This year, we're giving you new tools to *sculpt* a dynamic web

We've been crafting new features with *you* in mind.

Ready to see what we molded in 2025? The Chrome DevRel team will guide you through 22 CSS and UI features that landed on the Web Platform, fresh from the kiln

Customizable Components

The workshop was hot this year. We took the decades-old problem of styling dropdowns and fired it to perfection. We also delivered new core blocks like native anchor positioning and carousel scroll APIs.

Invoker Commands
Dialog Light Dismiss
popover=hint
Customizable select
::scroll-marker/button()
scroll-target-group
Anchored container queries
Interest invokers

Invoker Commands

Show a `<dialog>` modally (and more) without JavaScript!

Try the demo Chrome for Developers MDN

To open a `<dialog>` modally by clicking a `<button>` you typically need an `onclick` handler that calls the `showModal` method on that `<dialog>`.

```
<button onclick="document.querySelector('#my-dialog').showModal()
<dialog id="my-dialog">…</dialog>
```

With invoker commands–available from Chrome 135–buttons can now perform actions on other elements declaratively, without the need for any JavaScript.

```
<button commandfor="my-dialog" command="show-modal">Show Dialog</
<dialog id="my-dialog">…</dialog>
```

The `commandfor` attribute takes an ID—similar to the `for` attribute—while `command` accepts built-in values, enabling a more portable and intuitive approach.

Demo to show and hide a `<dialog>` and a `[popover]` with Invoker Commands. For browsers without support a polyfill is loaded.

Currently it is possible to send commands to `[popover]`s and `<dialog>` elements, with more types of elements possibly coming in the future. The commands to send to an element are mirrored after their JavaScript counterparts:

- `show-popover`: `el.showPopover()`
- `hide-popover`: `el.hidePopover()`
- `toggle-popover`: `el.togglePopover()`

It is also possible to set up custom commands to send to elements. These custom commands are prefixed by two dashes and are handled by the `toggle` event.

```
<button commandfor="some-element" command="--show-confetti">🎉</
```

```
document.querySelector('#some-element').addEventListener('comman
  if (e.command === "--show-confetti") {
    // …
  }
});
```

A polyfill for Invoker Commands is available.

Dialog Light Dismiss

Bringing a nice Popover API feature to `<dialog>`.

Try the demo **MDN**

<dialog closedby> 👍 12

Limited availability

One of the nice features introduced by the Popover API is the light dismiss

From Chrome 134, this light dismiss behavior is also available on `<dialog>`, through the new `closedby` attribute which controls the behavior:

- `<dialog closedby="none">`: No user-triggered closing of dialogs at all. This is the default behavior.
- `<dialog closedby="closerequest">`: Pressing ESC (or other close trigger) closes the dialog
- `<dialog closedby="any">`: Clicking outside the dialog, or pressing ESC, closes the dialog. Similar to `popover="auto"` behavior.

Demo that compares behavior of the various values for `closedby`.

popover=hint

Ephemeral popovers that don't close others.

Try the demo **Chrome for Developers** **MDN**

popover="hint" 👍 12

Limited availability

Hint popovers with `popover="hint"` are a new type of HTML popover designed for ephemeral layered UI patterns, such as tooltips or link previews. Opening a hint popover does not close other open auto or manual popovers, allowing layered UI elements to coexist. Hint popovers can also exist on links (`<a>` tags), unlike auto and manual popovers which require activation from button elements.

Set this up like any other popover:

```
<button interestfor="callout-1"></button>
<div id="callout-1" popover=hint>
  Product callout information here.
</div>
```

workarounds. This pairing allows for dual-purpose interaction patterns (for example, hover to preview, click to navigate) and better management of multiple on-screen layers.

0:00

Customizable select

You can finally style HTML select elements with CSS.

Try the demo Chrome for Developers MDN

Limited availability

The time has finally arrived: you can now fully customize the HTML `<select>` element using CSS!

