Fluent UI Web Components Overview

Article • 11/01/2021 • 2 minutes to read

Microsoft's Fluent UI Web Components are designed to help you build Fluent web apps using extensible Web Components. The package composes the `@microsoft/fast-foundation` Web Component package and styles it with the Fluent design language. You can use these Web Components flexibly, either from npm packages, from the CDN, or you can bring it into your project to customize and extend it.

With Fluent UI Web Components you can:

- Build a modern, standards-based, highly performant, highly accessible web front-end
- Build a web front end using only web platform code, no other frameworks, *or*
- Integrate with many popular frameworks like .NET, Blazor, Vue, React, etc.
- Build a web-standards UX built with W3C Web Component standards
- Leverage the existing Fluent UI design language as design tokens
- Customize the design language for your project by modifying and creating new design tokens
- Use the components out-of-the-box to build your web user experience *or*
- Customize, compose and build new Web Components based on the libraries that Fluent UI Web Components is built on, FAST
- Be part of a dynamic, open-web development community

What are Web Components?

"Web Components" is an umbrella term that refers to a collection of web standards focused on enabling the creation of custom HTML elements. Some of the standards under this umbrella include the ability to define new HTML tags, plug into a standard component lifecycle, encapsulate HTML rendering and CSS, parameterize CSS, skin components, and more. Each of these platform features is defined by the W3C and has shipped in every major browser today.

How does Fluent UI leverage Web Components?

Fluent UI Web Components are built directly on the W3C Web Component standards, and do not create their own separate component model. This allows our components to function the same as built-in, native HTML elements. You do not need a framework to

use Fluent UI components but you can use them in combination with any framework or library of your choice. See Integrations for more.

Joining the community

Looking to get answers to questions or engage with us in real time? Our community is most active on Discord. Submit requests and issues on GitHub, or join us by contributing on some good first issues via GitHub.

Next steps

- Quick start
- Install the packages
- View all the Fluent UI Web Components

Quickstart: Build web-based UX using components from the CDN

Article • 11/01/2021 • 2 minutes to read

A pre-bundled script with everything you need to start using the components is available on our Content Delivery Network (CDN). You can use it by adding script tag to your HTML markup with a `type="module"` attribute:

```
    <script type="module" src="https://unpkg.com/@fluentui/web-components">
</script>
```

Prerequisites

- Modern, standards-based browser (See browser compatibility table)
- Web server - Azure web app, StackBlitz, CodePen, etc.
  - Typescript pre-processor - while not necessary, most of our code examples will be in Typescript -- it's great to work with and we recommend it.

Import from the CDN

Add the a `script` tag to the head of your HTML file with a `type"="module"` attribute. This will include a pre-bundled script in your page with all the component APIs you need to use Fluent UI Web Components.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script type="module" src="https://unpkg.com/@fluentui/web-components">
</script>
  </head>
  <!-- ... -->
</html>
```

Add components

With the script in the head of your document, you can add any Fluent UI Web Component to your markup like you would any HTML tag:

```
<fluent-button>Hello world</fluent-button>
```

Next steps

If you want to install the Fluent UI Web Component packages in your web project, check the next article to learn how to set up your machine for development:

Install Fluent Web Component Packages

Tutorial: Install the Fluent UI Web Component packages

Article • 11/01/2021 • 2 minutes to read

In this tutorial, you learn how to:

- ✓ Install the packages with NPM or Yarn
- ✓ Register & use individual web components
- ✓ Register and use all the components

Prerequisites

Running locally on your machine, you should have:

- Node.js
- Node Package Manager (NPM) and/or Yarn
- A Typescript pre-processor - not strictly necessary but most of our examples are in TypeScript. *Recommended*

Install the Fluent UI Web Component packages with NPM or Yarn

To install the component packages using NPM:

```
npm install --save @fluentui/web-components
```

Or with yarn:

```
yarn add @fluentui/web-components
```

Register & use the web components

To use a Fluent UI Web Component as a custom element in HTML, the custom element must be registered for the browser to recognize it. The two steps to registering the

custom component are **Importing** the component definition into your JavaScript bundle, and **registering it with a `DesignSystem`**.

Here is a fluent Button getting imported and registered:

```
import { fluentButton, provideFluentDesignSystem } from '@fluentui/web-components';

provideFluentDesignSystem().register(fluentButton());
```

Register & use all web components

As a shortcut, if you wish to easily register all available components, rather than registering each one separately, you can use the following pattern with either design system:

```
import { allComponents, provideFluentDesignSystem } from '@fluentui/web-components';

provideFluentDesignSystem().register(allComponents);
```

> ⓘ Note
>
> When working with a tree-shaking bundler like **Webpack** or [Rollup]) (**https://rollupjs.org/guide/en/**), you will want to import and register the individual components, as in the first example above. This will ensure that any unused components are tree-shaken out of your builds.

> Looking to integrate with a front-end framework or bundler? Check out the integration docs.

Next steps

Learn how to customize the look of the components

Style your components

Styling the components

Article • 11/01/2021 • 8 minutes to read

The Fluent UI Web Components are designed to be stylistically flexible, allowing dramatic changes to visual design with minimal code changes. This is possible through the extensive use of Design Tokens and an adaptive color system.

Design Tokens

The following Design Tokens can be used to configure components stylistically.

Light and dark mode

The most common need for setting a token is to switch between light and dark mode.

- `baseLayerLuminance`: Set to `StandardLuminance.DarkMode` to switch into dark mode.

This is a decimal value, and the `LightMode` and `DarkMode` constants represent the standard points for light and dark mode. You could set it to any value `0` (black) to `1` (white) depending on your needs.

Layers and fill color

The second most common need for manually applying color is to define layers. When you adjust `baseLayerLuminance` as above, you're actually adjusting the `neutralLayer*` recipe colors.

- `fillColor`: Sets a value that *may* be applied to an element's styles and used as context for child color recipes. The default value is `neutralLayer1`. *Ex: Set to `neutralLayer2` for a 'lower' container, like beneath a Card or Accordion.*

Important: This token is easy to misuse and we're evaluating a more elegant solution for this common use case:

```
<div id="myCardContainer">
  <fluent-card>
    <fluent-button>Hello</fluent-button>
  </fluent-card>
  ...
</div>
```

```
const layer = document.getElementById('myCardContainer');
fillColor.setValueFor(layer, neutralLayer2);
```

```
#myCardContainer {
    background-color: ${fillColor};
}
```

The details: Avoid setting this to a fixed color value. The scenario above works because the neutral layer recipe colors *come from* the neutral palette. The `fillColor` token is used by most color recipes only as a *reference* for the *luminance* (or brightness) context. That's because the recipes are still drawing from their palette, and setting the `fillColor` does not *change* the palette.

Adjust neutral or accent colors

- `neutralBaseColor`: Set to a custom swatch to use for color recipes for layers and other neutral components.
  Ex: *`SwatchRGB.from(parseColorHexRGB('#A90000')!)`*
- `accentBaseColor`: Set to a custom swatch to use for color recipes for accent buttons, checkboxes, etc.

Typography

- `bodyFont`: Used to specify the font string to apply to components. Note that this does not import fonts, so they must either be web standard, assumed to be installed, or imported at the top of your app.

These tokens and values represent the default Fluent type ramp. The tokens should be used and adjusted relatively. For instance, if the type should be larger overall, increase the size of the entire type ramp instead of restyling a component to use "Plus 1" instead of "base".

Level	Font Size Token Name	Line Height Token Name
Minus 2 (smallest)	`typeRampMinus2FontSize`	`typeRampMinus2LineHeight`
Minus 1	`typeRampMinus1FontSize`	`typeRampMinus1LineHeight`

Level	Font Size Token Name	Line Height Token Name
Base (body)	`typeRampBaseFontSize`	`typeRampBaseLineHeight`
Plus 1	`typeRampPlus1FontSize`	`typeRampPlus1LineHeight`
Plus 2	`typeRampPlus2FontSize`	`typeRampPlus2LineHeight`
Plus 3	`typeRampPlus3FontSize`	`typeRampPlus3LineHeight`
Plus 4	`typeRampPlus4FontSize`	`typeRampPlus4LineHeight`
Plus 5	`typeRampPlus5FontSize`	`typeRampPlus5LineHeight`
Plus 6 (largest)	`typeRampPlus6FontSize`	`typeRampPlus6LineHeight`

Sizing

Here are the common sizing tokens you may want to adjust:

- `controlCornerRadius`: Sets the corner radius used by controls with backplates.
  Ex: Increase to `6px` for slightly rounder buttons and text fields.
- `layerCornerRadius`: Sets the corner radius used layers like cards, flyouts, and dialogs.
  Ex: Increase to `20px` for very round cards.
- `density` (in process): A modifier used with sizing tokens `baseHeightMultiplier` and `baseHorizontalSpacingMultiplier`.
  Ex: Set to `1` to increase control size or `-1` to decrease.

These are less common and more nuanced:

- `baseHeightMultiplier`: This value, multiplied by `designUnit`, sets the base height of most controls. Works with adaptive `density` values.
- `baseHorizontalSpacingMultiplier` (future): This value, multiplied by `designUnit`, sets the internal horizontal padding of most controls. Works with adaptive `density` values.
- `designUnit`: The unit size of the design grid. Used to calculate height and spacing sizes for controls.

Miscellaneous

Common:

- `direction`: The primary document direction (LTR or RTL).

Less common:

- `strokeWidth`: Controls the width of the stroke of a component that has a stroke.
- `focusStrokeWidth`: Controls with width of the stroke of a component that has a stroke when it has document focus.
- `disabledOpacity`: The opacity of disabled controls. Careful with values that are too high as the control may no longer look disabled. There are no contrast requirements for a disabled control.

Adaptive color system

The design tokens are built around an adaptive color system that provides some unique advantages:

- Ensure text meets WCAG contrast requirements.
- Easily swap from light mode to dark, or anywhere in-between.
- Color theming through palette tinting.
- Perceptually uniform UI across background colors.

To accomplish these goals, the web components make heavy use of algorithmic colors called Recipes. Recipes are a combination of an algorithm and input values that produce a desired result. Just as you can bake different types of cookies with different combinations of sugar, butter, flour, and salt, you can produce different design system treatments by altering recipe values (measurements) or algorithms (instructions).

The current base recipes are closely related to their algorithm, but that's a convention and not a requirement. What follows is a list of the algorithms, which function on like-named values. For instance, `accentFill` relies on `accentFillRestDelta`, `accentFillHoverDelta`, `accentFillActiveDelta`, and `accentFillFocusDelta`.

Recipes are currently used for color values, but they are not limited to that and their usage will be expanded soon.

To better visualize how this works, the FAST team built an application specifically for exploring the system. Check out the Color Explorer.

Common functionality

Most color recipes are based on a `palette`. Currently there is built-in support for `accent` and `neutral` palettes.

Most color recipes take a `reference Swatch`. This is a core concept of Adaptive UI which allows the recipes to vary based on the containing component's color. For instance, supporting a button with consistent treatment between light and dark modes is done with a single recipe.

Many recipes are "stateful", meaning they support rest, hover, active, and focus states for a component.

"**Fill**" means the recipe is intended to fill a larger area, commonly like a component backplate.

"**Foreground**" means the recipe is intended for text, icons, or other lightweight decorations where you need or want to meet contrast requirements.

"**Stroke**" means the recipe is intended for lines, either outline or divider.

Accent algorithms

accentFill

Stateful.

Relies on `textColor` and `contrastTarget` to find the closest colors from the supplied palette that can be used for component states. For instance, colors needed to support white text and a 14px font (which requires 4.5:1 contrast).

accentForeground

Stateful.

Commonly for link text or icon. Also for smaller elements that might not show up well using `accentFill`, for instance if your accent color is dark purple and you support a dark mode interface.

Like `accentFill` this relies on `textColor` and `contrastTarget` to find the closest colors from the supplied palette that can be used for component states.

foregroundOnAccent

Not stateful.

Technically this doesn't *use* the accent palette, but it's designed to be used *over* the accent palette. This algorithm simply returns black or white based on the provided

`contrastTarget`. It returns white if possible, as a common treatment for an accent button is white text over the accent color.

Neutral algorithms

neutralDivider

Not stateful.

Used for decorative dividers that do not need to meet contrast requirements.

neutralFill

Stateful.

The most basic fill used for buttons or other components.

neutralFillContrast

Stateful.

Often Used as a selected state or anywhere you want to draw attention. Meets contrast requirements with the containing background.

neutralFillInput

Stateful.

Another basic fill, applied to input elements to allow easy differentiation from other components like buttons.

neutralFillStealth

Stateful.

More subtle than `neutralFill` in that the resting state is transparent. Often used for low-priority features to draw less attention.

neutralForeground

Not stateful.

Most common recipe, used for plain text or icons.

neutralForegroundHint

Not stateful.

Used for subtle text. Meets 4.5:1 minimum contrast requirement.

neutralStroke

Stateful.

Used for strong outline, either alone or with a fill.

Layers

The layer recipes are used for different sections of an app or site. They are designed to be able to stack, but that is not required. When stacked in sequence, the layers will lighten on top of each other.

The key feature of layering is to support the primary container color for light or dark mode. This produces absolute colors based on the `baseLayerLuminance` value, which sets the luminance for layer one. This is any value between 0 for black or 1 for white.

The difference between each layer is defined with `neutralFillLayerRestDelta`.

Layers are not stateful.

neutralFillLayer

The only layer recipe that's relative to the container color instead of absolute. The most common example of this is a Card, which will be one layer color lighter than its container.

neutralLayer1, neutralLayer2, neutralLayer3, and neutralLayer4

Absolute layer colors derived from and starting at `baseLayerLuminance`. Layer one is lightest and the values darken as the layer number increases.

neutralLayerCardContainer

A special layer to support experiences primarily built with cards, especially in light mode, so cards can be white and the container color can be one layer darker.

neutralLayerFloating

A special layer for floating layers, like flyouts or menus. It will be lighter than any other layers if possible, but will also be white in default light mode, as will neutral layer one.

Adaptive Color "Don'ts"

The adaptive color system lives entirely in JavaScript, emitting CSS custom properties for styling purposes where appropriate. This means that you should consider the CSS custom properties emitted by color Design Tokens to be immutable. If you declare the CSS custom property in CSS, the adaptive Color System is unable to know that has happened and components will render with incorrect colors, which can lead to accessibility issues. If you need to change the values for those CSS custom properties, set the value using the DesignToken.setValueFor() API.

Next steps

- More about design tokens
- High contrast mode
- Integrating components into other frameworks
- See the components

Web components overview

Article • 11/01/2021 • 2 minutes to read

This section provides a live catalog of components to help you understand the full set of Fluent UI Web Components and their features.

The @fluentui/web-components library are web components built on top of Microsoft's web component and design system foundation, FAST. The Fluent UI Web Components are built using FAST's @microsoft/fast-foundation and @microsoft/fast-element libraries in a way expresses Microsoft's Fluent design language. The FAST libraries are referred to frequently in the code.

You can check out the code for the components in the GitHub repo for Web Components, part of the Fluent UI monorepo

Open source communities

The Fluent UI Web Components and its underlying frameworks (fast-foundation, fast-element) are open source projects with communities which you can join for help, issues, contributing code, or asking questions.

The Fluent UI Web Components is part of the whole Fluent UI web community. You can submit requests and issues on GitHub

Join our FAST Discord for more engagement -- there's a Fluent UI section where you can join the discussion.

More on the FAST community project

Next steps

- See the first component - accordion
- Customize the component styling
- See the integrations

Accordion

Article • 07/20/2022 • 2 minutes to read

As defined by the W3C:

> An accordion is a vertically stacked set of interactive headings that each contain a title, content snippet, or thumbnail representing a section of content. The headings function as controls that enable users to reveal or hide their associated sections of content. Accordions are commonly used to reduce the need to scroll when presenting multiple sections of content on a single page.

fluent-accordion

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentAccordion,
    fluentAccordionItem
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentAccordion(),
        fluentAccordionItem()
    );
```

Example

HTML CSS TypeScript Result EDIT ON CODEPEN

LIVE

```
<h4>Multi-expand</h4>
<fluent-accordion style="max-
width: 350px;">
  <fluent-accordion-item>
    <span slot="heading">Item
1</span>
    <div class="panel">
      Panel one content
    </div>
  </fluent-accordion-item>
  <fluent-accordion-item>
    <span slot="heading">Item
2</span>
    <div class="panel">
      Panel 2 content
    </div>
  </fluent-accordion-item>
  <fluent-accordion-item>
    <span slot="heading">Item
3</span>
    <div class="panel">
      Panel 3 content
    </div>
  </fluent-accordion-item>
</fluent-accordion>
<h4>Single-expand</h4>
<fluent-accordion expand-
mode="single">
  <fluent-accordion-item>
    <span slot="heading">Item
1</span>
    <div class="panel">
      Panel one content
    </div>
  </fluent-accordion-item>
  <fluent-accordion-item>
    <span slot="heading">Item
2</span>
    <div class="panel">
      Panel 2 content
    </div>
  </fluent-accordion-item>
  <fluent-accordion-item>
    <span slot="heading">Item
3</span>
    <div class="panel">
      Panel 3 content
    </div>
  </fluent-accordion-item>
```

Multi-expand

Item 1
Item 2
Item 3

Single-expand

Item 1
Item 2
Item 3
Item 4

Resources 1× 0.5× 0.25× Rerun

Additional Resources

- Code in GitHub
- W3C Component Aria Practices

Anchor

Article • 07/20/2022 • 2 minutes to read

`fluent-anchor` is a web component implementation of an HTML anchor element. The fluent-components anchor supports the same visual appearances as the button component (accent, lightweight, neutral, outline, stealth) as well as a hypertext appearance for use inline with text.

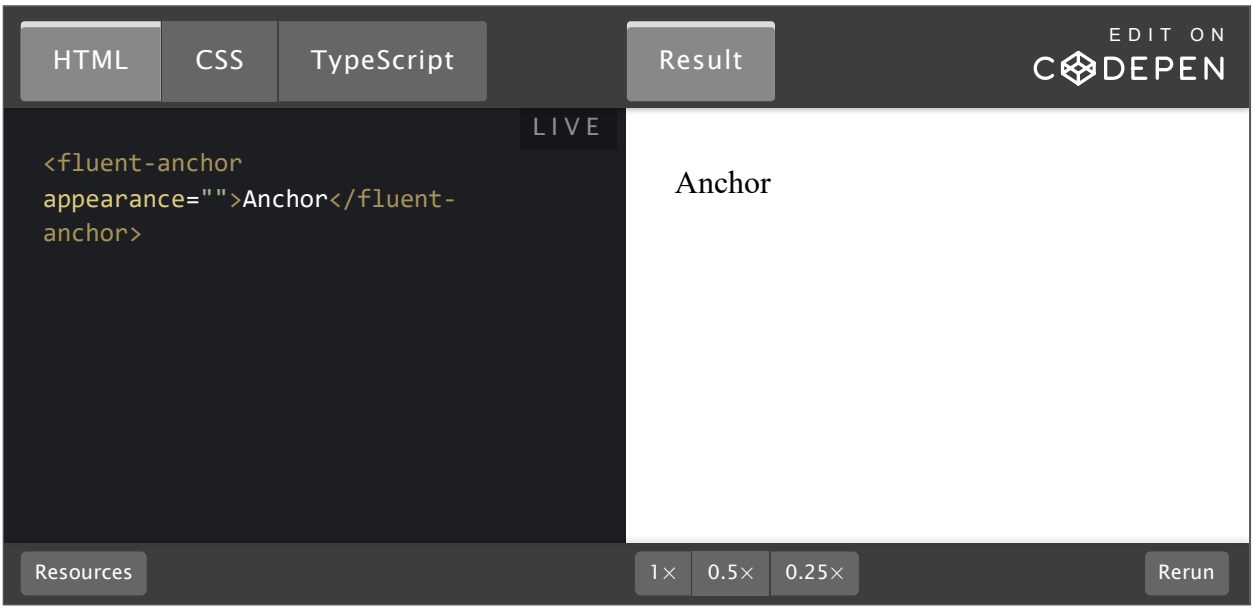
fluent-anchor

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentAnchor
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentAnchor()
    );
```

Example



> **Note**

`fluent-anchor` is built with the expectation that focus is delegated to the anchor element rendered into the shadow DOM.

Additional Resources

Code in GitHub W3C Component Aria Practices

Anchored Region

Article • 07/20/2022 • 2 minutes to read

An anchored region is a container component which enables authors to create layouts where the contents of the anchored region can be positioned relative to another "anchor" element. Additionally, the anchored region can react to the available space between the anchor and a parent "viewport" element such that the region is placed on the side of the anchor with the most available space, or even resize itself based on that space.

fluent-anchored-region

Setup

TypeScript

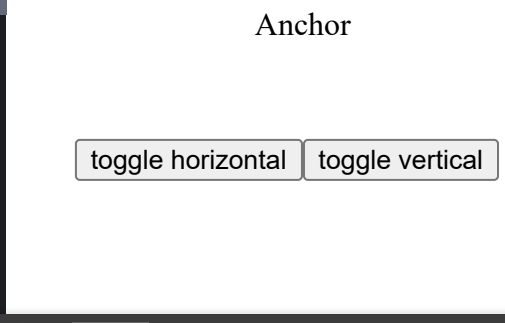
```
import {
    provideFluentDesignSystem,
    fluentAnchoredRegion
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentAnchoredRegion()
    );
```

Example

HTML CSS TypeScript Result EDIT ON CODEPEN

LIVE

```
<fluent-button
appearance="outline" id="anchor-
toggle-positions" style="margin-
left: 100px; margin-top: 100px">
  Anchor</fluent-button>

