Video

By Sam Dutton
Sam is a Developer Advocate

Users like videos; videos can be fun and informative. On mobile devices, videos can be an easier way to consume information. But videos take bandwidth and they don't always work the same across every platform. Users don't like waiting for videos to load; they don't like it when they press play and nothing happens. Read on to find the simplest way to add video to your site and ensure users get the best possible experience on any device.

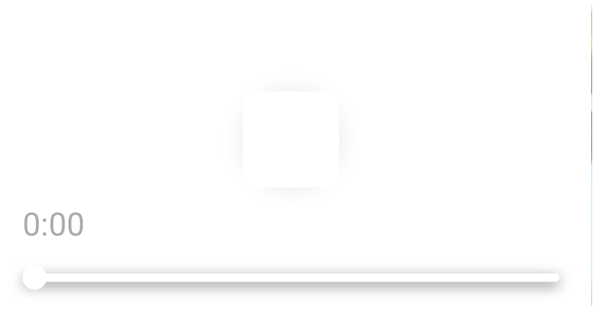
Add a video

TL;DR

- Use the `video` element to load, decode, and play video on your site.
- Produce video in multiple formats to cover a range of mobile platforms.
- Size videos correctly; ensure they don't overflow their containers.
- Accessibility matters; add the `track` element as a child of the `video` element.

Add the video element

Add the `video` element to load, decode, and play video in your site:



```
<video src="chrome.webm" type="video/webm">
    <p>Your browser does not support the video element.</p>
</video>
```

Specify multiple file formats

Not all browsers support the same video formats. The **`<source>`** element lets you specify multiple formats as a fallback in case the user's browser doesn't support one of them.

For example:

```
<video controls>
  <source src="https://storage.googleapis.com/webfundamentals-assets/videos/chrom
  <source src="https://storage.googleapis.com/webfundamentals-assets/videos/chrom
  <p>This browser does not support the video element.</p>
</video>
```

Try it

When the browser parses the **`<source>`** tags, it uses the optional **`type`** attribute to help decide which file to download and play. If the browser supports **`WebM`**, it plays chrome.webm; if not, it checks whether it can play MPEG-4 videos.

Check out A Digital Media Primer for Geeks to find out more about how video and audio work on the web.

This approach has several advantages over serving different HTML or server-side scripting, especially on mobile:

- Developers can list formats in order of preference.
- Native client-side switching reduces latency; only one request is made to get content.
- Letting the browser choose a format is simpler, quicker, and potentially more reliable than using a server-side support database with user-agent detection.

- Specifying each file source's type improves network performance; the browser can select a video source without having to download part of the video to "sniff" the format.

All of these points are especially important in mobile contexts, where bandwidth and latency are at a premium and the user's patience is likely to be limited. Not including a type attribute can affect performance when there are multiple sources with unsupported types.

Using your mobile browser developer tools, compare network activity with type attributes and without type attributes.

Also check the response headers in your browser developer tools to ensure your server reports the right MIME type; otherwise video source type checks won't work.

Specify start and end times

Save bandwidth and make your site feel more responsive: use the Media Fragments API to add start and end times to the video element.

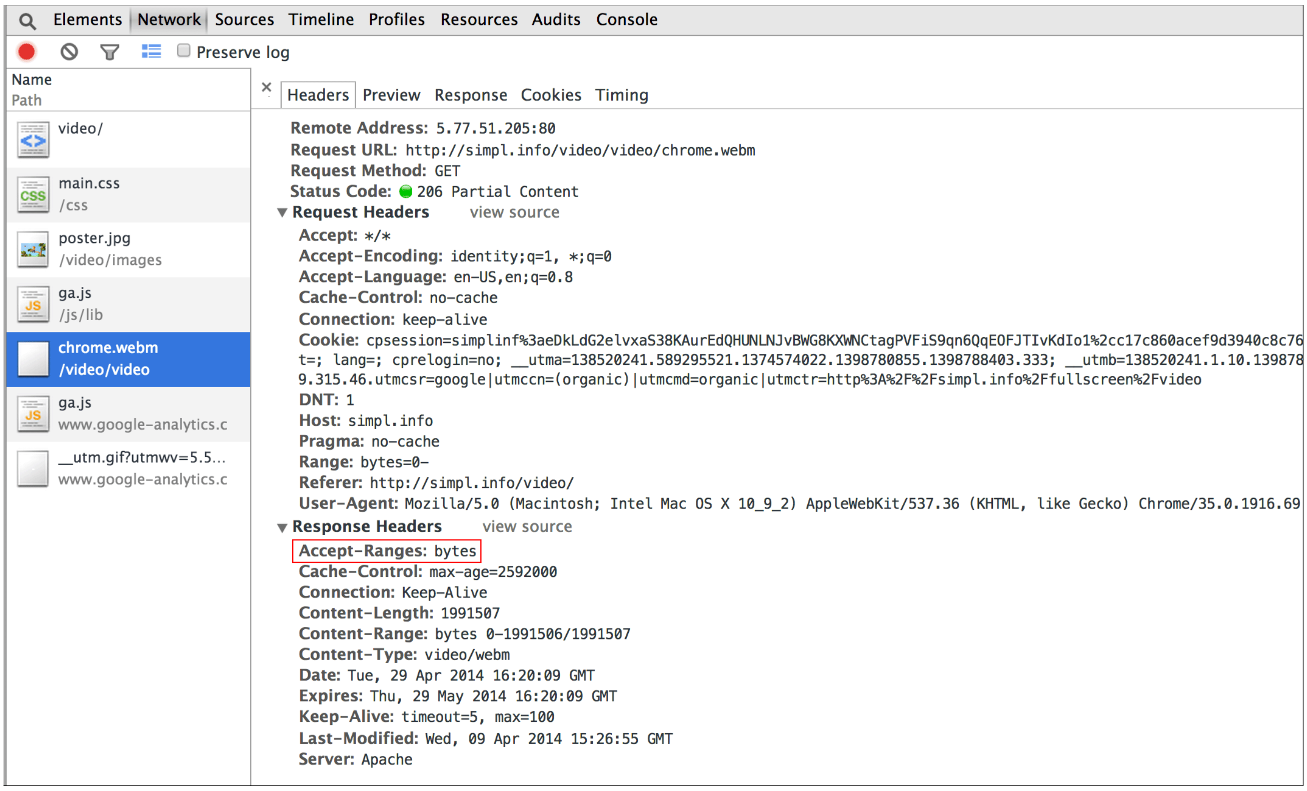
To add a media fragment, you simply add `#t=[start_time][,end_time]` to the media URL. For example, to play the video between seconds 5 through 10, specify:

```
<source src="video/chrome.webm#t=5,10" type="video/webm">
```