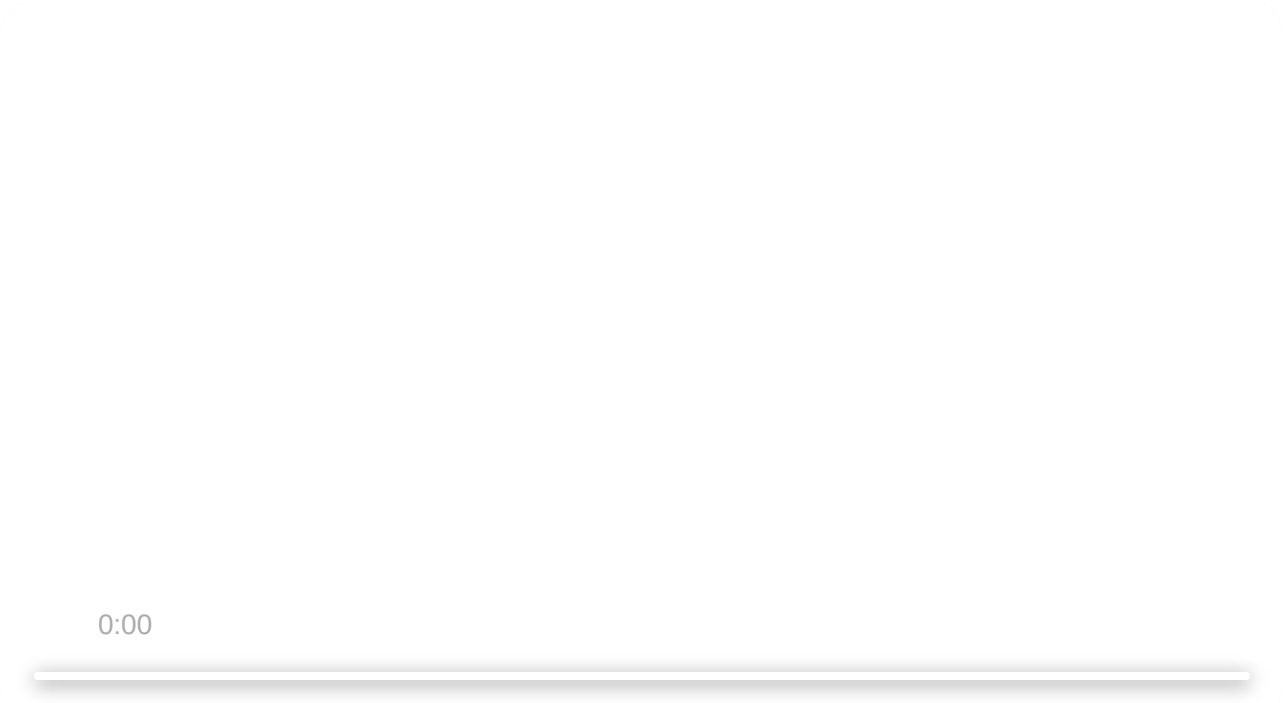
To get started, apply the `appearance: base-select` CSS property to your `<select>` element. This will switch it to a new, minimal state that's optimized for customization.

```
select {
&::picker(select) {
    appearance: base-select;
  }
}
```

Using base-select unlocks several powerful features including complete CSS customization. Every part of the select element, including the button, the dropdown list, and the options, can be styled with CSS. You can change colors, fonts, spacing, and even add animations to create a unique look and feel that matches your site's design.

clipped by parent containers. The browser also automatically handles positioning and flipping the dropdown based on available space in the viewport.

The new select also enables you to include and properly render HTML elements like `<img>` and `<span>` directly inside of the `<option>` elements. This means you can do something as simple as adding flag icons next to a country picker, or something as complex as creating a profile selection where you can see an icon, name, email, and ID. As long as you are not including interactive elements such as links, which are not allowed inside of customizable selects, you get full control over creating visually rich dropdown menus.



Another neat thing you can do with customizable select is use the new `<selectedcontent> element`. This element reflects the HTML content of the selected option. For complex selects, setting `display: none` on specific elements within `<selectedcontent>` lets you show part of the option content in the select button, or even just an icon to represent the selection. In the monster picker, you can hide the monster skills description by setting:

```
selectedcontent .description {
  display: none;
}
```

RESOURCES:

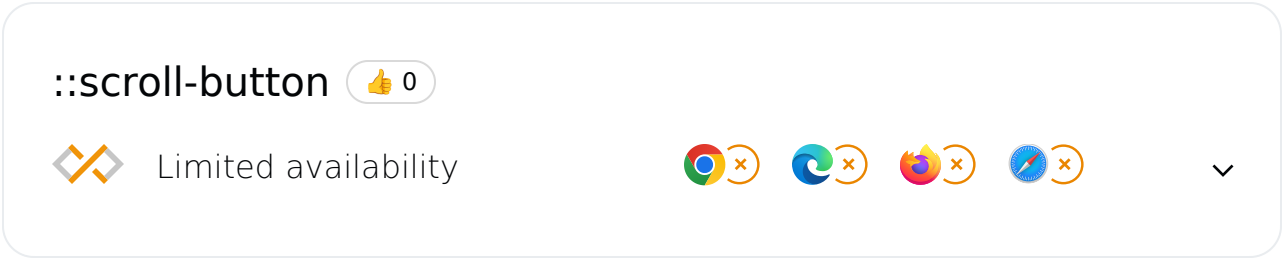
- Codepen collection of select demos
- OpenUI Explainer

::scroll-marker/button()

Carousel scroll affordances with native CSS pseudo-elements.

This year, creating carousels and other scrolling experiences in CSS became much easier with the introduction of two new pseudo-elements: `::scroll-button()` and `::scroll-marker()`. These features let you create native, accessible, and performant carousels with just a few lines of CSS, no JavaScript required.

A carousel is essentially a scrollable area with added UI affordances for navigation: buttons to scroll back and forth, and markers to indicate the current position and allow direct navigation to a specific item.



The `::scroll-button()` pseudo-element creates browser-provided, stateful, and interactive scroll buttons. These buttons are generated on a scroll container

You can create buttons for any scroll direction: `left`, `right`, `up`, or `down`, as well as logical directions like `block-start` and `inline-end`. When a scroll button is activated, it scrolls the container by approximately 85% of its visible area.

```
.carousel::scroll-button(left) {
  content: "←" / "Scroll Left";
}

.carousel::scroll-button(right) {
  content: "□" / "Scroll Right";
}
```

Scroll markers 👍 0

 Limited availability 

The `::scroll-marker` pseudo-element represents a marker for an element

in the scroller. This is useful for creating dot navigation for a carousel or a table of contents for a long document.

