Astro Cheat Sheet

Astro ist ein Web-Framework für **content-driven websites**, also vor allem:

- Blogs
- Marketing-Websites
- Dokumentationsseiten
- Portfolios
- Landing Pages
- Community-Seiten
- E-Commerce-Seiten

Astro ist besonders bekannt für:

- **sehr schnelle Ladezeiten**
- **gute SEO**
- **wenig JavaScript im Browser**
- **Server-first Architektur**
- **Islands Architecture**

Kurz gesagt:

> Wenn du eine Website willst, die viel Content anzeigt, schnell lädt und standardmäßig performant ist, ist Astro dafür gemacht.

---

2. Warum Astro?

Die Docs nennen als Grundideen:

1. **Content-driven**
   Astro wurde gebaut, um Inhalte schnell zum Leser zu bringen.
2. **Server-first**
   HTML wird bevorzugt auf dem Server gerendert.
3. **Fast by default**
   Es soll möglichst schwer sein, mit Astro eine langsame Website zu bauen.
4. **Easy to use**
   Wenn du HTML kannst, kannst du schon sehr viel mit Astro.

5. **Developer-focused**
   Gute CLI, VS Code Support, TypeScript, Community, Doku.

---

3. Die wichtigsten Features auf einen Blick

Kern-Features

- **Islands Architecture**
- **UI-agnostisch**
  - React
  - Preact
  - Svelte
  - Vue
  - Solid
  - HTMX
  - Web Components
- **Server-first**
- **Zero JS by default**
- **Content Collections**
- **viele Integrationen**
- **API Hooks**

---

4. Astro mental model: So musst du Astro denken

Wenn du Astro schnell lernen willst, merke dir dieses Modell:

Astro ist standardmäßig:

- **HTML-first**
- **Server-rendered**
- **ohne Client-JS**
- **für Content optimiert**

Das heißt:

- Du schreibst Seiten und Komponenten.
- Astro rendert daraus HTML.
- **Nur wenn du explizit willst**, kommt JavaScript in den Browser.

---

5. Astro vs typische SPA-Frameworks

Astro grenzt sich von klassischen SPA-Ansätzen ab.

SPA-Ansatz

Frameworks wie:

- Next.js
- Nuxt
- SvelteKit
- Remix

wurden laut Docs stark aus Richtung **client-side rendering** gedacht.

Astro-Ansatz

Astro setzt stärker auf:

- **Multi-Page App (MPA)**
- **Server-first**
- **selektive Hydration**
- **nur dort Interaktivität, wo nötig**

Das bedeutet in der Praxis:

- deine Seite lädt schneller
- weniger JS muss im Browser verarbeitet werden
- besser für Content-Seiten

---

6. Islands Architecture verstehen

Das ist eines der wichtigsten Konzepte in Astro.

Grundidee

Die meisten Teile der Seite werden als **statisches HTML** gerendert.

Nur kleine interaktive Bereiche werden als **Islands** geladen.

Beispiel:

- Header mit Dropdown → interaktive Island
- Sidebar → statisches HTML
- Text/Bilder → statisches HTML
- Bildkarussell → interaktive Island
- Footer → statisches HTML

Vorteile

- weniger JavaScript
- bessere Performance
- Komponenten laden unabhängig voneinander
- mehrere Frameworks auf einer Seite möglich

---

7. Was ist eine Island?

In Astro ist eine Island eine **verbesserte UI-Komponente auf einer sonst statischen HTML-Seite**.

Es gibt zwei Arten:

7.1 Client Island

Eine interaktive Komponente, die im Browser hydratisiert wird.

7.2 Server Island

Eine Komponente, deren serverseitige Berechnung separat und verzögert passiert.

---

8. Client Islands

Standardmäßig rendert Astro UI-Komponenten nur zu:

- HTML
- CSS

und entfernt Client-JS automatisch.

Interaktiv machen

Nutze eine `client:*` Direktive:

```
<MyReactComponent client:load />
```

Wichtige Client-Direktiven

- `client:load`
  - lädt sofort beim Laden der Seite
- `client:idle`
  - lädt, wenn der Browser idle ist
- `client:visible`
  - lädt erst, wenn die Komponente im Viewport sichtbar wird
- `client:media={QUERY}`
  - lädt nur bei passender Media Query
- `client:only="react"` / `"svelte"` etc.
  - rendert nur im Client, nicht auf dem Server

Merksatz

> Nur Komponenten mit `client:*` laufen im Browser.

---

9. Server Islands

Mit `server:defer` kannst du teure oder langsame serverseitige Komponenten aus dem Haupt-Renderprozess auslagern.

