Hello, OKTech.jp

An Introduction,
and some things you might find useful

Hello everyone.

My name is Chris, and I like to make websites.

When I came to Osaka earlier this year, I didn't know anyone.

So I went to an OWDDM event and I met some cool people.

I really appreciate having an anchor to the real world – a human connection with other tech people.

It's one of the reasons I decided to stay in Osaka, and why I volunteered to help out.

Thank You, Martin, being the pillar that allows this Meetup and it's Community to exist.

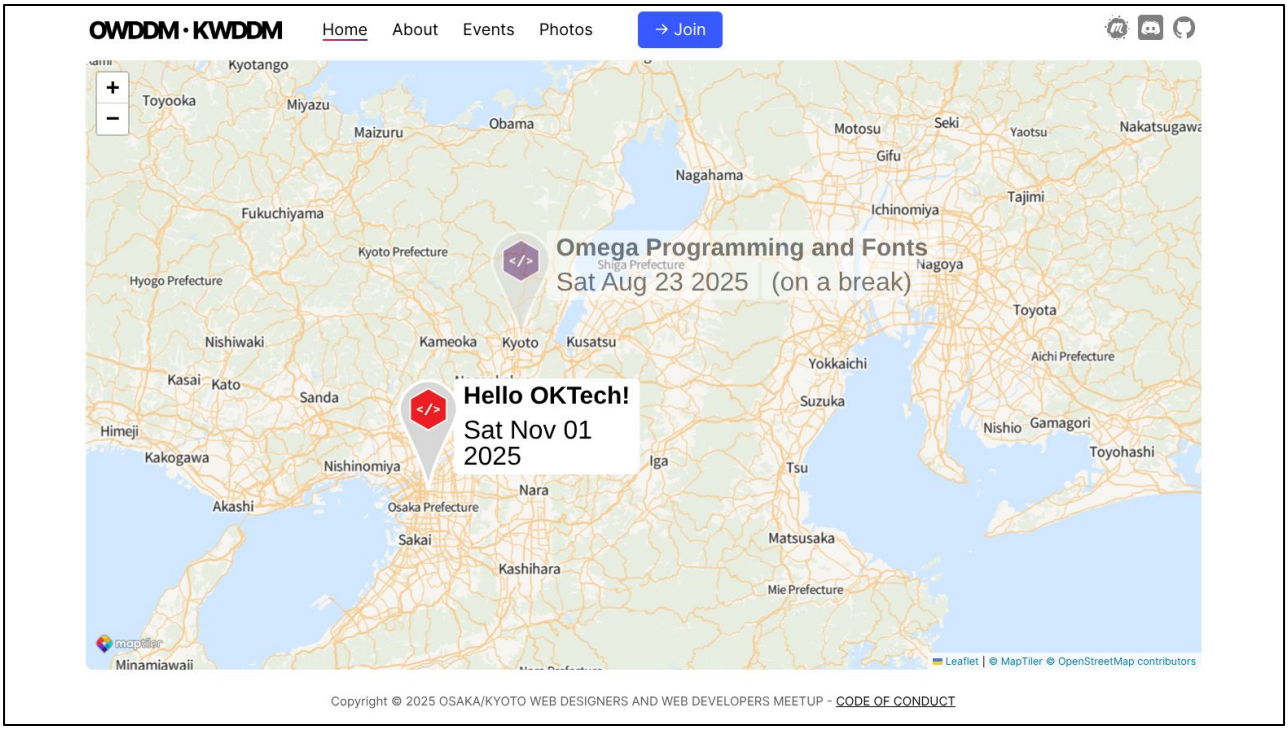
please clap

In this presentation, I thought it might be interesting to show you the new OKTech website, and some things I learned while making it.

Let's get cracking.

Goodbye,
owddm.com

Before we say Hello,



let’s say goodbye. to owddm.com

we will miss you

may your eternal soul live on in the internet archive.

Let Designers Design & Developers Develop

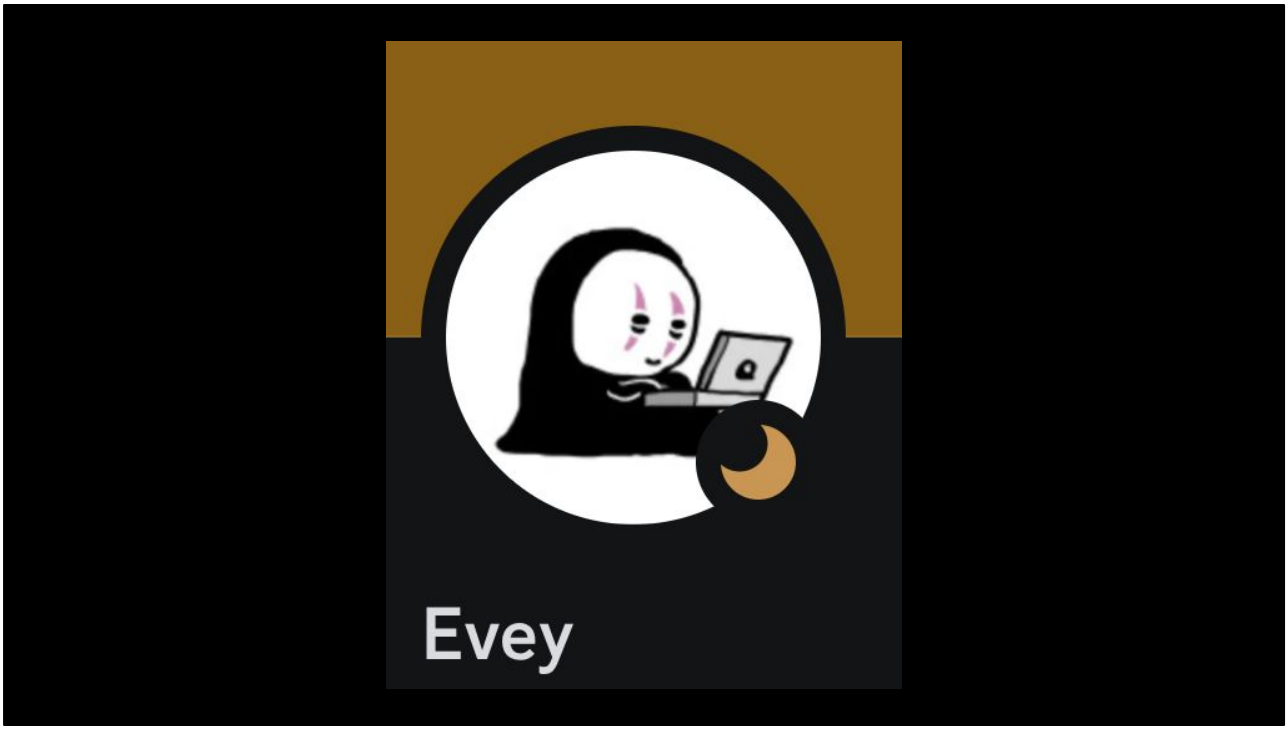
As you may know, OWDDM stood fo Osaka Web **Designers and Developers** Meetup.

I wanted to start by recognising the importance of non-developers – both in this community, and in tech projects in general.

We developers often can't help but think about the nitty gritty technical implementation.

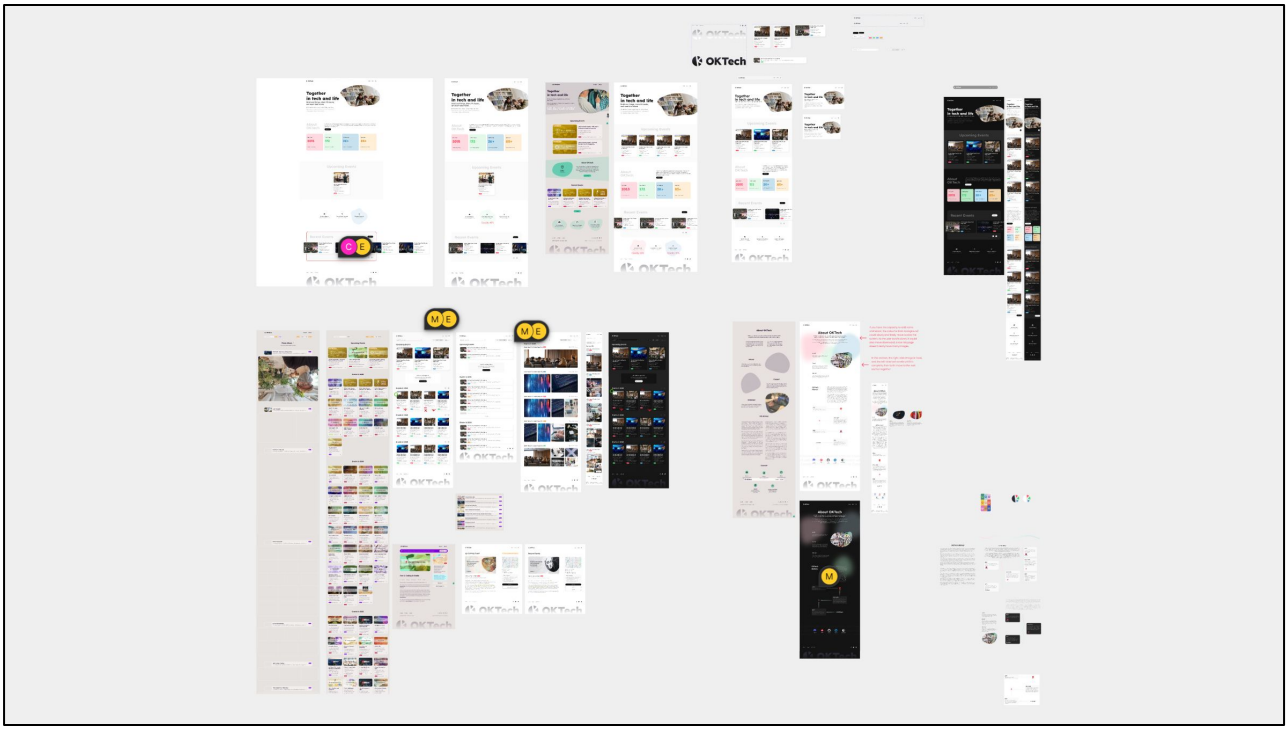
So when it comes to design, we're always thinking about what's **technically** easy, or **technically** fancy, rather than what's purely the best design.

That's why developers need non-developers, to guide us with their fresh perspectives.



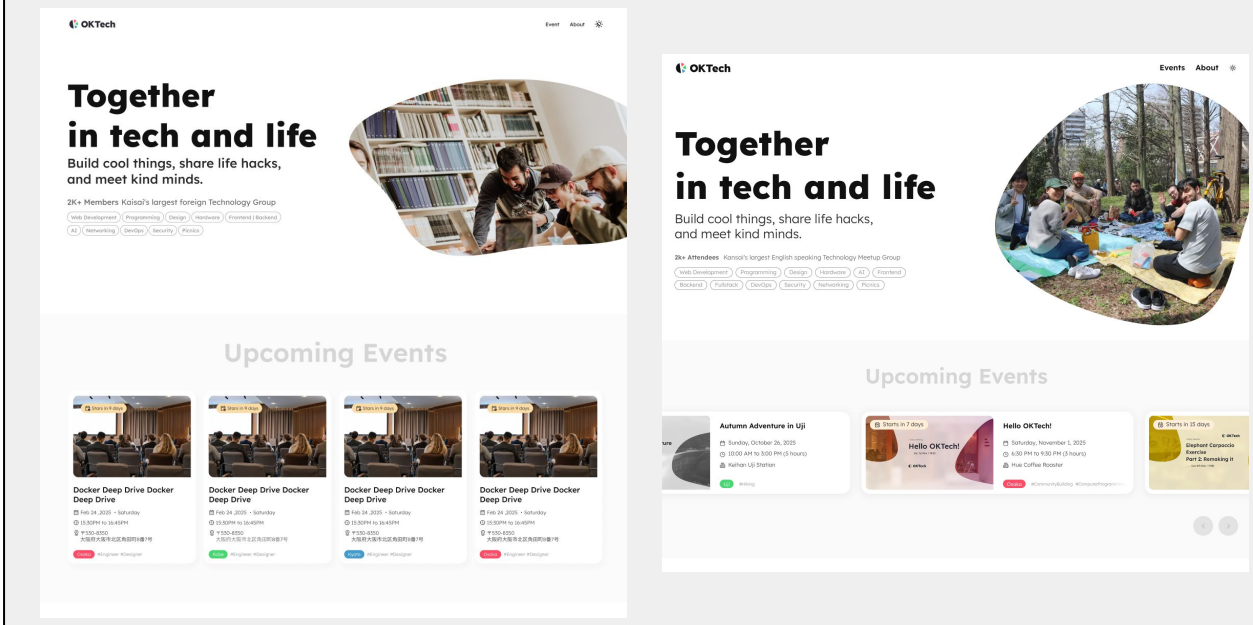
Lucky for [OKTech.jp](), a designer stepped forward and volunteered to help.

This is Evey, and, up until today, I only knew her by this avatar on discord.



Evey spent many hours mocking up and finessing designs for the new website in Figma.

Let’s zoom in a bit.



Here's the new OKTech landing page.

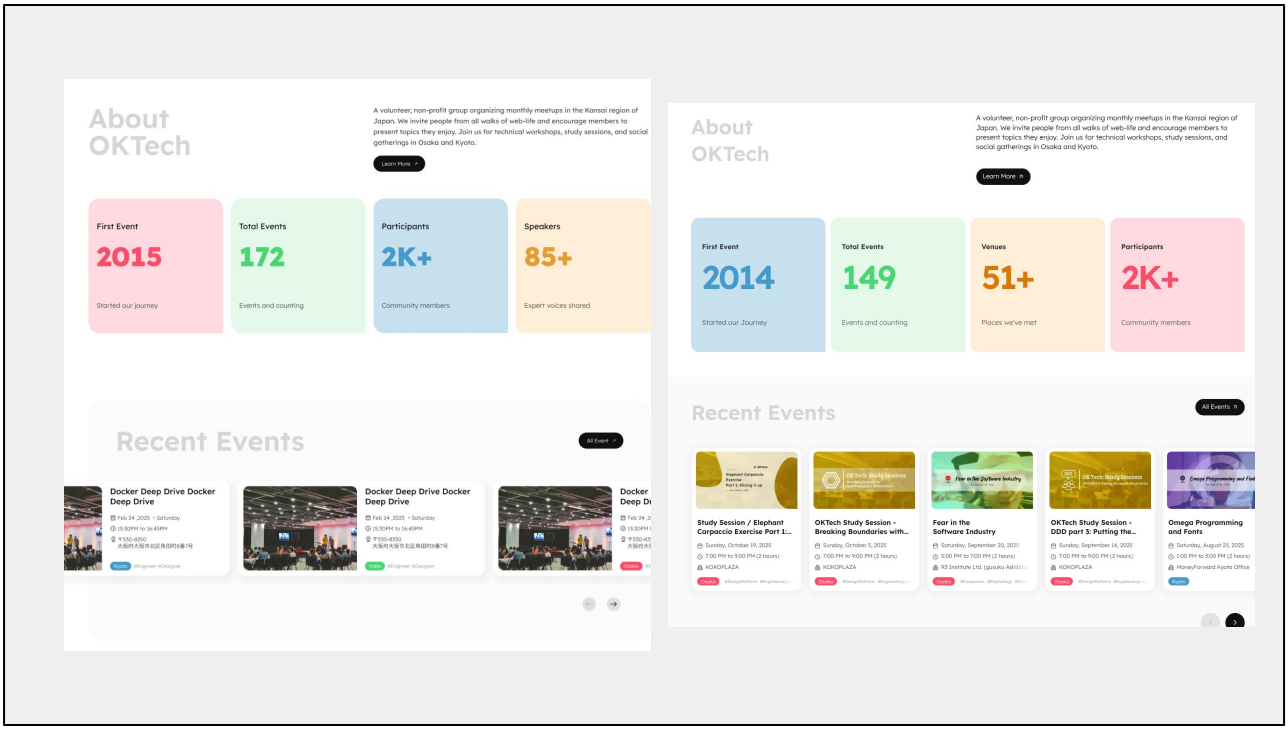
On the left, Evey's design in Figma.

On the right, my attempt to convert it to HTML and CSS.

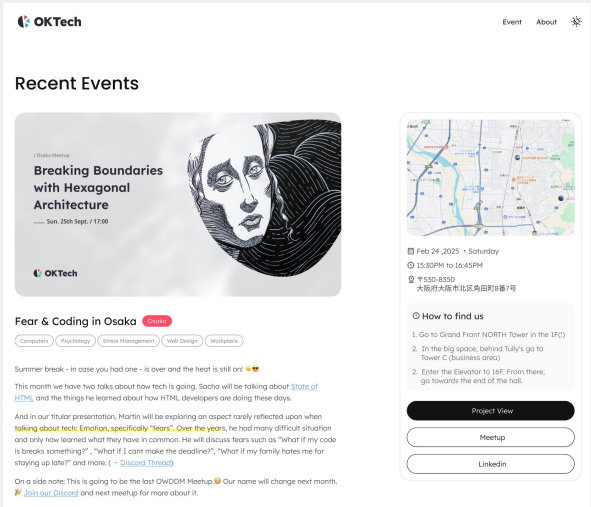
The new landing page is quite different from OWDDM, isn't it?

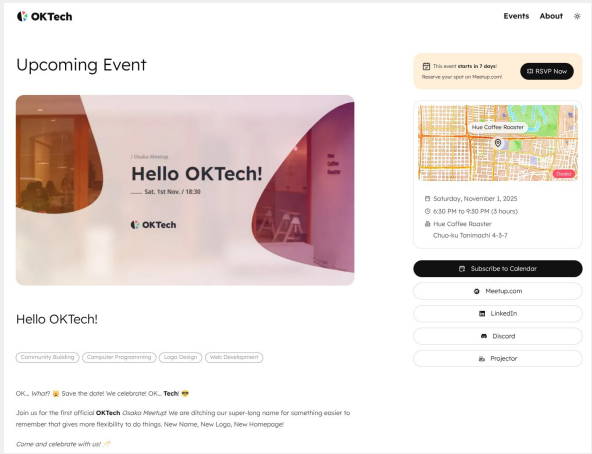
Instead of a clinical map of events, visitors are first presented with what OKTech is really about – human connection.

Welcoming, friendly, real people. Together, in Tech and Life.

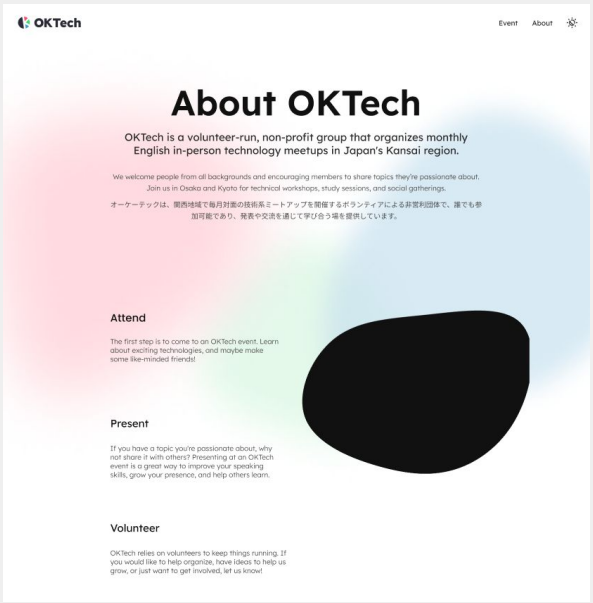


Scrolling down the landing page, and OKTech's proof-by-numbers is showcased.

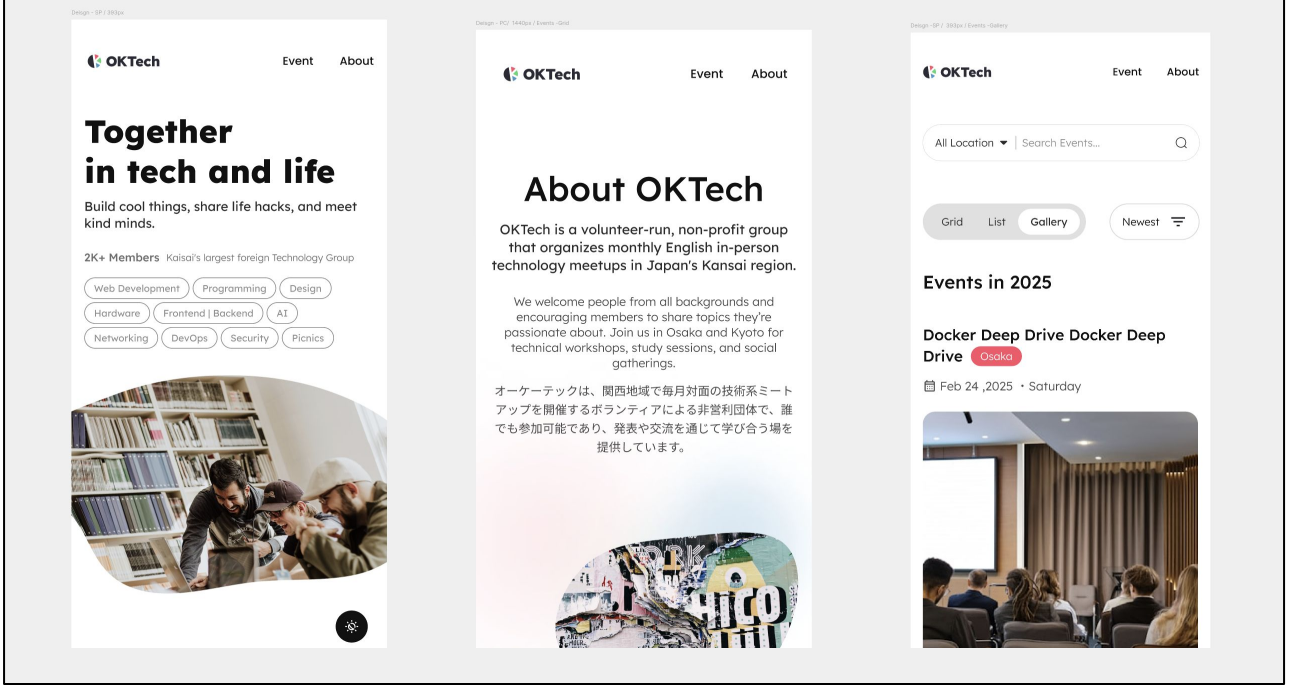




Each event has its own beautiful page, with clearly laid out details.

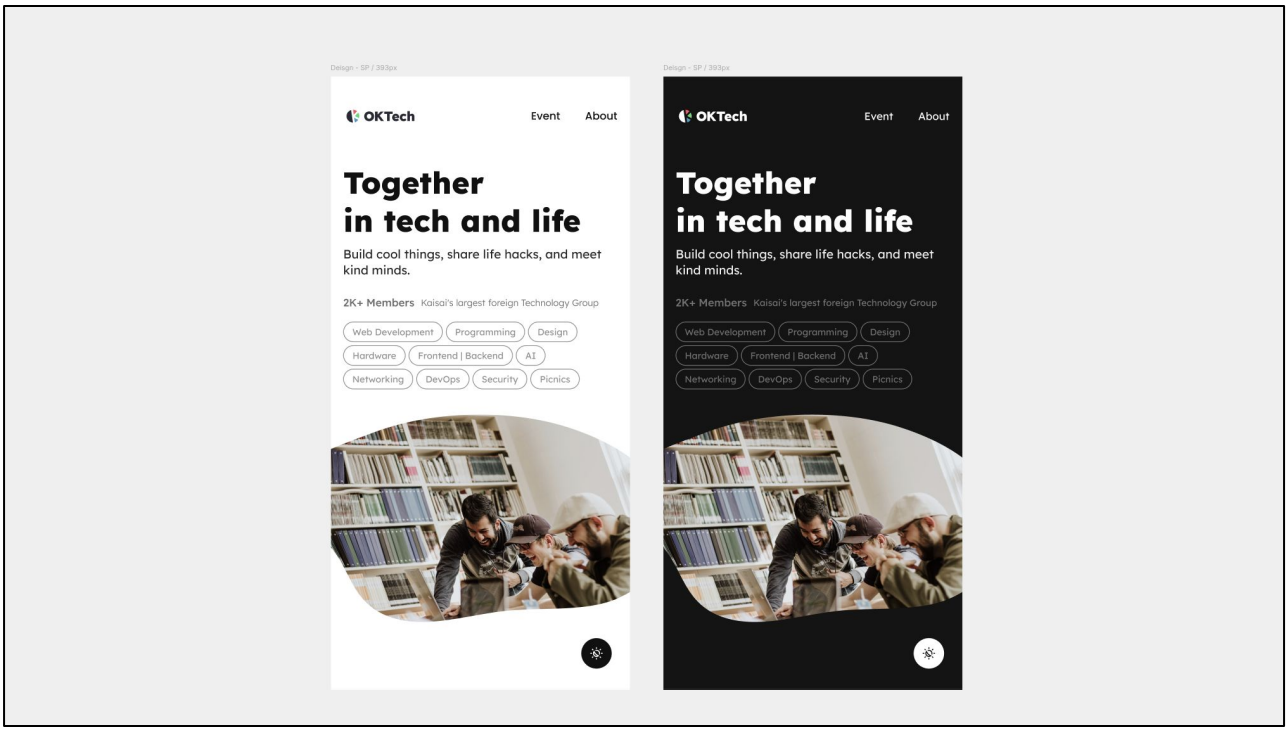


We have the most interactive and engaging About page I've had the pleasure of building.

