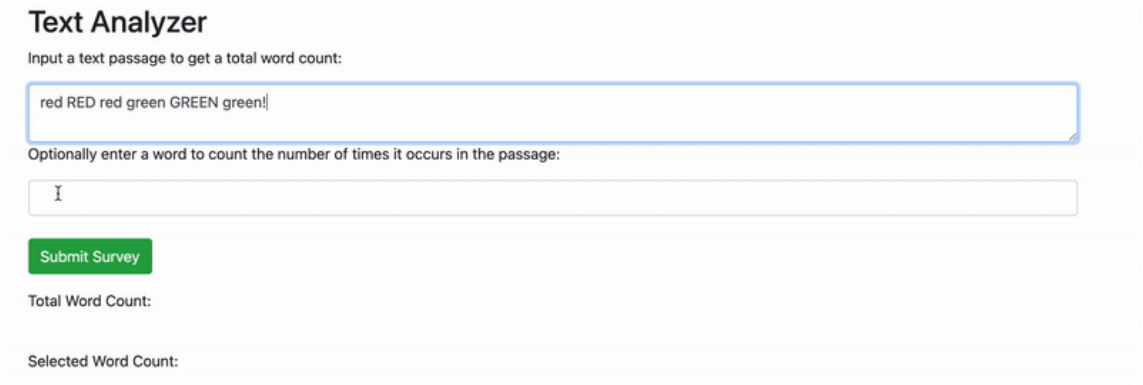
Lesson | Tuesday

Introduction to Programming (/introduction-to-programming) / Arrays and Looping (/introduction-to-programming/arrays-and-looping) / Separation of Logic: Fixing a Bug in Text Analyzer

Text | Cheat sheet

We have a bug in our Text Analyzer application. But what exactly is it? You'll find it if you test the application out in the browser. This GIF below demonstrates exactly what is happening. In this lesson, we'll solve the bug, but we'll also do it while keeping our UI and business logic separated. At the end of the lesson, we'll look at two examples of poorly separated logic.



Fixing the Bug While Keeping Logic Separated

If we enter a text passage into the first form field and a word into the second form field, everything works correctly. However, if we don't enter anything in the second form field, our application states that the selected word count is the same as the total word count. But how is that possible? It should be zero — or better yet, the selected word count shouldn't show at all if it's not entered.

Considering the latter option, couldn't we just hide the selected word count and only show it if a word is entered in the second form field?

Well, sure, we *could*. And at least from a user perspective, there'd no longer be a bug. But that would be like sweeping dust under the rug so no one else will see it. It's still there. And it could eventually morph into a terrifying dust monster from another dimension if it's left there. Okay, that's not technically true, but the point is, we can't just hide bugs. We need to fix them.

This also gives us a perfect opportunity to revisit some of the debugging techniques we covered in the last course section. Remember those? We are especially referring to the techniques in Debugging in JavaScript: Using debugger and Breakpoints (https://www.learnhowtoprogram.com/introduction-to-programming/javascript-and-web-browsers/debugging-in-javascript-using-debugger-and-breakpoints).

So let's solve this by adding a breakpoint! We'll review the techniques covered in the lesson linked above because they are absolutely essential tools for debugging your code. Consider this a refresher. Whether or not you are feeling good about using breakpoints yet, they are so important that it's worth covering them again.

