**Examining the Important Features**

Medium has a pretty interesting tooltip. Let's analyze what we're building and how we can achieve it.

**We'll cover the following**

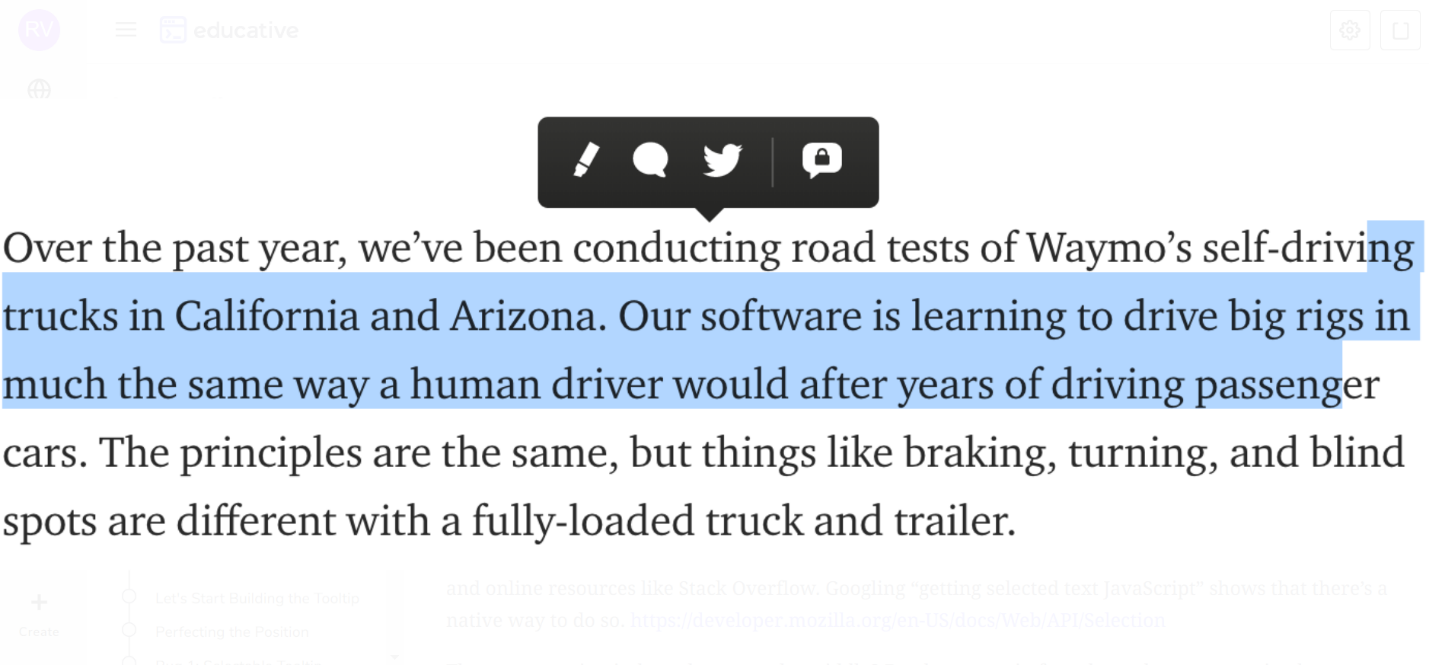
* + [Determining scope](https://www.educative.io/courses/intermediate-javascript/gx8GZWB7pO6#determining-scope)
  + [Searching for possible solutions](https://www.educative.io/courses/intermediate-javascript/gx8GZWB7pO6#searching-for-possible-solutions)
  + [Testing and exploring](https://www.educative.io/courses/intermediate-javascript/gx8GZWB7pO6#testing-and-exploring)

**Determining scope**[#](https://www.educative.io/courses/intermediate-javascript/gx8GZWB7pO6#determining-scope)

As always, we’ll begin by scoping the work ahead. It’s an imperfect process, and something will almost always be missed. It’s important that as we’re designing our solution, we can ask ourselves whether it’ll work for x, y, and z. If you have an example you’re building off of, play around with it and try to break it or see how it behaves given a variety of user interactions. To move past this stage, you shouldn’t have an unanswered, “how do they do that?” You don’t have to know how *it* is implemented, but you should know *a way* to implement it.