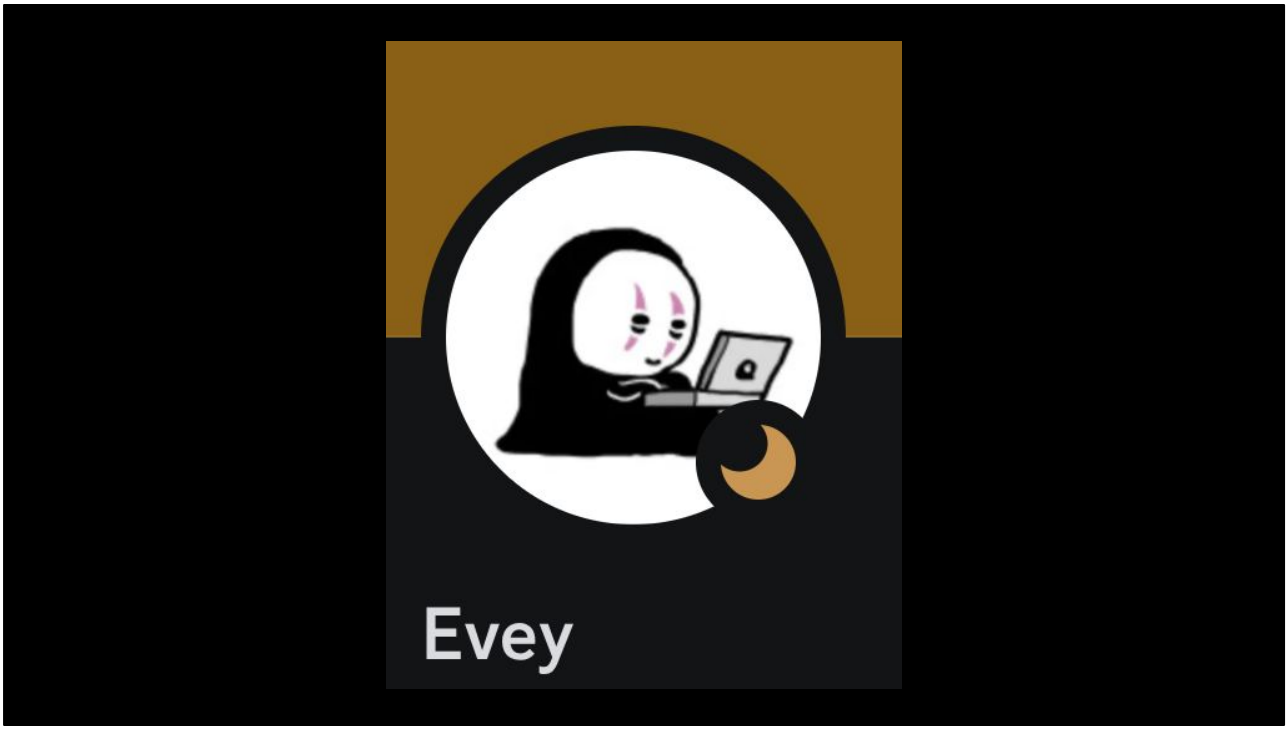


Not only do we have multiple page layouts,

But Evey also took the time to implement responsive design variants for mobile.



And perhaps most importantly, she provided themes for both light and dark mode!



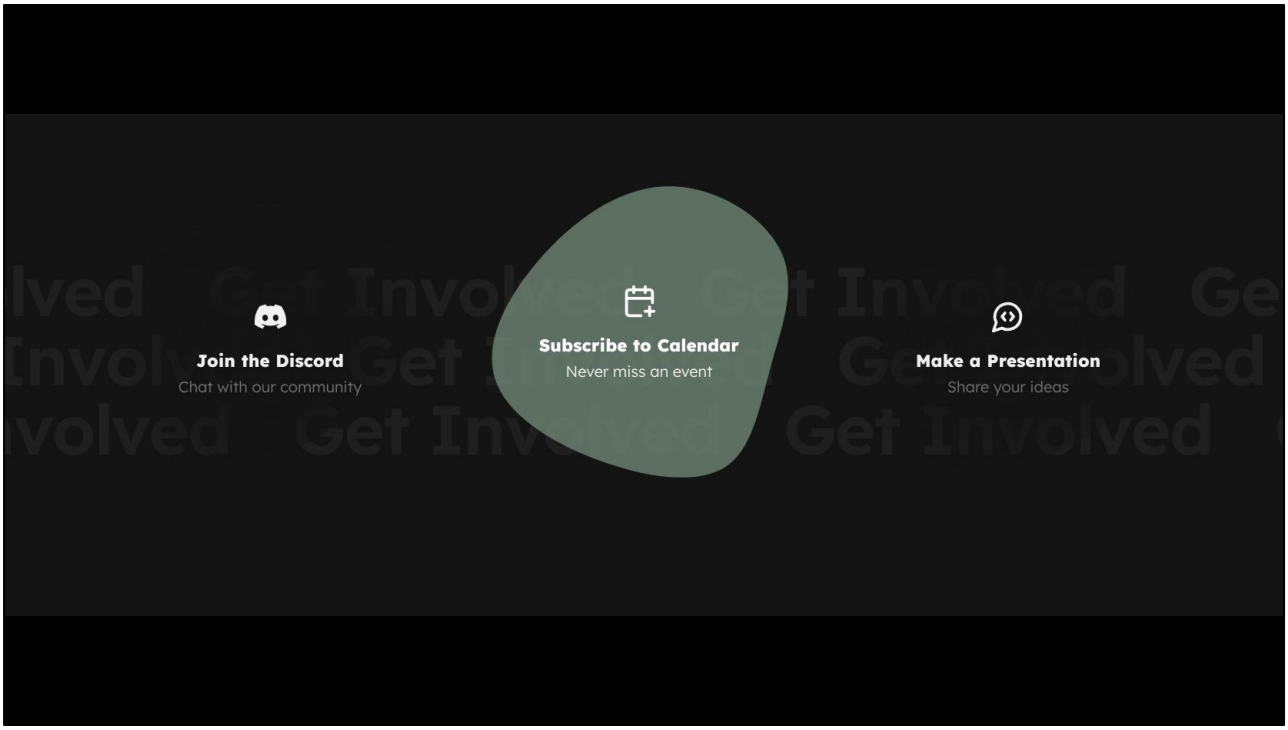
So please, let’s thanks Evey for her VERY hard work designing OKTech.jp!

Thank you Evey.

And a shout out also to Karim and Martin again for your continued input during the buildout, and of course, the Logo Crew.

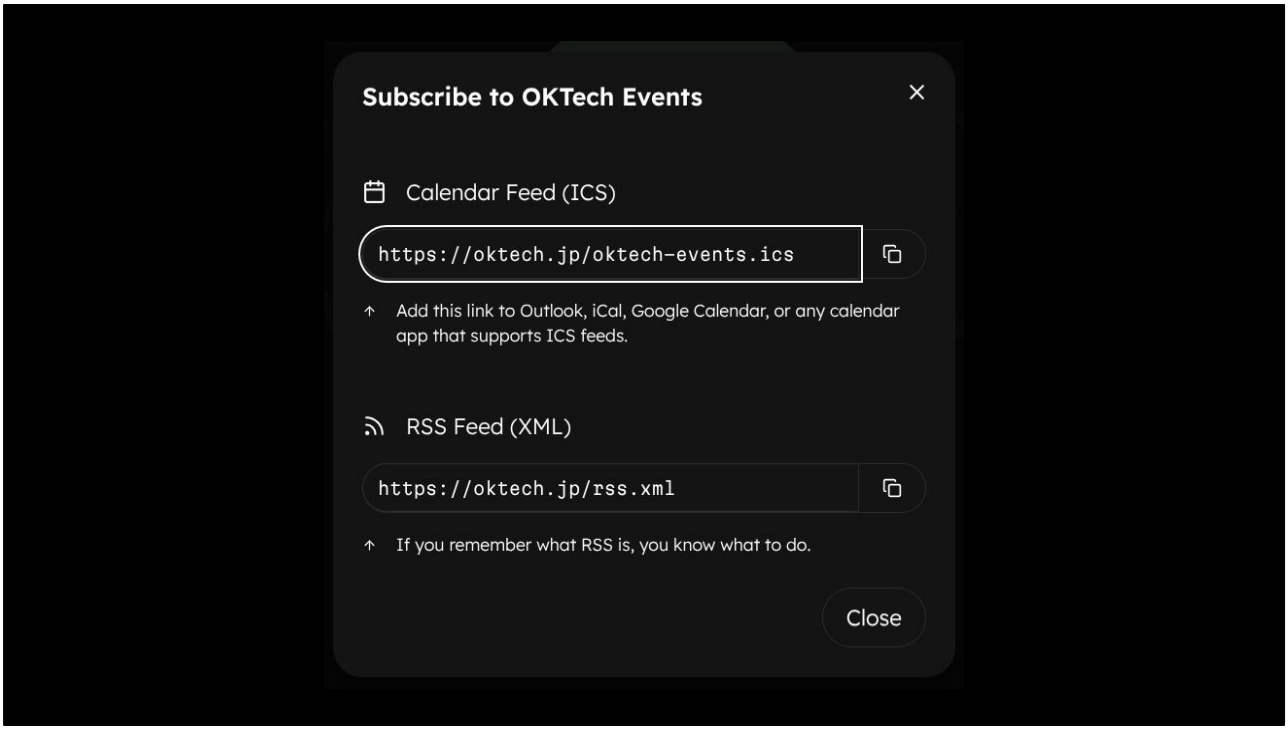
Please Visit OKTech.jp

The best way to show your appreciation would be to visit the website.

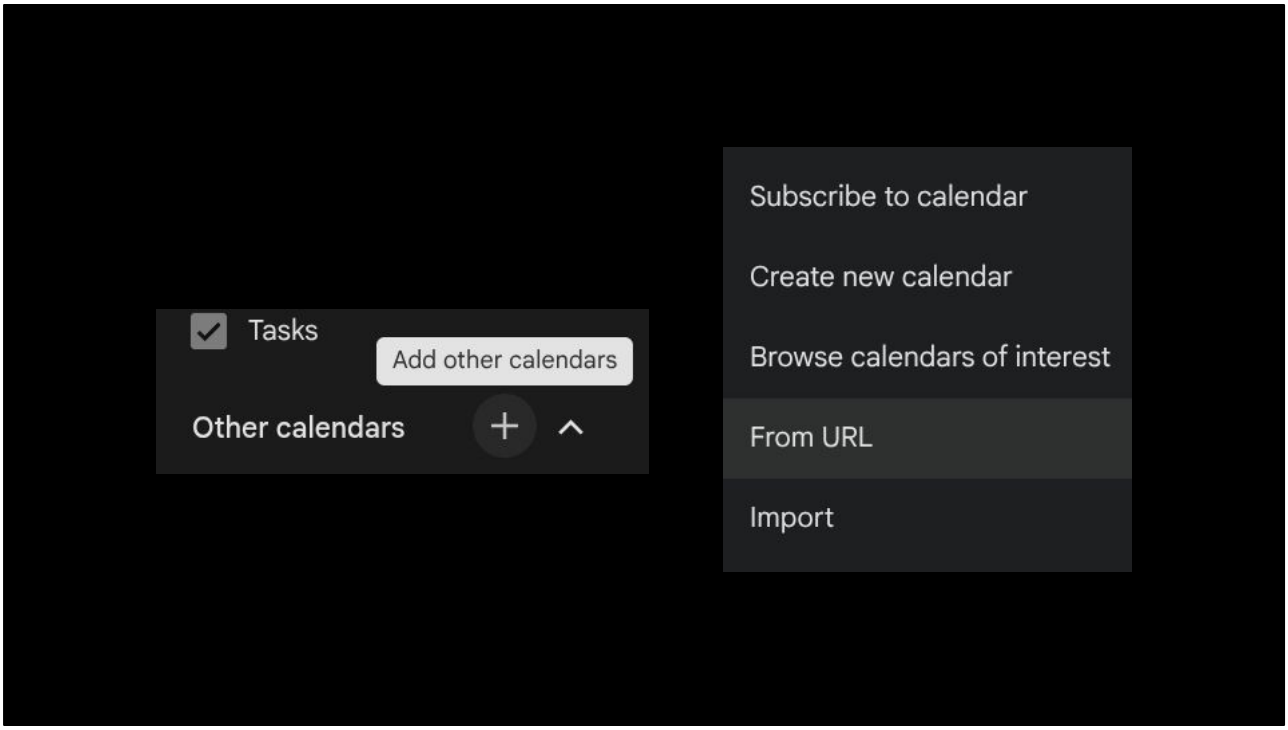


While you’re there, I ask that you Subscribe to the Calendar feed.

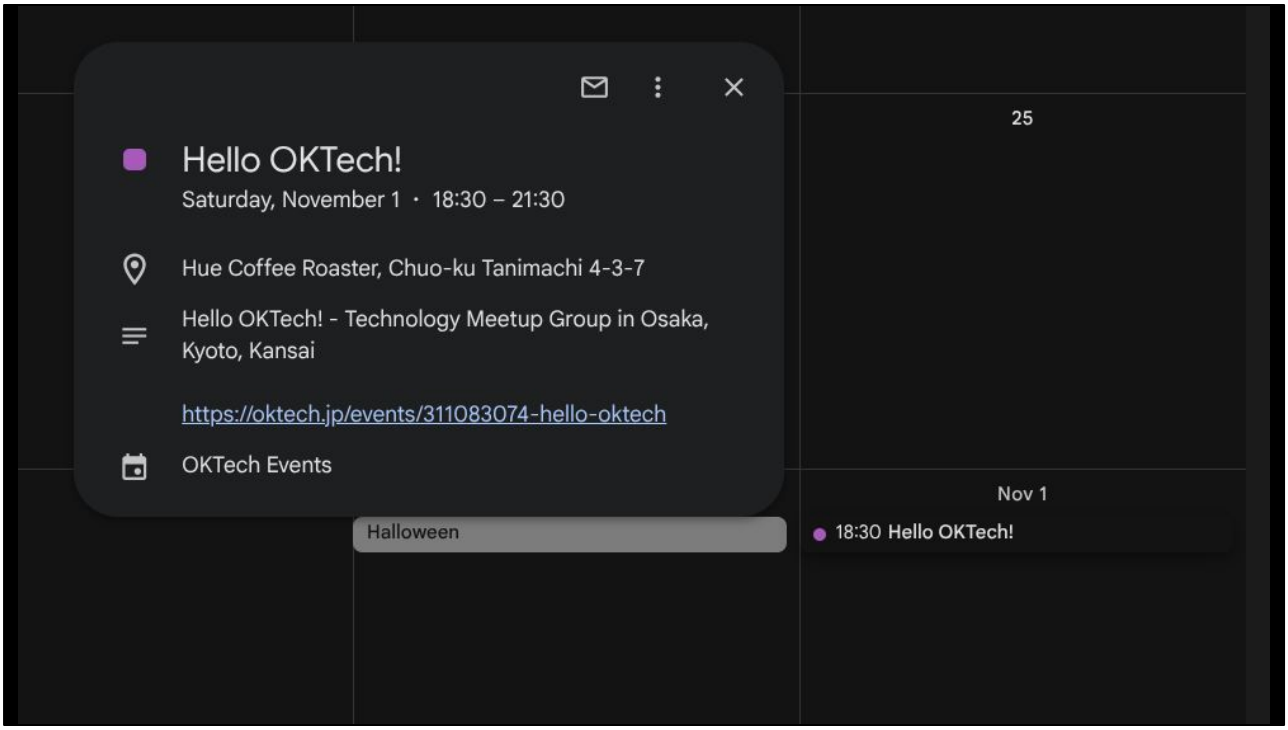
If you click this button,



You will be shown a URL that you can copy and paste into your favourite calendar app.



Here’s how you do it on Google Calendar.



And from then on, you'll automatically get updates about the latest OKTech events right in your calendar.

Many of us rely on Meetup.com, which is doing a fine job right now, but who knows what might happen in the future.

It's best to get updates straight from the source.

OKTech, Let's Tackle Tech Together!

Now,

No OKTech event would be complete without a bit of nerding out over technology.

So here is a collection of random tips, tricks and things I learned while making OKTech.jp.

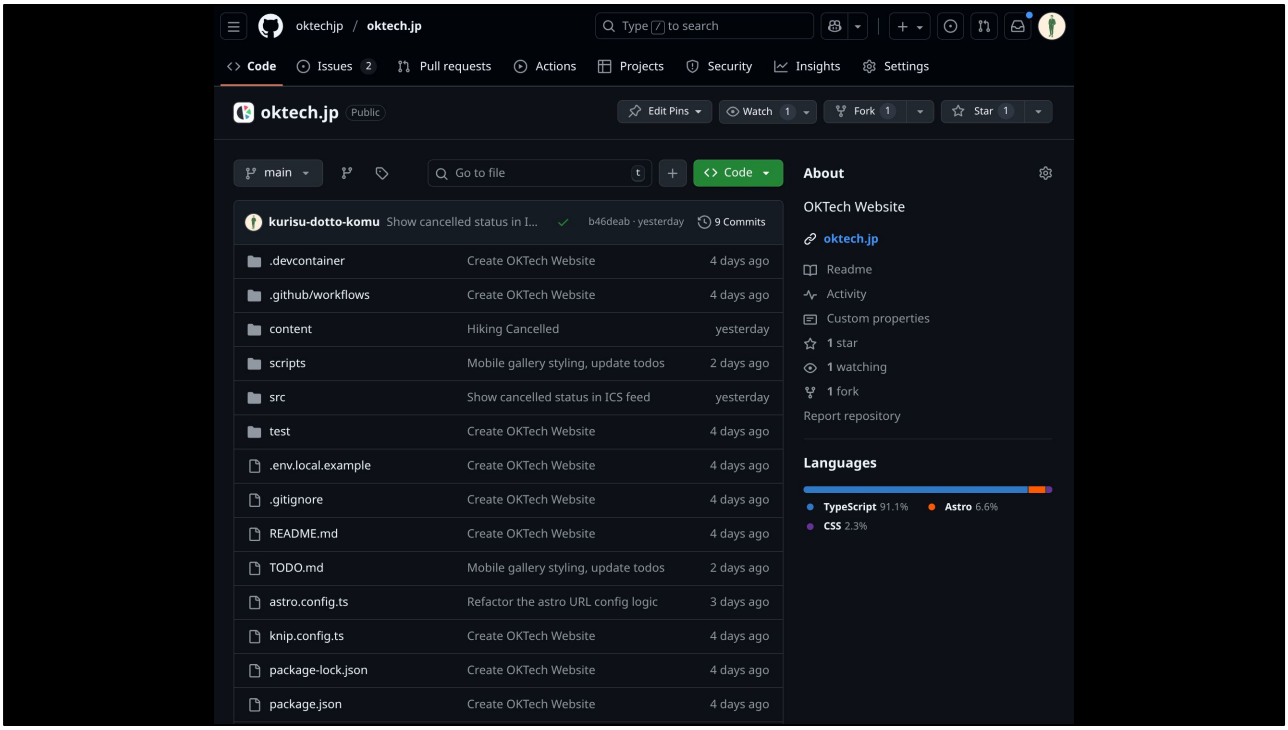
package.json
is your friend

The Web Devs probably already know this one,

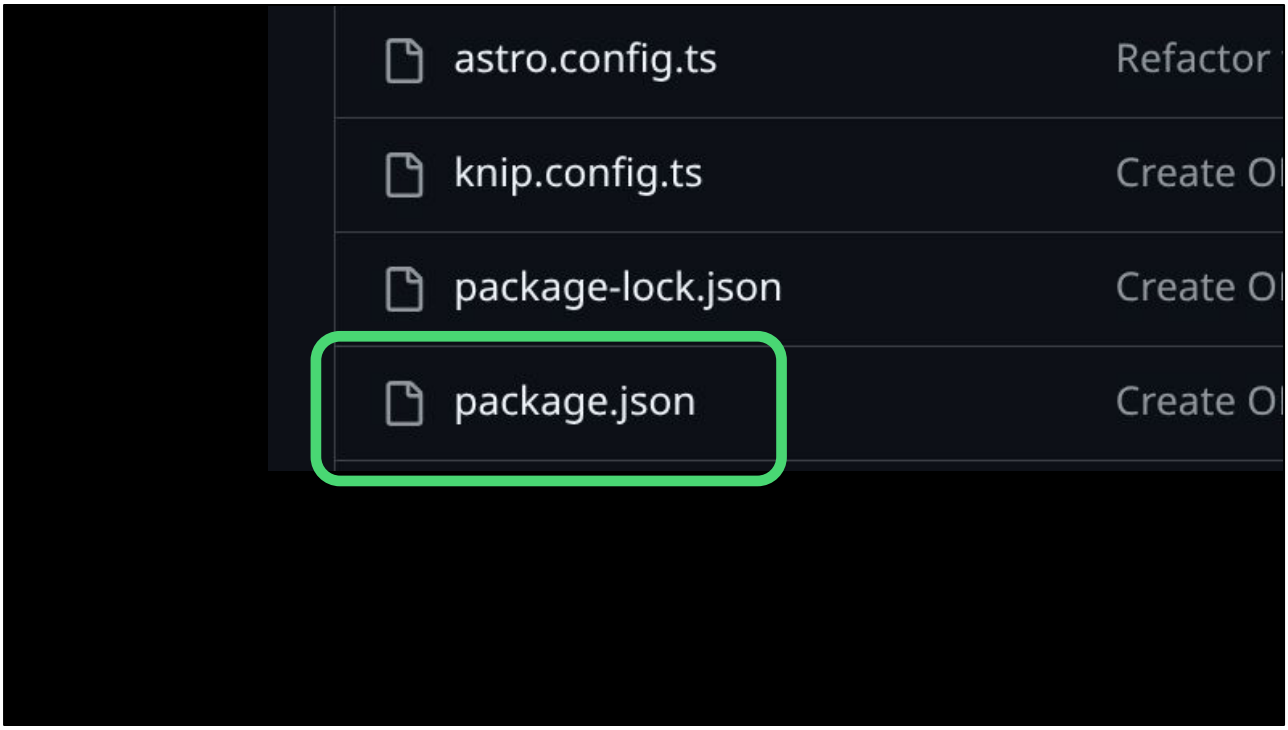
But if you happen to stumble upon a JavsaScript project in the wild,

a good way to understand what it's doing is to to take a peek inside

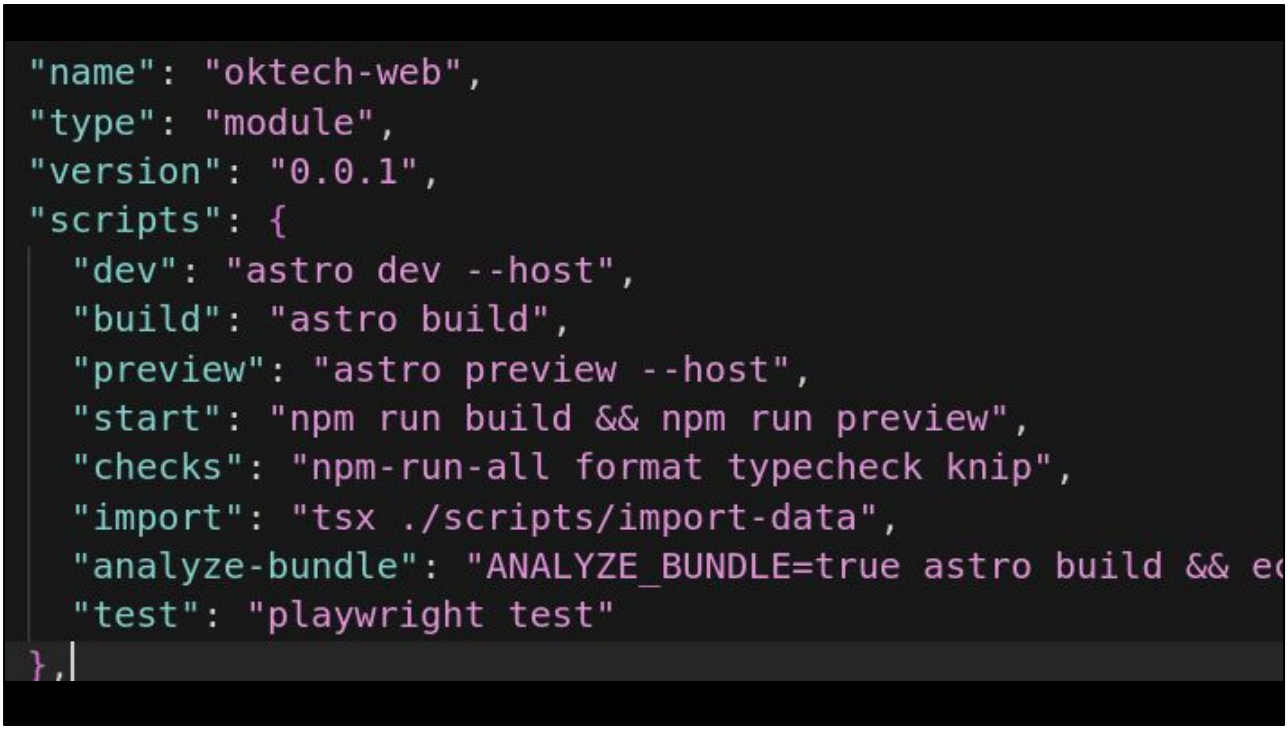
package.json



Here’s the OKTech website Github Repo - it’s open source of course.



