tailwind UI

Documentation

Getting set up

Requirements

All of the components in Tailwind UI are designed for Tailwind CSS >= v1.8. To make sure that you are on the latest version of Tailwind, update via npm or Yarn:

```
# Using npm
npm install tailwindcss@latest

# Using Yarn
yarn add tailwindcss@latest
```

Add the @tailwindcss/ui plugin

During early access, the components in Tailwind UI depend on some extensions we've added to the default Tailwind CSS config (like extra spacing values, updated, colors, additional shadows, etc.)

These extensions will make their way into Tailwind itself in the future but right now you need to install the `@tailwindcss/ui` plugin to add these extensions to your project:

```
# Using npm
npm install @tailwindcss/ui

# Using yarn
yarn add @tailwindcss/ui
```

Then add `@tailwindcss/ui` to your Tailwind plugin list:

```
// tailwind.config.js
module.exports = {
  plugins: [
    require('@tailwindcss/ui'),
  ]
}
```

> *Make sure you don't generate your Tailwind config using the `--full` flag or the extensions added by the plugin will be superseded by the values in your config. Keep any of your own modifications under `theme.extend` to ensure everything works as expected.*

We also have a Tailwind UI-compatible CDN build available if that's more your style:

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/@tailwindcss/ui@latest/dist/tailwind-ui.min.
```

You should really be using a proper build though if you can — the development build of Tailwind is pretty beefy, especially with the Tailwind UI extensions.

For more details on what exactly the plugin adds to Tailwind, see "How @tailwindcss/ui extends Tailwind".

Configuring sidebar breakpoints

If you are building a site that uses one of our sidebar layout application shells, you'll want to tweak the breakpoints in your app to account for the width of the sidebar. If you don't, you'll find that wide components like lists and tables may not look right at certain screen sizes because they are expecting to have more space available to them.

The easiest way to do this is using the `layout` option exposed by our `@tailwindcss/ui` plugin:

```
// tailwind.config.js
module.exports = {
  plugins: [
    require('@tailwindcss/ui')({
      layout: 'sidebar',
    })
  ]
}
```

This will automatically configure your breakpoints to accommodate our standard 256px sidebar width.

Optional: Add the Inter font family

We've used Inter font family for all of the Tailwind UI examples because it's a beautiful font for UI design and is completely open-source and free. Using a custom font is nice because it allows us to make the components look the same on all browsers and operating systems.

You can use any font you want in your own project of course, but if you'd like to use Inter, the easiest way is to first add it via the CDN:

```
<link rel="stylesheet" href="https://rsms.me/inter/inter.css">
```

Then add "Inter var" to your "sans" font family in your Tailwind config:

```
// tailwind.config.js
const defaultTheme = require('tailwindcss/defaultTheme')

module.exports = {
  theme: {
    extend: {
      fontFamily: {
        sans: ['Inter var', ...defaultTheme.fontFamily.sans],
      },
    },
  },
  plugins: [
    require('@tailwindcss/ui'),
  ]
}
```

Integrating with JavaScript frameworks

To make Tailwind UI as universal as possible, we've built all the components using HTML only.

The vast majority of components don't need JavaScript at all and are completely ready to go out of the box, but things that are interactive like dropdowns, modals, etc. require you to write some JS to make them work the way you'd expect.

In these situations we've provided some simple comments in the HTML to explain things like what classes you need use for different states *(like a toggle switch being on or off)*, or what classes we recommend for transitioning elements on to or off of the screen *(like a modal opening)*.

This guide explains how to take these comments and translate them into working code using some popular JS libraries.

Accessibility

We've done our best to ensure that all of the markup in Tailwind UI is as accessible as possible, but when you're building interactive components, **many accessibility best practices can only be implemented with JavaScript.**

For example:

- **Making sure components are properly keyboard accessible** *(dropdowns should be navigated with up/down arrow keys, modals should close when you press escape, tabs should be selected using the left/right arrow keys, etc.)*
- **Correctly handling focus** *(you shouldn't be able to tab to an element behind a modal, the first item in a dropdown should be auto-focused when the dropdown opens, etc.)*
- **Synchronizing ARIA attributes with component state** *(adding `aria-expanded="true"` when a dropdown is open, setting `aria-checked` to true when a toggle is on, updating `aria-activedescendant` when navigating the options in an autocomplete, etc.)*
- ...and many other concerns.

Because the components in Tailwind UI are HTML-only, it is up to you to follow accessibility best practices adding interactive behavior with JavaScript.

To learn more about building accessible UI components, we recommend studying the WAI-ARIA Authoring Practices published by the W3C.

React

The first thing you'll need to do to incorporate any Tailwind UI component into a React project is convert the HTML to valid JSX, making sure to update attributes like `class` to `className`, `for` to `htmlFor`, `fill-rule` to `fillRule`, etc.

