Astro & Vue: Architecture, Hydration, and Pragmatic Frontend Design

Mateusz Jabłoński

Who am I?

- programmer since 2011
- mainly: Javascript / Typescript / Java / formerly: PHP
- trainer / mentor since 2016
- privately a father and husband
- recently also an actor in a local theater and a player of paper RPGs

Arrangements

- Goal and agenda
- Mutual expectations
- Questions and discussion
- Flexibility
- Openness and honesty

Agenda, what's coming up?

1. Astro Introduction
2. Astro Fundamentals & Layouts
3. Vue 3 – Aligning Mental Model
4. Astro Islands & Hydration Strategies
5. Component Slicing
6. Hydration Issues & Debugging in Nuxt
7. Server Logic: Nuxt APIs & Astro Actions
8. State Management - Minimal & Pragmatic
9. Performance & Bundle Control

github.com/matwjablonski/astro-vue-0126

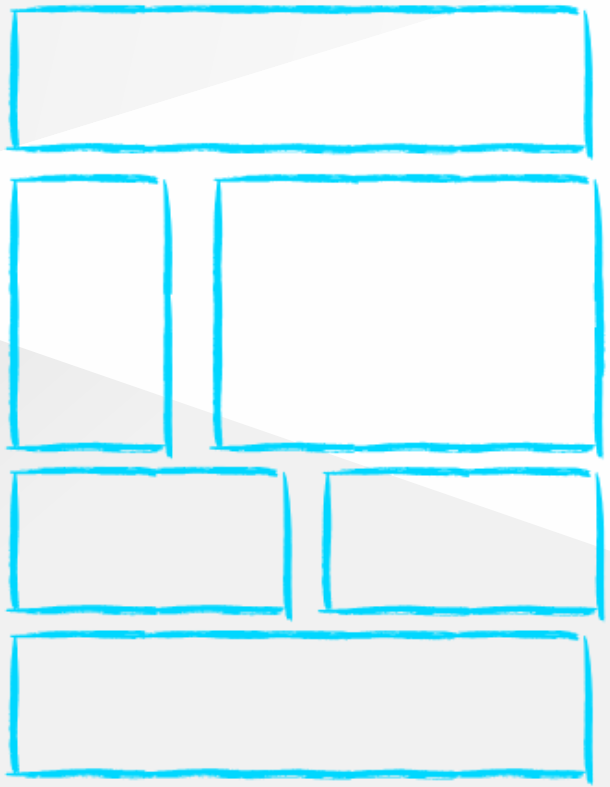
Astro Introduction

Island Architecture

- an approach to building applications where interactive parts of the page (so-called "islands") are rendered and managed independently
- the page consists of pre-generated HTML, where interactive components act as standalone modules
- the main page (content) is pre-generated (like in SSG), which ensures fast loading times
- dynamic elements, such as forms or carousels, are loaded and rendered individually on the client side
- each island is an independent module with its own code and state
- islands are loaded only when they are visible or needed, which reduces the initial page load time

Only specific islands are rehydrated, while the rest remain static.

SSR

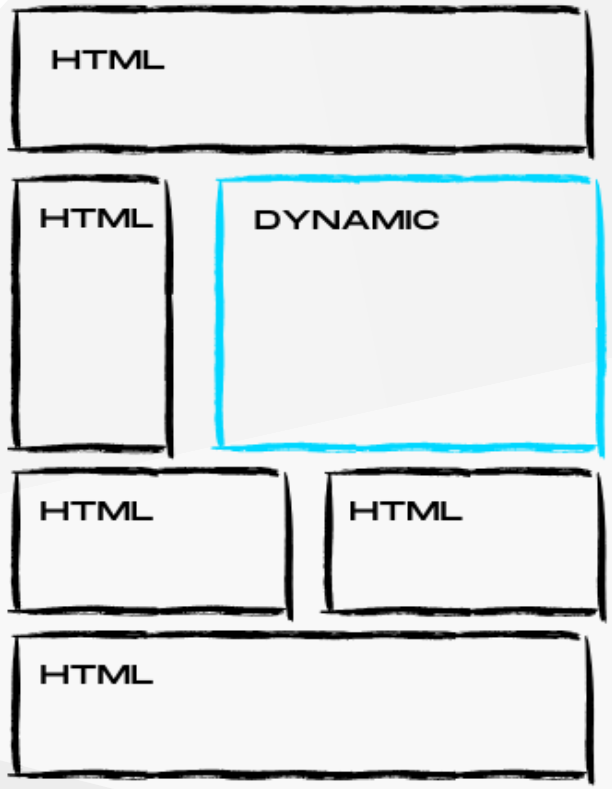


Rendering and hydrating all elements at the same time.

Progressive hydration

Rendering in the first phase, hydration as needed.

Island architecture



Static elements are returned as HTML from the server. JavaScript is needed only for islands.

Origin of Island Architecture

- the term "Islands Architecture" was coined by Katie Sylor-Miller (Frontend Architect at Etsy) in 2019 during a conversation with Jason Miller (Google) about an approach to building websites that combines static site generation with interactive components loaded on demand
- the concept was further described and popularized by Jason Miller (creator of Preact) in a blog post in 2020
- this architecture emerged as a response to the problem of "massive JavaScript" in traditional Single Page Applications (SPAs) and the problematic rehydration of entire pages in frameworks like Next.js or Nuxt.js

Progressive Enhancement on Steroids

- **Selective Hydration** - instead of hydrating the entire DOM tree (which blocks the main thread), the framework "hydrates" only specific, isolated components (islands), allowing for faster interactions with the rest of the page
- **interactivity on demand** - the developer decides when an island should become interactive (e.g., immediately, on mouse hover, or only when it appears in the viewport – `client:visible`)
- **total isolation** - an error inside one island (e.g., in the shopping cart) should not break the rest of the page, which remains a usable HTML document
- **reduced TBT (Total Blocking Time)** - by avoiding heavy rehydration of the entire page, the browser becomes responsive faster

Alternatives to Astro

Tool	Description
Fresh	framework for the Deno environment, sends 0 bytes of JS by default, islands are created in Preact
Marko	created by eBay, one of the pioneers of partial hydration, very efficient in e-commerce
Elder.js	Svelte-based framework focused on SEO, emphasizing building fast sites with islands
Enhance	framework focused on standard Web Components as islands
Qwik	although technically uses the Resumability mechanism, ideologically close to islands - allows interactivity without full rehydration

Who created Astro?

- The Astro Technology Company was announced as the organizational entity supporting the project
- the main creator and project leader is Fred K. Schott
- the first public version was announced on June 8, 2021
- Fred K. Schott is known "for his fight against SPA applications" and promoting the HTML-first approach in building frontend applications

HTML–first frontend architecture

- architecture based on the assumption that HTML is the fundamental building block of frontend applications
- JavaScript is used only where it is absolutely necessary
- the amount of JavaScript code is minimized, leading to faster page loading and better performance
- this approach promotes better accessibility and SEO, as content is available even without JavaScript

SUMMARY

HTML is the main and complete artifact of a frontend application, and JavaScript is an optional addition, not the foundation of the page's operation.

Key paradigm shift

SPA

The dominant model in building frontend applications for years has been SPA (Single Page Application). SPA assumes that the entire application is loaded as a single page, and user interactions are handled dynamically by JavaScript.

Server	Browser
large JS bundle	downloads and executes JS
minimal HTML	generates HTML on the fly

IMPORTANT

UI exists only after JS execution

HTML-first

HTML-first should not be treated as an optimization of SPA. It is a fundamental architectural difference.

Server	Browser
complete HTML	immediately renders UI
ready to use	JS optionally adds interaction

IMPORTANT

UI exists before JS runs

What does this change in practice?

1. HTML becomes a contract
2. Rendering stops being a "side effect of JS"
3. Interactivity stops being global

HTML as a contract defines the structure and content of the application. HTML must always work, JS is optional — therefore it should not be required for the existence of the UI.

HTML is the result of server-side rendering, JS no longer generates HTML on the fly. Rendering becomes predictable and independent of the client environment.

In HTML—first, interactivity is local and JS is precisely dosed.

HTML-first is not the absence of JavaScript

HTML-first changes the structure of responsibility. HTML-first does not eliminate JS, does not exclude frameworks, does not take us back to the era of static pages.

Layer	Role
HTML	Structure and accessibility
CSS	Appearance
JS	Interaction and logic

Why Astro at all?

Problem we are solving:

- more and more JS in places where it is not needed
- SPA as the default choice
- hydration is expensive and often unnecessary
- high performance does not translate to better UX

IMPORTANT

Astro is not "another SPA framework".

Main assumption of Astro

- HTML as the final artifact
- Astro renders HTML on the server
- JS is optional
- interactive components are loaded only where needed
- "island architecture" approach

What Astro is NOT?

Astro:

- is not an SPA framework
- is not a replacement for React/Vue/Svelte
- does not impose a single FE runtime

To orchestrator, not a monolith.

Monolith vs Orchestrator

Feature	Monolith (e.g. Nuxt / Next / SPA)	Orchestrator (e.g. Astro)
Dependency	limited to one ecosystem	technology agnostic - tool chosen for specific tasks
JS Control	the whole site is one big app, hard to disable JS for specific parts	default zero JS, developer decides which island needs JS
Architecture	everything is tightly coupled	components are isolated and independent
Hydration	usually full hydration at start	selective hydration - only what is specified as `client:visible`
Cognitive Load	you need to know very well the specifics of a given framework and its meta-framework	you focus on web standards (HTML/CSS), and frameworks are just plugins
Technical Debt	changing the framework requires rewriting the entire application from scratch	you can gradually replace components (e.g. from Vue to Svelte) without touching the rest

What happens when you treat Astro like a monolith?

Common mistakes:

- trying to transfer SPA logic to Astro
- "everything as an island"
- global state "because it's expected to add"
- Vue components doing layouts

Effect:

- larger bundle
- more hydration than necessary
- architectural chaos

Anti-pattern: Treating Astro as a monolith

The following example will not work. Why?

- the `.astro` file tries to handle application logic, but `.astro` is not a Vue (or other framework) component
- logic is mixed - render vs interaction

Example code:

```
---
import { ref } from 'vue'

const count = ref(0)

function increment() {
  count.value++
}
---

<button @click="increment">
  Clicked {{ count }} times
</button>
```

Correctly: separation of concerns

Example code:

```
---
import Counter from '../components/Counter.vue'
---
<Counter client:load />
```

Example code:

```
<template>
  <button @click="increment">
    Clicked {{ count }} times
  </button>
</template>
<script setup>
    import { ref } from 'vue'
    const count = ref(0)
    function increment() { count.value++ }
</script>
```

Anti-pattern: Everything is an island

Example code:

```
---
import Header from './components/Header.vue';
import Hero from './components/Hero.vue';
import Features from './components/Features.vue';
import Footer from './components/Footer.vue';
import NewsletterForm from './components/NewsletterForm.vue';
---
<Header client:load />
<Hero client:load />
<Features client:load />
<NewsletterForm client:load />
<Footer client:load />
```

What happened here?

- the whole site requires JS
- no benefits from Astro
- global hydration "through the back door"

Correctly: Selective islands

Example code:

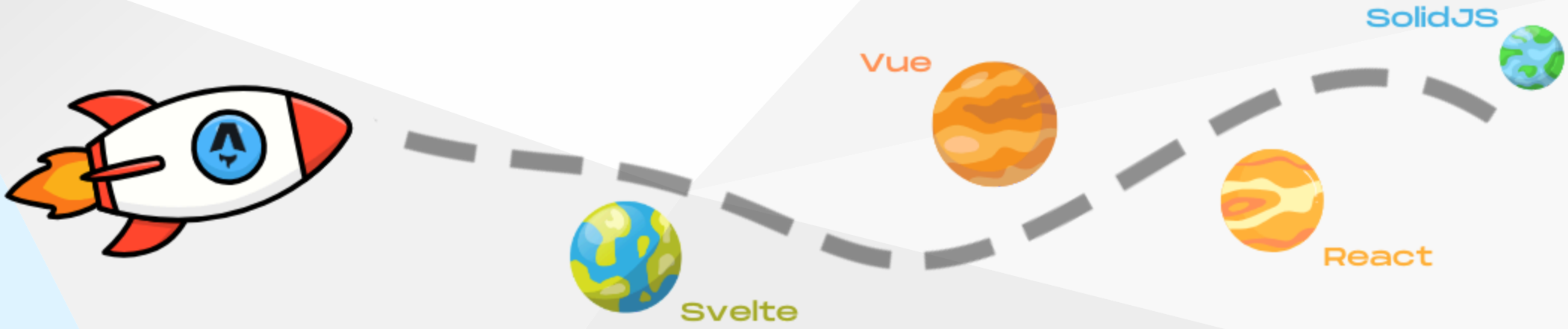
```
---
import Header from './components/Header.vue';
import Hero from './components/Hero.vue';
import Features from './components/Features.vue';
import Footer from './components/Footer.vue';
import NewsletterForm from './components/NewsletterForm.vue';
---
<Header />
<Hero />
<Features />
<NewsletterForm client:visible />
<Footer />
```

What happened here?