You can also use the Media Fragments API to deliver multiple views on the same video–like cue points in a DVD–without having to encode and serve multiple files.

Caution: Most platforms except iOS support the Media Fragments API. Also, make sure that your server supports Range Requests. By default, most servers enable Range Requests, but some hosting services may turn them off.

Using your browser developer tools, check for `Accept-Ranges: bytes` in the response headers:



```
Remote Address: 5.77.51.205:80
Request URL: http://simpl.info/video/video/chrome.webm
Request Method: GET
Status Code: ● 206 Partial Content
▼ Request Headers      view source
  Accept: */*
  Accept-Encoding: identity;q=1, *;q=0
  Accept-Language: en-US,en;q=0.8
  Cache-Control: no-cache
  Connection: keep-alive
  Cookie: cpsession=simplinf%3aeDkLdG2elvxaS38KAurEdQHUNLNJvBWG8KXWNCtagPVFiS9qn6QqEOFJTIvKdIo1%2cc17c860acef9d3940c8c76
  t=; lang=; cprelogin=no; __utma=138520241.589295521.1374574022.1398780855.1398788403.333; __utmb=138520241.1.10.139878
  9.315.46.utmcsr=google|utmccn=(organic)|utmcmd=organic|utmctr=http%3A%2F%2Fsimpl.info%2Ffullscreen%2Fvideo
  DNT: 1
  Host: simpl.info
  Pragma: no-cache
  Range: bytes=0-
  Referer: http://simpl.info/video/
  User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/35.0.1916.69
▼ Response Headers      view source
  Accept-Ranges: bytes
  Cache-Control: max-age=2592000
  Connection: Keep-Alive
  Content-Length: 1991507
  Content-Range: bytes 0-1991506/1991507
  Content-Type: video/webm
  Date: Tue, 29 Apr 2014 16:20:09 GMT
  Expires: Thu, 29 May 2014 16:20:09 GMT
  Keep-Alive: timeout=5, max=100
  Last-Modified: Wed, 09 Apr 2014 15:26:55 GMT
  Server: Apache
```

