Writing HTML Markup - Part 2

The topics we will cover in this lab are:

- Embedding media
- Responsive video and iframes
- Creating pop-ups with the dialog element

Embedding media in HTML5

For many, HTML5 first entered their vocabulary when Apple refused to add support for Flash technology in their iOS devices. Flash had gained market dominance (some would argue market stranglehold) as the plugin of choice to serve up video through a web browser. However, rather than using Adobe's proprietary technology, Apple decided to rely on HTML5 instead to handle rich media rendering. While HTML5 was making good headway in this area anyway, Apple's public support of HTML5 gave it a major leg up and helped its media tools gain greater traction in the wider community.

Adding video and audio in HTML

Here's a "simple as can be" example of how to link to a video file on your page:

```
<video src="myVideo.mp4"></video>
```

HTML allows a single `video` tag (or `audio` tag for audio) to do all the heavy lifting. It's also possible to insert text between the opening and closing tag to inform users when there is a problem. There are also additional attributes you'd ordinarily want to add, such as the height and width. Let's add these in:

```
<video src="myVideo.mp4" width="640" height="480">
    If you're reading this either the video didn't load or your browser is
    waaaayyyyyy old!
</video>
```

Now, if we add the preceding code snippet into our page and look at it in some browsers, it will appear, but there will be no controls for playback. To ensure we show the default playback controls, we need to add the `controls` attribute. For the sake of illustration, we could also add the `autoplay` attribute.

These days, you will also usually need to add the `muted` attribute if you want your video to autoplay. Everyone detested videos automatically blaring out audio into a crowded office (if you want to listen to a little Rick Astley at lunch, who am I to judge) so much that most browsers won't autoplay without the `muted` attribute present.

Here's an example with the `controls` and `autoplay` attributes added:

```
<video src="myVideo.mp4" width="640" height="480" controls autoplay>
    If you're reading this either the video didn't load or your browser is
    waaaayyyyyy old!
</video>
```

Further attributes include `preload` to control the preloading of media, `loop` to repeat the video, and `poster` to define a poster frame for the video-the image that shows while a video is loading.

To use an attribute, simply add it to the tag. Here's an example that includes all these attributes:

```
<video
    src="myVideo.mp4"
    width="640"
    height="480"
    controls
    autoplay
    muted
    preload="auto"
    loop
    poster="myVideoPoster.png"
>
    If you're reading this either the video didn't load or your browser is
    waaaayyyyyy old!
</video>
```

Providing alternate media sources

The `source` tag enables us to provide alternate sources for the media. For example, alongside providing an MP4 version of the video, if we wanted to provide support for a new format, we could easily do so. Further still, if the user didn't have any suitable playback technology in the browser, we could provide download links to the files themselves. Here's an example:

```
<video
    width="640"
    height="480"
    controls
    preload="auto"
    loop
    poster="myVideoPoster.png"
>
    <source src="video/myVideo.sp8" type="video/super8" />
    <source src="video/myVideo.mp4" type="video/mp4" />
    <p>
        <b>Download Video:</b>
        MP4 Format:
        <a href="myVideo.mp4">"MP4"</a>
    </p>
</video>
```

In that case, we first specified a source of a made-up video format called `super8`. The browser goes top to bottom deciding what to play, so if it doesn't support `super8`, it moves on to the next source; `mp4` in this case. And on and on it goes, moving down to the download link if it can't reconcile any of the sources listed. The `type` attribute tells the browser the MIME type of the resource. If you fail to specify this, the browser will fetch the content and try and play it anyway. But if you know the MIME type, you should add it in the `type` attribute. The prior code example and the sample video file (which, incidentally, is me appearing in the UK soap *Coronation Street* back when I had hair and hopes of starring alongside DeNiro) in MP4 format are in the `example_02-02.html` section of the lab code.

Audio and video tags work almost identically

The `audio` tag works on the same principles as the `video` tag, with the same attributes (excluding `width`, `height`, and `poster`). The main difference between the two is the fact that `audio` has no playback area for visible content.