**If you’re stuck, being more specific about what’s happening can often help.**

* When text is highlighted, a tooltip shows up.
* When text is highlighted, an HTML element is displayed in the middle of the highlighted text.
* When I click down on some text, drag my mouse over some text, and release the mouse, an HTML element is added to the page positioned over the middle of the text that was dragged over.
* When I click down on text in the div of the article, drag my mouse over some text, and release the mouse, an HTML element is added to the page centered slightly above the middle of the selected text. The exception is when the selection spans multiple lines, in which case it’s positioned over the middle of the first line.



## Searching for possible solutions [#](https://www.educative.io/courses/intermediate-javascript/gx8GZWB7pO6#searching-for-possible-solutions)

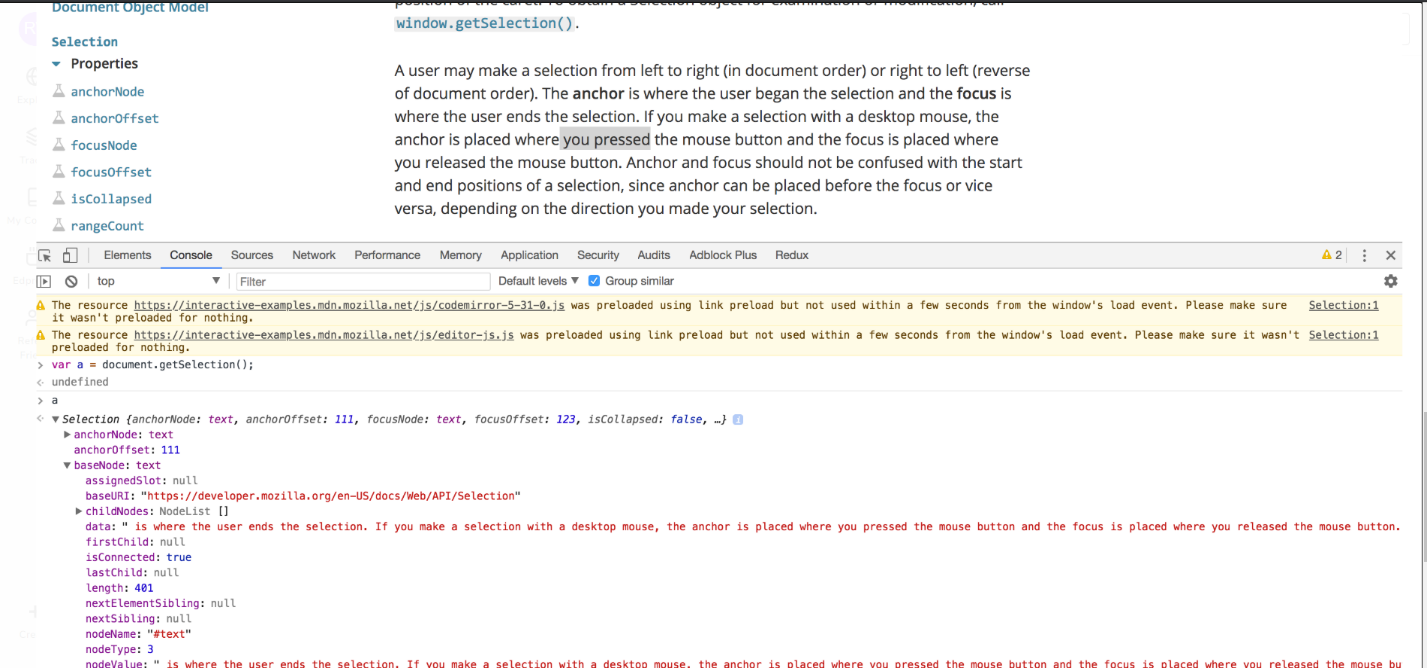
At this point, we should ask ourselves how can we reliably get the text? Tracking the position of the mouse when it was first clicked to where it was released and figuring out which text was selected based on that path seems tricky. As a web developer, you should liberally rely on searching documentation and online resources like Stack Overflow. Googling “getting selected text JavaScript” shows that there’s a native way to do so. <https://developer.mozilla.org/en-US/docs/Web/API/Selection>

The next question is, how do we get the middle? Do characters in fonts have the same spacing between each one such that I can multiply the number of characters selected with some constant and divide by two?

Googling “get middle of selected text JavaScript” yields nothing useful this time around, unfortunately (or maybe fortunately, since we get a teachable moment from it). Okay, let’s see. Googling “character width of fonts”, it appears only “monospaced” fonts have the attribute that each character is the same width. Since these are articles with aesthetically pleasing typography, we’re out of luck there.