And if you scroll down the list of files and and you'll see a **package.json**



```
"name": "oktech-web",
"type": "module",
"version": "0.0.1",
"scripts": {
  "dev": "astro dev --host",
  "build": "astro build",
  "preview": "astro preview --host",
  "start": "npm run build && npm run preview",
  "checks": "npm-run-all format typecheck knip",
  "import": "tsx ./scripts/import-data",
  "analyze-bundle": "ANALYZE_BUNDLE=true astro build && ec
  "test": "playwright test"
},
```

Inside that file, is a ‘scripts’ section, which will give you some clues as to what’s going on.

You can run any of these with `npm run`, `build` for example.

We got “dev” and “preview”, also using something called `astro`, and some other useful commands like “analyze-bundle”.

There's also a "dependencies" section.

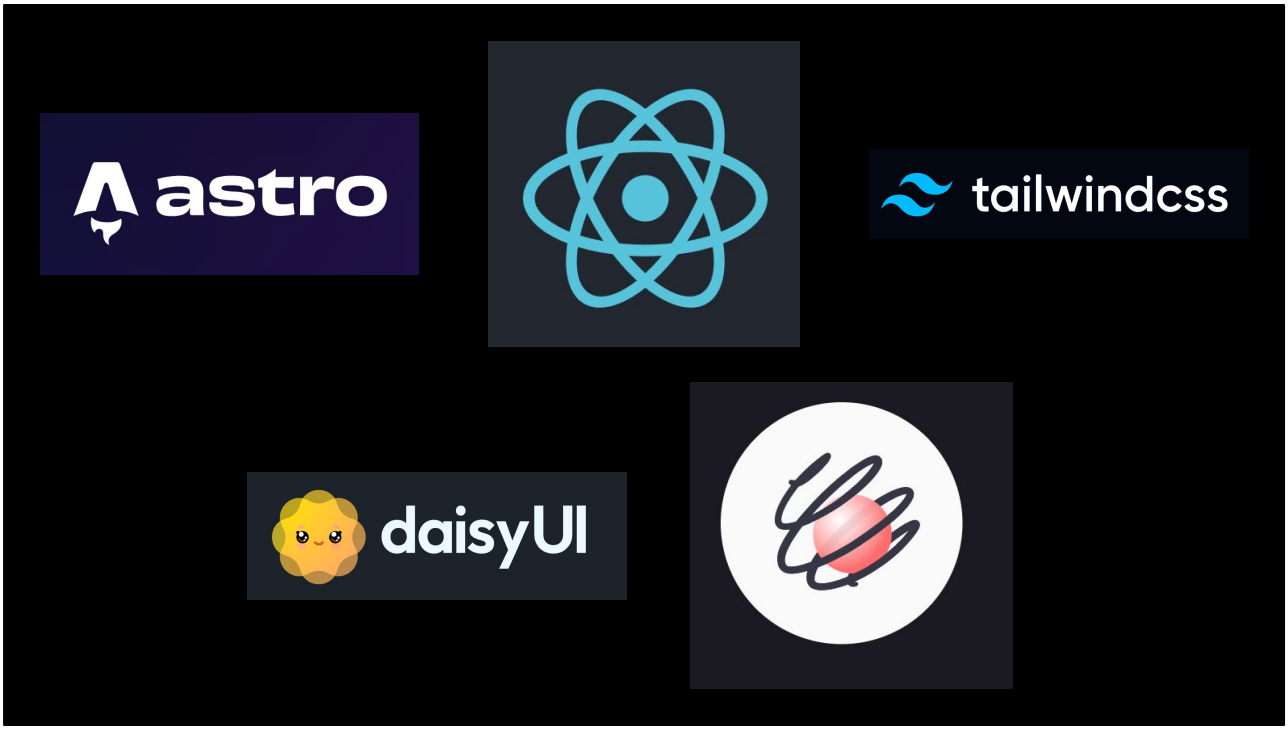
These are the third party libraries used by the project.

You can see Astro again.

Typescript, React, and Tailwind might be familiar.

Has anyone heard of Daisy UI?

How about React-spring?



That’s right, the presence of these libraries give you a good picture of how this project works.

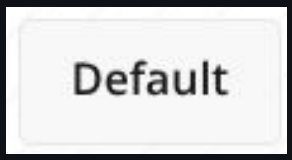
Astro is the foundation,
React for presentation,
Tailwind and Daisy decoration,
And React Spring does animation

Finally,
Daisy UI
is OK Tech

Since we were *just* talking about design, let's start with a closer look at Daisy UI

```
<button
  class="px-4 py-2 bg-gray-600 hover:bg-gray-700 text-black font-semibold rounded-lg shadow-md
  focus:outline-none focus:ring-2 focus:ring-gray-400 focus:ring-offset-2 transition"
>
  Default
</button>
```

```
<button class="btn">Default</button>
```

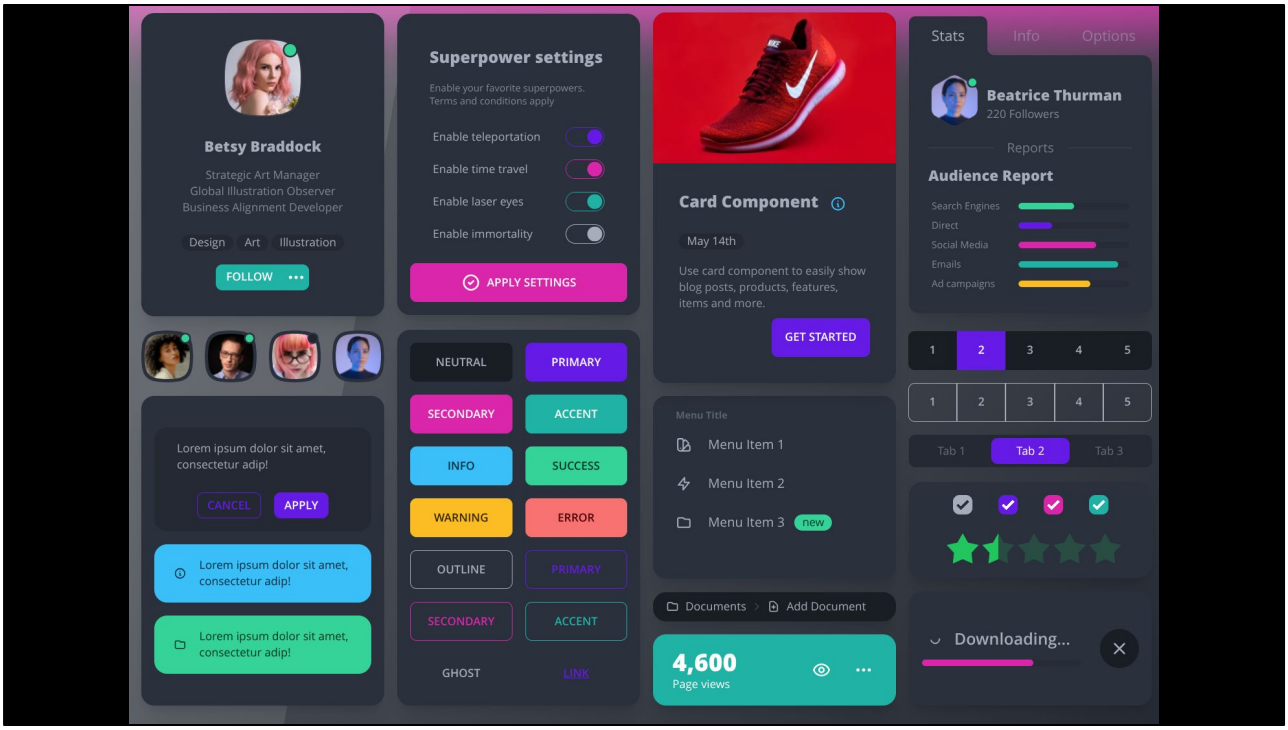


Daisy is the most popular free and open-source component library for the Tailwind CSS framework.

The basic idea is that it provides pre-made classes and components that don’t come with base Tailwind.

For example, instead of composing a bunch of classes yourself to construct a button,

you can just write btn.

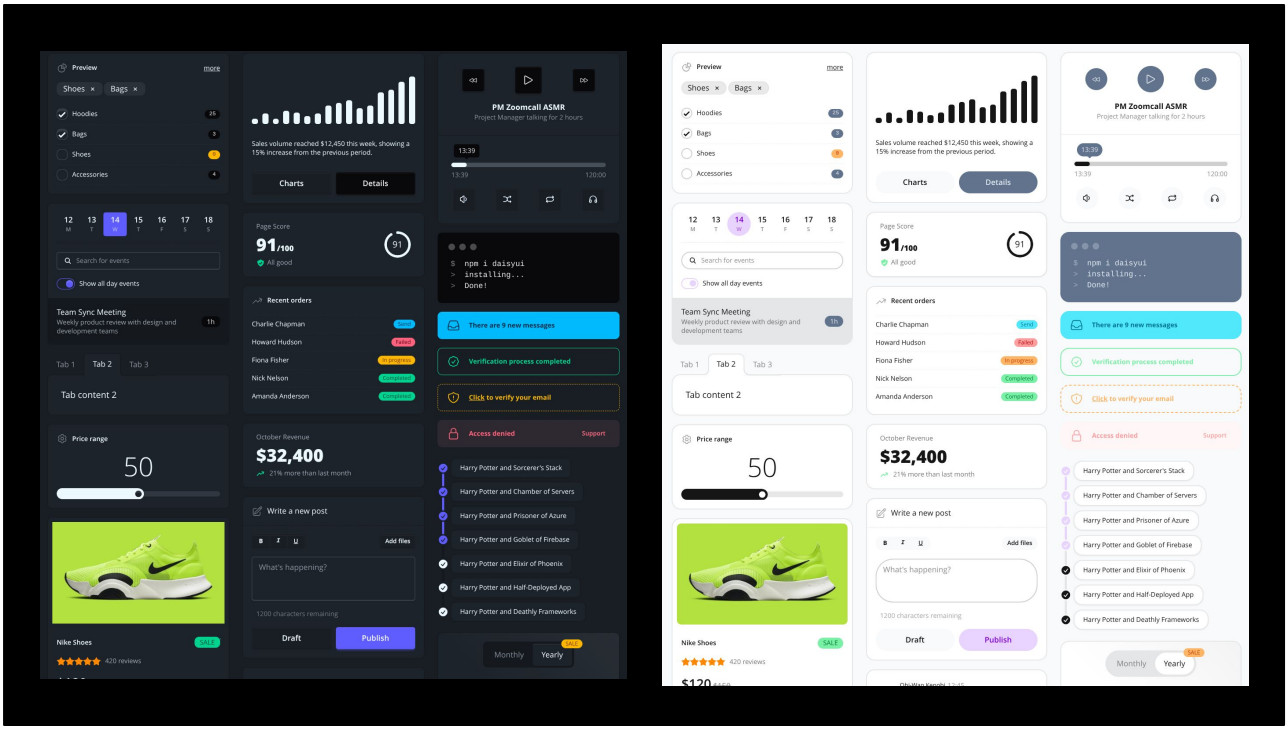


A few years ago I evaluated Daisy v3 in a project, and it was a pretty nice developer experience.

But to be frank, it was a little bit ugly.

Anyway, I thought I'd give it another chance for OKTech.jp

And with a bit of good timing, it turns out a new version of Daisy had just come out earlier this year.



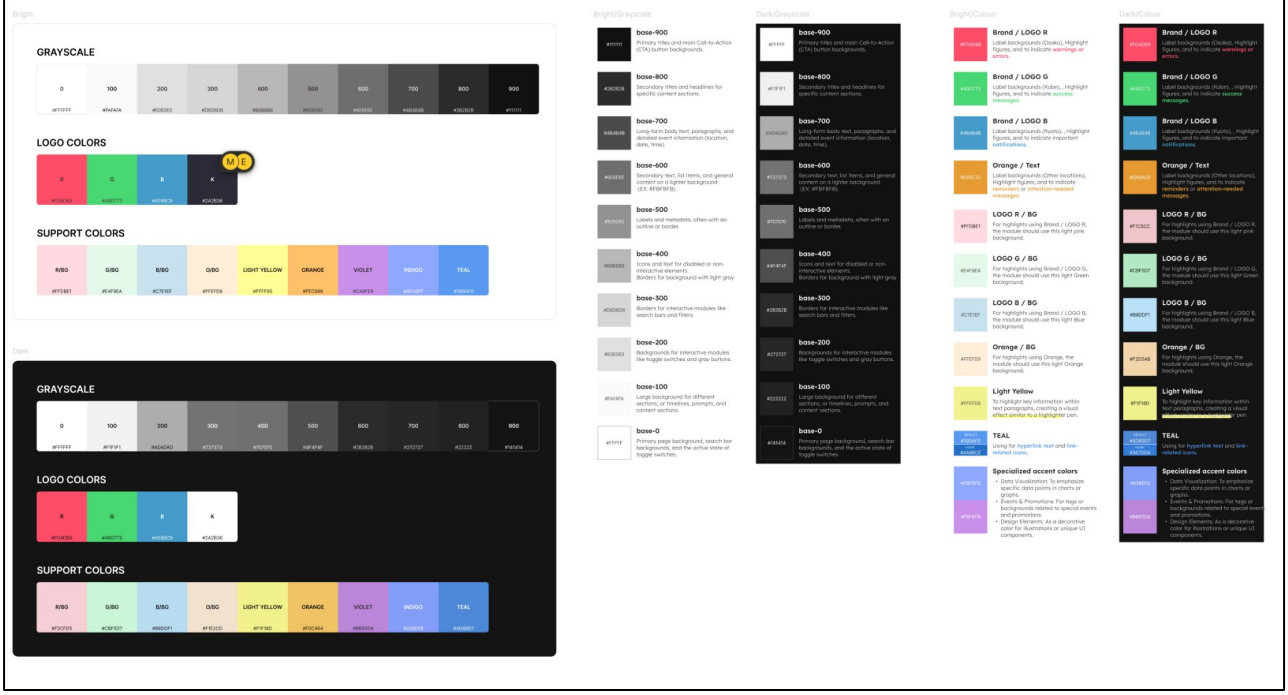
This is Daisy v5. Looking much more modern now.

All these components, and more, are provided.

Things like modals and popovers normally require custom logic in React, but Daisy handles this for you in native CSS.

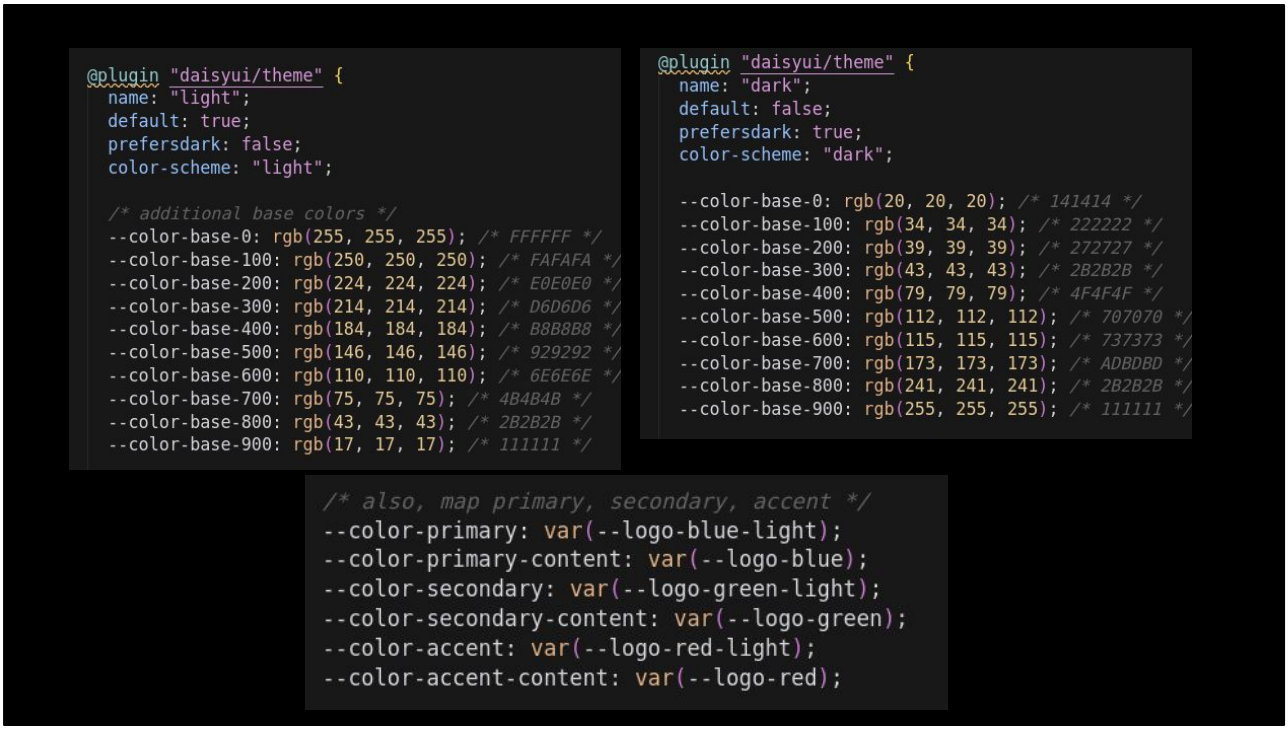
With Daisy, you can cover, like, 90% of common interactions, saving you time to focus on other stuff.

Since Daisy is built on top of Tailwind, you can also easily tweak components if they aren't perfect, by adding Tailwind classes.



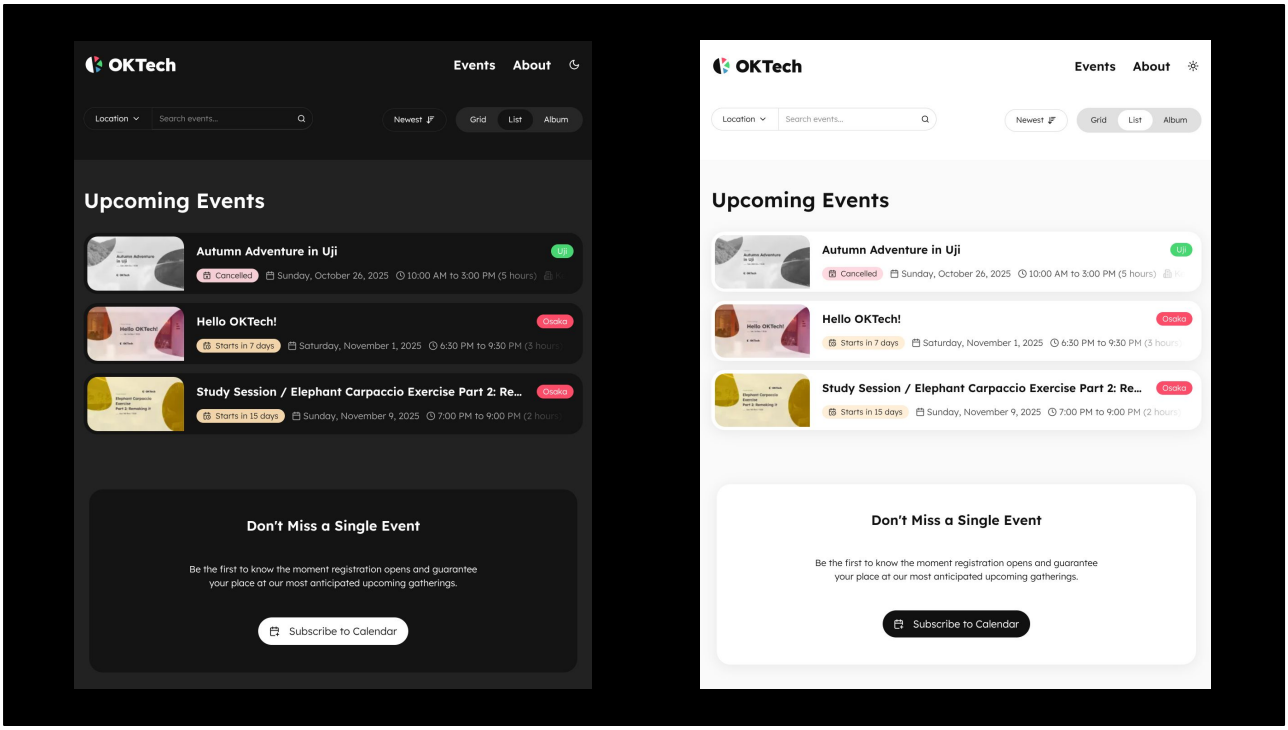
Moreover, Daisy has a robust theme system that integrates with Tailwind.

To create a custom OKTech theme, the color palettes provided by Evey could be simply copy and pasted into Daisy's config.



Defining CSS variables would either create new utility classes, like text-base-800

…or would override existing variables, and become magically integrated into the Daisy component library



And of course, Daisy supports dark mode!

Why should I care so much about Dark Mode, you ask?

Everybody Loves Dark Mode

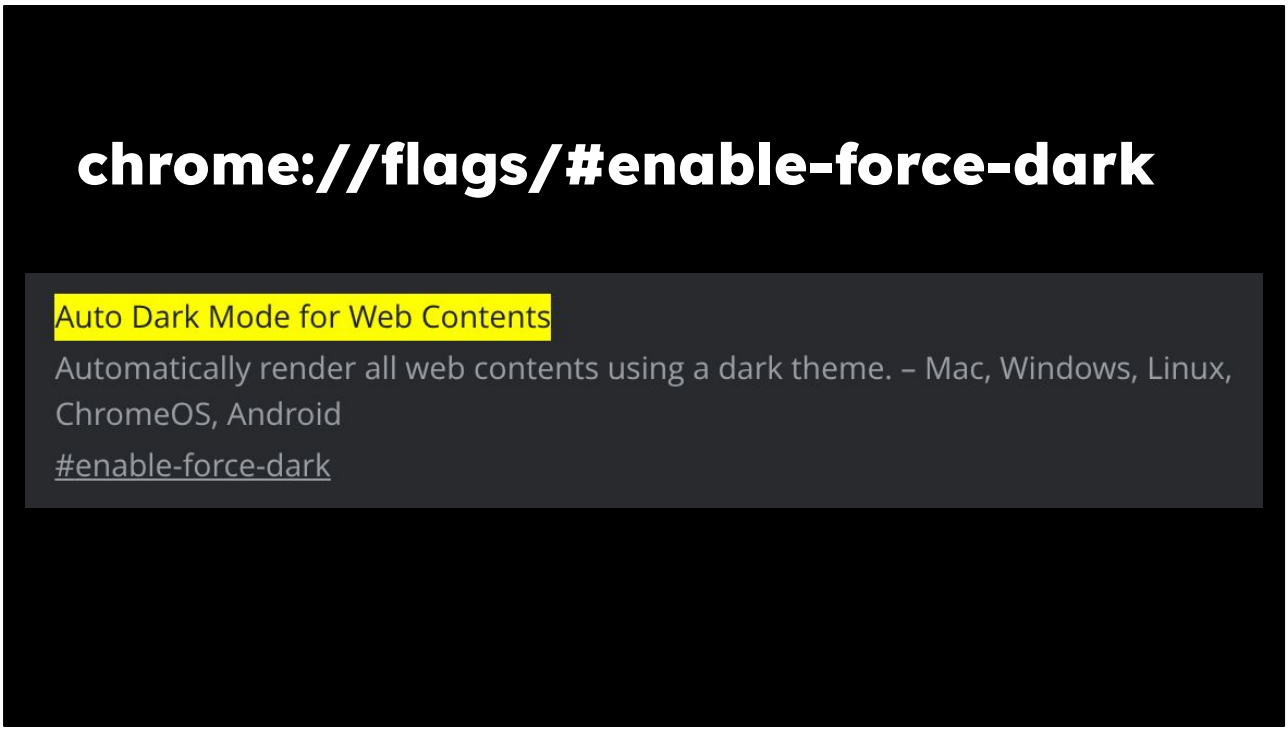
Well, *everyone* loves dark mode. Or at least they should.

What's not to love? Less battery wasted. Your eyes are less strained. You get better sleep.

With dark mode, people live happier, healthier, longer lives.

If everyone used Dark Mode, the world would be a better place.

For these reasons and more, Dark Mode is becoming trendy.



And before long, it will likely become standard practice.

