Summer Internship Project (R151340, R151244)

Project Title : Command line words game using *NodeJS*

Project - Guide : M. Muni babu sir

Team Memebers :

R151340 - S. Inamul Hassan
R151244 - M. Hari krishna

Introduction to NodeJs

What is Node.js ?

Node.js is an open source server environment
Node.js is free
Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
Node.js uses JavaScript on the server

Why Node.js ?

Node.js uses asynchronous programming!

A common task for a web server can be to open a file on the server and return the content to the client.

Here is how PHP or ASP handles a file request:

1. Sends the task to the computer's file system.
2. Waits while the file system opens and reads the file.
3. Returns the content to the client.
4. Ready to handle the next request.

Here is how Node.js handles a file request:

1. Sends the task to the computer's file system.
2. Ready to handle the next request.
3. When the file system has opened and read the file, the server returns the content to the client.

Node.js eliminates the waiting, and simply continues with the next request.

Node.js runs single-threaded, non-blocking, asynchronously programming, which is very memory efficient.

What Can Node.js Do?

Node.js can generate dynamic page content
Node.js can create, open, read, write, delete, and close files on the server
Node.js can collect form data
Node.js can add, delete, modify data in your database

The basics of async/await

There are two parts to using async/await in your code.

The async keyword

First of all we have the `async` keyword, which you put in front of a function declaration to turn it into an async function. An async function is a function that knows how to expect the possibility of the `await` keyword being used to invoke asynchronous code.

Try typing the following lines into your browser's JS console:

```
function hello() { return "Hello" };hello();
```

The function returns "Hello" — nothing special, right?

But what if we turn this into an async function? Try the following:

```
async function hello() { return "Hello" };hello();
```

Ah. Invoking the function now returns a promise. This is one of the traits of async functions — their return values are guaranteed to be converted to promises.

You can also create an async function expression, like so:

```
let hello = async function() { return "Hello" };hello();
```

And you can use arrow functions:

```
let hello = async () => { return "Hello" };
```

These all do basically the same thing.

To actually consume the value returned when the promise fulfills, since it is returning a promise, we could use a `.then()` block:

```
hello().then((value) => console.log(value))
```

or even just shorthand such as

```
hello().then(console.log)
```

Like we saw in the last article.

So the `async` keyword is added to functions to tell them to return a promise rather than directly returning the value.

The await keyword

The real advantage of async functions becomes apparent when you combine it with the await keyword — in fact, `await` **only works inside async functions**. This can be put in front of any async promise-based function to pause your code on that line until the promise fulfills, then return the resulting value. In the meantime, other code that may be waiting for a chance to execute gets to do so.

You can use `await` when calling any function that returns a Promise, including web API functions.

Here is a trivial example:

```
async function hello() {

  return greeting = await Promise.resolve("Hello");};

hello().then(alert);
```

Of course, the above example is not very useful, although it does serve to illustrate the syntax. Let's move on and look at a real example.

Rewriting promise code with async/await

Let's look back at a simple fetch example that we saw in the previous article:

```
fetch('coffee.jpg').then(response => {

  if (!response.ok) {

    throw new Error(`HTTP error! status: ${response.status}`);

  } else {

    return response.blob();

  }}).then(myBlob => {

  let objectURL = URL.createObjectURL(myBlob);

  let image = document.createElement('img');

  image.src = objectURL;

  document.body.appendChild(image);}).catch(e => {

  console.log('There has been a problem with your fetch operation: ' + e.message);});
```

By now, you should have a reasonable understanding of promises and how they work, but let's convert this to use async/await to see how much simpler it makes things:

```
async function myFetch() {

  let response = await fetch('coffee.jpg');


  if (!response.ok) {

    throw new Error(`HTTP error! status: ${response.status}`);

  } else {

    let myBlob = await response.blob();


    let objectURL = URL.createObjectURL(myBlob);

    let image = document.createElement('img');

    image.src = objectURL;

    document.body.appendChild(image);

  }}

myFetch().catch(e => {

  console.log('There has been a problem with your fetch operation: ' + e.message);});
```

It makes code much simpler and easier to understand — no more `.then()` blocks everywhere!

