Intersection Observer API

English ▼

Intersection observer concepts and usage
Interfaces
A simple example
Specifications
Browser compatibility
See also

Related Topics
Intersection Observer API
▼ Interfaces
`IntersectionObserver`
`IntersectionObserverEntry`

The Intersection Observer API provides a way to asynchronously observe changes in the intersection of a target element with an ancestor element or with a top-level document's viewport.

Historically, detecting visibility of an element, or the relative visibility of two elements in relation to each other, has been a difficult task for which solutions have been unreliable and prone to causing the browser and the sites the user is accessing to become sluggish. As the web has matured, the need for this kind of information has grown. Intersection information is needed for many reasons, such as:

- Lazy-loading of images or other content as a page is scrolled.
- Implementing "infinite scrolling" web sites, where more and more content is loaded and rendered as you scroll, so that the user doesn't have to flip through pages.
- Reporting of visibility of advertisements in order to calculate ad revenues.
- Deciding whether or not to perform tasks or animation processes based on whether or not the user will see the result.

Implementing intersection detection in the past involved event handlers and loops calling methods like `Element.getBoundingClientRect()` to build up the needed information for every element affected. Since all this code runs on the main thread, even one of these can cause performance problems. When a site is loaded with these tests, things can get downright ugly.

Consider a web page that uses infinite scrolling. It uses a vendor-provided library to manage the advertisements placed periodically throughout the page, has animated graphics here and there, and uses a custom library that draws notification boxes and the like. Each of these has its own intersection detection routines, all running on the main thread. The author of the web site may not even realize this is happening, since they may know very little about the inner workings of the two libraries they are using. As the user scrolls the page, these intersection detection routines are firing constantly during the scroll handling code, resulting in an experience that leaves the user frustrated with the browser, the web site, and their computer.

The Intersection Observer API lets code register a callback function that is executed whenever an element they wish to monitor enters or exits another element (or the viewport), or when the amount by which the two intersect changes by a requested amount. This way, sites no longer need to do anything on the main thread to watch for this kind of element intersection, and the browser is free to optimize the management of intersections as it sees fit.

One thing the Intersection Observer API can't tell you: the exact number of pixels that overlap or specifically which ones they are; however, it covers the much more common use case of "If they intersect by somewhere around *N*%, I need to do something."

Intersection observer concepts and usage

The Intersection Observer API allows you to configure a callback that is called when either of these circumstances occur:

- A **target** element intersects either the device's viewport or a specified element. That specified element is called the **root element** or **root** for the purposes of the Intersection Observer API.
- The first time the observer is initially asked to watch a target element.

Typically, you'll want to watch for intersection changes with regard to the element's closest scrollable ancestor, or, if the element isn't a descendant of a scrollable element, the viewport. To watch for intersection relative to the root element, specify `null`.

Whether you're using the viewport or some other element as the root, the API works the same way, executing a callback function you provide whenever the visibility of the target element changes so that it crosses desired amounts of intersection with the root.

The degree of intersection between the target element and its root is the **intersection ratio**. This is a representation of the percentage of the target element which is visible as a value between 0.0 and 1.0.

Creating an intersection observer

Create the intersection observer by calling its constructor and passing it a callback function to be run whenever a threshold is crossed in one direction or the other:

```
1  let options = {
2    root: document.querySelector('#scrollArea'),
3    rootMargin: '0px',
4    threshold: 1.0
5  }
6
7  let observer = new IntersectionObserver(callback, options);
```

A threshold of 1.0 means that when 100% of the target is visible within the element specified by the `root` option, the callback is invoked.

Intersection observer options

The `options` object passed into the `IntersectionObserver()` constructor let you control

the circumstances under which the observer's callback is invoked. It has the following fields:

`root`
: The element that is used as the viewport for checking visibility of the target. Must be the ancestor of the target. Defaults to the browser viewport if not specified or if `null`.

`rootMargin`
: Margin around the root. Can have values similar to the CSS `margin` property, e.g. "`10px 20px 30px 40px`" (top, right, bottom, left). The values can be percentages. This set of values serves to grow or shrink each side of the root element's bounding box before computing intersections. Defaults to all zeros.