Like `::scroll-button()`, `::scroll-marker`s are fully stylable with CSS. You can use images, text, or even counters to create a variety of marker styles. Additionally, the `:target-current` pseudo-class styles the active ("current") marker that aligns with the currently-scrolled-to item.

```
.carousel {
  scroll-marker-group: after;
}

.carousel > li::scroll-marker {
  content: ' ';
  width: 1em;
  height: 1em;
  border: 1px solid black;
  border-radius: 50%;
}

.carousel > li::scroll-marker:target-current {
  background: black;
}
```

Here is a demo that combines both `::scroll-button()` and `::scroll-marker()` to create a simple carousel:

Here is a more complex carousel that makes use of anchor positioning and scroll state queries:

RESOURCES:

- Carousel configurator
- Carousel gallery
- Make accessible carousels

scroll-target-group

Turn a list of anchor links into connected scroll-markers.

Try the demo MDN

In addition to the `::scroll-button()` and `::scroll-marker` pseudo-elements, CSS carousels includes another neat feature: `scroll-target-group`. This designates an element as a container for a group of navigation items, like a table of contents. Use this to transform a manually-created list of anchor links into scroll-markers which can be used to navigate the page.

Pair `scroll-target-group` with the `:target-current` pseudo-class to style the anchor element whose target is currently visible. This gives you the power of `::scroll-marker` from the CSS Carousel API, but with the flexibility of using your own HTML elements for the markers, giving you much more control over their styling and content.

To create a scroll-spy navigation, you need two things: A list of anchor links that point to different sections of your page. The `scroll-target-group: auto` property applied to the container of those links.

The following example creates a "scroll-spy" highlighting where you are on a page in an overview, or table of contents.

```
<nav class="toc">
  <ul>
    <li><a href="#section-1">Section 1</a></li>
    <li><a href="#section-2">Section 2</a></li>
    <li><a href="#section-3">Section 3</a></li>
  </ul>
</nav>

<main>
```

```
  <section id="section-3">...</section>
</main>
```

The following CSS creates the scroll-target-group, then styles the table of contents. The link corresponding to the section currently in view will be red and bold.

```
.toc {
  scroll-target-group: auto;
}

.toc a:target-current {
  color: red;
  font-weight: bold;
}
```

Anchored container queries

Style elements based on their anchor position.

Try the demo **Chrome for Developers**

Last year's CSS Wrapped covered CSS anchor positioning: an exciting update that changes the way you can position elements relative to each other. And since that coverage, it became a part of Interop 2025 , and browser support expanded.

However, while CSS could move an element to a fallback position, it had no way of knowing which fallback was chosen. This meant that if your tooltip flipped from the bottom to the top of the screen, the arrow would still be pointing the wrong way. This is now resolved with anchored container queries.

Anchor queries can be created with two steps:

- First, apply `container-type: anchored` to the positioned element, like your tooltip. This enables the element to be "aware" of its anchor position fallback.
- Next, use the `anchored(fallback: ...)` function within an `@container` block to style any child of your positioned element based on the active fallback value.

When you specify a fallback value, it can either be a custom fallback that you name and specify, or it can be one of the browser defaults like `flip-block`, or `flip-inline`.

Here's a quick demo of how you can use anchored container queries to automatically flip a tooltip's arrow when its position changes:

```
/* The positioned element (tooltip) */
.tooltip {
  position: fixed;
  position-anchor: --my-anchor;
  position-area: bottom;
  /* Reposition in the block direction */
  position-try-fallbacks: flip-block;

  /* Make it an anchored query container */
  container-type: anchored;

  /* Add a default "up" arrow */
  &::before {
    content: '▲';
    position: absolute;
    /* Sits on top of the tooltip, pointing up */
    bottom: 100%;
  }
}

/* Use the anchored query to check the fallback */
@container anchored(fallback: flip-block) {
  .tooltip::before {
    /* The 'top' fallback was used, so flip the arrow */
    content: '▼';
    bottom: auto;
    /* Move the arrow below the tooltip */
    top: 100%;
  }
}
```

0:00

This is a huge win for anchor positioning and component libraries, enabling more robust and self-contained UI elements with less code.

Interest invokers

Try the demo MDN

Interest invokers 👍 3

Limited availability

Hover and focus-triggered UI is everywhere on the web, from tooltips to rich hovercards and page previews. While this pattern often works well for mouse users, it can be inaccessible to other modalities like touchscreen. Additionally developers have to manually implement the logic for each input type, leading to inconsistent experiences.

The new `interestfor` attribute solves this by providing a native, declarative way to style an element when users "show interest" in it without fully activating it.

