Lab 3

Re-submit Assignment

Due Sunday by 11:55pm **Points** 100 **Submitting** a file upload **File Types** zip

CS-546 Lab 3

Asynchronous Code, Files, and Promises

This week will be interesting -- we're going to be working with files; particularly, reading them, creating metrics on them, and storing them using promises.

You'll need to write two modules this week, and one test file.

Important Note For Requirements: you will be explicitly marking the function as an *async function*, so remember to return the **value** of the result, rather than a promise. Remember to simply throw an error as normal. Async functions always return promises by design.

Note: you may leverage the bluebird node module as desired, to promisify methods. This is the suggested way to make your `fs` module use promises, as is done in lecture code!

Hints

- Async methods return promises, always.
- You do not need to manually create a promise anywhere
- Throwing inside an async function causes the method to return a rejected promise!
- Awaiting a rejected promise that throws, causes the async method to return a rejected promise

To recap from the lecture code and slides, the following is the general algorithm for using async methods:

```
const bluebird = require("bluebird");
const Promise = bluebird.Promise;

// We use bluebird to make a copy of fs
// that has all its normal methods, and
// {methodName}Async method versions that return
// promises as well; ie, you will have a copy
// of fs with fs.stat(path, callback) and
// fs.statAsync(path), which returns a promise
// thus allowing us to await it.
const fs = bluebird.promisifyAll(require("fs"));

async function getFileSizeInKilobytes(path) {
  // Throwing inside of an async method causes the method
  // To return a rejected promise, which means we can throw based
```

```
  // On arguments
  if (!path) throw "You must provide a path";
  const stats = await fs.statAsync(path);

  return stats.size / 1024;
}

async function main() {
    // We can await this; if it throws / rejects
  const kilos = await getFileSizeInKilobytes("./hello.txt");
  console.log(`That file is ${kilos}kb large!`);
}

main();
```

Your file module, fileData.js

This module will export four methods:

async getFileAsString(path)

This method must be an **async function**, and will implicitly return a promise. You will **await** any promises inside this method to get the result of said promise (such as the result of a file operation) in order to use it in later on in the method. If the method enters a state that should return a rejected promise, you should achieve this by **throwing**, as thrown exceptions inside async methods cause the returned promise to be in a rejected state.

This method will, when given a path, return a promise (implicitly, due to being defined as an async function) that resolves to a string with the contents of the files.

If no path is provided, it will return a rejected promise.

If there are any errors reading the file, the returned promise will reject rather than resolve, passing the error to the rejection callback.

async getFileAsJSON(path)

This method must be an **async function**, and will implicitly return a promise. You will **await** any promises inside this method to get the result of said promise (such as the result of a file operation) in order to use it in later on in the method. If the method enters a state that should return a rejected promise, you should achieve this by **throwing**, as thrown exceptions inside async methods cause the returned promise to be in a rejected state.

This method will, when given a path, return a promise that resolves to a JavaScript object. You can use the `JSON.parse` function to convert a string to a JavaScript object (if it's valid!).

If no path is provided, it will return a rejected promise.

If there are any errors reading the file or parsing the file, the returned promise will reject rather than resolve, passing the error to the rejection callback.

async saveStringToFile(path, text)

This method must be an **async function**, and will implicitly return a promise. You will **await** any promises inside this method to get the result of said promise (such as the result of a file operation) in order to use it in later on in the method. If the method enters a state that should return a rejected promise, you should achieve this by **throwing**, as thrown exceptions inside async methods cause the returned promise to be in a rejected state.

This method will take the `text` supplied, and store it in the file specified by `path`. The function should return a promise that will resolve to `true` when saving is completed.

If no path or text is provided, it will return a rejected promise.

If there are any errors writing the file, the returned promise will reject rather than resolve, passing the error to the rejection callback.

async saveJSONToFile(path, obj)

This method must be an **async function**, and will implicitly return a promise. You will **await** any promises inside this method to get the result of said promise (such as the result of a file operation) in order to use it in later on in the method. If the method enters a state that should return a rejected promise, you should achieve this by **throwing**, as thrown exceptions inside async methods cause the returned promise to be in a rejected state.

