Well-controlled scrolling with CSS Scroll Snap

- Background
  - The case for scroll snapping
- CSS Scroll Snap
  - Example: A horizontal gallery
  - Example: a journeyed product page
    - Scroll padding and margin
- Interaction with other scrolling APIs
  - DOM Scrolling API
  - Smooth scrolling
  - Overscroll behavior
- Caveats and best practices
  - Feature detection in CSS
  - Feature detection in JavaScript
- Future work

The CSS Scroll Snap feature allows web developers to create well-controlled scroll experiences by declaring scroll snapping positions. Paginated articles and image carousels are two commonly used examples of this. CSS Scroll Snap provides an easy-to-use and consistent API for building these popular UX patterns.

CSS Scroll Snap is supported in all major browsers. A previous version of the specification was implemented in some browsers, and may appear in tutorials and articles. If material includes the deprecated `scroll-snap-points-x` and `scroll-snap-points-y` properties, it should be considered outdated.

Background

The case for scroll snapping

Scrolling is a popular and natural way to interact with content on the web. It is the platform's native means of providing access to more information than is visible on the screen at once, becoming especially vital on mobile platforms with limited screen real estate. So it is no surprise that web authors increasingly prefer to organize content into scrollable flat lists as opposed to deep hierarchies.

Scrolling's main drawback is its lack of precision. Rarely does a scroll end up aligned to a paragraph or sentence. This is even more pronounced for paginated or itemized content with meaningful boundaries when the scroll finishes at the middle of the page or image, leaving it partially visible. These use cases benefit from a well-controlled scrolling experience.

Web developers have long relied on JavaScript-based solutions for controlling the scroll to help address this shortcoming. However, JavaScript-based solutions fail to provide a full fidelity solution due to lack of scroll customization primitives or access to composited scrolling. CSS Scroll Snap ensures a fast, high fidelity, and easy-to-use solution that works consistently across browsers.

CSS Scroll Snap allows web authors to mark each scroll container with boundaries for scroll operations at which to finish. Browsers then choose the most appropriate end position depending on the particulars of the scroll operation, scroll container's layout and visibility, and details of the snap positions, then smoothly animate to it. Going back to our earlier example, as the user finishes scrolling the carousel, its visible image snaps into place. No scroll adjustments needed by JavaScript.

> *Example of using css scroll snap with an image carousel.*
>
> Example of using css scroll snap with an image carousel. Here scroll snapping ensures at the end of scrolling an image horizontal center is aligned with the horizontal center of the scroll container.

CSS Scroll Snap

Scroll snapping is the act of adjusting the scroll offset of a scroll container to be at a preferred **snap position** once the scroll operation finishes.

A scroll container may be opted into scroll snapping by using the `scroll-snap-type` property. This tells the browser that it should consider snapping this scroll container to the snap positions produced by its descendants. `scroll-snap-type` determines the axis on which scrolling occurs: `x`, `y`, or `both`, and the snapping strictness: `mandatory`, `proximity`. More on these later.

A snap position can be produced by declaring a desired alignment on an element. This position is the scroll offset at which the nearest ancestor scroll container and the element are aligned as

specified for the given axis. The following alignments are possible on each axis: `start`, `end`, `center`.

A `start` alignment means that the scroll container snapport start edge should be flushed with the element snap area start edge. Similarly, the `end` and `center` alignments mean that the scroll container snapport end edge or center should be flushed with the element snap area end edge or center.

The snapport is the area of the scroll container to which the snap areas are aligned. By default, it is the same as the visual viewport of the scroll container, but it can be adjusted using the `scroll-padding` property.

Example of a various alignments on horizontal scrolling axis.

The following examples illustrate how to use these concepts.

Example: A horizontal gallery

A common use case for scroll snapping is an image carousel. For example, to create a horizontal image carousel that snaps to each image as you scroll, we can specify the scroll container to have a mandatory `scroll-snap-type` on the horizontal axis. set each image to `scroll-snap-align: center` to ensure that the snapping centers the image within the carousel.

```
#gallery {
  scroll-snap-type: x mandatory;
  overflow-x: scroll;
  display: flex;
}

#gallery img {
   scroll-snap-align: center;
}
```

```
<div id="gallery">
  <img src="cat.jpg">
  <img src="dog.jpg">
  <img src="another_cute_animal.jpg">
</div>
```

Because snap positions are associated with an element, the snapping algorithm can be smart about when and how it snaps given the element and the scroll container size. For example, consider the case where one image is larger than the carousel. A naïve snapping algorithm may

prevent the user from panning around to see the full image. But the specification requires implementations to detect this case and allow the user to freely scroll around within that image only snapping at its edges.