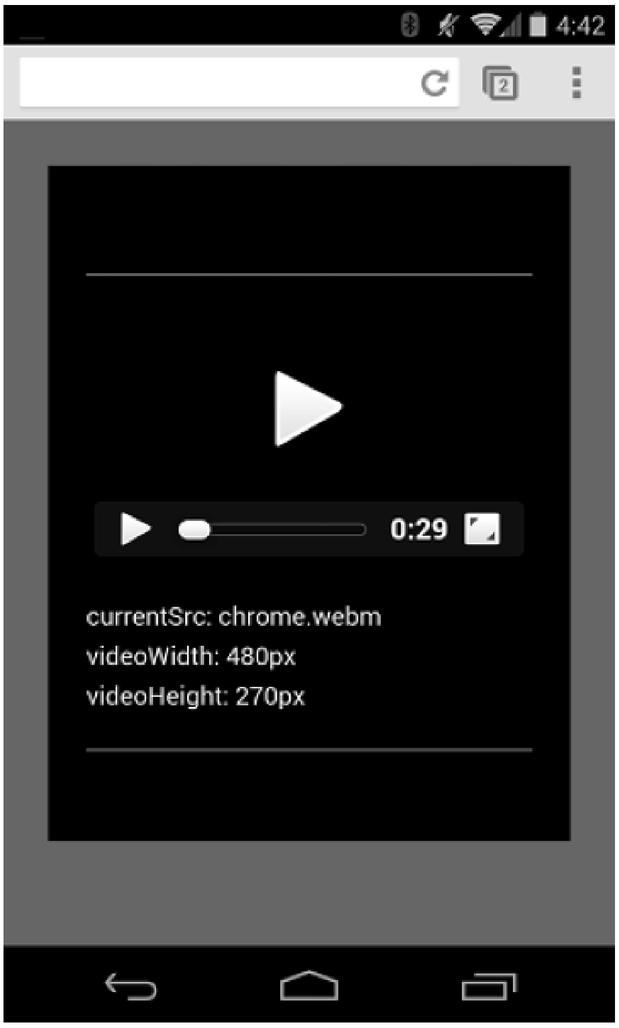
Include a poster image

Add a poster attribute to the `video` element so that your users have an idea of the content as soon as the element loads, without needing to download video or start playback.

```
<video poster="poster.jpg" ...>
  ...
</video>
```

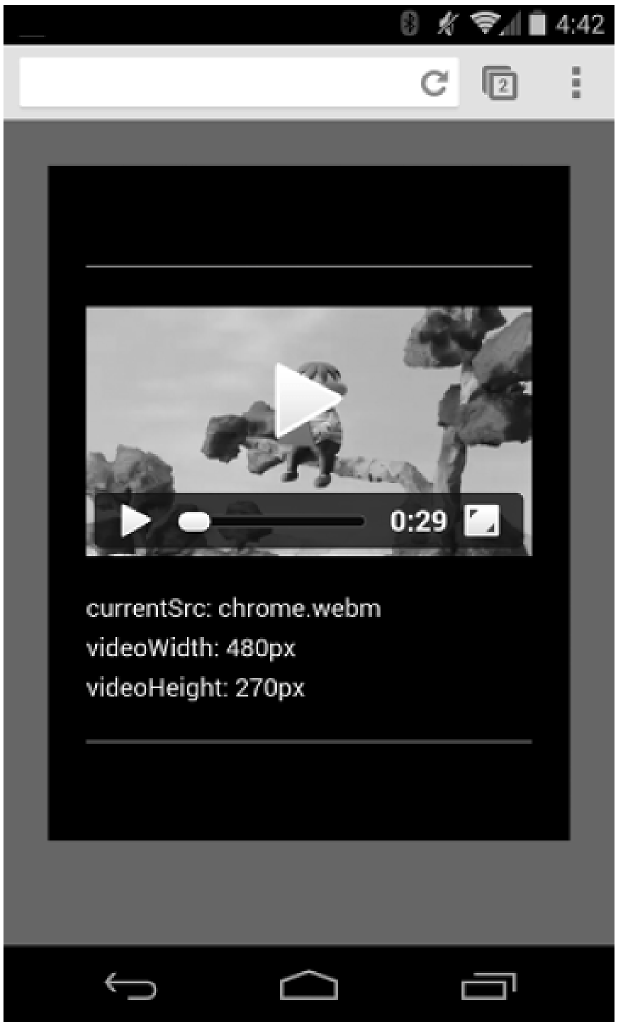
A poster can also be a fallback if the video `src` is broken or if none of the video formats supplied are supported. The only downside to poster images is an additional file request, which consumes some bandwidth and requires rendering. For more information see Image Optimization.

Here's a side-by-side comparison of videos without and with a poster image–we've made the poster image grayscale to prove it's not the video:



currentSrc: chrome.webm
videoWidth: 480px
videoHeight: 270px

Android Chrome screenshot, portrait: no poster



currentSrc: chrome.webm
videoWidth: 480px
videoHeight: 270px

Android Chrome screenshot, portrait: with poster

Provide alternatives for legacy platforms

Not all video formats are supported on all platforms. Check which formats are supported on the major platforms and make sure your video works in each of these.

Check which formats are supported

Use `canPlayType()` to find out which video formats are supported. The method takes a string argument consisting of a `mime-type` and optional codecs and returns one of the following values:

Return value and Description	
(empty string)	The container and/or codec isn't supported.

Return value and Description

`maybe`	The container and codec(s) might be supported, but the browser will need to download some video to check.
`probably`	The format appears to be supported.

Here are some examples of `canPlayType()` arguments and return values when run in Chrome:

Type and Response

`video/xyz`	(empty string)
`video/xyz; codecs="avc1.42E01E, mp4a.40.2"`	(empty string)
`video/xyz; codecs="nonsense, noise"`	(empty string)
`video/mp4; codecs="avc1.42E01E, mp4a.40.2"`	`probably`
`video/webm`	`maybe`
`video/webm; codecs="vp8, vorbis"`	`probably`

Produce video in multiple formats

There are lots of tools to help save the same video in different formats:

- Desktop tools: FFmpeg
- GUI applications: Miro, HandBrake, VLC
- Online encoding/transcoding services: Zencoder, Amazon Elastic Encoder

Check which format was used

Want to know which video format was actually chosen by the browser?

