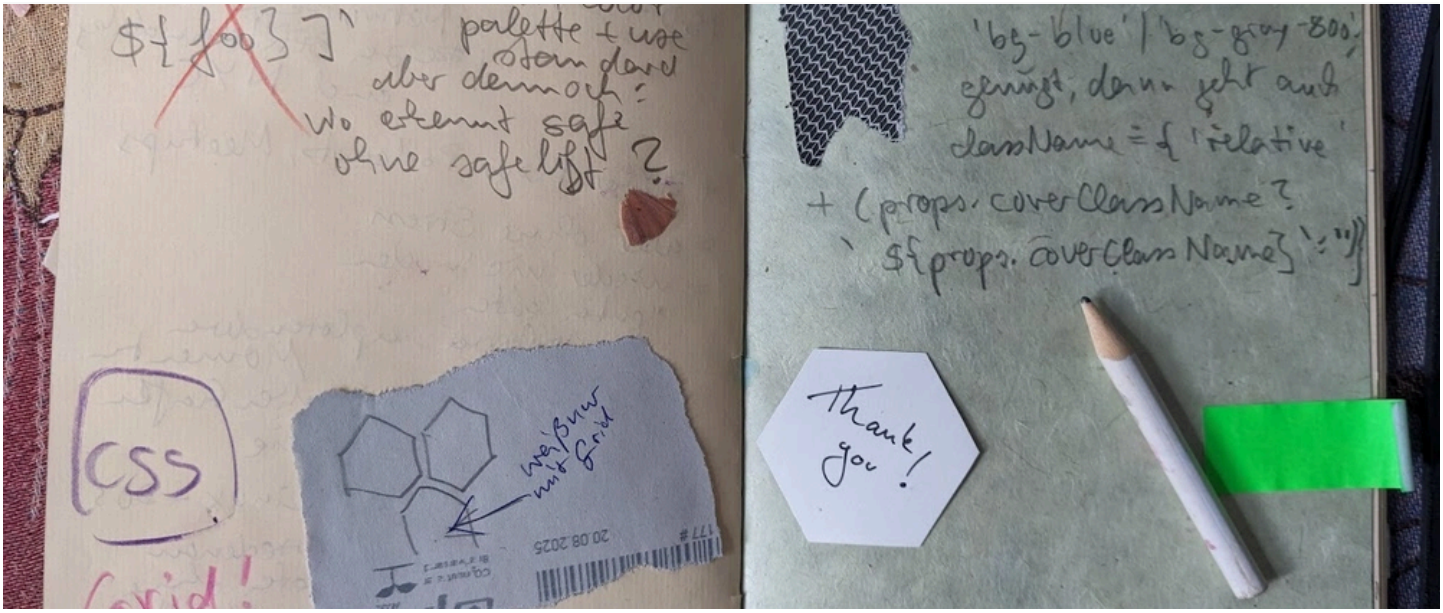


Ingo Steinke, web developer

Edit Stats

An Astro React Revival Project and its Tailwind Takeaways

As developers, we often have to maintain and extend existing software, also known as "legacy code", when we'd rather rewrite everything. When I was looking for a demo project idea to refresh my React skills, I decided to **revive** an abandoned **side project** stub started back in 2021.

I wrote this post to share a variety of takeaways and some seemingly simple issues that I didn't find straightforward answers for. Maybe you'll find some useful tips or answers here.

A Clean and Modern Frontend Tech Stack?

It wouldn't call it minimal to require 13 different notable tech tools and at least **30 npm package dependencies**, but several of these tools are **optional** and don't depend on each other, like Storybook previews or Netlify deployments, or are isolated enough to be **replaced without rewriting** the whole code base.

There are peer dependencies and recommendations, like using Zod to define content collection item attribute types when using Astro with TypeScript. Some tools are not explicitly required by but still reflected in the tech stack, like using VSCode/VSCodium or WebStorm/PhpStorm that might require speicific workarounds like a `// noinspection` line or a certain `.gitignore` entry.