- only the form requires JS
- the rest of the site is static and fast
- full utilization of Astro's potential

Astro along with frontend frameworks

Astro is not tied to any frontend framework. You can use any of them or even several at the same time. Astro allows integration with popular frameworks such as React, Vue, Svelte, Solid, or Angular.



Can you mix frameworks in one project?

Yes, technically you can. Astro allows it, but:

- increases cognitive load
- makes project maintenance harder
- may lead to inconsistencies in styles and component behaviors
- usually not worth doing in practice

IMPORTANT

Astro allows mixing different frameworks mainly for migration purposes (you can slowly move a project from Vue to React) or in specific cases of using ready-made libraries.

How to choose a frontend framework for Astro?

Criterion	Best choice
Small project, simplicity	Svelte, Solid
Large project, complexity	React, Vue
Performance and speed	Solid, Svelte
Rich ecosystem	React, Vue
Ease of learning	Vue, Svelte
Community support	React, Vue

Astro chooses where JS runs. The framework chooses how JS works.

Project structure in Astro

The standard structure of an Astro project looks as follows:

Example code:

```
.
├── public/
├── src/
│   ├── components/
│   ├── layouts/
│   ├── pages/
│   └── assets/
├── astro.config.mjs
├── package.json
└── tsconfig.json
```

Directory/File	Description
`public/`	directory for static assets such as images, fonts, etc., the contents of this directory are served directly
`src/`	main source directory of the project
`components/`	directory for components
`layouts/`	directory for page layouts
`pages/`	directory for application pages
`assets/`	directory for assets such as images, fonts, etc.
`astro.config.mjs`	Astro configuration file
`package.json`	file managing project dependencies and scripts
`tsconfig.json`	TypeScript configuration file

What is a page in Astro?

In Astro, a **page** is a special type of component that represents a single page in the application. Page files are located in the `src/pages/` directory and are automatically mapped to corresponding URL paths in the application.

For example, the file `src/pages/about.astro` will be available at the `/about` address in the application.

IMPORTANT

Astro does not provide a specific `Link` component for navigation. Use standard HTML `a` elements for navigation between pages.

Supported file types

- `.astro` - Astro components
- `.js` / `.ts` - JavaScript/TypeScript files
- `.md` / `.mdx` - Markdown/MDX files
- `.html` - HTML files

What should a page file contain?

- a `page` should provide a complete HTML structure for the given page
- Astro will automatically add the `<!DOCTYPE html>`, `<html>`, `<head>`, and `<body>` elements if they are not defined in the page file
- a `page` can contain a frontmatter section

What is a layout file in Astro?

- a `layout` is a special-purpose component that defines a common structure and layout for multiple pages in an application
- Layout files are located in the `src/layouts/` directory
- each layout can contain a `<slot />`, which acts as a placeholder for page-specific content

Task 1

15 minutes

1. Create a new Astro project with Vue 3 as the frontend framework
2. Create a new Contact page using the default layout

Example code:

```
npx create-astro@latest my-astro-vue-app
cd my-astro-vue-app
npm install
npx astro add vue
```

Astro Expression Language

Astro allows using:

- logical operators
- conditions (`&&` , `?:`)
- array mapping
- simple JS expressions

Example code:

```
<ul>
  {items.map(item => (
    <li>{item.label}</li>
  ))}
</ul>
```

IMPORTANT

It is for rendering HTML, not application logic.

Expressions other than all

To define expressions in Astro, we can use the `{}` syntax. Inside the curly braces, we can use:

1 Variables

Example code:

```
<h1>{title}</h1>
<p>{user.name}</p>
```

2 Logical expressions

Example code:

```
    {isLoggedIn && <p>Welcome back!</p>}
    {count > 0 ? <span>{count}</span> : <span>Empty</span>}
```

3 Array mapping

4 Mathematical expressions

Example code:

```
<p>Total: {price * quantity}</p>
```

5 Functions

Example code:

```
<p>{formatDate(post.date)}</p>
```

We can call functions defined in the frontmatter section or imported from other modules.

The syntax of expressions looks similar to JSX / Vue template, but it DOES NOT WORK the same way.

What is NOT in the Astro expression language?

- `@click`, `v-if`, `v-for`
- reactivity (`ref`, `computed`)
- lifecycle
- side effects

Astro does not respond to browser events.

Astro vs Vue

What in {}	Astro	Vue
Variables	✅	✅
Conditions	✅	✅
Mapping	✅	✅
Events	❌	✅
Reactivity	❌	✅
Runtime	❌	✅

Astro uses JavaScript as an expression language for generating HTML, not as a runtime programming language.

Special Astro directives

Directive	Description
`set:html`	injects raw HTML
`set:text`	injects raw text
`class:list`	conditionally adds CSS classes
`is:raw`	informs the compiler that the element is raw and should not be processed
`is:global`	indicates that the style or script is global and should not be scoped
`is:inline`	indicates that the style or script should be inlined directly in the HTML at the location where it was added

Example code:

```
---
const rawHtmlContent = "<strong>This is bold text</strong>";
const isActive = true, isDisabled = false;
---
<div set:html={rawHtmlContent}></div>
<div class:list={{ active: isActive, disabled: isDisabled }}></div>

<code is:raw>
  {isActive ? "Active" : "Inactive"}
</code>

<style is:global>
  body {
    margin: 0;
    font-family: Arial, sans-serif;
  }
</style>

<script is:inline>
  console.log("This script is inlined in the HTML");
</script>
```

Anti-patterns

Event handling

Example code:

```
<button onClick={handleClick}>Click</button>
```

Astro does not handle events in the template.

Reactivity / runtime JS

Example code:

```
{count++}
{ref.value}
```

Lack of reactivity and lifecycle.

Side effect operations

Example code:

```
{saveToDatabase()}
{fetchData()}
```

Fetch and logic - these should go to the frontmatter, not the template.

Browser API

Example code:

```
{window.innerWidth}
```

The template runs on the server, not in the browser.

Task 2

20 minutes

1. On the /contact page, display:
   - the page title: Contact
   - a list of contact channels (email, phone, working hours) rendered using `.map()`
   - a conditional status: "Open now" / "Closed" (based on a boolean variable)
   - display the working hours and information about how many hours until closing if currently open (use a numeric variable) or information about what time it will open if currently closed (use a numeric variable)
2. Place all data and helper functions in the page frontmatter and they must not use the browser API.

Astro Fundamentals & Layouts

Frontmatter vs Template

Frontmatter is a section of code placed at the beginning of an `.astro` file, surrounded by triple dashes `---`. It is used to define variables, import components, and execute JavaScript logic before rendering the template.

Template is the part of the `.astro` file that comes after the frontmatter section. It contains HTML code and Astro-specific syntax that defines the structure and appearance of the component or page.

Example code:

```
---
// This is frontmatter
import MyComponent from '../components/MyComponent.astro';
const title = "Welcome to Astro!";
---
<!-- This is template -->
<html>
  <head>
    <title>{title}</title>
  </head>
  <body>
    <MyComponent />
  </body>
</html>
```

Frontmatter – the brain of your component

- contains JavaScript logic (in `.astro` files)
- allows data fetching (Fetching) - you can use `await fetch()` directly in this section, and Astro will wait for the data before sending the page to the browser
- by default, Frontmatter supports TypeScript, so you can define interfaces and types

IMPORTANT

The frontmatter section runs only on the server. You cannot use browser APIs like `window` in it. The code inside Frontmatter never reaches the client browser.

Frontmatter in Markdown

Feature	In `.astro` files	In `.md` / `.mdx` files
Language	Full JavaScript / TypeScript	YAML format (data only)
Logic	You can write functions, loops, fetch	Only key-value pairs
Purpose	Building component logic	Metadata (title, date, author)

Example code:

```
---
title: "How to learn Vue in a weekend"
date: 2024-05-20
author: "John Doe"
tags: ["vue", "javascript", "frontend"]
draft: false
image: "/images/hero-vue.jpg"
description: "A short guide on how to start your journey with Vue.js and not go crazy."

---

# Here begins the post content

This is regular text in Markdown format. The above data (between the `---` lines) will not be displayed directly on the page, but Astro can use it to build lists of posts or SEO meta-tags.
```

How does Astro see this data?

In Astro, you can extract this data using the `Astro.glob()` function or through the `Content Collections` library.

Example code:

```
---
// Example of using Content Collections
import { getEntry } from 'astro:content';
const post = await getEntry('blog', 'how-to-learn-vue');
---

<article>
  <h1>{post.data.title}</h1> <p>Author: {post.data.author}</p>
  <img src={post.data.image} alt={post.data.title} />
</article>
```

TIP

In Markdown, frontmatter is static. If you need logic in it (e.g., calculations or imports), you must change the file extension to `.mdx` .

Astro.glob()

`Astro.glob()` is an older (deprecated) but still very useful method for quickly fetching multiple files at once. We use it when we don't want to configure full Content Collections, and just need to import a group of files from a specific folder.

Example code:

```
---
// We fetch all .md files from a specific folder
const posts = await Astro.glob('./posts/*.md');
const sortedPosts = posts.sort((a, b) =>
  new Date(b.frontmatter.date).getTime() - new Date(a.frontmatter.date).getTime()
);
const recentPosts = sortedPosts.slice(0, 3);
---
<ul>
    {recentPosts.map(post => (
        <li>
            <a href={post.url}>
                {post.frontmatter.title}
            </a>
            <span> - {post.frontmatter.date}</span>
        </li>
    ))}
</ul>
```

What does Astro.glob() return?

Each object in the list returned by `Astro.glob()` contains:

- `frontmatter` - all the data from the top of the `.md` file
- `url` - automatically generated URL (if the file is located in `src/pages/`)
- `file` - full system path to the file
- `content` - a component that you can render as `<post.Content />`

Astro.glob() vs getCollection()

Feature	`Astro.glob()`	`getCollection()` (Content Collections)
Where does it look?	Any folder (you provide the path)	Only in `src/content/`
Validation	None (you risk data errors)	Yes (checks format via `Zod`)
Use case	Quick prototypes, small projects	Professional, large websites
Performance	Good	Very high (optimized)

Validation Configuration

- Zod is by default integrated with Content Collections
- `src/content/config.ts` is the file where you should define validation schemas for each collection
- validation happens during the build process, so data errors will be caught early

Example code:

```ts
// src/content/config.ts
import { defineCollection, z } from 'astro:content';

const blog = defineCollection({
  type: 'content', // we inform that these are .md / .mdx files
  schema: z.object({
    title: z.string(),                  // Title must be a string
    date: z.date(),                     // Date must be a valid date format
    author: z.string().default('Anonymous'), // If no author, set 'Anonymous'
    tags: z.array(z.string()),          // Tags must be a list of strings
    isDraft: z.boolean().optional(),    // This field is optional
  }),
});
// We export the collection under the name 'blog'
export const collections = { blog };
```

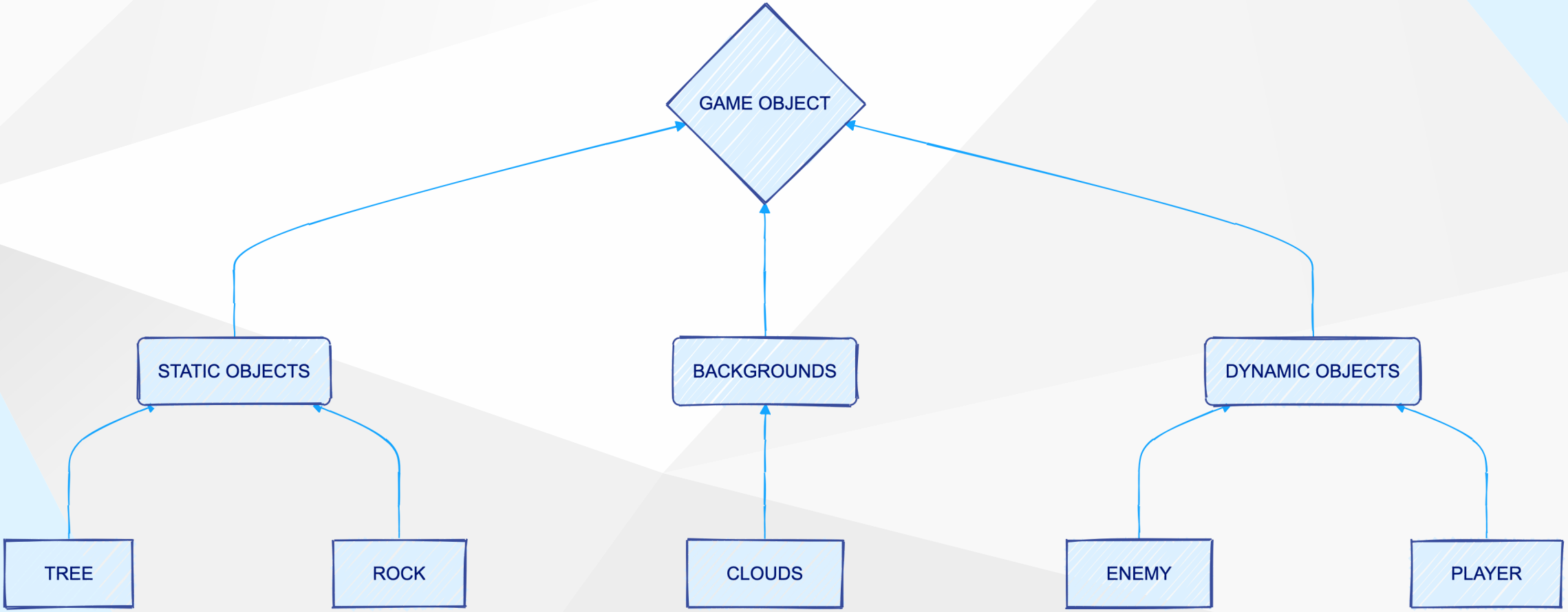
Task 3

20 minutes

1. Add `Content Collections` for the blog collection:
   - create a directory: `src/content/blog/`
   - add at least 3 .md files with fields: `title`, `date`, `author`, `tags`, `description`
2. Configure validation in `src/content/config.ts`:
   - use defineCollection and zod
   - `title`, `date`, `author` required
   - `tags` as z.array(z.string()), `description` as z.string()
3. Create a page `src/pages/blog.astro` that:
   - fetches entries in frontmatter using `getCollection('blog')`
   - sorts them in descending order by date and renders only the 3 most recent entries