In JavaScript, use the video's `currentSrc` property to return the source used.

Size videos correctly

When it comes to keeping your users happy, file size is important.

TL;DR

- Don't serve videos with a larger frame size or higher quality than the platform can handle.
- Don't make your videos any longer than they need be.
- Long videos can cause hiccups with download and seeking; some browsers may have to wait until the video downloads before beginning playback.

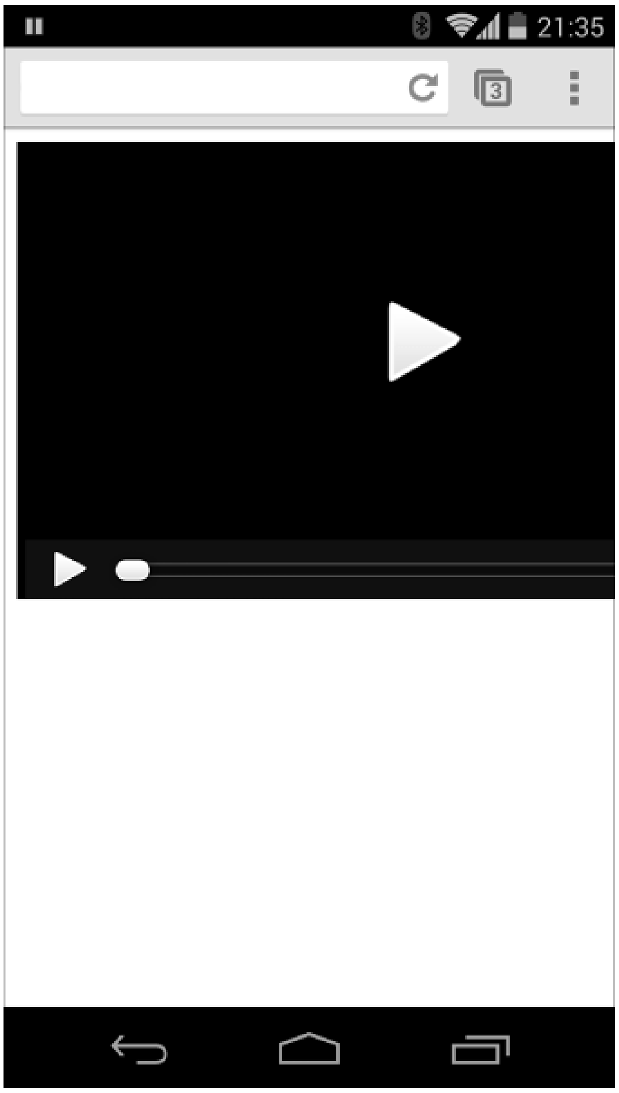
Check video size

The actual video frame size, as encoded, might be different from the video element dimensions (just as an image might not be displayed using its actual dimensions).

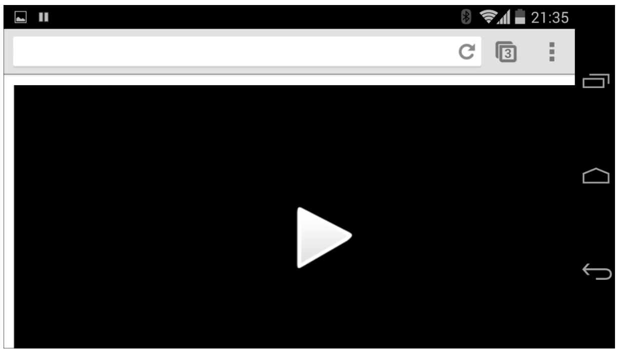
To check the encoded size of a video, use the video element `videoWidth` and `videoHeight` properties. `width` and `height` return the dimensions of the video element, which may have been sized using CSS or inline width and height attributes.

Ensure videos don't overflow containers

When video elements are too big for the viewport, they may overflow their container, making it impossible for the user to see the content or use the controls.



Android Chrome screenshot, portrait: unstyled video element overflows viewport



Android Chrome screenshot, landscape: unstyled video element overflows viewport

You can control video dimensions using JavaScript or CSS. JavaScript libraries and plugins such as FitVids make it possible to maintain appropriate size and aspect ratio, even for Flash videos from YouTube and other sources.

Use CSS media queries to specify the size of elements depending on the viewport dimensions; `max- width: 100%` is your friend.

For media content in iframes (such as YouTube videos), try a responsive approach (like the one proposed by John Surdakowski).

Caution: Don't force element sizing that results in an aspect ratio different from the original video. Squashed or stretched looks bad.

CSS:

```
.video-container {
    position: relative;
    padding-bottom: 56.25%;
    padding-top: 0;
    height: 0;
    overflow: hidden;
}

.video-container iframe,
.video-container object,
.video-container embed {
    position: absolute;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
}
```

HTML:

```
<div class="video-container">
  <iframe src="//www.youtube.com/embed/l-BA9Ee2XuM"
          frameborder="0" width="560" height="315">
  </iframe>
</div>
```

Try it

Compare the responsive sample to the unresponsive version.

