

Why Astro?

Astro is the web framework for building **content-driven websites** like blogs, marketing, and e-commerce. Astro is best-known for pioneering a new [frontend architecture](#) to reduce JavaScript overhead and complexity compared to other frameworks. If you need a website that loads fast and has great SEO, then Astro is for you.

Features

Astro is an all-in-one web framework. It includes everything you need to create a website, built-in. There are also hundreds of different [integrations](#) and [API hooks](#) available to customize a project to your exact use case and needs.

Some highlights include:

- [Islands](#): A component-based web architecture optimized for content-driven websites.
- [UI-agnostic](#): Supports React, Preact, Svelte, Vue, Solid, HTMX, web components, and more.
- [Server-first](#): Moves expensive rendering off of your visitors' devices.
- [Zero JS, by default](#): Less client-side JavaScript to slow your site down.
- [Content collections](#): Organize, validate, and provide TypeScript type-safety for your Markdown content.
- [Customizable](#): Partytown, MDX, and hundreds of integrations to choose from.

Design Principles

Here are five core design principles to help explain why we built Astro, the problems that it exists to solve, and why Astro may be the best choice for your project or team.

Astro is...

1. [Content-driven](#): Astro was designed to showcase your content.
2. [Server-first](#): Websites run faster when they render HTML on the server.
3. [Fast by default](#): It should be impossible to build a slow website in Astro.
4. [Easy to use](#): You don't need to be an expert to build something with Astro.
5. [Developer-focused](#): You should have the resources you need to be successful.

Content-driven

Astro was designed for building content-rich websites. This includes marketing sites, publishing sites, documentation sites, blogs, portfolios, landing pages, community sites, and e-commerce sites. If you have content to show, it needs to reach your reader quickly.

By contrast, most modern web frameworks were designed for building *web applications*. These frameworks excel at building more complex, application-like experiences in the browser: logged-in admin dashboards, inboxes, social networks, todo lists, and even native-like applications like [Figma](#) and [Ping](#). However with that complexity, they can struggle to provide great performance when delivering your content.

Astro's focus on content from its beginnings as a static site builder have allowed Astro to **sensibly scale up to performant, powerful, dynamic web applications** that still respect your content and your audience. Astro's unique focus on content lets Astro make tradeoffs and deliver unmatched performance features that wouldn't make sense for more application-focused web frameworks to implement.

Server-first

Astro leverages server rendering over client-side rendering in the browser as much as possible. This is the same approach that traditional server-side frameworks – PHP, WordPress, Laravel, Ruby on Rails, etc. – have been using for decades. But you don't need to learn a second server-side language to unlock it. With Astro, everything is still just HTML, CSS, and JavaScript (or TypeScript, if you prefer).

This approach stands in contrast to other modern JavaScript web frameworks like Next.js, SvelteKit, Nuxt, Remix, and others. These frameworks were built for client-side rendering of your entire website and include server-side rendering mainly to address performance concerns. This approach has been dubbed the **Single-Page App (SPA)**, in contrast with Astro's **Multi-Page App (MPA)** approach.

The SPA model has its benefits. However, these come at the expense of additional complexity and performance tradeoffs. These tradeoffs harm page performance – critical metrics like [Time to Interactive \(TTI\)](#) – which doesn't make much sense for content-focused websites where first-load performance is essential.

Astro's server-first approach allows you to opt in to client-side rendering only if, and exactly as, necessary. You can choose to add UI framework components that run on the client. You can take advantage of Astro's view transitions router for finer control over select page transitions and animations. Astro's server-first rendering, either pre-rendered or on-demand, provides performant defaults that you can enhance and extend.

Fast by default

Good performance is always important, but it is *especially* critical for websites whose success depends on displaying your content. It has been well-proven that poor performance loses you engagement, conversions, and money. For example:

- Every 100ms faster → 1% more conversions ([Mobify](#), earning +\$380,000/yr)
- 50% faster → 12% more sales ([AutoAnything](#))
- 20% faster → 10% more conversions ([Furniture Village](#))
- 40% faster → 15% more sign-ups ([Pinterest](#))
- 850ms faster → 7% more conversions ([COOK](#))
- Every 1 second slower → 10% fewer users ([BBC](#))

In many web frameworks, it is easy to build a website that looks great during development only to load painfully slow once deployed. JavaScript is often the culprit, since many phones and lower-powered devices rarely match the speed of a developer's laptop.

Astro's magic is in how it combines the two values explained above – a content focus with a server-first architecture – to make tradeoffs and deliver features that other frameworks cannot. The result is amazing web performance for every website, out of the box. Our goal: **It should be nearly impossible to build a slow website with Astro.**

An Astro website can [load 40% faster with 90% less JavaScript](#) than the same site built with the most popular React web framework. But don't take our word for it: watch Astro's performance leave Ryan Carniato (creator of Solid.js and Marko) [speechless](#).

Easy to use

Astro's goal is to be accessible to every web developer. Astro was designed to feel familiar and approachable regardless of skill level or past experience with web development.

The `.astro` UI language is a superset of HTML: any valid HTML is valid Astro templating syntax! So, if you can write HTML, you can write Astro components! But, it also combines some of our favorite features borrowed from other component languages like JSX expressions (React) and CSS scoping by default (Svelte and Vue). This closeness to HTML also makes it easier to use progressive enhancement and common accessibility patterns without any overhead.

We then made sure that you could also use your favorite UI component languages that you already know, and even reuse components you might already have. React, Preact, Svelte, Vue, Solid, and others, including web components, are all supported for authoring UI components in an Astro project.

Astro was designed to be less complex than other UI frameworks and languages. One big reason for this is that Astro was designed to render on the server, not in the browser. That means that you don't need to worry about hooks (React), stale closures (also React), refs (Vue), observables (Svelte), atoms, selectors, reactions, or derivations. There is no reactivity on the server, so all of that complexity melts away.

One of our favorite sayings is **opt in to complexity**. We designed Astro to remove as much "required complexity" as possible from the developer experience, especially as you onboard for the first time. You can build a "Hello World" example website in Astro with just HTML and CSS. Then, when you need to build something more powerful, you can incrementally reach for new features and APIs as you go.

Developer-focused

We strongly believe that Astro is only a successful project if people love using it. Astro has everything you need to support you as you build with Astro.

Astro invests in developer tools like a great CLI experience from the moment you open your terminal, an official VS Code extension for syntax highlighting, TypeScript and Intellisense, and documentation actively maintained by hundreds of community contributors and available in 14 languages.

Our welcoming, respectful, inclusive community on Discord is ready to provide support, motivation, and encouragement. Open a `#support` thread to get help with your project. Visit our dedicated `#showcase` channel for sharing your Astro sites, blog posts, videos, and even work-in-progress for safe feedback and constructive criticism. Participate in regular live events such as our weekly community call, "Talking and Doc'ing," and API/bug bashes.

As an open-source project, we welcome contributions of all types and sizes from community members of all experience levels. You are invited to join in roadmap discussions to shape the future of Astro, and we hope you'll contribute fixes and features to the core codebase, compiler, docs, language tools, websites, and other projects.

Islands architecture

Astro helped pioneer and popularize a new frontend architecture pattern called **Islands Architecture**. Islands architecture works by rendering the majority of your page to fast, static HTML with smaller "islands" of JavaScript added when interactivity or personalization is needed on the page (an image carousel, for example). This avoids the monolithic JavaScript payloads that slow down the responsiveness of many other, modern

JavaScript web frameworks.

A brief history

The term “component island” was first coined by Etsy’s frontend architect [Katie Saylor-Miller](#) in 2019. This idea was then expanded on and documented in [this post](#) by Preact creator Jason Miller on August 11, 2020.

The general idea of an “Islands” architecture is deceptively simple: render HTML pages on the server, and inject placeholders or slots around highly dynamic regions [...] that can then be “hydrated” on the client into small self-contained widgets, reusing their server-rendered initial HTML.

— Jason Miller, Creator of Preact

The technique that this architectural pattern builds on is also known as **partial** or **selective hydration**.

In contrast, most JavaScript-based web frameworks hydrate & render an entire website as one large JavaScript application (also known as a single-page application, or SPA). SPAs provide simplicity and power but suffer from page-load performance problems due to heavy client-side JavaScript usage.

SPAs have their place, even [embedded inside an Astro page](#). But, SPAs lack the native ability to selectively and strategically hydrate, making them a heavy-handed choice for most projects on the web today.

Astro became popular as the first mainstream JavaScript web framework with selective hydration built-in, using that same component islands pattern first coined by Saylor-Miller. We’ve since expanded and evolved on Saylor-Miller’s original work, which helped to inspire a similar component island approach to dynamically server-rendered content.

What is an island?

In Astro, an island is an enhanced UI component on an otherwise static page of HTML.

A [client island](#) is an interactive JavaScript UI component that is hydrated separately from the rest of the page, while a [server island](#) is a UI component that server-renders its dynamic content separately from the rest of the page.

Both islands run expensive or slower processes independently, on a per-component basis, for optimized page loads.

Island components

Astro components are the building blocks of your page template. They render to static HTML with no client-side runtime.

Think of a client island as an interactive widget floating in a sea of otherwise static, lightweight, server-rendered HTML. Server islands can be added for personalized or dynamic server-rendered elements, such as a logged in visitor’s profile picture.

Header (interactive island)

Sidebar (static HTML)

Static content like text, images, etc.

Image carousel (interactive island)

Footer (static HTML)

Source: [Islands Architecture: Jason Miller](#)

An island always runs in isolation from other islands on the page, and multiple islands can exist on a page. Client islands can still share state and communicate with each other, even though they run in different component contexts.

This flexibility allows Astro to support multiple UI frameworks like [React](#), [Preact](#), [Svelte](#), [Vue](#), and [SolidJS](#). Because they are independent, you can even mix several frameworks on each page.

Tip

Although most developers will stick to just one UI framework, Astro supports multiple frameworks in the same project. This allows you to:

- Choose the framework that is best for each component.
- Learn a new framework without needing to start a new project.
- Collaborate with others even when working in different frameworks.
- Incrementally convert an existing site to another framework with no downtime.

Client Islands

By default, Astro will automatically render every UI component to just HTML & CSS, **stripping out all client-side JavaScript automatically.**

src/pages/index.astro

```
1 <MyReactComponent />
```

This may sound strict, but this behavior is what keeps Astro websites fast by default and protects developers from accidentally sending unnecessary or unwanted JavaScript that might slow down their website.

Turning any static UI component into an interactive island requires only a `client:*` directive. Astro then automatically builds and bundles your client-side JavaScript for optimized performance.

src/pages/index.astro

```
1 <!-- This component is now interactive on the page!
2
3     The rest of your website remains static. -->
4
5 <MyReactComponent client:load />
```

With islands, client-side JavaScript is only loaded for the explicit interactive components that you mark using `client:*` directives.

And because interaction is configured at the component-level, you can handle different loading priorities for each component based on its usage. For example, `client:idle` tells a component to load when the browser becomes idle, and `client:visible` tells a component to load only once it enters the viewport.

Benefits of client islands

The most obvious benefit of building with Astro Islands is performance: the majority of your website is converted to fast, static HTML and JavaScript is only loaded for the individual components that need it. JavaScript is one of the slowest assets that you can load per-byte, so every byte counts.

Another benefit is parallel loading. In the example illustration above, the low-priority “image carousel” island doesn’t need to block the high-priority “header” island. The two load in parallel and hydrate in isolation, meaning that the header becomes interactive immediately without having to wait for the heavier carousel lower down the page.

Even better, you can tell Astro exactly how and when to render each component. If that image carousel is really expensive to load, you can attach a special [client directive](#) that tells Astro to only load the carousel when it becomes visible on the page. If the user never sees it, it never loads.

In Astro, it’s up to you as the developer to explicitly tell Astro which components on the page need to also run in the browser. Astro will only hydrate exactly what’s needed on the page and leave the rest of your site as static HTML.

Client islands are the secret to Astro’s fast-by-default performance story!

Read more about [using JavaScript framework components](#) in your project.

Server islands

Server islands are a way to move expensive or slow server-side code out of the way of the main rendering process, making it easy to combine high-performance static HTML and dynamic server-generated components.

Add the [server:defer directive](#) to any Astro component on your page to turn it into its own server island:

src/pages/index.astro

```
1  ---
2
3  import Avatar from "../components/Avatar.astro";
4
5  ---
6
7  <Avatar server:defer />
```

This breaks up your page with smaller areas of server-rendered content that each load in parallel.

Your page’s main content can be rendered immediately with placeholder content, such as a generic avatar, until your island’s own content is available. With server islands, having small components of personalized content does not delay the rendering of an otherwise static page.

This rendering pattern was built to be portable. It does not depend on any server infrastructure so it will work with any host, from a Node.js server in a Docker container to the serverless provider of your choice.

Benefits of server islands

One benefit of server islands is the ability to render the more highly dynamic parts of your page on the fly. This allows the outer shell and main content to be more aggressively cached, providing faster performance.

Another benefit is providing a great visitor experience. Server islands are optimized and load quickly, often even before the browser has even painted the page. But in the short time it takes for your islands to render, you can display custom fallback content and prevent any layout shift.

An example of a site that benefits from Astro's server islands is an e-commerce storefront. Although the main content of product pages change infrequently, these pages typically have some dynamic pieces:

- The user's avatar in the header.
- Special deals and sales for the product.
- User reviews.

Using server islands for these elements, your visitor will see the most important part of the page, your product, immediately. Generic avatars, loading spinners, and store announcements can be displayed as fallback content until the personalized parts are available.

Read more about [using server islands](#) in your project.

Install Astro

The [create astro CLI command](#) is the fastest way to start a new Astro project from scratch. It will walk you through every step of setting up your new Astro project and allow you to choose from a few different official starter templates.

You can also run the CLI command with the `template` flag to begin your project using any existing theme or starter template. Explore our [themes and starters showcase](#) where you can browse themes for blogs, portfolios, documentation sites, landing pages, and more!

To install Astro manually instead, see our [step-by-step manual installation guide](#).

Online previews

Prefer to try Astro in your browser? Visit [astro.new](#) to browse our starter templates and spin up a new Astro project without ever leaving your browser.

Prerequisites

- **Node.js** - `v22.12.0` or higher. Odd-numbered versions like `v23` are not supported.
- **Text editor** - We recommend [VS Code](#) with our [Official Astro extension](#).
- **Terminal** - Astro is accessed through its command-line interface (CLI).

Browser compatibility

Astro is built with Vite which targets browsers with modern JavaScript support by default. For a complete reference, you can see the [list of currently supported browser versions in Vite](#).

Install from the CLI wizard

You can run `create astro` anywhere on your machine, so there's no need to create a new empty directory for your project before you begin. If you don't have an empty directory yet for your new project, the wizard will help create one for you automatically.

1. Run the following command in your terminal to start the install wizard:

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | # create a new project with npm
2 |
3 | npm create astro@latest
```

If all goes well, you will see a success message followed by some recommended next steps.

2. Now that your project has been created, you can `cd` into your new project directory to begin using Astro.

3. If you skipped the “Install dependencies?” step during the CLI wizard, then be sure to install your dependencies before continuing.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npm install
```

4. You can now [start the Astro dev server](#) and see a live preview of your project while you build!

CLI installation flags

You can run the `create astro` command with additional flags to customize the setup process (e.g. answering “yes” to all questions, skipping the Houston animation) or your new project (e.g. install git or not, add integrations).

See [all the available `create astro` command flags](#).

Add integrations

You can start a new Astro project and install any [official integrations](#) or community integrations that support the `astro add` command at the same time by passing the `--add` argument to the `create astro` command.

Run the following command in your terminal, substituting any integration that supports the `astro add` command:

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 # create a new project with React and Partytown
2
3 npm create astro@latest -- --add react --add partytown
```

Use a theme or starter template

You can start a new Astro project based on an [official example](#) or the `main` branch of any GitHub repository by passing a `--template` argument to the `create astro` command.

Run the following command in your terminal, substituting the official Astro starter template name, or the GitHub username and repository of the theme you want to use:

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 # create a new project with an official examplenpm create astro@latest -- --template
  <example-name>
2 # create a new project based on a GitHub repository's main branchnpm create astro@latest
  -- --template <github-username>/<github-repo>
```

By default, this command will use the template repository's `main` branch. To use a different branch name, pass it as part of the `--template` argument: `<github-username>/<github-repo>#<branch>`.

Manual Setup

This guide will walk you through the steps to manually install and configure a new Astro project.

If you prefer not to use our automatic `create astro` CLI tool, you can set up your project yourself by following the guide below.

1. Create your directory

Create an empty directory with the name of your project, and then navigate into it.

Terminal window

```
1 mkdir my-astro-project
2
3 cd my-astro-project
```

Once you are in your new directory, create your project `package.json` file. This is how you will manage your project dependencies, including Astro. If you aren't familiar with this file format, run the following command to create one.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 npm init --yes
```

2. Install Astro

First, install the Astro project dependencies inside your project.

Important

Astro must be installed locally, not globally. Make sure you are *not* running `npm install -g astro` or `pnpm add -g astro` or `yarn add global astro`.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 npm install astro
```

Then, replace any placeholder “scripts” section of your `package.json` with the following:

package.json

```
1 {
2
3   "scripts": {
4
5     "test": "echo \"Error: no test specified\" && exit 1",
6
7     "dev": "astro dev",
8
9     "build": "astro build",
10  }
```

```
11     "preview": "astro preview"
12
13   },
14
15 }
```

You'll use these scripts later in the guide to start Astro and run its different commands.

3. Create your first page

In your text editor, create a new file in your directory at `src/pages/index.astro`. This will be your first Astro page in the project.

For this guide, copy and paste the following code snippet (including `---` dashes) into your new file:

`src/pages/index.astro`

```
1  ---
2
3  // Welcome to Astro! Everything between these triple-dash code fences
4
5  // is your "component frontmatter". It never runs in the browser.
6
7  console.log('This runs in your terminal, not the browser!');
8
9  ---
10
11  <!-- Below is your "component template." It's just HTML, but with
12
13       some magic sprinkled in to help you build great templates. -->
14
15  <html>
16
17    <body>
18
19      <h1>Hello, world!</h1>
20
21    </body>
22
23  </html>
24
25  <style>
26
27    h1 {
28
29      color: orange;
30
31    }
32
33  </style>
```

4. Create your first static asset

You will also want to create a `public/` directory to store your static assets. Astro will always include

these assets in your final build, so you can safely reference them from inside your component templates.

In your text editor, create a new file in your directory at `public/robots.txt`. `robots.txt` is a simple file that most sites will include to tell search bots like Google how to treat your site.

For this guide, copy and paste the following code snippet into your new file:

`public/robots.txt`

```
1 # Example: Allow all bots to scan and index your site.
2
3 # Full syntax: https://developers.google.com/search/docs/advanced/robots/create-robots-txt
4
5 User-agent: *
6
7 Allow: /
```

5. Create `astro.config.mjs`

Astro is configured using `astro.config.mjs`. This file is optional if you do not need to configure Astro, but you may wish to create it now.

Create `astro.config.mjs` at the root of your project, and copy the code below into it:

`astro.config.mjs`

```
1 import { defineConfig } from "astro/config";
2 // https://astro.build/configexport default defineConfig({});
```

If you want to include [UI framework components](#) such as React, Svelte, etc. or use other tools such as MDX or Partytown in your project, here is where you will [manually import and configure integrations](#).

Read Astro's [API configuration reference](#) for more information.

6. Add TypeScript support

TypeScript is configured using `tsconfig.json`. Even if you don't write TypeScript code, this file is important so that tools like Astro and VS Code know how to understand your project. Some features (like npm package imports) aren't fully supported in the editor without a `tsconfig.json` file.

If you do intend to write TypeScript code, using Astro's `strict` or `strictest` template is recommended. You can view and compare the three template configurations at [astro/tsconfigs/](#).

Create `tsconfig.json` at the root of your project, and copy the code below into it. (You can use `base`, `strict`, or `strictest` for your TypeScript template):

`tsconfig.json`

```
1 {
2
3   "extends": "astro/tsconfigs/base"
4
5 }
```

Read Astro's [TypeScript setup guide](#) for more information.

7. Next Steps

If you have followed the steps above, your project directory should now look like this:

Directorynode_modules/Directorypublic/robots.txtDirectorysrc/Directorypages/index.astroastro.config.mjspackage-lock.jsonor yarn.lock, pnpm-lock.yaml, etc.package.jsonsonconfig.json

8. You can now [start the Astro dev server](#) and see a live preview of your project while you build!

Project structure

Your new Astro project generated from the `create astro` CLI wizard already includes some files and folders. Others, you will create yourself and add to Astro's existing file structure.

Here's how an Astro project is organized, and some files you will find in your new project.

Directories and Files

Astro leverages an opinionated folder layout for your project. Every Astro project root should include the following directories and files:

- `src/*` - Your project source code (components, pages, styles, images, etc.)
- `public/*` - Your non-code, unprocessed assets (fonts, icons, etc.)
- `package.json` - A project manifest.
- `astro.config.mjs` - An Astro configuration file. (recommended)
- `tsconfig.json` - A TypeScript configuration file. (recommended)

Example Project Tree

A common Astro project directory might look like this:

Directorypublic/robots.txtfavicon.svgmy-cv.pdfDirectorysrc/Directoryblog/post1.mdpost2.mdpost3.mdDirectorycomponents/Header.astroButton.jsxDirectoryimages/image1.jpgimage2.jpgimage3.jpgDirectorylayouts/PostLayout.astroDirectorypages/Directoryposts/[post].astroabout.astroindex.astroindex.rss.xml.jsDirectorystyles/global.csscontent.config.tsastro.config.mjspackage.jsonsonconfig.json

`src/`

The `src/` folder is where most of your project source code lives. This includes:

- [Pages](#)
- [Layouts](#)
- [Astro components](#)
- [UI framework components \(React, etc.\)](#)
- [Styles \(CSS, Sass\)](#)
- [Markdown](#)
- [Images to be optimized and processed by Astro](#)

Astro processes, optimizes, and bundles your `src/` files to create the final website that is shipped to the browser. Unlike the static `public/` directory, your `src/` files are built and handled for you by Astro.

Some files (like Astro components) are not even sent to the browser as written but are instead rendered to static HTML. Other files (like CSS) are sent to the browser but may be optimized or bundled with other CSS files for performance.

Tip

While this guide describes some popular conventions used in the Astro community, the only directory reserved by Astro is `src/pages/`. You are free to rename and reorganize any other directories in a way that works best for you.

`src/pages`

Pages routes are created for your site by adding [supported file types](#) to this directory.

Caution

`src/pages` is a **required** sub-directory in your Astro project. Without it, your site will have no pages or routes!

`src/components`

Components are reusable units of code for your HTML pages. These could be [Astro components](#), or [UI framework components](#) like React or Vue. It is common to group and organize all of your project components together in this folder.

This is a common convention in Astro projects, but it is not required. Feel free to organize your components however you like!

`src/layouts`

[Layouts](#) are Astro components that define the UI structure shared by one or more [pages](#).

Just like `src/components`, this directory is a common convention but not required.

`src/styles`

It is a common convention to store your CSS or Sass files in a `src/styles` directory, but this is not required. As long as your styles live somewhere in the `src/` directory and are imported correctly, Astro will handle and optimize them.

`public/`

The `public/` directory is for files and assets in your project that do not need to be processed during Astro's build process. The files in this folder will be copied into the build folder untouched, and then your site will be built.

This behavior makes `public/` ideal for common assets that do not require any processing, like some images and fonts, or special files such as `robots.txt` and `manifest.webmanifest`.

You can place CSS and JavaScript in your `public/` directory, but be aware that those files will not be bundled or optimized in your final build.

Tip

As a general rule, any CSS or JavaScript that you write yourself should live in your `src/` directory.

`package.json`

This is a file used by JavaScript package managers to manage your dependencies. It also defines the scripts that are commonly used to run Astro (ex: `npm run dev`, `npm run build`).

There are [two kinds of dependencies](#) you can specify in a `package.json`: `dependencies` and `devDependencies`. In most cases, these work the same: Astro needs all dependencies at build time, and your package manager will install both. We recommend putting all of your dependencies in `dependencies` to start, and only use `devDependencies` if you find a specific need to do so.

For help creating a new `package.json` file for your project, check out the [manual setup](#) instructions.

`astro.config.mjs`

This file is generated in every starter template and includes configuration options for your Astro project. Here you can specify integrations to use, build options, server options, and more.

Astro supports several file formats for its JavaScript configuration file: `astro.config.js`, `astro.config.mjs` and `astro.config.ts`. We recommend using `.mjs` in most cases or `.ts` if you want to write TypeScript in your config file.

TypeScript config file loading is handled using `tsm` and will respect your project's `tsconfig` options.

See the [configuration reference](#) for complete details.

`tsconfig.json`

This file is generated in every starter template and includes TypeScript configuration options for your Astro project. Some features (like npm package imports) aren't fully supported in the editor without a `tsconfig.json` file.

See the [TypeScript Guide](#) for details on setting configurations.

Develop and build

Once you have an Astro project, now you're ready to build with Astro! 🚀

Edit your project

To make changes to your project, open your project folder in your code editor. Working in development mode with the dev server running allows you to see updates to your site as you edit the code.

You can also [customize aspects of your development environment](#) such as configuring TypeScript or installing the official Astro editor extensions.

Start the Astro dev server

Astro comes with a built-in development server that has everything you need for project development. The `astro dev` CLI command will start the local development server so that you can see your new website in action for the very first time.

Every starter template comes with a pre-configured script that will run `astro dev` for you. After navigating into your project directory, use your favorite package manager to run this command and start the Astro development server.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npm run dev
```

If all goes well, Astro will now be serving your project on <http://localhost:4321/>. Visit that link in your browser and see your new site!

Work in development mode

Astro will listen for live file changes in your `src/` directory and update your site preview as you build, so you will not need to restart the server as you make changes during development. You will always be able to see an up-to-date version of your site in your browser when the dev server is running.

When viewing your site in the browser, you'll have access to the [Astro dev toolbar](#). As you build, it will help you inspect your [islands](#), spot accessibility issues, and more.

If you aren't able to open your project in the browser after starting the dev server, go back to the terminal where you ran the `dev` command and check the message displayed. It should tell you if an error occurred, or if your project is being served at a different URL than <http://localhost:4321/>.

Build and preview your site

To check the version of your site that will be created at build time, quit the dev server (Ctrl + C) and run the appropriate build command for your package manager in your terminal:

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npm run build
```

Astro will build a deploy-ready version of your site in a separate folder (`dist/` by default) and you can watch its progress in the terminal. This will alert you to any build errors in your project before you deploy to production. If TypeScript is configured to `strict` or `strictest`, the `build` script will also check your project for type errors.

When the build is finished, run the appropriate `preview` command (e.g. `npm run preview`) in your terminal and you can view the built version of your site locally in the same browser preview window.

Note that this previews your code as it existed when the build command was last run. This is meant to give you a preview of how your site will look when it is deployed to the web. Any later changes you make to your code after building will **not** be reflected while you preview your site until you run the build command again.

Use (Ctrl + C) to quit the preview and run another terminal command, such as restarting the dev server to go back to [working in development mode](#) which does update as you edit to show a live preview of your code changes.

Read more about [the Astro CLI](#) and the terminal commands you will use as you build with Astro.

Tip

You may wish to [deploy your new site right away](#), before you begin to add or change too much code. This is helpful to get a minimal, working version of your site published and can save you extra time and effort troubleshooting your deployment later.

Next Steps

Success! You are now ready to start building with Astro! 🎉

Here are a few things that we recommend exploring next. You can read them in any order. You can even leave our documentation for a bit and go play in your new Astro project codebase, coming back here whenever you run into trouble or have a question.

Configure your dev environment

Explore the guides below to customize your development experience.

[Editor Setup](#)Customize your code editor to improve the Astro developer experience and unlock new features.

[Dev Toolbar](#)Explore the helpful features of the dev toolbar.

[TypeScript Configuration](#)Configure options for type-checking, IntelliSense, and more.

Explore Astro's Features

[Understand your codebase](#)Learn about Astro's file structure in our Project Structure guide.

[Create content collections](#)Add content to your new site with frontmatter validation and automatic type-safety.

[Add view transitions](#)Create seamless page transitions and animations.

[Learn about Islands](#)Read about Astro's islands architecture.

Take the introductory tutorial

Build a fully functional Astro blog starting from a single blank page in our [introductory tutorial](#).

This is a great way to see how Astro works and walks you through the basics of pages, layouts, components, routing, islands, and more. It also includes an optional, beginner-friendly unit for those newer to web development concepts in general, which will guide you through installing the necessary applications on your computer, creating a GitHub account, and deploying your site.

Configuration overview

Astro is a flexible, unopinionated framework that allows you to configure your project in many different ways. This means that getting started with a new project might feel overwhelming: there is no “one best way” to set up your Astro project!

The guides in this “Configuration” section will help you familiarize yourself with the various files that allow you to configure and customize aspects of your project and development environment.

If this is your first Astro project, or if it’s been a while since you’ve set up a new project, use the following guides and reference in the documentation for assistance.

The Astro config File

The [Astro config file](#) is a JavaScript file included at the root of every starter project:

astro.config.mjs

```
1 import { defineConfig } from "astro/config";
2 export default defineConfig({ // your configuration options here...});
```

It is only required if you have something to configure, but most projects will use this file. The `defineConfig()` helper provides automatic IntelliSense in your IDE and is where you will add all your configuration options to tell Astro how to build and render your project to HTML.

We recommend using the default file format `.mjs` in most cases, or `.ts` if you want to write TypeScript in your config file. However, `astro.config.js` is also supported.

Read Astro’s [configuration reference](#) for a full overview of all supported configuration options.

The TypeScript config File

Every Astro starter project includes a `tsconfig.json` file in your project. Astro’s [component script](#) is TypeScript, which provides Astro’s editor tooling and allows you to optionally add syntax to your JavaScript for type checking of your own project code.

Use the `tsconfig.json` file to configure the TypeScript template that will perform type checks on your code, configure TypeScript plugins, set import aliases, and more.

Read Astro’s [TypeScript guide](#) for a full overview of TypeScript options and Astro’s built-in utility types.

Development Experience

While you work in development mode, you can take advantage of your code editor and other tools to improve the Astro developer experience.

Astro provides its own official VS Code extension and is compatible with several other popular editor tools. Astro also provides a customizable toolbar that displays in your browser preview while the dev server is running. You can install and even build your own toolbar apps for additional functionality.

Read Astro's guides to [editor setup options](#) and [using the dev toolbar](#) to learn how to customize your development experience.

Common new project tasks

Here are some first steps you might choose to take with a new Astro project.

Add your deployment domain

For generating your sitemap and creating canonical URLs, configure your deployment URL in the [site](#) option. If you are deploying to a path (e.g. `www.example.com/docs`), you can also configure a [base](#) for the root of your project.

Additionally, different deployment hosts may have different behavior regarding trailing slashes at the end of your URLs. (e.g. `example.com/about` vs `example.com/about/`). Once your site is deployed, you may need to configure your [trailingSlash](#) preference.

astro.config.mjs

```
1 import { defineConfig } from "astro/config";
2 export default defineConfig({ site: "https://www.example.com", base: "/docs",
  trailingSlash: "always", });
```

Add site metadata

Astro does not use its configuration file for common SEO or meta data, only for information required to build your project code and render it to HTML.

Instead, this information is added to your page `<head>` using standard HTML `<link>` and `<meta>` tags, just as if you were writing plain HTML pages.

One common pattern for Astro sites is to create a `<Head />` [.astro component](#) that can be added to a common [layout component](#) so it can apply to all your pages.

src/components/MainLayout.astro

```
1 ---import Head from "../Head.astro";
2 const { ...props } = Astro.props;---<html> <head> <meta charset="utf-8"> <Head />
  <!-- Additional head elements --> </head> <body> <!-- Page content goes here -->
</body></html>
```

Because `Head.astro` is just a regular Astro component, you can import files and receive props passed from other components, such as a specific page title.

src/components/Head.astro

```
1  ---import Favicon from "../assets/Favicon.astro";import SomeOtherTags from
   "./SomeOtherTags.astro";
2  const { title = "My Astro website", ...props } = Astro.props;---<link rel="sitemap"
   href="/sitemap-index.xml"><title>{title}</title><meta name="description"
   content="welcome to my new Astro site!">
3  <!-- web analytics --><script data-goatcounter="https://my-
   account.goatcounter.com/count" async src="//gc.zgo.at/count.js"></script>
4  <!-- Open Graph tags --><meta property="og:title" content="My New Astro Website" /><meta
   property="og:type" content="website" /><meta property="og:url"
   content="http://www.example.com/" /><meta property="og:description" content="welcome to
   my new Astro site!" /><meta property="og:image"
   content="https://www.example.com/_astro/seo-banner.BZD7kegZ.webp"><meta
   property="og:image:alt" content="">
5  <SomeOtherTags />
6  <Favicon />
```

Editor setup

Customize your code editor to improve the Astro developer experience and unlock new features.

VS Code

[VS Code](#) is a popular code editor for web developers, built by Microsoft. The VS Code engine also powers popular in-browser code editors like [GitHub Codespaces](#).

Astro works with any code editor. However, VS Code is our recommended editor for Astro projects. We maintain an official [Astro VS Code Extension](#) that unlocks several key features and developer experience improvements for Astro projects.

- Syntax highlighting for `.astro` files.
- TypeScript type information for `.astro` files.
- [VS Code Intellisense](#) for code completion, hints and more.

To get started, install the [Astro VS Code Extension](#) today.

See how to [set up TypeScript](#) in your Astro project.

Zed

[Zed](#) is a high-performance, multiplayer code editor that is optimized for speed and large projects. Their [Astro extension](#) includes features like syntax highlighting for `.astro` files, code completion, formatting, diagnostics, and go-to-definition.

JetBrains IDEs

[Webstorm](#) is a JavaScript and TypeScript IDE that added support for the Astro Language Server in version 2024.2. This update brings features like syntax highlighting, code completion, and formatting.

Install the official plugin through [JetBrains Marketplace](#) or by searching for “Astro” in the IDE’s Plugins tab. You can toggle the language server in `Settings | Languages & Frameworks | TypeScript | Astro`.

For more information on Astro support in Webstorm, check out [the official Webstorm Astro Documentation](#).

Other Code Editors

Our amazing community maintains several extensions for other popular editors, including:

- [VS Code Extension on Open VSX](#) Official - The official Astro VS Code Extension, available on the Open VSX registry for editors like [Cursor](#) or [VSCodium](#).
- [Vim Plugin](#) Community - Provides syntax highlighting, indentation, and code folding support for Astro inside of Vim or Neovim
- Neovim [LSP](#) and [TreeSitter](#) Plugins Community - Provides syntax highlighting, treesitter parsing, and code completion for Astro inside of Neovim
- Emacs - See instructions for [Configuring Emacs and Eglot](#) Community to work with Astro
- [Astro syntax highlighting for Sublime Text](#) Community - The Astro package for Sublime Text, available on the Sublime Text package manager.
- [Nova Extension](#) Community - Provides syntax highlighting and code completion for Astro inside of Nova

In-Browser Editors

In addition to local editors, Astro also runs well on in-browser hosted editors, including:

- [StackBlitz](#) and [CodeSandbox](#) - online editors that run in your browser, with built-in syntax highlighting support for `.astro` files. No installation or configuration required!
- [GitHub.dev](#) - allows you to install the Astro VS Code extension as a [web extension](#), which gives you access to only some of the full extension features. Currently, only syntax highlighting is supported.

Other tools

ESLint

[ESLint](#) is a popular linter for JavaScript and JSX. For Astro support, [a community maintained plugin](#) can be installed.

See [the project’s User Guide](#) for more information on how to install and set up ESLint for your project.

Stylelint

[Stylelint](#) is a popular linter for CSS. [A community maintained Stylelint configuration](#) provides Astro support.

Installation instructions, editor integration, and additional information can be found in the project’s README.

Biome

[Biome](#) is an all-in-one linter and formatter for the web. [Biome currently has experimental support for `.astro` files](#), and can be used to lint and format the frontmatter in `.astro` files.

Prettier

[Prettier](#) is a popular formatter for JavaScript, HTML, CSS, and more. If you're using the [Astro VS Code Extension](#), code formatting with Prettier is included.

To add support for formatting `.astro` files outside of the editor (e.g. CLI) or inside editors that don't support our editor tooling, install [the official Astro Prettier plugin](#).

1. Install `prettier` and `prettier-plugin-astro`.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npm install --save-dev --save-exact prettier prettier-plugin-astro
```

2. Create a `.prettierrc` configuration file (or `.prettierrc.json`, `.prettierrc.mjs`, or [other supported formats](#)) in the root of your project and add `prettier-plugin-astro` to it.

In this file, also manually specify the parser for Astro files.

`.prettierrc`

```
1 | {
2 |
3 |   "plugins": ["prettier-plugin-astro"],
4 |
5 |   "overrides": [
6 |
7 |     {
8 |
9 |       "files": "*.astro",
10 |
11 |       "options": {
12 |
13 |         "parser": "astro"
14 |
15 |       }
16 |
17 |     }
18 |
19 |   ]
20 |
21 | }
```

3. Optionally, install other Prettier plugins for your project, and add them to the configuration file. These additional plugins may need to be listed in a specific order. For example, if you use Tailwind, `prettier-plugin-tailwindcss` must be [the last Prettier plugin in the plugins array](#).

`.prettierrc`

```
1  {
2
3    "plugins": [
4
5      "prettier-plugin-astro",
6
7      "prettier-plugin-tailwindcss" // needs to be last
8
9    ],
10
11    "overrides": [
12
13      {
14
15        "files": "*.astro",
16
17        "options": {
18
19          "parser": "astro"
20
21        }
22
23      }
24
25    ]
26
27  }
```

4. Run the following command in your terminal to format your files.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npx prettier . --write
```

See the [Prettier plugin's README](#) for more information about its supported options, how to set up Prettier inside VS Code, and more.

dprint

[dprint](#) is a highly-configurable code formatter that supports many languages, including JavaScript, TypeScript, CSS, and more. Support for `.astro` files can be added using the [markup_fmt plugin](#).

TypeScript

Astro ships with built-in support for [TypeScript](#). You can import `.ts` and `.tsx` files in your Astro project, write TypeScript code directly inside your [Astro component](#), and even use an `astro.config.ts` file for your Astro configuration if you like.

Using TypeScript, you can prevent errors at runtime by defining the shapes of objects and components in your code. For example, if you use TypeScript to [type your component's props](#), you'll get an error in your editor if you set a prop that your component doesn't accept.

You don't need to write TypeScript code in your Astro projects to benefit from it. Astro always treats your component code as TypeScript, and the [Astro VS Code Extension](#) will infer as much as it can to provide autocompletion, hints, and errors in your editor.

The Astro dev server won't perform any type checking, but you can use a [separate script](#) to check for type errors from the command line.

Setup

Astro starter projects include a `tsconfig.json` file in your project. Even if you don't write TypeScript code, this file is important so that tools like Astro and VS Code know how to understand your project. Some features (like npm package imports) aren't fully supported in the editor without a `tsconfig.json` file. If you install Astro manually, be sure to create this file yourself.

TSConfig templates

Three extensible `tsconfig.json` templates are included in Astro: `base`, `strict`, and `strictest`. The `base` template enables support for modern JavaScript features and is also used as a basis for the other templates. We recommend using `strict` or `strictest` if you plan to write TypeScript in your project. You can view and compare the three template configurations at [astro/tsconfigs/](#).

To inherit from one of the templates, use [the `extends` setting](#):

`tsconfig.json`

```
1 {  
2  
3   "extends": "astro/tsconfigs/base"  
4  
5 }
```

Additionally, we recommend setting `include` and `exclude` as follows to benefit from Astro types and avoid checking built files:

`tsconfig.json`

```
1 {
2
3   "extends": "astro/tsconfigs/base",
4
5   "include": [".astro/types.d.ts", "**/*"],
6
7   "exclude": ["dist"]
8
9 }
```

TypeScript editor plugin

The [Astro TypeScript plugin](#) can be installed separately when you are not using the [official Astro VS Code extension](#). This plugin is automatically installed and configured by the VS Code extension, and you do not need to install both.

This plugin runs only in the editor. When running `tsc` in the terminal, `.astro` files are ignored entirely. Instead, you can use [the astro check CLI command](#) to check both `.astro` and `.ts` files.

This plugin also supports importing `.astro` files from `.ts` files (which can be useful for re-exporting).

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 npm install @astrojs/ts-plugin
```

Then, add the following to your `tsconfig.json`:

`tsconfig.json`

```
1 {
2
3   "compilerOptions": {
4
5     "plugins": [
6
7       {
8
9         "name": "@astrojs/ts-plugin"
10
11       },
12
13     ],
14
15   }
16
17 }
```

To check that the plugin is working, create a `.ts` file and import an Astro component into it. You should have no warning messages from your editor.

UI Frameworks

If your project uses a [UI framework](#), additional settings depending on the framework might be needed. Please see your framework's TypeScript documentation for more information. ([Vue](#), [React](#), [Preact](#), [Solid](#), [Svelte](#))

Type Imports

Use explicit type imports and exports whenever possible.

```
1 import { SomeType } from "./script";
2
3 import type { SomeType } from "./script";
```

This way, you avoid edge cases where Astro's bundler may try to incorrectly bundle your imported types as if they were JavaScript.

You can configure TypeScript to enforce type imports in your `tsconfig.json` file. Set [verbatimModuleSyntax](#) to `true`. TypeScript will check your imports and tell you when `import type` should be used. This setting is enabled by default in all our presets.

`tsconfig.json`

```
1 {
2
3   "compilerOptions": {
4
5     "verbatimModuleSyntax": true
6
7   }
8
9 }
```

Import Aliases

Astro supports import aliases that you define in your `tsconfig.json` `paths` configuration. [Read our imports guide](#) to learn more.

`src/pages/about/nate.astro`

```
1 ---
2
3 import HelloWorld from "@components/HelloWorld.astro";
4
5 import Layout from "@layouts/Layout.astro";
6
7 ---
```

`tsconfig.json`

```

1  {
2
3    "compilerOptions": {
4
5      "paths": {
6
7        "@components/*": ["/src/components/*"],
8
9        "@layouts/*": ["/src/layouts/*"]
10
11      }
12
13    }
14
15  }

```

Extending global types

You can create `src/env.d.ts` as a convention for adding custom types declarations, or to benefit from Astro types if you don't have a `tsconfig.json`:

`src/env.d.ts`

```

1  // Custom types declarations
2  declare var myString: string;
3  // Astro types, not necessary if you already have a `tsconfig.json` <reference
4  path="../../astro/types.d.ts" />

```

window and globalThis

You may want to add a property to the global object. You can do this by adding top-level declarations using the `declare` keyword to your `env.d.ts` file:

`src/env.d.ts`

```

1  declare var myString: string;
2
3  declare function myFunction(): boolean;

```

This will provide typing to `globalThis.myString` and `globalThis.myFunction`, as well as `window.myString` and `window.myFunction`.

Note that `window` is only available in client-side code. `globalThis` is available both server-side and client-side, but its server-side value won't be shared with the client.

If you only want to type a property on the `window` object, provide a `window` interface instead:

`src/env.d.ts`

```

1 interface window {
2
3     myFunction(): boolean;
4
5 }

```

Add non-standard attributes

You may want to define a type for custom attributes or CSS properties. You can extend the default JSX definitions to add non-standard attributes by redeclaring the `astroHTML.JSX` namespace in a `.d.ts` file.

`src/env.d.ts`

```

1 declare namespace astroHTML.JSX { interface HTMLAttributes {     "data-count"?: number;
    "data-label"?: string;  }
2 // Add a CSS custom property to the style object interface CSSProperties {     "--
  theme-color"?: "black" | "white";  }}

```

Note

`astroHTML` is injected globally inside `.astro` components. To use it in TypeScript files, use a [triple-slash directive](#):

```

1 /// <reference types="astro/astro-jsx" />
2 type MyAttributes = astroHTML.JSX.ImgHTMLAttributes;

```

Using imports

You may want to extend global types by reusing types declared elsewhere in your project or from an external library. To do this, use [dynamic imports](#):

`src/env.d.ts`

```

1 type Product = { id: string; name: string; price: number;};
2 declare namespace App { interface Locals {     orders: Map<string, Product[]>
  session: import("../lib/server/session").Session | null;     user: import("my-external-
  library").User;  }}

```

A `.d.ts` file is an [ambient module](#) declaration. While its syntax is similar to ES modules, these files do not allow top-level imports/exports. If TypeScript encounters one, the file will be considered a [module augmentation](#) and this will break your global types.

Component Props

Astro supports typing your component props via TypeScript. To enable, add a TypeScript `Props` interface to your component frontmatter. An `export` statement may be used, but is not necessary. The [Astro VS Code Extension](#) will automatically look for the `Props` interface and give you proper TS support when you use that component inside another template.

`src/components/HelloProps.astro`

```

1  ---interface Props {  name: string;  greeting?: string;}
2  const { greeting = "Hello", name } = Astro.props;---<h2>{greeting}, {name}!</h2>

```

Common prop type patterns

- If your component takes no props or slotted content, you can use `type Props = Record<string, never>`.
- If your component must be passed children to its default slot, you can enforce this by using `type Props = { children: any; };`.

Type Utilities

Added in: `astro@1.6.0`

Astro comes with some built-in utility types for common prop type patterns. These are available under the `astro/types` entrypoint.

Built-in HTML attributes

Astro provides the `HTMLAttributes` type to check that your markup is using valid HTML attributes. You can use these types to help build component props.

For example, if you were building a `<Link>` component, you could do the following to mirror the default HTML attributes for `<a>` tags in your component's prop types.

`src/components/Link.astro`

```

1  ---import type { HTMLAttributes } from "astro/types";
2  // use a `type` type Props = HTMLAttributes<"a">;
3  // or extend with an `interface` interface Props extends HTMLAttributes<"a"> {  myProp?:
  boolean;}
4  const { href, ...attrs } = Astro.props;---<a href={href} {...attrs}>  <slot /></a>

```

ComponentProps type

Added in: `astro@4.3.0`

This type export allows you to reference the `Props` accepted by another component, even if that component doesn't export that `Props` type directly.

The following example shows using the `ComponentProps` utility from `astro/types` to reference a `<Button />` component's `Props` types:

`src/pages/index.astro`

```

1  ---import type { ComponentProps } from "astro/types";import Button from
  "./Button.astro";
2  type ButtonProps = ComponentProps<typeof Button>;---

```

Polymorphic type

Added in: `astro@2.5.0`

Astro includes a helper to make it easier to build components that can render as different HTML elements with full type safety. This is useful for components like `<Link>` that can render as either `<a>` or `<button>` depending on the props passed to it.

The example below implements a fully-typed, polymorphic component that can render as any HTML element. The `HTMLTag` type is used to ensure that the `as` prop is a valid HTML element.

```
1  ---import type { HTMLTag, Polymorphic } from "astro/types";
2  type Props<Tag extends HTMLTag> = Polymorphic<{ as: Tag }>;
3  const { as: Tag, ...props } = Astro.props;---<Tag {...props} />
```

Infer `getStaticPaths()` types

Added in: `astro@2.1.0`

Astro includes helpers for working with the types returned by your `getStaticPaths()` function for dynamic routes.

You can get the type of `Astro.params` with `InferGetStaticParamsType` and the type of `Astro.props` with `InferGetStaticPropsType` or you can use `GetStaticPaths` to infer both at once:

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

```
1  ---import type { InferGetStaticParamsType, InferGetStaticPropsType, GetStaticPaths, }
   from "astro";
2  export const getStaticPaths = (async () => { const posts = await getCollection("blog");
   return posts.map((post) => { return { params: { id: post.id }, props: {
   draft: post.data.draft, title: post.data.title }, }; });}) satisfies GetStaticPaths;
3  type Params = InferGetStaticParamsType<typeof getStaticPaths>; type Props =
   InferGetStaticPropsType<typeof getStaticPaths>;
4  const { id } = Astro.params as Params; // ^? { id: string; }
5  const { title } = Astro.props; // ^? { draft: boolean; title:
   string; }---
```

Type checking

To see type errors in your editor, please make sure that you have the [Astro VS Code extension](#) installed. Please note that the `astro start` and `astro build` commands will transpile the code with esbuild, but will not run any type checking. To prevent your code from building if it contains TypeScript errors, change your “build” script in `package.json` to the following:

`package.json`

```
1 {  
2  
3   "scripts": {  
4  
5     "build": "astro build",  
6  
7     "build": "astro check && astro build",  
8  
9   },  
10  
11 }
```

Note

`astro check` checks all the files included in your TypeScript project. To check types within Svelte and Vue files, you can use the [svelte-check](#) and the [vue-tsc](#) packages respectively.

Read more about [.ts file imports](#) in Astro.

Read more about [TypeScript Configuration](#).

Troubleshooting

Errors typing multiple JSX frameworks at the same time

An issue may arise when using multiple JSX frameworks in the same project, as each framework requires different, sometimes conflicting, settings inside `tsconfig.json`.

Solution: Set the [jsxImportSource](#) setting to `react` (default), `preact` or `solid-js` depending on your most-used framework. Then, use a [pragma comment](#) inside any conflicting file from a different framework.

For the default setting of `jsxImportSource: react`, you would use:

```
1 // For Preact/** @jsxImportSource preact */  
2 // For Solid/** @jsxImportSource solid-js */
```

Using environment variables

Astro gives you access to [Vite's built-in environment variables support](#) and includes some [default environment variables for your project](#) that allow you to access configuration values for your current project (e.g. `site`, `base`), whether your project is running in development or production, and more.

Astro also provides a way to [use and organize your environment variables with type safety](#). It is available for use inside the Astro context (e.g. Astro components, routes and endpoints, UI framework components, middleware), and managed with [a schema in your Astro configuration](#).

Vite's built-in support

Astro uses Vite's built-in support for environment variables, which are statically replaced at build time, and lets you [use any of its methods](#) to work with them.

Note that while *all* environment variables are available in server-side code, only environment variables prefixed with `PUBLIC_` are available in client-side code for security purposes.

`.env`

```
1 SECRET_PASSWORD=password123
2
3 PUBLIC_ANYBODY=there
```

In this example, `PUBLIC_ANYBODY` (accessible via `import.meta.env.PUBLIC_ANYBODY`) will be available in server or client code, while `SECRET_PASSWORD` (accessible via `import.meta.env.SECRET_PASSWORD`) will be server-side only.

Caution

`.env` files are not loaded inside [configuration files](#).

IntelliSense for TypeScript

By default, Astro provides a type definition for `import.meta.env` in `astro/client.d.ts`.

While you can define more custom env variables in `.env.[mode]` files, you may want to get TypeScript IntelliSense for user-defined env variables which are prefixed with `PUBLIC_`.

To achieve this, you can create an `env.d.ts` in `src/` to [extend the global types](#) and configure `ImportMetaEnv` like this:

`src/env.d.ts`

```
1 interface ImportMetaEnv {  readonly DB_PASSWORD: string;  readonly PUBLIC_POKEAPI:
  string;  // more env variables...}
2 interface ImportMeta {  readonly env: ImportMetaEnv;}
```

Default environment variables

Astro includes a few environment variables out of the box:

- `import.meta.env.MODE`: The mode your site is running in. This is `development` when running `astro dev` and `production` when running `astro build`.
- `import.meta.env.PROD`: `true` if your site is running in production; `false` otherwise.
- `import.meta.env.DEV`: `true` if your site is running in development; `false` otherwise. Always the opposite of `import.meta.env.PROD`.
- `import.meta.env.BASE_URL`: The base URL your site is being served from. This is determined by the [base config option](#).
- `import.meta.env.SITE`: This is set to [the site option](#) specified in your project's `astro.config`.

Use them like any other environment variable.

```
1 | const isProd = import.meta.env.PROD;  
2 |  
3 | const isDev = import.meta.env.DEV;
```

Setting environment variables

`.env` files

Environment variables can be loaded from `.env` files in your project directory.

Just create a `.env` file in the project directory and add some variables to it.

`.env`

```
1 | # This will only be available when run on the server!DB_PASSWORD="foobar"  
2 | # This will be available everywhere!PUBLIC_POKEAPI="https://pokeapi.co/api/v2"
```

You can also add `.production`, `.development` or a custom mode name to the filename itself (e.g `.env.testing`, `.env.staging`). This allows you to use different sets of environment variables at different times.

The `astro dev` and `astro build` commands default to `"development"` and `"production"` modes, respectively. You can run these commands with the `--mode` flag to pass a different value for `mode` and load the matching `.env` file.

This allows you to run the dev server or build your site connecting to different APIs:

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | # Run the dev server connected to a "staging" APIInpm run astro dev -- --mode staging  
2 | # Build a site that connects to a "production" API with additional debug informationnpm  
  | run astro build -- --devOutput  
3 | # Build a site that connects to a "testing" APIInpm run astro build -- --mode testing
```

For more on `.env` files, [see the Vite documentation](#).

In the Astro config file

Astro evaluates configuration files before it loads your other files. This means that you cannot use `import.meta.env` in `astro.config.mjs` to access environment variables that were set in `.env` files.

You can use `process.env` in a configuration file to access other environment variables, like those [set by the CLI](#).

You can also use [Vite's loadEnv helper](#) to manually load `.env` files.

`astro.config.mjs`

```
1 import { loadEnv } from "vite";
2 const { SECRET_PASSWORD } = loadEnv(process.env.NODE_ENV, process.cwd(), "");
```

Note

`pnpm` does not allow you to import modules that are not directly installed in your project. If you are using `pnpm`, you will need to install `vite` to use the `loadEnv` helper.

Terminal window

```
1 pnpm add -D vite
```

Using the CLI

You can also add environment variables as you run your project:

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 PUBLIC_POKEAPI=https://pokeapi.co/api/v2 npm run dev
```

Getting environment variables

Environment variables in Astro are accessed with `import.meta.env`, using the [import.meta feature added in ES2020](#), instead of `process.env`.

For example, use `import.meta.env.PUBLIC_POKEAPI` to get the `PUBLIC_POKEAPI` environment variable.

