Video and audio APIs

Previous Overview: Client-side web APIs Next

HTML5 comes with elements for embedding rich media in documents — `<video>` and `<audio>` — which in turn come with their own APIs for controlling playback, seeking, etc. This article shows you how to do common tasks such as creating custom playback controls.

Prerequisites:	JavaScript basics (see first steps, building blocks, JavaScript objects), the basics of Client-side APIs
Objective:	To learn how to use browser APIs to control video and audio playback.

HTML5 video and audio

The `<video>` and `<audio>` elements allow us to embed video and audio into web pages. As we showed in Video and audio content, a typical implementation looks like this:

```
<video controls>
  <source src="rabbit320.mp4" type="video/mp4">
  <source src="rabbit320.webm" type="video/webm">
  <p>Your browser doesn't support HTML5 video. Here is a <a href="rabbit32
</video>
```

This creates a video player inside the browser like so:

Below is a video that will play in all modern browsers

You can review what all the HTML features do in the article linked above; for our purposes here, the most interesting attribute is `controls`, which enables the default set of playback controls. If you don't specify this, you get no playback controls:

Below is a video that will play in all modern browsers

This is not as immediately useful for video playback, but it does have advantages. One big issue with the native browser controls is that they are different in each browser — not very good for cross-browser support! Another big issue is that the native controls in most browsers aren't very keyboard-accessible.

You can solve both these problems by hiding the native controls (by removing the `controls` attribute), and programming your own with HTML, CSS, and JavaScript. In the next section we'll look at the basic tools we have available to do this.

The HTMLMediaElement API

Part of the HTML5 spec, the `HTMLMediaElement` API provides features to allow you to control video and audio players programmatically — for example `HTMLMediaElement.play()`, `HTMLMediaElement.pause()`, etc. This interface is available to both `<audio>` and `<video>` elements, as the features you'll want to implement are nearly identical. Let's go through an example, adding features as we go.

Our finished example will look (and function) something like the following:

Sintel © copyright Blender Foundation | www.sintel.org.

Getting started

To get started with this example, download our media-player-start.zip and unzip it into a new directory on your hard drive. If you downloaded our examples repo, you'll find it in `javascript/apis/video-audio/start/`

At this point, if you load the HTML you should see a perfectly normal HTML5 video player, with the native controls rendered.

Exploring the HTML

Open the HTML index file. You'll see a number of features; the HTML is dominated by the video player and its controls:

```
<div class="player">
  <video controls>
    <source src="video/sintel-short.mp4" type="video/mp4">
    <source src="video/sintel-short.webm" type="video/webm">
    <!-- fallback content here -->
  </video>
  <div class="controls">
    <button class="play" data-icon="P" aria-label="play pause toggle"></bu
    <button class="stop" data-icon="S" aria-label="stop"></button>
    <div class="timer">
      <div></div>
      <span aria-label="timer">00:00</span>
```

```
    </div>
    <button class="rwd" data-icon="B" aria-label="rewind"></button>
    <button class="fwd" data-icon="F" aria-label="fast forward"></button>
  </div>
</div>
```

- The whole player is wrapped in a `<div>` element, so it can all be styled as one unit if needed.
- The `<video>` element contains two `<source>` elements so that different formats can be loaded depending on the browser viewing the site.
- The controls HTML is probably the most interesting:
  - We have four `<button>`s — play/pause, stop, rewind, and fast forward.
  - Each `<button>` has a `class` name, a `data-icon` attribute for defining what icon should be shown on each button (we'll show how this works in the below section), and an `aria-label` attribute to provide an understandable description of each button, since we're not providing a human-readable label inside the tags. The contents of `aria-label` attributes are read out by screenreaders when their users focus on the elements that contain them.
  - There is also a timer `<div>`, which will report the elapsed time when the video is playing. Just for fun, we are providing two reporting mechanisms — a `<span>` containing the elapsed time in minutes and seconds, and an extra `<div>` that we will use to create a horizontal indicator bar that gets longer as the time elapses. To get an idea of what the finished product will look like, check out our finished version.

