Astro Course

Flavio Copes

Updated August 3, 2026

Contents

Learn Astro from the ground up: project structure, pages, components, layouts, props, data, content collections, assets, islands, and deployment.

What you'll learn: Build and deploy a small content-driven Astro site that sends browser JavaScript only for the parts that need interactivity.

Start an Astro project

Create a project, understand its folders, and run the toolchain.

Why Astro

Understand Astro as a web framework that renders useful HTML first and makes browser JavaScript an explicit choice.

Astro starts from the HTML the visitor needs.

An `.astro` component runs while Astro renders the page. That can happen during a production build or on a server request. Its component script does not become browser JavaScript.

```
---
const title = 'My field notes'
---

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

The browser receives an `h1`. It does not receive the `title` variable or an Astro runtime.

Add browser JavaScript only where behavior needs it. A page can remain mostly links, forms, images, and text while one search box or chart becomes an interactive island.

This makes Astro a strong fit for content sites, documentation, marketing pages, and stores where most of the page is read rather than continuously manipulated.

Astro does not make every page fast automatically. Large images, third-party scripts, poor caching, and unnecessary hydrated components still cost time. Its architecture makes those decisions visible.

Use Astro when HTML-first rendering matches the product. A dense offline application with continuous local state may be clearer in an application framework.

View the page source for this example. Confirm that the useful heading exists before any client

JavaScript runs.

Create an Astro project

Run the official project wizard, install the dependencies, and open the first page in a local development server.

Run Astro's project wizard:

```
npm create astro@latest
```

Choose a small starter for this course and install the dependencies. Then enter the project directory and start development:

```
cd astro-notes
npm run dev
```

Open the local URL printed by Astro. Edit `src/pages/index.astro`:

```
---
const title = 'Astro notes'
---

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

Save and confirm the page updates.

The development server renders the page when you request it. Later, the default production build will render the static route into a file. The same component can run at a different time depending on the output and route configuration.

If the page fails, read the first terminal error. An unclosed tag, missing import, and wrong directory are different problems.

Keep the terminal and browser open side by side. For every example in this course, check both the render error and the final browser output.

Understand the project structure

Know which folders Astro owns and which folders you create as the site grows.

An Astro project separates source files from files copied unchanged. Here is the shape of a small site:

```
astro-notes/
  public/
      robots.txt
  src/
      components/
      layouts/
      pages/
      assets/
  astro.config.mjs
  package.json
  tsconfig.json
```

`src/pages/` defines routes, and it is the one folder whose name Astro enforces: files here become URLs. `src/components/` and `src/layouts/` are useful conventions for reusable building blocks, not requirements, so you can reorganize them if a project demands it. Most people keep the standard names because every Astro developer recognizes them. Other source code, content, styles, and processed assets also live under `src/`.

Astro processes files in `src/`. It can bundle scripts and styles, optimize imported images, and report broken imports. This is the useful part: rename an image in `src/assets/` without updating the component that imports it, and the build fails with the exact file and line. The mistake cannot reach production.

Files in `public/` bypass that pipeline. Astro copies them unchanged to the output root, so `public/robots.txt` is served at `/robots.txt`. Use it for `robots.txt`, a web manifest, or a file that must keep an exact public path. The trade-off: a reference to a public file is only a string, so a typo there survives the build and shows up later as a 404.

Three files at the root round out the picture. `astro.config.mjs` holds project-wide configuration, such as integrations and build options. `package.json` records dependencies and the `dev`, `build`, and `preview` commands. `tsconfig.json` configures TypeScript support, which Astro projects have from the start.

A static production build normally writes to `dist/`. Treat that folder as disposable output: never edit it by hand, because the next build overwrites it.

Move one image between `public/` and `src/`. Notice how its reference and build handling change, and which of the two locations catches a broken path at build time.

Develop, build, and preview

Use each Astro command for its intended stage and run the production build before deployment.

The three main commands answer different questions.

`npm run dev`

The development server favors fast feedback. A route is rendered when you open it locally.

`npm run build`

The production build checks imports and creates deployable output. In a static project, every known route is rendered to files under `dist/` by default.

`npm run preview`

Preview serves the built output locally. It lets you inspect the generated routes and assets without rebuilding on every request.

Development success is not build success. A dynamic route may be missing a path. A content entry may fail its schema. A source filename may differ only by case.

Preview is also not the real host. Redirect status codes, headers, and server routes can depend on the platform.

Run all three commands. Write down when the page code ran in each mode.

Configure Astro

Use the Astro configuration for site-wide build decisions, integrations, URL information, and output behavior.

A typical configuration imports `defineConfig`:

```
import { defineConfig } from 'astro/config'

export default defineConfig({
  site: 'https://notes.flaviocopes.com'
})
```

The `site` value gives Astro the deployment origin for features that need absolute URLs, such as sitemap entries and canonical URLs.

Integrations, redirects, output behavior, adapter settings, and build options also belong here. This file runs in the Astro toolchain, not in the visitor's browser.

Configuration changes can affect every route. Add one setting at a time and keep the reason close to the change.

Prefer the smallest configuration that expresses a real requirement. Defaults are easier to maintain than copied settings whose purpose is unclear.

Do not place runtime secrets in values that will be serialized into generated HTML or client bundles. Server-side configuration does not make every value used from it private.

Set `site`, build the project, and inspect one generated absolute URL.

Choose src or public for an asset

Put processed project assets under src and reserve public for files that must keep their exact name and bytes.

Choose an asset location from how the file should be built.

A public file keeps its name and bytes:

```
public/favicon.svg → /favicon.svg
public/downloads/guide.pdf → /downloads/guide.pdf
```

Reference it by its root URL. Do not write `/public/` in the browser URL.

A source asset can be imported:

```
---
import diagram from '../assets/request-flow.png'
---
```

Astro can check that the file exists, fingerprint the output, and process it through the asset pipeline.