```
1 // when import.meta.env.SSR === trueconst data = await db(import.meta.env.DB_PASSWORD);
2 // when import.meta.env.SSR === falseconst data =
  fetch(`${import.meta.env.PUBLIC_POKEAPI}/pokemon/squirtle`);
```

When using SSR, environment variables can be accessed at runtime based on the SSR adapter being used. With most adapters you can access environment variables with `process.env`, but some adapters work differently. For the Deno adapter, you will use `Deno.env.get()`. See how to [access the Cloudflare runtime](#) to handle environment variables when using the Cloudflare adapter. Astro will first check the server environment for variables, and if they don't exist, Astro will look for them in `.env` files.

Type safe environment variables

The `astro:env` API lets you configure a type-safe schema for [environment variables you have set](#). This allows you to indicate whether they should be available on the server or the client, and define their data type and additional properties.

Developing an adapter? See how to [make an adapter compatible with `astro:env`](#).

Basic Usage

Define your schema

To configure a schema, add the `env.schema` option to your Astro config:

astro.config.mjs

```
1 import { defineConfig } from "astro/config";
2 export default defineConfig({ env: { schema: { // ... } } })
```

You can then [register variables as a string, number, enum, or boolean](#) using the `envField` helper. Define the [kind of environment variable](#) by providing a `context` (`"client"` or `"server"`) and `access` (`"secret"` or `"public"`) for each variable, and pass any additional properties such as `optional` or `default` in an object:

astro.config.mjs

```
1 import { defineConfig, envField } from "astro/config";
2 export default defineConfig({ env: { schema: { API_URL: envField.string({
  context: "client", access: "public", optional: true }), PORT: envField.number({
  context: "server", access: "public", default: 4321 }), API_SECRET:
  envField.string({ context: "server", access: "secret" }), } } })
```

Types will be generated for you when running `astro dev` or `astro build`, but you can run `astro sync` to generate types only.

Use variables from your schema

Import and use your defined variables from the appropriate `/client` or `/server` module:

```
1 ---import { API_URL } from "astro:env/client";import { API_SECRET_TOKEN } from
  "astro:env/server";
2 const data = await fetch(`${API_URL}/users`, { method: "GET", headers: { "Content-
  Type": "application/json", "Authorization": `Bearer ${API_SECRET_TOKEN}` },})---
3 <script> import { API_URL } from "astro:env/client";
4   fetch(`${API_URL}/ping`)</script>
```

Variable types

There are three kinds of environment variables, determined by the combination of `context` (`"client"` or `"server"`) and `access` (`"secret"` or `"public"`) settings defined in your schema:

- **Public client variables:** These variables end up in both your final client and server bundles, and can be

accessed from both client and server through the `astro:env/client` module:

```
1 | import { API_URL } from "astro:env/client";
```

- **Public server variables:** These variables end up in your final server bundle and can be accessed on the server through the `astro:env/server` module:

```
1 | import { PORT } from "astro:env/server";
```

- **Secret server variables:** These variables are not part of your final bundle and can be accessed on the server through the `astro:env/server` module:

```
1 | import { API_SECRET } from "astro:env/server";
```

By default, all secrets are validated whenever anything is imported from the `astro:env/server` module. This means, secrets may be validated even when they are not imported. You may need to [pass dummy environment variables](#) to satisfy this validation during the build.

You can also enable validating secrets on start by [configuring `validateSecrets: true`](#).

Note

Secret client variables are not supported because there is no safe way to send this data to the client. Therefore, it is not possible to configure both `context: "client"` and `access: "secret"` in your schema.

Data types

There are currently four data types supported: strings, numbers, enums, and booleans:

```
1 | import { envField } from "astro/config";
2 | envField.string({ // context & access optional: true, default: "foo",})
3 | envField.number({ // context & access optional: true, default: 15,})
4 | envField.boolean({ // context & access optional: true, default: true,})
5 | envField.enum({ // context & access values: ["foo", "bar", "baz"], optional: true,
   | default: "baz",})
```

For a complete list of validation fields, see the [envField API reference](#).

Retrieving secrets dynamically

Despite defining your schema, you may want to retrieve the raw value of a given secret or to retrieve secrets not defined in your schema. In this case, you can use `getSecret()` exported from `astro:env/server`:

```
1 | import { FOO, // boolean getSecret } from "astro:env/server";
2 | getSecret("FOO"); // string | undefined
```

Learn more in [the API reference](#).

Limitations

`astro:env` is a virtual module which means it can only be used inside the Astro context. For example, you can use it in:

- Middlewares
- Astro routes and endpoints
- Astro components
- Framework components
- Modules

You cannot use it in the following and will have to resort to `process.env`:

- `astro.config.mjs`
- Scripts

Working with integrations

Astro integrations add new functionality and behaviors for your project with only a few lines of code. You can use an official integration, [integrations built by the community](#) or even [build a custom integration yourself](#).

Integrations can...

- Unlock React, Vue, Svelte, Solid, and other popular UI frameworks with a [renderer](#).
- Enable on-demand rendering with an [SSR adapter](#).
- Integrate tools like MDX, and Partytown with a few lines of code.
- Add new features to your project, like automatic sitemap generation.
- Write custom code that hooks into the build process, dev server, and more.

Integrations directory

Browse or search the complete set of hundreds of official and community integrations in our [integrations directory](#). Find packages to add to your Astro project for authentication, analytics, performance, SEO, accessibility, UI, developer tools, and more.

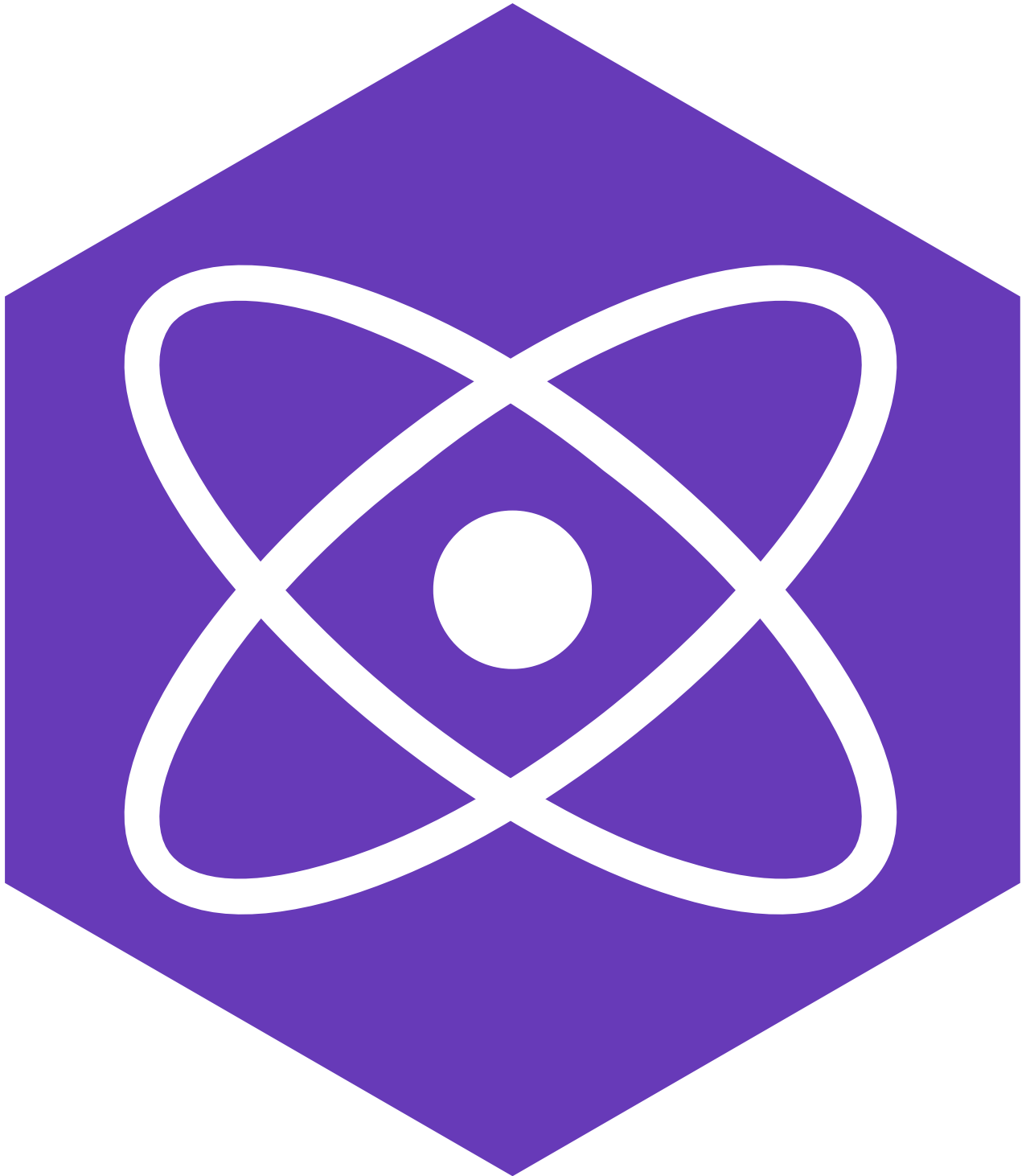
Official integrations

The following integrations are maintained by Astro.

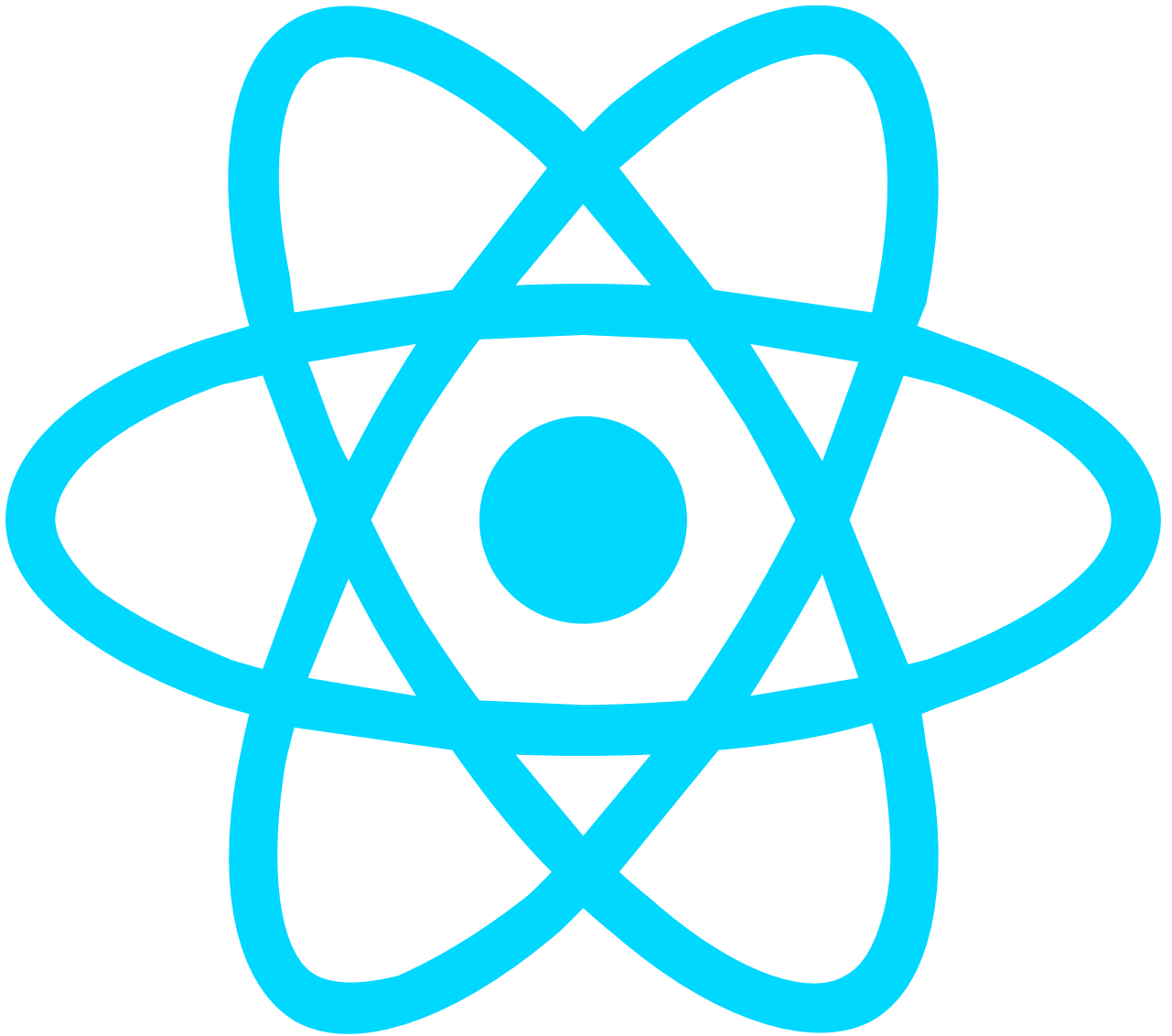
Front-end frameworks



- [@astrojs/alpinejs](https://astrojs.alpinejs)



- [@astrojs/preact](#)



- [@astrojs/react](#)



- [@astrojs/solid-js](#)



- [@astrojs/svelte](#)



- [@astrojs/vue](#)

Adapters



- [@astrojs/cloudflare](#)



- [@astrojs/netlify](https://astrojs.netlify.com)



- [@astrojs/node](#)

- [@astrojs/vercel](#)

Other integrations

DOC

- [@astrojs/markdoc](#)

- [@astrojs/mdx](#)



- [@astrojs/partytown](#)

- [@astrojs/sitemap](#)

Automatic integration setup

Astro includes an `astro add` command to automate the setup of official integrations. Several community plugins can also be added using this command. Please check each integration's own documentation to see whether `astro add` is supported, or whether you must [install manually](#).

Run the `astro add` command using the package manager of your choice and our automatic integration wizard will update your configuration file and install any necessary dependencies.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npx astro add react
```

It's even possible to add multiple integrations at the same time!

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npx astro add react sitemap partytown
```

Handling integration dependencies

If you see any warnings like `Cannot find package '[package-name]'` after adding an integration, your package manager may not have installed [peer dependencies](#) for you. To install these missing packages, run the following command:

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npm install [package-name]
```

Manual installation

Astro integrations are always added through the `integrations` property in your `astro.config.mjs` file.

There are three common ways to import an integration into your Astro project:

1. [Install an npm package integration](#).
2. Import your own integration from a local file inside your project.
3. Write your integration inline, directly in your config file.

`astro.config.mjs`

```
1 | import { defineConfig } from 'astro/config';import installedIntegration from
  '@astrojs/vue';import localIntegration from './my-integration.js';
2 | export default defineConfig({ integrations: [ // 1. Imported from an installed
    npm package    installedIntegration(), // 2. Imported from a local JS file
    localIntegration(), // 3. An inline object    { name: 'namespace:id', hooks: {
    /* ... */ } }, ]});
```

Check out the [Integration API](#) reference to learn all of the different ways that you can write an integration.

Installing an npm package

Install an npm package integration using a package manager, and then update `astro.config.mjs` manually.

For example, to install the `@astrojs/sitemap` integration:

1. Install the integration to your project dependencies using your preferred package manager:
 - [npm](#)
 - [pnpm](#)
 - [Yarn](#)

Terminal window

```
1 | npm install @astrojs/sitemap
```

2. Import the integration to your `astro.config.mjs` file, and add it to your `integrations[]` array, along with any configuration options:

`astro.config.mjs`

```
1 | import { defineConfig } from 'astro/config';import sitemap from '@astrojs/sitemap';
2 | export default defineConfig({ // ... integrations: [sitemap()], // ...});
```

Note that different integrations may have different configuration settings. Read each integration's

documentation, and apply any necessary config options to your chosen integration in `astro.config.mjs`.

Custom options

Integrations are almost always authored as factory functions that return the actual integration object. This lets you pass arguments and options to the factory function that customize the integration for your project.

```
1 integrations: [  
2  
3   // Example: Customize your integration with function arguments  
4  
5   sitemap({ filter: true })  
6  
7 ]
```

Toggle an integration

Falsy integrations are ignored, so you can toggle integrations on & off without worrying about left-behind `undefined` and boolean values.

```
1 integrations: [  
2  
3   // Example: Skip building a sitemap on windows  
4  
5   process.platform !== 'win32' && sitemap()  
6  
7 ]
```

Upgrading integrations

To upgrade all official integrations at once, run the `@astrojs/upgrade` command. This will upgrade both Astro and all official integrations to their latest versions.

Automatic upgrading

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 # Upgrade Astro and official integrations together to latest  
2  
3 npx @astrojs/upgrade
```

Manual upgrading

To upgrade one or more integrations manually, use the appropriate command for your package manager.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 # Example: upgrade React and Partytown integrations
2
3 npm install @astrojs/react@latest @astrojs/partytown@latest
```

Removing an integration

1. To remove an integration, first uninstall the integration from your project.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 npm uninstall @astrojs/react
```

2. Next, remove the integration from your `astro.config.*` file:

`astro.config.mjs`

```
1 import { defineConfig } from 'astro/config';import react from '@astrojs/react';
2 export default defineConfig({ integrations: [ react() ]});
```

Finding more integrations

You can find many integrations developed by the community in the [Astro Integrations Directory](#). Follow links there for detailed usage and configuration instructions.

Building your own integration

Astro's Integration API is inspired by Rollup and Vite, and designed to feel familiar to anyone who has ever written a Rollup or Vite plugin before.

Check out the [Integration API](#) reference to learn what integrations can do and how to write one yourself.

Publishing your integration to npm

Publishing an Astro component is a great way to reuse your existing work across your projects, and to share with the wider Astro community at large. Astro components can be published directly to and installed from npm, just like any other JavaScript package.

Looking for inspiration? Check out some of our favorite [themes](#) and [components](#) from the Astro community. You can also [search npm](#) to see the entire public catalog.

Don't want to go it alone?

Check out [Astro community's component template](#) for a community-supported, out-of-the-box template!

Quick start

To get started developing your component quickly, you can use a template already set up for you.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 # Initialize the Astro Component template in a new directory
2
3 npm create astro@latest my-new-component-directory -- --template component
```

Creating a package

Prerequisites

Before diving in, it will help to have a basic understanding of:

- [Node Modules](#)
- [Package Manifest \(`package.json` \)](#)
- [Workspaces](#)

To create a new package, configure your development environment to use **workspaces** within your project. This will allow you to develop your component alongside a working copy of Astro.

Directorymy-new-component-directory/Directorydemo/...for testing and demonstrationpackage.jsonDirectorypackages/Directorymy-component/index.jspackage.json...additional files used by the package

This example, named `my-project`, creates a project with a single package, named `my-component`, and a `demo/` directory for testing and demonstrating the component.

This is configured in the project root's `package.json` file:

```
1 {
2
3   "name": "my-project",
4
5   "workspaces": ["demo", "packages/*"]
6
7 }
```

In this example, multiple packages can be developed together from the `packages` directory. These packages can also be referenced from `demo`, where you can install a working copy of Astro.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 npm create astro@latest demo -- --template minimal
```

There are two initial files that will make up your individual package: `package.json` and `index.js`.

`package.json`

The `package.json` in the package directory includes all of the information related to your package, including its description, dependencies, and any other package metadata.

```
1 {
2
3   "name": "my-component",
4
5   "description": "Component description",
6
7   "version": "1.0.0",
8
9   "homepage": "https://github.com/owner/project#readme",
10
11  "type": "module",
12
13  "exports": {
14
15    ".": "./index.js",
16
17    "./astro": "./MyAstroComponent.astro",
18
19    "./react": "./MyReactComponent.jsx"
20  },
21
22  "files": ["index.js", "MyAstroComponent.astro", "MyReactComponent.jsx"],
23 }
```

```
24 |  
25 |   "keywords": ["astro-component", "withastro", "... etc", "... etc"]  
26 |  
27 | }
```

description

A short description of your component used to help others know what it does.

```
1 | {  
2 |  
3 |   "description": "An Astro Element Generator"  
4 |  
5 | }
```

type

The module format used by Node.js and Astro to interpret your `index.js` files.

```
1 | {  
2 |  
3 |   "type": "module"  
4 |  
5 | }
```

Use `"type": "module"` so that your `index.js` can be used as an entrypoint with `import` and `export` .

homepage

The url to the project homepage.

```
1 | {  
2 |  
3 |   "homepage": "https://github.com/owner/project#readme"  
4 |  
5 | }
```

This is a great way to direct users to an online demo, documentation, or homepage for your project.

exports

The entry points of a package when imported by name.

```

1  {
2
3    "exports": {
4
5      ".": "./index.js",
6
7      "./astro": "./MyAstroComponent.astro",
8
9      "./react": "./MyReactComponent.jsx"
10
11    }
12
13  }

```

In this example, importing `my-component` would use `index.js`, while importing `my-component/astro` or `my-component/react` would use `MyAstroComponent.astro` or `MyReactComponent.jsx` respectively.

files

An optional optimization to exclude unnecessary files from the bundle shipped to users via npm. Note that **only files listed here will be included in your package**, so if you add or change files necessary for your package to work, you must update this list accordingly.

```

1  {
2
3    "files": ["index.js", "MyAstroComponent.astro", "MyReactComponent.jsx"]
4
5  }

```

keywords

An array of keywords relevant to your component, used to help others [find your component on npm](#) and in any other search catalogs.

Add `astro-component`, `astro-integration`, or `withastro` as a special keyword to maximize its discoverability in the Astro ecosystem.

```

1  {
2
3    "keywords": ["astro-component", "withastro", "... etc", "... etc"]
4
5  }

```

Tip

Keywords are also used by our [integrations library](#)! [See below](#) for a full list of keywords we look for in npm.

index.js

The main **package entrypoint** used whenever your package is imported.

```
1 export { default as MyAstroComponent } from './MyAstroComponent.astro';
2
3 export { default as MyReactComponent } from './MyReactComponent.jsx';
```

This allows you to package multiple components together into a single interface.

Example: Using named imports

```
1 ---
2
3 import { MyAstroComponent } from 'my-component';
4
5 import { MyReactComponent } from 'my-component';
6
7 ---
8
9 <MyAstroComponent />
10
11 <MyReactComponent />
```

Example: Using namespace imports

```
1 ---
2
3 import * as Example from 'example-astro-component';
4
5 ---
6
7 <Example.MyAstroComponent />
8
9 <Example.MyReactComponent />
```

Example: Using individual imports

```
1 ---
2
3 import MyAstroComponent from 'example-astro-component/astro';
4
5 import MyReactComponent from 'example-astro-component/react';
6
7 ---
8
9 <MyAstroComponent />
10
11 <MyReactComponent />
```

Developing your package

Astro does not have a dedicated “package mode” for development. Instead, you should use a demo project to develop and test your package inside of your project. This can be a private website only used for development, or a public demo/documentation website for your package.

If you are extracting components from an existing project, you can even continue to use that project to develop your now-extracted components.

Testing your component

Astro does not currently ship a test runner. *(If you are interested in helping out with this, [join us on Discord!](#))*

In the meantime, our current recommendation for testing is:

1. Add a test `fixtures` directory to your `demo/src/pages` directory.
2. Add a new page for every test that you’d like to run.
3. Each page should include some different component usage that you’d like to test.
4. Run `astro build` to build your fixtures, then compare the output of the `dist/__fixtures__` directory to what you expected.

Directorymy-project/demo/src/pages/**fixtures**/test-name-01.astrotest-name-02.astrotest-name-03.astro

Publishing your component

Once you have your package ready, you can publish it to npm using the `npm publish` command. If that fails, make sure that you have logged in via `npm login` and that your `package.json` is correct. If it succeeds, you’re done!

Notice that there was no `build` step for Astro packages. Any file type that Astro supports natively, such as `.astro`, `.ts`, `.jsx`, and `.css`, can be published directly without a build step.

If you need another file type that isn’t natively supported by Astro, add a build step to your package. This advanced exercise is left up to you.

Integrations library

Share your hard work by adding your integration to our [integrations library](#)!

Tip

Do you need some help building your integration, or just want to meet other integrations builders? We have a dedicated `#integrations` channel on our [Discord server](#). Come say hi!

`package.json` data

The library is automatically updated weekly, pulling in every package published to npm with the `astro-component`, `astro-integration`, or `withastro` keyword.

The integrations library reads the `name`, `description`, `repository`, and `homepage` data from your `package.json`.

Avatars are a great way to highlight your brand in the library! Once your package is published you can [file a GitHub issue](#) with your avatar attached and we will add it to your listing.

Tip

Need to override the information our library reads from npm? No problem! [File an issue](#) with the updated information and we'll make sure the custom `name`, `description`, or `homepage` is used instead.

Categories

In addition to the required `astro-component`, `astro-integration`, or `withastro` keyword, special keywords are also used to automatically organize packages. Including any of the keywords below will add your integration to the matching category in our integrations library.

category	keywords
Accessibility	<code>ally</code> , <code>accessibility</code>
Adapters	<code>astro-adapter</code>
Analytics	<code>analytics</code>
CSS + UI	<code>css</code> , <code>ui</code> , <code>icon</code> , <code>icons</code> , <code>renderer</code>
Frameworks	<code>renderer</code>
Content Loaders	<code>astro-loader</code>
Images + Media	<code>media</code> , <code>image</code> , <code>images</code> , <code>video</code> , <code>audio</code>
Performance + SEO	<code>performance</code> , <code>perf</code> , <code>seo</code> , <code>optimization</code>
Dev Toolbar	<code>devtools</code> , <code>dev-overlay</code> , <code>dev-toolbar</code>
Utilities	<code>tooling</code> , <code>utils</code> , <code>utility</code>

Packages that don't include any keyword matching a category will be shown as `Uncategorized`.

Building Astro sites with AI tools

AI-powered editors and agentic coding tools generally have good knowledge of Astro's core APIs and concepts. However, some may use older APIs and may not be aware of newer features or recent changes to the framework.

This guide covers how to enhance AI tools with up-to-date Astro knowledge and provides best practices for building Astro sites with AI assistance.

Astro Docs MCP Server

You can ensure your AI tools have current Astro knowledge through the Astro Docs MCP (Model Context Protocol) server. This provides real-time access to the latest documentation, helping AI tools avoid outdated recommendations and ensuring they understand current best practices.

What is MCP?

[Model Context Protocol](#) (MCP) is a standardized way for AI tools to access external tools and data sources.

Unlike AI models trained on static data, the MCP server provides access to the latest Astro documentation. The server is free, open-source, and runs remotely with nothing to install locally.

The Astro Docs MCP server uses the [kapa.ai](#) API to maintain an up-to-date index of the Astro documentation.

Server Details

- **Name:** Astro Docs
- **URL:** `https://mcp.docs.astro.build/mcp`
- **Transport:** Streamable HTTP

Installation

The setup process varies depending on your AI development tool. You may see some tools refer to MCP servers as connectors, adapters, extensions, or plugins.

Manual setup

Many tools support a common JSON configuration format for MCP servers. If there are not specific instructions for your chosen tool, you may be able to add the Astro Docs MCP server by including the following configuration in your tool's MCP settings:

- [Streamable HTTP](#)
- [Local Proxy](#)

MCP Configuration

```
1 {  
2  
3   "mcpServers": {  
4  
5     "Astro docs": {  
6  
7       "type": "http",  
8  
9       "url": "https://mcp.docs.astro.build/mcp"  
10  
11     }  
12  
13   }  
14  
15 }
```

Claude Code CLI

[Claude Code](#) is an agentic coding tool that runs on the command line. Enabling the Astro Docs MCP server allows it to access the latest documentation while generating Astro code.

Install using the terminal command:

Terminal window

```
1 | claude mcp add --transport http astro-docs https://mcp.docs.astro.build/mcp
```

[More info on using MCP servers with Claude Code](#)

Claude Code GitHub Action

Claude Code also provides a GitHub Action that can be used to run commands in response to GitHub events. Enabling the Astro Docs MCP server allows it to access the latest documentation while answering questions in comments or generating Astro code.

You can configure it to use the Astro Docs MCP server for documentation access by adding the following to the workflow file:

.github/workflows/claude.yml

```
1 | # ...rest of your workflow configuration
2 |
3 | - uses: anthropics/claude-code-action@beta
4 |
5 |   with:
6 |
7 |     anthropic_api_key: ${ secrets.ANTHROPIC_API_KEY }
8 |
9 |     mcp_config: |
10 |
11 |       {
12 |
13 |         "mcpServers": {
14 |
15 |           "astro-docs": {
16 |
17 |             "type": "http",
18 |
19 |             "url": "https://mcp.docs.astro.build/mcp"
20 |
21 |           }
22 |
23 |         }
24 |
25 |       }
26 |
27 |     allowed_tools: "mcp__astro-docs__search_astro_docs"
```

[More info on using MCP servers with the Claude Code GitHub Action](#)

Codex CLI

Codex CLI is a command-line AI coding tool that can use the Astro Docs MCP server to access documentation while generating Astro code.

You can configure MCP servers at the global level in the `~/.codex/config.toml` file, or in a `.codex/config.toml` file in a project root.

`~/.codex/config.toml`

```
1 [mcp_servers.astro-docs]
2
3 command = "npx"
4
5 args = ["-y", "mcp-remote", "https://mcp.docs.astro.build/mcp"]
```

[More info on using MCP servers with Codex CLI](#)

Cursor

[Cursor](#) is an AI code editor. Adding the Astro Docs MCP server allows Cursor to access the latest Astro documentation while performing development tasks.

Install by clicking the button below:

[Add to Cursor](#)

[More info on using MCP servers with Cursor](#)

Visual Studio Code

[Visual Studio Code](#) supports MCP servers when using Copilot Chat. Adding the Astro Docs MCP server allows VS Code to access the latest Astro documentation when answering questions or performing coding tasks.

Install by clicking the button below:

[Add to VS Code](#)

[More info on using MCP servers with VS Code](#)

Warp

[Warp](#) (formerly Warp Terminal) is an agent development environment built for coding with multiple AI agents. Adding the Astro Docs MCP server allows Warp to access the latest Astro documentation when answering questions or performing coding tasks.

1. Open your Warp settings and go to AI > MCP Servers > Manage MCP Servers.
2. Click "Add".
3. Enter the following configuration. You can optionally configure the Astro MCP server to activate on startup using the `start_on_launch` flag:

MCP Configuration

```
1 {
2
```

```

3   "mcpServers": {
4
5       "Astro docs": {
6
7           "command": "npx",
8
9           "args": ["-y", "mcp-remote", "https://mcp.docs.astro.build/mcp"],
10
11          "env": {},
12
13          "working_directory": null,
14
15          "start_on_launch": true
16      }
17  }
18
19  }
20
21  }

```

4. Click “Save”.

[More info on using MCP servers with Warp](#)

Claude.ai / Claude Desktop

[Claude.ai](#) is a general-purpose AI assistant. Adding the Astro Docs MCP server allows it to access the latest documentation when answering Astro questions or generating Astro code.

1. Navigate to the [Claude.ai connector settings](#).
2. Click “Add custom connector”. You may need to scroll down to find this option.
3. Enter the server URL: `https://mcp.docs.astro.build/mcp`.
4. Set the name to “Astro docs”.

[More info on using MCP servers with Claude.ai](#)

Windsurf

[Windsurf](#) is an AI-powered agentic coding tool, available as editor plugins or a standalone editor. It can use the Astro Docs MCP server to access documentation while performing coding tasks.

Windsurf doesn’t support streaming HTTP, so it requires a local proxy configuration:

1. Open `~/codeium/windsurf/mcp_config.json` in your editor.
2. Add the following configuration to your Windsurf MCP settings:

MCP Configuration

```

1   {
2
3       "mcpServers": {
4
5           "Astro docs": {
6

```

```

7      "command": "npx",
8
9      "args": ["-y", "mcp-remote", "https://mcp.docs.astro.build/mcp"]
10
11    }
12
13  }
14
15 }

```

3. Save the configuration and restart Windsurf.

[More info on using MCP servers with Windsurf](#)

Gemini CLI

Gemini CLI is a command-line AI coding tool that can use the Astro Docs MCP server to access documentation while generating Astro code.

You can configure MCP servers at the global level in the `~/.gemini/settings.json` file, or in a `.gemini/settings.json` file in a project root.

`.gemini/settings.json`

```

1  {
2
3    "mcpServers": {
4
5      "Astro docs": {
6
7        "httpUrl": "https://mcp.docs.astro.build/mcp",
8
9      }
10
11    }
12
13  }

```

[More info on using MCP servers with Gemini CLI](#)

Google Antigravity

[Google Antigravity](#) is an agentic development platform.

1. Open `~/.gemini/antigravity/mcp_config.json` by following the [Connecting Custom MCP Servers guide](#).
2. Add the following configuration to `mcp_config.json`:
`mcp_config.json`

```

1  {
2
3    "mcpServers": {
4
5      "astro-docs": {
6
7        "serverUrl": "https://mcp.docs.astro.build/mcp"
8
9      }
10
11    }
12
13  }

```

3. Save the file and click “Refresh” in the “Manage MCPs” tab.

Zed

[Zed](#) supports MCP servers when using its AI capabilities. It can use the Astro Docs MCP server to access documentation while performing coding tasks.

1. Open `~/.config/zed/settings.json` in your editor.
2. Add the following configuration to your Zed MCP settings:

MCP Configuration

```

1  {
2
3    "context_servers": {
4
5      "Astro docs": {
6
7        "settings": {},
8
9        "enabled": true,
10
11        "url": "https://mcp.docs.astro.build/mcp"
12
13      }
14
15    }
16
17  }

```

3. Save the configuration.

[More info on using MCP servers with Zed](#)

ChatGPT

Refer to the [OpenAI MCP documentation](#) for specific setup instructions.

Raycast

[Raycast](#) can connect to MCP servers to enhance its AI capabilities. Adding the Astro Docs MCP server allows Raycast to access the latest Astro documentation while answering questions.

Install by clicking the button below:

[Add to Raycast](#)

[More info on using MCP servers with Raycast](#)

Opencode AI

[Opencode AI](#) is an open-source, terminal-based AI coding tool that can use the Astro Docs MCP server to access documentation while generating Astro code.

You can configure MCP servers in your Opencode configuration file, typically named `opencode.json`, located in your project root or your global configuration directory (e.g. `~/.config/opencode/opencode.json`).

MCP Configuration

```
1  {
2
3    "$schema": "https://opencode.ai/config.json",
4
5    "mcp": {
6
7      "Astro docs": {
8
9        "type": "remote",
10
11        "url": "https://mcp.docs.astro.build/mcp",
12
13        "enabled": true
14      }
15    }
16  }
17 }
18
19 }
```

[More info on using Opencode AI](#)

GitHub Copilot Coding Agent

[GitHub Copilot](#) can be used as a coding agent powered by GitHub Actions. Enabling the Astro Docs MCP server allows it to access the latest Astro documentation when answering questions or performing coding tasks.

You can configure it to use the Astro Docs MCP server for documentation access by adding the following to your repository's Copilot coding agent settings available at `https://github.com/<your-org>/<your-repo>/settings/copilot/coding_agent`:

MCP Configuration

```
1 {  
2  
3   "mcpServers": {  
4  
5     "astro-docs": {  
6  
7       "type": "http",  
8  
9       "url": "https://mcp.docs.astro.build/mcp",  
10  
11      "tools": ["mcp__astro-docs__search_astro_docs"]  
12    }  
13  }  
14  
15 }  
16  
17 }
```

Learn more about [extending GitHub Copilot coding agent with MCP servers](#).

Usage

Once configured, you can ask your AI tool questions about Astro, and it will retrieve information directly from the latest docs. Coding agents will be able to consult the latest documentation when performing coding tasks, and chatbots will be able to accurately answer questions about Astro features, APIs, and best practices.

Remember

The Astro Docs MCP server provides access to current documentation, but your AI tools are still responsible for interpretation and code generation. AI makes mistakes, so always review generated code carefully and test thoroughly.

Troubleshooting

If you encounter issues:

- Verify that your tool supports streamable HTTP transport.
- Check that the server URL is correct: `https://mcp.docs.astro.build/mcp`.
- Ensure your tool has proper internet access.
- Consult your specific tool's MCP integration documentation.

If you are still having problems, open an issue in the [Astro Docs MCP Server repository](#).

Discord AI Support

The same technology that powers Astro's MCP server is also available as a chatbot in the [Astro Discord](#) for self-serve support. Visit the `#support-ai` channel to ask questions about Astro or your project code in natural language. Your conversation is automatically threaded, and you can ask an unlimited number of follow-up questions.

Conversations with the chatbot are public, and are subject to the same server rules for language and behavior as the rest of our channels, but they are not actively visited by our volunteer support members. For assistance from the community, please create a thread in our regular `#support` channel.

Background mode

Added in: `astro@7.0.0` New

When an AI coding agent is detected, `astro dev` automatically starts the dev server as a detached background process. This prevents the dev server from blocking the agent's terminal and allows it to continue working while the server runs.

A lock file (`.astro/dev.json`) is written when the dev server starts, recording the server's URL, port, and PID. This prevents duplicate servers from being started for the same project.

If you are not using an AI coding agent, `astro dev` starts in the foreground process and logs to the terminal.

To opt out of automatic background mode, set the `ASTRO_DEV_BACKGROUND` environment variable before running `astro dev`:

Terminal window

```
1 | ASTRO_DEV_BACKGROUND=0 astro dev
```

See the [CLI reference](#) for the full list of `astro dev` flags and subcommands.

Health endpoint

The dev server exposes a `/_astro/status` endpoint that returns `{"ok": true}` as JSON. This allows agents and other tools to check programmatically whether the dev server is ready to accept requests. This endpoint is only available in the dev server and does not exist in production builds.

Tips for AI-Powered Astro Development

- **Start with templates:** Rather than building from scratch, ask AI tools to start with an existing [Astro template](#) or use `npm create astro@latest` with a template option.
 - **Use `astro add` for integrations:** Ask AI tools to use `astro add` for official integrations (e.g. `astro add tailwind`, `astro add react`). For other packages, install using the command for your preferred package manager rather than editing `package.json` directly.
 - **Verify current APIs:** AI tools may use outdated patterns. Ask them to check the latest documentation, especially for newer features like sessions and actions. This is also important for features that have seen significant changes since their initial launch, such as content collections, or previously experimental features that may no longer be experimental.
 - **Use project rules:** If your AI tool supports it, set up project rules to enforce best practices and coding standards, such as the ones listed above.
-

Dev toolbar

While the dev server is running, Astro includes a dev toolbar at the bottom of every page in your local browser preview.

This toolbar includes a number of useful tools for debugging and inspecting your site during development and can be [extended with more dev toolbar apps](#) found in the integrations directory. You can even [build your own toolbar apps](#) using the [Dev Toolbar API](#)!

This toolbar is enabled by default and appears when you hover over the bottom of the page. It is a development tool only and will not appear on your published site.

Built-in apps

Astro Menu

The Astro Menu app provides easy access to various information about the current project and links to extra resources. Notably, it provides one-click access to the Astro documentation, GitHub repository, and Discord server.

This app also includes a “Copy debug info” button which will run the `astro info` command and copy the output to your clipboard. This can be useful when asking for help or reporting issues.

Inspect

The Inspect app provides information about any [islands](#) on the current page. This will show you the properties passed to each island, and the client directive that is being used to render them.

Audit

The Audit app automatically runs a series of audits on the current page, checking for the most common performance and accessibility issues. When an issue is found, a red dot will appear in the toolbar. Clicking on the app will pop up a list of results from the audit and will highlight the related elements directly in the page.

Note

The basic performance and accessibility audits performed by the dev toolbar are not a replacement for dedicated tools like [Pa11y](#) or [Lighthouse](#), or even better, humans!

The dev toolbar aims to provide a quick and easy way to catch common issues during development, without needing to context-switch to a different tool.

Settings

The Settings app allows you to configure options for the dev toolbar, such as verbose logging, disabling notifications, and adjusting its placement on your screen.

Extending the dev toolbar

Astro integrations can add new apps to the dev toolbar, allowing you to extend it with custom tools that are specific to your project. You can find [more dev tool apps to install in the integrations directory](#) or using the [Astro Menu](#).

Install additional dev toolbar app integrations in your project just like any other [Astro integration](#) according to its own installation instructions.



Related recipe: [Create a dev toolbar app](#)

Disabling the dev toolbar

The dev toolbar is enabled by default for every site. You can choose to disable it for individual projects and/or users as needed.

Per-project

To disable the dev toolbar for everyone working on a project, set `devToolbar: false` in the [Astro config file](#).

`astro.config.mjs`

```
1 import { defineConfig } from "astro/config";
2 export default defineConfig({ devToolbar: { enabled: false } });
```

To enable the dev toolbar again, remove these lines from your configuration, or set `enabled: true`.

Per-user

To disable the dev toolbar for yourself on a specific project, run the [astro preferences](#) command.

Terminal window

```
1 astro preferences disable devToolbar
```

To disable the dev toolbar in all Astro projects for a user on the current machine, add the `--global` flag when running `astro-preferences`:

Terminal window

```
1 astro preferences disable --global devToolbar
```

The dev toolbar can later be enabled with:

Terminal window

```
1 astro preferences enable devToolbar
```

Pages

Pages are files that live in the `src/pages/` subdirectory of your Astro project. They are responsible for handling routing, data loading, and overall page layout for every page in your website.

Supported page files

Astro supports the following file types in the `src/pages/` directory:

- `.astro`
- `.md`
- `.mdx` (with the [MDX Integration installed](#))
- `.html`
- `.js` / `.ts` (as [endpoints](#))

File-based routing

Astro leverages a routing strategy called **file-based routing**. Each file in your `src/pages/` directory becomes an endpoint on your site based on its file path.

A single file can also generate multiple pages using [dynamic routing](#). This allows you to create pages even if your content lives outside of the special `/pages/` directory, such as in a [content collection](#) or a [CMS](#).

Read more about [Routing in Astro](#).

Link between pages

Write standard HTML [`elements`](#) in your Astro pages to link to other pages on your site. Use a **URL path relative to your root domain** as your link, not a relative file path.

For example, to link to `https://example.com/authors/sonali/` from any other page on `example.com`:

`src/pages/index.astro`

```
1 | Read more <a href="/authors/sonali/">about Sonali</a>.
```

Astro Pages

Astro pages use the `.astro` file extension and support the same features as [Astro components](#).

`src/pages/index.astro`

```
1 | ---
2 |
3 | ---
4 |
5 | <html lang="en">
6 |
7 |   <head>
8 |
```

```

 9      <title>My Homepage</title>
10
11    </head>
12
13    <body>
14
15      <h1>welcome to my website!</h1>
16
17    </body>
18
19  </html>

```

A page must produce a full HTML document. If not explicitly included, Astro will add the necessary `<!DOCTYPE html>` declaration and `<head>` content to any `.astro` component located within `src/pages/` by default. You can opt-out of this behavior on a per-component basis by marking it as a [partial](#) page.

To avoid repeating the same HTML elements on every page, you can move common `<head>` and `<body>` elements into your own [layout components](#). You can use as many or as few layout components as you'd like.

`src/pages/index.astro`

```

 1  ---
 2
 3  import MySiteLayout from "../layouts/MySiteLayout.astro";
 4
 5  ---
 6
 7  <MySiteLayout>
 8
 9    <p>My page content, wrapped in a layout!</p>
10
11  </MySiteLayout>

```

Read more about [layout components](#) in Astro.

Markdown/MDX Pages

Astro also treats any Markdown (`.md`) files inside of `src/pages/` as pages in your final website. If you have the [MDX Integration installed](#), it also treats MDX (`.mdx`) files the same way.

Tip

Consider creating [content collections](#) instead of pages for directories of related Markdown files that share a similar structure, such as blog posts or product items.

Markdown files can use the special `layout` frontmatter property to specify a [layout component](#) that will wrap their Markdown content in a full `<html>...</html>` page document.

`src/pages/page.md`

```

 1  ---layout: ../layouts/MySiteLayout.astrotitle: My Markdown page---# Title
 2  This is my page, written in **Markdown.**

```

Read more about [Markdown](#) in Astro.

HTML Pages

Files with the `.html` file extension can be placed in the `src/pages/` directory and used directly as pages on your site. Note that some key Astro features are not supported in [HTML Components](#).

Custom 404 Error Page

For a custom 404 error page, you can create a `404.astro` or `404.md` file in `src/pages`.

This will build to a `404.html` page. Most [deploy services](#) will find and use it.

Custom 500 Error Page

For a custom 500 error page to show for pages that are [rendered on demand](#), create the file `src/pages/500.astro`. This custom page is not available for prerendered pages.

If an error occurs rendering this page, your host's default 500 error page will be shown to your visitor.

Added in: `astro@4.10.3`

During development, if you have a `500.astro`, the error thrown at runtime is logged in your terminal, as opposed to being shown in the error overlay.

error

Added in: `astro@4.11.0`

`src/pages/500.astro` is a special page that is automatically passed an `error` prop for any error thrown during rendering. This allows you to use the details of an error (e.g. from a page, from middleware, etc.) to display information to your visitor.

The `error` prop's data type can be anything, which may affect how you type or use the value in your code:

`src/pages/500.astro`

```
1 ---interface Props { error: unknown;}
2 const { error } = Astro.props;---<div>{error instanceof Error ? error.message : "Unknown
  error"}</div>
```

To avoid leaking sensitive information when displaying content from the `error` prop, consider evaluating the error first, and returning appropriate content based on the error thrown. For example, you should avoid displaying the error's stack as it contains information about how your code is structured on the server.

Page Partial

Added in: `astro@3.4.0`

Caution

Page partials are intended to be used in conjunction with a front-end library, such as [htmx](#) or [Unpoly](#). You can also use them if you are comfortable writing low-level front-end JavaScript. For this reason they are an advanced feature.

Additionally, partials should not be used if the component contains scoped styles or scripts, as these elements will be stripped from the HTML output. If you need scoped styles, it is better to use regular, non-partial pages along with a frontend library that knows how to merge the contents into the head.

Partials are page components located within `src/pages/` that are not intended to render as full pages.

Like components located outside of this folder, these files do not automatically include the `<!DOCTYPE html>` declaration, nor any `<head>` content such as scoped styles and scripts.

However, because they are located in the special `src/pages/` directory, the generated HTML is available at a URL corresponding to its file path. This allows a rendering library (e.g. [htmx](#), [Stimulus](#), [jQuery](#)) to access it on the client and load sections of HTML dynamically on a page without a browser refresh or page navigation.

Partials, when combined with a rendering library, provide an alternative to [Astro islands](#) and `` tags` for building dynamic content in Astro.

Page files that can export a value for `partial` (e.g. `.astro` and `.mdx`, but not `.md`) can be marked as partials.

`src/pages/partial.astro`

```
1  ---
2
3  export const partial = true;
4
5  ---
6
7  <li>I'm a partial!</li>
```

Using with a library

Partials are used to dynamically update a section of a page using a library such as [htmx](#).

The following example shows an `hx-post` attribute set to a partial's URL. The content from the partial page will be used to update the targeted HTML element on this page.

`src/pages/index.astro`

```
1  <html> <head> <title>My page</title> <script
  src="https://unpkg.com/htmx.org@1.9.6" integrity="sha384-
  FhXw7b6A1E/jyjlZH5iHa/tTe9EpJ1Y55RjcgPbjewMskSxZt1v9qkxLJWNJaGni"
  crossorigin="anonymous"></script> </head> <body> <section> <div id="parent-
  div">Target here</div>
2    <button hx-post="/partials/clicked/" hx-trigger="click" hx-
  target="#parent-div" hx-swap="innerHTML" > Click Me! </button>
  </section> </body></html>
```

The `.astro` partial must exist at the corresponding file path, and include an export defining the page as a partial:

`src/pages/partials/clicked.astro`

```
1 ---
2
3 export const partial = true;
4
5 ---
6
7 <div>I was clicked!</div>
```

See the [htmx documentation](#) for more details on using htmx.

Routing

Astro uses **file-based routing** to generate your build URLs based on the file layout of your project `src/pages/` directory.

Navigating between pages

Astro uses standard HTML [`<a>` elements] (<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/a>) to navigate between routes. There is no framework-specific component provided.

`src/pages/index.astro`

```
1 <p>Read more <a href="/about/">about</a> Astro!</p>
2 <!-- with `base: "/docs"` configured --><p>Learn more in our <a
  href="/docs/reference/">reference</a> section!</p>
```

Static routes

`.astro` [page components](#) as well as Markdown and MDX Files (`.md`, `.mdx`) within the `src/pages/` directory **automatically become pages on your website**. Each page's route corresponds to its path and filename within the `src/pages/` directory.

