Safari HTML5 Audio and Video Guide

Developer

Contents

Figures, Tables, and Listings

About HTML5 Audio and Video

If you embed audio or video in your website, you should use HTML5.

HTML5 is the next major version of HTML, the primary standard that determines how web content interacts with browsers. HTML5 supports audio and video playback natively in the browser, without requiring a plug-in. With HTML5, you can add media to a webpage with just a line or two of code.

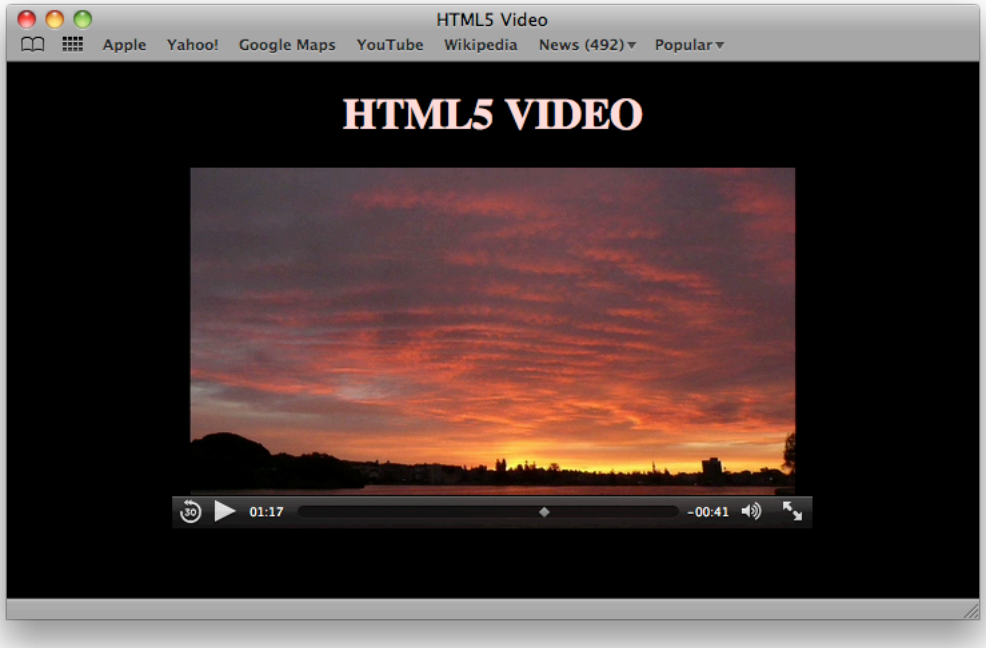
The HTML5 media elements provide simple fallback for browsers that still need to use plug-ins, so you can update your website to HTML5 today and still be compatible with older browsers.

When you use HTML5, you can create your own customized media controllers for rich interactivity using web-standard CSS and JavaScript.

The HTML5 `<audio>` and `<video>` tags make it simple to add media to your website. Just include the `<audio>` or `<video>` element, use the `src` attribute to identify the media source, and include the `controls` attribute.

```
<video src="Mymovie.mp4" controls> </video>
```

There are no plug-ins to install or configure. The audio or video downloads and plays in your webpage with built-in controls.



In Safari, the built-in video controls include a play/pause button, volume control, and a time scrubber. In Safari 5.0 and later on the desktop and on iOS 4.2 on the iPad, the controls also include a full-screen playback toggle on the lower right. The controls automatically fade out when the video is playing and fade in when the user hovers over the video or touches it.

If you want to provide your own media controller on the desktop or iPad, just leave out the `controls` attribute. HTML5 media elements expose a full set of methods, properties, and events to JavaScript for interactivity, and because the media elements are HTML, they can be styled using CSS to create exactly the look and feel you want.

In Safari 5.1 and later, you can choose any HTML element and expand it to fill the screen, allowing you to use your own custom controls while playing video in full-screen mode.

At a Glance

Safari supports the `<video>` and `<audio>` media elements on iOS 3.0 and later and in Safari 3.1 and later on the desktop (Mac OS X and Windows). Support for these media elements allows Safari and other HTML5-compliant browsers to play the indicated source media without using a plug-in.

To get the most out of HTML5 audio and video, you should first learn to create the HTML media elements, then learn how to control them using JavaScript, and finally learn to apply CSS styles to media elements and modify styles dynamically using JavaScript.

Create the HTML5 Media Elements

Relevant Chapter "Audio and Video HTML" (page 10)

To use HTML5 audio or video, start by creating an `<audio>` or `<video>` element, specifying a source URL for the media, and including the `controls` attribute.

```
<video src="http://Myserver.com/Path/Mymovie.mp4" controls>

</video>
```

Add Optional Attributes

You can set additional attributes to tell Safari that the media should autoplay or loop, for example, or specify a video height and width. You set boolean attributes such as `controls` or `autoplay` by including or omitting them—no value is required.

```
<video src="My.mp4"
       controls autoplay height="480" width="640">
</video>
```

For more information, see "Working with Attributes" (page 12).

Provide Alternate Sources

Not all browsers can play all media sources. Some browsers are able to play MPEG-4 or MP3 files, while others play only files compressed using codecs such as Ogg Vorbis. Desktop computers can typically play media using a wider assortment of compressors than mobile devices. Safari supports streaming delivery using HTTP Live Streaming, while some other browsers support only HTTP download. To provide the best experience for everyone, you can provide multiple versions of your media. List the sources in order of preference using separate `<source>` tags. The browser iterates through the list and plays the first source that it can.

```
<audio controls>
<source src="http://Example.com/mySong.aac">
<source src="mySong.oga">
</audio>
```

You don't have to rely on the file extension and delivery scheme to tell Safari about the media file. The `<source>` tag accepts attributes for MIME type and codecs as well. For details, see "Providing Multiple Sources" (page 13).

Fall Back in Good Order

Browsers that don't support HTML5 ignore the `<audio>` and `<video>` tags, and HTML5-savvy browsers ignore anything between the opening and closing tags except `<source>` tags, so it's easy to specify fallback behavior for older browsers. Just put the fallback HTML between the opening and closing `<audio>` or `<video>` tags (after any `<source>` tags).

```
<video src="My.mov" controls >
       <!-- fallback -->
       <p>Your browser does not support HTML5 video.</p>
</video>
```

Your fallback can be an `<object>` tag for a browser that needs a plug-in to play your media, a redirect to another page, or a simple error message telling the user what the problem is.

For more information, including examples of how to use a plug-in as a fallback, see "Specifying Fallback Behavior" (page 16).

Take Control Using JavaScript

Relevant Chapter "Controlling Media with JavaScript" (page 24)

HTML5 media elements expose methods, properties, and events to JavaScript. There are methods for playing, pausing, and changing the media source URL dynamically, for example. There are properties, such as duration, volume, and playback rate, that you can read or set (some properties are read-only). In addition, there are DOM events that notify you, for example, when a media element is able to play through, begins to play, is paused by the user, or completes.

For a complete list of methods, properties, and events that Safari supports, see *HTMLMediaElement Class Reference*, *HTMLVideoElement Class Reference*, and *HTMLAudioElement Class Reference*.

Some of the ways you can use JavaScript with HTML5 media elements are:

- Create your own interactive audio or video controller—for an example, see "A Simple JavaScript Media Controller and Resizer" (page 24).
- Enter full-screen video mode—for examples, see "Taking Video Full Screen" (page 34) and "Taking Your Custom Controls Full Screen" (page 36).
- Display a progress indicator that shows how much of the media has downloaded—for an example, see "Using DOM Events to Monitor Load Progress" (page 26).
- Load another audio or video when the current one finishes playing—for an example, see "Replacing a Media Source Sequentially" (page 28).
- Test whether Safari can play the specified media type or specified media file—for examples, see "Using JavaScript to Provide Fallback Content" (page 29) and "Handling Playback Failure" (page 31).

Set the Style with CSS3

Relevant Chapter "Adding CSS Styles" (page 40)

Because the `<audio>` and `<video>` elements are standard HTML, you can customize them using CSS—set the background color, control opacity, add a reflection, move the element smoothly across the screen, or even rotate it in 3D. You can combine CSS with JavaScript to change media properties dynamically, in response to user input or movie events.

You can also change the CSS properties of other parts of your webpage in response to media events. For example, you could darken the background and reduce the opacity of the rest of the page—effectively "dimming the lights"—when a movie is playing, or highlight the title of the currently-playing song in a playlist.

For more information, see "Changing Styles in Response to Media Events" (page 41) and "Adding CSS Styles to Video" (page 45).

For code examples, see "Example: Setting Opacity" (page 45), "Adding a Mask" (page 46), "Adding a Reflection" (page 48), and "Rotating Video in 3D" (page 52).

Prerequisites

You should be familiar with HTML and JavaScript. Familiarity with CSS is helpful. To create image masks, you should be able to work with transparency (alpha channels).