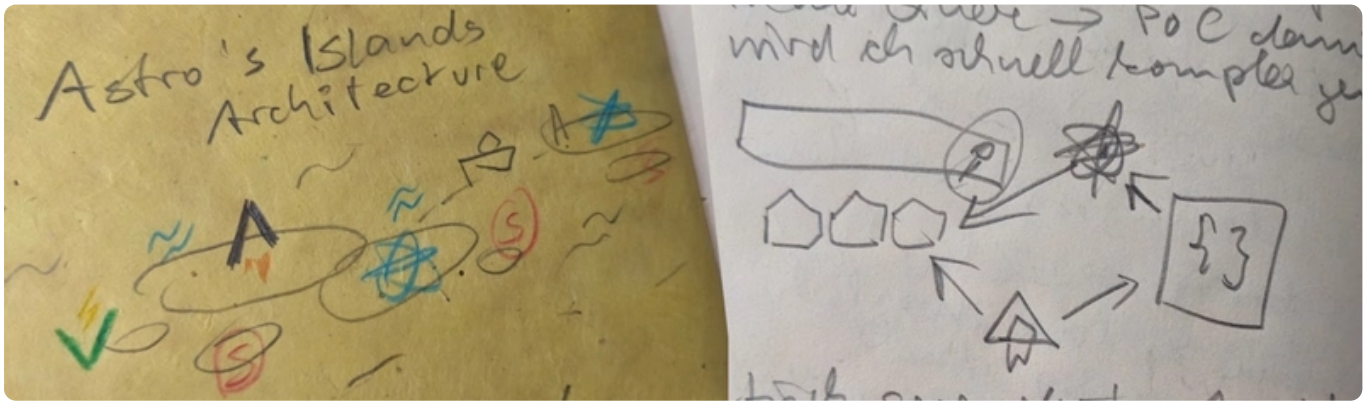
Cleaning up my 2021 full-stack code base to proceed with a clean and modern frontend tech stack, I detached the repository from its obsolete Tailwind-JIT-MERN starter template and removed all backend code. I changed the complex yarn workspaces monorepo back to a flat **npm** package, replaced Preact with the latest **React** version and Snowpack with **Vite**. I kept **Tailwind, TypeScript, ESLint, and Storybook**, and added **Vitest, and Playwright** for testing and quality control.

Regulated React Revival 🚀

Unlike Next.js, **Astro** is optimized for static, content-first websites and ships zero JavaScript by default. Astro is both a static site generator and a meta-framework designed to collaborate with React, Vue, Svelte, or anything similar, thanks to its so-called **Islands Architecture**.

Islands Architecture: Theory and Practice 🏝️🛟

Independent islands in theory can become intertwined peninsulas in practice, coupled by peer dependencies and restricted by incompatibilites, but Astro really helps to simplify and separate components and statically render those parts that need no interactive scripting.



Similarly, testing tools like Storybook, Vitest, Playwright, or Cypress can live next to the actual project files without adding frontend dependencies.

Interactive React Islands within static Astro Islands

How can we restrict interactivity to the smallest possible entities to profit from speed and performance of Astro's static rendering? Let's say, we have a set of cards, and each card has a star-shaped favorite button. In the initial view, the buttons are the only elements that need to be interactive. Consequentially, the favorited status can't be part of the card's state, but rather the toggle's state.

To ensure a component can be rendered statically, all interactive code must either be defined on a child compoents' level, or as global app functionality to prevent code duplication and make the components easier to read and maintain.

Tailwind Takeaways 〰️

CSS-savvy developers must learn an **additional abstraction layer** with arbitrary class names and a slightly unintuitive customization syntax to use Tailwind CSS. In Tailwind, `width: 20rem` becomes `w-80`, so I can't just copy values that I found out using devtools. However, presets like `cover` or `grid-cols-1` are easier to remember than their verbose CSS eqivalents.

Tailwind uses a smart compiler that ships a lightweight package without the style rules that are never used, provided that we write our code in a way that they can easily be detected.

Tailwind documenation archives have version prefixes, e.g. https://v3.tailwindcss.com/docs/ for Tailwind CSS v3.

We can use `@apply` not only for custom styles, but also for grouping frequently combined Tailwind built-ins to make our markup cleaner and easier to refactor.

```
/* global.css */
@tailwind base;
@tailwind components;
@tailwind utilities;

@layer base {
h1, .heading--primary {
  @apply text-6xl md:text-7xl lg:text-8xl lg:text-clamp-10ch;
}
@layer components {
  .my-grid-autofit {
    grid-template-columns: repeat(auto-fit, minmax(min-content, 1fr));
  }
```

```
/* tailwind.config.ts (v3 syntax) */
const config: Config = {
  theme: {
    extend: {
      fontSize: {
        'clamp-10ch': 'clamp(5rem, 10ch, 24rem)',
```

IDEs understand this implicit Tailwind 3 class generation thanks to a so-called "language" server. In Tailwind 4 we would use custom properties (CSS variables) instead. Astro 5.12 stil strongly recommends using Tailwind 3.