Beispiel:

```
---
import Avatar from "../components/Avatar.astro";
---
<Avatar server:defer />
```

Typische Anwendungsfälle

- Benutzer-Avatar

- personalisierte Header-Bereiche
- Reviews
- Deals / Rabattinfos
- dynamische, aber kleine Teilbereiche

Vorteile

- statischer Hauptinhalt rendert sofort
- kleinere dynamische Teile kommen parallel nach
- bessere Cachebarkeit
- bessere UX

Fallback-Inhalt

Du kannst Fallback-Inhalte mit Slot `"fallback"` angeben:

```
<Avatar server:defer>
  <GenericAvatar slot="fallback" />
</Avatar>
```

---

10. Installation

Voraussetzungen

- **Node.js v22.12.0 oder höher**
- **ungerade Versionen wie v23 werden nicht unterstützt**
- Editor, empfohlen: **VS Code**
- Terminal

Schnellstart per CLI

```
npm create astro@latest
```

Danach:

```
cd dein-projekt
npm install
npm run dev
```

Mit Integrationen direkt beim Erstellen

```
npm create astro@latest -- --add react --add partytown
```

Mit Template

```
npm create astro@latest -- --template <example-name>
```

oder

```
npm create astro@latest -- --template <github-user>/<repo>
```

---

11. Manuelle Installation

1. Projektordner erstellen

```
mkdir my-astro-project
cd my-astro-project
```

2. package.json anlegen

```
npm init --yes
```

3. Astro installieren

```
npm install astro
```

4. Scripts in `package.json`

```
{
  "scripts": {
    "dev": "astro dev",
    "build": "astro build",
    "preview": "astro preview"
  }
}
```

5. Erste Seite erstellen

Pfad:

```
src/pages/index.astro
```

Beispiel:

```
---
console.log('This runs in your terminal, not the browser!');
---

<html>
  <body>
    <h1>Hello, World!</h1>
  </body>
</html>
```

```
<style>
  h1 {
    color: orange;
  }
</style>
```

6. `public/robots.txt`

```
User-agent: *
Allow: /
```

7. `astro.config.mjs`

```
import { defineConfig } from "astro/config";
export default defineConfig({});
```

8. `tsconfig.json`

```
{
  "extends": "astro/tsconfigs/base"
}
```

---

12. Projektstruktur

Empfohlene Struktur:

```
src/
public/
package.json
astro.config.mjs
tsconfig.json
```

Wichtige Ordner

`src/`

Hier liegt dein Quellcode:

- Pages
- Layouts
- Astro-Komponenten
- Framework-Komponenten
- Styles
- Markdown
- Bilder

`src/pages/`

Pflichtordner.
Hier entstehen deine Routen.

`src/components/`

Wiederverwendbare Komponenten.

`src/layouts/`

Layouts für Seitenstrukturen.

`src/styles/`

CSS / Sass etc.

`public/`

Unverarbeitete Assets:

- Fonts
- Icons
- `robots.txt`
- `manifest.webmanifest`

Wichtig:

> Dateien in `public/` werden nicht von Astro optimiert oder gebundelt.

---

13. Entwickeln und Build

Dev-Server starten

```
npm run dev
```

Standardmäßig unter:

```
http://localhost:4321/
```

Build

```
npm run build
```

Output standardmäßig in:

```
dist/
```

Preview des Builds

```
npm run preview
```

---

14. Astro-Komponenten

Astro-Komponenten sind die Grundbausteine.

- Dateiendung: `.astro`
- kein Client-Runtime-Overhead
- rendern zu HTML

Aufbau einer Astro-Komponente

```
---
// Component Script
---

<!-- Component Template -->
```

Component Script

Im Frontmatter (`---`) kannst du:

- importieren
- Daten laden
- Variablen definieren
- APIs fetchen
- Props auslesen

Beispiel:

```
---
import SomeAstroComponent from '../components/SomeAstroComponent.astro';
import SomeReactComponent from '../components/SomeReactComponent.jsx';
import someData from '../data/pokemon.json';

const { title } = Astro.props;
const data = await fetch('SOME_SECRET_API_URL/users').then(r => r.json());
---
```

Component Template

Darunter kommt dein HTML + Astro-Syntax.

---

15. Props in Astro-Komponenten

Beispiel:

```
---
const { greeting, name } = Astro.props;
---
<h2>{greeting}, {name}!</h2>
```

Verwendung:

```
<GreetingHeadline greeting="Hi" name="Astro" />
```

Mit TypeScript

```
---
interface Props {
  name: string;
  greeting?: string;
}

const { greeting = "Hello", name } = Astro.props;
---
<h2>{greeting}, {name}!</h2>
```

---

16. Slots

Slots sind Platzhalter für Kind-Inhalte.

Default Slot

`Wrapper.astro`

```
<div>
  <slot />
</div>
```

Verwendung:

```
<Wrapper>
  <p>Inhalt</p>
</Wrapper>
```

Named Slots

Komponente:

```
<div>
  <slot name="after-header" />
  <slot />
  <slot name="after-footer" />
</div>
```

Verwendung:

```
<Wrapper>
  <img slot="after-header" src="/header.jpg" />
  <p>Hauptinhalt</p>
  <p slot="after-footer">Footer-Text</p>
</Wrapper>
```

Fallback Content

```
<slot>
  <p>Fallback, wenn kein Inhalt übergeben wurde</p>
</slot>
```

---

17. Layouts

Layouts sind normale Astro-Komponenten für gemeinsame Seitenstruktur.

