

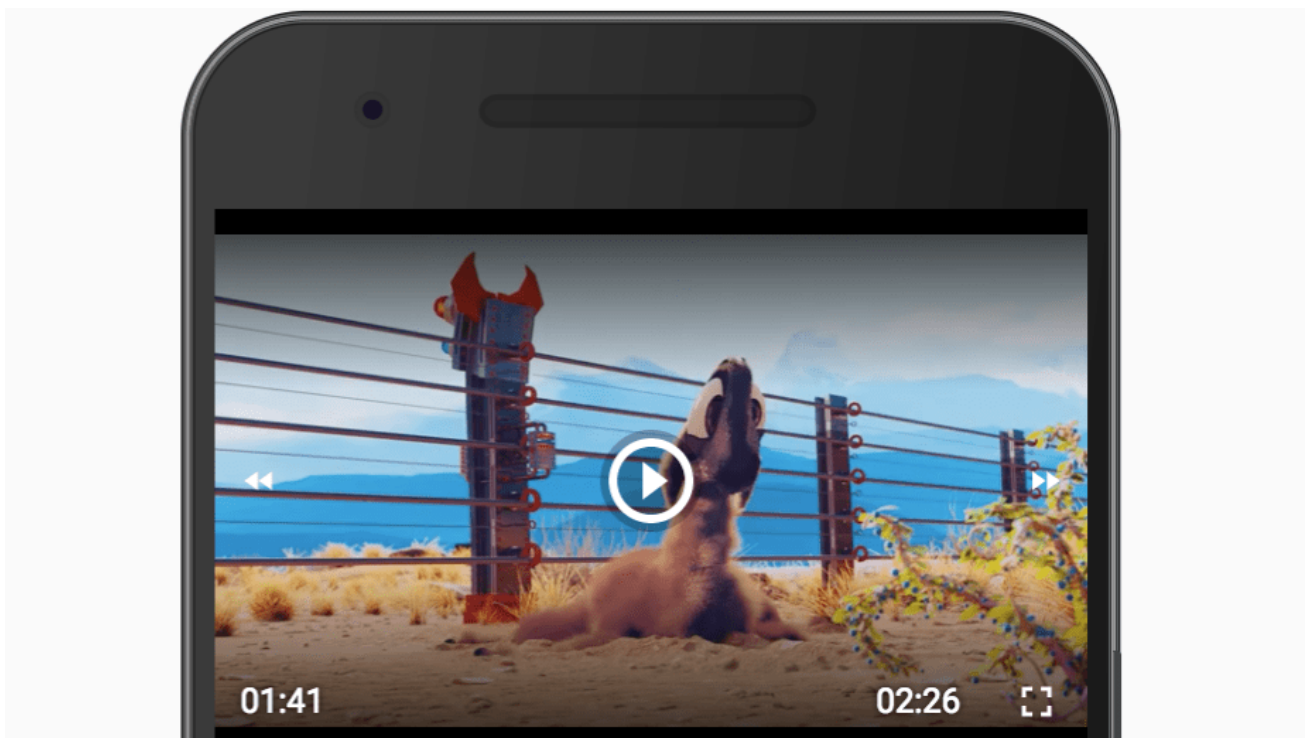
Mobile Web Video Playback



By [François Beaufort](#)

Dives into Chromium source code

How do you create the best mobile media experience on the Web? Easy! It all depends on user engagement and the importance you give to the media on a web page. I think we all agree that if video is THE reason for a user's visit, the user's experience has to be immersive and re-engaging.



In this article I show you how to enhance in a progressive way your media experience and make it more immersive thanks to a plethora of Web APIs. That's why we're going to build a simple mobile player experience with custom controls, fullscreen, and background playback. You can try the [sample](#) [↗](#) now and find [the code](#) [↗](#) in our GitHub repository.

