

# Media Project – Phase02 inClass Project

\*Video used for educational purposes only

In this inClass tutorial project you will be creating a video player app that displays a video along with custom controls done as icon fonts. There will also be a text effect added to the header text at the top. The finished app will look like this:



Keep in mind that this type of app could be just one component of a larger web page!

## Learning Objectives:

- CSS attributes and selection techniques
- CSS positioning for layout: relative and absolute positioning, float
- Advanced CSS topics such as: local storage, icon fonts, and ::before and ::after pseudo elements
- HTML and CSS related to video media
- JavaScript's Media API functions

## Starting Resources:

You have been provided a folder of starting files for this project under your lesson in E360. It contains the complete index.html file for the app as well as the starting CSS file (inside the css folder), fontawesome font-related files (fonts folder), images for the project (images folder), and a JavaScript file created via Modernizr.com (js folder).

Download the **startFiles** folder with its files and store it somewhere on your local machine so you can work with it.

If you don't already have the [Brackets](https://brackets.io) editor installed on your machine, go to [brackets.io](https://brackets.io) to download and install it. It is a free editor that was specifically made for working with HTML, CSS, and JavaScript and we will be using it during this class. *If you have another editor that you really like to use for web apps, you can feel free to use it as well as long as I can open your files.*

Once you have Brackets installed, you should be able to *right-click* on your **startFiles** folder and choose **Open as Brackets Project** in Windows. On a Mac, you can do the same thing by choosing the **Open Folder** option.

## Looking through the provided HTML:

Note: I may not show the entire HTML file here so be sure to look through the provided index.html file in your code editor in its entirety...

```
<html lang="en">
<head>

    <link href="css/controls.css" rel="stylesheet">
    <link href='http://fonts.googleapis.com/css?family=Rock+Salt' rel='stylesheet'
type='text/css'>

    <script src="js/modernizr.custom.js"></script>

</head>
```

We are linking in one CSS file we'll be working with in a bit as well as linking in a Google font named 'Rock Salt'. You can also see the Modernizr JavaScript file I'm referred to being

brought in via the script tag at the bottom of the head section (`modernizr.custom.js`). This file is provided as part of the project's start files and is used to determine if the user's browser supports certain features such as HTML input tags and input types. For more information about Modernizr, the feature detection JavaScript library for HTML5/CSS3, go to [modernizr.com](http://modernizr.com).

Ok? Let's continue with the HTML structure of the body...

```
<body onload="initializePlayer()">
```

The **onload** event will fire when the page's content finishes loading. When that happens the `initializePlayer()` event handler function will be called and executed...

```
<div id="wrapper">    wrap all of our content in a wrapper container
```

```
<h2>Needlegun Breechloader - 1866</h2>
```

```
<div id="aboveTheControls" class="shadowEffect">
```

```
    <!-- note: using just width (no height) controls bar stays pinned correctly -->
```

```
    <video class="shadowEffect" width="800px" controls preload="metadata">
```

```
        <source src="../media/Needlegun.webm">
```

```
        <source src="../media/Needlegun.mp4">
```

```
    </video>
```

Here, we set up our video in a div container. The **<video>** element was added in HTML5 to give us a way to add video media *natively* in the browser – no more Flash plugin needed – so they say. Setting just **width** (no height) causes browser to retain video's proportion. The **controls** attribute, if present, causes the default video controls to be shown (browser-specific). The **preload** attribute can be used to make suggestions to the browser on what to preload related to the video. A value of "**metadata**" suggests the browser preload any special (meta) data stored with the video such as duration, title, date created, producer info, etc...

```
</div>
```

```

```

We'll be using this image as a "poster" cover for our video so we don't just see frame 1 or in the case of an mp4, a pixelated first frame – yuk!

Ok, let's take a look at the next part of the body's structure which sets up the custom controls for our video.

```
<div id="controls">

    <div id="progressBar"><span id="played"></span></div>

    <div id="lowerControls">

        <button id="playPause" class="playBtn"></button>

        <button id="stopButton"></button>

        <span id="currentTime">0:00</span> /
        <span id="durationTime">0:00</span>

        <button id="fullScreen"></button>

        <input id="volumeSlider" class="sliderBar" type="range" min="0"
max="1" value="1" step="0.1">

        <button id="mute" class="mute"></button>

    </div> <!-- end div#lowerControls -->
</div>
</div>

<script src="http://code.jquery.com/jquery-1.12.4.min.js"></script>
<script src="js/controls.js"></script>

</body>
</html>
```

Things to note in the above code:

- Our video controls are wrapped in a **div** with a **class** of “controls”.
- The first child is a **div** with an **id** of “progressBar” which just contains an empty **span** tag that has an **id** of “played”. What's up with that? Well, we'll be dealing with bringing a progress bar to life for our video using this via our CSS and JavaScript code.