Typischer Inhalt:

- `<html>`
- `<head>`
- `<body>`
- `<slot />`

Beispiel:

```
---
const { title } = Astro.props;
---
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>{title}</title>
  </head>
  <body>
    <nav>...</nav>
    <main>
      <slot />
    </main>
  </body>
</html>
```

Nutzung:

```
---
import MySiteLayout from '../layouts/MySiteLayout.astro';
---
<MySiteLayout title="Home Page">
```

```
  <p>Mein Inhalt</p>
</MySiteLayout>
```

---

18. Routing

Astro verwendet **file-based routing**.

Beispiele

```
src/pages/index.astro        -> /
src/pages/about.astro        -> /about
src/pages/about/index.astro  -> /about
src/pages/about/me.astro     -> /about/me
src/pages/posts/1.md         -> /posts/1
```

Linking

Du nutzt normale HTML-Links:

```
<a href="/about/">About</a>
```

Keine spezielle Link-Komponente nötig.

---

19. Dynamische Routen

Beispiel:

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

Dann brauchst du in statischem Modus `getStaticPaths()`:

```
---
export function getStaticPaths() {
  return [
    { params: { dog: "clifford" } },
    { params: { dog: "rover" } },
    { params: { dog: "spot" } },
  ];
}

const { dog } = Astro.params;
---
<div>Good dog, {dog}!</div>
```

Mehrere Parameter

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

Rest-Parameter

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

Beispiel:

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

20. Redirects und Rewrites

Config-Redirects

In `astro.config.mjs`:

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

export default defineConfig({
  redirects: {
    "/old-page": "/new-page",
    "/blog": "https://example.com/blog"
  }
});
```

Mit Statuscode:

```
redirects: {
  "/old-page": {
    status: 302,
    destination: "/new-page"
  }
}
```

Dynamische Redirects

```
---
if (!isLoggedIn(cookie)) {
  return Astro.redirect("/login");
}
---
```

Rewrites

Mit Rewrite bleibt URL sichtbar, aber anderer Inhalt wird gezeigt.

```
---
return Astro.rewrite("/es/articles/introduction");
---
```

---

21. Page Partials

Partials sind Seiten in `src/pages/`, die **keine vollständige HTML-Seite** rendern.

Wichtig:

```
---
export const partial = true;
---
<li>I'm a partial!</li>
```

Geeignet für Libraries wie:

- htmx
- Stimulus
- jQuery
- Unpoly

---

22. Pagination

Astro hat eingebaute Pagination.

Beispiel:

```
---
export function getStaticPaths({ paginate }) {
  const astronautPages = [
    { astronaut: "Neil Armstrong" },
    { astronaut: "Buzz Aldrin" },
    { astronaut: "Sally Ride" },
```

```
    { astronaut: "John Glenn" },
  ];

  return paginate(astronautPages, { pageSize: 2 });
}

const { page } = Astro.props;
---
<h1>Page {page.currentPage}</h1>
<ul>
  {page.data.map(({ astronaut }) => <li>{astronaut}</li>)}
</ul>
```

Wichtige `page`-Properties

- `page.data`
- `page.start`
- `page.end`
- `page.total`
- `page.currentPage`
- `page.size`
- `page.lastPage`
- `page.url.current`
- `page.url.prev`
- `page.url.next`
- `page.url.first`
- `page.url.last`

---

23. Endpoints / API-Routen

Dateien in `src/pages/` mit `.js` oder `.ts` können Endpoints sein.

Beispiel:

```
src/pages/builtwith.json.ts
```

```
export function GET() {
  return new Response(
    JSON.stringify({
      name: "Astro",
      url: "https://astro.build/",
    }),
  );
}
```

Dynamische Endpoints

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

```
import type { APIRoute } from "astro";

const usernames = ["Sarah", "Chris", "Yan", "Elian"];

export const GET = (({ params }) => {
  const id = params.id;
  return new Response(JSON.stringify({
    name: usernames[id],
  }));
}) satisfies APIRoute;
```

HTTP-Methoden

Unterstützt:

- `GET`
- `POST`
- `DELETE`
- `ALL`
- usw.

---

24. Middleware

Mit Middleware kannst du Requests/Responses abfangen.

Datei:

```
src/middleware.ts
```

Beispiel:

```
export function onRequest(context, next) {
  context.locals.title = "New title";
  context.locals.property = "information";
  return next();
}
```

In Astro-Komponente:

```
---
const data = Astro.locals;
---
<h1>{data.title}</h1>
<p>{data.property}</p>
```

`locals`

`context.locals` dient zum Weitergeben request-spezifischer Daten.

Typische Verwendung:

- aktueller User
- Session
- Orders
- Funktionen
- Auth-Daten

Middleware typisieren

```
import { defineMiddleware } from "astro:middleware";

export const onRequest = defineMiddleware((context, next) => {
  return next();
});
```

Middleware chainen