```
1 # Example: Static routes
2
3 src/pages/index.astro      -> mysite.com/
4
5 src/pages/about.astro     -> mysite.com/about
6
7 src/pages/about/index.astro -> mysite.com/about
8
9 src/pages/about/me.astro   -> mysite.com/about/me
10
11 src/pages/posts/1.md       -> mysite.com/posts/1
```

Tip

There is no separate “routing config” to maintain in an Astro project! When you add a file to the `src/pages/` directory, a new route is automatically created for you. In static builds, you can customize the file output format using the `build.format` configuration option.

Dynamic routes

An Astro page file can specify dynamic route parameters in its filename to generate multiple, matching pages. For example, `src/pages/authors/[author].astro` generates a bio page for every author on your blog. `author` becomes a *parameter* that you can access from inside the page.

In Astro’s default static output mode, these pages are generated at build time, and so you must predetermine the list of `author`s that get a corresponding file. In SSR mode, a page will be generated on request for any route that matches.

Static (SSG) Mode

Because all routes must be determined at build time, a dynamic route must export a `getStaticPaths()` that returns an array of objects with a `params` property. Each of these objects will generate a corresponding route.

`[dog].astro` defines the dynamic `dog` parameter in its filename, so the objects returned by `getStaticPaths()` must include `dog` in their `params`. The page can then access this parameter using `Astro.params`.

`src/pages/dogs/[dog].astro`

```
1  ---export function getStaticPaths() { return [    { params: { dog: "clifford" } },    {  
    params: { dog: "rover" } },    { params: { dog: "spot" } },  ];}  
2  const { dog } = Astro.params;---<div>Good dog, {dog}!</div>
```

This will generate three pages: `/dogs/clifford`, `/dogs/rover`, and `/dogs/spot`, each displaying the corresponding dog name.

The filename can include multiple parameters, which must all be included in the `params` objects in `getStaticPaths()`:

`src/pages/[lang]-[version]/info.astro`

```
1  ---export function getStaticPaths() { return [    { params: { lang: "en", version: "v1"  
    } },    { params: { lang: "fr", version: "v2" } },  ];}  
2  const { lang, version } = Astro.params;---
```

This will generate `/en-v1/info` and `/fr-v2/info`.

Parameters can be included in separate parts of the path. For example, the file `src/pages/[lang]/[version]/info.astro` with the same `getStaticPaths()` above will generate the routes `/en/v1/info` and `/fr/v2/info`.

Decoding params

`params` returned by a `getStaticPaths()` function are not decoded. Use `decodeURI()` when you need to decode parameter values.

`src/pages/[slug].astro`

```
1  ---
2
3  export function getStaticPaths() {
4
5      return [
6
7          { params: { slug: decodeURI("%5Bpage%5D") }}, // decodes to "[page]"
8
9      ]
10
11  }
12
13  ---
```

Learn more about [getStaticPaths\(\)](#).



Related recipe: [Add i18n features](#)

Rest parameters

If you need more flexibility in your URL routing, you can use a [rest parameter](#) (`[...path]`) in your `.astro` filename to match file paths of any depth:

`src/pages/sequences/[...path].astro`

```
1  ---export function getStaticPaths() { return [    { params: { path: "one/two/three" }},
    { params: { path: "four" }},    { params: { path: undefined } }  ]
2  const { path } = Astro.params;---
```

This will generate `/sequences/one/two/three`, `/sequences/four`, and `/sequences`. (Setting the rest parameter to `undefined` allows it to match the top level page.)

Rest parameters can be used with **other named parameters**. For example, GitHub's file viewer can be represented with the following dynamic route:

```
1  /[org]/[repo]/tree/[branch]/[...file]
```

In this example, a request for `/withastro/astro/tree/main/docs/public/favicon.svg` would be split into the following named parameters:

```

1  {
2
3    org: "withastro",
4
5    repo: "astro",
6
7    branch: "main",
8
9    file: "docs/public/favicon.svg"
10
11 }

```

Example: Dynamic pages at multiple levels

In the following example, a rest parameter (`[...slug]`) and the [props](#) feature of `getStaticPaths()` generate pages for slugs of different depths.

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

```

1  ---export function getStaticPaths() { const pages = [ { slug: undefined,
  title: "Astro Store",      text: "Welcome to the Astro store!", }, { slug:
  "products",      title: "Astro products",      text: "We have lots of products for you",
  }, { slug: "products/astro-handbook",      title: "The ultimate Astro
  handbook",      text: "If you want to learn Astro, you must read this book.", }, ];
2  return pages.map(({ slug, title, text }) => { return { params: { slug },
  props: { title, text }, }; });
3  const { title, text } = Astro.props;---<html> <head> <title>{title}</title> </head>
  <body> <h1>{title}</h1> <p>{text}</p> </body></html>

```

On-demand dynamic routes

For [on-demand rendering](#) with an adapter, dynamic routes are defined the same way: include `[param]` or `[...path]` brackets in your file names to match arbitrary strings or paths. But because the routes are no longer built ahead of time, the page will be served to any matching route. Since these are not “static” routes, `getStaticPaths` should not be used.

For on-demand rendered routes, only one rest parameter using the spread notation may be used in the file name (e.g. `src/pages/[locale]/[...slug].astro` or `src/pages/[...locale]/[slug].astro`, but not `src/pages/[...locale]/[...slug].astro`).

`src/pages/resources/[resource]/[id].astro`

```

1  ---
2
3  export const prerender = false; // Not needed in 'server' mode
4
5  const { resource, id } = Astro.params;
6
7  ---
8
9  <h1>{resource}: {id}</h1>

```

This page will be served for any value of `resource` and `id`: `resources/users/1`, `resources/colors/blue`, etc.

Modifying the `[...slug]` example for SSR

Because SSR pages can't use `getStaticPaths()`, they can't receive props. The [previous example](#) can be adapted for SSR mode by looking up the value of the `slug` param in an object. If the route is at the root ("/"), the `slug` param will be `undefined`. If the value doesn't exist in the object, we redirect to a 404 page.

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

```
1  ---const pages = [ { slug: undefined, title: 'Astro Store', text: 'welcome to
  the Astro store!', }, { slug: 'products', title: 'Astro products', text: 'We
  have lots of products for you', }, { slug: 'products/astro-handbook', title:
  'The ultimate Astro handbook', text: 'If you want to learn Astro, you must read this
  book.', }];
2  const { slug } = Astro.params;const page = pages.find((page) => page.slug === slug);if
  (!page) return Astro.redirect("/404");const { title, text } = page;---<html> <head>
  <title>{title}</title> </head> <body> <h1>{title}</h1> <p>{text}</p> </body>
  </html>
```

Redirects

Sometimes you will need to redirect your readers to a new page, either permanently because your site structure has changed or in response to an action such as logging in to an authenticated route.

You can define rules to [redirect users to permanently-moved pages](#) in your Astro config. Or, [redirect users dynamically](#) as they use your site.

Configured Redirects

Added in: `astro@2.9.0`

You can specify a mapping of permanent redirects in your Astro config with the `redirects` value.

For internal redirects, this is a mapping of an old route path to the new route. As of Astro v5.2.0, it is also possible to redirect to external URLs that start with `http` or `https` and [can be parsed](#):

`astro.config.mjs`

```
1  import { defineConfig } from "astro/config";
2  export default defineConfig({ redirects: { "/old-page": "/new-page", "/blog":
  "https://example.com/blog" }});
```

These redirects follow [the same priority rules as file-based routes](#) and will always take lower precedence than an existing page file of the same name in your project. For example, `/old-page` will not redirect to `/new-page` if your project contains the file `src/pages/old-page.astro`.

Dynamic routes are allowed as long as both the new and old routes contain the same parameters, for example:

```

1 {
2
3   "/blog/[...slug]": "/articles/[...slug]"
4
5 }

```

Using SSR or a static adapter, you can also provide an object as the value, allowing you to specify the `status` code in addition to the new `destination`:

astro.config.mjs

```

1 import { defineConfig } from "astro/config";
2 export default defineConfig({ redirects: {
  "/old-page": { status: 302,
  destination: "/new-page" },
  "/news": { status: 302, destination:
  "https://example.com/news" } } });

```

When running `astro build`, Astro will output HTML files with the [meta refresh](#) tag by default. Supported adapters will instead write out the host's configuration file with the redirects.

The status code is `301` by default. If building to HTML files the status code is not used by the server.

Dynamic redirects

On the `Astro` global, the `Astro.redirect` method allows you to redirect to another page dynamically. You might do this after checking if the user is logged in by getting their session from a cookie.

src/pages/account.astro

```

1 ---import { isLoggedIn } from "../utils";
2 const cookie = Astro.request.headers.get("cookie");
3 // If the user is not logged in, redirect them to the login page if (!isLoggedIn(cookie))
  { return Astro.redirect("/login"); } ---

```

Because Astro uses [HTML streaming](#) in on-demand rendering, redirects must be done at the page level, not inside child components.

Rewrites

Added in: `astro@4.13.0`

A rewrite allows you to serve a different route without redirecting the browser to a different page. The browser will show the original address in the URL bar, but will instead display the content of the URL provided to `Astro.rewrite()`.

Tip

For content that has permanently moved, or to direct your user to a different page with a new URL (e.g. a user dashboard after logging in), use a [redirect](#) instead.

Rewrites can be useful for showing the same content at multiple paths (e.g. `/products/shoes/men/` and `/products/men/shoes/`) without needing to maintain two different source files.

Rewrites are also useful for SEO purposes and user experience. They allow you to display content that otherwise would require redirecting your visitor to a different page or would return a 404 status. One common use of rewrites is to show the same localized content for different variants of a language.

The following example uses a rewrite to render the `/es/` version of a page when the `/es-cu/` (Cuban Spanish) URL path is visited. When a visitor navigates to the URL `/es-cu/articles/introduction`, Astro will render the content generated by the file `src/pages/es/articles/introduction.astro`.

`src/pages/es-cu/articles/introduction.astro`

```
1 ---
2
3 return Astro.rewrite("/es/articles/introduction");
4
5 ---
```

Use `context.rewrite()` in your endpoint files to reroute to a different page:

`src/pages/api.js`

```
1 export function GET(context) {
2
3   if (!context.locals.allowed) {
4
5     return context.rewrite("/");
6
7   }
8
9 }
```

If the URL passed to `Astro.rewrite()` emits a runtime error, Astro will show the overlay error in development and return a 500 status code in production. If the URL does not exist in your project, a 404 status code will be returned.

You can intentionally create a rewrite to render your `/404` page, for example to indicate that a product in your e-commerce shop is no longer available:

`src/pages/[item].astro`

```
1 ---const { item } = Astro.params;
2 if (!itemExists(item)) { return Astro.rewrite("/404");}---
```

You can also conditionally rewrite based on an HTTP response status, for example to display a certain page on your site when visiting a URL that doesn't exist:

`src/middleware.mjs`

```

1 export const onRequest = async (context, next) => {
2
3   const response = await next();
4
5   if (response.status === 404) {
6
7     return context.rewrite("/");
8
9   }
10
11   return response;
12
13 }

```

Before displaying the content from the specified rewrite path, the function `Astro.rewrite()` will trigger a new, complete rendering phase. This re-executes any middleware for the new route/request.

See the [Astro.rewrite\(\) API reference](#) for more information.

Route Priority Order

It's possible for multiple defined routes to attempt to build the same URL path. For example, all of these routes could build `/posts/create`:

`Directorysrc/pages/[...slug].astro`
`Directoryposts/create.astro`
`[page].astro`
`[pid].ts`
`[...slug].astro`

Astro needs to know which route should be used to build the page. To do so, it sorts them according to the following rules in order:

- Astro [reserved routes](#)
- Routes with more path segments will take precedence over less specific routes. In the example above, all routes under `/posts/` take precedence over `/[...slug].astro` at the root.
- Static routes without path parameters will take precedence over dynamic routes. E.g. `/posts/create.astro` takes precedence over all the other routes in the example.
- Dynamic routes using named parameters take precedence over rest parameters. E.g. `/posts/[page].astro` takes precedence over `/posts/[...slug].astro`.
- Pre-rendered dynamic routes take precedence over server dynamic routes.
- Endpoints take precedence over pages.
- File-based routes take precedence over redirects.
- If none of the rules above decide the order, routes are sorted alphabetically based on the default locale of your Node installation.

Given the example above, here are a few examples of how the rules will match a requested URL to the route used to build the HTML:

- `pages/posts/create.astro` - Will build only `/posts/create`
- `pages/posts/[pid].ts` - Will build `/posts/abc`, `/posts/xyz`, etc. But not `/posts/create`
- `pages/posts/[page].astro` - Will build `/posts/1`, `/posts/2`, etc. But not `/posts/create`, `/posts/abc` nor `/posts/xyz`

- `pages/posts/[...slug].astro` - Will build `/posts/1/2`, `/posts/a/b/c`, etc. But not `/posts/create`, `/posts/1`, `/posts/abc`, etc.
- `pages/[...slug].astro` - Will build `/abc`, `/xyz`, `/abc/xyz`, etc. But not `/posts/create`, `/posts/1`, `/posts/abc`, etc.

Reserved routes

Internal routes take priority over any user-defined or integration-defined routes as they are required for Astro features to work. The following are Astro's reserved routes:

- `_astro/`: Serves all of the static assets to the client, including CSS documents, bundled client scripts, optimized images, and any Vite assets.
- `_server_islands/`: Serves the dynamic components deferred into a [server island](#).
- `_actions/`: Serves any defined [actions](#).

Pagination

Astro supports built-in pagination for large collections of data that need to be split into multiple pages. Astro will generate common pagination properties, including previous/next page URLs, total number of pages, and more.

Paginated route names should use the same `[bracket]` syntax as a standard dynamic route. For instance, the file name `/astronauts/[page].astro` will generate routes for `/astronauts/1`, `/astronauts/2`, etc, where `[page]` is the generated page number.

You can use the `paginate()` function to generate these pages for an array of values like so:

`src/pages/astronauts/[page].astro`

```
1  ---export function getStaticPaths({ paginate }) { const astronautPages = [    {
  astronaut: "Neil Armstrong" },    { astronaut: "Buzz Aldrin" },    { astronaut: "Sally
  Ride" },    { astronaut: "John Glenn" },  ];
2  // Generate pages from our array of astronauts, with 2 to a page return
  paginate(astronautPages, { pageSize: 2 });} // All paginated data is passed on the "page"
  propconst { page } = Astro.props;---<!-- Display the current page number.
  `Astro.params.page` can also be used! --><h1>Page {page.currentPage}</h1><ul> <!-- List
  the array of astronaut info --> {page.data.map(({ astronaut }) => <li>{astronaut}
  </li>)}</ul>
```

This generates the following pages, with 2 items to a page:

- `/astronauts/1` - Page 1: Displays “Neil Armstrong” and “Buzz Aldrin”
- `/astronauts/2` - Page 2: Displays “Sally Ride” and “John Glenn”

The `page` prop

When you use the `paginate()` function, each page will be passed its data via a `page` prop. The `page` prop has many useful properties that you can use to build pages and links between them:

```
1  interface Page<T = any> {
```

```
2
3  /** array containing the page's slice of data that you passed to the paginate()
function */
4
5  data: T[];
6
7  /** metadata */
8
9  /** the count of the first item on the page, starting from 0 */
10
11  start: number;
12
13  /** the count of the last item on the page, starting from 0 */
14
15  end: number;
16
17  /** total number of results */
18
19  total: number;
20
21  /** the current page number, starting from 1 */
22
23  currentPage: number;
24
25  /** number of items per page (default: 10) */
26
27  size: number;
28
29  /** number of last page */
30
31  lastPage: number;
32
33  url: {
34
35      /** url of the current page */
36
37      current: string;
38
39      /** url of the previous page (if there is one) */
40
41      prev: string | undefined;
42
43      /** url of the next page (if there is one) */
44
45      next: string | undefined;
46
47      /** url of the first page (if the current page is not the first page) */
48
49      first: string | undefined;
50
51      /** url of the last page (if the current page is not the last page) */
52
53      last: string | undefined;
```

```

54
55     };
56
57 }

```

The following example displays current information for the page along with links to navigate between pages:

`src/pages/astronauts/[page].astro`

```

1  ---
2
3  // Paginate same list of `{ astronaut }` objects as the previous example
4
5  export function getStaticPaths({ paginate }) { /* ... */ }
6
7  const { page } = Astro.props;
8
9  ---
10
11 <h1>Page {page.currentPage}</h1>
12
13 <ul>
14
15   {page.data.map(({ astronaut }) => <li>{astronaut}</li>)}
16
17 </ul>
18
19 {page.url.first ? <a href={page.url.first}>First</a> : null}
20
21 {page.url.prev ? <a href={page.url.prev}>Previous</a> : null}
22
23 {page.url.next ? <a href={page.url.next}>Next</a> : null}
24
25 {page.url.last ? <a href={page.url.last}>Last</a> : null}

```

Learn more about [the pagination `page` prop](#).

Nested Pagination

A more advanced use-case for pagination is **nested pagination**. This is when pagination is combined with other dynamic route params. You can use nested pagination to group your paginated collection by some property or tag.

For example, if you want to group your paginated Markdown posts by some tag, you would use nested pagination by creating a `/src/pages/[tag]/[page].astro` page that would match the following URLs:

- `/red/1` (tag=red)
- `/red/2` (tag=red)
- `/blue/1` (tag=blue)
- `/green/1` (tag=green)

Nested pagination works by returning an array of `paginate()` results from `getStaticPaths()`, one for each grouping.

In the following example, we will implement nested pagination to build the URLs listed above:

`src/pages/[tag]/[page].astro`

```
1 ---export function getStaticPaths({ paginate }) { const allTags = ["red", "blue",  
  "green"]; const allPosts = Object.values(import.meta.glob("../pages/post/*.md", {  
  eager: true })); // For every tag, return a `paginate()` result. // Make sure that you  
  pass `{ params: { tag }}` to `paginate()` // so that Astro knows which tag grouping the  
  result is for. return allTags.flatMap((tag) => { const filteredPosts =  
  allPosts.filter((post) => post.frontmatter.tag === tag); return  
  paginate(filteredPosts, { params: { tag }, pageSize: 10 }); });  
2 const { page } = Astro.props;const params = Astro.params;
```

Excluding pages

You can exclude pages or directories within `src/pages` from being built by prefixing their names with an underscore (`_`). Files with the `_` prefix won't be recognized by the router and won't be placed into the `dist/` directory.

You can use this to temporarily disable pages, and also to put tests, utilities, and components in the same folder as their related pages.

In this example, only `src/pages/index.astro` and `src/pages/projects/project1.md` will be built as page routes and HTML files.

Directorysrc/pages/Directory_hidden-directory/page1.mdpage2.md_hidden-
page.astroindex.astroDirectoryprojects/_SomeComponent.astro_utils.jsproject1.md

Advanced routing

Added in: `astro@7.0.0` New

By default, Astro handles every request with a built-in pipeline that runs the handlers in a fixed order: trailing-slash normalization, redirects, sessions, actions, user middleware, page rendering, i18n, and caching. This pipeline is designed to cover the most common use cases for routing and request handling, but it may not fit every project's needs.

Astro's advanced routing allows you to replace this pipeline with your own. You can choose which built-in features to use and where to use them. You can also add your own custom logic anywhere in the pipeline. This gives you full control over how Astro handles incoming requests.

Creating a custom entrypoint

When the default pipeline does not fit your needs, you can override it by creating a `src/fetch.ts` file that default-exports an object with a `fetch()` method. This method receives a standard [Request](#) and must return a [Response](#).

`src/fetch.ts`

```
1 import type { Fetchable } from 'astro';
2 export default { async fetch(request) { // Your custom request handling logic here
  return new Response("Hello from advanced routing!"); } } satisfies Fetchable;
```

Astro supports several file formats for its advanced routing endpoint: `.ts`, `.js`, `.mjs`, and `.mts`. We recommend using `.js` in most cases or `.ts` if you need TypeScript support.

Changing the endpoint filename

By default, Astro looks for `src/fetch.ts` as the advanced routing endpoint. You can change this by setting the `fetchFile` option in your Astro config file.

The following example tells Astro to look for `src/handler.ts` instead of `src/fetch.ts`:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';
2 export default defineConfig({ fetchFile: 'handler',});
```

Set `fetchFile` to `null` to disable the endpoint entirely. This is useful if you already have a `src/fetch.ts` file used for other purposes:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';
2 export default defineConfig({ fetchFile: null,});
```

Adding custom logic

The main benefit of advanced routing is the ability to insert custom logic into the request pipeline. You can run code before Astro touches the request, between pipeline stages, or after the response is produced.

You can do this in two ways:

- Use the [astro\(\) handler](#) to run the full built-in pipeline, and add logic before or after it.
- Compose individual handlers from [astro/fetch](#) or [astro/hono](#) for more control over execution order.

Running the full pipeline with `astro()`

Use [astro\(\)](#) when you want to keep Astro's built-in routing behavior, but need custom logic around it. This approach preserves the default pipeline order and lets you add pre-processing and post-processing in one place. For many use cases, such as adding auth guards, request logging, and custom headers, `astro()` is all you need.

The following example checks if a user can access a dashboard before running the Astro pipeline, and adds a custom header to the response once Astro has finished running:

src/fetch.ts

```

1 import { FetchState, astro } from 'astro/fetch';
2 export default { async fetch(request: Request): Promise<Response> {    const state =
  new FetchState(request);
3     // Custom pre-processing, runs before any Astro handler    const url = new
  URL(request.url);    if (url.pathname.startsWith('/dashboard')) {    const cookie =
  request.headers.get('cookie') ?? '';    if (!cookie.includes('session=')) {
  return new Response(null, {    status: 302,    headers: { Location: '/login'
  },    });    }    }
4     const response = await astro(state);
5     // Custom post-processing, runs after Astro renders    response.headers.set('X-
  Powered-By', 'Astro');    return response; },,};

```

Composing individual handlers

When you need more control over the pipeline execution order, or want to omit certain features, you can compose individual handler functions from [astro/fetch](#). Each handler operates on a [FetchState object](#) that tracks per-request data, such as the matched route, cookies, and session. You can call handlers in any order and insert custom logic between stages.

The following example runs only the handlers used in the project and adds custom logic after actions and before page rendering:

src/fetch.ts

```

1 import { FetchState, actions, middleware, pages, i18n, } from 'astro/fetch';
2 export default { async fetch(request: Request): Promise<Response> {    const state =
  new FetchState(request);
3     const actionResponse = await actions(state);    if (actionResponse) return
  actionResponse;
4     // Custom logic between actions and page rendering    console.log(`Rendering ${new
  URL(request.url).pathname}`);
5     const response = await middleware(state, (s) => pages(s));    return i18n(state,
  response); },,};

```

Using with Hono

Astro also provides Hono-compatible wrappers for all handler functions via [astro/hono](#). If you prefer to use [Hono](#) as your routing framework, you can export a Hono app from `src/fetch.ts`:

src/fetch.ts

```

1 import { Hono } from 'hono';import { logger } from 'hono/logger';import { actions,
  middleware, pages, i18n } from 'astro/hono';
2 const app = new Hono();
3 // Hono middlewareapp.use(logger());
4 // Astro handlers (as Hono
  middleware)app.use(actions());app.use(middleware());app.use(pages());app.use(i18n());
5 export default app;

```

Endpoints

Astro lets you create custom endpoints to serve any kind of data. You can use this to generate images, expose an RSS document, or use them as API Routes to build a full API for your site.

In statically-generated sites, your custom endpoints are called at build time to produce static files. If you opt in to [SSR](#) mode, custom endpoints turn into live server endpoints that are called on request. Static and SSR endpoints are defined similarly, but SSR endpoints support additional features.

Static File Endpoints

To create a custom endpoint, add a `.js` or `.ts` file to the `/pages` directory. The `.js` or `.ts` extension will be removed during the build process, so the name of the file should include the extension of the data you want to create. For example, `src/pages/data.json.ts` will build a `/data.json` endpoint.

Endpoints export a `GET` function (optionally `async`) that receives a [context object](#) with properties similar to the `Astro` global. Here, it returns a [Response](#) object with a `name` and `url`, and Astro will call this at build time and use the contents of the body to generate the file.

`src/pages/builtwith.json.ts`

```
1 // Outputs: /builtwith.json
2
3 export function GET({ params, request }) {
4
5     return new Response(
6
7         JSON.stringify({
8
9             name: "Astro",
10
11             url: "https://astro.build/",
12
13         }),
14
15     );
16
17 }
```

Since Astro v3.0, the returned `Response` object doesn't have to include the `encoding` property anymore. For example, to produce a binary `.png` image:

`src/pages/astro-logo.png.ts`

```
1 export async function GET({ params, request }) { const response = await fetch(
  "https://docs.astro.build/assets/full-logo-light.png", );
2   return new Response(await response.arrayBuffer());}
```

You can also get type safety in your endpoint functions using the `APIRoute` type with the `satisfies` operator:

```

1 import type { APIRoute } from "astro";
2 export const GET = (async ({ params, request }) => { /* ... */ }) satisfies APIRoute;

```

Note that endpoints whose URLs include a file extension (e.g. `src/pages/sitemap.xml.ts`) can only be accessed without a trailing slash (e.g. `/sitemap.xml`), regardless of your [build.trailingSlash](#) configuration.

params and Dynamic routing

Endpoints support the same [dynamic routing](#) features that pages do. Name your file with a bracketed parameter name and export a [getStaticPaths\(\)](#) function. Then, you can access the parameter using the `params` property passed to the endpoint function:

`src/pages/api/[id].json.ts`

```

1 import type { APIRoute } from "astro";
2 const usernames = ["Sarah", "Chris", "Yan", "Elian"];
3 export const GET = (({ params, request }) => { const id = params.id;
4   return new Response(    JSON.stringify({      name: usernames[id],    }),    );})
  satisfies APIRoute;
5 export function getStaticPaths() { return [    { params: { id: "0" } },    { params: {
  id: "1" } },    { params: { id: "2" } },    { params: { id: "3" } },    ];}

```

This will generate four JSON endpoints at build time: `/api/0.json`, `/api/1.json`, `/api/2.json` and `/api/3.json`. Dynamic routing with endpoints works the same as it does with pages. In static mode, you can [pass props to the endpoint using getStaticPaths\(\)](#). However, with on-demand rendering, since the endpoint is a function and not a component, passing props is not supported.

request

All endpoints receive a `request` property, but in static mode, you only have access to `request.url`. This returns the full URL of the current endpoint and works the same as [Astro.request.url](#) does for pages.

`src/pages/request-path.json.ts`

```

1 import type { APIRoute } from "astro";
2 export const GET = (({ params, request }) => { return new Response(    JSON.stringify({
  path: new URL(request.url).pathname,    }),    );}) satisfies APIRoute;

```

Server Endpoints (API Routes)

Everything described in the static file endpoints section can also be used in SSR mode: files can export a `GET` function which receives a [context object](#) with properties similar to the `Astro` global.

But, unlike in `static` mode, when you enable on-demand rendering for a route, the endpoint will be built when it is requested. This unlocks new features that are unavailable at build time, and allows you to build API routes that listen for requests and securely execute code on the server at runtime.

Your routes will be rendered on demand by default in `server` mode. In `static` mode, you must opt out of prerendering for each custom endpoint with `export const prerender = false`.



Related recipe: [Call endpoints from the server](#)

Note

Be sure to [enable an on-demand rendering mode](#) before trying these examples, and opt out of prerendering in `static` mode.

Server endpoints can access `params` without exporting `getStaticPaths`, and they can return a `Response` object, allowing you to set status codes and headers:

`src/pages/[id].json.js`

```
1 import { getProduct } from "../db";
2 export async function GET({ params }) { const id = params.id; const product = await
  getProduct(id);
3   if (!product) { return new Response(null, { status: 404, statusText: "Not
    found", }); }
4   return new Response(JSON.stringify(product), { status: 200, headers: {
    "Content-Type": "application/json", }, });}
```

This will respond to any request that matches the dynamic route. For example, if we navigate to `/helmet.json`, `params.id` will be set to `helmet`. If `helmet` exists in the mock product database, the endpoint will use a `Response` object to respond with JSON and return a successful [HTTP status code](#). If not, it will use a `Response` object to respond with a `404`.

In SSR mode, certain providers require the `Content-Type` header to return an image. In this case, use a `Response` object to specify a `headers` property. For example, to produce a binary `.png` image:

`src/pages/astro-logo.png.ts`

```
1 export async function GET({ params, request }) { const response = await fetch(
  "https://docs.astro.build/assets/full-logo-light.png", ); const buffer =
  Buffer.from(await response.arrayBuffer());
2   return new Response(buffer, { headers: { "Content-Type": "image/png" }, });}
```

HTTP methods

In addition to the `GET` function, you can export a function with the name of any [HTTP method](#). When a request comes in, Astro will check the method and call the corresponding function.

You can also export an `ALL` function to match any method that doesn't have a corresponding exported function. If there is a request with no matching method, it will redirect to your site's [404 page](#).

`src/pages/methods.json.ts`

```

1 export const GET = ({ params, request }) => { return new Response(    JSON.stringify({
    message: "This was a GET!",    }),    );}) satisfies APIRoute;
2 export const POST = ({ request }) => { return new Response(    JSON.stringify({
    message: "This was a POST!",    }),    );}) satisfies APIRoute;
3 export const DELETE = ({ request }) => { return new Response(    JSON.stringify({
    message: "This was a DELETE!",    }),    );}) satisfies APIRoute;
4 export const ALL = ({ request }) => { return new Response(    JSON.stringify({
    message: `This was a ${request.method}!`,    }),    );}) satisfies APIRoute;

```

If you define a `GET` function but no `HEAD` function, Astro will automatically handle `HEAD` requests by calling the `GET` function and stripping the body from the response.



Related recipes

- [Verify a Captcha](#)
- [Build forms with API routes](#)

request

In SSR mode, the `request` property returns a fully usable [Request](#) object that refers to the current request. This allows you to accept data and check headers:

`src/pages/test-post.json.ts`

```

1 export const POST = (async ({ request }) => { if (request.headers.get("Content-Type")
  === "application/json") {    const body = await request.json();    const name =
  body.name;
2    return new Response(    JSON.stringify({    message: "Your name was: " + name,
    },    {    status: 200,    },    ); }
3    return new Response(null, { status: 400 });}) satisfies APIRoute;

```

Redirects

The endpoint context exports a `redirect()` utility similar to `Astro.redirect`:

`src/pages/links/[id].js`

```

1 import { getLinkUrl } from "../db";
2 export async function GET({ params, redirect }) { const { id } = params; const link =
  await getLinkUrl(id);
3   if (!link) {    return new Response(null, {    status: 404,    statusText: "Not
  found",    }); }
4   return redirect(link, 307);}

```

Middleware

Middleware allows you to intercept requests and responses and inject behaviors dynamically every time a page or endpoint is about to be rendered. This rendering occurs at build time for all prerendered pages, but occurs when the route is requested for pages rendered on demand, making [additional SSR features like cookies and headers](#) available.

Middleware also allows you to set and share request-specific information across endpoints and pages by mutating a `locals` object that is available in all Astro components and API endpoints. This object is available even when this middleware runs at build time.

Basic Usage

1. Create `src/middleware.js|ts` (Alternatively, you can create `src/middleware/index.js|ts`.)
2. Inside this file, export an `onRequest()` function that can be passed a `context object` and `next()` function. This must not be a default export.

`src/middleware.js`

```
1 export function onRequest (context, next) {    // intercept data from a request
  // optionally, modify the properties in `locals`    context.locals.title = "New
  title";    context.locals.property = "information";
2    // return a Response or the result of calling `next()`    return next();};
```

3. Inside any `.astro` file, access response data using `Astro.locals`.

`src/components/Component.astro`

```
1 ---
2
3 const data = Astro.locals;
4
5 ---
6
7 <h1>{data.title}</h1>
8
9 <p>This {data.property} is from middleware.</p>
```

The `context` object

The `context` object includes information to be made available to other middleware, API routes and `.astro` routes during the rendering process.

This is an optional argument passed to `onRequest()` that may contain the `locals` object as well as any additional properties to be shared during rendering. For example, the `context` object may include cookies used in authentication.

Storing data in `context.locals`

`context.locals` is an object that can be manipulated inside the middleware.

This `locals` object is forwarded across the request handling process and is available as a property to `APIContext` and `AstroGlobal`. This allows data to be shared between middlewares, API routes, and `.astro` pages. This is useful for storing request-specific data, such as user data, across the rendering step.

Integration properties

[Integrations](#) may set properties and provide functionality through the `locals` object. If you are using an integration, check its documentation to ensure you are not overriding any of its properties or doing unnecessary work.

You can store any type of data inside `locals`: strings, numbers, and even complex data types such as functions and maps.

`src/middleware.js`

```
1 export function onRequest (context, next) { // intercept data from a request //
  optionally, modify the properties in `locals` context.locals.user = { id: 1, name:
  "John Wick" }; context.locals.welcomeTitle = () => { return "welcome back " +
  context.locals.user.name; }; context.locals.orders = new Map([["1", { product:
  "socks" }]]); context.locals.property = "information";
2 // return a Response or the result of calling `next()` return next();}
```

Then you can use this information inside any `.astro` file with `Astro.locals`.

`src/pages/orders.astro`

```
1 ---
2
3 const title = Astro.locals.welcomeTitle();
4
5 const orders = Array.from(Astro.locals.orders.entries());
6
7 const data = Astro.locals;
8
9 ---
10
11 <h1>{title}</h1>
12
13 <p>This {data.property} is from middleware.</p>
14
15 <ul>
16
17     {orders.map(order => {
18
19         return <li>{/* do something with each order */}</li>;
20
21     })}
22
23 </ul>
```

`locals` is an object that lives and dies within a single Astro route; when your route page is rendered, `locals` won't exist anymore and a new one will be created. Information that needs to persist across multiple page requests must be stored elsewhere.

Note

The value of `locals` cannot be overridden at run time. Doing so would risk wiping out all the information stored by the user. Astro performs checks and will throw an error if `locals` are overridden.

Example: redacting sensitive information

The example below uses middleware to replace “PRIVATE INFO” with the word “REDACTED” to allow you to render modified HTML on your page:

src/middleware.js

```
1 export const onRequest = async (context, next) => {    const response = await next();
  const html = await response.text();    const redactedHtml = html.replaceAll("PRIVATE
  INFO", "REDACTED");
2    return new Response(redactedHtml, {                status: 200,                headers:
  response.headers    });};
```

Middleware types

You can import and use the utility function [defineMiddleware\(\)](#) to take advantage of type safety:

src/middleware.ts

```
1 import { defineMiddleware } from "astro:middleware";
2 // `context` and `next` are automatically typedexport const onRequest =
  defineMiddleware((context, next) => {
3 });
```

Instead, if you're using JsDoc to take advantage of type safety, you can use `MiddlewareHandler`:

src/middleware.js

```
1 /** * @type {import("astro").MiddlewareHandler} */// `context` and `next` are
  automatically typedexport const onRequest = (context, next) => {
2 };
```

To type the information inside `Astro.locals`, which gives you autocomplete inside `.astro` files and middleware code, [extend the global types](#) by declaring a global namespace in the `env.d.ts` file:

src/env.d.ts

```
1 type User = { id: number; name: string;};
2 declare namespace App { interface Locals {    user: User;    welcomeTitle: () =>
  string;    orders: Map<string, object>;    session:
  import("../lib/server/session").Session | null;  }}
```

Then, inside the middleware file, you can take advantage of autocomplete and type safety.

Chaining middleware

Multiple middlewares can be joined in a specified order using `sequence()`:

src/middleware.js

```
1 import { sequence } from "astro:middleware";
2 async function validation(_, next) { console.log("validation request"); const
  response = await next(); console.log("validation response"); return response;}
3 async function auth(_, next) { console.log("auth request"); const response = await
  next(); console.log("auth response"); return response;}
4 async function greeting(_, next) { console.log("greeting request"); const response
  = await next(); console.log("greeting response"); return response;}
5 export const onRequest = sequence(validation, auth, greeting);
```

This will result in the following console order:

Terminal window

```
1 validation request
2
3 auth request
4
5 greeting request
6
7 greeting response
8
9 auth response
10
11 validation response
```

Rewriting

Added in: astro@4.13.0

The `APIContext` exposes a method called `rewrite()` which works the same way as [Astro.rewrite](#).

Use `context.rewrite()` inside middleware to display a different page's content without [redirecting](#) your visitor to a new page. This will trigger a new rendering phase, causing any middleware to be re-executed.

src/middleware.js

```
1 import { isLoggedIn } from "~/auth.js"export function onRequest (context, next) { if
  (!isLoggedIn(context)) { // If the user is not logged in, update the Request to
    render the `/login` route and // add header to indicate where the user should be sent
    after a successful login. // Re-execute middleware. return context.rewrite(new
    Request("/login", { headers: { "x-redirect-to": context.url.pathname }
    })); }
2 return next();};
```

You can also pass the `next()` function an optional URL path parameter to rewrite the current `Request` without retriggering a new rendering phase. The location of the rewrite path can be provided as a string, URL, or `Request`:

src/middleware.js

```
1 import { isLoggedIn } from "~/auth.js" export function onRequest (context, next) { if
  (!isLoggedIn(context)) { // If the user is not logged in, update the Request to
    render the `/login` route and // add header to indicate where the user should be sent
    after a successful login. // Return a new `context` to any following middlewares.
    return next(new Request("/login", { headers: { "x-redirect-to":
      context.url.pathname } })); }
2 return next();};
```

The `next()` function accepts the same payload of [the `Astro.rewrite\(\)` function](#). The location of the rewrite path can be provided as a string, URL, or `Request`.

When you have multiple middleware functions chained via [sequence\(\)](#), submitting a path to `next()` will rewrite the `Request` in place and the middleware will not execute again. The next middleware function in the chain will receive the new `Request` with its updated `context`.

Calling `next()` with this signature will create a new `Request` object using the old `ctx.request`. This means that trying to consume `Request.body`, either before or after this rewrite, will throw a runtime error. This error is often raised with [Astro Actions that use HTML forms](#). In these cases, we recommend handling rewrites from your Astro templates using `Astro.rewrite()` instead of using middleware.

src/middleware.js

```
1 // Current URL is https://example.com/blog
2 // First middleware function async function first(context, next) {
  console.log(context.url.pathname) // this will log "/blog" // Rewrite to a new route,
  the homepage // Return updated `context` which is passed to next function return
  next("/")
3 // Current URL is still https://example.com/blog
4 // Second middleware function async function second(context, next) { // Receives updated
  `context` console.log(context.url.pathname) // this will log "/" return next()
5 export const onRequest = sequence(first, second);
```

Error pages

Middleware will attempt to run for all on-demand rendered pages, even when a matching route cannot be found. This includes Astro's default (blank) 404 page and any custom 404 pages. However, it is up to the [adapter](#) to decide whether that code runs. Some adapters may serve a platform-specific error page instead.

Middleware will also attempt to run before serving a 500 error page, including a custom 500 page, unless the server error occurred in the execution of the middleware itself. If your middleware does not run successfully, then you will not have access to `Astro.locals` to render your 500 page.

Internationalization (i18n) Routing

Astro's internationalization (i18n) features allow you to adapt your project for an international audience. This routing API helps you generate, use, and verify the URLs that your multi-language site produces.

Astro's i18n routing allows you to bring your multilingual content with support for configuring a default language, computing relative page URLs, and accepting preferred languages provided by your visitor's browser. You can also specify fallback languages on a per-language basis so that your visitors can always be directed to existing content on your site.

Routing Logic

Astro uses a [middleware](#) to implement its routing logic. This middleware function is placed in the [first position](#) where it awaits every `Response` coming from any additional middleware and each page route before finally executing its own logic.

This means that operations (e.g. redirects) from your own middleware and your page logic are run first, your routes are rendered, and then the i18n middleware performs its own actions such as verifying that a localized URL corresponds to a valid route.

You can also choose to [add your own i18n logic in addition to or instead of Astro's i18n middleware](#), giving you even more control over your routes while still having access to the `astro:i18n` helper functions.

Configure i18n routing

In your `i18n` configuration, specify the list of all supported languages ([locales](#)) and set one of them as the default language ([defaultLocale](#)). You can also configure more specific routing and fallback behavior to match your desired URLs.

astro.config.mjs

```
1 import { defineConfig } from "astro/config"
2
3 export default defineConfig({
4
5   i18n: {
6
7     locales: ["es", "en", "pt-br"],
8
9     defaultLocale: "en",
10
11   }
12
13 })
```

Create localized folders

Organize your content folders with localized content by language. Create individual `/[locale]/` folders anywhere within `src/pages/` and Astro's [file-based routing](#) will create your pages at corresponding URL paths.

Your folder names must match the items in `locales` exactly. Include a localized folder for your `defaultLocale` only if you configure `prefixDefaultLocale: true` to show a localized URL path for your default language (e.g. `/en/about/`).

DirectorysrcDirectorypagesabout.astroindex.astroDirectoryesabout.astroindex.astroDirectorypt-brabout.astroindex.astro

Note

The localized folders do not need to be at the root of the `/pages/` folder.

Create links

With i18n routing configured, you can now compute links to pages within your site using the helper functions such as `getRelativeLocaleUrl()` available from the [astro:i18n module](#). These generated links will always provide the correct, localized route and can help you correctly use, or check, URLs on your site.

You can also still write the links manually.

src/pages/es/index.astro

```
1  ---import { getRelativeLocaleUrl } from 'astro:i18n';
2  // defaultLocale is "es"const aboutURL = getRelativeLocaleUrl("es", "about");---
3  <a href="/get-started/">¡Vamos!</a><a href={getRelativeLocaleUrl('es', 'blog')}>Blog</a>
   <a href={aboutURL}>Acerca</a>
```

routing

Astro's built-in file-based routing automatically creates URL routes for you based on your file structure within `src/pages/`.

When you configure i18n routing, information about this file structure (and the corresponding URL paths generated) is available to the i18n helper functions so they can generate, use, and verify the routes in your project. Many of these options can be used together for even more customization and per-language flexibility.

You can even choose to [implement your own routing logic manually](#) for even greater control.

prefixDefaultLocale

Added in: `astro@3.5.0`

This routing option defines whether or not your default language's URLs should use a language prefix (e.g. `/en/about/`).

All non-default supported languages **will** use a localized prefix (e.g. `/fr/` or `/french/`) and content files must be located in appropriate folders. This configuration option allows you to specify whether your default language should also follow a localized URL structure.

This setting also determines where the page files for your default language must exist (e.g. `src/pages/about/` or `src/pages/en/about`) as the file structure and URL structure must match for all languages.

- `"prefixDefaultLocale: false"` (default): URLs in your default language will **not** have a `/[locale]/` prefix. All other locales will.

- `"prefixDefaultLocale: true"`: All URLs, including your default language, will have a `/[locale]/` prefix.

`prefixDefaultLocale: false`

astro.config.mjs

```
1 import { defineConfig } from "astro/config"
2
3 export default defineConfig({
4
5   i18n: {
6
7     locales: ["es", "en", "fr"],
8
9     defaultLocale: "en",
10
11    routing: {
12
13      prefixDefaultLocale: false
14
15    }
16  }
17 }
18
19 })
```

This is the **default** value. Set this option when URLs in your default language will **not** have a `/[locale]/` prefix and files in your default language exist at the root of `src/pages/`:

DirectorysrcDirectorypagesabout.astroindex.astroDirectoryesabout.astroindex.astroDirectoryfrabout.astroindex.astro

- `src/pages/about.astro` will produce the route `example.com/about/`
- `src/pages/fr/about.astro` will produce the route `example.com/fr/about/`

`prefixDefaultLocale: true`

astro.config.mjs

```
1 import { defineConfig } from "astro/config"
2
3 export default defineConfig({
4
5   i18n: {
6
7     locales: ["es", "en", "fr"],
8
9     defaultLocale: "en",
10
11    routing: {
12
```

```

13     prefixDefaultLocale: true
14
15   }
16
17 }
18
19 })

```

Set this option when all routes will have their `/locale/` prefix in their URL and when all page content files, including those for your `defaultLocale`, exist in a localized folder:

DirectorysrcDirectorypagesindex.astro// Note: this file is always requiredDirectoryenindex.astroabout.astroDirectoryesabout.astroindex.astroDirectorypt-brabout.astroindex.astro

- URLs without a locale prefix, (e.g. `example.com/about/`) will return a 404 (not found) status code unless you specify a [fallback strategy](#).

Opting out of redirects for the home URL

Even with your default locale routes prefixed, this behaviour does not apply by default to your site's index page. This allows you to have a home page that exists outside of your configured locale structure, where all of your localized routes are prefixed except the home URL of your site.

You can opt out of this behavior so that your main site URL will also redirect to a prefixed, localized route for your default locale. When `prefixDefaultLocale: true` is set, you can additionally configure `redirectToDefaultLocale: true`. This will ensure that the home URL (`/`) generated by `src/pages/index.astro` will redirect to `/[defaultLocale]/`.

manual

Added in: astro@4.6.0

When this option is enabled, Astro will **disable** its i18n middleware so that you can implement your own custom logic. No other `routing` options (e.g. `prefixDefaultLocale`) may be configured with `routing: "manual"`.

You will be responsible for writing your own routing logic, or [executing Astro's i18n middleware manually](#) alongside your own.

astro.config.mjs

```

1  import { defineConfig } from "astro/config"
2
3  export default defineConfig({
4
5    i18n: {
6
7      locales: ["es", "en", "fr"],
8
9      defaultLocale: "en",
10
11     routing: "manual"

```

```

12
13   }
14
15 })

```

Astro provides helper functions for your middleware so you can control your own default routing, exceptions, fallback behavior, error catching, etc: [redirectToDefaultLocale\(\)](#), [notFound\(\)](#), and [redirectToFallback\(\)](#):

src/middleware.js

```

1  import { defineMiddleware } from "astro:middleware";
2
3  import { redirectToDefaultLocale } from "astro:i18n"; // function available with
  `manual` routing
4
5  export const onRequest = defineMiddleware(async (ctx, next) => {
6
7    if (ctx.url.startsWith("/about")) {
8
9      return next();
10
11    } else {
12
13      return redirectToDefaultLocale(302);
14
15    }
16
17  })

```

middleware function

The [middleware\(\)](#) function manually creates Astro's i18n middleware. This allows you to extend Astro's i18n routing instead of completely replacing it.

You can run [middleware\(\)](#) with [routing options](#) in combination with your own middleware, using the [sequence\(\)](#) utility to determine the order:

src/middleware.js

```

1  import { defineMiddleware, sequence } from "astro:middleware";import { middleware } from
  "astro:i18n"; // Astro's own i18n routing config
2  export const userMiddleware = defineMiddleware(async (ctx, next) => { // this response
  might come from Astro's i18n middleware, and it might return a 404  const response =
  await next(); // the /about page is an exception and we want to render it  if
  (ctx.url.pathname.startsWith("/about")) {    return new Response("About page", {
  status: 200,    }); } else {    return response;  });
3  export const onRequest = sequence( userMiddleware, middleware({
  redirectToDefaultLocale: false,    prefixDefaultLocale: true,    fallbackType:
  "redirect",  }));

```

domains

Added in: astro@4.9.0

This routing option allows you to customize your domains on a per-language basis for `server` rendered projects using the [@astrojs/node](#) or [@astrojs/vercel](#) adapter with a `site` configured.

Add `i18n.domains` to map any of your supported `locales` to custom URLs:

astro.config.mjs

```
1 import { defineConfig } from "astro/config"
2
3 export default defineConfig({
4
5   site: "https://example.com",
6
7   output: "server", // required, with no prerendered pages
8
9   adapter: node({
10
11     mode: 'standalone',
12
13   }),
14
15   i18n: {
16
17     locales: ["es", "en", "fr", "ja"],
18
19     defaultLocale: "en",
20
21     routing: {
22
23       prefixDefaultLocale: false
24
25     },
26
27     domains: {
28
29       fr: "https://fr.example.com",
30
31       es: "https://example.es"
32
33     }
34
35   }
36
37 })
```

All non-mapped `locales` will follow your `prefixDefaultLocale` configuration.

With the above configuration:

- The file `/fr/about.astro` will create the URL `https://fr.example.com/about`.
- The file `/es/about.astro` will create the URL `https://example.es/about`.
- The file `/ja/about.astro` will create the URL `https://example.com/ja/about`.
- The file `/about.astro` will create the URL `https://example.com/about`.

The above URLs will also be returned by the `getAbsoluteLocaleUrl()` and `getAbsoluteLocaleUrlList()` functions.

Fallback

When a page in one language doesn't exist (e.g. a page that is not yet translated), instead of displaying a 404 page, you can choose to display fallback content from another `locale` on a per-language basis. This is useful when you do not yet have a page for every route, but you want to still provide some content to your visitors.

Your fallback strategy consists of two parts: choosing which languages should fallback to which other languages ([i18n.fallback](#)) and choosing whether to perform a [redirect](#) or a [rewrite](#) to show the fallback content ([i18n.routing.fallbackType](#) added in Astro v4.15.0).

For example, when you configure `i18n.fallback: { fr: "es" }`, Astro will ensure that a page is built in `src/pages/fr/` for every page that exists in `src/pages/es/`.

If any page does not already exist, then a page will be created depending on your `fallbackType`:

- With a redirect to the corresponding `es` route (default behavior).
- With the content of the `/es/` page (`i18n.routing.fallbackType: "rewrite"`).

For example, the configuration below sets `es` as the fallback locale for any missing `fr` routes. This means that a user visiting `example.com/fr/my-page/` will be shown the content for `example.com/es/my-page/` (without being redirected) instead of being taken to a 404 page when `src/pages/fr/my-page.astro` does not exist.

`astro.config.mjs`

```
1 import { defineConfig } from "astro/config"
2
3 export default defineConfig({
4
5   i18n: {
6
7     locales: ["es", "en", "fr"],
8
9     defaultLocale: "en",
10
11    fallback: {
12
13      fr: "es"
14
15    },
16
17    routing: {
18
19      fallbackType: "rewrite"
20
21    }
22  }
23 })
```

```

21     }
22
23     }
24
25 })

```

Custom locale paths

In addition to defining your site's supported `locales` as strings (e.g. "en", "pt-br"), Astro also allows you to map an arbitrary number of [browser-recognized language codes](#) to a custom URL `path`. While locales can be strings of any format as long as they correspond to your project folder structure, `codes` must follow the browser's accepted syntax.

Pass an object to the `locales` array with a `path` key to define a custom URL prefix, and `codes` to indicate the languages mapped to this URL. In this case, your `/[locale]/` folder name must match exactly the value of the `path` and your URLs will be generated using the `path` value.

This is useful if you support multiple variations of a language (e.g. "fr", "fr-BR", and "fr-CA") and you want to have all these variations mapped under the same URL `/fr/`, or even customize it entirely (e.g.

`/french/`):

astro.config.mjs

```

1  import { defineConfig } from "astro/config"
2
3  export default defineConfig({
4
5    i18n: {
6
7      locales: ["es", "en", "fr"],
8
9      locales: ["es", "en", {
10
11        path: "french", // no slashes included
12
13        codes: ["fr", "fr-BR", "fr-CA"]
14
15      }],
16
17      defaultLocale: "en",
18
19      routing: {
20
21        prefixDefaultLocale: true
22
23      }
24    }
25  })
26
27

```

When using functions from the [astro:i18n virtual module](#) to compute valid URL paths based on your configuration (e.g. `getRelativeLocaleUrl()`), [use the `path` as the value for `locale`](#).

Limitations

This feature has some restrictions:

- The `site` option is mandatory.
- The `output` option must be set to `"server"`.
- There cannot be any individual prerendered pages.

Astro relies on the following headers in order to support the feature:

- [X-Forwarded-Host](#) and [Host](#). Astro will use the former, and if not present, will try the latter.
- [X-Forwarded-Proto](#) and [URL#protocol](#) of the server request.

Make sure that your server proxy/hosting platform is able to provide this information. Failing to retrieve these headers will result in a 404 (status code) page.

Browser language detection

Astro's i18n routing allows you to access two properties for browser language detection in pages rendered on demand: `Astro.preferredLocale` and `Astro.preferredLocaleList`. All pages, including static prerendered pages, have access to `Astro.currentLocale`.

These combine the browser's `Accept-Language` header, and your `locales` (strings or `codes`) to automatically respect your visitor's preferred languages.

- [Astro.preferredLocale](#): Astro can compute a **preferred locale** for your visitor if their browser's preferred locale is included in your `locales` array. This value is undefined if no such match exists.
- [Astro.preferredLocaleList](#): An array of all locales that are both requested by the browser and supported by your website. This produces a list of all compatible languages between your site and your visitor. The value is `[]` if none of the browser's requested languages are found in your `locales` array. If the browser does not specify any preferred languages, then this value will be [i18n.locales](#).
- [Astro.currentLocale](#): The locale computed from the current URL, using the syntax specified in your `locales` configuration. If the URL does not contain a `/[locale]/` prefix, then the value will default to [i18n.defaultLocale](#).

In order to successfully match your visitors' preferences, provide your `codes` using the same pattern [used by the browser](#).

Prefetch

Page load times play a big role in the usability and overall enjoyment of a site. Astro's **opt-in prefetching** brings the benefits of near-instant page navigations to your multi-page application (MPA) as your visitors interact with the site.

Enable prefetching

You can enable prefetching with the `prefetch` config:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';
2 export default defineConfig({ prefetch: true});
```

A prefetch script will be added to all pages of your site. You can then add the `data-astro-prefetch` attribute to any `<a />` links on your site to opt-in to prefetching. When you hover over the link, the script will fetch the page in the background.

```
1 <a href="/about" data-astro-prefetch>
```

Note that prefetching only works for links within your site, and not external links.

Prefetch configuration

The `prefetch` config also accepts an option object to further customize prefetching.

Prefetch strategies

Astro supports 4 prefetch strategies for various use cases:

- `hover` (default): Prefetch when you hover over or focus on the link.
- `tap`: Prefetch just before you click on the link.
- `viewport`: Prefetch as the links enter the viewport.
- `load`: Prefetch all links on the page after the page is loaded.

You can specify a strategy for an individual link by passing it to the `data-astro-prefetch` attribute:

```
1 <a href="/about" data-astro-prefetch="tap">About</a>
```

Each strategy is fine-tuned to only prefetch when needed and save your users' bandwidth. For example:

- If a visitor is using [data saver mode](#) or has a [slow connection](#), prefetch will fallback to the `tap` strategy.
- Quickly hovering or scrolling over links will not prefetch them.

Default prefetch strategy

The default prefetch strategy when adding the `data-astro-prefetch` attribute is `hover`. To change it, you can configure `prefetch.defaultStrategy` in your `astro.config.mjs` file:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';
2 export default defineConfig({ prefetch: { defaultStrategy: 'viewport' }});
```

Prefetch all links by default

If you want to prefetch all links, including those without the `data-astro-prefetch` attribute, you can set `prefetch.prefetchAll` to `true`:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';
2 export default defineConfig({ prefetch: { prefetchAll: true } });
```

You can then opt-out of prefetching for individual links by setting `data-astro-prefetch="false"`:

```
1 <a href="/about" data-astro-prefetch="false">About</a>
```

The default prefetch strategy for all links can be changed with `prefetch.defaultStrategy` as shown in the [Default prefetch strategy section](#).

Prefetch programmatically

As some navigation might not always appear as `<a />` links, you can also prefetch programmatically with the `prefetch()` API from the `astro:prefetch` module:

```
1 <button id="btn">Click me</button>
2 <script> import { prefetch } from 'astro:prefetch';
3   const btn = document.getElementById('btn'); btn.addEventListener('click', () => {
4     prefetch('/about'); });</script>
```

The `prefetch()` API includes the same [data saver mode](#) and [slow connection](#) detection so that it only prefetches when needed.

To ignore slow connection detection, you can use the `ignoreSlowConnection` option:

```
1 // Prefetch even on data saver mode or slow connection
2
3 prefetch('/about', { ignoreSlowConnection: true });
```

eagerness

Type: `'immediate' | 'eager' | 'moderate' | 'conservative'`

Default: `'immediate'`

Added in: `astro@5.6.0`

With the experimental [clientPrerender](#) flag enabled, you can use the `eagerness` option on `prefetch()` to suggest to the browser how eagerly it should prefetch/prerender link targets.

This follows the same API described in the [Speculation Rules API](#) and defaults to `immediate` (the most eager option). In decreasing order of eagerness, the other options are `eager`, `moderate`, and `conservative`.

The `eagerness` option allows you to balance the benefit of reduced wait times against bandwidth, memory, and CPU costs for your site visitors. Some browsers, such as Chrome, have [limits in place to guard against over-speculating](#) (prerendering/prefetching too many links).

```
1  -----<script>// Control prefetching eagerness with `experimental.clientPrerender`import
   { prefetch } from 'astro:prefetch';
2  // This page is resource-intensiveprefetch('/data-heavy-dashboard', { eagerness:
   'conservative' });
3  // This page is critical to the visitor's journeyprefetch('/getting-started'); //
   defaults to `{ eagerness: 'immediate' }`
4  // This page may not be visitedprefetch('/terms-of-service', { eagerness: 'moderate' });
   </script>
```

To use `prefetch()` programmatically with large sets of links, you can set `eagerness: 'moderate'` to take advantage of [First In, First Out \(FIFO\)](#) strategies and browser heuristics to let the browser decide when to prerender/prefetch them and in what order:

```
1  <a class="link-moderate" href="/nice-link-1">A Nice Link 1</a><a class="link-moderate"
   href="/nice-link-2">A Nice Link 2</a><a class="link-moderate" href="/nice-link-3">A Nice
   Link 3</a><a class="link-moderate" href="/nice-link-4">A Nice Link 4</a>...<a
   class="link-moderate" href="/nice-link-20">A Nice Link 20</a>
2  <script> import { prefetch } from 'astro:prefetch';
3  const linkModerate = document.getElementsByClassName('link-moderate');
   linkModerate.forEach((link) => prefetch(link.getAttribute('href'), {eagerness:
   'moderate'}));
4  </script>
```

Make sure to only import `prefetch()` in client-side scripts as it relies on browser APIs.

Using with View Transitions

When you use [Astro's `<View Transitions>`] (<https://docs.astro.build/en/guides/view-transitions/#enabling-view-transitions-spa-mode>) on a page, prefetching will also be enabled by default. It sets a default configuration of `{ prefetchAll: true }` which enables [prefetching for all links](#) on the page.

You can customize the prefetch configuration in `astro.config.mjs` to override the default. For example:

`astro.config.mjs`

```
1  import { defineConfig } from 'astro/config';
2  export default defineConfig({ // Disable prefetch completely prefetch: false});
```

`astro.config.mjs`

```
1  import { defineConfig } from 'astro/config';
2  export default defineConfig({ // Keep prefetch, but only prefetch for links with `data-
   astro-prefetch` prefetch: { prefetchAll: false } });
```

Browser support

Astro's prefetching uses `<link rel="prefetch">` if supported by the browser, and falls back to the `fetch()` API otherwise.

The most common browsers support Astro's prefetching with subtle differences:

Chrome

Chrome supports `<link rel="prefetch">`. Prefetching works as intended.

It also fully supports `<script type="speculationrules">` from the [Speculation Rules API](#), which can be used to further describe [prefetching strategies and rules](#), enhancing user experience for your Chrome users. You'll need to enable `clientPrerender` experiment to utilize this functionality with `prefetch()`

Firefox

Firefox supports `<link rel="prefetch">` but may display errors or fail entirely:

- Without an explicit cache header (e.g. [Cache-Control](#) or [Expires](#)), prefetching will error with `NS_BINDING_ABORTED`.
- Even in the event of an error, if the response has a proper `ETag` header, it will be re-used on navigation.
- Otherwise, if it errors with no other cache headers, the prefetch will not work.