`threshold`
: Either a single number or an array of numbers which indicate at what percentage of the target's visibility the observer's callback should be executed. If you only want to detect when visibility passes the 50% mark, you can use a value of 0.5. If you want the callback to run every time visibility passes another 25%, you would specify the array [0, 0.25, 0.5, 0.75, 1]. The default is 0 (meaning as soon as even one pixel is visible, the callback will be run). A value of 1.0 means that the threshold isn't considered passed until every pixel is visible.

Targeting an element to be observed

Once you have created the observer, you need to give it a target element to watch:

```
let target = document.querySelector('#listItem');
observer.observe(target);

// the callback we setup for the observer will be executed now for the fir
// it waits until we assign a target to our observer (even if the target i
```

Whenever the target meets a threshold specified for the `IntersectionObserver`, the callback is invoked. The callback receives a list of `IntersectionObserverEntry` objects and the observer:

```
let callback = (entries, observer) => {
  entries.forEach(entry => {
    // Each entry describes an intersection change for one observed
    // target element:
    //   entry.boundingClientRect
    //   entry.intersectionRatio
    //   entry.intersectionRect
    //   entry.isIntersecting
    //   entry.rootBounds
    //   entry.target
    //   entry.time
  });
};
```

The list of entries received by the callback includes one entry for each target which reporting a change in its intersection status. Check the value of the `isIntersecting` property to see if the entry represents an element that currently intersects with the root.

Be aware that your callback is executed on the main thread. It should operate as quickly as possible; if anything time-consuming needs to be done, use `Window.requestIdleCallback()`.

Also, note that if you specified the `root` option, the target must be a descendant of the root element.

How intersection is calculated

All areas considered by the Intersection Observer API are rectangles; elements which are irregularly shaped are considered as occupying the smallest rectangle which encloses all of the element's parts. Similarly, if the visible portion of an element is not rectangular, the element's intersection rectangle is construed to be the smallest rectangle that contains all the visible portions of the element.

It's useful to understand a bit about how the various properties provided by `IntersectionObserverEntry` describe an intersection.

The intersection root and root margin

Before we can track the intersection of an element with a container, we need to know what that container is. That container is the **intersection root**, or **root element**. This can be either a specific element in the document which is an ancestor of the element to be observed, or `null` to use the document's viewport as the container.

The **root intersection rectangle** is the rectangle used to check against the target or targets. This rectangle is determined like this:

- If the intersection root is the implicit root (that is, the top-level `Document`), the root intersection rectangle is the viewport's rectangle.
- If the intersection root has an overflow clip, the root intersection rectangle is the root element's content area.
- Otherwise, the root intersection rectangle is the intersection root's bounding client rectangle (as returned by calling `getBoundingClientRect()` on it).

The root intersection rectangle can be adjusted further by setting the **root margin**, `rootMargin`, when creating the `IntersectionObserver`. The values in `rootMargin` define offsets added to each side of the intersection root's bounding box to create the final intersection root bounds (which are disclosed in `IntersectionObserverEntry.rootBounds` when the callback is executed).

Thresholds

Rather than reporting every infinitesimal change in how much a target element is visible, the Intersection Observer API uses **thresholds**. When you create an observer, you can provide

one or more numeric values representing percentages of the target element which are visible. Then, the API only reports changes to visibility which cross these thresholds.

For example, if you want to be informed every time a target's visibility passes backward or forward through each 25% mark, you would specify the array [0, 0.25, 0.5, 0.75, 1] as the list of thresholds when creating the observer.