It's invoked similarly to the `commandfor` attribute, but, instead of a click, `interestfor` is activated when a user "shows interest" in an element, such as by hovering over it with a mouse or focusing it with a keyboard. When paired with `popover="hint"`, it becomes incredibly easy to create layered UI elements like tooltips and hovercards without any custom JavaScript.

```
<button interestfor="callout-1"></button>

<div id="callout-1" popover="hint">
  Product callout information here.
</div>
```

Note: Unlike command invokers, which only work on button elements, interest invokers can be set on links (`<a>` tags) as well as buttons.

Here's a demo that uses `interestfor` to create product callouts on an image. Hovering over the buttons on the image will reveal more information about each product.

Try the demo Chrome for Developers MDN

Next-gen Interactions

With this new interaction toolkit, you can now animate between pages with view transitions and sculpt gorgeous, scroll-based experiences.

Scroll-
Tree counting functions
scrollIntoView() container
Nested View Transition Groups
DOM State-Preserving Move

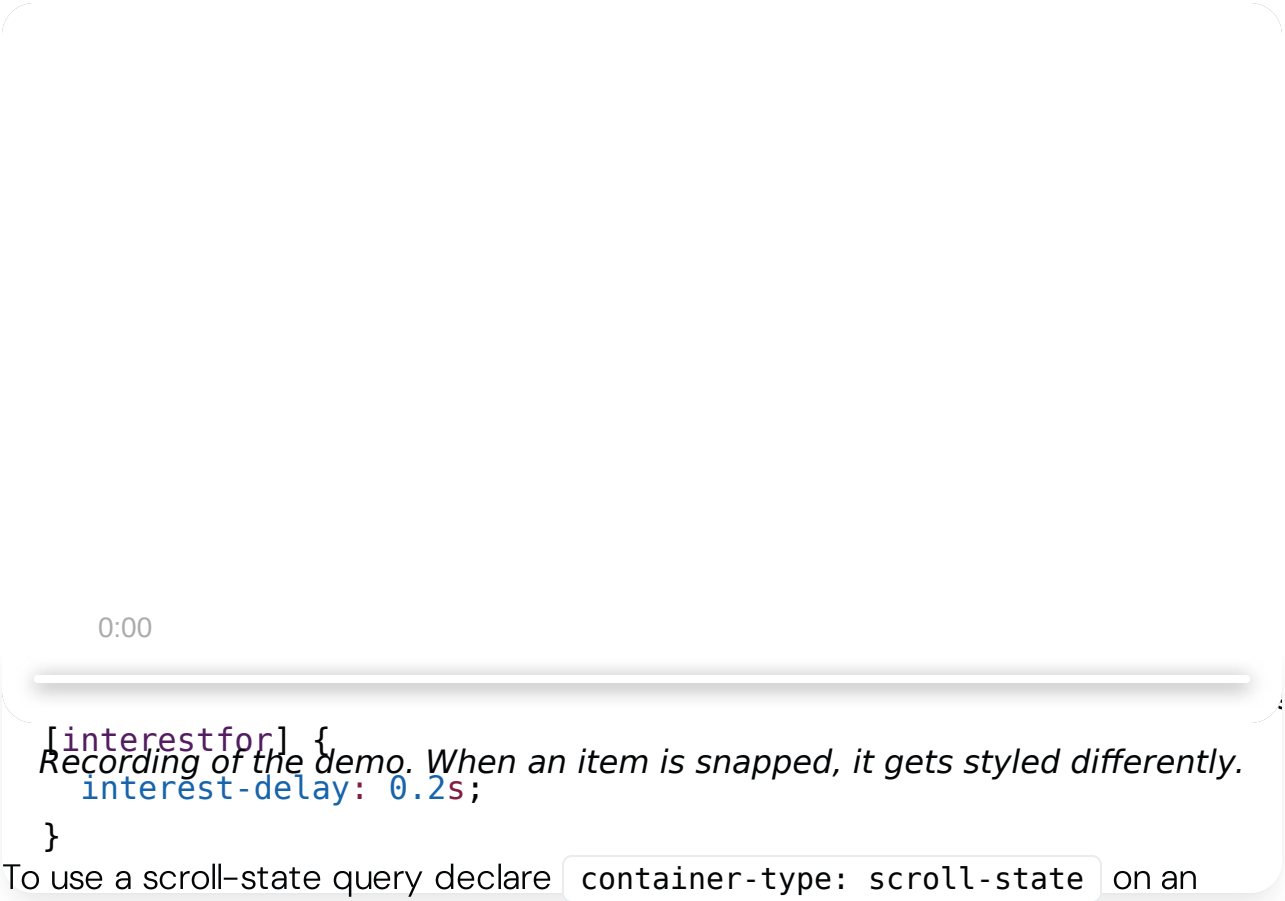
Scroll-state queries

Style descendants based on whether something is scrollable, stuck, or snapped.

Limited availability

To determine if an element is stuck, snapped, or scrollable you could use a bunch of JavaScript … which isn't always easy to do because you have to attach timeouts to scroll events and so on.

Thanks to scroll-state queries–available from Chrome 133–you can use CSS to declaratively, and more performantly, style elements in these states.