Still, learning Tailwind wasn't my biggest challenge.

Tuning Autocompletion Suggestions

JetBrains autocompletion adds two curly braces `{}` when I type `className=` in JSX. With Tailwind, we rarely need that. I changed the "add for JSX" behavior in Settings → Editor → Code Style → HTML → Other from "braces" to "based on type" which inserts double quotes (`"`) instead, ignoring both its Code Style → JavaScript|TypeScript → Punctutation "use single quotes always" and my project settings in `.prettierrc` (`"singleQuote": true`) and `.editorconfig`.

```
[*.{js,jsx,ts,tsx,vue,html,css,scss,json,md}]
quote_type = single
```

Typing `Ctrl` + `Alt` + `L` to autoformat my TSX code doesn't apply single quotes here either. If that's a bug in WebStorm/PhpStorm, I don't care much, as long as it's correct syntax without obsolete braces.

Toggling Tailwind Class Names in JSX ⚛️

Remember to write `className` is JSX/TSX/React, but `class` in `.astro`!

Questions that betray my vanilla JS and CSS mindset:

- What's the equivalent of `classList.toggle` in a JSX `className` property?
- What's the best practice for toggling a single CSS `className` in React while keeping all other existing class names in the class list untouched?
- How to use className template literals in a way that the Tailwind JIT compiler can detect the used class names without needing to enumerate them in a safelist?

The React equivalent of direct DOM manipulation is updating the component's state. The best practice for toggling a single CSS class while keeping others is to use a

template literal with a conditional expression inside the className attribute, and without breaking class names apart:

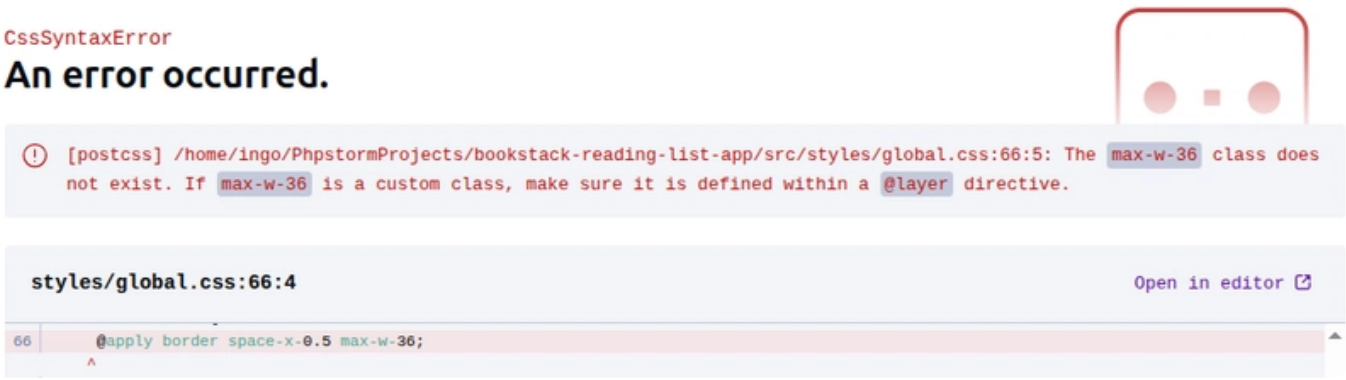
```
return <svg
  className={`w-6 h-6 text-blue-500 ${selected ? 'fill-current' : ''}`}
>
```

Tailwind CSS 3 detects class names in source files by scanning the project files as plain text and looking for tokens that resemble utility class names, not executing any code but performing simple pattern matching to find complete, static class names in our code. So, it should find `fill-current` in the case above or `isFavorite` in the code below:

```
<article className="{{isFavorite ? 'isFavorite' : '' }}
```

Tailwind's class name detection doesn't even seem to work consistently in a static `@apply` rule. If `@apply` works well with `border` and `space-x-0.5`, why would it fail at `max-w-36`?

```
@layer components {
  .omc-hex-tile {
    @apply border space-x-0.5 max-w-36;
```



