Formatting the Autocomplete Suggestions

Let's style the autocomplete suggestions in this lesson to make them look more like Google's.

In this section, we'll address any outstanding functionality that relies on JavaScript.

Outstanding Functionality

- Using a template for the suggestion and auxiliary data.
  - We need to insert it between `<li>` elements, so it conforms to regular HTML inside a `<ul>`, which is where we're inserting these suggestions.
- Determine which sections need to be bolded
  - Using a comparison between the input value and the given suggestion
- Add classes to elements as necessary for them to change their styles dynamically
  - The buttons look like they merge into the autosuggestions list, so I plan to apply a `--autosuggest` modifier class to the element when the autosuggestion results are shown.

Using a template

Wrapping API results into a template has gotten a lot easier since ES6 introduced template literals. The alternative before this was to just concatenate sections of the template with dynamic variables, either through

`+` or have them in a list to join. For example, `"<div>" + name + "</div>"`. With template literals, we just use `${variable}` anywhere in our template string.

```
function createSuggestionElement({suggestion, auxiliaryData}) {
  const auxiliaryString = auxiliaryData ? ` - ${auxiliaryData}` : "";
  return `<li class="search__suggestions__list__result">${suggestion}${auxiliaryString}</li>`
}

function onSuggestionsResponse(data) {
  const suggestionsElement = document.getElementsByClassName('search__suggestions__list')[0];
  let suggestionsHTML = "";
  for (const suggestion of data) {
    suggestionsHTML += createSuggestionElement({
      suggestion: suggestion.suggestion,
      auxiliaryData: suggestion.auxiliary
    });
  }
  suggestionsElement.innerHTML = suggestionsHTML;
}
```

Making the above changes to our working example gives us:

Output
JavaScript
HTML

Google search   I'm Feeling Lucky

We don't have to stick the variables directly into one big template. We can do

preprocessing, such as only showing the hyphen when the auxiliary data is present.

Mixing HTML and Javascript has actually become a lot more trendy and acceptable recently. JSX is a syntax extension that makes this feel even more natural. It can get messy, but you should try to maintain good code etiquette when possible, which in this case means avoiding having multi-paragraph interpolations, which can get confusing. Split things into functions just as you would when writing a longer function, e.g. `<h1>${renderHeader()}</h2>` .

Determine bolding

We need to bold everything in the results except characters at the beginning of the results that matches the input value. It's an all or nothing match, so if you type `squrea` and the result has `square` , all of it is bolded. Then, we also need to wrap those characters that are bolded in a `<b>` . Auxiliary data is never bolded.

Given those specs, can you write a function that wraps text that's supposed to be bolded in `<b>` ?

```
/**
inputValue: "blah"
suggestion: "blah blah"
return: "blah<b> blah</b>"

inputValue: "blah"
suggestion: "foo"
return: "<b>foo</b>"
*/
function wrapBoldedCharacters({inputValue, suggestion}) {
  // TODO
  return "";
}
```

Give it a try before moving on.

Hopefully you came up with something similar:

Output

Google search   I'm Feeling Lucky

Toggling active classes

Finally, we use JavaScript to perform the common task of adding and removing classes based on conditions.

We'll need to add a class to the div that wraps the buttons when the autosuggest is open, so that we can style it accordingly to look merged.

Output

JavaScript

HTML
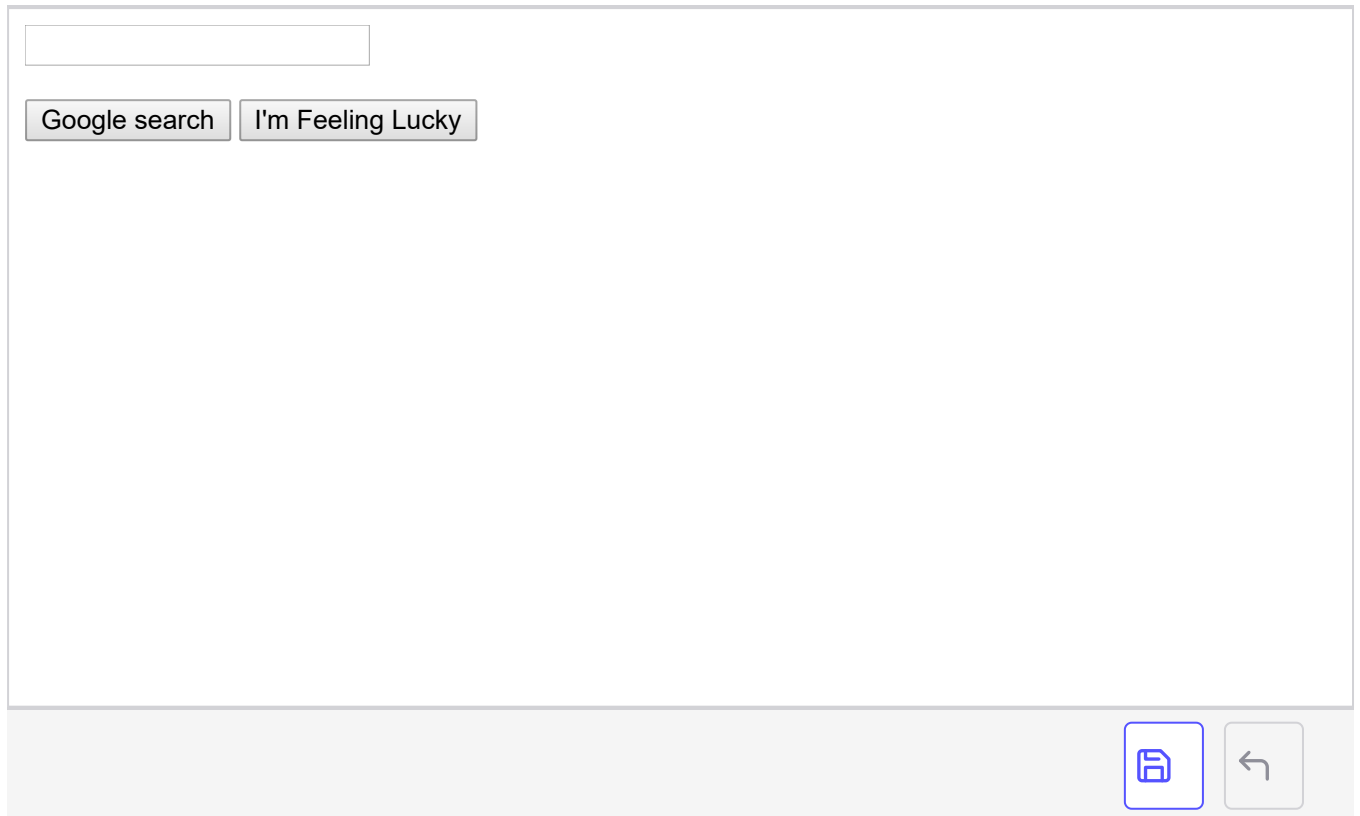
As I was thinking about when to remove the class, I realized I had neglected the case where the autosuggest doesn't show up – when I remove all the characters and when the results come up empty (remember that we set this as past a certain character limit). So part of our changes this time was only to make an API call if the input value is not empty.

The operation of adding and removing classes is straightforward. Just target the element and call `classList.add` or `classList.remove`.

I also did some minor refactoring to put the targeting of the elements at the top level.

That should be it. Let's add the styling and wrap this up!