```
import { sequence } from "astro:middleware";

export const onRequest = sequence(validation, auth, greeting);
```

---

25. Pages

Unterstützte Dateitypen in `src/pages/`:

- `.astro`
- `.md`
- `.mdx` (mit MDX-Integration)
- `.html`
- `.js` / `.ts` (Endpoints)

404-Seite

```
src/pages/404.astro
```

500-Seite

```
src/pages/500.astro
```

`500.astro` erhält automatisch ein `error`-Prop.

---

26. HTML-Komponenten

`.html`-Dateien können importiert und benutzt werden.

Einschränkungen:

- kein Frontmatter
- keine serverseitigen Imports
- keine dynamischen Expressions
- `<script>` bleibt ungebundelt
- nur Assets aus `public/`

---

27. Styling in Astro

Astro macht CSS sehr einfach.

Lokale Styles

```
<style>
  h1 { color: red; }
</style>
```

Scoped Styles

Standardmäßig sind Styles **gescoped**.

Das bedeutet:

- Styles gelten nur in dieser Komponente
- kein Leak in andere Komponenten

Globale Styles

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

Gemischt mit `:global()`

```
<style>
  h1 { color: red; }
  article :global(h1) {
    color: blue;
  }
</style>
```

---

28. `class:list`

Dynamische Klassen in Astro:

```
---
const { isRed } = Astro.props;
---
<div class:list={['box', { red: isRed }]}>
  <slot />
</div>
```

---

29. CSS-Variablen mit `define:vars`

```
---
const foregroundColor = "rgb(221 243 228)";
const backgroundColor = "rgb(24 121 78)";
```

```
---
<style define:vars={{ foregroundColor, backgroundColor }}>
  h1 {
    background-color: var(--backgroundColor);
    color: var(--foregroundColor);
  }
</style>
```

---

30. CSS importieren

Lokales Stylesheet

```
---
import '../styles/utils.css';
---
```

Aus npm

```
---
import 'package-name/styles.css';
---
```

Wenn keine Dateiendung verwendet wird, ggf. `vite.ssr.noExternal` setzen.

---

31. Tailwind mit Astro

Tailwind 4 hinzufügen

```
npx astro add tailwind
```

Dann in z. B. `src/styles/global.css`:

```
@import "tailwindcss";
```

Und in Layout/Page importieren:

```
---
import "../styles/global.css";
---
```

Legacy Tailwind 3

Für Tailwind 3 braucht man:

- `tailwindcss@3`
- `@astrojs/tailwind`

---

32. TypeScript in Astro

Astro hat eingebauten TypeScript-Support.

Wichtige Empfehlung

Verwende in `tsconfig.json` möglichst:

- `astro/tsconfigs/strict`
- oder `strictest`

Typ-Checken

Der Dev-Server prüft Typen **nicht vollständig**.

Dafür:

```
astro check
```

oder im Build-Script:

```
{
  "scripts": {
    "build": "astro check && astro build"
  }
}
```

---

33. Props typisieren

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

---

34. Nützliche Typ-Utilities

`HTMLAttributes`

Für HTML-Prop-Typen:

```
---
import type { HTMLAttributes } from "astro/types";

type Props = HTMLAttributes<"a">;
---
```

`ComponentProps`

Props einer anderen Komponente referenzieren:

```
---
import type { ComponentProps } from "astro/types";
import Button from "./Button.astro";

type ButtonProps = ComponentProps<typeof Button>;
---
```

`Polymorphic`

Für polymorphe Komponenten.

---

35. Environment Variables

Astro nutzt `import.meta.env`.

Beispiel `.env`

```
SECRET_PASSWORD=password123
PUBLIC_ANYBODY=there
```

Zugriff

- Server + Client: nur `PUBLIC_*`
- Nur Server: alle Variablen

```
import.meta.env.PUBLIC_ANYBODY
import.meta.env.SECRET_PASSWORD
```

Wichtiger Hinweis

`.env`-Dateien werden **nicht** in `astro.config.mjs` geladen.

Dort ggf. `process.env` oder `loadEnv` verwenden.

---

36. Type-safe env mit `astro:env`

Im Config-File:

```
import { defineConfig, envField } from "astro/config";

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"
      }),
    }
  }
});
```

Verwendung:

```
import { API_URL } from "astro:env/client";
import { API_SECRET } from "astro:env/server";
```

---

37. Integrationen

Integrationen erweitern Astro.

Offizielle Integrationen

Frameworks

- `@astrojs/react`
- `@astrojs/preact`
- `@astrojs/vue`
- `@astrojs/svelte`
- `@astrojs/solid-js`
- `@astrojs/alpinejs`

Adapter

- `@astrojs/node`
- `@astrojs/netlify`
- `@astrojs/vercel`
- `@astrojs/cloudflare`

Weitere

- `@astrojs/mdx`
- `@astrojs/partytown`
- `@astrojs/sitemap`
- `@astrojs/markdoc`

Integration automatisch hinzufügen

```
npx astro add react
```

Mehrere:

```
npx astro add react sitemap partytown
```

Manuelle Konfiguration

```
import { defineConfig } from 'astro/config';
import sitemap from '@astrojs/sitemap';

export default defineConfig({
  integrations: [sitemap()]
});
```

---

38. Frontend-Frameworks mit Astro

Du kannst React, Vue, Svelte etc. direkt verwenden.

Statisches Rendering

```
---
import MyReactComponent from '../components/MyReactComponent.jsx';
---
<MyReactComponent />
```

Standardmäßig rendert Astro das als **statisches HTML**.

Interaktiv machen

```
<MyReactComponent client:load />
```

Mehrere Frameworks mischen

```
---
import MyReactComponent from '../components/MyReactComponent.jsx';
import MySvelteComponent from '../components/MySvelteComponent.svelte';
import MyVueComponent from '../components/MyVueComponent.vue';
---
<div>
  <MySvelteComponent />
  <MyReactComponent />
  <MyVueComponent />
</div>
```

Wichtig:

> Nur `.astro`-Dateien dürfen mehrere Frameworks mischen.

---

39. Props und Children an Framework-Komponenten

Props übergeben:

```
<TodoList initialTodos={["learn Astro", "review PRs"]} />
```

Unterstützte serialisierbare Prop-Typen

- plain object
- number
- string
- Array
- Map
- Set
- RegExp
- Date
- BigInt
- URL
- Uint8Array
- Uint16Array
- Uint32Array
- Infinity

Nicht unterstützt für Hydration:

- Funktionen
- zirkuläre Referenzen

Children übergeben

```
<MyReactSidebar>
  <p>Sidebar Content</p>
</MyReactSidebar>
```

Named Slots an Framework-Komponenten

Geht ebenfalls.

---

40. Scripts in Astro

Client-Script in Astro-Komponente

```
<button data-confetti-button>Celebrate!</button>

<script>
  import confetti from 'canvas-confetti';

  const buttons = document.querySelectorAll('[data-confetti-button]');
  buttons.forEach((button) => {
    button.addEventListener('click', () => confetti());
  });
</script>
```

Was Astro mit `<script>` macht

Standardmäßig, wenn keine zusätzlichen Attribute vorhanden sind:

- TypeScript-Support
- Bundling
- `type="module"`
- Deduplication
- kleine Scripts werden inline eingebettet

Unprocessed Script

Wenn du ein Attribut hinzufügst oder `is:inline` setzt, dann wird Astro das Script **nicht** verarbeiten:

```
<script is:inline>
  console.log("raw script");
</script>
```

---

41. Typische Script-Muster

Event Handling

In Astro kein React-`onClick={}`.

Stattdessen:

```
<button class="alert">Click me!</button>
<script>
  const buttons = document.querySelectorAll('button.alert');
  buttons.forEach((button) => {
    button.addEventListener('click', () => {
      alert('Button was clicked!');
    });
  });
</script>
```

Frontmatter-Werte an Script übergeben

Über `data-*` Attribute:

```
---
const { message = 'Welcome, world!' } = Astro.props;
---
<astro-greet data-message={message}>
  <button>Say hi!</button>
</astro-greet>
```

---

42. Markdown

Astro unterstützt Markdown nativ.

`.md` in `src/pages/`

wird automatisch zu einer Seite.

Markdown-Layout über Frontmatter

```
---
layout: ../layouts/BlogPostLayout.astro
title: My Markdown page
---
# Title
This is my page, written in **Markdown.**
```

Wichtig

Wenn du ein Layout nutzt, musst du in diesem Layout selbst setzen:

```
<meta charset="utf-8">
```

---

43. Content Collections

Für strukturierte Inhalte ist das **eine der wichtigsten Astro-Funktionen**.

Ideal für:

- Blogposts
- Produktdaten
- Autorenprofile

- Dokus
- Rezepte
- CMS-Daten

Vorteile

- Struktur
- Validation mit Zod
- Type-Safety
- Intellisense
- gute Query-API
- skalierbar

---

44. Content Collections definieren

Datei:

```
src/content.config.ts
```

Beispiel:

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

const blog = defineCollection({
  loader: glob({ base: './src/content/blog', pattern: '**/*.{md,mdx}' }),
  schema: z.object({
    title: z.string(),
    description: z.string(),
    pubDate: z.coerce.date(),
    updatedDate: z.coerce.date().optional(),
  }),
});
```

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

45. Collections abfragen

Ganze Collection

```
---
import { getCollection } from 'astro:content';
const posts = await getCollection('blog');
---
```

Einzelnen Entry

```
---
import { getEntry } from 'astro:content';
const post = await getEntry('blog', 'post-1');
---
```

Sortieren

```
---
const posts = (await getCollection('blog')).sort(
  (a, b) => b.data.pubDate.valueOf() - a.data.pubDate.valueOf()
);
---
```

Filtern

```
---
const publishedBlogEntries = await getCollection('blog', ({ data }) => {
  return data.draft !== true;
});
---
```

46. Content rendern

Nach dem Query:

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

const entry = await getEntry("blog", "post-1");
if (!entry) throw new Error("Entry not found");

const { Content } = await render(entry);
---
<h1>{entry.data.title}</h1>
<Content />
```

47. Referenzen in Collections

Beispiel: Blogpost referenziert Autor.