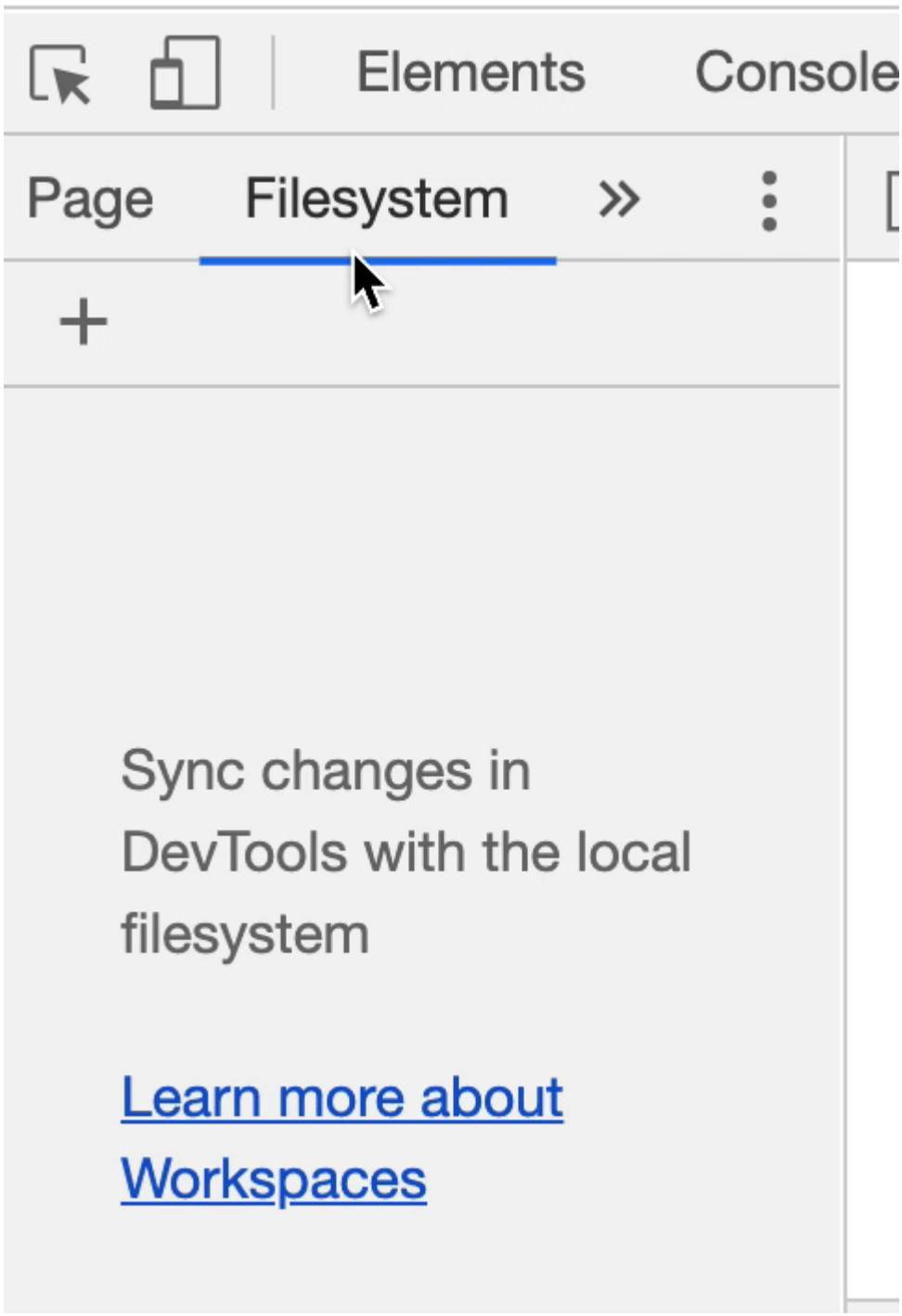
Finding the Bug

At this point, we don't know what's causing the error. It seems like we've tested our functions pretty thoroughly, so maybe it's coming from the UI. However, it's still possible we missed something.

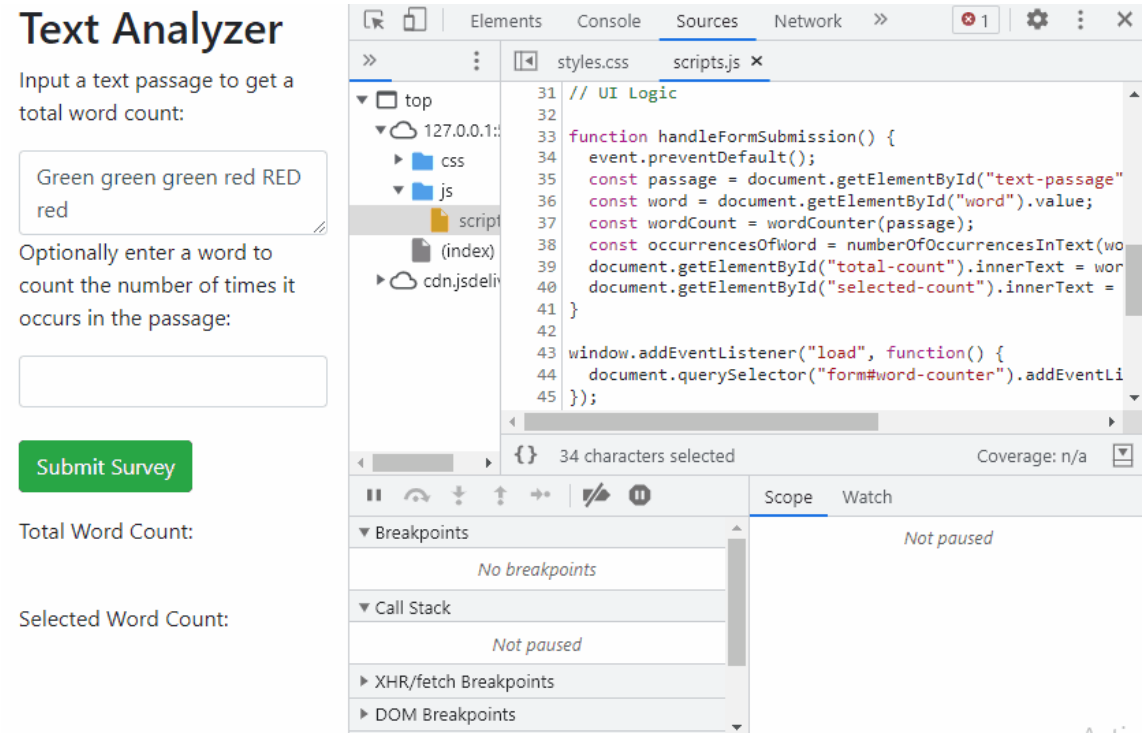
Start by opening the DevTools console if you haven't already. We always want to keep an eye on the console when we are testing our code in the browser.

