[Reset timer] Hey everyone! I'm Blake. I'm super excited to be sharing in some tech nerdery with you guys. I want to share a few things up front quickly.

blakewatson.com/magnolia2025

If you want to follow along or if you want to reference this talk later, there is a full text version available at this URL. This will have links to things I mentioned, some CodePen demos, and stuff like that.

Frontend Engineer

MRI Technologies

A little about me. I work remotely as a Frontend Engineer at MRI Technologies in Houston, TX. We build a web-based spacesuit hardware management system called COSMIC for teams at NASA.

Aside from work. I'm a side-project enthusiast, which is kind of what my last Magnolia talk was about. I have a disability called spinal muscular atrophy so I use a power wheelchair and a lot of assistive tech. I enjoy playing Dungeons & Dragons. I enjoy blogging and creative writing. And I have a bit of a font-hoarding problem.

Community Sponsor
htmlforpeople.com

I'm proud to support MagnoliaConf as a sponsor this year. I know it's a lot of work to put this conference on and I really appreciate the work of Kayla and Richard and everyone involved. I'm using my sponsorship to promote this totally free online book I wrote last year called HTML for People. HTML isn't only for people working in the tech field. *Anyone* should be able to make a website with HTML if they want. Pass it along to your non-coder friends. It's super beginner friendly. That's htmlforpeople.com.

Frontend build tools

Ok yall. I have a confession to make. I'm fed up. I'm absolutely sick of frontend build tools. I'm talking about Webpack, Vite, Rollup, ESBuild—anything that processes your frontend code into a bundled version that can be read by the browser. It seems like every project you look at shoves an npm installation in your face. I'm not saying there aren't things to like about them. But they add complexity and some damn annoying technical debt in the long term.

Thing is, browsers have been getting better. They have native support for features that previously *required* build tools. Now's the time to jump ship and leverage the power of the browser directly.

To that end, we'll go on a whirlwind tour of the features and techniques available in today's browsers. But first, we should consider what build tools do for us.

Reasons for using build tools

Languages and framework support

TypeScript

Sass

JSX

`.svelte`, `.vue`, etc.

A huge reason for turning to build tools is to support languages and framework file formats that the browser doesn't recognize. The biggest examples are:

TypeScript for type-aware JavaScript
Sass for advanced stylesheets
JSX for expressive templates in React
Framework specific file formats like Vue and Svelte.
Tailwind for radically atomic CSS
And many more...

Reasons for using build tools

Easily use third-party code

```
npm install everything@latest --yolo
```

Build tools also make it easy to install and use third-party libraries and packages. You `npm install` them, import them, and then you are good to go. The build tool takes care of dependency management, module resolution, bundling, minification, and other optimizations.

Reasons for using build tools

Optimizations

A build tool typically bundles all of your code into one file for the browser to read and minifies it to reduce its size.

It eliminates parts of your codebase that aren't being used.

It can do things like chunking to break up your JavaScript bundle and serve code in a just-in-time manner.

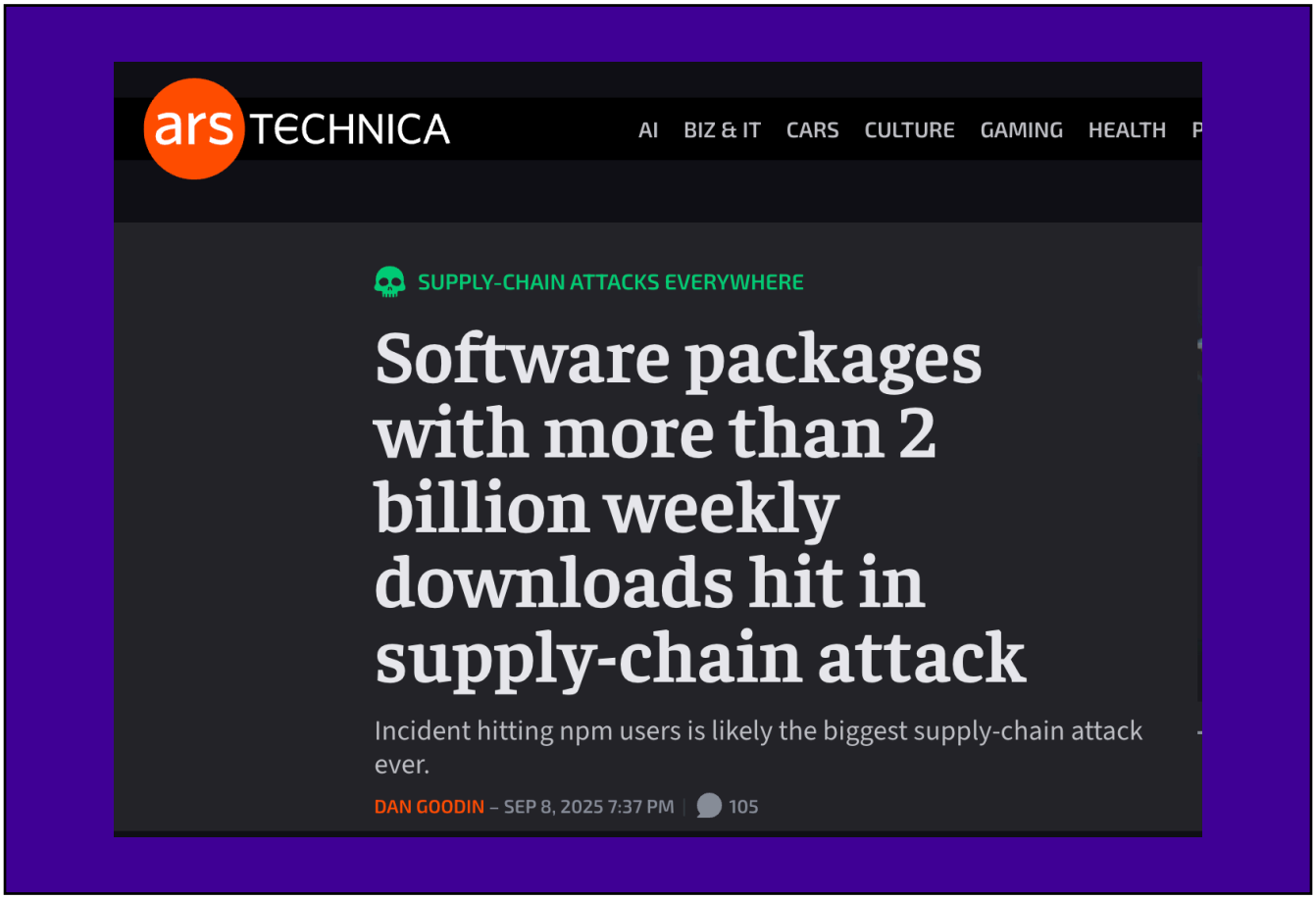
And they can process other assets like images to automatically optimize them for the web.

It sounds like I'm doing a marketing pitch *for* build tools. I'm just trying to be fair okay? Don't say I didn't give build tools a fair shake, okay? But they add so much complexity.

Let me tell you a story. It goes a little something like this. You start a new project. You npm install a thing. Then you realize you'd like another thing, so you npm install that. That thing has you npm install a few other things. And now you're really cooking with grease. You start building out your project. Everything is going great. You ship the project and pat yourself on the back.

A year later, you get a request to make a small, easy change to the project. So you open it up. You run your build. And the build fails. You poke around and realize that some package or other needs to be updated. But that package can't be updated because it relies on another package that needs to be updated. And that package can't be updated without making more changes to your codebase. Before you know it, you're looking at hours of work trying to get your build process up again, all so you can make a change that should have taken five minutes.