Component Architecture

- components can be used in multiple applications
- shortens the time to build new applications and reduces project costs - by using existing components
- by using existing components, first of all, we do not have to write them from scratch, secondly test and document them
- individual components can be created in parallel
- by properly specifying the interfaces of individual components, it is easy to outsource the implementation of components
- supports so-called application modifiability - specific functionality is focused on dedicated software components, so in case of the need to make a change, this process usually concerns the modification of one or at most a few components, not the entire application
- coherence of components supports modifiability - the greater the coherence, the easier it is to modify the application


GAME OBJECT
STATIC OBJECTS
BACKGROUNDS
DYNAMIC OBJECTS
TREE
ROCK
CLOUDS
ENEMY
PLAYER

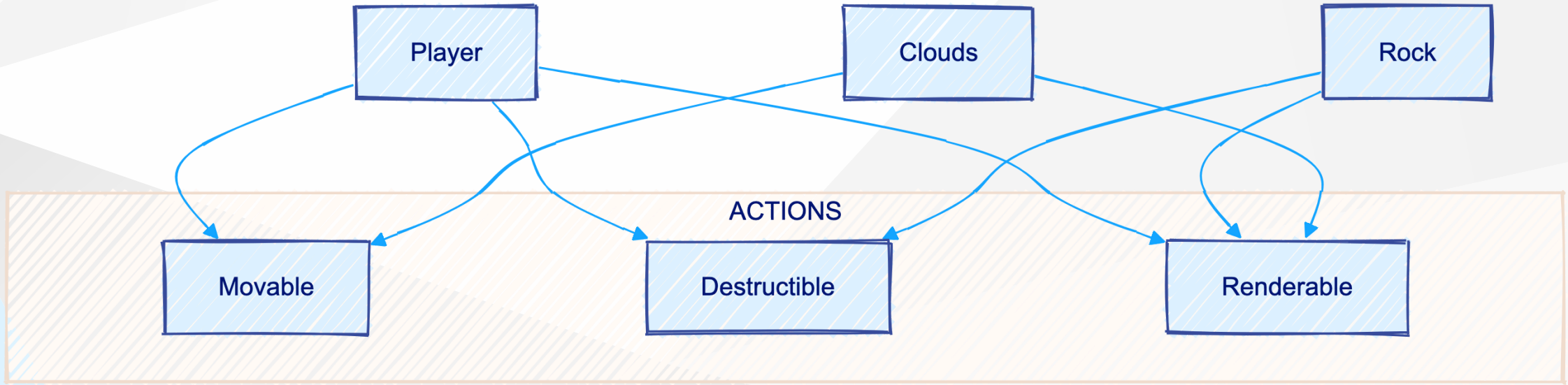
Loose Coupling

In an ideal architecture, components should know as little about each other as possible. The `Player` component should not know how the `Tree` component is implemented. **Communication happens through defined contracts (interfaces).**

This allows one component to be replaced with another (e.g., changing `Tree` from a 2D version to a 3D version) without touching the rest of the game logic.

Composition over Inheritance

- modern architecture (e.g., ECS in games `Entity-component-system` or components in Vue/Astro) promotes composition
- instead of creating complex class hierarchies, we build applications from small, isolated components
- components are combined (composed) into larger functional units
- this facilitates code reuse and testing
- changes in one component have minimal impact on the rest of the system


Player
Clouds
Rock
ACTIONS
Movable
Destructible
Renderable

Contract

- every component should have a clearly defined interface (contract)
- the contract specifies what data the component accepts
- the contract should be explicit
- the contract should follow the Open/Closed principle (open for extension, closed for modification) - the component should be open for extension (through props/slots), but closed for modification (we do not need to change its source code to change its behavior)
- the component's API should be stable and well-documented
- changes in the API require discipline: `semver`, `deprecations`, `changelog`

Component architecture works best when components have high cohesion, low coupling, and stable contracts, and their reusability is supported by versioning, contract testing, and documentation.

Astro Components

1 Astro component is the basic building block of an Astro application

Components allow you to create reusable UI fragments that can be rendered on the server and/or client.

2 File with the `.astro` extension.

The component in the template part can use variables through `{}` .

3 Props and Slots

Astro components can accept input data (props) and contain dynamic content (slots).

4 Importing Components

Astro components can import and use other components, both Astro and from frontend frameworks.

5 Client Directives

Rendering of Astro components can be controlled using directives such as `client:load`, `client:idle`, `client:visible`, and `client:media`.

Task 4

15 minutes

1. Create a `Header.astro` component in the src/components/ directory.
2. The component should include:
   - a page header (`<header>`)
   - the service name (e.g., MediaHub)
   - navigation with links to: **Home (/)** , **Contact (/contact)** , **Blog (/blog)**
3. Use props to pass:
   - the service name (title)

Ways of Rendering in Astro

In Astro, there are two levels of rendering: output mode for pages and client:* for components (islands).

Rendering Mode	Description	Use Case
`static`	SSG, default mode, components are rendered at build time and delivered as static HTML	pages that do not require interactivity
`server`	SSR, components are rendered on the server on demand, with each request	dynamic pages that need to be updated in real-time

IMPORTANT

Changing the rendering mode can be done in the configuration file `astro.config.mjs` or directly in the `.astro` files.

How to apply SSR in Astro?

Example code:

```
---
// This code will run on the SERVER on every page refresh,
// if you have SSR enabled (output: 'server')
const cookie = Astro.cookies.get("user-session");
const response = await fetch(`https://api.com/user/${cookie.value}`);
const userData = await response.json();
---
<h1>Hello, {userData.name}!</h1>
<p>Your data was fetched directly from the database just now.</p>
```

Example of using rendering modes in Astro

Example code:

```
---
import InteractiveMap from '../components/InteractiveMap.astro';
import StaticContent from '../components/StaticContent.astro';
---
<StaticContent />
<InteractiveMap client:visible />
<style>
  /* Component styles */
</style>
```

Example code:

```
---
import UserProfile from '../components/UserProfile.astro';
const title = "Lista użytkowników";
const response = await fetch('https://api.example.com/users');
const users = await response.json();

const activeUsers = users.filter(u => u.isActive); // will run on the server
const isLoggedIn = true;
---

<h1>{title}</h1>

{isLoggedIn ? (<p>Welcome back!</p>) : (
  <p>Please log in to see more.</p>
)}

<ul>
  {activeUsers.map((user) => (
    <li>
      <UserProfile name={user.name} />
      {user.isAdmin && <span> (Admin)</span>}
    </li>
  ))}
</ul>

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

Hybrid Mode

Astro allows mixing rendering modes within a single application. You can have both static and dynamic pages in the same project.

Example code:

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

export default defineConfig({
  output: 'hybrid',
});
```

Static Page (SSG)

Example code:

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

<h1>Welcome</h1>
<p>This page is fully static.</p>
```

Dynamic Page (SSR)

Example code:

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

const session = Astro.cookies.get('session');

if (!session) {
  return Astro.redirect('/login');
}

const response = await fetch(
  `https://api.example.com/user/${session.value}`
);
const user = await response.json();
---
<h1>Hello, {user.name}</h1>
<p>Your email: {user.email}</p>
```

Client Directives for Components

Client Directive	Description	Use Case
`client:load`	component is loaded and rendered on the client immediately after the page loads	interactive components that need to be available right away
`client:idle`	component is loaded when the browser is idle	components that are not critical for initial rendering
`client:visible`	component is loaded when it becomes visible in the browser viewport	components below the initial viewport
`client:media="(min-width: 800px)"`	component is loaded according to the media query	components tailored to specific screen sizes

Props – what are they?

Props in Astro are input data passed to a component at the time of HTML rendering. They allow dynamic customization of the content and behavior of components.

Key features:

- props are static during render
- they are not reactive
- they do not change on the client side
- they are used to configure HTML, not application behavior

SUMMARY

Props in Astro are part of the rendering process, not the runtime.

How do we pass props?

Example code:

```
<MyCard title="Astro" />
<MyCard count={10} />
<MyCard user={user} />
```

Props can be: string, number, object, array, result of a JS expression.

Receiving props in a component

Example code:

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

<h2>{title}</h2>
<p>{count}</p>
<p>{user.name}</p>
```

IMPORTANT

Props are available in the frontmatter section through `Astro.props` .

Typing props

Example code:

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

const { title, featured = false } = Astro.props as Props;
---

<h2>{title}</h2>
{featured && <span>This is featured</span>}
```

Summary

1. Props are one-time
2. We cannot observe their changes
3. They do not cause re-render

IMPORTANT

If something needs to change → it is no longer a problem for Astro, but for islands (Vue/React).

Slot

Slot is a special place in an Astro component that allows you to inject content from outside. With slots, you can create more flexible and reusable components that can accept different content depending on the context in which they are used.

Default and Named Slots

- **Default slot** - allows you to insert content without specifying a name. This content will be placed where the `<slot />` is in the component.
- **Named slot** - allows you to define multiple places in the component where each part of the content can be inserted into a specific slot using the `slot` attribute. Named slots are defined using `<slot name="name" />`.

Example code:

```
---
// Component with slots
---
<div class="card en">
  <header>
    <slot name="header">Default header</slot>
  </header>
  <main>
    <slot>Default content</slot>
  </main>
  <footer>
    <slot name="footer">Default footer</slot>
  </footer>
</div>
```

Example code:

```
---
import Card from '../components/Card.astro';
---
<Card>
  <h2 slot="header">Custom header</h2>
  <p>This is custom card content.</p>
  <p slot="footer">Custom footer</p>
</Card>
```

set:html and set:text with Slots

By default, code inserted via `set:html` and `set:text` will be rendered in the default slot location.

Example code:

```
---
import Card from '../components/Card.astro';
const rawHtml = '<strong>This is bold text</strong>';
---
<Card set:html={rawHtml}></Card>
```

We can also use `set:html` and `set:text` with named slots:

Example code:

```
---
import Card from '../components/Card.astro';
const footerText = 'This is the footer set by set:text';
---
<Card>
    <div slot="footer" set:text={footerText}></div>
</Card>
```

Task 5

20 minutes

1. Create a component: `src/components/ArticleCard.astro`
2. The component should accept props: **title: string**, **excerpt?: string**, **highlighted?: boolean (default false)**
3. The component should have slots: `default slot` – main content and `named slot meta` – author/date/tags
4. In the component, use directives:
   - `class:list` for conditionally adding a class for highlighted
   - `set:text` for injecting excerpt (without XSS risk)
   - `set:html` for injecting badgeHtml (with a comment that this requires a trusted source)
   - `is:global` for global styles (e.g., .sr-only or link resets)
   - `is:inline` for a short script (e.g., console.log('ArticleCard rendered')) – for demonstration only
5. On the `/blog` page, use ArticleCard at least 2 times

Layout is more than just a frame

- `Layout` is a component that defines a stable structure in which only specific content fragments change
- The page "promises" to provide data (e.g., title, SEO description), and the Layout "promises" to place them in the appropriate HTML tags and ensure a consistent appearance

Separation of responsibilities:

Responsibility	Page	Layout
Providing content	Yes	No
Providing metadata (title, description)	Yes	No
HTML structure	No	Yes
Appearance (CSS)	No	Yes

Anatomy of the Contract (Props and Slot)

In Astro, the layout contract is realized through:

- **Interface Props** - definition of metadata (SEO)
- **Default Slot** - place for the main content of the page
- **Named Slots** - special places (e.g., sidebar, scripts section)

Example code:

```
---
// src/layouts/MainLayout.astro
interface Props {
  title: string;
  description?: string;
}
const { title, description = "Default description" } = Astro.props;
---
<html>
  <head>
    <title>{title}</title>
    <meta name="description" content={description} />
  </head>
  <body>
    <nav>Menu</nav>
    <main>
      <slot /> </main>
  </body>
</html>
```

Typing the Contract

Using TypeScript in Layouts, we enforce the programmer to provide the necessary data already at the coding stage.