See Also

- *Safari DOM Additions Reference* —DOM events, JavaScript functions and properties added to Safari to support HTML5 audio and video, touch events, and CSS transforms and transitions.
- *Safari CSS Visual Effects Guide* —How to use CSS transitions and effects in Safari.
- *Safari CSS Reference* —Complete list of CSS properties, rules, and property functions supported in Safari, with syntax and usage.
- *Safari HTML Reference* —The HTML elements and attributes supported by different Safari and WebKit applications.
- *WebKit DOM Programming Topics* —How to get the most out of using DOM events in Safari.
- *iOS Human Interface Guidelines* —User interface guidelines for designing webpages and web applications for Safari on iOS.

Audio and Video HTML

In their simplest form, the `<audio>` and `<video>` tags require only a `src` attribute to identify the media, although you generally want to set the `controls` attribute as well. Safari allocates space, provides a default controller, loads the media, and plays it when the user clicks the play button. It's all automatic.

There are optional attributes as well, such as `autoplay`, `loop`, `height`, and `width`.

Basic Syntax

The attributes for the `<audio>` and `<video>` tags are summarized in Table 1-1 (page 10). The only difference between the `<audio>` and `<video>` tag attributes is the option to specify a height, width, and poster image for video.

Table 1-1 Attributes of the `<audio>` and `<video>` elements

Attribute	Value	Description
preload *This attribute was formerly named autobuffer, and was boolean.*	`none | metadata | auto`	`none`—Safari should not preload media data. `metadata`—Safari should preload only media metadata. `automatic`—Safari should preload all media data.
autoplay	Boolean—any value sets this to `true`	If present, asks the browser to play the media automatically.
controls	Boolean—any value sets this to `true`	If present, causes the browser to display the default media controls.
height (video only)	*pixels*	The height of the video player.
loop	Boolean—any value sets this to `true`	If present, causes the media to loop indefinitely. This attribute is supported in iOS 5.0 and later.
poster (video only)	*url of image file*	If present, shows the poster image until the first frame of video has downloaded.

Attribute	Value	Description
src	*url*	The URL of the media.
width (video only)	*pixels*	The width of the video player.

Important Several of the attributes are boolean. No value is required. Set the attribute true by including it; set it false by omitting it. Although boolean attributes can be set to `false` using JavaScript, *any* value in the HTML tag sets them to `true`. `Controls="controls"`, for example, is the same as `controls=true` or simply `controls`. Even `controls=false` sets controls to `true` in HTML.

The `preload` attribute is a hint to the browser, telling it your preferences. There is no guarantee that you will get the behavior that you prefer. Safari does not support all preferences on all devices, and does not currently support the Metadata preference on any device. For more information, see "iOS-Specific Considerations" (page 20).

Listing 1-1 shows an HTML page that autoplays a video at a specified height and width with the built-in user controls.

Listing 1-1 Creating a simple movie player

```
<!DOCTYPE html>
<html>
   <head>
      <title>Simple Movie Player</title>
   </head>
   <body>
      <video src="http://Domain.com/path/My.mov"
             controls
             autoplay
             height=270 width=480
      >
      </video>
  </body>
</html>
```

Working with Attributes

There are several ways you can control media playback directly in HTML by setting attributes appropriately.

Resizing the Video

In Safari on iOS, the native size of the video is unknown until the user initiates a download. If no height or width is specified, a default size of 150 x 300 pixels is allocated in the webpage. On iPad, the video currently plays in this default space. On iPhone or iPod touch, the video plays in fullscreen mode once the user initiates it, and the default space is allocated to a placeholder on the webpage.

In Safari on the desktop, the movie metadata is downloaded automatically whenever possible, so the native video size is used as the default. If only the `height` or `width` parameter is specified, the video is scaled up or down to that height or width while maintaining the native aspect ratio of the movie. If both height and width are specified, the video is presented at that size. If neither is specified, the video is displayed at its native size.

Enabling the Controller

In Safari, the default video controller is slightly translucent and is overlaid on the bottom of the video. The controller is not normally visible when the movie is playing—it appears only when the movie is paused, when the user touches the video, or when the mouse pointer hovers over the playing movie. In cases where it is crucial that the bottom of the video never be obscured, omit the `controls` attribute. (You cannot set the attribute to `false` explicitly in HTML—you set it to `false` *implicitly* by leaving the attribute out.)

If you do not set the `controls` attribute, you must either set the `autoplay` attribute, create a controller using JavaScript, or play the movie programmatically from JavaScript. Otherwise the user has no way to play the movie.

Warning To prevent unsolicited downloads over cellular networks at the user's expense, embedded media cannot be played automatically in Safari on iOS—the user always initiates playback. A controller is automatically supplied on iPhone or iPod touch once playback in initiated, but for iPad you must either set the `controls` attribute or provide a controller using JavaScript.

Preloading

If you set the `preload` attribute to `auto`, it tells Safari you want the specified media file to start buffering immediately, making it more likely that it will begin promptly and play through smoothly when the user plays it.

If you have multiple movies on the page, you should set the `preload` attribute to `none`, to prevent all the movies from downloading at once.

Safari does not currently support the `metadata` setting.

> **Note** The `preload` attribute is supported in Safari 5.0 and later. Safari on iOS never preloads.

Showing a Poster

Setting a poster image normally has a transitory effect—the poster image is shown only until the first frame of the video is available, which is typically a second or two. On iOS, however, the first frame is not shown until the user initiates playback, and a poster image is recommended, as shown in Listing 1-2.

Listing 1-2 Showing a poster

```
<!DOCTYPE html>
<html>
   <head>
      <title>Movie Player with Poster</title>
   </head>
   <body>
      <video src="http://Domain.com/path/My.mov"
             controls
             height=340 width=640
             poster="http://Domain.com/path/Poster.jpg"
      >
      </video>
  </body>
</html>
```

Providing Multiple Sources

Not all types of audio and video can play on all devices and browsers. Fortunately, the `<audio>` and `<video>` elements allow you to list as many `<source>` elements as you like. The browser iterates through them until it finds one it can play or runs out of sources. Instead of using the `src` attribute in the media element itself, follow the `<audio>` or `<video>` tag with one or more `<source>` elements, prior to the closing tag.

Specifying Multiple Media Formats

List the alternate media sources in the order of most desirable to least desirable. The browser chooses the first listed source that it thinks it can play. For example, if you have an AAC audio file, an Ogg Vorbis audio file, and a WAVE audio file, listed in that order, Safari plays the AAC file. A browser that cannot play AAC but can play Ogg plays the Ogg file. A browser that can play neither Ogg nor AAC might still be able to play the WAVE file.

Listing 1-3 is a simple example of using multiple sources for an audio file.

Listing 1-3 Specifying multiple audio sources

```
<!DOCTYPE html>
<html>
   <head>
      <title>Multi-Source Audio Player</title>
   </head>
   <body>
      <audio controls autoplay >
            <source src="http://Domain.com/path/MyAudio.m4a">
            <source src="http://Domain.com/path/MyAudio.oga">
            <source src="http://Domain.com/path/MyAudio.wav">
      </audio>
  </body>
</html>
```

The `<source>` element can have a `type` attribute, specifying the MIME type, to help the browser decide whether or not it can play the media without having to load the file.

```
<audio controls=controls>
<source src="mySong.aac" type="audio/mp4">
<source src="mySong.oga" type="audio/ogg">
</audio>
```

The `type` attribute can take an additional `codecs` parameter, to specify exactly which versions of which codecs are needed to play this particular media.

```
<audio controls=controls>
<source src="mySongComplex.aac" type="audio/mp4; codecs=mp4a.40.5">
```

```
<source src="mySongSimple.aac" type="audio/mp4; codecs=mp4a.40.2">
</audio>
```

Notice that you don't have to know which browsers support what formats, and then sniff the user agent string for the browser name to decide what to do. Just list your available media, preferred format first, second choice next, and so on. The browser plays the first one it can.

Currently, most browsers support low-complexity AAC and MP3 audio and basic profile MPEG-4. Most browsers that do not support these formats support Theora video and Vorbis audio using the Ogg file format. Generally speaking, if you provide media in MPEG-4 (basic profile) and Ogg formats, your media can play in browsers that support HTML5.

> **Note** Safari on iOS supports low-complexity AAC audio, MP3 audio, AIF audio, WAVE audio, and baseline profile MPEG-4 video. Safari on the desktop (Mac OS X and Windows) supports all media supported by the installed version of QuickTime, including any installed third-party codecs.

Specifying Multiple Delivery Schemes

You can also use multiple source elements to specify different delivery schemes. Let's say you have a large real-time video streaming service that uses RTSP streaming, and you want to add support for Safari on iOS, including iPad, using HTTP Live Streaming, along with a progressive download for browsers that can't handle either kind of streaming. As shown in Listing 1-4, with HTML5 video it's quite straightforward.

Listing 1-4 Specifying multiple delivery schemes

```
<!DOCTYPE html>
<html>
   <head>
      <title>Multi-Scheme Video Player</title>
   </head>
   <body>
      <video controls autoplay >
            <source src="http://HttpLiveStream.m3u8">
            <source src="rtsp://LegacyStream.3gp">
            <source src="http://ProgressiveDownload.m4v">
      </video>
  </body>
</html>
```

Again, the browser picks the first source it can handle. Safari on the desktop plays the RTSP stream, while Safari on iOS plays the HTTP Live stream. Browsers that support neither m3u8 playlists nor RTSP URLs play the progressive download version.