If we load our page, there aren't any errors, which means there's nothing obvious just yet.

Next, click on the *Sources* tab. If you dragged `index.html` into your browser to open the application, your `scripts.js` code should already be showing. If not, you may need to manually add the code by clicking *Filesystem* in the left pane of *Sources* and then clicking the + icon as the GIF below shows.



Once we have our code open, we need to insert a breakpoint. The GIF below demonstrates the whole debugging process using a breakpoint. We'll also explain it in detail after the GIF and you should walk through the process on your own as well.



First, we click on the line of code where we want the code to pause. Since the error is occurring in the line where we'd expect to see `numberOfOccurrencesInText()` correctly rendered, let's start by verifying that our `numberOfOccurrencesInText()` function is working correctly. That means we will add the breakpoint to the line right *after* the line `const occurrencesOfWord = numberOfOccurrencesInText(word, passage);`. This is line 39 in the GIF, though it might be a slightly different line number in your code.

Next, we manually submit the form again. Our code will execute and when it reaches the breakpoint, it will pause. The Sources tab will also show us the result of each line of code, which is very helpful. For instance, we can see the value of word from the input is `""`. We can also see that `occurrencesOfWord` is equal to 6. That's not at all what we want.

In other words, the call was coming from inside the house all along! It's not the UI — the bug is coming from the function `numberOfOccurrencesInText()` that we've already tested.

In the GIF above, we then move to the Console tab and manually test the function ourselves with different arguments, but still passing in an empty string for the `word` parameter.

Specifically, we do the following:

```
numberOfOccurrencesInText("", "hi");
```

The result is `1`, which isn't correct. It's often very helpful to try out code within the scope of the breakpoint. Remember, you can just switch over to the console and check the value of any functions or variables that exist in that scope when your code is paused. And if they don't exist when you expected they would, well, you've got a scope problem. While the values in the Sources tab generally give you a lot of information, we still recommend using the console to test your code regularly within breakpoints.

This particular problem isn't a scope problem, though. We have a bug in our `numberOfOccurrencesInText()` function. As a review, here's the code for the `numberOfOccurrencesInText()` function:

```
function numberOfOccurrencesInText(word, text) {
  const textArray = text.split(" ");
  let wordCount = 0;
  textArray.forEach(function(element) {
    if (element.toLowerCase().includes(word.toLowerCase()))
{
      wordCount++;
    }
  });
  return wordCount;
}
```

When we originally used TDD to write this function, we wrote a test that checks if the text passage is an empty string but we never tested what would happen if the word itself is an empty string. That's one mistake we made. However, even if we had written that test early on, it would've broken later. That's because we introduced a new bug when we refactored our function to use `String.prototype.includes()`. Check this out:

```
"hi".includes("");
true
```

It turns out that when we use `String.prototype.includes()`, *every* string includes `""`. A really tiny bug, right? Well, look how quickly it came back to bite us. It's easy to forget a simple test like this. In fact, we just did.

If we were using Jest and we'd already written a test for this, Jest would automatically find and notify us of the failed test. As we'll discover in a few course sections, automated testing is a very powerful tool. However, it only works as well as the tests we write, which means we need to work on good test-writing practices now with our pseudocode tests.

Fixing the Bug

So let's fix this issue. Here are two possible fixes. Which one do you think is better?

- Update our UI so that a user can't pass in an empty string or otherwise update the UI so the `numberOfOccurrencesInText()` function won't be called for an empty string.
- Update our function so it can handle an empty string input for a word.

If you guessed the second option, you are correct!

Remember, our goal right now is to keep our logic separate. Our function should be robust on its own. If we tried to fix the problem by addressing the UI, our function would get more dependent on external logic (in this case, code that checks the user input in the UI) — and that is not good separation.