Custom controls

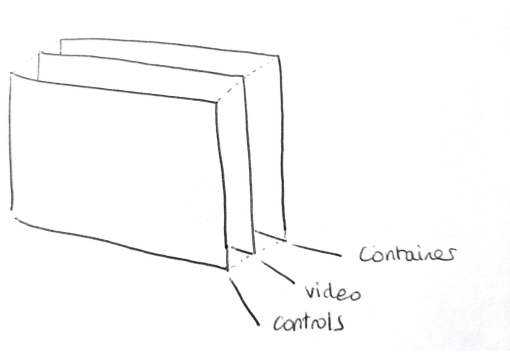


Figure 1.HTML Layout

As you can see, the HTML layout we're going to use for our media player is pretty simple: a `<div>` root element contains a `<video>` media element and a `<div>` child element dedicated to video controls.

Video controls we will cover later, include: a play/pause button, a fullscreen button, seek backward and forward buttons, and some elements for current time, duration and time tracking.

```
<div id="videoContainer">
  <video id="video" src="file.mp4"></video>
  <div id="videoControls"></div>
</div>
```



Read video metadata

First, let's wait for the video metadata to be loaded to set the video duration, the current time, and initialize the progress bar. Note that the `secondsToTimeCode()` function is a custom utility function I've written that converts a number of seconds to a string in "hh:mm:ss" format which is better suited in our case.

```
<div id="videoContainer">
  <video id="video" src="file.mp4"></video>
  <div id="videoControls">
    <div id="videoCurrentTime"></div>
    <div id="videoDuration"></div>
    <div id="videoProgressBar"></div>
  </div>
</div>
```



```
video.addEventListener('loadedmetadata', function() {
  videoDuration.textContent = secondsToTimeCode(video.duration);
  videoCurrentTime.textContent = secondsToTimeCode(video.currentTime);
```



```
videoProgressBar.style.transform = `scaleX(${video.currentTime / video.duration
});
```

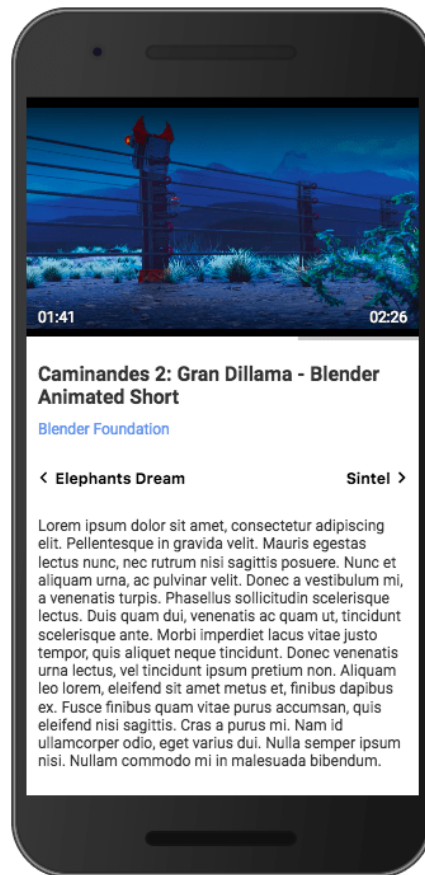


Figure 2. Media Player showing video metadata

Play/pause video

Now that video metadata are loaded, let's add our first button that lets user play and pause video with `video.play()` and `video.pause()` depending on its playback state.

```
<div id="videoContainer">
  <video id="video" src="file.mp4"></video>
  <div id="videoControls">
    <button id="playPauseButton"></button>
    <div id="videoCurrentTime"></div>
    <div id="videoDuration"></div>
    <div id="videoProgressBar"></div>
  </div>
</div>
```



```
playPauseButton.addEventListener('click', function(event) {
  event.stopPropagation();
  if (video.paused) {
    video.play();
  } else {
    video.pause();
  }
});
```



Note: I call `event.stopPropagation()` to prevent parent handlers (e.g. video controls) from being notified of the click event.

