**Displaying Autocomplete Suggestions**

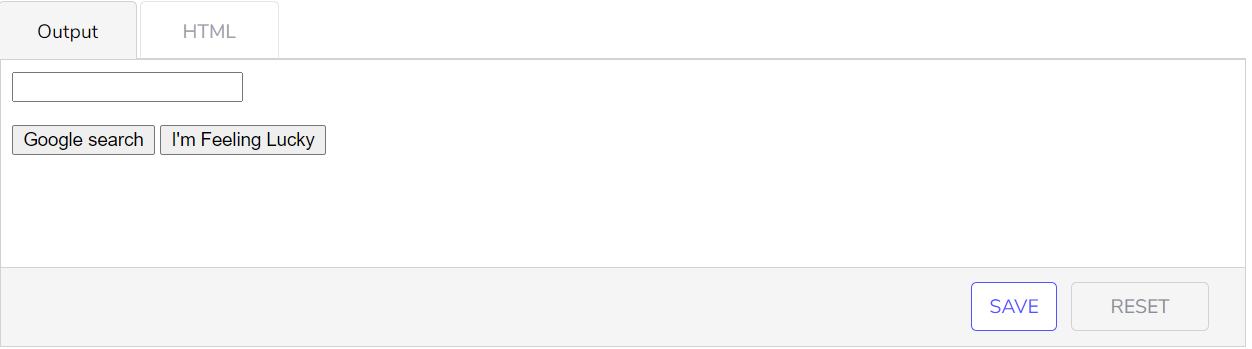
Let's connect the data from the backend and make the suggestions visible.

**We'll cover the following**

* + [Input event listener](https://www.educative.io/courses/intermediate-javascript/gxZAOwDB7pZ#input-event-listener)
  + [Getting server responses](https://www.educative.io/courses/intermediate-javascript/gxZAOwDB7pZ#getting-server-responses)
    - * [Random Integer](https://www.educative.io/courses/intermediate-javascript/gxZAOwDB7pZ#random-integer)
      * [Random String](https://www.educative.io/courses/intermediate-javascript/gxZAOwDB7pZ#random-string)
  + [Extracting text from the response objects](https://www.educative.io/courses/intermediate-javascript/gxZAOwDB7pZ#extracting-text-from-the-response-objects)

**Input event listener**[#](https://www.educative.io/courses/intermediate-javascript/gxZAOwDB7pZ#input-event-listener)

As you’ll recall, we’re working with just a basic input bar and a ul inside a div to show the list of results.



<!DOCTYPE html>

<html>

<head>

</head>

<body>

  <div class="search">

    <div class="search\_\_bar">

      <input type="text" class="search\_\_bar\_\_input" />

    </div>

    <div class="search\_\_suggestions">

      <ul class="search\_\_suggestions\_\_list">

      </ul>

    </div>

    <div class="search\_\_actions">

      <button class="search\_\_actions\_\_action search\_\_actions\_\_action--search">Google search</button>

      <button class="search\_\_actions\_\_action search\_\_actions\_\_action--lucky">I'm Feeling Lucky</button>

    </div>

  </div>

</body>

</html>

Since results show up every time the input changes, we’ll add an event listener on the input element. Whenever it changes, whether that’s a new character, a backspace or a copy-paste, this event listener should read the new value and get new results for us.

Let’s add an event listener to the input and test that it works by just having it echo the results. Usually, I’d have the output be the console (part of the browser devtools mentioned earlier), but for demonstration purposes, writing HTML works too.

We’re just doing basic DOM targeting and editing. I added the <br> as formatting to see more clearly that it’s getting the right values. <br> is like HTML’s newline.

Next, we’ll bring in our server and make an API call on each input.

## Getting server responses

## 

const HOST = 'server.com/';

const searchInput = document.getElementsByClassName('search\_\_bar\_\_input')[0];

function onSuggestionsResponse(data) {

  const suggestionsElement = document.getElementsByClassName('search\_\_suggestions\_\_list')[0];

  suggestionsElement.innerHTML += (data + '<br>');

}

function onNewInput(event) {

  api.get(HOST + 'autocomplete', searchInput.value, onSuggestionsResponse);

}

searchInput.oninput = onNewInput;

// Server

function getRandomString({length}) {

  const characterChoices = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789 ";

  const characters = [];

  while (characters.length < length) {

    const randomIndex = Math.floor(Math.random() \* characterChoices.length);

    characters.push(characterChoices[randomIndex]);

  }

  return characters.join('');

}

function getRandomInteger({min, max}) {

  return Math.floor(Math.random() \* (max - min) + min);

}

function generateSuggestion(prefix) {

  const RATIO\_EXACT\_MATCH = 0.3;

  const RATIO\_AUTOCORRECT = 0.1;

  if (Math.random() < RATIO\_AUTOCORRECT) {

    return getRandomString({ length: getRandomInteger({min: 1, max: prefix.length}) })