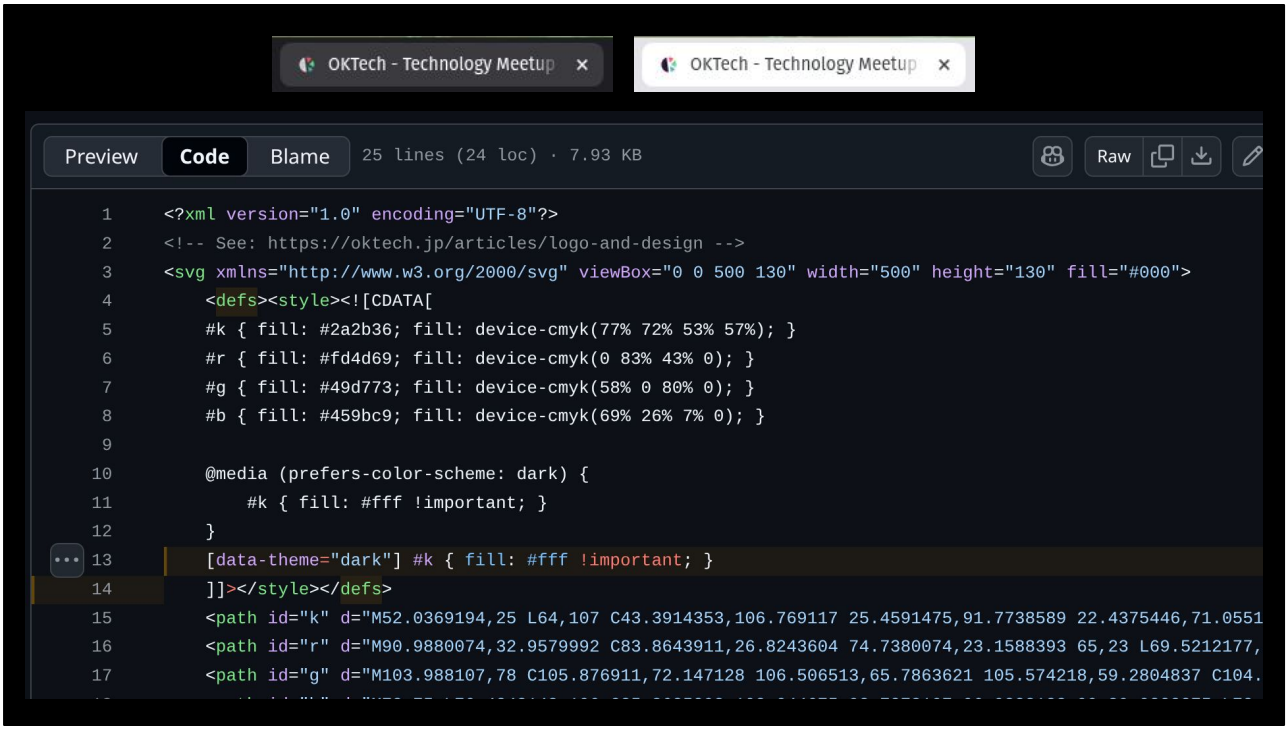
Did you know If you have a Chrome-based browser and enter this in the URL bar,

You can force all websites to render in dark mode.

Right now this feature is a hidden flag, but more browsers are enabling it as a non-hidden option.

If your website doesn't have a dark mode already, you'll soon lose control over what your visitors see when they visit your website.

Providing both a light and dark theme will ensure your visitors get an experience that you determine.



And did you know that SVG's support Dark Mode?

Here, you can see the same OKTech SVG favicon appearing in both dark mode and light mode, based on the operating system user preferences.

This is achieved with the `prefers-color-scheme` media selector, which can be used to override individual shape colors.

It's even applied in contexts outside the dom, such as bookmarks.

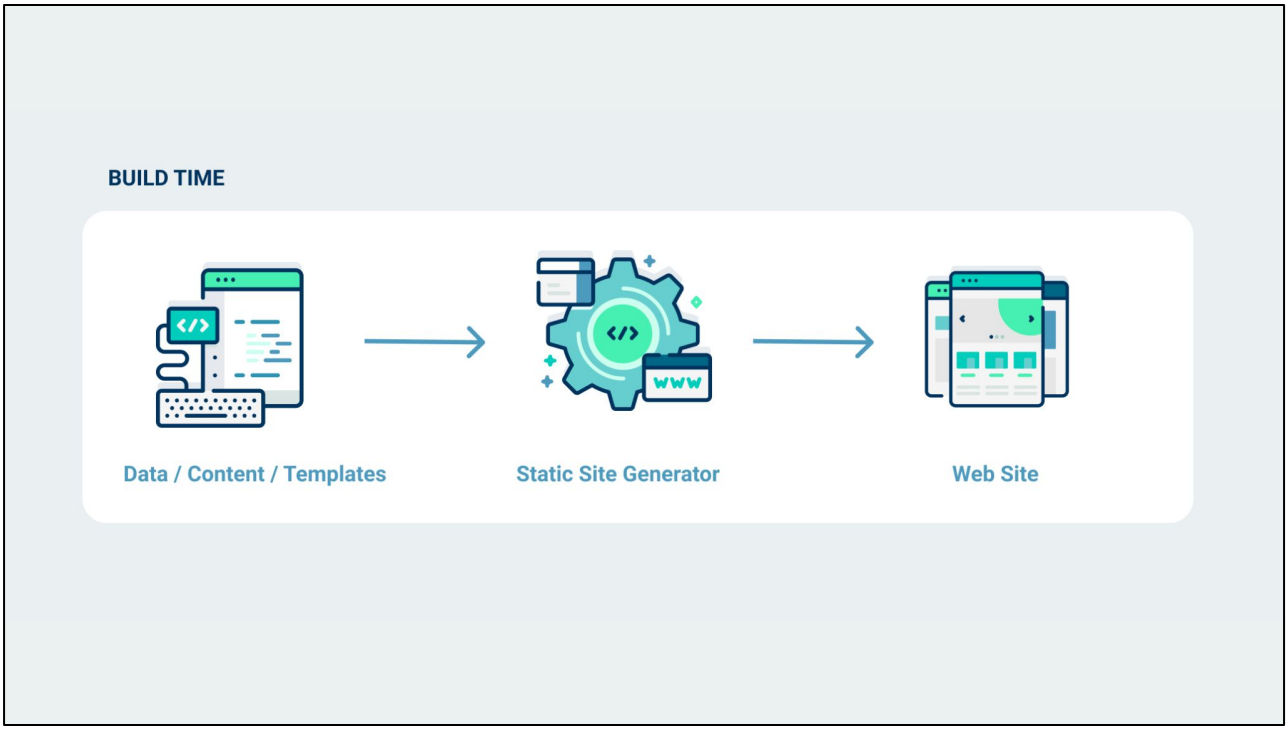
Just keep in mind that this doesn’t work on IE6 yet.

Dynamic
Static Site Generation

Let's move away form visuals now, and on to the logistics of how we deliver OKTech,jp.

You might have heard of **Static Site Generation**.

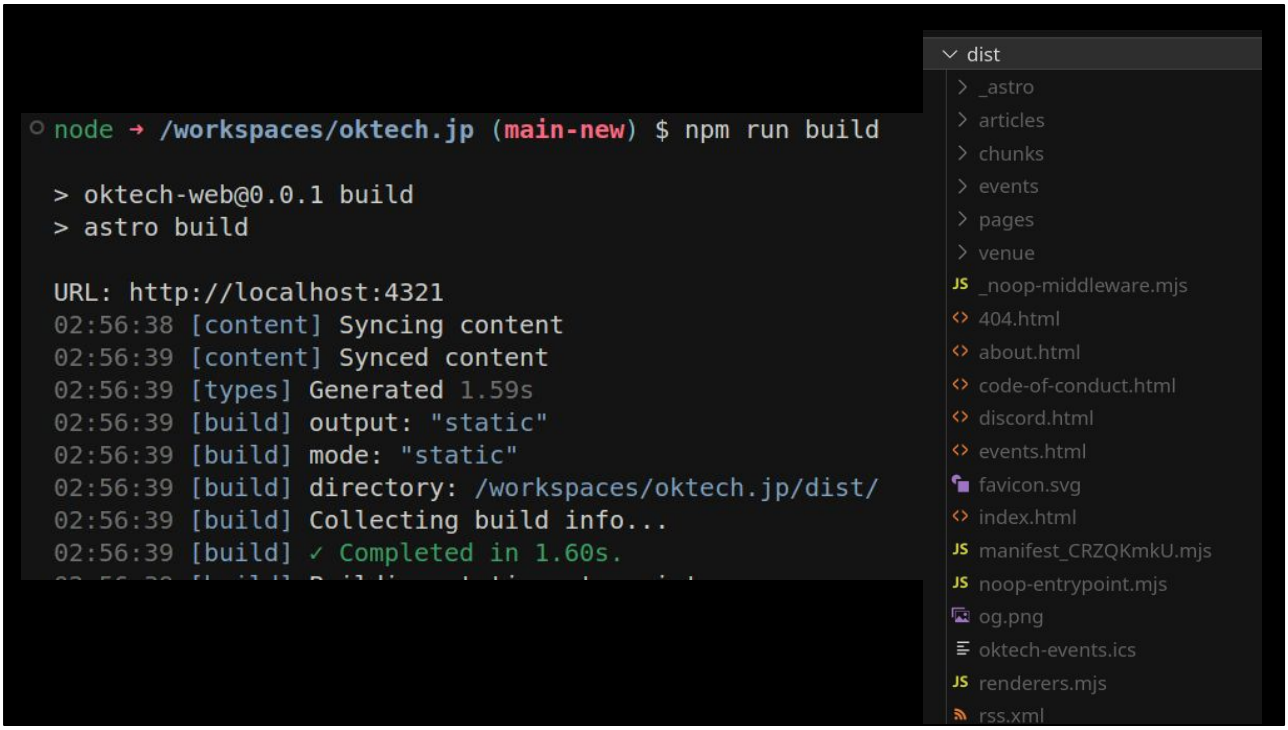
But I'm going be a bit special and call what we're doing **Dynamic** SSG.



To recap, “Static Site Generation”, is a technique used by frameworks like Jekyll or Gatsby, and started off as a way to create sites with very static content like a blog or API documentation.

Basically, you have a bunch of markdown and templates, you run a build process, and it outputs HTML and CSS that you can host wherever you like.

“Static” is good because, with no application server involved, it’s far less hackable, and incredibly cheap to serve.

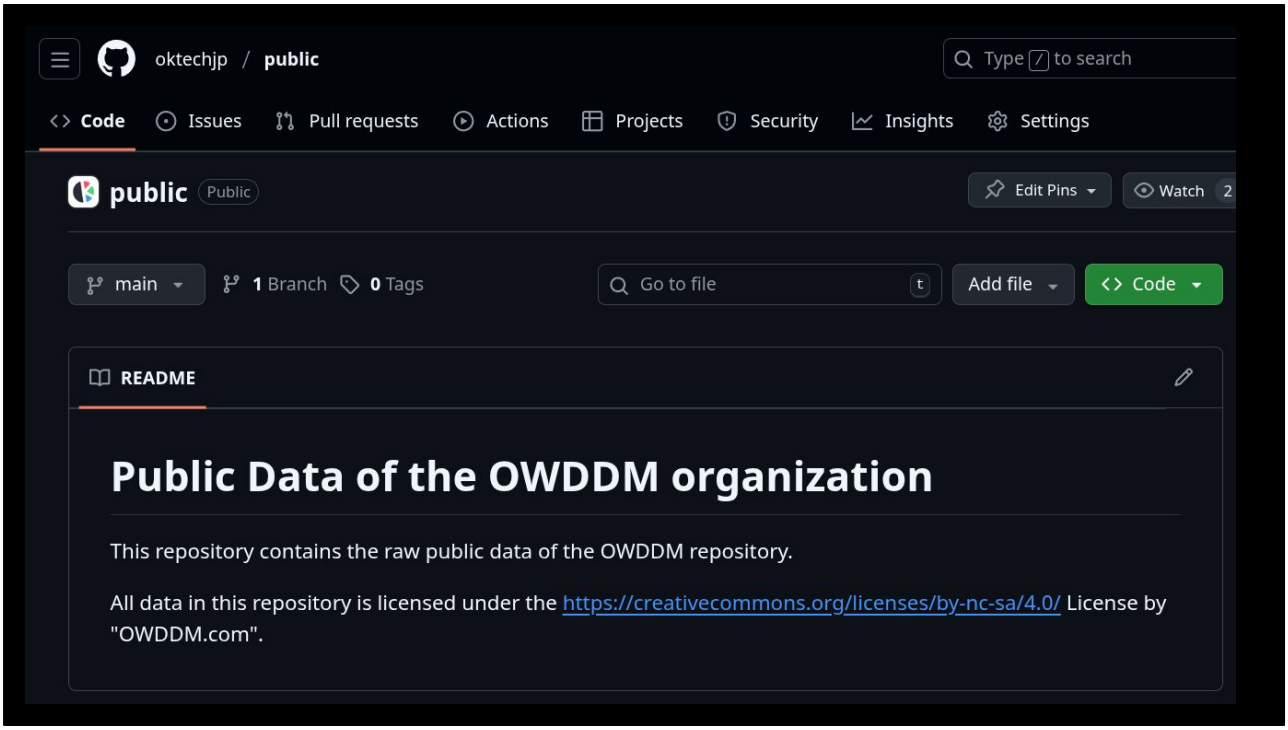


Indeed, for [OKTech.jp](), an `**npm run build**` step outputs a `dist` folder with a bunch of HTML files.

This output can be published to Github Pages, which provides free hosting.

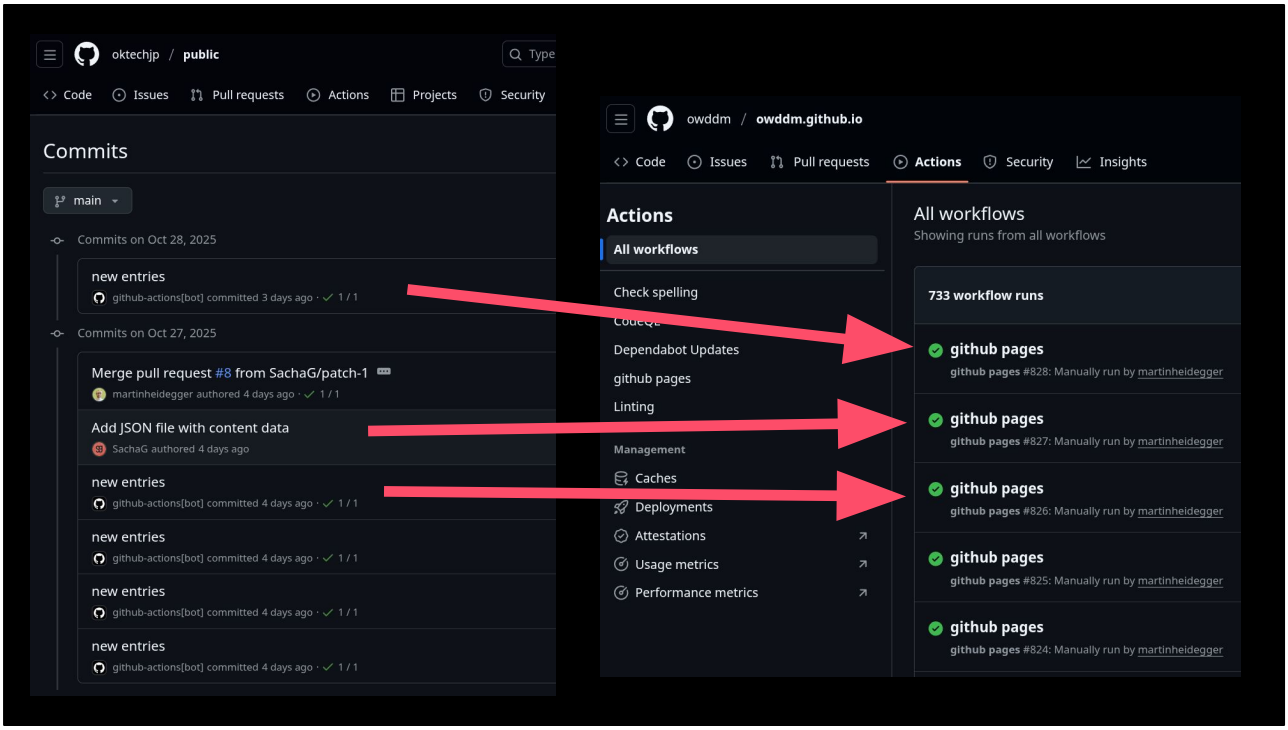
Normally, you'd set up your Github Repo to build and deploy any time files in the repo get updated.

But we're not normal.

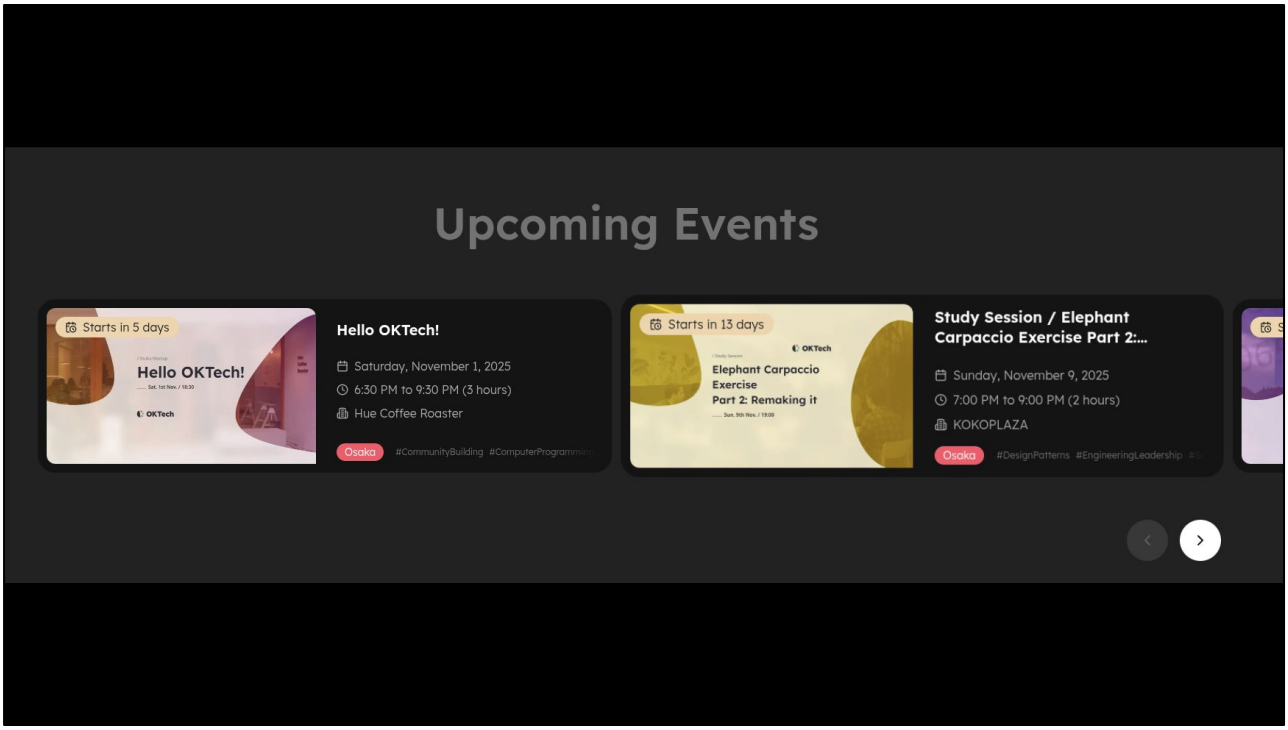


In the OKTech Github, there's this repo called 'public' where all kinds of data is stored.

It includes event information, but also images, business cards, graphic design, and more.



On the old site, whenever the **public** repo was updated, it would trigger a build of the **website** repo.



The ‘one trigger, one build’ approach is simple and works most of the time, but it’s not perfect.

Consider the new “upcoming events” section.

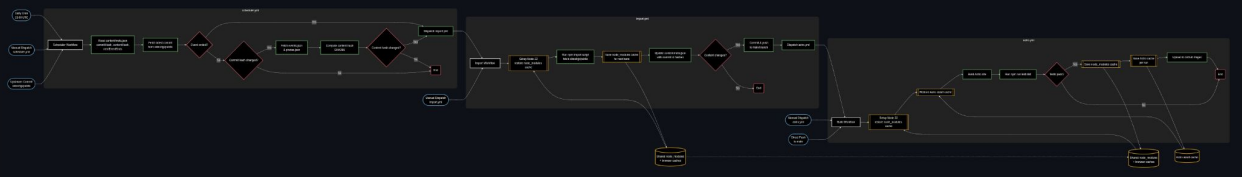
If we only trigger a build when we add a new event, then the this section will still show stale recently-ended events until a new event is added.

Github Workflows Are Super Powerful

So, we created some custom Github Workflows to provide upgraded functionality and optimized builds.

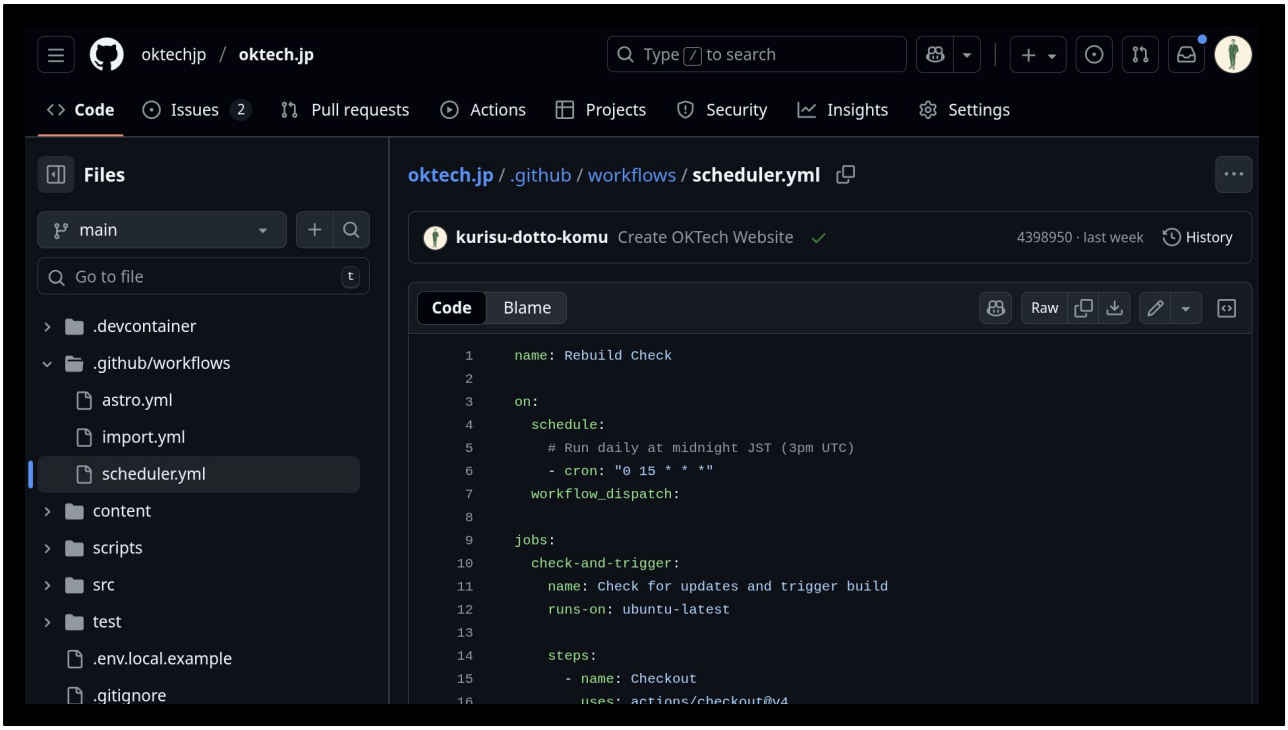
And I want to encourage you developers to try more complex Github Workflows, with the help of AI it’s not that difficult, and can be quite powerful.

Schedule -> Import -> Astro (Build)



In the case of OKTech, the new website has 3 Workflow files.

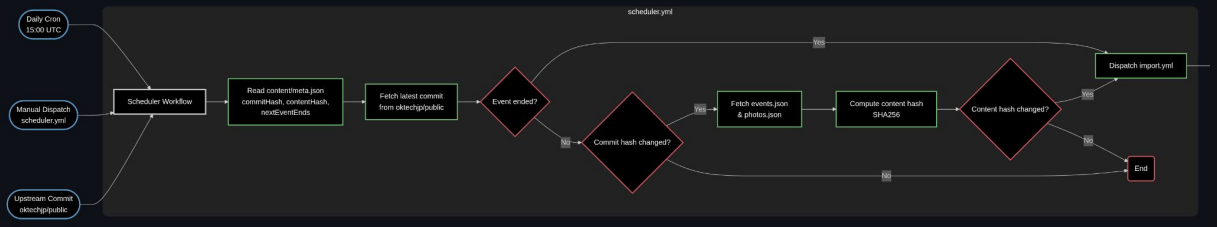
Scheduler, Import, and Build.



First, the Scheduler.

Github Workflows has a **chron** feature, which allows us to automatically trigger this workflow every day at midnight.