Recording of the demo. When an item is snapped, it gets styled differently.

```
[interestfor] {
  interest-delay: 0.2s;
}
```

To use a scroll-state query declare `container-type: scroll-state` on an element.

```
.parent {
  container-type: scroll-state;
}
```

Once you have that in place, children of that element can then query whether

- Stuck state: when the element is stuck.
- Snapped state: when the element is snapped.
- Scrollable state: when the element is overflowing.

For example, to style the snapped element differently, use the `snapped` scroll-state-query:

```
.scroller {
  overflow-x: scroll;
  scroll-snap-type: x mandatory;

  > div {
    container-type: scroll-state;
    scroll-snap-align: center;

    @supports (container-type: scroll-state) {
      > * {
        transition: opacity .5s ease;

        @container not scroll-state(snapped: x) {
          opacity: .25;
        }
      }
    }
  }
}
```

RESOURCES:

- Open UI explainer
- `interestfor` Polyfill

A demo that highlights the currently snapped item. Other, non-snapped, items have a reduced opacity.

Staggered animations, anyone?

Try the demo Chrome for Developers MDN

sibling-count() and sibling-index() 👍 13

Limited availability

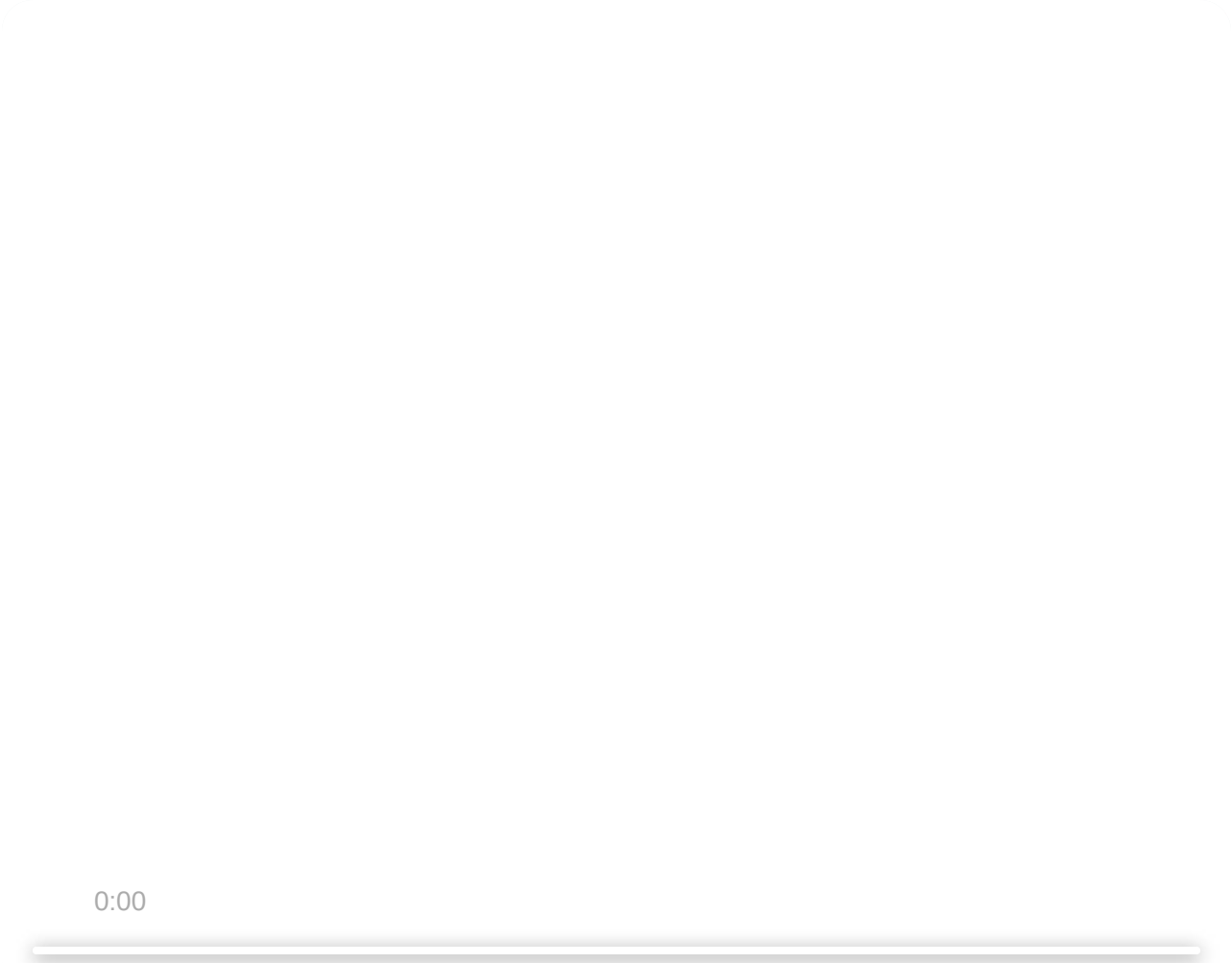
The usual method to create staggered animations for list items, where each item appears sequentially, requires you to count DOM elements and hard-code these values into custom properties (for example, `--index: 1;`, `--index: 2;`) using `:nth-child` selectors. This method is cumbersome, fragile, and not scalable, especially when the number of items changes dynamically.