Multiple Data Rate Sources

HTML5 does not support selection from multiple sources based on data rate. If you supply multiple sources, the browser chooses the first it can play based on scheme, file format, profile, and codecs. Bandwidth is not tested. To provide multiple bandwidths, you must provide a `src` attribute that specifies a source capable of supporting data rate selection itself.

For example, if one of your sources is an HTTP Live Stream, the playlist file can specify multiple streams, and Safari selects the best stream for the current bandwidth dynamically as network bandwidth changes. For more information, see *HTTP Live Streaming Overview*.

Similarly, if the source is a QuickTime reference movie, it can include alternate sources at different data rates, and Safari chooses the best source for the current bandwidth when the video is first requested (though it does not dynamically switch between sources if available bandwidth subsequently changes). For more information, see Technical Note TN2266 "Creating Reference Movies—MakeRefMovie"

Specifying Fallback Behavior

It's easy to specify fallback behavior for browsers that don't support the `<audio>` or `<video>` elements—just put the fallback content between the opening and closing media tags, after any `<source>` elements. See Listing 1-5.

Listing 1-5 Adding simple fallback behavior

```
<!DOCTYPE html>
<html>
   <head>
      <title>Simple Movie Player with Trivial Fallback Behavior</title>
   </head>
   <body>
      <!-- opening tag -->
      <video src="My.mov" controls >

       <!-- fallback content -->
```

```
        Your browser does not support the video element.

      <!-- closing tag -->
      </video>
  </body>
</html>
```

Browsers that don't support the `<audio>` or `<video>` tags simply ignore them. Browsers that support these elements ignore all content between the opening and closing tags (except `<source>` tags).

> **Note** Browsers that understand the `<audio>` and `<video>` tags do not display fallback content, even if they cannot play any of the specified media. To provide fallback content in case no specified media is playable, see "Using JavaScript to Provide Fallback Content" (page 29).

Fall Back to the QuickTime Plug-in

There is a simple way to fall back to the QuickTime plug-in that works for nearly all browsers—download the prebuilt JavaScript file provided by Apple, `AC_QuickTime.js`, from http://developer.apple.com/internet/licensejs.html and include it in your webpage by inserting the following line of code into your HTML head:

```
<script src="AC_QuickTime.js" type="text/javascript">
</script>
```

Once you've included the script, add a call to `QT_WriteOBJECT()` between the opening and closing tags of the `<audio>` or `<video>` element, passing in the URL of the movie, its height and width, and the version of the ActiveX control for Internet Explorer (just leave this parameter blank to use the current version). See Listing 1-6 for an example.

Listing 1-6 Falling back to the QuickTime plug-in

```
<!DOCTYPE html>
<html>
   <head>
      <title>Simple Movie Player with QuickTime Fallback</title>
      <script src="AC_QuickTime.js" type="text/javascript">
      </script>
   </head>
```

```
   <body>
      <video controls >
         <source src="myMovie.mov" type="video/quicktime">
         <source src="myMovie.3gp" type="video/3gpp">
       <!-- fallback -->
       <script type="text/javascript">
        QT_WriteOBJECT('myMovie.mov' , '320', '240', '');
       </script>
      </video>
  </body>
</html>
```

You can pass more than twenty additional parameters to the QuickTime plug-in. For more about how to work with the QuickTime plug-in and the `AC_QuickTime.js` script, see *HTML Scripting Guide for QuickTime*.

Fall Back to Any Plug-In

Since most browsers now support the `<audio>` and `<video>` elements, you can simplify the process of coding for plug-ins by including only the version of the `<object>` tag that works with Internet Explorer as your fallback for HTML5 media.

Listing 1-7 uses HTTP Live Streaming for browsers that support it, MPEG-4 and Ogg Vorbis by progressive download for browsers that support those formats, and falls back to a plug-in for versions of Internet Explorer that don't support HTML5.

Listing 1-7 Falling back to a plug-in for IE

```
<!DOCTYPE html>
<html>
   <head>
      <title>Simple Movie Player with Plug-In Fallback</title>
   </head>
   <body>
       <video controls >
       <source src="HttpLiveStream.m3u8" type="vnd.apple.mpegURL">
       <source src="ProgressiveDowload.mp4" type="video/mp4">
       <source src="OggVorbis.ogv" type="video/ogg">
```

```
        <!-- fallback -->
        <object CLASSID="clsid:02BF25D5-8C17-4B23-BC80-D3488ABDDC6B"
          CODEBASE="http://www.apple.com/qtactivex/qtplugin.cab"
          HEIGHT="320"
          WIDTH="240"
        >
            <PARAM NAME="src" VALUE="ProgressiveDownload.mp4">
        </object>
        </video>
    </body>
</html>
```

Listing 1-7 uses the QuickTime plug-in. To use a different plug-in, change the `CLASSID` and `CODEBASE` parameters to those of your preferred plug-in and provide the `PARAM` tags the plug-in requires.

iOS-Specific Considerations

There are a handful of device-specific considerations you should be aware of when embedding audio and video using HTML5.

Optimization for Small Screens

Currently, Safari optimizes video presentation for the smaller screen on iPhone or iPod touch by playing video using the full screen—video controls appear when the screen is touched, and the video is scaled to fit the screen in portrait or landscape mode. Video is not presented within the webpage. The `height` and `width` attributes affect only the space allotted on the webpage, and the `controls` attribute is ignored. This is true only for Safari on devices with small screens. On Mac OS X, Windows, and iPad, Safari plays video inline, embedded in the webpage.

> **Note** Prior to iOS 4.0, iPhone and iPod touch did not play audio inline. Audio was presented in full-screen mode. Audio plays inline on iOS 4.0 and later, on all devices.

User Control of Downloads Over Cellular Networks

In Safari on iOS (for all devices, including iPad), where the user may be on a cellular network and be charged per data unit, preload and autoplay are disabled. No data is loaded until the user initiates it. This means the JavaScript `play()` and `load()` methods are also inactive until the user initiates playback, unless the `play()` or `load()` method is triggered by user action. In other words, a user-initiated Play button works, but an `onLoad="play()"` event does not.

This plays the movie: `<input type="button" value="Play" onClick="document.myMovie.play()">`

This does nothing on iOS: `<body onLoad="document.myMovie.play()">`

Default Height and Width on iOS

Because the native dimensions of a video are not known until the movie metadata loads, a default height and width of 150 x 300 is allocated on devices running iOS if the height or width is not specified. Currently, the default height and width do not change when the movie loads, so you should specify the preferred height and width for the best user experience on iOS, especially on iPad, where the video plays in the allocated space.

iPhone Video Placeholder

On iPhone and iPod touch, a placeholder with a play button is shown until the user initiates playback, as shown in Figure 2-1. The placeholder is translucent, so the background or any poster image shows through. The placeholder provides a way for the user to play the media. If the iOS device cannot play the specified media, there is a diagonal bar through the control, indicating that it cannot play.

Figure 2-1 The iPhone video placeholder

On the desktop and iPad, the first frame of a video displays as soon as it becomes available. There is no placeholder.

Media Playback Controls

Controls are always supplied during fullscreen playback on iPhone and iPod touch, and the placeholder allows the user to initiate fullscreen playback. On the desktop or iPad, you must either include the `controls` attribute or provide playback controls using JavaScript. It is especially important to provide user controls on iPad because autoplay is disabled to prevent unsolicited cellular download.

Supported Media

Safari on the desktop supports any media the installed version of QuickTime can play. This includes media encoded using codecs QuickTime does not natively support, provided the codecs are installed on the user's computer as QuickTime codec components.

Safari on iOS (including iPad) currently supports uncompressed WAV and AIF audio, MP3 audio, and AAC-LC or HE-AAC audio. HE-AAC is the preferred format.

Safari on iOS (including iPad) currently supports MPEG-4 video (Baseline profile) and QuickTime movies encoded with H.264 video (Baseline profile) and one of the supported audio types.

iPad and iPhone 3G and later support H.264 Baseline profile 3.1. Earlier versions of iPhone support H.264 Baseline profile 3.0.

Multiple Simultaneous Audio or Video Streams

Currently, all devices running iOS are limited to playback of a single audio or video stream at any time. Playing more than one video—side by side, partly overlapping, or completely overlaid—is not currently supported on iOS devices. Playing multiple simultaneous audio streams is also not supported. You can change the audio or video source dynamically, however. See "Replacing a Media Source Sequentially" (page 28) for details.

Volume Control in JavaScript

On the desktop, you can set and read the `volume` property of an `<audio>` or `<video>` element. This allows you to set the element's audio volume relative to the computer's current volume setting. A value of 1 plays sound at the normal level. A value of 0 silences the audio. Values between 0 and 1 attenuate the audio.

This volume adjustment can be useful, because it allows the user to mute a game, for example, while still listening to music on the computer.

On iOS devices, the audio level is always under the user's physical control. The `volume` property is not settable in JavaScript. Reading the `volume` property always returns 1.

Playback Rate in JavaScript

You can set the audio or video `playbackRate` property to nonzero values to play media in slow motion (values >0 and <1) or fast forward (values >1) in Safari on the desktop. Setting `playbackRate` is not currently supported on iOS.