Since an `async` keyword turns a function into a promise, you could refactor your code to use a hybrid approach of promises and await, bringing the second half of the function out into a new block to make it more flexible:

```
async function myFetch() {

  let response = await fetch('coffee.jpg');

  if (!response.ok) {

    throw new Error(`HTTP error! status: ${response.status}`);
```

```
  } else {
    return await response.blob();
  }}
myFetch().then((blob) => {
  let objectURL = URL.createObjectURL(blob);
  let image = document.createElement('img');
  image.src = objectURL;
  document.body.appendChild(image);}).catch(e => console.log(e));
```

You can try typing in the example yourself, or running our live example (see also the source code).

But how does it work?

You'll note that we've wrapped the code inside a function, and we've included the `async` keyword before the `function` keyword. This is necessary — you have to create an async function to define a block of code in which you'll run your async code; as we said earlier, `await` only works inside of async functions.

Inside the `myFetch()` function definition you can see that the code closely resembles the previous promise version, but there are some differences. Instead of needing to chain a `.then()` block on to the end of each promise-based method, you just need to add an `await` keyword before the method call, and then assign the result to a variable. The `await` keyword causes the JavaScript runtime to pause your code on this line, allowing other code to execute in the meantime, until the async function call has returned its result. Once that's complete, your code continues to execute starting on the next line. For example:

```
let response = await fetch('coffee.jpg');
```

The response returned by the fulfilled `fetch()` promise is assigned to the `response` variable when that response becomes available, and the parser pauses on this line until that occurs. Once the response is available, the parser moves to the next line, which creates a `Blob` out of it. This line also invokes an async promise-based method, so we use `await` here as well. When the result of operation returns, we return it out of the `myFetch()` function.

This means that when we call the `myFetch()` function, it returns a promise, so we can chain a `.then()` onto the end of it inside which we handle displaying the blob onscreen.

You are probably already thinking "this is really cool!", and you are right — fewer `.then()` blocks to wrap around code, and it mostly just looks like synchronous code, so it is really intuitive.

Game Plan

Create a command line dictionary tool using foutytwowrods words api.
Check API here - http://fourtytwowords.herokuapp.com/

API KEY: (expiry: 2021-01-17T05:23:28.002Z)
a98eff3917981ec80a86523e17be5f61287bd0a6595728ef9feb6a9cf50f354db16fe8aa5e96d7405
784d4771876d1ff84d8b644c569371bd70ce16fa49d2fff5e15de4c572b47f55792f763df03a2c7

Node Modules Used

1. Node-fetch 2.Readline-async 3.Readline-sync

Steps to install depencies :

Make sure that nodejs and npm is installed in your machine first.

```
For Ubuntu :

Sudo apt install nodejs npm
```

Then open the command line and enter the follwing commands :

```
npm install node-fetch

npm install readline-async

npm install readline-sync
```

Requirements

The command line tool should have following functions -
The output should be nicely formatted on console, and show all relevant information.

1. Word Definitions

Display definitions of a word.
./dict def <word> OR node dict.js def <word>

2. Word Synonyms

Display synonyms of a word.
./dict syn <word> OR node dict.js syn <word>

3. Word Antonyms

Display antonyms of a word
./dict ant <word> OR node dict.js ant <word>

4. Word Examples

Display examples of a word
./dict ex <word> OR node dict.js ex <word>

5. Word Full Dict

Display all above details for a word
./dict <word> OR node dict.js <word> or node dict.js dict <word>

6. Word of the Day Full Dict

Display all above details of word of the day
./dict OR node dict.js

7. Word Game

./dict play OR node dict.js play

The program should select a random work and display a definition, or synonym, or antonym to user, and ask the user to guess the word

If correct guess is entered, program should tell that the word is correct, and start with new word to guess.

Other Synonyms (not yet displayed as hits) of the word should be accepted as correct answer.

If incorrect word is entered, program should ask for

- 1. try again

  Lets user enter guess again

- 2. hint

  Display a hint, and let user enter guess again

  Hint can be randomly chosen from following, and should not be repeated

-

  Another definition of the word

  OR Synonym of the word

  OR Antonym of the word

  If all definitions, synonyms and antonyms have already been displayed as hint, then display the word randomly jumbled (eg. 'atc' for word 'cat')

-3 skip

  Display the word, its full dict, and start with new word to guess

Game Scoring

Each correct answer gives 10 points.

Each hint reduces 3 point.

Each wrong try reduces 2 points.

Skip reduces 4 points.

Keep displaying current score of user.

Areas of Focus

- Code quality
- Code reuse
- Code structure
- Use of high level language features