Building a Server Endpoint for Autocomplete

Let's modify the server we have already built so that it returns data that's a little less hardcoded and more varied for a search engine example!

WE'LL COVER THE FOLLOWING

Skeleton for suggestion stubs

The server powering this search engine we're building needs to respond with the actual suggestions. Let's copy the mock server we built in the last lesson and modify it for this case.

```
// Server

const endpoints = {
  "/": {
    "get": () => "hello world"
  }
}

// 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
};
```

We need to add a new endpoint for autocomplete results. Let's call it `/autocomplete`, which will be an API endpoint that just responds with a list of results.

```
function getAutocompleteHandler(data) {
  const results = [];
  for (let i = 0; i < 10; i++) {
    results.push("asdf");
  }
  return results
}

const endpoints = {
  ...
  "/autocomplete": {
    "get": getAutocompleteHandler
  }
}
```

Q Will this work for our development purposes?

COMPLETED 0%

1 of 1

Adding auxiliary information

This server is lacking considering the feature set we're aiming to include in

This server is lacking considering the feature set we're aiming to include in our component. We want to be able to have varied results so that we can test

that the bolding of individual letters is applied correctly and that different inputs lead to different results.

We also have an edge case where multiple results sharing the same string prefix include additional information. We don't want to merely append the detail to the result string (e.g., the raw result is "educative - the company") because that assumes formatting on the front-end, which you'll want to avoid (a client-side design change should not require a back-end change).

```
function getAutocompleteHandler(data) {
  const NUM_AUTOCOMPLETE_RESULTS = 10;
  const results = [];
  for (let i = 0; i < NUM_AUTOCOMPLETE_RESULTS; i++) {
    results.push({
      suggestion: data + "asdf",
      auxillery: "asdf"
    });
  }
  return results;
}
```

Adding some randomness

Take a moment to see the changes. We've satisfied the two limitations of our last implementation, but I think it's still missing something. This implementation is a natural iteration after identifying what we're missing from the first try, so it's not immediately apparent that something is wrong. If we take a moment to anticipate how this will look on the client, I think you'd agree that the data appears too uniform. Everything has the same length and has auxiliary information. We need some entropy in the data as we'd get in a real Google search result.

Also, we're missing the case where the autocomplete result is precisely what is typed along with the case where nothing is returned.

```
function getRandomString({length}) { ... }

function getRandomInteger({min, max}) { ... }

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 (Math.random() < RATIO_AUXILIARY_DATA) {
      results.push({
        suggestion,
        auxillery: getRandomString({ length: getRandomInteger({min: 5, max: 15}) })
      });
    } else {
      results.push({ suggestion, auxillery: "" });
    }
  }
  return results;
}
```

I've defined two functions with the implementation hidden since it's not important.

By the way, I would recommend using this trick for rapid prototyping. You have all these functions you know should exist, but the implementation might not necessarily be important, so you can just define a function signature as a form of note-taking. Proper function signatures should tell developers everything they need to know to use the function.

Changelist

- To simulate the case of the empty set, the server will return an empty list whenever the input exceeds a certain length. This condition feels right since the more you type, the less likely autosuggest will have results for you.
- We define a rate for which an exact match will show up. It shouldn't dominate the list of suggestions, and it doesn't always exist for a given

input, so I've set `RATIO_EXACT_MATCH` to 0.3. For each of the results generated, `Math.random()` will give me a random decimal from 0 to 1, which should be less than `RATIO_EXACT_MATCH` 30% of the time. If it is indeed less than, our suggestion is defined as the data input. Otherwise, append some random string to the end.

- A similar technique is done to simulate the rate of auxiliary data appearing, as well as the autocorrect.

I've intentionally left something out. Let's revisit the edge cases we listed in our scoping phase – can you figure out which isn't satisfied?

Q Which edge case will not be fully tested with this server code?

COMPLETED 0%

1 of 1

Avoiding duplicates

Only the first three cases involve the server, and we've taken into

consideration all three. However, for the first case, the missing piece is that there are “multiple results **of the same string**.” Our code as-is will generate only one suggestion with auxiliary data. We can just add a `for` loop to make it generate multiple results. In fact, the inverse is also unfortunately true: multiple results of the same string can appear without auxiliary data. To fix the inverse, we should skip putting generated suggestions into our results if it’s already included.

```
function getRandomString({length}) { ... }

function getRandomInteger({min, max}) { ... }

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_AUXILLERY_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_AUXILLERY_DATA) {
      for (let i = 0; i < 2; i++) {
        results.push({
          suggestion,
          auxillery: getRandomString({ length: getRandomInteger({min: 5, max: 15}) })
        });
      }
    } else {
      results.push({ suggestion, auxillery: "" });
    }
  }
  return results;
}
```

Changelist

- I added a `results.find` check to see if there's a result with the same suggestion as the one generated. If so, I just restart the loop so it can generate a new result.
- I added a for a loop in the auxiliary data inclusion case so that we have multiple results when auxiliary data is added.

As a side note, our code now won't have data that matches the probabilities we've defined. For example, even though we define exact matches as 0.3, don't expect it to show up 30% of the time since we discard it in certain cases and double it in others. The math isn't important anyway; they're just knobs we can dial as needed if it doesn't give the test data we're looking for.

That should be it for the server portion of our component. It might feel like a lot of work to build out something that isn't even part of our component, but it's work that gets you into the mindset of testing. Even if there's a live server ready to go, you don't want to be making actual network calls in automated tests(for many reasons – slow test-runs, using server resources, etc.).

In addition, when working in a team, there won't always be perfect alignment of dependencies. You should never have to wait on the server portion to be completed before you begin working on a frontend component.