Safari

Safari does not support `<link rel="prefetch">` and will fall back to the `fetch()` API which requires cache headers (e.g. [Cache-Control](#), [Expires](#), and [ETag](#)) to be set. Otherwise, the prefetch will not work.

Edge case: `ETag` headers do not work in private windows.

Recommendations

To best support all browsers, make sure your pages have the proper cache headers.

For static or prerendered pages, the `ETag` header is often automatically set by the deployment platform and is expected to work out of the box.

For dynamic and server-side rendered pages, set the appropriate cache headers yourself based on the page content. Visit the [MDN documentation on HTTP caching](#) for more information.

Migrating from `@astrojs/prefetch`

The `@astrojs/prefetch` integration was deprecated in v3.5.0 and is no longer maintained. Use the following instructions to migrate to Astro's built-in prefetching which replaces this integration.

1. Remove the `@astrojs/prefetch` integration and enable the `prefetch` config in `astro.config.mjs`:

```
1 import { defineConfig } from 'astro/config';import prefetch from
  '@astrojs/prefetch';
2 export default defineConfig({ integrations: [prefetch()], prefetch: true});
```

2. Convert from `@astrojs/prefetch`'s configuration options:

- The deprecated integration used the `selector` config option to specify which links should be prefetched upon entering the viewport.

Add `data-astro-prefetch="viewport"` to these individual links instead.

```
1 | <a href="/about" data-astro-prefetch="viewport">
```

- The deprecated integration used the `intentSelector` config option to specify which links should be prefetched when they were hovered over or focused.

Add `data-astro-prefetch` or `data-astro-prefetch="hover"` to these individual links instead:

```
1 | <!-- You can omit the value if `defaultStrategy` is set to `hover` (default)
   | --><a href="/about" data-astro-prefetch>
2 | <!-- Otherwise, you can explicitly define the prefetch strategy --><a
   | href="/about" data-astro-prefetch="hover">
```

- The `throttles` option from `@astrojs/prefetch` is no longer needed as the new prefetch feature will automatically schedule and prefetch optimally.

View transitions

[View transitions](#) are animated transitions between different website views. They are a popular design choice for preserving visual continuity as visitors move between states or views of an application.

Astro's view transitions and client-side routing support is powered by the [View Transitions browser API](#) and also includes:

- A few [built-in animation options](#), such as `fade`, `slide`, and `none`.
- Support for both forwards and backwards navigation animations.
- The ability to fully [customize all aspects of transition animation](#), and build your own animations.
- A way to carry HTML elements from the current page to the next during navigation.
- The option to [prevent client-side navigation for non-page links](#).
- [Control over fallback behavior](#) for browsers that do not yet support the View Transition APIs.
- Automatic support for [prefers-reduced-motion](#).

Note

By default, every page will use regular, full-page, browser navigation. You must opt in to view transitions and can use them either on a per-page basis or site-wide.

Differences between browser-native view transitions and Astro's `<ClientRouter />`

[Browser-native, cross-document view transitions](#) can be used in Astro to animate the navigation between documents in a multi-page app (MPA), often providing the experience of client-side routing of single-page applications. They don't alter the core functionality of a multi-page application, nor do they affect any existing scripts or add additional JavaScript to your page load. They simply add animations.

For enhanced client-side routing and view transition features not yet fully supported by the View Transition API, Astro provides a built-in, lightweight component to enable client-side routing and turn your multi-page app into a [single-page app](#) with smooth animations on navigation.

That comes with some benefits, like shared state across pages and persistent elements, and some drawbacks, such as needing to manually reinitialize scripts or state after navigation.

Adding Astro's built-in `<ClientRouter />` component:

- [intercepts page navigation](#) and gives you considerable control over this process.
- extends and enhances some View Transition/Navigation API features.
- allows you to [configure fallback strategies](#) for when [native browser support is lacking](#).

However, as browser APIs and web standards evolve, using Astro's `<ClientRouter />` for this additional functionality [will increasingly become unnecessary](#). We recommend keeping up with the current state of browser APIs so you can [decide whether you still need Astro's client-side routing](#) for the specific features you use.

Enabling view transitions (SPA mode)

Import and add the `<ClientRouter />` component to your common `<head>` or shared layout component. Astro will create default page animations based on the similarities between the old and new page, and will also provide fallback behavior for unsupported browsers.

The example below shows adding Astro's default page navigation animations site-wide, including the default fallback control option for non-supporting browsers, by importing and adding this component to a

`<CommonHead />` Astro component:

src/components/CommonHead.astro

```
1  ---import { ClientRouter } from "astro:transitions";---<link rel="icon"
  type="image/svg+xml" href="/favicon.svg" /><meta name="generator" content=
  {Astro.generator} />
2  <!-- Primary Meta Tags --><title>{title}</title><meta name="title" content={title} />
  <meta name="description" content={description} />
3  <ClientRouter />
```

No other configuration is necessary to enable Astro's default client-side navigation!

Use [transition directives](#) or [override default client-side navigation](#) on individual elements for finer control.

Transition Directives

Astro will automatically assign corresponding elements found in both the old page and the new page a shared, unique `view-transition-name`. This pair of matching elements is inferred by both the type of element and its location in the DOM.

Use optional `transition:*` directives on page elements in your `.astro` components for finer control over the page transition behaviour during navigation.

- `transition:name`: Allows you to override Astro's default element matching for old/new content animation and [specify a transition name](#) to associate a pair of DOM elements.
- `transition:animate`: Allows you to override Astro's default animation while replacing the old element with the new one by specifying an animation type. Use Astro's [built-in animation directives](#) or [create custom transition animations](#).
- `transition:persist`: Allows you to override Astro's default replacing old elements for new ones and instead [persist components and HTML elements](#) when navigating to another page.

Naming a transition

In some cases, you may want or need to identify the corresponding view transition elements yourself. You can specify a name for a pair of elements using the `transition:name` directive.

src/pages/old-page.astro

```
1 | <aside transition:name="hero">
```

src/pages/new-page.astro

```
1 | <aside transition:name="hero">
```

Note that the provided `transition:name` value can only be used once on each page. Set this manually when Astro can't infer a proper name itself, or for more fine control over matching elements.

Maintaining State

Added in: `astro@2.10.0`

You can persist components and HTML elements (instead of replacing them) across page navigations using the `transition:persist` directive.

For example, the following `<video>` will continue to play as you navigate to another page that contains the same video element. This works for both forwards and backwards navigation.

src/components/Video.astro

```

1 <video controls muted autoplay transition:persist>
2
3   <source
4
5     src="https://ia804502.us.archive.org/33/items/GoldenGa1939_3/GoldenGa1939_3_512kb.mp4"
6
7     type="video/mp4"
8
9   />
10
11 </video>

```

You can also place the directive on an [Astro island](#) (a UI framework component with a [client: directive](#)). If that component exists on the next page, the island from the old page **with its current state** will continue to be displayed, instead of replacing it with the island from the new page.

In the example below, the component's internal state of the count will not be reset when navigating back and forth across pages that contain the `<Counter />` component with the `transition:persist` attribute.

components/Header.astro

```

1 | <Counter client:load transition:persist initialCount={5} />

```

Known limitations

Not all state can be preserved in this way. The restart of CSS animations and the reload of iframes cannot be avoided during view transitions even when using `transition:persist`.

You can also [manually identify corresponding elements](#) if the island/element is in a different component between the two pages.

src/pages/old-page.astro

```

1 <video
2
3   controls
4
5   muted
6
7   autoplay
8
9   transition:name="media-player"
10
11   transition:persist
12
13 />

```

src/pages/new-page.astro

```

1 <MyVideo
2
3   controls
4
5   muted
6
7   autoplay
8
9   transition:name="media-player"
10
11  transition:persist
12
13 />

```

As a convenient shorthand, `transition:persist` can alternatively take a transition name as a value.

`src/pages/index.astro`

```
1 <video controls muted autoplay transition:persist="media-player">
```

transition:persist-props

Added in: `astro@4.5.0`

This allows you to control whether or not an island's props should be persisted upon navigation.

By default, when you add `transition:persist` to an island, the state is retained upon navigation, but your component will re-render with new props. This is useful, for example, when a component receives page-specific props such as the current page's `title`.

You can override this behavior by setting `transition:persist-props` in addition to `transition:persist`. Adding this directive will keep an island's existing props (not re-render with new values) in addition to maintaining its existing state.

Built-in Animation Directives

Astro comes with a few built-in animations to override the default `fade` transition. Add the `transition:animate` directive to individual elements to customize the behavior of specific transitions.

- `fade` (default): An opinionated crossfade animation. The old content fades out and the new content fades in.
- `initial`: Opt out of Astro's opinionated crossfade animation and use the browser's default styling.
- `slide`: An animation where the old content slides out to the left and new content slides in from the right. On backwards navigation, the animations are the opposite.
- `none`: Disable the browser's default animations. Use on a page's `<html>` element to disable the default fade for every element on the page.

Combine directives for full control over your page animation. Set a page default on the `<html>` element, and override on any individual elements as desired.

The example below produces a slide animation for the body content while disabling the browser's default fade animation for the rest of the page:

```
1 ---import CommonHead from "../components/CommonHead.astro";---
2 <html transition:name="root" transition:animate="none"> <head> <CommonHead />
  </head> <body> <header> ... </header> <!-- override your page default on
  a single element --> <main transition:animate="slide"> ... </main> </body>
  </html>
```

Customizing Animations

You can customize all aspects of a transition with any CSS animation properties.

To customize a built-in animation, first import the animation from `astro:transitions`, and then pass in customization options.

The example below customizes the duration of the built-in `fade` animation:

```
1 ---
2
3 import { fade } from "astro:transitions";
4
5 ---
6
7 <header transition:animate={fade({ duration: "0.4s" })}>
```

You can also define your own animations for use with `transition:animate` by defining both the forwards and backwards behavior, as well as new and old pages, according to the following types:

```
1 export interface TransitionAnimation { name: string; // The name of the keyframe
  delay?: number | string; duration?: number | string; easing?: string; fillMode?:
  string; direction?: string;}
2 export interface TransitionAnimationPair { old: TransitionAnimation |
  TransitionAnimation[]; new: TransitionAnimation | TransitionAnimation[];}
3 export interface TransitionDirectionalAnimations { forwards: TransitionAnimationPair;
  backwards: TransitionAnimationPair;}
```

The following example shows all the necessary properties to define a custom `bump` animation inside a `<style is:global>` tag in your root layout file:

src/layouts/Layout.astro

```
1 ---import { ClientRouter } from "astro:transitions";---<html lang="en"> <head>
  <ClientRouter /> </head> <body> <slot /> </body></html>
2 <style is:global> @keyframes bump { 0% { opacity: 0; transform: scale(1)
  translateX(200px); } 50% { opacity: 0.5; transform: scale(1.1); }
  100% { opacity: 1; transform: scale(1) translateX(0); } }</style>
```

The animation's behavior must be defined in the frontmatter of every component using the animation:

src/pages/index.astro

```

1  ---const anim = { old: { name: "bump", duration: "0.5s", easing: "ease-in",
    direction: "reverse", }, new: { name: "bump", duration: "0.5s", easing:
    "ease-in-out", },};
2  const customTransition = { forwards: anim, backwards: anim,};---<header
    transition:animate={customTransition}> ... </header>

```

You have great flexibility when defining custom animations. To achieve your desired result, you may wish to consider unusual combinations such as using different objects for forward and backward, or providing separate keyframe animations for old and new.

Router control

The `<ClientRouter />` router handles navigation by listening to:

- Clicks on `<a>` elements.
- Backwards and forwards navigation events.

The following options allow you to further control when navigation occurs within the router:

- `data-astro-reload`: an `<a>` tag attribute to [force a full-page navigation](#)
- `data-astro-history="auto | push | replace"`: an `<a>` tag attribute to [control the browser's history](#)
- `navigate(href, options)`: a method available to any client script or client component to [trigger navigation](#)

Preventing client-side navigation

There are some cases where you cannot navigate via client-side routing since both pages involved must use the `<ClientRouter />` router to prevent a full-page reload. You may also not want client-side routing on every navigation change and would prefer a traditional page navigation on select routes instead.

You can opt out of client-side routing on a per-link basis by adding the `data-astro-reload` attribute to any `<a>` or `<form>` tag. This attribute will override any existing `<ClientRouter />` component and instead trigger a browser refresh during navigation.

The following example shows preventing client-side routing when navigating to an article from the home page only. This still allows you to have animation on shared elements, such as a hero image, when navigating to the same page from an article listing page:

src/pages/index.astro

```

1  <a href="/articles/emperor-penguins" data-astro-reload>

```

src/pages/articles.astro

```

1  <a href="/articles/emperor-penguins">

```

Links with the `data-astro-reload` attribute will be ignored by the router and a full-page navigation will occur.

Trigger navigation

You can also trigger client-side navigation via events not normally listened to by the `<ClientRouter />` router using `navigate()`. This function from the `astro:transitions/client` module can be used in scripts, and in framework components that are hydrated with a [client directive](#).

The following example shows an Astro component that navigates a visitor to another page they select from a menu:

src/components/Form.astro

```
1 <script> import { navigate } from "astro:transitions/client";
2   // Navigate to the selected option automatically.
  document.querySelector("select").onchange = (event) => {    let href =
event.target.value;    navigate(href);  };</script><select> <option
value="/play">Play</option> <option value="/blog">Blog</option> <option
value="/about">About</option> <option value="/contact">Contact</option></select>
```

src/pages/index.astro

```
1 ---
2
3 import Form from "../components/Form.astro";
4
5 import { ClientRouter } from "astro:transitions";
6
7 ---
8
9 <html>
10
11   <head>
12
13     <ClientRouter />
14
15   </head>
16
17   <body>
18
19     <Form />
20
21   </body>
22
23 </html>
```

The following example implements the same using `navigate()` in a React `<Form />` component:

src/components/Form.jsx

```

1 import { navigate } from "astro:transitions/client";
2 export default function Form() { return (
  <select onChange={e =>
    navigate(e.target.value)}>
    <option value="/play">Play</option>
    <option value="/blog">Blog</option>
    <option value="/about">About</option>
    <option value="/contact">Contact</option>
  </select>
);}

```

The `<Form />` component can then be rendered on an Astro page that uses the `<ClientRouter />` router, with a client directive:

src/pages/index.astro

```

1 ---
2
3 import Form from "../components/Form.jsx";
4
5 import { ClientRouter } from "astro:transitions";
6
7 ---
8
9 <html>
10
11   <head>
12
13     <ClientRouter />
14
15   </head>
16
17   <body>
18
19     <Form client:load />
20
21   </body>
22
23 </html>

```

For backward and forward navigation through the browser history, you can combine `navigate()` with the built-in `history.back()`, `history.forward()` and `history.go()` functions of the browser. If `navigate()` is called during the server-side render of your component, it has no effect.

See the `astro:transitions` reference for more information about the [navigate\(\) options](#).

Replace entries in the browser history

Normally, each time you navigate, a new entry is written to the browser's history. This allows navigation between pages using the browser's `back` and `forward` buttons.

The `<ClientRouter />` router allows you to overwrite history entries by adding the `data-astro-history` attribute to any individual `<a>` tag.

The `data-astro-history` attribute can be set to the same three values as the [history option of the navigate\(\) function](#):

- `"push"`: the router will use `history.pushState` to create a new entry in the browser history.
- `"replace"`: the router will use `history.replaceState` to update the URL without adding a new entry into navigation.
- `"auto"` (default): the router will attempt `history.pushState`, but if the URL is not one that can be transitioned to, the current URL will remain with no changes to the browser history.

The following example navigates to the `/main` page but does not add a new entry to the browsing history. Instead, it reuses the current entry in the history (`/confirmation`) and overwrites it.

`src/pages/confirmation.astro`

```
1 | <a href="/main" data-astro-history="replace">
```

This has the effect that if you go back from the `/main` page, the browser will not display the `/confirmation` page, but the page before it.

Transitions with forms

Added in: `astro@4.0.0`

The `<ClientRouter />` router will trigger in-page transitions from `<form>` elements, supporting both `GET` and `POST` requests.

By default, Astro submits your form data as `multipart/form-data` when your `method` is set to `POST`. If you want to match the default behavior of web browsers, use the `enctype` attribute to submit your data encoded as `application/x-www-form-urlencoded`:

`src/components/Form.astro`

```
1 | <form
2 |
3 |   action="/contact"
4 |
5 |   method="POST"
6 |
7 |   enctype="application/x-www-form-urlencoded"
8 |
9 | >
```

You can opt out of router transitions on any individual form using the `data-astro-reload` attribute:

`src/components/Form.astro`

```
1 | <form action="/contact" data-astro-reload>
```

Navigating with user input

The `navigate()` API does not perform sanitization on the URLs passed to it. If you are using user input to determine the URL to navigate to, you should validate the input before passing it to `navigate()`.

For example, a `?redirect` query parameter could be used to navigate away from your site (`?redirect=http://example.com`) or to execute arbitrary code (`?redirect=javascript:alert('Evil code')`) if the value is not sanitized before use.

One way to implement this safely is to ensure only a set of known paths can be redirected to:

src/pages/index.astro

```
1 <script> import { navigate } from 'astro:transitions/client';
2   const params = new URLSearchParams(window.location.search); const redirect =
  params.get('redirect'); const allowedPaths = ['/home', '/about', '/contact'];
3   if (allowedPaths.includes(redirect)) {    navigate(redirect); }</script>
```

The exact kind of sanitization you need will depend on your site and what you want to allow.

Consider enabling Astro's [Content Security Policy feature](#) to help protect against cross-site scripting (XSS) risks if using user input with the `navigate()` API.

Fallback control

The `<ClientRouter />` router works best in browsers that support View Transitions (i.e. Chromium browsers), but also includes default fallback support for other browsers. Even if the browser does not support the View Transitions API, Astro's client router can still provide in-browser navigation using one of the fallback options.

Depending on browser support, you may need to explicitly set the `name` or `animate` [transition directives](#) on the elements you wish to animate for a comparable experience across all browsers:

src/pages/about.astro

```
1 ---
2
3 import Layout from "../layouts/LayoutUsingClientRouter.astro";
4
5 ---
6
7 <title transition:animate="fade">About my site</title>
```

You can override Astro's default fallback support by adding a `fallback` property on the `<ClientRouter />` component and setting it to `swap` or `none`:

- `animate` (default, recommended): Astro will simulate view transitions using custom attributes before updating page content.
- `swap`: Astro will not attempt to animate the page. Instead, the old page will be immediately replaced by the new one.
- `none`: Astro will not do any animated page transitions at all. Instead, you will get full page navigation in non-supporting browsers.

```
1 ---import { ClientRouter } from "astro:transitions";---<title>My site</title>
2 <ClientRouter fallback="swap" />
```

Known limitations

The `initial` browser animation is not simulated by Astro. So any element using this animation will not currently be animated.

Client-side navigation process

When using the `<ClientRouter />` router, the following steps occur to produce Astro's client-side navigation:

1. A visitor to your site triggers navigation by any of the following actions:
 - Clicking an `<a>` tag linking internally to another page on your site.
 - Clicking the back button.
 - Clicking the forward button.
2. The router starts fetching the next page.
3. The router adds the `data-astro-transition` attribute to the HTML element with a value of `"forward"` or `"back"` as appropriate.
4. The router calls `document.startViewTransition`. This triggers the browser's own [view transition process](#). Importantly, the browser screenshots the current state of the page.
5. Inside the `startViewTransition` callback, the router performs a **swap**, which consists of the following sequence of events:
 - The contents of the `<head>` are swapped out, with some elements kept:
 - Stylesheet DOM nodes are left in if they exist on the new page, to prevent FOUC.
 - Scripts are left in if they exist on the new page.
 - Any other head elements with `transition:persist` are left in if there is a corresponding element in the new page.
 - The `<body>` is completely replaced with the new page's body.
 - Elements marked `transition:persist` are moved over to the new DOM if they exist on the new page.
 - Scroll position is restored if necessary.
 - The `astro:after-swap` event is triggered on the `document`. This is the end of the **swap** process.
6. The router waits for any new stylesheets to load before resolving the transition.
7. The router executes any new scripts added to the page.
8. The `astro:page-load` event fires. This is the end of the navigation process.

Script behavior with view transitions

When you add view transitions to an existing Astro project, some of your scripts may no longer re-run after page navigation like they did with full-page browser refreshes. Use the following information to ensure that your scripts execute as expected.

Script order

When navigating between pages with the `<ClientRouter />` component, scripts are run in sequential order to match browser behavior.

Script re-execution

[Bundled module scripts](#), which are the default scripts in Astro, are only ever executed once. After initial execution they will be ignored, even if the script exists on the new page after a transition.

Unlike bundled module scripts, [inline scripts](#) have the potential to be re-executed during a user's visit to a site if they exist on a page that is visited multiple times. Inline scripts might also re-execute when a visitor navigates to a page without the script, and then back to one with the script.

With view transitions, some scripts may no longer re-run after page navigation like they do with full-page browser refreshes. There are several [events during client-side routing that you can listen for](#), and fire events when they occur. You can wrap an existing script in an event listener to ensure it runs at the proper time in the navigation cycle.

The following example wraps a script for a mobile “hamburger” menu in an event listener for `astro:page-load` which runs at the end of page navigation to make the menu responsive to being clicked after navigating to a new page:

src/scripts/menu.js

```
1 document.addEventListener("astro:page-load", () => {
2
3   document.querySelector(".hamburger").addEventListener("click", () => {
4
5     document.querySelector(".nav-links").classList.toggle("expanded");
6
7   });
8
9 });
```

The following example shows a function that runs in response to the `astro:after-swap` event, which happens immediately after the new page has replaced the old page and before the DOM elements are painted to the screen. This avoids a flash of light mode theme after page navigation by checking and, if necessary, setting the dark mode theme before the new page is rendered:

src/components/ThemeToggle.astro

```
1 <script is:inline> function applyTheme() {   localStorage.theme === "dark"       ?
  document.documentElement.classList.add("dark")       :
  document.documentElement.classList.remove("dark"); }
2   document.addEventListener("astro:after-swap", applyTheme);  applyTheme();</script>
```

data-astro-rerun

Added in: astro@4.5.0

To force inline scripts to re-execute after every transition, add the `data-astro-rerun` property. Adding any attribute to a script also implicitly adds `is:inline`, so this is only available for scripts that are not bundled and processed by Astro.

```
1 <script is:inline data-astro-rerun>...</script>
```

To ensure that a script runs every time a page is loaded during client-side navigation, it should be executed by a [lifecycle event](#). For example, event listeners for `DOMContentLoaded` can be replaced by the [astro:page-load](#) lifecycle event.

If you have code that sets up a global state in an inline script, this state will need to take into account that the script might execute more than once. Check for the global state in your `<script>` tag, and conditionally execute your code where possible. This works because `window` is preserved.

```
1 <script is:inline>
2
3   if (!window.SomeGlobal) {
4
5       window.SomeGlobal = {};
6
7   }
8
9 </script>
```

Lifecycle events

The `<ClientRouter />` router fires a number of events on the `document` during navigation. These events provide hooks into the lifecycle of navigation, allowing you to do things like show indicators that a new page is loading, override default behavior, and restore state as navigation is completing.

The navigation process involves a **preparation** phase, when new content is loaded; a **DOM swap** phase, where the old page's content is replaced by the new page's content; and a **completion** phase where scripts are executed, loading is reported as completed and clean-up work is carried out.

Astro's View Transition API lifecycle events in order are:

- [astro:before-preparation](#)
- [astro:after-preparation](#)
- [astro:before-swap](#)
- [astro:after-swap](#)
- [astro:page-load](#)

Tip

`before-` events allow you to influence and modify actions that are about to take place, and `after-` events are notifications that a phase is complete.

While some actions can be triggered during any event, some tasks can only be performed during a specific event for best results, such as displaying a loading spinner before preparation or overriding animation pairs before swapping content.

astro:before-preparation

Added in: astro@3.6.0

An event that fires at the beginning of the preparation phase, after navigation has started (e.g. after the user has clicked a link), but before content is loaded.

This event is used:

- To do something before loading has started, such as showing a loading spinner.
- To alter loading, such as loading content you've defined in a template rather than from the external URL.
- To change the `direction` of the navigation (which is usually either `forward` or `backward`) for custom animation.

Here is an example of using the `astro:before-preparation` event to load a spinner before the content is loaded and stop it immediately after loading. Note that using the [loader callback](#) in this way allows asynchronous execution of code.

```
1 <script is:inline>
2
3   document.addEventListener("astro:before-preparation", (event) => {
4
5     const originalLoader = event.loader;
6
7     event.loader = async function () {
8
9       const { startSpinner } = await import("./spinner.js");
10
11       const stop = startSpinner();
12
13       await originalLoader();
14
15       stop();
16
17     };
18
19   });
20
21 </script>
```

astro:after-preparation

Added in: astro@3.6.0

An event that fires at the end of the preparation phase, after the new page's content has been loaded and parsed into a document. This event occurs before the view transitions phase.

This example uses the `astro:before-preparation` event to start a loading indicator and the `astro:after-preparation` event to stop it:

```
1 <script is:inline>
2
3   document.addEventListener("astro:before-preparation", () => {
4
5     document.querySelector("#loading").classList.add("show");
6
7   });
8
9   document.addEventListener("astro:after-preparation", () => {
10
11     document.querySelector("#loading").classList.remove("show");
12
13   });
14
15 </script>
```

This is a simpler version of loading a spinner than the example shown above: if all of the listener's code can be executed synchronously, there is no need to hook into the `loader` callback.

`astro:before-swap`

Added in: `astro@3.6.0`

An event that fires before the new document (which is populated during the preparation phase) replaces the current document. This event occurs inside of the view transition, where the user is still seeing a snapshot of the old page.

This event can be used to make changes before the swap occurs. The `newDocument` property on the event represents the incoming document. Here is an example of ensuring the browser's light or dark mode preference in `localStorage` is carried over to the new page:

```
1 <script>
2
3   document.addEventListener("astro:before-swap", (event) => {
4
5     event.newDocument.documentElement.dataset.theme =
6
7       localStorage.getItem("darkMode") ? "dark" : "light";
8
9   });
10
11 </script>
```

The `astro:before-swap` event can also be used to change the *implementation* of the swap. The default swap implementation diffs head content, moves **persistent** elements from the old document to the `newDocument`, and then replaces the entire `body` with the body of the new document.

At this point of the lifecycle, you could choose to define your own swap implementation, for example to diff the entire contents of the existing document (which some other routers do):

```
1 <script is:inline>
2
3   document.addEventListener("astro:before-swap", (event) => {
4
5     event.swap = () => {
6
7       diff(document, event.newDocument);
8
9     };
10
11   });
12
13 </script>
```

Building a custom swap function

Added in: `astro@4.15.0`

The [swapFunctions object](#) of the `astro:transitions/client` module provides five utility functions that handle specific swap-related tasks, including handling document attributes, page elements, and script execution. These functions can be used directly to define a custom swap implementation.

The following example demonstrates how to use these functions to recreate Astro's built-in swap implementation:

```
1 <script> import { swapFunctions } from "astro:transitions/client";
2   // substitutes `window.document` with `doc` function mySwap(doc: Document) {
3     swapFunctions.deselectScripts(doc);    swapFunctions.swapRootAttributes(doc);
    swapFunctions.swapHeadElements(doc);    const restoreFocusFunction =
    swapFunctions.saveFocus();    swapFunctions.swapBodyElement(doc.body, document.body);
    restoreFocusFunction();  }
4   document.addEventListener("astro:before-swap", (event) => {    event.swap = () =>
    mySwap(event.newDocument);  });</script>
```

Custom swap implementations can start with this template and add or replace individual steps with custom logic as needed.

astro:after-swap

An event that fires immediately after the new page replaces the old page. You can listen to this event on the `document` and trigger actions that will occur before the new page's DOM elements render and scripts run.

This event, when listened to on the **outgoing page**, is useful to pass along and restore any state on the DOM that needs to transfer over to the new page.

This is the latest point in the lifecycle where it is still safe to, for example, add a dark mode class name (`<html class="dark-mode">`), though you may wish to do so in an earlier event.

The `astro:after-swap` event occurs immediately after the browser history has been updated and the scroll position has been set. Therefore, one use of targeting this event is to override the default scroll restore for history navigation. The following example resets the horizontal and vertical scroll position to the top left corner of the page for each navigation.

```
1 document.addEventListener("astro:after-swap", () =>
2
3   window.scrollTo({ left: 0, top: 0, behavior: "instant" }),
4
5 );
```

`astro:page-load`

An event that fires at the end of page navigation, after the new page is visible to the user and blocking styles and scripts are loaded. You can listen to this event on the `document`.

The `<ClientRouter />` component fires this event both on initial page navigation for a pre-rendered page and on any subsequent navigation, either forwards or backwards.

You can use this event to run code on every page navigation, for example to set up event listeners that would otherwise be lost during navigation.

```
1 <script>
2
3   document.addEventListener("astro:page-load", () => {
4
5     // This runs on first page load and after every navigation.
6
7     setupStuff(); // e.g. add event listeners
8
9   });
10
11 </script>
```

Accessibility

Enabling client-side routing and animating page transitions both come with accessibility challenges, and Astro aims to make sites opting in to View Transitions as accessible-by-default as possible.

Route announcement

Added in: `astro@3.2.0`

The `<ClientRouter />` component includes a route announcer for page navigation during client-side routing. No configuration or action is needed to enable this.

Assistive technologies let visitors know that the page has changed by announcing the new page title after navigation. When using server-side routing with traditional full-page browser refreshes, this happens by default after the new page loads. In client-side routing, the `<ClientRouter />` component performs this action.

To add route announcement to client-side routing, the component adds an element to the new page with the `aria-live` attribute set to `assertive`. This tells AT (assistive technology) to announce immediately. The component also checks for the following, in priority order, to determine the announcement text:

- The `<title>`, if it exists.
- The first `<h1>` it finds.
- The `pathname` of the page.

We strongly recommend you always include a `<title>` in each page for accessibility.

prefers-reduced-motion

Astro's `<ClientRouter />` component includes a CSS media query that disables *all* view transition animations, including fallback animation, whenever the `prefers-reduced-motion` setting is detected. Instead, the browser will simply swap the DOM elements without an animation.

Components

Astro components are the basic building blocks of any Astro project. They are HTML-only templating components with no client-side runtime and use the `.astro` file extension.

Note

If you know HTML, you already know enough to write your first Astro component.

Learn more in the [Astro syntax reference](#).

Astro components are extremely flexible. An Astro component can be as small as a snippet of HTML, like a collection of common `<meta>` tags that make SEO easy to work with. Components can be reusable UI elements, like a header or a profile card. Astro components can even contain an entire page layout or, when located in the special `src/pages/` folder, be an entire page itself.

The most important thing to know about Astro components is that they **don't render on the client**. They render to HTML either at build-time or on-demand. You can include JavaScript code inside of your component frontmatter, and all of it will be stripped from the final page sent to your users' browsers. The result is a faster site, with zero JavaScript footprint added by default.

When your Astro component does need client-side interactivity, you can add [standard HTML `` tags](#) or [UI Framework components](#) as "client islands".

For components that need to render personalized or dynamic content, you can defer their server rendering by adding a [server directive](#). These "server islands" will render their content when it is available, without delaying the entire page load.

Component Structure

An Astro component is made up of two main parts: the **Component Script** and the **Component Template**. Each part performs a different job, but together they provide a framework that is both easy to use and expressive enough to handle whatever you might want to build.

`src/components/EmptyComponent.astro`

```
1  ---
2
3  // Component Script (JavaScript)
4
5  ---
6
7  <!-- Component Template (HTML + JS Expressions) -->
```

The Component Script

Astro uses a code fence (`---`) to identify the component script in your Astro component. If you've ever written Markdown before, you may already be familiar with a similar concept called *frontmatter*. Astro's idea of a component script was directly inspired by this concept.

You can use the component script to write any JavaScript code that you need to render your template. This can include:

- importing other Astro components
- importing other framework components, like React
- importing data, like a JSON file
- fetching content from an API or database
- creating variables that you will reference in your template

`src/components/MyComponent.astro`

```
1  ---import SomeAstroComponent from '../components/SomeAstroComponent.astro';import
   SomeReactComponent from '../components/SomeReactComponent.jsx';import someData from
   '../data/pokemon.json';
2  // Access passed-in component props, like `<X title="Hello, world" />`const { title } =
   Astro.props;
3  // Fetch external data, even from a private API or databaseconst data = await
   fetch('SOME_SECRET_API_URL/users').then(r => r.json());---<!-- Your template here! -->
```

The code fence is designed to guarantee that the JavaScript that you write in it is “fenced in.” It won't escape into your frontend application, or fall into your user's hands. You can safely write code here that is expensive or sensitive (like a call to your private database) without worrying about it ever ending up in your user's browser.

Note

The Astro component script is TypeScript, which allows you to add additional syntax to JavaScript for editor tooling, and error checking.

Read more about Astro's [built-in TypeScript support](#).

The Component Template

The component template is below the code fence and determines the HTML output of your component.

If you write plain HTML here, your component will render that HTML in any Astro page it is imported and used.

However, [Astro's component template syntax](#) also supports **JavaScript expressions**, Astro `__` and `__` tags, **imported components**, and [special Astro directives](#). Data and values defined in the component script can be used in the component template to produce dynamically-created HTML.

src/components/MyFavoritePokemon.astro

```
1  ---// Your component script here!import Banner from '../components/Banner.astro';import
  Avatar from '../components/Avatar.astro';import ReactPokemonComponent from
  '../components/ReactPokemonComponent.jsx';const myFavoritePokemon = [/* ... */];const {
  title } = Astro.props;---<!-- HTML comments supported! -->{/* JS comment syntax is also
  valid! */}
2  <Banner /><h1>Hello, world!</h1>
3  <!-- Use props and other variables from the component script: --><p>{title}</p>
4  <!-- Delay component rendering and provide fallback loading content: --><Avatar
  server:defer> <svg slot="fallback" class="generic-avatar" transition:name="avatar">...
  </svg></Avatar>
5  <!-- Include other UI framework components with a `client:` directive to hydrate: -->
  <ReactPokemonComponent client:visible />
6  <!-- Mix HTML with JavaScript expressions, similar to JSX: --><ul>
  {myFavoritePokemon.map((data) => <li>{data.name}</li>)}</ul>
7  <!-- Use a template directive to build class names from multiple strings or even
  objects! --><p class:list=[["add", "dynamic", { classNames: true }]] />
```

Component-based design

Components are designed to be **reusable** and **composable**. You can use components inside of other components to build more and more advanced UI. For example, a `Button` component could be used to create a `ButtonGroup` component:

src/components/ButtonGroup.astro

```
1  ---
2
3  import Button from './Button.astro';
4
5  ---
6
7  <div>
8
9    <Button title="Button 1" />
10
11   <Button title="Button 2" />
12
13   <Button title="Button 3" />
14
15 </div>
```

Component Props

An Astro component can define and accept props. These props then become available to the component template for rendering HTML. Props are available on the `Astro.props` global in your frontmatter script.

Here is an example of a component that receives a `greeting` prop and a `name` prop. Notice that the props to be received are destructured from the global `Astro.props` object.

src/components/GreetingHeadline.astro

```
1  ---
2
3  // Usage: <GreetingHeadline greeting="Howdy" name="Partner" />
4
5  const { greeting, name } = Astro.props;
6
7  ---
8
9  <h2>{greeting}, {name}!</h2>
```

This component, when imported and rendered in other Astro components, layouts or pages, can pass these props as attributes:

src/components/GreetingCard.astro

```
1  ---
2
3  import GreetingHeadline from './GreetingHeadline.astro';
4
5  const name = 'Astro';
6
7  ---
8
9  <h1>Greeting Card</h1>
10
11 <GreetingHeadline greeting="Hi" name={name} />
12
13 <p>I hope you have a wonderful day!</p>
```

You can also define your props with TypeScript with a `Props` type interface. Astro will automatically pick up the `Props` interface in your frontmatter and give type warnings/errors. These props can also be given default values when destructured from `Astro.props`.

src/components/GreetingHeadline.astro

```
1  ---interface Props {  name: string;  greeting?: string;}
2  const { greeting = "Hello", name } = Astro.props;---<h2>{greeting}, {name}!</h2>
```

Component props can be given default values to use when none are provided.

src/components/GreetingHeadline.astro

```

1  ---
2
3  const { greeting = "Hello", name = "Astronaut" } = Astro.props;
4
5  ---
6
7  <h2>{greeting}, {name}!</h2>

```

Slots

The `<slot />` element is a placeholder for external HTML content, allowing you to inject (or “slot”) child elements from other files into your component template.

By default, all child elements passed to a component will be rendered in its `<slot />`.

Note

Unlike *props*, which are attributes passed to an Astro component available for use throughout your component with `Astro.props`, *slots* render child HTML elements where they are written.

`src/components/Wrapper.astro`

```

1  ---import Header from './Header.astro';import Logo from './Logo.astro';import Footer
   from './Footer.astro';
2  const { title } = Astro.props;---<div id="content-wrapper">  <Header />  <Logo />  <h1>
   {title}</h1>  <slot />  <!-- children will go here -->  <Footer /></div>

```

`src/pages/fred.astro`

```

1  ---
2
3  import wrapper from '../components/wrapper.astro';
4
5  ---
6
7  <wrapper title="Fred's Page">
8
9    <h2>All about Fred</h2>
10
11    <p>Here is some stuff about Fred.</p>
12
13  </wrapper>

```

This pattern is the basis of an [Astro layout component](#): an entire page of HTML content can be “wrapped” with `<SomeLayoutComponent></SomeLayoutComponent>` tags and sent to the component to render inside of common page elements defined there.

See the [Astro.slots utility functions](#) for more ways to access and render slot content.

Named Slots

An Astro component can also have named slots. This allows you to pass only HTML elements with the corresponding slot name into a slot's location.

Slots are named using the `name` attribute:

src/components/Wrapper.astro

```
1  ---import Header from './Header.astro';import Logo from './Logo.astro';import Footer
   from './Footer.astro';
2  const { title } = Astro.props;---<div id="content-wrapper"> <Header /> <!-- children
   with the `slot="after-header"` attribute will go here --> <slot name="after-header" />
   <Logo /> <h1>{title}</h1> <!-- children without a `slot`, or with `slot="default"`
   attribute will go here --> <slot /> <Footer /> <!-- children with the `slot="after-
   footer"` attribute will go here --> <slot name="after-footer" /></div>
```

To inject HTML content into a particular slot, use the `slot` attribute on any child element to specify the name of the slot. All other child elements of the component will be injected into the default (unnamed) `<slot />`.

src/pages/fred.astro

```
1  ---
2
3  import wrapper from '../components/wrapper.astro';
4
5  ---
6
7  <wrapper title="Fred's Page">
8
9      
10
11     <h2>All about Fred</h2>
12
13     <p>Here is some stuff about Fred.</p>
14
15     <p slot="after-footer">Copyright 2022</p>
16
17 </wrapper>
```

Tip

Use a `slot="my-slot"` attribute on the child element that you want to pass through to a matching `<slot name="my-slot" />` placeholder in your component.

To pass multiple HTML elements into a component's `<slot/>` placeholder without a wrapping `<div>`, use the `slot=""` attribute on [Astro's ``component`](#):

src/components/CustomTable.astro

```

1  ---
2
3  // Create a custom table with named slot placeholders for header and body content
4
5  ---
6
7  <table class="bg-white">
8
9      <thead class="sticky top-0 bg-white"><slot name="header" /></thead>
10
11      <tbody class="[_tr:nth-child(odd)]:bg-gray-100"><slot name="body" /></tbody>
12
13  </table>

```

Inject multiple rows and columns of HTML content using a `slot=""` attribute to specify the `"header"` and `"body"` content. Individual HTML elements can also be styled:

src/components/StockTable.astro

```

1  ---import CustomTable from './CustomTable.astro';---<CustomTable> <Fragment
  slot="header"> <!-- pass table header -->    <tr><th>Product name</th><th>Stock
  units</th></tr> </Fragment>
2    <Fragment slot="body"> <!-- pass table body -->    <tr><td>Flip-flops</td><td>64</td>
  </tr>    <tr><td>Boots</td><td>32</td></tr>    <tr><td>Sneakers</td><td class="text-red-
  500">0</td></tr> </Fragment></CustomTable>

```

Note that named slots must be an immediate child of the component. You cannot pass named slots through nested elements.

Tip

Named slots can also be passed to [UI framework components](#)!

Note

It is not possible to dynamically generate an Astro slot name, such as within a [map](#) function. If this feature is needed within UI framework components, it might be best to generate these dynamic slots within the framework itself.

Fallback Content for Slots

Slots can also render **fallback content**. When there are no matching children passed to a slot, a `<slot />` element will render its own placeholder children.

src/components/Wrapper.astro

```

1  ---import Header from './Header.astro';import Logo from './Logo.astro';import Footer
  from './Footer.astro';
2  const { title } = Astro.props;---<div id="content-wrapper"> <Header /> <Logo /> <h1>
  {title}</h1> <slot>    <p>This is my fallback content, if there is no child passed into
  slot</p> </slot> <Footer /></div>

```

Fallback content will only be displayed when there are no matching elements with the `slot="name"` attribute being passed in to a named slot.

Astro will pass an empty slot when a slot element exists but has no content to pass. Fallback content cannot be used as a default when an empty slot is passed. Fallback content is only displayed when no slot element can be found.

Transferring slots

Slots can be transferred to other components. For example, when creating nested layouts:

src/layouts/BaseLayout.astro

```
1  ---
2
3  ---
4
5  <html lang="en">
6
7    <head>
8
9      <meta charset="utf-8" />
10
11     <link rel="icon" type="image/svg+xml" href="/favicon.svg" />
12
13     <meta name="viewport" content="width=device-width" />
14
15     <meta name="generator" content={Astro.generator} />
16
17     <slot name="head" />
18
19   </head>
20
21   <body>
22
23     <slot />
24
25   </body>
26
27 </html>
```

src/layouts/HomeLayout.astro

```

1  ---
2
3  import BaseLayout from './BaseLayout.astro';
4
5  ---
6
7  <BaseLayout>
8
9      <slot name="head" slot="head" />
10
11     <slot />
12
13 </BaseLayout>

```

Note

Named slots can be transferred to another component using both the `name` and `slot` attributes on a `<slot />` tag.

Now, the default and `head` slots passed to `HomeLayout` will be transferred to the `BaseLayout` parent.

`src/pages/index.astro`

```

1  ---
2
3  import HomeLayout from '../layouts/HomeLayout.astro';
4
5  ---
6
7  <HomeLayout>
8
9      <title slot="head">Astro</title>
10
11     <h1>Astro</h1>
12
13 </HomeLayout>

```

HTML Components

Astro supports importing and using `.html` files as components or placing these files within the `src/pages/` subdirectory as pages. You may want to use HTML components if you're reusing code from an existing site built without a framework, or if you want to ensure that your component has no dynamic features.

HTML components must contain only valid HTML, and therefore lack key Astro component features:

- They don't support frontmatter, server-side imports, or dynamic expressions.
- Any `<script>` tags are left unbundled, treated as if they had an [is:inline directive](#).
- They can only [reference assets that are in the `public/` folder](#).

Note

A ``element`` (<https://docs.astro.build/en/basics/astro-components/#slots>) inside an HTML component will work as it would in an Astro component. In order to use the `[HTML Web Component Slot]` (<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/slot>) element instead, add `is:inline to your `element`.

Layouts

Layouts are [Astro components](#) used to provide a reusable UI structure, such as a page template.

We conventionally use the term “layout” for Astro components that provide common UI elements shared across pages such as headers, navigation bars, and footers. A typical Astro layout component provides [Astro, Markdown or MDX pages](#) with:

- a **page shell** (`<html>`, `<head>` and `<body>` tags)
- a ``` to specify where individual page content should be injected.

But, there is nothing special about a layout component! They can [accept props](#) and [import and use other components](#) like any other Astro component. They can include [UI frameworks components](#) and [client-side scripts](#). They do not even have to provide a full page shell, and can instead be used as partial UI templates.

However, if a layout component does contain a page shell, its `<html>` element must be the parent of all other elements in the component.

Layout components are commonly placed in a `src/layouts` directory in your project for organization, but this is not a requirement; you can choose to place them anywhere in your project. You can even colocate layout components alongside your pages by [prefixing the layout names with `](#).

Sample Layout

`src/layouts/MySiteLayout.astro`

```
1  ---
2
3  import BaseHead from '../components/BaseHead.astro';
4
5  import Footer from '../components/Footer.astro';
6
7  const { title } = Astro.props;
8
9  ---
10
11 <html lang="en">
12
13   <head>
14
15     <meta charset="utf-8">
16
17     <meta name="viewport" content="width=device-width, initial-scale=1">
18
19     <BaseHead title={title}/>
20
```

```

21 </head>
22
23 <body>
24
25   <nav>
26
27     <a href="#">Home</a>
28
29     <a href="#">Posts</a>
30
31     <a href="#">Contact</a>
32
33   </nav>
34
35   <h1>{title}</h1>
36
37   <article>
38
39     <slot /> <!-- your content is injected here -->
40
41   </article>
42
43   <Footer />
44
45 </body>
46
47 <style>
48
49   h1 {
50
51     font-size: 2rem;
52
53   }
54
55 </style>
56
57 </html>

```

src/pages/index.astro

```

1  ---
2
3  import MySiteLayout from '../layouts/MySiteLayout.astro';
4
5  ---
6
7  <MySiteLayout title="Home Page">
8
9    <p>My page content, wrapped in a layout!</p>
10
11 </MySiteLayout>

```

Learn more about [slots](#).

Using TypeScript with layouts

Any Astro layout can be modified to introduce type safety & autocompletion by providing the types for your props:

src/components/MyLayout.astro

```
1  ---
2
3  interface Props {
4
5      title: string;
6
7      description: string;
8
9      publishDate: string;
10
11     viewCount: number;
12
13 }
14
15 const { title, description, publishDate, viewCount } = Astro.props;
16
17 ---
18
19 <html lang="en">
20
21     <head>
22
23         <meta charset="UTF-8">
24
25         <meta name="description" content={description}>
26
27         <title>{title}</title>
28
29     </head>
30
31     <body>
32
33         <header>
34
35             <p>Published on {publishDate}</p>
36
37             <p>Viewed by {viewCount} folks</p>
38
39         </header>
40
41         <main>
42
43             <slot />
44
```

```
45     </main>
46
47     </body>
48
49 </html>
```

Markdown Layouts

Page layouts are especially useful for individual Markdown pages which otherwise would not have any page formatting.

Astro provides a special `layout` frontmatter property intended for [individual .md files located within src/pages/ using file-based routing](#) to specify which `.astro` component to use as the page layout. This component allows you to provide `<head>` content like meta tags (e.g. `<meta charset="utf-8">`) and styles for the Markdown page. By default, this specified component can automatically access data from the Markdown file.

This is not recognized as a special property when using [content collections](#) to query and render your content.

src/pages/page.md

```
1  ---layout: ../layouts/BlogPostLayout.astrotitle: "Hello, world!"author: "Matthew
  Phillips"date: "09 Aug 2022"---All frontmatter properties are available as props to an
  Astro layout component.
2  The `layout` property is the only special one provided by Astro.
3  You can use it in Markdown files located within `src/pages/`.
```

A typical layout for a Markdown page includes:

1. The `frontmatter` prop to access the Markdown page's frontmatter and other data.
2. A default `<body>` to indicate where the page's Markdown content should be rendered.

src/layouts/BlogPostLayout.astro

```
1  ---
2
3  // 1. The frontmatter prop gives access to frontmatter and other data
4
5  const { frontmatter } = Astro.props;
6
7  ---
8
9  <html>
10
11    <head>
12
13      <!-- Add other Head elements here, like styles and meta tags. -->
14
15      <meta name="viewport" content="width=device-width, initial-scale=1">
16
17      <meta charset="utf-8">
18
```

```

19     <title>{frontmatter.title}</title>
20
21 </head>
22
23 <body>
24
25     <!-- Add other UI components here, like common headers and footers. -->
26
27     <h1>{frontmatter.title} by {frontmatter.author}</h1>
28
29     <!-- 2. Rendered HTML will be passed into the default slot. -->
30
31     <slot />
32
33     <p>Written on: {frontmatter.date}</p>
34
35 </body>
36
37 </html>

```

You can set a layout's [Props type](#) with the `MarkdownLayoutProps` helper:

`src/layouts/BlogPostLayout.astro`

```

1  ---import type { MarkdownLayoutProps } from 'astro';
2  type Props = MarkdownLayoutProps<{ // Define frontmatter props here  title: string;
   author: string;  date: string;}>;
3  // Now, `frontmatter`, `url`, and other Markdown layout properties// are accessible with
   type safetyconst { frontmatter, url } = Astro.props;---<html>  <head>    <meta
   charset="utf-8">    <link rel="canonical" href={new URL(url, Astro.site).pathname}>
   <title>{frontmatter.title}</title>  </head>  <body>    <h1>{frontmatter.title} by
   {frontmatter.author}</h1>    <slot />    <p>Written on: {frontmatter.date}</p>  </body>
   </html>

```

Markdown Layout Props

A Markdown layout will have access to the following information via `Astro.props`:

- `file` - The absolute path of this file (e.g. `/home/user/projects/.../file.md`).
- `url` - The URL of the page (e.g. `/en/guides/markdown-content`).
- `frontmatter` - All frontmatter from the Markdown or MDX document.
 - `frontmatter.file` - The same as the top-level `file` property.
 - `frontmatter.url` - The same as the top-level `url` property.
- `headings` - A list of headings (`h1` -> `h6`) in the Markdown or MDX document with associated metadata. This list follows the type: `{ depth: number; slug: string; text: string }[]`.
- `rawContent()` - A function that returns the raw Markdown document as a string.
- `compiledContent()` - An async function that returns the Markdown document compiled to an HTML string.

Note

A Markdown layout will have access to all the Markdown file's [available properties](#) from `Astro.props` with **two key differences**:

- Heading information (i.e. `h1` -> `h6` elements) is available via the `headings` array, rather than a `getHeadings()` function.
- `file` and `url` are *also* available as nested `frontmatter` properties (i.e. `frontmatter.url` and `frontmatter.file`).

Importing Layouts Manually (MDX)

You can also use the special Markdown layout property in the frontmatter of MDX files to pass `frontmatter` and `headings` props directly to a specified layout component in the same way.

To pass information to your MDX layout that does not (or cannot) exist in your frontmatter, you can instead import and use a `<Layout />` component. This works like any other Astro component, and will not receive any props automatically. Pass it any necessary props directly:

`src/pages/posts/first-post.mdx`

```
1 ---layout: ../../layouts/BaseLayout.astrotitle: 'My first MDX post'publishDate: '21
  September 2022'---import BaseLayout from '../../layouts/BaseLayout.astro';
2 export function fancyJsHelper() { return "Try doing that with YAML!";}
3 <BaseLayout title={frontmatter.title} fancyJsHelper={fancyJsHelper}> welcome to my new
  Astro blog, using MDX!</BaseLayout>
```

Then, your values are available to you through `Astro.props` in your layout, and your MDX content will be injected into the page where your `<slot />` component is written:

`src/layouts/BaseLayout.astro`

```
1 ---
2
3 const { title, fancyJsHelper } = Astro.props;
4
5 ---
6
7 <html>
8
9   <head>
10
11     <!-- -->
12
13     <meta charset="utf-8">
14
15   </head>
16
17   <body>
18
19     <!-- -->
20
```

```

21     <h1>{title}</h1>
22
23     <slot /> <!-- your content is injected here -->
24
25     <p>{fancyJsHelper()}</p>
26
27     <!-- -->
28
29 </body>
30
31 </html>

```

When using any layout (either through the frontmatter `Layout` property or by importing a layout), you must include the `<meta charset="utf-8">` tag in your layout as Astro will no longer add it automatically to your MDX page.

Learn more about Astro's Markdown and MDX support in our [Markdown guide](#).

Nesting Layouts

Layout components do not need to contain an entire page worth of HTML. You can break your layouts into smaller components, and combine layout components to create even more flexible, page templates. This pattern is useful when you want to share some code across multiple layouts.

For example, a `BlogPostLayout.astro` layout component could style a post's title, date and author. Then, a site-wide `BaseLayout.astro` could handle the rest of your page template, like navigation, footers, SEO meta tags, global styles, and fonts. You can also pass props received from your post to another layout, just like any other nested component.

`src/layouts/BlogPostLayout.astro`

```

1  ---
2
3  import BaseLayout from './BaseLayout.astro';
4
5  const { frontmatter } = Astro.props;
6
7  ---
8
9  <BaseLayout url={frontmatter.url}>
10
11     <h1>{frontmatter.title}</h1>
12
13     <h2>Post author: {frontmatter.author}</h2>
14
15     <slot />
16
17 </BaseLayout>

```

Styles and CSS

Astro was designed to make styling and writing CSS a breeze. Write your own CSS directly inside of an Astro component or import your favorite CSS library like [Tailwind](#). Advanced styling languages like [Sass](#) and [Less](#) are also supported.

Styling in Astro

Styling an Astro component is as easy as adding a `<style>` tag to your component or page template. When you place a `<style>` tag inside of an Astro component, Astro will detect the CSS and handle your styles for you, automatically.

src/components/MyComponent.astro

```
1 <style>
2
3   h1 { color: red; }
4
5 </style>
```

Scoped Styles

Astro `<style>` CSS rules are automatically **scoped by default**. Scoped styles are compiled behind-the-scenes to only apply to HTML written inside of that same component. The CSS that you write inside of an Astro component is automatically encapsulated inside of that component.