Use `public/` when an exact path matters or processing is unwanted. Use `src/` for project images, scripts, and styles that benefit from imports, optimization, and build errors.

A broken public URL may survive the build because it is only a string. A broken source import fails early. That feedback is valuable.

Rename one public file and one imported source file. Compare when each mistake is reported.

Use TypeScript in Astro components

Type props and component code while remembering that the types disappear from runtime output.

Astro component scripts support TypeScript syntax inside the normal `.astro` extension.

```
---
interface Props {
  title: string
  published?: boolean
}

const { title, published = false } = Astro.props
---

<h2>{title}</h2>
{published && <p>Published</p>}
```

The interface checks component calls during development and build. It documents what the template expects.

Types disappear from runtime output. They do not validate a database response, form submission, environment value, or parsed JSON file.

If data crosses an untrusted boundary, validate it at runtime before treating it as the typed shape. Content collection schemas are one Astro-native example.

Keep types near the boundary they describe. A small `Props` interface in the component is often clearer than a distant shared type used once.

Pass a number to `title` and confirm the checker reports the call. Then parse invalid JSON and notice that a TypeScript annotation alone does not reject it.

Check your understanding: Astro project basics

Check project creation, folders, public files, commands, configuration, static output, and TypeScript support.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Pages and routing

Turn files and data into static routes and endpoints.

Create routes with files

Map files under src/pages to URLs without creating a separate route table.

Files under `src/pages/` become public routes. The file's path decides the URL:

```
src/pages/index.astro         → /
src/pages/about.astro         → /about/
src/pages/docs/install.astro → /docs/install/
```

Astro uses the file and folder names as the route table. You do not register these pages in another configuration file, and there is no router object to maintain. Adding a route is creating a file. Removing a route is deleting one.

This is **file-based routing**, and its practical benefit goes beyond saving setup. The mapping works in both directions. Given a file, you know its URL. Given a URL, you know which file renders it. If `/docs/install/` fails, you should be able to find its source without searching the entire project — so keep the hierarchy obvious.

Pages can be `.astro`, Markdown, MDX when configured, HTML, or endpoint files such as `.ts`. The extension affects how the route produces its response: an `.astro` page renders a template, a Markdown page renders its content, a `.ts` endpoint returns whatever response your code constructs.

Try it. Create two new routes:

```
mkdir -p src/pages/guides
touch src/pages/contact.astro src/pages/guides/index.astro
```

Give each file a minimal heading, then open `/contact/` and `/guides/` in the dev server. Run a production build and find their generated output under `dist/` — one folder with an `index.html` per route in the default configuration.

Two things to watch. First, avoid creating two page files that compete for the same route, such as `src/pages/blog.astro` next to `src/pages/blog/index.astro`. Both want `/blog/`, and you should not have to remember which one wins. Keep one.

Second, remember that your host may normalize trailing slashes or filenames differently from development. A URL that works locally as `/about` may redirect or 404 in production depending on the platform. Verify important URLs on the deployed site, not only in the dev server.

Write an Astro page

Combine a server-side component script with the HTML template it renders.

An Astro page is a component under `src/pages/` that answers a route with a complete HTML document. Like every Astro component, it has two parts: a component script and a template.

```
---
import SiteHeader from '../components/SiteHeader.astro'

const title = 'About'
---

<html lang="en">
  <head>
    <title>{title}</title>
  </head>
  <body>
    <SiteHeader />
    <main>
      <h1>{title}</h1>
    </main>
  </body>
</html>
```

The fenced script between the two `---` markers runs while Astro renders. It can import components, read props, load data, and calculate values. Astro removes this script from the HTML response, so nothing you write there reaches the visitor's browser. That makes it the right place for work you would never ship to a client, like reading files or preparing data.

The template below the fence describes the output. Expressions such as `{title}` are evaluated during rendering, and the resulting HTML is what gets sent.

What separates a page from an ordinary component is responsibility for the document. This file emits `<html>`, `<head>`, and `<body>` itself. A button component only produces its own fragment; a page owns the whole response for its URL. Once several pages repeat that shell, you extract it into a layout, which a later lesson covers.

The title is dynamic at build or request time, but it is not reactive in the browser. Changing a browser variable later will not rerender this Astro component. This trips up people arriving from React or Vue: the component script is not a live client runtime, it runs once per render on the server side.

If the page needs browser behavior, use a normal `<script>` tag or a hydrated framework component. Keep server-only work in the component script.

View the page source and search for `SiteHeader`, `const title`, and the final heading. The import and the variable are gone. Only the rendered HTML should remain, with the header component already expanded into its markup.

Use nested and index pages

Create clean directory routes and choose an index file when the URL should end at a folder.

An `index` page represents the folder URL. When a visitor requests `/blog/`, Astro answers with the index file inside the `blog` folder:

```
src/pages/blog/index.astro   → /blog/
src/pages/blog/archive.astro → /blog/archive/
```

This mirrors a familiar static-site structure and keeps related routes together. Web servers have served `index.html` for directory URLs for decades, and Astro's build output makes that literal. With the default static output, `src/pages/blog/index.astro` becomes `dist/blog/index.html`, and `archive.astro` becomes `dist/blog/archive/index.html`.

A deeper index works the same way, at any level:

```
src/pages/docs/api/index.astro → /docs/api/
```

When should you switch from a flat file to a folder with an index? Both `src/pages/blog.astro` and `src/pages/blog/index.astro` produce `/blog/`, so pick one form and stay with it. My advice is to create the folder as soon as a section grows a second page. Then `/blog/` and everything under it live in one directory, and the source tree reads like the sitemap.

Use folders when they reflect the public information hierarchy. The URL is part of the interface: `/docs/api/` tells the reader where they are before the page loads. Do not nest files only to organize the source if you do not want that nesting in the URL, because every folder under `src/pages/` becomes a visible URL segment.