By the way, are the thousands of packages I just stuffed into this project like... safe?



Nope. We're having to put too much trust into too many authors and packages and infrastructure. And with every package we install, our attack surface becomes a larger.

Browsers are good actually

Embrace runtime

Build tools and bundlers have been around for a while now because they offered a developer experience that we weren't getting when we were using the browser directly. But with some of the features the browsers have gained, we can still have nice things without the build step.

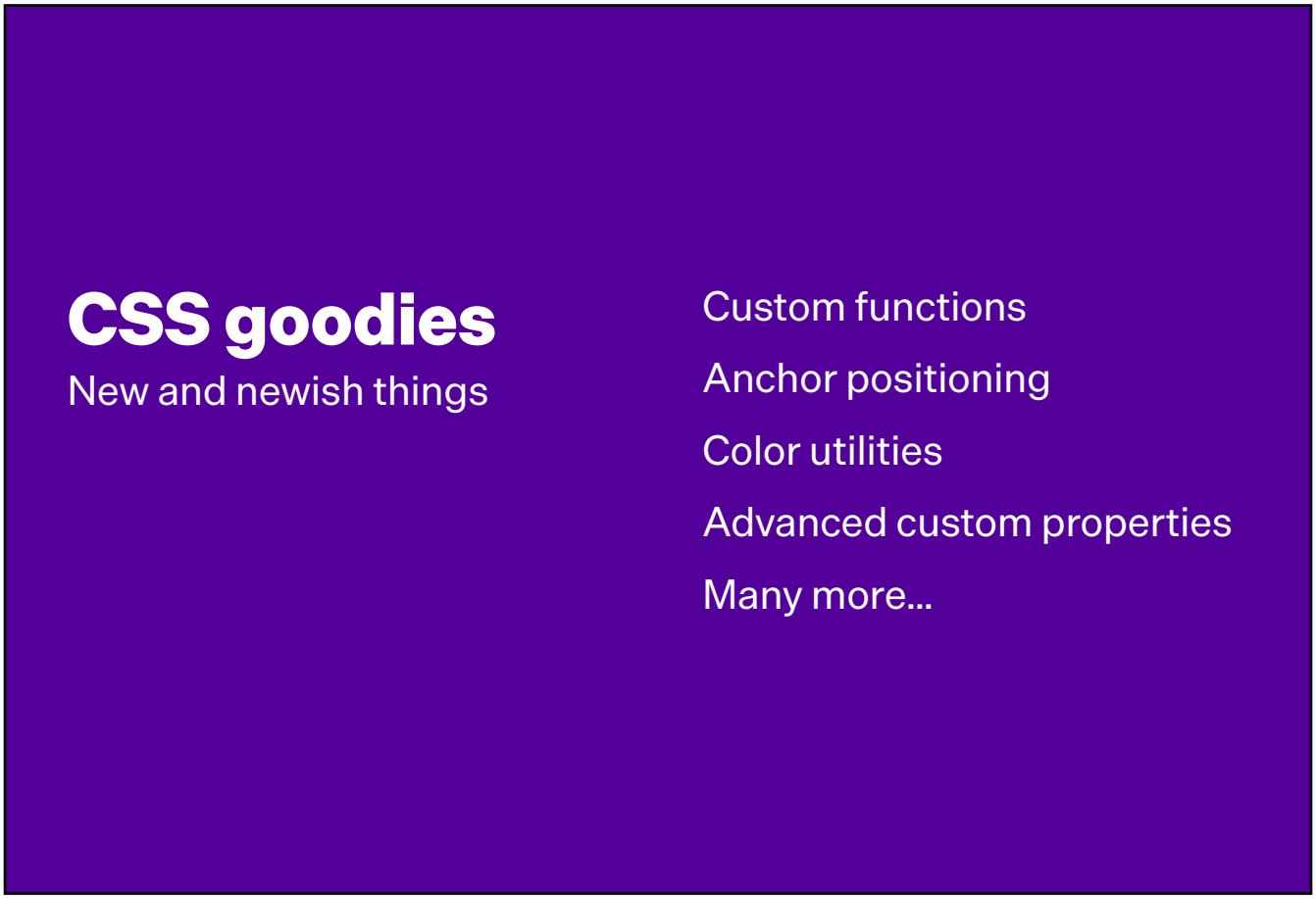
So no more compiling. Let's write the code the browser is going to run. Let's embrace runtime. We're starting our tour off with styling.

I'd say a lot of projects gained a build step because the developers wanted to use Sass and probably because they wanted to use variables and they wanted to use nesting.

Well, the days of needing Sass for them are over.

You've probably seen CSS custom properties (variables) by now. They begin with two dashes and you can access them using the `var` function. These are actually way better than Sass variables because they can be changed at runtime. And not only by CSS code, but also by JavaScript! This combo gives you endless possibilities to change your design based on user actions.

CSS nesting is Baseline 2023. Later in 2026 it will be considered widely available, meaning you've got browser support going back for about two and a half years. We're basically there.



And there are all sorts of goodies coming in CSS. I'm serious, I need to take a course in CSS again. If you're like me and it's been a minute since you brushed up on your CSS knowledge, then you're probably missing out on a ton of really cool stuff. But yes, some new and newish things are:

Custom functions
Anchor positioning
Color utilities
Advanced custom properties
Many more...

You might not need JavaScript

So we've seen some modern CSS, but how about JavaScript? Do you even really need it? Okay, probably, but let's take a look at some HTML components that you previously may have used JavaScript to build. A lot of this is inspired by a blog post by Lyra Rebane and I've linked it in the slide notes.

Reveal and collapse with `<details>` and `<summary>`

```
<details>
  <summary>
    Why did Han Solo refuse the
steak dinner?
  </summary>

  <p>It was Chewy.</p>
</details>
```

A common need in web pages is to click an element to reveal some content. Previously, this required a bit of fancy CSS or, commonly, a bit of JavaScript. HTML can do this very easily with the built-in `<details>` and `<summary>` elements. These have been around for a while now, so you may have heard of them, but if not, here's a quick example of how they work.