Advantage: It is impossible to create a page without a title if the Layout requires it.
Compilation error: Astro will show an error if you forget a required prop

Example code:

```
---
// Example usage - if you don't provide a title, the IDE will highlight an error
import Layout from '../layouts/MainLayout.astro';
---
<Layout title="Home page">
  <p>Page content...</p>
</Layout>
```

Nested Layouts

- Layouts can be nested, allowing for the creation of more complex structures
- each layout can have its own contract (props and slots)

1 Top level: BaseLayout.astro

Example code:

```
---
// src/layouts/BaseLayout.astro
const { title } = Astro.props;
---
<html lang="pl">
  <head>
    <title>{title}</title>
  </head>
  <body>
    <nav>Main navigation</nav>
    <slot /> <footer>Global footer</footer>
  </body>
</html>
```

2 Middle level: BlogLayout.astro

This layout is intended only for the blog section. It uses `BaseLayout` but adds a `sidebar` .

Example code:

```
---
// src/layouts/BlogLayout.astro
import BaseLayout from './BaseLayout.astro';
const { frontmatter } = Astro.props;
---
<BaseLayout title={frontmatter.title}>
  <div class="blog-container en">
    <article>
      <header>
        <h1>{frontmatter.title}</h1>
        <p>Autor: {frontmatter.author}</p>
      </header>
      <slot />
    </article>
    <aside>
      <h3>Kategorie bloga</h3>
      </aside>
  </div>
</BaseLayout>
```

3 The Lowest level: post.md

The content file does not need to know about `BaseLayout`. It is only concerned with the contract of `BlogLayout`.

Example code:

```
---
layout: ../layouts/BlogLayout.astro
title: "How to nest layouts in Astro"
author: "John Doe"
---

This is the content of my post. It will be injected into `BlogLayout`,
which in turn will be injected into `BaseLayout`.
```

Why does this matter?

1. DRY principle - changes to common elements are made in one place, for the entire site
2. Specialization - we can create layouts dedicated to specific sections of the site
3. SEO support - data flows from the page level to the layout, making it easier to manage meta tags

Task 6

20 minutes

1. Prepare a base layout `BaseLayout.astro` that includes:
   - a header with the service name (e.g., AstroVueApp)
   - a footer with copyright information (e.g., © 2024 AstroVueApp)
   - a main section for the page content (use `<slot />` to insert page content)
2. Prepare a layout `BlogLayout.astro` that extends `BaseLayout.astro` and additionally includes:
   - a sidebar with blog categories (e.g., Technology, Lifestyle, Travel)
3. Modify the `/blog` page to use `BlogLayout.astro` as the page layout.

Vue 3 — Aligning Mental Model

Vue components vs Vue application

In SPA:

- component is part of the application
- long-lived runtime
- global context

In Astro:

- Vue component is an isolated island
- short-lived runtime
- no shared global context between islands

SUMMARY

Vue in Astro is a tool for interaction, not the foundation of architecture.

Vue Components (Composition API)

Example code:

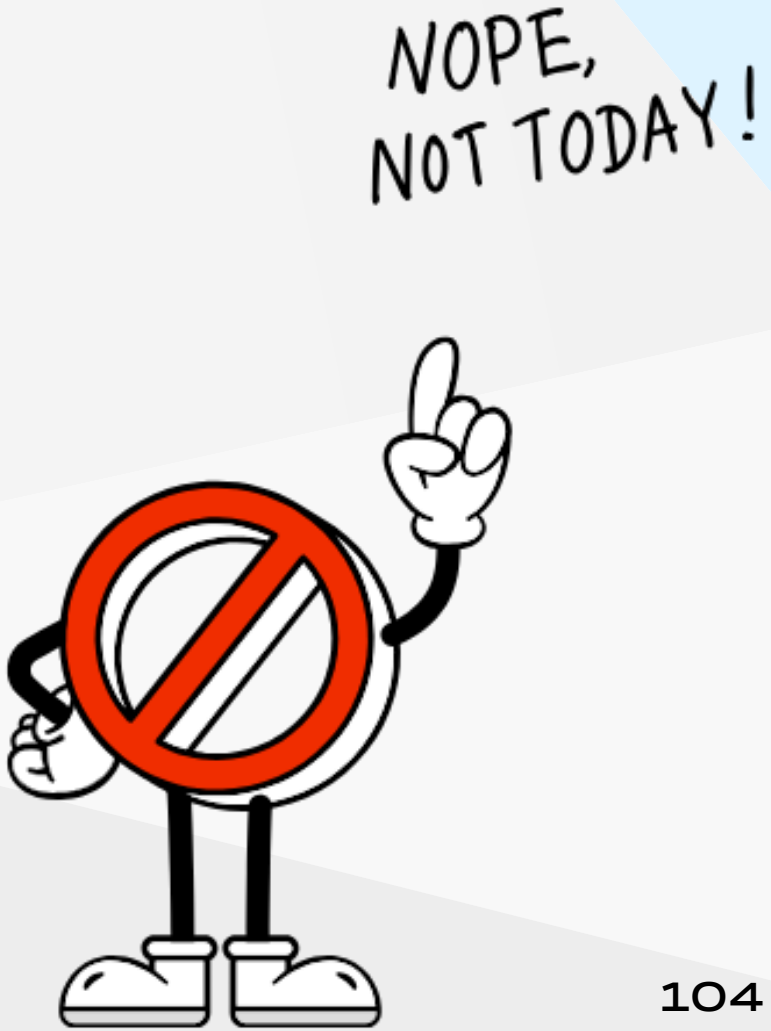
```
<script setup lang="ts">
import { ref } from 'vue'

const count = ref(0)

function increment() {
  count.value++
}
</script>
<template>
  <button @click="increment">
    Clicked {{ count }} times
  </button>
</template>
```

- no store
- no router
- no side-effects at the application level
- component does one thing

Don't count on it!

Vue component in Astro should not:

- manage navigation
- initialize global services
- hold application state



Vue Composables

Composables are the heart of reusability in Vue 3. We can think of them as "closed boxes of logic" that we can plug into different components.

In Astro, a composable most often represents a local use-case, not shared application state

Example code:

```
import { ref } from 'vue'
export function useCounter() {
  const count = ref(0)

  function increment() {
    count.value++
  }

  return { count, increment }
}
```

Example code:

```
<script setup lang="ts">
import { useCounter } from '../composables/useCounter'
const { count, increment } = useCounter()
</script>
<template>
  <button @click="increment">
    Clicked {{ count }} times
  </button>
</template>
```

Options API vs Composition API

Criterion	Options API ("option bags")	Composition API (Composable API)
Code organization	By types (data, methods, watch)	By responsibility / use-cases
Logic readability	Logic of one function scattered across the file	All logic in one place
Component scaling	Chaos grows quickly	Scales linearly
Refactoring	Difficult, risky	Simple, predictable
Logic reusability	Mixins (problematic)	Composables (pure functions)
Testability	Harder	Easy (pure functions)
TypeScript	Limited, often unreadable	Full support, strong typing

Criterion	Options API ("option bags")	Composition API (Composable API)
Reactivity	Often excessive (watch, sync state)	Explicit and controlled
Cognitive load	High for larger components	Low, code "reads from top"
Side-effects	Hidden in watch, mounted	Explicit, easy to find
Integration with Astro	Heavier, "SPA mindset"	Natural for islands
Recommendation for new projects	Rather no	Yes (default choice)
Code colocation	None	Full colocation (logic, reactivity, side-effects)

Vue in Astro is a tool for interaction, not the foundation of architecture.

Reactivity in Vue

Vue 3 introduces a new reactivity system based on `Proxy`, which is more efficient and flexible than the previous system based on `Object.defineProperty`. Reactivity in Vue allows automatic tracking of changes in data and updating the view in response to those changes.

How did Object.defineProperty work?

In Vue 2, reactivity was implemented using `Object.defineProperty`, which had some limitations:

- you couldn't add new properties to an object after its creation without losing reactivity
- you couldn't track changes in arrays (e.g., adding or removing elements)
- performance was lower with large objects or arrays

Example code:

```
const obj = {};
Object.defineProperty(obj, 'message', {
    get() {
        console.log('Getting message');
        return 'Hello, Vue 2!';
    },
    set(value) {
        console.log('Setting message to', value);
    }
});
obj.message; // Getting message
obj.message = 'Hi!'; // Setting message to Hi!
```

Proxy in JS

`Proxy` is an object that allows you to intercept and define custom behavior for fundamental operations on objects, such as reading, writing, deleting properties, etc.

Example code:

```
const target = { message: "Hello, World!" };
const handler = {
  get: (obj, prop) => {
    console.log(`Getting property: ${prop}`);
    return obj[prop];
  },
  set: (obj, prop, value) => {
    console.log(`Setting property: ${prop} to ${value}`);
    obj[prop] = value;
    return true;
  }
};
const proxy = new Proxy(target, handler);
console.log(proxy.message); // Getting property: message
proxy.message = "Hello, Vue!"; // Setting property: message to Hello, Vue!
```

Proxy allows you to "intercept" operations on an object and add your own logic before those operations are executed.

Syntax system

Example code:

```
<script>
import { reactive, computed, watch } from 'vue'

export default {
  setup() {
    const state = reactive({ count: 0 })
    const doubleCount = computed(() => state.count * 2)

    watch(
      () => state.count,
      (newVal, oldVal) => {
        console.log(`Count changed from ${oldVal} to ${newVal}`)
      }
    )

    function increment() {
      state.count++
    }

    return { state, doubleCount, increment }
  },
}
</script>
```

Example code:

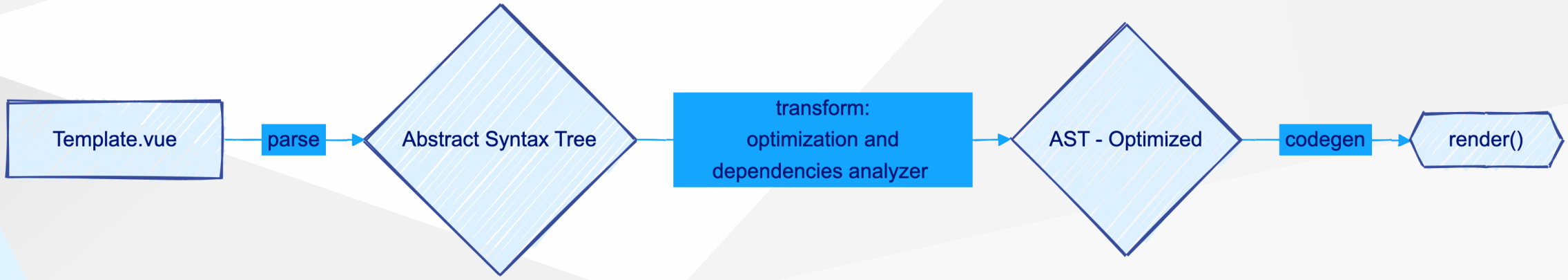
```
<script setup>
    import { reactive, ref, computed, watch } from 'vue'

    const state = reactive({ count: 0 })
    const doubleCount = computed(() => state.count * 2)

    watch(
      () => state.count,
      (newVal, oldVal) => {
        console.log(`Count changed from ${oldVal} to ${newVal}`)
      }
    )

    function increment() {
      state.count++
    }
</script>
```

IMPORTANT

Both versions are correct and equivalent. In the `script setup` syntax, you don't need to use `return` because everything declared in the `script setup` block is automatically available in the component's template.