- Next, we wrap the rest of the tags in a **div** with an **id** of “**lowerControls**”. This div will indeed contain the rest of our custom video controls which will all be placed *below* our progress bar.
- Speaking of our controls, we now see a couple of **button** tags for our play/pause and stop buttons – standard fare for audio/video controls. Note the **class** of “**playBtn**” on the play/pause button – we’ll be using that shortly in our CSS.

Then, we see two **span** tags that will be used to display time-related values for our video. First, the current time of our playhead and secondly, the video’s duration.


Next, we see another button tag that will be used to toggle between normal vs full screen viewing.

Ok, a lot of control-related tags here, but hang in there a little longer...

Now we see an interesting element – an **input** tag that sets up a *range slider* that will allow the user to adjust the volume of our video as it plays by sliding a thumb slider. This one will also get a little browser specific as its support varies.

Finally, we have a mute **button** for our video volume that toggles between mute/unmute.

- Lastly, we have a couple of **<script>** tags to import the **jQuery** core library (check [www.jquery.com](http://www.jquery.com) for most current version) and our local **js/controls.js** file where we will be doing all of our JavaScript work later in the `initializePlayer()` function called when the page finishes loading (via `onload` in our `<body>` tag).

In Brackets click on the lightning bolt in the upper right corner of the window (Live Preview). 

Answer OK to any dialog that pops up and a Chrome browser window will launch showing your web page in its current state. It should look like the following right now:

## Needlegun Breechloader - 1866



0:00 / 1:30



0:00 / 0:00

Now let's begin working on our CSS by looking at the provided **controls.css** file.

Let's start with the provided code in [controls.css](#):

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

body {
  background-color: #222;      dark background for our video to sit on
  color: #fff;                white text color for our dark background
  padding: 0;                 browser resets for padding and margin...
  margin: 0;
}

#wrapper {
  width: 53.75rem /* 860px */ Set width for our page...
  margin: 0 auto; /* center our div#wrapper on page */
  position: relative; /* positioning parent */
  /*border: 1px solid green;*/
}
```

Use the [@font-face](#) at-rule to define and use an extended font set. In this case we will be using a subset of the fontawesome font set (see [fontawesome.io](#)) that was provided in our project's start files under the fonts folder. The fontawesome font set consists of **icon fonts** *which then can be treated like **text*** in our HTML and CSS. This means we don't need to use images for these symbols which improves our page's performance. Yay!

You can go to [fontawesome.io](#), generate a subset of fonts that you want to use in your app, and download them for use. We'll see later that we can access the symbol we wish from the font set using a Unicode character.

Then set up some styles for our page and our wrapper div.

Ok, time to get to stylin' the rest of our content in the **controls.css** file.

We are going to set up a fairly sophisticated text effect on our **h2** at the top of the page by animating its *text shadow*. We will be using CSS *keyframe animation* to do this. Be sure to save and test the effect of *each* style attribute as you add them.

```
h2 {  
  font-family: Impact, 'Trebuchet MS', Arial;  
  font-size: 2.6em;  
  font-weight: normal;           make letters less bold (Impact is very bold)  
  margin-bottom: 0.2em;  
  letter-spacing: 0.1em;        space out the letters a bit more  
  text-align: center;  
  opacity: 0.8;  
  color: #222;                  make same color as background – stay tuned...
```

The following text shadow will use layers to create a glow effect around the text to make it visible once again.

```
text-shadow: -1px -1px 0px rgba(255, 255, 255, 0.3),  
             0 -2px 6px rgba(0, 0, 0, 0.8),  
             1px 0 6px rgba(255, 255, 255, 0.4),  
             -2px -8px 20px rgba(255, 255, 255, 0.4);
```

```
animation: smoke 10s 1 forwards;
```

Set up a CSS keyframe animation named “smoke” that will run for 10 seconds, repeat once, and once ended, will retain the values it ended with (via the keyword *forwards*). The CSS animation attribute is a shorthand for several animation properties. See the [MDN animation CSS attribute](#) page for more info on the properties that are available. This animation will have the look of smoke that appears and then disperses as if a breeze is blowing from the left.

We are setting the following properties here:

**animation-name:** set to “smoke”

**animation-duration:** set to 10 seconds

**animation-iteration-count:** set to just one time here

**animation-fill-mode:** set to “forwards” which means our h2 will retain the computed text shadow values set by the last keyframe in our keyframe animation (coming up shortly).



The **h2** at the top of the page now should look like this:

## Needlegun Breechloader - 1866

Let's keep going... Time for our CSS keyframes animation code – go *get the following code between the {}'s from provided file **smokeAnimationSettings.txt** from E360.*