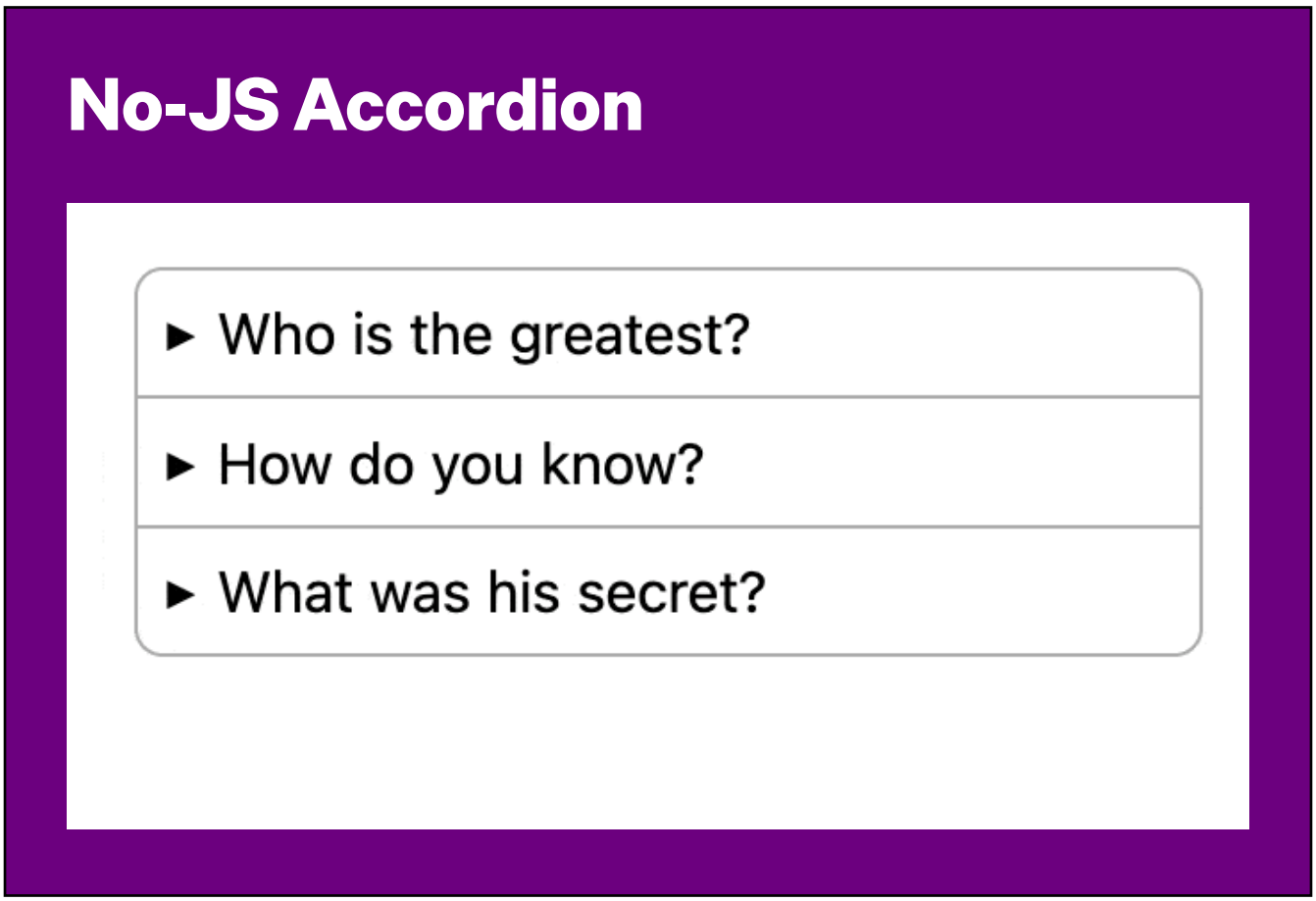
That gives us something like this.

```html
<div class="accordion">
  <details name="ali">
    <summary>Who is the greatest?</summary>
    <p>Muhammad Ali</p>
  </details>

  <details name="ali">
    <summary>How do you know?</summary>
    <p>Because he said it over and over.</p>
  </details>

  <details name="ali">
    <summary>What was his secret?</summary>
    <p>Float like a butterfly, sting like a bee.</p>
  </details>
</div>
```

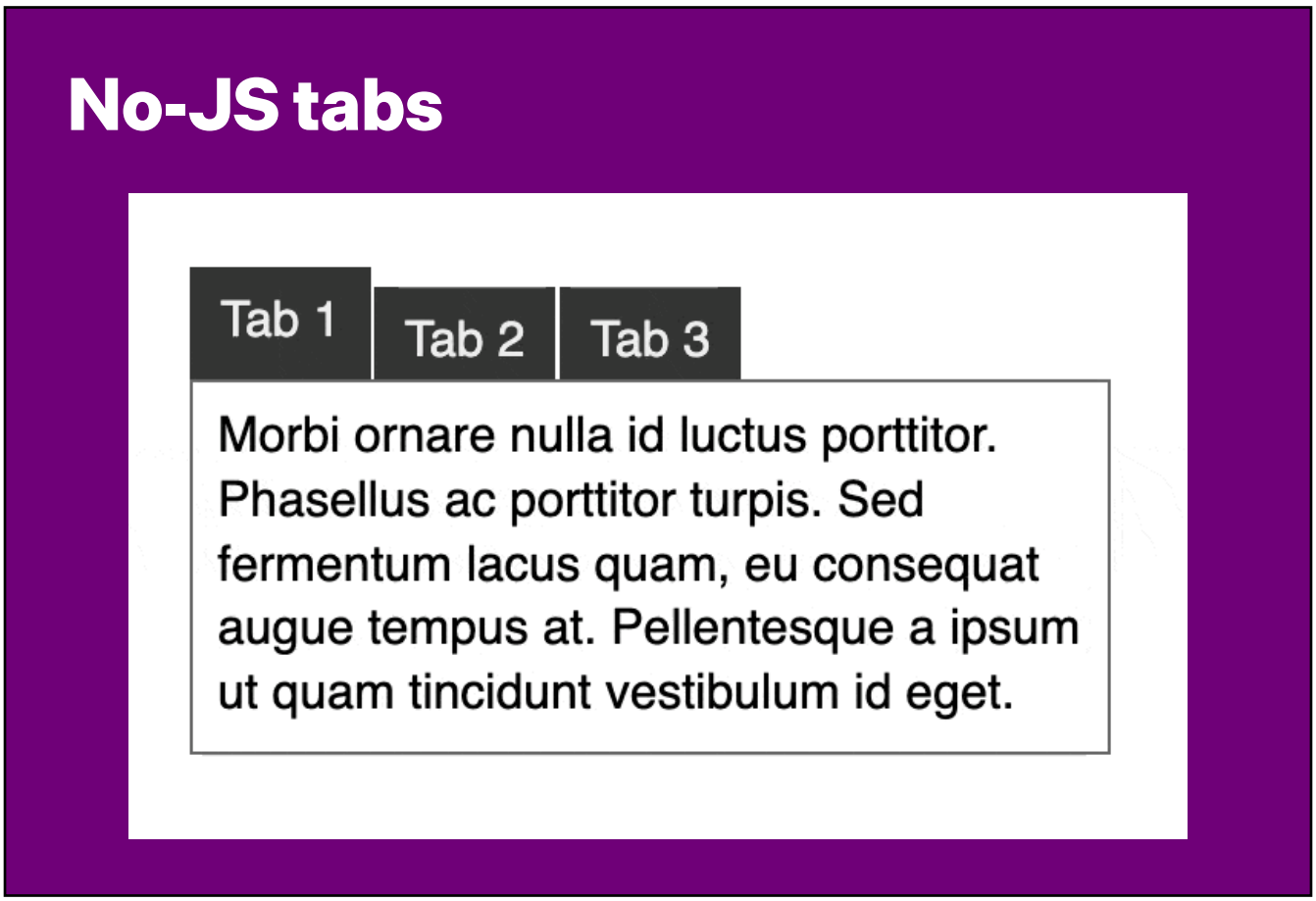
We can take this a bit further and combine multiple `<details>` elements to create an accordion-style control. By giving each element a `name` attribute, the group becomes mutually exclusive—where opening one element closes the others.

```
.accordion {
  ...

  details {
    ...

    /* styles that target the open state of the element */
    &[open] {
      ...

      /* animate everything inside the <details> element except for the
<summary> */
      & > *:not(summary) {
        animation: fade-in 0.3s ease-out forwards;
      }
    }
  }
}
```

I stripped out some of this styling to make it readable on the slide, but these are the important bits of it. You can find the full code in a CodePen in the slide notes, but I wanted to demonstrate that you can style a `<details>` element based on whether it's open or not with an attribute selector.



That gives us something that looks like this. And we did it with zero JavaScript!