Render pipeline

- **Template** - is declarative and describes how the UI should look based on the application state
- **render function** - is the result of compiling the template, compilation is done by the Vue Compiler
- **reactive component state** - the component state that is tracked by Vue's reactivity system
- **virtual DOM** - an in-memory representation of the DOM that allows efficient comparison and updating of the actual DOM
- **actual DOM** - what the user sees in the browser


Template.vue
parse
Abstract Syntax Tree
transform:
optimization and
dependencies analyzer
AST - Optimized
codegen
render()

Example of a render function

Example code:

```
function render(_ctx, _cache) {
  return openBlock(), createElementBlock("ul", null, [
    (openBlock(true),
     createElementBlock(Fragment, null,
       renderList(_ctx.items, (item) => {
         return openBlock(), createElementBlock("li", {
           key: item.id
         }, toDisplayString(item.name))
       }),
       128 /* KEYED_FRAGMENT */
     )
    )
  ])
}
```

Components of a render function

	Description
`openBlock`	creates a new render block that helps Vue track changes in the virtual DOM
`createElementBlock`	creates an HTML element
`renderList`	helper function for rendering lists based on an array
`toDisplayString`	converts a value to a string for display in the DOM
`_ctx`	component context, contains data, methods, and other properties
`_cache`	used to store computation results between renders for performance optimization
`Fragment`	a special type of virtual DOM that allows grouping multiple elements without adding an extra node to the DOM
`128 /*KEYED_FRAGMENT*/`	an optimization flag indicating that the fragment contains keyed elements, which helps in efficiently updating the list

openBlock and performance

- `openBlock` helps Vue track changes in the virtual DOM (instead of comparing the entire DOM tree on each update, Vue can focus only on the changed blocks)
- allows Vue to optimize the update process by minimizing the number of DOM operations
- improves rendering performance, especially for complex user interfaces with many dynamic elements

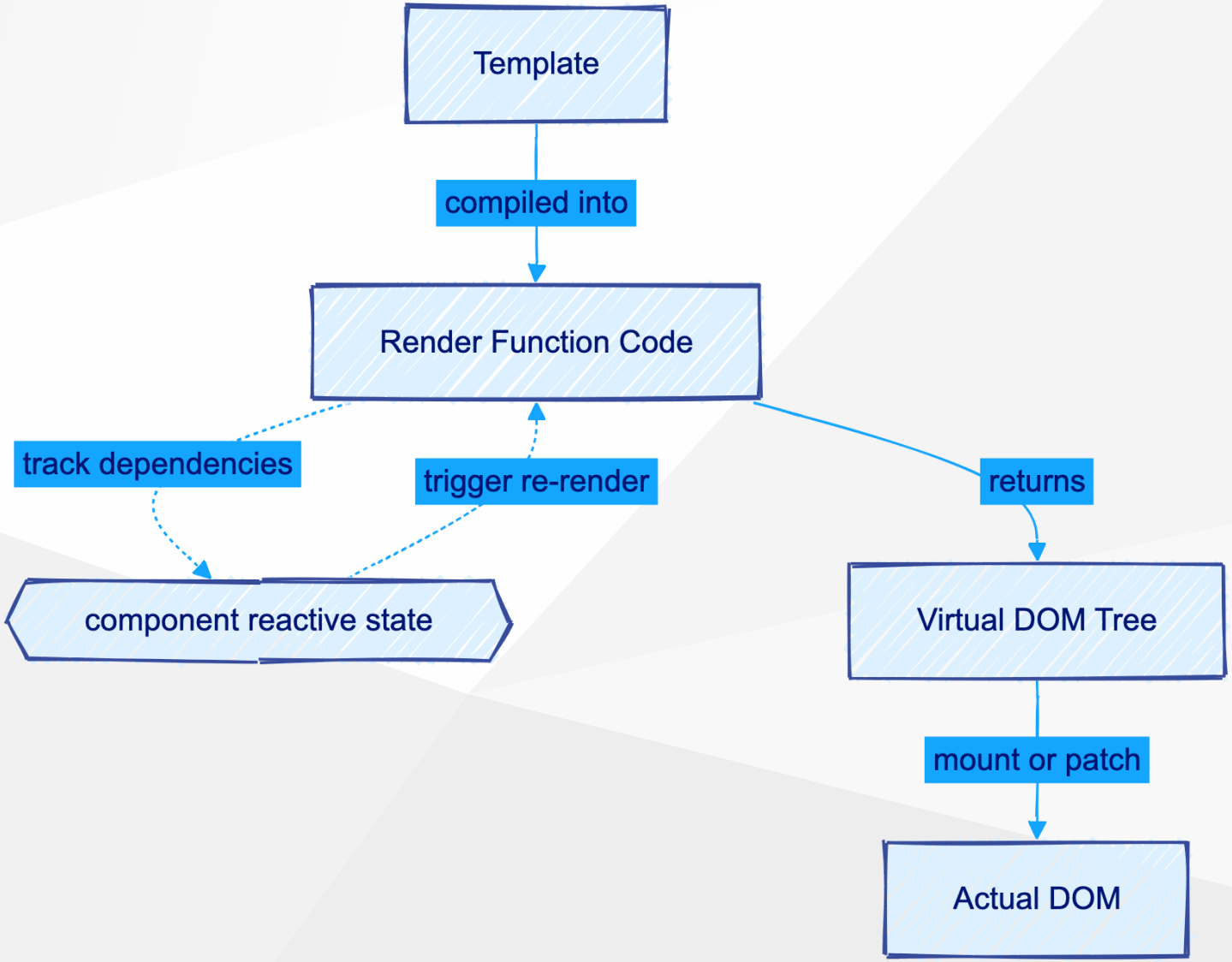
Patch Flags

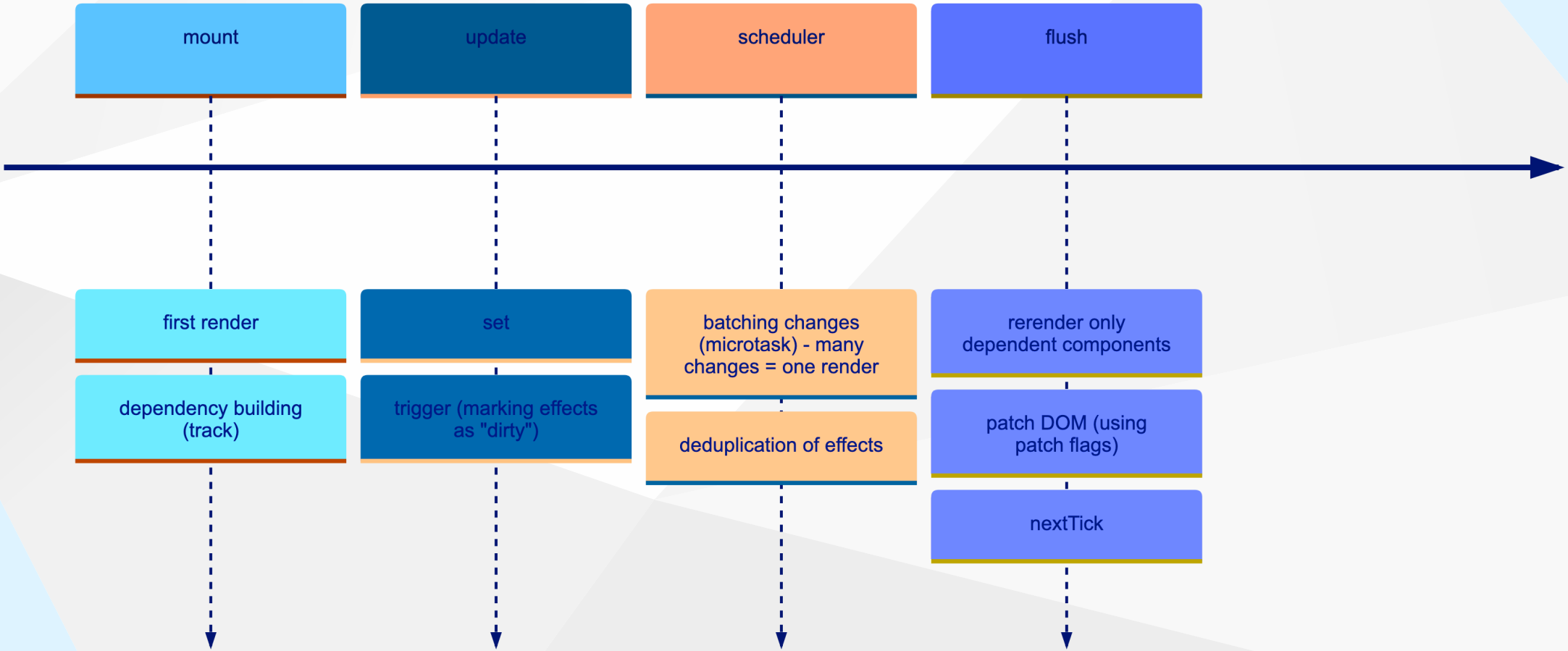
Flag	Description
`1 /* TEXT */`	indicates that the text content of the element may change
`2 /* CLASS */`	indicates that the CSS classes of the element may change
`4 /* STYLE */`	indicates that the inline styles of the element may change
`8 /* PROPS */`	indicates that the element's attributes may change
`16 /* FULL_PROPS */`	indicates that many attributes of the element may change
`32 /* HYDRATE_EVENTS */`	indicates that event handling must be initialized during hydration
`64 /* STABLE_FRAGMENT */`	indicates that the fragment is stable and will not change its structure

Flag	Description
`128 /* KEYED_FRAGMENT */`	indicates that the fragment contains keyed elements, which helps in efficiently updating the list
`256 /* UNKEYED_FRAGMENT */`	indicates that the fragment does not contain keyed elements
`512 /* NEED_PATCH */`	indicates that the element requires patching during the next render
`1024 /* DYNAMIC_SLOTS */`	indicates that the component's slots may change dynamically

DID YOU KNOW

Powers of two are used as bit flags, which allows efficient combination of multiple flags into a single integer using bitwise operations.


Template
compiled into
Render Function Code
track dependencies
trigger re-render
returns
component reactive state
Virtual DOM Tree
mount or patch
Actual DOM


mount
update
scheduler
flush
first render
dependency building (track)
set
trigger (marking effects as "dirty")
batching changes (microtask) - many changes = one render
deduplication of effects
rerender only dependent components
patch DOM (using patch flags)
nextTick

ref vs reactive

Feature	`ref(val)`	`reactive(obj)`
Data type	Any (primitives and objects)	Only objects / arrays
Access (script)	`.value` (e.g., `count.value`)	Direct (e.g., `state.count`)
Access (template)	Unwrapped (e.g., {{ count }})	Direct (e.g., {{ state.count }})
Reassign	Can replace the entire value	Cannot replace the entire object

Loss of reactivity when destructuring

Example code:

```
<script setup>
import { reactive } from 'vue'
const state = reactive({ count: 0 })
const { count } = state // loss of reactivity
function increment() {
  count++ // reactivity does not work
}
</script>
```

Why does this happen?

- destructuring creates a new variable `count` that is no longer linked to the original `state` object
- changing `count` does not affect `state.count`, so Vue cannot track this change and does not trigger a view update

Reactivity in Vue works only when data access goes through a Proxy. Destructuring bypasses the Proxy.

Solution with toRefs

Example code:

```
<script setup>
import { reactive, toRefs } from 'vue'
const state = reactive({ count: 0 })

const { count } = toRefs(state) // preserving reactivity
function increment() {
  count.value++ // reactivity works
}
</script>
```

Why does this work?

- `toRefs` creates references to the properties of the `state` object, preserving their reactivity
- changing `count.value` updates `state.count`, allowing Vue to track changes and update the view

Computed values

A computed value is a special kind of reactive value that is calculated based on other reactive data. It is used to create derived values that automatically update when their dependencies change. It should not create side effects and is cached until its dependencies change.

Feature	`computed`
Data type	Any (primitives and objects)
Access (script)	`.value` (e.g., `doubleCount.value`)
Access (template)	Unwrapped (e.g., {{ doubleCount }})
Purpose	Creating derived values based on reactive data

IMPORTANT

`computed` is cached and does not recompute unless its dependencies change.

Example code:

```
<script setup>
import { ref, computed } from 'vue'
const count = ref(0)
const doubleCount = computed(() => count.value * 2)
function increment() {
  count.value++
}
</script>
<template>
  <div>
    <p>Count: {{ count }}</p>
    <p>Double Count: {{ doubleCount }}</p>
    <button @click="increment">Increment</button>
  </div>
</template>
```

watch vs watchEffect

Feature	`watch`	`watchEffect`
Purpose	Watching specific reactive sources	Automatically tracking all used reactive sources
Dependency definition	Manual (you must specify what to watch)	Automatic (Vue tracks used sources)
Access to old value	Yes (via the second argument of the function)	No
Use case	When you want to react to changes in specific data	When you want to perform a side effect based on multiple reactive sources
Execution time	After the watched source changes (unless `{ immediate: true }`)	Immediately after creation and whenever dependencies change

Example code:

```
<script setup>
import { ref, computed, watch, watchEffect } from 'vue'

const count = ref(0)
const doubled = computed(() => count.value * 2)

watch(count, (newVal, oldVal) => {
  console.log(`Count changed from ${oldVal} to ${newVal}`)
})

watchEffect(() => {
  // example of a side effect dependent on reactive sources
  document.title = `Count: ${count.value}`
})

function increment() {
  count.value++
}
</script>

<template>
  <div>
    <p>Count: {{ count }}</p>
    <p>Doubled: {{ doubled }}</p>
    <button @click="increment">Increment</button>
  </div>
</template>
```

computed vs watch

Feature	`computed`	`watch`
Purpose	Creating derived values based on reactive data	Reacting to changes in reactive data by performing side effects
Returned value	Returns a new reactive value	Does not return a value, performs a side effect
Caching	Yes, the value is cached until dependencies change	No, the effect runs every time the watched source changes
Usage	When you need a value that depends on other reactive data	When you need to run code in response to data changes
Example use case	Calculating a value based on other values	Reacting to data changes, e.g., updating the page title

Cost of reactivity

- `ref` - low memory usage, ideal for simple values
- `reactive` - moderate memory usage, better for complex objects
- `computed` - moderate memory usage, but can be expensive for complex computations
- `watch` - can be expensive if you watch many sources or perform heavy operations
- `watchEffect` - similar to `watch`, but can be more expensive if the side effect is complex or depends on many sources

Derived state through watchEffect (not recommended)

- double reactivity
- unnecessary triggers
- no cache
- harder debugging

Example code:

```
<script setup>
import { reactive, watchEffect } from 'vue'

const state = reactive({
  firstName: 'John',
  lastName: 'Smith',
  fullName: ''
});
watchEffect(() => {
  state.fullName = `${state.firstName} ${state.lastName}`;
});
</script>
```

Derived state through computed (best approach)

- single source of truth
- optimization through caching
- lazy evaluation
- Read-Only value - protects against accidental modifications by default

Example code:

```
<script setup>
import { reactive, computed } from 'vue'

const state = reactive({
  firstName: 'John',
  lastName: 'Smith'
});

const fullName = computed(() => `${state.firstName} ${state.lastName}`);
</script>
```

Deep Reactivity – hidden cost

Placing nested objects in `reactive` causes Vue to track changes at every level of nesting. This can lead to a significant increase in the number of effects and triggers, which impacts application performance.