```
- <button type="button" class="inline-flex items-center px-4 py-2 border border-transparent text-sm le
```

```
+ <button type="button" className="inline-flex items-center px-4 py-2 border border-transparent text-s
-   <svg class="-ml-1 mr-2 h-5 w-5" fill="currentColor" viewBox="0 0 20 20">
+   <svg className="-ml-1 mr-2 h-5 w-5" fill="currentColor" viewBox="0 0 20 20">
-     <path fill-rule="evenodd" clip-rule="evenodd" d="M2.003 5.884L10 9.882l7.997-3.998A2 2 0 0016 4H
+     <path fillRule="evenodd" clipRule="evenodd" d="M2.003 5.884L10 9.882l7.997-3.998A2 2 0 0016 4H4a
    </svg>
    Button text
  </button>
```

Dynamic classes

When an element needs different classes applied based on some state *(like a toggle being on or off)*, we list the classes for each state in a comment directly above the element:

```
<!-- On: "bg-indigo-600", Off: "bg-gray-200" -->
<span aria-checked="false" class="bg-gray-200 relative inline-block flex-shrink-0 h-6 w-11 border-2 bo
  <!-- On: "translate-x-5", Off: "translate-x-0" -->
  <span aria-hidden="true" class="translate-x-0 inline-block h-5 w-5 rounded-full bg-white shadow tran
</span>
```

The HTML we provide is always pre-configured for one of the defined states, and the classes that you need to change when switching states are always at the very beginning of the class list so they are easy to find.

To make this work in React, conditionally include the necessary classes on each element by checking a prop or piece of state:

```
import { useState } from 'react'

function SimpleToggle() {
  const [isOn, setIsOn] = useState(false)
  return (
    <span
      role="checkbox"
      aria-checked={isOn}
      tabindex="0"
      onClick={() => setIsOn(!isOn)}
      class={`${isOn ? 'bg-indigo-600' : 'bg-gray-200'} relative inline-block flex-shrink-0 h-6 w-11 b
    >
      <span
        aria-hidden="true"
        class={`${isOn ? 'translate-x-5' : 'translate-x-0'} inline-block h-5 w-5 rounded-full bg-white
      ></span>
```

```
      </span>
    )
  }
```

> *We've included a basic click handler here to demonstrate the general idea, but **please reference the WAI-ARIA Authoring Practices when building components like this** to ensure you implement all of the necessary keyboard interactions and properly manage any required ARIA attributes.*

Enter/Leave transitions

For elements that should be dynamically shown or hidden *(like the panel on a dropdown)*, we include the recommended transition styles in a comment directly above the dynamic element:

```
<div class="relative ...">
  <button type="button" class="...">
    Options
  </button>

  <!--
    Show/hide this element based on the dropdown state

    Entering: "transition ease-out duration-100 transform"
      From: "opacity-0 scale-95"
      To: "opacity-100 scale-100"

    Closing: "transition ease-in duration-75 transform"
      From: "opacity-100 scale-100"
      To: "opacity-0 scale-95"
  -->
  <div class="origin-top-right absolute right-0 mt-2 w-56 rounded-md shadow-lg">
    <div class="rounded-md bg-white shadow-xs">
      <!-- Snipped  -->
    </div>
  </div>
</div>
```

React doesn't include a first-party transition component out of the box, so we've built one ourselves that's tailor-made for the Tailwind workflow, and published it as part of @tailwindui/react, a set of Tailwind-ready component primitives we're working on.

To get started, install the @tailwindui/react library.

To implement a transition, wrap the dynamic element in the `Transition` component, passing through the following props:

- `show` — whether or not the dynamic element should be showing

- `show`, whether or not the dynamic element should be showing
- `enter`, a list of classes that should be present during the entire enter phase
- `enterFrom`, a list of classes that represent the element's state at the beginning of the enter phase
- `enterTo`, a list of classes that represent the element's state at the end of the enter phase
- `leave`, a list of classes that should be present during the entire leave phase
- `leaveFrom`, a list of classes that represent the element's state at the beginning of the leave phase
- `leaveTo`, a list of classes that represent the element's state at the end of the leave phase

Here's what that looks like when applied to the dropdown example from above:

```
import { useState } from 'react'
import { Transition } from '@tailwindui/react'

function Dropdown() {
  const [isOpen, setIsOpen] = useState(false)
  return (
    <div className="relative ...">
      <button type="button" onClick={() => setIsOpen(!isOpen)} className="...">
        Options
      </button>

      <Transition
        show={isOpen}

        enter="transition ease-out duration-100 transform"
        enterFrom="opacity-0 scale-95"
        enterTo="opacity-100 scale-100"
        leave="transition ease-in duration-75 transform"
        leaveFrom="opacity-100 scale-100"
        leaveTo="opacity-0 scale-95"
      >
        {(ref) => (
          <div ref={ref} className="origin-top-right absolute right-0 mt-2 w-56 rounded-md shadow-lg">
            <div className="rounded-md bg-white shadow-xs">
              {/* Snipped */}
            </div>
          </div>
        )}
      </Transition>
    </div>
  )
}
```

> *We've included a basic click handler here to demonstrate the general idea, but* ***please reference the WAI-ARIA Authoring Practices when building components like this*** *to ensure you implement all of the necessary keyboard interactions and properly manage any required ARIA attributes.*

Vue.js

Since Vue templates are no different than regular HTML, the first step to pulling a Tailwind UI component into a Vue project is to just paste the code inside a `<template>` tag of a single file Vue component.

Dynamic classes

When an element needs different classes applied based on some state *(like a toggle being on or off)*, we list the classes for each state in a comment directly above the element:

```
<!-- On: "bg-indigo-600", Off: "bg-gray-200" -->
<span aria-checked="false" class="bg-gray-200 relative inline-block flex-shrink-0 h-6 w-11 border-2 bo
  <!-- On: "translate-x-5", Off: "translate-x-0" -->
  <span aria-hidden="true" class="translate-x-0 inline-block h-5 w-5 rounded-full bg-white shadow trar
</span>
```

The HTML we provide is always pre-configured for one of the defined states, and the classes that you need to change when switching states are always at the very beginning of the class list so they are easy to find.

To make this work in Vue, conditionally include the necessary classes on each element by checking a prop or piece of state:

```
<template>
  <span
    role="checkbox"
    :aria-checked="isOn"
    tabindex="0"
    @click="isOn = !isOn"
    :class="isOn ? 'bg-indigo-600' : 'bg-gray-200'"
    class="relative inline-block flex-shrink-0 h-6 w-11 border-2 border-transparent rounded-full cursc
  >
    <span
      aria-hidden="true"
      :class="isOn ? 'translate-x-5' : 'translate-x-0'"
      class="inline-block h-5 w-5 rounded-full bg-white shadow transform transition ease-in-out durati
    ></span>
  </span>
</template>
```

```
<script>
  export default {
    data: () => ({
      isOn: false,
    })
  }
</script>
```

> *We've included a basic click handler here to demonstrate the general idea, but* ***please reference the WAI-ARIA Authoring Practices when building components like this*** *to ensure you implement all of the necessary keyboard interactions and properly manage any required ARIA attributes.*

Enter/Leave transitions

For elements that should be dynamically shown or hidden *(like the panel on a dropdown)*, we include the recommended transition styles in a comment directly above the dynamic element:

```
<div class="relative ...">
  <button type="button" class="...">
    Options
  </button>

  <!--

    Show/hide this element based on the dropdown state

    Entering: "transition ease-out duration-100 transform"
      From: "opacity-0 scale-95"
      To: "opacity-100 scale-100"
    Closing: "transition ease-in duration-75 transform"
      From: "opacity-100 scale-100"
      To: "opacity-0 scale-95"
  -->
  <div class="origin-top-right absolute right-0 mt-2 w-56 rounded-md shadow-lg">
    <div class="rounded-md bg-white shadow-xs">
      <!-- Snipped  -->
    </div>
  </div>
</div>
```

To implement transitions like this in Vue, use Vue's built-in `<transition>` component and pass the necessary transition styles through the custom transition class props:

```
<template>
  <div class="relative ...">
    <button type="button" @click="isOpen = !isOpen" class="...">
      Options
    </button>

