INFO3139 Lab 4

Rev 1.0

General JavaScript Topic #3 – Comparing loops

- **for, map, for of, forEach**

We've used some of these looping constructs a bit so far in the course, but let's do a little comparison between them.

1. Create a new folder called **week3\class1** in nodeexercises

2. Add a new file called **loop_tests.js** with the following code:

```
const generateTestArray = () => {
  const bigArray = [];
  for (let i = 0; i < 1_000_000; ++i) { // newish way to do big ints
    bigArray.push({
      a: i,
      b: i / 2,
      r: 0,
    });
  }
  return bigArray;
};

const readArrayWithMap = () => {
  const mapArray = generateTestArray();
  const startMap = performance.now();
  let newArray = mapArray.map((x) => x.b + x.b);
  const endMap = performance.now();
  return `Speed [map]: ${endMap - startMap} miliseconds`;
};

const readArrayWithFor = () => {
  const bigArray = generateTestArray();
  const startFor = performance.now();
  for (let i = 0; i < bigArray.length; ++i) {
    bigArray[i].b + bigArray[i].b;
  }
  const endFor = performance.now();
  return `Speed [for]: ${endFor - startFor} miliseconds`;
};

const readArrayWithForOf = () => {
  let x;
  const bigArray = generateTestArray();
  const startForOf = performance.now();
  for (x of bigArray) {
    x.b + x.b;
  }
  const endForOf = performance.now();
  return `Speed [forOf]: ${endForOf - startForOf} miliseconds`;
};

const readArrayWithForEach = () => {
  const bigArray = generateTestArray();
  const startForEach = performance.now();
  bigArray.forEach((x) => x.b + x.b);
  const endForEach = performance.now();
```

```
  return `Speed [forEach]: ${endForEach - startForEach} miliseconds`;
};

export {
  readArrayWithMap,
  readArrayWithFor,
  readArrayWithForOf,
  readArrayWithForEach,
};
```

Notice a couple of new things in this code; the way the million constant is defined with **underscores**, this is new in JavaScript. Also notice we're calling a function called **performance.now()** that returns a timestamp in milliseconds this is a newish feature in node.

3. Create another file called **compare_loops.js** with the following code:

```
import * as tstLib from "./loop_tests.js";

console.log(tstLib.readArrayWithMap());
console.log(tstLib.readArrayWithFor());
console.log(tstLib.readArrayWithForOf());
console.log(tstLib.readArrayWithForEach());
```

4. Execute compare_loops **a few times**:

```
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1> node compare_loops
Speed [map]: 36.003700003027916 miliseconds
Speed [for]: 11.402300000190735 miliseconds
Speed [forOf]: 30.118499994277954 miliseconds
Speed [forEach]: 14.208800002932549 miliseconds
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1> node compare_loops
Speed [map]: 30.37389999628067 miliseconds
Speed [for]: 10.09679999947548 miliseconds
Speed [forOf]: 26.500100001692772 miliseconds
Speed [forEach]: 14.736699998378754 miliseconds
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1> node compare_loops
Speed [map]: 34.44969999790192 miliseconds
Speed [for]: 10.763999998569489 miliseconds
Speed [forOf]: 25.630400002002716 miliseconds
Speed [forEach]: 17.331999994814396 miliseconds
```

Typically, the ranking is:

1. for
2. forEach
3. forOf
4. map

But remember speed is just one aspect of coding, readability of code is also important and really what's a few milliseconds when you have a million elements?

Using Node to Read/Write To a Local File System

We had a brief introduction in using Node's fs (file system) library in week 1. Today will take a more comprehensive look at it and then incorporate it with our knowledge of promises and async functions and put together a lab that will help support the first case study.

If you recall back in week 1, we did something like this that utilized **a callback** function (bolded code):

```
const fs = require('fs')

fs.readFile('./sampledata/users','utf8', (err,contents) => {
    let users = contents
    console.log(users)
})
```

Let's rewrite that example using the **fs promise library** and then call the promise from an asynchronous enabled function:

1. Copy the **users** file from the week1 content into the week3/class1 folder

2. Create a new file called **file_routines.js** with this code to start (notice how the read code has been converted to use the **fs promise-based library**):

```
// different fs library from week 1
import { promises as fsp } from "fs";

const readFileFromFSPromise = async (fname) => {
  let rawData;

  try {
    rawData = await fsp.readFile(fname); // returns promise
  } catch (error) {
    console.log(error);
  } finally {
    if (rawData !== undefined) return rawData;
  }
};

export {
  readFileFromFSPromise,
};
```

