**Astro Notes**

# What is Astro?

Astro is an **all-in-one** **web framework** for building **fast,** **content-focused** websites.

## Key Features

* **Component Islands:** A new web architecture for building faster websites.
* **Server-first API design:** Move expensive hydration off of your users’ devices.
* **Zero JS, by default:** No JavaScript runtime overhead to slow you down.
* **Edge-ready:** Deploy anywhere, even a global edge runtime like Deno or Cloudflare.
* **Customizable:** Tailwind, MDX, and 100+ other integrations to choose from.
* **UI-agnostic:** Supports React, Preact, Svelte, Vue, Solid, Lit and more.

# Installation

npm create astro@latest

yarn create astro

# Prettier Configuration

## Using in VS Code

First install the [VS Code Prettier extension](https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode) and add the following settings to your VS Code configuration so VS Code is aware that Prettier can be used for Astro files:

{

"prettier.documentSelectors": ["\*\*/\*.astro"]

}

Additionally, you should set Prettier as the default formatter for Astro files or VS Code will ask you to choose a formatter everytime you format since the Astro VS Code extension also includes a formatter for Astro files:

{

"[astro]": {

"editor.defaultFormatter": "esbenp.prettier-vscode"

}

}

# Why Astro…

Astro is….

1. [Content-focused](https://docs.astro.build/en/concepts/why-astro/#content-focused): Astro was designed for content-rich websites. This includes most marketing sites, publishing sites, documentation sites, blogs, portfolios, and some ecommerce sites.

By contrast, most modern web frameworks are designed for building web applications. These frameworks work best for 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](https://figma.com/) and [Ping](https://ping.gg/).. Check out [Next.js](https://nextjs.org/) for a great application-focused web framework alternative.

1. [Server-first](https://docs.astro.build/en/concepts/why-astro/#server-first): Websites run faster when they render HTML on the server. **Astro leverages server-side rendering over client-side rendering as much as possible.**

This approach stands in contrast to other modern JavaScript web frameworks like Next.js, SvelteKit, Nuxt, Remix, and others. These frameworks require 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.

1. [Fast by default](https://docs.astro.build/en/concepts/why-astro/#fast-by-default): It should be impossible to build a slow website in Astro. Good performance is always important, but it is *especially* critical for content-focused websites. It has been well-proven that poor performance loses you engagement, conversions, and money

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 users 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 MPA architecture -- to make tradeoffs and deliver features that other frameworks cannot

An Astro website can [load 40% faster with 90% less JavaScript](https://twitter.com/t3dotgg/status/1437195415439360003) than the same site built with the most popular React web framework

1. [Easy to use](https://docs.astro.build/en/concepts/why-astro/#easy-to-use): You don’t need to be an expert to build something with Astro. **Astro’s goal is to be accessible to every web developer**

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.

1. [Fully-featured, but flexible](https://docs.astro.build/en/concepts/why-astro/#fully-featured-but-flexible): Over 100+ Astro integrations to choose from.

**Astro is an all-in-one web framework that comes with everything you need to build a website.** Astro includes component syntax, file-based routing, asset handling, a build process, bundling, optimizations, data-fetching, and more. You can build great websites without ever reaching outside of Astro’s core feature set.

# MPAs vs. SPAs

**A Multi-Page Application (MPA)** is a website consisting of multiple HTML pages, mostly rendered on a server. When you navigate to a new page, your browser requests a new page of HTML from the server. **Astro is an MPA framework**

**A Single-Page Application (SPA)** is a website consisting of a single JavaScript application that loads in the user’s browser and then renders HTML locally. SPAs may *also* generate HTML on the server, but SPAs are unique in their ability to run your website as a JavaScript application in the browser to render a new page of HTML when you navigate. Next.js, Nuxt, SvelteKit, Remix, Gatsby, and Create React App are all examples of SPA frameworks.

**There are three main differences to be aware of when comparing MPAs vs. SPAs:**

#### Server rendering (MPA) vs. client rendering (SPA)

In MPAs, most of your page’s HTML is rendered on the server. In SPAs, most HTML is rendered locally by running JavaScript in the browser. This has a dramatic impact on site behavior, performance, and SEO.

#### Server routing (MPA) vs. client routing (SPA)

Where does your website router live? In an MPA, every request to the server decides which HTML to respond with, so the routing logic lives in the server. In a SPA, your router locally runs in the browser and hijacks any navigation to render the new page without ever hitting a server.

#### Server state management (MPA) vs. client state management (SPA)

SPAs are the superior architecture for websites that deal with complex, multi-page state management (think: Gmail). This is because an SPA runs the entire website as a single JavaScript application, which lets the application maintain state and memory across multiple pages. Interactive, data-driven experiences like inboxes and admin dashboards do well as SPAs because the website itself is inherently “app-like”.