<fluent-anchored-region
id="toggle-positions-region"
anchor="anchor-toggle-positions"
vertical-positioning-
mode="locktodefault" vertical-
default-position="top"
horizontal-positioning-
mode="locktodefault" horizontal-
default-position="right">
```



Resources 1× 0.5× 0.25× Rerun

Badge

Article • 07/20/2022 • 2 minutes to read

The `fluent-badge` is used to highlight an item, attract attention or flag status.

fluent-badge

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentBadge
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentBadge()
    );
```

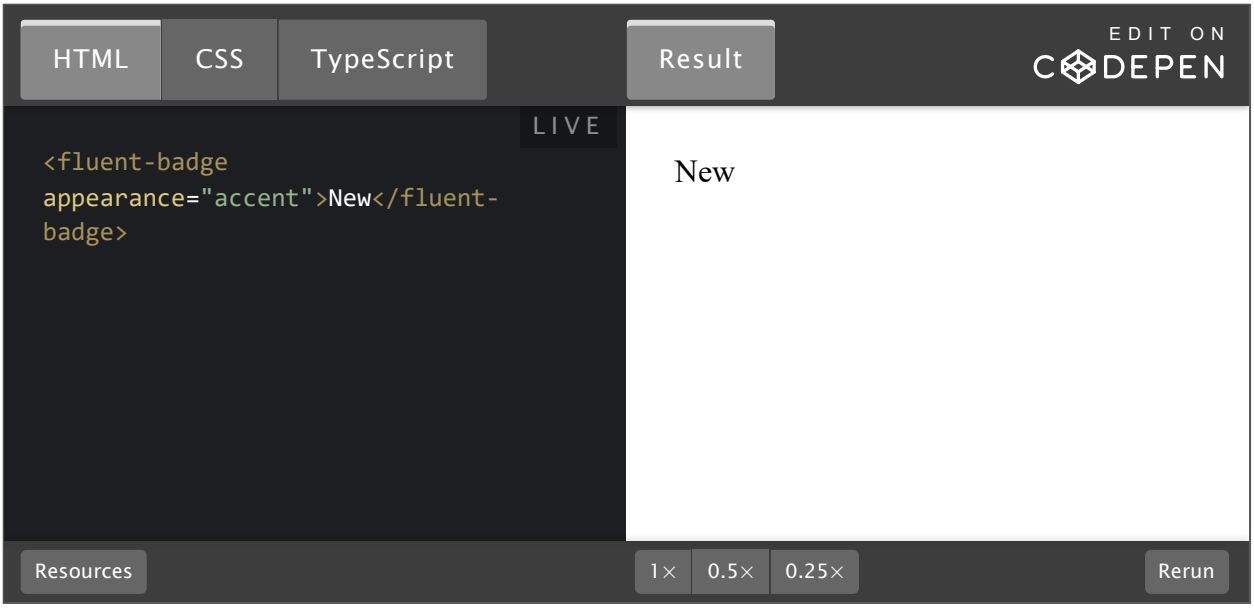
Usage

The `fill` and `color` attributes of the badge create CSS custom properties which can be used to style the control

css

```
fluent-badge {
    --badge-fill-primary: #00FF00;
    --badge-fill-danger: #FF0000;
    --badge-color-light: #FFFFFF;
    --badge-color-dark: #000000;
}
```

Example



Additional Resources

Code in GitHub W3C Component Aria Practices

Breadcrumb

Article • 07/20/2022 • 2 minutes to read

As defined by the W3C:

> A breadcrumb trail consists of a list of links to the parent pages of the current page in hierarchical order. It helps users find their place within a website or web application. Breadcrumbs are often placed horizontally before a page's main content.

fluent-breadcrumb

Setup

Basic Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentBreadcrumb,
    fluentBreadcrumbItem
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentBreadcrumb(),
        fluentBreadcrumbItem()
    );
```

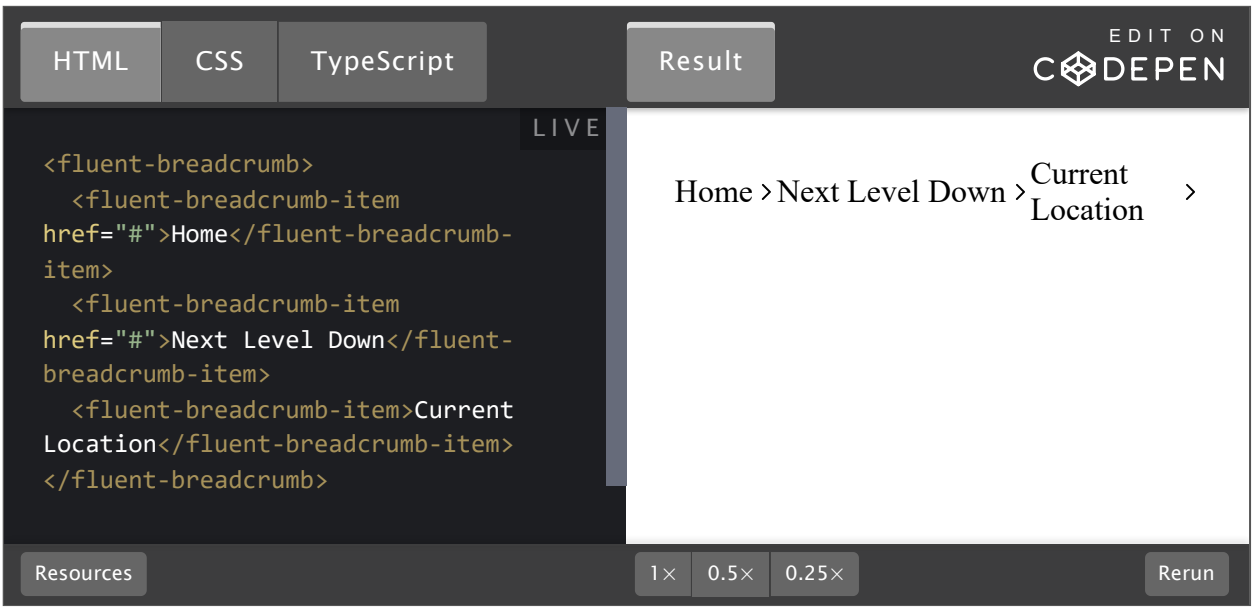
Custom Separator

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentBreadcrumb,
    fluentBreadcrumbItem
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentBreadcrumb(),
```

```
        fluentBreadcrumbItem({
            separator: " -> "
        })
    );
```

Example



> **Note**
>
> This component is built with the expectation that focus is delegated to the anchor element rendered into the shadow DOM.

Button

Article • 07/20/2022 • 2 minutes to read

The fluent-button is a web component implementation of an HTML button element. The fluent-components button supports several visual appearances (accent, lightweight, neutral, outline, stealth).

fluent-button

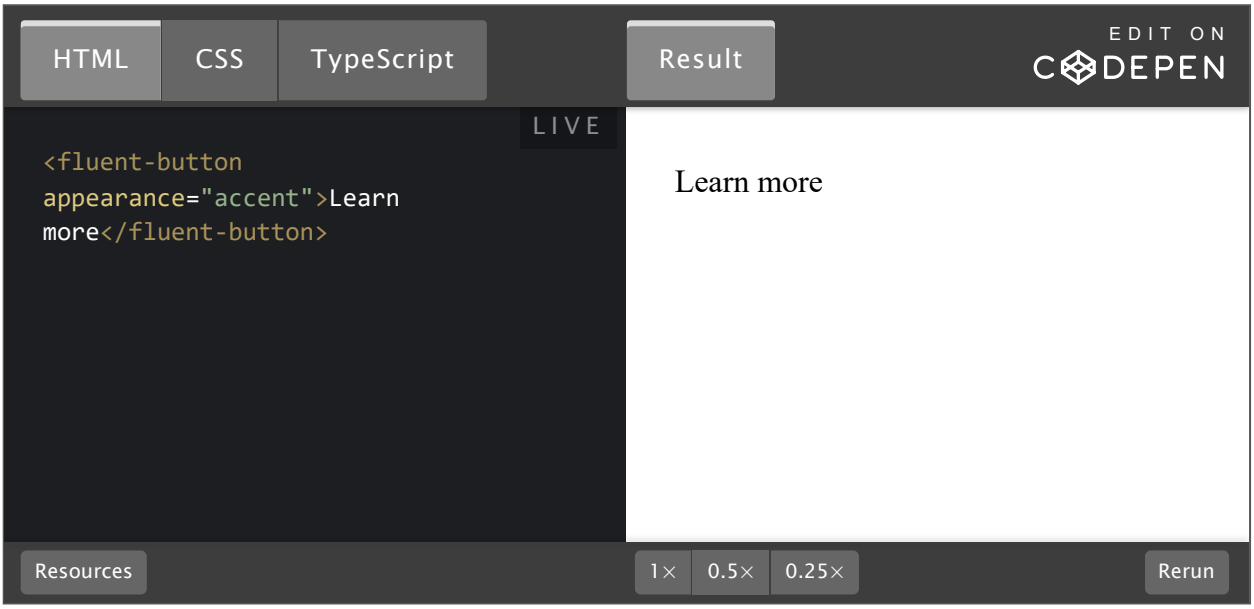
Setup

Basic Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentButton
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentButton()
    );
```

Example



Additional Resources

Code in GitHub W3C Component Aria Practices

Card

Article • 07/20/2022 • 2 minutes to read

The fluent-card is a visual container without semantics that takes children. Cards are snapshots of content that are typically used in a group to present collections of related information.

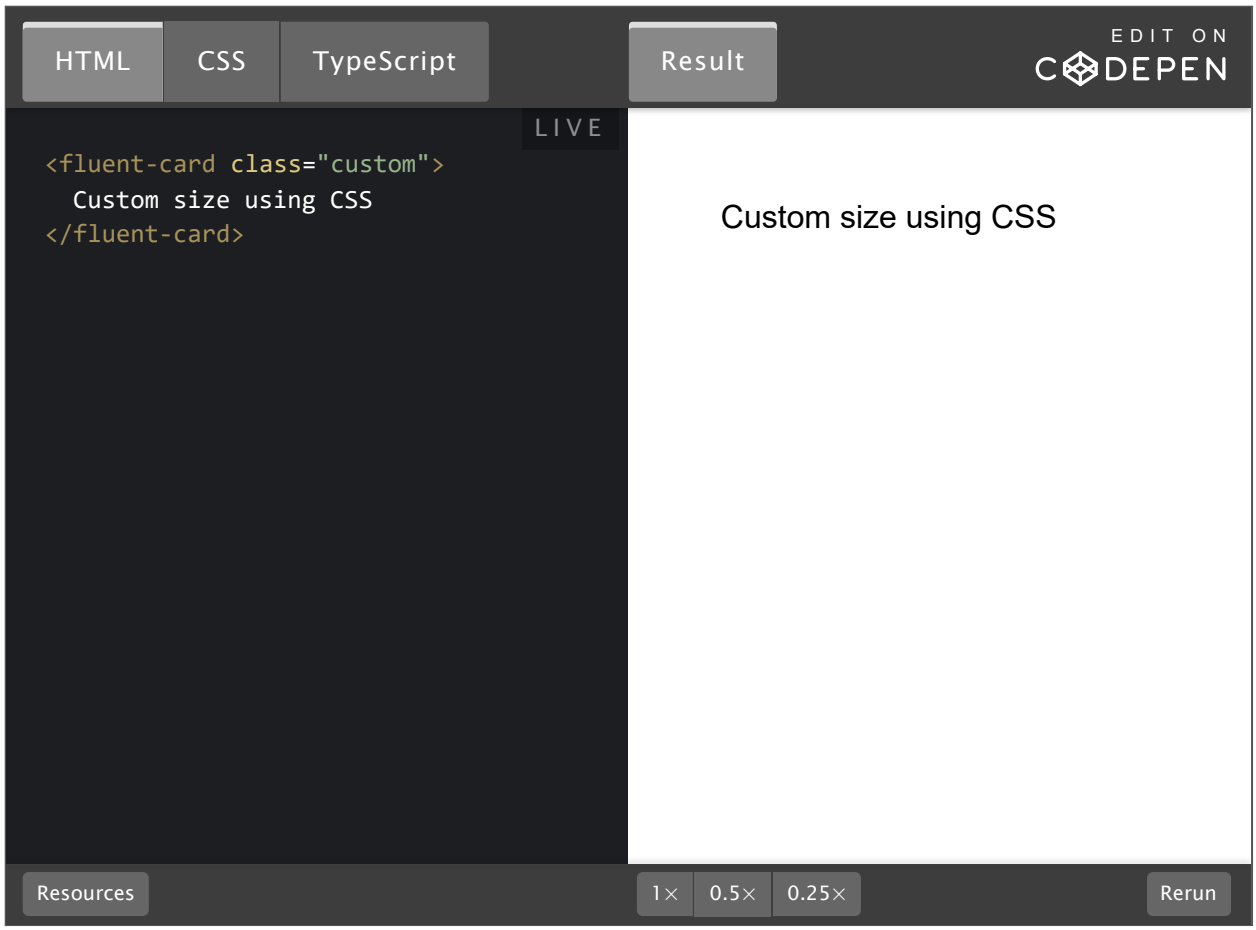
fluent-card

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentCard
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentCard()
    );
```

Example


HTML
CSS
TypeScript
Result
EDIT ON
CODEPEN
LIVE
<fluent-card class="custom">
Custom size using CSS
</fluent-card>
Custom size using CSS
Resources
1×
0.5×
0.25×
Rerun

Checkbox

Article • 07/20/2022 • 2 minutes to read

A implementation of a checkbox as a form-connected web-component.

fluent-checkbox

Setup

Basic Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentCheckbox
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentCheckbox()
    );
```

Customizing the check indicator

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentCheckbox
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentCheckbox({
            checkedIndicator: `...your checked indicator...`,
            indeterminateIndicator: `...your indeterminate indicator...`,
        })
    );
```

Example

HTML CSS TypeScript Result EDIT ON CODEPEN

LIVE

```
<fluent-checkbox>Did you check
this?</fluent-checkbox><br />
<fluent-checkbox checked
disabled>Is this disabled?</fluent-
checkbox><br />
<fluent-checkbox checked>Checked by
default?</fluent-checkbox>
```

Resources 1× 0.5× 0.25× Rerun

Combobox

Article • 07/20/2022 • 2 minutes to read

The W3C definition:

> A combobox is an input widget with an associated popup that enables users to select a value for the combobox from a collection of possible values. In some implementations, the popup presents allowed values, while in other implementations, the popup presents suggested values, and users may either select one of the suggestions or type a value. The popup may be a listbox, grid, tree, or dialog. Many implementations also include a third optional element -- a graphical Open button adjacent to the combobox, which indicates availability of the popup. Activating the Open button displays the popup if suggestions are available.

fluent-combobox

Setup

Basic Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentCombobox,
    fluentOption
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentCombobox(),
        fluentOption()
    );
```

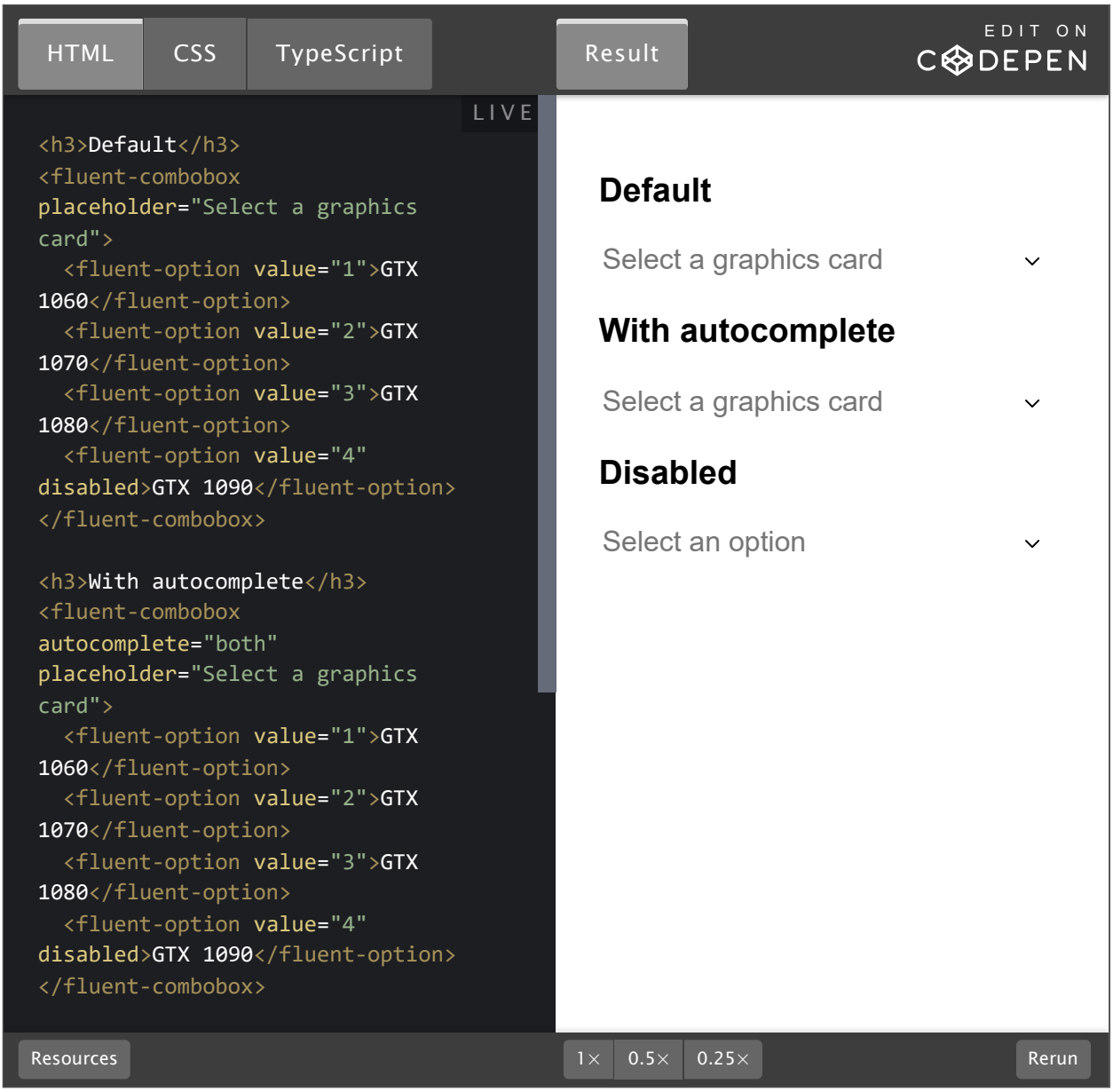
Customizing the indicator

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentCombobox,
    fluentOption
```

```
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentCombobox({
          indicator: `...your indicator...`
        }),
        fluentOption()
    );
```

Example



Data Grid

Article • 07/20/2022 • 2 minutes to read

The fluent-data-grid component is used to display tabular data. The fluent-data-grid-row and fluent-data-grid-cell components are typically created programmatically by the parent grid but some authors may find it useful to create them manually.

fluent-data-grid

Setup

TypeScript

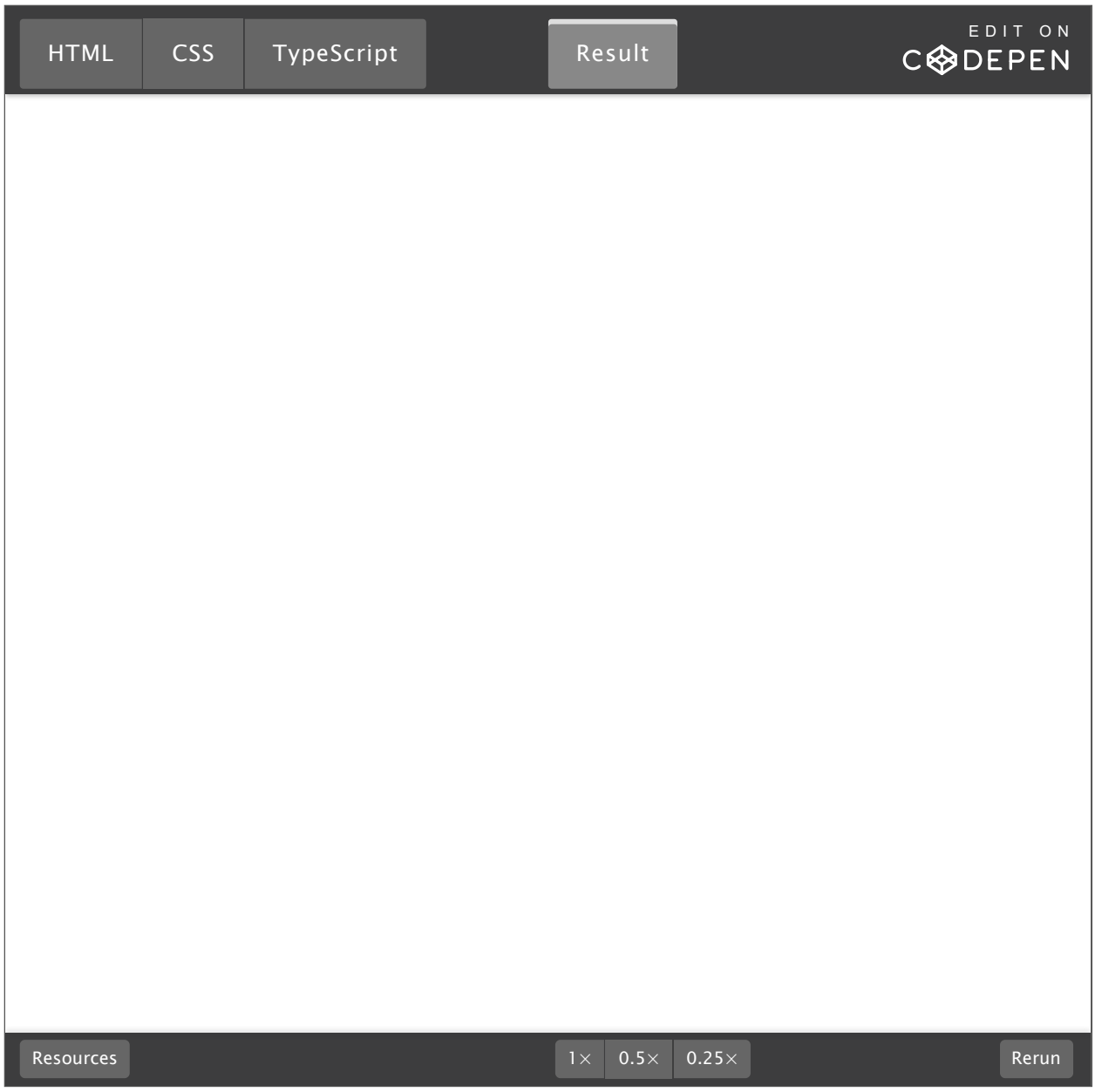
```
import {
    provideFluentDesignSystem,
    fluentDataGridCell,
    fluentDataGridRow,
    fluentDataGrid
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentDataGridCell(),
        fluentDataGridRow(),
        fluentDataGrid()
    );
```

Example

Note: data must be provided to the grid via a property.