## Testing and exploring [#](https://www.educative.io/courses/intermediate-javascript/gx8GZWB7pO6#testing-and-exploring)

Let’s see what the selection api gives us. The nice thing about web development is that trying out some code on a whim is extremely easy to do. Open up the console, and there’s your REPL (interface to evaluate expressions).



Hmm, the value doesn’t seem to be quite what we selected. I wonder if it’s thrown off at all by the fact that we defocus the selection when we focus on the console. We can write a simple script to test that.

1

2

3

setInterval(() => {

  console.log(document.getSelection());

}, 1000);

Putting this in an HTML will have the current selection printed to the console every second. After running the test, I see that the act of running the earlier test didn’t interfere with the result – getSelection gives us all the text in the element that the selection took place. Of course, the documentation tells us the same thing, but you shouldn’t hesitate to run quick and dirty tests to confirm or deny assumptions about some JavaScript code you’re unfamiliar with!

So the library gives us the two HTML nodes (node has roughly the same meaning as an element) that the selection begins and ends at, and the character offset into each node. I’m not convinced that helps us, since again, we don’t know the exact spacing of each character selected.

We’ll continue exploring this problem in the next lesson.

**How to Determine the Middle of Cursor?**

One of the most interesting aspects was that the tooltip appears in the middle of our cursor. How can we do that? Let's find out in this lesson.

**We'll cover the following**

* + [What we have so far](https://www.educative.io/courses/intermediate-javascript/YMlO4rW66k2#what-we-have-so-far)
  + [Exploring solutions](https://www.educative.io/courses/intermediate-javascript/YMlO4rW66k2#exploring-solutions)
  + [Employing devtools](https://www.educative.io/courses/intermediate-javascript/YMlO4rW66k2#employing-devtools)

**What we have so far**[#](https://www.educative.io/courses/intermediate-javascript/YMlO4rW66k2#what-we-have-so-far)

So far, we’ve investigated if we can use constant character widths to determine the middle, and we’ve tried to use the getSelection method alone to provide all the information we need.

We know that we’ll certainly have to use *some* information from getSelection, since it reliably gives us the selected characters. So the problem becomes this: given a string of characters in an HTML node, find the x and y offsets of the middle (so that we can have absolute position for the tooltip with top and left properties).

We can divide this into subproblems. If we can find the x and y offsets of *any* HTML element, one idea is to wrap the selected text in a span, get the x and y offsets and the width of that span, and consider the middle to be (x + width/2, y).

Googling “get x and y of html node,” I find the API getBoundingClientRect(), which has the properties we’re looking for. I’m not entirely sure how to determine if the selection spans multiple lines. Maybe we can use the height to determine that. We’ll see.

Let’s write some code to test if this method works.

I’m using console.log to print often, and continuously test if my intermediary steps work, like getting the selected text.

document.onmouseup = () => {

  const selection = document.getSelection();

  console.log(selection);

  const anchorNode = selection.anchorNode;

  const focusNode = selection.focusNode;