```
@keyframes smoke {  
  
  0% {text-shadow: -1px -1px 0px rgba(255, 255, 255, 0.3),  
                0px -2px 0px rgba(0, 0, 0, 0.8),  
                1px 0px 6px rgba(255, 255, 255, 0.4);}   
  
  30% {text-shadow: -1px -1px 0px rgba(255, 255, 255, 0.3),  
                0px -2px 0px rgba(0, 0, 0, 0.8),  
                1px 0px 6px rgba(255, 255, 255, 0.8),  
                0px -8px 26px rgba(255, 255, 255, 0.6),  
                2px -20px 30px rgba(255, 255, 255, 0.4);}   
  
  50% {text-shadow: -1px -1px 0px rgba(255, 255, 255, 0.3),  
                0px -2px 0px rgba(0, 0, 0, 0.8),  
                1px 0px 6px rgba(255, 255, 255, 0.4),  
                -1px -4px 20px rgba(255, 255, 255, 0.2),  
                40px -35px 28px rgba(255, 255, 255, 0.3);}   
  
  75% {text-shadow: -1px -1px 0px rgba(255, 255, 255, 0.3),  
                0px -2px 0px rgba(0, 0, 0, 0.8),  
                1px 0px 6px rgba(255, 255, 255, 0.4),  
                -1px -4px 20px rgba(255, 255, 255, 0.2),  
                55px -35px 45px rgba(255, 255, 255, 0.1);}   
  
  85% {text-shadow: -1px -1px 0px rgba(255, 255, 255, 0.3),  
                0px -2px 0px rgba(0, 0, 0, 0.8),  
                1px 0px 6px rgba(255, 255, 255, 0.2),  
                -1px -4px 20px rgba(255, 255, 255, 0.1),  
                65px -35px 45px rgba(255, 255, 255, 0.1);}   
  
  100% {text-shadow: -1px -1px 0px rgba(255, 255, 255, 0.3),  
                0px -2px 0px rgba(0, 0, 0, 0.8),  
                1px 0px 6px rgba(255, 255, 255, 0.4),  
                -1px -4px 20px rgba(255, 255, 255, 0),  
                70px -35px 55px rgba(255, 255, 255, 0);}   
  
}
```

Whoa, that is a lot of typing! Fortunately for you, I've provided the above CSS keyframe animation code in your E360 lesson. It is called **smokeAnimationSettings.txt**. Download it now, copy its code, and paste it between your **smoke @keyframes** at-rule's curly braces.

What this code is doing is setting up a CSS *keyframes* animation which allows you to make changes to whatever CSS properties you are animating (here our h2's text-shadow property) some percentage of the way along the animation's timeline (0% is at beginning of the animation, 20% of the way, etc..., up to 100% which is at the end of the animation). Keyframe is an animation term that describes a frame in an animation's timeline where the object is changed in some way.

**Note:** I made one change I like better in the provided keyframe animation code. I changed the 75% to 51% so that part runs very quickly after arriving at halfway through the animation. You can play around with the values too to see if you come up with something you like better!

## Onwards and upwards!

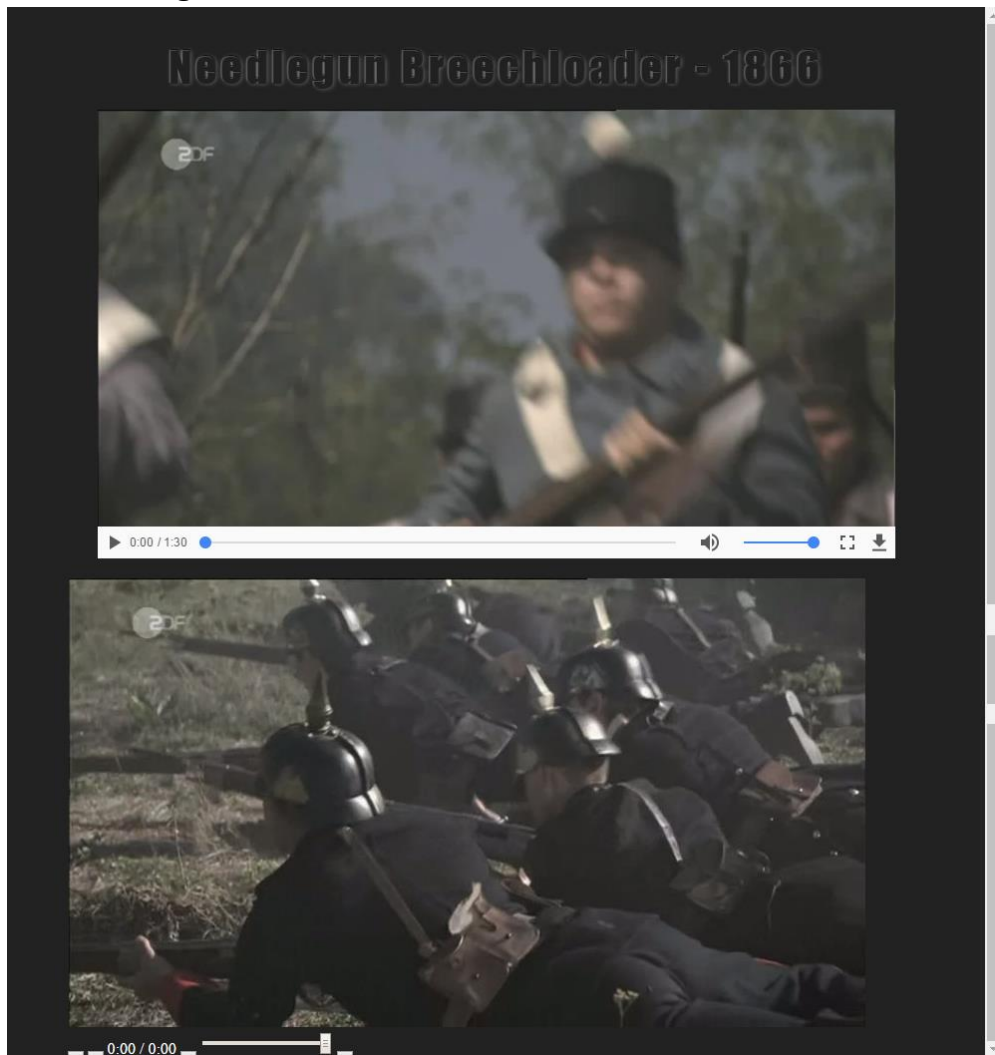
Select the **<div>** that contains our video element, set its width to match the video and give it a bit of padding on top. Then use margin to center it within its *div#wrapper* container and separate it from the next element that will be below it by 1em.