TypeScript

```
document.getElementById("samplegrid").rowsData = [
    { item1: "value 1-1", item2: "value 2-1" },
    { item1: "value 1-2", item2: "value 2-2" },
    { item1: "value 1-3", item2: "value 2-3" },
];
```


HTML
CSS
TypeScript
Result
EDIT ON
CODEPEN
Resources
1×
0.5×
0.25×
Rerun

Dialog

Article • 07/20/2022 • 2 minutes to read

As defined by the W3C:

> A dialog is a window overlaid on either the primary window or another dialog window. Windows under a modal dialog are inert. That is, users cannot interact with content outside an active dialog window. Inert content outside an active dialog is typically visually obscured or dimmed so it is difficult to discern, and in some implementations, attempts to interact with the inert content cause the dialog to close.

> Like non-modal dialogs, modal dialogs contain their tab sequence. That is, Tab and Shift + Tab do not move focus outside the dialog. However, unlike most non-modal dialogs, modal dialogs do not provide means for moving keyboard focus outside the dialog window without closing the dialog.

fluent-dialog

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentDialog
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentDialog()
    );
```

Example

Dialog

Show Dialog

Divider

Article • 07/20/2022 • 2 minutes to read

As the name implies, it divides one section of content from another with a line. A web component implementation of a horizontal rule.

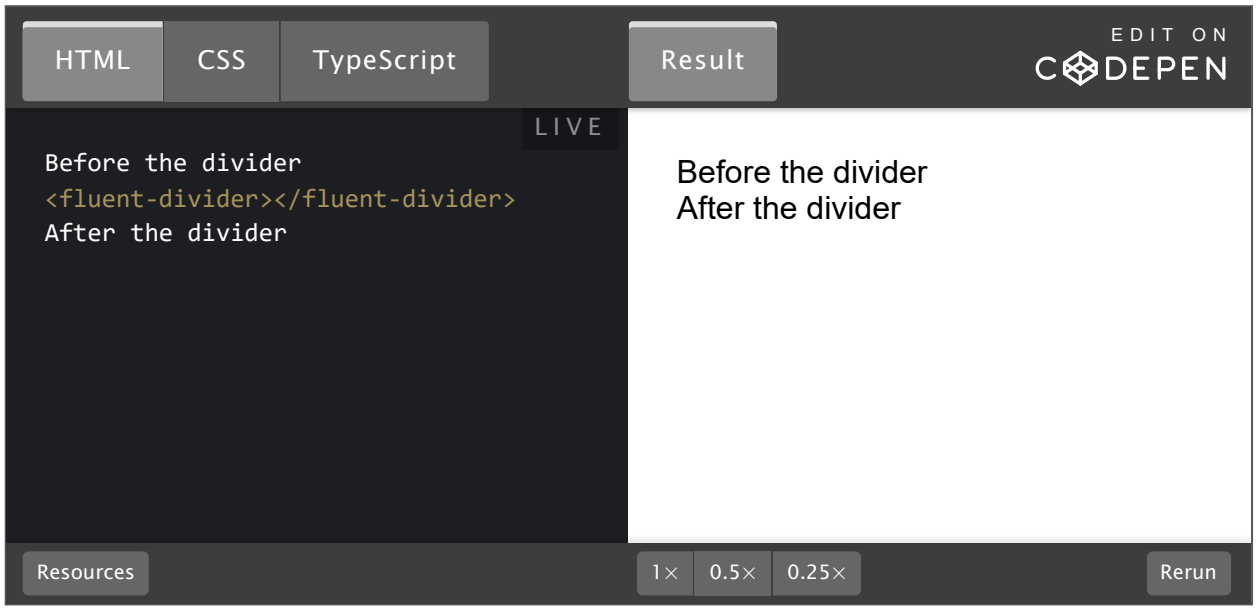
fluent-divider

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentDivider
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentDivider()
    );
```

Example



Customize the design

TypeScript

```
import { Divider, dividerTemplate as template } from "@microsoft/fast-
foundation";
import { dividerStyles as styles } from "./my-divider.styles";

export const myDivider = Divider.compose({
    baseName: "divider",
    template,
    styles,
});
```

Flipper

Article • 07/20/2022 • 2 minutes to read

The flipper is most often used to page through blocks of content or collections of ui elements. As flippers are often a supplemental form of navigation, they are often hidden by default to avoid duplicate keyboard interaction. Passing an attribute of aria-hidden="false" will expose the flippers to assistive technology.

fluent-flipper

Setup

TypeScript

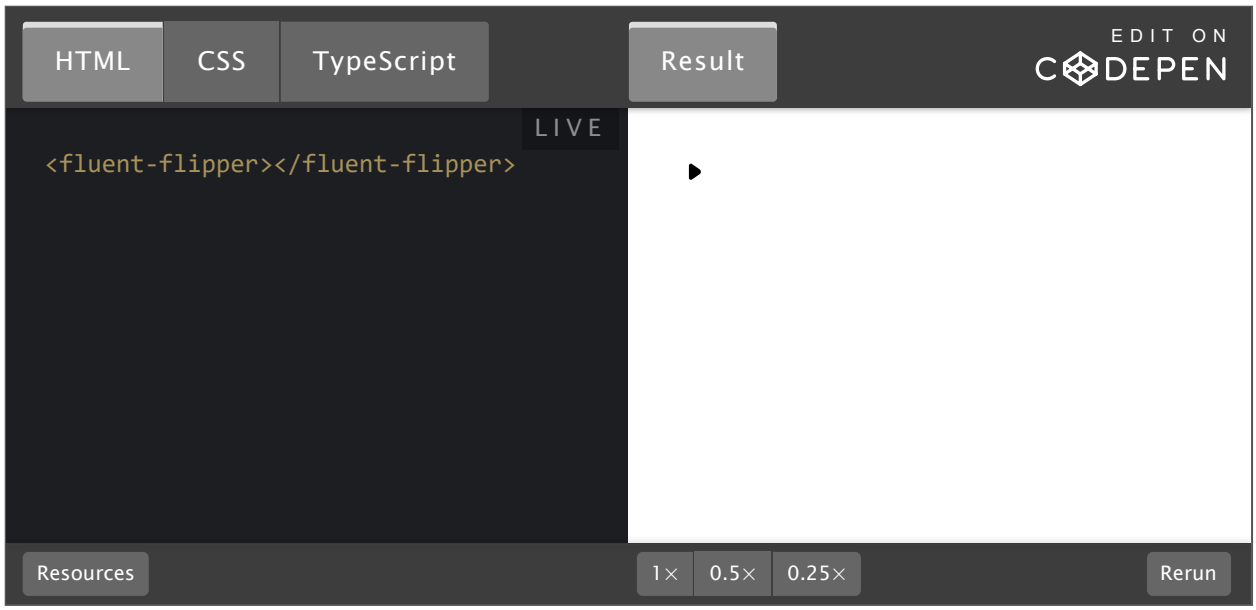
```
import {
    provideFluentDesignSystem,
    fluentFlipper
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentFlipper()
    );
```

Customize the icons

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentFlipper
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentFlipper({
            next: `...your next icon...`,
            previous: `...your previous icon...`,
        })
    );
```

Example


HTML
CSS
TypeScript
Result
EDIT ON
CODEPEN
LIVE
<fluent-flipper></fluent-flipper>
Resources
1×
0.5×
0.25×
Rerun

Horizontal Scroll

Article • 07/20/2022 • 2 minutes to read

An implementation of a content scroller as a web-component.

fluent-horizontal-scroll

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentHorizontalScroll,
    fluentFlipper
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentHorizontalScroll(),
        fluentFlipper()
    );
```

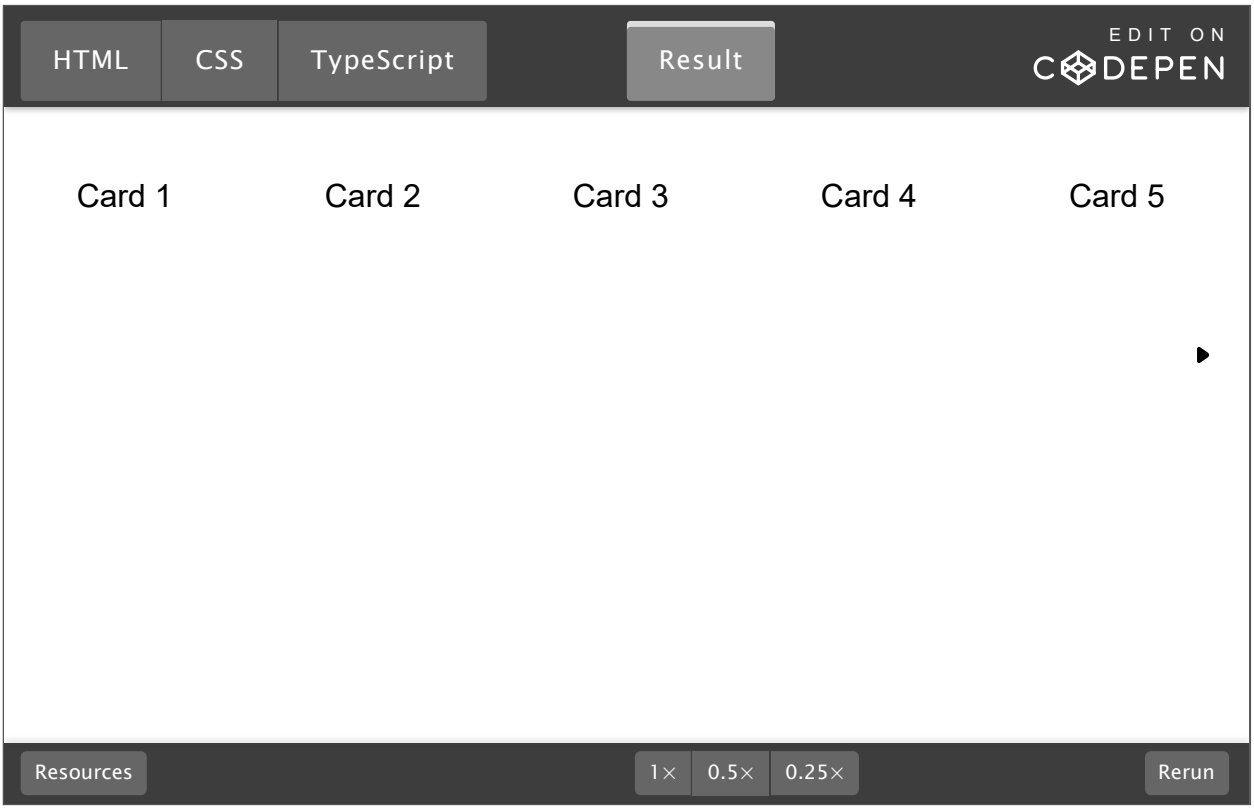
Customizing flippers

TypeScript

```
import { html } from "@microsoft/fast-element";
import {
    provideFluentDesignSystem,
    fluentHorizontalScroll
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentHorizontalScroll({
            nextFlipper: html<HorizontalScroll>`
                <fluent-flipper
                    @click="${x => x.scrollToNext()}"
                    aria-hidden="${x => x.flippersHiddenFromAT}"
                ></fluent-flipper>
            `,
            previousFlipper: html<HorizontalScroll>`
                <fluent-flipper
                    @click="${x => x.scrollToPrevious()}"
```

```
                    direction="previous"
                    aria-hidden="${x => x.flippersHiddenFromAT}"
                ></fluent-flipper>
            `
        })
    );
```

Example



Listbox

Article • 07/20/2022 • 2 minutes to read

An implementation of a listbox. While any DOM content is permissible as a child of the listbox, only fluent-option elements, `option elements`, and slotted items with `role="option"` will be treated as options and receive keyboard support.

The `fluent-listbox` component has no internals related to form association. For a form-associated listbox, see the fluent-select component.

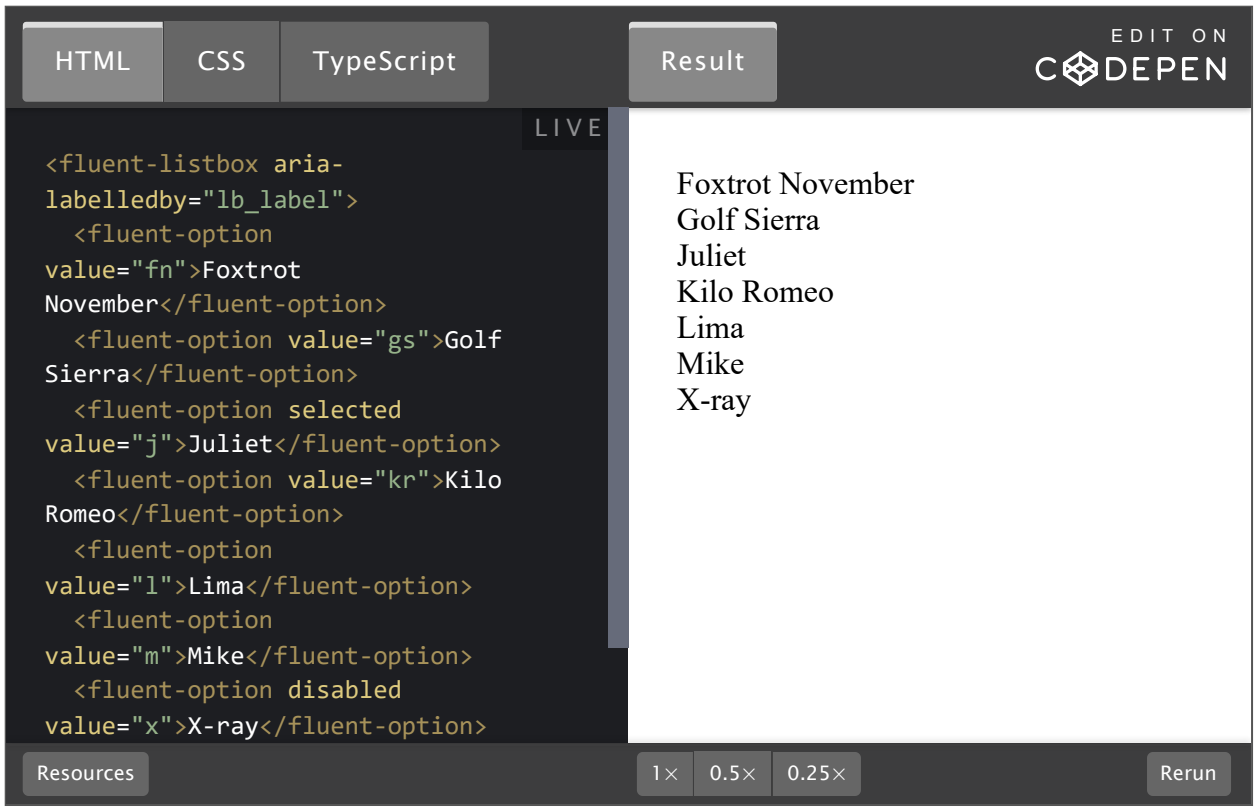
fluent-listbox

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentListbox,
    fluentOption
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentListbox(),
        fluentOption()
    );
```

Example



Additional Resources

Code in GitHub W3C Component Aria Practices

Menu

Article • 07/20/2022 • 2 minutes to read

As defined by the W3C:

> A menu is a widget that offers a list of choices to the user, such as a set of actions or functions. Menu widgets behave like native operating system menus, such as the menus that pull down from the menubars commonly found at the top of many desktop application windows. A menu is usually opened, or made visible, by activating a menu button, choosing an item in a menu that opens a sub menu, or by invoking a command, such as Shift + F10 in Windows, that opens a context specific menu. When a user activates a choice in a menu, the menu usually closes unless the choice opened a submenu.

While any DOM content is permissible as a child of the menu, only fluent-menu-item's and slotted content with a role of menuitem, menuitemcheckbox, or menuitemradio will receive keyboard support.

fluent-menu applies fluent-menu-item's startColumnCount property based on an evaluation of all of the fluent-menu-items so the content text vertically aligns across all fluent-menu-items. If any fluent-menu-item does not have a roll of checkbox or radio or the start slot is not passed, startColumnCount is set to 0 which applies a indent-0 class to all the fluent-menu-items. If any fluent-menu-item has a roll of checkbox or radio or the start slot exists, startColumnCount is set to 1 which applies a indent-1 class to all the fluent-menu-items. Or if any fluent-menu-item has a roll of checkbox or radio and the start slot exists, startColumnCount is set to 2 which applies a indent-2 class to all the fluent-menu-items.

fluent-menu

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentMenu,
    fluentMenuItem
} from "@fluentui/web-components";

provideFluentDesignSystem()
```

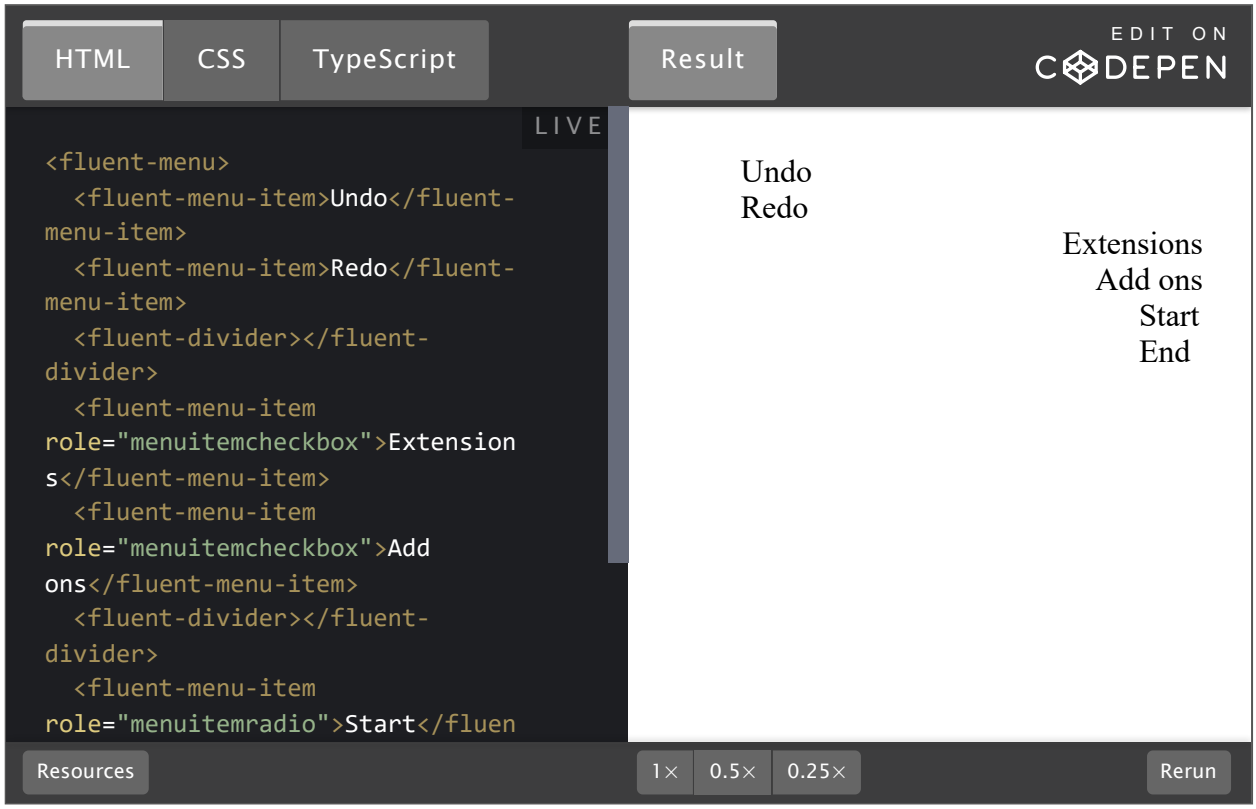
```
    .register(
        fluentMenu(),
        fluentMenuItem()
    );
```