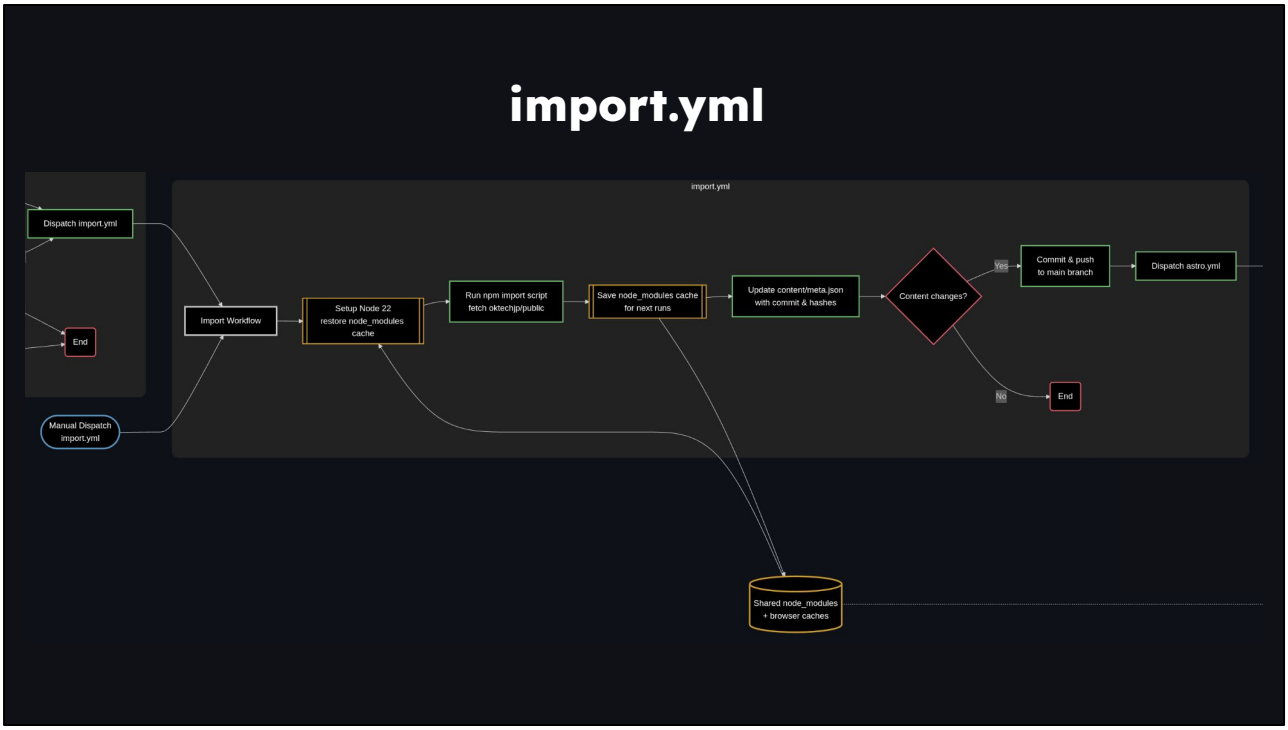
scheduler.yml



It can also be triggered manually, or whenever our that public repo gets updated, as before.

The scheduler will check if latest event has ended, or if the content hash has changed, and will trigger the next step if needed.

Otherwise, it does nothing, avoiding unnecessary builds.



Next, the Import script runs.

It pulls data from upstream sources, using a cache to speed things up.

If new data arrives, the import script will do some image optimizations, write markdown, generate maps, and commit changes, along with a content hash, to the website repo.

But wait, why are we committing data that already exists in public?

To Commit
Or Not To Commit?

Now THIS is a question I find myself asking far too often.

But in the context of the OKTech website, I think it makes a lot of sense to commit.

By committing the data, rather than just referencing it as external source, we get a number of benefits.

Reproducibility, Portability, Hotfixes, State Transitions, and more. I can explain more afterwards if you care.

—

Not only can we clone the repo and guarantee that we have a reproducible state, which helps for development and debugging.

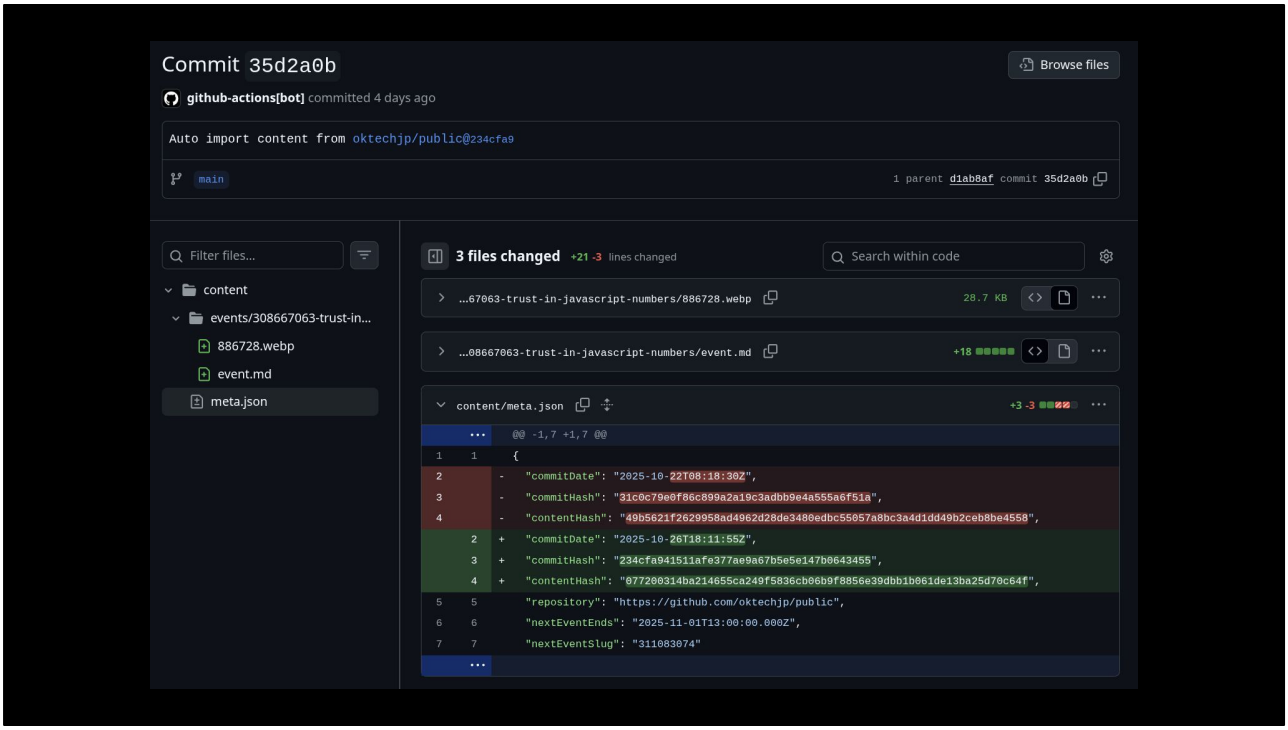
But we,

- Guarantees that the published build will look as we intend
- Can make emergency edits and hotfixes more easily
- Gain the ability to check a committed content hash to avoid unnecessary rebuilds

- Further save time on builds by not having to re-process images on a cold cache
- Give ourselves flexibility if we need to migrate to a different or multiple data sources in the future
- Can do more exotic transforms, such as map generation

And

- Easily compare state changes, allowing us to, for example, create an RSS feed entry if the status of an event is changed



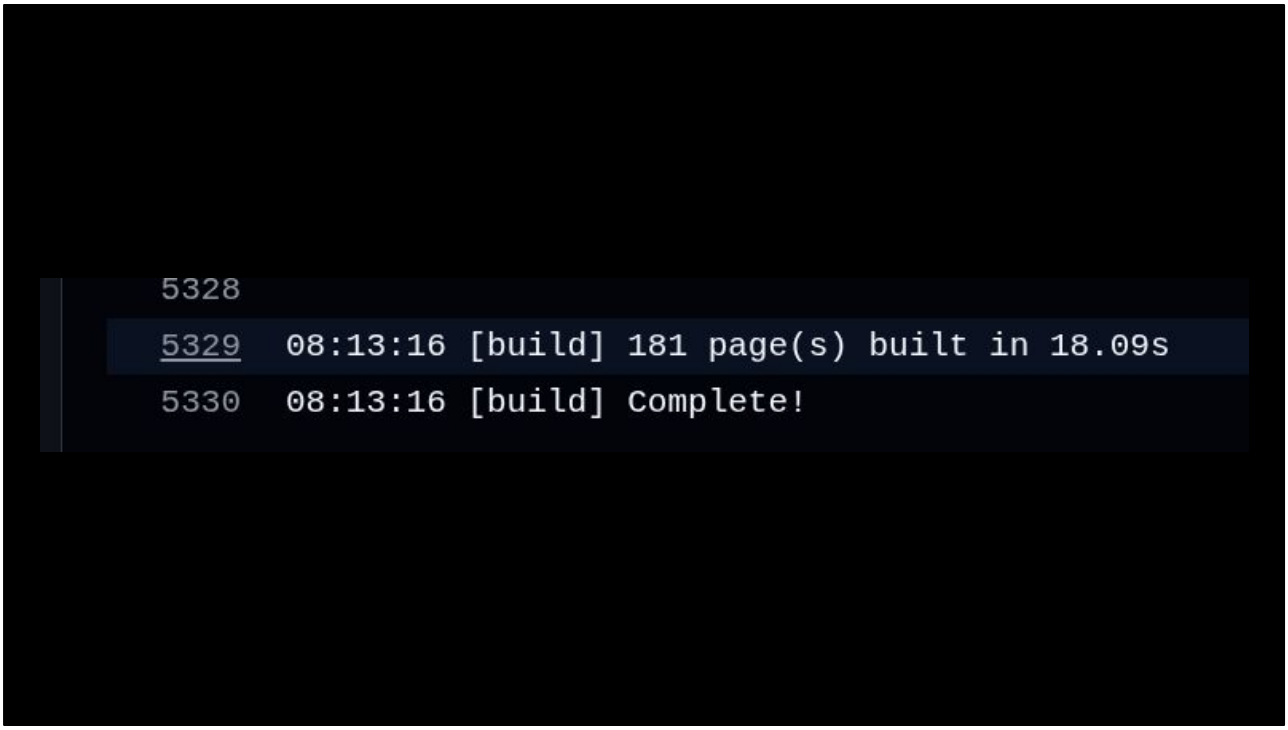
Based on these advantages, the workflow is set up so that as new data come in,

it creates a new commit, including a content hash in meta.json so the scheduler can easily check if it needs to trigger a build next time.

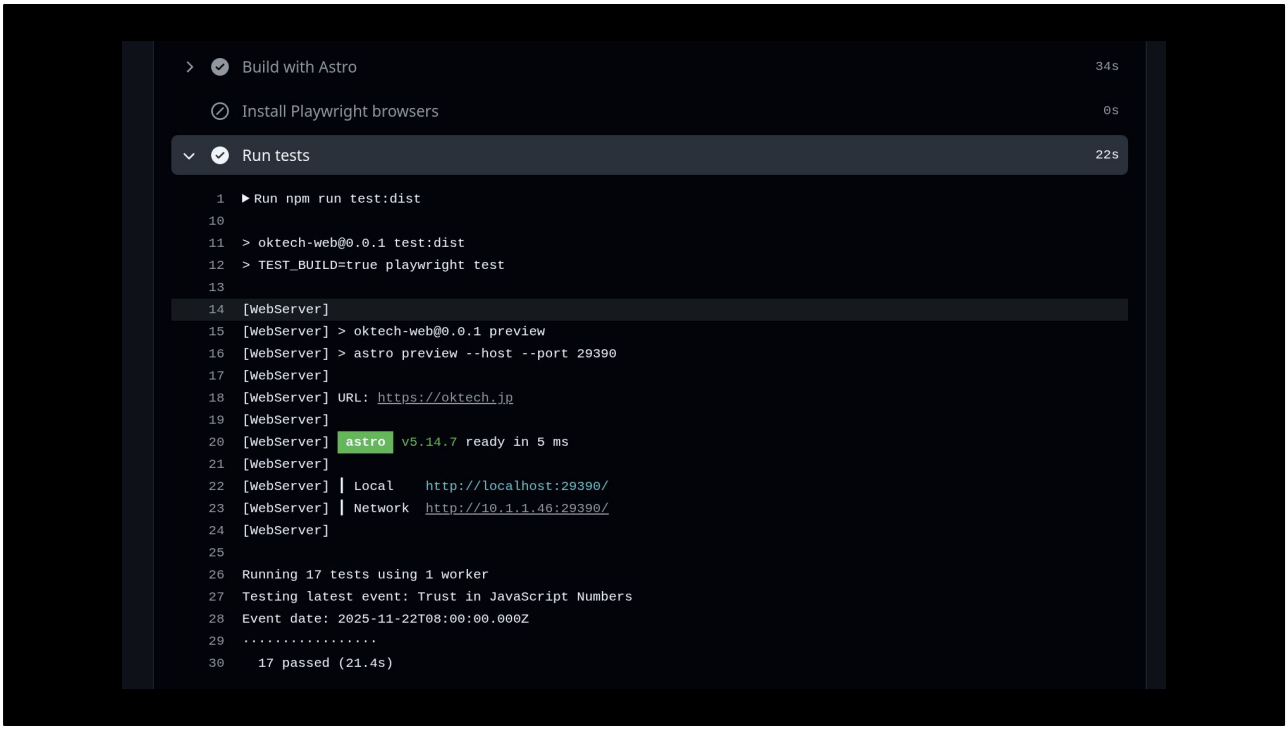
astro.yml

The final workflow is the build itself.

Again, with heavy use of caching,

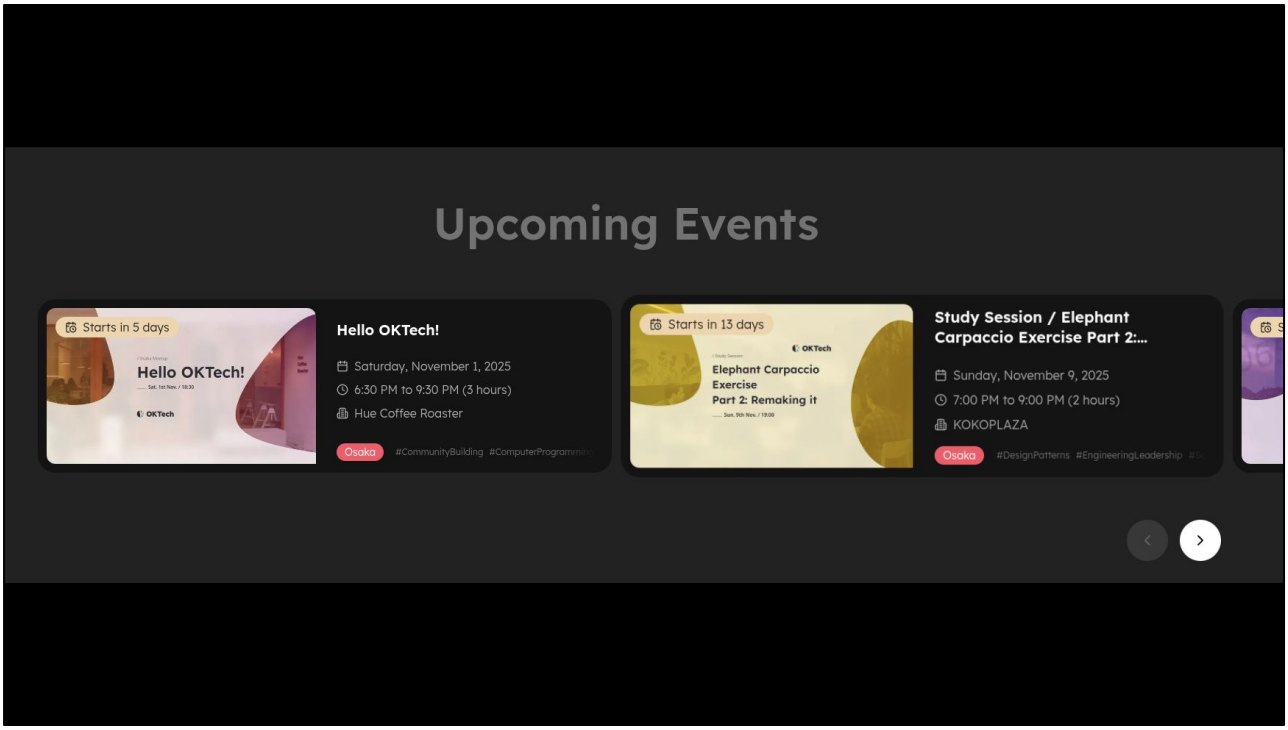


Even with thousands of images, Astro breezes through the build in a few seconds.

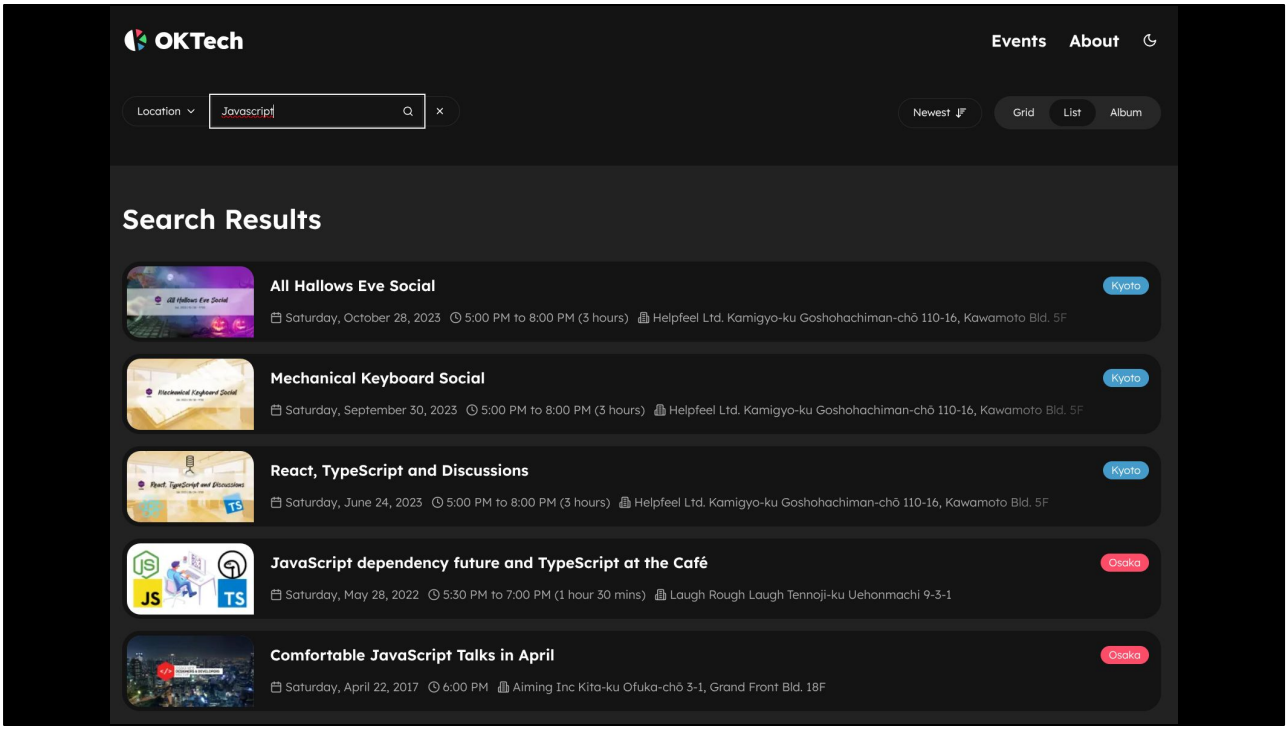


Before we actually deploy, we run a sanity check test against the distribution.

We spin up playwright, and ensure the latest event is visible on the front page and the events page.



And the result is that we now have “dynamic” static HTML, and an upcoming events section that is always up to date.



Of course, the other “Dynamic” aspect of our SSG is that:

Even though the HTML files are static, we can still run Javascript in the client.

Since Astro lets us use React on the frontend, so we can opt-in to Client Side Rendering whenever we want to do something more ‘Dynamic’, like Searching or Animations.

When using the search box, this feels like a typical client-server interaction.

But we’re actually just fuzzy-filtering, with a client-side with a library called fuse.js.

Astro is Hands-Down The Best DSSG Framework

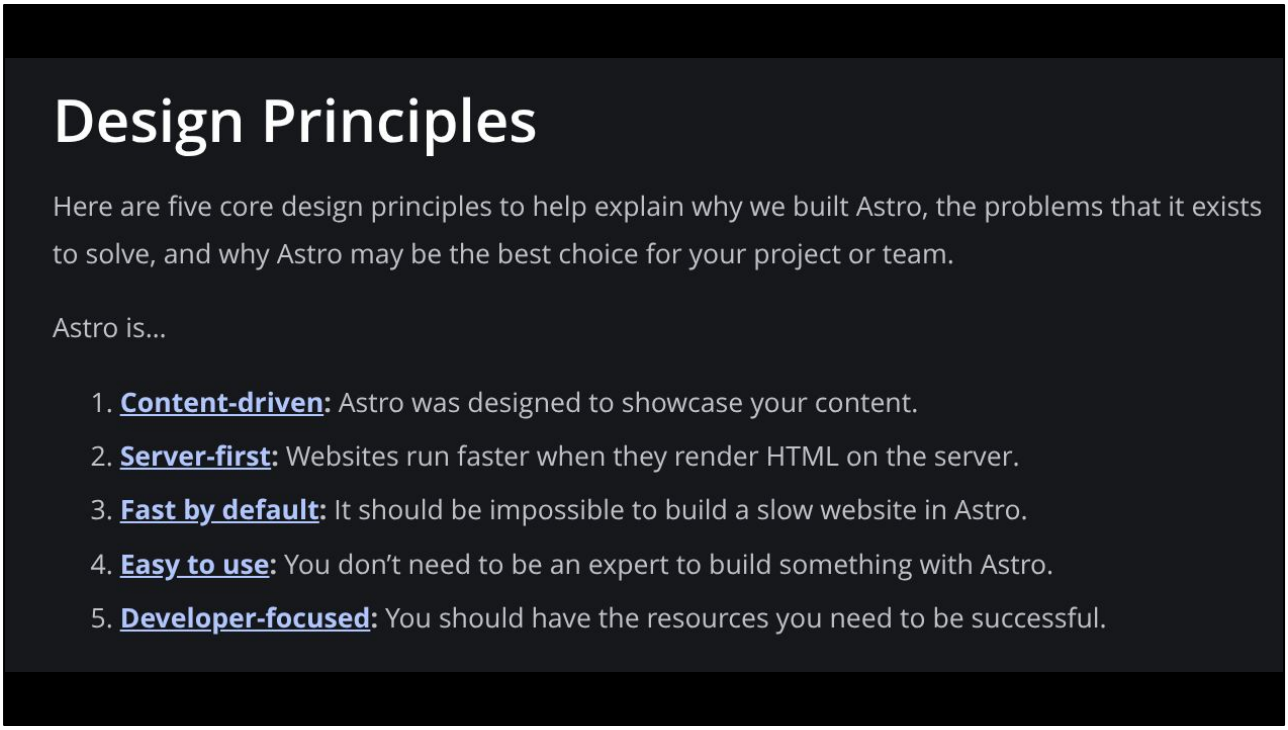
One of the reasons I was excited to help with OKTech.jp is because it would give me an opportunity to use Astro for the first time.

I've built SSG sites before in Gatsby and Next, but those tools have drawbacks.

Needless to say, I am now an Astro convert.

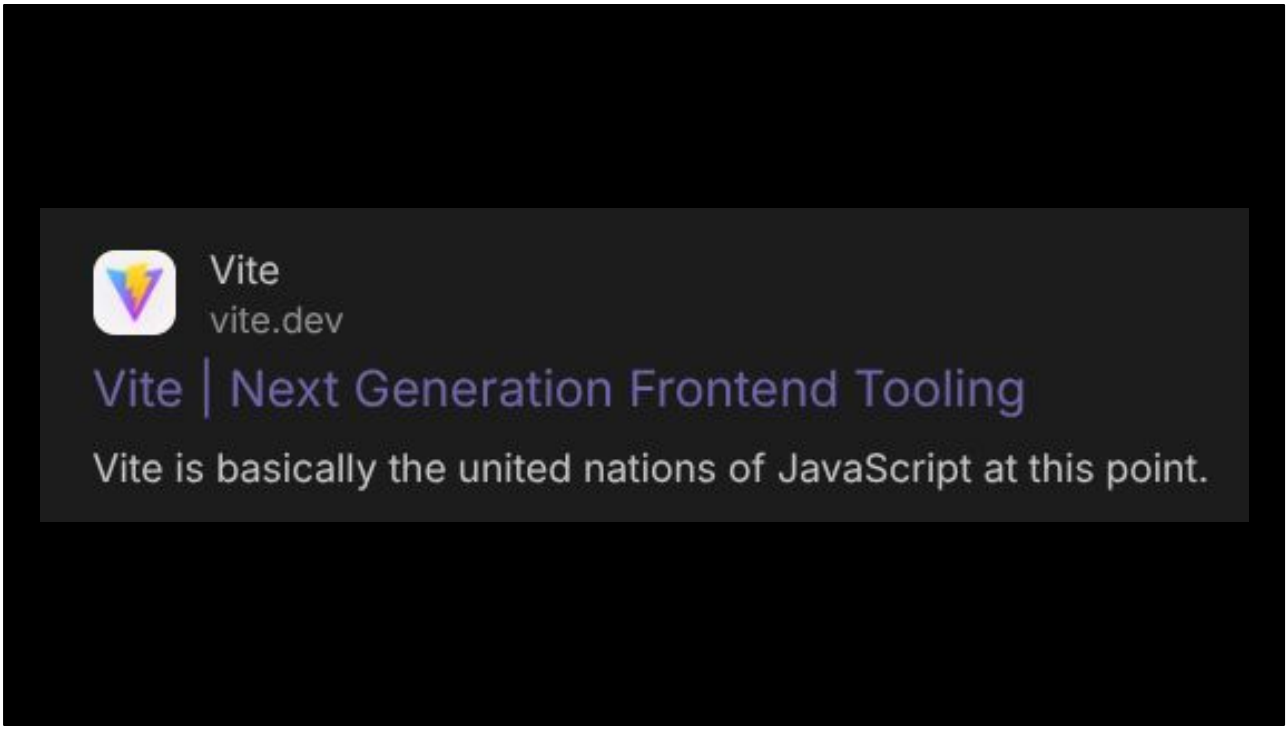
Would I recommend it for everything? No.

