Build a Rule-based Weather bot

Exercises

- Vanilla: the rule-based weather bot
  In the previous example (Week 1), we learned how to design an interactive interface for our chatbot. In this exercise, we show how implement the rules for the chatbot.

  Notice: since that we implement our ideas with the nodejs, in each section the nodejs command will be explained as well.

  Exe 1 In order to implement a pattern such as the following for our chatbot:

```
pattern: \b(Hi|Hello|Hey)\b
intent: Hello
```

  We should create a directory namely "patterns" in our vanilla project. Afterward, we create an index.js file in the patterns directory. This is where that we should define all required patterns of our chatbot. This is simple example of the pattern dictionary for Vanilla:

```
const patternDict = [{
  pattern: '\\b(Hi|Hello|Hey)\\b',
  intent: 'Hello'
  },{
  pattern:'\\b(bye|exit)\\b'
  intent: 'Exit'
}];

module.exports = patternDict;
```

  nodejs notice: the `const` keyword defines a constant reference to a value in javascript, variables must be assigned to a value when they

are declared. If we assign a primitive value to a constant, we cannot change the primitive value[1].

nodejs notice: A dictionary is defined as the following in javascript:

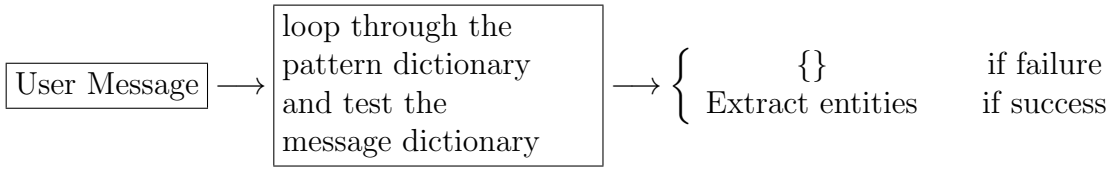
```
const input = [
{key:"key1", value:"value1"},
{key:"key2", value:"value2"}
];
```

nodejs notice: \ is an escape character, when followed by a non-special character it doesn't become a literal \. Instead, you have to double it \\. For this reason the \b in the regex string is written as \\b.

nodejs notice: Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via NPM[2], so that Node.js community can use it. For example, if you need to use this index.js in your previously made app.js and connect to the index.js then you can create a module for it, which can be reused in your application.
In the above example of index.js module, we have created a pattern dictionary. At the end, we have assigned this object to `module.exports`. The `module.exports` in the above example introduces a pattern dictionary object as a module. The `module.exports` is a special object existed in every JS file in the Node.js application by default.

Question write some patterns + intent and add them to your pattern dictionary.

Exe 2 Now we need a function to find the matches pattern and implement them. In *Exe 4*, you will design a simple *matcher function* for your chatbot such as the following model:



[1]for more details check `https://www.w3schools.com/js/js_const.asp`

[2]npm is the world's largest Software Registry. The registry contains over 800,000 code packages.Open-source developers use npm to share software.

Extracting entities can be like:

```
{
  intent: 'get weather',
  entities:{
    city: 'Paris',
    time: 'tomorrow',
  }
}
```

To implement our matcher function, create a directory namely matcher in vanillas folder with a js file namely index.js as well.
Hint: The entities should be extracted from the given sentence by the user while the pattern should be defined accordingly. For instance, for taking two entities from a phrase as "what is the weather like in Paris today?" we need to define a pattern as (add the new pattern to vanilla/pattern/index.js with other existed patterns):

```
  {
    pattern: "\\b(weather\\slike\\sin\\s\\b(?<city>[a-z
       ]+[ a-z]+?)\\b(?<time>tomorrow|today)$",
    intent: "get weather"
  }
```

where `city` and `time` are the entities for the pattern.

Exe 3 Since we need a package for using RegEx, we suggest installing the xregexp on your computer. **XRegExp** provides augmented (and extensible) JavaScript regular expressions. You get modern syntax and flags beyond what browsers support natively. XRegExp is also a regex utility belt with tools to make your grepping and parsing easier, while freeing you from regex cross-browser inconsistencies and other annoyances. If you have any problem in installing it try:
`npm -i xregexp --save --save-exact`

To check the dependencies of your project on the xregexp, check your package.json file, and the "dependencies" attribute in the file.