Here’s another example of what you can accomplish with HTML and a bit of smart CSS. The tabs across the top are actually radio inputs. With CSS we can hide the radio input itself and style the label to look like a tab. Based on which radio button is checked, we make one of the three content divs visible. If you're interested in seeing the code for this, I have a CodePen demo you can find in the slide notes. That's just two little examples, there is much much more where that came from. But I just wanted to give you a taste of HTML/CSS only solutions.

Use TypeScript without a build tool

You might say, okay Blake, but I have a real app and I do need JavaScript. And in fact, I need TypeScript.

TypeScript doesn’t run in the browser, so you will need to compile it into JavaScript first, which means you'll need a build step. But *writing* TypeScript is not the only way to *use* TypeScript. Most modern code editors use TypeScript's language server to type check your regular JavaScript files.

A quick way to make that explicit is to put the following comment at the beginning of your JavaScript file. Even with this alone you will get some basic type checking of your JavaScript code. But if you want to go the extra mile and add type notation of your own, you can do it with JSDoc.

Add type notation with JSDoc

```
/**
 * Add two numbers together.
 * @param {number} a
 * @param {number} b
 * @returns {number}
 */
function add(a, b) {
  return a + b;
}

/** @type {number} */
const sum = add(13, 29); // 42

console.log(add(6, '7')); // '67'
- type error
```

JSDoc is a powerful tool that can generate documentation from your codebase based on a special comment syntax. In our case, we're not actually going to generate any documentation. But by using the JSDoc comment syntax, we can tell the editor what types we expect. For example, we have this `add` function.

We put a comment above it and start with `/**` for it to be recognized as JSDoc. Then we can use a handful of tags to inform the editor about what the function expects. We're using the `@param` tag to tell the editor that we expect both parameters to be numbers. And we use the `@returns` tag to type the return value as a number. Inside the curly brackets, you can typically use any valid TypeScript expression.

```
/**
 * @typedef {Object} User
 * @property {string} name
 * @property {number} age
 */

/** @type {User} */
const currentUser = { name: 'Tilly', age: 31 };
```

We can even define custom object types using the `@typedef` tag.

So in this example we've created a type called `User`. It should have a name and an age. And then we're creating a variable that represents a `User`. If we tried to use the string 31 for age rather than a number, we would get a little red squiggly underline in the editor, and hovering over it would tell us what's wrong.

If you are used to TypeScript, then this syntax should feel familiar, if a bit more verbose. VS Code in particular has great editor support for type-checking this way.

jsconfig.json

```
{
  "compilerOptions": {
    "checkJs": true,
    "target": "ES2022",
    "module": "nodenext", // TS uses
NodeNext ESM rules; closest to
browser ESM
    "lib": ["ES2022", "DOM",
"DOM.Iterable"],
    "verbatimModuleSyntax": true //
keeps your import/export syntax
  },
  "include": ["js/**/*"],
  "exclude": ["node_modules",
"dist"]
}
```

Now if you have a lot of JavaScript files, you can enable type checking in all of them with a jsconfig file. This sample one is specifically tuned for no-build situations. It the editor to check your JavaScript files, that you're using newish JavaScript features, that you might use JavaScript modules which we'll look at shortly. And that you are in a browser environment so you want DOM objects available.

Use the include option to specify what files to check. Use `exclude` to skip over `node_modules` and any build folders you may have. (Which is hopefully none).

Use a TypeScript declaration file

types.d.ts

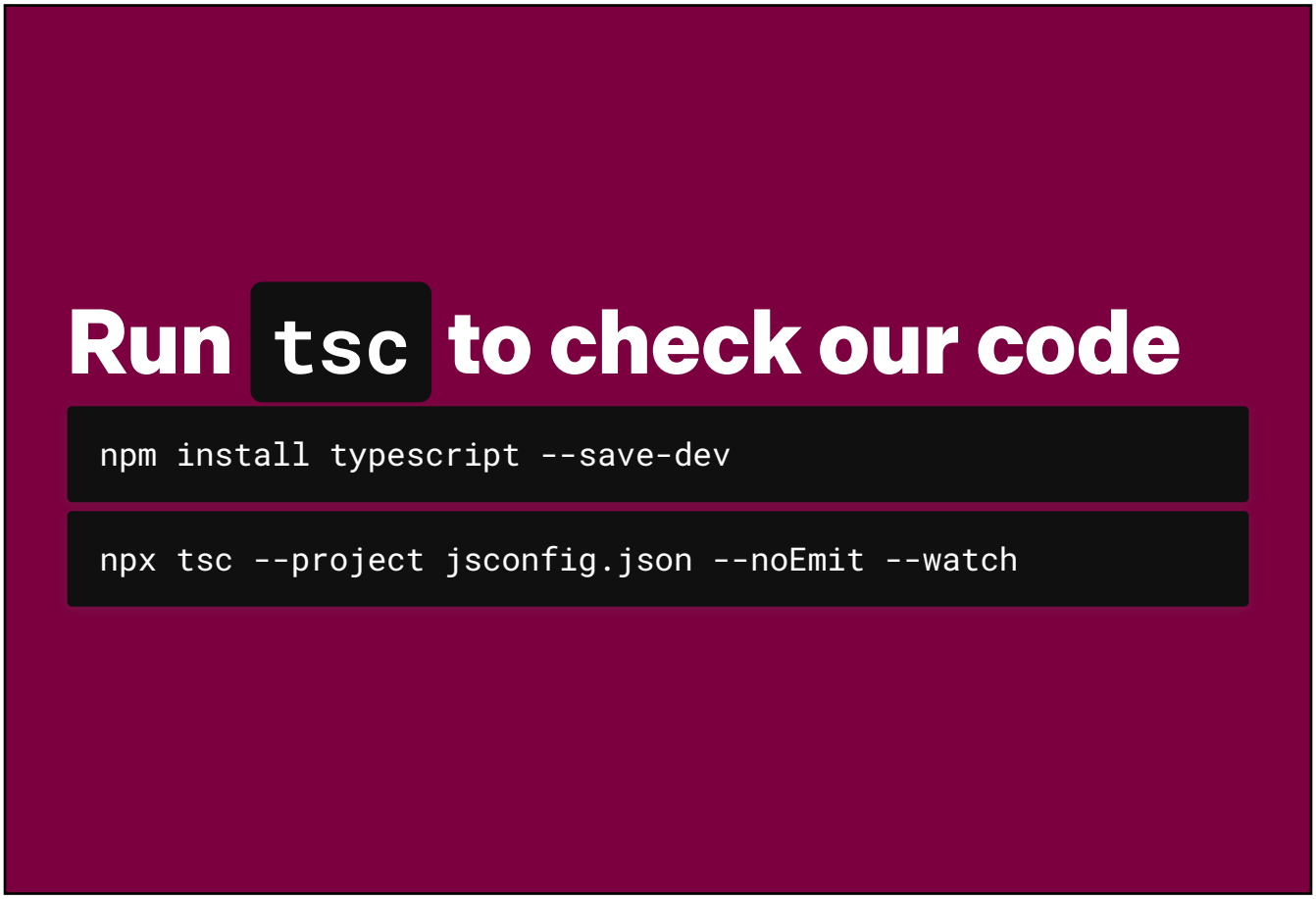
```ts
export type Person = {
  id: string;
  name: string;
  email: string;
  age: number;
  roles: string[];
};
```