The new `sibling-index()` and `sibling-count()` functions make your life easier here, as these functions provide native awareness of an element's position among its siblings. The `sibling-index()` function returns a 1-based integer representing the element's position, while `sibling-count()` returns the total number of siblings.

These let you write concise, mathematical formulas for layouts and animations that automatically adapt to the number of elements in the DOM.

```
li {
  /* Create a staggered delay. */
  /* We subtract 1 because sibling-index() starts at 1, */
  /* ensuring the first item starts immediately (0s). */
  transition: opacity 0.25s ease, translate 0.25s ease;
  transition-delay: calc(0.1s * (sibling-index() - 1));

  @starting-style {
    opacity: 0;
    translate: 1em 0;
```

Demo showing a staggered entry animation on the 4 images. Hit the shuffle button to randomize the order.



Recording of the demo.

scrollIntoView() container

Sometimes, scrolling only the nearest ancestor scroller is all you want.

Try the demo MDN

The `container` option for `Element.scrollIntoView` lets you perform a `scrollIntoView` only scrolling the nearest ancestor scroll container. This is extremely useful if you have nested scroll containers. With the option set to `"nearest"`, calling `scrollIntoView` won't scroll all of the scroll containers to the viewport.

```
slideList.addEventListener('click', (evt) => {
  // scrollIntoView will automatically determine the position.
  evt.target.targetSlide.scrollIntoView({container: 'nearest', be
});
```

Recording showing a `scrollIntoView` action without and with `container` set to `"nearest"`

Demo featuring a JavaScript-based carousel that uses `scrollIntoView` to scroll to the specific slide in the carousel. Use the toggle at the top left to control whether `container: "nearest"` should be used or not.

Nested View Transition Groups

Try the demo Chrome for Developers

Nested view transition groups is an extension to view transitions that lets you nest `::view-transition-group` pseudo-elements within each other.

When view transition groups are nested, instead of putting them all as siblings under a single `::view-transition` pseudo-element, it's possible to retain 3D and clipping effects during the transition.

To nest `::view-transition-group` elements in another group, use the `view-transition-group` property on either the parent or children.

```
.card {
  view-transition-name: card;
  overflow: clip;
}

.card img {
  view-transition-name: photo;
  view-transition-group: nearest;
}
```

The nested groups get placed inside a new `::view-transition-group-children(…)` pseudo-element in the tree. To reinstate the clipping used in the original DOM, apply `overflow: clip` on that pseudo-element.

```
::view-transition-group-children(card) {
  overflow: clip;
}
```

Demo for Nested View Transition Groups. Without nested view transition groups, the avatar and name don't rotate along with the card. But when the option is checked, the 3D effect can be restored.

For browsers with no support, check out this recording:

Recording of the demo showing the demo. It shows the behavior without and with nested view transition groups.

DOM State-Preserving Move

Move iframes and videos across the DOM without reloading them.

Try the demo Chrome for Developers MDN

moveBefore() 👍 2

Using `insertBefore` to move an element in the DOM is destructive. If you move a playing video or an iframe using `insertBefore`, it reloads and loses its state completely.

However, from Chrome 133, you can use `moveBefore`. It works exactly like `insertBefore`, but it keeps the element alive during the move.

```
const $newSibling = getRandomElementInBody();
const $iframe = document.querySelector('iframe');
document.body.moveBefore($iframe, $newSibling);
```

This means videos keep playing, iframes don't reload, CSS animations don't restart, and input fields keep their focus—even while you are actively reparenting them across your layout.

Optimized ergonomics

These modules aren't just plug-and-play; they're true chameleons, allowing users to redefine their interface, functionality, and aesthetic down to the atomic level.

Advanced attr() function
ToggleEvent.source
text-box features
shape() function
if() statements
Custom Functions
Expanded range syntax
Stretch sizing keyword
corner-shape

Advanced attr() function

Typed values for `attr()` beyond simple strings.

Try the demo Chrome for Developers MDN

attr() 👍 16

Demo to compare behavior of `insertBefore` and `moveBefore`.

Limited availability

For browsers with no support, check out this recording.

Previously, `attr()` could only be used within the content property of pseudo-elements and could only return values as a CSS string. The updated `attr()` function expands its capabilities, allowing `attr()` to be used with any CSS property, including custom properties. It can now also parse attribute values into various data types beyond just strings, like colors, lengths, and custom identifiers.

With the new attribute, you can set an element's `color` property based on a `data-color` attribute, parsing it as a `<color>` type with a fallback.

```
div {
  color: attr(data-color type(<color>), red);
}
```

Recording of the demo showing a YouTube embed that is playing. When the `iframe` gets moved with `moveBefore`, the video keeps playing. When it gets moved with `insertBefore`, the iframe reloads.