Responsive HTML5 video and iframes

The only problem with our lovely HTML video implementation is that it's not responsive. For HTML embedded video, the fix is easy. Simply remove any height and width attributes in the markup (for example, remove `width="640" height="480"`) and add the following in the CSS:

```
video {
    max-width: 100%;
    height: auto;
}
```

However, while that works fine for files that we might be hosting locally, it doesn't solve the problem of videos embedded within an `iframe` (take a bow YouTube, Vimeo, et al.). The following code will add a film trailer for *Midnight Run* from YouTube:

```
<iframe
    width="960"
    height="720"
    src="https://www.youtube.com/watch?v=B1_N28DA3gY"
    frameborder="0"
    allowfullscreen
></iframe>
```

You don't even need to work out the aspect ratio and plug it in yourself; there's an online service that can do it for you. Just head to [https://embedresponsively.com/] and paste your `iframe` URL in. It will spit you out a simple chunk of code you can paste into your page. For example, our *Midnight Run* trailer results in this (note the `padding-bottom` value to define the aspect ratio):

```
<style>
    .embed-container {
        position: relative;
        padding-bottom: 56.25%;
        height: 0;
        overflow: hidden;
        max-width: 100%;
    }
    .embed-container iframe,
    .embed-container object,
    .embed-container embed {
        position: absolute;
        top: 0;
        left: 0;
        width: 100%;
        height: 100%;
    }
</style>
<div class="embed-container">
    <iframe
        src="https://www.youtube.com/embed/B1_N28DA3gY"
        frameborder="0"
        allowfullscreen
```

```
    ></iframe>
</div>
```

That's all there is to it. Simply add to your page and you're done: we now have a fully responsive YouTube video (note: kids, don't pay any attention to Mr. DeNiro; smoking is bad)!

The modern way to embed iframes and keep aspect ratio

If you only need to support more modern browsers (check at [https://caniuse.com/mdn-css_properties_aspect-ratio]), you can set the aspect ratio of something like an `iframe` with the `aspect-ratio` CSS property. For an `iframe` we want to keep in a 16:9 ratio, such as a YouTube video, we could add a suitable class and do this:

```
.iframe-16-9 {
    aspect-ratio: 16/9;
    max-width: 100%;
}
```

You can see this in action for yourself in `example_02-04.html` of the lab code. Of course, the `aspect-ratio` property is not limited to `iframe`s -- you can use it on any box in CSS that you want to maintain certain proportions with.

The loading attribute

If you find yourself adding a lot of images or `iframe` elements to your page, consider the `loading` attribute. Setting this attribute to `lazy` means the browser will only go and fetch the item if the user scrolls it into view. This is really useful for any media you know will certainly not be visible when the page first displays, meaning the cost of loading that item is saved until it is actually needed. It's literally as simple as adding it like this:

```
<img src="home-image.png" alt="A rendering of the home page" loading="lazy" />
```

Or, if you were adding something like a YouTube video embed, you could make the same economy like this:

```
<iframe
    loading="lazy"
    width="560"
    height="315"
    src="https://www.youtube.com/embed/ukzFI9rgwfU"
    title="YouTube video player"
    frameborder="0"
    allow="accelerometer; autoplay; clipboard-write; encrypted-media;
    gyroscope; picture-in-picture"
    allowfullscreen
></iframe>
```

In case you were wondering, the opposite of `lazy` in this context is `eager`, which is the default if you don't add the attribute.

The <dialog> element

You know what a dialog is, even if you know it by a different name. It's the pop-up you get on a site asking you to subscribe to a newsletter, or log in to a service. Coding this kind of thing has always been problematic. Worse still, if

you didn't think they were problematic it is likely because you didn't realize there were accessibility problems with your implementation. It's likely users using only a keyboard could struggle to get focus into the dialog or be unable to leave it.

Thankfully, we now have a specific element to handle these use cases. In the code for this lab, you will want to consider `example_02-05.html`.

Inside that HTML file is also the relevant CSS and JavaScript to produce a simple but fully working `dialog`.

When the modal `dialog` is opened, it automatically gets centered, focused, and the `backdrop` of the dialog --- which is automatically inserted --- covers everything below, preventing any of the underlying content from being interacted with. You can dismiss the `dialog` with the `button` inside it, or a keyboard press of the *Escape* key; the keyboard support is functionality you get for free!