Example code:

```
<script setup>
import { reactive } from 'vue'
const state = reactive({
  user: {
    name: 'John',
    address: {
      city: 'Berlin',
    }
  }
});
</script>
```

IMPORTANT

Vue does not assume watchers on everything upfront. Vue creates lazy reactivity, meaning it only tracks properties when they are accessed. The cost arises when deeply nested fields are frequently read or modified.

How to reduce the cost of reactivity?

- use `shallowReactive` or `shallowRef` for nested objects that do not require deep reactivity
- use `markRaw` for objects that should not be reactive
- normalize data before placing it in reactive state

shallowReactive / shallowRef

- creates a reactive object or reference, but only at the first level
- nested objects or arrays are not reactive

Example code:

```
<script setup>
import { shallowReactive, shallowRef } from 'vue'
const state = shallowReactive({
  user: {
    name: 'John',
    address: {
      city: 'Berlin',
    }
  }
});
const count = shallowRef(0);
</script>
```

markRaw

- marks an object as "raw", meaning Vue will not track it for changes

Example code:

```
<script setup>
import { reactive, markRaw } from 'vue'
const nonReactiveObject = markRaw({
  id: 1,
  data: 'This will not be reactive'
});
const state = reactive({
  rawData: nonReactiveObject
});
</script>
```

data normalization

- data should be transformed into a simpler structure before placing it in reactive state
- avoiding deeply nested data structures is a best practice

Example code:

```
// Before normalization
<script setup>
import { reactive } from 'vue'
const state = reactive({
  users: [
    { id: 1, name: 'John', city: 'Berlin' },
    { id: 2, name: 'Anna', city: 'Cracow' }
  ]
});
</script>
```

In the above example, any change to a user object will cause Vue to track the entire `users` array, which can be costly with a large number of users.

Example code:

```
// After normalization
<script setup>
import { reactive } from 'vue'
const state = reactive({
  usersById: {
    1: { id: 1, name: 'John', city: 'Berlin' },
    2: { id: 2, name: 'Anna', city: 'Cracow' }
  },
  userIds: [1, 2]
});
</script>
```

After normalization, a change in a single user does not require tracking the entire `users` array, which reduces the cost of reactivity.

Reactivity cost is not "Vue magic", but the number of effects and triggers that we generate ourselves through poor tool selection.

Task 7

20 minutes

1. Create a Vue component `src/components/SubscribeForm.vue` :
   - a form with an email field (input type="email") and a "Subscribe" button
   - email field validation (required, valid email format)
2. Create an Astro component `src/components/NewsletterSection.astro` that:
   - renders a static header + description (HTML-first)
   - embeds a Vue island only for the form
   - passes props to Vue: placeholder (e.g., "your@email.com")
3. Use `NewsletterSection.astro` on the `/blog` page
4. The "Subscribe" button should be active only when the email is valid

Astro Islands & Hydration Strategies

What really is an island?

- island is a fragment of the UI that is statically rendered on the server and then "hydrated" on the client using JavaScript
- islands are isolated from each other, meaning each island manages its own state and logic
- islands can be rendered using different hydration strategies, such as `client:load`, `client:visible`, `client:idle`, and `client:media`

An Astro Island is an isolated fragment of the UI that receives JavaScript only when necessary.

Island as a boundary of responsibility

What Astro does	What Island does
Layout, routing	Interaction
Local state	Static data
Composition	Short-lived runtime

IMPORTANT

An island is a boundary of JS responsibility, not application architecture.

Hydration strategies

Hydration is the process where interactive components on a page are "brought to life" by JavaScript on the client side.

Types of hydration:

- **full hydration** - the entire component is loaded and rendered on the client
- **progressive hydration** - components are loaded and rendered on the client as needed
- **selective hydration** - only some parts of the component are loaded and rendered on the client
- **partial hydration** - only selected components are loaded and rendered on the client

Full Hydration

- classic approach to hydration
- the server generates static HTML, and the browser displays it
- then a JS file is loaded that "brings the whole page to life"

IMPORTANT

The user sees the page but cannot interact with it for a moment (the so-called "uncanny valley") until all the JS is downloaded and executed

DID YOU KNOW

The Uncanny Valley is a term borrowed from robotics. It describes a phenomenon where objects that look almost human evoke feelings of unease and revulsion in observers. In the context of websites, it refers to situations where the user sees a static page but cannot interact with it, leading to frustration.

Progressive Hydration

- after the page is loaded, components are loaded and rendered on the client as needed
- allows for faster page loading because not all components need to be interactive immediately
- faster time to first interaction (TTI)

Selective Hydration

- mechanism that manages the order and priority of component loading
- hydration is based on activating elements when the user interacts with them
- allows interactivity only where necessary, even if other parts are still loading
- found, for example, in React 18 using `<Suspense />`

Partial Hydration

- architectural approach where only selected components are loaded and rendered on the client
- the page consists of pre-generated HTML that largely does not require JavaScript to function
- it is the developer, not the framework, who decides which components should be interactive
- examples of use include: `Astro`, `React Server Components`

client:* as decisions, not defaults

- hydration strategies in Astro (`client:load` , `client:visible` , `client:idle` , `client:media`) are developer decisions about when and how components should be hydrated
- by default, components in Astro are static (no hydration)
- the choice of hydration strategy should be conscious and based on user needs and site performance

client:load

- JS loaded immediately after the page loads
- the most expensive strategy in terms of performance

When to use?

- critical above-the-fold interaction
- element must be active immediately

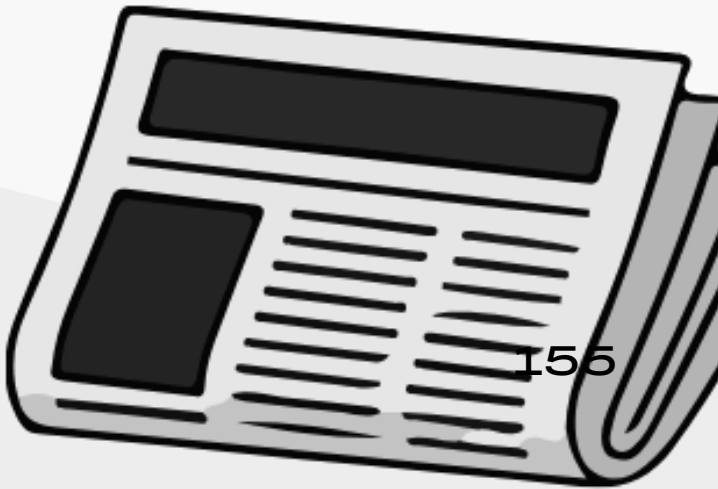
When not to use?

- forms at the bottom of the page
- auxiliary elements

Above the fold

DID YOU KNOW

Above the fold is a term borrowed from newspapers. In the web world, it refers to content visible at the top of the page without scrolling immediately after loading. The code needed to display this content should be loaded as quickly as possible.



client:idle

- JS loaded when the browser is idle

When to use?

- "nice to have" things
- elements low on the page
- widgets, chatbots

TIP

The statement "let's put everything on idle, it will be faster" is incorrect. If everything is loaded on idle, then nothing is loaded on idle.

client:visible

- JS loaded when the element appears in the viewport

When to use?

- modals
- dropdowns
- sections at the bottom of the page

IMPORTANT

This is often the best default, but it is still a decision.

client:media

- JS loaded after a media query is met

When to use?

- desktop-only interactions
- heavy components only on large screens

client:only

- no SSR, component rendered only on the client

Use only when:

- library is not SSR-safe
- library assumes runtime = browser
- SSR works, but hydration is unstable (depends on viewport, randomness)

IMPORTANT

client:only is not a workaround for SSR. It is a conscious decision to forgo SSR when the component has no HTML value or cannot be safely executed on the server.

Common Hydration Mistakes

Hydrate everything

Symptoms:

- a lot of JS
- poor performance
- Astro starts behaving like an SPA

Cause:

- "framework-first" thinking
- transferring habits from React / Vue

Treating Islands like applications

Example code:

```
// too much logic and too broad a scope of responsibility
```

Symptoms:

- heavy components and slow loading
- difficult to debug

Cause:

- lack of division into smaller components
- lack of boundaries of responsibility

IMPORTANT

An Island has no router, no global store, and should not initialize services.

Shared state between islands

Does not work with:

- `pinia`
- `redux`
- `global singleton`

Why does it not work?

- each island is a separate runtime
- no shared context

IMPORTANT

If you need shared state, it's a sign that you're designing an SPA.

Unconscious client:load

Always ask:

QUESTION

Does the user need this immediately?

QUESTION

Is this above-the-fold?

QUESTION

Can the JS wait?

Task 8

20 minutes

1. Create a component `src/components/ArticleTeaser.astro` that statically renders:
   - the article title
   - a short description (excerpt)
   - a list of tags (statically in HTML)
   - a "Read more" link
2. Add only one interaction to this section, as a separate Vue island:
   - component src/components/BookmarkButton.vue
   - local state: bookmarked (boolean)
   - UI: Bookmark / Bookmarked
3. In `ArticleTeaser.astro`, embed the Vue island so that only the BookmarkButton is hydrated and the rest is pure HTML.
4. Choose appropriate hydration strategies for the Vue island.

Component Slicing

Atomic Design

- concept created by Brad Frost in 2013
- it was a response to the growing popularity of RWD and the need to create consistent visual systems for large applications
- the advantages of this approach are: good communication with design, UI consistency, ease of maintenance, and code scalability
- before Atomic Design, interfaces were designed "top-down," starting from full pages and ending with individual elements

Five Levels of Atomic Design

1 Atoms

These are the basic building blocks of the interface. Atoms may contain simple interactions but do not carry domain meaning on their own.

Examples: buttons, form fields, icons, colors, fonts.

2 Molecules

These are groups of atoms combined to perform a specific function.

Examples: label + input + button together form a search form.

3 Organisms

Complex sections of the interface composed of molecules and/or atoms. They form distinct parts of a page. Organisms often serve as the boundary between pure UI and application logic.

Examples: headers, footers, product cards.

4 Templates

A template defines the structure of content without specific data (based on placeholders).

Purpose: focus on proportions and layout of elements rather than the content itself.

5 Pages

These are specific instances of templates with real content and data.

Purpose: to test the template in realistic conditions.

DID YOU KNOW

Atomic Design and Atomic CSS are different concepts - often confused with each other. Atomic CSS is an approach to writing styles, while Atomic Design concerns the structure and organization of UI components.

Directory Structure

In modern frontend applications, it is common to use a directory structure that reflects the levels of Atomic Design.

Example code:

```
src/
  components/
    atoms/
      Button.jsx
    molecules/
      SearchBar.jsx
    organisms/
      Navbar.jsx
  templates/
    MainLayout.jsx
  pages/
    HomePage.jsx
```

IMPORTANT

A directory structure that mirrors Atomic Design 1:1 can be problematic in large applications.

Atomic Design is not a rigid set of rules but a conceptual model. The most important lesson from this method is understanding that the components are more important than the finished pages.

Problems with the Atomic Design approach

- structure based on form does not always reflect functionality
- excessive splitting into small components can lead to complexity and difficulty in management
- it is not always easy to determine the boundaries between levels
- it can lead to over-design and delays in delivering functionality
- problems with components like ButtonIconWrapper (an atom with organism functionality)

When (not) to use Atomic Design?

Atomic Design works well when:

- you are building a design system / UI library
- you have strong collaboration with design
- the UI is relatively stable

Atomic Design does not work well when:

- the domain is complex
- logic is more important than appearance
- components are highly contextual

Philosophy of division: Atomic vs Function-based

- The philosophy of Atomic Design (Atoms, Molecules, Organisms) focuses on the hierarchy of UI components, where each component has a defined role and complexity.
- The function-based approach focuses on the functionality and responsibility of components, regardless of their size or complexity.
- In Astro, it is often better to think of components as functionalities that perform specific tasks, rather than rigidly classifying them by size

IMPORTANT

A well-designed function-based component can serve as an atom, molecule, or organism depending on the context of use.