Safe-Listing Class Names Explicitly ❤️‍🩹

Tailwind's **safelist configuration property**, a workaround to ensure that the compiler exports otherwise undetected class names, accepts class names, patterns, and variants:

```
const config: Config = {
  safelist: [
    'bg-gray-800',
    {
      pattern: /bg-(red|green|blue)-(100|200|300)/,
```

```
      variants: ['hover', 'focus'],
    },
```

This should never be necessary! The Tailwind compiler expects complete strings somewhere, like an explicit enumeration in a TypeScript interface:

```
// Book.tsx
interface CardProps {
  coverClassName?: 'bg-blue-700'|'bg-gray-800';
...
className={'relative' + (props.coverClassName ?
  ` ${props.coverClassName}`
  : '')}  >
```

Further reading: [Tailwind core concepts: detecting classes in source files](#).

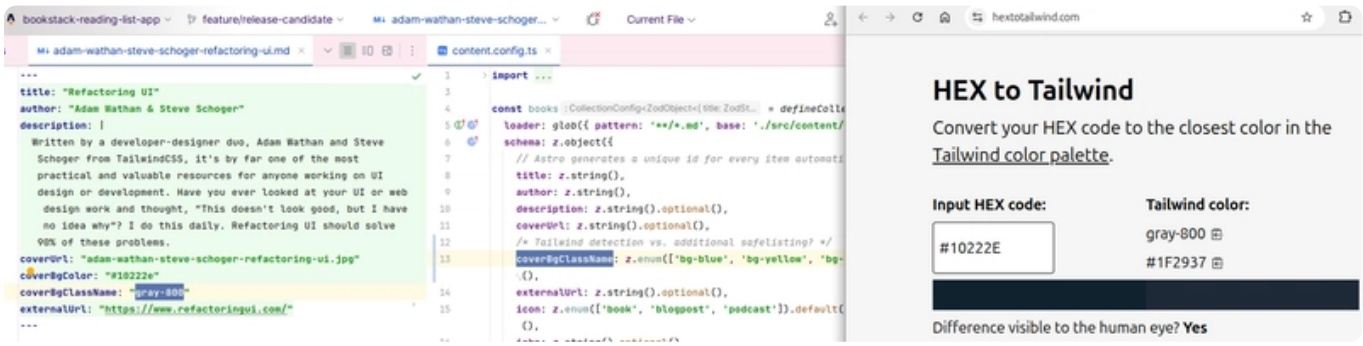
Arbitrary Values: Square-Brackets Syntax 🟦

If safelisting doesn't help, maybe the problematic class name doesn't exist as a Tailwind built-in, even when the doc claim that it does. In the case of `max-w-36`, we might explicitly create the expected style using [Tailwind's arbitrary value syntax](#) in square brackets:

```
    @apply max-w-[9rem]; /* same as max-w-36 */
```

Using the bracket syntax for calculations like `calc(100% - 2px)` we must *omit the spaces around the operator, e.g. `w-[calc(100%-2px)]`;

If we prefer to stick to built-in preset class names, there are handy tools like [HEX to Tailwind](#) for finding the nearest Tailwind class name that matches any given hex color code.



Modelling Data

What's a minimal, but extensible, way to model data and handle state? Each book item needs a unique key to persist a reading list in a local storage, synchronize

favorites with favorization icons, and use favorite status as a filter when searching.

After entering the first ISBN as a file name, I decideded to use human-readable slugs like `stephanie-walter-user-journey-mapping` in this project. Another reason is that not every book has an ISBN, and there is nothing like a "canonical ISBN" to use for normalizing real world data, even when we assume that nobody makes a typo when entering data. While directory (folder) structures and file names are also potentially arbitrary or opinionated, depending on the tutorials and examples, it's worth mentioning is Astro's `public` folder, often parallel to `src`, the contents of which are published unchanged.

Story-Driven Development

Storybook is a DevUX tool similar to Fractal, but while I use Fractal as framework for static site generation, Storybook integrates well with React and has a much larger community on GitHub than Fractal or Histoire, a more Vite-specific Storybook alternative.

Storybook runs its own server instance on a distinct localhost port, so we can use Storybook and our application simultaneously.