The `dialog` is a wonderful addition to the web platform.

Here is the relevant HTML from our example:

```
        <button id="launchDialog">Where is the dialog?</button>
        <p id="formResult"></p>
        <dialog id="dialogEle">
            <form method="dialog">
                <h1>How about this? A native Dialog.</h1>
                <p>So, only thing left to do is dismiss me.</p>
                <button value="Dismissed!">Dismiss Dialog</button>
            </form>
        </dialog>
```

Worth noting here that I have included the content of the `dialog` in a `form`. That isn't necessary, but doing so allows you to use the special `method="dialog"` attribute, which "returns" the value of the `button` doing the submitting of the form; more in-built functionality that can prove useful!

Also worth noting is that when you click the `button` to open a `dialog`, it automatically gets an `open` attribute, which you can see in the dev tools. This can be used as a styling hook. I've added that in the styles when animating the modal `backdrop` when it first appears, although it isn't actually necessary in this instance.

Let's take a look at the styles:

```
            dialog {
                border-radius: 10px;
            }
            dialog::backdrop {
                animation: fade 0.5s ease forwards;
            }
            button {
                height: 40px;
                padding: 5px 10px;
            }
            button:focus {
                outline: 2px solid #f90;
            }
            @keyframes fade {
                0% {
                    background-color: transparent;
```

```
            }
            100% {
                background-color: rgb(0 57 24 / 0.6);
            }
        }
```

None of the styles are necessary for the `dialog` to function, but I do always like to add a stronger `:focus` style than the default for better accessibility. However, I thought it would also be interesting to bring the `::backdrop` pseudo-element to your attention, as that is how you can style the area behind the modal `dialog`.

If any of the syntaxes or styles in there are unfamiliar to you, don't worry. We cover all of those in detail later in the course. All that is necessary here is to understand that you can style the `dialog` and `backdrop` however you see fit.

Let's understand the `dialog` functions:

```
const dialogEle = document.getElementById("dialogEle");
const launchBtn = document.getElementById("launchDialog");
const formResult = document.getElementById("formResult");
launchBtn.addEventListener("click", () => dialogEle.showModal());
dialogEle.addEventListener("close", () => {
    formResult.textContent = dialogEle.returnValue;
});
```

Besides creating `const` references to the `dialog` itself, the `button` that opens it, and a `p` tag to display the value of the `button` that can be clicked inside the `dialog`, we have two event listeners. The first opens the `dialog` when the `launchBtn` button is clicked with the `showModal()` method, and the second listens for the `close` event. The `close` is unique to `dialog` and gets triggered when submitting a form inside a `dialog` that has a `method` of `dialog`, or if the user exits the form with, say, a press of *Escape* on their keyboard `.`

The default `type` of a `button` is `submit`, so in our example above, where we want to `submit` the `form` inside the `dialog`, there is no need to declare it. However, imagine you are building a form, as we will in detail in *Lab 13*, *Forms*, and you have a `button` in your form for some other purpose; you don't want the form submitting when the user clicks that, rather than the correct submit button. As such, I recommend being in the habit of always adding `type="button"` whenever you use the `button` element.

In our example, we are using the `returnValue` of the `dialog` to set the text of our `p` tag. That `returnValue` is set to the `value` attribute of the `button` that submits the `form` in the `dialog`. Not essential, but potentially useful depending on your use case.

It's also possible to show a dialog in a non-modal manner. To do that, instead of using the `showModal()` method, use the `show()` method. In this situation, no `backdrop` is inserted and the `dialog` is inserted using absolute positioning by default.

You now know how to create, control, and style a fully working `dialog`.

Summary

We've covered a lot in this lab. Everything from the basics of creating a page that validates as HTML5 through to embedding rich media (video and audio) into our markup and ensuring it behaves responsively. Although not specific to responsive designs, we've also covered how we can write semantically rich and meaningful code and considered how we might ensure pages are meaningful and usable for users who are relying on assistive technology.

You should now have a decent understanding of the wide and varied vocabulary of HTML elements at your disposal.