Loop Attribute

You can set the audio or video `loop` attribute in Safari on the desktop and on iOS 5.0 and later to cause media to repeat endlessly. To loop audio or video in a manner compatible with earlier versions of iOS, use JavaScript to install the `play()` method as an event listener for the `"ended"` event. This technique is illustrated in Listing 2-1.

Listing 2-1 Backward-compatible looping audio

```
<!DOCTYPE html>
<html>
<head>
    <title>Looping Audio</title>
<script type="text/javascript">

    function init() {
    var myAudio = document.getElementById("audio");
    myAudio.addEventListener('ended', loopAudio, false);
    }

    function loopAudio() {
    var myAudio = document.getElementById("audio");
    myAudio.play();
    }

</script>
</head>
<body onload="init();">
    <audio id="audio" src="myAudio.m4a" controls></audio>
</body>
</html>
```

Controlling Media with JavaScript

Because the `<audio>` and `<video>` elements are part of the HTML5 standard, there are standard JavaScript methods, properties, and DOM events associated with them.

There are methods for loading, playing, pausing, and jumping to a time, for example. There are also properties you can set programmatically, such as the `src` URL and the height and width of a video, as well as read-only properties such as the video's native height. Finally, there are DOM events you can listen for, such as load progress, media playing, media paused, and media done playing.

This chapter shows you how to do the following:

- Use JavaScript to create a simple controller.
- Change the size of a movie dynamically.
- Display a progress indicator while the media is loading.
- Replace one movie with another when the first finishes.
- Provide fallback content using JavaScript if none of the media sources are playable.
- Enter and exit fullscreen mode.
- Take your custom video player and controls into fullscreen mode.

For a complete description of all the methods, properties, and DOM events associated with HTML5 media, see *Safari DOM Additions Reference*. All the methods, properties, and DOM events associated with `HTMLMediaElement`, `HTMLAudioElement`, and `HTMLVideoElement` are exposed to JavaScript.

A Simple JavaScript Media Controller and Resizer

The example in Listing 3-1creates a simple play/pause movie control in JavaScript, with additional controls to toggle the video size between normal and doubled. It is intended to illustrate, in the simplest possible way, addressing a media element, reading and setting properties, and calling methods.

Any of the standard ways to address an HTML element in JavaScript can be used with `<audio>` or `<video>` elements. You can assign the element a name or an id, use the tag name, or use the element's place in the DOM hierarchy. The example in Listing 3-1 uses the tag name. Since there is only one `<video>` element in the example, it is the 0th item in the array of elements with the "video" tag name.

Listing 3-1 Adding simple JavaScript controls

```
<!DOCTYPE html>
<html>
  <head>
    <title>Simple JavaScript Controller</title>
    <script type="text/javascript">

       function playPause() {
       var myVideo = document.getElementsByTagName('video')[0];
       if (myVideo.paused)
           myVideo.play();
       else
           myVideo.pause();
       }

       function makeBig() {
       var myVideo = document.getElementsByTagName('video')[0];
       myVideo.height = (myVideo.videoHeight * 2 ) ;
       }

       function makeNormal() {
       var myVideo = document.getElementsByTagName('video')[0];
       myVideo.height = (myVideo.videoHeight) ;
       }

    </script>
  </head>

  <body>
     <div class="video-player" align="center">
      <video src="myMovie.m4v" poster="poster.jpg" ></video>
      <br>
      <a href="javascript:playPause();">Play/Pause</a> <br>
      <a href="javascript:makeBig();">2x Size</a> |
      <a href="javascript:makeNormal();">1x Size</a> <br>
```

```
    </div>
  </body>
</html>
```

The previous example gets two read-only properties: `paused` and `videoHeight` (the native height of the video). It calls two methods: `play()` and `pause()`. And it sets one read/write property: `height`. Recall that setting only the height or width causes the video to scale up or down while retaining its native aspect ratio.

> **Note** Safari on iOS version 3.2 does not support dynamically resizing video on the iPad.

Using DOM Events to Monitor Load Progress

One of the common tasks for a movie controller is to display a progress indicator showing how much of the movie has loaded so far. One way to do this is to constantly poll the media element's `buffered` property, to see how much of the movie has buffered, but this is a waste of time and energy. It wastes processor time and often battery charge, and it slows the loading process.

A much better approach is to create an event listener that is notified when the media element has something to report. Once the movie has begun to load, you can listen for `progress` events. You can read the `buffered` value when the browser reports `progress` and display it as a percentage of the movie's `duration`.

Another useful DOM event is `canplaythrough`, a logical point to begin playing programmatically.

You can install event listeners on the media element or any of its parents, up to and including the document body.

Listing 3-2 loads a large movie from a remote server so you can see the progress changing. It installs an event listener for `progress` events and another for the `canplaythrough` event. It indicates the percentage loaded by changing the inner HTML of a paragraph element.

You can copy and paste the example into a text editor and save it as HTML to see it in action.

Listing 3-2 Installing DOM event listeners

```
<!DOCTYPE html>
<html>
<head>
    <title>JavaScript Progress Monitor</title>
```

```
    <script type="text/javascript">

       function getPercentProg() {
       var myVideo = document.getElementsByTagName('video')[0];
       var endBuf = myVideo.buffered.end(0);
       var soFar = parseInt(((endBuf / myVideo.duration) * 100));
       document.getElementById("loadStatus").innerHTML =  soFar + '%';
       }

       function myAutoPlay() {
       var myVideo = document.getElementsByTagName('video')[0];
       myVideo.play();
       }

       function addMyListeners(){
       var myVideo = document.getElementsByTagName('video')[0];
       myVideo.addEventListener('progress',getPercentProg,false);
       myVideo.addEventListener('canplaythrough',myAutoPlay,false);
       }

    </script>
</head>

  <body onLoad="addMyListeners()">
     <div align=center>
        <video controls
        src="http://homepage.mac.com/qt4web/sunrisemeditations/myMovie.m4v" >
        </video>
        <p ID="loadStatus">
        MOVIE LOADING...
        </p>
      </div>
  </body>
</html>
```

> **Note** On the iPad, Safari does not begin downloading until the user clicks the poster or placeholder. Currently, downloads begun in this manner do not emit `progress` events.

The `buffered` property is a `TimeRanges` object, essentially an array of start and stop times, not a single value. Consider what happens if the person watching the media uses the time scrubber to jump forward to a point in the movie that hasn't loaded yet—the movie stops loading and jumps forward to the new point in time, then starts buffering again from there. So the `buffered` property can contain an array of discontinuous ranges. The example simply seeks to the end of the array and reads the last value, so it actually shows the percentage into the movie duration for which there is data. To determine precisely what percentage of a movie has loaded, taking possible discontinuities into account, iterate through the array, summing the seekable ranges, as illustrated in Listing 3-3

Listing 3-3 Summing a TimeRanges object

```
var myBuffered = document.getElementsByTagName('video')[0].buffered;
var total = 0;
for (ndx = 0; ndx < myBuffered.length; ndx++)
    total += (seekable.end(ndx) – seekable.start(ndx));
```

The `buffered`, `played`, and `seekable` properties are all `TimeRanges` objects.

Replacing a Media Source Sequentially

Another common task for a website programmer is to create a playlist of audio or video—to put together a radio set or to follow an advertisement with a program, for example. To do this, you can provide a function that listens for the `ended` event. The `ended` event is triggered only when the media ends (plays to its complete duration), not if it is paused.

When your listener function is triggered, it should change the media's `src` property, then call the `load` method to load the new media and the `play` method to play it, as shown in Listing 3-4.

Listing 3-4 Replacing media sequentially

```
<!DOCTYPE html>
<html>
<head>
    <title>Sequential Movies</title>
```

```
    <script type="text/javascript">

        // listener function changes src
        function myNewSrc() {
        var myVideo = document.getElementsByTagName('video')[0];
        myVideo.src="http://homepage.mac.com/qt4web/myMovie.m4v";
        myVideo.load();
        myVideo.play();
        }
        // add listener function to ended event -->
        function myAddListener(){
        var myVideo = document.getElementsByTagName('video')[0];
        myVideo.addEventListener('ended',myNewSrc,false);
        }
    </script>
</head>
<body onload="myAddListener()">
    <video controls
    src="http://homepage.mac.com/qt4web/A-chord.m4v">
    </video>
</body>
</html>
```

The previous example works on both Safari for the desktop and Safari for iOS, as the `load()` and `play()` methods are enabled even on cellular networks once the user has started playing the first media element.

Using JavaScript to Provide Fallback Content

It's easy to provide fallback content for browsers that don't support the `<audio>` or `<video>` tag using HTML (see "Specifying Fallback Behavior" (page 16)). But if the browser understands the tag and can't play any of the media you've specified, you need JavaScript to detect this and provide fallback content.

To test whether the browser can play any of the specified media, iterate through your source types using the `canPlayType` method.

Important The HTML5 specification has changed. Browsers conforming to the earlier version of the specification, including Safari 4.0.4 and earlier, return "no" if they cannot play the media type. Browsers conforming to the newer version return an empty string (“”)instead. You need to check for either response, or else check for a positive response rather than a negative one.