Rather than adjusting our video controls in the `click` event listener, we use the `play` and `pause` video events. Making our controls events based helps with flexibility (as we'll see later with the Media Session API) and will allow us to keep our controls in sync if the browser intervenes in the playback. When video starts playing, we change the button state to "pause" and hide the video controls. When the video pauses, we simply change button state to "play" and show the video controls.

```
video.addEventListener('play', function() {
  playPauseButton.classList.add('playing');
});

video.addEventListener('pause', function() {
  playPauseButton.classList.remove('playing');
});
```



When time indicated by video `currentTime` attribute changed via the `timeupdate` video event, we also update our custom controls if they're visible.

```
video.addEventListener('timeupdate', function() {
  if (videoControls.classList.contains('visible')) {
    videoCurrentTime.textContent = secondsToTimeCode(video.currentTime);
    videoProgressBar.style.transform = `scaleX(${video.currentTime / video.duration})`;
  }
});
```



When the video ends, we simply change button state to "play", set video `currentTime` back to 0 and show video controls for now. Note that we could also choose to load automatically another video if the user has enabled some kind of "AutoPlay" feature.

```
video.addEventListener('ended', function() {
  playPauseButton.classList.remove('playing');
```



```
    video.currentTime = 0;
  });
```

Seek backward and forward

Let's continue and add "seek backward" and "seek forward" buttons so that user can easily skip some content.

```
<div id="videoContainer">
  <video id="video" src="file.mp4"></video>
  <div id="videoControls">
    <button id="playPauseButton"></button>
    <button id="seekForwardButton"></button>
    <button id="seekBackwardButton"></button>
    <div id="videoCurrentTime"></div>
    <div id="videoDuration"></div>
    <div id="videoProgressBar"></div>
  </div>
</div>
```



```
var skipTime = 10; // Time to skip in seconds
```



```
seekForwardButton.addEventListener('click', function(event) {
  event.stopPropagation();
  video.currentTime = Math.min(video.currentTime + skipTime, video.duration);
});
```

```
seekBackwardButton.addEventListener('click', function(event) {
  event.stopPropagation();
  video.currentTime = Math.max(video.currentTime - skipTime, 0);
});
```

As before, rather than adjusting video styling in the `click` event listeners of these buttons, we'll use the fired `seeking` and `seeked` video events to adjust video brightness. My custom `seeking` CSS class is as simple as `filter: brightness(0);`.

```
video.addEventListener('seeking', function() {
  video.classList.add('seeking');
});
```




```
video.addEventListener('seeked', function() {
  video.classList.remove('seeking');
});
```

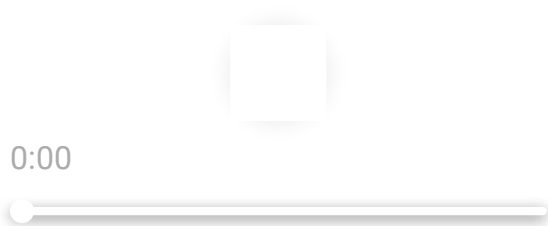
Here's below what we have created so far. In the next section, we'll implement the fullscreen button.



Fullscreen

Here we are going to take advantage of several Web APIs to create a perfect and seamless fullscreen experience. To see it in action, check out the [sample](#) .

Obviously, you don't have to use all of them. Just pick the ones that make sense to you and combine them to create your custom flow.



Prevent automatic fullscreen


On iOS, video elements automatically enter fullscreen mode when media playback begins. As we're trying to tailor and control as much as possible our media experience across mobile browsers, I recommend you set the **playsinline** attribute of the video element to force it to play inline on iPhone and not enter fullscreen mode when playback begins. Note that this has no side effects on other browsers.

```
<div id="videoContainer">
  <video id="video" src="file.mp4" playsinline></video>
  <div id="videoControls">...</div>
</div>
```



Caution: Set **playsinline** only if you provide your own media controls or show native controls with **<video controls>**.

Toggle fullscreen on button click

Now that we prevent automatic fullscreen, we need to handle ourselves the fullscreen mode for the video with the [Fullscreen API](#) . When user clicks the "fullscreen button", let's exit fullscreen mode with `document.exitFullscreen()` if fullscreen mode is currently in use by the document. Otherwise, request fullscreen on the video container with the method `requestFullscreen()` if available or fallback to `webkitEnterFullscreen()` on the video element only on iOS.