Customizing Indicators and Glyphs

TypeScript

```typescript
import {
    provideFluentDesignSystem,
    fluentMenu,
    fluentMenuItem
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentMenu(),
        fluentMenuItem({
            expandCollapseGlyph: `...your expand/collapse glyph...`,
            checkboxIndicator: `...your checkbox indicator...`,
            radioIndicator: `...your radio indicator...`,
        })
    );
```

Example



Additional Resources

Code in GitHub W3C Component Aria Practices

Number Field

Article • 07/20/2022 • 2 minutes to read

An implementation of a text field as a form-connected web-component. The fluent-number-field supports two visual appearances, outline and filled, with the control defaulting to the outline appearance.

fluent-number-field

Setup

TypeScript

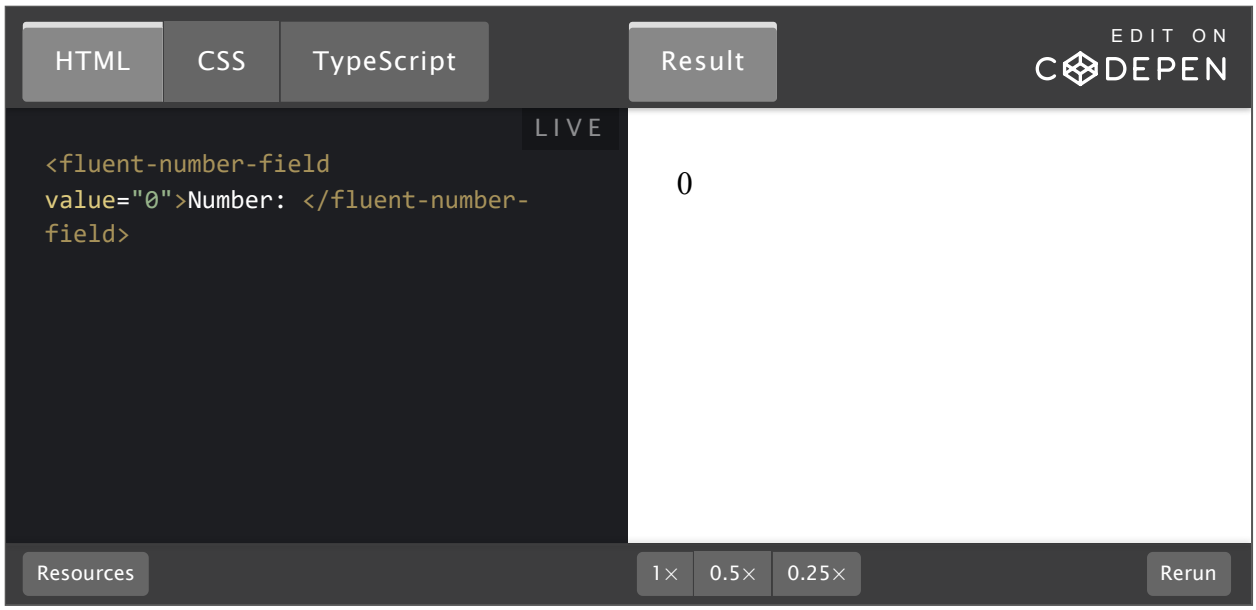
```
import {
    provideFluentDesignSystem,
    fluentNumberField
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentNumberField()
    );
```

Customizing glyphs

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentNumberField
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentNumberField({
            stepDownGlyph: `...your step down glyph...`,
            stepUpGlyph: `...your setup up glyph...`,
        })
    );
```

Example



Note

This component is built with the expectation that focus is delegated to the input element rendered into the shadow DOM.

Progress

Article • 07/20/2022 • 2 minutes to read

Progress and progress ring are used to display the length of time a process will take or to visualize percentage value (referred to as a determinate state) and to represent an unspecified wait time (referred to as an indeterminate state). Progress components are typically visually represented by a circular or linear animation. When the value attribute is passed the state is determinate, otherwise it is indeterminate.

For progress components which have a linear visual appearance, use fluent-progress. For progress implementations which are circular, use fluent-progress-ring.

fluent-progress

Setup

TypeScript

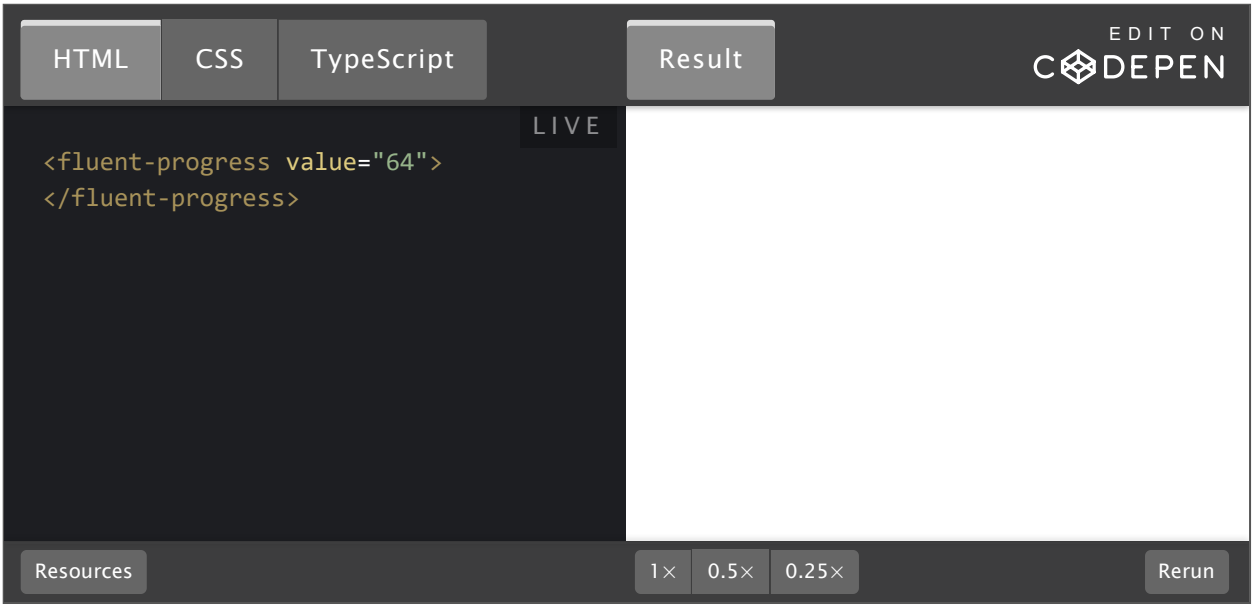
```
import {
    provideFluentDesignSystem,
    fluentProgress,
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentProgress(),
    );
```

Customizing indicators

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentProgress,
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentProgress({
            indeterminateIndicator1: `...your indeterminate indicator...`,
            indeterminateIndicator2: `...your indeterminate indicator...`
        }),
    );
```

Example



Additional Resources

See also progress-ring W3C Component Aria Practices

Progress Ring

Article • 07/20/2022 • 2 minutes to read

Progress and progress ring are used to display the length of time a process will take or to visualize percentage value (referred to as a determinate state) and to represent an unspecified wait time (referred to as an indeterminate state). Progress components are typically visually represented by a circular or linear animation. When the value attribute is passed the state is determinate, otherwise it is indeterminate.

For progress components which have a linear visual appearance, use fluent-progress. For progress implementations which are circular, use fluent-progress-ring.

fluent-progress-ring

Setup

TypeScript

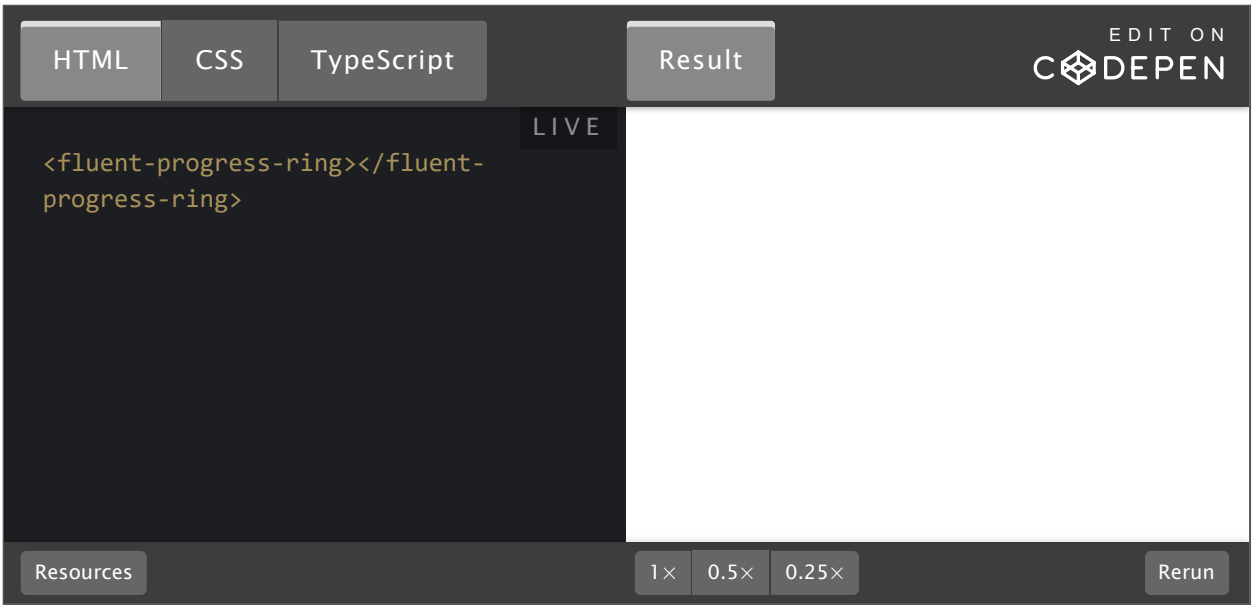
```
import {
    provideFluentDesignSystem,
    fluentProgressRing
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentProgressRing()
    );
```

Customizing indicators

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentProgressRing
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentProgressRing({
            indeterminateIndicator: `...your indeterminate indicator...`
        })
    );
```

Example



Additional Resources

See also progress W3C Component Aria Practices

Radio and Radio Group

Article • 07/20/2022 • 2 minutes to read

As defined by the W3C:

A radio group is a set of checkable buttons, known as radio buttons, where no more than one of the buttons can be checked at a time. Some implementations may initialize the set with all buttons in the unchecked state in order to force the user to check one of the buttons before moving past a certain point in the workflow.

While any DOM content is permissible as a child of the radiogroup, only fluent-radio's and slotted content with a role of radio will receive keyboard support.

fluent-radio and fluent-radio-group

Setup

TypeScript

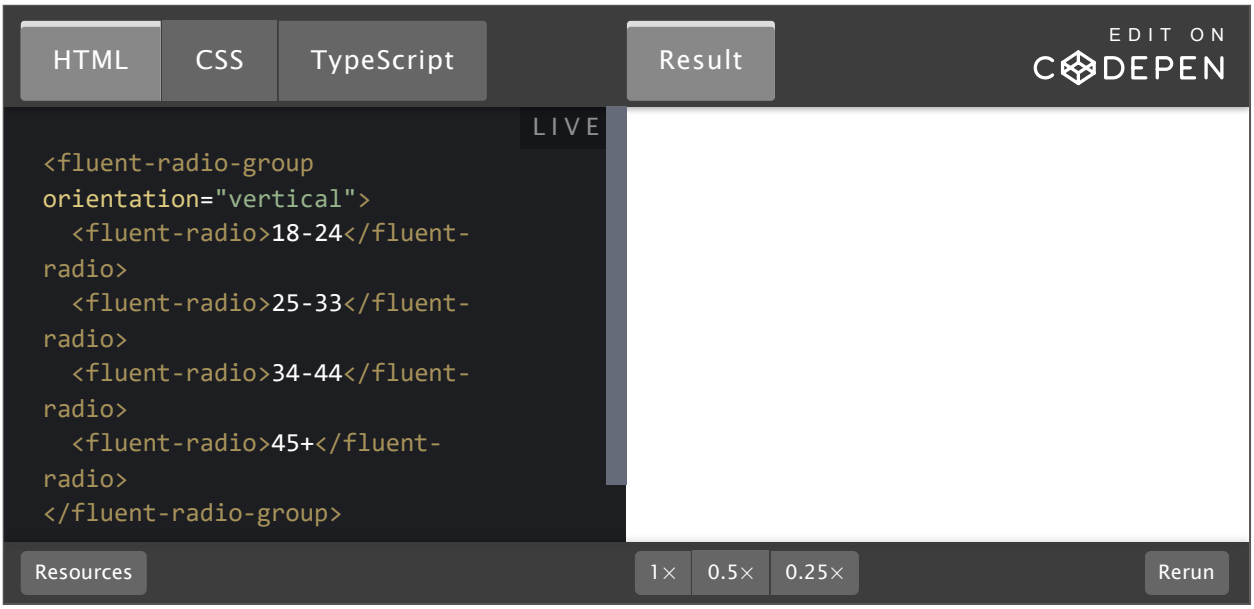
```
import {
    provideFluentDesignSystem,
    fluentRadio,
    fluentRadioGroup
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentRadio(),
        fluentRadioGroup()
    );
```

Customize your design

TypeScript

```
import { RadioGroup, radioGroupTemplate as template } from "@fluentui/fast-
foundation";
import { radioGroupStyles as styles } from "./my-radio-group.styles";

export const myRadioGroup = RadioGroup.compose({
    baseName: "radio-group",
    template,
    styles,
});
```

Example



Additional Resources

See also checkbox W3C Component Aria Practices

Select

Article • 07/20/2022 • 2 minutes to read

An implementation of an HTML select element as a form-connected web-component.

fluent-select

Setup

TypeScript

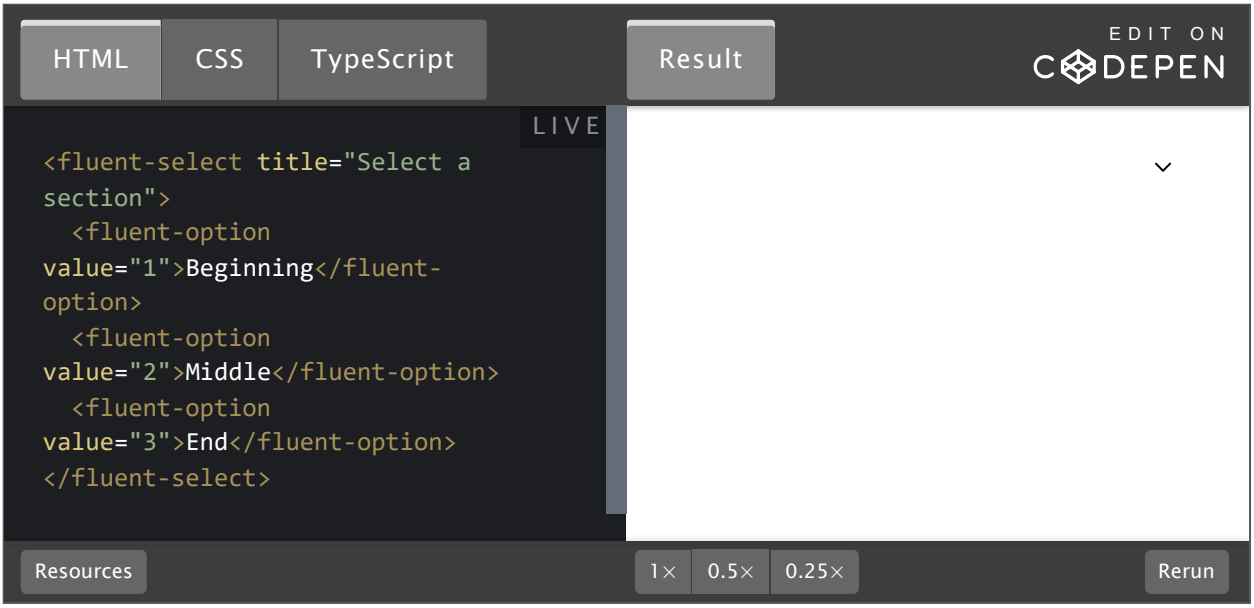
```
import {
    provideFluentDesignSystem,
    fluentSelect,
    fluentOption
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentSelect(),
        fluentOption()
    );
```

Customize the indicator

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentSelect,
    fluentOption
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentSelect({
          indicator: `...your indicator...`
        }),
        fluentOption()
    );
```

Example



Additional Resources

See also listbox See also combobox

Skeleton

Article • 07/20/2022 • 2 minutes to read

The skeleton component is used as a visual placeholder for an element while it is in a loading state and usually presents itself as a simplified wireframe-like version of the UI it is representing.

fluent-skeleton

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentSkeleton
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentSkeleton()
    );
```

Pattern

A URL for an image asset may be passed to the pattern attribute. In this mode, the fluent-skeleton component is used as a container for a transparent SVG that may express a more complex placeholder

HTML

```
<fluent-skeleton
    style="
        border-radius: 4px;
        width: 500px;
        height: 250px;
    "
    shape="rect"
    pattern="https://static.fast.design/assets/skeleton-test-pattern.svg"
></fluent-skeleton>
```

Shimmer

The shimmer boolean attribute will activate the component's shimmer effect.

Custom SVG

TypeScript

```
<fluent-skeleton
    style="
        border-radius: 4px;
        width: 500px;
        height: 250px;
    "
    shape="rect"
    shimmer
>
    <svg
        style="position: absolute; left: 0; top: 0;"
        id="pattern"
        width="100%"
        height="100%"
    >
        <defs>
            <mask id="mask" x="0" y="0" width="100%" height="100%">
                <rect x="0" y="0" width="100%" height="100%" fill="#ffffff"
/>
                <rect x="0" y="0" width="100%" height="45%" rx="4" />
                <rect x="25" y="55%" width="90%" height="15px" rx="4" />
                <rect x="25" y="65%" width="70%" height="15px" rx="4" />
                <rect x="25" y="80%" width="90px" height="30px" rx="4" />
            </mask>
        </defs>
        <rect
            x="0"
            y="0"
            width="100%"
            height="100%"
            mask="url(#mask)"
            fill="#ffffff"
        />
    </svg>
</fluent-skeleton>
```

Example

HTML CSS TypeScript Result EDIT ON CODEPEN

LIVE

```
<fluent-skeleton style="height:
50px; width: 250px" shape="rect"
shimmer="false"></fluent-skeleton>
```


Resources 1× 0.5× 0.25× Rerun

Slider

Article • 07/20/2022 • 2 minutes to read

An implementation of a range slider as a form-connected web-component. Note that if the slider is in vertical orientation by default the component will get a height using the css var --fluent-slider-height, by default that equates to (10px * var(--thumb-size)) or 160px. Inline styles will override that height.

fluent-slider

Setup

TypeScript

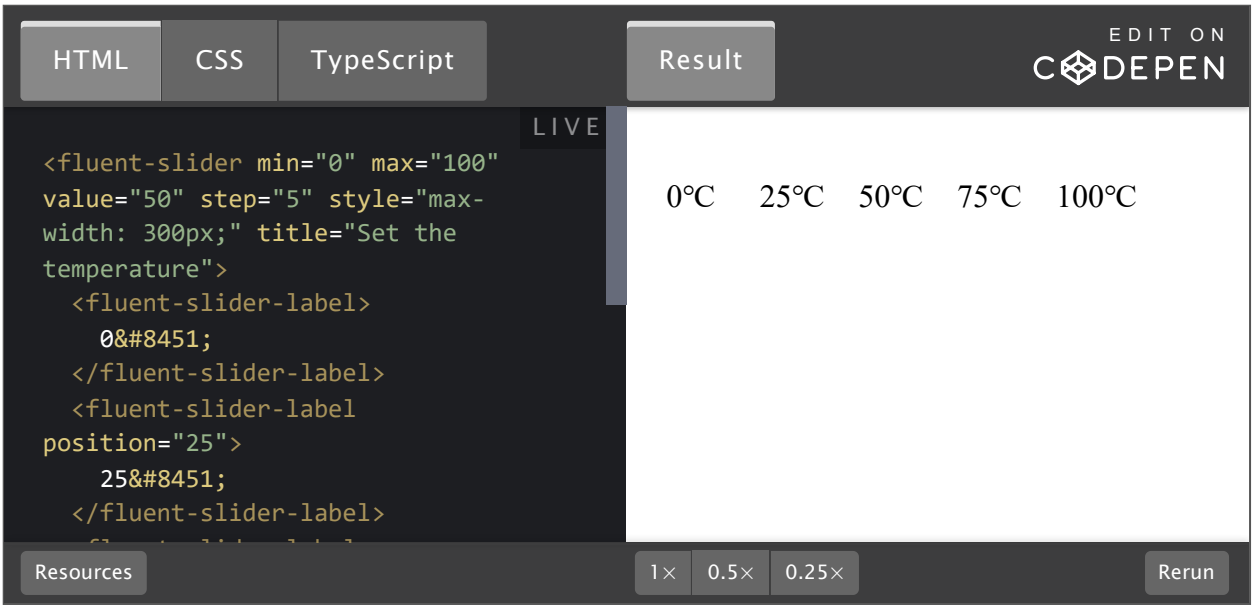
```
import {
    provideFluentDesignSystem,
    fluentSlider,
    fluentSliderLabel
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentSlider(),
        fluentSliderLabel()
    );
```

Customizing the thumb

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentSlider,
    fluentSliderLabel
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentSlider({
            thumb: `...your thumb...`
        }),
        fluentSliderLabel()
    );
```

Example



Additional Resources

Code in GitHub W3C Component Aria Practices

Switch

Article • 07/20/2022 • 2 minutes to read

An implementation of a switch as a form-connected web-component.