Shared components do not belong under `src/pages/` unless they should become routes. Put a reusable blog card under `src/components/`, not beside page files where it could be mistaken for a

public page. If a helper file must sit near the pages that use it, prefix its name with an underscore, such as `src/pages/blog/_PostCard.astro`. Astro excludes underscore-prefixed files from routing.

The realistic failure here is a silent URL change. Moving `about.astro` into a folder moves its public address, and inbound links to the old URL start returning 404 without any build warning.

Move a page into a folder and run the build. Confirm the old URL no longer exists under `dist/`, then add a redirect when the URL was already public.

Declare a dynamic route

Capture a URL segment with square brackets when one template renders several related pages.

A dynamic segment lets one page template represent several URLs.

`src/pages/posts/[slug].astro`

The bracket name becomes a route parameter. A request for `/posts/hello-astro/` has a `slug` value of `hello-astro`.

Inside the page:

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

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

The parameter tells you which route matched. It is still input. Validate it before using it in a database query, file path, or authorization decision.

Rendering mode changes how the value becomes available. A static project needs the complete set of slugs during the build, provided by `getStaticPaths()`. An on-demand route can receive the current request parameter at runtime but needs a compatible adapter.

Use `[...slug].astro` for a rest parameter that can match several path segments. Choose it only when nested paths are part of the content model.

Create one dynamic route and try a known and unknown value. Decide whether the unknown path should be built, return a 404, or be handled on demand.

Generate static dynamic routes

Return params and optional props from getStaticPaths so Astro can build one page for each data item.

A static dynamic route must tell Astro every URL to build.

```
---
const posts = [
  { slug: 'hello-astro', title: 'Hello Astro' },
  { slug: 'islands', title: 'Understanding islands' }
]

export function getStaticPaths() {
  return posts.map(post => ({
```

```
    params: { slug: post.slug },
    props: { post }
  }))
}

const { post } = Astro.props
---

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

Each `params` object identifies a public route. `props` carries the data directly into that page render.

During a static build, Astro runs `getStaticPaths()` once, then renders the page for every returned entry. The browser never calls this function.

Parameter values must be strings, numbers converted for the route, or `undefined` where a rest parameter allows it. Keep the URL field stable and separate from a display title.

Returning no entry for a slug means no static page exists for it. A link to that URL becomes a production 404.

Avoid fetching the same data again inside every page render when it can travel through `props` from the path generation step.

Add a third post, build, and inspect the three generated routes. Then remove one entry and confirm its file disappears.

Create a page from Markdown

Use Markdown for content that maps directly to a route and choose content collections when many entries share one content model.

A Markdown file under `src/pages/` becomes a route.

```
---
title: Installation
description: Install the project locally.
---

# Installation

Run the project wizard, then start the development server.
```

At build time, Astro turns the Markdown into HTML. The frontmatter values are available to the page and any configured layout; they are not sent as a JavaScript object to the browser by default.

Direct Markdown pages are convenient for a few standalone documents whose file path is their URL.

Use a content collection when many posts, notes, or docs share one model. Collections add a schema, query API, and a clear place to create dynamic routes from entries.

Markdown does not remove the need for semantic structure. Use one clear main heading, descriptive links, accurate image text, and a layout that supplies document metadata.

Create one Markdown page and inspect its generated HTML. Then break a required layout value and see whether your current setup catches it.

Create a static endpoint

Return JSON, text, XML, or another response from a .js or .ts route file.

An endpoint returns a `Response` instead of page HTML.

```
export function GET() {
  return new Response(JSON.stringify({ ok: true }), {
    headers: { 'Content-Type': 'application/json' }
  })
}
```

Place this in a route such as `src/pages/api/status.ts`. In a static build, Astro runs the GET handler during the build and writes the response body to a file.

That static response cannot change per visitor. It also cannot handle a POST after deployment.

An on-demand endpoint runs for an incoming request. It can read request headers, validate input, and return current data, but it requires a compatible adapter and server runtime.

Return accurate headers and status codes. JSON needs a JSON content type. A missing record should not return a successful status with an error string.

Treat route parameters, headers, and request bodies as untrusted. Being inside an Astro endpoint does not provide authentication or validation automatically.

Build the static endpoint and open its output. Then change a value and notice it stays stale until the next build.

Handle redirects and missing pages

Create an explicit redirect and a useful custom 404 instead of leaving broken navigation ambiguous.

Redirect an old route when content has moved:

```
---
return Astro.redirect('/new-path/', 301)
---
```

Choose the status from the change. A permanent move usually uses 301 or 308. A temporary destination uses a temporary status.

Static redirect output depends on the adapter and host. Without platform support, a static build may produce an HTML redirect rather than an edge HTTP response. Test the deployed headers instead of assuming local navigation proves the status.

Create `src/pages/404.astro` for missing pages. Include a clear explanation, the site navigation, and links back to useful sections. Do not turn every unknown URL into a 200 response showing "not found"; the HTTP status matters.

Redirects should preserve useful inbound links. Avoid chains such as old → older → current. Point every old URL directly to the final destination.

After deployment, request the old URL without automatically following redirects, then request an unknown URL. Verify the status, `Location` header, and final page.

Check your understanding: pages and routing

Check index and nested routes, dynamic segments, getStaticPaths, params, props, Markdown pages, endpoints, redirects, and 404 pages.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Components and layouts

Compose typed templates with props, slots, and scoped styles.

The anatomy of an Astro component

Separate render-time logic from the template and understand where each part executes.

An `.astro` file combines render-time JavaScript with an HTML template:

```
---
const name = "Ada"
const greeting = name.toUpperCase()
---

<h2>Hello {greeting}</h2>
```

The fenced script runs when Astro renders the component: during the build for a static page, or on the server for an on-demand page. It can import files, read props, fetch data, and calculate values. Its code is not sent to the browser.

The template uses those values to produce HTML. View the page source and you will see `<h2>Hello ADA</h2>`, not the JavaScript that created it.

This distinction is the foundation of Astro:

- code between the fences prepares the document
- template expressions decide which HTML is emitted
- a `<script>` tag runs later in the visitor's browser

Add `console.log("rendering")` inside the fenced script and another log inside a `<script>` tag. The first appears in the build or server terminal. The second appears in browser DevTools. That small experiment makes the two runtimes concrete.

Import and compose components

Build pages from small components without turning every HTML element into an abstraction.

Import a component in frontmatter, then use its capitalized name in the template:

```
---
import Card from "../components/Card.astro"
---

<section class="cards">
  <Card title="HTML first" />
```

```
  <Card title="JavaScript when needed" />
</section>
```

Astro resolves the import while rendering. Each `<Card>` produces ordinary HTML; the browser does not download a component runtime or fetch `Card.astro`.

Components are useful boundaries for repeated UI, a distinct responsibility, or a piece with its own styles and data. A `Header`, `ProductCard`, or `NewsletterForm` usually earns a component. A one-off paragraph usually does not.

Composition also makes data flow visible. The page chooses which cards exist and passes their values. The card owns how one item is presented. Neither needs to know the other's whole implementation.

After extracting a component, inspect the generated HTML. If the output becomes needlessly nested or the reader must jump between several files to understand simple markup, the abstraction is costing more than it saves. Astro makes components cheap at runtime, but they still have a maintenance cost for humans.

Pass component props

Give a component values from its parent and read them through Astro.props.

Props are how a parent passes values into a component. If you know React, Vue, or Svelte, the concept is the same, and Astro components support it too.

At the call site, props look like HTML attributes:

```
<Hello name="Flavio" />
```

Inside `src/components/Hello.astro`, the values arrive on `Astro.props`. You can read them directly in the template:

```
<p>Hello {Astro.props.name}!</p>
```

That works, but the common style is to destructure props into variables in the component script. It keeps the template clean when a component takes several values, and it gives you one place to set defaults:

```
---
const { name, message = 'Hello' } = Astro.props
---

<p>{message} {name}!</p>
```

The `message = 'Hello'` part is a default for a prop that might be unset. `<Hello name="Flavio" />` renders ”Hello Flavio!”, while `<Hello name="Flavio" message="Welcome" />` renders ”Welcome Flavio!”.

Quoted attributes are always strings. To pass anything else — a boolean, a number, an array, an object — use braces, which accept a JavaScript expression:

```
<Card title="First post" featured={post.isFeatured} />
```

Inside `Card.astro`, that value keeps its real type, so you can use it in logic, for example toggling a class:

```
---
const { title, featured = false } = Astro.props
```

```
---

<article class:list={{ featured }}>
  <h2>{title}</h2>
</article>
```

Watch the quoting mistake here. `featured="true"` passes the string `"true"`, not a boolean. The string is truthy, so the class still appears, but any comparison like `featured === true` fails. When the value is not text, use braces.

Props flow one way, from parent to child. The parent owns the data and the child owns its presentation. They are available while Astro renders the component; they do not create reactive browser state, so changing a value later in client-side code will not rerender anything.

Use a default for an optional prop, as with `featured = false`. For required values, fail clearly during development instead of silently rendering an empty heading. In the next lesson, TypeScript makes that contract explicit with a typed `Props` interface.

Type component props

Declare the component contract with a Props interface and provide defaults for optional values.

In the previous lesson, a component's props were an informal agreement. Nothing stopped a parent from omitting a required value or passing the wrong type. TypeScript turns that agreement into a checked contract, and the mechanism is one interface with a reserved name.

Astro recognizes an interface named `Props` in the component script:

```
---
interface Props {
  title: string
  featured?: boolean
}

const { title, featured = false } = Astro.props
---
```

The name matters. Declare `interface Props` and Astro applies it to `Astro.props` and to every place the component is used. Now `<Card featured="yes" />` is a type error, because a quoted attribute is a string and `featured` expects a boolean. Omitting `title` is a type error too. The interface documents the component at the point where people use it: hover a `<Card>` tag in the editor and you see exactly what it accepts.

Optional properties use `?`, and their defaults belong in the destructuring statement. Keep the type and the default consistent:

```
---
interface Props {
  title: string
  count?: number
}

const { title, count = 0 } = Astro.props
---
```

If you declare `count?: number` but default it to `'0'`, the annotation and the actual value disagree, and every consumer downstream inherits the confusion. The pattern above keeps the declaration and the fallback in one visible pair.

Where do the errors surface? In the editor as you type, and when you run `astro check` or a build that includes type checking. The mistake is expecting more than that. This is not runtime validation: the interface disappears from the compiled output. If props originate in a URL, form submission, CMS, or external API, validate that input before passing it to the component.

Think of `Props` as an internal programming contract, not a security boundary. It prevents accidental misuse by your code; it cannot make untrusted data truthful.

Add a `Props` interface to a component you already have, then deliberately break one call site with a wrong type. Confirm the editor flags the usage, not just the component file.

Use template expressions

Insert values, choose markup, and render arrays with JavaScript expressions.

Use braces for expressions:

```
<h1>{title}</h1>
{featured && <strong>Featured</strong>}
<ul>
  {items.map(item => <li>{item.name}</li>)}
</ul>
```

The expressions run while Astro renders the template. They choose the resulting HTML; they do not keep watching values in the browser. If `featured` changes later in client-side code, this block will not rerender by itself.

Use normal JavaScript before the template for anything that would make the markup hard to read:

```
---
const visibleItems = items.filter(item => !item.draft)
---

{visibleItems.length > 0 ? (
  <ul>{visibleItems.map(item => <li>{item.name}</li>)}</ul>
) : (
  <p>No items yet.</p>
)}
```

Astro resembles JSX, but it follows HTML more closely: use `class`, and multiple top-level elements are allowed. Values inserted with `{value}` are escaped, which is the safe default for untrusted text. Treat any API that deliberately inserts raw HTML as a security-sensitive exception.

Render the page, then view its source. You should be able to connect each branch and loop to the static HTML it produced.

Compose child markup with slots

Let a parent provide HTML content for a component to place inside its own structure.

A slot is a place where a component renders markup supplied by its parent. Here is a simple panel:

```
<!-- Panel.astro -->
<aside class="panel">
  <header><slot name="heading">Details</slot></header>
  <slot />
</aside>
```

The parent fills both slots:

```
<Panel>
  <h2 slot="heading">Shipping</h2>
  <p>Orders leave within two working days.</p>
</Panel>
```

`Details` is fallback content: Astro uses it only when the parent does not provide the named slot. The unnamed children go into the default `<slot />`.

Props pass values that the component interprets. Slots pass a piece of markup that the parent controls. Use a prop for `variant="warning"`; use a slot when callers need links, emphasis, or several elements.

In Astro 7, do not provide several sibling elements separately to the same named slot. Only one sibling assigned to that slot is rendered. Wrap the group in one fragment:

```
<Fragment slot="heading">
  <span>Shipping</span>
  <small>Updated today</small>
</Fragment>
```

Layouts use this same mechanism: the layout owns the document shell, while each route supplies its page-specific content.

Build a page layout

Share the complete document structure and page metadata while keeping each route responsible for its own content.

When a site grows past one page, shared HTML starts to duplicate. Every page repeats the doctype, the `<head>`, the navigation, the footer. Then you change something in the header and you are editing five files, hoping you caught them all.

A **layout** fixes this. A layout is an Astro component that owns the shared page structure while a route supplies the unique content. In other tools, like Hugo, you would use partials. In Astro it is all components, and by convention layouts live in `src/layouts/`.

Here is `src/layouts/Layout.astro`:

```
---
interface Props {
  title: string
  description: string
}

const { title, description } = Astro.props
---
```

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="description" content={description} />
    <title>{title}</title>
  </head>
  <body>
    <nav><a href="/">Home</a></nav>
    <main><slot /></main>
    <footer>Built with Astro</footer>
  </body>
</html>
```

The `<slot />` element marks where page content will render. A page imports the layout and uses it like any other component:

```
---
import Layout from "../layouts/Layout.astro"
---

<Layout title="About" description="Learn who built this site">
  <h1>About</h1>
  <p>This content fills the layout's slot.</p>
</Layout>
```

Everything between `<Layout>` and `</Layout>` replaces the slot. The props fill the metadata.

The split of responsibilities is the useful part. The page remains responsible for accurate metadata: it knows its own title and description. The layout remains responsible for emitting a valid, consistent document: one place owns the doctype, the language attribute, and the navigation.

Layouts compose, because they are components. A blog can create a `PostLayout` that imports the base `Layout`, adds the post header inside it, and passes `title` and `description` through. Pages then wrap themselves in the most specific layout they need.

Like other Astro components, the layout runs while rendering. It does not stay alive in the browser, so it adds no client-side JavaScript to the page.

A classic mistake is forgetting `<slot />`. The layout renders fine, and every page using it comes out with an empty `<main>`. If page content silently disappears inside a correct shell, check the layout for a missing slot before debugging the page.

View two generated pages and verify that their shell is shared but their title, description, and main content differ.

Style an Astro component

Use scoped component styles by default and make global styling an explicit decision.

A normal `<style>` tag belongs to the component:

```
<article class="card">
  <slot />
```

```
</article>

<style>
  .card {
    border: 1px solid currentColor;
    padding: 1rem;
  }
</style>
```

Astro rewrites the selector and adds a generated scope attribute to matching markup. Another component can use `.card` without accidentally receiving this rule.

Scoping follows component boundaries. A parent's scoped selector is not a reliable way to reach deep into a child component. Give the child a prop or class it deliberately supports, or move a genuinely shared rule into a global stylesheet.

Import global CSS from a layout for resets, design tokens, typography, and rules intentionally shared by the whole site. Use `:global()` only for a small, explicit escape from scoping; broad global selectors make components harder to reason about.

Inspect the rendered HTML and CSS once. The generated attributes explain why the rule is local and help diagnose selectors that do not reach the element you expected.

Add a browser script

Distinguish a script tag that Astro bundles for the browser from frontmatter that runs only while rendering.

Use a `<script>` tag when the finished HTML needs behavior in the browser:

```
<button data-menu-button aria-expanded="false">Menu</button>
<nav data-menu hidden>...</nav>

<script>
  const button = document.querySelector("[data-menu-button]")
  const menu = document.querySelector("[data-menu]")

  button?.addEventListener("click", () => {
    const open = button.getAttribute("aria-expanded") === "true"
    button.setAttribute("aria-expanded", String(!open))
    if (menu instanceof HTMLElement) menu.hidden = open
  })
</script>
```

Astro processes a normal script: imports can be bundled, TypeScript is supported, and repeated use of the component does not imply repeated copies of the same bundled script. An `is:inline` script is emitted as written and skips that processing, so use it only when inline behavior is specifically required.

This code is public. Never include secrets, private environment values, or authorization logic in it.

Prefer native HTML before JavaScript. Links should navigate with `<a>`, forms should submit with `<form>`, and disclosure behavior may fit `<details>`. Add a script for behavior HTML cannot provide clearly, then test the page with JavaScript disabled to see whether its essential content still

works.

Check your understanding: components and layouts

Check frontmatter, templates, props, typed contracts, slots, layouts, scoped styles, HTML attributes, roots, and browser scripts.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Data, content, and assets

Load data, validate content, and optimize project images.

Load data before rendering

Use top-level await in frontmatter and know whether the work happens during a build or an on-demand request.

An Astro component can await data before it produces HTML:

```
---
const response = await fetch("https://api.example.com/products")

if (!response.ok) {
  throw new Error(`Product request failed: ${response.status}`)
}

const products = await response.json()
---

<ul>
  {products.map(product => <li>{product.name}</li>)}
</ul>
```

There is no client-side loading state here. Astro waits for the fetch, renders the list, and sends the resulting HTML.