  if (anchorNode != focusNode) {

    // Cross-paragraph selection

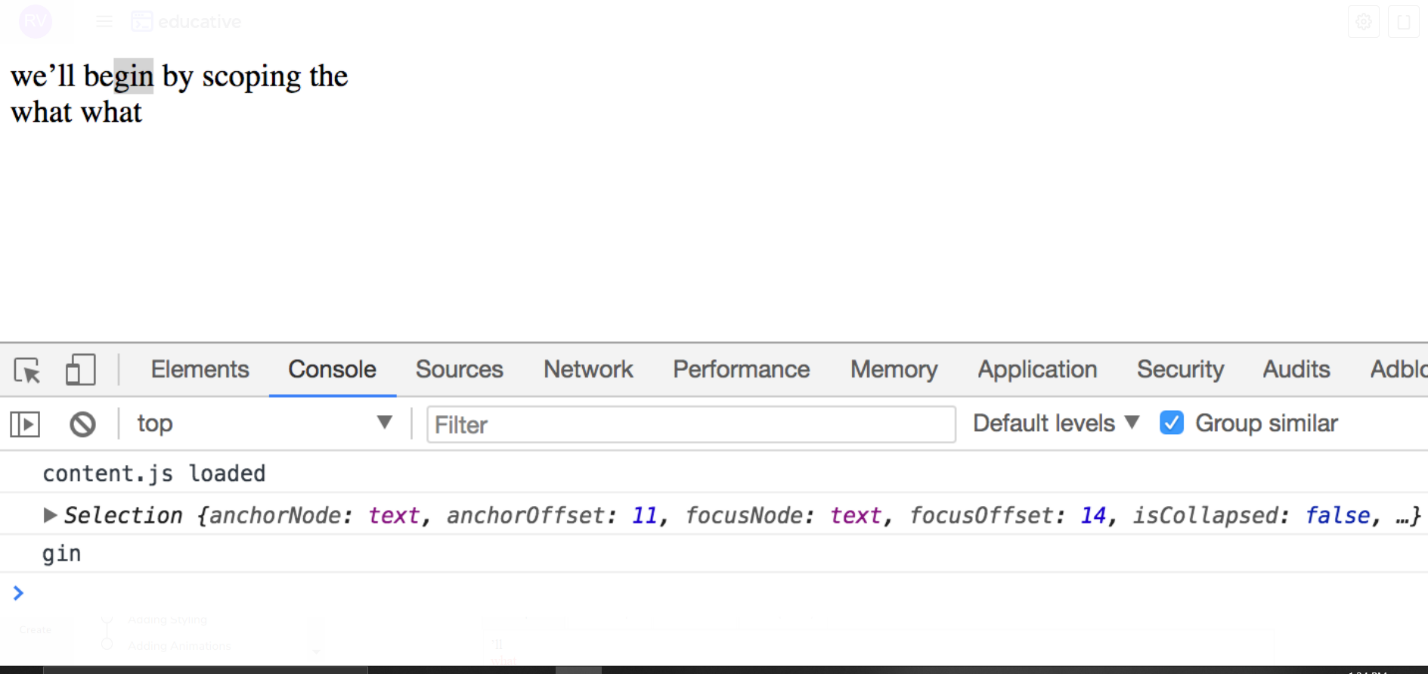
    return;

  }

  const selectedText = anchorNode.data.substring(selection.anchorOffset, selection.focusOffset);

  console.log(selectedText);

}



The result is this:

Html page:

<!DOCTYPE html>

<html>

<head>

</head>

<body>

  <div>

  we’ll begin by scoping the

  </div>

  <div>

    what what

  </div>

</body>

</html>

Css page:

.highlighted {

  color: red;

}

Js page:

document.onmouseup = () => {

  const selection = document.getSelection();

  console.log(selection);

  const anchorNode = selection.anchorNode;

  const focusNode = selection.focusNode;

  if (anchorNode != focusNode) {

    // Cross-paragraph selection

    return;

  }

  const selectedText = anchorNode.data.substring(selection.anchorOffset, selection.focusOffset);

  let withSelectionSpanned = anchorNode.data.substring(0, selection.anchorOffset);

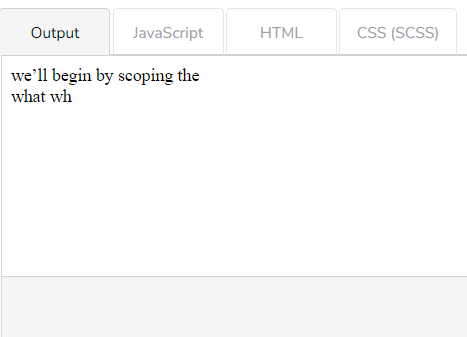
  withSelectionSpanned += ("<span class='highlighted'>" + selectedText + "</span>");

  withSelectionSpanned += anchorNode.data.substring(selection.focusOffset, anchorNode.data.length);

  anchorNode.parentElement.innerHTML = withSelectionSpanned;

}

Output:



Try highlighting text. Beware, it only works once! If you want to play with it more, switching tabs (between, for example, the JavaScript and output) will cause a refresh.