fluent-switch

Setup

TypeScript

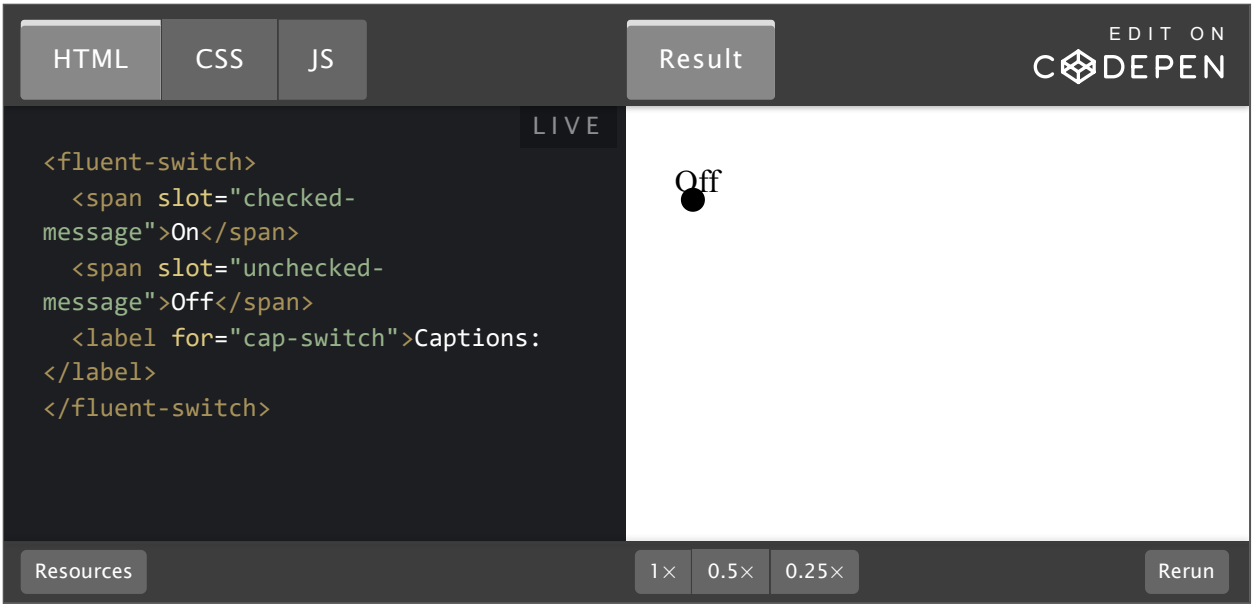
```
import {
    provideFluentDesignSystem,
    fluentSwitch
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentSwitch()
    );
```

Customizing the indicator

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentSwitch
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentSwitch({
            switch: `...your switch indicator...`
        })
    );
```

Example



Additional Resources

W3C Component Aria Practices

Tabs

Article • 07/20/2022 • 2 minutes to read

Tabs are a set of layered sections of content that display one panel of content at a time. Each tab panel has an associated tab element, that when activated, displays the panel. The list of tab elements is arranged along one edge of the currently displayed panel.

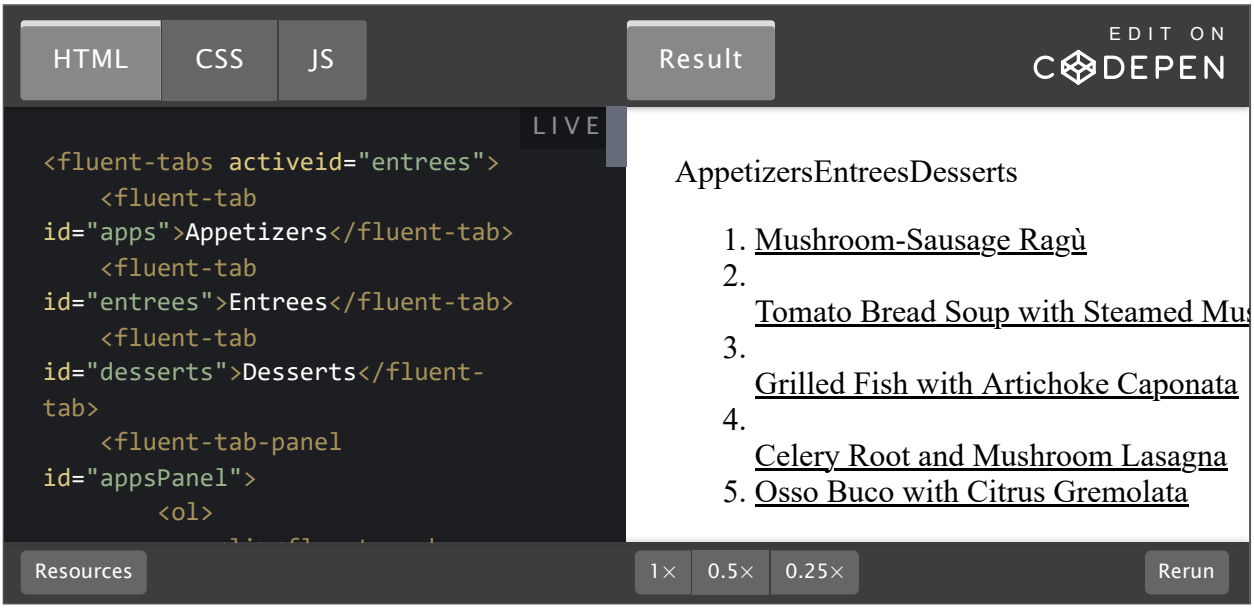
fluent-tabs

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentTab,
    fluentTabPanel,
    fluentTabs
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentTab(),
        fluentTabPanel(),
        fluentTabs()
    );
```

Example



Additional Resources

Code in GitHub W3C Component Aria Practices

Text Area

Article • 07/20/2022 • 2 minutes to read

An implementation of an HTML textarea element as a form-connected web-component. The fluent-text-area supports two visual appearances, outline and filled, with the control defaulting to the outline appearance.

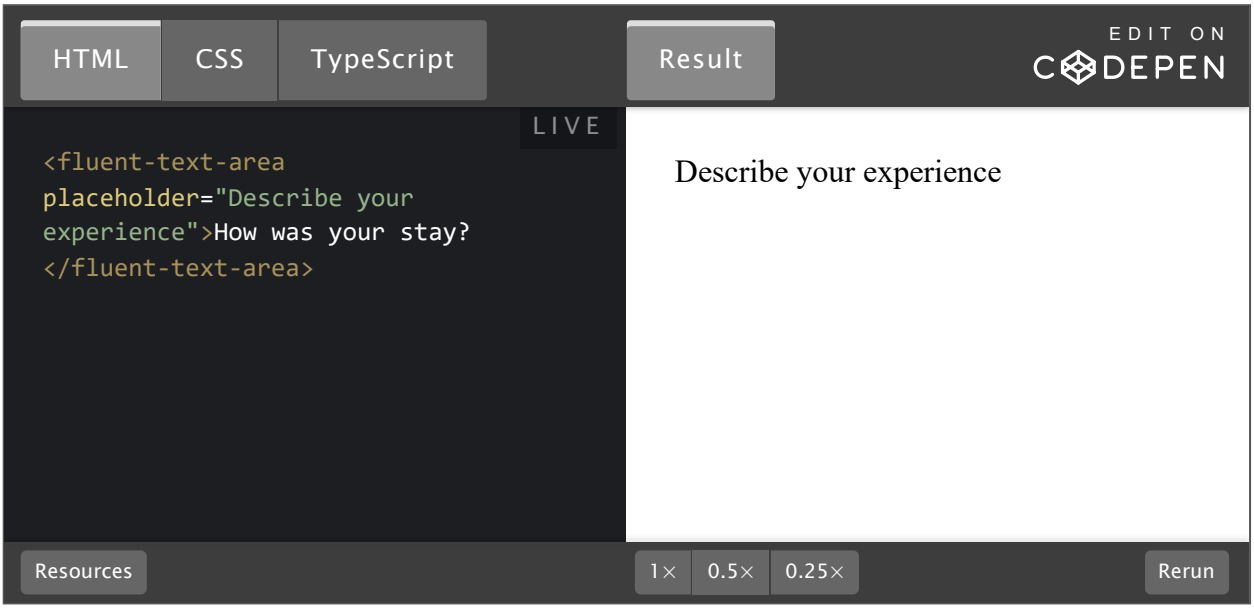
fluent-text-area

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentTextArea
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentTextArea()
    );
```

Example



Text Field

Article • 07/20/2022 • 2 minutes to read

An implementation of a text field as a form-connected web-component. The fluent-text-field supports two visual appearances, outline and filled, with the control defaulting to the outline appearance.

> ⓘ **Note**
>
> This component filters out slotted text nodes that are only white space to properly hide the label when the label is not in use.

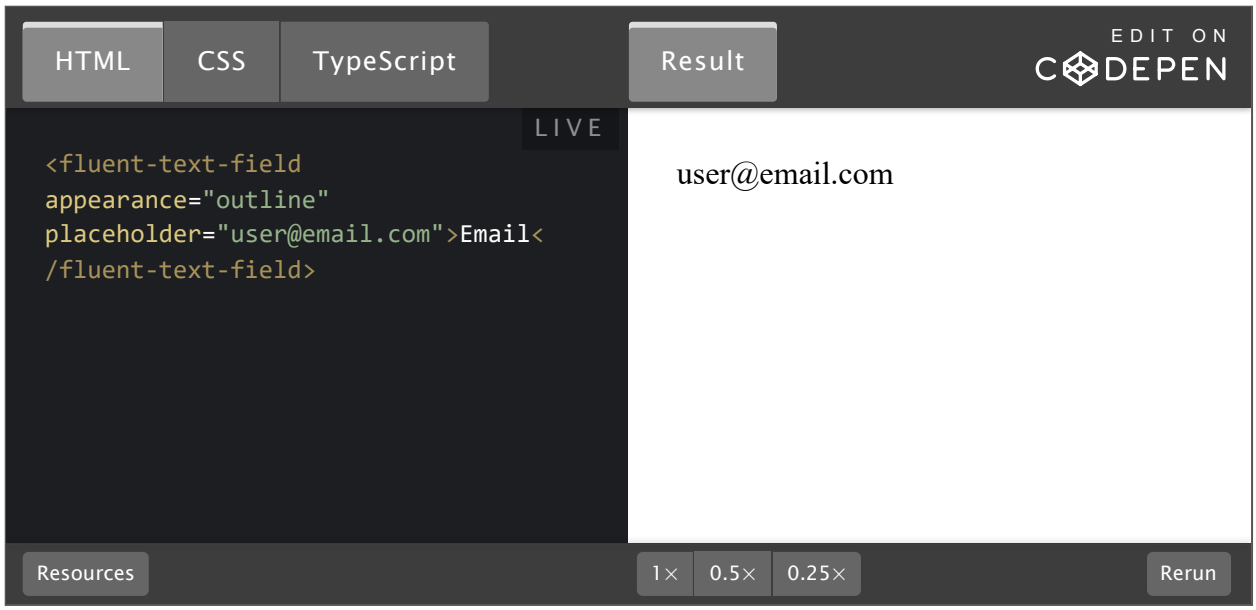
fluent-text-field

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentTextField
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentTextField()
    );
```

Example


HTML
CSS
TypeScript
Result
EDIT ON
CODEPEN
LIVE
<fluent-text-field
appearance="outline"
placeholder="user@email.com">Email<
/fluent-text-field>
user@email.com
Resources
1×
0.5×
0.25×
Rerun

Tooltip

Article • 07/20/2022 • 2 minutes to read

The tooltip component is used provide extra information about another element when it is hovered.

fluent-tooltip

Setup

TypeScript

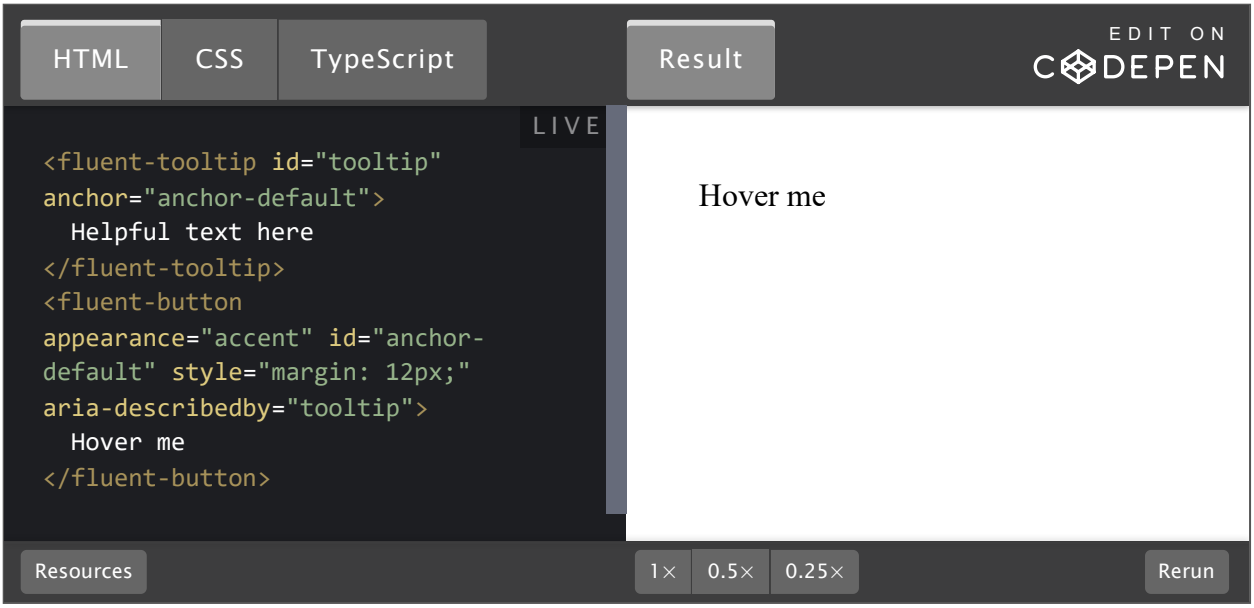
```
import {
    provideFluentDesignSystem,
    fluentTooltip
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentTooltip()
    );
```

Markup

HTML

```
<div>
    <fluent-button id="anchor">Hover me</fluent-button>
    <fluent-tooltip anchor="anchor">Tooltip text</fluent-tooltip>
</div>
```

Example



Additional Resources

Code in GitHub W3C Component Aria Practices

Tree View

Article • 07/20/2022 • 2 minutes to read

As defined by the W3C:

> A tree view widget presents a hierarchical list. Any item in the hierarchy may have child items, and items that have children may be expanded or collapsed to show or hide the children. For example, in a file system navigator that uses a tree view to display folders and files, an item representing a folder can be expanded to reveal the contents of the folder, which may be files, folders, or both.

fluent-tree-view

Setup

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentTreeItem,
    fluentTreeView
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentTreeItem(),
        fluentTreeView()
    );
```

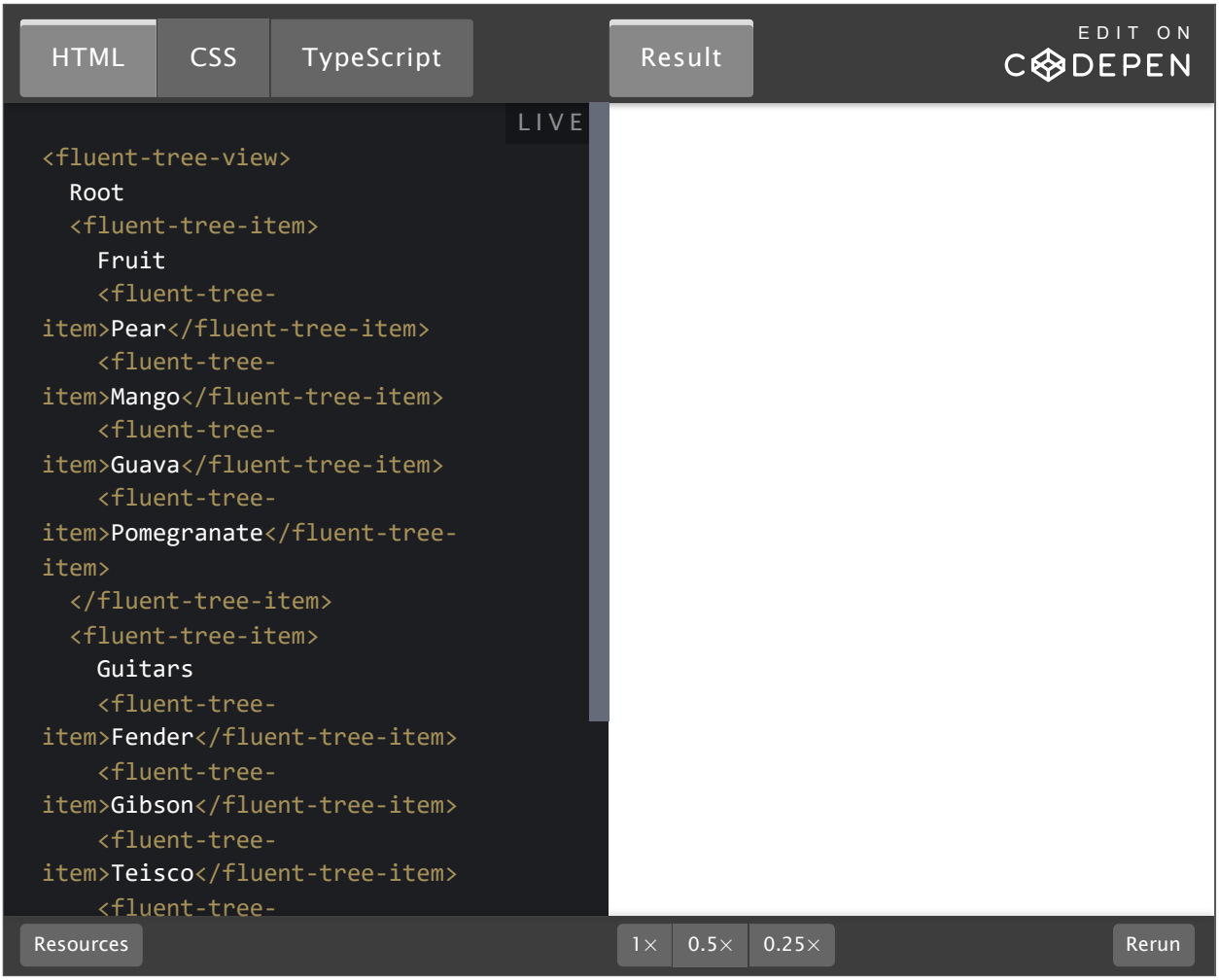
Customize the glyph

TypeScript

```
import {
    provideFluentDesignSystem,
    fluentTreeItem,
    fluentTreeView
} from "@fluentui/web-components";

provideFluentDesignSystem()
    .register(
        fluentTreeItem({
            expandCollapseGlyph: `...your expand/collapse glyph`
        }),
```

```
        fluentTreeView()
    );
```

Example



Additional Resources

Code in GitHub W3C Component Aria Practices

Introduction to Integrations

Article • 11/01/2021 • 2 minutes to read

The Fluent UI Web Component libraries, composed from FAST Components, can be used on their own to build modern web sites and applications, but are also designed to be used in combination with a wide variety of existing technologies. This section is dedicated to helping you get Fluent UI web components working with your preferred stack.

- Angular
- ASP.NET
- Aurelia
- Blazor
- React
- Vue
- Webpack

Not seeing an integration for your preferred technology? We'd be happy to work with you to add it. Feel free to kick off a discussion by opening an issue on the FAST component GitHub.

Next steps

- View the Components
- Check Browser Support

Use Fluent UI Web Components with Angular

Article • 11/01/2021 • 2 minutes to read

Fluent UI Web Components integrate nicely with Angular. Let's take a look at how you can set up an Angular project, starting from scratch.

Setting up the Angular project

First, you'll need to make sure that you have Node.js installed. You can learn more and download that on the official site.

With Node.js installed, you can run the following command to install the Angular CLI:

```shell
npm install -g @angular/cli
```

With the CLI installed, you have access to the `ng` command-line interface. This can be used to create a new Angular project. For example, to create a new Angular App named "fluent-angular", you would use the following command:

```shell
ng new fluent-angular
```

Follow the prompts, answering each question in turn. When the CLI completes, you should have a basic runnable Angular application.

Configuring packages

Next, we'll install the Fluent packages, along with supporting libraries. To do that, run this command from your new project folder:

```Bash
npm install --save @fluentui/web-components @microsoft/fast-element lodash-es
```

Using the components

With all the basic pieces in place, let's run our app in dev mode with `ng serve --open`. The Angular CLI should build your project and make it available on localhost. Right now, it displays a basic welcome message, since we haven't added any code or interesting HTML. Let's change that.

First, open your `src/main.ts` file and add the following code:

ts

```ts
import { provideFluentDesignSystem, fluentCard, fluentButton,
fluentTextField } from '@microsoft/fluent-components';

provideFluentDesignSystem().register(fluentCard(), fluentButton(),
fluentTextField());
```

This code uses the Fluent Design System to register `<fluent-card>`, `<fluent-button>` and `<fluent-text-field>` components. Once you save, the dev server will rebuild and refresh your browser. However, you still won't see anything. To get some UI showing up, we need to write some HTML that uses our components. Replace the HTML template in your `app/app.component.html` file with the following markup:

HTML

```html
<fluent-card>
  <h2>{{title}}</h2>
  <fluent-text-field
    [(ngModel)]="exampleTextField"
    name="exampleTextField"
    ngDefaultControl
    placeholder="Enter Some Text"
  ></fluent-text-field>
  <fluent-button appearance="accent" (click)="onClick()">Click Me</fluent-
button>
</fluent-card>
```

Replace the code in your `app/app.component.ts` file contents with this:

ts

```ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
```

```
})
export class AppComponent {
  title = 'fluent-angular';

  exampleTextField = '';

  onClick() {
    console.log(this.exampleTextField);
  }
}
```

To allow an NgModule to contain Non-Angular element names, add the following code in your `app/app.module.ts` file:

ts

```ts
import { CUSTOM_ELEMENTS_SCHEMA } from '@angular/core';

@NgModule({
  schemas: [ CUSTOM_ELEMENTS_SCHEMA ]
})
```

To add a splash of style, replace the `app/app.component.css` file contents with this:

css

```css
fluent-card {
  padding: 16px;
  display: flex;
  flex-direction: column;
}

fluent-text-field {
  margin-bottom: 12px;
}

h2 {
  font-size: var(--type-ramp-plus-5-font-size);
  line-height: var(--type-ramp-plus-5-line-height);
}

fluent-card > fluent-button {
  align-self: flex-end;
}
```

ⓘ Note

Third party controls require a ControlValueAccessor for writing a value and listening to changes on input elements. Add ngDefaultControl attribute to your component

to have two-way binding working with FormControlDirective, FormControlName, or NgModel directives:

HTML

```
<fluent-text-field placeholder="name" id="name" formControlName="name"
ngDefaultControl></fluent-text-field>
```

Congratulations! You're now set up to use Fluent and Angular!

Next steps

- See the components

Use Fluent UI Web Components with ASP.NET

Article • 11/01/2021 • 2 minutes to read

Fluent UI Web Components work naturally with ASP.NET server-side development, by simply adding a script tag and using the custom HTML elements. Let's take a look at how to set things up.

Setting up the ASP.NET project

First, you'll need to make sure that you have the .NET SDK installed. You can learn more and download that on the official site.

With the SDK installed, you have access to the `dotnet` command-line interface. This can be used to create a new ASP.NET project. For example, to create a new ASP.NET Core MVC Web App named "fluent-aspnet", you would use the following command:

shell

```
dotnet new mvc -o fluent-aspnet
```

Create a project using the command above if you don't already have one. When the CLI completes, you should have a basic runnable ASP.NET Core MVC application.

Configuring scripts

Now that we've got our basic project setup, we need to add our web components script and update ASP.NET accordingly. You can either add the script from our CDN directly, or you can install it with NPM and then add that.

To add a CDN script for `fluent-components` use the following markup:

HTML

```
<script type="module" src="https://unpkg.com/@fluentui/web-components">
</script>
```

The best place to put this is typically in your `_Layout.cshtml` file in the script section at the bottom of the `<body>`.

If you wish to leverage NPM instead, run the following command:

```shell
npm install --save @fluentui/web-components
```

You can locate the single file script build in the following location:

```shell
node_modules/@fluentui/web-components/dist/fluent-components.min.js
```

Copy this to your `wwwroot/js` folder and reference it with a script tag as described above.

Should you wish to go one step further and leverage a client-side bundler, such as Webpack, there is some additional setup to integrate with ASP.NET that is beyond the scope of this tutorial. Basic Webpack instructions for Fluent UI Web Components can be found here. The most important detail with respect to Fluent UI Web Components is that you'll want to install a few more packages. Use the following command if this is your preferred setup:

```shell
npm install --save @fluentui/web-components @microsoft/fast-element lodash-es
```

In this case, because Webpack can tree-shake unused components, you'll also want to be sure to register the components you want to use somewhere in your own JavaScript code. See our Webpack guide for an example.

Using the components

With your script tag added (or your client bundle in place), you can use any component in any of your views. For example, you could put something like this in your `Index.cshtml` file:

```HTML
@{ ViewData["Title"] = "Home Page"; }

<fluent-card>
  <h2>@ViewData["Title"]</h2>
```

```
  <fluent-button id="button" appearance="accent">Click Me</fluent-button>
</fluent-card>
```

For a splash of style, add the following to your `wwwroot/css/site.css` file:

```css
:not(:defined) {
  visibility: hidden;
}

fluent-card {
  padding: 16px;
  display: flex;
  flex-direction: column;
}

h2 {
  font-size: var(--type-ramp-plus-5-font-size);
  line-height: var(--type-ramp-plus-5-line-height);
}

#button {
  align-self: flex-end;
}
```

Congratulations! You're now set up to use Fluent UI Web Components with ASP.NET. You can use more components, build your own components, and when you are ready, build and deploy your website or app to production.

Next steps

- See the components

Use Fluent UI Web Components with Aurelia

Article • 11/01/2021 • 5 minutes to read

Fluent UI Web Components works flawlessly with both Aurelia 1 and Aurelia 2, with full integration into the binding engine and component model. Let's take a look at how you can set up an Aurelia project, starting from scratch.

Aurelia 2

Setting up the Aurelia 2 project

First, you'll need to make sure that you have Node.js installed. You can learn more and download that on the official site.

With Node.js installed, you can run the following command to create a new Aurelia 2 project:

```shell
npx makes Aurelia
```

Follow the prompts, answering each question in turn. It is recommended that you select the "Default TypeScript Aurelia 2 App" when prompted unless you have previous experience with the CLI. Be sure to choose to install dependencies when asked.

When the CLI completes, you should have a basic runnable Aurelia 2 application.

Configuring packages

Next, we'll install the Fluent UI Web Components packages, along with supporting libraries. To do that, run this command from your new project folder:

```shell
npm install --save @fluentui/web-components @microsoft/fluent-element
lodash-es
```

Using the components

With all the basic pieces in place, let's run our app in dev mode with `npm start`. Webpack should build your project and open your default browser with your `index.html` page. Right now, it should only have a hello message, since we haven't added any code or interesting HTML. Let's change that.

First, open your `src/main.ts` file and add the following code:

ts

```ts
import { provideFluentDesignSystem, fluent, fluentButton } from
'@fluentui/web-components';

provideFluentDesignSystem().register(fluentCard(), fluentButton());
```

This code uses the Fluent UI Web Components Design System to register the `<fluent-card>` and `<fluent-button>` components. Once you save, the dev server will rebuild and refresh your browser. However, you still won't see anything. To get some UI showing up, we need to write some HTML that uses our components. Replace your `my-app.html` file with the following markup:

HTML

```html
<fluent-card>
  <h2>${message}</h2>
  <fluent-button appearance="accent" click.trigger="onClick()">Click
Me</fluent-button>
</fluent-card>
```

Replace your `my-app.ts` with this:

ts

```ts
export class MyApp {
  public message = 'Hello World!';

  onClick() {
    console.log('clicked!');
  }
}
```

To add a splash of style, replace your `my-app.css` content with this:

css

```css
fluent-card {
  padding: 16px;
```

```
  display: flex;
  flex-direction: column;
}

h2 {
  font-size: var(--type-ramp-plus-5-font-size);
  line-height: var(--type-ramp-plus-5-line-height);
}

fluent-card > fluent-button {
  align-self: flex-end;
}
```

Enabling two-way bindings

Aurelia knows by default how to listen for changes in native elements. Now we need to teach it how to listen for changes in Fluent UI elements. You can do so by extending its templating syntax.

You can either use a wrapper developed by the community or teach Aurelia manually:

Import and register `aurelia-fast-adapter`

Start by installing the adapter

ts

```
npm install aurelia-fast-adapter
```

and then simply register it from your `src/main.ts`:

ts

```
// src/main.ts

import { FASTAdapter } from 'aurelia-fast-adapter';

Aurelia.register(FASTAdapter) // add this line
  // other registrations...
  .start();
```

If you use Fluent UI Web Components in its default configuration that's all you need to do. But if you changed the prefix of your components to something else, you can customize the adapter as such:

ts

```
// src/main.ts

import { FASTAdapter } from 'aurelia-fast-adapter';

Aurelia
  .register(FASTAdapter.customize({withPrefix: 'my-custom-prefix'}) //
customized with prefix
  .start();
```

Also, in case you have local components that require two-way binding, you can adjust the adapter before to register it as such:

ts

```
// src/main.ts

import { FASTAdapter } from 'aurelia-fast-adapter';

// this line will tell the adapter that it must use two-way binding on the
<my-custom-prefix-date-field> component and use this two-way binding on the
`value` property. It's possible to add several properties at once if
necessary
FASTAdapter.tags['DATE-FIELD'] = ['value'];

Aurelia
  .register(FASTAdapter.customize({withPrefix: 'my-custom-prefix'})
  .start();
```

Congratulations! You're now set up to use Fluent UI Web Components and Aurelia 2!

Manually teach Aurelia 2 about two-way binding:

If the example doesn't seem obvious, the following prerequisite reads are recommended:

- extending Aurelia templating syntax

The following is a code example of how to teach Aurelia to work seamlessly with Microsoft Fluent UI Web Components.

TypeScript

```
import { AppTask, IContainer, IAttrMapper, NodeObserverLocator } from
'aurelia';

Aurelia.register(
  AppTask.beforeCreate(IContainer, container => {
    const attrMapper = container.get(IAttrMapper);
```

```
    const nodeObserverLocator = container.get(NodeObserverLocator);
    attrMapper.useTwoWay((el, property) => {
      switch (el.tagName) {
        case 'FAST-SLIDER':
        case 'FAST-TEXT-FIELD':
        case 'FAST-TEXT-AREA':
          return property === 'value';
        case 'FAST-CHECKBOX':
        case 'FAST-RADIO':
        case 'FAST-RADIO-GROUP':
        case 'FAST-SWITCH':
          return property === 'checked';
        case 'FAST-TABS':
          return property === 'activeid';
        default:
          return false;
      }
    });

    // Teach Aurelia what events to use to observe properties of elements.
    // Because FAST components all use a single change event to notify,
    // we can use a single common object
    const valuePropertyConfig = { events: ['input', 'change'] };
    nodeObserverLocator.useConfig({
      'FAST-CHECKBOX': {
        checked: valuePropertyConfig,
      },
      'FAST-RADIO': {
        checked: valuePropertyConfig,
      },
      'FAST-RADIO-GROUP': {
        value: valuePropertyConfig,
      },
      'FAST-SLIDER': {
        value: valuePropertyConfig,
      },
      'FAST-SWITCH': {
        checked: valuePropertyConfig,
      },
      'FAST-TABS': {
        activeid: valuePropertyConfig,
      },
      'FAST-TEXT-FIELD': {
        value: valuePropertyConfig,
      },
      'FAST-TEXT-AREA': {
        value: valuePropertyConfig,
      },
    });
  }),
);
```

Aurelia 1

Setting up the Aurelia 1 project

First, you'll need to make sure that you have Node.js installed. You can learn more and download that on the official site.

With Node.js installed, you can run the following command to install the Aurelia 1 CLI:

```shell
npm install -g aurelia-cli
```

And then use the CLI like this:

```shell
au new fast-aurelia
```

Follow the prompts, answering each question in turn. It is recommended that you select the "Default TypeScript App" when prompted unless you have previous experience with the CLI. Be sure to choose to install dependencies when asked.

When the CLI completes, you should have a basic runnable Aurelia 1 application.

Configuring packages

Next, we'll install the Fluent UI Web Components packages, along with supporting libraries. To do that, run this command from your new project folder:

```shell
npm install --save @fluentui/web-components @microsoft/fluent-element
lodash-es
```

Using the components

With all the basic pieces in place, let's run our app in dev mode with `npm start`. Webpack should build your project and make it available at `http://localhost:8080/`. If you visit this address it should only have a hello message, since we haven't added any code or interesting HTML. Let's change that.

First, open your `src/main.ts` file and add the following code:

ts

```ts
import { provideFluentDesignSystem, fluentCard, fluentButton } from
'@fluentui/web-components';

provideFluentDesignSystem().register(fluentCard(), fluentButton());
```

This code uses the Fluent UI Design System to register the `<fluent-card>` and `<fluent-button>` components. Once you save, the dev server will rebuild and refresh your browser. However, you still won't see anything. To get some UI showing up, we need to write some HTML that uses our components. Replace your `app.html` file with the following markup:

HTML

```html
<template>
  <fluent-card>
    <h2>${message}</h2>
    <fluent-button appearance="accent" click.trigger="onClick()">Click
Me</fluent-button>
  </fluent-card>
</template>
```

Replace your `app.ts` with this:

ts

```ts
export class App {
  public message: string = 'Hello World!';

  onClick() {
    console.log('clicked!');
  }
}
```

To add a splash of style, add the following to your `app.html` template:

HTML

```html
<style>
  fluent-card {
    padding: 16px;
    display: flex;
    flex-direction: column;
  }
```

```
  h2 {
    font-size: var(--type-ramp-plus-5-font-size);
    line-height: var(--type-ramp-plus-5-line-height);
  }

  fluent-card > fluent-button {
    align-self: flex-end;
  }
</style>
```

Congratulations! You're now set up to use Fluent UI Web Components and Aurelia 1!

Next steps

- See the components

Use Fluent UI Web Components with Blazor

Article • 07/20/2022 • 7 minutes to read

Fluent UI Web Components work seamlessly with Blazor, including integration with Blazor's binding engine and components. Let's take a look at how to set things up.

Setting up the Blazor project

First, you'll need to make sure that you have the .NET 6 SDK installed. You can learn more and download that on the official site.

With the SDK installed, you have access to the `dotnet` command-line interface. This can be used to create a new Blazor project. For example, to create a new Blazor App named "fluent-blazor", you would use the following command:

```shell
dotnet new blazorwasm -o fluent-blazor
```

Create a project using the command above if you don't already have one. When the CLI completes, you should have a basic runnable Blazor application. For more information on setting up and using Blazor, see the official Blazor Getting Started guide

Configuring the Fluent UI Web Components

To get started using the Fluent UI Web Components for Blazor, you will first need to install the official Nuget package for Fluent UI. You can use the following command:

```shell
dotnet add package Microsoft.Fast.Components.FluentUI
```

Next, you need to add the web components script. You can either add the script from CDN directly, or you can install it with NPM, whichever you prefer.

To add the script from CDN use the following markup:

```HTML
```

```
<script type="module" src="https://cdn.jsdelivr.net/npm/@fluentui/web-components/dist/web-components.min.js"></script>
```

The markup above always references the latest release of the components. When deploying to production, you will want to ship with a specific version. Here's an example of the markup for that:

HTML

```
<script type="module" src="https://cdn.jsdelivr.net/npm/@fluentui/web-components@2.0.2/dist/web-components.min.js"></script>
```

The best place to put the script tag is typically in your `index.html` file in the script section at the bottom of the `<body>`.

If you wish to leverage NPM instead, run the following command:

shell

```
npm install --save @fluentui/web-components
```

You can locate the single file script build in the following location:

shell

```
node_modules/@fluentui/web-components/dist/web-components.min.js
```

Copy this to your `wwwroot/script` folder and reference it with a script tag as described above.

> ⓘ **Important**
>
> If you are setting up Fluent UI Web Components on a Blazor Server project, you will need to escape the `@` character by repeating it in the source link. For more information check out the **Razor Pages syntax documentation**.

Using the FluentUI Web Components

With the package installed and the script configured, you can begin using the Fluent UI Web Components in the same way as any other Blazor component. Just be sure to add the following using statement to your views:

HTML

```
@using Microsoft.Fast.Components.FluentUI
```

Here's a small example of a `FluentCard` with a `FluentButton` that uses the Fluent "Accent" appearance:

HTML

```
@using Microsoft.Fast.Components.FluentUI

<FluentCard>
    <h2>Hello World!</h2>
    <FluentButton Appearance="@Appearance.Accent">Click Me</FluentButton>
</FluentCard>
```

Tip

You can add `@using Microsoft.Fast.Components.FluentUI` to the namespace collection in `_Imports.razor`, so that you can avoid repeating it in every single razor page.

If you are using the .NET CLI, you can run your project with the following command from the project folder:

shell

```
dotnet watch run
```

Congratulations! You're now set up to use the Fluent UI Web Components with Blazor!

Configuring the Design System

The Fluent UI Web Components are built on FAST's Adaptive UI technology, which enables design customization and personalization, while automatically maintaining accessibility. This is accomplished through setting various "Design Tokens". As of version 1.4 you can use all of the (160) individual Design Tokens, both from code as in a declarative way in your `.razor` pages. See /fluent-ui/web-components/design-system/design-tokens for more information on how Design Tokens work

Option 1: Using Design Tokens from C# code

Given the following `.razor` page fragment:

HTML

```
<FluentButton @ref="ref1" Appearance="Appearance.Filled">A
button</FluentButton>
<FluentButton @ref="ref2" Appearance="Appearance.Filled">Another
button</FluentButton>
<FluentButton @ref="ref3" Appearance="Appearance.Filled">And one
more</FluentButton>
<FluentButton @ref="ref4" Appearance="Appearance.Filled"
@onclick=OnClick>Last button</FluentButton>
```

You can use Design Tokens to manipulate the styles from C# code as follows:

C#

```
[Inject]
private BaseLayerLuminance BaseLayerLuminance { get; set; } = default!;

[Inject]
private AccentBaseColor AccentBaseColor { get; set; } = default!;

[Inject]
private BodyFont BodyFont { get; set; } = default!;

[Inject]
private StrokeWidth StrokeWidth { get; set; } = default!;

[Inject]
private ControlCornerRadius ControlCornerRadius { get; set; } = default!;

private FluentButton? ref1;
private FluentButton? ref2;
private FluentButton? ref3;
private FluentButton? ref4;

protected override async Task OnAfterRenderAsync(bool firstRender)
{
    if (firstRender)
    {
        //Set to dark mode
        await BaseLayerLuminance.SetValueFor(ref1!.Element, (float)0.15);

        //Set to Excel color
        await AccentBaseColor.SetValueFor(ref2!.Element,
"#185ABD".ToSwatch());

        //Set the font
        await BodyFont.SetValueFor(ref3!.Element, "Comic Sans MS");
```

```
        //Set 'border' width for ref4
        await StrokeWidth.SetValueFor(ref4!.Element, 7);
        //And change conrner radius as well
        await ControlCornerRadius.SetValueFor(ref4!.Element, 15);

        StateHasChanged();
    }

}

public async Task OnClick()
{
    //Remove the wide border
    await StrokeWidth.DeleteValueFor(ref4!.Element);
}
```

As can be seen in the code above (with the `ref4.Element`), it is posible to apply multiple tokens to the same component.

For Design Tokens that work with a color value, you must call the `ToSwatch()` extension method on a string value or use one of the `Swatch` constructors. This makes sure the color is using a format that Design Tokens can handle. A `Swatch` has a lot of commonality with the `System.Drawing.Color` struct. Instead of the values of the components being between 0 and 255, in a `Swatch` they are expressed as a value between 0 and 1.

> **ⓘ Important**
>
> The Design Tokens are manipulated through JavaScript interop working with an `ElementReference`. There is no JavaScript element until after the component is rendered. This means you can only work with the Design Tokens from code after the component has been rendered in `OnAfterRenderAsync` and not in any earlier lifecycle methods.

Option 2: Using Design Tokens as components

The Design Tokens can also be used as components in a `.razor` page directely. It looks like this:

HTML

```
<BaseLayerLuminance Value="(float?)0.15">
    <FluentCard BackReference="@context">
        <div class="contents">
            Dark
```

```
            <FluentButton
Appearance="Appearance.Accent">Accent</FluentButton>
            <FluentButton
Appearance="Appearance.Stealth">Stealth</FluentButton>
            <FluentButton
Appearance="Appearance.Outline">Outline</FluentButton>
            <FluentButton
Appearance="Appearance.Lightweight">Lightweight</FluentButton>
        </div>
    </FluentCard>
</BaseLayerLuminance>
```

To make this work, a link needs to be created between the Design Token component and its child components. This is done with the `BackReference="@context"` construct.

> ⓘ Note
>
> Only one Design Token component at a time can be used this way. If you need to set more tokens, use the code approach as described in Option 1 above.

Option 3: Using the `<FluentDesignSystemProvider>`

The third way to customize the design in Blazor is to wrap the entire block you want to manipulate in a `<FluentDesignSystemProvider>`. This special element has a number of properties you can set to configure a subset of the tokens. **Not all tokens are available/supported** and we recommend this to only be used as a fall-back mechanism. The preferred mehod of working with the desgn tokens is to manipulate them from code as described in option 1.

Here's an example of changing the "accent base color" and switching the system into dark mode (in the file `app.razor`):

HTML

```
<FluentDesignSystemProvider AccentBaseColor="#464EB8"
BaseLayerLuminance="0">
    <Router AppAssembly="@typeof(App).Assembly">
        <Found Context="routeData">
            <RouteView RouteData="@routeData"
DefaultLayout="@typeof(MainLayout)" />
        </Found>
        <NotFound>
            <PageTitle>Not found</PageTitle>
            <LayoutView Layout="@typeof(MainLayout)">
                <p role="alert">Sorry, there's nothing at this address.</p>
            </LayoutView>
```