Exploring the CSS

Now open the CSS file and have a look inside. The CSS for the example is not too complicated, but we'll highlight the most interesting bits here. First of all, notice the `.controls` styling:

```
.controls {
  visibility: hidden;
  opacity: 0.5;
  width: 400px;
  border-radius: 10px;
  position: absolute;
  bottom: 20px;
  left: 50%;
  margin-left: -200px;
```

```
  background-color: black;
  box-shadow: 3px 3px 5px black;
  transition: 1s all;
  display: flex;
}

.player:hover .controls, player:focus .controls {
  opacity: 1;
}
```

- We start off with the `visibility` of the custom controls set to `hidden`. In our JavaScript later on, we will set the controls to `visible`, and remove the `controls` attribute from the `<video>` element. This is so that, if the JavaScript doesn't load for some reason, users can still use the video with the native controls.
- We give the controls an `opacity` of 0.5 by default, so that they are less distracting when you are trying to watch the video. Only when you are hovering/focusing over the player do the controls appear at full opacity.
- We lay out the buttons inside the control bar out using Flexbox (`display`: flex), to make things easier.

Next, let's look at our button icons:

```
@font-face {
    font-family: 'HeydingsControlsRegular';
    src: url('fonts/heydings_controls-webfont.eot');
    src: url('fonts/heydings_controls-webfont.eot?#iefix') format('embedded
         url('fonts/heydings_controls-webfont.woff') format('woff'),
         url('fonts/heydings_controls-webfont.ttf') format('truetype');
    font-weight: normal;
    font-style: normal;
}

button:before {
  font-family: HeydingsControlsRegular;
  font-size: 20px;
  position: relative;
  content: attr(data-icon);
  color: #aaa;
  text-shadow: 1px 1px 0px black;
}
```

First of all, at the top of the CSS we use a `@font-face` block to import a custom web font. This is an icon font — all the characters of the alphabet equate to common icons you might want to use in an application.

Next we use generated content to display an icon on each button:

- We use the `::before` selector to display the content before each `<button>` element.
- We use the `content` property to set the content to be displayed in each case to be equal to the contents of the `data-icon` attribute. In the case of our play button, `data-icon` contains a capital "P".
- We apply the custom web font to our buttons using `font-family`. In this font, "P" is actually a "play" icon, so therefore the play button has a "play" icon displayed on it.

Icon fonts are very cool for many reasons — cutting down on HTTP requests because you don't need to download those icons as image files, great scalability, and the fact that you can use text properties to style them — like `color` and `text-shadow`.

Last but not least, let's look at the CSS for the timer:

```
.timer {
  line-height: 38px;
  font-size: 10px;
  font-family: monospace;
  text-shadow: 1px 1px 0px black;
  color: white;
  flex: 5;
  position: relative;
}

.timer div {
  position: absolute;
  background-color: rgba(255,255,255,0.2);
  left: 0;
  top: 0;
  width: 0;
  height: 38px;
  z-index: 2;
}

.timer span {
  position: absolute;
  z-index: 3;
```

```
  left: 19px;
}
```

- We set the outer `.timer <div>` to have flex: 5, so it takes up most of the width of the controls bar. We also give it `position : relative`, so that we can position elements inside it conveniently according to it's boundaries, and not the boundaries of the `<body>` element.
- The inner `<div>` is absolutely positioned to sit directly on top of the outer `<div>`. It is also given an initial width of 0, so you can't see it at all. As the video plays, the width will be increased via JavaScript as the video elapses.
- The `<span>` is also absolutely positioned to sit near the left hand side of the timer bar.
- We also give our inner `<div>` and `<span>` the right amount of `z-index` so that the timer will be displayed on top, and the inner `<div>` below that. This way, we make sure we can see all the information — one box is not obscuring another.

Implementing the JavaScript

We've got a fairly complete HTML and CSS interface already; now we just need to wire up all the buttons to get the controls working.