This CSS:

src/pages/index.astro

```
1 <style> h1 {   color: red;  }
2   .text {   color: blue;  }</style>
```

Compiles to this:

```
1 <style> h1[data-astro-cid-hhnqfkh6] {   color: red;  }
2   .text[data-astro-cid-hhnqfkh6] {   color: blue;  }</style>
```

Scoped styles don't leak and won't impact the rest of your site. In Astro, it is okay to use low-specificity selectors like `h1 {}` or `p {}` because they will be compiled with scopes in the final output.

Scoped styles also won't apply to other Astro components contained inside of your template. If you need to style a child component, consider wrapping that component in a `<div>` (or other element) that you can then style.

The specificity of scoped styles is preserved, allowing them to work consistently alongside other CSS files or CSS libraries while still preserving the exclusive boundaries that prevent styles from applying outside the component.

Global Styles

While we recommend scoped styles for most components, you may eventually find a valid reason to write global, unscoped CSS. You can opt-out of automatic CSS scoping with the `<style is:global>` attribute.

src/components/GlobalStyles.astro

```
1 <style is:global>
2
3   /* Unscoped, delivered as-is to the browser.
4
5     Applies to all <h1> tags on your site. */
6
7   h1 { color: red; }
8
9 </style>
```

You can also mix global & scoped CSS rules together in the same `<style>` tag using the `:global()` selector. This becomes a powerful pattern for applying CSS styles to children of your component.

src/components/MixedStyles.astro

```
1 <style>
2
3   /* Scoped to this component, only. */
4
5   h1 { color: red; }
6
7   /* Mixed: Applies to child `h1` elements only. */
8
9   article :global(h1) {
10
11     color: blue;
12
13   }
14
15 </style>
16
17 <h1>Title</h1>
18
19 <article><slot /></article>
```

This is a great way to style things like blog posts, or documents with CMS-powered content where the contents live outside of Astro. But be careful: components whose appearance differs based on whether or not they have a certain parent component can become difficult to troubleshoot.

Scoped styles should be used as often as possible. Global styles should be used only as-needed.

Combining classes with `class:list`

If you need to combine classes on an element dynamically, you can use the `class:list` utility attribute in `.astro` files.

src/components/ClassList.astro

```
1 ---const { isRed } = Astro.props;---<!-- If `isRed` is truthy, class will be "box red".  
--><!-- If `isRed` is falsy, class will be "box". --><div class:list={['box', { red:  
  isRed }}><slot /></div>  
2 <style> .box { border: 1px solid blue; } .red { border-color: red; }</style>
```

See our [directives reference](#) page to learn more about `class:list`.

CSS Variables

Added in: astro@0.21.0

The Astro `<style>` can reference any CSS variables available on the page. You can also pass CSS variables directly from your component frontmatter using the `define:vars` directive.

src/components/DefineVars.astro

```
1 ---  
2  
3 const foregroundColor = "rgb(221 243 228)";  
4  
5 const backgroundColor = "rgb(24 121 78)";  
6  
7 ---  
8  
9 <style define:vars={{ foregroundColor, backgroundColor }}>  
10  
11   h1 {  
12  
13     background-color: var(--backgroundColor);  
14  
15     color: var(--foregroundColor);  
16  
17   }  
18  
19 </style>  
20  
21 <h1>Hello</h1>
```

See our [directives reference](#) page to learn more about `define:vars`.

Passing a `class` to a child component

In Astro, HTML attributes like `class` do not automatically pass through to child components.

Instead, accept a `class` prop in the child component and apply it to the root element. When destructuring, you must rename it, because `class` is a [reserved word](#) in JavaScript.

Using the default scoped style strategy, you must also pass the `data-astro-cid-*` attribute. You can do this by passing the `...rest` of the props to the component. If you have changed `scopedStyleStrategy` to `'class'` or `'where'`, the `...rest` prop is not necessary.

`src/components/MyComponent.astro`

```
1  ---
2
3  const { class: className, ...rest } = Astro.props;
4
5  ---
6
7  <div class={className} {...rest}>
8
9      <slot/>
10
11 </div>
```

`src/pages/index.astro`

```
1  ---
2
3  import MyComponent from "../components/MyComponent.astro"
4
5  ---
6
7  <style>
8
9      .red {
10
11          color: red;
12
13      }
14
15 </style>
16
17 <MyComponent class="red">This will be red!</MyComponent>
```

Scoped styles from parent components

Because the `data-astro-cid-*` attribute includes the child in its parent's scope, it is possible for styles to cascade from parent to child. To avoid this having unintended side effects, ensure you use unique class names in the child component.

Inline styles

You can style HTML elements inline using the `style` attribute. This can be a CSS string or an object of CSS properties:

src/pages/index.astro

```
1 // These are equivalent:
2
3 <p style={{ color: "brown", textDecoration: "underline" }}>My text</p>
4
5 <p style="color: brown; text-decoration: underline;">My text</p>
```

External Styles

There are two ways to resolve external global stylesheets: an ESM import for files located within your project source, and an absolute URL link for files in your `public/` directory, or hosted outside of your project.

Read more about using [static assets](#) located in `public/` or `src/`.

Import a local stylesheet

Using an npm package?

You may need to update your `astro.config` when importing from npm packages. See the [“import stylesheets from an npm package” section](#) below.

You can import stylesheets in your Astro component frontmatter using ESM import syntax. CSS imports work like [any other ESM import in an Astro component](#), which should be referenced as **relative to the component** and must be written at the **top** of your component script, with any other imports.

src/pages/index.astro

```
1 ---
2
3 // Astro will bundle and optimize this CSS for you automatically
4
5 // This also works for preprocessor files like .scss, .styl, etc.
6
7 import '../styles/utils.css';
8
9 ---
10
11 <html><!-- Your page here --></html>
```

CSS `import` via ESM are supported inside of any JavaScript file, including JSX components like React & Preact. This can be useful for writing granular, per-component styles for your React components.

Import a stylesheet from an npm package

You may also need to load stylesheets from an external npm package. This is especially common for utilities like [Open Props](#). If your package **recommends using a file extension** (i.e. `package-name/styles.css` instead of `package-name/styles`), this should work like any local stylesheet:

src/pages/random-page.astro

```
1  ---
2
3  import 'package-name/styles.css';
4
5  ---
6
7  <html><!-- Your page here --></html>
```

If your package **does not suggest using a file extension** (i.e. `package-name/styles`), you'll need to update your Astro config first!

Say you are importing a CSS file from `package-name` called `normalize` (with the file extension omitted). To ensure we can prerender your page correctly, add `package-name` to [the `vite.ssr.noExternal` array](#):

astro.config.mjs

```
1  import { defineConfig } from 'astro/config';
2  export default defineConfig({ vite: { ssr: { noExternal: ['package-name'], }
3  }})
```

Note

This is a [Vite-specific setting](#) that does *not* relate to (or require) [Astro SSR](#).

Now, you are free to import `package-name/normalize`. This will be bundled and optimized by Astro like any other local stylesheet.

src/pages/random-page.astro

```
1  ---
2
3  import 'package-name/normalize';
4
5  ---
6
7  <html><!-- Your page here --></html>
```

Load a static stylesheet via “link” tags

You can also use the `<link>` element to load a stylesheet on the page. This should be an absolute URL path to a CSS file located in your `/public` directory, or an URL to an external website. Relative `<link>` href values are not supported.

src/pages/index.astro

```

1 <head>
2
3   <!-- Local: /public/styles/global.css -->
4
5   <link rel="stylesheet" href="/styles/global.css" />
6
7   <!-- External -->
8
9   <link rel="stylesheet"
10  href="https://cdn.jsdelivr.net/npm/prismjs@1.24.1/themes/prism-tomorrow.css" />
11 </head>

```

Because this approach uses the `public/` directory, it skips the normal CSS processing, bundling and optimizations that are provided by Astro. If you need these transformations, use the [Import a Stylesheet](#) method above.

Cascading Order

Astro components will sometimes have to evaluate multiple sources of CSS. For example, your component might import a CSS stylesheet, include its own `<style>` tag, *and* be rendered inside a layout that imports CSS.

When conflicting CSS rules apply to the same element, browsers first use *specificity* and then *order of appearance* to determine which value to show.

If one rule is more *specific* than another, no matter where the CSS rule appears, its value will take precedence:

`src/components/MyComponent.astro`

```

1 <style>
2
3   h1 { color: red }
4
5   div > h1 {
6
7     color: purple
8
9   }
10
11 </style>
12
13 <div>
14
15   <h1>
16
17     This header will be purple!
18
19   </h1>
20
21 </div>

```

If two rules have the same specificity, then the *order of appearance* is evaluated, and the last rule's value will take precedence:

src/components/MyComponent.astro

```
1 <style>
2
3   h1 { color: purple }
4
5   h1 { color: red }
6
7 </style>
8
9 <div>
10
11   <h1>
12
13     This header will be red!
14
15   </h1>
16
17 </div>
```

Astro CSS rules are evaluated in this order of appearance:

- `<link>` **tags in the head** (lowest precedence)
- **imported styles**
- **scoped styles** (highest precedence)

Scoped Styles

Depending on your chosen value for `scopedStyleStrategy`, scoped styles may or may not increase the [CLASS column specificity](#).

However, [scoped styles](#) will always come last in the order of appearance. These styles will therefore take precedence over other styles of the same specificity. For example, if you import a stylesheet that conflicts with a scoped style, the scoped style's value will apply:

src/components/make-it-purple.css

```
1 h1 {
2
3   color: purple;
4
5 }
```

src/components/MyComponent.astro

```
1 ---
2
3 import './make-it-purple.css'
```

```

4  ---
5
6
7  <style>
8
9      h1 { color: red }
10
11 </style>
12
13 <div>
14
15     <h1>
16
17         This header will be red!
18
19     </h1>
20
21 </div>

```

Scoped styles will be overwritten if the imported style is more specific. The style with a higher specificity will take precedence over the scoped style:

src/components/make-it-purple.css

```

1  #intro {
2
3      color: purple;
4
5  }

```

src/components/MyComponent.astro

```

1  ---
2
3  import "../make-it-purple.css"
4
5  ---
6
7  <style>
8
9      h1 { color: red }
10
11 </style>
12
13 <div>
14
15     <h1 id="intro">
16
17         This header will be purple!
18
19     </h1>
20

```

Import Order

When importing multiple stylesheets in an Astro component, the CSS rules are evaluated in the order that they are imported. A higher specificity will always determine which styles to show, no matter when the CSS is evaluated. But, when conflicting styles have the same specificity, the *last one imported* wins:

src/components/make-it-purple.css

```
1 | div > h1 {  
2 |  
3 |   color: purple;  
4 |  
5 | }
```

src/components/make-it-green.css

```
1 | div > h1 {  
2 |  
3 |   color: green;  
4 |  
5 | }
```

src/components/MyComponent.astro

```
1 | ---  
2 |  
3 | import './make-it-green.css'  
4 |  
5 | import './make-it-purple.css'  
6 |  
7 | ---  
8 |  
9 | <style>  
10 |  
11 |   h1 { color: red }  
12 |  
13 | </style>  
14 |  
15 | <div>  
16 |  
17 |   <h1>  
18 |  
19 |     This header will be purple!  
20 |  
21 |   </h1>  
22 |  
23 | </div>
```

While `<style>` tags are scoped and only apply to the component that declares them, *imported* CSS can “leak”. Importing a component applies any CSS it imports, even if the component is never used:

src/components/PurpleComponent.astro

```
1  ---
2
3  import "../make-it-purple.css"
4
5  ---
6
7  <div>
8
9    <h1>I import purple CSS.</h1>
10
11  </div>
```

src/components/MyComponent.astro

```
1  ---
2
3  import "../make-it-green.css"
4
5  import PurpleComponent from "../PurpleComponent.astro";
6
7  ---
8
9  <style>
10
11    h1 { color: red }
12
13  </style>
14
15  <div>
16
17    <h1>
18
19      This header will be purple!
20
21    </h1>
22
23  </div>
```

Tip

A common pattern in Astro is to import global CSS inside a [Layout component](#). Be sure to import the Layout component before other imports so that it has the lowest precedence.

Link Tags

Style sheets loaded via [link tags](#) are evaluated in order, before any other styles in an Astro file. Therefore, these styles will have lower precedence than imported stylesheets and scoped styles:

src/pages/index.astro

```
1 ---import "../components/make-it-purple.css"---
2 <html lang="en"> <head>   <meta charset="utf-8" />   <link rel="icon"
  type="image/svg+xml" href="/favicon.svg" />   <meta name="viewport"
  content="width=device-width" />   <meta name="generator" content={Astro.generator} />
  <title>Astro</title>   <link rel="stylesheet" href="/styles/make-it-blue.css" />
</head> <body>   <div>       <h1>This will be purple</h1>   </div> </body></html>
```

Tailwind

Astro comes with support for adding popular CSS libraries, tools, and frameworks to your project like [Tailwind](#) and more!

Astro supports both Tailwind 3 and 4. You can [add Tailwind 4 support through a Vite plugin](#) to your project with a CLI command, or install legacy dependencies manually to add [Tailwind 3 support through an Astro integration](#).

To [upgrade your Astro project from Tailwind 3 to 4](#) you will need to both add Tailwind 4 support, and remove legacy Tailwind 3 support.

Add Tailwind 4

In Astro `>=5.2.0`, use the `astro add tailwind` command for your package manager to install the official Vite Tailwind plugin. To add Tailwind 4 support to earlier versions of Astro, follow the [instructions in the Tailwind docs](#) to add the `@tailwindcss/vite` Vite plugin manually.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npx astro add tailwind
```

Then, import `tailwindcss` into `src/styles/global.css` (or another CSS file of your choosing) to make Tailwind classes available to your Astro project. This file including the import will be created by default if you used the `astro add tailwind` command to install the Vite plugin.

src/styles/global.css

```
1 | @import "tailwindcss";
```

Import this file in the pages where you want Tailwind to apply. This is often done in a layout component so that Tailwind styles can be used on all pages sharing that layout:

src/layouts/Layout.astro

```
1 ---
2
3 import "../styles/global.css";
4
5 ---
```

Upgrade from Tailwind 3

Follow the steps to update an existing Astro project using Tailwind v3 (using the `@astrojs/tailwind` integration) to Tailwind 4 (using [the @tailwindcss/vite plugin](#)).

1. [Add Tailwind 4 support to your project](#) through the CLI for the latest version of Astro, or by adding the Vite plugin manually.
2. Uninstall the `@astrojs/tailwind` integration from your project:
 - [npm](#)
 - [pnpm](#)
 - [Yarn](#)

Terminal window

```
1 npm uninstall @astrojs/tailwind
```

3. Remove the `@astrojs/tailwind` integration from your `astro.config.mjs`:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import tailwind from
  '@astrojs/tailwind';
2 export default defineConfig({ // ... integrations: [tailwind()], // ...});
```

4. Then, upgrade your project according to [Tailwind's v4 upgrade guide](#).

Legacy Tailwind 3 support

To add (or keep) support for Tailwind 3, you will need to have both `tailwindcss@3` and the official Astro Tailwind integration `@astrojs/tailwind` installed. Installing these dependencies manually is only used for legacy Tailwind 3 compatibility, and is not required for Tailwind 4. You will also need a [legacy Tailwind configuration](#):

1. Install Tailwind and the Astro Tailwind integration to your project dependencies using your preferred package manager:
 - [npm](#)
 - [pnpm](#)

- [Yarn](#)

Terminal window

```
1 | npm install tailwindcss@3 @astrojs/tailwind
```

2. Import the integration to your `astro.config.mjs` file, and add it to your `integrations[]` array:

`astro.config.mjs`

```
1 | import { defineConfig } from 'astro/config';import tailwind from
  | '@astrojs/tailwind';
2 | export default defineConfig({ // ... integrations: [tailwind()], // ...});
```

3. Create a `tailwind.config.mjs` file in your project's root directory. You can use the following command to generate a basic configuration file for you:

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npx tailwindcss init
```

4. Add the following basic configuration to your `tailwind.config.mjs` file:

`tailwind.config.mjs`

```
1 | /** @type {import('tailwindcss').Config} */
2 |
3 | export default {
4 |
5 |   content: ['./src/**/*.astro,html,js,jsx,md,mdx,svelte,ts,tsx,vue'],
6 |
7 |   theme: {
8 |
9 |     extend: {},
10 |
11 |   },
12 |
13 |   plugins: [],
14 |
15 | };
```



Related recipe: [Style rendered Markdown with Tailwind Typography](#)

CSS Preprocessors

Astro supports CSS preprocessors such as [Sass](#), [Stylus](#), and [Less](#) through [Vite](#).

Sass and SCSS

Terminal window

```
1 | npm install sass
```

Use `<style lang="scss">` or `<style lang="sass">` in `.astro` files.

Stylus

Terminal window

```
1 | npm install stylus
```

Use `<style lang="styl">` or `<style lang="stylus">` in `.astro` files.

Less

Terminal window

```
1 | npm install less
```

Use `<style lang="less">` in `.astro` files.

LightningCSS

Terminal window

```
1 | npm install lightningcss
```

Update your `vite` configuration in `astro.config.mjs`:

`astro.config.mjs`

```
1 | import { defineConfig } from 'astro/config'
2 | export default defineConfig({ vite: { css: { transformer: "lightningcss", }, }, })
```

In framework components

You can also use all of the above CSS preprocessors within JS frameworks as well! Be sure to follow the patterns each framework recommends:

- **React / Preact:** `import styles from './styles.module.scss';`
- **Vue:** `<style lang="scss">`

- **Svelte:** `<style lang="scss">`

PostCSS

Astro comes with PostCSS included as part of [Vite](#). To configure PostCSS for your project, create a `postcss.config.cjs` file in the project root. You can import plugins using `require()` after installing them (for example `npm install autoprefixer`).

`postcss.config.cjs`

```
1 module.exports = {
2
3   plugins: [
4
5     require('autoprefixer'),
6
7     require('cssnano'),
8
9   ],
10
11 };
```

Frameworks and Libraries

React / Preact

`.jsx` files support both global CSS and CSS Modules. To enable the latter, use the `.module.css` extension (or `.module.scss` / `.module.sass` if using Sass).

`src/components/MyReactComponent.jsx`

```
1 import './global.css'; // include global CSS
2
3 import styles from './styles.module.css'; // Use CSS Modules (must end in `.module.css`,
  `.module.scss`, or `.module.sass`!)
```

Vue

Vue in Astro supports the same methods as `vue-loader` does:

- [vue-loader - Scoped CSS](#)
- [vue-loader - CSS Modules](#)

Svelte

Svelte in Astro also works exactly as expected: [Svelte Styling Docs](#).

Markdown Styling

Any Astro styling methods are available to a [Markdown layout component](#), but different methods will have different styling effects on your page.

You can apply global styles to your Markdown content by adding [imported stylesheets](#) to the layout that wraps your page content. It is also possible to style your Markdown with ``` tags` in the layout component. Note that any styles added are subject to [Astro's cascading order](#), and you should check your rendered page carefully to ensure your styles are being applied as intended.

You can also add CSS integrations including [Tailwind](#). If you are using Tailwind, the [typography plugin](#) can be useful for styling Markdown.

Production

Bundle control

When Astro builds your site for production deployment, it minifies and combines your CSS into chunks. Each page on your site gets its own chunk, and additionally, CSS that is shared between multiple pages is further split off into their own chunks for reuse.

However, when you have several pages sharing styles, some shared chunks can become really small. If all of them were sent separately, it would lead to many stylesheets requests and affect site performance. Therefore, by default Astro will link only those in your HTML above 4kB in size as `<link rel="stylesheet">` tags, while inlining smaller ones into `<style type="text/css">`. This approach provides a balance between the number of additional requests and the volume of CSS that can be cached between pages.

You can configure the size at which stylesheets will be linked externally (in bytes) using the `assetsInlineLimit` vite build option. Note that this option affects script and image inlining as well.

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';
2 export default defineConfig({ vite: { build: { assetsInlineLimit: 1024, } } });
```

If you would rather all project styles remain external, you can configure the `inlineStylesheets` build option.

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';
2 export default defineConfig({ build: { inlineStylesheets: 'never' } });
```

You can also set this option to `'always'` which will inline all stylesheets.

Advanced

Caution

Be careful when bypassing Astro's built-in CSS bundling! Styles won't be automatically included in the built output, and it is your responsibility to make sure that the referenced file is properly included in the final page output.

?raw CSS Imports

For advanced use cases, CSS can be read directly from disk without being bundled or optimized by Astro. This can be useful when you need complete control over some snippet of CSS, and need to bypass Astro's automatic CSS handling.

This is not recommended for most users.

src/components/RawInlineStyles.astro

```
1  ---
2
3  // Advanced example! Not recommended for most users.
4
5  import rawStylesCSS from '../styles/main.css?raw';
6
7  ---
8
9  <style is:inline set:html={rawStylesCSS}></style>
```

See [Vite's docs](#) for full details.

?url CSS Imports

For advanced use cases, you can import a direct URL reference for a CSS file inside of your project `src/` directory. This can be useful when you need complete control over how a CSS file is loaded on the page. However, this will prevent the optimization of that CSS file with the rest of your page CSS .

This is not recommended for most users. Instead, place your CSS files inside of `public/` to get a consistent URL reference.

Caution

Importing a smaller CSS file with `?url` may return the base64 encoded contents of the CSS file as a data URL in your final build. Either write your code to support encoded data URLs (`data:text/css;base64,...`) or set the [vite.build.assetsInlineLimit](#) config option to `0` to disable this feature.

src/components/RawStylesUrl.astro

```
1  ---
2
3  // Advanced example! Not recommended for most users.
4
5  import stylesUrl from '../styles/main.css?url';
6
7  ---
8
9  <link rel="preload" href={stylesUrl} as="style">
10
11 <link rel="stylesheet" href={stylesUrl}>
```

See [Vite's docs](#) for full details.

Using custom fonts

This guide will show you how to add [web fonts](#) to your project and use them in your components.

Astro provides a way to use fonts from your filesystem and various font providers (e.g. Fontsource, Google) through a unified, [fully customizable](#), and type-safe API.

Web fonts can impact page performance at both load time and rendering time. This API helps you keep your site performant with automatic [web font optimizations](#) including preload links, optimized fallbacks, and opinionated defaults. [See common usage examples](#).

The Fonts API focuses on performance and privacy by downloading and caching fonts so they're served from your site. This can avoid sending user data to third-party sites, and also ensures that a consistent set of fonts is available to all your visitors.

Configuring custom fonts

Registering custom fonts for your Astro project is done through [the fonts option](#) in your Astro config.

For each font you want to use, you must specify its [name](#), a [CSS variable](#), and an Astro font provider.

Astro provides [built-in support for the most popular font providers](#): Adobe, Bunny, Fontshare, Fontsource, Google, Google Icons and NPM, as well as for using your own local font files. Additionally, you can [further customize your font configuration](#) to optimize performance and visitor experience.

Using a local font file

This example will demonstrate adding a custom font using the font file `DistantGalaxy.woff2`.

1. Add your font file inside the [src/ directory](#), for example `src/assets/fonts/`.
2. Create a new font family in your Astro config file using the [local font provider](#) and specify the variants to be included:

astro.config.mjs

```
1 import { defineConfig, fontProviders } from "astro/config";
2 export default defineConfig({ fonts: [{ provider: fontProviders.local(),
  name: "DistantGalaxy", cssVariable: "--font-distant-galaxy", options: {
  variants: [{ src: ['./src/assets/fonts/DistantGalaxy.woff2'], weight:
    'normal', style: 'normal' }] } } ]});
```

3. Your font is now configured and ready to be [added to your page head](#) so that it can be used in your project.

Using Fontsource

Astro supports [several font providers](#) out of the box, including support for [Fontsource](#) that simplifies using Google Fonts and other open-source fonts.

The following example will use Fontsource to add custom font support, but the process is similar for any of Astro's built-in font providers (e.g. [Adobe](#), [Bunny](#)).

1. Find the font you want to use in [Fontsource's catalog](#). This example will use [Roboto](#).

2. Create a new font family in your Astro config file using the [Fontsource provider](#):

astro.config.mjs

```
1 import { defineConfig, fontProviders } from "astro/config";
2 export default defineConfig({ fonts: [{ provider: fontProviders.fontsource(),
  name: "Roboto", cssVariable: "--font-roboto", }]});
```

3. Your font is now configured and ready to be [added to your page head](#) so that it can be used in your project.

Applying custom fonts

After [a font is configured](#), it must be added to your page head with an identifying CSS variable. Then, you can use this variable when defining your page styles.

1. Import and include the `[`](https://docs.astro.build/en/reference/modules/astro-assets/#font-)`]` component with the required `cssVariable` property in the head of your page, usually in a dedicated `Head.astro`` component or in a [layout](#) component directly:

src/layouts/Layout.astro

```
1 ---import { Font } from "astro:assets";---
2 <html> <head> <Font cssVariable="--font-distant-galaxy" /> </head> <body>
  <slot /> </body></html>
```

2. In any page rendered with that layout, including the layout component itself, you can now define styles with your font's `cssVariable` to apply your custom font.

In the following example, the `<h1>` heading will have the custom font applied, while the paragraph `<p>` will not.

src/pages/example.astro

```
1 ---import Layout from "../layouts/Layout.astro";---<Layout> <h1>In a galaxy far,
  far away...</h1>
2 <p>Custom fonts make my headings much cooler!</p>
3 <style> h1 { font-family: var(--font-distant-galaxy); } </style></Layout>
```

Preloading fonts

Font preloading should be done sparingly, as it can block the loading of other important resources or download fonts that are unnecessary for the current page. Consider preloading only the most essential fonts, necessary for displaying content visible above the fold.

To preload a font, pass the [preload property](#) to the corresponding `<Font />` component. If multiple files correspond to a font, you can also specify which one to preload by passing an array.

src/layouts/Layout.astro

```

1  ---import { Font } from "astro:assets";---
2  <html> <head>    <Font cssVariable="--font-distant-galaxy" preload />  </head>  <body>
    <slot />  </body></html>

```

Register fonts in Tailwind

If you are using [Tailwind](#) for styling, you will not apply your styles with the `font-face` CSS property.

Instead, after [configuring your custom font](#) and [adding it to your page head](#), you will need to update your Tailwind configuration to register your font:

- [Tailwind CSS 4.0](#)
- [Tailwind CSS 3.0](#)

src/styles/global.css

```

1  @import "tailwindcss";
2  @theme inline { --font-sans: var(--font-roboto);}

```

See [Tailwind's docs on adding custom font families](#) for more information.

Using variable fonts

To use [variable fonts](#) in your project, specify the available weight range instead of individual weights in your provider's configuration.

- [Local provider](#)
- [Other providers](#)

When [using a local font file](#), you can specify that the font is variable by setting the [weight property of the variant](#) to a string corresponding to the exact weight range available for the font.

The following example configures Inter as a local variable font with the available weight range:

astro.config.mjs

```

1  import { defineConfig, fontProviders } from "astro/config";
2  export default defineConfig({  fonts: [{  provider: fontProviders.local(),  name:
  "Inter",  cssVariable: "--font-inter",  options: {  variants: [  {
  weight: "100 900",  style: "normal",  src:
  ["../src/assets/fonts/InterVariable.woff2"],  },  ],  },  }]);

```

Customizing font fallbacks

Fallback fonts are used when the primary font has not yet loaded, contains missing characters, or cannot be loaded for any reason. When the fallback font differs significantly from the primary font, layout shifts may occur during page loading.

To avoid this, Astro automatically tries to generate optimized fallback fonts from the last [defined fallback](#) if it is a [generic font family](#). It uses `sans-serif` by default, but it may not match the desired appearance of your primary font. You can adjust it in your font configuration:

astro.config.mjs

```
1 import { defineConfig, fontProviders } from "astro/config";
2 export default defineConfig({ fonts: [{ provider: fontProviders.fontsource(),
  name: "Cousine", cssVariable: "--font-cousine", fallbacks: ["monospace"] }]]});
```

You can also opt out of the default optimization by setting `font.optimizedFallbacks` to `false` in your font configuration. Astro will then use the fallback fonts specified in your configuration without any additional automatic processing.

Accessing font data programmatically

Astro exposes low-level APIs for accessing data programmatically:

- Font family data through the `fontData` object
- Font file URLs with the `experimental_getFontFileURL()` function.

This can be useful for advanced use cases where you need direct access to font files, such as generating OpenGraph images with [Satori](#) in an [API Route](#).

The `fontData` object gives you access to all font files downloaded by Astro for your project, along with their metadata. This means that you are responsible for filtering font files to find the specific file you need, and for fetching data after resolving URLs.

The following example generates an OpenGraph image in a static file endpoint, assuming that only [one font and its format have been configured](#) with a [format supported by Satori](#):

src/pages/og.png.ts

```
1 import type { APIRoute } from "astro";import { fontData, experimental_getFontFileURL }
  from "astro:assets";import satori from "satori";import { html } from "satori-
  html";import sharp from "sharp";
2 export const GET: APIRoute = async (context) => { const fontPath = fontData["--font-
  roboto"][0]?.src[0]?.url;
3   if (fontPath === undefined) { throw new Error("Cannot find the font path."); }
4   const url = experimental_getFontFileURL(fontPath, context.url); const data = await
  fetch(url).then((res) => res.arrayBuffer());
5   const svg = await satori( html`<div style="color: black;">hello, world</div>`, {
    width: 600, height: 400, fonts: [ { name: "Roboto",
    data, weight: 400, style: "normal", }, ],, );
6   const pngBuffer = await sharp(Buffer.from(svg)) // requires `@types/node` in your
  dependencies for type safety .resize(600, 400) .png() .toBuffer();
7   return new Response(new Uint8Array(pngBuffer), { headers: { "Content-Type":
  "image/png", }, });};
```

Granular font configuration

A font family is defined by a combination of properties such as weights and styles (e.g. `weights: [500, 600]` and `styles: ["normal", "bold"]`), but you may want to download only certain combinations of these.

For greater control over which font files are downloaded, you can specify the same font (ie. with the same `cssVariable`, `name`, and `provider` properties) multiple times with different combinations. Astro will merge the results and download only the required files. For example, it is possible to download normal `500` and `600` while downloading only italic `500`:

astro.config.mjs

```
1 import { defineConfig, fontProviders } from "astro/config";
2 export default defineConfig({ fonts: [ { name: "Roboto", cssVariable: "--roboto", provider: fontProviders.google(), weights: [500, 600], styles: ["normal"] }, { name: "Roboto", cssVariable: "--roboto", provider: fontProviders.google(), weights: [500], styles: ["italic"] } ]});
```

Caching

The Fonts API caching implementation was designed to be practical in development and efficient in production. During builds, font files are copied to the `_astro/fonts` output directory, so they can benefit from HTTP caching of static assets (usually a year).

To clear the cache in development, remove the `.astro/fonts` directory. To clear the build cache, remove the `node_modules/.astro/fonts` directory.

Examples

Astro's font feature is based on flexible configuration options. Your own project's font configuration may look different from simplified examples, so the following are provided to show what various font configurations might look like when used in production.

astro.config.mjs

```
1 import { defineConfig, fontProviders } from "astro/config";
2 export default defineConfig({ fonts: [ { name: "Roboto", cssVariable: "--font-roboto", provider: fontProviders.google(), // Default included: // weights: [400] , // styles: ["normal", "italic"], // subsets: ["latin"], // fallbacks: ["sans-serif"], // formats: ["woff2"], }, { name: "Inter", cssVariable: "--font-inter", provider: fontProviders.fontsource(), // Specify weights that are actually used weights: [400, 500, 600, 700], // Specify styles that are actually used styles: ["normal"], // Download only font files for characters used on the page subsets: ["latin", "cyrillic"], // Download more font formats formats: ["woff2", "woff"], }, { name: "JetBrains Mono", cssVariable: "--font-jetbrains-mono", provider: fontProviders.fontsource(), // Download only font files for characters used on the page subsets: ["latin", "latin-ext"], // Use a fallback font family matching the intended appearance fallbacks: ["monospace"], }, { name: "Poppins", cssVariable: "--font-poppins", provider: fontProviders.local(), options: { // weight and style are not specified so Astro // will try to infer them for each variant variants: [ { src: [ "./src/assets/fonts/Poppins-regular.woff2", ".src/assets/fonts/Poppins-regular.woff", ] }, { src: [ "./src/assets/fonts/Poppins-bold.woff2", ".src/assets/fonts/Poppins-bold.woff", ] } ] } ] });
```

Syntax Highlighting

Astro comes with built-in support for [Shiki](#) and [Prism](#). This provides syntax highlighting for:

- all [code fences](#) (`````) used in a Markdown or MDX file.
- content within the [built-in ``` component](https://docs.astro.build/en/guides/syntax-highlighting/#code-) (powered by Shiki) in `.astro`` files.
- content within the [``` component](https://docs.astro.build/en/guides/syntax-highlighting/#prism-) (powered by Prism) in `.astro`` files.

Add [community integrations such as Expressive Code](#) for even more text marking and annotation options in your code blocks.

Markdown code blocks

A Markdown code block is indicated by a block with three backticks ````` at the start and end. You can indicate the programming language being used after the opening backticks to indicate how to color and style your code to make it easier to read.

```
1  ```js
2
3  // JavaScript code with syntax highlighting.
4
5  var fun = function lang(l) {
6
7      dateFormat.i18n = require('./lang/' + l);
8
9      return true;
10
11  };
12
13  ```
```

Astro's Markdown code blocks are styled by Shiki by default, preconfigured with the `github-dark` theme. The compiled output will be limited to inline `style`s without any extraneous CSS classes, stylesheets, or client-side JS.

You can [add a Prism stylesheet and switch to Prism's highlighting](#), or disable Astro's syntax highlighting entirely, with the `markdown.syntaxHighlight` configuration option.

See the full [markdown.shikiConfig reference](#) for the complete set of Markdown syntax highlighting options available when using Shiki.

Setting a default Shiki theme

You can configure any [built-in Shiki theme](#) for your Markdown code blocks in your Astro config:

`astro.config.mjs`

```

1 import { defineConfig } from 'astro/config';
2 export default defineConfig({ markdown: { shikiConfig: { theme: 'dracula',
  }, }, });

```

See the full [Shiki config reference](#) for the complete set of Markdown code block options.

Setting light and dark mode themes

You can specify dual Shiki themes for light and dark mode in your Astro config:

astro.config.mjs

```

1 import { defineConfig } from 'astro/config';
2 export default defineConfig({ markdown: { shikiConfig: { themes: {
  light: 'github-light',      dark: 'github-dark',    }, }, });

```

Then, [add Shiki's dark mode CSS variables via media query or classes](#) to apply to all your Markdown code blocks by default. Replace the `.shiki` class in the examples from Shiki's documentation with `.astro-code`:

src/styles/global.css

```

1 @media (prefers-color-scheme: dark) {
2
3   .shiki,
4
5   .shiki span {
6
7   .astro-code,
8
9   .astro-code span {
10
11     color: var(--shiki-dark) !important;
12
13     background-color: var(--shiki-dark-bg) !important;
14
15     /* Optional, if you also want font styles */
16
17     font-style: var(--shiki-dark-font-style) !important;
18
19     font-weight: var(--shiki-dark-font-weight) !important;
20
21     text-decoration: var(--shiki-dark-text-decoration) !important;
22
23   }
24
25 }

```

See the full [Shiki config reference](#) for the complete set of Markdown code block options.

Adding your own Shiki theme

Instead of using one of Shiki's predefined themes, you can import a custom Shiki theme from a local file.

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import customTheme from './my-shiki-theme.json';
2 export default defineConfig({  markdown: {    shikiConfig: {      theme: customTheme,
  },  },});
```

Customizing Shiki themes

You can follow [Shiki's own theme documentation](#) for more customization options for themes, [light vs dark mode toggles](#), or styling via [CSS variables](#).

You will need to adjust the examples from Shiki's documentation for your Astro project by making the following substitutions:

- Code blocks are styled using the `.astro-code` class instead of `.shiki`
- When using the `css-variables` theme, custom properties are prefixed with `--astro-code-` instead of `--shiki-`

Components for code blocks

There are two Astro components available for `.astro` and `.mdx` files to render code blocks: `<Code />` and `<CodeBlock />`.

You can reference the `Props` of these components using the [ComponentProps_type](#) utility.

```
<Code />
```

This component is powered internally by Shiki. It supports all popular Shiki themes and languages as well as several other Shiki options such as custom themes, languages, [transformers](#), and default colors.

These values are passed to the `<Code />` component using the `theme`, `lang`, [embeddedLangs](#), [transformers](#), and `defaultColor` attributes respectively as props. The `<Code />` component will not inherit your `shikiConfig` settings for Markdown code blocks.

```
1 ---
2
3 import { Code } from 'astro:components';
4
5 ---
6
7 <!-- Syntax highlight some JavaScript code. -->
8
9 <Code code={`const foo = 'bar';`} lang="js" />
10
11 <!-- Optional: Customize your theme. -->
12
13 <Code code={`const foo = 'bar';`} lang="js" theme="dark-plus" />
14
```

```

15 <!-- Optional: Enable word wrapping. -->
16
17 <Code code={`const foo = 'bar';`} lang="js" wrap />
18
19 <!-- Optional: Output inline code. -->
20
21 <p>
22
23   <Code code={`const foo = 'bar';`} lang="js" inline />
24
25   will be rendered inline.
26
27 </p>
28
29 <!-- Optional: defaultColor -->
30
31 <Code code={`const foo = 'bar';`} lang="js" defaultColor={false} />

```

embeddedLangs

Type: `string[] | undefined`

Added in: `astro@6.0.0`

Any additional languages to be included for syntax highlighting by Shiki.

A `lang` value may include support for highlighting some additional languages by default (e.g. `lang="svelte"` will also provide highlighting for `ts`).

Use `embeddedLangs` to include support for additional, non-standard language combinations (e.g. `jsx` support when `lang="vue"`).

`src/pages/index.astro`

```

1  ---import { Code } from 'astro:components'
2  const code = `
```

An array of [Shiki transformers](#) to be applied to your `code`. Since Astro v4.14.0, you can also provide a string for [Shiki's `meta` attribute](#) to pass options to transformers.

Note that `transformers` only applies classes and you must provide your own CSS rules to target the elements of your code block.

src/pages/index.astro

```
1  ---import { transformerNotationFocus, transformerMetaHighlight } from
   '@shikijs/transformers'import { Code } from 'astro:components'const code = `const foo =
   'hello'const bar = ' world'console.log(foo + bar) // [!code focus]`---<Code code={code}
   lang="js" transformers=[[transformerMetaHighlight()]] meta="{1,3}"/>
2  <style is:global> pre.has-focused .line:not(.focused) { filter: blur(1px); }
   </style>
```

<Prism />

This component provides language-specific syntax highlighting for code blocks by applying Prism's CSS classes. Note that you must [provide a Prism CSS stylesheet](#) (or bring your own) to style the classes.

To use the `Prism` highlighter component, you must install the `@astrojs/prism` package:

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1  npm install @astrojs/prism
```

Then, you can import and use the `<Prism />` component like any other Astro component, passing a language and the code to render.

```
1  ---
2
3  import { Prism } from '@astrojs/prism';
4
5  ---
6
7  <Prism lang="js" code={`const foo = 'bar';`} />
```

In addition to the [list of languages supported by Prism](#), you can also use `lang="astro"` to display Astro code blocks.

Add a Prism stylesheet

If you opt to use Prism (either by configuring `markdown.syntaxHighlight: 'prism'` or with the `<Prism />` component), Astro will apply Prism's CSS classes instead of Shiki's to your code. You will need to bring your own CSS stylesheet for syntax highlighting to appear.

1. Choose a premade stylesheet from the available [Prism Themes](#).
2. Add this stylesheet to [your project's public/ directory](#).
3. Load this into your page's `<head>` in a [layout component](#) via a `<link>` tag. (See [Prism basic usage](#).)

You can also visit the [list of languages supported by Prism](#) for options and usage.

Scripts and event handling

You can send JavaScript to the browser and add functionality to your Astro components using `<script>` tags in the component template.

Scripts add interactivity to your site, such as handling events or updating content dynamically, without the need for a [UI framework](#) like React, Svelte, or Vue. This avoids the overhead of shipping framework JavaScript and doesn't require you to know any additional framework to create a full-featured website or application.

Client-Side Scripts

Scripts can be used to add event listeners, send analytics data, play animations, and everything else JavaScript can do on the web.

Astro automatically enhances the HTML standard `<script>` tag with bundling, TypeScript, and more. See [how astro processes scripts](#) for more details.

`src/components/ConfettiButton.astro`

```
1 <button data-confetti-button>Celebrate!</button>
2 <script> // Import from npm package. import confetti from 'canvas-confetti';
3   // Find our component DOM on the page. const buttons =
  document.querySelectorAll('[data-confetti-button]');
4   // Add event listeners to fire confetti when a button is clicked.
  buttons.forEach((button) => {    button.addEventListener('click', () => confetti());
  });</script>
```

See [when your scripts will not be processed](#) to troubleshoot script behavior, or to learn how to opt-out of this processing intentionally.

Script processing

By default, Astro processes `<script>` tags that contain no attributes (other than `src`) in the following ways:

- **TypeScript support:** All scripts are TypeScript by default.
- **Import bundling:** Import local files or npm modules, which will be bundled together.
- **Type Module:** Processed scripts become `type="module"` automatically.

- **Deduplication:** If a component that contains a `<script>` is used multiple times on a page, the script will only be included once.
- **Automatic inlining:** If the script is small enough, Astro will inline it directly into the HTML to reduce the number of requests.

src/components/Example.astro

```
1 <script>
2
3   // Processed! Bundled! TypeScript!
4
5   // Importing local scripts and from npm packages works.
6
7 </script>
```

Unprocessed scripts

Astro will not process a `<script>` tag if it has any attribute other than `src`.

You can add the `is:inline` directive to intentionally opt out of processing for a script.

src/components/InlineScript.astro

```
1 <script is:inline>
2
3   // will be rendered into the HTML exactly as written!
4
5   // Not transformed: no TypeScript and no import resolution by Astro.
6
7   // If used inside a component, this code is duplicated for each instance.
8
9 </script>
```

Include JavaScript files on your page

You may want to write your scripts as separate `.js/.ts` files or need to reference an external script on another server. You can do this by referencing these in a `<script>` tag's `src` attribute.

Import local scripts

When to use this: when your script lives inside of `src/`.

Astro will process these scripts according to the [script processing rules](#).

src/components/LocalScripts.astro

```
1 <!-- relative path to script at `src/scripts/local.js` --><script
  src="../../scripts/local.js"></script>
2 <!-- also works for local TypeScript files --><script src="../../script-with-types.ts">
  </script>
```

Load external scripts

When to use this: when your JavaScript file lives inside of `public/` or on a CDN.

To load scripts outside of your project's `src/` folder, include the `is:inline` directive. This approach skips the JavaScript processing, bundling, and optimizations that are provided by Astro when you import scripts as described above.

`src/components/ExternalScripts.astro`

```
1 <!-- absolute path to a script at `public/my-script.js` --><script is:inline src="/my-script.js"></script>
2 <!-- full URL to a script on a remote server --><script is:inline src="https://my-analytics.com/script.js"></script>
```

Common script patterns

Handle `onClick` and other events

Some UI frameworks use custom syntax for event handling like `onClick={...}` (React/Preact) or `@click="..."` (Vue). Astro follows standard HTML more closely and does not use custom syntax for events.

Instead, you can use `addEventListener` in a `<script>` tag to handle user interactions.

`src/components/AlertButton.astro`

```
1 <button class="alert">Click me!</button>
2 <script> // Find all buttons with the `alert` class on the page.  const buttons =
  document.querySelectorAll('button.alert');
3 // Handle clicks on each button.  buttons.forEach((button) => {
  button.addEventListener('click', () => {      alert('Button was clicked!');    }); });
</script>
```

If you have multiple `<AlertButton />` components on a page, Astro will not run the script multiple times. Scripts are bundled and only included once per page. Using `querySelectorAll` ensures that this script attaches the event listener to every button with the `alert` class found on the page.

Web components with custom elements

You can create your own HTML elements with custom behavior using the Web Components standard. Defining a [custom element](#) in a `.astro` component allows you to build interactive components without needing a UI framework library.

In this example, we define a new `<astro-heart>` HTML element that tracks how many times you click the heart button and updates the `<span>` with the latest count.

`src/components/AstroHeart.astro`

```

1 <!-- wrap the component elements in our custom element "astro-heart". --><astro-heart>
  <button aria-label="Heart">♥</button> × <span>0</span></astro-heart>
2 <script> // Define the behavior for our new type of HTML element. class AstroHeart
  extends HTMLElement { connectedCallback() { let count = 0;
3     const heartButton = this.querySelector("button"); const countSpan =
  this.querySelector("span");
4     // Each time the button is clicked, update the count. if (heartButton &&
  countSpan) { heartButton.addEventListener("click", () => { count++;
  countSpan.textContent = count.toString(); }); } } }
5 // Tell the browser to use our AstroHeart class for <astro-heart> elements.
  customElements.define("astro-heart", AstroHeart);</script>

```

There are two advantages to using a custom element here:

1. Instead of searching the whole page using `document.querySelector()`, you can use `this.querySelector()`, which only searches within the current custom element instance. This makes it easier to work with only the children of one component instance at a time.
2. Although a `<script>` only runs once, the browser will run our custom element's `connectedCallback()` method each time it finds `<astro-heart>` on the page. This means you can safely write code for one component at a time, even if you intend to use this component multiple times on a page.

You can learn more about custom elements in [web.dev's Reusable Web Components guide](#) and [MDN's introduction to custom elements](#).

Pass frontmatter variables to scripts

In Astro components, the code in [the frontmatter](#) (between the `---` fences) runs on the server and is not available in the browser.

To pass server-side variables to client-side scripts, store them in [data-* attributes](#) on HTML elements. Scripts can then access these values using the `dataset` property.

In this example component, a `message` prop is stored in a `data-message` attribute, so the custom element can read `this.dataset.message` and get the value of the prop in the browser.

src/components/AstroGreet.astro

```

1 ---const { message = 'welcome, world!' } = Astro.props;---
2 <!-- Store the message prop as a data attribute. --><astro-greet data-message={message}>
  <button>Say hi!</button></astro-greet>
3 <script> class AstroGreet extends HTMLElement { connectedCallback() { // Read
  the message from the data attribute. const message = this.dataset.message;
  const button = this.querySelector('button'); button?.addEventListener('click', ()
  => { alert(message); }); } }
4 customElements.define('astro-greet', AstroGreet);</script>

```

Now we can use our component multiple times and be greeted by a different message for each one.

src/pages/example.astro

```
1 ---import AstroGreet from '../components/AstroGreet.astro';---
2 <!-- Use the default message: "welcome, world!" --><AstroGreet />
3 <!-- Use custom messages passed as a props. --><AstroGreet message="Lovely day to build
  components!" /><AstroGreet message="Glad you made it! 🙌" />
```

Did you know?

This is actually what Astro does behind the scenes when you pass props to a component written using a UI framework like React! For components with a `client:*` directive, Astro creates an `<astro-island>` custom element with a `props` attribute that stores your server-side props in the HTML output.

Combining scripts and UI Frameworks

Elements rendered by a UI framework may not be available yet when a `<script>` tag executes. If your script also needs to handle [UI framework components](#), using a custom element is recommended.

Front-end frameworks

Build your Astro website without sacrificing your favorite component framework. Create Astro [islands](#) with the UI frameworks of your choice.

Official front-end framework integrations

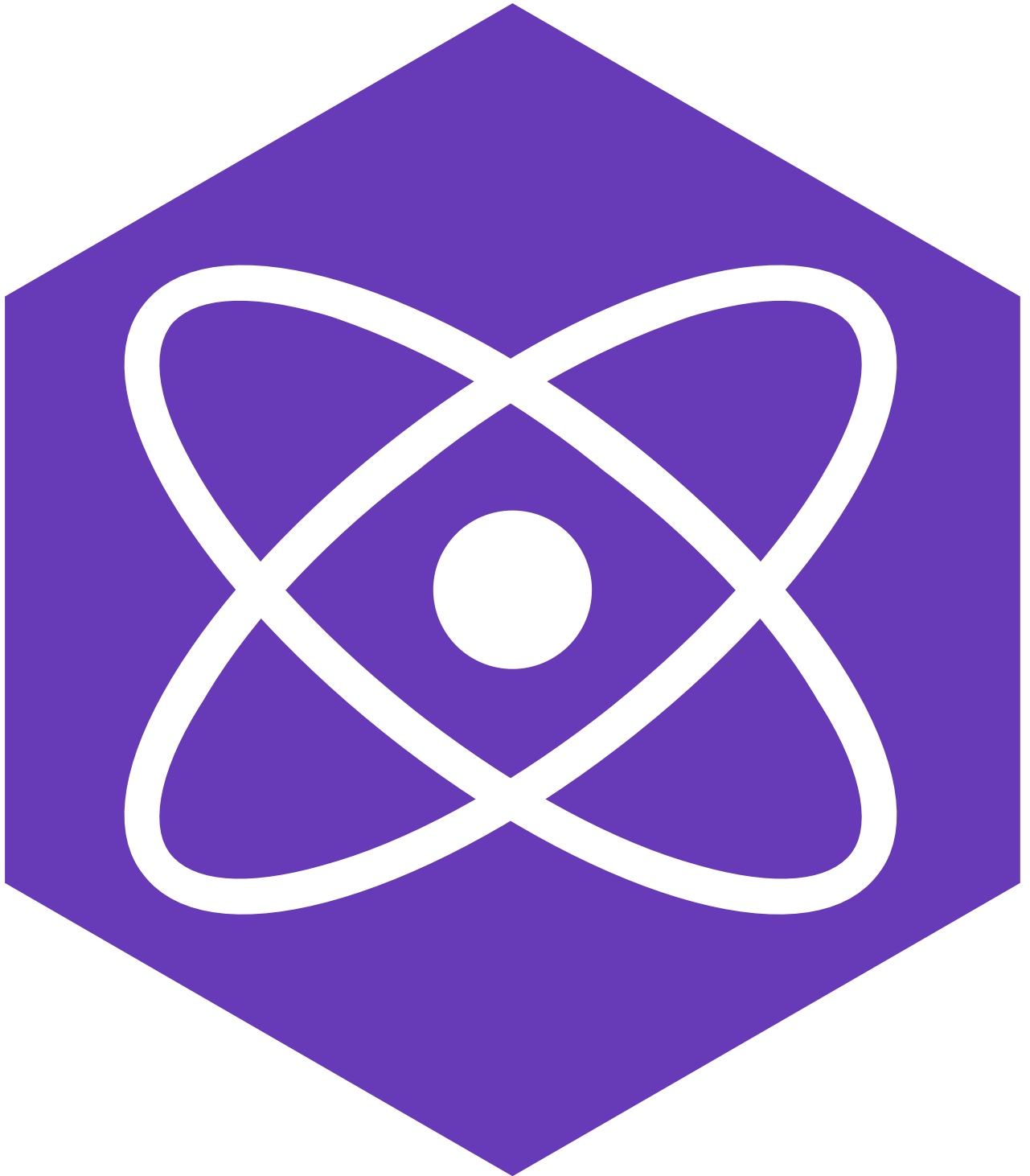
Astro supports a variety of popular frameworks including [React](#), [Preact](#), [Svelte](#), [Vue](#), [SolidJS](#), and [AlpineJS](#) with official integrations.

Find even more [community-maintained framework integrations](#) (e.g. Angular, Qwik, Elm) in our integrations directory.

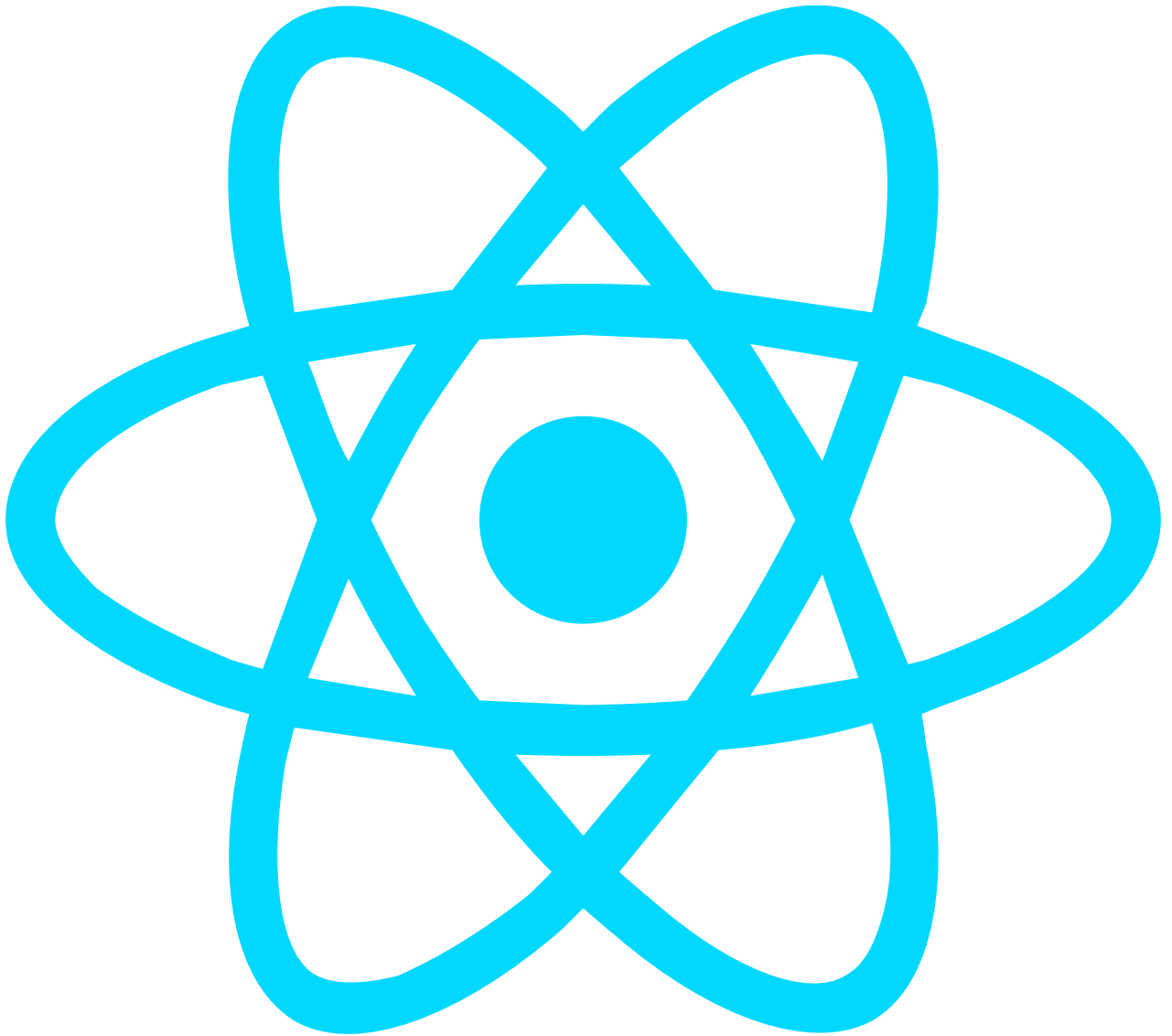
Front-end frameworks



- [@astrojs/alpinejs](https://twitter.com/astrojs/alpinejs)



- [@astrojs/preact](#)



- [@astrojs/react](#)



- [@astrojs/solid-js](#)



- [@astrojs/svelte](#)



- [@astrojs/vue](#)

Installing integrations

One or several of these Astro integrations can be installed and configured in your project.

See the [Integrations Guide](#) for more details on installing and configuring Astro integrations.

Tip

Want to see an example for the framework of your choice? Visit [astro.new](#) and select one of the framework templates.

Using framework components

Use your JavaScript framework components in your Astro pages, layouts and components just like Astro components! All your components can live together in `/src/components`, or can be organized in any way you like.

To use a framework component, import it from its relative path in your Astro component script. Then, use the component alongside other components, HTML elements and JSX-like expressions in the component template.

`src/pages/static-components.astro`

