



Astro

Version 5.4 · verified 2026-10-18
Component islands, zero-JS by default,
React hydration, content collections,
dynamic endpoints, and AI agent
integration.

Contents

Mental model	2
Core architecture and benefits	2
React component integration	4
Island state and cross-component communication . .	6
Content collections and type-safe schemas	8
Server endpoints and API routes	10
Optimizing Astro for AI agents	11
Further reading	13

Mental model

Astro is a content-focused web framework designed for speed, utilizing an Islands architecture. It renders HTML on the server by default with zero client-side JavaScript overhead. Interactive UI framework components (React, Vue, Svelte) are isolated into standalone “islands” and hydrated individually on demand.

Core architecture and benefits

Feature	Server Execution	Client Impact
Zero JS baseline	Renders static HTML	Fast initial load
Island hydration	Isolated JS bundles	Low memory footprint
Multi-framework	Renders React/Vue/etc	Mix framework UI

Astro compiles `.astro` components into static HTML at build or request time. Frontmatter between `---` fences

runs exclusively on the server to fetch data, access environment variables, or process markup. Any HTML rendered in the template ships to the browser without framework runtime overhead.

```
---
import Card from '../components/Card.astro';
const items = ['Fast', 'Light', 'Flexible'];
---
<ul>
  {items.map((item) => (
    <li><Card title={item} /></li>
  ))}
</ul>
```

Gotcha: Frontmatter script (---) executes on the server only. Browser APIs like `window`, `document`, or `localStorage` are undefined in frontmatter and throw runtime errors if accessed there.

React component integration

Directive	Hydration Timing	Typical Use Case
<code>client:load</code>	Immediately on load	Critical header UI
<code>client:idle</code>	Browser idle period	Low-priority widgets
<code>client:visible</code>	Viewport entry	Below-fold components
<code>client:only="react"</code>	Client-side only	Non-SSR browser UI

React components render as static HTML on the server unless given an explicit `client:*` hydration directive in an `.astro` template. When hydrated, Astro loads only the React runtime and component code required for that specific island, keeping the rest of the page unburdened by client JavaScript.

```
---
import Counter from '../components/Counter.jsx';
import Chart from '../components/Chart.jsx';
---
<!-- Hydrates when browser becomes idle -->
<Counter client:idle initial={5} />

<!-- Hydrates only when scrolled into view -->
<Chart client:visible data={[1, 2, 3]} />
```

Gotcha: Without a `client:*` directive, `<Counter />` renders static HTML and ignores user clicks, event handlers, and React state updates.

Island state and cross-component communication

Strategy	Scope	Advantage
Shared store	Client module scope	Framework-agnostic sync
Custom Events	DOM window / node	Decoupled event signaling
URL State	Browser query params	Deep-linkable UI state

Because each hydrated component is an isolated React root, standard React Context cannot pass state between separate islands on the same page. Module-level reactive stores or native DOM `CustomEvent` dispatchers sync state across React, Vue, Svelte, and native JS components without re-rendering the surrounding page.

```
type Listener = (count: number) => void;  
let count = 0;
```

```
const listeners = new Set<Listener>();

export const store = {
  get: () => count,
  inc: () => {
    count++;
    listeners.forEach((fn) => fn(count));
  },
  sub: (fn: Listener) => {
    listeners.add(fn);
    return () => listeners.delete(fn);
  },
};
```

Gotcha: React Context providers wrapped around one island do not reach another island. Use module-scope state or global browser events to communicate across island boundaries.

Content collections and type-safe schemas

Feature	Config Function	Purpose
Collection config	<code>defineCollection()</code>	Defines collection rules
Schema validation	<code>z.object()</code>	Validates YAML/JSON data
Data query	<code>getCollection()</code>	Fetches typed entries

Astro Content Collections organize Markdown, MDX, JSON, or YAML content with compile-time type safety. Schemas defined using Zod validate frontmatter fields during build, preventing broken routes or missing metadata before deployment.

```
---  
  
// src/content.config.ts  
import { defineCollection, z } from
```

```
'astro:content';

const blog = defineCollection({
  type: 'content',
  schema: z.object({
    title: z.string(),
    pubDate: z.date(),
    draft: z.boolean().default(false),
  }),
});

export const collections = { blog };
---
```

Gotcha: `getCollection('blog')` returns unrendered content objects. Call `await entry.render()` on a single entry to obtain its rendered `<Content />` component.

Server endpoints and API routes

Method Export	HTTP Trigger	Expected Return
export const GET	GET request	Response object
export const POST	POST request	Response object
export const ALL	Any HTTP method	Response object

Server endpoints defined in `src/pages/` as `.ts` files output raw data instead of HTML. They handle HTTP requests, process webhooks, or return JSON payloads for client components and external service calls.

```
type APIRoute = (  
  ctx: { request: Request },  
) => Response;  
  
export const GET: APIRoute = ({ request }) => {
```

```
const url = new URL(request.url);
const q = url.searchParams.get('q') ?? '';
return new Response(JSON.stringify({ q }), {
  headers: {
    'Content-Type': 'application/json',
  },
});
};
```

Gotcha: In static output mode (`output: 'static'`), API endpoints execute at build time unless individual routes specify `export const prerender = false` with a server adapter configured.

Optimizing Astro for AI agents

AI Pattern	Implementation	Purpose
<code>/llms.txt</code>	Static text route	Machine-readable catalog

AI Pattern	Implementation	Purpose
Clean Markdown API	Raw endpoint export	Scraping- friendly context
Structured JSON	Server API endpoint	Agent tool invocation

AI agents (LLMs, crawlers, coding assistants) consume web content best when it is structured, lightweight, and free of browser execution requirements. Serving plain text `/llms.txt` routes and Markdown content endpoints allows AI agents to inspect page indexes, query documentation, and consume knowledge bases directly.

```

type APIRoute = () => Response;

export const GET: APIRoute = () => {
  const body = `# Site Map for AI Agents
- /docs/intro: Getting started guide
- /api/v1/schema: OpenAPI specs`;
  return new Response(body, {

```

```
headers: { 'Content-Type': 'text/plain' },  
});  
};
```

Gotcha: AI crawlers cannot execute JavaScript or client hydration scripts. Ensure agent routes return static Markdown or JSON rather than relying on hydrated React islands to populate content.

Further reading

- [Astro Documentation](#)
- [Island Architecture Guide](#)
- [Framework Components](#)
- [Content Collections](#)
- [Server Endpoints](#)