```
import { defineCollection, reference } from "astro:content";
import { z } from "astro/zod";

const blog = defineCollection({
  schema: z.object({
    title: z.string(),
    author: reference("authors"),
    relatedPosts: z.array(reference("blog")),
  }),
```

```
});
```

48. Dynamische Seiten aus Collections generieren

Static Build mit `getStaticPaths()`

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

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

const { post } = Astro.props;
const { Content } = await render(post);
---
<h1>{post.data.title}</h1>
<Content />
```

49. Daten fetchen

Direkt in Astro

```
---
const response = await fetch("https://randomuser.me/api/");
const data = await response.json();
const randomUser = data.results[0];
---
<h2>{randomUser.name.first} {randomUser.name.last}</h2>
```

In Framework-Komponenten

Geht auch, aber beachte Server/Client-Kontext.

GraphQL

Auch via `fetch()`.

---

50. On-demand Rendering / SSR

Standardmäßig baut Astro statische Seiten.

Wenn du SSR brauchst:

1. Adapter installieren
2. bei einzelner Route:

```
---
export const prerender = false;
---
```

Dann wird die Route on demand gerendert.

Komplettes Projekt server-first

In Config:

```
output: "server"
```

Dann sind standardmäßig alle Seiten SSR.

Einzelne Seiten kannst du statisch machen mit:

```
---
export const prerender = true;
---
```

---

51. Cookies, Request, Response im SSR

Cookies

```
---
let counter = 0
if (Astro.cookies.has('counter')) {
  const cookie = Astro.cookies.get('counter')
  const value = cookie?.number()
  if (value !== undefined && !isNaN(value)) counter = value + 1
}
Astro.cookies.set('counter', String(counter))
---
<h1>Counter = {counter}</h1>
```

Response

```
---
Astro.response.status = 404;
Astro.response.statusText = 'Not found';
---
```

Headers setzen

```
---
Astro.response.headers.set('Cache-Control', 'public, max-age=3600');
---
```

Request auslesen

```
---
const cookie = Astro.request.headers.get('cookie');
console.log(Astro.request.method);
---
```

---

52. Sessions

Sessions sind für serverseitig gespeicherten Request-/User-State.

Typische Use-Cases:

- Userdaten
- Warenkorb
- Formularstatus

Beispiel

```
---
const cart = await Astro.session?.get('cart');
---
<a href="/checkout">□□{cart?.length ?? 0} items</a>
```

Session setzen in API oder Middleware

Geht über `context.session`.

---

53. Actions

Actions sind type-safe Backend-Funktionen für Client-Server-Kommunikation.

Sie reduzieren Boilerplate gegenüber klassischen API-Routes.

Definieren

Datei:

```
src/actions/index.ts
```

Beispiel:

```
import { defineAction } from 'astro:actions';
import { z } from 'astro/zod';

export const server = {
  getGreeting: defineAction({
    input: z.object({
      name: z.string(),
    }),
```

```
    handler: async (input) => {
      return `Hello, ${input.name}!`
    }
  })
}
```

Aufrufen

```
<script>
  import { actions } from 'astro:actions';

  async function run() {
    const { data, error } = await actions.getGreeting({ name: "Houston" });
    if (!error) alert(data);
  }
</script>
```

Form-Daten akzeptieren

```
comment: defineAction({
  accept: 'form',
  input: z.object({...}),
  handler: async (input) => { ... }
})
```

Fehlerbehandlung mit `ActionError`

```
import { ActionError } from "astro:actions";

throw new ActionError({
  code: "UNAUTHORIZED",
  message: "User must be logged in."
});
```

54. Prefetch

Astro kann Seiten vorladen.

Aktivieren

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

export default defineConfig({
  prefetch: true
});
```

Pro Link aktivieren

```
<a href="/about" data-astro-prefetch>About</a>
```

Strategien

- `hover`
- `tap`
- `viewport`
- `load`

Beispiel:

```
<a href="/about" data-astro-prefetch="tap">About</a>
```

Alle Links prefetchen

```
prefetch: {
  prefetchAll: true
```

```
}
```

Opt-out pro Link:

```
<a href="/about" data-astro-prefetch="false">About</a>
```

---

55. View Transitions

Mit Astro kannst du Übergänge zwischen Seiten animieren.

Aktivieren

```
---
import { ClientRouter } from "astro:transitions";
---
<ClientRouter />
```

Meist in einem gemeinsamen Layout/Head.

Wichtige Direktiven

- `transition:name`
- `transition:animate`
- `transition:persist`
- `transition:persist-props`

Built-in Animationen

- `fade`
- `initial`
- `slide`
- `none`

Beispiel:

```
<main transition:animate="slide">
  ...
</main>
```

Persistente Elemente

```
<video controls muted autoplay transition:persist>
```

oder für Island:

```
<Counter client:load transition:persist initialCount={5} />
```

---

56. i18n Routing

Astro hat eingebaute i18n-Routing-Funktionen.

Grundkonfiguration

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

export default defineConfig({
  i18n: {
    locales: ["es", "en", "pt-br"],
    defaultLocale: "en",
  }
})
```

Lokalisierte Ordner

```
src/pages/es/about.astro
src/pages/en/about.astro
```

`prefixDefaultLocale`

- `false` (default): Default-Sprache ohne Prefix
- `true`: auch Default-Sprache mit Prefix

---

57. Bilder

Astro bietet starke Bildunterstützung.