If the method returns “no” or the empty string (“”) for all the source types, the browser knows it can’t play any of the media, and you need to supply fallback content. If the method returns “maybe” or “probably” for any of the types, it will attempt to play the media and no fallback should be needed.

The following example creates an array of types, one for each source, and iterates through them to see if the browser thinks it can play any of them. If it exhausts the array without a positive response, none of the media types are supported, and it replaces the video element using `innerHTML`. Listing 3-5 displays a text message as fallback content. You could fall back to a plug-in or redirect to another page instead.

Listing 3-5 Testing for playability using JavaScript

```
<!DOCTYPE html>
<html>
 <head>
   <title>JavaScript Fallback</title>
   <script type="text/javascript">

     function checkPlaylist() {
       var playAny = 0;
       myTypes = new Array ("video/mp4","video/ogg","video/divx");
       var nonePlayable = "Your browser cannot play these movie types."
       var myVideo = document.getElementsByTagName('video')[0];
       for (x = 0; x < myTypes.length; x++)
       { var canPlay = myVideo.canPlayType(myTypes[x]);
         if ((canPlay=="maybe") || (canPlay=="probably"))
            playAny = 1;
       }
       if (playAny==0)
         document.getElementById("video-player").innerHTML = nonePlayable;
     }

   </script>
```

```
 </head>
 <body onload="checkPlaylist()" >
   <div id="video-player" align=center>
     <video controls height="200" width="400">
     <source src="myMovie.m4v" type="video/mp4">
     <source src="myMovie.oga" type="video/ogg">
     <source src="myMovie.dvx" type="video/divx">
     </video>
   </div>
  </body>
</html>
```

Handling Playback Failure

Even if a source type is playable, that's no guarantee that the source *file* is playable—the file may be missing, corrupted, misspelled, or the `type` attribute supplied may be incorrect. If Safari 4.0.4 or earlier attempts to play a source and cannot, it emits an `error` event. However, it still continues to iterate through the playable sources, so the `error` event may indicate only a momentary setback, not a complete failure. It's important to check which source has failed to play.

Changes in the HTML5 specification now require the media element to emit an error only if the *last* playable source fails, so this test is not necessary in Safari 5.0 or later.

The example in Listing 3-5 iterates through the source types to see if any are playable. It saves the filename of the last playable source. If there are no playable types, it triggers a fallback. If there are playable types, it installs an `error` event listener. The event listener checks to see if the current source contains the last playable filename before triggering a failure fallback. (The `currentSrc` property includes the full path, so the test is for inclusion, not equality.)

Notice that when adding a listener for the `error` event you need to set the `capture` property to `true`, whereas for most events you set it to `false`.

Listing 3-6 Testing for failure using JavaScript

```
<!DOCTYPE html>
<html>
<head>
    <title>JavaScript Fallback</title>
```

```
<script type="text/javascript">

var lastPlayable
myTypes = new Array ("video/mp4","video/ogg","video/divx");
mySrc = new Array ("myMovie.mp4","myMovie.oga","myMovie.dvx");

function errorFallback() {
var errorLast = "An error occurred playing " ;
var myVideo = document.getElementsByTagName('video')[0];
if (myVideo.currentSrc.match(lastPlayable))
    { errorLast = errorLast + lastPlayable ;
    document.getElementById("video-player").innerHTML = errorLast;
    }
 }

 function checkPlaylist() {
 var noPlayableTypes = "Your browser cannot play these movie types";
 var myVideo = document.getElementsByTagName('video')[0];
 var playAny = 0;
 for (x = 0; x < myTypes.length; x++)
     { var canPlay = myVideo.canPlayType(myTypes[x]);
     if ((canPlay=="maybe") || (canPlay=="probably"))
         { playAny = 1;
         lastPlayable=mySrc[x];
         }
     }
 if (playAny==0)
     {
     document.getElementById("video-player").innerHTML = noPlayableTypes;
     } else {
     myVideo.addEventListener('error',errorFallback,true);
     }
  }
```

```
      </script>
</head>
<body onload="checkPlaylist()" >
    <video controls >
    <source src="myMovie.mp4" type="video/mp4">
    <source src="myMovie.oga" type="video/ogg">
    <source src="myMovie.dvx" type="video/divx
    </video>
</body>
</html>
```

Resizing Movies to Native Size

If you know the dimensions of your movie in advance, you should specify them. Specifying the dimensions is especially important for delivering the best user experience on iPad. But you may not know the dimensions when writing the webpage. For example, your source movies may not be the same size, or sequential movies may have different dimensions. If you install a listener function for the `loadedmetadata` event, you can resize the video player to the native movie size dynamically using JavaScript as soon as the native size is known. The `loadedmetadata` event fires once for each movie that loads, so a listener function is called any time you change the source. Listing 3-7 shows how.

Listing 3-7 Resizing movies programmatically

```
<!DOCTYPE html>
<html>
<head>
    <title>Resizing Movies</title>
    <script type="text/javascript">

     // set height and width to native values
     function naturalSize() {
     var myVideo = document.getElementsByTagName('video')[0];
     myVideo.height = myVideo.videoHeight;
     myVideo.width = myVideo.videoWidth;
     }
     // install listener function on metadata load
```

```
    function myAddListener(){
    var myVideo = document.getElementsByTagName('video')[0];
    myVideo.addEventListener('loadedmetadata',naturalSize,false);
    }
    </script>
</head>
<body onload="myAddListener()" >
    <video src="http://homepage.mac.com/qt4web/myMovie.m4v" controls >
    </video>
</body>
</html>
```

Taking Video Full Screen

Safari 5.0 and later, and iOS 3.2 and later on iPad, include a fullscreen button on the video controller, allowing the user to initiate full-screen video mode.

Safari 5.0 and iOS 4.2 and later add JavaScript properties and DOM events that your scripts can use to determine when the browser has entered or exited full-screen video mode, as well as the methods to enter and exit full-screen video mode programatically. See *HTMLMediaElement Class Reference* for a full description of the full-screen DOM events, properties, and methods.

The following example, Listing 3-8, adds a button that puts Safari into full-screen video mode. The Boolean property `webkitSupportsFullscreen` is tested to verify that the current media is capable of being played in full-screen mode. Audio-only files cannot be played in full-screen mode, for example. The Full-screen button is hidden until the test is performed.

Note The `webkitSupportsFullscreen` property is not valid until the movie metadata has loaded. You can detect when the metada is loaded by installing an event listener for the `loadedmetadata` event.

Important The `webkitEnterFullscreen()` method can be invoked only in response to a user action, such as clicking a button. You cannot invoke `webkitEnterFullscreen()` in response to an `onload()` event, for example.

Listing 3-8 Using webkitEnterFullscreen()

```
<html>
<head>
  <title>Fullscreen Video</title>

<script type="text/javascript">
var vid;

function init() {
    vid = document.getElementById("myVideo");
    vid.addEventListener("loadedmetadata", addFullscreenButton, false);
}

function addFullscreenButton(){
    if (vid.webkitSupportsFullscreen) {
        var fs=document.getElementById("fs");
        fs.style.visibility="visible";
    }
}

function goFullscreen() {
        vid.webkitEnterFullscreen();
}
</script>

</head>
```

```
<body onload="init()"


    <h1>Fullscreen Video</h1>


    <video src="myMovie.m4v" id="myVideo" autoplay controls>
    </video>
     <BR>
     <input type="button" id="fs" value="Fullscreen" onclick="goFullscreen()"
style="visibility:hidden">


</body>
</html>
```

Taking Your Custom Controls Full Screen

In Safari 5.1 and later for Mac OS X and Windows, you can not only take your video into full-screen mode, you can take *any* HTML element into full-screen mode. If you enclose a video and custom controls inside a `div` element, for example, you can take the `div` element and all its contents into full-screen mode by calling `myDiv.webkitRequestFullscreen()`.

Use the following functions to take any element into and out of full-screen mode:

- `theElement.webkitRequestFullScreen()`
- `theElement.webkitCancelFullScreen()`

When you enter full-screen mode programatically, it is important to remember that the user can exit full-screen mode at any time by pressing the Esc key.

Important The `webkitRequestFullScreen()` method can be invoked *only* in response to a user action, such as clicking a button. You cannot invoke `webkitRequestFullScreen()` in response an `onload()` event, for example.

Full-Screen Event and Properties

Safari emits a `webkitfullscreenchange` event when an element enters or exits full-screen mode. Listen for this event to detect changes.

You can test the `document.webkitIsFullScreen` property to see whether Safari has most recently entered or exited full-screen mode. The property is `true` when in full-screen mode.

The `document.webkitCurrentFullScreenElement` property contains the element that is in full-screen mode. Use this property if you have more than one element that could be in full-screen mode.

Resizing Enclosed Video

When a `video` element alone is taken into full-screen mode, the video is automatically scaled to fill the screen. When other elements are taken full screen, however, they are not necessarily resized. Instead, normal HTML rules are followed, so a `div` element widens to fill the screen, but an enclosed video retains its height and width. If your video is inside an element that you take full screen, you are responsible for resizing the video when Safari enters and exits full-screen mode.