```
        </NotFound>
    </Router>
</FluentDesignSystemProvider>
```

> **Note**
>
> Provider token attributes can be changed on-the-fly like any other Blazor component attribute.

Colors for integration with specific Microsoft products

If you are attempting to configure the components for integration into a specific Microsoft product, the following table provides `AccentBaseColor` values you can use:

Product	AccentBaseColor
Office	#D83B01
Word	#185ABD
Excel	#107C41
PowerPoint	#C43E1C
Teams	#6264A7
OneNote	#7719AA
SharePoint	#03787C
Stream	#BC1948

For a list of all available token attributes, see here. More examples for other components can be found in the `examples` folder of this repository.

Web components / Blazor components mapping, implementation status and remarks

Web component	Blazor component	Status	Remarks
<fluent-accordion>	`<FluentAccordion>`	✔	-
`<fluent-accordion-item>`	`<FluentAccordionItem>`	✔	-

Web component	Blazor component	Status	Remarks
<fluent-anchor>	`<FluentAnchor>`	✔	-
<fluent-anchored-region>	`<FluentAnchoredRegion>`	✔	-
<fluent-badge>	`<FluentBadge>`	✔	-
<fluent-breadcrumb>	`<FluentBreadcrumb>`	✔	-
`<fluent-breadcrumb-item>`	`<FluentBreadcrumbItem>`	✔	-
<fluent-button>	`<FluentButton>`	✔	-
<fluent-card>	`<FluentCard>`	✔	-
<fluent-checkbox>	`<FluentCheckbox>`	✔	-
<fluent-combobox>	`<FluentCombobox>`	✔	-
<fluent-data-grid>	`<FluentDataGrid>`	✔	-
`<fluent-data-grid-cell>`	`<FluentDataGridCell>`	✔	-
`<fluent-data-grid-row>`	`<FluentDataGridRow>`	✔	-
<fluent-design-system-provider>	`<FluentDesignSystemProvider>`	✔	-
<fluent-dialog>	`<FluentDialog>`	✔	-
<fluent-divider>	`<FluentDivider>`	✔	-
<fluent-flipper>	`<FluentFlipper>`	✔	-
<fluent-horizontal-scroll>	`<FluentHorizontalScroll>`	✔	-
No web component	`<FluentIcon>`	✔	-
<fluent-listbox>	`<FluentListbox>`	✔	-
<fluent-menu>	`<FluentMenu>`	✔	-
`<fluent-menu-item>`	`<FluentMenuItem>`	✔	-
<fluent-number-field>	`<FluentNumberField>`	✔	-
`<fluent-option>`	`<FluentOption>`	✔	-
<fluent-progress>	`<FluentProgress>`	✔	-
<fluent-progress-ring>	`<FluentProgressRing>`	✔	-

Web component	Blazor component	Status	Remarks
<fluent-radio>	`<FluentRadio>`	✔	-
<fluent-radio-group>	`<FluentRadioGroup>`	✔	-
<fluent-select>	`<FluentSelect>`	✔	-
<fluent-skeleton>	`<FluentSkeleton>`	✔	-
<fluent-slider>	`<FluentSlider>`	✔	-
`<fluent-slider-label>`	`<FluentSliderLabel>`	✔	-
<fluent-switch>	`<FluentSwitch>`	✔	-
<fluent-tabs>	`<FluentTabs>`	✔	-
`<fluent-tab>`	`<FluentTab>`	✔	-
`<fluent-tab-panel>`	`<FluentTabPanel>`	✔	-
<fluent-text-area>	`<FluentTextArea>`	✔	-
<fluent-text-field>	`<FluentTextField>`	✔	-
<fluent-toolbar>	`<FluentToolbar>`	✔	-
<fluent-tooltip>	`<FluentTooltip>`	✔	-
<fluent-tree-view>	`<FluentTreeView>`	✔	-
`<fluent-tree-item>`	`<FluentTreeItem>`	✔	-

To report issues or provide feedback on `Microsoft.Fast.Components.FluentUI`, please visit the microsoft/fast-blazor repository.

Next steps

- See the components

Use Fluent UI Web Components with Ember

Article • 11/01/2021 • 2 minutes to read

Fluent UI Web Components and Ember work great together.

Setting up the Ember project

First, you'll need to make sure that you have Node.js installed. You can learn more and download that on the official site.

With Node.js installed, you can run the following command to install the Ember CLI:

```shell
npm install -g ember-cli
```

With the CLI installed, you have access to the `ember` command-line interface. This can be used to create a new Ember project. For example, to create a new Ember App named "fluent-ember", you would use the following command:

```shell
ember new fluent-ember --lang en
```

When the CLI completes, you should have a basic runnable Ember application.

Configuring packages

Next, we'll install the Fluent UI Web Component packages, along with supporting libraries. To do that, run this command from your new project folder:

```shell
npm install --save @fluentui/web-components @microsoft/fast-element lodash-es
```

Using the components

With all the pieces in place, let's run our app in dev mode with `npm start`. The Ember CLI should build your project and make it available on localhost. Right now, it displays a basic welcome message, since we haven't added any code or interesting HTML. Let's change that.

First, open your `app/app.js` file and add the following code:

ts

```ts
import { provideFluentDesignSystem, fluentCard, fluentButton,
fluentTextField } from '@fluentui/web-components';

provideFluentDesignSystem().register(fluentCard(), fluentButton(),
fluentTextField());
```

This code uses the Fluent Design System to register the `<fluent-card>`, `<fluent-button>`, and `<fluent-text-field>` components. Once you save, the dev server will rebuild and refresh your browser. However, you still won't see anything. Open your `application.hbs` file and replace the `<WelcomePage />` component with the following HTML and then save again.

HTML

```HTML
    <h2>Fluent Ember</h2>
    <fluent-text-field placeholder="Enter Some Text"></fluent-text-field>
    <fluent-button appearance="accent">Click Me</fluent-button>
  </fluent-card>
```

Now you should see the Fluent UI Web Components displayed in your Ember application.

Next, let's improve this by refactoring it into a component. Stop the CLI and run the following command to scaffold a new Ember component that will be called `fluent-demo`.

ts

```ts
ember generate component fluent-demo
```

Copy your original HTML above and use it to replace the HTML in your `app/components/fluent-demo.hbs` file. Next replace that same HTML in `templates/application.hbs` file with the following Ember component use:

HTML

```HTML
<FluentDemo />
```

Run `npm start` again and you should see the same output, but now we have moved our web components into a `fluentDemo` Ember component.

Let's take it a little further. Create a `fluent-demo.js` file in the same folder as your `fluent-demo.hbs` file and paste the following code:

JavaScript

```
import Component from '@glimmer/component';
import { action } from '@ember/object';
import { tracked } from '@glimmer/tracking';

export default class FluentDemoComponent extends Component {
  @tracked exampleTextField = '';

  @action
  onClick() {
    console.log(this.exampleTextField);
  }

  @action
  onInput(event) {
    this.exampleTextField = event.target.value;
  }
}
```

Next, update the `fluent-demo.hbs` file with the following HTML:

HTML

```
<fluent-card>
  <h2>fluent Ember</h2>
  <fluent-text-field placeholder="Enter Some Text"
                  value="{{this.exampleTextField}}"
                  {{on "input" this.onInput}}
  ></fluent-text-field>
  <fluent-button appearance="accent" {{on "click" this.onClick}}>Click
Me</fluent-button>
</fluent-card>
```

With this code in place, you now have Fluent UI Web Components fully binding to data and handling user interactions, all from inside an Ember component.

Congratulations! You're now set up to use Fluent and Ember!

Next steps

- See the components

Use Fluent UI Web Components with React

Article • 11/01/2021 • 5 minutes to read

Fluent UI Web Components can be used in React applications. Let's take a look at how you can set up a project, starting from scratch.

Setting up the React project

First, you'll need to make sure that you have Node.js >= 8.2 and npm >= 5.6 installed. You can learn more and download that on the official site.

With Node.js installed, you can use create-react-app to create a new React project.

```shell
npx create-react-app fluent-app
```

Configuring packages

Next, we'll install the Fluent UI Web Component package, along with supporting FAST libraries. To do that, run this command from your new project folder:

```shell
npm install --save @fluentui/web-components @microsoft/fast-foundation
@microsoft/fast-element @microsoft/fast-react-wrapper
```

Configure create-react-app

create-react-app ships with an eslint rule that makes working with Fluent UI Web Components difficult. There are two changes that will need to be made in the `package.json`:

Set the EXTEND_ESLINT environment variable in start, build, and test scripts

```jsonc
{
  //...
```

```
  "scripts": {
    "start": "EXTEND_ESLINT=true react-scripts start",
    "build": "EXTEND_ESLINT=true react-scripts build",
    "test": "EXTEND_ESLINT=true react-scripts test"
  }
  // ...
}
```

Override the `eslintConfig` field to turn off the 'no-unused-expressions' rule

jsonc

```
{
  //..
  "eslintConfig": {
    "extends": "react-app",
    "rules": {
      "no-unused-expressions": "off"
    }
  }
  //..
}
```

See configuring eslint for more information.

Using the components

With all the basic pieces in place, let's run our app in dev mode with `npm start`. Right now, it displays the React logo and some editing instructions, since we haven't added any code or interesting HTML. Let's change that.

First, open your `src/app.js` file and add the following code:

With all the basic pieces in place, let's run our app in dev mode with `npm start`. Right now, it displays the React logo and some editing instructions, since we haven't added any code or interesting HTML. Let's change that.

First, open your `src/app.js` file and add the following code:

JavaScript

```
import { provideFluentDesignSystem, fluentCard, fluentButton } from
'@fluentui/web-components';
import { provideReactWrapper } from '@microsoft/fast-react-wrapper';
import React from 'react';

const { wrap } = provideReactWrapper(React, provideFluentDesignSystem());
```

```
export const FluentCard = wrap(fluentCard());
export const FluentButton = wrap(fluentButton());
```

This code uses the Fluent Design System to register the `<fluent-card>` and `<fluent-button>` components while automatically wrapping them into React components. Once you save, the dev server will rebuild and refresh your browser. However, you still won't see anything. To get some UI showing up, we need to write some HTML that uses our components. Replace the App component in your `src/app.js` file with the following:

jsx

```
function App() {
  return (
    <FluentCard>
      <h2>Fluent React</h2>
      <FluentButton appearance="accent" onClick={() =>
console.log('clicked')}>
        Click Me
      </FluentButton>
    </FluentCard>
  );
}
```

To add a splash of style, add the following to the `src/App.css`:

css

```
fluent-card {
  padding: 16px;
  display: flex;
  flex-direction: column;
}

h2 {
  font-size: var(--type-ramp-plus-5-font-size);
  line-height: var(--type-ramp-plus-5-line-height);
}

fluent-card > fluent-button {
  align-self: flex-end;
}
```

Congratulations! You're now set up to use Fluent and React!

Using the React Wrapper

Above, we leveraged the `@microsoft/fast-react-wrapper` library to enable seamless integration of Fluent Components. There are a few additional ways to use this API for different web component scenarios.

Wrapping Design System Components

Previously, you've seen that we can wrap a Design System component by passing its registration function to the `wrap` method as follows:

ts

```ts
const { wrap } = provideReactWrapper(React, provideFluentDesignSystem());

export const FluentButton = wrap(fluentButton());
```

This code creates a wrapper that is configured with a React-compatible API and a Design System instance. When passing a Design System as the second parameter, you can then pass component registration functions to the `wrap` helper. The helper will both register the web component with the Design System and wrap it in a type-safe React component, all with a single call.

Alternatively, you can skip providing the Design System to the wrapper, and use the generated registry to manually register all previously wrapped components.

ts

```ts
const { wrap, registry } = provideReactWrapper(React);

export const FluentButton = wrap(fluentButton());

provideFluentDesignSystem().register(registry);
```

The final option is to handle everything by hand:

ts

```ts
const { wrap } = provideReactWrapper(React);

export const FluentButton = wrap(fluentButton());

provideFluentDesignSystem().register(fluentButton());
```

Configuring Custom Events

If the wrapped component uses custom events that you intend to use from React, you will need to manually configure a mapping from React event name to the native event name. Here's an example of what that would look like if you wanted to leverage the FAST MenuItem's `expanded-change` event:

```ts
const { wrap } = provideReactWrapper(React, provideFluentDesignSystem());

export const FluentMenuItem = wrap(fluentMenuItem(), {
  events: {
    onExpandedChange: 'expanded-change',
  },
});
```

Additional Notes

create-react-app

FAST makes use of decorators to define components. At this time, `create-react-app` does not support decorators. This won't be a problem when using components *imported* from FAST because they have already been transpiled by TypeScript - but to *create* components in a `create-react-app` application you'll need to do one of the following:

- Define components without decorators
- Eject `create-react-app` and change Babel to support decorators
- Use an intermediary like react-app-rewired

Data Binding

HTML Attributes

React is capable of rendering custom HTML elements and binding data to them, but it is beneficial to understand *how* React does this. React will apply all *props* to a custom HTML element as *HTML attributes* - including non-primitive types such as arrays and objects. Where some UI libraries provide binding syntaxes to distinguish setting properties, attributes, and events, React does not. This means that it can be very easy to end up with `my-prop="[object Object]"` in your HTML. React is exploring solutions in this issue. See the section on interop layers for a work-around for this issue.

Custom events

React's synthetic eventing system comes with an unfortunate side-effect of being incapable of declaratively applying CustomEvent listeners. Interop layers can be used to address this issue. Alternatively, a `ref` can be used on the custom element to imperatively apply the event listener to the HTML element directly.

Interop layers: @skatejs/val and reactify-wc

@skatejs/val is a small library that wraps React's `createElement` function and provides the ability direct React *props* explicitly to HTML attributes, DOM properties, or to declarative event listeners.

Another good option is reactify-wc. It provides similar capabilities as `@skatejs/val` but does so by creating component wrappers.

TypeScript and TSX support

If you're using TypeScript, you'll need to augment the `JSX.IntrinsicElements` interface to use custom elements in TSX. To do so, create a `custom-elements.d.ts` file in your source directory and add the following:

ts

```ts
// custom-elements.d.ts
declare namespace JSX {
  interface IntrinsicElements {
    /**
     *  React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>,
HTMLElement> allows setting standard HTML attributes on the element
     */
    'my-element': React.DetailedHTMLProps<React.HTMLAttributes<HTMLElement>,
HTMLElement> & {
      'my-attribute-name': string;
    };
  }
}
```

Next steps

- See the components

Use Fluent UI Web Components with Vue

Article • 11/01/2021 • 2 minutes to read

Vue is fun to use and Fluent UI Web Components work great with it. Let's take a look at how you can set up a Vue project, starting from scratch.

Setting up the Vue project

First, you'll need to make sure that you have Node.js installed. You can learn more and download that on the official site.

With Node.js installed, you can run the following command to install the Vue CLI:

```shell
npm install -g @vue/cli
```

With the CLI installed, you have access to the `vue` command-line interface. This can be used to create a new Vue project. For example, to create a new Vue App named "fluent-vue", you would use the following command:

```shell
vue create fluent-vue
```

When prompted to select options, choose "Manually select features". Follow the prompts, answering each question in turn. It is recommended that you select "TypeScript" when prompted.

When the CLI completes, you should have a basic runnable Vue application.

Configuring packages

Next, we'll install the Fluent UI Web Component packages, along with supporting libraries. To do that, run this command from your new project folder:

```shell
npm install --save @fluentui/web-components @microsoft/fast-element lodash-es
```

Using the components

With all the basic pieces in place, let's run our app in dev mode with `npm run serve`. The Vue CLI should build your project and make it available on localhost. Right now, it displays a basic welcome message, since we haven't added any code or interesting HTML. Let's change that.

First, open your `src/main.ts` file and add the following code:

ts

```ts
import { provideFluentDesignSystem, fluentCard, fluentButton } from
'@fluentui/web-components';

provideFluentDesignSystem().register(fluentCard(), fluentButton());
```

This code uses the Fluent Design System to register the `<fluent-card>` and `<fluent-button>` components. Once you save, the dev server will rebuild and refresh your browser. However, you still won't see anything. To get some UI showing up, we need to write some HTML that uses our components. Replace the HTML template in your `components/HelloWorld.vue` file with the following markup:

HTML

```html
<template>
  <fluent-card>
    <h2>{{msg}}</h2>
    <fluent-button appearance="accent" v-on:click="onClick">Click
Me</fluent-button>
  </fluent-card>
</template>
```

Replace your script tag with this:

HTML

```html
<script>
  export default {
    name: 'HelloWorld',
    props: {
      msg: String,
    },
    methods: {
      onClick: function () {
        console.log('clicked!');
```

```
      },
    },
  };
</script>
```

To add a splash of style, replace the `style` tag with this:

HTML

```
<style scoped>
  fluent-card {
    padding: 16px;
    display: flex;
    flex-direction: column;
  }

  h2 {
    font-size: var(--type-ramp-plus-5-font-size);
    line-height: var(--type-ramp-plus-5-line-height);
  }

  fluent-card > fluent-button {
    align-self: flex-end;
  }
</style>
```

You're now set up to use Fluent UI Web Components and Vue!

Next steps

- See the components

Use Fluent UI Web Components with Webpack and Typescript

Article • 11/01/2021 • 4 minutes to read

Fluent UI Web Components work great with TypeScript and Webpack, using a fairly standard setup. Let's take a look at how you can set up a TypeScript+Webpack project, starting from scratch.

Setting up the package

First, let's make a directory for our new project. From the terminal:

```shell
mkdir fluent-webpack
```

Next, let's move into that directory, where we'll set up our project:

```shell
cd fluent-webpack
```

From here, we'll initialize npm:

```shell
npm init
```

Follow the prompts from npm, answering each question in turn. You can always accept the defaults at first and then make changes later in the package.json file.

Next, we'll install the Fluent packages, along with supporting libraries. To do that, run this command:

```shell
npm install --save @fluentui/web-components @microsoft/fast-element lodash-es
```

We also need to install the Webpack build tooling:

```
```

shell

```
npm install --save-dev clean-webpack-plugin ts-loader typescript webpack
webpack-cli webpack-dev-server
```

Adding configuration and source

Now that we've got our basic package and dependencies set up, let's create some source files and get things configured. Since we're going to be writing a bit of code, now is a great time to involve a code editor in the process. If you're looking for a great editor for TypeScript and front-end in general, we highly recommend VS Code.

Open the `fluent-webpack` folder in your favorite editor. You should see your `package.json` along with a `package-lock.json` and a `node_modules` folder.

First, let's create a `src` folder where we'll put all our TypeScript code. In the `src` folder, add a `main.ts` file. You can leave the file empty for now. We'll come back it in a bit.

Next, in the root of your project folder, add a `tsconfig.json` file to configure the TypeScript compiler. Here's an example starter config that you can put into that file:

JSON

```
{
  "compilerOptions": {
    "pretty": true,
    "target": "ES2015",
    "module": "ES2015",
    "moduleResolution": "node",
    "importHelpers": true,
    "experimentalDecorators": true,
    "declaration": true,
    "declarationMap": true,
    "sourceMap": true,
    "noEmitOnError": true,
    "strict": true,
    "outDir": "dist/build",
    "rootDir": "src",
    "lib": ["dom", "esnext"]
  },
  "include": ["src"]
}
```

You can learn more about `tsconfig.json` options in the official TypeScript documentation.

Next, create a `webpack.config.js` file in the root of your project folder with the following source:

JavaScript

```
const { CleanWebpackPlugin } = require("clean-webpack-plugin");
const path = require('path');
module.exports = function(env, { mode }) {
  const production = mode === 'production';
  return {
    mode: production ? 'production' : 'development',
    devtool: production ? 'source-map' : 'inline-source-map',
    entry: {
      app: ['./src/main.ts']
    },
    output: {
      filename: 'bundle.js'
      filename: 'bundle.js',
      publicPath:'/'
    },
    resolve: {
      extensions: ['.ts', '.js'],
      modules: ['src', 'node_modules']
    },
    devServer: {
      port: 9000,
      historyApiFallback: true,
      open: !process.env.CI,
      devMiddleware: {
        writeToDisk: true,
      },
      static: {
        directory: path.join(__dirname, './')
      }
    },
    plugins: [
      new CleanWebpackPlugin()
    ],
    module: {
      rules: [
        {
          test: /\.ts$/i,
          use: [
            {
              loader: 'ts-loader'
            }
          ],
          exclude: /node_modules/
        }
      ]
    }
  }
}
```

This setup uses `ts-loader` to process TypeScript. It will also enable both a production mode and a development mode that watches your source, recompiling and refreshing your browser as things change. You can read more about Webpack configuration in the official Webpack documentation.

To enable easy execution of both our production and development builds, let's add some script commands to our `package.json` file. Find the `scripts` section of your `package.json` file and add the following two scripts:

JSON

```
"scripts": {
  "build": "webpack --mode=production",
  "dev": "webpack serve"
}
```

The `build` script will build your TypeScript for production deployment while the `dev` script will run the development web server so you can write code and see the results in your browser. You can run these scripts with `npm run build` and `npm run dev` respectively.

To complete our setup, we need to add an `index.html` file to the root of our project. We'll start with some basic content as follows:

HTML

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Fluent Webpack</title>
  </head>
  <body>
    <script src="dist/bundle.js"></script>
  </body>
</html>
```

There's nothing special about the HTML yet other than the `script` tag in the `body` that references the `bundle.js` file that our Webpack build will produce.

Using the components

With all the basic pieces in place, let's run our app in dev mode with `npm run dev`. Webpack should build your project and open your default browser with your

`index.html` page. Right now, it should be blank, since we haven't added any code or interesting HTML. Let's change that.

First, open your `src/main.ts` file and add the following code:

ts

```ts
import { provideFluentDesignSystem, fastCard, fastButton } from
'@fluentui/web-components';

provideFluentDesignSystem().register(fastCard(), fastButton());
```

This code uses the Fluent Design System to register the `<fluent-card>` and `<fluent-button>` components. Once you save, the dev server will rebuild and refresh your browser. However, you still won't see anything. To get some UI showing up, we need to write some HTML that uses our components. Replace the contents of the `<body>` in your `index.html` file with the following markup:

HTML

```html
<body>
  <fluent-card>
    <h2>Hello World!</h2>
    <fluent-button appearance="accent">Click Me</fluent-button>
  </fluent-card>
  <style>
    :not(:defined) {
      visibility: hidden;
    }

    fluent-card {
      padding: 16px;
      display: flex;
      flex-direction: column;
    }

    h2 {
      font-size: var(--type-ramp-plus-5-font-size);
      line-height: var(--type-ramp-plus-5-line-height);
    }

    fluent-card > fluent-button {
      align-self: flex-end;
    }
  </style>
  <script src="dist/bundle.js"></script>
</body>
```

After saving your `index.html` file, refresh your browser and you should see a card with text and a button.