Note: I'm going to use a [tiny shim](#) for the Fullscreen API in code snippets below that will take care of prefixes as the API is not unprefixed yet at that time. You may want to use [screenfull.js](#) wrapper as well.

```
<div id="videoContainer">  
  <video id="video" src="file.mp4"></video>  
  <div id="videoControls">  
    <button id="playPauseButton"></button>  
    <button id="seekForwardButton"></button>  
    <button id="seekBackwardButton"></button>  
    <button id="fullscreenButton"></button>  
    <div id="videoCurrentTime"></div>  
    <div id="videoDuration"></div>  
    <div id="videoProgressBar"></div>  
  </div>  
</div>
```



```
fullscreenButton.addEventListener('click', function(event) {  
  event.stopPropagation();  
  if (document.fullscreenElement) {  
    document.exitFullscreen();  
  } else {  
    requestFullscreenVideo();  
  }  
});
```



```
function requestFullscreenVideo() {  
  if (videoContainer.requestFullscreen) {  
    videoContainer.requestFullscreen();  
  } else {  
    video.webkitEnterFullscreen();  
  }  
}
```

```

    }
}

document.addEventListener('fullscreenchange', function() {
    fullscreenButton.classList.toggle('active', document.fullscreenElement);
});

```

0:00

Toggle fullscreen on screen orientation change

As user rotates device in landscape mode, let's be smart about this and automatically request fullscreen to create an immersive experience. For this, we'll need the [Screen Orientation API](#) [↗](#) which is not yet supported everywhere and still prefixed in some browsers at that time. Thus, this will be our first progressive enhancement.

How does this work? As soon as we detect the screen orientation changes, let's request fullscreen if the browser window is in landscape mode (that is, its width is greater than its height). If not, let's exit fullscreen. That's all.

```

if ('orientation' in screen) {
    screen.orientation.addEventListener('change', function() {
        // Let's request fullscreen if user switches device in landscape mode.
        if (screen.orientation.type.startsWith('landscape')) {
            requestFullscreenVideo();
        } else if (document.fullscreenElement) {
            document.exitFullscreen();
        }
    });
}

```



Note: This may silently fail in browsers that don't [allow requesting fullscreen from the orientation change event](#).

Lock screen in landscape on button click

As video may be better viewed in landscape mode, we may want to lock screen in landscape when user clicks the "fullscreen button". We're going to combine the previously used [Screen Orientation API](#) [↗](#) and some [media queries](#) [↗](#) to make sure this experience is the best.

Locking screen in landscape is as easy as calling `screen.orientation.lock('landscape')`. However, we should do this only when device is in portrait mode with `matchMedia('(orientation: portrait)')` and can be held in one hand with `matchMedia('(max-device-width: 768px)')` as this wouldn't be a great experience for users on tablet.

```
fullscreenButton.addEventListener('click', function(event) {  
  event.stopPropagation();  
  if (document.fullscreenElement) {  
    document.exitFullscreen();  
  } else {  
    requestFullscreenVideo();  
    lockScreenInLandscape();  
  }  
});
```




```
function lockScreenInLandscape() {  
  if (!('orientation' in screen)) {  
    return;  
  }  
  // Let's force landscape mode only if device is in portrait mode and can be held  
  if (matchMedia('(orientation: portrait) and (max-device-width: 768px)').matches  
      screen.orientation.lock('landscape');  
}
```




Unlock screen on device orientation change

You may have noticed the lock screen experience we've just created isn't perfect though as we don't receive screen orientation changes when screen is locked.