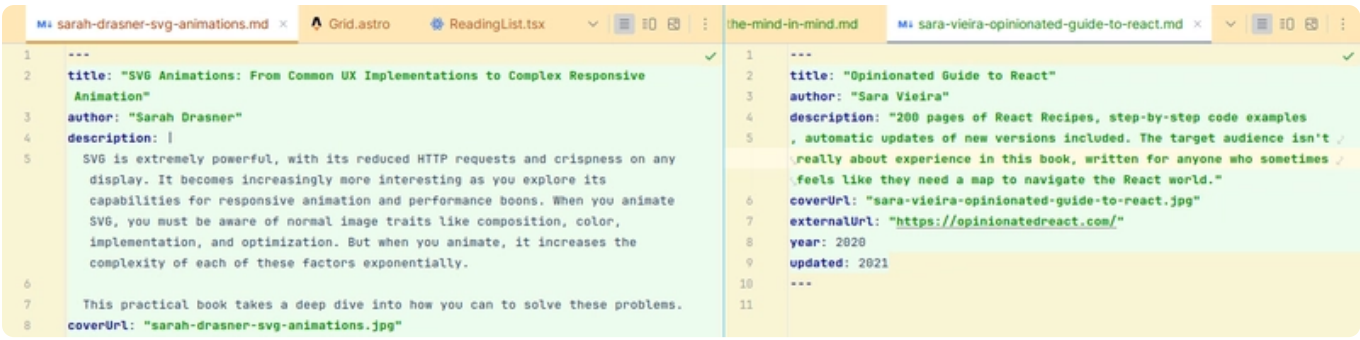
Storybook's play functions are written as JavaScript/TypeScript functions and executed without the need for user intervention. The Interactions panel shows the step-by-step flow.

We can still use another Playwright instance for end-to-end integration tests. That also means that we don't necessarily have to test every single static component in Storybook, which is good, because Storybook can't understand Astro.

Tech Stack Diversity 💐

However, Astro renders TSX to static HTML when we don't specify any `client:` directive, so we could write all of our components in JSX, even those without any interactivity, to include them in our Storybook.

We can use Astro's content collection files and front matter to pass data to our components, whether static (React, Astro) or interactive. But before you start to rewrite every static Astro component to React, pause and think twice. Astro and markdown (mdx) files are easier to write, read, and maintain than React's more verbose TSX syntax, and more forgiving than JSON or XML where a single mistyped character is likely to invalidate the whole document.



Multi-line strings are best written as a YAML literal block scalars using the `|` pipe character in Astro's "fenced content" front matter inside `.md` or `.mdx` markdown files that expect YAML content, while the fenced part of `.astro` files defaults to JavaScript (or TypeScript). Astro provides a type-safe schema validation based on Zod. Astro's content collection querying tutorial doesn't explicitly show how to use content collections in TypeScript, or I got lost in its edge case details before finding an appropriate code example.

Some of those errors break the build while not all cause warnings in the IDE, and the error messages might be misleading. A "bad indentation of a mapping entry" does not necessarily mean a literal indentation error, it can also point to a misssing or extra comma or quotation mark, while a missing indentation results in "can not read a block mapping entry; a multiline key may not be an implicit key". Similarly, "data does not match collection schema" could be caused by a missing triple-dash-"fence" (`---`).

Despite too little min-width and image object cover style, some book covers don't fill their tile completely. Background colors also act as visual placeholders before images are loaded, but how to ensure that Tailwind will detect which one we are using? The following code is verbose and repetitive, but it works.

```
import { glob } from 'astro/loaders';

const books = defineCollection({
  loader: glob({ pattern: '**/*.md', base: './src/content/books' }),
  schema: z.object({
    author: z.string(),
    description: "z.string(),"
    icon: z.enum(['book', 'blogpost', ...]);
    coverBgClassName: z.enum(['bg-blue', 'bg-yellow', 'bg-green']).optional()
    pubYear: z.number().int().optional(),
// ...
export const collections = { book: books };
```

... defining content ...

```
---
author: "Alla Kholmatova"
description: |
  A practical guide to creating design systems
  that empower teams to create digital products at scale.
pubYear: 2017
---
```

... and then using it with or without explicit sorting:

```
---
import Book from './Book.tsx';
import { getCollection, type CollectionEntry } from 'astro:content';
const books: CollectionEntry<'book'>[] = await getCollection('book');
---
<ul class="flex flex-wrap md:flex-col">
  {
    books.map(({ id, data }) => (
      <li>
        <Book
          id={id} /* auto-generated by Astro */
          author={data.author}
```

Content Editing as a Single Point of Failure?

Astro's MDX syntax might look robust and forgiving, but beware!