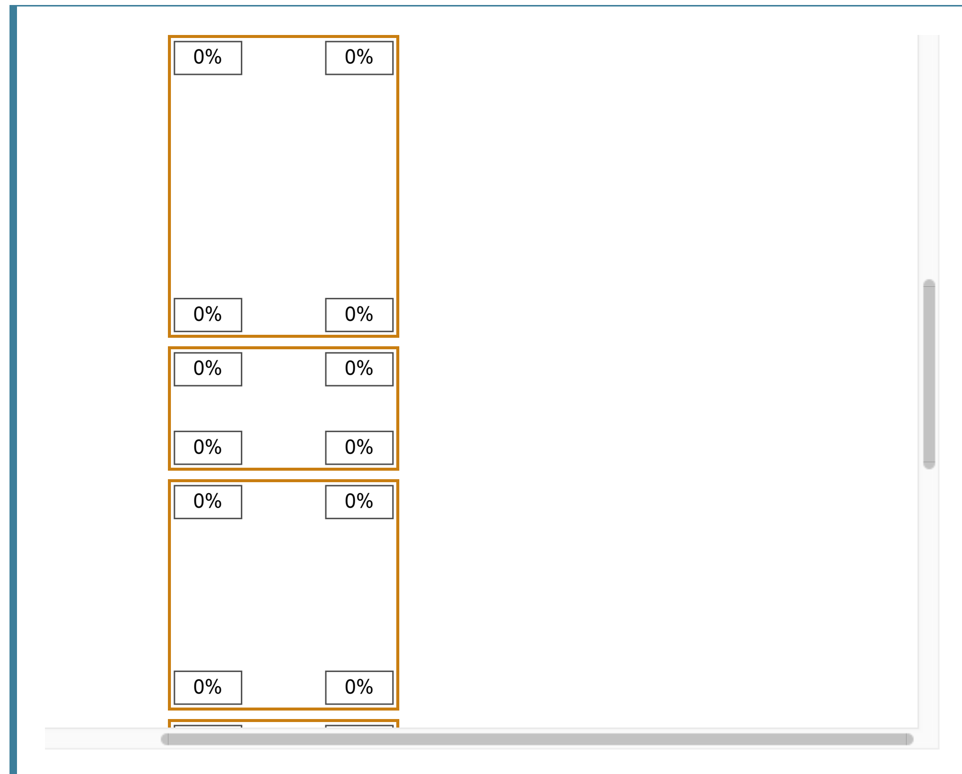
When the callback is invoked, it receives a list of `IntersectionObserverEntry` objects, one for each observed target which has had the degree to which it intersects the root change such that the amount exposed crosses over one of the thresholds, in either direction.

You can see if the target *currently* intersects the root by looking at the entry's `isIntersecting` property; if its value is `true`, the target is at least partially intersecting the root element or document. This lets you determine whether the entry represents a transition from the elements intersecting to no longer intersecting or a transition from not intersecting to intersecting.

Note that it's possible to have a non-zero intersection rectangle, which can happen if the intersection is exactly along the boundary between the two or the area of `boundingClientRect` is zero. This state of the target and root sharing a boundary line is not considered enough to be considered transitioning into an intersecting state.

To get a feeling for how thresholds work, try scrolling the box below around. Each colored box within it displays the percentage of itself that's visible in all four of its corners, so you can see these ratios change over time as you scroll the container. Each box has a different set of thresholds:

- The first box has a threshold for each percentage point of visibility; that is, the `IntersectionObserver.thresholds` array is `[0.00, 0.01, 0.02, ..., 0.99, 1.00]`.
- The second box has a single threshold, at the 50% mark.
- The third box has thresholds every 10% of visibility (0%, 10%, 20%, etc.).
- The last box has thresholds each 25%.



Clipping and the intersection rectangle

The browser computes the final intersection rectangle as follows; this is all done for you, but it can be helpful to understand these steps in order to better grasp exactly when intersections will occur.

1. The target element's bounding rectangle (that is, the smallest rectangle that fully encloses the bounding boxes of every component that makes up the element) is obtained by calling `getBoundingClientRect()` on the target. This is the largest the intersection rectangle may be. The remaining steps will remove any portions that don't intersect.
2. Starting at the target's immediate parent block and moving outward, each containing block's clipping (if any) is applied to the intersection rectangle. A block's clipping is determined based on the intersection of the two blocks and the clipping mode (if any) specified by the `overflow` property. Setting `overflow` to anything but `visible` causes clipping to occur.
3. If one of the containing elements is the root of a nested browsing context (such as the document contained in an `<iframe>`, the intersection rectangle is clipped to the containing context's viewport, and recursion upward through the containers continues with the container's containing block. So if the top level of an `<iframe>` is reached, the intersection rectangle is clipped to the frame's viewport, then the frame's parent element is the next block recursed through toward the intersection root.
4. When recursion upward reaches the intersection root, the resulting rectangle is mapped to the intersection root's coordinate space.
5. The resulting rectangle is then updated by intersecting it with the root intersection rectangle.
6. This rectangle is, finally, mapped to the coordinate space of the target's `document`.

