Your mission, should you choose to accept it, is the following:

You'll be building a webapp with a nodejs serverless backend and a react + redux frontend.

1) Create a free AWS account (if you don't already have one)

2) Following this serverless tutorial, add a DynamoDB table and two endpoints to create and retrieve Ingredient objects. You'll be running and testing these endpoints locally using the serverless offline plugin (you can see examples of using the plugin about halfway through in the serverless tutorial).

Your endpoints will be

1. GET /ingredient/{key}
Retrieves an ingredient from the table
Response (status 200) an ingredient object
Parameters

Parameter	Description	Data type	Parameter type
key	Unique ingredient identifier	string	path

2. PUT /add-ingredients
Adds multiple ingredients to the table at once
Response / **Parameters:** up to you how you would like to design this. We'll ask you about the choices that you made here on the follow-up call.

The Ingredients you'll be adding to the table are in this ingredients dataset, which looks like:

```
{
 "abelmosk": {
"text": "abélmösk",
 "tags": {
  "SPICE_HERB": 1
 }
},
 "absinthe wormwood": {
"text": "absinthe wormwood",
 "tags": {
  "SPICE_HERB": 1
 }
},
 "achiote": {
"text": "achiote",
  "tags": {
   "SPICE_HERB": 1
  }
 },
 ...
}
```

Each ingredient in the data consists of a
- a key, like `"abelmosk"`,
- the human readable string “text”, like `"abélmösk"`,
- a set of tags, like `"SPICE_HERB"`.

in other words, each ingredient looks like:

```
"<key>": {
"text": "<text>",
 "tags": {
  "<TAG_1>": 1,
  "<TAG_2>": 1,
  ...
  "<TAG_K>": 1
 }
}
```

where the number of tags *K* varies for each line.

Note: think about how you could design the database to make the tags efficient to query.

3) Create a script which adds all of the ingredients from the ingredients dataset into your database table using the /add-ingredients endpoint, and test that you’re able to retrieve them using /ingredient/{key}.

4) Create a new GET endpoint /fuzzy-search that will retrieve an ingredient based on user input text with fuzzy matching. We’re providing a util library (StringUtils.js) for you to handle the fuzzy matching. To transform the user input text to the ingredient database key, use myKey = myInputText.toHashKey();

For example if a user queries `"abélmosk"`, the output should be:

```
{
"text": "abélmösk",
 "tags": {
  "SPICE_HERB": 1
 }
}
```

The fuzzy matching is done with `"abélmosk".toHashKey() which returns “abelmosk”, which is the key of that ingredient.`

GET /fuzzy-search
Retrieves an ingredient from the table
Response (status 200) an ingredient object
Parameters

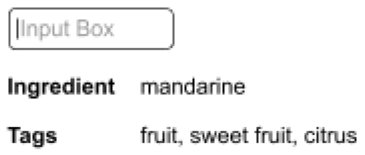
Parameter	Description	Data type	Parameter type
input-text	User input text	string	query

If there is no matching ingredient, the response should be null instead of an Ingredient object.

5) Using React + Redux, create a simple web front-end that will talk to your localhost backend. The frontend will consist of an input field and a matching ingredient view, which will display the ingredient in the database that matches the user input text. The ingredient view will update as the user is typing (you'll be making a new request whenever the onchange event is triggered).

(Note: StringUtils.js should be hidden from the client, i.e. it should be called server side.)

Here's a rough wireframe for the UI with the input field and the ingredient view:



Bonus questions:
5) Add to the web app database update functionalities (allowing a user to add a new entry, with some tags, to the database).

6) Use the myInputText.toHashKey().getVariations() functions from StringUtils.js to get key variations, when the initial query fails. (getVariations handles case like plural vs singular, etc). For example if a user queries `"abelmosks"`, it will fail because of the "s" at the end. getVariations() provides additional keys to handle grammatical variations.

7) Implement a text auto-complete mechanism when a user queries an entry. For example, when a user starts entering `"abel"`, `"abélmösk"` should be recommended as a valid query.

You will be evaluated on:
- quality of the code (20 points)
- documentation of the code, and coding style, according to Airbnb standards for javascript, and Google standards for the rest (10 points)
- the solution you'll choose to solve the problem, error handling, etc:
  question 1) 5 points
  question 2) 25 points
  question 3) 20 points
  question 4) 20 points
Bonus questions:
  question 5) bonus 15 points
  question 6) bonus 5 points
  question 7) bonus 30 points

Deadline: 1 week (i.e. from May 9th to May 16th)
+10 bonus points if sent by this Sunday.
-10 penalty points per late day.

Results: (expected May 16th)
We will go over your results, and challenges you faced, together next week. Also, let us know by the end of the day if you need more time or if you need to shift the project dates.

"As always, should you or any of your IM Force be caught or killed, the Secretary will disavow any knowledge of your actions. This tape will self-destruct in ten seconds. Good luck, agent XXX!"