To solve a common UI challenge, you can dynamically set the `view-transition-name` for multiple elements using their `id` attribute, parsed as a `<custom-ident>`. This avoids repetitive CSS rules for each element.

```
  view-transition-name: attr(id type(<custom-ident>), none);
  view-transition-class: card;
}
```

Finally, this demo shows how to use the `attr()` function in multiple ways. First use the `data-rating` to determine a percent-fill to visually fill the star mask and represent the rating. Then use the same data attribute in the `content` property to insert the value in a pseudo-element.

```
.star-rating {
  --percent-fill: calc(attr(data-rating type(<number>)) * 20%);
  /*  hard breakpoint gradient background  */
  background: linear-gradient(to right, gold var(--percent-fill)

  &::after {
    content: attr(data-rating);
  }
}
```

ToggleEvent.source

Find out which element was responsible for toggling the target.

Try the demo MDN

ToggleEvent source 👍 0

Limited availability

When a popover, `<dialog>`, or `<details>` element gets toggled, it can be interesting to know which element was responsible for toggling it. For example, knowing if the user pressed the “Accept Cookies” or “Reject Cookies” button to dismiss a *cookie banner* is a very important detail.

The `source` attribute of the `ToggleEvent` lets you know exactly that, as it contains the element which triggered the event to be fired, if applicable. Based on that source you can take different actions.

```
<div id="cookiebanner" popover="auto">
 <p>Would you like a cookie?</p>
 <button id="yes" commandfor="cookiebanner" command="hide-popove
 <button id="no" commandfor="cookiebanner" command="hide-popover
</div>

<script>
 const $btnYes = document.getElementById('yes');
 const $btnNo = document.getElementById('no');
 const $cookiebanner = document.getElementById('cookiebanner');
```

```
    } else if (event.source == $btnNo) {
      // Don't give the user a cookie
    }
  });
</script>
```

Cookie banner demo that uses `ToggleEvent.source`. The demo also uses Invoker Commands

text-box features

text-box 👍 15

Limited availability

A font's content box is defined by internal metrics—specifically the ascent and descent that reserve space for accents and hanging characters.

Illustration showing the ascender and descender line of a typeface. (Source: Material Design)

Because the visual boundaries of Latin text are the cap height and the alphabetic baseline, rather than the ascent and descent, text will appear optically off-center even when it is mathematically centered within a container.

Illustrations showing the cap height and baseline of a typeface. (Source: Material Design)

The `text-box` properties make finer control of vertical alignment of text possible, letting you flawlessly center text vertically. The `text-box-trim` property specifies the sides to trim, above or below (or both), and the `text-box-edge` property specifies the metrics to use for `text-box-trim` effects.

When trimming both edges and setting the over edge metric to `cap` and the under edge metric to `alphabetic`, text will be visually centered.

```
h1, button {
  text-box: trim-both cap alphabetic;
}
```

Interactive CSS text-box demo

shape() function

Try the demo Chrome for Developers MDN

shape() 👍 4

 Limited availability 

The new `shape()` function lets you clip an element to a complex, non-polygonal, responsive shape in CSS. This is a great option for clipping masks using `clip-path: path()`, and works seamlessly with CSS custom properties to define coordinates and control points, making it more maintainable than SVG shapes. This also means you can animate your custom properties within `shape()` to create dynamic and interactive clipping.

Here's how to create a flag shape with curved top and bottom edges using `shape()`:

```
.flag {
  clip-path: shape(from 0% 20px,
    curve to 100% 20px with 25% 0% / 75% 40px,
    vline to calc(100% - 20px),
    curve to 0% calc(100% - 20px)
      with 75% 100% / 25% calc(100% - 40px),
    close
  );
}
```

In this example, the horizontal coordinates use percentages to scale with the element's width, while the vertical coordinates for the curve's height use fixed pixel values, creating a responsive effect where the flag's wave remains constant regardless of the element's size.

if() statements

Conditionals in your CSS for dynamic styling.

Try the demo Chrome for Developers MDN

if() 👍 14

Limited availability

The `if()` function in CSS lets you set different values for a property based on a conditional test. Think of it like a ternary operator in JavaScript, but for your stylesheets. It provides a cleaner and more concise way to handle dynamic styling compared to writing multiple, verbose `@media` or `@supports` blocks for single property changes.

The syntax is straightforward. The `if()` function takes a series of condition-value pairs, separated by semicolons. The first condition that evaluates to `true` will have its corresponding value applied. You can also provide an `else fallback` value.

```
if(condition-1: value-1; condition-2: value-2; else: fallback-va
```

Currently, `if()` can be used with three types of queries:

- `media()`: For media queries.
- `supports()`: For feature queries.
- `style()`: For style queries.

For example, you can create a responsive layout that changes from a column to a row based on viewport orientation:

```
.responsive-layout {
  display: flex;
  flex-direction: if(media(orientation: landscape): row; else: co
}
```

This approach is more concise than a traditional media query, which requires you to define the styles in two separate places. With if(), you can keep the logic for a single property in one place, making your CSS easier to read and maintain. Change the orientation of the layout in this CodePen by opening the CSS or HTML side pane:

Custom Functions

Reusable functions for cleaner, maintainable styles.

Try the demo MDN

@function 👍 12

Limited availability 

CSS custom functions are a fantastic new addition to the CSS language, and make it much easier to write composable, reusable, and clear functional styling logic. A custom function is made up of the `@function` statement, a function name prefixed with a double dash (`--`), a series of arguments, and a `result` block. The arguments can also have default, or fallback, values.