```
#aboveTheControls {  
  width: 800px;  
  padding: 0.6em 0 0;  
  margin: 0 auto 1em;  
}
```

Select the **<img>** tag containing the image we want to use as a "poster" cover for our video. Otherwise, the video will show its first frame (which may be pixelated in the case of an mp4). Absolutely position it relative to the edges of its closest positioning parent (*div#wrapper* because of setting position: relative on it). Move its top edge down 4.3em's below *div#wrapper*'s top edge and its left edge 1.9em's to the right of *div#wrapper*'s right edge.

```
#videoCover {  
  position: absolute;  
  top: 4.3em;  
  left: 1.9em;  
}
```

Screenshot showing prior to repositioning the video cover image (notice, cover image is below the video which is showing its first frame and default browser controls at bottom of video).



And after repositioning the video cover image so it covers the video...



Remember, you should be looking anything up you don't fully understand (or emailing to ask me) and saving and testing in your browser as you go!

Select the elements with the class “shadowEffect”, make them **positioning parents** so we can absolutely position any of their descendant elements *relative to the positioning parent's edges*.

```
.shadowEffect {  
  position: relative; /* positioning parent */  
}
```

We are going to add **::before** and **::after** children to these elements. **::before** creates a **first child** element on the element(s) selected before the **::**. **::after** creates a **last child** element on it. These children elements are not in your .html file and they require a CSS **content** attribute to be set on them even if you set **content's** value to the empty string (like we are doing below).

These two new children will be styled to create drop shadows at each lower corner of our video making it look like its two bottom corners are curling up off the page a bit.

```
.shadowEffect::before,  
.shadowEffect::after {  
  content: ""; /* empty string as these will be used for shadows */  
  position: absolute; /* position edges relative to edges of .shadowEffect tags */  
  bottom: 15px; /* bottom edge 15px up from bottom edge of posParent */  
  left: 10px; /* left edge 10px right of left edge of posParent */  
  top: 80%; /* top edge 80% down from top edge of posParent */  
  z-index: -1; /* make sure it is behind its posParent so it is peeking out */  
  max-width: 300px;  
  width: 50%; /* width will be half the width of its parent (.shadowEffect tags) */  
  background: #111;  
  box-shadow: 0 18px 10px #111; /* this will create the shadow – vertical with blur */  
  transform: rotate(-3deg); /* will make rightmost shadow look funny – is fixed soon */  
}
```

Now, select just the **::after** child to set some CSS properties on that child only