An easy way to resize video automatically is to define a full-screen pseudo style in CSS for the element enclosing the video. For example, if the ID of the enclosing `div` is “video-player” this CSS snippet expands the enclosed video when the `div` element is in full-screen mode:

```
#video-player:-webkit-full-screen video {
    width: 100%;
}
```

A key advantage to using CSS is that it expands the video when its parent is in full-screen mode, then returns the video to its normal size when its parent leaves full-screen mode.

It can be tricky to expand a video to use the full screen while preserving its aspect ratio. Here are some guidelines:

- If your video aspect ratio is 16 x 9, setting the width to 100% usually works best without setting the height explicitly—your video is scaled to the correct width, and the height is scaled to preserve the aspect ratio. Most displays are 4 x 3, 16 x 9, or slightly taller, so there is normally enough display height to prevent clipping.
- If your video aspect ratio is 4 x 3, setting the width to 75% gives the maximum image size for screens with 16 x 9 aspect ratios, while still using most of a 4 x 3 display. (Setting the width to 100% clips off the top and bottom of the image on widescreen displays.) Alternatively, you can use JavaScript to read the screen height and width, then set the width to 100% on 4 x 3 displays and 75% on wider displays.
- If your video is taller than it is wide, setting the height to 100% and leaving the width unset gives you the maximum image size on any landscape display.
- If your controls appear under the video, instead of floating on top of the video, reduce the width or height setting by 10% or so to leave room for the controls.

Full-Screen Video with Custom Controls Example

The example in Listing 3-9 creates a `div` element enclosing a video and a simple Play/Pause control. Beneath the `div` element is a Full-screen control. When the Full-screen control is clicked, the example takes the `div` element enclosing the video and Play/Pause control into full-screen mode.

CSS styles are used to expand the `div` element itself to 100% of the screen width and height, and to expand the enclosed video element to 100% of the `div` element's width. Only the video's width is specified, so that the video scales up while retaining its native aspect ratio. The example also gives the `div` element a black background when in fullscreen mode.

More elaborate CSS could be used to hide the controls while in fullscreen mode and reveal them when the user touches or hovers over the video. For more about styling video controllers using CSS, see "Adding CSS Styles" (page 40).

Listing 3-9 Full-screen video with custom controls

```
<!DOCTYPE html>
<html>
<head>
    <title>Fullscreen JavaScript Controller</title>
<style>
    #video-player:-webkit-full-screen {
        width: 100%;
        height: 100%;
        background-color: black;
    }

    #video-player:-webkit-full-screen video {
        width: 100%;
    }
</style>

<script type="text/javascript">
    function playPause() {
        var myVideo = document.getElementsByTagName('video')[0];
        if (myVideo.paused)
            myVideo.play();
```

```
        else
            myVideo.pause();
     }

    function goFullscreen() {
        var myPlayer = document.getElementById('video-player');
        myPlayer.webkitRequestFullScreen();
    }

</script>
</head>

<body >
    <div align="center">
        <div id="video-player">
            <video src="myMovie.m4v"></video>
            <p> <a href="javascript:playPause();">Play/Pause</a> </p>
        </div>
        <a href="javascript:goFullscreen();">Full-screen</a>
      </div>
</body>
</html>
```

Adding CSS Styles

Because the `<video>` element is standard HTML, you can modify its appearance and behavior using CSS styles. You can modify the opacity, add a border, position the element dynamically on the page, and much more. You can also use CSS styles to enhance elements that interact with audio or video, such as custom controllers and progress bars. In addition, you can listen for media events in JavaScript and change the properties of other website elements in response.

> **Note** You may want to create your own stylish movie controller—including a progress bar and a time scrubber—that slides smoothly out of the way when not in use. To see an example of just that, download the *ConcertDemo* sample code from `http://developer.apple.com/safari/library/samplecode/HTML5VideoPlayer/index.html`. The sample code contains complete HTML, JavaScript, CSS, and video.

This chapter illustrates some methods of adding CSS styles to video, as well as how to modify the style of non-video elements in response to media events. For more information on using CSS styles in Safari, see *Safari CSS Visual Effects Guide*, *Safari Graphics, Media, and Visual Effects Coding How-To's*, and *Safari CSS Reference*.

Changing Styles in Response to Media Events

You can use JavaScript to modify the style of website elements in response to media events. For example, you could update a music playlist and highlight the name of the song currently being played. The example in Listing 4-1 (page 42) changes the webpage styles when a movie is playing—darkening the background and fading the rest of the webpage by making it partly transparent—then restores things when the movie is paused or completes. Figures III-1 and III-2 show the webpage with the movie playing and paused.

Figure 4-1 Page with movie paused

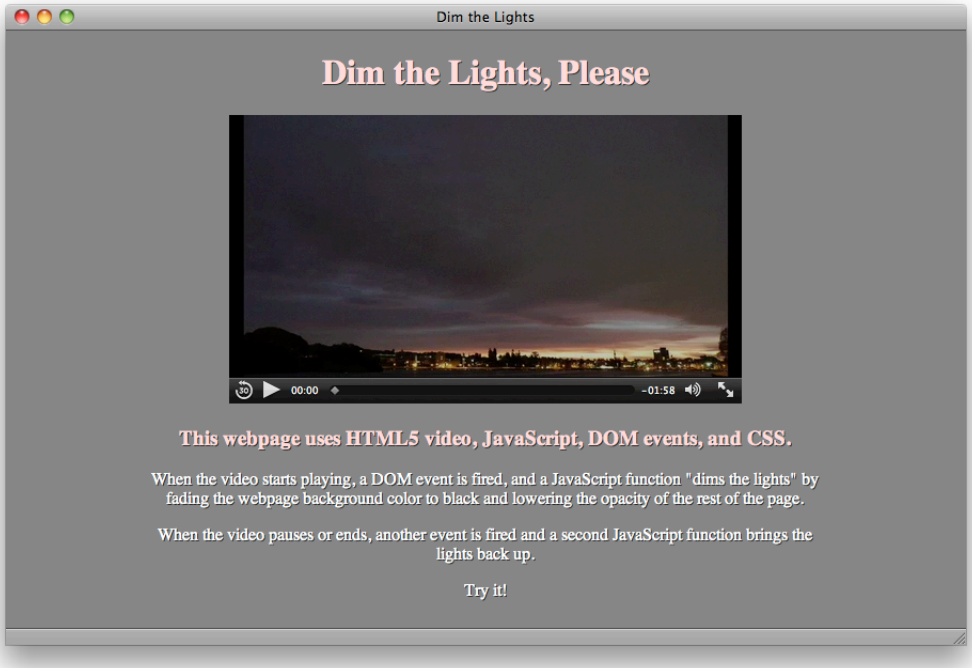
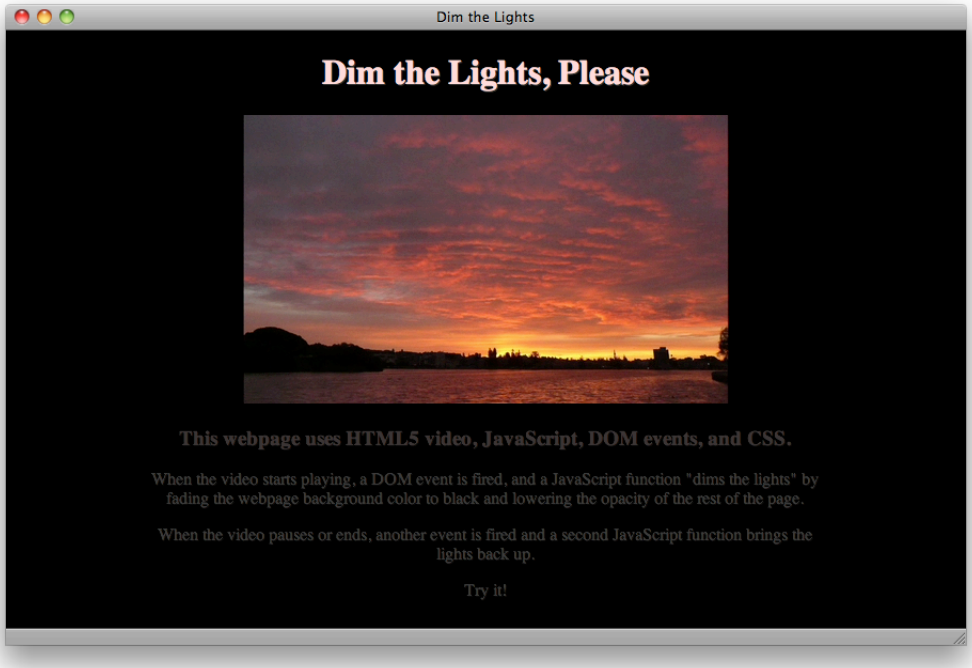


Figure 4-2 Page with movie playing



The example listens for the `playing`, `pause`, and `ended` events. When the `playing` event fires, a function stores the current background color, sets the background color to black, and sets the opacity of the rest of the page to 0.25. When the `pause` or `ended` event fires, a second function restores the background color and sets the opacity to 1.