Function-based slicing

- a component exists because it does one thing well
- a component can be small or large, but its responsibility is clearly defined
- components are designed around the functions they need to fulfill

Examples:

- UserMenu
- SearchBox
- NewsletterSignup

Each of them:

- has a clear purpose
- can contain multiple UI elements
- hides implementation details

Atomic vs function-based

Atomic	Function-based
form	responsibility
visual reusability	logical reusability
design-first	behavior-first
shallow meaning	semantic value

Page vs Island vs UI Component

- **Page** - full application page, responsible for HTML, data, and composition, but not interaction
- **Island** - interactive component that is loaded and rendered independently of the rest of the page (e.g., using `client:load`, `client:visible`)
- **UI Component** - a single user interface component that can be used both on pages and in islands

Page Component (.astro page)

Responsible for:

- routing
- data (fetch)
- SEO
- layout

Not responsible for:

- interaction
- stan UI

TIP

Page fetches data needed to generate HTML. Interactive data, dependent on the user or events, belongs to the Island.

Example code:

```
---
import Header from '../components/Header.astro';
import UserProfile from '../components/UserProfile.astro';
const response = await fetch('https://api.example.com/user');
const user = await response.json();
---
<Header />
<main>
  <UserProfile user={user} />
</main>
```

IMPORTANT

Page composes, it does not implement.

Island Component (client island)

Responsible for:

- interaction
- local state
- events

Characteristics:

- short-lived runtime
- no global context
- clearly defined scope

IMPORTANT

Island is the boundary of JS, not a container for an entire section of the page.

Anatomy of an Island

An island should not be "an entire section" of the page if 90% of that section is static text.

Bad practice:

`NewsletterSection.vue` – contains a title, description, image, and form. The whole thing is hydrated.

Good practice:

- `Newsletter.astro` – static layout, SEO, images.
- `SubscribeForm.vue` – only input fields and submission logic.

IMPORTANT

An island should be as small as possible to minimize the "Reactivity Cost".

UI Component

Responsible for:

- presentation
- no knowledge of data
- no knowledge of environment

Characteristics:

- clean
- testable
- easy to replace

Difficult relationships

TIP

UI Component never knows if it is in a Page or in an Island.

Example code:

```
Page
  ├── Layout
  ├── UI Components
  └── Islands
        └── UI Components
```

Component boundaries

1 Responsibility boundary

A component ends where another domain begins.

2 Hydration boundary

In Astro, the component boundary is often the boundary between server and client. If a component needs to use window, localStorage, or onMounted, it should become an island.

3 Reusability boundary

If you copy the same HTML in three places, it's time for an Astro component.

Where does a component end

- a UI component should be responsible for a single aspect of the user interface
- components should not manage application state or business logic
- components should be easy to understand, test, and maintain

Principles:

- if a component grows too large or complex, consider splitting it into smaller, more specialized components
- avoid creating "all-in-one" components that try to do too many things at once
- focus on readability and simplicity of UI components
- a component does not decide on routing, data, or global application state

A good component: knows what it does, doesn't know where it is used, and doesn't know who uses it.

Hydration Issues & Debugging in Nuxt

What is Hydration Mismatch?

Hydration is the process where Vue (or another framework) tries to "bring to life" the static HTML delivered by the server by overlaying it with a reactivity system and event listeners.

Symptoms of mismatch:

- text "flickers" during loading (changes suddenly)
- missing elements that should be there
- console throws error: `Hydration completed but contains mismatches`

Worst case scenario: Vue abandons the existing DOM and re-renders everything from scratch, which kills performance.

IMPORTANT

Hydration bugs are often not logical errors. They are runtime errors.

Why does mismatch occur?

IMPORTANT

The main rule is that the render on the server and the first render on the client must be identical.

Most common causes:

1 Invalid HTML

 inside <p>. The browser "fixes" the DOM before JS starts, causing the structure in Vue's memory to no longer match what it found in the browser.

2 Access to browser APIs

Using window, localStorage, or document directly in setup() or template.

3 Dates and numbers

Rendering new Date() without consistent formatting (seconds may change between server and client).

4 State mismatch

Different data in initialState on the server than what is fetched on the client.

Problematic code examples

Example code:

```
// Error: using Date without formatting
<template>
  <p>Today is: {{ new Date() }}</p>
</template>
```

Example code:

```
// Error: accessing window in setup()
<template>
  <p>Window width: {{ width }}</p>
</template>
```

Example code:

```
// Error: state mismatch
<template>
  <p>Number of notifications: {{ notifications.length }}</p>
</template>
```

Example code:

```
// Error: invalid HTML
<template>
  <p>This is <div>invalid</div> HTML</p>
</template>
```

Why are these errors difficult?

- they don't always reproduce locally
- dependent on:
  - time
  - environment
  - devices
  - often don't break the build

These are runtime bugs, not compile-time.

How to avoid Hydration Mismatch?

- use Content Collections for deterministic data fetching on the server (`Astro.glob()` works but is deprecated)
- avoid direct access to browser APIs in components rendered on the server
- format dates and numbers deterministically
- validate and test HTML
- use `client:only` or `client:load` mode for components that need to use browser APIs

SSR vs client-only logic

In Nuxt the same component:

- renders on the server
- hydrates on the client
- ...and must generate an identical DOM tree in both environments

Problem:

server and **client** may have different conditions (data, time, environment), leading to mismatch.

Where does the code run?

Phase	Code in `setup()`	`onMounted()`	Template
Server (SSR)	Yes	No	Yes
Client (Hydration)	Yes	Yes	Yes

IMPORTANT

Everything in the template section and directly in setup (except lifecycle hooks) must be safe for both environments.

In Nuxt you constantly have to think about what happens on the server and what happens on the client. This increases complexity.

What is defensive rendering?

Defensive rendering is a technique where code is written to avoid differences between server-side and client-side rendering.

We must assume that:

- something will execute too early
- something will execute twice
- something will execute in a different context

Defensive rendering practices:

- using `v-if="isClient"` to hide client-dependent elements
- checking for the existence of `window` before use
- using placeholders while loading data

Common defensive techniques in Nuxt

process.client / process.server

Example code:

```
const theme = ref('light');

if (import.meta.client) {
    // only in the browser
    theme.value = localStorage.getItem('theme') || 'light';
}
```

Works, but:

- it scatters logic
- it blurs responsibilities

client-only

Example code:

```
<ClientOnly>
  <Chart />
</ClientOnly>
```

- you disable SSR
- you delegate responsibility to the client, knowing that the component should not be rendered on the server

Lazy rendering after mount

Example code:

```
<script setup>
    import { ref, onMounted } from 'vue'
    const mounted = ref(false)
    onMounted(() => mounted.value = true)
</script>

<template>
  <div v-if="mounted">
    <!-- client-dependent elements -->
  </div>
</template>
```

Conditional rendering:

- server - no element
- client - element appears after `mount`

Defensive rendering is the cost you pay for global hydration.

Why does Astro look different?

Nuxt	Astro
hydration of everything	selective hydration
defensive rendering	avoiding hydration
global runtime	isolated runtimes
SSR + hydrate	SSR first, JS optional

SUMMARY

In Nuxt you learn how to fix hydration. In Astro you learn how to avoid it.

Debugging: How to find the cause?

When you see a hydration error, don't search blindly:

1 Enable "View Source"

Check the raw HTML from the server (Ctrl+U) and compare it with what you see in "Elements" (F12).

2 Use Nuxt DevTools

In Nuxt, you can use dedicated tools in DevTools. Check hydration warnings in the console.

3 Binary Search

Cut half of the components from the section where the error occurs until you find the culprit.

4 Strict HTML

Make sure you are not breaking the HTML specification. Remember: <ul> can only have <li> as children, and <a> cannot contain other links.

Server Logic: Nuxt APIs & Astro Actions

Concept of Serverless Functions (API Routes)

Most meta-frameworks natively support creating server functions through a file-based structure.

How does it work?

- files in a specific directory (e.g., `api/`, `functions/`, `server/`) are treated as HTTP endpoints
- each file exports a function that handles HTTP requests (GET, POST, etc.)
- functions are executed on demand, meaning you don't have to manage a server
- functions can be deployed on serverless platforms like Vercel, Netlify, AWS Lambda, etc.

Example of a serverless function in Astro:

Example code:

```
// src/pages/api/hello.js
export async function get({ request }) {
  return { body: JSON.stringify({ message: 'Hello from Astro Serverless Function!' }) };
}
```

Comparison of Popular Approaches

Meta-framework	Function Directory	Target Platforms	Languages/Technologies
Next.js	`pages/api/`	Vercel, AWS Lambda, etc.	JavaScript, TypeScript
Nuxt.js	`server/api/`	Vercel, Netlify, etc.	JavaScript, TypeScript
Astro	`src/pages/api/`	Vercel, Netlify, etc.	JavaScript, TypeScript

TIP

Astro does not have a dedicated directory for server functions. Instead, files in `src/pages/api/` are treated as HTTP endpoints.

In meta-frameworks, "server as a function" means moving from the model "Frontend talks to Backend" to the model "The application has functions that run where it is safest/fastest."

BFF (Backend For Frontend)

The BFF pattern involves creating a dedicated backend for a specific frontend (e.g., web application, mobile app). Serverless functions can serve as a BFF by aggregating data from various sources and delivering it in a format optimized for the given client.

Advantages of BFF:

- reduces frontend complexity
- optimizes performance (fewer network requests)
- provides better control over data and business logic

Example of BFF in Astro:

Example code:

```
// src/api/user-profile.js
import { fetchUserData } from '../lib/user-service.js';
export async function get({ request }) {
  const userId = new URL(request.url).searchParams.get('id');
  const userData = await fetchUserData(userId);
  return { body: JSON.stringify(userData) };
}
```

What are server routes in Nuxt?

- HTTP endpoints running on the server side
- part of the application
- shared runtime with the frontend
- access to request/response

Example code:

```
// server/api/user.get.ts
export default defineEventHandler(() => {
  return { name: 'Mateusz' }
});
```

Characteristics of Nuxt Server Routes

This is the application backend:

- business logic
- validation
- authorization
- integrations with DB / API
- long-running processes

SUMMARY

The application has a frontend and a backend. Both in one repository, sharing the same runtime.

Advantages

- full control over HTTP
- consistency with SPA/SSR application

A good choice for applications requiring:

- auth
- API for multiple clients
- dashboards
- stateful applications

Cost

- global runtime
- larger surface area (more potential bugs and attack vectors)
- more responsibilities:
  - security
  - versioning
  - API contracts

IMPORTANT

Server routes are a full-fledged backend. They require a responsible approach to security and maintenance.

Nuxt Server API: Directory server/

- Nuxt relies on file conventions
- each file in server/api is an automatic endpoint
- Nuxt uses the lightweight H3 framework underneath
- supports various HTTP methods (GET, POST, etc.) by exporting functions with appropriate names, e.g., `login.post.ts`
- auto-imports and integration with Nuxt Modules

Example code:

```
// server/api/login.post.ts
export default defineEventHandler(async (event) => {
  const body = await readBody(event);
  // login logic
});
```

Astro Actions

Astro introduced Actions to further simplify the "function" model. Instead of creating HTTP endpoints and using `fetch()`, you define server-side actions that you call in a component like regular JavaScript functions.

- **function context** - when you call an action in Astro, the framework under the hood performs a POST request to an automatically generated server function
- **security** - the code inside the action runs only on the server, so you can safely use database API keys there that never reach the client's browser
- **type-safety** - thanks to TypeScript, the frontend knows exactly what type of data the "server function" returns, eliminating typing errors in communication

No more fetch('/api/...')

Astro Actions allow you to define actions that are fully typed (TypeScript) and secure.

Actions: RPC-like invocation (the framework generates the transport), ideal for forms and UI.

API Endpoints: classic HTTP (REST/GraphQL/RSS/.json file), when you need a public contract or multiple clients.

Example code:

```ts
// src/actions/index.ts
import { defineAction } from 'astro:actions'
import { z } from 'astro/zod'

export const server = {
  getNewsletter: defineAction({
    input: z.object({ email: z.string().email() }),
    handler: async (input) => {
      // Server logic: saving to the database
      return { success: true };
    },
  }),
};
```

Using Actions in a Vue Component

Example code:

```
<script setup>
import { actions } from 'astro:actions';
const email = ref('');

async function subscribe() {
  const { data, error } = await actions.getNewsletter({ email: email.value });
  if (!error) alert('Subscribed!');
}
</script>
```

Summary

Astro Actions are declarative server-side logic directly tied to the UI.