Intersection change callbacks

When the amount of a target element which is visible within the root element crosses one of the visibility thresholds, the `IntersectionObserver` object's callback is executed. The callback receives as input an array of all of `IntersectionObserverEntry` objects, one for each threshold which was crossed, and a reference to the `IntersectionObserver` object itself.

Each entry in the list of thresholds is an `IntersectionObserverEntry` object describing one threshold that was crossed; that is, each entry describes how much of a given element is intersecting with the root element, whether or not the element is considered to be intersecting or not, and the direction in which the transition occurred.

The code snippet below shows a callback which keeps a counter of how many times elements transition from not intersecting the root to intersecting by at least 75%. For a threshold value of 0.0 (default) the callback is called approximately upon transition of the boolean value of `isIntersecting`. The snippet thus first checks that the transition is a positive one, then determines whether `intersectionRatio` is above 75%, in which case it increments the counter.

```
intersectionCallback(entries) => {
  entries.forEach(entry => {
    if (entry.isIntersecting) {
      let elem = entry.target;

      if (entry.intersectionRatio >= 0.75) {
        intersectionCounter++;
      }
    }
  });
}
```

Interfaces

`IntersectionObserver`

The primary interface for the Intersection Observer API. Provides methods for creating and managing an observer which can watch any number of target elements for the same intersection configuration. Each observer can asynchronously observe changes in the intersection between one or more target elements and a shared ancestor element or with their top-level `Document`'s viewport. The ancestor or viewport is referred to as the **root**.

`IntersectionObserverEntry`

Describes the intersection between the target element and its root container at a specific moment of transition. Objects of this type can only be obtained in two ways: as an input to your `IntersectionObserver` callback, or by calling `IntersectionObserver.takeRecords()`.

A simple example

This simple example causes a target element to change its color and transparency as it becomes more or less visible. At Timing element visibility with the Intersection Observer API, you can find a more extensive example showing how to time how long a set of elements (such as ads) are visible to the user and to react to that information by recording statistics or by updating elements..

HTML

The HTML for this example is very short, with a primary element which is the box that we'll be targeting (with the creative ID `"box"`) and some contents within the box.

```
<div id="box">
  <div class="vertical">
    Welcome to <strong>The Box!</strong>
  </div>
</div>
```

CSS

The CSS isn't terribly important for the purposes of this example; it lays out the element and establishes that the `background-color` and `border` attributes can participate in CSS transitions, which we'll use to affect the changes to the element as it becomes more or less obscured.

```
#box {
  background-color: rgba(40, 40, 190, 255);
  border: 4px solid rgb(20, 20, 120);
  transition: background-color 1s, border 1s;
  width: 350px;
  height: 350px;
  display: flex;
  align-items: center;
  justify-content: center;
  padding: 20px;
}

.vertical {
  color: white;
  font: 32px "Arial";
}

.extra {
  width: 350px;
  height: 350px;
  margin-top: 10px;
```

```
22    border: 4px solid rgb(20, 20, 120);
23    text-align: center;
24    padding: 20px;
25  }
```

JavaScript

Finally, let's take a look at the JavaScript code that uses the Intersection Observer API to make things happen.

Setting up

First, we need to prepare some variables and install the observer.

```
1  const numSteps = 20.0;
2
3  let boxElement;
4  let prevRatio = 0.0;
5  let increasingColor = "rgba(40, 40, 190, ratio)";
6  let decreasingColor = "rgba(190, 40, 40, ratio)";
7
8  // Set things up
9  window.addEventListener("load", (event) => {
10   boxElement = document.querySelector("#box");
11
12   createObserver();
13 }, false);
```

The constants and variables we set up here are:

`numSteps`
A constant which indicates how many thresholds we want to have between a visibility ratio of 0.0 and 1.0.

`prevRatio`
This variable will be used to record what the visibility ratio was the last time a threshold was crossed; this will let us figure out whether the target element is becoming more or less visible.