The important question is *when* the route renders. A static route fetches during `astro build`. Its data stays in the generated page until the next build. This is fast and resilient for visitors, but a price change in the API will not appear automatically.

An on-demand route fetches on the server for a request. Its data can be fresher, but visitors now depend on the API and your server response time.

Choose based on how stale the page may safely be. Then handle failures explicitly: check `response.ok`, decide whether the build should stop, and avoid silently publishing an empty page when an upstream service fails.

Keep server data private

Use private credentials while rendering without leaking them into markup, public files, or client scripts.

Frontmatter code runs while rendering, so it can use a private credential without shipping the credential's source code to the browser:

```
---
const response = await fetch("https://api.example.com/account", {
  headers: {
    Authorization: `Bearer ${import.meta.env.API_TOKEN}`
  }
})

if (!response.ok) throw new Error("Account request failed")

const account = await response.json()
const displayName = account.displayName
---

<p>Welcome {displayName}</p>
```

Only `displayName` is rendered. The token never belongs in the template.

The boundary is the output, not the file extension. A secret still leaks if you interpolate it into HTML, include it in a `data-*` attribute, log it into a public response, or pass it as a prop to a hydrated framework component. Everything delivered to browser JavaScript is public to the visitor.

Also distinguish build-time secrecy from user-specific authorization. A static page built with a private token is still the same public file for everyone. Private account pages require on-demand rendering plus real access control.

Return the smallest safe projection of external data. Before deploying, inspect the generated HTML and browser network responses for the secret's name and value.

Create a content collection

Group related content entries under one typed model rather than reading arbitrary files throughout page templates.

Content collections give repeated content one loading and validation boundary. They fit posts, documentation, products, and any group whose entries share a shape.

In Astro 7, define build-time collections in `src/content.config.ts`. A loader says where the data comes from:

```
import { defineCollection } from "astro:content"
import { glob } from "astro/loaders"

const notes = defineCollection({
  loader: glob({
    base: "./src/content/notes",
```

```
    pattern: "**/*.{md,mdx}"
  })
})

export const collections = { notes }
```

Each Markdown file under that directory becomes an entry with an `id`, its frontmatter in `data`, and renderable content. The collection name `notes` becomes the stable handle used by pages.

The loader is required. `glob()` loads separate local files, `file()` can turn one JSON, YAML, or TOML file into entries, and custom loaders can retrieve remote content.

This is better than scattering filesystem reads across page templates. Routes can ask for a typed collection without knowing its storage details. If the source later moves from Markdown to a CMS, that change stays at the loading boundary rather than spreading across every component.

Validate content data

Give content fields a schema so a missing title, invalid date, or wrong value fails before deployment.

A schema makes the assumptions in your templates executable. Add it beside the collection loader:

```
import { defineCollection } from "astro:content"
import { glob } from "astro/loaders"
import { z } from "astro/zod"

const notes = defineCollection({
  loader: glob({ base: "./src/content/notes", pattern: "**/*.md" }),
  schema: z.object({
    title: z.string(),
    description: z.string(),
    publishedAt: z.coerce.date(),
    topic: z.enum(["astro", "css", "javascript"]),
    draft: z.boolean().default(false)
  })
})

export const collections = { notes }
```

Astro validates entries as it loads the collection. A missing title, impossible topic, or malformed date stops the build close to the bad source instead of failing later inside a page.

Use required fields when every template needs the value. Use optional fields only for genuine variants, not as a way to avoid cleaning inconsistent data. Defaults are useful when an omitted value has one unambiguous meaning.

Break an entry deliberately: set `topic: cooking`, run the build, and read the error back to the exact field. Then restore it. A useful schema is strict enough to catch mistakes and accurate enough to describe real content.

Query and render a collection

Load entries, filter drafts, sort dates, and pass one consistent content shape into the page.

Once a collection is defined, pages query it with `getCollection()` from `astro:content`. The function returns an array of entries, and each entry carries an `id` plus its frontmatter under `data`.

Here is a listing page:

```
---
import { getCollection } from "astro:content"

const notes = await getCollection("notes", ({ data }) => !data.draft)

notes.sort(
  (a, b) => b.data.publishedAt.valueOf() - a.data.publishedAt.valueOf()
)
---

<ul>
  {notes.map(note => (
    <li><a href={`/notes/${note.id}/`}>{note.data.title}</a></li>
  ))}
</ul>
```

The second argument is a filter callback. Every entry it rejects disappears from the result, so drafts never produce a link here. Filter drafts before creating links or routes, not after, so unpublished URLs are not accidentally generated anywhere.

Do not rely on the loader's entry order; it reflects how files were read, not any publishing policy. Sort explicitly where the listing policy belongs, as the `publishedAt` comparison above does.

For a detail route, select the entry in `getStaticPaths()` and render its body:

```
---
import { getCollection, render } from "astro:content"

export async function getStaticPaths() {
  const notes = await getCollection("notes", ({ data }) => !data.draft)
  return notes.map(note => ({ params: { id: note.id }, props: { note } }))
}

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

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

`render()` compiles the entry's Markdown and hands back a `<Content />` component you place in the template like any other component. Each generated page receives its own entry through props, so the detail route never re-queries the whole collection at render time.

Keep selection and sorting near the route. Presentational components should receive a small, consistent shape instead of knowing how the entire collection is stored.

A mistake worth recognizing: forgetting `await` in front of `getCollection()`. You get a Promise

instead of an array, and the first symptom is usually `notes.sort is not a function` or an empty listing. Check the query line before suspecting the collection itself.

Build the site and count the generated note pages under `dist/notes/`. The number should match the entries that pass the draft filter, not the total files in the collection.

Use local images

Import source images and let Astro determine dimensions and create optimized output.

Images imported from `src/` enter Astro's asset pipeline:

```
---
import { Image } from "astro:assets"
import workshop from "../assets/workshop.jpg"
---

<Image
  src={workshop}
  alt="Two developers sketching a data flow on a whiteboard"
  widths={[480, 800, 1200]}
  sizes="(max-width: 700px) 100vw, 700px"
/>
```

Astro knows the imported image's dimensions and can generate optimized output. The dimensions reserve layout space before the file loads, reducing visual movement. Responsive widths let the browser choose an appropriately sized file.

Files in `public/` follow a different path: reference them with a root URL such as `/logo.svg`, and Astro copies them without source-image processing. Use `public/` when a file must keep an exact name or should bypass transformation. Use a source import when you want build-time validation and optimization.

Alternative text describes the image's purpose in this context, not its filename. Use `alt=""` for a decorative image already explained by nearby content. Optimization improves delivery; it does not make the image accessible.

Temporarily rename the imported file and run the build. The failure is useful: source imports turn broken asset references into build errors instead of production 404s.

Build a small content site

Connect collections, dynamic routes, layouts, components, and optimized images in one practical project.

Build a small notes site that connects the boundaries from this module.

Create `src/content/notes/` with three Markdown entries. Give each entry a title, description, publication date, topic, and draft flag. In `src/content.config.ts`, load the files with `glob()` and validate every field. Make one note a draft.

Then build two routes:

- `/notes/` queries the collection, excludes drafts, sorts newest first, and links by entry ID
- `/notes/[id].astro` uses `getStaticPaths()` to create one route per published note and

`render()` to display its body

Wrap both routes in a layout that accepts title and description props. Add a small card component for list items, but keep filtering and sorting in the route. Components present data; routes decide which content becomes public.

Import one local image through `astro:assets`, give it contextual alt text, and render responsive sizes. Do not move it to `public/` just to avoid the import—the build-time reference check is part of the value.

Now test the failure paths deliberately:

1. Give one entry an invalid topic and confirm that the schema identifies it.
2. Restore the topic, rename the image, and confirm that the asset import fails.
3. Restore the image and run the production build.
4. Inspect `dist/notes/` and confirm there is no page for the draft.

Finally, add a fourth published note and rebuild. The index and detail route should appear without changing either page template. That is the payoff of the collection: adding valid content becomes data work, while loading, validation, routing, and presentation remain stable.

Check your understanding: data, content, and assets

Check render-time data, private values, collections, schemas, queries, imported images, public assets, and static freshness.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Islands and deployment

Add focused interactivity, build the site, and deploy it.

Start with zero JavaScript

Inspect an Astro page as HTML first and add client code only for behavior the browser must perform.

Astro components produce HTML without shipping a client runtime by default. A loop, conditional, or imported component can make a rich document while leaving no JavaScript for the visitor to execute.

Before adding client code, ask what behavior is actually missing. The browser already provides:

- links for navigation
- forms and built-in validation for data submission
- `<details>` and `<summary>` for disclosures
- audio and video controls
- CSS for responsive layout and many visual states

These features work before a bundle downloads, support browser conventions, and usually survive assistive technology better than a custom imitation.

Inspect a plain Astro page in the Network panel. Filter requests by JavaScript and confirm there

is no framework runtime. Then add a `<details>` element and verify that it still opens and closes. Interactivity does not automatically mean JavaScript.

Use client code when the browser must hold changing state, respond to custom events, or call a browser-only API. Zero JavaScript is the starting point, not a contest: the goal is to make every shipped byte explain its purpose.

Add a UI framework integration

Install one supported framework only when a component benefits from its client-side state and component model.

Install an official integration when one part of the page genuinely benefits from a framework's state and component model. For React:

```
npx astro add react
```

The command installs the packages and updates `astro.config.mjs`. You can then import a React component from an Astro file:

```
---
import Counter from "../components/Counter.jsx"
---

<Counter />
```

This renders the component to HTML, but it does not make the counter interactive. Hydration is a separate decision expressed with a `client:*` directive.

The integration also does not convert the whole site to React. Astro pages, Markdown content, Vue islands, and React islands can coexist. That flexibility is useful for gradual migration, but mixing frameworks increases dependencies and the amount visitors may download.

Use one framework by default and add another only for a concrete constraint, such as reusing a valuable existing component. After installation, inspect `astro.config.mjs` and the browser network requests so you know what changed—and what did not.

Choose a client directive

Match hydration timing to the importance and visibility of the interaction.

A framework component is static HTML until you give it a client directive. The directive decides when its JavaScript downloads and hydrates:

```
<SearchBox client:load />
<NewsletterForm client:idle />
<Comments client:visible />
<MobileMenu client:media="(max-width: 48rem)" />
```

Choose by user impact:

- `client:load` is for interaction needed immediately
- `client:idle` waits until the initial page work is less urgent
- `client:visible` waits until an off-screen component approaches the viewport
- `client:media` waits until a media query matches
- `client:only="react"` skips server rendering and runs only in the browser

`client:only` removes the useful initial HTML, so reserve it for components that truly cannot render on the server because they depend on browser APIs. It may show nothing until JavaScript runs.

Hydration makes the existing HTML interactive; it is not the same as rendering the component for the first time. Start with no directive, confirm the static output is useful, then add the least urgent directive that still meets the interaction requirement.

Test the result with throttled JavaScript and with JavaScript disabled. A low-priority widget may appear late, but essential content and navigation should not disappear with it.

Render a framework component without hydration

Use React, Vue, Svelte, or another supported component as static HTML when it has no browser interaction.

A framework integration can render a component without hydrating it:

```
---
import ProductCard from "../components/ProductCard.jsx"
---

<ProductCard name="Notebook" price={12} />
```

Astro runs the framework renderer during the build or server request and sends the resulting HTML. It does not ship the framework runtime for this component.

This is useful when you are reusing an existing component that only formats props. But any event handler or state transition ends at the rendering boundary:

```
export function ProductCard({ name }) {
  return <button onClick={() => alert(name)}>{name}</button>
}
```

Without a `client:*` directive, the browser receives a button but React never attaches `onClick`. The element may look interactive while doing nothing, which is worse than clearly static output.

Use a normal link or form when native HTML covers the action. Add hydration only when local state, custom event handling, or a browser API is necessary. Inspect the Network panel afterward: the static version should not load a framework bundle for that component.

Keep islands independent

Use small hydration roots so one low-priority widget does not delay unrelated interaction.

Each client island is its own hydration root. A menu can use `client:load` while an off-screen chart uses `client:visible`; the chart does not delay the menu.

That independence is strongest when the components do not share live state. Pass each island the small, serializable initial data it needs:

```
<CartButton client:load initialCount={cart.count} />
<Reviews client:visible productId={product.id} />
```

Functions, database clients, and server secrets cannot cross this boundary. Props are serialized into data the browser can receive, so treat them as public.

When islands must coordinate, choose deliberately. A browser custom event works for occasional

messages. A shared external store fits state used by several islands in the same framework. If the interface changes as one tightly coupled unit, make it one larger island instead of building a fragile messaging system between tiny ones.

Use the Network and Performance panels to verify the architecture. The below-the-fold island should not load before its directive requires it, and interacting with one island should not wait for unrelated code.

Choose static or on-demand rendering

Start from static output and add a server adapter when request-time behavior is a real requirement.

Astro prerenders routes by default. Their frontmatter runs during the build and the result becomes files that a static host can serve directly. This is a strong default for content that changes only when you deploy.

Choose on-demand rendering when the response must depend on the request: a session cookie, authenticated user, frequently changing inventory, or server endpoint. Install the adapter for your deployment runtime, then opt out on that route:

```
---
export const prerender = false

const session = Astro.cookies.get("session")
---

<p>{session ? "Signed in" : "Guest"}</p>
```

The rest of a default static project can remain prerendered. If most routes are dynamic, `output: "server"` reverses the default; individual pages can still use `export const prerender = true`.

This choice changes the data lifetime and failure model. A static route can publish only data available during the build, but it needs no server execution per visit. An on-demand route can personalize each response, but it now depends on a compatible runtime, environment variables, latency, and uptime.

Choose per route, not per project slogan. Write down what request-specific input forces each dynamic route to exist.

Build and inspect the site

Treat the generated routes, assets, and browser requests as evidence of what will be deployed.

The production build is evidence, not a ceremony:

```
npm run build
npm run preview
```

Fix every build error before inspecting the result. Development mode can generate routes lazily, while the build must enumerate dynamic paths, validate content, resolve assets, and produce deployment output.

Check the built site from several angles:

- open every route type, including one dynamic page and an unknown URL
- follow internal links instead of testing only copied URLs

- verify titles, descriptions, heading order, and useful HTML with JavaScript disabled
- confirm image dimensions, alt text, and production asset URLs
- filter the Network panel by JavaScript and explain every bundle
- test endpoint status codes and response headers, not only response bodies

Look in `dist/` as well. Static routes should have generated output. A missing route often points to `getStaticPaths()`, draft filtering, or an incorrect filename. Unexpected scripts often point to an unnecessary client directive.

Preview is close to production output, but it does not reproduce every hosting behavior. Redirects, caching, environment variables, and platform functions still need verification after deployment.

Deploy a static Astro site

Upload the build output to a static host and verify the real production URL, redirects, and fallback behavior.

A static Astro deployment has two essential settings:

```
Build command: npm run build
Publish directory: dist
```

The host builds the project and serves the generated files. A purely static site does not need an Astro server adapter. Add an adapter only when you use on-demand routes or a platform feature that requires one.

Set the production origin when Astro must generate absolute URLs:

```
import { defineConfig } from "astro/config"

export default defineConfig({
  site: "https://example.com"
})
```

Build-time environment variables must exist in the host's build settings. Runtime secrets matter only when a deployed server function reads them. Confusing these lifetimes is a common reason a local build works while production fails.

After deployment, test the real domain. Check a nested route, an unknown route, redirects, canonical URLs, image assets, and endpoint status codes. Inspect response headers because caching and redirect behavior belong to the host as well as your code.

Finally, load one island with throttling and with JavaScript disabled. The production Network panel—not the source tree—shows what the visitor actually pays for.

Finish the Astro site

Deploy the content project with one deliberate island and record why every piece of browser JavaScript exists.

Finish the notes site with one deliberate island: a topic filter.

Render the complete published-note list as HTML first. The page must remain readable and every note link must work before JavaScript runs. Then add a small framework component that enhances the list by hiding notes outside the selected topic.

Choose its client directive from the experience you want. Use `client:load` only if filtering is part of the page's immediate purpose. If the filter sits below a useful introduction, `client:idle` may be enough. Write the reason next to your implementation notes; “this is what I always use” is not a reason.

Run the production build and verify:

1. the collection schema accepts every published entry
2. draft notes produce neither links nor detail pages
3. every dynamic route has a title, description, and one main heading
4. local images have generated dimensions and useful alt text
5. the page still exposes all content and links without JavaScript
6. only the filter's framework code loads in the browser

Preview the build, then deploy it. On the real domain, test a nested note URL, the custom 404, an optimized image, and the filter under network throttling. Check response status codes and browser requests rather than relying on appearances.

Finally, record three facts: how many routes were generated, which routes are static or on demand, and why each client-side script exists. If you cannot explain a script, remove its directive and test whether the site improves.

You now have the mental model for larger Astro projects: data is loaded and validated at a defined boundary, routes decide when HTML is produced, components compose that HTML, and islands add browser execution only where the user experience requires it.

Check your understanding: islands and deployment

Check zero-JavaScript output, hydration directives, framework components, island boundaries, adapters, builds, and deployment verification.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/astro/.