An example of a simple CSS function is the "negate" function which returns the inverse value of a number:

```
/* Negate function returns the negative of a value */
@function --negate(--value) {
  result: calc(-1 * var(--value));
}

/* Usage */
html {
  --gap: 1em;
  padding: --negate(var(--gap));
}
```

conditionally rounded border radius. The following function removes an element's `border-radius` when it gets within a specified distance of the viewport edge (defaulting to 4px), otherwise applying the desired radius. You can provide one argument for the radius, or a second to override the edge distance:

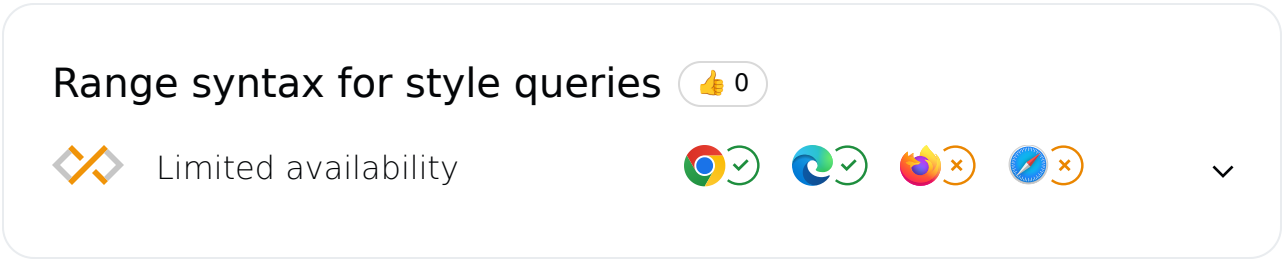
0:00

```
/* Conditionally apply a radius until you are (default: 4px, or s
@function --conditional-radius(--radius, --edge-dist: 4px) {
  result: clamp(0px, ((100vw - var(--edge-dist)) - 100%) * 1e5, v
}

/* usage */
.box {
  /*  1rem border radius, default (4px) distance  */
  border-radius: --conditional-radius(1rem);
}

.box-2 {
  /*  1rem border radius, right at the edge (0px distance)  */
  border-radius: --conditional-radius(1rem, 0px);
}
```

Expanded range syntax

Range syntax in style queries and if() statements.

Try the demo MDN



One nice update that landed this year is the ability to use range syntax in style queries and `if()` statements. Media queries and container queries already supported this capability, but before Chrome 142, style queries required an exact

Now, you can type your values and use them with comparison operators like `<`, `>`, `<=`, and `>=`. This enables many new architectural capabilities directly in your CSS.

The following demo uses stylized cards to visualize the daily weather. The HTML markup includes data, such as the chance of rain, which is indicated by the value of data-rain-percent.

```
<li class="card-container" style="--cloudy: true;" data-rain-per
  <div class="weather-card">
    <h2>Today's Weather</h2>
    <p>Chance of rain: 60%</p>
  </div>
</li>
```

In CSS, convert `data-rain-percent` into a custom property, give it a type using `attr()`, and then use it within a range style query:

```
.card-container {
  container-name: weather;
  --rain-percent: attr(data-rain-percent type(<percentage>));
}

@container style(--rain-percent > 45%) {
  .weather-card {
    background: linear-gradient(140deg, blue, lightblue);
  }
}
```

Now, if the chance of rain is greater than 45%, the card will get a blue background.

Range queries can also be used in `if()` statements now as well, meaning more concise phrasing for styles. For example, you can write the above code even more concisely using inline `if()`:

```
.weather-card {
  background: if(style(--rain-percent > 45%): blue; else: gray;
  );
}
```

RESOURCES:

- Range syntax for style queries blog post

Stretch sizing keyword

Make an element fill its containing block, regardless of the `box-sizing`.

Try the demo MDN

stretch 👍 4

Limited availability

The `stretch` keyword is a keyword for use with CSS sizing properties (such as `width` and `height`) that lets elements grow to exactly fill their containing block's available space.

It's similar to `100%`, except the resulting size is applied to the margin box of the element instead of the box determined by `box-sizing`.

```
.element {
  height: stretch;
}
```

Using this keyword lets the element keep its margins while still being as large as possible.

Demo to compare behavior of `height` being set to `auto`, `100vh`, `100%`, or `stretch`.

corner-shape

Corner shapes beyond rounded edges.

Try the demo MDN

corner-shape 👍 17

This year, CSS gives us more control over the shape of our elements with the new `corner-shape` property. This experimental feature lets you customize the shape of corners beyond the standard rounded corners available with `border-radius`.

You can now create a variety of corner styles, including:

- `round`
- `bevel`
- `notch`
- `scoop`
- `squircle`

This property opens up a world of creative possibilities. From flower-like shapes to hexagonal grids , and even enabling a simple squircle; this CSS feature is small but mighty. You can even animate between different corner shapes for dynamic and engaging user interfaces, making this a great option for hover effects and interest states.

For even more control, you can use the `superellipse()` function to create any continuous curve, allowing for fine-tuned and unique corner designs.

We can't wait to see what you build!

Made with ❤️ by The Chrome DevRel Team