Congratulations! You're now set up to use Fluent, TypeScript, and Webpack. You can import and use more components, build your own components, and when you are ready, build and deploy your website or app to production.

Next steps

- See the components

Design Tokens for Fluent UI Web Components

Article • 11/01/2021 • 5 minutes to read

Fluent UI Web Components provides first-class support for Design Tokens and makes setting, getting, and using Design Tokens simple.

What is a Design Token

A Design Token is a semantic, named variable used to describe a Design System. They often describe design concepts like typography, color, sizes, UI spacing, etc. Fluent UI encourages checking out the Design Tokens Community Group for more information on Design Tokens themselves.

Fluent UI Design Tokens

The `@fluentui/web-components` have extensive support for predefined design tokens. See styling the components for details on adjusting or using the existing tokens, or read on to create your own design tokens.

Create a Token

> ⓘ Note
>
> This example uses color because it's an easy concept to describe, but we generally discourage the use of fixed colors as they don't benefit from the **adaptive color system** with support for light and dark mode and other adjustments.

The first step to using a token is to create it:

ts

```ts
import { DesignToken } from '@microsoft/fast-foundation';

export const specialColor = DesignToken.create<string>('special-color');
```

> ⓘ Note

The Fluent UI Web Components are built using Microsoft's `fast-foundation` and `fast-element` libraries, so customizing requires using the original FAST types that underlie the Fluent UI Web Components.

The type assertion informs what types the token can be set to (and what type will be retrieved), and the name parameter will serve as the CSS Custom Property name (more on that later).

Setting Values

A `DesignToken` *value* is set for a `FASTElement` or `HTMLBodyElement` node. This allows tokens to be set to different values for distinct DOM trees:

ts

```ts
const ancestor = document.querySelector('my-element') as FASTElement;
const descendent = ancestor.querySelector('my-element') as FASTElement;

specialColor.setValueFor(ancestor, '#FFFFFF');
specialColor.setValueFor(descendent, '#F7F7F7');
```

Setting a Default Value

A default value can be set for a token, so that the default value is returned from `getValueFor()` in cases where no other token value is found for a node tree.

ts

```ts
specialColor.withDefault('#FFFFFF');
```

Getting Values

Once the value is set for a node, the value is available to use for that node or any descendent node. The value returned will be the value set for the nearest ancestor (or the element itself).

ts

```ts
specialColor.getValueFor(ancestor); // "#FFFFFF"
specialColor.getValueFor(descendent); // "#F7F7F7"
```

Deleting Values

Values can be deleted for a node. Doing so causes retrieval of the nearest ancestor's value instead:

ts

```
specialColor.deleteValueFor(descendent);
specialColor.getValueFor(descendent); // "#FFFFFF"
```

CSS Custom Property emission

Unless configured not to, a DesignToken emits a token to CSS automatically whenever the value is set for an element. In the case when a DesignToken is assigned a derived value, the CSS custom property will also be emitted when any dependent tokens change.

A DesignToken can be configured **not** to emit to a CSS custom property by passing a configuration with `cssCustomPropertyName` set to `null` during creation:

ts

```
DesignToken.create<number>({
  name: 'my-token',
  cssCustomPropertyName: null,
});
```

A DesignToken can also be configured to emit to a CSS custom property that is different than the provided name by providing a CSS custom property name to the configuration:

ts

```
DesignToken.create<number>({
  name: 'my-token',
  cssCustomPropertyName: 'my-css-custom-property-name', // Emits to --my-
css-custom-property-name
});
```

Values with a 'createCSS' method

It is sometimes useful to be able to set a token to a complex object but still use that value in CSS. If a DesignToken is assigned a value with a `createCSS` method on it, the

product of that method will be used when emitting to a CSS custom property instead of the Design Token value itself:

ts

```ts
interface RGBColor {
  r: number;
  g: number;
  b: number;
  createCSS(): string;
}

const extraSpecialColor = DesignToken.create<RGBColor>('extra-special-color');

const value = {
  r: 255,
  g: 0,
  b: 0,
  createCSS() {
    return `rgb(${this.r}, ${this.g}, ${this.b})`;
  },
};

extraSpecialColor.setValueFor(descendent, value);
```

Subscription

`DesignToken` supports subscription, notifying a subscriber when a value changes. Subscriptions can subscribe to *any* change throughout the document tree or they can subscribe changes for specific elements.

Example: Subscribe to changes for any element

ts

```ts
const subscriber = {
  handleChange(record) {
    console.log(`DesignToken ${record.token} changed for element ${record.target}`);
  },
};

specialColor.subscribe(subscriber);
```

Example: Subscribe to changes a specific element

ts

```ts
// ...
const target = document.body.querySelector('#my-element');

specialColor.subscribe(subscriber, target);
```

Subscribers can be unsubscribed using the `unsubscribe()` method:

ts

```ts
// ...
specialColor.unsubscribe(subscriber);
specialColor.unsubscribe(subscriber, target);
```

Using Design Tokens in CSS

Any token can be used directly in a stylesheet by using the Design Token as a CSS directive. Assuming the token value has been set for the element or some ancestor element, the value of the token embedded in the stylesheet will be the token value for that element instance.

ts

```ts
import { css } from '@microsoft/fast-element';

const styles = css`
  :host {
    background: ${specialColor};
  }
`;
```

At runtime, the directive is replaced with a CSS custom property, and the Directive ensures that the CSS custom property is added for the element.

Derived Design Token Values

In the examples above, the design token is always being set to a simple string value. But, we can also set a Design Token to be a function that *derives* a value. A derived value receives the target element as its only argument and must return a value with a type matching the Design Token:

ts

```ts
const token = DesignToken.create<number>('token');
token.setValueFor(target, element => 12);
```

The above example is contrived, but the target element can be used to retrieve *other* Design Tokens:

Example: A derived token value that uses another design token

ts

```ts
const foregroundColor = DesignToken.create<string>('foreground-color');

foregroundColor.setValueFor(target, element =>
  specialColor.getValueFor(element) === '#FFFFFF' ? '#2B2B2B' : '#262626',
);
```

For derived Design Token values, any change to dependent tokens will force the derived value to update (and update the CSS custom property if applicable). The same is true if an observable property is used by the derived value:

ts

```ts
import { observable } from '@microsoft/fast-element';

class ThemeManager {
  @observable
  theme: 'blue' | 'red' = 'blue';
}

const themeManager = new ThemeManager();

specialColor.setValueFor(target, () => (themeManager.theme === 'blue' ?
'#0000FF' : '#FF0000'));

themeManager.theme = 'red'; // Forces the derived tokens to re-evaluate and
CSS custom properties to update if applicable
```

Aliasing Design Tokens

In some design systems, Design Tokens may have complex hierarchies with tokens referencing other tokens. This can be accomplished by setting a Design Token to another Design Token.

ts

```ts
const specialColor = DesignToken.create<string>('special-color');
const buttonSpecialColor = DesignToken.create<string>('button-special-
color');
```

```
specialColor.setValueFor(target, '#EDEDED');
buttonSpecialColor.setValueFor(target, specialColor);

buttonSpecialColor.getValueFor(target); // "#EDEDED"
```

Next steps

- High contrast
- Integrations

Windows high contrast mode

Article • 11/01/2021 • 4 minutes to read

Fluent UI Web Components provides first-class support for accessibility including built in support for Windows High Contrast mode.

Styling components using forced-colors

High contrast mode uses the CSS media feature, forced-colors. When `forced-colors` is set to `active`, the user agent will apply a limited color palette to the component.

Simple forced-color example

```css
@media (forced-colors: active) {
  :host {
    background: ButtonFace;
  }
}
```

Fluent UI Web Components implements FAST's forcedColorsStylesheetBehavior utility function that is used to construct `forced-colors` in a stylesheet. This function is passed to the `withBehavior` function from the `css` tagged template object.

> **Note**
>
> The reason for this behavior is to avoid the runtime cost of applying `forced-color` style rules when the UA does not match the `forced-colors` @media query. Fluent UI Web Components exposes a behavior that conditionally adds and removes stylesheets based on this media query, so forced-colors' stylesheets can then be conditionally applied where necessary.

Forced-color example

```ts
export const ComponentStyles = css`
  /* ... */
`.withBehaviors(
  forcedColorsStylesheetBehavior(
    css`
      :host {
        background: ButtonFace;
      }
    `,
  ),
);
```

System Color Keyword

In `forced-colors` mode, the colors on the component are reduced to a limited color palette chosen by the user. The System Color keywords defined by the CSS Color Module Level 4 specification expose these user-chosen colors.

Fluent UI Web Components (via FAST components) provides a SystemColors enum to use when setting the color value keywords in a `forced-colors` stylesheet.

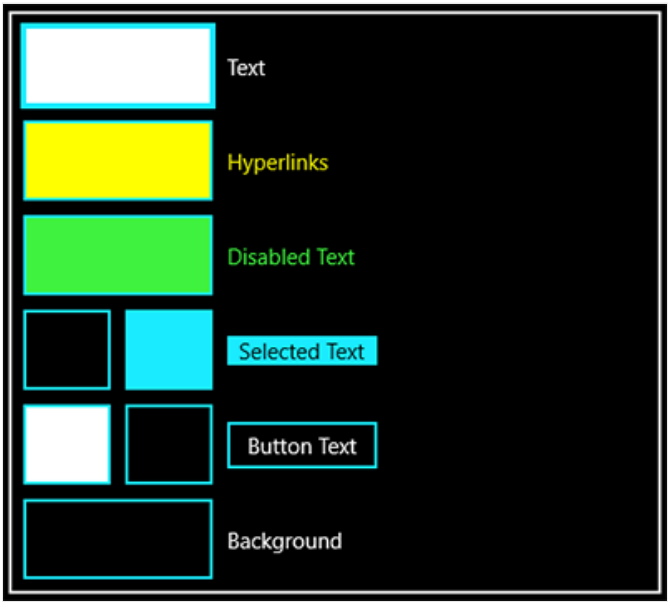
System color example

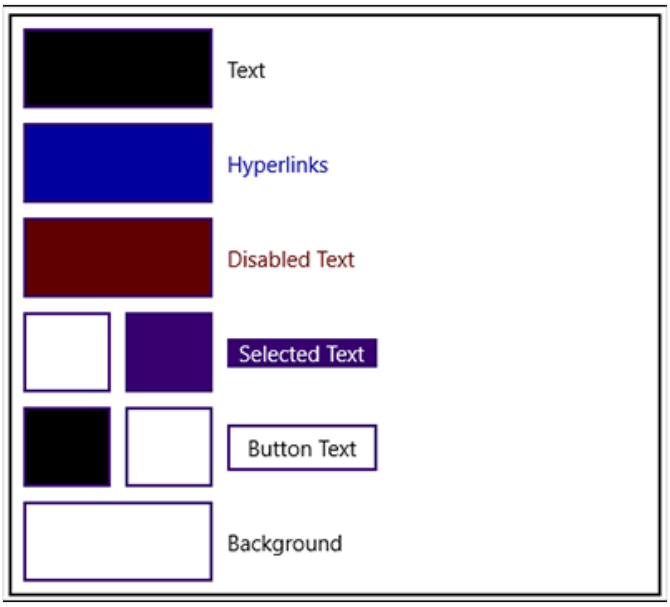
ts

```ts
export const ComponentStyles = css`
  /* ... */
`.withBehaviors(
  forcedColorsStylesheetBehavior(
    css`
      :host {
        background: ${SystemColors.ButtonFace};
      }
    `,
  ),
);
```

Forced colors and Windows High Contrast themes

`forced-colors` works with Windows high contrast mode in Windows, located in Ease of Access within Settings. There are two default themes to test high contrast, `High Contrast Black` and `High Contrast White`.



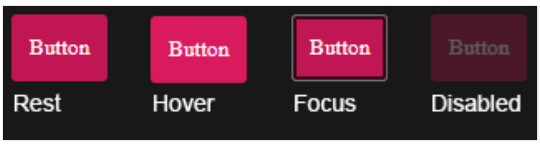


Here is a 1:1 map between the `forced-colors` keywords and Windows high contrast resource names.

forced-colors	Windows
`CanvasText`	`Text`
`LinkText`	`Hyperlinks`
`GrayText`	`Disabled Text`
`HighlightText` `Highlight`	`Selected Text`
`ButtonText` `ButtonFace`	`Button Text`
`Canvas`	`Background`

Quick demo

Here is a simple example of adding high contrast to style an accent button. It has selectors for rest, active, hover, focus, and disabled. Note: the base styles shown in the following images use the FAST coloring, but High Contrast for Fluent UI is the same.

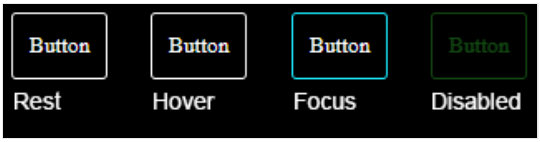


ts

```ts
export const AccentButtonStyles = css`
  :host([appearance='accent']) {
    background: ${accentFillRest};
    color: ${foregroundOnAccentRest};
  }
  :host([appearance='accent']:hover) {
    background: ${accentFillHover};
  }
  :host([appearance='accent']:active) .control:active {
    background: ${accentFillActive};
  }
  :host([appearance="accent"]) .control:${focusVisible} {
```

```
    box-shadow: 0 0 0 calc(${focusStrokeWidth} * 1px) inset ${focusStrokeInner};
  }
  :host([appearance='accent'][disabled]) {
    opacity: ${disabledOpacity};
    background: ${accentFillRest};
  }
`;
```

When high contrast is enabled, the system will try to apply the correct color. In the case of this accent button, the system is missing a few things. We do not have a background, `rest` and `hover` states are the same, `focus` is not following the button's `focus` design, and the `disabled` state is too dim.

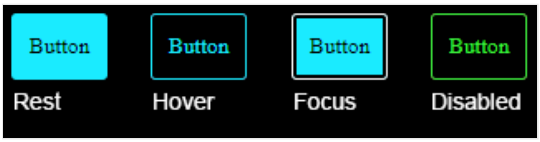


To fix this, we will pass a forcedColorsStylesheetBehavior object to `withBehaviors`, using similar selectors, and setting property values with the `SystemColors` keyword.

ts

```ts
export const AccentButtonStyles = css`
  /* ... */
`.withBehaviors(
  forcedColorsStylesheetBehavior(
    css`
      :host([appearance='accent']) .control {
        forced-color-adjust: none;
        background: ${SystemColors.Highlight};
        color: ${SystemColors.HighlightText};
      }
      :host([appearance='accent']) .control:hover,
      :host([appearance='accent']:active) .control:active {
        background: ${SystemColors.HighlightText};
        border-color: ${SystemColors.Highlight};
        color: ${SystemColors.Highlight};
      }
      :host([appearance="accent"]) .control:${focusVisible} {
        border-color: ${SystemColors.ButtonText};
        box-shadow: 0 0 0 2px ${SystemColors.HighlightText} inset;
      }
      :host([appearance='accent'][disabled]),
      :host([appearance='accent'][disabled]) .control,
      :host([appearance='accent'][disabled]) .control:hover {
        background: ${SystemColors.ButtonFace};
        border-color: ${SystemColors.GrayText};
        color: ${SystemColors.GrayText};
        opacity: 1;
      }
    `,
  ),
);
```

After adding `forced-colors` and applying `SystemColors` keywords, the accent button now uses `Highlight` as a background for its `rest` state. On the `hover` and `active` states, the background and color from the `rest` state are swapped. A double border treatment is applied when in the `focus` state, and the `disabled` has opacity set to 1 and uses the disabled color, `GrayText`, for color on the border and content.



> ⓘ Note
>
> forced-color-adjust, controls whether the UA system theme color override, should be applied to an element and its descendants. The example is set to `none`, because we are overriding to remove the backplate on the text content in the control, that the UA sets on text elements.

Further resources

Color contrast comparison chart

To help determine whether a pair of high contrast colors will meet a color luminosity contrast ratio of at least 10:1, this table uses the high contrast theme color resource names you see in Windows Ease of Access.

How to read this table:

- **YES** - indicates that it is safe to assume this pair of colors will meet high contrast requirements, even in custom themes.
- `YES*` - indicates that this specific pair of colors meets the high contrast requirements in both High Contrast Black and High Contrast White themes.
- NO - indicates that you should never use this pair of colors as they do not meet high contrast requirements in High Contrast Black and High Contrast White themes.

	Text	**Hyperlink**	**Disabled Text**	**Selected Text (Foreground)**	**Selected Text (Background)**	**Button Text (Foreground)**	**Button Text (Background)**	**Background**
Text	NO	NO	NO	NO	NO	NO	**YES**	**YES**
Hyperlink	NO	NO	NO	`YES*`	NO	NO	`YES*`	**YES**
Disabled Text	NO	NO	NO	`YES*`	NO	NO	**YES**	**YES**
Selected Text (Foreground)	NO	`YES*`	`YES*`	NO	**YES**	`YES*`	NO	NO
Selected Text (Background)	NO	NO	NO	**YES**	NO	NO	`YES*`	`YES*`
Button Text (Foreground)	NO	NO	NO	`YES*`	NO	NO	**YES**	**YES**
Button Text (Background)	**YES**	`YES*`	**YES**	NO	`YES*`	**YES**	NO	NO
Background	**YES**	**YES**	**YES**	NO	`YES*`	**YES**	NO	NO

Microsoft Edge blog

Microsoft Edge blog has excellent and in-depth information on styling for Windows high contrast using forced-colors. Styling for Windows high contrast with new standards for forced colors

Next steps

- Integrations
- Resources

Browser support

Article • 11/01/2021 • 2 minutes to read

The following browsers have native support for the Web Components features used by fast-element and our components:

- Microsoft Edge 79+
- Mozilla Firefox 63+
- Google Chrome 67+
- Apple Safari 10.1+
- Opera 41+
- iOS Safari 10.3+
- Android Browser 81+
- Opera Mobile 46+
- Chrome for Android 81+
- Firefox for Android 68+
- Samsung Internet 6.2+x

Next steps

- Integrations
- Frequently Asked Questions

Frequently Asked Questions

Article • 11/01/2021 • 3 minutes to read

What's the difference between Fluent and FAST?

Fluent is Microsoft's design language, independent of any particular implementation technology. FAST is an agnostic tech stack that enables implementing web components, design systems, and apps. `fast-element` is the lowest level of FAST, providing core features for building performant web components. `fast-foundation` is layered on top of `fast-element` and provides primarily two things: core features for building design systems, and a core set of components that are design-system-independent. With `fast-foundation` you can implement any design system. For example, you could implement Fluent Design, Material Design, Lightning Design, Bootstrap, etc. Once the design system is implemented, it can connect with any component built with `fast-element` or `fast-foundation` to enable a particular component to present itself using the visual language of the chosen design system. The FAST team ships two design systems: `fast-components`, which provide our team's own FAST Design system, and `@fluentui/web-components`, located in the Fluent UI repo, which provides Microsoft's Fluent Design system. If you want to build an app or site with Fluent Design, and you want to use web components as a technology solution, you can use `@fluentui/web-components` to meet that need today.

Who is behind FAST?

The Microsoft FAST team builds and maintains all the `@microsoft/fast-*` packages as well as the `@fluentui/web-components` package. We are a collection of UX Engineers and Designers who are passionate about solving real-world UX challenges using web standard technologies. You can find us on GitHub.

How does `fast-element` compare to "Framework X"?

At this time, `fast-element` has a focus that's a bit different from the typical front-end framework. Rather than focusing on being a "mega SPA framework", `fast-element` endeavors to enable the creation of web components. As a result, you can use `fast-`

`element` or any component built on `fast-element` in tandem with your favorite front-end framework.

What are Web Components?

"Web Components" is an umbrella term that refers to a collection of web standards focused on enabling the creation of custom HTML elements. Some of the standards that are under the umbrella include the ability to define new HTML tags, plug into a standard component lifecycle, encapsulate HTML rendering and CSS, parameterize CSS, skin components, and more. Each of these platform features is defined by the W3C and has shipped in every major browsers today.

Why does FAST have components that are already available in HTML?

Various members of our community have wondered why FAST has components that seem to mirror native elements. Examples include `fast-anchor`, `fast-button`, and `fast-divider`. There are several reasons for this:

- CSS Encapsulation - By using Shadow DOM, FAST is able to provide a set of styles for these elements and guarantee that they will not conflict with your site or app. Your site styles will not break FAST and FAST will not break your site.
- CSS Behaviors - Custom elements enable FAST to dynamically add/remove styles based on runtime conditions, such as toggling high contrast mode. They also enable components to hook into the *design system* and respond to design changes over time.
- Enhanced Anatomies - The FAST team refers to the DOM structure of a component as its "anatomy". This is an important detail of a component spec. Our research as part of OpenUI has revealed common anatomies across many design systems and component libraries that are not reflected by a single standard HTML element. We leverage this research to design the structure of our FAST components so that they are built in a way that meets real-world needs, particularly regarding composition with other components. Some basic components, such as `anchor`, benefit from an expanded anatomy, based on industry usage. Through custom elements, we are able to implement this anatomy without inflicting an HTML authoring burden on our community.

Next steps

- Integrations
- Resources