Exe 4 Return back to the index.js inside the matcher directory. In order to be able using any modules in the js file we should call the modules, for instance we start by:

```
'use strict';
const patterns = require('../patterns');
const XRegExp = require('xregexp');

let matchPattern = (str, cb) => {

//should be written later

}

module.exports = matchPattern;
```

nodejs notice: The `use strict` is not a statement, but a literal expression, ignored by earlier versions of JavaScript. The purpose of "use strict" is to indicate that the code should be executed in "strict mode". With strict mode, you can not, for example, use undeclared variables. All modern browsers support "use strict" except Internet Explorer 9 and lower. It should be declared just at the beginning of a script or a function.

Now, we need to define a `matchPattern` function containing two arguments string and callback `(str, cb)` where it gets the string written by user and returns back an intent, if it finds any.

nodejs notice: There's a very simple and concise syntax for creating functions, that's often better than Function Expressions. It's called "arrow functions", because it looks like this:

```
let func = (arg1, arg2, ...argN) => expression
```

This creates a function `func` that has arguments `arg1..argN`, evaluates the expression on the right side with their use and returns its result.

In other words, its roughly the same as:

```
let func = function(arg1, arg2, ...argN) {
  return expression;
};
```

But much more concise. Let's see an example:

```
let sum = (a, b) => a + b;

/* The arrow function is a shorter form of:

let sum = function(a, b) {
```

```
  return a + b;
};
*/

alert( sum(1, 2) ); // 3
```

Question try to write the function by yourself, using the `patterns` and `XRegExp` modules in the code.

To help you advancing with the code, take the following code, if you didn't find the correct function after enough thinking!

```
 let matchPattern = (str, cb) => {
  let getResult = patterns.find(item => {
    if(XRegExp.test(str, XRegExp(item.pattern, "i"))){
      return true;
    }
  });

  if(getResult){
    return cb({
      intent: getResult.intent
    });
  }
  else{
    return cb({});
  }
}
```

Exe 5 Now we want to use our modules in the original app.js (we made it in the first week). We rewrite it hear:

```
'use strict';

const Readline = required('readline'); // for including
   readline module in your application
const rl = Readline.createInterface({ //Creates an
   Interface object
  input: process.stdin,
  output: process.stdout,
  terminal: false
});

const matcher = require('./matcher'); //to use the
   matcher module here

```

```
rl.setPrompt('> ');
rl.prompt();
rl.on('line', reply => {
  matcher(reply, cb => {
    // do it by yourself
  });
} );
```

Question: write the matcher function by calling its `cb` parameter. Hint: Define a switching loop on `cb.intent` and verify different intents cases. For example, regarding the defined patterns in exercise 1, you have various options including:
1) code your chatbot reaction to the user and ask for its next message.
2) define a `default` reaction to the undefined intention for your chatbot.
3) for the exit case, reply back the user and exit the app.js.

Test your chatbot with `nodemon app.js` every time.

nodejs notice: `rl.createInterface(input, output, completer)`:

Takes two streams and creates a readline interface. The completer function is used for autocompletion. When given a substring, it returns `[[substr1, substr2, ...], originalsubstring]`. `createInterface` is commonly used with process.stdin and process.stdout in order to accept user input.

nodejs notice: The `rl.prompt()` method writes the `readline.Interface` instances configured prompt to a new line in output in order to provide a user with a new location at which to provide input.

The `rl.setPrompt()` method sets the prompt that will be written to output whenever `rl.prompt()` is called.[3]

nodejs notice *Event: 'line'*: The 'line' event is emitted whenever the input stream receives an end-of-line input (\n, \r, or \r\n). This usually occurs when the user presses the <Enter>, or <Return> keys. The listener function is called with a string containing the single line of received input.

```
rl.on('line', (input) => {
  console.log(`Received: ${input}`);
});
```

[3]for more details check this link: `https://nodejs.org/api/readline.html`

Exe 6 named capturing groups and **named backreferences**: Long regular expressions with lots of groups and backreferences may be hard to read. They can be particularly difficult to maintain as adding or removing a capturing group in the middle of the regex upsets the numbers of all the groups that follow the added or removed group.

(?P<name>group) captures the match of group into the backreference "name". name must be an alphanumeric sequence starting with a letter. group can be any regular expression. You can reference the contents of the group with the named backreference (?P=name). The question mark, P, angle brackets, and equals signs are all part of the syntax. Though the syntax for the named backreference uses parentheses, it's just a backreference that doesn't do any capturing or grouping. In our chatbot the "greeting" named capturing group is defined as:

```
'\\b(?<greeting>Hi|Hello|Hey)\\b'
```

implment it in the vanilla/patterns/index.js.