1. Create a new JavaScript file in the same directory level as your index.html file. Call it `custom-player.js`.

2. At the top of this file, insert the following code:

```
var media = document.querySelector('video');
var controls = document.querySelector('.controls');

var play = document.querySelector('.play');
var stop = document.querySelector('.stop');
var rwd = document.querySelector('.rwd');
var fwd = document.querySelector('.fwd');

var timerWrapper = document.querySelector('.timer');
var timer = document.querySelector('.timer span');
var timerBar = document.querySelector('.timer div');
```

   Here we are creating variables to hold references to all the objects we want to manipulate. We have three groups:

- The `<video>` element, and the controls bar.
- The play/pause, stop, rewind, and fast forward buttons.
- The outer timer wrapper `<div>`, the digital timer readout `<span>`, and the inner `<div>` that gets wider as the time elapses.

3. Next, insert the following at the bottom of your code:

```
media.removeAttribute('controls');
controls.style.visibility = 'visible';
```

These two lines remove the default browser controls from the video, and make the custom controls visible.

Playing and pausing the video

Let's implement probably the most important control — the play/pause button.

1. First of all, add the following to the bottom of your code, so that the `playPauseMedia()` function is invoked when the play button is clicked:

```
play.addEventListener('click', playPauseMedia);
```

2. Now to define `playPauseMedia()` — add the following, again at the bottom of your code:

```
function playPauseMedia() {
  if(media.paused) {
    play.setAttribute('data-icon','u');
    media.play();
  } else {
    play.setAttribute('data-icon','P');
    media.pause();
  }
}
```

He we use an `if` statement to check whether the video is paused. The `HTMLMediaElement.paused` property returns true if the media is paused, which is any time the video is not playing, including when it is sat at 0 duration after it first loads. If it is paused, we set the `data-icon` attribute value on the play button to "u",

which is a "paused" icon, and invoke the `HTMLMediaElement.play()` method to play the media.

On the second click, the button will be toggled back again — the "play" icon will be shown again, and the video will be paused with `HTMLMediaElement.paused()`.

Stopping the video

1. Next, let's add functionality to handle stopping the video. Add the following `addEventListener()` lines below the previous one you added:

```
stop.addEventListener('click', stopMedia);
media.addEventListener('ended', stopMedia);
```

The `click` event is obvious — we want to stop the video by running our `stopMedia()` function when the stop button is clicked. We do however also want to stop the video when it finishes playing — this is marked by the `ended` event firing, so we also set up a listener to run the function on that event firing too.

2. Next, let's define `stopMedia()` — add the following function below `playPauseMedia()`:

```
function stopMedia() {
  media.pause();
  media.currentTime = 0;
  play.setAttribute('data-icon','P');
}
```

there is no `stop()` method on the HTMLMediaElement API — the equivalent is to `pause()` the video, and set its `currentTime` property to 0. Setting `currentTime` to a value (in seconds) immediately jumps the media to that position.

All there is left to do after that is to set the displayed icon to the "play" icon. Regardless of whether the video was paused or playing when the stop button is pressed, you want it to be ready to play afterwards.

Seeking back and forth

There are many ways that you can implement rewind and fast forward functionality; here we are showing you a relatively complex way of doing it, which doesn't break when the

different buttons are pressed in an unexpected order.

1. First of all, add the following two `addEventListener()` lines below the previous ones:

```
rwd.addEventListener('click', mediaBackward);
fwd.addEventListener('click', mediaForward);
```

2. Now on to the event handler functions — add the following code below your previous functions to define `mediaBackward()` and `mediaForward()`:

```
var intervalFwd;
var intervalRwd;

function mediaBackward() {
  clearInterval(intervalFwd);
  fwd.classList.remove('active');

  if(rwd.classList.contains('active')) {
    rwd.classList.remove('active');
    clearInterval(intervalRwd);
    media.play();
  } else {
    rwd.classList.add('active');
    media.pause();
    intervalRwd = setInterval(windBackward, 200);
  }
}

function mediaForward() {
  clearInterval(intervalRwd);
  rwd.classList.remove('active');

  if(fwd.classList.contains('active')) {
    fwd.classList.remove('active');
    clearInterval(intervalFwd);
    media.play();
  } else {

    fwd.classList.add('active');
    media.pause();
    intervalFwd = setInterval(windForward, 200);
```

```
    }
}
```

You'll notice that first we initialize two variables — `intervalFwd` and `intervalRwd` — you'll find out what they are for later on.