`increasingColor`
A string defining a color we'll apply to the target element when the visibility ratio is increasing. The word "ratio" in this string will be replaced with the target's current visibility ratio, so that the element not only changes color but also becomes increasingly opaque as it becomes less obscured.

`decreasingColor`
Similarly, this is a string defining a color we'll apply when the visibility ratio is decreasing.

We call `Window.addEventListener()` to start listening for the `load` event; once the page has finished loading, we get a reference to the element with the ID `"box"` using `querySelector()`, then call the `createObserver()` method we'll create in a moment to handle building and installing the intersection observer.

Creating the intersection observer

The `createObserver()` method is called once page load is complete to handle actually creating the new `IntersectionObserver` and starting the process of observing the target element.

```
function createObserver() {
  let observer;

  let options = {
    root: null,
    rootMargin: "0px",
    threshold: buildThresholdList()
  };

  observer = new IntersectionObserver(handleIntersect, options);
  observer.observe(boxElement);
}
```

This begins by setting up an `options` object containing the settings for the observer. We want to watch for changes in visibility of the target element relative to the document's viewport, so `root` is `null`. We need no margin, so the margin offset, `rootMargin`, is specified as "0px". This causes the observer to watch for changes in the intersection between the target element's bounds and those of the viewport, without any added (or subtracted) space.

The list of visibility ratio thresholds, `threshold`, is constructed by the function `buildThresholdList()`. The threshold list is built programmatically in this example since there are a number of them and the number is intended to be adjustable.

Once `options` is ready, we create the new observer, calling the `IntersectionObserver()` constructor, specifying a function to be called when intersection crosses one of our thresholds, `handleIntersect()`, and our set of options. We then call `observe()` on the returned observer, passing into it the desired target element.

We could opt to monitor multiple elements for visibility intersection changes with respect to the viewport by calling `observer.observe()` for each of those elements, if we wanted to do so.

Building the array of threshold ratios

The `buildThresholdList()` function, which builds the list of thresholds, looks like this:

```
function buildThresholdList() {
  let thresholds = [];
  let numSteps = 20;
```

```
4
5     for (let i=1.0; i<=numSteps; i++) {
6       let ratio = i/numSteps;
7       thresholds.push(ratio);
8     }
9
10    thresholds.push(0);
11    return thresholds;
12  }
```

This builds the array of thresholds—each of which is a ratio between 0.0 and 1.0, by pushing the value `i/numSteps` onto the `thresholds` array for each integer `i` between 1 and `numSteps`. It also pushes 0 to include that value. The result, given the default value of `numSteps` (20), is the following list of thresholds:

#	Ratio	#	Ratio
1	0.05	**11**	0.55
2	0.1	**12**	0.6
3	0.15	**13**	0.65
4	0.2	**14**	0.7
5	0.25	**15**	0.75
6	0.3	**16**	0.8
7	0.35	**17**	0.85
8	0.4	**18**	0.9
9	0.45	**19**	0.95
10	0.5	**20**	1.0

We could, of course, hard-code the array of thresholds into our code, and often that's what you'll end up doing. But this example leaves room for adding configuration controls to adjust the granularity, for example.

Handling intersection changes

When the browser detects that the target element (in our case, the one with the ID `"box"`) has been unveiled or obscured such that its visibility ratio crosses one of the thresholds in our list, it calls our handler function, `handleIntersect()`:

```
1  function handleIntersect(entries, observer) {
2    entries.forEach((entry) => {
3      if (entry.intersectionRatio > prevRatio) {
4        entry.target.style.backgroundColor = increasingColor.replace("ratio"
5      } else {
6        entry.target.style.backgroundColor = decreasingColor.replace("ratio"
7      }
```

```
8
9       prevRatio = entry.intersectionRatio;
10    });
11  }
```