```
.shadowEffect::after {  
  right: 10px; /* pin this child 10px left of posParent's right edge */  
  left: auto; /* let browser set left edge position based on width */  
  transform: rotate(3deg); /* fix rotation of rightmost shadow */  
}
```

Here is what that shadow looks like before and after rightmost shadow fix noted above:



Now, with both shadows in place:



Ok, let's keep going...

Select the `video` (this would actually select all video tags it finds on our page), make it a *positioning parent* and set a **CSS transition** on it such that if any (via the `all` value) of our video element's CSS attribute values *change*, the change should take *two seconds* to occur with a CSS easing value of *'ease'* applied. This creates a simple animation if any of our video's CSS attribute values get changed.

If we wanted a transition animation on only the video's opacity attribute we would code it like this in video's {}'s below: `transition: opacity 3s;`. Now, if the opacity of our video is changed at any time (including by JavaScript code), the change will take 3 seconds to occur creating a simple fade animation. What if you wanted to place a transition on more than one of a selected element's CSS attributes to have an animation on them if their values change?

Like this for opacity and width: `transition: opacity 3s, width 1.5s ease-in-out .2s;` `opacity` changes occur over 3 seconds, while any change to `width` would take 1.5 seconds with an easing value of *ease-in-out* and a delay of two-tenths of a second before the transition begins (transition-delay property).

```
video {  
    position: relative;      /* makes our video a positioning parent */  
    transition: all 2s ease; /* changing any attribute value takes 2s to occur */  
}
```

Select `div#controls` which is the `<div>` that contains all of our video's controls. It should be 90% the width of its parent element – `div#wrapper` and be centered within it. Notice I left in a commented out border. I like to use 1px wide border's with varying colors to see my elements as I lay them out to see if everything is where it is supposed to be. This helps me in debugging my HTML structure layout and is particularly useful when using positioning and float's. You should remove any commented out code in the production version of your code before submitted it.

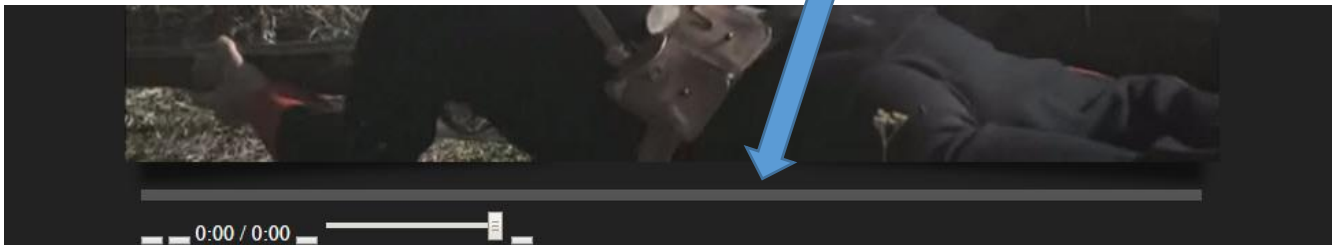
```
#controls {  
    width: 90%;  
    margin: 0 auto;  
    /*border: 1px solid yellow;*/      turn on the border and check it out...  
}
```

## Onward to the styling of our video's progress bar!

Select our `div#progressBar` and set its background color to a dark gray. It will have a short height of 8px. Separate it from any elements below it by setting a margin on the bottom of 6px. Remember that margin is the space outside an element's border that separates the element from an adjacent element.

```
#progressBar {  
  background-color: #555;  
  height: 8px;  
  margin-bottom: 6px;  
}
```

It should look this now below our video:



Select the `<span>` tag that has the id of “**played**”. It is a child of `div#progressBar` and is empty in the HTML. Why would that be? Well, we are going to do something with its “content” through styles. We are going to set its *height*, make it *block* rather than inline, round its top and bottom right corners a bit and give it a nice blue color that fits our theme.

Oh, and we are going to use *relative positioning* to nudge it down just 1px from where it would normally appear so we still see a 1px line of gray on top of it and below it from its parent, the `#progressBar <div>`. We won't see this yet as its *width* will start at **0%**, but don't worry, we'll make it grow later in our JavaScript based on the video's play progress – cool!

```
#played {  
  background-color: #47a3da;  
  position: relative;  
  top: 1px;          /* move its top edge down 1px from where it would normally appear */  
  width: 0%;         /* we won't see our #played span yet because of this */  
  height: 6px;       /* 2px less than the height of its parent – div#progressBar */  
  display: block;    /* span tags are normally display: inline */  
  border-top-right-radius: 20%;  
  border-bottom-right-radius: 20%;  
}
```



## Next up..., the rest of our video's controls of course!

Select `div#lowerControls` which contains the rest of our video's control elements. In this `<div>` we'll simply set a font on the text (which will only affect our time values) and set the font size – *yawn...* But don't go to sleep yet, our icon fonts are on the way!

```
#lowerControls {
  font-family: 'Trebuchet MS', Arial;
  font-size: 0.9em;
}
```

Select all of the **button** and **input** elements on our page and set their outline attribute to a value 'none' which removes any outline that might be showing. This only affects certain browsers (ahem, older versions of IE for one) that would show outlines on form elements by default – not cool!