**Few Noticeable Points**:

* An SPA will consistently perform slower on first page load vs. an MPA, unless server rendering is used. This is because an SPA needs to download, parse, and run an entire JavaScript application in the browser just to render any HTML on the page
* MPAs render all HTML on a remote server and often don’t require much (if any) JavaScript to run. This gives MPAs a much faster first load experience than SPAs, which is essential for content-focused websites.
* SPAs can also offer more powerful transitions across page navigation because they control so much about page rendering. To match this support, MPAs leverage tools like Hotwire’s [Turbo](https://turbo.hotwired.dev/) that mimic client routing by also controlling navigation in the browser.

## Are MPAs Better than SPAs?

When comparing MPAs vs SPAs, there is no “better” or “worse” choice. It all comes down to tradeoffs.

**Astro prioritizes the performance and simplicity of MPAs because it makes the most sense for our usecase of content-focused websites.** More stateful, interaction-heavy websites may benefit more from the app-like architecture that SPAs bring at the expense of first-load performance.

# Astro Islands

**Astro Islands** (aka Component Islands) are a pattern of web architecture pioneered by Astro. The idea of “islands architecture” was first coined by Etsy’s frontend architect [Katie Sylor-Miller](https://twitter.com/ksylor) in 2019, and expanded on in [this post](https://jasonformat.com/islands-architecture/) by Preact creator Jason Miller

## What is an Astro Island?

The term “Astro Island” refers to an interactive UI component on an otherwise static page of HTML. Multiple islands can exist on a page, and ***an island always renders in isolation***. Think of them as islands in a sea of static, non-interactive HTML.

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

## How Do Islands Work in Astro?

**Astro generates every website with zero client-side JavaScript, by default.** Use a frontend UI component built with [React](https://reactjs.org/), [Preact](https://preactjs.com/), [Svelte](https://svelte.dev/), [Vue](https://vuejs.org/), [SolidJS](https://www.solidjs.com/), [AlpineJS](https://alpinejs.dev/), or [Lit](https://lit.dev/) and Astro will automatically render it to HTML ahead of time and then strip out all of the JavaScript. This keeps every site fast by default by removing all unused JavaScript from the page.

src/pages/index.astro

---

// Example: Use a static React component on the page, without JavaScript.

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

---

<!-- 100% HTML, Zero JavaScript loaded on the page! -->

<MyReactComponent />

But sometimes, client-side JavaScript is required for creating interactive UI. Instead of forcing your entire page to become an SPA-like JavaScript application, Astro asks you to create an island.

src/pages/index.astro

---

// Example: Use a dynamic React component on the page.

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

---

<!-- This component is now interactive on the page!

The rest of your website remains static and zero JS. -->

<MyReactComponent client:load />

With Astro Islands, the vast majority of your site remains pure, lightweight HTML and CSS. In the example above, you have just added a single, isolated **island of interactivity** without also changing the rest of the page.

## What are the benefits of Islands?

The most obvious benefit to 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

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.

**Islands are the secret to Astro’s fast-by-default performance story!**

# Project Structure

### Example Project Tree

A common Astro project directory might look like this:

├── src/

│ ├── components/

│ │ ├── Header.astro

│ │ └-─ Button.jsx

│ ├── layouts/

│ │ └-─ PostLayout.astro

│ └── pages/

│ │ ├── posts/

│ │ │ ├── post1.md

│ │ │ ├── post2.md

│ │ │ └── post3.md

│ │ └── index.astro

│ └── styles/

│ └-─ global.css

├── public/

│ ├── robots.txt

│ ├── favicon.svg

│ └-─ social-image.png

├── astro.config.mjs

└── package.json

# Components

The most important thing to know about Astro components is that they **render to HTML during your build.** Even if you run JavaScript code inside of your components, it will all run ahead of time, stripped from the final page that you send to your users. The result is a faster site, with zero JavaScript footprint added by default.

## Component Structure

An Astro component is made up of two main parts: the **Component Script** and the **Component Template**.

src/components/EmptyComponent.astro

---

// Component Script (JavaScript)

---

<!-- Component Template (HTML + JS Expressions) -->

**Few Noticeable Points**

* 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.*
* You can also use **dynamic tags** by setting a variable to an HTML tag name or a component import:

src/components/DynamicTags.astro

* ---
* import MyComponent from "./MyComponent.astro";
* const Element = 'div'
* const Component = MyComponent;
* ---
* <Element>Hello!</Element> <!-- renders as <div>Hello!</div> -->
* <Component /> <!-- renders as <MyComponent /> -->

When using dynamic tags:

* + **Variable names must be capitalized.**
  + **Hydration directives are not supported**
* An Astro component template can render multiple elements with no need to wrap everything in a single <div> or <>, unlike JavaScript or JSX.
* However, when using an expression to dynamically create multiple elements, you should wrap these elements inside a **fragment** as you would in JavaScript or JSX. Astro supports using either <Fragment> </Fragment> or the shorthand <> </>.
* ---
* const items = ["Dog", "Cat", "Platypus"];
* ---
* <ul>
* {items.map((item) => (
* <>
* <li>Red {item}</li>
* <li>Blue {item}</li>
* <li>Green {item}</li>
* </>
* ))}
* </ul>
* Fragments can also be useful to avoid wrapper elements when adding [set:\* directives](https://docs.astro.build/en/reference/directives-reference/#sethtml), as in the following example:
* src/components/SetHtml.astro
* ---
* const htmlString = '<p>Raw HTML content</p>';
* ---
* <Fragment set:html={htmlString} />
* In Astro, you use the standard kebab-case format for all HTML attributes instead of the camelCase used in JSX
* In Astro, you can use standard HTML comments where JSX would use JavaScript style comments.
* An Astro component can define and accept props. Props are available on the **Astro.props** global in your frontmatter script.
* Component props can be given default values to use when none are provided.
* src/components/GreetingHeadline.astro
* ---
* const { greeting = "Hello", name = "Astronaut" } = Astro.props;
* ---
* <h2>{greeting}, {name}!</h2>
* 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.
* 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.
* <div id="content-wrapper">
* <Header />
* <slot name="after-header"/> <!-- children with the `slot="after-header"` attribute will go here -->
* <Logo />
* <h1>{title}</h1>
* <slot /> <!-- children without a `slot`, or with `slot="default"` attribute will go here -->
* <Footer />
* <slot name="after-footer"/> <!-- children with the `slot="after-footer"` attribute will go here -->
* </div>
* src/pages/fred.astro
* ---
* import Wrapper from '../components/Wrapper.astro';
* ---
* <Wrapper title="Fred's Page">
* <img src="https://my.photo/fred.jpg" slot="after-header">
* <h2>All about Fred</h2>
* <p>Here is some stuff about Fred.</p>
* <p slot="after-footer">Copyright 2022</p>
* </Wrapper>
* 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.
* <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>

## CSS Styles

* **CSS <style> tags** are also supported inside of the component template.
* They can be used to style your components, and all style rules are automatically scoped to the component itself to prevent CSS conflicts in large apps.
* src/components/StyledHeading.astro
* ---
* // Your component script here!
* ---
* <style>
* /\* scoped to the component, other H1s on the page remain the same \*/
* h1 { color: red }
* </style>
* <h1>Hello, world!</h1>
* **CAUTION**
* The styles defined here apply only to content written directly in the component’s own component template. Children, and any imported components will **not** be styled by default.

## Client-Side Scripts

* To send JavaScript to the browser without [using a framework component](https://docs.astro.build/en/core-concepts/framework-components/) (React, Svelte, Vue, Preact, SolidJS, AlpineJS, Lit) or an [Astro integration](https://astro.build/integrations/) (e.g. astro-XElement), you can use a <script> tag in your Astro component template and send JavaScript to the browser that executes in the global scope.
* By default, <script> tags are processed by Astro.
  + Any imports will be bundled, allowing you to import local files or Node modules.
  + The processed script will be injected into your page’s <head> with [type="module"](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Modules).
  + TypeScript is fully supported, including importing TypeScript files
  + If your component is used several times on a page, the script tag will only be included once.
* <script>
* // Processed! Bundled! TypeScript-supported! ESM imports work, even to npm packages.
* </script>
* To avoid bundling the script, you can use the is:inline attribute.
* <script is:inline>
* // Will be rendered into the HTML exactly as written!
* // ESM imports will not be resolved relative to the file.
* </script>
* Multiple <script> tags can be used in the same .astro file using any combination of the methods above.
* **NOTE**
* Adding type="module" or any other attribute to a <script> tag will disable Astro’s default bundling behavior, treating the tag as if it had an is:inline directive.

### Loading External Scripts

* **When to use this:** If your JavaScript file lives inside of public/.
* Note that this approach skips the JavaScript processing, bundling and optimizations that are provided by Astro when you use the import method described below.
* // absolute URL path
* <script is:inline src="/some-external-script.js"></script>

### Using Hoisted Scripts

* **When to use this:** If your external script lives inside of src/ *and* it supports the ESM module type.
* Astro detects these JavaScript client-side imports and then builds, optimizes, and adds the JS to the page automatically.
* // ESM import
* <script>
* import './some-external-script.js';
* </script>

## HTML Components

Astro supports importing and using .html files as components or placing these files within the src/pages subdirectory as pages.

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 is:inline.
* They can only [reference assets that are in the public/ folder](https://docs.astro.build/en/guides/images/#public).

**NOTE**

* A [<slot /> element](https://docs.astro.build/en/core-concepts/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 <slot> element.