Let's step through `mediaBackward()` (the functionality for `mediaForward()` is exactly the same, but in reverse):

1. We clear any classes and intervals that are set on the fast forward functionality — we do this because if we press the `rwd` button after pressing the `fwd` button, we want to cancel any fast forward functionality and replace it with the rewind functionality. If we tried to do both at one, the player would break.
2. We use an `if` statement to check whether the `active` class has been set on the `rwd` button, indicating that it has already been pressed. The `classList` is a rather handy property that exists on every element — it contains a list of all the classes set on the element, as well as methods for adding/removing classes, etc. We use the `classList.contains()` method to check whether the list contains the `active` class. This returns a boolean `true/false` result.
3. If `active` has been set on the `rwd` button, we remove it using `classList.remove()`, clear the interval that has been set when the button was first pressed (see below for more explanation), and use `HTMLMediaElement.play()` to cancel the rewind and start the video playing normally.
4. If it hasn't yet been set, we add the `active` class to the `rwd` button using `classList.add()`, pause the video using `HTMLMediaElement.pause()`, then set the `intervalRwd` variable to equal a `setInterval()` call. When invoked, `setInterval()` creates an active interval, meaning that it runs the function given as the first parameter every x milliseconds, where x is the value of the 2nd parameter. So here we are running the `windBackward()` function every 200 milliseconds — we'll use this function to wind the video backwards constantly. To stop a `setInterval()` running, you have to call `clearInterval()`, giving it the identifying name of the interval to clear, which in this case is the variable name `intervalRwd` (see the `clearInterval()` call earlier on in the function).

3. last of all for this section, we need to define the `windBackward()` and `windForward()` functions invoked in the `setInterval()` calls. Add the following below your two previous functions:

```
function windBackward() {
```

```
  if(media.currentTime <= 3) {
    rwd.classList.remove('active');
    clearInterval(intervalRwd);
    stopMedia();
  } else {
    media.currentTime -= 3;
  }
}

function windForward() {
  if(media.currentTime >= media.duration - 3) {
    fwd.classList.remove('active');
    clearInterval(intervalFwd);
    stopMedia();
  } else {
    media.currentTime += 3;
  }
}
```

Again, we'll just run through the first one of these functions as they work almost identically, but in reverse to one another. In `windBackward()` we do the following — bear in mind that when the interval is active, this function is being run once every 200 milliseconds.

1. We start off with an `if` statement that checks to see whether the current time is less than 3 seconds, i.e., if rewinding by another three seconds would take it back past the start of the video. This would cause strange behaviour, so if this is the case we stop the video playing by calling `stopMedia()`, remove the `active` class from the rewind button, and clear the `intervalRwd` interval to stop the rewind functionality. If we didn't do this last step, the video would just keep rewinding forever.
2. If the current time is not within 3 seconds of the start of the video, we simply remove three seconds from the current time by executing `media.currentTime -= 3`. So in effect, we are rewinding the video by 3 seconds, once every 200 milliseconds.

Updating the elapsed time

The very last piece of our media player to implement is the time elapsed displays. To do this we'll run a function to update the time displays every time the `timeupdate` event is fired on the `<video>` element. This frequency with which this event fires depends on your browser, CPU power, etc (see this stackoverflow post).