```
button, input {
  outline: none;
}
```

Ok, let's style our first control that will use an icon font – our `#playPause button` element (notice this button tag has a class of `"playBtn"` as well which we'll be using in a bit).

```
#playPause {
  position: relative;
  left: 0.2em;           /* nudge it to the right just a bit */
  background: none;
  border: none;
  padding: 0;
  cursor: default;      /* reset cursor */
  margin-right: 1em;     /* put some space between this and button to its right */
}
```

Wait, what? Where did our play/pause button go (hint: look at your browser now)? Well, by removing its `background` and `border` we took away what HTML usually uses to show a button. It's ok though. We'll be styling in a `::before` child on this button in a moment that will set up its icon.



Select our `#playPause button` element and add a *before child pseudo-element* to it as its first child. We'll use the `'fontawesome'` font that we included earlier via `@font-face`. It allows us to use icons that are treated like text and it has icons people are used to seeing for audio/video players such as play, pause, stop, mute, and full-screen icons.

```
#playPause::before {
  font-family: 'fontawesome'; /* fontawesome icon font family is, well, "awesome" */
  font-size: 1.4em;          /* May need to tweak this value to match it other controls */
  color: #47a3da;             /* our nice blue color again */
  font-style: normal;
  font-weight: normal;
}
```

But I don't see anything yet you say. Patience young Skywalker<sup>©</sup> Did you notice anything missing from the `::before` pseudo-element above that I stated earlier every before and after child MUST have? Yes, good job, that's right – a `content` attribute. We don't see anything yet because we haven't given our new child any content!

Select our play button element this time using its class `"playBtn"` and set its content to the Unicode characters for a play button icon in the `'fontawesome'` font set.

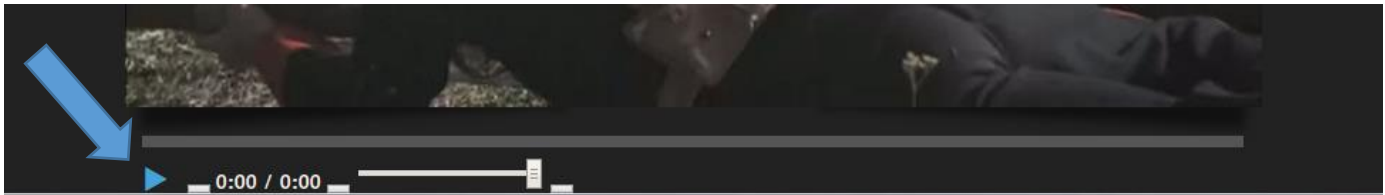
```
.playBtn::before {
  content: "\f04b"; /* get Unicode from the play button icon's page at fontawesom.io */
}
```

Check out these changes in your browser and you should see our play button. But, why did we select our button using its class this time? Great question! This button needs to toggle between being a **play** and **pause** button.

We are about to set up alternative content for this button (fontawesome's Unicode for a **pause** button icon) on a different class of `"pauseBtn"`. We'll toggle the button's look later in JavaScript when the user clicks/taps the button. We will need to remove its current class and add the alternative class. That way the CSS is already in place to show the right fontawesome icon based on what class our button element currently has.

For example, when the button is the play icon and the user clicks it, the video will begin to play and the button (through our JavaScript wizardry) will automatically have its class changed from `"playBtn"` to `"pauseBtn"` so the icon automagically changes to be the pause icon.

Here is what the bottom of our page looks like now – notice the play button?



Ok, let's select the before child of element(s) with a class of "pauseBtn" so we can set its content. Remember, our #playPause button doesn't have this class at the moment, but it will in the future through JavaScript.

```
.pauseBtn::before {  
  content: "\f04c";      /* fontawesome Unicode for a pause icon */  
}
```

Add a hover effect on the button's before child that flips the color to an off-white. Notice the :hover has to go on the selected element just prior to ::before. After doing this you should notice the play button now changes color on hover. Test it out!

```
#playPause:hover::before {  
  color: #ddd;  
}
```

Next we'll set up our other video controls and many of them will be very similar to the code we just did for our #playPause button.

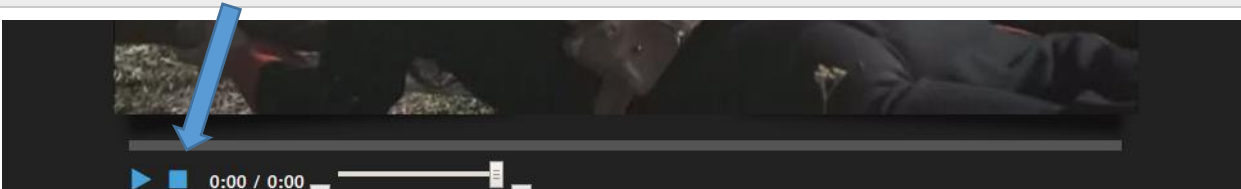
Next up, the Stop Button..

So, you are going to notice that the styling of our `#stopButton` element is going to be very similar to the play/pause button. It just won't need to be a toggle button so no classes will be needed. Notice that again, we won't see the button once we set `background` and `border` to 'none', but it will come back into view when we add and set up its `::before` child with a fontawesome font icon.