app.js

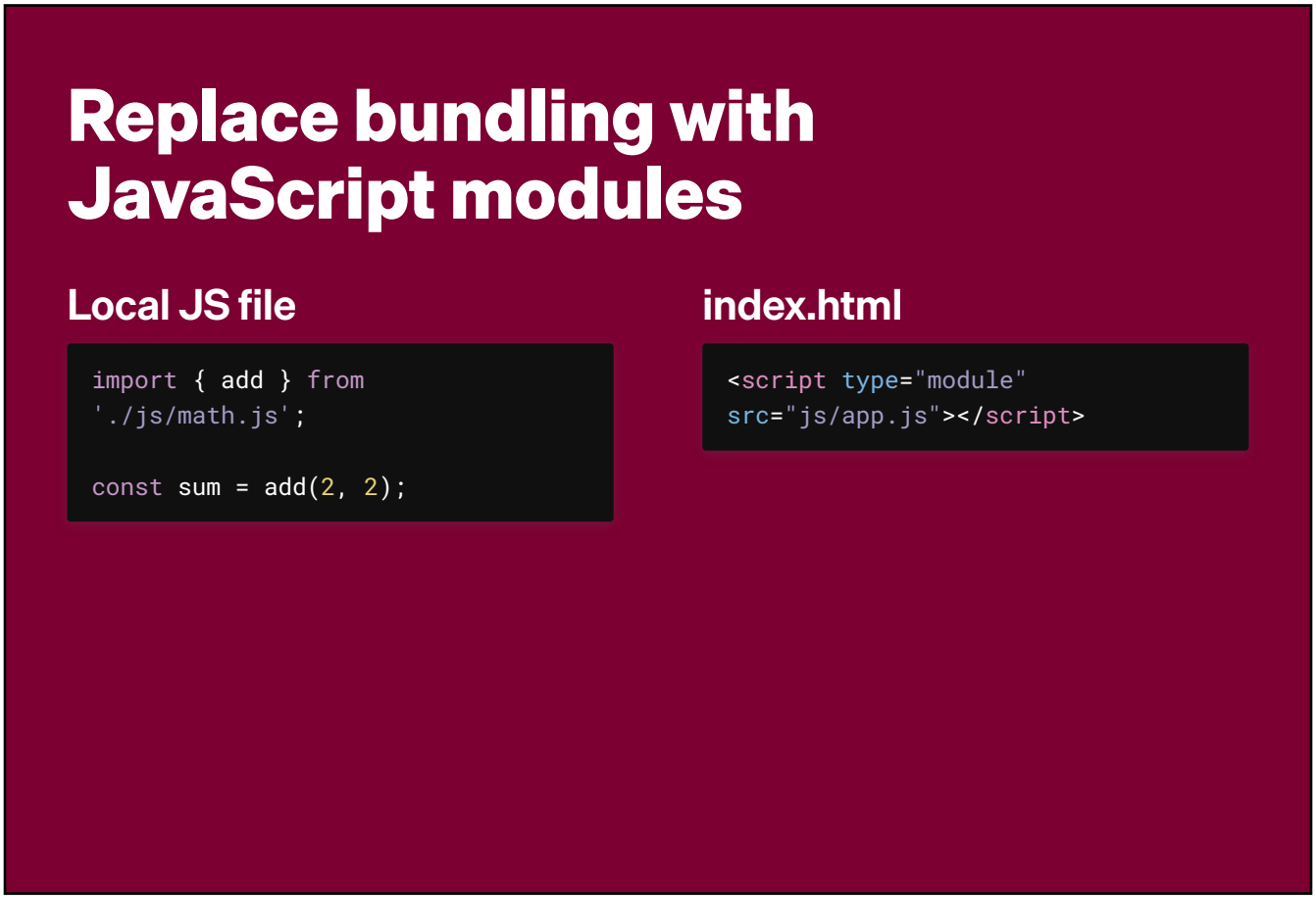
```js
/** @typedef
{import("../types.d.ts").Person}
Person */

/** @type {Person} */
let person = {
  id: "c0ffee",
  name: "Blake Watson",
  email: "blake@blakewatson.com",
  age: 40,
  roles: ["developer",
"dungeon_master"]
};
```

If you're a TypeScript veteran and you really can't be bothered to make a bunch of custom types with JSDoc, you could use a TypeScript declaration file and put a bunch of your types there. Or more than one. I mean there's kind of a million ways to get types into your project. And then using the import function syntax you can import types in JSDoc to use in your regular JavaScript files.

If you do this, just make sure your TypeScript files are included in your `jsconfig.json`.



You can even type check with `tsc` if you want. This snippet will watch your code and report any errors. Handy for local development and if you want a CI/CD pipeline that fails when there are type errors.



Alright, continuing the JavaScript train. You're probably familiar with npm installing something and then importing it into your code.

It turns out that browsers support JavaScript modules natively. So you can break your project up and import things and the browser can read those just fine. When you add your script to the page like you normally would, you just give the script tag a `type="module"` attribute to let it know you're loading a module.

It's worth noting that this also automatically defers the script—same as if you used the `defer` attribute.

Okay, that's cool and all, but that doesn't help me stuff a bunch of NPM packages into my no-build project. So how do I go about doing that? If you're using modules, you need the ESM-compatible version of the library or framework that you are after. (ESM meaning ECMAScript module, ECMAScript meaning the standard that JavaScript is based off of which isn’t called JavaScript because Oracle’s army of lawyers said no).

Finding the ESM build of a project if it exists is easier said than done.

```
https://unpkg.com/vue@3.5.22/dist/vue.esm-browser.js
```

```
<div id="app">{{ message }}</div>

<script type="module">
  import { createApp, ref } from './js/vue.esm-browser.js'

  createApp({
    setup() {
      const message = ref('Hello Vue!')
      return {
        message
      }
    }
  }).mount('#app')
</script>
```

An example of one that's easy is Vue.js. You can grab the global version which is totally fine but in this case we're going to grab the ESM compatible version. They have instructions on how to get it and they boil down to grabbing it from a CDN.

You could import it from the CDN directly, but I prefer to download it and host it myself.

So here is a small Vue.js app using imports that can run directly in the browser.