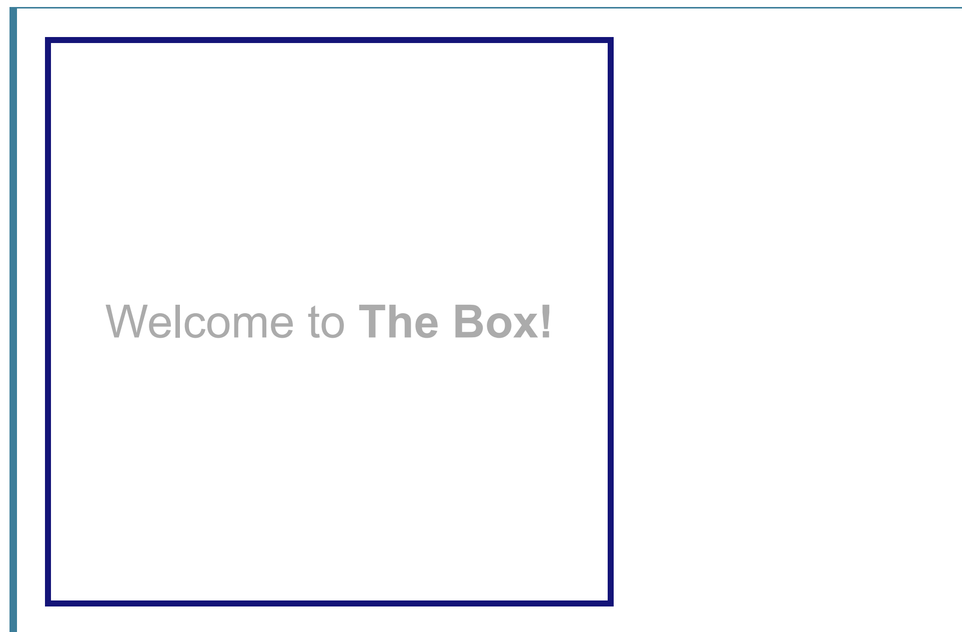
For each `IntersectionObserverEntry` in the list `entries`, we look to see if the entry's `intersectionRatio` is going up; if it is, we set the target's `background-color` to the string in `increasingColor` (remember, it's `"rgba(40, 40, 190, ratio)"`), replaces the word "ratio" with the entry's `intersectionRatio`. The result: not only does the color get changed, but the transparency of the target element changes, too; as the intersection ratio goes down, the background color's alpha value goes down with it, resulting in an element that's more transparent.

Similarly, if the `intersectionRatio` is going down, we use the string `decreasingColor` and replace the word "ratio" in that with the `intersectionRatio` before setting the target element's `background-color`.

Finally, in order to track whether the intersection ratio is going up or down, we remember the current ratio in the variable `prevRatio`.

Result

Below is the resulting content. Scroll this page up and down and notice how the appearance of the box changes as you do so.



There's an even more extensive example at Timing element visibility with the Intersection Observer API.

Specifications

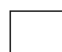
Specification	Status	Comment
Intersection Observer	WD Working Draft	

Browser compatibility

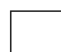
Update compatibility data on GitHub

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet
`IntersectionObserver`	51	15	55	No	38	12.1	51	51	?	41	12.2	5.0
`IntersectionObserver()` constructor	51	15	55	No	38	12.1	51	51	?	?	12.2	5.0
`disconnect`	51	15	55	No	Yes	?	51	51	?	?	?	5.0
`observe`	51	15	55	No	Yes	12.1	51	51	?	?	12.2	5.0
`root`	51	15	55	No	Yes	12.1	51	51	?	?	12.2	5.0
`rootMargin`	51	15	55	No	Yes	12.1	51	51	?	?	12.2	5.0
`takeRecords`	51	15	55	No	Yes	?	51	51	?	?	?	5.0
`thresholds`	51	15	55	No	Yes	12.1	51	51	?	?	12.2	5.0
`unobserve`	51	15	55	No	Yes	12.1	51	51	?	?	12.2	5.0

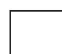
What are we missing?



Full support



No support



Compatibility unknown

Experimental. Expect behavior to change in the future.

See implementation notes.

User must explicitly enable this feature.

See also

- Intersection Observer polyfill
- Timing element visibility with the Intersection Observer API
- `IntersectionObserver` and `IntersectionObserverEntry`

Last modified: Aug 29, 2020, by MDN contributors

Learn the best of web development

Get the latest and greatest from MDN delivered straight to your inbox.

you@example.com

Sign up now