Empfohlen: Bilder in `src/`

Dann kann Astro sie optimieren.

`Image`-Komponente

```
---
import { Image } from 'astro:assets';
import myImage from '../assets/my_image.png';
---
<Image src={myImage} alt="Beschreibung" />
```

`Picture`-Komponente

```
---
import { Picture } from 'astro:assets';
import myImage from '../assets/my_image.png';
---
<Picture src={myImage} formats={['avif', 'webp']} alt="Beschreibung" />
```

HTML-`img`

Auch möglich, aber ohne Optimierung:

```
<img src="/images/public-cat.jpg" alt="A sleeping cat.">
```

Remote Images authorisieren

```
export default defineConfig({
  image: {
    domains: ["astro.build"],
  }
});
```

oder:

```
export default defineConfig({
  image: {
    remotePatterns: [{ protocol: "https" }],
  }
});
```

---

58. SVGs

SVG-Dateien können als Astro-Komponenten importiert werden.

```
---
import Logo from './path/to/svg/file.svg';
---
<Logo />
```

Mit Props:

```
<Logo width={64} height={64} fill="currentColor" />
```

---

59. Syntax Highlighting

Astro unterstützt standardmäßig:

- **Shiki**
- **Prism**

Markdown-Codeblöcke

Standardmäßig mit **Shiki** und Theme `github-dark`.

Theme setzen

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

export default defineConfig({
  markdown: {
    shikiConfig: {
      theme: 'dracula',
    },
  },
});
```

Light/Dark Themes

```
export default defineConfig({
  markdown: {
    shikiConfig: {
      themes: {
        light: 'github-light',
        dark: 'github-dark',
      },
    },
```

```
  },
});
```

`<Code />`

```
---
import { Code } from 'astro:components';
---
<Code code={`const foo = 'bar';`} lang="js" />
```

`<Prism />`

Erfordert Installation:

```
npm install @astrojs/prism
```

Dann:

```
---
import { Prism } from '@astrojs/prism';
---
<Prism lang="js" code={`const foo = 'bar';`} />
```

---

60. Dev Toolbar

Astro hat eine eingebaute Dev Toolbar im Browser.

Built-in Apps

- **Astro Menu**
- **Inspect**
- **Audit**
- **Settings**

Deaktivieren pro Projekt

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

export default defineConfig({
  devToolbar: {
    enabled: false
  }
});
```

---

61. Editor-Setup

Empfohlen: VS Code

Offizielle Astro VS Code Extension mit:

- Syntax Highlighting
- TypeScript-Typinfos
- IntelliSense

Weitere unterstützte Editoren laut Docs:

- Zed
- WebStorm / JetBrains
- Vim / Neovim
- Emacs
- Sublime Text
- Nova
- StackBlitz / CodeSandbox
- GitHub.dev

---

62. Nützliche Tools

ESLint

Community Plugin.

Stylelint

Community-Konfiguration.

Biome

Experimenteller Support für `.astro`.

Prettier

Mit Astro Plugin:

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

`.prettierrc`:

```
{
  "plugins": ["prettier-plugin-astro"],
  "overrides": [
    {
      "files": "*.astro",
      "options": {
        "parser": "astro"
      }
    }
  ]
}
```

Formatieren:

```
npx prettier . --write
```

63. Häufige Fehler / Gotchas

„document/window is not defined“

Du greifst serverseitig auf Browser-APIs zu.

Lösung:

- in `<script>` verschieben
- oder bei Frameworks Lifecycle + `client:*`

Komponente rendert nicht

Check:

- Import korrekt?
- Pfad korrekt?
- Name korrekt?
- Dateiendung korrekt?

Komponente ist nicht interaktiv

Wahrscheinlich fehlt `client:*`.

`<head>` in Komponenten

Astro verschiebt `<head>` nicht automatisch nach oben.
Am besten nur ein zentrales `<head>` im Layout.

64. Wichtige Best Practices

1. Denke zuerst in HTML

Astro ist HTML-first. Nutze diese Stärke.

2. So wenig Client-JS wie möglich

Füge `client:*` nur dort hinzu, wo wirklich Interaktivität nötig ist.

3. Nutze Layouts

Zentrale Struktur in Layouts halten.

4. Nutze Content Collections für strukturierte Inhalte

Für Blogs, Docs, Produkte etc. fast immer sinnvoll.