How to get ESM compatible builds

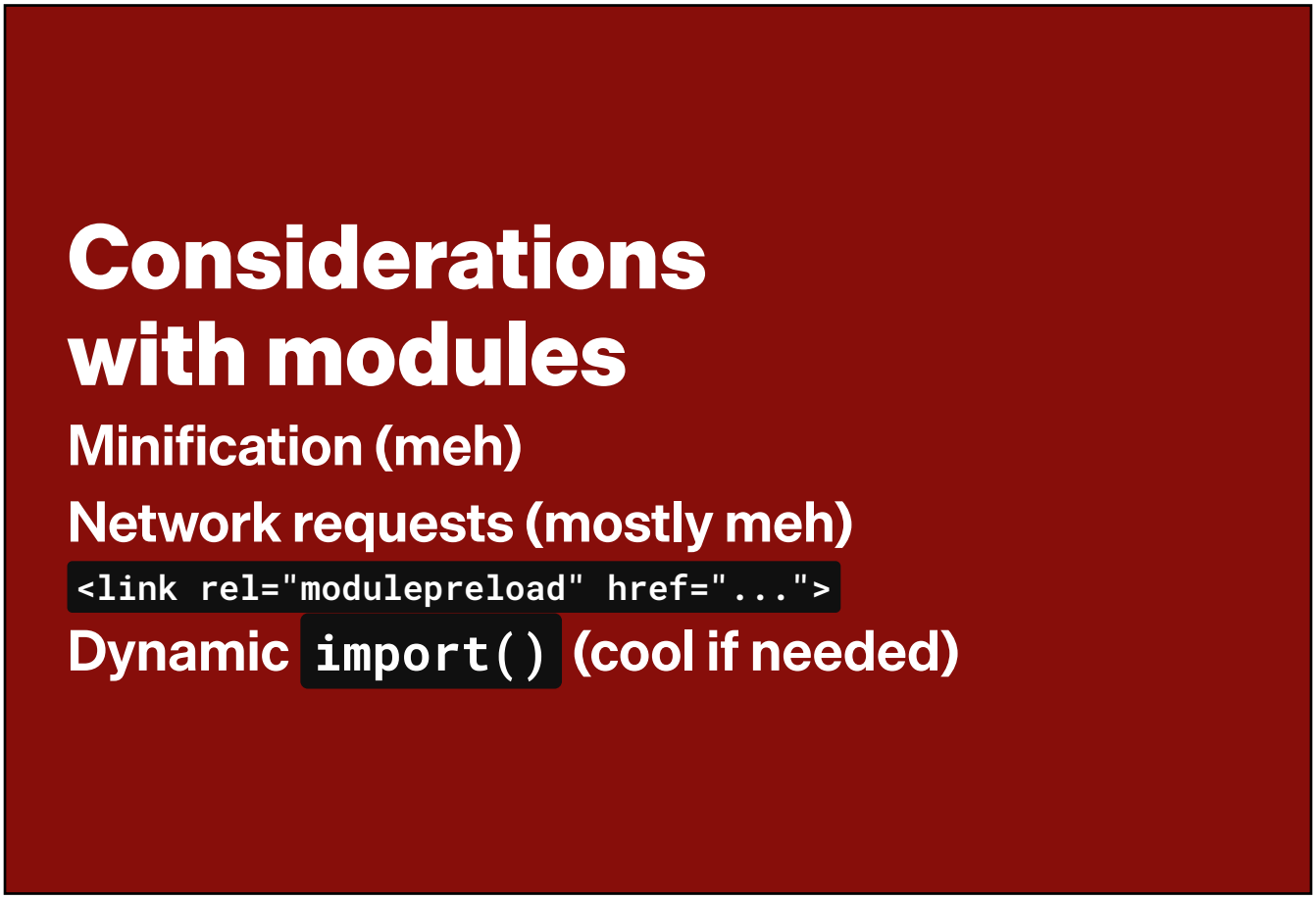
- npmjs.com Code tab
- jsDelivr.com
- esm.sh
- download-esm

It's hard to find ESM compatible builds sometimes because skipping the build step is a very intentional choice these days. Projects like to be installed with npm and they don't make it easy to do otherwise sometimes.

But I've had luck using these methods. Often times you can go directly to npm and go to a package and pop over to the code tab. If they have ESM builds there, you can grab them.

jsDelivr and esm.sh are CDNs that often have automated ESM versions of packages. And jsDelivr has a nice UI for browsing. So you can look for them that way.

Sometimes ESM files from a CDN will turn around and import other files from a CDN which may not be what you want. In that case, you can use this tool by Simon Wilson called download-esm which given a URL to an ESM package on a CDN will download a flat list of all the dependencies for you.



There are more things I didn't have time to cover. But I wanted to quickly address some of the trade-offs that you're making when you use modules.

No-build means no minification. But that's not nearly as important as gzipping, which your server is probably already doing anyway.

Modules do mean more network requests, but this mostly isn't an issue. The thing that can get you into trouble is a bunch of deeply nested module dependencies. If you have that situation, you can use the link tag with `rel=modulepreload` to tell the browser ahead of time what modules you're going to be loading.

We don't have dead code elimination, but we do have the import function, which you could use inside of conditionals to import extra JavaScript if you need to.

What about JavaScript frameworks? Giving up the build step doesn't mean you have to give up on fancy reactive JavaScript frameworks. You have a decent number of choices here and, if you combine them with the JavaScript module pattern we looked at previously, you can get a no-build setup that feels pretty nice.

Vue.js

```
<div id="app">{{ message }}</div>

<script type="module">
  import { createApp, ref } from './js/vue.esm-browser.js'

  createApp({
    setup() {
      const message = ref('Hello Vue!')
      return {
        message
      }
    }
  }).mount('#app')
</script>
```

We looked at how you could use Vue earlier as they have an ESM compatible build. That's totally an option. You can use the CDN global version of Vue as well if you wanted to. But I think for many projects there is a better, easier option. And that's AlpineJS.

No-build frameworks

Alpine.js

```
<script src="//unpkg.com/alpinejs" defer></script>

<div x-data="{ open: false }">
    <button @click="open = true">
      Expand
    </button>

    <span x-show="open">
        Content...
    </span>
</div>
```

I was introduced to this framework a couple of years ago by my brother, Matt, who is chilling in the back over there.

At first I was unimpressed but then I started using it and it's actually kind of amazing. You plop it into a script tag and just start using it. No build or configuration necessary. It uses attributes in your HTML.

So in this example, by putting `x-data` on this `<div>`, you're creating an Alpine component. That attribute defines whatever state your component needs. So in this case, it's the boolean value `open`.

Then you have a button, it has an `@click` attribute, which should look familiar if you've used Vue before. So clicking that button changes the value to `true`. And then below the button, we have a `<span>` with an `x-show` attribute on it. That attribute is a conditional, so it will hide or show the element based on the value you provide it, which in this case is our boolean value, `open`. So this span will show up if it's true, or hide if it's false.

Alpine is nice because it has a relatively small API. There's not as much stuff to wrap your head around as some other frameworks. It works right out of the box with no configuration. And it really works great with the no-build philosophy.

```
<div x-data="{ todos: [], newTask: '' }" x-init="
  const data = await fetch('https://api.jsonbin.io/v3/b/68f4e966ae596e708f1cb665',
    { headers: { 'x-access-key': '$2a$10$c5RwWN2B7YgIKM5vTLdPSOvpsf43zuT.Nsbsi4f3sYNPp.WvIZqSe' } }
  );
  todos = (await data.json()).record;
">
  <h1>Todos</h1>
  <ul>
    <template x-for="todo in todos">
      <li x-text="todo"></li>
    </template>
  </ul>

  <form @submit.prevent="todos.push(newTask); newTask = '';">
    <label>
      New todo
      <input type="text" x-model="newTask">
    </label>

    <button>Add</button>
  </form>
</div>
```

I want to show you this slightly more complex example just to give you a better idea of what's possible with Alpine.

In this example we have a super simple to-do list. First we create a component by adding `x-data` and some state. We then add an `x-init` attribute. This is code that will run when the component is first initialized. The wild thing is that we can write asynchronous JavaScript inside most Alpine attributes. So I can fetch some existing to-dos from an API and populate the to-dos array from that. Yes, it's fugly, but it works and it's kind of amazing. But don't worry there are ways to write organized Alpine code if you don't like this style.

Alright, so we can loop over our to-dos using the `x-for` attribute. In Alpine, you need to put that on a template tag, but no biggie. Each list item will set its inner text to the content of the to-do using the `x-text` attribute.

And then below the list we have a form. It has a text input and a button. We use Alpine's `x-model` attribute to keep track of what's in the text input. If you'll look back up at our `x-data` attribute at the top of the component, you can see we're tracking a string called `newTask`. So our input is bound to that value.

Then our form has a submit handler which will push the new task to the list and then clear the input. It also calls `preventDefault` so that we don't get a page reload.