There's certainly nothing wrong with doing *both* things. For instance, we could *also* do something in the UI as well. But they are still two completely different things. On the one hand, we want a robust function that always correctly returns the number of occurrences of a word. On the other, we want to notify a user if they don't input a word. Technically, our function could handle both things but it shouldn't. That would mean our code isn't properly separated.

So what's the next step?

No, it's not time to fix the code yet. *First*, we need to write a test. Always. No matter how small the fix.

Here's the test. (This should be the last test in the **Describe** block for `numberOfOccurrencesInText()`).

```
Test: "If an empty string is passed in as a word, it should
return 0."
Code:
const word = "";
const text = "red RED Red!";
numberOfOccurrencesInText(word, text);
Expected Output: 0
```

The fix is simple:

js/scripts.js

```
function numberOfOccurrencesInText(word, text) {
  if (word.trim().length === 0) {
    return 0;
  }
  const textArray = text.split(" ");
  let wordCount = 0;
  textArray.forEach(function(element) {
    if (element.toLowerCase().includes(word.toLowerCase()))
{
      wordCount++;
    }
  });
  return wordCount;
}
```

We add a conditional to check if the `word` parameter has 0 characters in it after being trimmed, and if so, return 0:

js/scripts.js

```
...
if (word.trim().length === 0) {
  return 0;
}
...
```

Remember, we can use `String.prototype.trim()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/trim) to trim all whitespace from both ends of a string. Since the string is all whitespace, that will reduce it to `""`, which has a length of `0`.

So our application has a functioning UI now and we've also fixed a bug. However, take careful note of the fact that we still kept business and UI logic separate in the process.

Examples of Bad Separation of Logic

Example 1: Accessing or Updating the DOM from your Business Logic

Let's say that we wanted to return a message to the user in the UI if they don't enter a word. We aren't actually going to do this — we just want to illustrate two examples where logic isn't properly separated. One of these examples is very obvious while the other is less so.

Here's the obviously bad example of mixing UI and business logic in our function:

```
// This is bad! Don't put any logic to alter the DOM in you
r function!

function numberOfOccurrencesInText(word, text) {
  if (word.trim().length === 0) {
    // I'm directly altering the DOM from my business logi
c! This is bad.
    document.getElementById("total-count").innerText = 0;
  }
  const textArray = text.split(" ");
  let wordCount = 0;
  textArray.forEach(function(element) {
    if (element.toLowerCase().includes(word.toLowerCase()))
{
      wordCount++;
    }
  });
  // I'm directly altering the DOM from my business logic!
This is bad.
  document.getElementById("total-count").innerText = wordCo
unt;
}
```

This example should be really obvious. Instead of returning the number of occurrences, the function doesn't return anything at all. It directly alters the DOM — which should be part of our user interface logic. Whenever you see a function altering the DOM in business logic, it's time to refactor. You shouldn't be doing it in the first place.

Again, if you have code that directly accesses or alters the DOM in any way, keep it out of your business logic. That's a job for the UI logic.

Example 2: Including a Message to the User in the Values Returned from Business Logic Functions

Now let's look at a slightly less obvious example.

```
// Not so good either, but for less obvious reasons.

function numberOfOccurrencesInText(word, text) {
  if (word.trim().length === 0) {
    return "You need to enter a word!";
  }
  const textArray = text.split(" ");
  let wordCount = 0;
  textArray.forEach(function(element) {
    if (element.toLowerCase().includes(word.toLowerCase()))
{
      wordCount++;
    }
  });
  return "There are " + wordCount + " total matches!";
}
```

Here there's no alteration of the DOM. However, our function still has a problem with separation of logic. How so? Well, it's really nice when our function *just* returns the number of occurrences of a word in a text. It shouldn't care about anything else. That includes providing a message for a user. A message like `"You need to enter`

`a word!"` really belongs in the user interface because it's a specific message for a user. If you want to return these kinds of messages, put them in the UI logic instead.

Here's a silly example to hammer home the point. What if `"xyz".split("")` didn't return the array `["x", "y", "z"]`, but instead returned the following message:

```
"Hooray. You've successfully split the string into three el
ements:  ['x', 'y', 'z']!"
```

`String.prototype.split()` would go from being one of the most useful methods in JavaScript to being absolutely worthless. Don't do the same to your own code!

In the next lesson, we'll give one more example of UI logic versus business logic. This will hopefully really drive home the difference between the two.

Previous (/introduction-to-programming/arrays-and-looping/separation-of-logic-adding-a-ui-to-text-analyzer)
Next (/introduction-to-programming/arrays-and-looping/separation-of-concerns-in-text-analyzer-boldpassage-ui-function)

Lesson 29 of 50
Last updated February 28, 2023

disable dark mode

(http://www.epicodus.com)