```
1  ---
2
3  import MyReactComponent from '../components/MyReactComponent.jsx';
4
5  ---
6
7  <html>
8
9    <body>
10
11      <h1>Use React components directly in Astro!</h1>
12
13      <MyReactComponent />
14
15    </body>
16
17  </html>
```

By default, your framework components will only render on the server, as static HTML. This is useful for templating components that are not interactive and avoids sending any unnecessary JavaScript to the client.

Hydrating interactive components

A framework component can be made interactive (hydrated) using a [client:* directive](#). These are component attributes that determine when your component's JavaScript should be sent to the browser.

With all client directives except `client:only`, your component will first render on the server to generate static HTML. Component JavaScript will be sent to the browser according to the directive you chose. The component will then hydrate and become interactive.

`src/pages/interactive-components.astro`

```
1  ---// Example: hydrating framework components in the browser.import InteractiveButton
   from '../components/InteractiveButton.jsx';import InteractiveCounter from
   '../components/InteractiveCounter.jsx';import InteractiveModal from
   '../components/InteractiveModal.svelte';---<!-- This component's JS will begin importing
   when the page loads --><InteractiveButton client:load />
2  <!-- This component's JS will not be sent to the client untilthe user scrolls down and
   the component is visible on the page --><InteractiveCounter client:visible />
3  <!-- This component won't render on the server, but will render on the client when the
   page loads --><InteractiveModal client:only="svelte" />
```

The JavaScript framework (React, Svelte, etc.) needed to render the component will be sent to the browser along with the component's own JavaScript. If two or more components on a page use the same framework, the framework will only be sent once.

Accessibility

Most framework-specific accessibility patterns should work the same when these components are used in Astro. Be sure to choose a client directive that will ensure any accessibility-related JavaScript is properly loaded and executed at the appropriate time!

Available hydration directives

There are several hydration directives available for UI framework components: `client:load`, `client:idle`, `client:visible`, `client:media={QUERY}` and `client:only={FRAMEWORK}`.

See our [directives reference](#) page for a full description of these hydration directives, and their usage.

Mixing frameworks

You can import and render components from multiple frameworks in the same Astro component.

`src/pages/mixing-frameworks.astro`

```
1  ---
2
3  // Example: Mixing multiple framework components on the same page.
4
5  import MyReactComponent from '../components/MyReactComponent.jsx';
6
7  import MySvelteComponent from '../components/MySvelteComponent.svelte';
8
9  import MyVueComponent from '../components/MyVueComponent.vue';
10
11  ---
12
13  <div>
14
15      <MySvelteComponent />
16
17      <MyReactComponent />
18
19      <MyVueComponent />
20
21  </div>
```

Astro will recognize and render your component based on its file extension. To distinguish between frameworks that use the same file extension, [additional configuration when rendering multiple JSX frameworks](#) (e.g. React and Preact) is required.

Caution

Only **Astro** components (`.astro`) can contain components from multiple frameworks.

Passing props to framework components

You can pass props from Astro components to framework components:

src/pages/frameworks-props.astro

```
1  ---
2
3  import TodoList from '../components/TodoList.jsx';
4
5  import Counter from '../components/Counter.svelte';
6
7  ---
8
9  <div>
10
11    <TodoList initialTodos=[["learn Astro", "review PRs"]] />
12
13    <Counter startingCount={1} />
14
15  </div>
```

Props that are passed to interactive framework components [using a `client:\*` directive](#) must be [serialized](#): translated into a format suitable for transfer over a network, or storage. However, Astro does not serialize every type of data structure. Therefore, there are some limitations on what can be passed as props to hydrated components.

The following prop types are supported: plain object, `number`, `string`, `Array`, `Map`, `Set`, `RegExp`, `Date`, `BigInt`, `URL`, `Uint8Array`, `Uint16Array`, `Uint32Array`, and `Infinity`

Non-supported data structures passed to components, such as functions, can only be used during the component's server rendering and cannot be used to provide interactivity. For example, passing functions to hydrated components is not supported because Astro cannot pass functions from the server in a way that makes them executable on the client.

Passing children to framework components

Inside of an Astro component, you **can** pass children to framework components. Each framework has its own patterns for how to reference these children: React, Preact, and Solid all use a special prop named `children`, while Svelte and Vue use the `<slot />` element.

src/pages/component-children.astro

```

1  ---
2
3  import MyReactSidebar from '../components/MyReactSidebar.jsx';
4
5  ---
6
7  <MyReactSidebar>
8
9    <p>Here is a sidebar with some text and a button.</p>
10
11  </MyReactSidebar>

```

Additionally, you can use [Named Slots](#) to group specific children together.

For React, Preact, and Solid, these slots will be converted to a top-level prop. Slot names using `kebab-case` will be converted to `camelCase`.

src/pages/named-slots.astro

```

1  ---
2
3  import MySidebar from '../components/MySidebar.jsx';
4
5  ---
6
7  <MySidebar>
8
9    <h2 slot="title">Menu</h2>
10
11    <p>Here is a sidebar with some text and a button.</p>
12
13    <ul slot="social-links">
14
15      <li><a href="https://twitter.com/astrodotbuild">Twitter</a></li>
16
17      <li><a href="https://github.com/withastro">GitHub</a></li>
18
19    </ul>
20
21  </MySidebar>

```

src/components/MySidebar.jsx

```

1  export default function MySidebar(props) {
2
3    return (
4
5      <aside>
6
7        <header>{props.title}</header>
8
9        <main>{props.children}</main>

```

```

10
11     <footer>{props.socialLinks}</footer>
12
13 </aside>
14
15 )
16
17 }

```

For Svelte and Vue these slots can be referenced using a `<slot>` element with the `name` attribute. Slot names using `kebab-case` will be preserved.

src/components/MySidebar.svelte

```

1 <aside>
2
3   <header><slot name="title" /></header>
4
5   <main><slot /></main>
6
7   <footer><slot name="social-links" /></footer>
8
9 </aside>

```

Nesting framework components

Inside of an Astro file, framework component children can also be hydrated components. This means that you can recursively nest components from any of these frameworks.

src/pages/nested-components.astro

```

1 ---
2
3 import MyReactSidebar from '../components/MyReactSidebar.jsx';
4
5 import MyReactButton from '../components/MyReactButton.jsx';
6
7 import MySvelteButton from '../components/MySvelteButton.svelte';
8
9 ---
10
11 <MyReactSidebar>
12
13   <p>Here is a sidebar with some text and a button.</p>
14
15   <div slot="actions">
16
17     <MyReactButton client:idle />
18
19     <MySvelteButton client:idle />
20
21   </div>

```

```
22 |
23 | </MyReactSidebar>
```

Caution

Remember: framework component files themselves (e.g. `.jsx`, `.svelte`) cannot mix multiple frameworks.

This allows you to build entire “apps” in your preferred JavaScript framework and render them, via a parent component, to an Astro page.

Note

Astro components are always rendered to static HTML, even when they include framework components that are hydrated. This means that you can only pass props that don’t do any HTML rendering. Passing React’s “render props” to framework components from an Astro component will not work, because Astro components can’t provide the client runtime behavior that this pattern requires. Instead, use named slots.

Can I use Astro components inside my framework components?

Any UI framework component becomes an “island” of that framework. These components must be written entirely as valid code for that framework, using only its own imports and packages. You cannot import `.astro` components in a UI framework component (e.g. `.jsx` or `.svelte`).

You can, however, use [the Astro ``pattern``](https://docs.astro.build/en/basics/astro-components/#slots) to pass static content generated by Astro components as children to your framework components **inside an `.astro`` component**.

`src/pages/astro-children.astro`

```
1  ---
2
3  import MyReactComponent from '../components/MyReactComponent.jsx';
4
5  import MyAstroComponent from '../components/MyAstroComponent.astro';
6
7  ---
8
9  <MyReactComponent>
10
11    <MyAstroComponent slot="name" />
12
13  </MyReactComponent>
```

Can I hydrate Astro components?

If you try to hydrate an Astro component with a `client:` modifier, you will get an error.

[Astro components](#) are HTML-only templating components with no client-side runtime. But, you can use a `<script>` tag in your Astro component template to send JavaScript to the browser that executes in the global scope.

Learn more about [client-side `` tags in Astro components](<https://docs.astro.build/en/guides/client-side-scripts/>)

Markdown in Astro

[Markdown](#) is commonly used to author text-heavy content like blog posts and documentation. Astro includes built-in support for Markdown files that can also include [frontmatter YAML](#) (or [TOML](#)) to define custom properties such as a title, description, and tags.

In Astro, you can author content in [GitHub Flavored Markdown](#), then render it in `.astro` components. This combines a familiar writing format designed for content with the flexibility of Astro's component syntax and architecture.

Tip

For additional functionality, such as including components and JSX expressions in Markdown, add the [@astrojs/mdx integration](#) to write your Markdown content using [MDX](#).

Organizing Markdown files

Your local Markdown files can be kept anywhere within your `src/` directory. Markdown files located within `src/pages/` will automatically generate [Markdown pages on your site](#).

Your Markdown content and frontmatter properties are available to use in components through [local file imports](#) or when [queried and rendered from data fetched by a content collections helper function](#).

File imports vs content collections queries

Local Markdown can be imported into `.astro` components using an `import` statement for a single file and [Vite's `import.meta.glob\(\)`](#) to query multiple files at once. The [exported data from these Markdown files](#) can then be used in the `.astro` component.

If you have groups of related Markdown files, consider [defining them as collections](#). This gives you several advantages, including the ability to store Markdown files anywhere on your filesystem or remotely.

Collections use content-specific, optimized APIs for [querying and rendering your Markdown content](#) instead of file imports. Collections are intended for sets of data that share the same structure, such as blog posts or product items. When you define that shape in a schema, you additionally get validation, type safety, and Intellisense in your editor.

See more about [when to use content collections](#) instead of file imports.

Dynamic JSX-like expressions

After importing or querying Markdown files, you can write dynamic HTML templates in your `.astro` components that include frontmatter data and body content.

`src/pages/posts/great-post.md`

```
1 ---title: 'The greatest post of all time'author: 'Ben'---
2 Here is my _great_ post!
```

src/pages/my-posts.astro

```
1  ---import * as greatPost from "./posts/great-post.md";const compiled = await
   greatPost.compiledContent();const posts = Object.values(import.meta.glob("./posts/*.md",
   { eager: true }));---
2  <p>{greatPost.frontmatter.title}</p><p>Written by: {greatPost.frontmatter.author}</p>
3  <Fragment set:html={compiled} />
4  <p>Post Archive:</p><ul> {    posts.map((post: any) => (    <li>          <a href=
   {post.url}>{post.frontmatter.title}</a>          </li>    )) }</ul>
```

Available Properties

Markdown from content collections queries

When fetching data from your collections with the helper functions `getCollection()` or `getEntry()`, your Markdown's frontmatter properties are available on a `data` object (e.g. `post.data.title`). Additionally, `body` contains the raw, uncompiled body content as a string.

The `render()` function returns your Markdown body content, a generated list of headings, as well as a modified frontmatter object after any Markdown plugins have been applied.

Read more about [using content returned by a collections query](#).

Importing Markdown

The following exported properties are available in your `.astro` component when importing Markdown using `import` or `import.meta.glob()`:

- `file` - The absolute file path (e.g. `/home/user/projects/.../file.md`).
- `url` - The URL of the page (e.g. `/en/guides/markdown-content`).
- `frontmatter` - Contains any data specified in the file's YAML (or TOML) frontmatter.
- `<Content />` - A component that returns the full, rendered contents of the file.
- `rawContent()` - A function that returns the raw Markdown document as a string.
- `compiledContent()` - An async function that returns the Markdown document compiled to an HTML string.
- `getHeadings()` - An async function that returns an array of all headings (`<h1>` to `<h6>`) in the file with the type: `{ depth: number; slug: string; text: string }[]`. Each heading's `slug` corresponds to the generated ID for a given heading and can be used for anchor links.

An example Markdown blog post may pass the following `Astro.props` object:

```
1  Astro.props = {
2
3    file: "/home/user/projects/.../file.md",
4
5    url: "/en/guides/markdown-content/",
6
7    frontmatter: {
8
```

```

 9  /** Frontmatter from a blog post */
10
11  title: "Astro 0.18 Release",
12
13  date: "Tuesday, July 27 2021",
14
15  author: "Matthew Phillips",
16
17  description: "Astro 0.18 is our biggest release since Astro launch.",
18
19  },
20
21  getHeadings: () => [
22
23    {"depth": 1, "text": "Astro 0.18 Release", "slug": "astro-018-release"},
24
25    {"depth": 2, "text": "Responsive partial hydration", "slug": "responsive-partial-
hydration"}
26
27    /* ... */
28
29  ],
30
31  rawContent: () => "# Astro 0.18 Release\nA little over a month ago, the first public
beta [...]",
32
33  compiledContent: () => "<h1>Astro 0.18 Release</h1>\n<p>A little over a month ago,
the first public beta [...]</p>",
34
35  }

```

The `<Content />` Component

The `<Content />` component is available by importing `Content` from a Markdown file. This component returns the file's full body content, rendered to HTML. You can optionally rename `Content` to any component name you prefer.

You can similarly [render the HTML content of a Markdown collection entry](#) by rendering a `<Content />` component.

src/pages/content.astro

```

1  ---// Import statementimport {Content as PromoBanner} from
  '../components/promoBanner.md';
2  // Collections queryimport { getEntry, render } from 'astro:content';
3  const product = await getEntry('products', 'shirt');const { Content } = await
  render(product);---<h2>Today's promo</h2><PromoBanner />
4  <p>Sale Ends: {product.data.saleEndDate.toDateString()}</p><Content />

```

Heading IDs

Writing headings in Markdown will automatically give you anchor links so you can link directly to certain sections of your page.

src/pages/page-1.md

```
1 ---title: My page of content---## Introduction
2 I can link internally to [my conclusion](#conclusion) on the same page when writing
  Markdown.
3 ## Conclusion
4 I can visit `https://example.com/page-1/#introduction` in a browser to navigate directly
  to my Introduction.
```

Astro generates heading `id`s based on `github-slugger`. You can find more examples in [the github-slugger documentation](#).

Heading IDs and plugins

Astro injects an `id` attribute into all heading elements (`<h1>` to `<h6>`) in Markdown and MDX files. You can retrieve this data from the `getHeadings()` utility available as a [Markdown exported property](#) from an imported file, or from the `render()` function when [using Markdown returned from a content collections query](#).

You can customize these heading IDs with a [Markdown processor](#) plugin that injects `id` attributes (e.g. `rehype-slug`). Your custom IDs, instead of Astro's defaults, will be reflected in the HTML output and the items returned by `getHeadings()`.

Astro injects `id` attributes after your custom plugins have run, so any ID set by a plugin is preserved. If one of your custom plugins needs to access the IDs injected by Astro, you can import Astro's heading ids plugin and place it before any plugins that rely on it:

- [Satteri](#)
- [Unified](#)

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import { satteri, satteriHeadingIdsPlugin }
  from '@astrojs/markdown-satteri';import { otherPluginThatReliesOnHeadingIDs } from
  'some/plugin/source';
2 export default defineConfig({ markdown: { processor: satteri({      hastPlugins: [
    satteriHeadingIdsPlugin(),      otherPluginThatReliesOnHeadingIDs,    ],    }),
  },});
```

Markdown Plugins

Astro renders Markdown using a configurable **Markdown processor**. By default, this is [Sätteri](#), Astro's native Markdown and MDX pipeline, included with Astro.

Astro applies [GitHub-Flavored Markdown](#) and [SmartyPants](#) automatically. This brings some niceties like generating clickable links from text, and formatting for quotations and em-dashes.

You can extend Markdown processing with plugins, enable additional parser features, or [switch to a different processor entirely](#). See the full list of [Markdown configuration options](#).

Choosing a Markdown processor

The `markdown.processor` option controls which engine renders your `.md` and `.mdx` files. Astro provides two official options:

- `satteri()` (default): the native [Sätteri](#) pipeline, a fast Markdown and MDX compiler with its own plugin model. Included with Astro, so no extra installation is needed.
- `unified()`: the [remark](#) and [rehype](#) pipeline, with its large plugin ecosystem. Provided by `@astrojs/markdown-remark`, which you install separately.

Using Sätteri plugins and features

The default `satteri()` processor accepts its own plugins through `mdastPlugins` and `hastPlugins`, and toggles optional parser features through `features`. See the [Sätteri documentation](#) for the available plugins and features.

Import `satteri` from `@astrojs/markdown-satteri` and pass your options to it through `markdown.processor`. This example adds an mdast plugin and enables the `directive` parser feature:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import { satteri } from '@astrojs/markdown-satteri';import { myMdastPlugin } from './my-satteri-plugin.mjs';
2 export default defineConfig({ markdown: { processor: satteri({ mdastPlugins: [myMdastPlugin()], features: { directive: true }, }), }, });
```

Switching to the unified processor

To use [remark and rehype](#) instead of Sätteri, install `@astrojs/markdown-remark`, then import `unified` and pass it to `markdown.processor`:

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 npm install @astrojs/markdown-remark
```

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import { unified } from '@astrojs/markdown-remark';
2 export default defineConfig({ markdown: { processor: unified(), },});
```

Switching processors replaces S  tteri for both `.md` and `.mdx` files. Any S  tteri plugins in your config will not apply. To use remark and rehype only for `.mdx` files, set the [processor option on the MDX integration](#) instead.

Using remark and rehype plugins

The `unified()` processor accepts third-party [remark](#) and [rehype](#) plugins. These plugins allow you to extend your Markdown with new capabilities, like [auto-generating a table of contents](#), [applying accessible emoji labels](#), and [styling your Markdown](#).

We encourage you to browse [awesome-remark](#) and [awesome-rehype](#) for popular plugins! See each plugin's own README for specific installation instructions.

After [switching to the `unified\(\)` processor](#), pass your plugins to it through `markdown.processor`. This example applies [remark-toc](#) and [rehype-accessible-emojis](#) to Markdown files:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import { unified } from '@astrojs/markdown-remark';import remarkToc from 'remark-toc';import { rehypeAccessibleEmojis } from 'rehype-accessible-emojis';
2 export default defineConfig({ markdown: { processor: unified({ remarkPlugins: [[remarkToc, { heading: 'toc', maxDepth: 3 }]], rehypePlugins: [rehypeAccessibleEmojis], }, }, });
```

Customizing a remark or rehype plugin

In order to customize a plugin, provide an options object after it in a nested array.

The example below adds the [heading option to the `remarkToc` plugin](#) to change where the table of contents is placed, and the [behavior option to the `rehype-autolink-headings` plugin](#) in order to add the anchor tag after the headline text.

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import { unified } from '@astrojs/markdown-remark';import remarkToc from 'remark-toc';import rehypeSlug from 'rehype-slug';import rehypeAutolinkHeadings from 'rehype-autolink-headings';
2 export default defineConfig({ markdown: { processor: unified({ remarkPlugins: [[remarkToc, { heading: 'contents' }]], rehypePlugins: [rehypeSlug, [rehypeAutolinkHeadings, { behavior: 'append' }]], }, }, });
```

Modifying frontmatter programmatically

When using the [remark/rehype processor](#), you can add frontmatter properties to all of your Markdown and MDX files with a remark or rehype plugin.

1. Append a `customProperty` to the `data.astro.frontmatter` property from your plugin's `file` argument:

example-remark-plugin.mjs

```
1 export function exampleRemarkPlugin() {
2
3   // All remark and rehype plugins return a separate function
4
5   return function (tree, file) {
6
7     file.data.astro.frontmatter.customProperty = 'Generated property';
8
9   }
10
11 }
```

Tip

Added in: `astro@2.0.0`

`data.astro.frontmatter` contains all properties from a given Markdown or MDX document. This allows you to modify existing frontmatter properties, or compute new properties from this existing frontmatter.

2. Apply this plugin to your `markdown` or `mdx` integration config:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import { unified } from
  '@astrojs/markdown-remark';import { exampleRemarkPlugin } from './example-remark-
  plugin.mjs';
2 export default defineConfig({ markdown: { processor: unified({ remarkPlugins:
  [exampleRemarkPlugin] }) }, },});
```

or

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import mdx from '@astrojs/mdx';import {
  unified } from '@astrojs/markdown-remark';import { exampleRemarkPlugin } from
  './example-remark-plugin.mjs';
2 export default defineConfig({ integrations: [ mdx({ processor: unified({
  remarkPlugins: [exampleRemarkPlugin] }) }, ], },});
```

Now, every Markdown or MDX file will have `customProperty` in its frontmatter, making it available when [importing your markdown](#) and from [the `Astro.props.frontmatter` property in your layouts](#).



Related recipe: [Add reading time](#)

Extending Markdown config from MDX

Astro's MDX integration will extend [your project's existing Markdown configuration](#) by default, including the [Markdown processor](#). To override individual options, you can specify their equivalent in your MDX configuration.

The following example uses a different syntax highlighter and a different set of plugins for `.mdx` files than for `.md` files:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import { satteri } from '@astrojs/markdown-satteri';import mdx from '@astrojs/mdx';
2 export default defineConfig({ markdown: { syntaxHighlight: 'prism', processor: satteri({ mdastPlugins: [mdastPlugin1] }) }, integrations: [ mdx({ // `markdown.syntaxHighlight` gets overridden for `.mdx` files by this option syntaxHighlight: 'shiki',
3 // `markdown.processor` gets overridden for `.mdx` files with a different plugin processor: satteri({ mdastPlugins: [mdastPlugin2] }) }, ) ]});
```

To avoid extending your Markdown config from MDX, set [the `extendMarkdownConfig` option](#) (enabled by default) to `false`:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import { satteri } from '@astrojs/markdown-satteri';import mdx from '@astrojs/mdx';
2 export default defineConfig({ markdown: { processor: satteri({ mdastPlugins: [mdastPlugin] }) }, integrations: [ mdx({ // Markdown config now ignored extendMarkdownConfig: false, // Default `satteri()` processor used with no plugins }) ]});
```

Individual Markdown pages

Tip

[Content collections](#) and [importing Markdown into `.astro` components](#) provide more features for rendering your Markdown and are the recommended way to handle most of your content. However, there may be times when you want the convenience of just adding a file to `src/pages/` and having a simple page automatically created for you.

Astro treats [any supported file inside of the `/src/pages/` directory](#) as a page, including `.md` and other Markdown file types.

Placing a file in this directory, or any sub-directory, will automatically build a page route using the pathname of the file and display the Markdown content rendered to HTML. Astro will automatically add a `<meta charset="utf-8">` tag to your page to allow easier authoring of non-ASCII content.

src/pages/page-1.md

```

1 ---title: Hello, world---
2 # Hi there!
3 This Markdown file creates a page at `your-domain.com/page-1/`
4 It probably isn't styled much, but Markdown does support:- bold and italics._-
  lists- [links](https://astro.build)- <p>HTML elements</p>- and more!

```

Frontmatter `layout` property

To help with the limited functionality of individual Markdown pages, Astro provides a special frontmatter `layout` property which is a relative path to an Astro [Markdown layout component](#). `layout` is not a special property when using [content collections](#) to query and render your Markdown content, and is not guaranteed to be supported outside of its intended use case.

If your Markdown file is located within `src/pages/`, create a layout component and add it in this layout property to provide a page shell around your Markdown content.

`src/pages/posts/post-1.md`

```

1 ---
2
3 layout: ../../layouts/BlogPostLayout.astro
4
5 title: Astro in brief
6
7 author: Himanshu
8
9 description: Find out what makes Astro awesome!
10
11 ---
12
13 This is a post written in Markdown.

```

This layout component is a regular Astro component with [specific properties automatically available](#) through `Astro.props` for your Astro template. For example, you can access your Markdown file's frontmatter properties through `Astro.props.frontmatter`:

`src/layouts/BlogPostLayout.astro`

```

1 ---
2
3 const {frontmatter} = Astro.props;
4
5 ---
6
7 <html>
8
9   <head>
10
11     <!-- ... -->
12
13     <meta charset="utf-8"> // no longer added by default

```

```

14
15     </head>
16
17     <!-- ... -->
18
19     <h1>{frontmatter.title}</h1>
20
21     <h2>Post author: {frontmatter.author}</h2>
22
23     <p>{frontmatter.description}</p>
24
25     <slot /> <!-- Markdown content is injected here -->
26
27     <!-- ... -->
28
29 </html>

```

When using the frontmatter `layout` property, you must include the `<meta charset="utf-8">` tag in your layout as Astro will no longer add it automatically. You can now also [style your Markdown](#) in your layout component.

Learn more about [Markdown Layouts](#).

Fetching Remote Markdown

Astro's internal Markdown processor is not available for processing remote Markdown.

To fetch remote Markdown for use in [content collections](#), you can [build a custom loader](#) with access to a [renderMarkdown\(\) function](#).

To fetch remote Markdown directly and render it to HTML, you will need to install and configure your own Markdown parser from NPM. This will not inherit from any of Astro's built-in Markdown settings that you have configured.

Be sure that you understand these limitations before implementing this in your project, and consider fetching your remote Markdown using a content collections loader instead.

`src/pages/remote-example.astro`

```

1  ---
2
3  // Example: Fetch Markdown from a remote API
4
5  // and render it to HTML, at runtime.
6
7  // Using "marked" (https://github.com/markedjs/marked)
8
9  import { marked } from 'marked';
10
11  const response = await fetch('https://raw.githubusercontent.com/adam-p/markdown-
12  here/Markdown-Cheatsheet.md');
13
13  const markdown = await response.text();

```

```
14
15 const content = marked.parse(markdown);
16
17 ---
18
19 <article set:html={content} />
```

Content collections

Added in: `astro@2.0.0`

Content collections are the best way to manage sets of content in any Astro project: blog posts, product descriptions, character profiles, recipes, or any structured content. Collections help to organize and query your documents, enable Intellisense and type checking in your editor, and provide automatic TypeScript type-safety for all of your content.

Astro provides performant, scalable APIs to load, query, and render content from anywhere: stored locally in your project, hosted remotely, or fetched live from frequently-updating sources.

What are Content Collections?

A content collection is a set of related, structurally identical data. This data can be stored in one or several files locally (e.g. a folder of individual Markdown files of blog posts, a single JSON file of product descriptions) or fetched from remote sources such as a database, CMS, or API endpoint. Each member of the collection is called an entry.

Directorysrc/Directory**newsletter**/the “newsletter” collectionweek-1.mda collection entryweek-2.mda collection entryweek-3.mda collection entryDirectory**authors**/the “author” collectionauthors.jsona single file containing all collection entries

Collections are defined by the location and shape of its entries and provide a convenient way to query and render your content and associated metadata. You can create a collection any time you have a group of related data or content, stored in the same location, that shares a common structure.

[Two types of content collections](#) are available to allow you to work with data fetched either at build time or at request time. Both build-time collections and live updating collections use:

- A required `loader` to retrieve your content and metadata from wherever it is stored and make it available to your project through content-focused APIs.
- An optional collection `schema` that allows you to define the expected shape of each entry for type safety, autocomplete, and validation in your editor.

Collections stored locally in your project or on your filesystem can use one of Astro’s [provided build-time loaders](#) to fetch data from Markdown, MDX, Markdoc, YAML, TOML, or JSON files. Point Astro to the location of your content, define your data shape, and you’re good to go with a blog or similarly content-heavy, mostly static site in no time!

With [community-built loaders](#) or by building a [custom build-time collection loader](#) or [live loader](#) yourself, you can fetch remote data from any external source, such as a CMS, database, or headless payment system, either at build time or live on demand.

Types of collections

[Build-time content collections](#) are updated at build time, and data is saved to a storage layer. This provides excellent performance for most content, but may not be suitable for frequently updating data sources requiring up-to-the-moment data freshness, such as live stock prices.

For the best performance and scalability, use build-time content collections when one or more of these is true:

- **Performance is critical** and you want to prerender data at build time.
- **Your data is relatively static** (e.g., blog posts, documentation, product descriptions).
- **You want to benefit from build-time optimization** and caching.
- **You need to process MDX or perform image optimization.**
- **Your data can be fetched once and reused** across multiple builds.

Quick start

See [the official Astro blog starter template](#) to get up and running quickly with an example of using the [built-in glob\(\) loader](#) and [defining a schema](#) for a collection of local Markdown or MDX blog posts.

[Live content collections](#) fetch their data at runtime rather than build time. This allows you to access frequently updated data from CMSs, APIs, databases, or other sources using a unified API, without needing to rebuild your site when the data changes. However, this can come at a performance cost since data is fetched at each request and returned directly with no data store persistence.

Live content collections are designed for data that changes frequently and needs to be up-to-date when a page is requested. Consider using them when one or more of these is true:

- **You need real-time information** (e.g. user-specific data, current stock levels).
- **You want to avoid constant rebuilds** for content that changes often.
- **Your data updates frequently** (e.g. up-to-the-minute product inventory, prices, availability).
- **You need to pass dynamic filters** to your data source based on user input or request parameters.
- **You're building preview functionality** for a CMS where editors need to see draft content immediately.

Both kinds of collections can exist in the same project, so you can always choose the best type of collection for each individual data source. For example, a build-time collection can manage product descriptions, while a live collection can manage content inventory.

Both types of collections use similar APIs (e.g. `getCollection()` and `getLiveCollection()`), so that working with collections will feel familiar no matter which one you choose, while still ensuring that you always know which type of collection you are working with.

We suggest using build-time content collections whenever possible, and using live collections when your content needs updating in real time and the performance tradeoffs are acceptable. Additionally, live content collections have some limitations compared to build-time collections:

- **No MDX support:** MDX cannot be rendered at runtime
- **No image optimization:** Images cannot be processed at runtime
- **Performance considerations:** Data is fetched on each request (unless cached)
- **No data store persistence:** Data is not saved to the content layer data store

When to create a collection

Define your data as a collection when:

- You have multiple files or data to organize that share the same overall structure (e.g. a directory of blog posts written in Markdown which all have the same frontmatter properties).
- You have existing content stored remotely, such as in a CMS, and want to take advantage of the collections helper functions instead of using `fetch()` or SDKs.
- You need to fetch (tens of) thousands of related pieces of data at build time, and need a querying and caching method that handles at scale.

Much of the benefit of using collections comes from:

- Defining a common data shape to validate that an individual entry is “correct” or “complete”, avoiding errors in production.
- Content-focused APIs designed to make querying intuitive (e.g. `getCollection()` instead of `import.meta.glob()`) when importing and rendering content on your pages.
- Access to both built-in loaders and access to the low-level [Content Loader API](#) for retrieving your content. There are additionally several third-party and community-built loaders available, and you can build your own custom loader to fetch data from anywhere.
- Performance and scalability. Build-time content collections data can be cached between builds and is suitable for tens of thousands of content entries.

When not to create a collection

Collections provide excellent structure, safety, and organization when you have multiple pieces of content that must share the same properties.

Collections may not be your solution if:

- You have only one or a small number of different content pages. Consider [making individual page components](#) such as `src/pages/about.astro` with your content directly instead.
- You are displaying files that are not processed by Astro, such as PDFs. Place these static assets in the [public/ directory](#) of your project instead.
- Your data source has its own SDK/client library for imports that is incompatible with or does not offer a content loader, and you prefer to use it directly.

TypeScript configuration for collections

Content collections rely on TypeScript to provide Zod validation, Intellisense, and type checking in your editor. By default, Astro configures a [strict TypeScript template](#) when you create a new project using the `create astro` CLI command. Both of Astro’s `strict` and `strictest` templates include the TypeScript settings your project needs for content collections.

If you changed this setting to `base` because you are not writing TypeScript in your project, or are not using any of Astro’s built-in templates, you will need to also add the following `compilerOptions` in your `tsconfig.json` to use content collections:

`tsconfig.json`

```

1  {
2
3    "extends": "astro/tsconfigs/base",
4
5    // not needed for `strict` or `strictest`
6
7    "compilerOptions": {
8
9      "strictNullChecks": true,
10
11      "allowJs": true
12
13    }
14
15  }

```

Defining build-time content collections

All of your build-time content collections are defined in a special `src/content.config.ts` file (`.js` and `.mjs` extensions are also supported) using `defineCollection()`, and then a single collections object is exported for use in your project.

Each individual collection configures:

- [a build-time loader](#) for a data source (required)
- [a build-time schema](#) for type safety (optional, but highly recommended!)

`src/content.config.ts`

```

1  // 1. Import utilities from `astro:content`import { defineCollection } from
   'astro:content';
2  // 2. Import loader(s)import { glob, file } from 'astro/loaders';
3  // 3. Import Zodimport { z } from 'astro/zod';
4  // 4. Define a `loader` and `schema` for each collectionconst blog = defineCollection({
   loader: glob({ base: './src/content/blog', pattern: '**/*.md,mdx' }), schema:
   z.object({ title: z.string(), description: z.string(), pubDate:
   z.coerce.date(), updatedAt: z.coerce.date().optional(), }),});
5  // 5. Export a single `collections` object to register your collection(s)export const
   collections = { blog };

```

You can then use the dedicated `getCollection()` and `getEntry()` functions to [query your content collections data](#) and render your content.

You can choose to [generate page routes](#) from your build-time collection entries at build time for an entirely static, prerendered site. Or, you can render your build-time collections on demand, choosing to delay building your page until it is first requested. This is useful when you have a large number of pages (e.g. thousands or tens of thousands) and want to delay building a static page until it is needed.

Build-time collection loaders

Astro provides two built-in loaders (`glob()` and `file()`) for fetching your local content at build time. Pass the location of your data in your project or on your filesystem, and these loaders will automatically handle your data and update the persistent data store content layer.

To fetch remote data at build time, you can [build a custom loader](#) to retrieve your data and update the data store. Or, you can use any [third-party or community-published loader integration](#). Several already exist for popular content management systems as well as common data sources such as Obsidian vaults, GitHub repositories, or Bluesky posts.

The `glob()` loader

The [`glob\(\)` loader](#) fetches entries from directories of Markdown, MDX, Markdoc, JSON, YAML, or TOML files from anywhere on the filesystem. If you store your content entries locally as separate files, such as a directory of blog posts, then the `glob()` loader is all you need to access your content.

This loader requires a `pattern` of entry files to match using glob patterns supported by [micromatch](#), and a `base` file path of where your files are located. A unique `id` for each entry will be automatically generated from its file name, but you can [define custom IDs](#) if needed.

src/content.config.ts

```
1 import { defineCollection } from 'astro:content';import { glob } from 'astro/loaders';
2 const blog = defineCollection({ loader: glob({ pattern: "**/*.md", base:
  "./src/data/blog" }),});
3 export const collections = { blog };
```

Defining custom IDs

When using the [`glob\(\)` loader](#) with Markdown, MDX, Markdoc, JSON, or TOML files, every content entry `id` is automatically generated in an URL-friendly format based on the content filename. This unique `id` is used to query the entry directly from your collection. It is also useful when [creating new pages and URLs from your content](#).

You can override a single entry's generated `id` by adding your own `slug` property to the file frontmatter or data object for JSON files. This is similar to the “permalink” feature of other web frameworks.

src/blog/1.md

```
1 ---
2
3 title: My Blog Post
4
5 slug: my-custom-id/supports/slashes
6
7 ---
8
9 Your blog post content here.
```

src/categories/1.json

```

1 {
2
3   "title": "My Category",
4
5   "slug": "my-custom-id/supports/slashes",
6
7   "description": "Your category description here."
8
9 }

```

You can also pass options to the `glob()` loader's [generateID\(\) helper function](#) when you define your build-time collection to adjust how `ids` are generated. For example, you may wish to revert the default behavior of converting uppercase letters to lowercase for each collection entry:

`src/content.config.ts`

```

1 import { glob } from "astro/loaders"; import { defineCollection } from "astro:content";
2 const authors = defineCollection({ /* Retrieve all JSON files in your authors directory
   while retaining * uppercase letters in the ID. */ loader: glob({ pattern:
   "**/*.json", base: "./src/data/authors", generateId: ({ entry }) =>
   entry.replace(/\.json$/, ""), }, {})});

```

The `file()` loader

The [file\(\) loader](#) fetches multiple entries from a single local file defined in your collection. The `file()` loader will automatically detect and parse (based on the file extension) a single array of objects from JSON and YAML files, and will treat each top-level table as an independent entry in TOML files.

`src/content.config.ts`

```

1 import { defineCollection } from "astro:content"; import { file } from "astro/loaders";
2 const dogs = defineCollection({ loader: file("src/data/dogs.json"), });
3 export const collections = { dogs };

```

Each entry object in the file must have a unique `id` key property so that the entry can be identified and queried. Unlike the `glob()` loader, the `file()` loader will not automatically generate IDs for each entry.

You can provide your entries as an array of objects with an `id` property, or in object form where the unique `id` is the key:

`src/data/dogs.json`

```

1 // Specify an `id` property in each object of an array
2
3 [
4
5   { "id": "poodle", "coat": "curly", "shedding": "low" },
6
7   { "id": "afghan", "coat": "short", "shedding": "low" }
8
9 ]

```

src/data/dogs.json

```
1 // Each key will be used as the `id`
2
3 {
4
5   "poodle": { "coat": "curly", "shedding": "low" },
6
7   "afghan": { "coat": "silky", "shedding": "low" }
8
9 }
```

Parsing other data formats

Support for parsing single JSON, YAML, and TOML files into collection entries with the `file()` loader is built-in (unless you have a [nested JSON document](#)). To load your collection from unsupported file types, such as `.csv`, you will need to create a [parser function](#). This function can be made async if required (e.g. to fetch files from the web, or if your parser is asynchronous).

The following example shows importing a third-party CSV parser then passing a custom `parser` function to the `file()` loader:

src/content.config.ts

```
1 import { defineCollection } from "astro:content";import { file } from
  "astro/loaders";import { parse as parseCsv } from "csv-parse/sync";
2 const cats = defineCollection({ loader: file("src/data/cats.csv", { parser: (text)
  => parseCsv(text, { columns: true, skipEmptyLines: true }), })});
```

Nested `.json` documents

The `parser()` argument can be used to load a single collection from a nested JSON document. For example, this JSON file contains multiple collections:

src/data/pets.json

```
1 { "dogs": [{}], "cats": [{}]}
```

You can separate these collections by passing a custom `parser()` function to the `file()` loader for each collection, using Astro's built-in JSON parsing:

src/content.config.ts

```
1 import { file } from "astro/loaders";import { defineCollection } from "astro:content";
2 const dogs = defineCollection({ loader: file("src/data/pets.json", { parser: (text) =>
  JSON.parse(text).dogs })});const cats = defineCollection({ loader:
  file("src/data/pets.json", { parser: (text) => JSON.parse(text).cats })});
```

Custom build-time loaders

You can [build a custom loader](#) using the Content Loader API to fetch remote content from any data source, such as a CMS, a database, or an API endpoint.

Then you can import and define your custom loader in your collections config, passing any required values:

src/content.config.ts

```
1 import { defineCollection } from "astro:content";import { myLoader } from "../loader.ts";
2 const blog = defineCollection({ loader: myLoader({ url:
  "https://api.example.com/posts", apiKey: "my-secret", }),});
```

Tip

Find community-built and third-party loaders in the [Astro integrations directory](#).

Using a custom loader to fetch your data will automatically create a collection from your remote data. This gives you all the benefits of local collections, including collection-specific API helpers such as `getCollection()` and `render()` to [query and display your data](#), as well as schema validation.

Similar to creating an Astro integration or Vite plugin, you can [distribute your loader as an npm package](#) that others can use in their projects.

See the full [Content Loader API](#) for examples of how to build your own loader.

Defining the collection schema

Schemas enforce consistent frontmatter or entry data within a collection through Zod validation. A schema **guarantees** that this data exists in a predictable form when you need to reference or query it. If any file violates its collection schema, Astro will provide a helpful error to let you know.

Schemas also power Astro's automatic TypeScript typings for your content. When you define a schema for your collection, Astro will automatically generate and apply a TypeScript interface to it. The result is full TypeScript support when you query your collection, including property autocomplete and type-checking.

Tip

In order for Astro to recognize a new or updated schema, you may need to restart the dev server or [sync the content layer](#) (`s + enter`) to define the `astro:content` module.

Providing a `schema` is optional, but highly recommended! If you choose to use a schema, then every frontmatter or data property of your collection entries must be defined using a [Zod data type](#):

src/content.config.ts

```

1 import { defineCollection } from "astro:content";import { z } from "astro/zod";import {
  glob, file } from "astro/loaders";
2 const blog = defineCollection({ loader: glob({ pattern: "**/*.md", base:
  "./src/data/blog" }), schema: z.object({ title: z.string(), description:
  z.string(), pubDate: z.coerce.date(), updatedAt: z.coerce.date().optional(),
  })),});const dogs = defineCollection({ loader: file("src/data/dogs.json"), schema:
  z.object({ id: z.string(), breed: z.string(), temperament: z.array(z.string()),
  })),});
3 export const collections = { blog, dogs };

```

Defining datatypes with Zod

Astro uses [Zod](#) to power its content schemas. With Zod, Astro is able to validate every file's data within a collection *and* provide automatic TypeScript types when you query content from inside your project.

To use Zod in Astro, import the `z` utility from `"astro/zod"`. This is a re-export of the Zod library, and it supports all of the features of Zod 4.

See the [z utility reference](#) for a cheatsheet of common datatypes and to learn how Zod works and what features are available.

Zod schema methods

All [Zod schema methods](#) (e.g. `.parse()`, `.transform()`) are available, with some limitations. Notably, performing custom validation checks on images using `image().refine()` is unsupported.

Defining collection references

Collection entries can also “reference” other related entries.

With the [reference\(\) function](#), you can define a property in a collection schema as an entry from another collection. For example, you can require that every `space-shuttle` entry includes a `pilot` property which uses the `pilot` collection's own schema for type checking, autocomplete, and validation.

A common example is a blog post that references reusable author profiles stored as JSON, or related post URLs stored in the same collection:

src/content.config.ts

```

1 import { defineCollection, reference } from "astro:content";import { glob } from
  "astro/loaders";import { z } from "astro/zod";
2 const blog = defineCollection({ loader: glob({ base: "./src/content/blog", pattern:
  "**/*.md,mdx" }), schema: z.object({ title: z.string(), // Reference a single
  author from the `authors` collection by `id` author: reference("authors"), //
  Reference an array of related posts from the `blog` collection by `id` relatedPosts:
  z.array(reference("blog")), })),});
3 const authors = defineCollection({ loader: glob({ pattern: "**/*.json", base:
  "./src/data/authors" }), schema: z.object({ name: z.string(), portfolio: z.url(),
  })),});
4 export const collections = { blog, authors };

```

This example blog post specifies the `id`s of related posts and the `id` of the post author:

src/content/blog/welcome.md

```
1 ---
2
3 title: "welcome to my blog"
4
5 author: ben-holmes # references `src/data/authors/ben-holmes.json`
6
7 relatedPosts:
8
9 - about-me # references `src/content/blog/about-me.md`
10
11 - my-year-in-review # references `src/content/blog/my-year-in-review.md`
12
13 ---
```

These references will be transformed into objects containing a `collection` key and an `id` key, allowing you to easily [query them in your templates](#).

Querying build-time collections

Astro provides helper functions to query a build-time collection and return one or more content entries.

- `getCollection()` fetches an entire collection and returns an array of entries.
- `getEntry()` fetches a single entry from a collection.

These return entries with a unique `id`, a `data` object with all defined properties, and will also return a `body` containing the raw, uncompiled body of a Markdown, MDX, or Markdoc document.

src/pages/index.astro

```
1 ---import { getCollection, getEntry } from 'astro:content';
2 // Get all entries from a collection.// Requires the name of the collection as an
  argument.const allBlogPosts = await getCollection('blog');
3 // Get a single entry from a collection.// Requires the name of the collection and
  `id`const poodleData = await getEntry('dogs', 'poodle');---
```

The sort order of generated collections is non-deterministic and platform-dependent. This means that if you are calling `getCollection()` and need your entries returned in a specific order (e.g. blog posts sorted by date), you must sort the collection entries yourself:

src/pages/blog.astro

```
1 ---import { getCollection } from 'astro:content';
2 const posts = (await getCollection('blog')).sort( (a, b) => b.data.pubDate.valueOf() -
  a.data.pubDate.valueOf(),);---
```

See the full list of properties returned by the [CollectionEntry type](#).

Using content in Astro templates

After querying your collections, you can access each entry's content and metadata directly inside of your Astro component template.

For example, you can create a list of links to your blog posts, displaying information from your entry's frontmatter using the `data` property:

src/pages/index.astro

```
1  ---
2
3  import { getCollection } from 'astro:content';
4
5  const posts = await getCollection('blog');
6
7  ---
8
9  <h1>My posts</h1>
10
11 <ul>
12
13   {posts.map(post => (
14
15     <li><a href={` /blog/${post.id} `}>{post.data.title}</a></li>
16
17   ))}
18
19 </ul>
```

Rendering body content

Once queried, you can render Markdown and MDX entries to HTML using the [render\(\)](#) function from `astro:content`. Calling this function gives you access to rendered HTML content, including both a `<Content />` component and a list of all rendered headings.

src/pages/blog/post-1.astro

```
1  ---import { getEntry, render } from "astro:content";
2  const entry = await getEntry("blog", "post-1");
3  if (!entry) { throw new Error("Entry not found");}
4  const { Content } = await render(entry);---
5  <h1>{entry.data.title}</h1><p>Published on: {entry.data.pubDate.toDateString()}</p>
   <Content />
```

When working with MDX entries, you can also [pass your own components to ``](#) to replace HTML elements with custom alternatives.

Passing content as props

A component can also pass an entire collection entry as a prop.

You can use the `CollectionEntry` utility to correctly type your component's props using TypeScript. This utility takes a string argument that matches the name of your collection schema and will inherit all of the properties of that collection's schema.

src/components/BlogCard.astro

```
1  ---import type { CollectionEntry } from 'astro:content';interface Props {  post:
    CollectionEntry<'blog'>;}
2  // `post` will match your 'blog' collection schema typeconst { post } = Astro.props;---
```

Filtering collection queries

`getCollection()` takes an optional “filter” callback that allows you to filter your query based on an entry's `id` or `data` properties.

You can use this to filter by any content criteria you like. For example, you can filter by properties like `draft` to prevent any draft blog posts from publishing to your blog:

src/pages/blog.astro

```
1  ---
2
3  // Example: Filter out content entries with `draft: true`
4
5  import { getCollection } from 'astro:content';
6
7  const publishedBlogEntries = await getCollection('blog', ({ data }) => {
8
9      return data.draft !== true;
10
11  });
12
13  ---
```

You can also create draft pages that are available when running the dev server, but not built in production:

src/pages/blog.astro

```

1  ---
2
3  // Example: Filter out content entries with `draft: true` only when building for
  production
4
5  import { getCollection } from 'astro:content';
6
7  const blogEntries = await getCollection('blog', ({ data }) => {
8
9      return import.meta.env.PROD ? data.draft !== true : true;
10
11  });
12
13  ---

```

The filter argument also supports filtering by nested directories within a collection. Since the `id` includes the full nested path, you can filter by the start of each `id` to only return items from a specific nested directory:

`src/pages/blog.astro`

```

1  ---
2
3  // Example: Filter entries by sub-directory in the collection
4
5  import { getCollection } from 'astro:content';
6
7  const englishDocsEntries = await getCollection('docs', ({ id }) => {
8
9      return id.startsWith('en/');
10
11  });
12
13  ---

```

Accessing referenced data

To access [references defined in your schema](#), first query your collection entry. Your references will be available on the returned `data` object. (e.g. `entry.data.author` and `entry.data.relatedPosts`)

Then, you can use the `getEntry()` function again (or `getEntries()` to retrieve multiple referenced entries) by passing those returned values. The `reference()` function in your schema transforms those values into one or more `collection` and `id` objects as a convenient way to query this related data.

`src/pages/blog/adventures-in-space.astro`

```

1  ---import { getEntry, getEntries } from "astro:content";
2  // First, query a blog postconst blogPost = await getEntry("blog", "Adventures in
   Space");
3  // If the blog post doesn't exist, throw an errorif (!blogPost) { throw new Error("Blog
   post not found");}
4  // Retrieve a single reference item: the blog post's author// Equivalent to querying
   `{collection: "authors", id: "ben-holmes"}`const author = await
   getEntry(blogPost.data.author);
5  // Retrieve an array of referenced items: all the related posts// Equivalent to querying
   `[collection: "blog", id: "visiting-mars"], {collection: "blog", id: "leaving-earth-
   for-the-first-time"}`const relatedPosts = await
   getEntries(blogPost.data.relatedPosts);---
6  <h1>{blogPost.data.title}</h1><p>Author: {author.data.name}</p>
7  <!-- ... -->
8  <h2>You might also like:</h2>{relatedPosts.map((post) => <a href={post.id}>
   {post.data.title}</a>)}

```

Generating Routes from Content

Content collections are stored outside of the `src/pages/` directory. This means that no pages or routes are generated for your collection items by default by Astro's [file-based routing](#).

You will need to manually create a new [dynamic route](#) if you want to generate HTML pages for each of your collection entries, such as individual blog posts. Your dynamic route will map the incoming request param (e.g. `Astro.params.id` in `src/pages/blog/[...id].astro`) to fetch the correct entry for each page.

The exact method for generating routes will depend on whether your pages are prerendered (default) or rendered on demand by a server.

Building for static output (default)

If you are building a static website (Astro's default behavior) with build-time collections, use the [getStaticPaths\(\)](#) function to create multiple pages from a single page component (e.g. `src/pages/[id].astro`) during your build.

Call `getCollection()` inside of `getStaticPaths()` to have your collection data available for building static routes. Then, create the individual URL paths using the `id` property of each content entry. Each page receives the entire collection entry as a prop for [use in your page template](#).

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

```

1  ---
2
3  import { getCollection, render } from 'astro:content';
4
5  // 1. Generate a new path for every collection entry
6
7  export async function getStaticPaths() {
8
9      const posts = await getCollection('blog');
10
11     return posts.map(post => ({

```

```

12
13     params: { id: post.id },
14
15     props: { post },
16
17   }));
18
19 }
20
21 // 2. For your template, you can get the entry directly from the prop
22
23 const { post } = Astro.props;
24
25 const { Content } = await render(post);
26
27 ---
28
29 <h1>{post.data.title}</h1>
30
31 <Content />

```

This will generate a page route for every entry in the `blog` collection. For example, an entry at `src/blog/hello-world.md` will have an `id` of `hello-world`, and therefore its final URL will be `/posts/hello-world/`.

Note

If your custom slugs contain the `/` character to produce URLs with multiple path segments, you must use a [rest parameter (e.g. `[...id\]`)](<https://docs.astro.build/en/guides/routing/#rest-parameters>) in the `.astro` filename for this dynamic routing page.

Building routes on demand at request time

With an adapter installed for [on-demand rendering](#), you can generate your dynamic page routes at request time. First, examine the request (using `Astro.request` or `Astro.params`) to find the slug on demand, and then fetch it using one of Astro's content collection helper functions:

- `getEntry()` for build-time collection pages that are generated once, upon first request.
- `getLiveEntry()` for live collection pages where data is (re)etched at each request time.

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

```

1  ---export const prerender = false; // Not needed in 'server' mode
2  import { getEntry, render } from "astro:content";
3  // 1. Get the slug from the incoming server requestconst { id } = Astro.params;if (id
   === undefined) { return Astro.redirect("/404");}
4  // 2. Query for the entry directly using the request slugconst post = await
   getEntry("blog", id);
5  // 3. Redirect if the entry does not existif (post === undefined) { return
   Astro.redirect("/404");}
6  // 4. Render the entry to HTML in the templateconst { Content } = await render(post);---
   <h1>{post.data.title}</h1><Content />

```

Tip

Explore the `src/pages/` folder of the [blog tutorial demo code on GitHub](#) to see full examples of creating dynamic pages from your collections for blog features like a list of blog posts, tags pages, and more!

Live content collections

Live collections use a different API than build-time content collections, although the configuration and helper functions are designed to feel familiar.

Key differences include:

1. **Execution time:** Run at request time instead of build time
2. **Configuration file:** Use `src/live.config.ts` instead of `src/content.config.ts`
3. **Collection definition:** Use `defineLiveCollection()` instead of `defineCollection()`
4. **Loader API:** Implement `loadCollection` and `loadEntry` methods instead of the `load` method
5. **Data return:** Return data directly instead of storing in the data store
6. **User-facing functions:** Use `getLiveCollection()` / `getLiveEntry()` instead of `getCollection()` / `getEntry()`

Additionally, you must have an adapter configured for [on-demand rendering](#) of live collection data.

Define your live collections in the special file `src/live.config.ts` (separate from your `src/content.config.ts` for build-time collections, if you have one).

Each individual collection configures:

- a [live loader](#) for your data source, and optionally for type safety (required)
- a [live collection schema](#) for type safety (optional)

Unlike for build-time collections, there are no built-in live loaders available. You will need to [create a custom live loader](#) for your specific data source or find a third-party loader to pass to your live collection's `loader` property.

You can optionally [include type safety in your live loaders](#). Therefore, [defining a Zod schema](#) for live collections is optional. However, if you provide one, it will take precedence over the live loader's types.

`src/live.config.ts`