If you are a diehard React fan, bless your heart, then your problem becomes finding a way to run JSX. Browsers don't know JSX but they do know how to process tagged template literals.

htm

```
import { html, Component } from 'https://unpkg.com/htm/preact/standalone.module.js';

export class Logs extends Component {
  toggle () {
    this. setState({ show: !this.state.show });
  }
  render ({ logs = [], ...props }, { show }) {
    return html`
      <div class="logs" ...${props}>
        <button onClick=${() => this.toggle()}>🔻</button>
        <!-- If expanded, render all logs -->
        ${show && html`
          <section class="logs" ...${props}>
            <!-- maps and values work just like JSX. -->
            ${logs.map(log => html`
              <${Log}...${l09} >
            `)}
          </section>
        `}
      </div>
    `;
  }
}
```

Fortunately, the creator of Preact made htm which is a utility for processing template literals in a way that makes it feel like JSX.

You can use HTM with ReactDOM or Preact. And if you use Preact you can get both HTM and Preact together in an ESM compatible file.

I won't go line by line on this, but it's just to show that the syntax is very similar to JSX. The cool thing about it is that this is totally runnable by the browser natively, which means you can use this code without a build step. Pretty incredible. If you end up using this, definitely get a VS Code extension that will syntax highlight HTML in template literals. You can look at Lit or FAST, they both have extensions for that.

Web components

```
<ajax-form
    prevent-default
    msg-submitting="Signing up..."
    msg-success="Congrats! You're on the list!"
    target="#item-list"
>
    <form method="post" action="/subscribe">
        <label for="email">Add your email to join the
newsletter</label>
        <input type="email" id="email" name="email">
        <button>Join</button>
    </form>
</ajax-form>
```

I could give an entire talk on web components, but unfortunately I only have time to give them one slide, but I've had the chance to use these in my job and they are pretty incredible.

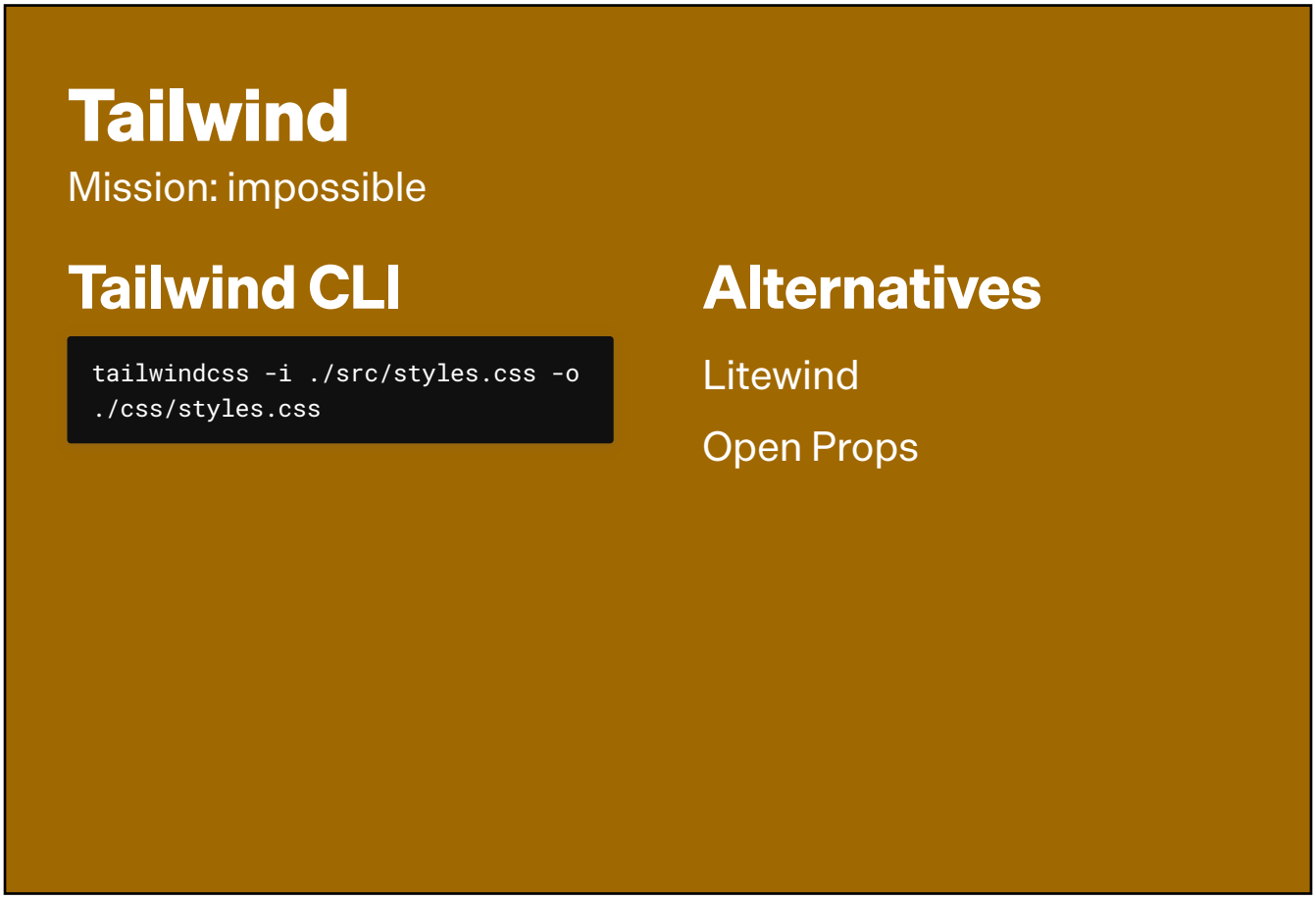
A web component is essentially a custom HTML element that you create with JavaScript. And this `ajax-form` web component from my internet buddy Chris Ferdinandi is a really cool one.

The great thing about it is that it's essentially a normal form at heart and if JavaScript is unavailable, this form will submit to the back end like any normal form would. But if JavaScript *is* available the web component takes over and turns this form submission into an Ajax request. You can give this component a few different props to customize it. And you're off to the races.

A well made web component will long outlive your build step. And they are interoperable with whatever framework you decide to use.

Finally, let's talk about Tailwind. Can you have tailwind in a no build project?