3. Add another JavaScript file called **read_userfile.js** with the following code:

```
import { readFileFromFSPromise } from "./file_routines.js";

const readUserFile = async () => {
  try {
    let users = [];
    let rawData = await readFileFromFSPromise("./users");
```

```
    rawData
      .toString()
      .split("\r\n")
      .forEach((user) => {
        if (user.length > 0) {
          let userJson = { Username: user, Email: user + "@abc.com" };
          users.push(userJson);
        }
      });

    users.forEach((user) =>
      console.log(`user ==>${user.Username}, email==>${user.Email}`)
    );
  } catch (err) {
    console.log(err);
  }
};

readUserFile();
```

4. Execute the **read_userfile.js** file in the terminal window

```
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1> node read_userfile
user ==>Evan, email==>Evan@abc.com
user ==>George, email==>George@abc.com
user ==>Sally, email==>Sally@abc.com
```

Now that we know how to read a file from the file system using promises, let us **try writing one**. We created an array of objects in step 3, let's write the array out to the file system now as new file called **users.json**.

5. Add another function to fileroutines.js with this code (note this code is using the **rest operator** … which you remember is used when an indeterminate number of arguments will be passed and must be the last argument for a method). Also, notice we're using the **forEach** syntax when iterating through the raw data which as we saw at the start of this class combines performance with readability:

```
const writeFileFromFSPromise = async (fname, ...rawdata) => {
  let filehandle;

  try {
    filehandle = await fsp.open(fname, "w");
    let dataToWrite = "";
    rawdata.forEach((element) => (dataToWrite += JSON.stringify(element))); // concatentate
    await fsp.writeFile(fname, dataToWrite); // returns promise
  } catch (err) {
    console.log(err);
  } finally {
    if (filehandle !== undefined) {
      await filehandle.close();
    }
  }
};
```

6. Update the export line to reflect the addition of the new routine

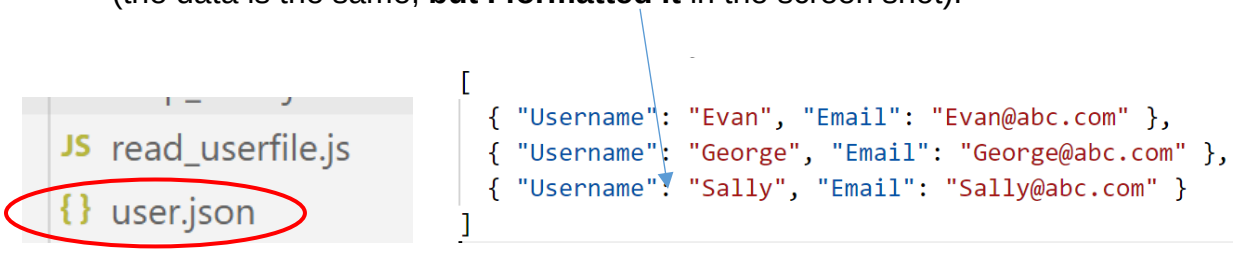
7. Create another file called **write_userjson.js** with the following:

```
import {
  readFileFromFSPromise,
  writeFileFromFSPromise,
} from "./file_routines.js";
const writeUserJson = async () => {
  try {
    let users = [];
    // rawData is returned as a buffer
    let rawData = await readFileFromFSPromise("./users");

    rawData
      .toString()
      .split("\r\n")
      .forEach((user) => {
        if (user.length > 0) {
          let userJson = { Username: user, Email: user + "@abc.com" };
          users.push(userJson);
        }
      });

    await writeFileFromFSPromise("./user.json", users);
    console.log("user json file written to file system");
  } catch (err) {
    console.log(err);
    console.log("user json file not written to file system");
  }
};

writeUserJson();
```

8. Run the **write_userjson.js** from the terminal console once more:

```
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1> node write_userjson
user json file written to file system
```

9. Notice the folder now has the new **user.json** file and its contents look like this (the data is the same, **but I formatted it** in the screen shot):



One other piece of functionality we could utilize in this setup is to write a function to determine if we **already have** a users.json file. If we have one already we won't bother doing the reading and writing logic, just inform the user that the file is already present. We can grab the statistics for the file via the **stat** method, and if stat variable gets populated we know the file exists.