This method will take the `obj` supplied and convert it into a JSON string so that it may stored as in a file. The function should return a promise that will resolve to `true` when saving is completed.

If no path or obj is provided, it will return a rejected promise.

If there are any errors writing the file, the returned promise will reject rather than resolve, passing the error to the rejection callback.

Your metric module, textMetrics.js

Firstly, this module will export a method, `simplify(text)`. This method will take a string of text and will:

1. Convert the text to lowercase
2. Remove all characters **except** for letters and whitespace characters
3. Convert all white space to simple spaces (new lines become spaces; tabs become spaces, spaces stay the same, etc)
4. Convert all duplicate spaces to be single spaces; ie, `hello phil` with 5 spaces between each word becomes `hello phil` with one space between each word
5. Trim the whitespace off the start and end of the text
6. Return the result.

Secondly, this module will export a method, `createMetrics(text)` which will scan through the text, simplify the text, and return an object with the following information based on the **simplified** text:

```
{
    totalLetters: total number of letter characters in the simplified text,
    totalWords: total number of words in the simplified text,
    uniqueWords: total number of unique words that appear in the simplified text,
    longWords: number of words in the text that are 6 or more letters long; this is a total count of ind
ividual words, not unique words,
    averageWordLength: the average number of letters in a word in the text; this is counting the individ
```

```
ual words, not unique words,
    wordOccurrences: a dictionary of each word and how many times each word occurs in the text.
 }
```

So running:

```
createMetrics("Helllo, my -! This is a great day to say helllo.\n\n\tHelllo! 2 3 4 23")
```

Will return:

```
{
    totalLetters: 40 // (helllomythisisagreatdaytosayhelllohelllo),
    totalWords: 11 //(["helllo","my","this","is","a","great","day","to","say","helllo","helllo"] is 11 w
ords),
    uniqueWords: 9 //(helllo, my, this, is, a, great, day, to, say),
    longWords: 3,
    averageWordLength: 3.6363636363636362 // (this will round differently on each machine",
    wordOccurrences: {
        a: 1
        day: 1
        great: 1
        helllo: 3
        is: 1
        my: 1
        say: 1
        this: 1
        to: 1
    } // this may or may not sort in your system; order DOES NOT MATTER
}
```

The comments above have been added for your convenience only.

app.js

Write a file, app.js, which will perform the following operation on each of these files (found in the Canvas Lecture 3 Module):

- chapter1.txt
- chapter2.txt
- chapter3.txt

Write an async function that you will then call, that will do the following for each file:

1. Check if a corresponding result file already exists for the chapter file specified. If one does exist, gt the results and print the results that have already been stored for that file.
2. If no result file is found, get the contents of the file using `getFileAsString`
3. Simplify the text, and store that text in a file named `fileName.debug.txt`
4. Run the text metrics, and store those metrics in `fileName.result.json`
5. Print the resulting metrics

So for example, with `chapter1.txt`, you will:

1. Check if `chapter1.result.json` exists; if it does, query and print the resulting object
2. If no result is found, perform `getFileAsString(chapter1.txt)`
3. simplify the text and store the result in `chapter1.debug.txt`
4. Run the text metrics and store those results in `chapter1.result.json`
5. Print the resulting metrics

General Requirements

1. You **must not submit** your node_modules folder
2. You **must remember** to save your dependencies to your package.json folder (if you use any)
3. You must do basic error checking in each function
    1. Check for arguments existing and of proper type.
    2. Throw if anything is out of bounds (ie, trying to perform an incalculable math operation or accessing data that does not exist)
    3. If a function should return a promise, you should mark the method as an `async` function and return the value. Any promises you use inside of that, you should *await* to get their result values. If the promise should reject, ten you should throw inside of that promise in order to return a rejected promise automatically. Thrown exceptions will bubble up from any awaited call that throws as well, unless they are caught in the async method.