    <transition
      enter-active-class="transition ease-out duration-100 transform"
      enter-class="opacity-0 scale-95"
      enter-to-class="opacity-100 scale-100"
      leave-active-class="transition ease-in duration-75 transform"
      leave-class="opacity-100 scale-100"
      leave-to-class="opacity-0 scale-95"
    >
      <div v-show="isOpen" class="origin-top-right absolute right-0 mt-2 w-56 rounded-md shadow-lg">
        <div class="rounded-md bg-white shadow-xs">
          <!-- Snipped  -->
        </div>
      </div>
    </transition>
  </div>
</template>

<script>

  export default {
    data: () => ({
      isOpen: false,
    })
  }
</script>
```

> *We've included a basic click handler here to demonstrate the general idea, but **please reference the WAI-ARIA Authoring Practices when building components like this** to ensure you implement all of the necessary keyboard interactions and properly manage any required ARIA attributes.*

Alpine

Alpine.js is a fairly new, heavily Vue-inspired library designed to make it easy to add interactive behavior to traditional server-rendered websites, using a light-weight declarative syntax directly in your HTML. It's what we use for the Tailwind UI site itself, and is without a doubt what I'd recommend if you'd otherwise be writing jQuery or vanilla JS.

Dynamic classes

When an element needs different classes applied based on some state *(like a toggle being on or off)*, we list the classes for each state in a comment directly above the element:

```
<!-- On: "bg-indigo-600", Off: "bg-gray-200" -->
<span aria-checked="false" class="bg-gray-200 relative inline-block flex-shrink-0 h-6 w-11 border-2 bo
  <!-- On: "translate-x-5", Off: "translate-x-0" -->
  <span aria-hidden="true" class="translate-x-0 inline-block h-5 w-5 rounded-full bg-white shadow trar
</span>
```

The HTML we provide is always pre-configured for one of the defined states, and the classes that you need to change when switching states are always at the very beginning of the class list so they are easy to find.

To make this work in Alpine, conditionally include the necessary classes on each element based on some state you've declared in `x-data`:

```
<span
  x-data="{ isOn: false }"
  @click="isOn = !isOn"
  :aria-checked="isOn"
  :class="{'bg-indigo-600': isOn, 'bg-gray-200': !isOn }"
  class="bg-gray-200 relative inline-block flex-shrink-0 h-6 w-11 border-2 border-transparent rounded-
  role="checkbox"
  tabindex="0"
>
  <span
    aria-hidden="true"
    :class="{'translate-x-5': isOn, 'translate-x-0': !isOn }"
    class="translate-x-0 inline-block h-5 w-5 rounded-full bg-white shadow transform transition ease-i
  ></span>
</span>
```

> *We've included a basic click handler here to demonstrate the general idea, but **please reference the WAI-ARIA Authoring Practices when building components like this** to ensure you implement all of the necessary keyboard interactions and properly manage any required ARIA attributes.*

Enter/Leave transitions

For elements that should be dynamically shown or hidden *(like the panel on a dropdown)*, we include the recommended transition styles in a comment directly above the dynamic element:

```
<div class="relative ...">
  <button type="button" class="...">
```

```
    Options
  </button>

  <!--
    Show/hide this element based on the dropdown state

    Entering: "transition ease-out duration-100 transform"
      From: "opacity-0 scale-95"
      To: "opacity-100 scale-100"
    Closing: "transition ease-in duration-75 transform"
      From: "opacity-100 scale-100"
      To: "opacity-0 scale-95"
  -->
  <div class="origin-top-right absolute right-0 mt-2 w-56 rounded-md shadow-lg">
    <div class="rounded-md bg-white shadow-xs">
      <!-- Snipped  -->
    </div>
  </div>
</div>
```

To implement transitions like this in Alpine, use the `x-transition` directives:

```
<div x-data="{ isOpen: false }" class="relative ...">
  <button type="button" @click="isOpen = !isOpen" class="...">
    Options
  </button>