Note that the `<style>` section of the page sets the `webkit-transition-property` and `webkit-transition-duration` for the background color and opacity, so these properties change smoothly over a few seconds instead of changing abruptly. The background color and opacity change work in any HTML5-compliant browser, but the gradual state change happens only in Safari and other WebKit-based browsers.

Alternatively, you could set a JavaScript timer and darken the background and reduce the page opacity incrementally over several steps.

Listing 4-1 Dim the lights

```
<!DOCTYPE html>
<html>
<head>
    <title>Dim the Lights</title>
```

```
<script type="text/javascript">

var restoreColor;

function dimLights() {
restoreColor = document.body.style.backgroundColor ;
var theBody = document.body ;
theBody.style.backgroundColor = "black" ;
var theRest = document.getElementById("restofpage") ;
theRest.style.opacity = 0.25 ;
}

function lightsUp() {
var theBody = document.body ;
theBody.style.backgroundColor = restoreColor ;
var theRest = document.getElementById("restofpage") ;
theRest.style.opacity = 1 ;
}

function addListener() {
var myVideo = document.getElementsByTagName('video')[0];
myVideo.addEventListener('playing',dimLights,false);
myVideo.addEventListener('pause',lightsUp,false);
myVideo.addEventListener('ended',lightsUp,false);
}

</script>

<style>
body { background-color:#888888;
color:#ffdddd;
text-shadow: 0.1em 0.1em #333;
-webkit-transition-property: background-color;
-webkit-transition-duration: 4s;
}
```

```
    #restofpage { color:#ffffff;
    width: 640px;
    -webkit-transition-property: opacity;
    -webkit-transition-duration: 4s;
    }
    </style>

</head>
<body onload = "addListener()">

    <div align="center">
        <h1>Dim the Lights, Please</h1>
        <video id="player" controls
        <src="http://homepage.mac.com/qt4web/myMovie.m4v">
        </video>
        <div id="restofpage">
            <h3 style="color:#ffdddd;" >
            This webpage uses HTML5 video, JavaScript, DOM events, and CSS.
            </h3>
            <p>
             When the video starts playing, a DOM event is fired, and a JavaScript
            function "dims the lights" by fading the webpage background color
            to black and lowering the opacity of the rest of the page.
            </p>
            <p>
            When the video pauses or ends, another event is fired and a second
            JavaScript function brings the lights back up.
            </p>
            <p>
            Try it!
            </p>
        </div>
    </div>
```

```
</body>
</html>
```

Adding CSS Styles to Video

You can use CSS to apply styles to the video element itself. For example, you can change the video opacity, position the video on top of another element, or add a border.

Example: Setting Opacity

To allow elements under the video to show through, set the opacity of the video. Opacity has a range from 0 (completely transparent) to 1 (completely opaque). You can set opacity directly in CSS—applying it to video generally, or to a class of video, or to a particular video element:

Declaring the opacity of the `video` element: `video { opacity: 0.5 }`

Declaring the opacity of a class: `.seeThroughVideo { opacity: 0.3 }`

Declaring the opacity of an element using its ID: `#theVideo { opacity: 0.7 }`

To change video opacity dynamically, use JavaScript. You can address the video element by ID or tag name, and you can set the opacity property either directly or by changing the element's class name to a class with a style applied to it.

The following code snippet declares the opacity of a class. The snippet then defines a JavaScript function that sets a video element's classname to the declared class. A second JavaScript function illustrates setting the opacity of a video element directly, instead of changing the classname.

```
<style>
    .seeThroughVideo { opacity: 0.3 }
</style>

<script type = "text/javascript">
    function makeSeeThrough() {
    document.getElementById("theVideo").className = "seeThroughVideo" ;
    }

    function setOpacityDirectly (val) {
    document.getElementsByTagName("video")[0].style.opacity = val ;
```

```
    }
```

Using WebKit Properties

In addition to the standard CSS properties, you can apply WebKit properties to video in Safari or other WebKit-based browsers. WebKit properties are prefixed with `–webkit–`. Many CSS properties begin as WebKit properties, are proposed as CSS standards, and go through review before becoming standardized. When and if a WebKit property becomes a standard property, `–webkit–` is dropped from the name. The `–webkit–` prefixed version is maintained for backward compatibility, however.

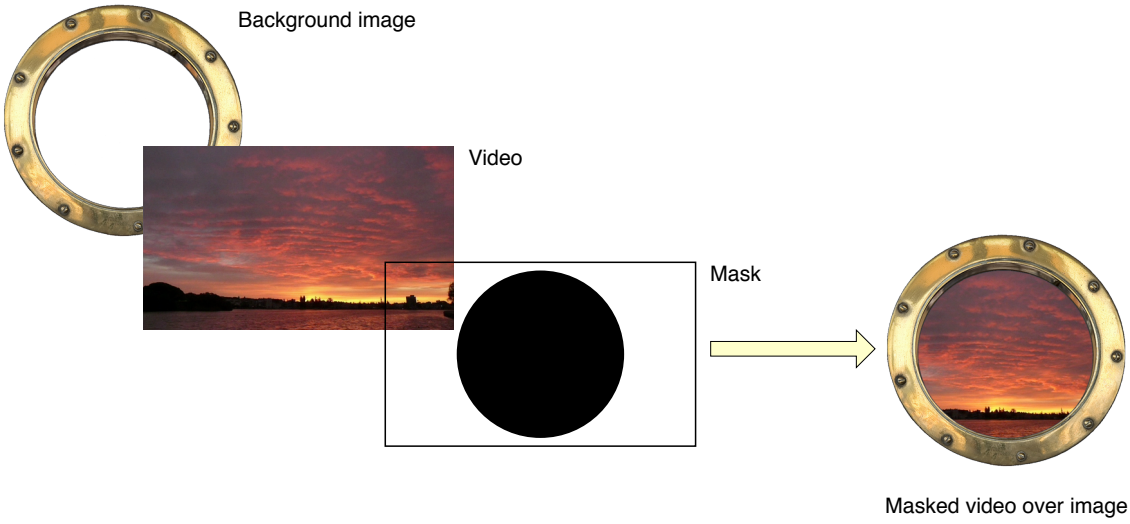
You can safely add WebKit properties to your webpages. WebKit-based browsers, such as Safari and Chrome, recognize WebKit properties. Other browsers just ignore them. Use WebKit properties to enhance general websites, but verify that your site still looks good using other browsers, unless your site is Safari-specific or designed solely for iOS devices.

Three noteworthy WebKit properties are masks, reflections, and 3D rotation.

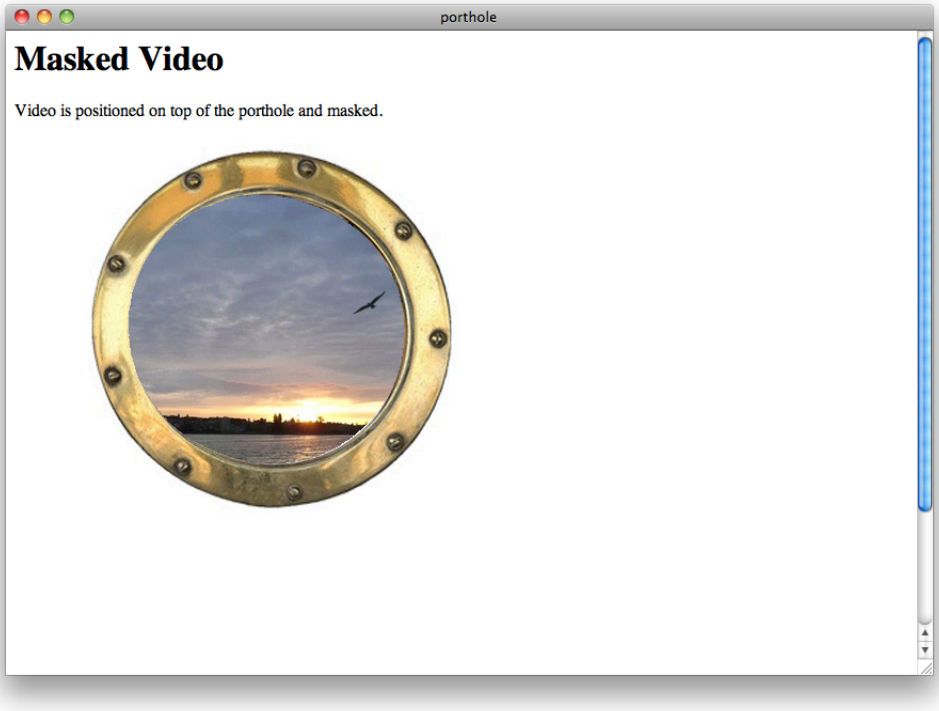
Adding a Mask

You can make a video element appear non-rectangular or discontinuous by masking part of the video out using `–webkit–mask–box–image`. Specify an image to use as a mask. The mask image should have the same dimensions as the video, be opaque where you want the video to show, and be transparent where you want the video to be hidden. Making areas of the mask semi-opaque makes the corresponding areas of the video proportionally semi-transparent.