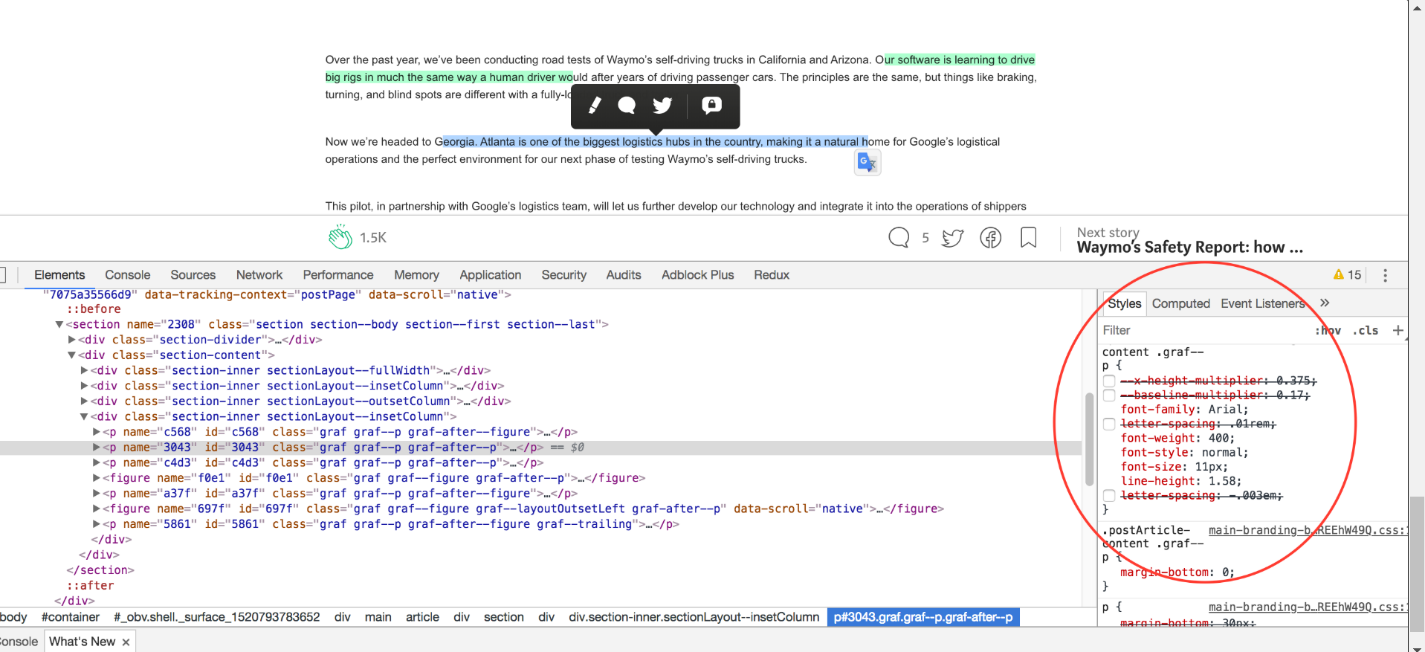
## Exploring solutions [#](https://www.educative.io/courses/intermediate-javascript/YMlO4rW66k2#exploring-solutions)

I applied a CSS property to turn the class red for some visual confirmation of what I’m doing. The big issue now is that replacing innerHTML seems to remove my highlighted text. That’s not acceptable. Maybe JavaScript can force a re-highlight of the original selection? Googling around, I can’t find anything. Now we want to make sure that what Medium is doing isn’t just applying a background color to appear highlighted. If that were the case, highlighting some text and doing a copy-paste operation should not work, yet it does on Medium.

At a higher level, it makes sense that innerHTML replacement removes my original highlight. The text that was highlighted is technically no more (even if the text output looks the same). They’re not the same object. Inserting spans, however, means modifying the HTML. There doesn’t seem to be a way to reconcile that the text highlight is maintained properly given this method.

## Employing devtools [#](https://www.educative.io/courses/intermediate-javascript/YMlO4rW66k2#employing-devtools)

I’m curious if Medium is just doing character width estimations to get the middle, so I go into the devtools again and start removing properties related to font. I change the font-family to Arial, change the font size and character spacing. The tooltip seems to be able to adapt.



# Alternative Approach to find the Middle

The previous approach didn't work, so we'll try to use a different approach to find the middle of the cursor.

###### We'll cover the following

* + [The range property](https://www.educative.io/courses/intermediate-javascript/gxnRy9wKQ8G#the-range-property)
  + [Trying out a solution](https://www.educative.io/courses/intermediate-javascript/gxnRy9wKQ8G#trying-out-a-solution)

## The range property [#](https://www.educative.io/courses/intermediate-javascript/gxnRy9wKQ8G#the-range-property)

Our previous attempt of wrapping the text we want in a span and getting the position of the span didn’t work. Time to Google for another lead. Searching for “get position of text”, I find a range property which I saw in the getSelection API but hadn’t paid attention to.