Astro Actions are not:

- REST API
- application backend

Astro Actions are:

- server functions tied to components
- a safe place for server-side logic
- typed interfaces for communication between frontend and backend

Example code:

```
// src/actions/subscribe.ts
import { defineAction } from 'astro:actions'
import { z } from 'zod'

export const subscribe = defineAction({
  input: z.object({
    email: z.string().email(),
  }),
  handler: async ({ email }) => {
    // saving, email, webhook
    return { success: true }
  },
})
```

Example code:

```
<form method="post" action={actions.subscribe}>
  <input name="email" />
  <button>Subscribe</button>
</form>
```

Limitations of Astro Actions

- work only in SSR or hybrid mode
- cannot be used in `.md` or `.mdx` files
- do not support file uploads (yet)
- cannot define actions dynamically (must be defined at build time)
- no support for WebSockets or long polling
- do not directly support sessions or cookies (must be handled manually)
- no custom HTTP headers in responses (e.g., CORS)

Astro Actions vs Nuxt Server Routes

Feature	Astro Actions	Nuxt API Routes
Typing	Native via Zod	Via `useFetch` and custom contracts
Invocation	Functional (RPC)	Via Fetch/HTTP
Validation	Built-in (Zod)	Manual / any library (e.g., Zod)
Main purpose	Form interactions	Full REST/GraphQL API

Static endpoints in Astro

Astro allows you to create static HTTP endpoints through files in the `src/pages/api` directory.

Example code:

```
// src/pages/api/hello.json.ts
export function GET() {
  return new Response(
    JSON.stringify({ message: 'Hello from Astro endpoint!' }),
    { headers: { 'Content-Type': 'application/json' } }
  )
}
```

When to use which tool?

Question	Astro Actions	Nuxt Server Routes
Is the logic tied to the UI?	✅	❌
Is it a form / submit?	✅	❌
Do you need a JSON API?	❌	✅
Do multiple clients use the API?	❌	✅
Is it domain logic?	❌	✅
Do you want zero JS?	✅	❌

IMPORTANT

Don't try to build a backend in Astro Actions. And don't build a REST API just to handle a form.

State Management — Minimal & Pragmatic

Global state management simplifies communication between components but introduces architectural, performance, and cognitive costs.

Global state management

- a single global object accessible for reading and modification from any part of our application
- all changes and data can be tracked from one place
- a centralized place for storing and managing data
- useful when data is needed by many components in the application, as it ensures they have access to the latest and most up-to-date version of the data
- components can listen to state changes and automatically update their data when a change occurs in the store
- centralization helps avoid chaos related to manual data passing between components, making the application easier to scale

When to use global state management?

- **data is shared between components** - when components from different parts of the application need the same data, such as: shopping cart in ecommerce, profile data, token, theme, language, user preferences
- **complex dependencies between components** - when data is passed through many levels of the application, even though intermediate components do not need this data (in React this phenomenon is called "prop drilling")
- **state is frequently modified** - when the application state is frequently changed in various places in the application, global state allows maintaining consistency, example: notification system
- **asynchronous queries and their state** - when the application sends asynchronous queries, and their result affects the application state in various places, e.g.: product/promotion list - unless we use dedicated server state management tools (e.g., TanStack Query)
- **user interface state management** - for handling global view elements such as modals, toasts, spinners
- **requirement for data consistency** - when the application requires data consistency across different components - one change affects several other places

When NOT to use global state management?

- small applications - usually local state management is sufficient in small applications
- simple applications without complex logic - adding a global state management tool will add unnecessary complexity
- data is isolated - e.g., when data is specific per component
- when components are isolated islands - e.g., in island architecture, where each component manages its own state

IMPORTANT

Global state management is a COST, not a benefit.

Local state

- it is used in a single component
- does not need to survive a page reload
- is not shared with other components

Example code:

```
<script setup>
import { ref } from 'vue'

const isOpen = ref(false)
</script>
```

Default choice. Cheapest. Safest.

Shared state

- it is used in more than one component
- needs to be synchronized
- often comes from the domain (auth, cart, filters)

Shared state can be shared locally (composable) or globally (singleton / store) — they differ in cost and scope.

Example code:

```
// store.js
// example of a simple global store with Vue 3
import { reactive } from 'vue'
export const store = reactive({
  isAuthenticated: false,
})
```

Shared state is a cost.

QUESTION

Who is the sole owner of this data?
If everyone can change it, then no one controls it.

Most common anti-pattern

Let's create a global store because it will definitely be useful

Effect:

- harder debugging
- more hydration
- implicit dependencies
- chaos of responsibilities

IMPORTANT

In Astro islands, global state very quickly destroys island isolation.

Decision hierarchy

1. Local state
2. Props + events
3. Composable
4. URL state: filters/sort/pagination that should be linkable and restorable after refresh
5. Server state
6. Global store (Pinia)
7. Query cache (TanStack)

As you can see, the global store is at the end of the list - this means it should be a last resort.

When "NONE" (and that's OK)

You don't need state management when:

- data is static
- state is directly derived from props
- UI does not change over time
- everything is rendered on the server side

Example code:

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

TIP

No state management is the best state management.

Composables as "light shared state"

Example code:

```ts
// useFilters.ts
import { ref } from 'vue'
const filters = ref({})
export function useFilters() {
  return { filters }
}
```

Advantages:

- no global runtime
- easy refactor
- low coupling
- great for islands
- if a composable holds state in a module (outside the function), it is a singleton (i.e., "quasi-global")

When Pinia?

Pinia makes sense when:

- the state is domain-global
- multiple islands need to share it
- the state has a long lifecycle (auth, cart)
- you need devtools

Examples:

- user session
- cart
- feature flags

Example code:

```
// stores/user.ts
import { defineStore } from 'pinia'

export const useUserStore = defineStore('user', {
  state: () => ({
    user: null,
  }),
})
```

IMPORTANT

Pinia should not be the default choice in Astro islands.

Pinia in Astro islands

Example code:

```
// stores/user.ts
import { defineStore } from 'pinia'

export const useUserStore = defineStore('user', {
  state: () => ({
    user: null as null | { id: number; name: string },
  }),

  actions: {
    login(name: string) {
      this.user = { id: 1, name }
    },
  },
})
```

- domain state (auth)
- truly shared
- long lifecycle and that justifies Pinia

Pinia configuration in islands

Example code:

```
// src/pinia.ts
import { createPinia } from 'pinia'

export const pinia = createPinia()
```

Example code:

```
<script setup lang="ts">
import { setActivePinia } from 'pinia'
import { pinia } from '../pinia'

setActivePinia(pinia)
</script>
```

Building a global Pinia

IMPORTANT

Pinia in Astro is not automatically global. Globality must be consciously built.

To make Pinia global within a page, you need to:

1. create a single Pinia instance
2. share it with all islands
3. explicitly set it as active

Why does Astro do it intentionally?

Astro:

- promotes isolation
- promotes state locality
- does not assume a global runtime

"Global Pinia" in Astro usually means sharing within the same page (document), not "application-wide globality like in SPA".

When TanStack Query?

TanStack Query is not global state management in the classical sense, but a query cache.

Use it when:

- data comes from an API
- you need caching
- refetch, retry, invalidation
- loading/error handling

Example code:

```
const { data, isLoading } = useQuery({
  queryKey: ['users'],
  queryFn: fetchUsers,
})
```

Pinia vs TanStack — key difference

Feature	Pinia	TanStack
State type	UI / domain	Server state
Source	Client	API
Cache	❌	✅
Synchronization	Manual	Automatic
Ideal for	auth, cart	lists, data, fetch

IMPORTANT

Do not keep API data in Pinia unless you have to.

Ask yourself:

QUESTION

Does the state need to be shared?

QUESTION

Does it need to survive a reload?

QUESTION

Does it come from an API?

QUESTION

Is it UI or data?

Performance & Bundle Control

Lazy Loading in Astro

IMPORTANT

Don't hydrate if you don't have to.

Example code:

```
<MyChart client:visible />
<MyForm client:idle />
<MyWidget client:media="(min-width: 1024px)" />
```

- `client:load` - most expensive
- `client:visible` - best default
- `client:idle` - secondary interactions

Bundle size awareness

IMPORTANT

Every import has a cost.

Avoid:

- global UI libraries
- "barrel exports" in islands
- heavy utils in every island

Think:

- does this code NEED to be on the client side?
- can it be moved to Astro (SSR)?

Cost of "Barrel"

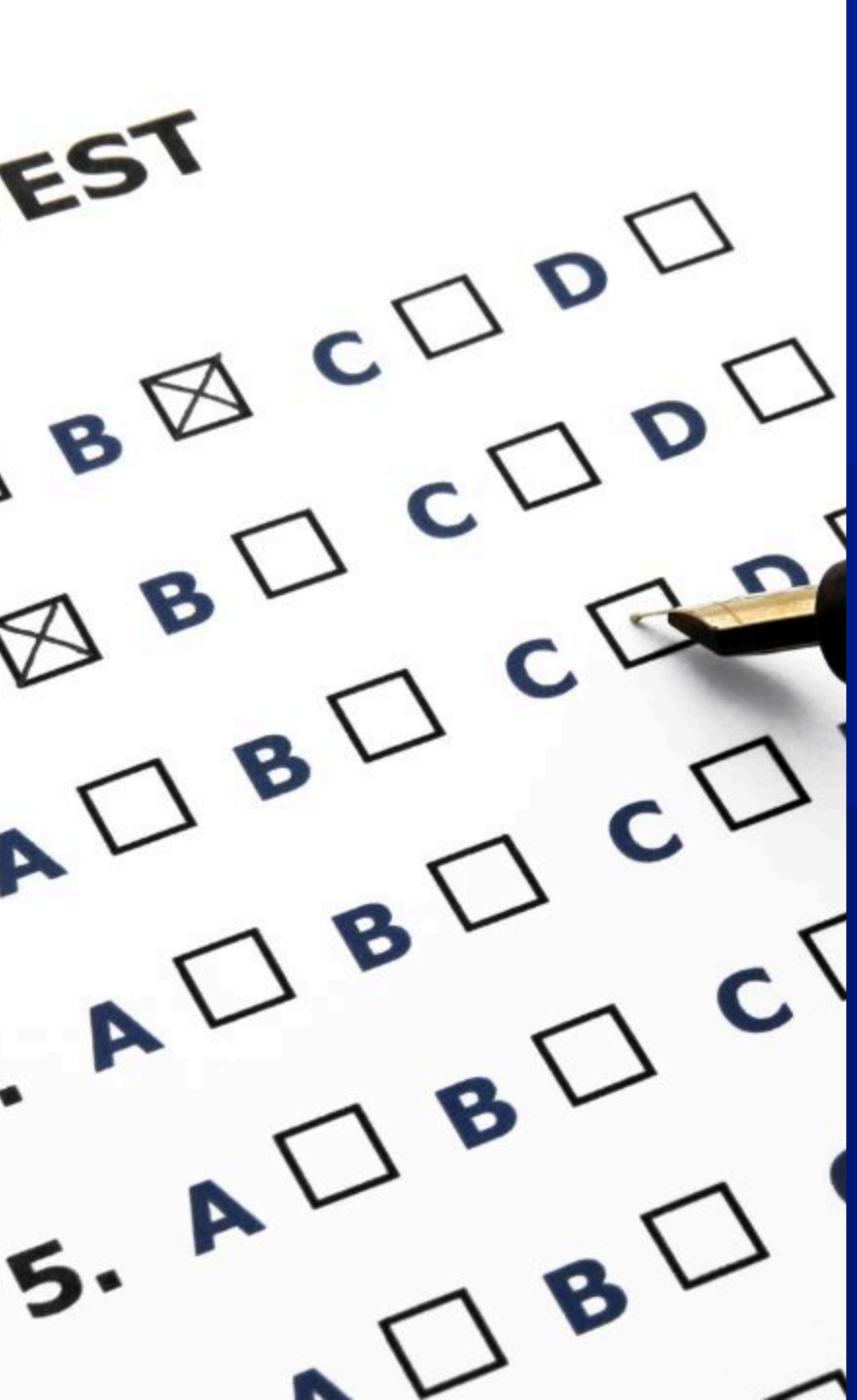
- **Tree Shaking** - some older tools (bundlers) might have trouble "shaking off" unused code. By importing one thing from a "barrel," you might accidentally import 50 other components into the bundle.
- **Circular Dependencies** - this is the most common problem. File A imports from a barrel, which imports file B, which... tries to import file A. This leads to undefined errors at runtime.
- **Developer Performance** - in large projects, editors (like VS Code) might index files more slowly when everything goes through multi-layered barrels.

Checklist "production ready"

- ✓ only necessary islands have JS
- ✓ no global store without justified need
- ✓ `client:*` chosen consciously
- ✓ no deep reactivity for large objects
- ✓ API data is different from UI state
- ✓ bundle checked (`vite build --report`)

The best optimization is code that never reaches the browser

Any additional questions?



KAHOOT

Join at **kahoot.it**
or through the **Kahoot!** app

XXX

Questionnaire

so how did I do?

mateuszjablonski.com/en/poll/57
0126berlin

Thanks for your attention!

Mateusz Jabłoński

mail@mateuszjablonski.com
mateuszjablonski.com


JABŁOŃSKI
Orb
Experts