  }

  if (Math.random() < RATIO\_EXACT\_MATCH) {

    return prefix;

  }

  return prefix + getRandomString({ length: getRandomInteger({min: 1, max: 10}) })

}

function getAutocompleteHandler(data) {

  const MAX\_CHARS = 10;

  const NUM\_AUTOCOMPLETE\_RESULTS = 10;

  const RATIO\_AUXILIARY\_DATA = 0.1;

  if (data.length > MAX\_CHARS) {

    return [];

  }

  const results = [];

  while (results.length < NUM\_AUTOCOMPLETE\_RESULTS) {

    const suggestion = generateSuggestion(data)

    if (results.find(result => result.suggestion === suggestion)) {

      continue;

    }

    if (Math.random() < RATIO\_AUXILIARY\_DATA) {

      for (let i = 0; i < 2; i++) {

        results.push({

          suggestion,

          auxiliary: getRandomString({ length: getRandomInteger({min: 5, max: 15}) })

        });

      }

    } else {

      results.push({ suggestion, auxiliary: "" });

    }

  }

  return results;

}

const endpoints = {

  "/": {

    "get": () => "hello world"

  },

  "/autocomplete": {

    "get": getAutocompleteHandler

  }

}

// API library

function getFunction(url, data, callback) {

  const domain = url.substring(0, url.indexOf("/"));

  const endpoint = url.substring(url.indexOf("/"), url.length);

  callback(endpoints[endpoint]["get"](data));

}

const api = {

  get: getFunction

};

I’ve copied​ and pasted the server I wrote from the last section, with the implementation of getRandomString and getRandomInteger. The code should hopefully be self-explanatory if you take some time to think about it. It’s not important if you don’t get it- these things you can always just Google. But, for completeness:

#### Random Integer [**#**](https://www.educative.io/courses/intermediate-javascript/gxZAOwDB7pZ#random-integer)

Math.random() gives random decimal from 0 to 1. It is a standard trick to do \*(max-min) + min since that will just map it to our range. For example, if we have x = 0.5, max = 10, and min = 5, then this formula gives 7.5.

#### Random String [**#**](https://www.educative.io/courses/intermediate-javascript/gxZAOwDB7pZ#random-string)

I manually defined a list of choices, create an empty array, and then just push random selections until the length is reached.

From here on out, I’ll hide the server code since it won’t be changing, and I want you to focus on the client code. You can always revisit the above code snippet as needed.

But when we start typing, something unexpected happens! The output does not look like what we want. It’s no doubt a minor issue that you’ll run into again and again.

What do you think happened?

## Ans- We’re printing a JavaScript object whose string representation is “[object Object]” Extracting text from the response objects [#](https://www.educative.io/courses/intermediate-javascript/gxZAOwDB7pZ#extracting-text-from-the-response-objects)

If you look back to the server code, you’ll notice that it’s returning a list of objects, because we need data about the suggestion as well as any auxiliary information. Since we’re printing the response directly, we expected only to get the string representation of a list of objects — nothing mysterious here.

To correct this, we need to specifically target the suggestion inside the object to print.​

After typing for a little, it looks like this works!

Going down the checklist:

* Whenever auxiliary data shows up (non-empty after the -), there are multiple results of the same suggestion.
* There’s a small set of results that are unrelated to our input, which simulates autocorrect.
* Some results are exactly the input.
* We have regular suggestions that have the input prefixed

We’ve gotten the suggestions to show up. In the next lesson, let’s continue tweaking this and making the suggestions look somewhat like the way we want it to. This isn’t all just static CSS though like the dropdown menu – we have to be smart about bolding suggestions​ and doing things like not generating results when there isn’t input.

const HOST = 'server.com/';

const searchInput = document.getElementsByClassName('search\_\_bar\_\_input')[0];

function onSuggestionsResponse(data) {

  const suggestionsElement = document.getElementsByClassName('search\_\_suggestions\_\_list')[0];

  let suggestionsHTML = "";

  for (const suggestion of data) {

    suggestionsHTML += (suggestion.suggestion + ' - ' + suggestion.auxiliary + '<br>');

  }

  suggestionsElement.innerHTML = suggestionsHTML;

}

function onNewInput(event) {

  api.get(HOST + 'autocomplete', searchInput.value, onSuggestionsResponse);

}

searchInput.oninput = onNewInput;

# Formatting the Autocomplete Suggestions

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

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

* + - [Outstanding Functionality](https://www.educative.io/courses/intermediate-javascript/xVovQB3Rknq#outstanding-functionality)
    - [Using a template](https://www.educative.io/courses/intermediate-javascript/xVovQB3Rknq#using-a-template)
    - [Determine bolding](https://www.educative.io/courses/intermediate-javascript/xVovQB3Rknq#determine-bolding)
    - [Toggling active classes](https://www.educative.io/courses/intermediate-javascript/xVovQB3Rknq#toggling-active-classes)