In order to fix this, let's use the [Device Orientation API](#)  if available. This API provides information from the hardware measuring a device's position and motion in space: gyroscope and digital compass for its orientation, and accelerometer for its velocity. When we detect a device orientation change, let's unlock screen with `screen.orientation.unlock()` if user holds device in portrait mode and screen is locked in landscape mode.

```
function lockScreenInLandscape() {  
  if (!('orientation' in screen)) {  
    return;  
  }  
  // Let's force landscape mode only if device is in portrait mode and can be hel  
  if (matchMedia('(orientation: portrait) and (max-device-width: 768px)').matches  
    screen.orientation.lock('landscape')  
    .then(function() {  
      listenToDeviceOrientationChanges();  
    }));  
}
```

```
function listenToDeviceOrientationChanges() {  
  if (!('DeviceOrientationEvent' in window)) {  
    return;  
  }  
  var previousDeviceOrientation, currentDeviceOrientation;  
  window.addEventListener('deviceorientation', function onDeviceOrientationChange  
    // event.beta represents a front to back motion of the device and  
    // event.gamma a left to right motion.  
    if (Math.abs(event.gamma) > 10 || Math.abs(event.beta) < 10) {  
      previousDeviceOrientation = currentDeviceOrientation;  
      currentDeviceOrientation = 'landscape';  
      return;  
    }  
    if (Math.abs(event.gamma) < 10 || Math.abs(event.beta) > 10) {  
      previousDeviceOrientation = currentDeviceOrientation;  
      // When device is rotated back to portrait, let's unlock screen orientation  
      if (previousDeviceOrientation == 'landscape') {  
        screen.orientation.unlock();  
        window.removeEventListener('deviceorientation', onDeviceOrientationChange  
      }  
    }  
  });  
}
```

As you can see, this is the seamless fullscreen experience we were looking for. To see this in action, check out the [sample](#) .



0:00

Background playback

When you detect a web page or a video in the web page is not visible anymore, you may want to update your analytics to reflect this. This could also affect the current playback as in picking a different track, pause it, or even show custom buttons to the user for instance.

Pause video on page visibility change

With the [Page Visibility API](#), we can determine the current visibility of a page and be notified of visibility changes. Code below pauses video when page is hidden. This happens when screen lock is active or when you switch tabs for instance.

As most mobile browsers now offer controls outside of the browser that allow resuming a paused video, I recommend you set this behaviour only if user is allowed to play in the background.

```
document.addEventListener('visibilitychange', function() {  
  // Pause video when page is hidden.  
  if (document.hidden) {  
    video.pause();  
  }  
});
```



Note: Chrome for Android already pauses videos when page is hidden.

Show/hide mute button on video visibility change

If you use the new [Intersection Observer API](#), you can be even more granular at no cost. This API lets you know when an observed element enters or exits the browser's viewport.

Let's show/hide a mute button based on the video visibility in the page. If video is playing but not currently visible, a mini mute button will be shown in the bottom right corner of the page to give user control over video sound. The `volumechange` video event is used to update the mute button styling.

Note: If there are a lot of videos on a page, and it is using the Intersection Observer API to pause / mute offscreen video, you may want to reset video source with `video.src = null` instead since it will release significant resources in an infinite scroll case.

```
<button id="muteButton"></button>
```



```
if ('IntersectionObserver' in window) {  
  // Show/hide mute button based on video visibility in the page.  
  function onIntersection(entries) {  
    entries.forEach(function(entry) {  
      muteButton.hidden = video.paused || entry.isIntersecting;  
    });  
  }  
  var observer = new IntersectionObserver(onIntersection);  
  observer.observe(video);  
}  
  
muteButton.addEventListener('click', function() {  
  // Mute/unmute video on button click.  
  video.muted = !video.muted;  
});  
  
video.addEventListener('volumechange', function() {  
  muteButton.classList.toggle('active', video.muted);  
});
```



0:00



Play only one video at a time

If there are more than one video on a page, I would suggest you only play one and pause the other ones automatically so that user doesn't have to hear multiple audio tracks playing simultaneously.

```
// Note: This array should be initialized once all videos have been added.
var videos = Array.from(document.querySelectorAll('video'));

videos.forEach(function(video) {
  video.addEventListener('play', pauseOtherVideosPlaying);
});

function pauseOtherVideosPlaying(event) {
  var videosToPause = videos.filter(function(video) {
    return !video.paused && video !== event.target;
  });
  // Pause all other videos currently playing.
  videosToPause.forEach(function(video) { video.pause(); });
}
```

Customize Media Notifications

With the [Media Session API](#), you can also customize media notifications by providing metadata for the currently playing video. It also allows you to handle media related events such as seeking or track changing which may come from notifications or media keys. To see this in action, check out the [sample](#).