15

document.onmouseup = () => {

  const selection = document.getSelection();

  console.log(selection);

  const anchorNode = selection.anchorNode;

  const focusNode = selection.focusNode;

  if (anchorNode != focusNode) {

    // Cross-paragraph selection

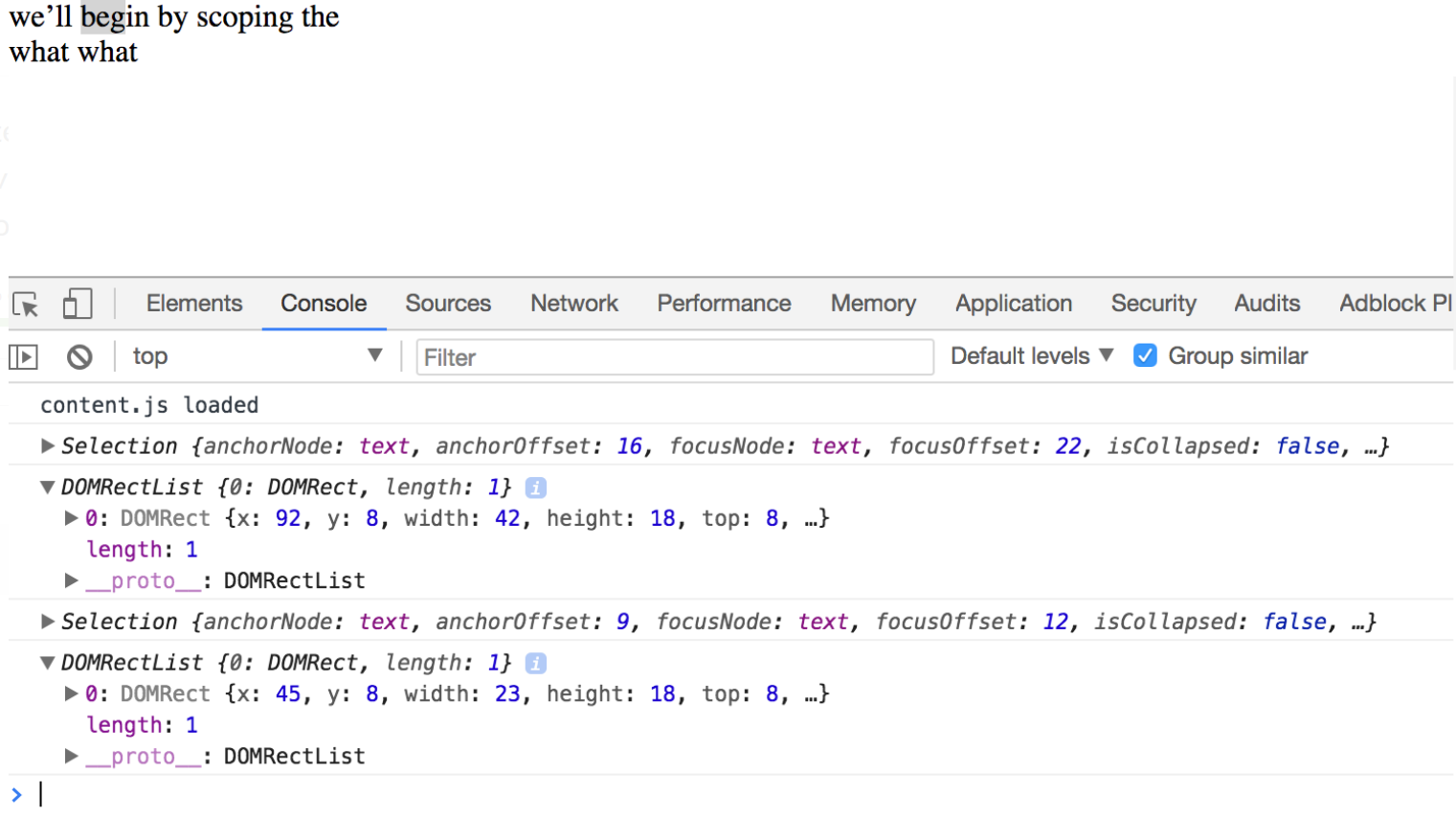
    return;

  }

  const selectedText = anchorNode.data.substring(selection.anchorOffset, selection.focusOffset);

  console.log(selection.getRangeAt(0).getClientRects());

}



Ah, this is exactly what we need! Sometimes we do extra work trying to use the tools we’re familiar with. Part of being an experienced frontend engineer is just knowing what exists. As you progress, you gain valuable intuition about what’s possible and what’s impossible, what tasks are straightforward, and which are minefields (ahem- CSS animations), which solutions degrade the user experience and which solutions are cheap on CPU, and then some. This doesn’t just apply to frontend development, by the way. Experienced engineers often choose the best solutions partly due to just being **aware** of so many other solutions and their outcomes.