Astro treats invalid content files as build-breaking errors rather than skipping them as optional content. A single missing or mistyped character can break the build of the whole app, turning content editing into a single point of failure. For dynamic imports or `Astro.glob()`, a wrapper around Vite's `import.meta.glob`, it's possible to handle errors in code with `try-catch` to avoid breaking the build during runtime, but this does not prevent build failures caused by content validation errors upfront.

If Astro doesn't fix this, we don't need to wonder why end-user-focused content management solutions like WordPress remain so popular.

Don't Overengineer Content-Focused Websites

We don't need to write every line of code in TypeScript, and we shouldn't: some technological islands *require* an old syntax, like `postcss.config.cjs` or `.astro/content-modules.mjs`, we shouldn't overengineer content-focused websites, and we can define eslint overrides where it makes sense. And if a short-sighted

warning like "Unused constant collections. Remove unused constant collections" still won't go away, and `eslint` shows no errors, it must be some unhelpful IDE inspection that we can disable globally or per next-line directive. Here's how to do it for WebStorm:

```
// noinspection JSUnusedGlobalSymbols
export const collections = { book: books };
```

Reading

I like reading! My [original reading list project post], back in 2021, featured a picture of a small book shop in a small town in England.

Recommended Reading for Devs and Designers 📚

Beyond acting as a proof of concept demo and as a learning project for Astro and React development, my reading list contains several interesting books about UX, UI, web design, web development, AI, and ethical aspects of digitisation.

I sometimes code in a public library, but I hadn't borrowed books for quite some time. It struck me how many printed books keep coming out for contemporary topics. I also checked my unread books at home and in my eBook library and rediscovered Hannah Fry's [Hello World: How to be human in the age of the machine]. It was published in 2018, but it aged well. Joy Buolamwini's [Unmasking AI] (2024) is the perfect sequel, and Zsike Peter's [Thinkbait] (2025) has just been released.

From the Past to the Future

My initial reading recommendations list goes back to 1994 (Design Patterns, by Erich Gamma and the "gang of four") and forward to 2026 (Joana Cerejo's Anticipatory

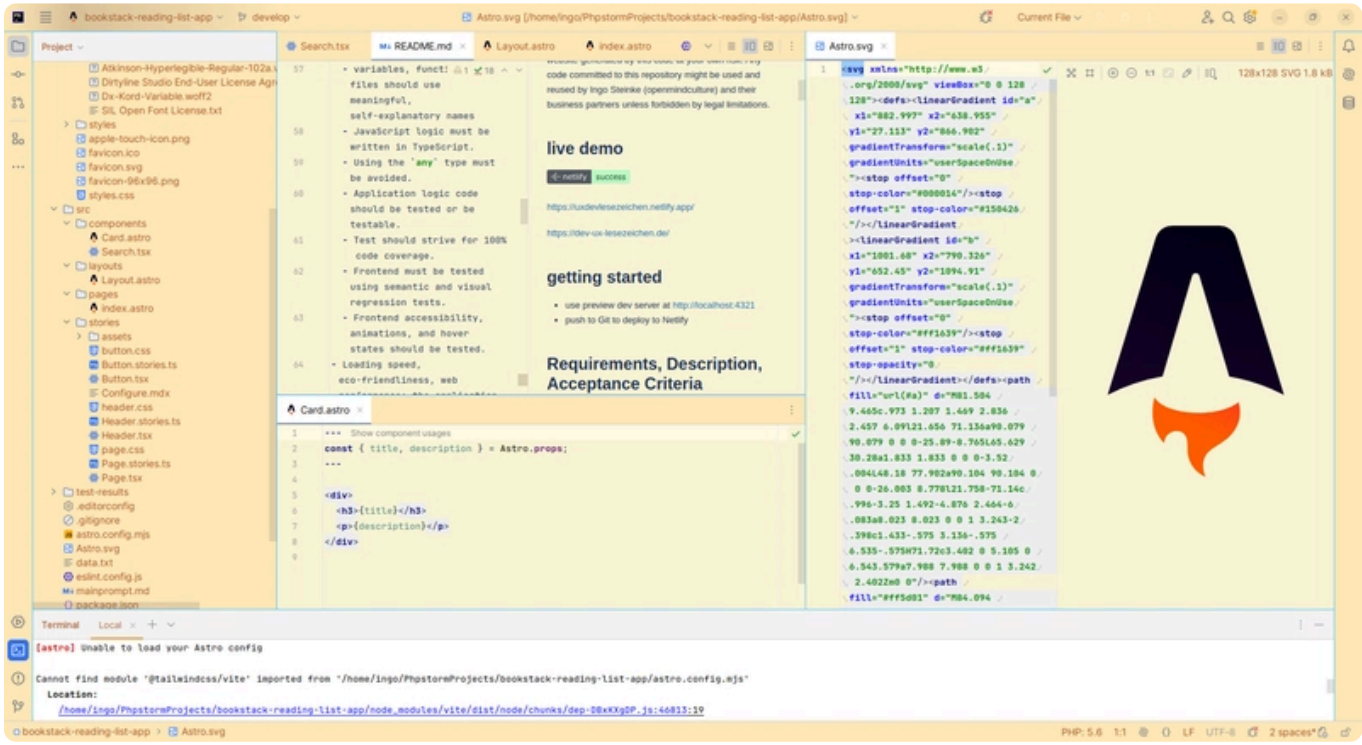
Design Playbook, an upcoming book that can be pre-ordered later this year), but the majority of my virtual book stack are all the books that I bought as downloads to read while commuting in the 2010s and early 2020s, many of which I still haven't finished, many of which by authors that I have met in person or at least online.

