.data.title}</h1>
  <Content />
</BaseLayout>
```

This is Chapter 3's own `getStaticPaths()` pattern and Chapter 5's own `props`-passing shortcut, working together exactly as designed.

An Island for the Newsletter Signup

```jsx
// src/components/NewsletterSignup.jsx
import { useState } from 'react';

export default function NewsletterSignup() {
  const [email, setEmail] = useState('');
  const [sent, setSent] = useState(false);

  async function handleSubmit(e) {
    e.preventDefault();
    await fetch('/api/subscribe', {
      method: 'POST',
      body: JSON.stringify({ email }),
    });
    setSent(true);
  }

  return sent
    ? <p>Thanks — you're subscribed!</p>
    : (
      <form onSubmit={handleSubmit}>
        <input value={email} onChange={(e) => setEmail(e.target.value)} />
        <button>Subscribe</button>
      </form>
    );
}
```

Placed on the blog index below the fold via `<NewsletterSignup client:visible />` — Chapter 7's own genuine performance payoff: its JS never even downloads unless a visitor scrolls that far.

The API Endpoint — and Why It Forces a Rendering-Mode Decision

```ts
// src/pages/api/subscribe.ts
export async function POST({ request }) {
  const { email } = await request.json();
  // in a real app: validate and store the email
  return new Response(JSON.stringify({ status: 'ok' }), {
    headers: { 'Content-Type': 'application/json' },
  });
}
```

A real newsletter signup has to accept whatever email a visitor types, at the moment they submit it — exactly the case Chapter 10 identified as needing server output, not a static build. So the finished project's config reflects that decision:

```
// astro.config.mjs
import { defineConfig } from 'astro/config';
import react from '@astrojs/react';
import mdx from '@astrojs/mdx';
import node from '@astrojs/node';

export default defineConfig({
  integrations: [react(), mdx()],
  output: 'server',
  adapter: node({ mode: 'standalone' }),
});
```

A REAL TRADE-OFF, MADE DELIBERATELY

Choosing `output: 'server'` for the whole project means the blog index and post pages now render per-request instead of once at build time — a genuine cost, since Chapter 3's own `getStaticPaths()` isn't even necessary anymore in this mode. A more finely-tuned version of this same project could keep the blog pages statically pre-rendered (`export const prerender = true` on just those two files, per Chapter 10) while leaving only `subscribe.ts` dynamic — the right call depends on how much of the site is genuinely static versus genuinely dynamic, exactly the judgment call Chapter 10 raised.

Every Chapter's Own Contribution

PIECE	CHAPTER(S)
Zero-JS static rendering as the baseline	1
Frontmatter, props, markup expressions	2
Blog index/detail routing, `getStaticPaths()`	3
BaseLayout, `<slot />`	4
The blog Content Collection & schema	5
Scoped and global dark-theme styling	6
`client:visible` on the newsletter island	7
The React integration itself	8
The `/api/subscribe` POST endpoint	9
`output: 'server'` + the Node adapter	10
The MDX post with its own embedded island	11

Coding Challenges

CHALLENGE 1

Build this capstone's own blog collection, BaseLayout, index page, and [slug].astro route from scratch, with at least two real Markdown posts.

View solution

CHALLENGE 2

Add the NewsletterSignup island with client:visible, and the /api/subscribe endpoint it posts to, confirming a real submission returns { status: 'ok' }.

View solution

CHALLENGE 3

Refine the project by marking the blog index and [slug] pages export const prerender = true while leaving subscribe.ts dynamic, and confirm the site still builds correctly with this mixed approach.

View solution

CHAPTER 12 QUICK REFERENCE

- **Content Collections + Zod** — one schema, mixed `.md` / `.mdx` entries
- `getStaticPaths()` + `props` — dynamic routing without a second lookup
- `client:visible` — an island whose JS never downloads until scrolled into view
- **A real POST endpoint** — the concrete reason a project needs server output, not static
- **Per-page** `prerender` — the finer-grained alternative to an all-or-nothing rendering mode

★ Astro Course Complete — 12 / 12 chapters

From zero-JS-by-default and the islands architecture through Content Collections, styling, cross-framework composition, data fetching, and rendering modes. Chapters 3 and 5 in particular — arbitrary-depth routing and structured content — are the direct foundation a future Website Rebuild with Astro course can now build on.