View demo | Source

Example: a journeyed product page

Another common case that can benefit from scroll snapping are pages with multiple logical sections to vertically scroll through, for example, a typical product page. `scroll-snap-type: y proximity;` is a more natural fit for cases like this. It does not interfere when a user scrolls to the middle of a particular section but also snaps and brings attention to a new section when they scroll close enough.

Here is how this can be achieved:

```
article {
  scroll-snap-type: y proximity;
  /* Reserve space for header plus some extra space for sneak peeking. */
  scroll-padding-top: 15vh;
  overflow-y: scroll;
}
section {
  /* Snap align start. */
  scroll-snap-align: start;
}
header {
  position: fixed;
  height: 10vh;
}
```

```
<article>
  <header> Header </header>
  <section> Section One </section>
  <section> Section Two </section>
  <section> Section Three </section>
</article>
```

Scroll padding and margin

The product page has a fixed position top header. The design also asked for some of the top section to remain visible when the scroll container is snapped in order to provide a design cue to users about the content above.

The `scroll-padding` property is a new css property that can be used to adjust the effective viewable region of the scroll container, or snapport, which is used when calculating scroll snap alignments. The property defines an inset against the scroll container's padding box. In our example, `15vh` additional inset was added to the top, which instructs the browser to consider a lower position, `15vh` below the top edge of the scroll container, as its vertical start edge for scroll snapping. When snapping, the start edge of the snap target element will become flushed with this new position, thus leaving space above.

The `scroll-margin` property defines the outset amount used to adjust the snap target effective box similar to how `scroll-padding` functions on the snap scroll container.

You may have noticed that these two properties do not have the word "`snap`" in them. This is intentional as they actually modify the box for all relevant scroll operations and are not just scroll snapping. For example, Chrome takes them into account when calculating page size for paging scroll operations such as PageDown and PageUp, and also when calculating scroll amount for the `Element.scrollIntoView()` operation.

> View demo | Source

Interaction with other scrolling APIs

DOM Scrolling API

Scroll snapping happens **after** all scroll operations including those initiated by script. When you are using APIs like `Element.scrollTo`, the browser will calculate the intended scroll position of the operation, then apply appropriate snapping logic to find the final snapped location. Thus, there is no need for user script to do any manual calculations for snapping.

Smooth scrolling

Smooth scrolling controls the behavior of a programmatic scroll operation while scroll snap determines its destination. Since they control orthogonal aspects of scrolling, they can be used together and complement each other.

Overscroll behavior

Overscroll behavior API controls how scroll is chained across multiple elements and it is not affected by scroll snap.

Caveats and best practices

Avoid using mandatory snapping when target elements are widely spaced apart. This can cause content in between the snap positions to become inaccessible.

In many cases scroll-snapping can be added as an enhancement without needing to feature detect. If required, use `@supports` or `CSS.supports` to detect support for CSS Scroll Snap. Avoid using `scroll-snap-type` which is also present in the deprecated specification.

Feature detection in CSS

```
@supports (scroll-snap-align: start) {
  article {
    scroll-snap-type: y proximity;
    scroll-padding-top: 15vh;
    overflow-y: scroll;
  }
}
```

Feature detection in JavaScript

```
if (CSS.supports('scroll-snap-align: start')) {
  // use css scroll snap
} else {
  // use fallback
}
```

Do not assume that programmatically scrolling APIs such as `Element.scrollTo` always finish at the requested scroll offset. Scroll snapping may adjust the scroll offset after programmatic scrolling is complete. Note that this was not a good assumption even before scroll snap since scrolling may have been interrupted for other reasons, but it is especially the case with scroll snapping.

There is an upcoming proposal to change various scrolling APIs to return a promise. This promise is resolved when the user agent either completes or aborts that scrolling operation. Once this is standardized and implemented, it provides an ergonomic and efficient way for following up a user script initiated scroll with other actions.

Future work

Scroll experience was the focus of a recent survey by the Chrome team. The survey results identified several areas that need additional work to shrink the gap between plugin libraries and CSS. Upcoming work will focus on `scroll-snap`, including:

1. API availability and compatibility across browsers.
2. Work on new CSS APIs like `scroll-start`.
3. Work on new JS events like `snapChanged()`.

□□□ #3
□ □□□ □□□ 2026-10-31 21:47:31 UTC
□ cocobook-sync □□□ 2026-11-04 17:36:58 UTC