When your web app is playing audio or video, you can already see a media notification sitting in the notification tray. On Android, Chrome does its best to show appropriate information by using the document's title and the largest icon image it can find.

Let's see how to customize this media notification by setting some media session metadata such as the title, artist, album name, and artwork with the [Media Session API](#).

```
playPauseButton.addEventListener('click', function(event) {
  event.stopPropagation();
  if (video.paused) {
    video.play()
    .then(function() {
      setMediaSession();
    });
  } else {
    video.pause();
  }
});
```



```
function setMediaSession() {
  if (!('mediaSession' in navigator)) {
    return;
  }
  navigator.mediaSession.metadata = new MediaMetadata({
    title: 'Never Gonna Give You Up',
    artist: 'Rick Astley',
    album: 'Whenever You Need Somebody',
    artwork: [
      { src: 'https://dummyimage.com/96x96', sizes: '96x96', type: 'image/png' },
      { src: 'https://dummyimage.com/128x128', sizes: '128x128', type: 'image/png' },
      { src: 'https://dummyimage.com/192x192', sizes: '192x192', type: 'image/png' },
      { src: 'https://dummyimage.com/256x256', sizes: '256x256', type: 'image/png' },
      { src: 'https://dummyimage.com/384x384', sizes: '384x384', type: 'image/png' },
      { src: 'https://dummyimage.com/512x512', sizes: '512x512', type: 'image/png' }
    ]
  });
}
```

Once playback is done, you don't have to "release" the media session as the notification will automatically disappear. Keep in mind that current `navigator.mediaSession.metadata` will be used when any playback starts. This is why you need to update it to make sure you're always showing relevant information in the media notification.

If your web app provides a playlist, you may want to allow the user to navigate through your playlist directly from the media notification with some "Previous Track" and "Next Track" icons.



```
if ('mediaSession' in navigator) {
  navigator.mediaSession.setActionHandler('previoustrack', function() {
    // User clicked "Previous Track" media notification icon.
    playPreviousVideo(); // load and play previous video
  });
  navigator.mediaSession.setActionHandler('nexttrack', function() {
    // User clicked "Next Track" media notification icon.
    playNextVideo(); // load and play next video
  });
}
```

Note that media action handlers will persist. This is very similar to the event listener pattern except that handling an event means that the browser stops doing any default behaviour and uses this as a signal that your web app supports the media action. Hence, media action controls won't be shown unless you set the proper action handler.

By the way, unsetting a media action handler is as easy as assigning it to `null`.

The Media Session API allows you to show "Seek Backward" and "Seek Forward" media notification icons if you want to control the amount of time skipped.



```
if ('mediaSession' in navigator) {  
  let skipTime = 10; // Time to skip in seconds  
  
  navigator.mediaSession.setActionHandler('seekbackward', function() {  
    // User clicked "Seek Backward" media notification icon.  
    video.currentTime = Math.max(video.currentTime - skipTime, 0);  
  });  
  navigator.mediaSession.setActionHandler('seekforward', function() {  
    // User clicked "Seek Forward" media notification icon.  
    video.currentTime = Math.min(video.currentTime + skipTime, video.duration);  
  });  
}
```

The "Play/Pause" icon is always shown in the media notification and the related events are handled automatically by the browser. If for some reason the default behaviour doesn't work out, you can still [handle "Play" and "Pause" media events](#).

The cool thing about the Media Session API is that the notification tray is not the only place where media metadata and controls are visible. The media notification is synced automagically to any paired wearable device. And it also shows up on lock screens.



Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#). For details, see our [Site Policies](#). Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 2, 2018.