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

### Outstanding Functionality [**#**](https://www.educative.io/courses/intermediate-javascript/xVovQB3Rknq#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 [**#**](https://www.educative.io/courses/intermediate-javascript/xVovQB3Rknq#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:

const HOST = 'server.com/';

const searchInput = document.getElementsByClassName('search\_\_bar\_\_input')[0];

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;

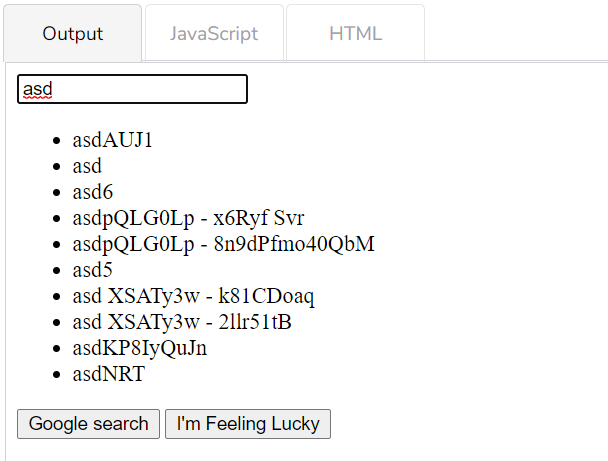
}

function onNewInput(event) {

  api.get(HOST + 'autocomplete', searchInput.value, onSuggestionsResponse);

}

searchInput.oninput = onNewInput;



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 [**#**](https://www.educative.io/courses/intermediate-javascript/xVovQB3Rknq#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 "";

}

After changes being made

const HOST = 'server.com/';

const searchInput = document.getElementsByClassName('search\_\_bar\_\_input')[0];

function wrapBoldedCharacters({inputValue, suggestion}) {

  if (suggestion.startsWith(inputValue)) {

    return `${suggestion.substring(0, inputValue.length)}<b>${suggestion.substring(inputValue.length, suggestion.length)}</b>`;

  }

  return `<b>${suggestion}</b>`;

}

function createSuggestionElement({suggestion, auxiliaryData}) {

  const auxiliaryString = auxiliaryData ? ` - ${auxiliaryData}` : "";

  const boldProcessedSuggestion = wrapBoldedCharacters({

    inputValue: searchInput.value,

    suggestion

  });

  return `<li class="search\_\_suggestions\_\_list\_\_result">${boldProcessedSuggestion}${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;

}

function onNewInput(event) {

  api.get(HOST + 'autocomplete', searchInput.value, onSuggestionsResponse);

}

searchInput.oninput = onNewInput;

### Toggling active classes [**#**](https://www.educative.io/courses/intermediate-javascript/xVovQB3Rknq#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.

const HOST = 'server.com/';

const searchInput = document.getElementsByClassName('search\_\_bar\_\_input')[0];

const suggestionsElement = document.getElementsByClassName('search\_\_suggestions\_\_list')[0];

const actionsElement = document.getElementsByClassName('search\_\_actions')[0];

function wrapBoldedCharacters({inputValue, suggestion}) {

  if (suggestion.startsWith(inputValue)) {

    return `${suggestion.substring(0, inputValue.length)}<b>${suggestion.substring(inputValue.length, suggestion.length)}</b>`;

  }

  return `<b>${suggestion}</b>`;

}

function createSuggestionElement({suggestion, auxiliaryData}) {

  const auxiliaryString = auxiliaryData ? ` - ${auxiliaryData}` : "";

  const boldProcessedSuggestion = wrapBoldedCharacters({

    inputValue: searchInput.value,

    suggestion

  });

  return `<li class="search\_\_suggestions\_\_list\_\_result">${boldProcessedSuggestion}${auxiliaryString}</li>`

}

function onSuggestionsResponse(data) {

  let suggestionsHTML = "";

  for (const suggestion of data) {

    suggestionsHTML += createSuggestionElement({

      suggestion: suggestion.suggestion,

      auxiliaryData: suggestion.auxiliary

    });

  }

  suggestionsElement.innerHTML = suggestionsHTML;

  if (suggestionsHTML) {

    actionsElement.classList.add('search\_\_actions--autosuggest');

  } else {

    actionsElement.classList.remove('search\_\_actions--autosuggest');

  }

}

function onNewInput(event) {

  if (searchInput.value) {

    api.get(HOST + 'autocomplete', searchInput.value, onSuggestionsResponse);

  } else {

    suggestionsElement.innerHTML = '';

    actionsElement.classList.remove('search\_\_actions--autosuggest');

  }

}

searchInput.oninput = onNewInput;

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!