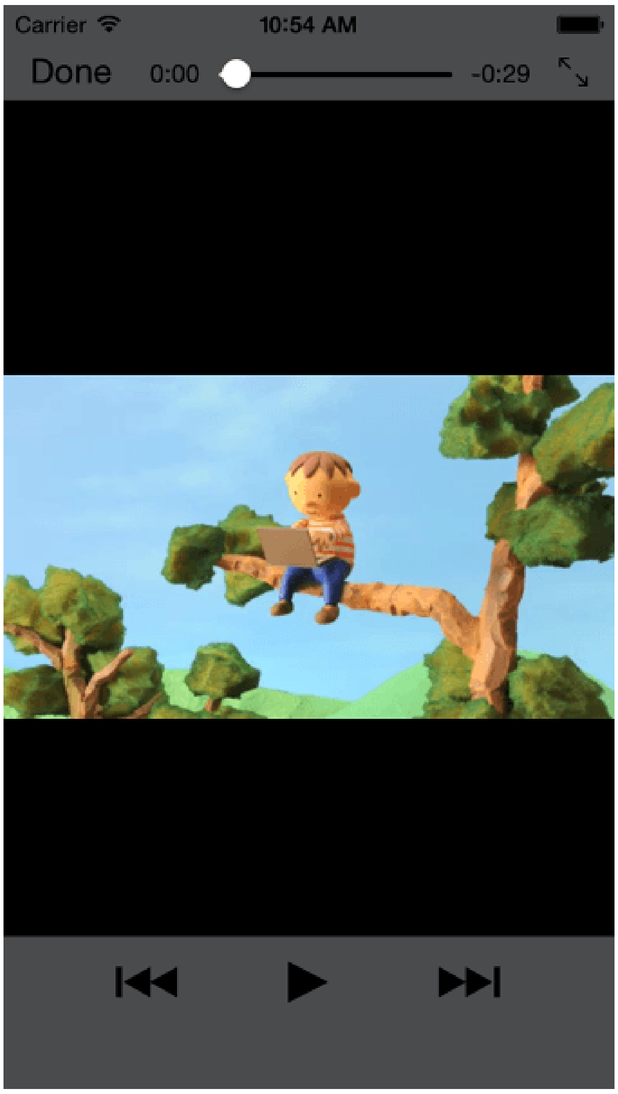
Customize the video player

Different platforms display video differently. Mobile solutions need to consider device orientation. Use the Fullscreen API to control the fullscreen view of video content.

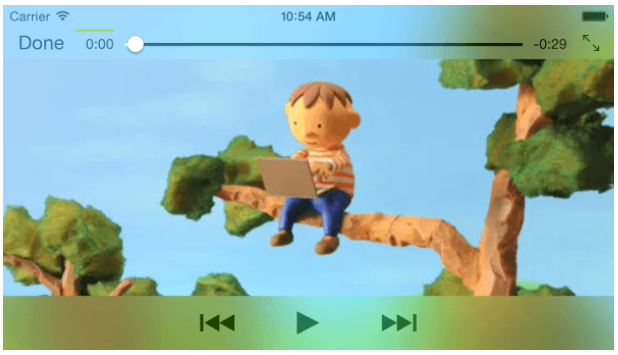
How device orientation works across devices

Device orientation isn't an issue for desktop monitors or laptops, but it's hugely important when considering web page design for mobile and tablets.

Safari on iPhone does a good job of switching between portrait and landscape orientation:

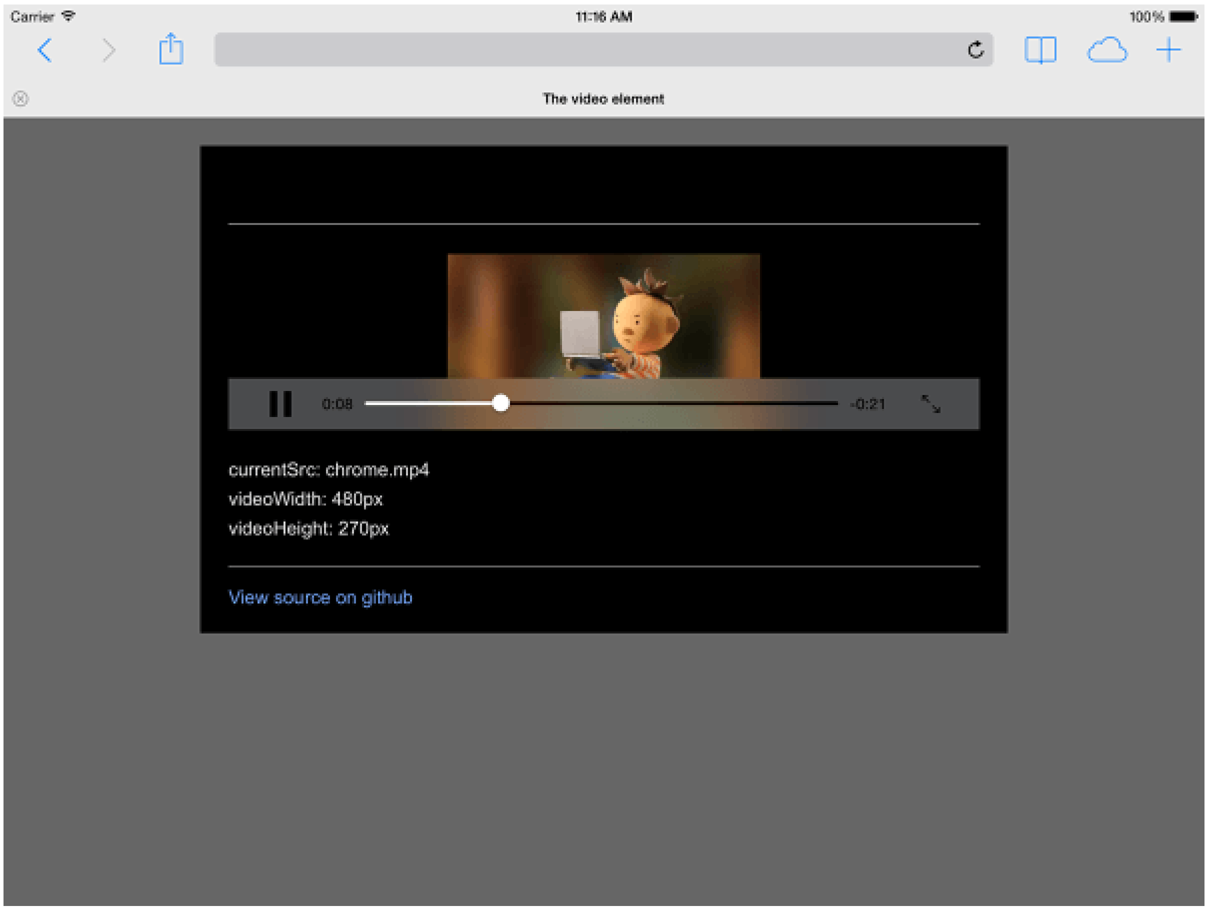


Screenshot of video playing in Safari on iPhone, portrait



Screenshot of video playing in Safari on iPhone, landscape

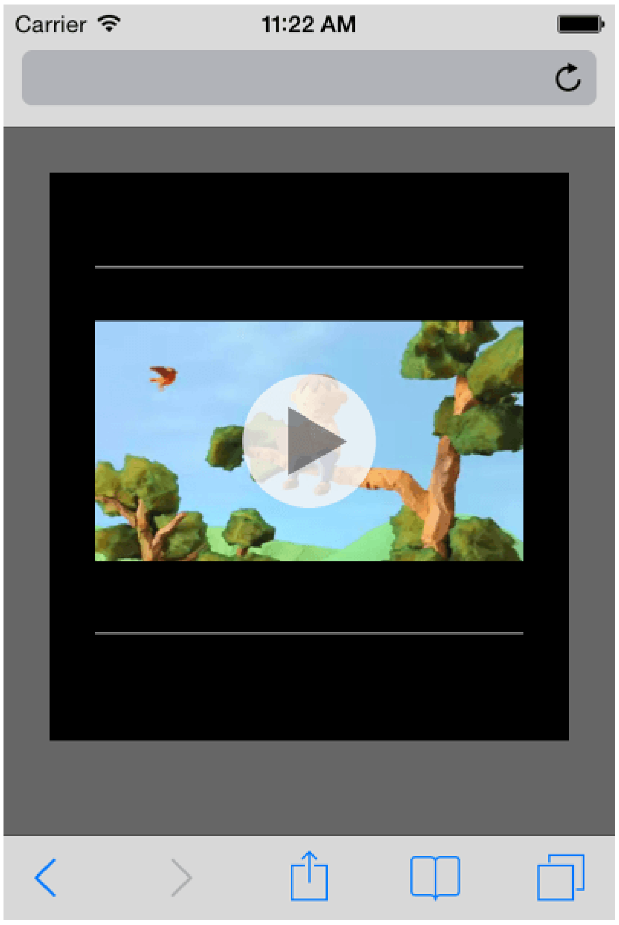
Device orientation on an iPad and Chrome on Android can be problematic. For example, without any customization a video playing on an iPad in landscape orientation looks like this:



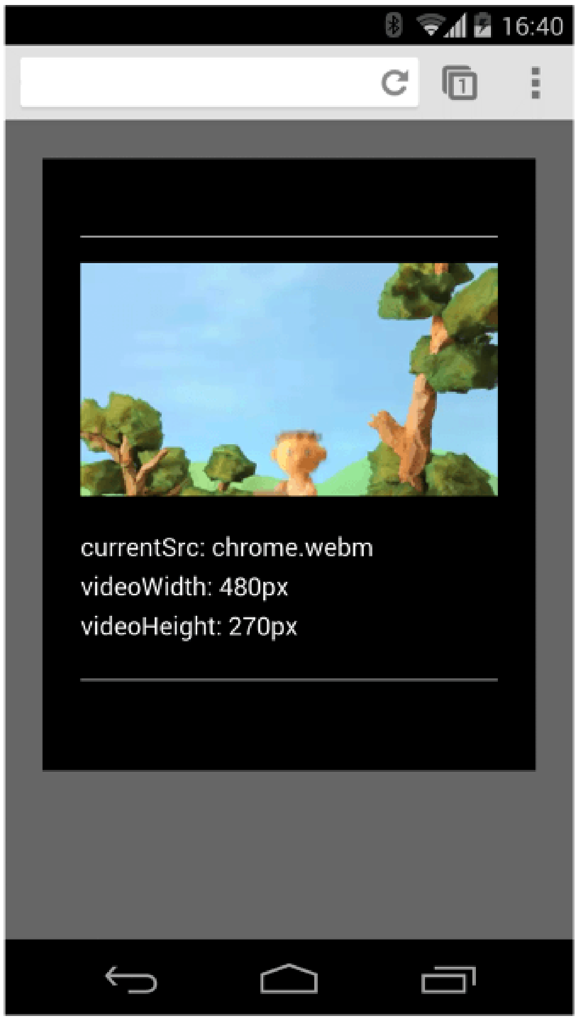
currentSrc: chrome.mp4
videoWidth: 480px
videoHeight: 270px

Setting the video `width: 100%` or `max-width: 100%` with CSS can resolve many device orientation layout problems. You may also want to consider fullscreen alternatives.

Inline or fullscreen display

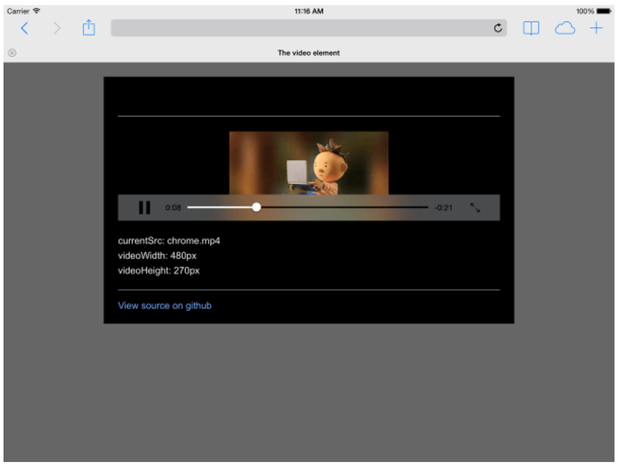


Different platforms display video differently. Safari on an iPhone displays a video element inline on a web page, but plays video back in fullscreen mode:



currentSrc: chrome.webm
videoWidth: 480px
videoHeight: 270px

On Android, users can request request fullscreen mode by clicking the fullscreen icon. But the default is to play video inline:



currentSrc: chrome.mp4
videoWidth: 480px
videoHeight: 270px

View source on github

Safari on an iPad plays video inline:

Control fullscreening of content

For platforms that do not force fullscreen video playback, the Fullscreen API is widely supported. Use this API to control fullscreening of content, or the page.

To full screen an element, like a video:

```
elem.requestFullScreen();
```

To full screen the entire document:

```
document.body.requestFullScreen();
```

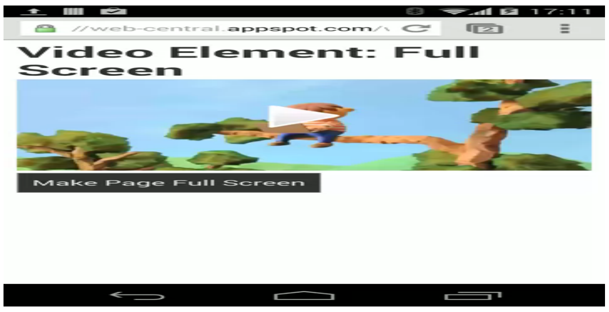
You can also listen for fullscreen state changes:

```
video.addEventListener("fullscreenchange", handler);
```

Or, check to see if the element is currently in fullscreen mode:

```
console.log("In full screen mode: ", video.displayingFullscreen);
```

You can also use the CSS `:fullscreen` pseudo-class to change the way elements are displayed in fullscreen mode.



On devices that support the Fullscreen API, consider using thumbnail images as placeholders for video:

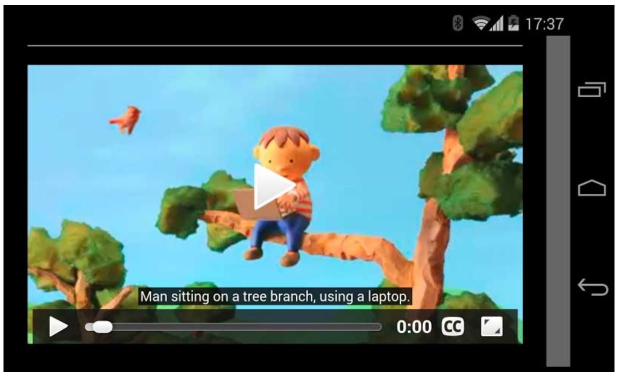
To see this in action, check out the demo.