Web development is unique in that it’s constantly evolving at a pace much quicker compared to a lot of domains in software engineering like databases or iOS. The “right way” to do things in web development changes rapidly. Frameworks trend and fade on the order of a small number of years (whereas popular databases stay the same for decades). The number of open-source libraries that could potentially fit what you’re doing is vast and is growing every day. Even JavaScript itself is iterating drastically in features. A senior JavaScript developer who’s been in a coma since 2014 would hardly recognize web code written today.

But that’s alright, because if they have the foundation, they can adapt quickly. Frameworks, libraries, and even syntax can change, but keep digging into the source code, and it often distills to more or less of the same. Whether you’re calling render() in React, or $(.class).html() in jQuery, they rely on JavaScript primitives like document.getElementsByClassName and innerHTML. The constant change can be a point of contention and scare newcomers away, but we should embrace the change because the improvements are in our favor. There are smart people evaluating changes made by smart people. Change for the sake of change will be disregarded, and change that empowers developers more will be embraced. The former you will never have to hear about, and the latter you won’t regret learning. I don’t doubt that in subsequent years, there will continue to be changes made to popular JavaScript that allow us to do more with less, paradigm shifts that allow codebases to more sustainably be worked on in large teams, tooling that helps how we catch bugs before they occur, and other improvements.

As you’ve seen in this lesson, there’s no need to memorize exactly how everything is done. Non-programmers looking in would be surprised at the lack of implementation detail we carry in our working memory. Far more important than memorizing the syntax for what you need to do is having a vague awareness of what’s possible and how to Google for what you need.

Anyway, we’ve just discovered range. Speaking of which, there it is in the React (the most popular framework as of this writing) source code! <https://github.com/facebook/react/blob/24f824250fde6418569222f6e33b35ba9c1f1f46/packages/react-dom/src/client/ReactDOMSelection.js#L190>

## Trying out a solution [#](https://www.educative.io/courses/intermediate-javascript/gxnRy9wKQ8G#trying-out-a-solution)

Let’s add a div and absolute position it given the information provided by range. Google how to create a div in JavaScript

document.onmouseup = () => {

  const selection = document.getSelection();

  console.log(selection);

  const anchorNode = selection.anchorNode;

  const focusNode = selection.focusNode;

  if (anchorNode != focusNode) {

    // Cross-paragraph selection

    return;

  }

  const selectedText = anchorNode.data.substring(selection.anchorOffset, selection.focusOffset);

  const rangeRect = selection.getRangeAt(0).getClientRects()[0];

  const dot = document.createElement("div");

  dot.style.width = "5px";

  dot.style.height = "5px";

  dot.style.background = "red";

  dot.style.position = "absolute";

  // Middle

  dot.style.left = `${rangeRect.x + (rangeRect.width/2)}px`;

  dot.style.top = `${rangeRect.y}px`;

  document.body.appendChild(dot);

}

SAVERESET

Try highlighting text. We’ve finally found the middle!.

# Let's Start Building the Tooltip

Now that we've found a way to get the middle, let's get a skeleton of a tooltip to display. In this lesson, we will code a basic layout of a tooltip in HTML which we will further extend in the upcoming lessons.

###### We'll cover the following

* + [SVG or PNG?](https://www.educative.io/courses/intermediate-javascript/xV9vlmDPJAE#svg-or-png)
  + [Rendering the tooltip](https://www.educative.io/courses/intermediate-javascript/xV9vlmDPJAE#rendering-the-tooltip)

## SVG or PNG? [#](https://www.educative.io/courses/intermediate-javascript/xV9vlmDPJAE#svg-or-png)

At the end of the last lesson, we found the middle of some text we highlight. There’s still work to be done, like getting the middle of the first line when multiple are selected, and only having the event respond to selections (as of now it responds to clicks too), but I think the hard part is over. In a similar fashion, if you’re rapidly iterating on a product and you’re coding towards some work-in-progress designs, it would be wise to get 80% of the way there in CSS. Chances are, the last 20% of making things pixel perfect take disproportionately longer, and your efforts will be doubled when the design inevitably changes.

So, let’s make 80% of the tooltip.

We won’t spend too much time on styling. It’s a rectangle with a dark background and four icons separated by a thin divider. For the icons, they seem to use SVG over PNG.

SVG tells the browser how to draw something, whereas PNG tells the browser where to put pixels. There are some tradeoffs between the two, like how SVG takes more computing power but doesn’t lose sharpness when increased in size. Which one should you use? Often, the answer is just whichever format you can get your icons in.