But for content-first SSG, Astro the winner.



The key point of is that astro is FAST.

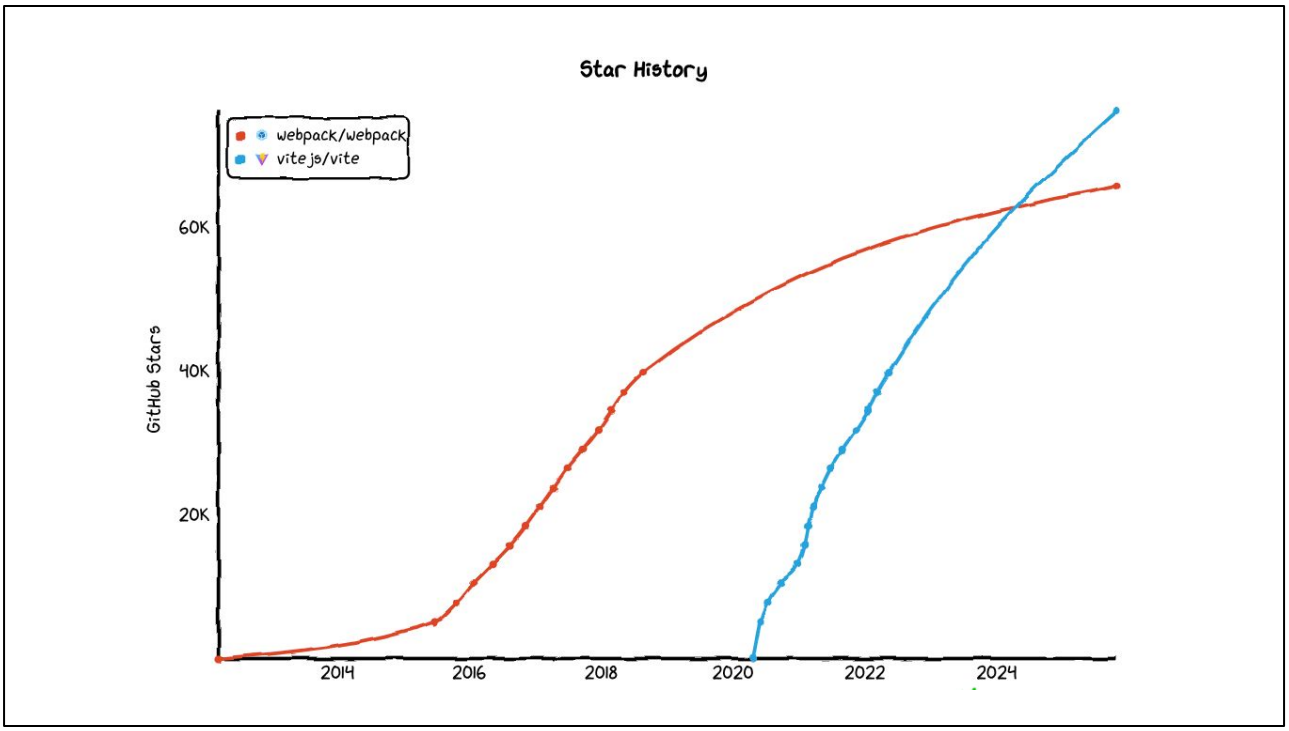
Not just for the end user, but for the developer too.



One not-so-secret weapon of Astro is that under the hood, it uses Vite.

A modern, blazing-fast bundler and frontend build tool.

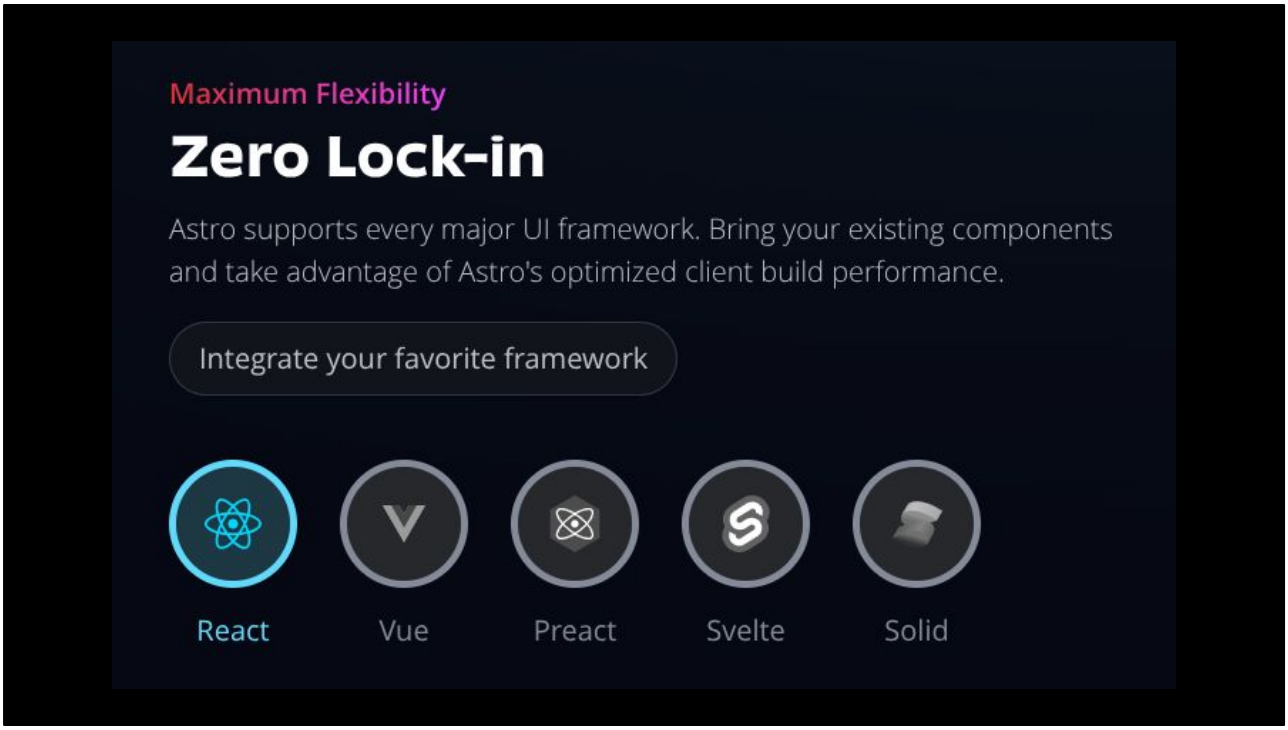
According to Vite, it’s the United Nations of Javascript at this point.



You can think of it like the post-covid webpack.

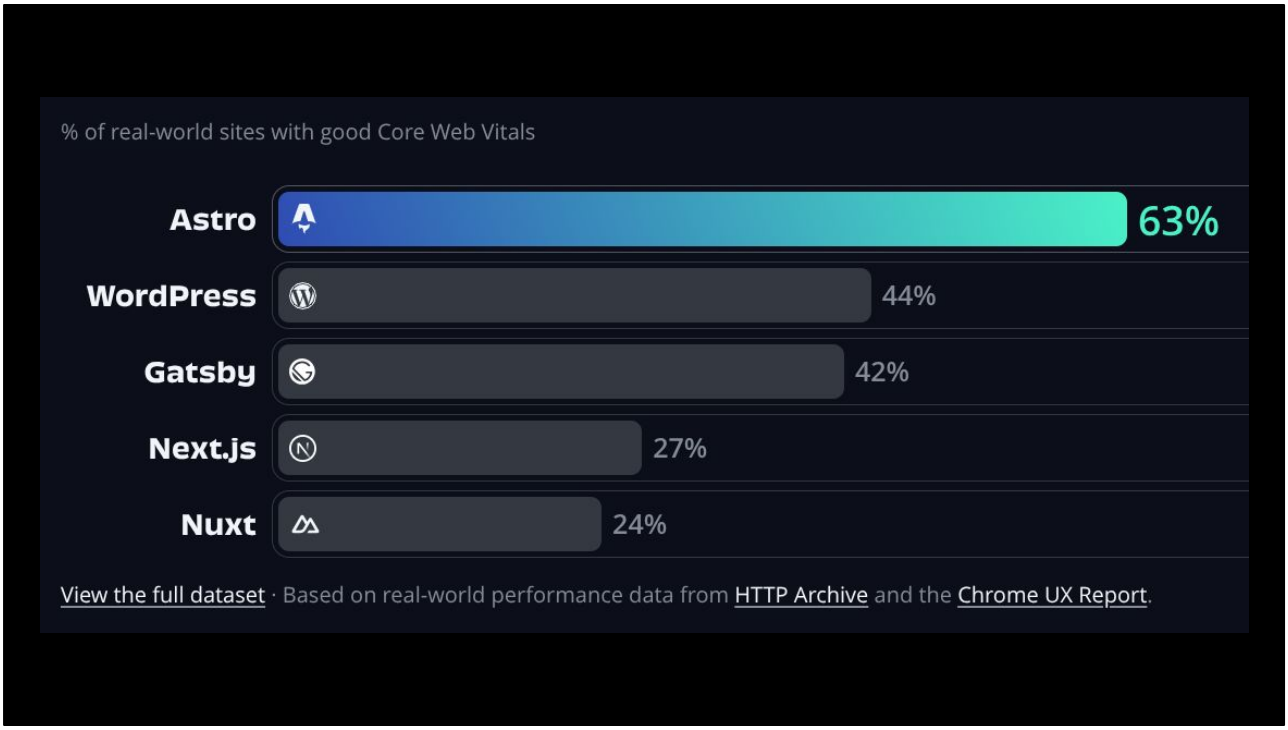
Unless your framework already handles bundling, it’s basically the defacto standard for frontend Javascript projects.

Definitely something worth getting familiar with.



Partly because of Vite, one key appeal of Astro is that it works with popular UI frameworks out of the box.

You can mix and match Astro Templates with React, Vue, Svelte, or even bring your own.



But the real Killer Feature is that Astro outputs extremely well-optimized distributions.

Astro beats the competition in *Core Web Vitals* - a set of metrics that measure real-world user experience for loading performance.

Faster page loads means better UX, means happier users, better SEO, more conversions, and is an easy to measure guiding light.

Using Astro?
It's Worth Considering Content Collections

Since Astro is designed to be SSG-first, it also solves SSG-specific problems, like data-management.

src/pages/page-1.md

```
---
title: My page of content
---
## Introduction

I can link internally to [my conclusion](#conclusion) on the same page when writing

## Conclusion

I can visit `https://example.com/page-1/#introduction` in a browser to navigate dire
```

Most SSG frameworks, including Astro, will let you use the classic pattern of

“a bunch of markdown files inside a folder.”

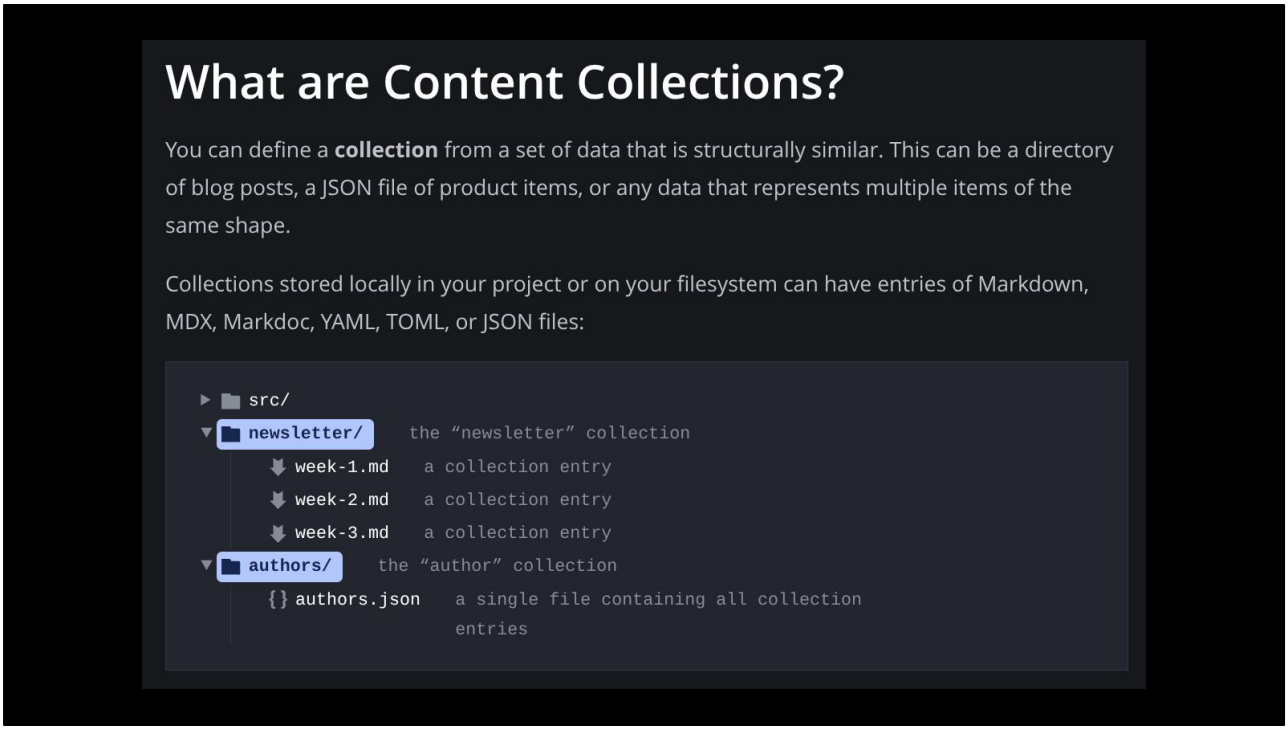
The framework will read the filesystem, ingest the markdown, apply a template, and output HTML based on the filename.

In this case, page-1.md yields page-1.html.

But what if we have more complex data types?

What if we want to join things like Events and Venues together?

What if each markdown file has an associated Image Gallery?

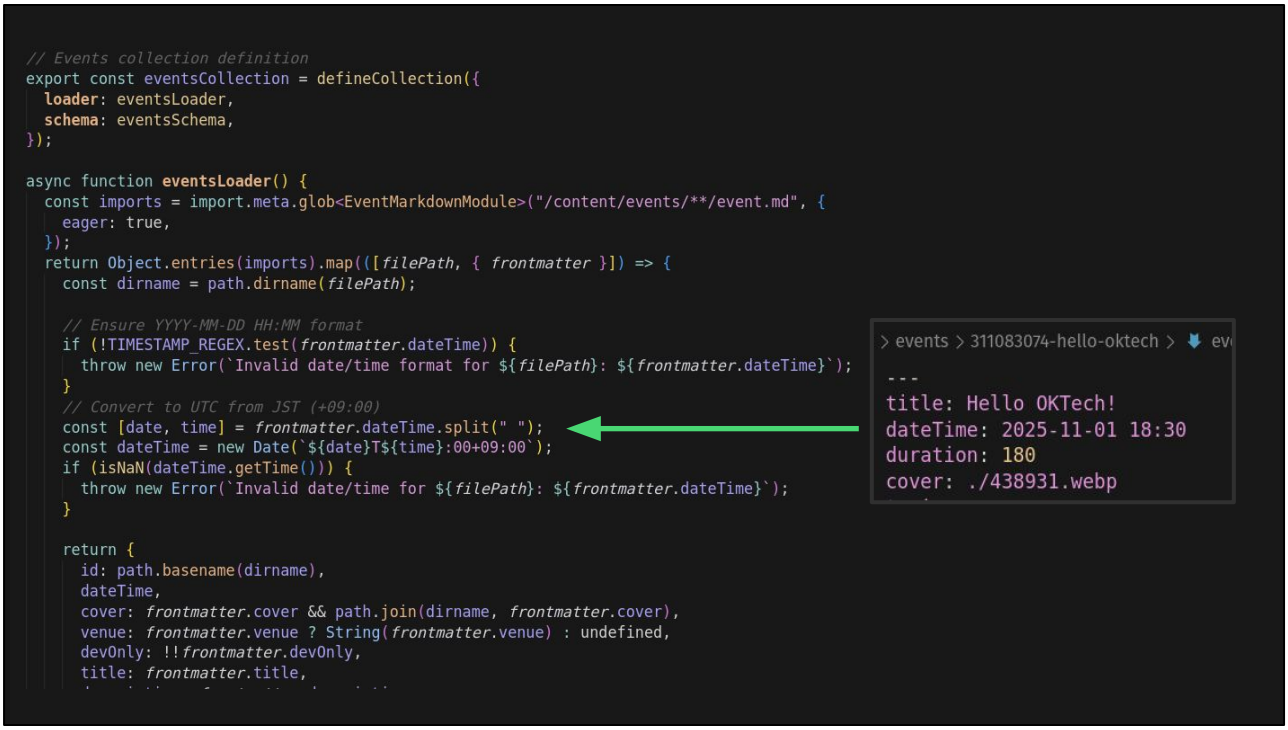


In a typical web application, a Database ORM could be be used for this, but since we’re doing SSG, we don’t have that luxury.

So Astro’s solution is a kind of in-memory ORM, called Content Collections.

The key advantage is that we can keep our data stored in a way that makes the most sense for maintainability and aesthetics.

While creating a standardized compatibility layer in code, providing total flexibility to read, transform, and extend it as we like.



With Content Collections we define arbitrary document types, with loaders and schemas.

For example, we can import markdown files with glob query.

But you aren't limited to that – it could be reading the filesystem for images, or parsing YAML, fetching from an API, or whatever.

Before any other parts of our app know about it, we can filter it and transform the data to ensure it's conforming to the structure we expect.

A perfect example, shown here, is the start time for events.

In our markdown, it's in a shorthand format and in Japan Time for easy editability.

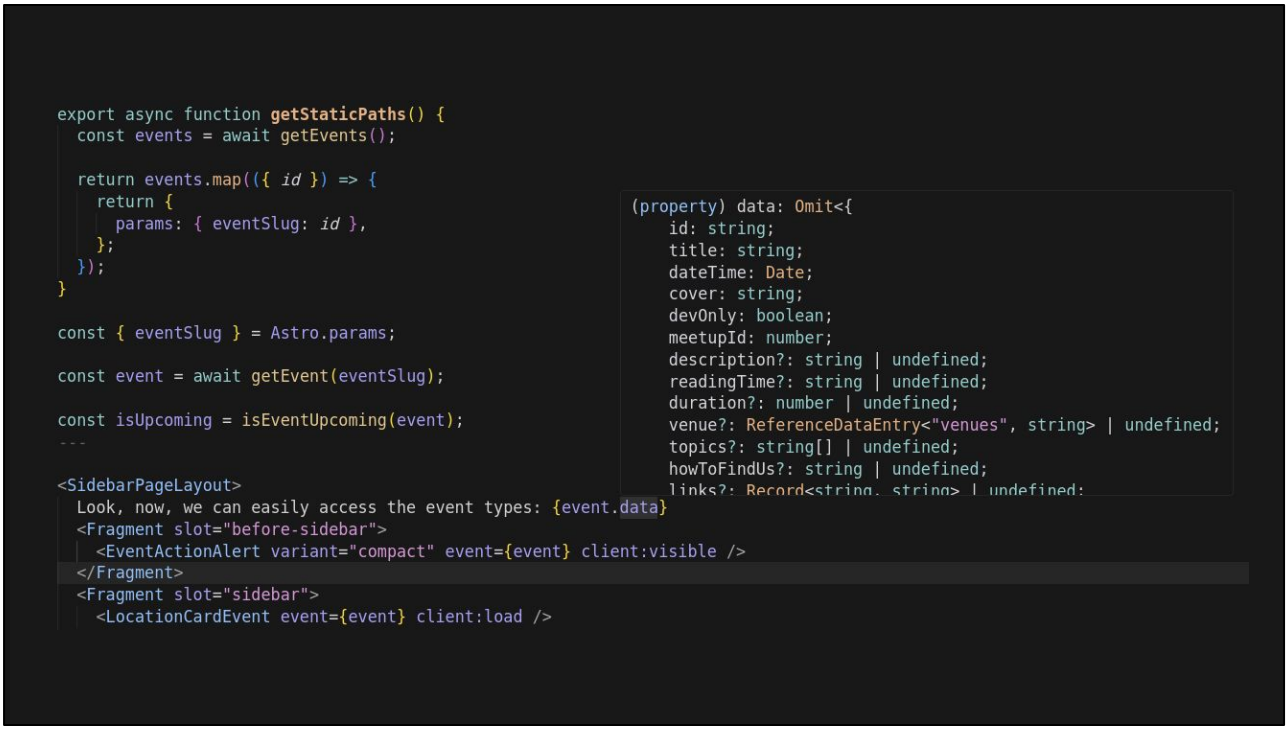
But within our app, we always want this as a UTC timestamp.

Instead of recalculating this in different parts of the app, we can just transform it once here.

```
function eventsSchema() {
  return z.object({
    id: z.string(),
    title: z.string(),
    description: z.string().optional(),
    readingTime: z.string().optional(),
    dateTime: z.date(),
    duration: z.number().optional(),
    cover: z.string(),
    devOnly: z.boolean().optional().default(false),
    venue: reference("venues").optional(),
    topics: z.array(z.string()).optional(),
    howToFindUs: z.string().optional(),
    meetupId: z.number(),
    links: z.record(z.string()).optional(),
    isCancelled: z.boolean().optional(),
    attachments: z.array(attachmentSchema).optional(),
  });
}
```

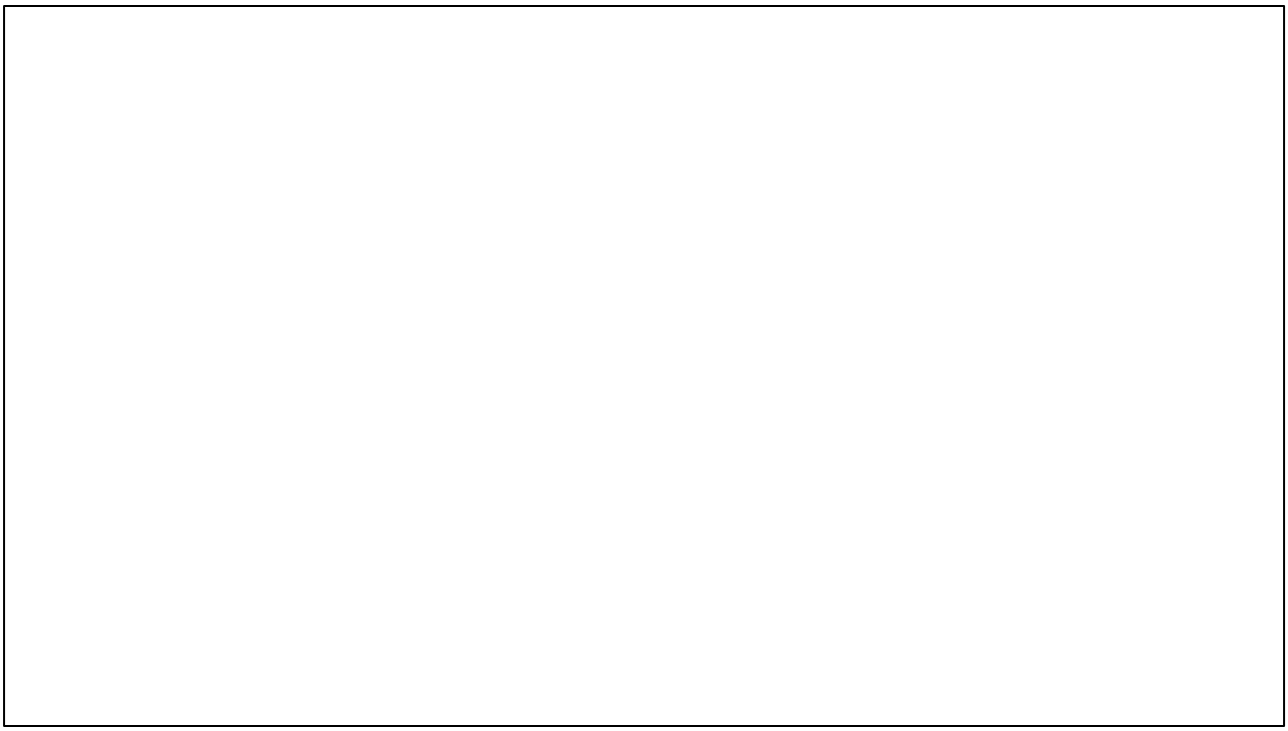
Content Collections can also enforce Schemas, which ensures build-time validation and creates Typescript types.

You can even reference collections with each other based on an ID, like a database join.



The result is that throughout your app you know exactly what kind of data you're working with.

And your build will fail with granular error messages if there's something wrong.