These books are a perfect match to recap, update, and extend my knowledge, as they remind me of meetups, conferences, and past pair programming sessions, bridging theory and practice and giving each chapter a personal voice and a point in time, making it easier to evaluate each advice from today's perspective.

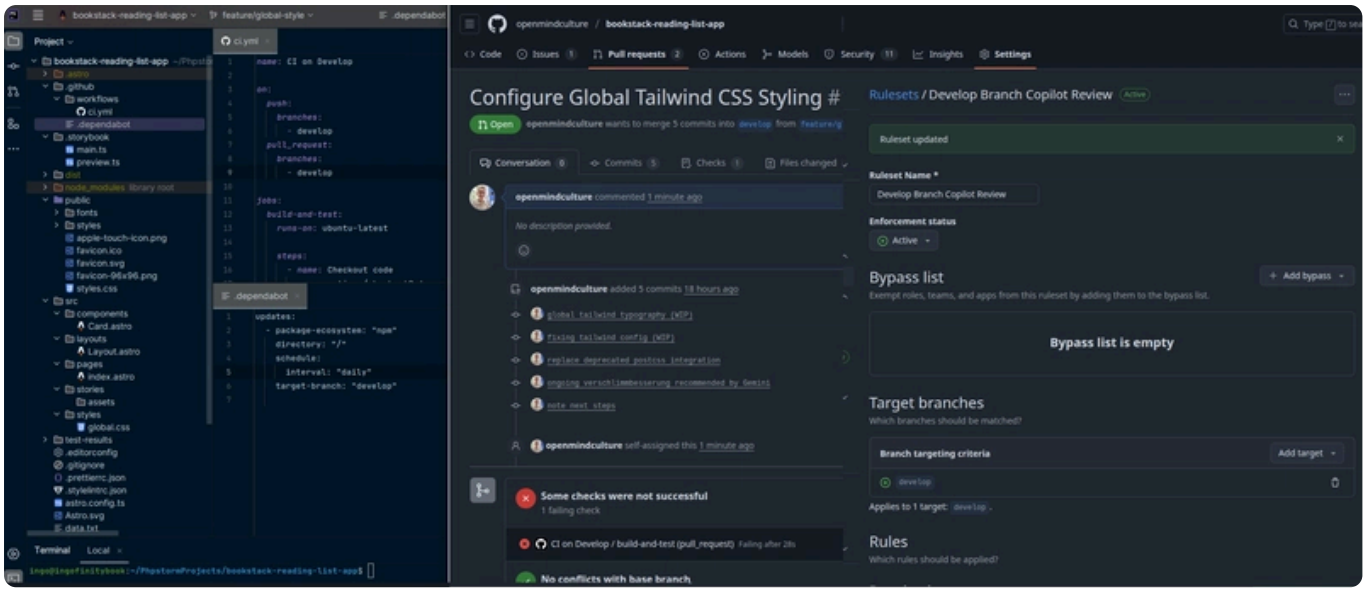
From Reading to Coding ✍️

I used this content to build a practical example based on Astro's official documentation tutorial. My web app's landing page will start with a small selection of featured books. Further content can be loaded later, requested by an interactive, React-powered, search form. A static, Astro-generated JSON file containing the complete, unfiltered, content, is good enough for a first proof of concept.