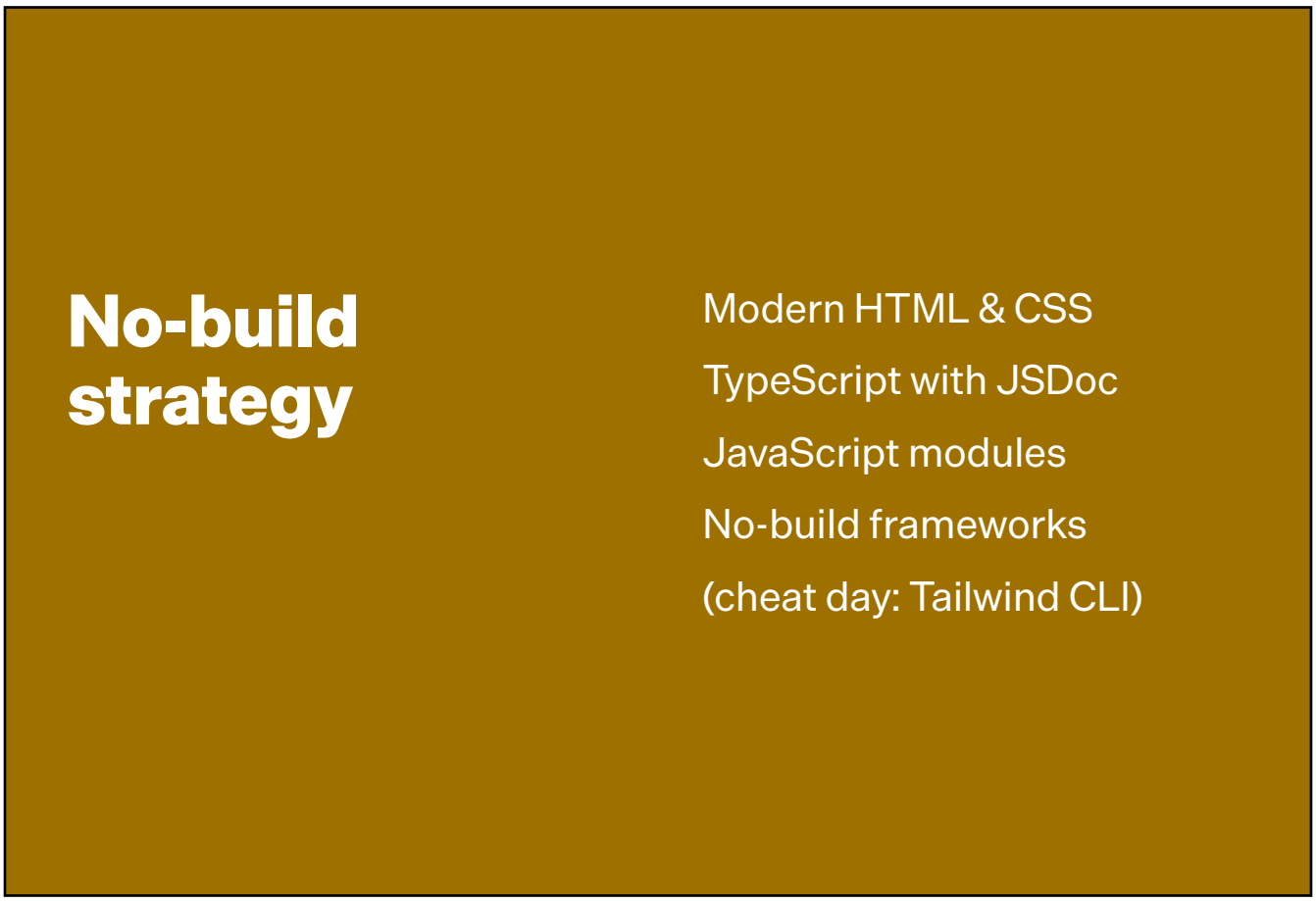
There's really just no good way to use Tailwind without a build step. There is a CDN version, but it's huge and does a lot of processing on the fly, and it's only made for messing around, not for production code.



If you really want to use Tailwind, and I wouldn't blame you if you did, I suggest keeping it as simple as possible and using their CLI and making that the only build step you have.

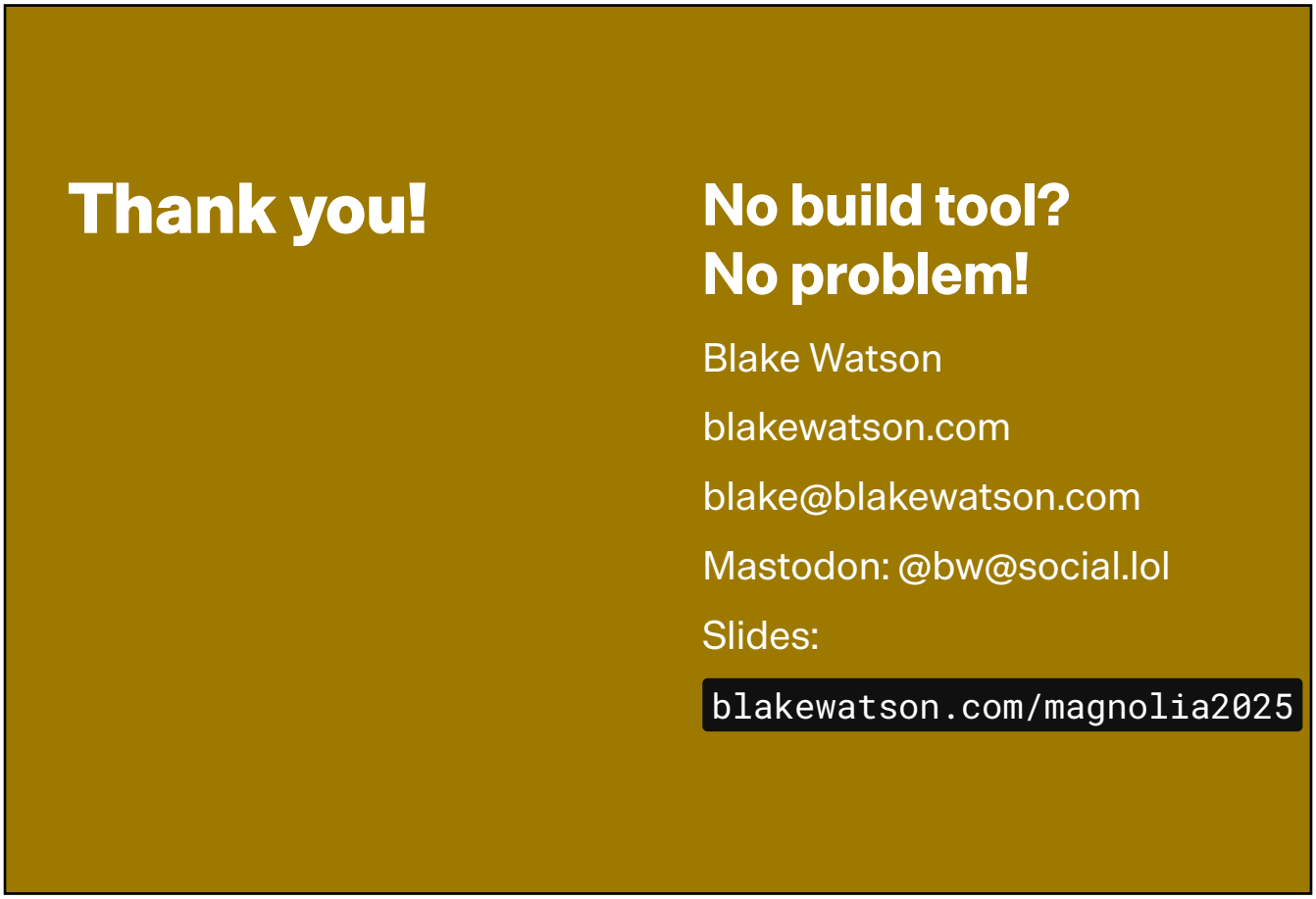
If you'd like to go absolutely no build step, you could try the alternatives. One is Litewind, which is essentially a static version of all of the classes from Tailwind 3. There's no processing so you can't do any of the custom bracket stuff.

The other option is not a set of utility classes but a set of predefined CSS variables that you can use. It's called Open Props. I used it on the HTML for people website. It's nice because it gives you a coherent set of colors and spacing and other values that you can reuse across your styles. So it's not really utility classes but if what you like about Tailwind is having some go-to design tokens, you can get that with Open Props.



We covered a ton of stuff in this talk and you could deep dive on every one of them probably, but to recap, you don't have to follow the npm installation instructions of your favorite framework. You can choose to adopt a no-build strategy.

And the options you have available are to use the modern technologies available in the browser. You can use the capability of modern editors to give you type checking in your regular JavaScript files. You can take advantage of JavaScript modules without a bundler. You can choose frameworks that don't need to be bundled or that have no build versions. And if you absolutely want to use Tailwind, then you'll have to have a cheat day and use the Tailwind CLI.



Thanks for listening to Old Man Yells at Clouds. I know everyone's ready to get lunch. If you have a question or want to get in touch with me about any of this stuff, you can go to my website, email me, get me on Mastodon, or just come up to me. I'll be at the conference today and tomorrow. Thanks everyone!