```
#stopButton {
  position: relative;
  top: -0.08em;           /* nudge it up a bit to match up with play button */
  left: 0;
  background: none;
  border: none;           /* button no longer visible – for now */
  padding: 0;
  cursor: default;
  margin-right: 1em;      /* set some space separation to its right */
}

#stopButton::before {
  content: '\f04d';       /* fontawesome Unicode for a stop button icon */
  font-family: 'fontawesome';
  font-size: 1.2em;
  color: #47a3da;
  font-style: normal;
  font-weight: normal;
}

#stopButton:hover::before { /* on hover change color to off-white */
  color: #ddd;
}
```



The *time* text is already styled and looks ok where it is, so let's jump to the *mute/unmute* button. Now this will be one of those toggle buttons like our *play/pause* button. Clicking it will toggle between the *mute/unmute* states. So, its styles will look very much like what we did with the *play/pause* button with `classes` used to set the content.

This will allow us to get the button icon we want by simply toggling the class on the element via JavaScript code. Let's get it done!

Select our `#mute button` element and set its styles very much like we did on our `#playPause` element. If needed, refer back to that part of this document to read about the details of why we are doing what you see below.

```
#mute {  
  float: right;           /* move it to the right edge of its container element */  
  position: relative;  
  top: -.25em;           /* nudge it up a bit to align with other controls */  
  background: none;  
  border: none;          /* make button invisible for now */  
  padding: 0;  
  cursor: default;  
  margin-right: .4em;     /* give it a bit of space by nudging it back to the left */  
}
```

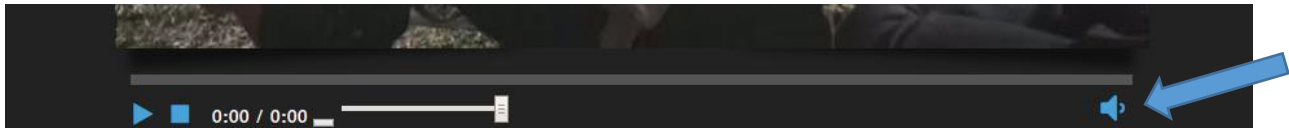
Ok, now add a `::before` child again for this button's icon font. Set its size to roughly match the other controls, and make it blue for our theme. Again, we are not setting its fontawesome Unicode content value here as we need to provide *two possible content values* depending on what *version* (via the `class` value) of the button we should display – **mute** vs **unmute**.

```
#mute::before {  
  font-family: 'fontawesome';  
  font-size: 2.2em;  
  color: #47a3da;  
  font-style: normal;  
  font-weight: normal;  
}
```

```
.mute::before {  
  content: "\f027";       /* fontawesome Unicode value for mute button icon */  
}
```

```
.unmute::before {  
  content: "\f026";       /* fontawesome Unicode value for unmute button icon */  
}
```

```
#mute:hover::before {    /* same type of hover effect the other icons have */  
  color: #ddd;  
}
```

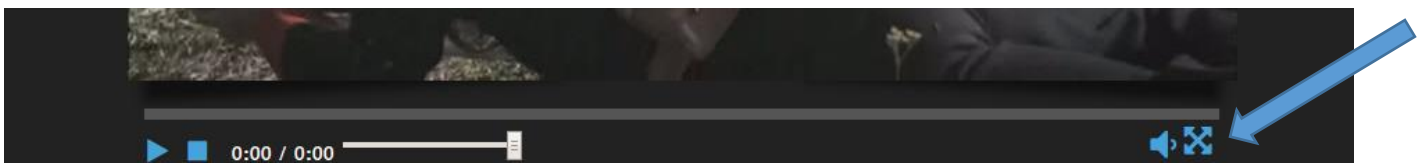


## Full Screen time!

Select our `#fullScreen` button. These will be the same type of styles as we saw for our `#stopButton`. I won't comment them much here so refer back to previous code for hints and practice looking up attributes to find out what they do. Or, cut them, see what changes when you save and refresh (or in Brackets Live Preview), then paste them back and watch the change.