```
1 // Define live collections for accessing real-time data
import { defineLiveCollection } from 'astro:content';
import { storeLoader } from '@mystore/astro-loader';

2 const products = defineLiveCollection({ loader: storeLoader({ apiKey:
  process.env.STORE_API_KEY, endpoint: 'https://api.mystore.com/v1', }),});

3 // Export a single `collections` object to register your collection(s)
export const collections = { products };
```

You can then use the dedicated `getLiveCollection()` and `getLiveEntry()` functions to [access your live data](#) and render your content.

You can [generate page routes](#) from your live collection entries on demand, fetching your data fresh at runtime upon each request without needing a rebuild of your site like [build-time collections](#) do. This is useful when accessing live, up-to-the-moment data is more important than having your content available in a performant data storage layer that persists between site builds.

Creating a live loader

You can build a custom [live loader](#) using the Live Loader API to fetch remote content fresh upon request from any data source, such as a CMS, a database or an API endpoint. You will have to tell your live loader how to fetch and return content entries from your desired data source, as well as provide error handling for unsuccessful data requests.

Using a live loader to fetch your data will automatically create a collection from your remote data. This gives you all the benefits of Astro's content collections, including collection-specific API helpers such as `getLiveCollection()` and `render()` to [query and display your data](#), as well as helpful error handling.

Tip

Find community-built and third-party live loaders in the [Astro integrations directory](#).

See the basics of [building a live loader](#) using the Live Loader API

Using Zod schemas with live collections

You can use Zod schemas with live collections to validate and transform data at runtime. This Zod validation works the same way as [schemas for build-time collections](#).

When you define a schema for a live collection, it takes precedence over [the live loader's types](#) when you query the collection:

src/live.config.ts

```
1 import { defineLiveCollection } from 'astro:content';import { z } from
  'astro/zod';import { apiLoader } from './loaders/api-loader';
2 const products = defineLiveCollection({ loader: apiLoader({ endpoint:
  process.env.API_URL }), schema: z.object({ id: z.string(), name:
  z.string(), price: z.number(), // Transform the API's category format
  category: z.string().transform((str) => str.toLowerCase().replace(/\s+/g, '-')), //
  Coerce the date to a Date object createdAt: z.coerce.date(), })
  .transform((data) => ({ ...data, // Add a formatted price field
  displayPrice: `${data.price.toFixed(2)}`, })),});
3 export const collections = { products };
```

When using Zod schemas with live collections, validation errors are automatically caught and returned as `AstroError` objects:

src/pages/store/index.astro

```

1  ---export const prerender = false; // Not needed in 'server' mode
2  import { LiveCollectionValidationError } from 'astro/content';import {
  getLiveEntry } from 'astro:content';
3  const { entry, error } = await getLiveEntry('products', '123');
4  // You can handle validation errors specificallyif
  (LiveCollectionValidationError.is(error)) { console.error(error.message); return
  Astro.rewrite('/500');}
5  // TypeScript knows entry.data matches your Zod schema, not the loader's
  typeconsole.log(entry?.data.displayPrice); // e.g., "$29.99"---
```

See [Zod's README](#) for complete documentation on how Zod works and what features are available.

Accessing live data

Astro provides live collection helper functions to access live data on each request and return one (or more) content entries. These can be used similarly to their [build-time collection counterparts](#).

- `getLiveCollection()` fetches an entire collection and returns an array of entries.
- `getLiveEntry()` fetches a single entry from a collection.

These return entries with a unique `id`, and `data` object with all defined properties from the live loader. When using third-party or community loaders distributed as npm packages, check their own documentation for the expected shape of data returned.

You can use these functions to access your live data, passing the name of the collection and optionally filtering conditions.

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

```

1  ---export const prerender = false; // Not needed in 'server' mode
2  import { getLiveCollection, getLiveEntry } from "astro:content";
3  if (!Astro.params.slug) { return Astro.redirect('/404');}
4  // Use loader-specific filtersconst { entries: draftArticles } = await
  getLiveCollection("articles", { status: "draft", author: "john-doe",});
5  // Get a specific product by IDconst { entry: product } = await getLiveEntry("products",
  Astro.params.slug);---
```

Rendering content

If your live loader [returns a `rendered` property](#), you can use [the `render\(\)` function and ``component``](#) to render your content directly in your pages, using the same method as build-time collections.

You also have access to any [error returned by the live loader](#), for example, to rewrite to a 404 page when content cannot be displayed:

`src/pages/store/[id].astro`

```

1  ---export const prerender = false; // Not needed in 'server' mode
2  if (!Astro.params.id) { return Astro.redirect("/404");}
3  import { getLiveEntry, render } from "astro:content";const { entry, error } = await
  getLiveEntry("articles", Astro.params.id);if (!entry || error) { return
  Astro.redirect("/404");}
4  const { Content } = await render(entry);---
5  <h1>{entry.data.title}</h1><Content />

```

Error handling

Live loaders can fail due to network issues, API errors, or validation problems. The API is designed to make error handling explicit.

When you call `getLiveCollection()` or `getLiveEntry()`, the error will be one of:

- The error type defined by the loader (if it returned an error)
- A `LiveEntryNotFoundError` if the entry was not found
- A `LiveCollectionValidationError` if the collection data does not match the expected schema
- A `LiveCollectionCacheHintError` if the cache hint is invalid
- A `LiveCollectionError` for other errors, such as uncaught errors thrown in the loader

You can use `instanceof` to check the type of an error at runtime:

`src/pages/store/[id].astro`

```

1  ---export const prerender = false; // Not needed in 'server' mode
2  import { LiveEntryNotFoundError } from "astro/content/runtime";import { getLiveEntry }
  from "astro:content";
3  if (!Astro.params.id) { return Astro.redirect("/404");}
4  const { entry, error } = await getLiveEntry("products", Astro.params.id);
5  if (error) { if (error instanceof LiveEntryNotFoundError) { console.error(`Product
  not found: ${error.message}`); Astro.response.status = 404; } else {
  console.error(`Error loading product: ${error.message}`); return
  Astro.redirect("/500"); }}---

```

Caching live data

Added in: `astro@7.0.0` New

When live loaders provide [cache hints](#), `getLiveEntry()` and `getLiveCollection()` return a `cacheHint` object. This allows you to control the [caching behavior](#) of your routes without manually setting headers.

Cache hints include [tags](#) for targeted invalidation and [lastModified](#) to keep cached data fresh. You can also combine the cache hints with your own [cache options](#) to further customize caching behavior.

See the [Content Loader Reference](#) for more about implementing cache hints in your live loaders.

Entry-level cache hints

Pass the `cacheHint` returned by `getLiveEntry()` to `Astro.cache.set()` to apply the loader's recommended caching strategy.

The following example passes the loader's cache hint and adds a `maxAge` to control how long the response stays fresh:

`src/pages/products/[id].astro`

```
1  ---import { getLiveEntry } from 'astro:content';
2  const { entry, error, cacheHint } = await getLiveEntry('products', Astro.params.id);
3  if (error) { return Astro.redirect('/404');}
4  if (cacheHint) { Astro.cache.set(cacheHint);}
5  Astro.cache.set({ maxAge: 300 });---
6  <h1>{entry.data.name}</h1>
```

You can also pass a `LiveDataEntry` directly to let Astro extract its `cacheHint` automatically:

`src/pages/products/[id].astro`

```
1  ---import { getLiveEntry } from 'astro:content';
2  const { entry, error } = await getLiveEntry('products', Astro.params.id);
3  if (error) { return Astro.redirect('/404');}
4  Astro.cache.set(entry);Astro.cache.set({ maxAge: 300, swr: 60 });---
5  <h1>{entry.data.name}</h1>
```

Collection-level cache hints

When fetching a full collection with `getLiveCollection()`, Astro merges cache hints from the collection response and all individual entries: tags are accumulated, and the most recent `lastModified` wins.

The following example passes the merged cache hint from a collection and sets a 10-minute freshness window:

`src/pages/products/index.astro`

```
1  ---import { getLiveCollection } from 'astro:content';
2  const { entries, error, cacheHint } = await getLiveCollection('products');
3  if (error) { return new Response('Error loading products', { status: 500 });}
4  if (cacheHint) { Astro.cache.set(cacheHint);}Astro.cache.set({ maxAge: 600 });---
5  <ul> {entries.map((p) => <li>{p.data.name}</li>)}</ul>
```

Invalidating by entry

You can invalidate cached entries by passing a `LiveDataEntry` to `cache.invalidate()`. This allows you to target specific cached responses for invalidation based on the entry's cache tags, without needing to clear the entire cache.

The following example invalidates the cached response for a specific product entry:

`src/pages/api/revalidate.ts`

```
1 import { getLiveEntry } from 'astro:content';
2 export async function POST(context) { const { entry } = await getLiveEntry('products',
  'featured'); if (entry) { await context.cache.invalidate(entry); } return
  Response.json({ ok: true });}
```

Using JSON Schema files in your editor

Added in: `astro@4.13.0`

Astro auto-generates [JSON Schema](#) files for collections, which you can use in your editor to get IntelliSense and type-checking for data files.

A JSON Schema file is generated for each collection in your project and output to the `.astro/collections/` directory. For example, if you have two collections, one named `authors` and another named `posts`, Astro will generate `.astro/collections/authors.schema.json` and `.astro/collections/posts.schema.json`.

Use JSON Schemas in JSON files

You can manually point to an Astro-generated schema by setting the `$schema` field in your JSON file. The value should be a relative file path from the data file to the schema. In the following example, a data file in `src/data/authors/` uses the schema generated for the `authors` collection:

`src/data/authors/armand.json`

```
1 {
2
3   "$schema": "../../../.astro/collections/authors.schema.json",
4
5   "name": "Armand",
6
7   "skills": ["Astro", "Starlight"]
8
9 }
```

Use a schema for a group of JSON files in VS Code

In VS Code, you can configure a schema to apply to all files in a collection using the [json.schemas setting](#). In the following example, all files in the `src/data/authors/` directory will use the schema generated for the `authors` collection:

```
1 {
2
3   "json.schemas": [
4
5     {
6
7       "fileMatch": ["/src/data/authors/**"],
8
9       "url": "../../../.astro/collections/authors.schema.json"
10
11     }
12 ]
```

```
12
13   ]
14
15 }
```

Use schemas in YAML files in VS Code

In VS Code, you can add support for using JSON schemas in YAML files using the [Red Hat YAML](#) extension. With this extension installed, you can reference a schema in a YAML file using a special comment syntax:

src/data/authors/armand.yml

```
1 # yaml-language-server: $schema=../../.astro/collections/authors.schema.json
2
3 name: Armand
4
5 skills:
6
7   - Astro
8
9   - Starlight
```

Use schemas for a group of YAML files in VS Code

With the Red Hat YAML extension, you can configure a schema to apply to all YAML files in a collection using the `yaml.schemas` setting. In the following example, all YAML files in the `src/data/authors/` directory will use the schema generated for the `authors` collection:

```
1 {
2
3   "yaml.schemas": {
4
5     "../../.astro/collections/authors.schema.json": ["src/content/authors/*.yaml"]
6
7   }
8
9 }
```

See [“Associating schemas”](#) in the Red Hat YAML extension documentation for more details.

Images

Astro provides several ways for you to use images on your site, whether they are stored locally inside your project, linked to from an external URL, or managed in a CMS or CDN.

Astro provides built-in ```` and `__` Astro components, [Markdown image syntax](#) (`![]()`) processing, [SVG components](#), and [an image generating function](#) to optimize and/or transform your images. Additionally, you can configure [automatically resizing responsive images](#) by default, or set responsive properties on individual image and picture components.

You can always choose to use images and SVG files using native HTML elements in `.astro` or Markdown files, or the standard way for your file type (e.g. `<img />` in MDX and JSX). However, Astro does not perform any processing or optimization of these images.

There is also no native video support in Astro, and we recommend choosing a [hosted video service](#) to handle the demands of optimizing and streaming video content.

See the full API reference for the `Image` and `Picture` components.

Where to store images

`src/` vs `public/`

We recommend that local images are kept in `src/` when possible so that Astro can transform, optimize, and bundle them. Files in the `public/` directory are always served or copied into the build folder as-is, with no processing.

Your local images stored in `src/` can be used by all files in your project: `.astro`, `.md`, `.mdx`, `.mdoc`, and other UI frameworks as file imports. Images can be stored in any folder, including alongside your content.

Store your images in the `public/` folder if you want to avoid any processing. These images are available to your project files as URL paths on your domain and allow you to have a direct public link to them. For example, your site favicon will commonly be placed in the root of this folder where browsers can identify it.

Remote images

You can also choose to store your images remotely, in a [content management system \(CMS\)](#) or [digital asset management \(DAM\)](#) platform. Astro can fetch your data remotely using APIs or display images from their full URL path.

For extra protection when dealing with external sources, Astro's image components and helper function will only process (e.g. optimize, transform) images from [authorized image sources specified in your configuration](#). Remote images from other sources will be displayed with no processing.

Images in `.astro` files

Options: `<Image />`, `<Picture />`, `<img>`, `<svg>`, SVG components

Astro's templating language allows you to render optimized images with the Astro `Image` component and generate multiple sizes and formats with the Astro `Picture` component. Both components also accept [responsive image properties](#) for resizing based on container size and responding to device screen size and resolution.

Additionally, you can import and use [SVG files as Astro components](#) in `.astro` components.

All native HTML tags, including `<img>` and `<svg>`, are also available in `.astro` components. [Images rendered with HTML tags](#) will not be processed (e.g. optimized, transformed) and will be copied into your build folder as-is.

For all images in `.astro` files, **the value of the image `src` attribute is determined by the location of your image file:**

- A local image from your project `src/` folder uses an import from the file's relative path.

The image and picture components use the named import directly (e.g. `src={rocket}`), while the `<img>` tag uses the `src` object property of the import (e.g. `src={rocket.src}`).

- Remote and `public/` images use a URL path.

Provide a full URL for remote images (e.g. `src="https://www.example.com/images/my-remote-image.jpg"`), or a relative URL path on your site that corresponds to your file's location in your `public/` folder (e.g. `src="/images/my-public-image.jpg"` for an image located in `public/images/my-public-image.jpg`).

`src/pages/blog/my-images.astro`

```
1 ---import { Image } from 'astro:assets';import localBirdImage from
  '.././images/subfolder/localBirdImage.png';---<Image src={localBirdImage} alt="A bird
  sitting on a nest of eggs." /><Image src="/images/bird-in-public-folder.jpg" alt="A
  bird." width="50" height="50" /><Image src="https://example.com/remote-bird.jpg" alt="A
  bird." width="50" height="50" />
2 <img src={localBirdImage.src} alt="A bird sitting on a nest of eggs.">
```

See the full API reference for the `Image` and `Picture` components including required and optional properties.



Related recipe: [Dynamically import images](#)

Images in Markdown files

Options: `![]()`, `<img>` (with public or remote images)

Use standard Markdown `![alt](src)` syntax in your `.md` files. Your local images stored in `src/` and remote images will be processed and optimized. When you [configure responsive images globally](#), these images will also be [responsive](#).

Images stored in the `public/` folder are never optimized.

`src/pages/post-1.md`

```
1 # My Markdown Page
2 <!-- Local image stored in src/assets/ --><!-- Use a relative file path or import alias
  -->![A starry night sky.](../assets/stars.png)
3 <!-- Image stored in public/images/ --><!-- Use the file path relative to public/ -->![A
  starry night sky.](/images/stars.png)
4 <!-- Remote image on another server --><!-- Use the full URL of the image -->![Astro]
  (https://example.com/images/remote-image.png)
```

The HTML `<img>` tag can also be used to display images stored in `public/` or remote images without any image optimization or processing. However, `<img>` is not supported for your local images in `src`.

The `<Image />` and `<Picture />` components are unavailable in `.md` files. If you require more control over your image attributes, we recommend using [Astro's MDX integration](#) to add support for `.mdx` file format. MDX allows additional [image options available in MDX](#), including combining components with Markdown syntax.

Images in MDX files

Options: `<Image />`, `<Picture />`, `<img />`, `![]()`, SVG components

You can use Astro's `<Image />` and `<Picture />` components in your `.mdx` files by importing both the component and your image. Use them just as they are [used in .astro files](#). The JSX `<img />` tag is also supported for unprocessed images and [uses the same image import as the HTML `` tag](#).

Additionally, there is support for [standard Markdown `!\[alt\]\(src\)` syntax](#) with no import required.

src/pages/post-1.mdx

```
1  ---title: My Page title---import { Image } from 'astro:assets';import rocket from
   './assets/rocket.png';
2  # My MDX Page
3  // Local image stored in the same folder![Houston in the wild](houston.png)
4  // Local image stored in src/assets/<Image src={rocket} alt="A rocketship in space." />
   <img src={rocket.src} alt="A rocketship in space." />![A rocketship in space]
   (./assets/rocket.png)
5  // Image stored in public/images/<Image src="/images/stars.png" alt="A starry night
   sky." />![A starry night sky.]
   (/images/stars.png)
6  // Remote image on another server<Image src="https://example.com/images/remote-
   image.png" />![Astro]
   (https://example.com/images/remote-image.png)
```

See the full API reference for the ```` and ```` components.

Images in UI framework components

Image options: the framework's own image syntax (e.g. `<img />` in JSX, `<img>` in Svelte)

[Local images must first be imported](#) to access their image properties such as `src`. Then, they can be rendered as you normally would in that framework's own image syntax:

src/components/ReactImage.jsx

```
1  import stars from "../assets/stars.png";
2  export default function ReactImage() { return (    <img src={stars.src} alt="A starry
   night sky." />  )}
```

src/components/SvelteImage.svelte

```
1 <script> import stars from '../assets/stars.png';</script>
2 <img src={stars.src} alt="A starry night sky." />
```

Astro components (e.g. `<Image />`, `<Picture />`, SVG components) are unavailable inside UI framework components because [a client island must contain only valid code for its own framework](#).

But, you can pass the static content generated by these components to a framework component inside a `.astro` file [as children](#) or using a `named ```:

`src/components/ImageWrapper.astro`

```
1 ---import ReactComponent from './ReactComponent.jsx';import { Image } from
  'astro:assets';import stars from '~/stars/docline.png';---
2 <ReactComponent> <Image src={stars} alt="A starry night sky." /></ReactComponent>
```

Astro components for images

Astro provides two built-in Astro components for images (`<Image />` and `<Picture />`) and also allows you to import SVG files and use them as Astro components. These components may be used in any files that can import and render `.astro` components.

`<Image />`

Use the built-in `<Image />` Astro component to display optimized versions of:

- your local images located within the `src/` folder
- [configured remote images](#) from authorized sources

`<Image />` can transform a local or authorized remote image's dimensions, file type, and quality for control over your displayed image. This transformation happens at build time for prerendered pages. When your page is rendered on demand, this transformation will occur on the fly when the page is viewed. The resulting `<img>` tag includes `alt`, `loading`, and `decoding` attributes and infers image dimensions to avoid Cumulative Layout Shift (CLS).

What is Cumulative Layout Shift?

[Cumulative Layout Shift \(CLS\)](#) is a Core Web Vital metric for measuring how much content shifted on your page during loading. The `<Image />` component optimizes for CLS by automatically setting the correct `width` and `height` for your images.

`src/components/MyComponent.astro`

```

1  ---// import the Image component and the imageimport { Image } from
   'astro:assets';import myImage from '../assets/my_image.png'; // Image is 1600x900---
2  <!-- `alt` is mandatory on the Image component --><Image src={myImage} alt="A
   description of my image." />
3  <!-- Prerendered output --><!-- Image is optimized, proper attributes are enforced -->
   
4  <!-- Output rendered on demand--><!-- src will use an endpoint generated on demand-->
   />

```

The `<Image />` component accepts [several component properties](#) as well as any attributes accepted by the HTML `<img>` tag.

The following example provides a `class` to the image component which will apply to the final `<img>` element.

src/pages/index.astro

```

1  ---import { Image } from 'astro:assets';import myImage from '../assets/my_image.png';--
   -
2  <!-- `alt` is mandatory on the Image component --><Image src={myImage} alt=""
   class="my-class" />
3  <!-- Prerendered output -->
4
5  

```

Tip

You can also use the `<Image />` component for images in the `public/` folder, or remote images not specifically configured in your project, even though these images will not be optimized or processed. The resulting image will be the same as using the HTML `<img>`.

However, using the image component for all images provides a consistent authoring experience and prevents Cumulative Layout Shift (CLS) even for your unoptimized images.

<Picture />

Added in: astro@3.3.0

Use the built-in `<Picture />` Astro component to generate a `<picture>` tag with multiple formats and/or sizes of your image. This allows you to specify preferred file formats to display and at the same time, provide a fallback format. Like the [`component`](#), images will be processed at build time for prerendered pages. When your page is rendered on demand, processing will occur on the fly when the page is viewed.

The following example uses the `<Picture />` component to transform a local `.png` file into a web-friendly `avif` and `webp` format as well as the `.png` `<img>` that can be displayed as a fallback when needed:

src/pages/index.astro

```
1  ---import { Picture } from 'astro:assets';import myImage from '../assets/my_image.png';
   // Image is 1600x900---
2  <!-- `alt` is mandatory on the Picture component --><Picture src={myImage} formats=
   {['avif', 'webp']} alt="A description of my image." />
3  <!-- Prerendered output -->
4
5  <picture>
6
7     <source srcset="/_astro/my_image.hash.avif" type="image/avif" />
8
9     <source srcset="/_astro/my_image.hash.webp" type="image/webp" />
10
11    
26
27 </picture>
```

See details about [the `component`` properties]

(<https://docs.astro.build/en/reference/modules/astro-assets/#picture->) in the `astro:assets`` reference.

Responsive image behavior

Added in: `astro@5.10.0`

Responsive images are images that adjust to improve performance across different devices. These images can resize to fit their container, and can be served in different sizes depending on your visitor's screen size and resolution.

With the [layout property](#) applied to the `<Image />` or `<Picture />` components, Astro will automatically generate the required `srcset` and `sizes` values for your images, and apply the necessary [styles to ensure they resize correctly](#).

When this responsive behavior is [configured globally with `image.layout`](#), it will apply to all image components and also to any local and remote images using [the Markdown `!\[\]\(\)` syntax](#).

Images in your `public/` folder are never optimized, and responsive images are not supported.

Note

A single responsive image will generate multiple images of different sizes so that the browser can show the best one to your visitor.

For prerendered pages, this happens during the build and may increase the build time of your project, especially if you have a large number of images.

For pages rendered on-demand, the images are generated as-needed when a page is visited. This has no impact on build times but may increase the number of image transformations performed when an image is displayed. Depending on your image service this may incur additional costs.

Read more about [responsive images on MDN web docs](#).

Generated HTML output for responsive images

When a layout is set, either by default or on an individual component, images have automatically generated `srcset` and `sizes` attributes based on the image's dimensions and the layout type. Images with `constrained` and `full-width` layouts will have styles applied to ensure they resize according to their container.

`src/components/MyComponent.astro`

```
1 ---
2
3 import { Image } from 'astro:assets';
4
5 import myImage from '../assets/my_image.png';
6
7 ---
8
9 <Image src={myImage} alt="A description of my image." layout='constrained' width={800}
  height={600} />
```

This `<Image />` component will generate the following HTML output on a prerendered page:

```
1 

```

Responsive image styles

Setting `image.responsiveStyles: true` applies a small number of global styles to ensure that your images resize correctly. In most cases, you will want to enable these as a default; your images will not be responsive without additional styles.

However, if you prefer to handle responsive image styling yourself, or need to [override these defaults when using Tailwind 4](#), leave the default `false` value configured.

The global styles applied by Astro will depend on the layout type, and are designed to produce the best result for the generated `srcset` and `sizes` attributes. These are the default styles:

Responsive Image Styles

```

1  :where([data-astro-image]) {
2

```

```

3 |   object-fit: var(--fit);
4 |
5 |   object-position: var(--pos);
6 |
7 | }
8 |
9 | :where([data-astro-image='full-width']) {
10 |
11 |   width: 100%;
12 |
13 | }
14 |
15 | :where([data-astro-image='constrained']) {
16 |
17 |   max-width: 100%;
18 |
19 | }

```

The styles use the `:where()` [pseudo-class](#), which has a [specificity](#) of 0, meaning that it is easy to override with your own styles. Any CSS selector will have a higher specificity than `:where()`, so you can easily override the styles by adding your own styles to target the image.

You can override the `object-fit` and `object-position` styles on a per-image basis by setting the `fit` and `position` props on the `<Image />` or `<Picture />` component.

Responsive images with Tailwind 4

Tailwind 4 is compatible with Astro's default responsive styles. However, Tailwind uses [cascade layers](#), meaning that its rules are always lower specificity than rules that don't use layers, including Astro's responsive styles. Therefore, Astro's styling will take precedence over Tailwind styling. To use Tailwind rules instead of Astro's default styling, do not enable [Astro's default responsive styles](#).

SVG components

Added in: `astro@5.7.0`

Astro allows you to import SVG files and use them as Astro components. Astro will inline the SVG content into your HTML output.

Reference the default import of any local `.svg` file. Since this import is treated as an Astro component, you must use the same conventions (e.g. capitalization) as when [using dynamic tags](#).

`src/components/MyAstroComponent.astro`

```

1 | ---import Logo from './path/to/svg/file.svg';---
2 | <Logo />

```

Your SVG component, like `<Image />` or any other Astro component, is unavailable inside UI framework components, but can [be passed to a framework component](#) inside a `.astro` component.

SVG component attributes

You can pass props such as `width`, `height`, `fill`, `stroke`, and any other attribute accepted by the [native `element`](<https://developer.mozilla.org/en-US/docs/Web/SVG/Element/svg>). These attributes will

automatically be applied to the underlying `element`.

If a property is present in the original `.svg`` file and is passed to the component, the value passed to the component will override the original value.

`src/components/MyAstroComponent.astro`

```
1 ---import Logo from '../assets/logo.svg';---
2 <Logo width={64} height={64} fill="currentColor" />
```

SvgComponent Type

Added in: `astro@5.14.0`

You can also enforce type safety for your `.svg` assets using the `SvgComponent` type:

`src/components/Logo.astro`

```
1 ---import type { SvgComponent } from "astro/types";import HomeIcon from "../Home.svg";
2 interface Link { url: string; text: string; icon: SvgComponent;}
3 const links: Link[] = [ { url: "/", text: "Home", icon: HomeIcon, },];---
```

Creating custom image components

You can create a custom, reusable image component by wrapping the `<Image />` or `<Picture />` component in another Astro component. This allows you to set default attributes and styles only once.

For example, you could create a component for your blog post images that receives attributes as props and applies consistent styles to each image:

`src/components/BlogPostImage.astro`

```
1 ---import { Image } from "astro:assets";
2 const { alt, src, ...attrs } = Astro.props;---
3 <Image alt={alt} src={src} {...attrs} />
4 <style> img { margin-block: 2.5rem; border-radius: 0.75rem; }</style>
```

Display unprocessed images with the HTML `<img>` tag

The [Astro template syntax](#) also supports writing an `<img>` tag directly, with full control over its final output. These images will not be processed and optimized. It accepts all HTML `<img>` tag properties, and the only required property is `src`. However, it is strongly recommended to include [the `alt` property for accessibility](#).

images in `src/`

Local images must be imported from the relative path from the existing `.astro` file, or you can configure and use an [import alias](#). Then, you can access the image's `src` and other properties to use in the `<img>` tag.

Imported image assets match the [ImageMetadata type](#) and have the following signature:

```
1 interface ImageMetadata {
2
3   src: string;
4
5   width: number;
6
7   height: number;
8
9   format: string;
10
11 }
```

The following example uses the image's own `height` and `width` properties to avoid Cumulative Layout Shift (CLS) and improve Core Web Vitals:

`src/pages/posts/post-1.astro`

```
1 ---
2
3 // import local images
4
5 import myDog from '../images/pets/local-dog.jpg';
6
7 ---
8
9 // access the image properties
10
11 <img src={myDog.src} width={myDog.width} height={myDog.height} alt="A barking dog." />
```

Images in `public/`

For images located within `public/` use the image's file path relative to the public folder as the `src` value:

```
1 
```

Remote images

For remote images, use the image's full URL as the `src` value:

```
1 | 
```

Choosing `<Image />` vs `<img>`

The `<Image />` component optimizes your image and infers width and height (for images it can process) based on the original aspect ratio to avoid CLS. It is the preferred way to use images in `.astro` files whenever possible.

Use the HTML `<img>` element when you cannot use the `<Image />` component, for example:

- for unsupported image formats
- when you do not want your image optimized by Astro
- to access and change the `src` attribute dynamically client-side

Using Images from a CMS or CDN

Image CDNs work with [all Astro image options](#). Use an image's full URL as the `src` attribute in the `<Image />` component, an `<img>` tag, or in Markdown notation. For image optimization with remote images, also [configure your authorized domains or URL patterns](#).

Alternatively, the CDN may provide its own SDKs to more easily integrate in an Astro project. For example:

- Cloudinary supports an [Astro SDK](#) which allows you to easily drop in images with their `CldImage` component or a [Node.js SDK](#) that can generate URLs to use with an `<img>` tag in a Node.js environment.
- ImageKit provides an [Astro integration](#) that registers an image service. This means Astro's built-in `<Image />` and `<Picture />` components, along with Markdown `![]()` and MDX images, are all routed through ImageKit automatically. It adds CDN delivery, automatic AVIF/WebP conversion, responsive `srcset`, and on-the-fly transformations without changing your existing image syntax.

See the full API reference for the `__`` and `__`` components.

Authorizing remote images

You can configure lists of authorized image source URL domains and patterns for image optimization using [image.domains](#) and [image.remotePatterns](#). This configuration is an extra layer of safety to protect your site when showing images from an external source.

Remote images from other sources will not be optimized, but using the `<Image />` component for these images will prevent Cumulative Layout Shift (CLS).

For example, the following configuration will only allow remote images from `astro.build` to be optimized:

astro.config.mjs

```

1 export default defineConfig({
2
3   image: {
4
5     domains: ["astro.build"],
6
7   }
8
9 });

```

The following configuration will only allow remote images from HTTPS hosts:

astro.config.mjs

```

1 export default defineConfig({
2
3   image: {
4
5     remotePatterns: [{ protocol: "https" }],
6
7   }
8
9 });

```

Images in content collections

You can declare an associated image for a content collections entry, such as a blog post's cover image, in your frontmatter using its path relative to the current folder:

src/content/blog/my-post.md

```

1 ---title: "My first blog post"cover: "./firstpostcover.jpeg" # will resolve to
  "src/content/blog/firstpostcover.jpeg"coverAlt: "A photograph of a sunset behind a
  mountain range."---
2 This is a blog post

```

The `image` helper for the content collections schema lets you validate and import the image.

src/content.config.ts

```

1 import { defineCollection } from 'astro:content';import { z } from 'astro/zod';
2 const blogCollection = defineCollection({ schema: ({ image }) => z.object({ title:
  z.string(), cover: image(), coverAlt: z.string(), }),});
3 export const collections = { blog: blogCollection,};

```

The image will be imported and transformed into metadata, allowing you to pass it as a `src` to `<Image/>`, `<img>`, or `getImage()` in an Astro component.

The example below shows a blog index page that renders the cover photo and title of each blog post from the previous schema:

src/pages/blog.astro

```

1  ---import { Image } from "astro:assets";import { getCollection } from
   "astro:content";const allBlogPosts = await getCollection("blog");---
2  { allBlogPosts.map((post) => (    <div>        <Image src={post.data.cover} alt=
   {post.data.coverAlt} />        <h2>            <a href={" /blog/" + post.id}>{post.data.title}
   </a>            </h2>        </div>    ))}

```

Generating images with `getImage()`

The `getImage()` function is intended for generating images destined to be used somewhere else than directly in HTML, for example in an [API Route](#). When you need options that the `<Picture>` and `<Image>` components do not currently support, you can use the `getImage()` function to create your own custom `<Image />` component.

`getImage()` can only be used on the server. If you need to use the resulting image URL on the client (e.g. in a client-side script or framework component), call `getImage()` inside the frontmatter and pass the resulting `src` to the client:

src/components/ClientImage.astro

```

1  ---import { getImage } from "astro:assets";import myBackground from "../background.png";
2  const optimizedBackground = await getImage({ src: myBackground, format: "avif" });---
3  <div id="background" data-src={optimizedBackground.src}></div>
4  <script> const src = document.getElementById("background")?.dataset.src; // use src
   client-side as needed</script>

```

See more in the [getImage\(\) reference](#).



Related recipe: [Build a custom image component](#)

Alt Text

Not all users can see images in the same way, so accessibility is an especially important concern when using images. Use the `alt` attribute to provide [descriptive alt text](#) for images.

This attribute is required for both the `<Image />` and `<Picture />` components. If no alt text is provided, a helpful error message will be provided reminding you to include the `alt` attribute.

If the image is merely decorative (i.e. doesn't contribute to the understanding of the page), set `alt=""` so that screen readers know to ignore the image.

Default image service

[Sharp](#) is the default image service used for `astro:assets`. You can further configure the image service using the [image.service](#) option.

Note

When using a [strict package manager](#) like `pnpm`, you may need to manually install Sharp into your project even though it is an Astro dependency:

Terminal window

```
1 | npm add sharp
```

Configure no-op passthrough service

If your [adapter](#) does not support Astro's built-in Sharp image optimization (e.g. Cloudflare), you can configure a no-op image service to allow you to use the `<Image />` and `<Picture />` components. Note that Astro does not perform any image transformation and processing in these environments. However, you can still enjoy the other benefits of using `astro:assets`, including no Cumulative Layout Shift (CLS), the enforced `alt` attribute, and a consistent authoring experience.

Configure the `passthroughImageService()` to avoid Sharp image processing:

astro.config.mjs

```
1 | import { defineConfig, passthroughImageService } from 'astro/config';
2 | export default defineConfig({ image: { service: passthroughImageService() } });
```

Asset Caching

Astro stores processed image assets in a cache directory during site builds for both local and [remote images from authorized sources](#). By preserving the cache directory between builds, processed assets are reused, improving build time and bandwidth usage.

The default cache directory is `./node_modules/.astro`, however this can be changed using the [cacheDir](#) configuration setting.

Remote Images

Remote images in the asset cache are managed based on [HTTP Caching](#), and respect the [Cache-Control header](#) returned by the remote server. Images are cached if the Cache-Control header allows, and will be used until they are no longer [fresh](#).

Revalidation

Added in: `astro@5.1.0`

[Revalidation](#) reduces bandwidth usage and build time by checking with the remote server whether an expired cached image is still up-to-date. If the server indicates that the image is still fresh, the cached version is reused, otherwise the image is redownloaded.

Revalidation requires that the remote server send [Last-Modified](#) and/or [Etag \(entity tag\)](#) headers with its responses. This feature is available for remote servers that support the [If-Modified-Since](#) and [If-None-Match](#) headers.

Community Integrations

There are several third-party [community image integrations](#) for optimizing and working with images in your Astro project.

Data fetching

`.astro` files can fetch remote data to help you generate your pages.

`fetch()` in Astro

All [Astro components](#) have access to the [global `fetch\(\)` function](#) in their component script to make HTTP requests to APIs using the full URL (e.g. `https://example.com/api`). Additionally, you can construct a URL to your project's pages and endpoints that are rendered on demand on the server using [`new URL\("/api", Astro.url\)`](#).

This fetch call will be executed at build time, and the data will be available to the component template for generating dynamic HTML. If [SSR](#) mode is enabled, any fetch calls will be executed at runtime.

💡 Take advantage of [top-level `await`](#) inside of your Astro component script.

💡 Pass fetched data to both Astro and framework components, as props.

src/components/User.astro

```
1  ---import Contact from "../components/Contact.jsx";import Location from
   "../components/Location.astro";
2  const response = await fetch("https://randomuser.me/api/");const data = await
   response.json();const randomUser = data.results[0];---<!-- Data fetched at build can be
   rendered in HTML --><h1>User</h1><h2>{randomUser.name.first} {randomUser.name.last}</h2>
3  <!-- Data fetched at build can be passed to components as props --><Contact client:load
   email={randomUser.email} /><Location city={randomUser.location.city} />
```

Note

Remember, all data in Astro components is fetched when a component is rendered.

Your deployed Astro site will fetch data **once, at build time**. In dev, you will see data fetches on component refreshes. If you need to re-fetch data multiple times client-side, use a [framework component](#) or a [client-side script](#) in an Astro component.

`fetch()` in Framework Components

The `fetch()` function is also globally available to any [framework components](#):

src/components/Movies.tsx

```

1 import type { FunctionalComponent } from 'preact';
2 const data = await fetch('https://example.com/movies.json').then((response) =>
  response.json());
3 // Components that are build-time rendered also log to the CLI.// When rendered with a
  `client:*` directive, they also log to the browser console.console.log(data);
4 const Movies: FunctionalComponent = () => { // Output the result to the page return
  <div>{JSON.stringify(data)}</div>;};
5 export default Movies;

```

GraphQL queries

Astro can also use `fetch()` to query a GraphQL server with any valid GraphQL query.

`src/components/Film.astro`

```

1 ---const response = await fetch( "https://swapi-
  graphql.netlify.app/.netlify/functions/index", { method: "POST", headers: {
    "Content-Type": "application/json" }, body: JSON.stringify({ query: `
  query getFilm ($id:ID!) {          film(id: $id) {          title
  releaseDate          }          }          `, variables: {          id: "ZmlsbXM6MQ==",
  },    },    });
2
3 const json = await response.json();const { film } = json.data;---<h1>Fetching
  information about Star Wars: A New Hope</h1><h2>Title: {film.title}</h2><p>Year:
  {film.releaseDate}</p>

```

Fetch from a Headless CMS

Astro components can fetch data from your favorite CMS and then render it as your page content. Using [dynamic routes](#), components can even generate pages based on your CMS content.

See our [CMS Guides](#) for full details on integrating Astro with headless CMSes including Storyblok, Contentful, and WordPress.

Community resources

- [Creating a fullstack app with Astro + GraphQL](#)

On-demand rendering

Your Astro project code must be **rendered** to HTML in order to be displayed on the web.

By default, Astro pages, routes, and API endpoints will be pre-rendered at build time as static pages. However, you can choose to render some or all of your routes on demand by a server when a route is requested.

On-demand rendered pages and routes are generated per visit, and can be customized for each viewer. For example, a page rendered on demand can show a logged-in user their account information or display freshly updated data without requiring a full-site rebuild.

On-demand rendering on the server at request time is also known as **server-side rendering (SSR)**.

Server adapters

To render any page on demand, you need to add an **adapter**. Each adapter allows Astro to output a script that runs your project on a specific **runtime**: the environment that runs code on the server to generate pages when they are requested (e.g. Netlify, Cloudflare).

You may also wish to add an adapter even if your site is entirely static and you are not rendering any pages on demand. For example, the [Netlify adapter](#) enables Netlify's Image CDN, and [server islands](#) require an adapter installed to use `server:defer` on a component.

Adapters



- [@astrojs/cloudflare](#)



- [@astrojs/netlify](https://astrojs.netlify.com)



- [@astrojs/node](#)

- [@astrojs/vercel](#)

Astro maintains official adapters for [Node.js](#), [Netlify](#), [Vercel](#), and [Cloudflare](#). You can find both [official and community adapters in our integrations directory](#). Choose the one that corresponds to your [deployment environment](#).

Add an Adapter

You can add any of the [official adapter integrations maintained by Astro](#) with the following `astro add` command. This will install the adapter and make the appropriate changes to your `astro.config.mjs` file in one step.

For example, to install the Netlify adapter, run:

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npx astro add netlify
```

You can also [add an adapter manually by installing the NPM package](#) (e.g. `@astrojs/netlify`) and updating `astro.config.mjs` yourself.

Note that different adapters may have different configuration settings. Read each adapter's documentation, and apply any necessary config options to your chosen adapter in `astro.config.mjs`

Enabling on-demand rendering

By default, your entire Astro site will be prerendered, and static HTML pages will be sent to the browser. However, you may opt out of prerendering on any routes that require server rendering, for example, a page that checks for cookies and displays personalized content.

First, [add an adapter integration](#) for your server runtime to enable on-demand server rendering in your Astro project.

Then, add `export const prerender = false` at the top of the individual page or endpoint you want to render on demand. The rest of your site will remain a static site:

`src/pages/page-rendered-on-demand.astro`

```
1 | ---
2 |
3 | export const prerender = false
4 |
5 | ---
6 |
7 | <html>
8 |
9 | <!--
10 |
11 | This content will be server-rendered on demand!
12 |
13 | Just add an adapter integration for a server runtime!
14 |
15 | All other pages are statically-generated at build time!
```

```

16
17 -->
18
19 </html>

```

The following example shows opting out of prerendering in order to display a random number each time the endpoint is hit:

src/pages/randomnumber.js

```

1 export const prerender = false;
2 export async function GET() { let number = Math.random(); return new Response(
  JSON.stringify({ number, message: `Here's a random number: ${number}`, }),
  );}

```

'server' mode

For a **highly dynamic app**, after adding an adapter, you can [set your build output configuration to `output: 'server'`](#) to **server-render all your pages by default**. This is the equivalent of opting out of prerendering on every page.

Then, if needed, you can choose to prerender any individual pages that do not require a server to execute, such as a privacy policy or about page.

src/pages/about-my-app.astro

```

1 ---
2
3 export const prerender = true
4
5 ---
6
7 <html>
8
9 <!--
10
11 `output: 'server'` is configured, but this page is static!
12
13 The rest of my site is rendered on demand!
14
15 -->
16
17 </html>

```

Add `export const prerender = true` to any page or route to prerender a static page or endpoint:

src/pages/myendpoint.js

```

1 export const prerender = true;
2 export async function GET() { return new Response( JSON.stringify({ message:
  `This is my static endpoint`, }), );}

```

Tip

Start with the default `'static'` mode until you are sure that **most or all** of your pages will be rendered on demand! This ensures that your site is as performant as possible, not relying on a server function to render static content.

The `'server'` output mode does not bring any additional functionality. It only switches the default rendering behavior.

See more about the [output setting](#) in the configuration reference.

On-demand rendering features

HTML streaming

With HTML streaming, a document is broken up into chunks, sent over the network in order, and rendered on the page in that order. Astro uses HTML streaming in on-demand rendering to send each component to the browser as it renders them. This makes sure the user sees your HTML as fast as possible, although network conditions can cause large documents to be downloaded slowly, and waiting for data fetches can block page rendering.



Related recipe: [Using streaming to improve page performance](#)

Caution

Features that modify the [Response headers](#) are only available at the **page level**. (You can't use them inside of components, including layout components.) By the time Astro runs your component code, it has already sent the Response headers and they cannot be modified.

Cookies

A page or API endpoint rendered on demand can check, set, get, and delete cookies.

The example below updates the value of a cookie for a page view counter:

`src/pages/index.astro`

```
1 ---export const prerender = false; // Not needed in 'server' mode
2 let counter = 0
3 if (Astro.cookies.has('counter')) { const cookie = Astro.cookies.get('counter') const
  value = cookie?.number() if (value !== undefined && !isNaN(value)) counter = value + 1}
4 Astro.cookies.set('counter', String(counter))---<html> <h1>Counter = {counter}</h1>
  </html>
```

See more details about [Astro.cookies](#) and the [AstroCookie type](#) in the API reference.

Response

`Astro.response` is a standard `ResponseInit` object. It can be used to set the response status and headers.

The example below sets a response status and status text for a product page when the product does not exist:

`src/pages/product/[id].astro`

```
1 ---export const prerender = false; // Not needed in 'server' mode
2 import { getProduct } from '../api';
3 const product = await getProduct(Astro.params.id);
4 // No product foundif (!product) { Astro.response.status = 404;
  Astro.response.statusText = 'Not found';}---<html> <!-- Page here... --></html>
```

`Astro.response.headers`

You can set headers using the `Astro.response.headers` object:

`src/pages/index.astro`

```
1 ---export const prerender = false; // Not needed in 'server' mode
2 Astro.response.headers.set('Cache-Control', 'public, max-age=3600');---<html> <!-- Page
  here... --></html>
```

Return a `Response` object

You can also return a `Response` object directly from any page using on-demand rendering either manually or with `Astro.redirect`.

The example below looks up an ID in the database on a dynamic page and either it returns a 404 if the product does not exist, or it redirects the user to another page if the product is no longer available, or it displays the product:

`src/pages/product/[id].astro`

```
1 ---export const prerender = false; // Not needed in 'server' mode
2 import { getProduct } from '../api';
3 const product = await getProduct(Astro.params.id);
4 // No product foundif (!product) { return new Response(null, { status: 404,
  statusText: 'Not found' });}
5 // The product is no longer availableif (!product.isAvailable) { return
  Astro.redirect("/products", 301);}---<html> <!-- Page here... --></html>
```

Request

`Astro.request` is a standard `Request` object. It can be used to get the `url`, `headers`, `method`, and even the body of the request.

You can access additional information from this object for pages that are not statically generated.

Astro.request.headers

The headers for the request are available on `Astro.request.headers`. This works like the browser's [Request.headers](#). It is a [Headers](#) object where you can retrieve headers such as the cookie.

src/pages/index.astro

```
1 ---export const prerender = false; // Not needed in 'server' mode
2 const cookie = Astro.request.headers.get('cookie');// ...---<html> <!-- Page here... --></html>
```

Astro.request.method

The HTTP method used in the request is available as `Astro.request.method`. This works like the browser's [Request.method](#). It returns the string representation of the HTTP method used in the request.

src/pages/index.astro

```
1 ---export const prerender = false; // Not needed in 'server' mode
2 console.log(Astro.request.method) // GET (when navigated to in the browser)---
```

See more details about [Astro.request](#) in the API reference.

Server Endpoints

A server endpoint, also known as an **API route**, is a special function exported from a `.js` or `.ts` file within the `src/pages/` folder. A powerful feature of server-side rendering on demand, API routes are able to securely execute code on the server.

The function takes an [endpoint context](#) and returns a [Response](#).

To learn more, see our [Endpoints Guide](#).

Server islands

Server islands allow you to on-demand render dynamic or personalized “islands” individually, without sacrificing the performance of the rest of the page.

This means your visitor will see the most important parts of your page sooner, and allows your main content to be more aggressively cached, providing faster performance.

Server island components

A server island is a normal server-rendered [Astro component](#) that is instructed to delay rendering until its contents are available.

Your page will be rendered immediately with any specified [fallback content as a placeholder](#). Then, the component's own contents are fetched on the client and displayed when available.

With [an adapter installed](#) to perform the delayed rendering, add the [server:defer directive](#) to any component on your page to turn it into its own island:

src/pages/index.astro

```
1 ---
2
3 import Avatar from '../components/Avatar.astro';
4
5 ---
6
7 <Avatar server:defer />
```

These components can do [anything you normally would in an on-demand rendered page](#) using an adapter, such as fetch content, and access cookies:

src/components/Avatar.astro

```
1 ---
2
3 import { getUserAvatar } from '../sessions';
4
5 const userSession = Astro.cookies.get('session');
6
7 const avatarURL = await getUserAvatar(userSession);
8
9 ---
10
11 <img alt="User avatar" src={avatarURL} />
```

Passing props to server islands

Props provided to server island components must be [serializable](#): able to be translated into a format suitable for transfer over a network, or storage. Additionally, Astro does not serialize every type of serializable data structure. Therefore, there are some limitations on what can be passed as props to a server island.

Notably, functions cannot be passed to components marked with `server:defer` as they cannot be serialized. Objects with circular references are also not serializable.

The following prop types are supported: plain object, `number`, `string`, `Array`, `Map`, `Set`, `RegExp`, `Date`, `BigInt`, `URL`, `Uint8Array`, `Uint16Array`, `Uint32Array`, and `Infinity`

Server island fallback content

When using the `server:defer` attribute on a component to delay its rendering, you can “slot” in default loading content using the included named `"fallback"` slot.

Your fallback content will be rendered along with the rest of the page initially on page load and will be replaced with your component’s content when available.

To add fallback content, add `slot="fallback"` on a child (other components or HTML elements) passed to your server island component:

```
1 ---
2
3 import Avatar from '../components/Avatar.astro';
4
5 import GenericAvatar from '../components/GenericAvatar.astro';
6
7 ---
8
9 <Avatar server:defer>
10
11   <GenericAvatar slot="fallback" />
12
13 </Avatar>
```

This fallback content can be things like:

- A generic avatar instead of the user's own.
- Placeholder UI such as custom messages.
- Loading indicators such as spinners.

How it works

Server island implementation happens mostly at build-time where component content is swapped out for a small script.

Each of the islands marked with `server:defer` is split off into its own special route which the script fetches at run time. When Astro builds your site it will omit the component and inject a script in its place, and any content you've marked with `slot="fallback"`.

When the page loads in the browser, these components will be requested to a special endpoint that renders them and returns the HTML. This means that users will see the most critical parts of the page instantly. Fallback content will be visible for a short amount of time before the dynamic islands are then loaded.

Each island is loaded independently from the rest. This means a slower island won't delay the rest of your personalized content from being available.

This rendering pattern was built to be portable. It does not depend on any server infrastructure so it will work with any host you have, from a Node.js server in a Docker container to the serverless provider of your choice.

Caching

The data for server islands is retrieved via a `GET` request, passing props as an encrypted string in the URL query. This allows caching data with the [Cache-Control HTTP header](#) using standard `Cache-Control` directives.

However, [the browser limits URLs to a maximum length of 2048 bytes](#) for practical reasons and to avoid causing denial-of-service problems. If your query string causes your URL to exceed this limit, Astro will instead send a `POST` request that contains all props in the body.

`POST` requests are not cached by browsers because they are used to submit data, and could cause data integrity or security issues. Therefore, any existing caching logic in your project will break. Whenever possible, pass only necessary props to your server islands and avoid sending entire data objects and arrays to keep your query small.

Accessing the page URL in a server island

In most cases you, your server island component can get information about the page rendering it by [passing props](#) like in normal components.

However, server islands run in their own isolated context outside of the page request. `Astro.url` and `Astro.request.url` in a server island component both return a URL that looks like `/_server-islands/Avatar` instead of the current page's URL in the browser. Additionally, if you are prerendering the page you will not have access to information such as query parameters in order to pass as props.

To access information from the page's URL, you can check the [Referer](#) header, which will contain the address of the page that is loading the island in the browser:

```
1 ---const referer = Astro.request.headers.get("Referer");
2 if (!referer) { throw new Error("Referer header is missing");}
3 const url = new URL(referer);const productId = url.searchParams.get("product");---
```

Reusing the encryption key

Astro uses [cryptography](#) to encrypt props passed to server islands, protecting sensitive data from accidental exposure. This encryption relies on a new, random key that is generated on each build and embedded in the server bundle.

Most deploy hosts will handle keeping your front end and back end in sync automatically. However, you may need a constant encryption key if you are using rolling deployments, multi-region hosting or a CDN that caches pages containing server islands.

In environments with rolling deployments (e.g., Kubernetes) where your frontend assets (which encrypt props) and your backend functions (which decrypt props) may be temporarily using different keys, or when a CDN is still serving pages built with an old key, encrypted props passed to your server island cannot be decrypted.

In these situations, use the Astro CLI to generate a reusable, encoded encryption key to set as an environment variable in your build environment:

Terminal window

```
1 astro create-key
```

Use this value to configure the `ASTRO_KEY` environment variable (e.g. in a `.env` file) and include it in your CI/CD or host's build settings. This ensures the same key is always reused in the generated bundle so that encryption and decryption remain in sync.

Actions

Added in: `astro@4.15`

Astro Actions allow you to define and call backend functions with type-safety. Actions perform data fetching, JSON parsing, and input validation for you. This can greatly reduce the amount of boilerplate needed compared to using an [API endpoint](#).

Use actions instead of API endpoints for seamless communication between your client and server code and to:

- Automatically validate JSON and form data inputs using [Zod validation](#).
- Generate type-safe functions to call your backend from the client and even [from HTML form actions](#). No need for manual `fetch()` calls.
- Standardize backend errors with the [ActionError](#) object.

Basic usage

Actions are defined in a `server` object exported from `src/actions/index.ts`:

`src/actions/index.ts`

```
1 import { defineAction } from 'astro:actions';import { z } from 'astro/zod';
2 export const server = { myAction: defineAction({ /* ... */ })}
```

Your actions are available as functions from the `astro:actions` module. Import `actions` and call them client-side within a [UI framework component](#), [a form POST request](#), or by using a `<script>` tag in an Astro component.

When you call an action, it returns an object with either `data` containing the JSON-serialized result, or `error` containing thrown errors.

`src/pages/index.astro`

```
1 -----
2 <script>import { actions } from 'astro:actions';
3 async () => { const { data, error } = await actions.myAction({ /* ... */ });}</script>
```

Write your first action

Follow these steps to define an action and call it in a `script` tag in your Astro page.

1. Create a `src/actions/index.ts` file and export a `server` object.

`src/actions/index.ts`

```
1 export const server = {
2
3   // action declarations
4
5 }
```

2. Import the `defineAction()` utility from `astro:actions`, and the `z` object from `astro/zod`.

`src/actions/index.ts`

```

1 import { defineAction } from 'astro:actions';import { z } from 'astro/zod';
2 export const server = { // action declarations}

```

- Use the `defineAction()` utility to define a `getGreeting` action. The `input` property will be used to validate input parameters with a [Zod schema](#) and the `handler()` function includes the backend logic to run on the server.