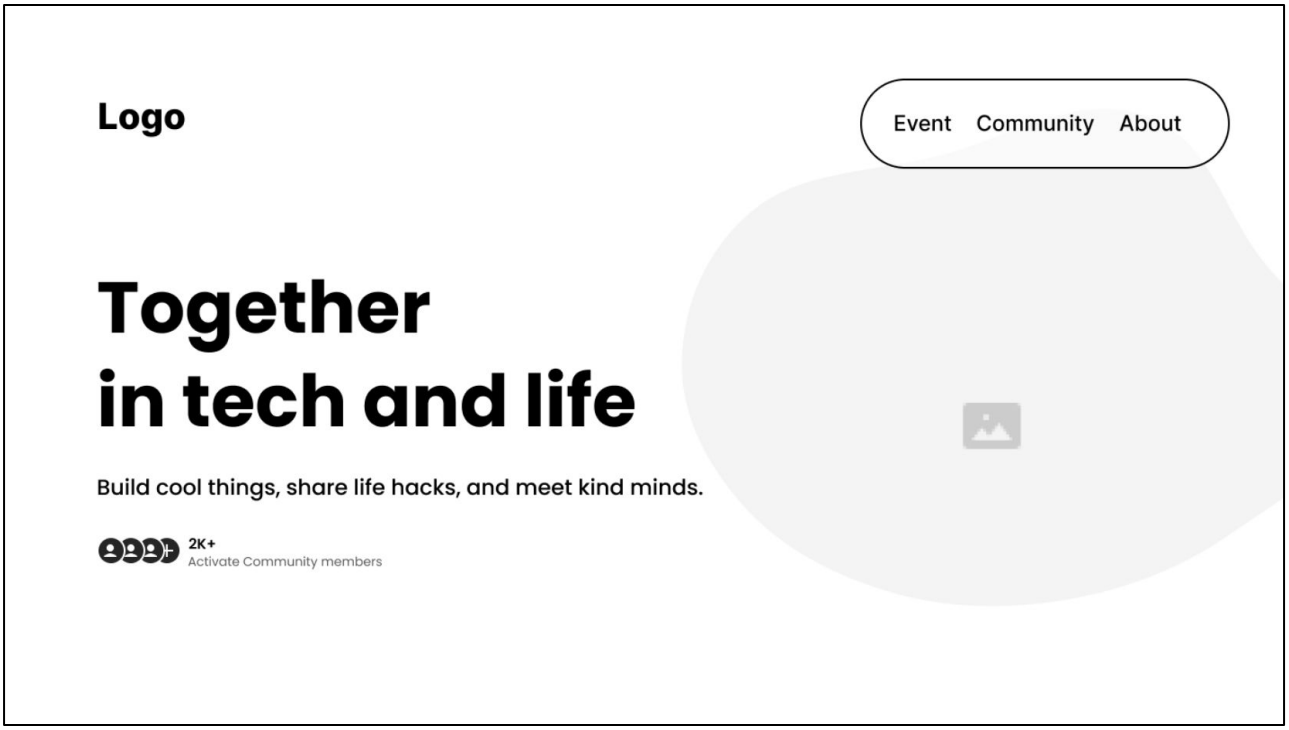
Okay, sorry. We may have gone in a bit too deep there.

I can see that many people’s eyes glaze over when I talk about Typescript.

Let’s talk about something - Lighter.

It’s Time to Talk About Blobs

It’s time to talk about blobs.



The blob graced us early in the design process.

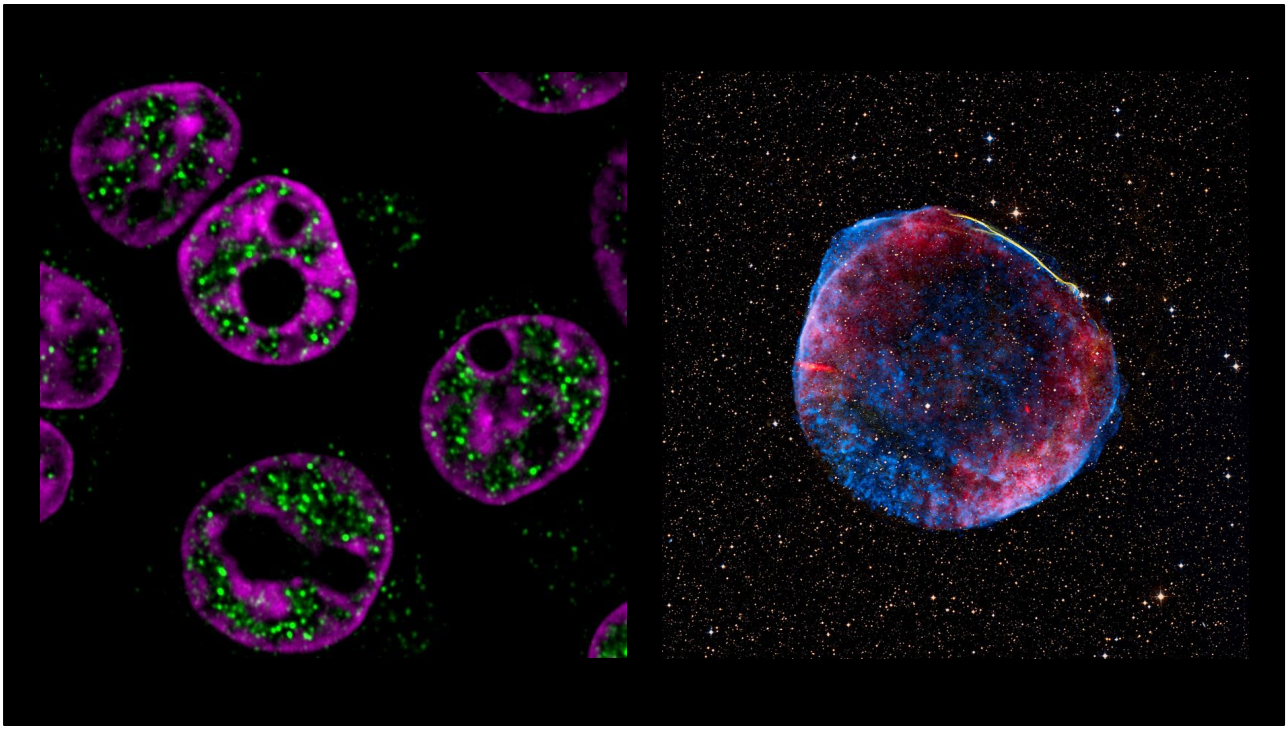
It first showed up in one of Evey's prototype mockups.

The moment I saw the Blob, I immediately recognized its beauty.

The blob is a vector, a path, a clipping mask.

The blob is unique, imperfect, human.

The blob is Together, in Tech and Life.



From the smallest cell, to the largest supernova.

Blobs are Within us and Without Us.

No Blobs. No Life.

The Blob was a memetic contagion.

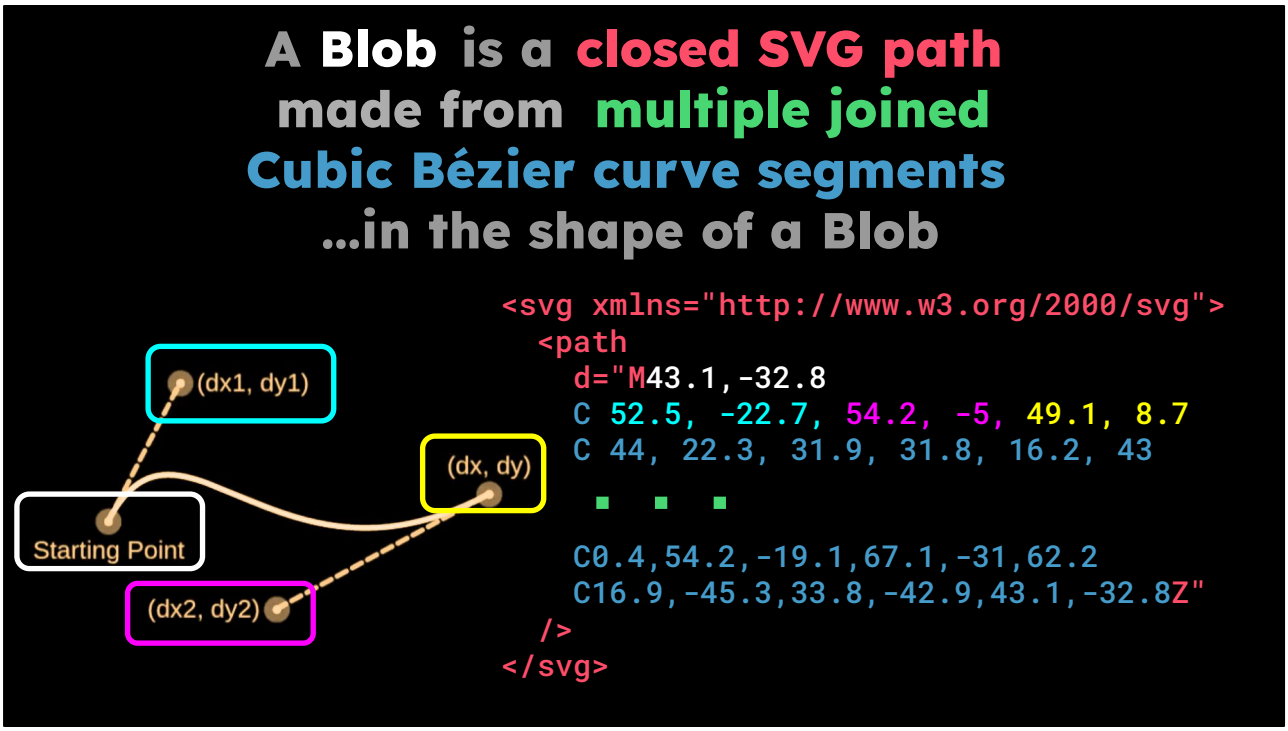
The blob evolved and adapted independently.

We didn’t choose the blob.

The blob chose us.

What is a Blob?

But wait. What, exactly, is a Blob?

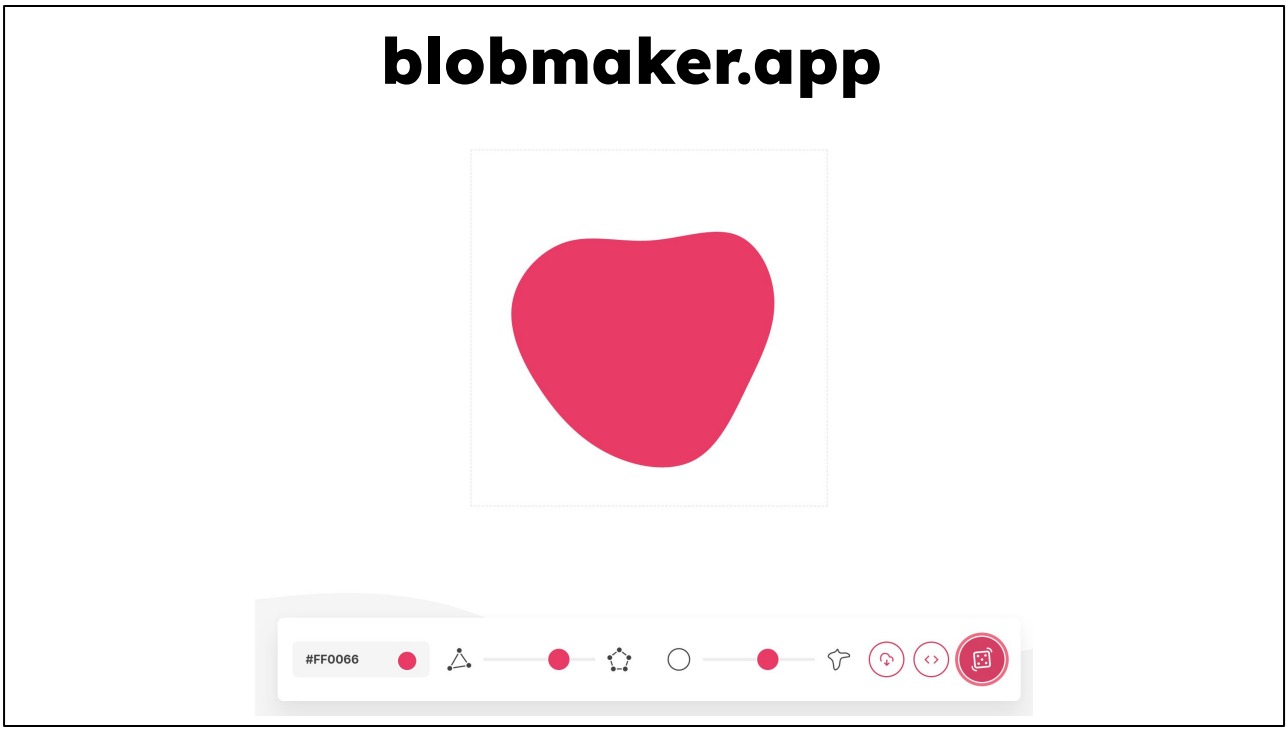


A Blob is a closed SVG path

made from multiple joined

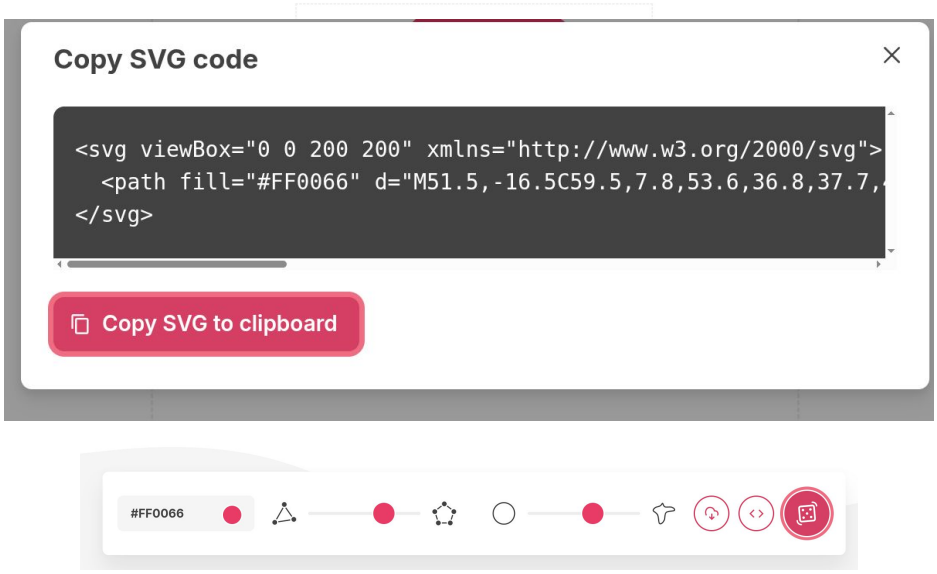
Cubic Bézier curve segments

…in the shape of a Blob



Blobmaker.app is the official blob provider of the OKTech Website.

blobmaker.app



With blobmaker.app, you are just a click away

from a practically infinite number of

beautifully blobbily

Closed joined cubic bezier curve segments.

```
// Created with https://www.blobmaker.app/, then scaled with scripts/precompute-blobs.
// you must ensure the same number of vertexes?
export const BLOBS = [
  "M87.73,11.83C96.03,20.74,97.53,36.36,93.03,48.46C88.53,60.46,77.85,68.84,63.99,78.7
  "M85.32,26.04C93.72,34.37,100.00,46.64,97.95,57.60C95.91,68.55,85.61,78.20,73.70,83.
  "M86.07,18.85C94.77,27.34,100.00,41.07,97.08,52.55C94.16,64.10,83.22,73.40,69.84,81.
  "M92.39,21.72C100.00,28.87,98.29,45.73,92.54,58.08C86.87,70.51,77.23,78.36,67.13,81.
  "M91.24,31.67C97.54,44.57,99.30,56.16,97.47,68.04C95.56,79.84,90.14,91.94,81.20,95.9
  "M89.20,22.02C94.04,33.81,89.54,47.31,86.33,61.01C83.13,74.71,81.29,88.62,73.24,94.3
  "M91.67,14.25C100.00,23.20,95.61,43.90,88.68,60.04C81.84,76.18,72.54,87.85,61.67,91.
  "M79.00,25.98C86.70,39.11,89.52,52.65,85.67,62.68C81.75,72.71,71.17,79.24,59.35,86.0
  "M86.61,20.99C94.22,33.06,94.72,48.40,89.57,59.08C84.48,69.77,73.85,75.86,61.72,83.4
];
```

And we ripped those things like nobody’s business.

```
<svg width={0} height={0} className="absolute">
  <defs>
    <mask id={maskId} maskUnits="objectBoundingBox" maskContentUnits="objectBoundingBox">
      <rect x="0" y="0" width="1" height="1" fill="black" />
      <path
        fill="white"
        transform="translate(0 0) scale(0.01)"
        d={getCurrentPath()}
        style={{
          transition: `d ${transitionSpeed}ms cubic-bezier(0.68,-0.55,0.265,1.55)`,
        }}
      />
    </mask>
  </defs>
</svg>
```

Did you know that you can even natively animate between Blob paths with pure CSS transitions?

Well, I didn’t, until ChatGPT told me I could. And it worked!

But, unfortunately, once again, it didn’t work in IE6.

And we couldn’t deprive such an important user base from realistic blob transitions.

Seriously, though, Safari did force us to find the much better alternative:

The react-spring animation library.

Now our blobs not only animated on all modern platforms, but had more realistic blobbiness to them.

Here’s the magic – an hidden inline SVG with a shared mask id that animates a neighbouring div’s clipping path.

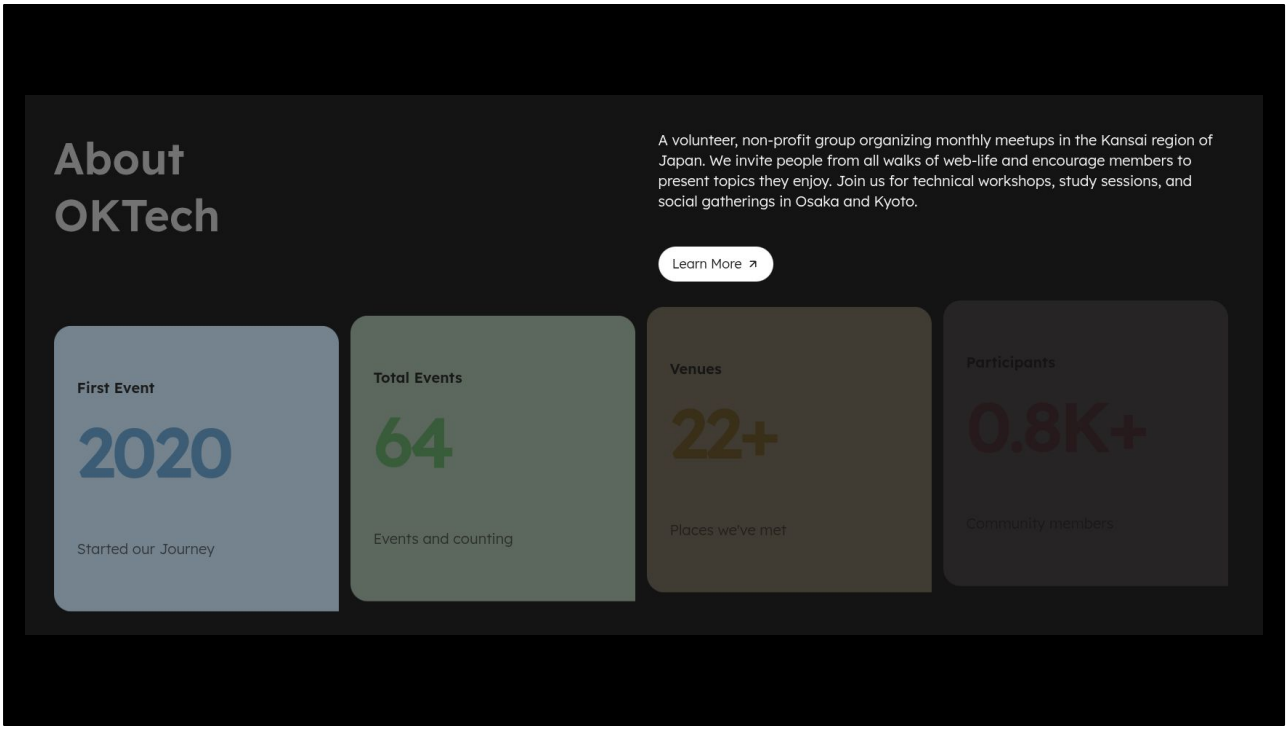
Animation is Low Hanging Fruit

And this is a learning I would like the share.

Developers don't really normally think about animations too much.

But nowadays, with modern animation libraries, they are very LOW COST and HIGH VALUE.

It's super easy to add extra PANASH, for very little programming.

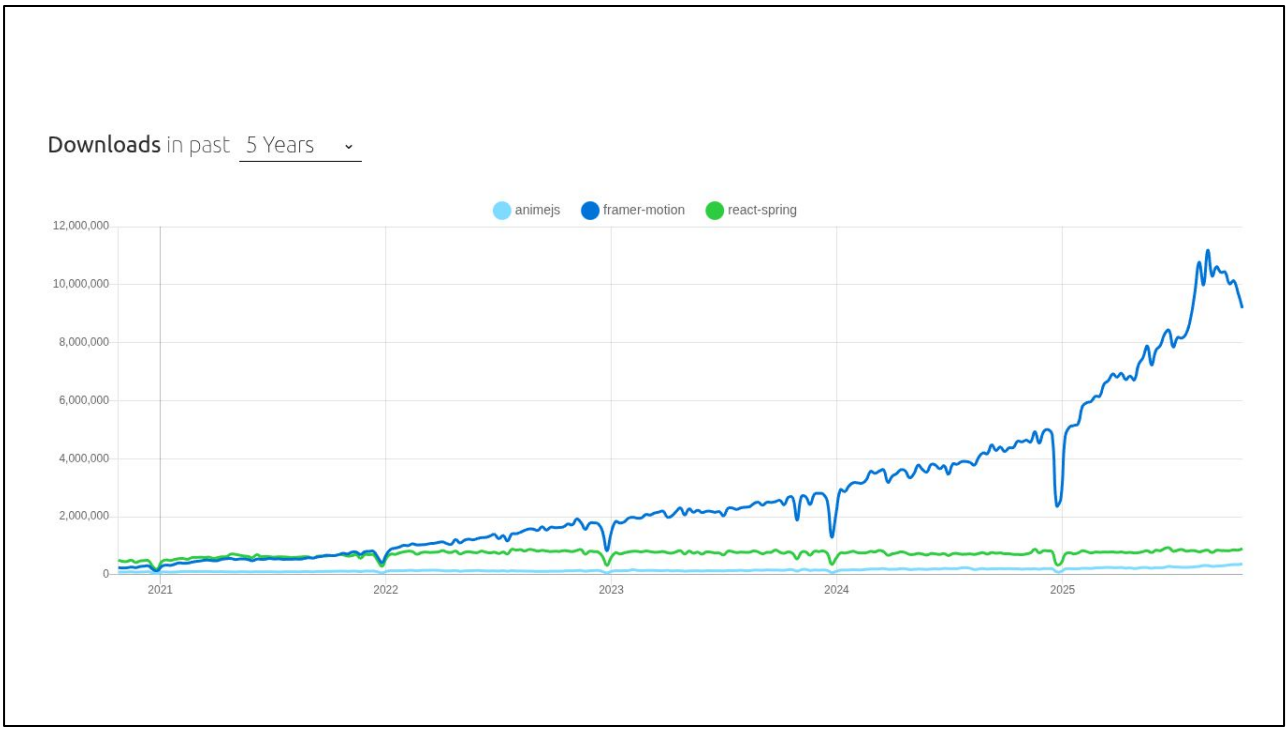


Take for example this grid. We could just keep them as static squares.

```tsx
export default function StatsGrid({ stats }: StatsGridProps) {
  const trail = useTrail(stats.length, {
    from: { opacity: 0, transform: "translate3d(0,-80px,0)" },
    to: { opacity: 1, transform: "translate3d(0,0px,0)" },
    config: { mass: 2, tension: 60, friction: 30 },
    delay: 300,
  });

  return (
    <div className="grid grid-cols-2 gap-4 md:grid-cols-4">
      {trail.map((style, index) => (
        <animated.div key={index} style={style} className="aspect-square">
          <StatsItem stat={stats[index]} colorClass={tileColors[index % tileColors.length]} />
        </animated.div>
      ))}
    </div>
  );
}
```

But with a just **few lines of code**, this otherwise boring component is *transformed* into an engaging bonanza of kodawari.



I used react-spring on OKTech.jp, but that maybe wasn't the right choice.