## Rendering the tooltip [#](https://www.educative.io/courses/intermediate-javascript/xV9vlmDPJAE#rendering-the-tooltip)

<!DOCTYPE html>

<html>

<head>

</head>

<body>

  <div class="tooltip">

    <svg class="tooltip\_\_icon">

      <path d="M19.074 21.117c-1.244 0-2.432-.37-3.532-1.096a7.792 7.792 0 0 1-.703-.52c-.77.21-1.57.32-2.38.32-4.67 0-8.46-3.5-8.46-7.8C4 7.7 7.79 4.2 12.46 4.2c4.662 0 8.457 3.5 8.457 7.803 0 2.058-.85 3.984-2.403 5.448.023.17.06.35.118.55.192.69.537 1.38 1.026 2.04.15.21.172.48.058.7a.686.686 0 0 1-.613.38h-.03z" fill-rule="evenodd"></path>

    </svg>

    <svg class="tooltip\_\_icon">

      <path d="M19.074 21.117c-1.244 0-2.432-.37-3.532-1.096a7.792 7.792 0 0 1-.703-.52c-.77.21-1.57.32-2.38.32-4.67 0-8.46-3.5-8.46-7.8C4 7.7 7.79 4.2 12.46 4.2c4.662 0 8.457 3.5 8.457 7.803 0 2.058-.85 3.984-2.403 5.448.023.17.06.35.118.55.192.69.537 1.38 1.026 2.04.15.21.172.48.058.7a.686.686 0 0 1-.613.38h-.03z" fill-rule="evenodd"></path>

    </svg>

    <svg class="tooltip\_\_icon">

      <path d="M19.074 21.117c-1.244 0-2.432-.37-3.532-1.096a7.792 7.792 0 0 1-.703-.52c-.77.21-1.57.32-2.38.32-4.67 0-8.46-3.5-8.46-7.8C4 7.7 7.79 4.2 12.46 4.2c4.662 0 8.457 3.5 8.457 7.803 0 2.058-.85 3.984-2.403 5.448.023.17.06.35.118.55.192.69.537 1.38 1.026 2.04.15.21.172.48.058.7a.686.686 0 0 1-.613.38h-.03z" fill-rule="evenodd"></path>

    </svg>

    <svg class="tooltip\_\_icon">

      <path d="M19.074 21.117c-1.244 0-2.432-.37-3.532-1.096a7.792 7.792 0 0 1-.703-.52c-.77.21-1.57.32-2.38.32-4.67 0-8.46-3.5-8.46-7.8C4 7.7 7.79 4.2 12.46 4.2c4.662 0 8.457 3.5 8.457 7.803 0 2.058-.85 3.984-2.403 5.448.023.17.06.35.118.55.192.69.537 1.38 1.026 2.04.15.21.172.48.058.7a.686.686 0 0 1-.613.38h-.03z" fill-rule="evenodd"></path>

    </svg>

  </div>

  <div>

  we’ll begin by scoping the

  </div>

  <div>

    what what

  </div>

</body>

</html>

Css page:

.tooltip {

  width: 10rem;

  height: 2rem;

  background: black;

  text-align: center;

}

.tooltip\_\_icon {

  width: 1.8rem;

  height: 2rem;

  fill: white;

  padding: 0.1rem;

  text-align: center;

}

.tooltip\_\_icon:last-child {

  border-left: 1px solid white;

}

The work is 80% there! There’s some padding, color, and miscellaneous things (rounded corners, that triangle thing at the bottom), but we can add that at the end. We need something resembling the finished product to iterate on the functionality visually.

# Perfecting the Position

As you can see, the Tooltip that currently appears needs to be repositioned. In this lesson, we will try to reposition it to get a neater look!

###### We'll cover the following

* + [Moving the tooltip to Javascript](https://www.educative.io/courses/intermediate-javascript/Bn8PMllro6Q#moving-the-tooltip-to-javascript)
  + [Avoiding multiple tooltips](https://www.educative.io/courses/intermediate-javascript/Bn8PMllro6Q#avoiding-multiple-tooltips)
  + [Adding the arrow end](https://www.educative.io/courses/intermediate-javascript/Bn8PMllro6Q#adding-the-arrow-end)

## Moving the tooltip to Javascript [#](https://www.educative.io/courses/intermediate-javascript/Bn8PMllro6Q#moving-the-tooltip-to-javascript)

Let’s move the tooltip generation into JavaScript and have it replace the dot that we were testing with.