Dogfood: `requestFullScreen()` may be vendor prefixed and may require extra code for full cross browser compatibility.

Accessibility matters

Accessibility isn't a feature. Users who can't hear or see won't be able to experience a video at all without captions or descriptions. The time it takes to add these to your video is much less than the bad experience you're delivering to users. Provide at least a base experience for all users.

Include captions to improve accessibility



To make media more accessible on mobile, include captions or descriptions using the track element.

Add track element

It's very easy to add captions to your video–simply add a track element as a child of the video element:

```
<video controls>
  <source src="https://storage.googleapis.com/webfundamentals-assets/videos/chrom
  <source src="https://storage.googleapis.com/webfundamentals-assets/videos/chrom
  <track src="chrome-subtitles-en.vtt" label="English captions"
         kind="captions" srclang="en" default>
  <p>This browser does not support the video element.</p>
</video>
```

Try it

The track element `src` attribute gives the location of the track file.

Define captions in track file

A track file consists of timed "cues" in WebVTT format:

```
WEBVTT

00:00.000 --> 00:04.000
Man sitting on a tree branch, using a laptop.

00:05.000 --> 00:08.000
The branch breaks, and he starts to fall.

...
```

Quick Reference

Video element attributes

For the complete list of video element attributes and their definitions, see the video element spec.

Attribute	Description
`src`	Address (URL) of the video.
`poster`	Address (URL) of an image file that the browser can show as soon as the video element is displayed without downloading video content.
`preload`	Hints to the browser that preloading metadata (or some video) in advance of playback is worthwhile. Options are none, metadata, or auto (see Preload section for details).
`autoplay`	Start download and playback as soon as possible (see Autoplay section for details).
`loop`	Loop the video.
`controls`	Show the default video controls (play, pause, etc.).

Autoplay

On desktop, `autoplay` tells the browser to download and play the video immediately. On mobile, don't assume `autoplay` will always work. See the WebKit blog, for instance.

Even on platforms where autoplay is possible, you need to consider whether it's a good idea to enable it:

- Data usage can be expensive.
- Causing media to download and playback to begin without asking first, can unexpectedly hog bandwidth and CPU, and thereby delay page rendering.
- Users may be in a context where playing video or audio is intrusive.

Autoplay behavior is configurable in the Android WebView via the WebSettings API. It defaults to true but a WebView app can choose to disable it.

Preload

The `preload` attribute provides a hint to the browser as to how much information or content to preload.

Value & Description

`none`	The user may not even watch the video–don't preload anything.
`metadata`	Metadata (duration, dimensions, text tracks) should be preloaded, but with minimal video.
`auto`	Downloading the entire video right away is considered desirable.

The `preload` attribute has different effects on different platforms. For example, Chrome buffers 25 seconds of video on desktop but none on iOS or Android. This means that on mobile, there may be playback startup delays that don't happen on desktop. See Steve Souders' blog for full details.

JavaScript

The HTML5 Rocks Video article does a great job of summarizing the JavaScript properties, methods, and events that can be used to control video playback.

Properties

Property & Description

`currentTime`	Get or set playback position in seconds.
`volume`	Get or set current volume level for the video.
`muted`	Get or set audio muting.

Property & Description	
`playbackRate`	Get or set playback rate; 1 is normal speed forward.
`buffered`	Information about how much of the video has been buffered and is ready to play.
`currentSrc`	The address of the video being played.
`videoWidth`	Width of the video in pixels (which may be different from the video element width).
`videoHeight`	Height of the video in pixels (which may be different from the video element height).

Methods

Method & Description	
`load()`	Load or reload a video source without initiating playback: for example, when the video src is changed using JavaScript.
`play()`	Play the video from its current location.
`pause()`	Pause the video at its current location.
`canPlayType('format')`	Find out which formats are supported (see Check which formats are supported).

On mobile `play()` and `pause()` don't work unless called in response to user action such as clicking a button: see the demo. (Likewise, you can't initiate playback for content such as embedded YouTube videos.)

Events

These are only a subset of the media events that may be fired. Refer to the Media events page on the Mozilla Developer Network for a complete listing.

Event & Description	
`canplaythrough`	Fired when enough data is available that the browser believes it can play the video completely without interruption.
ended	Fired when video has finished playing.

Event & Description	
`error`	Fired if an error occurs (see demo).
`playing`	Fired when video starts playing for the first time, after being paused, or when restarting.
`progress`	Fired periodically to indicate download progress.
`waiting`	Fired when an action is delayed pending completion of another action.
`loadedmetadata`	Fired when browser finishes loading metadata for video: duration, dimensions, and text tracks.