10. Add a new function to the file routines module with the following code:

```
const fileStatsFromFSPromise = async (fname) => {
  let stats;
  try {
    stats = await fsp.stat(fname);
  } catch (error) {
    error.code === "ENOENT" // doesn't exist
      ? console.log("./user.json does not exist")
      : console.log(error.message);
  }
  return stats;
};
```

11. Update the export again to include the new function
12. Create a new file called **userfile_exists.js** with the following (notice we've swtiched to a wildcard import here):

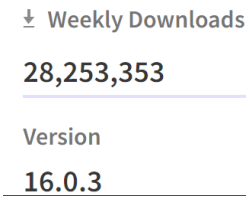
```
import * as rtnLib from "./file_routines.js";

const userFileExists = async () => {
  let filename = "./user.json";
  try {
    let fileStats = await rtnLib.fileStatsFromFSPromise(filename);
    if (!fileStats) {
      let users = [];
      let rawData = await rtnLib.readFileFromFSPromise("./users");

      rawData
        .toString()
        .split("\r\n")
        .map((user) => {
          if (user.length > 0) {
            let userJson = { Username: user, Email: user + "@abc.com" };
            users.push(userJson);
          }
        });

      await rtnLib.writeFileFromFSPromise(filename, JSON.stringify(users));
      console.log(`${filename} file written to file system`);
    } else {
      console.log(`${filename} already exists`);
    }
  } catch (err) {
    console.log(err);
    console.log(`${filename} file not written to file system`);
  }
};

userFileExists();
```

13. Delete the user.json file
14. Run **userfile_exists.js** twice, the first execution should create the file, and the second execution should indicate the file already exists:

```
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1> node userfile_exists
./user.json does not exist
./user.json file written to file system
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1> node userfile_exists
./user.json already exists
```

dotenv

To **reduce hardcoding** of names for files and locations which can be problematic, it would be nice to have a repository where we can place those kinds of items. Fortunately, there is an npm package called **dotenv** that does just that. dotenv is a zero-dependency module that loads environment variables from a file called **.env** into the node environment (process.env). npmJS statistics for this module were at the time this was written were:



15. Remember to install this package in the root **nodeexercises** folder so it can update the current package.json by issuing:

```
PS E:\winter2023\info3139\programming\nodeexercises> npm i dotenv

added 1 package, and audited 40 packages in 1s

12 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

```
12    "dependencies": {
13      "dotenv": "^16.0.3",
14      "got": "^12.5.2",
15      "yargs": "^17.6.0"
```

16. Create a new file in the week3\class1 folder called **.env** with the following contents notice how VSCode recognizes the file and changes the icon for it:

```
USERSRAW=./users
USERSJSON=./user.json
```

17. Create another file called **config.js** with the following (this file just gives us a way to do a bit more abstraction from the data in the .env file so we don't need to enter **process.env**… everywhere):

```
import { config } from "dotenv";
config();
export const rawdata = process.env.USERSRAW;
export const userobjects = process.env.USERSJSON;
```

18. Add a new file called **write_with_dotenv.js** with the following:

```
import * as rtnLib from "./file_routines.js";
import * as cfg from "./config.js";

const dotEnvWrite = async () => {
  try {
    let fileStats = await rtnLib.fileStatsFromFSPromise(cfg.userobjects);
    if (!fileStats) {
      let users = [];
      let rawData = await rtnLib.readFileFromFSPromise(cfg.rawdata);

      rawData
        .toString()
        .split("\r\n")
        .forEach((user) => {
          if (user.length > 0) {
            let userJson = { Username: user, Email: user + "@abc.com" };
            users.push(userJson);
          }
        });

      await rtnLib.writeFileFromFSPromise(cfg.userobjects, users);
      console.log(`${cfg.userobjects} file written to file system`);
    } else {
      console.log(`${cfg.userobjects} already exists`);
    }
  } catch (err) {
    console.log(err);
    console.log(`${cfg.userobjects} file not written to file system`);
  }
};

dotEnvWrite();
```

19. After making all of the changes, **delete the users.json file** run the new code twice, and you should see:

```
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1> node write_with_dotenv
./user.json does not exist
./user.json file written to file system
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1> node write_with_dotenv
./user.json already exists
```

Lab 4 – 2.5%

Part A - (2%) - Rework labs 1 and 2 and incorporate the file techinques and dotenv code we've covered in today's in work.

20. Create a new folder called week3\class1**lab4**

21. Create a file called week3\class1\lab4**.env** that contains the following 2 environment variables:

```
COUNTRIES=./countries.json
ISOCOUNTRIES= https://raw.githubusercontent.com/elauersen/info3139/master/isocountries.json
```

22. Create a corresponding week3\class1\lab4**config.js** like we did in the exercise today (step 17)

23. Create a JavasScript file called **lab4.js**. This lab4.js should

    a. Utilize the **yargs** module. This time set up yargs to use an **optional** switch to indicate whether or not the user wants to download a fresh copy of the iso country web file (ISOCOUNTRIES).
    b. Should utilize **a single** async function. This **async function** will minimally call **4 promise-based functions** contained in a separate module file with the name **iso_country_routines.js**.

1. The logic for the lab4.js code **should always** invoke **function 1** (an async stats function using fsp) passing the COUNTRIES environment variable to it. In return it should receive a variable containing the statistics for the local **countries.json** file, if the file exists or a null if it doesn't. Note the promise is intrinsic in the fsp package so you don't need to formally define this one.

2. If the local file does not exist or the optional yargs switch was used, the driver program (lab4.js) should invoke **function 2** and utilize the ISOCOUNTRIES environment variable to locate it to return a json array containing the country data **from the internet**. To do this we can use **got** with a slightly different syntax that utilizes its **.json method** which returns the JSON in a promise (see below).

3. **Function 3** (a write function using fsp) should be invoked and utilize the COUNTRIES environment variable and the json array. After writing the file you should make another call to get the statistics for the newly created file. Again, the promise is intrinsic for this one.

4. **Function 4** (a read function using fsp) is executed to read the local file, from here you can determine the # of elements in the resulting array. Promise is intrinsic again.

To summarize your **iso_country_routines.js** would have a method layout like:

```
1  import got from "got";
2  import { promises as fsp } from "fs";
3
4  const fileStatsFromFSPromise = async (fname) => {···
14 };
15
16 const getJSONFromWWWPromise = (url) => got(url).json();
17
18 const writeFileFromFSPromise = async (fname, ...rawdata) => {···
36 };
37
38 const readFileFromFSPromise = async (fname) => {···
51 };
```

24. Delete the countries.json file **prior** to starting.

25. Execute lab4.js **4 times**:
    a. The first execution should just show the yargs help with whatever option you set configured in yargs (--**refresh** was used in the screen shot below)
    b. Then issue a second execution (**no arguments**) which should indicate that the json file does not exist and a new file was written and should also show the creation date and number of elements in the json array
    c. The 3rd execution should use your option switch (again --refresh was used as the argument in the example), indicate that a new file was written and should also show the creation date and number of elements in the json array
    d. The 4th execution (no arguments) should indicate that the existing file was used and show the creation date and number of elements in the json array. You can obtain the creation date from the stats, use the **.ctime** property (create time)

No names should be hardcoded anywhere in the JavaScript code (use the .env and config.js files similarly to the example given in today's notes)

26. Submit **a single word doc** that contains:
    a. The source code for both .js files
    b. a single screen shot showing all 4 executions:

```
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1\lab4> node lab4 -h
Options:
      --version  Show version number                                   [boolean]
      --refresh  is a fresh copy from the web required?                 [string]
  -h, --help     Show help                                             [boolean]
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1\lab4> node lab4
./countries.json does not exist
A new ./countries.json file was written.
./countries.json was created on Sat Nov 05 2022 16:00:13 GMT-0400 (Eastern Daylight Time).
There are 249 codes in ./countries.json
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1\lab4> node lab4 --refresh
A new ./countries.json file was written.
./countries.json was created on Sat Nov 05 2022 16:00:24 GMT-0400 (Eastern Daylight Time).
There are 249 codes in ./countries.json
PS E:\winter2023\info3139\programming\nodeexercises\week3\class1\lab4> node lab4
An existing ./countries.json file was read from the file system.
./countries.json was created on Sat Nov 05 2022 16:00:24 GMT-0400 (Eastern Daylight Time).
There are 249 codes in ./countries.json
```

Lab 4 Part B - (.5%) - Lab 4 Theory Quiz on FOL - 10 questions

Review Lab 4

1. What is the best performing loop construct?
2. What is a reason for **not** using it?
3. What library module is required to work with files that return promises?
4. What parameter(s) is/are passed to the fsp.readFile method?
5. What parameter(s) is/are passed to the fsp.writeFile method?
6. Explain how we determined whether a local file existed.
7. Explain how the .split function works?
8. What did we split on?
9. When manually building the json array, what array method was used to add each user object into the array?
10. What package was added to deal with programming constants?
11. What file did we actually place the constants in?
12. What was the role of the config.js file?
13. What syntax was used to utilize the config.js in the driver program?
14. What url was the ISO country data obtained from?
15. How many functions return promises in lab4's countrycoderroutines.js?
16. What was the purpose of each of these functions?
17. Explain the difference between JSON.parse and JSON.stringify?
18. What iteration syntax did we utilize in the writeFileFromFSPromise method?

Homework Reading - Prior to next class, you can do yourself a favour and read the following link to get a feel for the NoSQL database we'll be using in this course MongoDB - https://www.guru99.com/what-is-mongodb.html