5. Nutze `src/` für Bilder

Damit Astro optimieren kann.

6. Halte globale Styles minimal

Scoped Styles bevorzugen.

7. Bleib erstmal im `static`-Modus

Nur auf `output: "server"` wechseln, wenn wirklich viele SSR-Seiten nötig sind.

8. Nutze `astro add`

Integrationen lieber mit CLI hinzufügen statt manuell.

9. TypeScript auch dann nutzen, wenn du „kein TS magst“

Schon allein für:

- bessere IntelliSense
- Content Collections
- sichere Props
- Editorhilfe

10. Performance zuerst

Astros größter Vorteil ist Performance. Verspiel ihn nicht mit unnötigen Client-Bundles.

---

65. Schnellstart-Workflow für echte Projekte

Wenn du **morgen ein Astro-Projekt bauen** willst, arbeite am besten so:

Für Blog / Doku / Content-Seite

1. Projekt erstellen

   ```
   npm create astro@latest
   ```

2. Dev-Server starten

   ```
   npm run dev
   ```

3. Layout anlegen
   - `src/layouts/Layout.astro`
4. Seiten anlegen
   - `src/pages/index.astro`
   - `src/pages/about.astro`
5. Styles global + scoped kombinieren
   - `src/styles/global.css`
   - pro Komponente `<style>`
6. Content Collection einrichten
   - `src/content.config.ts`
   - `src/content/blog/*.md`
7. Blog-Liste mit `getCollection()`
8. Blog-Detailseiten per dynamischer Route
9. Bilder mit `<Image />`
10. Nur notwendige interaktive Komponenten per `client:*`

Für Marketing-Website

1. Astro installieren
2. Layout + Sections als Komponenten
3. Bilder optimieren
4. wenn nötig:
   - React/Svelte-Komponente für Slider, Modal, Formular
5. `prefetch` aktivieren
6. bei Bedarf View Transitions aktivieren

Für E-Commerce / dynamischere Seite

1. Astro + Adapter
2. statische Teile prerendern
3. personalisierte Teile als `server:defer`
4. API-Routes / Actions für Warenkorb / Checkout
5. Sessions für Cart/User-State
6. SSR nur dort, wo wirklich nötig

---

66. Die wichtigsten Astro-Dateien im Alltag

Fast immer relevant

- `astro.config.mjs`
- `src/pages/`
- `src/components/`
- `src/layouts/`
- `src/styles/`
- `src/content.config.ts`
- `public/`

Oft relevant

- `src/actions/index.ts`
- `src/middleware.ts`
- `src/live.config.ts`
- `tsconfig.json`

---

67. Mini-Referenz: Was verwende ich wofür?

Ich will eine Seite erstellen

→ `src/pages/*.astro`

Ich will gemeinsame Struktur

→ `src/layouts/*.astro`

Ich will Wiederverwendung

→ `src/components/*.astro`

Ich will Markdown-Blogposts

→ Content Collections + `src/content/`

Ich will React-Komponente einbauen

→ `@astrojs/react` + Import in `.astro`

Ich will Interaktivität

→ `client:*`

Ich will SSR

→ Adapter + `export const prerender = false`

Ich will serverseitig dynamische Teilbereiche

→ `server:defer`

Ich will API-Routen

→ `src/pages/api/*.ts`

Ich will Form-Handling modern und type-safe

→ Actions

Ich will User-State serverseitig

→ Sessions

Ich will Auth / Request-Kontext

→ Middleware + `locals`

Ich will Bilder optimieren

→ `<Image />` / `<Picture />`

68. Die 20 wichtigsten Dinge, die du dir merken solltest

1. Astro ist für **content-driven websites** gebaut.
2. Standardmäßig ist Astro **server-first**.
3. Standardmäßig sendet Astro **kein unnötiges Client-JS**.
4. Interaktive Komponenten brauchen `client:*`.
5. Kleine dynamische Server-Bereiche gehen mit `server:defer`.
6. Routing ist **dateibasiert**.
7. `src/pages/` ist Pflicht.
8. Astro-Komponenten sind `.astro`.
9. Props kommen aus `Astro.props`.
10. Kind-Inhalte kommen über `<slot />`.
11. Styles sind standardmäßig **scoped**.
12. Content Collections sind der Standard für strukturierte Inhalte.
13. Bilder möglichst in `src/` lagern.
14. Optimierte Bilder mit `<Image />` oder `<Picture />`.
15. Für SSR brauchst du einen **Adapter**.
16. Für Cookies/Headers/Request musst du SSR oder on-demand nutzen.
17. Actions sind oft besser als klassische API-Routen für Form- und Client-Server-Calls.
18. TypeScript bringt dir in Astro sehr viel, auch wenn du wenig TS schreibst.
19. Markdown kann direkt Seiten erzeugen oder über Collections geladen werden.
20. Astro ist am stärksten, wenn du **so wenig JS wie möglich** in den Browser schickst.

Revision #1
Created 2026-06-29 12:02:49 UTC by art10m
Updated 2026-06-29 12:04:28 UTC by art10m