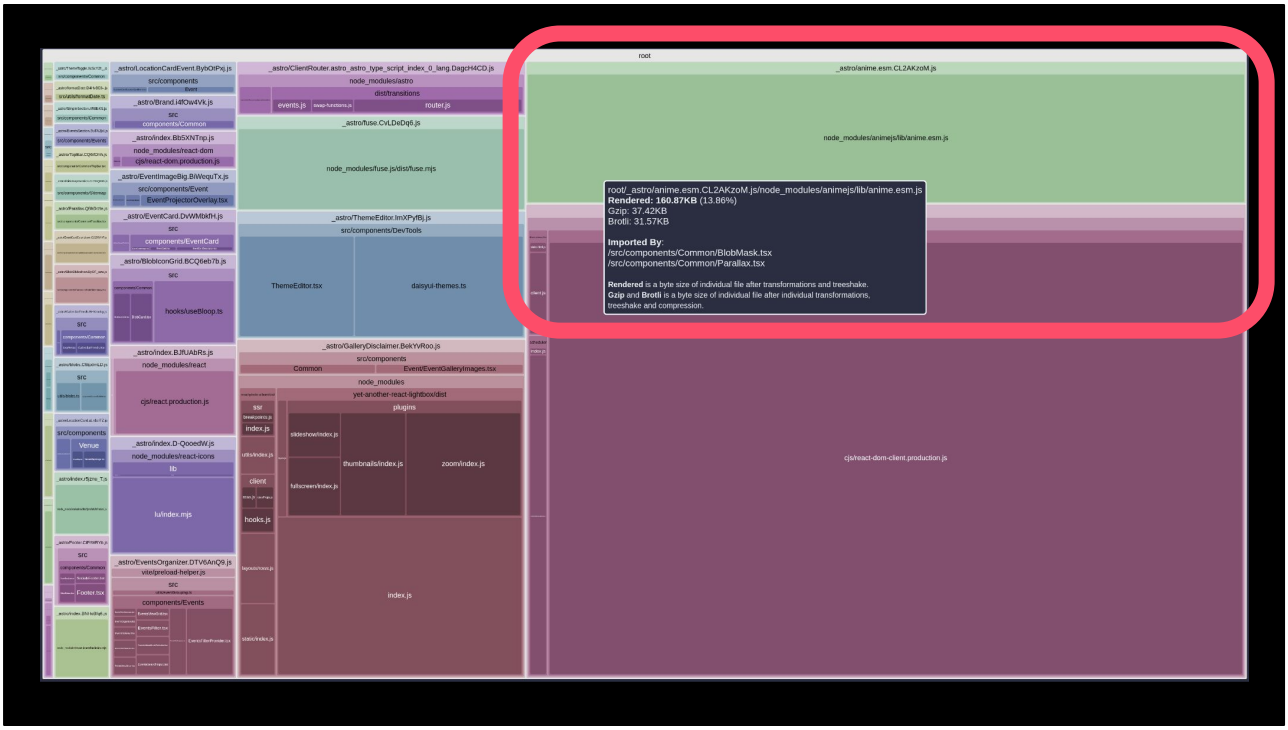
Don't get me wrong, I'm happy with it. But my rule of thumb lean towards whatever is popular.

So next time, I'll give framer-motion a try.

I actually first tried out Anime.js, largely because it sounded cool, but I quickly removed it for the following reason:

Auditing and Optimizing

Auditing and Optimizing



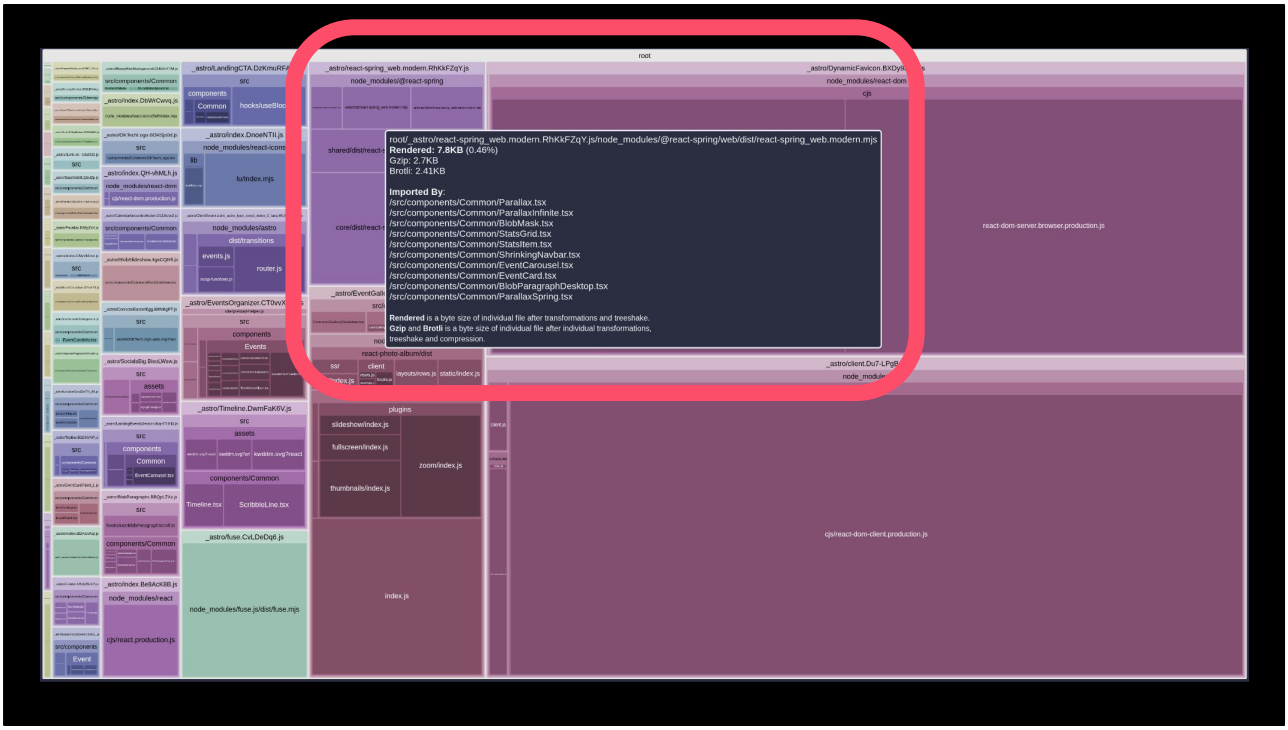
If you, like me, love to analyze bundles. You can run `npm run analyze-bundle` on this project.

You'll see this lovely visualization of which files in your distribution are heavy and light.

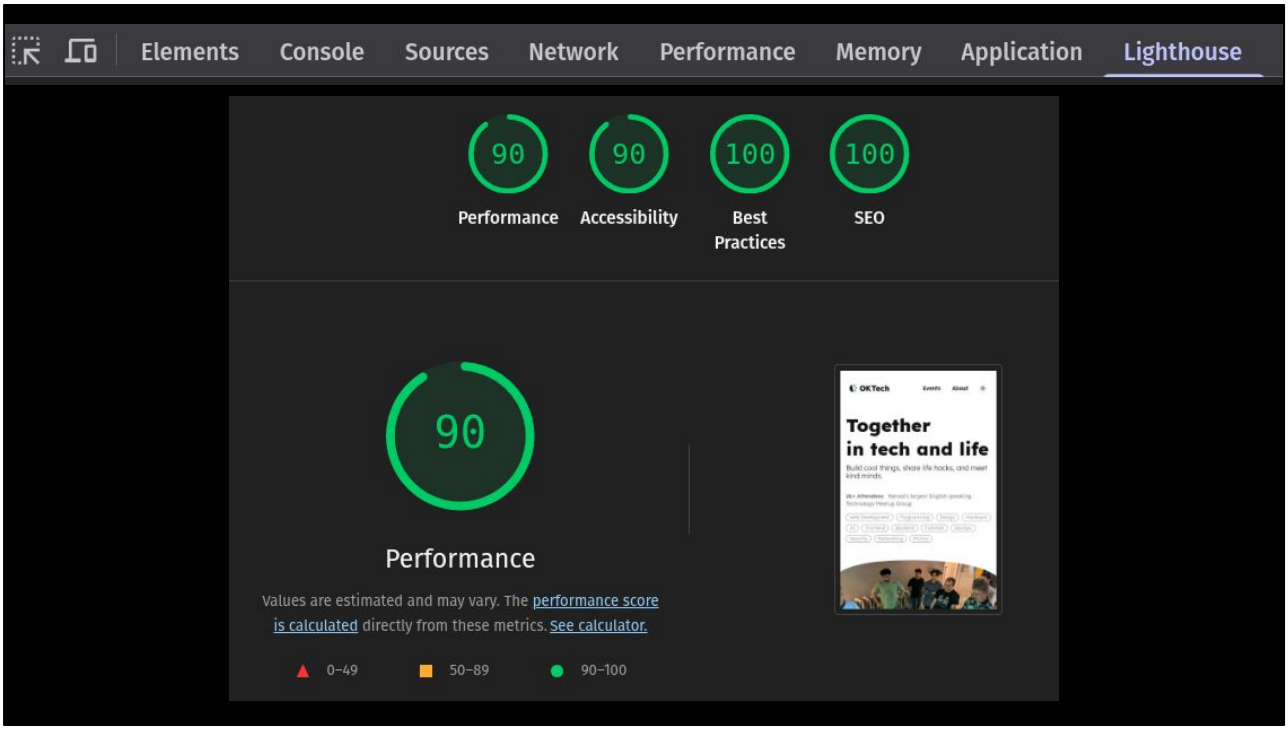
Much to my dismay, when I first analyzed the bundle, too much of it of it was taken up by Anime js.

Turns out it didn't support modern tree-shaking!

So, I switch to the lighter-weight react-spring.



Ah, Much better!



Another essential auditing tool is Lighthouse.

Built into Chrome Developer Tools, it will generate a report and check the **Core Web Vitals** we talked about before.

It's always worth using this tool, as it's quick and easy and can help troubleshoot performance issues.

All green, and you're good to go.

But this might require some tedious code changes, particularly to reduce the amount of data needed to load your site.

Save The Planet
Use Image Source Sets

One thing that your Lighthouse report will recommend is the use of Image Source Sets

As we all know, the internet is a series of tubes.

Some tubes are small, some tubes are big.

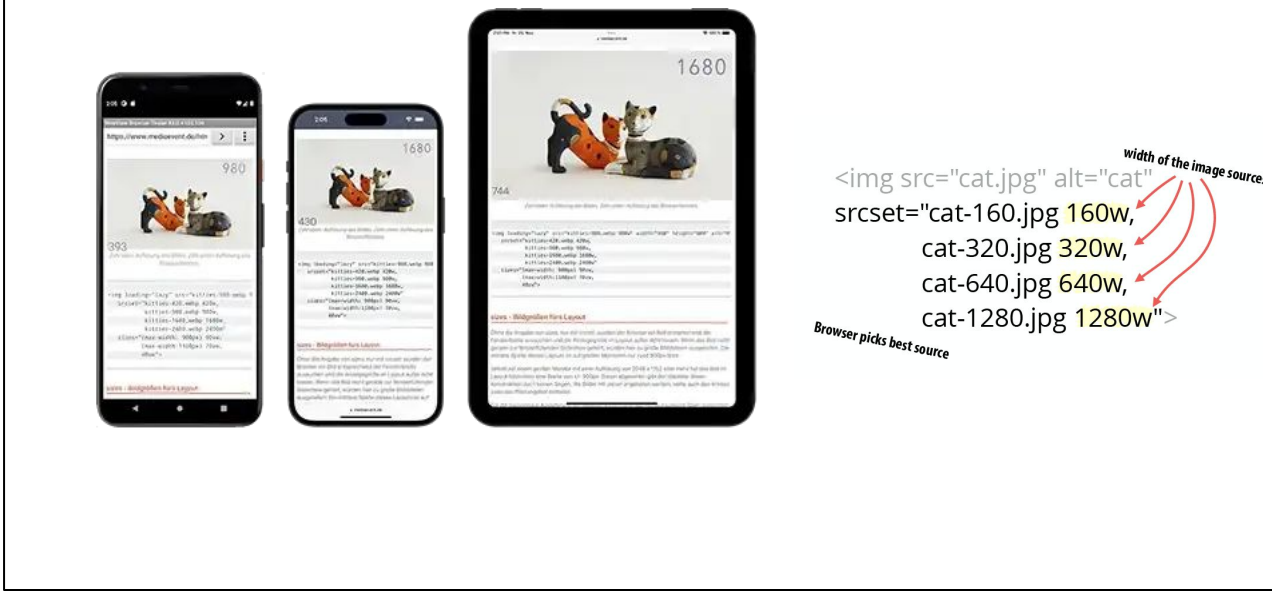
And the primary purpose of these tubes sending pictures of cats.

But how do we ensure that the tube is big enough for the cat?

How do we make sure that we only send a small cat to the small tube?

And a big cat through the big tube?

The industry terminology for this concept CTR – or Cat Tube Ratio, which you want to be as close to 1 as possible.

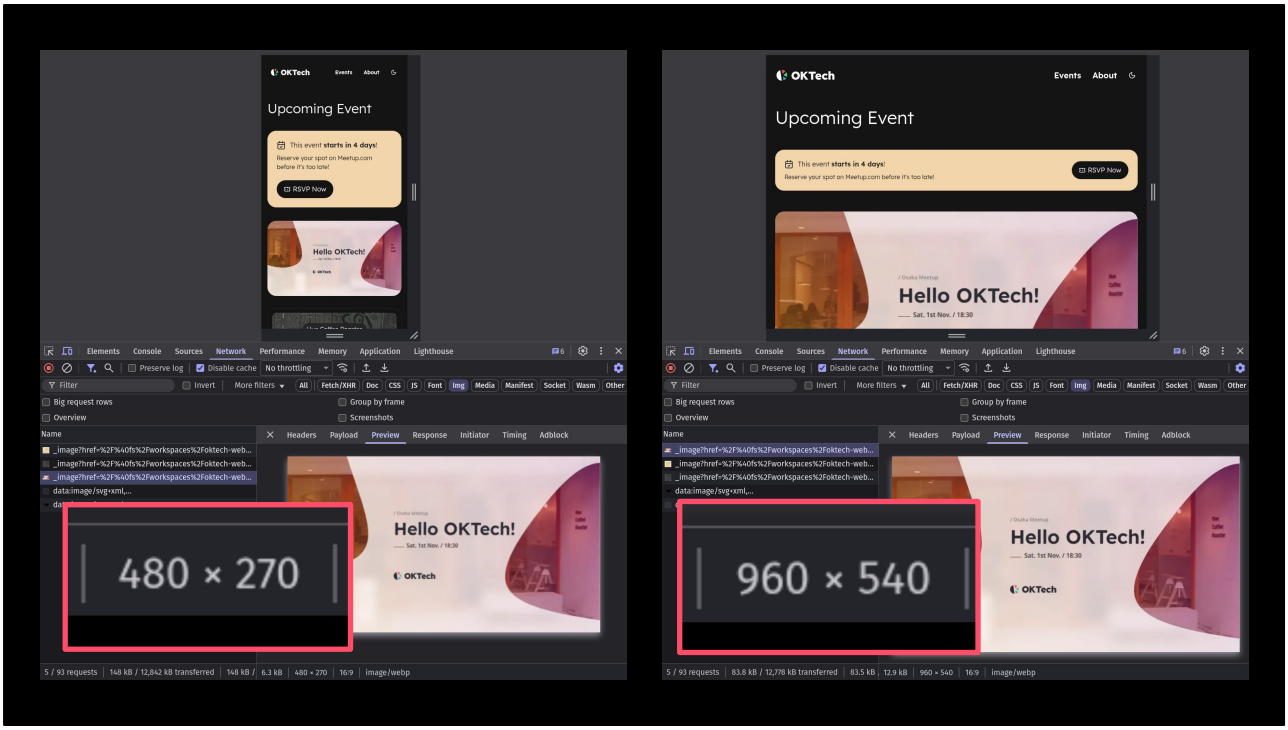


To optimize your CTR, this is where a Source Set can help.

An underutilized feature of the HTML spec,

Basically, since the browser knows how wide it’s viewport is, it can intelligently request the best image for that size.

Instead of trying to squeeze big cats through a small pipe, the cat we send is exactly the size that’s needed!



Here it is in action on oktech.jp

Especially with image-heavy pages, it can seriously reduce data transfer.

It saves your visitors valuable data.

Things feel faster - it's better UX, especially on mobile.

And in turn, it will help out with Search Engine Optimization.

But it does require some extra steps – including preparing the images in multiple dimensions before we send them.

Thankfully, Astro does this for us.

src/components/MyComponent.astro

```
---
import { Image } from 'astro:assets';
import myImage from '../assets/my_image.png';
---
<Image src={myImage} alt="A description of my image." layout='constrained' width={80
```

This `<Image />` component will generate the following HTML output on a prerendered page:

```
<img
  src="/_astro/my_image.hash3.webp"
  srcset="/_astro/my_image.hash1.webp 640w,
      /_astro/my_image.hash2.webp 750w,
      /_astro/my_image.hash3.webp 800w,
      /_astro/my_image.hash4.webp 828w,
      /_astro/my_image.hash5.webp 1080w,
      /_astro/my_image.hash6.webp 1280w,
      /_astro/my_image.hash7.webp 1600w"
  alt="A description of my image"
  sizes="(min-width: 800px) 800px, 100vw"
```

When using the Image component, Astro will prepare your various sized cats automagically.

Unfortunately, however, this only works with .astro components, and we're using React.

Beware of Astro's Idiosyncrasies

And this leads us to my main gripe with Astro.

While it does support multiple frontend frameworks as advertised,

They're kind of a second class citizens in the Astro world.

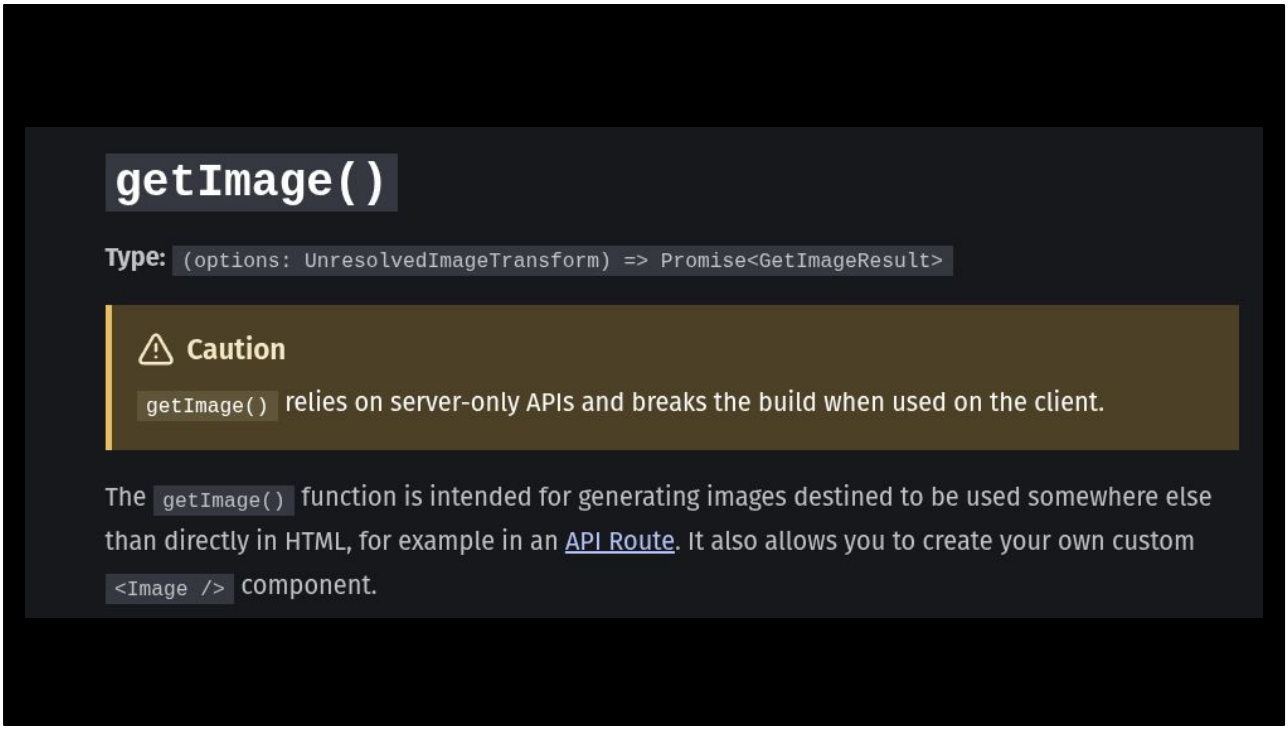
Unlike in Next.js, where React is deeply integrated and can be interwoven between client and server,

In Astro, while you can render React Serverside and ship the HTML,

For anything interactive, you have to use these weird Client Directives to tell Astro when we want to opt-in.

The problem is that once you opt-in to a client component, that entire component tree also becomes a client component.

These child components can't use Astro's server-side features, including the magic Image component.



The workaround is to implement your own image generation method with `getImage`,

You can either include the references in your Content Collection,

prop-drill them down the component tree,

or write a custom component wrapper.

BlobParagraphs.astro
BlobParagraphs.tsx

```astro
---
import BlobParagraphsClient from "./BlobParagraphs";
import { getResponsiveImage } from "@/utils/responsiveImage";

interface Paragraph {
  title: string;
  text: string;
  images: string[];
  blobs?: number[];
}

interface Props {
  paragraphs: Paragraph[];
  blobs?: number[];
}

const { paragraphs, blobs } = Astro.props;

const processedParagraphs = await Promise.all(
  paragraphs.map(async (paragraph) => ({
    ...paragraph,
    images: await Promise.all(
      paragraph.images.map((path) => getResponsiveImage(path, "blobSlideshow")),
    ),
  })),
);
---

<BlobParagraphsClient paragraphs={processedParagraphs} blobs={blobs} client:visible />
```

If you have an interactive component that that also uses responsive images,

You end up having to do something like this,

Where we have a Server Side Astro wrapper, providing data and generating images, passed to a Client Side React component.

It’s fine, I guess, but I hope Astro can improve this experience in future, and maybe embrace React’s native RSC model.

But apart from this, Astro is pretty good.

It’s worth giving it a try.

What's Next, OKTech.jp?

So that concludes my bag of tricks. I hope you've learned something useful.

And I just wanted to end with a call to action.

There are some features in the future we'd like to implement, such as

Speaker Profiles, A Map Visualization, OG Image Generation,

and even some interactivity during events with Q&A, Voting, etc.

If you have any other ideas, feedback or suggestions,

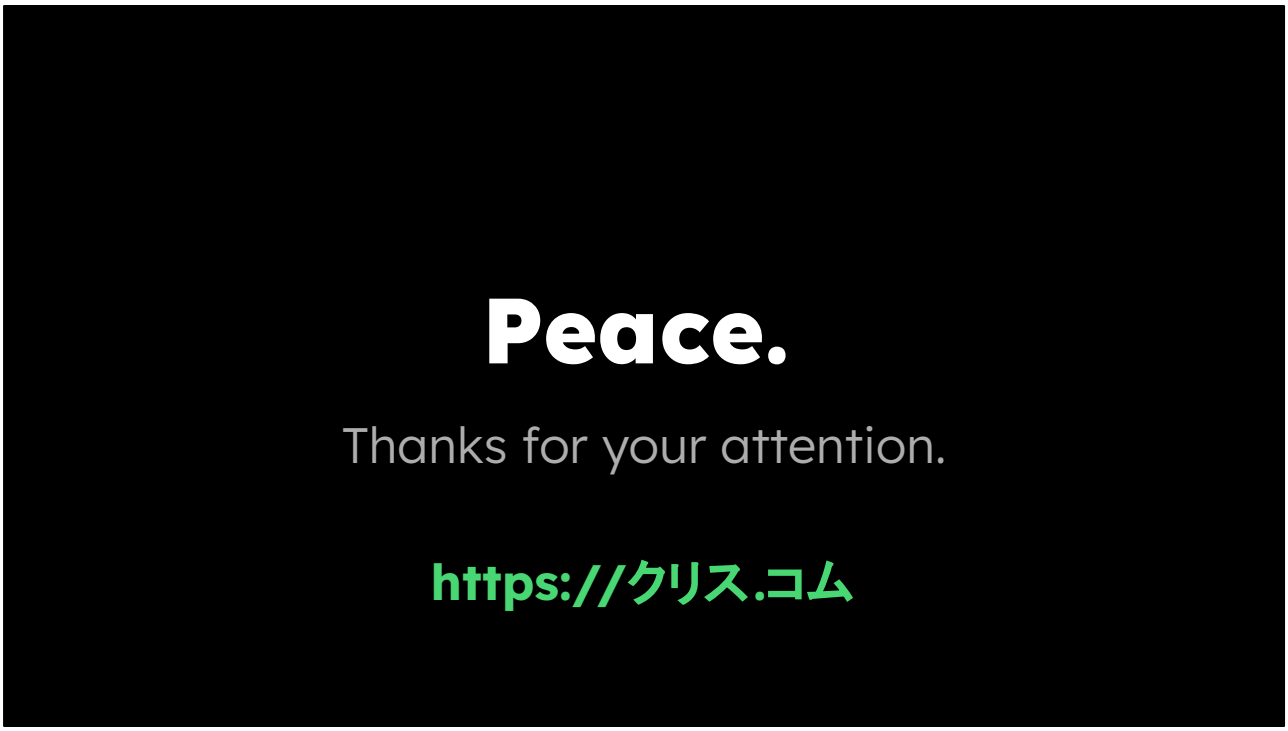
or if you'd like to learn by contributing code to this modern stack,

It would be my pleasure help you and collaborate.

Just reach out to me on the OKTech discord.

Or even better, chat with me now.

Please take a one of my limited edition business cards!



Thank you for your attention!

PEACE.