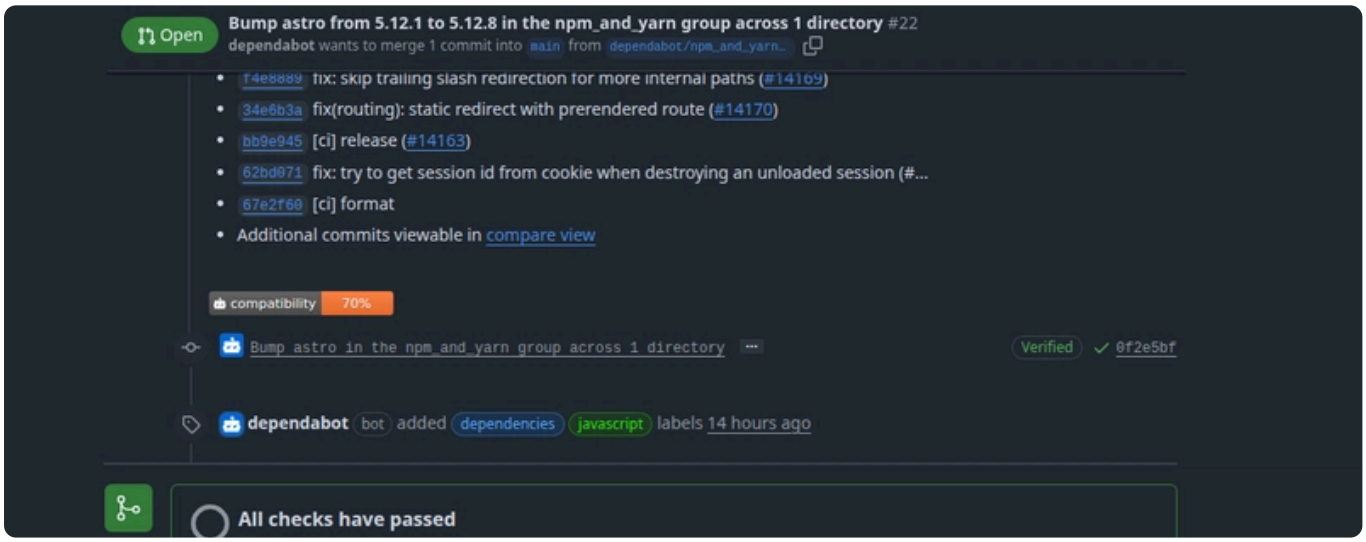
Here is a screenshot of an early development stage.



Repository Agents, Rules and Tools to Enforce Quality 🚨



GitHub provides repository-based agents, including Dependabot and **Copilot**. GitHub workflows files allow to integrate third-party tools as well. Using **branch protection rules**, we can mandate that certain checks must have past before we can merge a feature branch.



Dependabot's compatibility scores are calculated as a a percentage based on how many other projects using the same dependency update had their Continuous Integration (CI) tests pass, without evaluating the code in the current repository. Still, dependabot merge requests include linked lists to changelog details, and we can see if our automated checks have passed, so it's an easier process than a manual check or a generic `npm outdated` and `npm upgrade` query.

Conclusion: Showdev and Roadmap(s) 🔀

This is an unfinished work in progress side project for the purpose of learning.

I have reached my goal: I have learned how to use Astro, Tailwind, Vite, and Vitest, and I have updated my TypeScript and React skills, while other web developers are

over React and turning to alternatives like Vue, Svelte, or SolidJS. Thanks to Astro, I could use Svelte to implement another independent feature like adding and editing book data online or transform a content editing recommendation into a GitHub pull request or issue.

I could focus on adding and describing content and improving the frontend's design and behavior. I could add full-stack backend features, use deno or its hottest latest successor, take a glimpse at Rust and R, or learn some more Python (probably the most obvious choice based on my experience and the latest job descriptions).

I could also deepen my experience with devOps and cloud hosting.

I already configured continuous deployment of my main branch to a production demo, currently hosted by Netlify, called Dev-UX-Lesezeichen ("Dev-UX Bookmarks"):

Website: https://dev-ux-lesezeichen.de/

Contribution

Have a look, leave a comment, browse the code or open a GitHub issue at https://github.com/openmindculture/bookstack-reading-list-app . I can't guarantee to publish every suggestion, but I'm looking forward to your input, especially if you know good, new books about dev, UX, accessibility, and inclusion, preferably open-minded and intersectional, that are still missing on my list!