`src/actions/index.ts`

```

1 import { defineAction } from 'astro:actions';import { z } from 'astro/zod';
2 export const server = { getGreeting: defineAction({ input: z.object({
  name: z.string(),  }), handler: async (input) => { return `Hello,
  ${input.name}!`  } })}

```

- Create an Astro component with a button that will fetch a greeting using your `getGreeting` action when clicked.

`src/pages/index.astro`

```

1 -----
2 <button>Get greeting</button>
3 <script>const button =
  document.querySelector('button');button?.addEventListener('click', async () => {
  // Show alert pop-up with greeting from action});</script>

```

- To use your action, import `actions` from `astro:actions` and then call `actions.getGreeting()` in the click handler. The `name` option will be sent to your action's `handler()` on the server and, if there are no errors, the result will be available as the `data` property.

`src/pages/index.astro`

```

1 -----
2 <button>Get greeting</button>
3 <script>import { actions } from 'astro:actions';
4 const button = document.querySelector('button');button?.addEventListener('click',
  async () => { // Show alert pop-up with greeting from action const { data, error
  } = await actions.getGreeting({ name: "Houston" }); if (!error) alert(data);}
  </script>

```

See the full Actions API documentation for details on [defineAction\(\)](#) and its properties.

Organizing actions

All actions in your project must be exported from the `server` object in the `src/actions/index.ts` file. You can define actions inline or you can move action definitions to separate files and import them. You can even group related functions in nested objects.

For example, to colocate all of your user actions, you can create a `src/actions/user.ts` file and nest the definitions of both `getUser` and `createUser` inside a single `user` object.

`src/actions/user.ts`

```
1 import { defineAction } from 'astro:actions';
2 export const user = { getUser: defineAction(/* ... */), createUser: defineAction(/*
... */), }
```

Then, you can import this `user` object into your `src/actions/index.ts` file and add it as a top-level key to the `server` object alongside any other actions:

`src/actions/index.ts`

```
1 import { user } from './user';
2 export const server = { myAction: defineAction({ /* ... */ }), user, }
```

Now, all of your user actions are callable from the `actions.user` object:

- `actions.user.getUser()`
- `actions.user.createUser()`

Handling returned data

Actions return an object containing either `data` with the type-safe return value of your `handler()`, or an `error` with any backend errors. Errors may come from validation errors on the `input` property or thrown errors within the `handler()`.

Actions return a custom data format that can handle Dates, Maps, Sets, and URLs [using the Devalue library](#). Therefore, you can't easily inspect the response from the network like you can with regular JSON. For debugging, you can instead inspect the `data` object returned by actions.

[See the `handler\(\)` API reference](#) for full details.

Checking for errors

It's best to check if an `error` is present before using the `data` property. This allows you to handle errors in advance and ensures `data` is defined without an `undefined` check.

```
1 const { data, error } = await actions.example();
2 if (error) { // handle error cases return; } // use `data`
```

Accessing `data` directly without an error check

To skip error handling, for example while prototyping or using a library that will catch errors for you, use the `.orThrow()` property on your action call to throw errors instead of returning an `error`. This will return the action's `data` directly.

This example calls a `likePost()` action that returns the updated number of likes as a `number` from the action `handler`:

```
1 const updatedLikes = await actions.likePost.orThrow({ postId: 'example' });
2
3 //    ^ type: number
```

Handling backend errors in your action

You can use the provided `ActionError` to throw an error from your action `handler()`, such as “not found” when a database entry is missing, or “unauthorized” when a user is not logged in. This has two main benefits over returning `undefined`:

- You can set a status code like `404 - Not found` or `401 - Unauthorized`. This improves debugging errors in both development and in production by letting you see the status code of each request.
- In your application code, all errors are passed to the `error` object on an action result. This avoids the need for `undefined` checks on data, and allows you to display targeted feedback to the user depending on what went wrong.

Creating an `ActionError`

To throw an error, import the `ActionError()` class from the `astro:actions` module. Pass it a human-readable status `code` (e.g. `"NOT_FOUND"` or `"BAD_REQUEST"`), and an optional `message` to provide further information about the error.

This example throws an error from a `likePost` action when a user is not logged in, after checking a hypothetical “user-session” cookie for authentication:

`src/actions/index.ts`

```
1 import { defineAction, ActionError } from "astro:actions";import { z } from "astro/zod";
2 export const server = { likePost: defineAction({ input: z.object({ postId:
  z.string() }), handler: async (input, ctx) => { if (!ctx.cookies.has('user-
  session')) { throw new ActionError({ code: "UNAUTHORIZED",
  message: "User must be logged in.", }); } // otherwise, like the post
}, });};
```

Handling an `ActionError`

To handle this error, you can call the action from your application and check whether an `error` property is present. This property will be of type `ActionError` and will contain your `code` and `message`.

In the following example, a `LikeButton.tsx` component calls the `likePost()` action when clicked. If an authentication error occurs, the `error.code` attribute is used to determine whether to display a login link:

`src/components/LikeButton.tsx`

```
1 import { actions } from 'astro:actions';import { useState } from 'preact/hooks';
2 export function LikeButton({ postId }: { postId: string }) { const [showLogin,
  setShowLogin] = useState(false); return ( <> { showLogin && <a
  href="/signin">Log in to like a post.</a> } <button onClick={async () => {
  const { data, error } = await actions.likePost({ postId }); if (error?.code
  === 'UNAUTHORIZED') setShowLogin(true); // Early return for unexpected errors
  else if (error) return; // update likes }> Like </button>
</> )}
```

Handling client redirects

When calling actions from the client, you can integrate with a client-side library like `react-router`, or you can use Astro's `navigate()` function to redirect to a new page when an action succeeds.

This example navigates to the homepage after a `logout` action returns successfully:

`src/pages/LogoutButton.tsx`

```
1 import { actions } from 'astro:actions';import { navigate } from
  'astro:transitions/client';
2 export function LogoutButton() { return ( <button onClick={async () => {      const
  { error } = await actions.logout();      if (!error) navigate('/');    }}>      Logout
  </button> );}
```

Accepting form data from an action

Actions accept JSON data by default. To accept form data from an HTML form, set `accept: 'form'` in your `defineAction()` call:

`src/actions/index.ts`

```
1 import { defineAction } from 'astro:actions';import { z } from 'astro/zod';
2 export const server = { comment: defineAction({      accept: 'form',      input:
  z.object(/* ... */),      handler: async (input) => { /* ... */ },    })}
```

Using validators with form inputs

When your action is [configured to accept form data](#), you can use any Zod validators to validate your fields (e.g. `z.coerce.date()` for date inputs). Extension functions including `.refine()`, `.transform()`, and `.pipe()` are also supported on the `z.object()` validator.

Additionally, Astro provides special handling under the hood for your convenience to validate the following types of field inputs:

- Inputs of type `number` can be validated using `z.number()`
- Inputs of type `checkbox` can be validated using `z.coerce.boolean()`
- Inputs of type `file` can be validated using `z.instanceof(File)`
- Multiple inputs of the same `name` can be validated using `z.array(/* validator */)`
- All other inputs can be validated using `z.string()`

When your form is submitted with empty inputs, the output type may not match your `input` validator. Empty values are converted to `null` except when validating arrays or booleans. For example, if an input of type `text` is submitted with an empty value, the result will be `null` instead of an empty string (`""`).

To apply a union of different validators, use the `z.discriminatedUnion()` wrapper to narrow the type based on a specific form field. This example accepts a form submission to either “create” or “update” a user, using the form field with the name `type` to determine which object to validate against:

`src/actions/index.ts`

```

1 import { defineAction } from 'astro:actions';import { z } from 'astro/zod';
2 export const server = {  changeUser: defineAction({    accept: 'form',    input:
  z.discriminatedUnion('type', [    z.object({      // Matches when the `type` field
has the value `create`      type: z.literal('create'),      name: z.string(),
email: z.email(),    }),    z.object({      // Matches when the `type` field has
the value `update`      type: z.literal('update'),      id: z.number(),      name:
z.string(),      email: z.email(),    }),    ]),    async handler(input) {      if
(input.type === 'create') {        // input is { type: 'create', name: string, email:
string }      } else {        // input is { type: 'update', id: number, name: string,
email: string }      }    },  }},);

```

Validating form data

Actions will parse submitted form data to an object, using the value of each input's `name` attribute as the object keys. For example, a form containing `<input name="search">` will be parsed to an object like `{ search: 'user input' }`. Your action's `input` schema will be used to validate this object.

To receive the raw `FormData` object in your action handler instead of a parsed object, omit the `input` property in your action definition.

The following example shows a validated newsletter registration form that accepts a user's email and requires a "terms of service" agreement checkbox.

1. Create an HTML form component with unique `name` attributes on each input:

`src/components/Newsletter.astro`

```

1 <form>
2
3   <label for="email">E-mail</label>
4
5   <input id="email" required type="email" name="email" />
6
7   <label>
8
9     <input required type="checkbox" name="terms">
10
11     I agree to the terms of service
12
13   </label>
14
15   <button>Sign up</button>
16
17 </form>

```

2. Define a `newsletter` action to handle the submitted form. Validate the `email` field using the `z.email()` validator, and the `terms` checkbox using `z.boolean()`:

`src/actions/index.ts`

```

1 import { defineAction } from 'astro:actions';import { z } from 'astro/zod';
2 export const server = { newsletter: defineAction({ accept: 'form', input:
  z.object({ email: z.email(), terms: z.boolean(), }), handler: async
  ({ email, terms }) => { /* ... */ }, })}

```

See the [input API reference](#) for all available form validators.

3. Add a `<script>` to the HTML form to submit the user input. This example overrides the form's default submit behavior to call `actions.newsletter()`, and redirects to `/confirmation` using the `navigate()` function:

src/components/Newsletter.astro

```

1 <form> <label for="email">E-mail</label> <input id="email" required type="email"
  name="email" /> <label> <input required type="checkbox" name="terms"> I
  agree to the terms of service </label> <button>Sign up</button></form>
2 <script> import { actions } from 'astro:actions'; import { navigate } from
  'astro:transitions/client';
3   const form = document.querySelector('form'); form?.addEventListener('submit',
  async (event) => {   event.preventDefault();   const formData = new
  FormData(form);   const { error } = await actions.newsletter(formData);   if
  (!error) navigate('/confirmation'); })</script>

```

See ["Call actions from an HTML form action"](#) for an alternative way to submit form data.

Displaying form input errors

You can validate form inputs before submission using [native HTML form validation attributes](#) like `required`, `type="email"`, and `pattern`. For more complex `input` validation on the backend, you can use the provided [isInputError\(\)](#) utility function.

To retrieve input errors, use the `isInputError()` utility to check whether an error was caused by invalid input. Input errors contain a `fields` object with messages for each input name that failed to validate. You can use these messages to prompt your user to correct their submission.

The following example checks the error with `isInputError()`, then checks whether the error is in the email field, before finally creating a message from the errors. You can use JavaScript DOM manipulation or your preferred UI framework to display this message to users.

```

1 import { actions, isInputError } from 'astro:actions';
2 const form = document.querySelector('form');const formData = new FormData(form);const {
  error } = await actions.newsletter(formData);if (isInputError(error)) { // Handle input
  errors. if (error.fields.email) {   const message = error.fields.email.join(', '); }}

```

Call actions from an HTML form action

Note

Pages must be on-demand rendered when calling actions using a form action. [Ensure prerendering is disabled on the page](#) before using this API.

You can enable zero-JS form submissions with standard attributes on any `<form>` element. Form submissions without client-side JavaScript may be useful both as a fallback for when JavaScript fails to load, or if you prefer to handle forms entirely from the server.

Calling `Astro.getActionResult()` on the server returns the result of your form submission (`data` or `error`), and can be used to dynamically redirect, handle form errors, update the UI, and more.

To call an action from an HTML form, add `method="POST"` to your `<form>`, then set the form's `action` attribute using your action, for example `action={actions.logout}`. This will set the `action` attribute to use a query string that is handled by the server automatically.

For example, this Astro component calls the `logout` action when the button is clicked and reloads the current page:

`src/components/LogoutButton.astro`

```
1 ---import { actions } from 'astro:actions';---
2 <form method="POST" action={actions.logout}> <button>Log out</button></form>
```

Additional attributes on the `<form>` element may be necessary for proper schema validation with Zod. For example, to include file uploads, add `enctype="multipart/form-data"` to ensure that files are sent in a format correctly recognized by `z.instanceof(File)`:

`src/components/FileUploadForm.astro`

```
1 ---
2
3 import { actions } from 'astro:actions';
4
5 ---
6
7 <form method="POST" action={actions.upload} enctype="multipart/form-data" >
8
9   <label for="file">Upload File</label>
10
11   <input type="file" id="file" name="file" />
12
13   <button type="submit">Submit</button>
14
15 </form>
```

Redirect on action success

If you need to redirect to a new route on success, you can use an action's result on the server. A common example is creating a product record and redirecting to the new product's page, e.g. `/products/[id]`.

For example, say you have a `createProduct` action that returns the generated product id:

`src/actions/index.ts`

```

1 import { defineAction } from 'astro:actions';import { z } from 'astro/zod';
2 export const server = { createProduct: defineAction({ accept: 'form', input:
  z.object({ /* ... */ }), handler: async (input) => { const product = await
  persistToDatabase(input); return { id: product.id }; }, })}

```

You can retrieve the action result from your Astro component by calling `Astro.getActionResult()`. This returns an object containing `data` or `error` properties when an action is called, or `undefined` if the action was not called during this request.

Use the `data` property to construct a URL to use with `Astro.redirect()`:

`src/pages/products/create.astro`

```

1 ---import { actions } from 'astro:actions';
2 const result = Astro.getActionResult(actions.createProduct);if (result && !result.error)
  { return Astro.redirect(`/products/${result.data.id}`);}---
3 <form method="POST" action={actions.createProduct}> <!--...--></form>

```

Handle form action errors

Calling `Astro.getActionResult()` in the Astro component containing your form gives you access to the `data` and `error` objects for custom error handling.

The following example displays a general failure message when a `newsletter` action fails:

`src/pages/index.astro`

```

1 ---import { actions } from 'astro:actions';
2 const result = Astro.getActionResult(actions.newsletter);---
3 {result?.error && ( <p class="error">Unable to sign up. Please try again later.</p>)}
  <form method="POST" action={actions.newsletter}> <label> E-mail <input required
  type="email" name="email" /> </label> <button>Sign up</button></form>

```

For more customization, you can [use the `isInputError\(\)` utility](#) to check whether an error is caused by invalid input.

The following example renders an error banner under the `email` input field when an invalid email is submitted:

`src/pages/index.astro`

```

1 ---import { actions, isInputError } from 'astro:actions';
2 const result = Astro.getActionResult(actions.newsletter);const inputErrors =
  isInputError(result?.error) ? result.error.fields : {};---
3 <form method="POST" action={actions.newsletter}> <label> E-mail <input required
  type="email" name="email" aria-describedby="error" /> </label> {inputErrors.email &&
  <p id="error">{inputErrors.email.join(',')}</p> <button>Sign up</button></form>

```

Preserve input values on error

Inputs will be cleared whenever a form is submitted. To persist input values, you can [enable view transitions](#) and apply the `transition:persist` directive to each input:

```
1 <input transition:persist required type="email" name="email" />
```

Update the UI with a form action result

To use an action's return value to display a notification to the user on success, pass the action to `Astro.getActionResult()`. Use the returned `data` property to render the UI you want to display.

This example uses the `productName` property returned by an `addToCart` action to show a success message.

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

```
1 ---import { actions } from 'astro:actions';
2 const result = Astro.getActionResult(actions.addToCart);---
3 {result && !result.error && ( <p class="success">Added {result.data.productName} to
  cart</p>)}
4 <!--...-->
```

Advanced: Persist action results with a session

Added in: `astro@5.0.0`

Action results are displayed as a POST submission. This means that the result will be reset to `undefined` when a user closes and revisits the page. The user will also see a “confirm form resubmission?” dialog if they attempt to refresh the page.

To customize this behavior, you can add middleware to handle the result of the action manually. You may choose to persist the action result using a cookie or session storage.

Start by [creating a middleware file](#) and importing [the `getActionContext\(\)` utility](#) from `astro:actions`. This function returns an `action` object with information about the incoming action request, including the action handler and whether the action was called from an HTML form. `getActionContext()` also returns the `setActionResult()` and `serializeActionResult()` functions to programmatically set the value returned by `Astro.getActionResult()`:

`src/middleware.ts`

```
1 import { defineMiddleware } from 'astro:middleware';import { getActionContext } from
  'astro:actions';
2 export const onRequest = defineMiddleware(async (context, next) => { const { action,
  setActionResult, serializeActionResult } = getActionContext(context); if
  (action?.calledFrom === 'form') { const result = await action.handler(); // ...
  handle the action result setActionResult(action.name, serializeActionResult(result));
  } return next();});
```

A common practice to persist HTML form results is the [POST / Redirect / GET pattern](#). This redirect removes the “confirm form resubmission?” dialog when the page is refreshed, and allows action results to be persisted throughout the user’s session.

This example applies the POST / Redirect / GET pattern to all form submissions using session storage with the [Netlify server adapter](#) installed. Action results are written to a session store using [Netlify Blob](#), and retrieved after a redirect using a session ID:

src/middleware.ts

```
1 import { defineMiddleware } from 'astro:middleware';import { getActionContext } from
  'astro:actions';import { randomUUID } from "node:crypto";import { getStore } from
  "@netlify/blobs";
2 export const onRequest = defineMiddleware(async (context, next) => { // Skip requests
  for prerendered pages if (context.isPrerendered) return next();
3   const { action, setActionResult, serializeActionResult } =
  getActionContext(context); // Create a Blob store to persist action results with
  Netlify Blob const actionStore = getStore("action-session");
4   // If an action result was forwarded as a cookie, set the result // to be accessible
  from `Astro.getActionResult()` const sessionId = context.cookies.get("action-session-
  id")?.value; const session = sessionId ? await actionStore.get(sessionId, {
  type: "json", }) : undefined;
5   if (session) { setActionResult(session.actionName, session.actionResult);
6     // Optional: delete the session after the page is rendered. // Feel free to
  implement your own persistence strategy await actionStore.delete(sessionId);
  context.cookies.delete("action-session-id"); return next(); }
7   // If an action was called from an HTML form action, // call the action handler and
  redirect to the destination page if (action?.calledFrom === "form") { const
  actionResult = await action.handler();
8     // Persist the action result using session storage const sessionId =
  randomUUID(); await actionStore.setJSON(sessionId, { actionName: action.name,
  actionResult: serializeActionResult(actionResult), });
9     // Pass the session ID as a cookie // to be retrieved after redirecting to the
  page context.cookies.set("action-session-id", sessionId);
10    // Redirect back to the previous page on error if (actionResult.error) {
  const referer = context.request.headers.get("Referer"); if (!referer) {
  throw new Error("Internal: Referer unexpectedly missing from Action POST
  request.", ); } return context.redirect(referer); } // Redirect
  to the destination page on success return context.redirect(context.originPathname);
  }
11  return next();});
```

Security when using actions

Actions are accessible as public endpoints based on the name of the action. For example, the action `blog.like()` will be accessible from `/_actions/blog.like`. This is useful for unit testing action results and debugging production errors. However, this means you **must** use same authorization checks that you would consider for API endpoints and on-demand rendered pages.

Authorize users from an action handler

To authorize action requests, add an authentication check to your action handler. You may want to use [an authentication library](#) to handle session management and user information.

Actions expose [a subset of the `APIContext` object](#) to access properties passed from middleware using `context.locals`. When a user is not authorized, you can raise an `ActionError` with the `UNAUTHORIZED` code:

src/actions/index.ts

```
1 import { defineAction, ActionError } from 'astro:actions';
2 export const server = { getUserSettings: defineAction({ handler: async (_input,
  context) => { if (!context.locals.user) { throw new ActionError({ code:
    'UNAUTHORIZED' }); } return { /* data on success */ }; }) }
```

Gate actions from middleware

Added in: astro@5.0.0

Astro recommends authorizing user sessions from your action handler to respect permission levels and rate-limiting on a per-action basis. However, you can also gate requests to all actions (or a subset of actions) from middleware.

Use the [getActionContext\(\) function](#) from your middleware to retrieve information about any inbound action requests. This includes the action name and whether that action was called using a client-side remote procedure call (RPC) function (e.g. `actions.blog.like()`) or an HTML form.

The following example rejects all action requests that do not have a valid session token. If the check fails, a “Forbidden” response is returned. Note: this method ensures that actions are only accessible when a session is present, but is *not* a substitute for secure authorization.

src/middleware.ts

```
1 import { defineMiddleware } from "astro:middleware";import { getActionContext } from
  "astro:actions";
2 export const onRequest = defineMiddleware(async (context, next) => { const { action } =
  getActionContext(context); // Check if the action was called from a client-side
  function if (action?.calledFrom === "rpc") { // If so, check for a user session
    token if (!context.cookies.has("user-session")) { return new
    Response("Forbidden", { status: 403 }); } }
3 context.cookies.set("user-session", "session-token-value"); return next();});
```

Call actions from Astro components and server endpoints

You can call actions directly from Astro component scripts using the `Astro.callAction()` wrapper (or `context.callAction()` when using a [server endpoint](#)). This is common to reuse logic from your actions in other server code.

Pass the action as the first argument and any input parameters as the second argument. This returns the same `data` and `error` objects you receive when calling actions on the client:

src/pages/products.astro

```

1  ---import { actions } from 'astro:actions';
2  const searchQuery = Astro.url.searchParams.get('search');if (searchQuery) { const {
  data, error } = await Astro.callAction(actions.findProduct, { query: searchQuery }); //
  handle result}---
```

Sessions

Added in: `astro@5.7.0`

Sessions are used to share data between requests for [on-demand rendered pages](#).

Unlike [cookies](#), sessions are stored on the server, so you can store larger amounts of data without worrying about size limits or security issues. They are useful for storing things like user data, shopping carts, and form state, and they work without any client-side JavaScript:

`src/components/CartButton.astro`

```

1  ---export const prerender = false; // Not needed with 'server' outputconst cart = await
  Astro.session?.get('cart');---
2  <a href="/checkout"><img alt="shopping cart icon" data-bbox="305 405 320 415" style="vertical-align: middle;"/> {cart?.length ?? 0} items</a>
```

Configuring sessions

Sessions require a storage driver to store the session data. The [Node](#), [Cloudflare](#), and [Netlify](#) adapters automatically configure a default driver for you, but other adapters currently require you to [specify a driver manually](#).

`astro.config.mjs`

```

1  import { defineConfig, sessionDrivers } from 'astro/config'import vercel from
  '@astrojs/vercel'
2  export default defineConfig({ adapter: vercel() session: { driver:
  sessionDrivers.lruCache({ max: 800, }), })})
```

See [the session configuration option](#) for more details on setting a storage driver, and other configurable options.

Overriding the configuration at runtime

By default, session drivers are configured at build time, and any [environment variables](#) used will be inlined into the build. This means you cannot override the configuration at runtime.

When you need a different configuration (e.g. to connect to an external service), define it in a separate file. Then use that file as the [driver's entrypoint](#).

The following example takes advantage of [Unstorage compatibility](#) to configure the Redis driver in its own entrypoint:

1. Install the [unstorage package](#):

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npm install unstorage
```

2. Create a file for the driver configuration (e.g. `src/session-driver.ts`) and export a default function that returns the driver instance:

`src/session-driver.ts`

```
1 | import type { SessionDriver } from "astro";import redisDriver from
   | "unstorage/drivers/redis";import { REDIS_HOST, REDIS_PORT } from "astro:env";
2 | export default function (): SessionDriver { return redisDriver({ host:
   | REDIS_HOST, port: REDIS_PORT, });}
```

3. Use this file as the driver's entrypoint in your Astro configuration:

`astro.config.mjs`

```
1 | import { defineConfig, envField, sessionDrivers } from "astro/config";import vercel
   | from "@astrojs/vercel";
2 | export default defineConfig({ adapter: vercel(), env: { REDIS_HOST:
   | envField.string({ context: "server", access: "public", default: "localhost" }),
   | REDIS_PORT: envField.number({ context: "server", access: "public", default: 6379
   | }), }, session: { driver: { entrypoint: new URL('./src/session-
   | driver.ts', import.meta.url), } }});
```

Interacting with session data

The [session object](#) allows you to interact with the stored user state (e.g. adding items to a shopping cart) and the session ID (e.g. deleting the session ID cookie when logging out). The object is accessible as `Astro.session` in your Astro components and pages and as `context.session` object in API endpoints, middleware, and actions.

The session is generated automatically when it is first used and can be regenerated at any time with [session.regenerate\(\)](#) or destroyed with [session.destroy\(\)](#).

For many use cases, you will only need to use [session.get\(\)](#) and [session.set\(\)](#).

See [the Sessions API reference](#) for more details.

Astro components and pages

In `.astro` components and pages, you can access the session object via the global `Astro` object. For example, to display the number of items in a shopping cart:

`src/components/CartButton.astro`

```
1  ---export const prerender = false; // Not needed with 'server' outputconst cart = await
    Astro.session?.get('cart');---
2  <a href="/checkout">🛒 {cart?.length ?? 0} items</a>
```

API endpoints

In API endpoints, the session object is available on the `context` object. For example, to add an item to a shopping cart:

`src/pages/api/addToCart.ts`

```
1  import type { APIContext } from "astro";
2  export async function POST(context: APIContext) {   const cart = await
    context.session?.get('cart') || [];   const data = await context.request.json();
    if(!data?.item) {     return new Response('Item is required', { status: 400 });   }
    cart.push(data.item);   await context.session?.set('cart', cart);   return
    Response.json(cart);}
```

Actions

In actions, the session object is available on the `context` object. For example, to add an item to a shopping cart:

`src/actions/addToCart.ts`

```
1  import { defineAction } from 'astro:actions';import { z } from 'astro/zod';
2  export const server = {   addToCart: defineAction({     input: z.object({ productId:
    z.string() }),     handler: async (input, context) => {       const cart = await
    context.session?.get('cart');       cart.push(input.productId);       await
    context.session?.set('cart', cart);       return cart;     },   }},);
```

Middleware

Note

Sessions are not supported in edge middleware.

In middleware, the session object is available on the `context` object. For example, to set the last visit time in the session:

`src/middleware.ts`

```
1 import { defineMiddleware } from 'astro:middleware';
2 export const onRequest = defineMiddleware(async (context, next) => {
  context.session?.set('lastvisit', new Date()); return next();});
```

Session data types

By default session data is untyped, and you can store arbitrary data in any key. Values are serialized and deserialized using [devalue](#), which is the same library used in content collections and actions. This means that supported types are the same, and include strings, numbers, `Date`, `Map`, `Set`, `URL`, arrays, and plain objects.

You can optionally [define TypeScript types](#) for your session data by creating a `src/env.d.ts` file and adding a declaration for the `App.SessionData` type:

`src/env.d.ts`

```
1 declare namespace App {
2
3   interface SessionData {
4
5     user: {
6
7       id: string;
8
9       name: string;
10
11     };
12
13     cart: string[];
14
15   }
16
17 }
```

This will allow you to access the session data with type-checking and auto-completion in your editor:

`src/components/CartButton.astro`

```
1 ---const cart = await Astro.session?.get('cart');// const cart: string[] | undefined
2 const something = await Astro.session?.get('something');// const something: any
3 Astro.session?.set('user', { id: 1, name: 'Houston' });// Error: Argument of type '{ id:
  number; name: string }' is not assignable to parameter of type '{ id: string; name:
  string; }'.---
```

Caution

This is only used for type-checking and does not affect the runtime behavior of the session. Take extra care if you change the type when users have stored data in the session, as this could cause runtime errors.

Route caching

Added in: `astro@7.0.0` New

Astro provides a platform-agnostic API for caching responses from [on-demand rendered](#) pages and endpoints. Cache directives set in your routes are translated into the appropriate headers or runtime behavior depending on your configured cache provider.

Route caching builds on standard [HTTP caching semantics](#), including `max-age` and `stale-while-revalidate`, with support for tag-based and path-based invalidation, config-level route rules, and pluggable cache providers that adapters can set automatically.

Configure caching

Route caching requires a cache provider to determine how caching is implemented at runtime. A [built-in in-memory provider](#) is available, and custom providers can be implemented for advanced use cases and specific runtimes.

To enable this feature, set a [cache provider](#) in your Astro config:

`astro.config.mjs`

```
1 import { defineConfig, memoryCache } from 'astro/config';import node from
  '@astrojs/node';
2 export default defineConfig({ adapter: node({ mode: 'standalone' }), cache: {
  provider: memoryCache(), },});
```

You can then use `Astro.cache` in your `.astro` pages (or `context.cache` for API routes and middleware) to control caching per request. Cache defaults for groups of routes can also be defined declaratively in your config using [routeRules](#).

If you deploy to Netlify, Vercel, or Cloudflare, you can use their respective adapters' experimental [CDN cache provider](#) instead of the in-memory provider.

Adapter cache providers

Astro's first-party adapters for Netlify, Vercel, and Cloudflare each provide an experimental CDN cache provider that maps cache directives to the platform's native cache headers and invalidation API. Rather than storing responses in memory, these push your caching directives to the host's edge network, and cache hits are served straight from the CDN without invoking your server function.

During the experimental phase these providers need to be manually enabled, as shown below. In a future release, the Netlify and Vercel providers will be enabled automatically by their adapters. The Cloudflare provider requires a feature that is currently in private beta.

Each provider automatically tags cached responses with the request path, so `cache.invalidate({ path })` works on platforms that only support tag-based purges.

Netlify

Added in: `@astrojs/netlify@8.0.0` New

Import `cacheNetlify()` from `@astrojs/netlify/cache` and set it as your cache provider:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import netlify from
  '@astrojs/netlify';import { cacheNetlify } from '@astrojs/netlify/cache';
2 export default defineConfig({ adapter: netlify(), cache: { provider:
  cacheNetlify(), },});
```

The provider sets `Netlify-CDN-Cache-Control` and `Netlify-Cache-Tag` headers. Cached responses use Netlify's [durable cache](#) so they are shared across all edge nodes, reducing function invocations. Both tag-based and path-based invalidation are supported.

Vercel

Added in: `@astrojs/vercel@11.0.0` New

Import `cacheVercel()` from `@astrojs/vercel/cache` and set it as your cache provider:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import vercel from '@astrojs/vercel';import
  { cacheVercel } from '@astrojs/vercel/cache';
2 export default defineConfig({ adapter: vercel(), cache: { provider: cacheVercel(),
  },});
```

The provider sets `vercel-CDN-Cache-Control` and `vercel-Cache-Tag` headers. Both tag-based and path-based invalidation are supported. Tag invalidation is a soft invalidation: cached responses are marked as stale and revalidated in the background using stale-while-revalidate.

Cloudflare

Added in: `@astrojs/cloudflare@14.0.0` New

Private beta

The Cloudflare provider requires the Cloudflare Workers Cache feature, which is currently in private beta. Without beta access, the headers are ignored and calling `cache.invalidate()` will throw an error.

Import `cacheCloudflare()` from `@astrojs/cloudflare/cache` and set it as your cache provider:

astro.config.mjs

```
1 import { defineConfig } from 'astro/config';import cloudflare from
  '@astrojs/cloudflare';import { cacheCloudflare } from '@astrojs/cloudflare/cache';
2 export default defineConfig({ adapter: cloudflare(), cache: { provider:
  cacheCloudflare(), },});
```

The provider sets `Cloudflare-CDN-Cache-Control` and `Cache-Tag` headers. The adapter enables Worker caching automatically when a Cloudflare cache provider is configured. Both tag-based and path-based invalidation are supported.

Interacting with the cache

The [cache object](#) provides methods for setting cache options, invalidating entries, and checking the current cache state. This object is available in your `.astro` pages as `Astro.cache`, and in API routes and middleware as `context.cache`.

See the [Cache API reference](#) for more details.

Checking if caching is enabled

When caching is not configured, `cache.set()`, `cache.tags`, and `cache.options` log a warning, and `cache.invalidate()` throws an error. To avoid this, wrap your caching logic in a conditional check using [cache.enabled](#). Its value is always `false` when no provider is configured or in development mode.

`src/pages/products/[id].astro`

```
1  ---
2
3  if (Astro.cache.enabled) {
4
5      const tags = await getProductTags(Astro.params.id);
6
7      Astro.cache.set({ maxAge: 3600, tags });
8
9  }
10
11  ---
```

Setting cache options

Call [cache.set\(\)](#) with an options object to enable caching for the current response.

The following example caches a page for 2 minutes, serves stale content for 1 minute while revalidating, and tags the response for targeted invalidation:

`src/pages/index.astro`

```
1  ---export const prerender = false; // Not needed in 'server' mode
2  Astro.cache.set({ maxAge: 120, swr: 60, tags: ['home'],});---
3  <html><body>Cached page</body></html>
```

In API routes and middleware, use `context.cache`:

`src/pages/api/data.ts`

```

1 export function GET(context) {
2
3   context.cache.set({
4
5     maxAge: 300,
6
7     tags: ['api', 'data'],
8
9   });
10
11   return Response.json({ ok: true });
12
13 }

```

Opting out of caching

Call `cache.set()` with `false` to explicitly opt a request out of caching. This is useful when a matched [route rule](#) would otherwise cache the response:

`src/pages/dashboard.astro`

```

1 ---
2
3 if (isPersonalized) {
4
5   Astro.cache.set(false);
6
7 }
8
9 ---

```

Reading cache state

You can access the current accumulated cache options via `cache.options`. This is useful for debugging or when you want to conditionally modify caching based on the current state:

`src/pages/api/debug.ts`

```

1 const { maxAge, swr, tags } = context.cache.options;

```

Invalidating cache entries

You can purge cached entries by tag or path using `cache.invalidate()`. This is useful for programmatically clearing cached content when it becomes stale, such as after a content update or user action.

The following example creates an API route that invalidates by tag and by path:

`src/pages/api/revalidate.ts`

```

1 export async function POST(context) { // Invalidate all entries tagged 'data'  await
  context.cache.invalidate({ tags: ['data'] });
2   // Invalidate a specific path  await context.cache.invalidate({ path: '/api/data' });
3   return Response.json({ purged: true });}

```

Tag-based invalidation removes all cached entries whose tags include any of the provided tags. Path-based invalidation is exact-match only (no [glob](#) or wildcard patterns).

Merge behavior

Multiple calls to [cache.set\(\)](#) within a single request are merged according to the following rules:

- **Scalar values** (`maxAge`, `swr`, `etag`): last-write-wins
- `lastModified`: most recent date wins
- `tags`: accumulate across all calls

Middleware, layouts, content loaders, and page code can each contribute cache directives independently.

Dev mode behavior

In dev mode, the cache API is available so that route code does not need conditional checks, but no actual caching occurs. [cache.enabled](#) is `false`, and [cache.set\(\)](#) and [cache.invalidate\(\)](#) are no-ops. To test your caching locally, build then preview your site.

Route rules

Route rules allow you to define caching behavior for groups of routes declaratively in your config. This is useful for applying caching to large groups of routes at once.

The following example caches all API routes with stale-while-revalidate, product pages with a 1-hour freshness window, and blog posts for 5 minutes:

astro.config.mjs

```

1 import { defineConfig, memoryCache } from 'astro/config';import node from
  '@astrojs/node';
2 export default defineConfig({ adapter: node({ mode: 'standalone' }), cache: {
  provider: memoryCache(), }, routeRules: {   '/api/[...path]': { swr: 600 },
  '/products/[...slug]': { maxAge: 3600, tags: ['products'] },   '/blog/[...slug]': {
    maxAge: 300, swr: 60 }, },});

```

The following route patterns are supported:

- **Static paths:** `/about`, `/api/health`
- **Dynamic parameters:** `/products/[id]`, `/blog/[slug]`
- **Rest parameters:** `/docs/[...path]`

Patterns use the same syntax, matching, and priority rules as Astro's [file-based routing](#), so more specific patterns take precedence. Glob wildcards such as `*` are not supported; use a `[...rest]` parameter to match a group of routes (for example, `/api/[...path]` to match everything under `/api`).

Per-route `cache.set()` calls merge with config-level route rules. Route code can override or extend the defaults set in config. For example, a route rule might set a default `maxAge` for all product pages, but individual pages can call `cache.set()` to customize or disable caching as needed.

Upgrade Astro

This guide covers how to update your version of Astro and related dependencies, how to learn what has changed from one version to the next, and how to understand Astro's versioning system and corresponding documentation updates.

What has changed?

The latest release of Astro is v7.0.3.

You can find an exhaustive list of all changes in [Astro's changelog](#), and important instructions for upgrading to each new [major version](#) in our [upgrade guides](#).

Upgrade to the latest version

Update your project's version of Astro and all official integrations to the latest versions with one command using your package manager:

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 # Upgrade Astro and official integrations together
2
3 npx @astrojs/upgrade
```

Manual Upgrading

To update Astro and integrations to their current versions manually, use the appropriate command for your package manager.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 # Example: upgrade Astro with React and Partytown integrations
2
3 npm install astro@latest @astrojs/react@latest @astrojs/partytown@latest
```

Install a specific version number

To install a specific [version of Astro](#) or integrations, use the appropriate command for your package manager.

- [npm](#)
- [pnpm](#)
- [Yarn](#)

Terminal window

```
1 | npm install astro@4.5.3 @astrojs/react@3.0.10
```

Documentation updates

This documentation is updated for each [minor release](#) and [major version release](#). When new features are added, or existing usage changes, the docs will update to reflect the **current behavior of Astro**. If your project is not updated, then you may notice some behaviors do not match the up-to-date documentation.

New features are added to docs with the specific version number in which they were added. This means that if you have not updated to the latest release of Astro, some documented features may be unavailable. Always check the `Added in:` version number and make sure your project is updated before attempting to use new features!

If you have not upgraded to the latest major version of Astro, you may encounter significant differences between the Astro documentation and your project's behavior. We strongly recommend upgrading to the current major version of Astro as soon as you are able. Both the code and the documentation for earlier versions is unsupported.

Upgrade Guides

After every [major version release](#), you will find an **upgrade guide** with information about important changes and instructions for upgrading your project code.

The main Astro documentation pages are always **accurate for the latest released version of Astro**. They do not describe or compare to how things worked in previous versions, nor do they highlight updated or changed behavior.

See the upgrade guides below for an explanation of changes, comparing the new version to the old. The upgrade guides include everything that could require you to change your own code: breaking changes, deprecations, feature removals and replacements as well as updated usage guidance. Each change to Astro includes a "What should I do?" section to help you successfully update your project code.

- [Upgrade to v7](#)
- [Upgrade to v6](#)
- [Upgrade to v5](#)
- [Upgrade to v4](#)

- [Upgrade to v3](#)
- [Upgrade to v2](#)
- [Upgrade to v1](#)

Older docs (unmaintained)

Documentation for older versions of Astro is not maintained, but is available as a static snapshot. Use these versions of docs if you are unable to upgrade your project, but still wish to consult guides and reference:

- [unmaintained v6.4.8 snapshot](#)
- [unmaintained v5.18.0 snapshot](#)
- [unmaintained v4.16.17 snapshot](#)
- [unmaintained v3.6.3 snapshot](#)
- [unmaintained v2.10.15 snapshot](#)

Semantic versioning

Astro attempts to adhere as much as possible to [semantic versioning](#), which is a set of rules developers use to determine how to assign a version number to a release. Semantic version follows a predictable pattern to inform users of the kind of changes they can expect from one version to the next.

Semantic versioning enforces a pattern of `X.Y.Z` for software version numbers. These values represent **major (X)**, **minor (Y)**, and **patch (Z)** updates.

Patch changes

Patch changes are the least disruptive changes. They do not change the way you use Astro, and no change to your own code is required when you update.

When Astro issues a “patch” version, the last number increases. (e.g. `astro@4.3.14` -> `astro@4.3.15`)

Patches may be released for reasons such as:

- Internal changes that do not change Astro’s functionality:
 - refactors
 - performance improvements
 - increase or change in test coverage
 - aligning with stated documentation and expected behavior
- Improvements to logging and error messages.
- Re-releases after a failed release.

Patch changes also include **most bug fixes**, even in cases where users were taking advantage of existing unintended or undesirable behavior.

Minor changes

Minor releases primarily introduce new features and improvements that you may wish to try, but require no changes to your code. Some existing features may also be **deprecated** (marked for deletion in a future version while continuing to function) in a minor release, giving you the opportunity to prepare for their eventual removal.

Minor releases include changes such as:

- **Deprecations** of existing features/options with a warning that they will be removed in an upcoming major release.
- Introduction of new functionalities.
- Introduction of new options in the integration hooks.
- Introduction of new functionalities in `astro/app`, notably used for creating new adapters.

A minor release may also include smaller, patch changes at the same time.

Major changes

Major releases will include breaking changes to at least some existing code. These breaking changes are always documented in an [“Upgrade to vX” guide](#) in Astro.

Major releases allow Astro to make significant changes not only to internal logic, but also to intended behavior and usage. Documentation will be updated to reflect the latest version only, and **static, unmaintained snapshots of older docs** are available as a historical record for older projects that are not yet upgraded.

Major releases include changes such as:

- Removal of previously deprecated functionalities.
- Changes of existing functionalities.
- Changes of existing options in the integration hooks.
- Changes of existing options and functionalities in `astro/app`, notably used for creating new adapters.

A major release may also include some non-breaking changes and improvements that would normally be released separately in a minor or patch release.

Exceptions

- **Experimental features.** Releasing versions of Astro without adhering to semantic versioning allows Astro developers the greatest flexibility to explore, and even radically change course, during the development of experimental features. Therefore, the behavior of these features can break in minor and patch changes.

These features are usually accompanied by an ongoing, public [Request for Consideration \(RFC\) stage 3](#). It is expected that beta users will follow for updates, and leave early feedback on the discussion to help guide development of these features.

Once these features are out of their experimental period, they will follow the normal semantic versioning contract.

- **Improvements to the documentation** (e.g. reference and error messages). They are built from source

for the `docs` repository. This allows Astro to quickly update docs fixes and improvements in the cases where documentation source content is stored in the main `astro` repository.

Node.js support and upgrade policies

Support

- Astro supports the [latest ***Maintenance* LTS** version of Node.js](#).
- Astro supports the [current ***Active* LTS** version of Node.js](#)
- Astro can support odd versions of Node.js.

Upgrade

The following rules define when Astro may deprecate, drop, or add support for versions of Node.js:

- Odd versions of Node.js can be deprecated and/or dropped when the next even version of Node.js published. This change can occur in a **minor** release of Astro, after a reasonable period of extended support as decided by the Astro Core team.
- Upgrading the minimum ***Maintenance* LTS** (within the same major range, e.g. from `22.14.*` to `22.20.*`) version of Node.js can occur in a **minor** release of Astro.
 - Security exception: If a security flaw in Node.js that **affects Astro** is disclosed and fixed, the Core team can bump the minimum version of the ***Maintenance* LTS** in a **patch** release.
- Upgrading minor or major versions of Node.js (**not** Maintenance LTS) occurs only in major versions of Astro.
 - Security exception: If a security flaw in Node.js that **affects Astro** is disclosed and fixed, the Core team can bump the minimum version in a **minor** release.

Extended maintenance

The Core team will provide extended maintenance **for security fixes only** for one previous major version. This means that if the current major is `v4.*`, the Core team will back port security fixes and issue a new `v3.*` release.

Troubleshooting

Astro provides several different tools to help you troubleshoot and debug your code.

Tips and tricks

Debugging with `console.log()`

`console.log()` is a simple-but-popular method of debugging your Astro code. Where you write your `console.log()` statement will determine where your debugging output is printed:

```
1 ---console.log('Hi! I'm the server. This is logged in the terminal where Astro is
  running. ');---
2 <script>console.log('Hi! I'm the client. This is logged in browser dev console. ');
  </script>
```

A `console.log()` statement in Astro frontmatter will always output to the **terminal** running the Astro CLI. This is because Astro runs on the server, and never in the browser.

Code that is written or imported inside of an Astro `<script>` tag is run in the browser. Any `console.log()` statements or other debug output will be printed to the **console in your browser**.

Debugging framework components

[Framework components](#) (like React and Svelte) are unique: They render server-side by default, meaning that `console.log()` debug output will be visible in the terminal. However, they can also be hydrated for the browser, which may cause your debug logs to also appear in the browser.

This can be useful for debugging differences between the server output and the hydrated components in the browser.

Astro `<Debug />` component

To help you debug your Astro components, Astro provides a built-in `<Debug />` component which renders any value directly into your component HTML template.

This component provides a way to inspect values on the client-side, without any JavaScript. It can be useful for quick debugging in the browser without having to flip back-and-forth between your terminal and your browser.

```
1 ---import { Debug } from 'astro:components';const sum = (a, b) => a + b;---
2 <!-- Example: Outputs {answer: 6} to the browser --><Debug answer={sum(2, 4)} />
```

The Debug component supports a variety of syntax options for even more flexible and concise debugging:

```
1 ---
2
3 import { Debug } from 'astro:components';
4
5 const sum = (a, b) => a + b;
6
7 const answer = sum(2, 4);
8
9 ---
10
11 <!-- Example: All three examples are equivalent. -->
12
13 <Debug answer={sum(2, 4)} />
14
15 <Debug {...{answer: sum(2, 4)}} />
16
17 <Debug {answer} />
```

Common Error Messages

Here are some common error messages you might see in the terminal, what they might mean, and what to do about them. See our [full error reference guide](#) for a complete list of Astro errors you may encounter.

Cannot use import statement outside a module

In Astro components, `<script>` tags are loaded as [JS modules](#) by default. If you have included the `is:inline` directive or any other attribute in your tag, this default behavior is removed.

Solution: If you have added any attributes to your `<script>` tag, you must also add the `type="module"` attribute to be able to use import statements.

Status: Expected Astro behavior, as intended.

Not sure that this is your problem?

Check to see if anyone else has reported [this issue](#)!

document (or window) is not defined

This error occurs when trying to access `document` or `window` on the server.

Astro components run on the server, so you can't access these browser-specific objects within the frontmatter.

Framework components run on the server by default, so this error can occur when accessing `document` or `window` during rendering.

Solution: Determine the code that calls `document` or `window`. If you aren't using `document` or `window` directly and still getting this error, check to see if any packages you're importing are meant to run on the client.

- If the code is in an Astro component, move it to a `<script>` tag outside of the frontmatter. This tells Astro to run this code on the client, where `document` and `window` are available.
- If the code is in a framework component, try to access these objects after rendering using lifecycle methods (e.g. `useEffect()` in React, `onMounted()` in Vue, and `onMount()` in Svelte). Tell the framework component to hydrate client-side by using a `client:` directive, like `client:load`, to run these lifecycle methods. You can also prevent the component from rendering on the server at all by adding the `client:only` directive.

Status: Expected Astro behavior, as intended.

Expected a default export

This error can be thrown when trying to import or render an invalid component, or one that is not working properly. (This particular message occurs because of the way importing a UI component works in Astro.)

Solution: Try looking for errors in any component you are importing and rendering, and make sure it's working correctly. Consider opening an Astro starter template from [astro.new](#) and troubleshooting just your component in a minimal Astro project.

Status: Expected Astro behavior, as intended.

Refused to execute inline script

You may see the following error logged in the browser console:

Refused to execute inline script because it violates the following Content Security Policy directive: ...

This means that your site's [Content Security Policy](#) (CSP) disallows running inline `<script>` tags, which Astro outputs by default.

Solution: Update your CSP to include `script-src: 'unsafe-inline'` to allow inline scripts to run.

Alternatively, you can use a third-party integration such as [astro-shield](#) to generate the CSP headers for you.

Common gotchas

My component is not rendering

First, check to see that you have **imported the component** in your [.astro component script](#) or [.mdx file](#).

Then check your import statement:

- Is your import linking to the wrong place? (Check your import path.)
- Does your import have the same name as the imported component? (Check your component name and that it [follows the .astro syntax](#).)
- Have you included the extension in the import? (Check that your imported file contains an extension. e.g. `.astro`, `.md`, `.vue`, `.svelte`. Note: File extensions are **not** required for `.js(x)` and `.ts(x)` files only.)

My component is not interactive

If your component is rendering (see above) but is not responding to user interaction, then you may be missing a [client:* directive](#) to hydrate your component.

By default, a [UI Framework component is not hydrated in the client](#). If no `client:*` directive is provided, its HTML is rendered onto the page without JavaScript.

Tip

[Astro components](#) are HTML-only templating components with no client-side runtime. But, you can use a `<script>` tag in your Astro component template to send JavaScript to the browser that executes in the global scope.

Cannot find package 'X'

If you see a `"Cannot find package 'react'"` (or similar) warning when you start up Astro, that means that you need to install that package into your project. Not all package managers will install peer dependencies for you automatically. If you are on Node v16+ and using npm, you should not need to worry about this section.

React, for example, is a peer dependency of the `@astrojs/react` integration. That means that you should install the official `react` and `react-dom` packages alongside your integration. The integration will then pull from these packages automatically.

Terminal window

```
1 # Example: Install integrations and frameworks together
2
3 npm install @astrojs/react react react-dom
```

See [Astro's integration guide](#) for instructions on adding framework renderers, CSS tools and other packages to Astro.

Using Astro with Yarn 2+ (Berry)

Yarn 2+, a.k.a. Berry, uses a technique called [Plug'n'Play \(PnP\)](#) to store and manage Node modules, which can [cause problems](#) while initializing a new Astro project using `create astro` or while working with Astro. A workaround is to set the [nodeLinker property](#) in `.yarnrc.yml` to `node-modules`:

`.yarnrc.yml`

```
1 nodeLinker: "node-modules"
```

Adding dependencies to Astro in a monorepo

When working with Astro in a monorepo setup, project dependencies should be added in each project's own `package.json` file.

However, you may also want to use Astro in the root of the monorepo (e.g. [Nx projects recommend installing dependencies at the root](#)). In this case, manually add Astro-related dependencies (e.g. `@astrojs/vue`, `astro-component-lib`) to the `vite.ssr.noExternal` part of Astro's config to ensure that these dependencies are properly installed and bundled:

`astro.config.mjs`

```
1 import { defineConfig } from 'astro/config'
2
3 export default defineConfig({
4
5   vite: {
6
7     ssr: {
8
9       noExternal: [
10
11         '@astrojs/vue',
12
13         'astro-component-lib',
14
15       ]
16
17     }
18
19   }
20
21 })
```

Using `<head>` in a component

In Astro, using a `<head>` tag works like any other HTML tag: it does not get moved to the top of the page or merged with the existing `<head>`. Because of this, you usually only want to include one `<head>` tag throughout a page. We recommend writing that single `<head>` and its contents in a [layout component](#).

An unexpected `<style>` is included

You may notice an imported component's `<style>` tag included in your HTML source even if that component doesn't appear in the final output. For example, this will occur with [conditionally rendered](#) components that are not displayed.

Astro's build process works on the module graph: once a component is included in the template, its `<style>` tag is processed, optimized, and bundled, whether it appears in the final output or not.

Escaping special characters in Markdown

Certain characters have a special meaning in Markdown. You may need to use a different syntax if you want to display them. To do this, you can use [HTML entities](#) for these characters instead.

For example, to prevent `<` being interpreted as the beginning of an HTML element, write `<`.

Creating minimal reproductions

When troubleshooting your code, it can be helpful to create a **minimal reproduction** of the issue that you can share. This is a smaller, simplified Astro project that demonstrates your issue. Having a working reproduction in a new project helps to confirm that this is a repeatable problem, and is not caused by something else in your personal environment or existing project.

Sharing a minimal reproduction is helpful when asking for help in our support threads and is often required when filing a bug report to Astro.

Create a StackBlitz via [astro.new](#)

You can use [astro.new](#) to create a new Astro project with a single click. For minimal reproductions, we strongly recommend starting from the minimal (empty) example running in [StackBlitz](#), with as little extra code as possible.

StackBlitz will run this Astro project in the browser, outside of your local environment. It will also provide you with a shareable link so that any Astro maintainer or support squad member can view your minimal reproduction outside of their own local environment. This means that everyone is viewing the exact same project, with the same configuration and dependencies. This makes it easy for someone else to help troubleshoot your code. If the issue is reproducible, it allows you to verify that the issue lies within the Astro code itself and you can feel confident submitting a bug report.

Note that not every issue is reproducible in StackBlitz. For example, your issue might be dependent on a specific environment or package manager, or it may involve HTML Streaming, which isn't supported in StackBlitz. In this case, create a new minimal (empty) Astro project using the CLI, reproduce the issue, and upload it to a GitHub repository. Instead of sharing a StackBlitz URL, provide a link to the GitHub repository of your minimal reproduction.

Minimal code

Once your empty project is set up, go through the steps to reproduce the issue. This can include adding packages, changing configuration, and writing code.

You should only add the minimum amount of code necessary to reproduce the issue. Do not reproduce other elements of your existing project, and remove all code that is not directly related to the issue.

Create an issue

If your issue can be reproduced, then it is time to create an issue and file a bug report!

Go to the appropriate Astro repository on GitHub and open a new issue. Most repositories have an issue template that will ask questions or require information in order to submit. It's important that you follow these templates because if you don't provide the information we need, then we have to ask you for it... and no one is working on your issue!

Include the link to your minimal reproduction on StackBlitz (or GitHub repository, if necessary). Start with a description of the expected versus actual behavior to provide context for the issue. Then, include clear, step-by-step instructions on how to replicate the issue in an Astro project.

Need more?

Come and chat with us on [Discord](#) and explain your issue in the `#support` forum channel. We're always happy to help!

Visit the current [open Issues in Astro](#) to see if you are encountering a known problem or file a bug report.

You can also visit [Roadmap Discussions](#) to see whether you've found a known limitation of Astro, and check to see whether there are current proposals related to your use case.