```
#fullScreen {  
  float: right; /* is to the right of the mute button as it comes before it in the HTML */  
  position: relative;  
  top: -0.15em;  
  background: none;  
  border: none;  
  padding: 0;  
  cursor: default;  
  margin-right: .4em;  
}  
  
#fullScreen::before {  
  content: '\f0b2'; /* fontawesome Unicode value for full screen button icon */  
  font-family: 'fontawesome';  
  font-size: 1.7em;  
  color: #47a3da;  
  font-style: normal;  
  font-weight: normal;  
}  
  
#fullScreen:hover::before {  
  color: #ddd;  
}
```

Ok, here is what you should be seeing now:



The CSS finish line is in sight and you can make it! Let's style the volume bar. We'll put it between the mute and full screen buttons. Here we go.

Select our `#volumeSlider` `input` element. This HTML form element looks like this in our `.html` file:

```
<input id="volumeSlider" class="sliderBar" type="range" min="0" max="1" value="1" step="0.1">
```

The `type="range"` makes it a range slider where the slider thumb, as it is called, can slide between a range of values from 0 to 1 and it is initially set to its max value of 1. As the user slides the slider thumb it changes by .1 (one-tenth) as it moves between the min (0) and max (1) values.

```
#volumeSlider {  
  float: right; /* will be to the right of the mute button as it comes before it in the HTML */  
  position: relative;  
  top: .6em;  
  width: 12%; /* set width that looks good */  
  margin-right: 1em; /* place space between it and the full screen button */  
}
```

Select all `input` tags with a `type` attribute that is set to a value of `"range"`. Of course, for us, that gets our one `range slider` element...

The first attribute we'll use, `appearance`, indicates if we want to display form elements (in this case our range slider) using `platform-native` (browser is platform) styling based on the underlying operating system's theme. We don't want this so we'll set its value to `"none"`. This tells the browser **not** to use its native styling on this element. Instead, we'll style its look ourselves. *Note:* this doesn't play well with IE prior to Edge.

Be sure to test each of these as you go so you can see the changes take place as you style it.

```
input[type='range'] {  
  -webkit-appearance: none; /* browser prefix for Chrome and Safari support – Moz is good */  
  appearance: none; /* turn off default platform-native styling */  
  background: #000; /* make background a bit darker than page background */  
  border: 1px solid #666; /* put a thin medium gray border around the range slider */  
  height: 4px;  
  border-radius: 34%; /* round the corners for a kinda-cool look */  
}
```

One last thing: let's change the thumb slider.

Select our range slider input element's **slider thumb** which is the little rectangular shape you click onto and drag. We are going to make it look like a circular button instead through styles. Notice we need to use **browser prefixes** (-webkit-, -moz-, and -ms-) as most of the browsers do not yet support this as a standard (slider-thumb).

-webkit- is the browser prefix for Chrome and Safari browsers

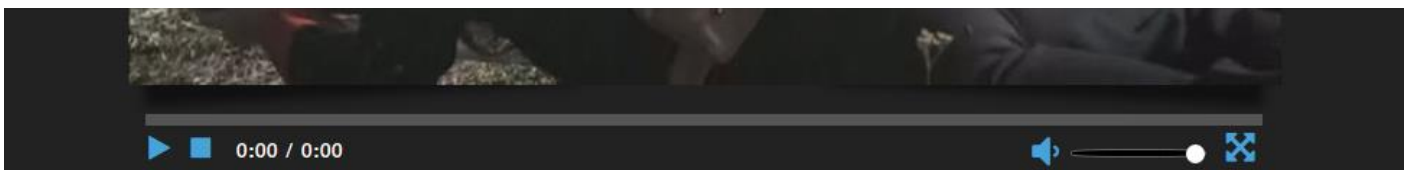
-moz- is for Mozilla Firefox

-ms- is for IE and Edge although it doesn't work some (well most) of the time.

```
input[type='range']::-webkit-slider-thumb,  
-moz-range-thumb,  
-ms-thumb {  
  -webkit-appearance: none;  
  -moz-appearance: none;  
  -ms-appearance: none;  
  appearance: none;      /* don't use its native style appearance */  
  background: #fff;      /* make it a 13px by 13px white circle */  
  height: 13px;  
  width: 13px;  
  border-radius: 50%;  
}
```

We now should have a custom range slider volume bar. Of course, it doesn't affect the actual video volume yet. Details, details... We'll set that up when we tie our JavaScript code to it.

Now, here is what we should have at this point for our controls:



Here is what our page should look like with the finished CSS:

# Needlegun Breechloader - 1866



0:00 / 1:30

Ok, next we work on our **JavaScript**. We will be building a **controls.js** file which will be using the HTML5 **Media API** (Application Programming Interface) that was added to JavaScript to support media elements like **audio** and **video**.

This will be done via a video tutorial consisting of [video lessons for creating our video-related JavaScript](#) in **controls.js**.