The following example uses CSS to position a video over the image of a porthole, then masks the video with a circular image so the movie appears to be seen through the porthole.



```
<!DOCTYPE html>
<html>
<head>
    <title>porthole</title>
</head>
<body>
    <h1>Masked Video</h1>
    <p>
    Video is positioned on top of the porthole and masked.
    </p>
    <img src="porthole.jpg" width="430" height="365">

    <video src = "myMovie.m4v" autoplay
    style = "position:relative ; top: -325px ;
    -webkit-mask-box-image: url(portholemask.png) ;">
    </video>
</body>
</html>
```



Note If your mask conceals the built-in controls, they cannot be used to play the video—be sure to provide a JavaScript controller or to set the `autoplay` attribute, bearing in mind that `autoplay` is disabled on cellular-capable devices such as iPhone and iPad.

Adding a Reflection

Add a reflection of a video using the `-webkit-box-reflect` property. Specify whether the reflection should appear on the left, right, above, or below the video, and specify any offset space between the video and the reflection. The following snippet adds a reflection immediately below the video.

```
<video src = "myMovie.m4v" autoplay style = "-webkit-box-reflect: below 0px;" >
```

If you set the `controls` attribute, the video controls will also appear mirrored in the reflection. Since the mirrored controls are backwards and nonfunctional, this is not usually what you want. Video reflections are best used with a JavaScript controller.

You can mask out part of the reflection by including the url of a mask image. The following example adds a masked reflection below the video.

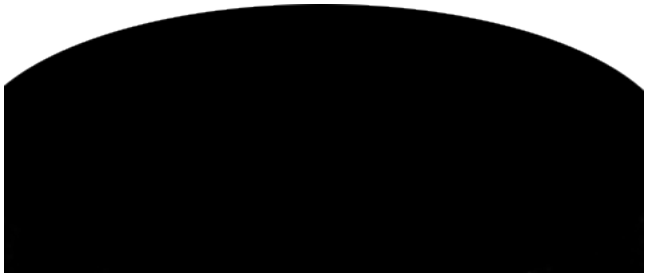
```
<html>
<head>
    <title>Reflection with Mask</title>
    <script type="text/javascript">

    function playPause() {
    var myVideo = document.getElementById('theVideo');
    if (myVideo.paused)
        myVideo.play();
    else myVideo.pause();
    }
    </script>
</head>
<body align="center" style = "background-color: 404040">
    <h1 style="color:#ffeeee">
    Reflection with Mask
    </h1>
    <input type="button" value="Play/Pause" onclick="playPause()">
    <BR><BR>
    <video  src = "myMovie.m4v" id = "theVideo"
    style = "-webkit-box-reflect: below 0px url(mask.png);">
    </video>
</body>
```

```
</html>
```

Figure 4-3 Reflection with mask image

Important The mask image is reflected, so flip it 180° on the axis of reflection from the way you want it to appear.

Figure 4-4 Mask image



You can specify a gradient instead of an image to mask the reflection. The following snippet creates a gradient that fades to transparency.

```
<video style = "-webkit-box-reflect: below 0px

-webkit-gradient(linear, left top, left bottom, from(transparent), to(black));" >
```

Because the gradient is reflected, it is specified from transparent to black—its reflection goes from black to transparent. Because the gradient is a mask, only its opacity matters. It could also be specified from transparent to white.

The following snippet adds a color-stop to the gradient, causing it to reach transparency at four-tenths of its length.

```
<video style = "-webkit-box-reflect: below 0px

-webkit-gradient(linear, left top, left bottom, from(transparent), to(black),
```

```
color-stop(0.4, transparent));">
```

Figure 4-5 Reflection with gradient mask and color stop

Rotating Video in 3D

You can use WebKit CSS properties to rotate video around the x or y axis. Rotating the video 180° allows the user to see it playing from "behind". Rotating the video 90° causes it to play edge-on, making it invisible and revealing whatever is behind it. Rotation can be set statically using CSS or dynamically using JavaScript. The following snippet sets the video rotation at 45° about the y axis.

```
<video style="-webkit-transform: rotateY(45deg);" src="myVideo.m4v" controls>
```

> **Note** When a video element is rotated between 90° and 270°, the controls are either reversed or edge-on, and are not operable.

Dynamic rotation is best used in conjunction with the `webkit-transition` property. Setting `webkit-transition` causes the video element to rotate smoothly over a specified duration, without having to set JavaScript timers or stepper functions. You set the transition property, specify the duration, and any time you change the specified property, the change is animated smoothly over the specified duration. The property to animate for rotation is `-webkit-transform`. When the animation completes, a `webkitTransitionEnd` event is fired.

The following snippet sets the transition property to `-webkit-transform` and sets the animation duration to 4 seconds.

```
<video src="myVideo.m4v" controls
style="-webkit-transition-property: -webkit-transform;
-webkit-transition-duration: 4s">
```

You can add perspective to the rotation, so that parts of the video element closer to the viewer along the z axis appear larger, and parts further away appear smaller. To add perspective, set the `-webkit-perspective` property of one of the video element's parents, such as a parent `div` or the document body.

Listing 4-2 dynamically flips a playing video 180° around its y axis, intermittently revealing the text behind the video. The video has perspective when it rotates, due to the `-webkit-perspective` property of the `body` element.

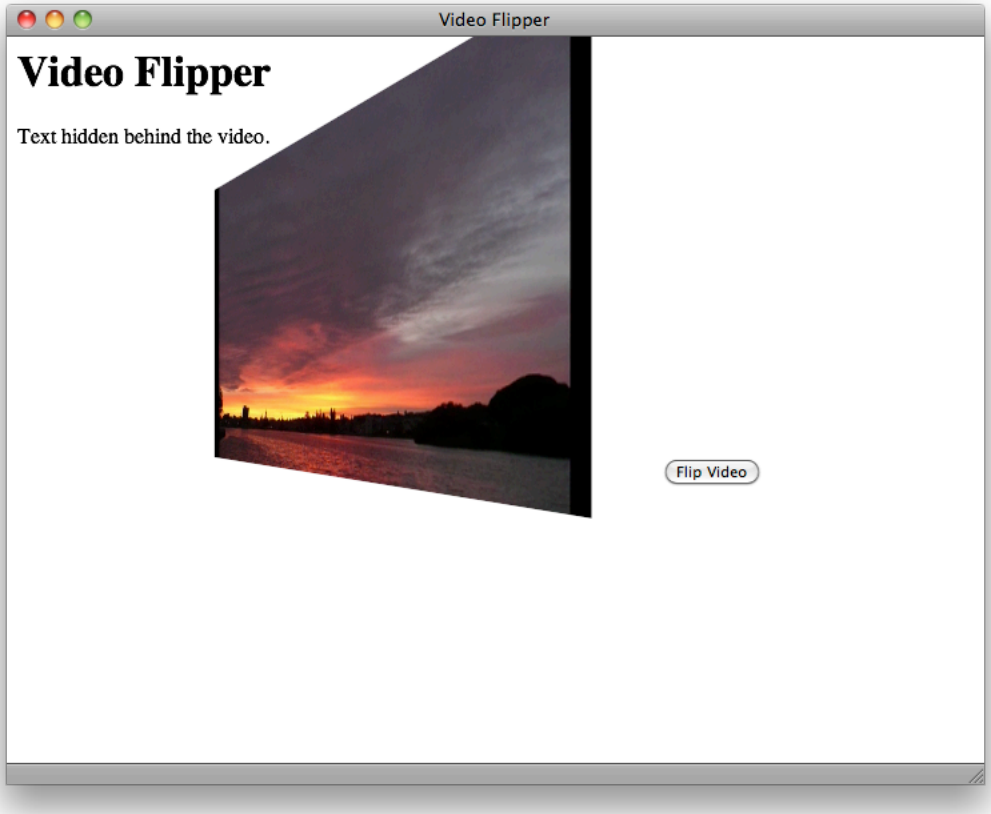
Listing 4-2 3D rotation with perspective

```
<html>
<head>
    <title>Video Flipper</title>

<style>
    video {
    -webkit-transition-property: -webkit-transform;
    -webkit-transition-duration: 4s;
    }

    body { -webkit-perspective: 500;}
```

```
</style>

<script type="text/javascript">
var flipped = 0 ;

function flipper() {
    if (flipped)
        myVideo.style.webkitTransform = "rotateY(0deg)";
    else
        myVideo.style.webkitTransform = "rotateY(180deg)";
    flipped = !flipped ;
}

</script>

</head>
<body>

    <h1>
    Video Flipper
    </h1>

    <p>
    Text hidden behind the video.
    </p>
    <div  style = " position:relative; top: -40px;" >
        <video src="myMovie.m4v" id="myVideo" autoplay controls>
        </video>
        <input type="button" value="Flip Video" onclick="flipper()">
    </div>
</body>
```

```
</html>
```

Figure 4-6 Video in mid-rotation



Document Revision History

This table describes the changes to *Safari HTML5 Audio and Video Guide*.

Date	Notes
2011-07-05	Updated with description of fullscreen mode for HTML elements.
2010-12-16	Updated to describe inline audio on iPhone and iPod touch.
2010-11-15	Updated for Safari 5.02 and iOS 4.2.
2010-03-18	Corrected typos in sample code and text. Minor changes throughout.
2010-03-09	New document showing how to use the HTML5 <audio> and <video> elements using HTML, JavaScript methods, DOM events, and CSS styles.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010