Add the following `addEventListener()` line just below the others:

```
media.addEventListener('timeupdate', setTime);
```

Now to define the `setTime()` function. Add the following at the bottom of your file:

```
function setTime() {
  var minutes = Math.floor(media.currentTime / 60);
  var seconds = Math.floor(media.currentTime - minutes * 60);
  var minuteValue;
  var secondValue;

  if (minutes < 10) {
    minuteValue = '0' + minutes;
  } else {
    minuteValue = minutes;
  }

  if (seconds < 10) {
    secondValue = '0' + seconds;
  } else {
    secondValue = seconds;
  }

  var mediaTime = minuteValue + ':' + secondValue;
  timer.textContent = mediaTime;

  var barLength = timerWrapper.clientWidth * (media.currentTime/media.dura
  timerBar.style.width = barLength + 'px';
}
```

This is a fairly long function, so let's go through it step by step:

1. First of all, we work out the number of minutes and seconds in the `HTMLMediaElement.currentTime` value.
2. Then we initialize two more variables — `minuteValue` and `secondValue`.
3. The two `if` statements work out whether the number of minutes and seconds are less than 10. If so, they add a leading zero to the values, in the same way that a digital clock display works.

4. The actual time value to display is set as `minuteValue` plus a colon character plus `secondValue`.
5. The `Node.textContent` value of the timer is set to the time value, so it displays in the UI.
6. The length we should set the inner `<div>` to is worked out by first working out the width of the outer `<div>` (any element's `clientWidth` property will contain its length), and then multiplying it by the `HTMLMediaElement.currentTime` divided by the total `HTMLMediaElement.duration` of the media.
7. We set the width of the inner `<div>` to equal the calculated bar length, plus "px", so it will be set to that number of pixels.

Fixing play and pause

There is one problem left to fix. If the play/pause or stop buttons are pressed while the rewind or fast forward functionality is active, they just don't work. How can we fix it so that they cancel the `rwd`/`fwd` button functionality and play/stop the video as you'd expect? This is fairly easy to fix.

First of all, add the following lines inside the `stopMedia()` function — anywhere will do:

```
rwd.classList.remove('active');
fwd.classList.remove('active');
clearInterval(intervalRwd);
clearInterval(intervalFwd);
```

Now add the same lines again, at the very start of the `playPauseMedia()` function (just before the start of the `if` statement).

At this point, you could delete the equivalent lines from the `windBackward()` and `windForward()` functions, as that functionality has been implemented in the `stopMedia()` function instead.

Note: You could also further improve the efficiency of the code by creating a separate function that runs these lines, then calling that anywhere it is needed, rather than repeating the lines multiple times in the code. But we'll leave that one up to you.

Summary

I think we've taught you enough in this article. The `HTMLMediaElement` API makes a wealth of functionality available for creating simple video and audio players, and that's only the tip of the iceberg. See the "See also" section below for links to more complex and interesting functionality.

Here are some suggestions for ways you could enhance the existing example we've built up:

1. The time display currently breaks if the video is an hour long or more (well, it won't display hours; just minutes and seconds). Can you figure out how to change the example to make it display hours?

2. Because `<audio>` elements have the same `HTMLMediaElement` functionality available to them, you could easily get this player to work for an `<audio>` element too. Try doing so.

3. Can you work out a way to turn the timer inner `<div>` element into a true seek bar/scrobbler — i.e., when you click somewhere on the bar, it jumps to that relative position in the video playback? As a hint, you can find out the X and Y values of the element's left/right and top/bottom sides via the `getBoundingClientRect()` method, and you can find the coordinates of a mouse click via the event object of the click event, called on the `Document` object. For example:

```
document.onclick = function(e) {
  console.log(e.x) + ',' + console.log(e.y)
}
```

See also

- `HTMLMediaElement`
- Video and audio content — simple guide to `<video>` and `<audio>` HTML.
- Audio and video delivery — detailed guide to delivering media inside the browser, with many tips, tricks, and links to further more advanced tutorials.
- Audio and video manipulation — detailed guide to manipulating audio and video, e.g. with Canvas API, Web Audio API, and more.
- `<video>` and `<audio>` reference pages.
- Media formats supported by the HTML audio and video elements.

Previous Overview: Client-side web APIs Next

In this module

- Introduction to web APIs
- Manipulating documents
- Fetching data from the server
- Third party APIs
- Drawing graphics
- Video and audio APIs
- Client-side storage