Question Go back to the index.js file in the matcher folder, the code is given in **Exe 4**. Add a function `createEntities` for extracting the entities from the user message. For instance, this function should extracts the pattern from the following phrase: Hello our beautiful chatbot!

```
let createEntities = (str, pattern) => {
  return XRegExp.exec(str, XRegExp(pattern, "i"))
}
```

use this function for calling entities in returning the `cb` callback:

```

return cb({
  intent: getResults.intent,
  entities: createEntities(str, //it is your duty to call
      the pattern here
  )
});
```

By going back to **Exe 5**, you can call these entities by calling the following variable:

```
`${cb.entities.greeting}`
```

Question modify your `matcher(reply, cb => {});` by using this variable in your interactive interface. Test your app.js.

Exe 7 Assume that the user asks on weather situation in a city. For instance he always asks the similar questions for the weather forecast:

- how is the weather in Paris
- what is the time in London?
- what is the weather in New York?

Question write a RegEx for `in city`. Notice that cities can include two words as well. Add your RegEx to your patterns dictionary code with an intent namely "Current Weather". Call the name capturing group as city.

Add another case to your app.js file with a case namely CurrentWeather. At the moment, print only the name of city to the user. Notice that you can get the city name by calling `cb.entities.city`. Test your chatbot again.

Exe 8 After extracting city names and entities, we need to connect with a weather API to find the weather situation of each city:
chatbot $\leftrightarrow$ API weather

You can use any API weather you prefer. We suggest you a list of APIs as bellow:

- `https://openweathermap.org/`
- `https://developer.accuweather.com/`
- `https://darksky.net/dev`
- `https://www.climacell.co/weather-api/`

Sign up in their website for getting a free access key to the api. Save your API key somewhere. In order to communicate with apis in javascript, it is required installing a package namely axios:
$>$ `npm i axios`

Exe 9 Add a weather module in your project by making a folder namely weather in vanilla folder and creating an index.js in the folder. In order to use the axios package, using our api key and getting the weather of the given location from the api, we need to define a function namely

"getWeather". This function receives a 'location', connects to the api and get weather information on the location forecast.

```
"use strict";
const axios = require("axios");
const apikey = ; //your api key to the api generated by
   the website

const getWeather = location => {
  return new Promise(async (resolve, reject) => {
    try{
      const weatherConditions = await axios.get( //get
         weather info from the api
       // write your api address access here, for
          instance: "http://api.apixu.com/v1/forecast.
          json",
       {
        params: {
          key: apikey,
          q: location,
          days: 3
        }
       });

       resolve(weatherConditions.data) //returns back the
           results to the chatbot
    }
    catch(error){
      reject(error);
    }
  });
}

module.exports = getWeather;
```

nodejs notice: A promise is an object which can be returned synchronously from an asynchronous function. It will be in one of 3 possible states:[4]

- Fulfilled: onFulfilled() will be called (e.g., resolve() was called)
- Rejected: onRejected() will be called (e.g., reject() was called)
- Pending: not yet fulfilled or rejected

[4]for more details check a blog

nodejs notice Axios is a very popular JavaScript library you can use to perform HTTP requests, that works in both Browser and Node.js platforms. It is promise-based, and this lets us write async/await code to perform XHR requests very easily. One convenient way to use Axios is to use the modern (ES2017) async/await syntax. This requires to retrieve a list of all information using axios.get()[5].

Exe 10 Use the weather module in app.js:

```
1 const weather = require("./weather")
```

In the `CurrentWeather` pattern case of your code call `weather(cb.entities.city)` and show its results to the chatbot user [6]. Test your app.js by asking "what is the weather in Los Angeles" or similar questions. You can define a function to print a limited set of parameters coming from the axios api.

Small Project: As a small project, try to finish your weather bot by implementing the following steps or adding your proper ideas to make your chatbot as user friendly as possible.

After finishing your project upload it to the BrightSpace on the space for weather chatbot homework.

- The api results are detailed information on each city including its exact temperature. Design a method that replies back in a human way i.e; expressing temperatures in words such as:
  human: how is the weather in Paris? bot: it is very cold in Paris, France. With -2 degrees Celsius.
- may be color some important words in your bot response.
- modify your chatbots in a way that it is capable of answering the following questions: is it cold in Berlin today?
  will it be rainy in Paris tomorrow?
  will it be sunny in Paris the day after tomorrow?

[5] for more details check `https://flaviocopes.com/axios/`

[6] Consider `.then()` and `.catch()` for extracting the data getting from the weather promise.

to implement your method, define the related Regex, add it to pattern dictionaries, define a module for extracting the entities and finding a response for your user question and finally add it to your app.js as a new case.