  <div
    x-show="isOpen"
    x-transition:enter="transition ease-out duration-100 transform"
    x-transition:enter-start="opacity-0 scale-95"
    x-transition:enter-end="opacity-100 scale-100"
    x-transition:leave="transition ease-in duration-75 transform"
    x-transition:leave-start="opacity-100 scale-100"
    x-transition:leave-end="opacity-0 scale-95"
    class="origin-top-right absolute right-0 mt-2 w-56 rounded-md shadow-lg"
  >
    <div class="rounded-md bg-white shadow-xs">
      <!-- Snipped  -->
    </div>
  </div>
</div>
```

> *We've included a basic click handler here to demonstrate the general idea, but **please reference the WAI-ARIA Authoring Practices when building components like this** to ensure you implement all of the necessary keyboard interactions and properly manage any required ARIA attributes.*

Creating reusable components and partials

Since Tailwind UI provides components using simple static HTML, there is a lot of repetition that wouldn't actually be there in a real-world project where the HTML was being generated from some dynamic data source. We might give you a list component with 5 list items for example that have all the utilities duplicated on each one, whereas in your project you'll actually be generating those list items in a loop.

We can't predict what tools/frameworks/template languages everyone might be working with so we think this static HTML approach is the best way to make these components available to everyone no matter what ecosystem they normally work in, but to keep your project maintainable **we recommend creating reusable template partials or JS components for the Tailwind UI components you use using whatever tools you like.**

Learn more about this in the "Extracting Components" documentation on the Tailwind CSS website →

Customizing colors

The new color palette we've included with the @tailwindcss/ui plugin is designed to make the colors much more interchangeable than they were in the past.

For example, every 600 shade has a similar perceived brightness, which means you can take an indigo-600 button with white text and change it to teal-600 without worrying about the contrast being completely different.

That means that to change the colors in a component, it's usually as simple as just doing a global find and replace for "indigo" and swapping it to "teal".

Dealing with inherently light colors

There are some cases where this might not work the way you first expect, like yellow for example. To get a yellow that has the same brightness as a color like blue, the yellow has to actually be more like brown, so swapping blue-600 for yellow-600 isn't going to give you a bright yellow theme like you might be looking for.

That's because you can't use white text on a bright yellow button for example without the text being impossible to read — you actually need to use a dark text color, like say yellow-700 or 800 for text, and yellow-200 for the background.

In these cases you'll have to swap the colors with a bit more care.

In the future we're planning to provide some more "rules" around this to help you make these sorts of changes more confidently, even if you don't trust your own judgment with color.

Icons

The icons we've used in all of the Tailwind UI components are from a custom set we designed called "Heroicons" and are available completely for free here:

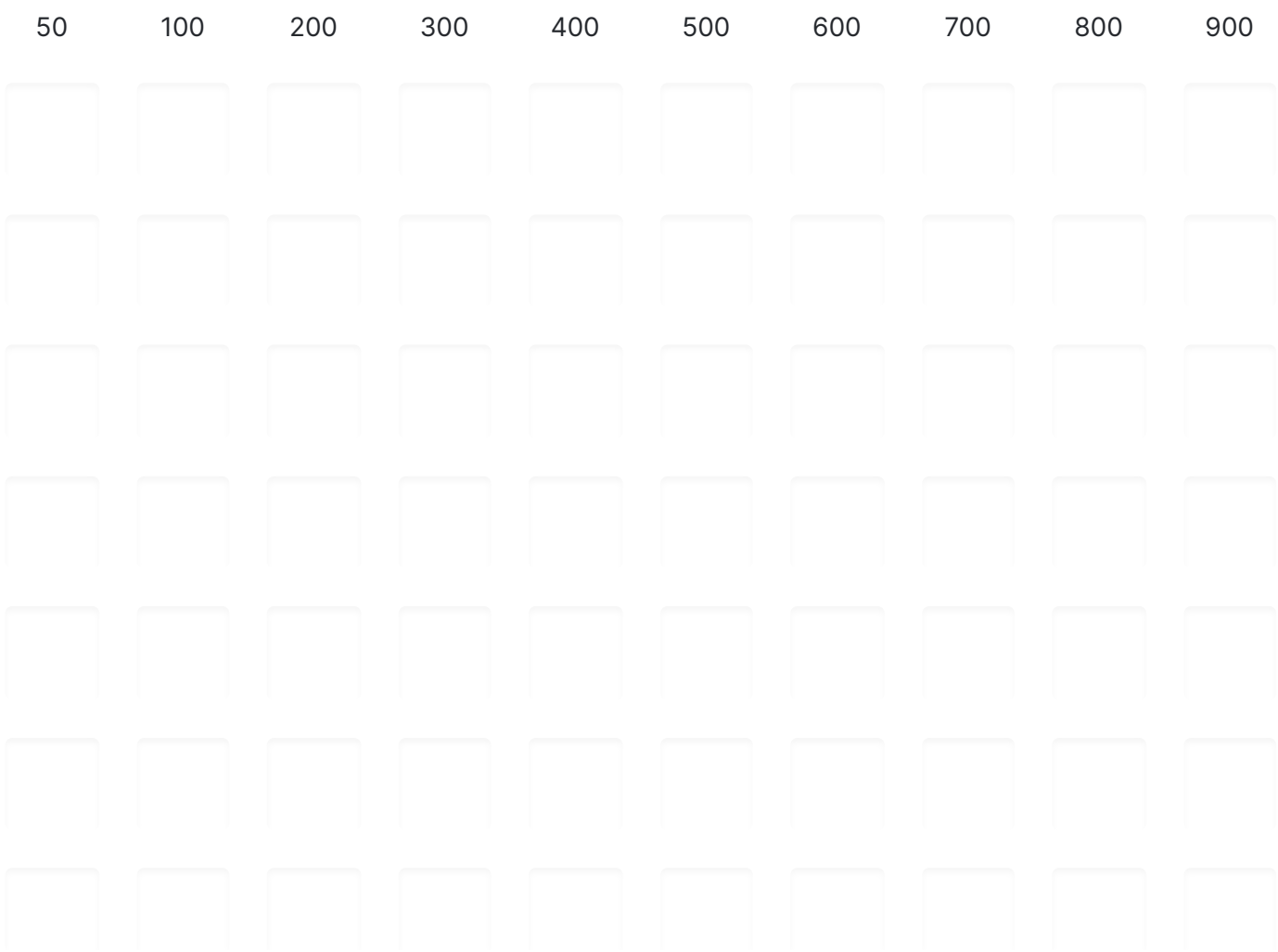
https://github.com/refactoringui/heroicons

How @tailwindcss/ui extends Tailwind

Updated color palette

Tailwind UI includes a brand new color palette that is more consistent across colors, and more vivid than the default Tailwind CSS color palette.

It also includes a new "50" shade, so there are now 10 shades per color instead of 9.



New gray

The default Tailwind gray is *extremely* saturated with blue and looks really good with certain colors, but it's a bit too opinionated to work with every color.

We've re-worked gray to be much more neutral (although still with a cool tint to it) and created a new "cool-gray" that has the same sort of look as the default Tailwind gray for people who prefer that one.

New default gray

Cool gray

Interchangeable colors

We've done our best to make the colors as interchangeable as possible, so for example if you see a component you like and it's using indigo for all of the colored sections but you want to use blue, you can just find and replace "indigo" for "blue" without changing any of the numbers and you will get really good results.

Button text Button text

Some colors are inherently lighter than others (like yellow) so swapping indigo for yellow is going to look much more like brown than yellow so you will probably want to rethink the color combinations a bit

in some cases, but generally switching colors is much easier than it was before.

Note that gray isn't swappable with the other colors — it's use cases are a bit different (every site uses gray in combination with brand colors) so that palette has been designed independently. The "gray" and "cool-gray" colors are perfectly interchangeable with each other though.

Solid box shadow for cut-out effects

We've added `shadow-solid` which is a 2px box shadow with no blur that adopts the current text color so you can use it for cut-out effects, for example:

Colored focus shadows

Tailwind includes a blue `shadow-outline` utility out of the box for focus styles, but when building the Tailwind UI components we realized we needed colored shadows for focus states on branded elements, high severity buttons, etc.

We've added a `shadow-outline-{color}` utility for each color so you can have nice focus states for any element:

adamwathan

group-focus variant

We've added a `group-focus` variant that works just like `group-hover`, but for focus. We use this a lot for things like changing the color of an icon in a link when the link is focused if the icon needs to be a different color than the text.

New max-w-7xl utility

We've added `max-w-7xl` which conveniently matches our 1280px breakpoint. This is useful for sidebar layouts where you want to constrain the content area to 1280px, but `max-w-screen-xl` doesn't actually do what you need since the `xl` breakpoint has been increased to account for the sidebar width.

More variants enabled by default

We've enabled group-hover, group-focus, focus-within, and active for many of the utilities that don't have them by default.

Bundled the custom forms plugin

Tailwind UI includes the **@tailwindcss/custom-forms** plugin out of the box so you can actually use custom selects, checkboxes, and radio buttons.