50.003 - Node.js

Learning Outcomes

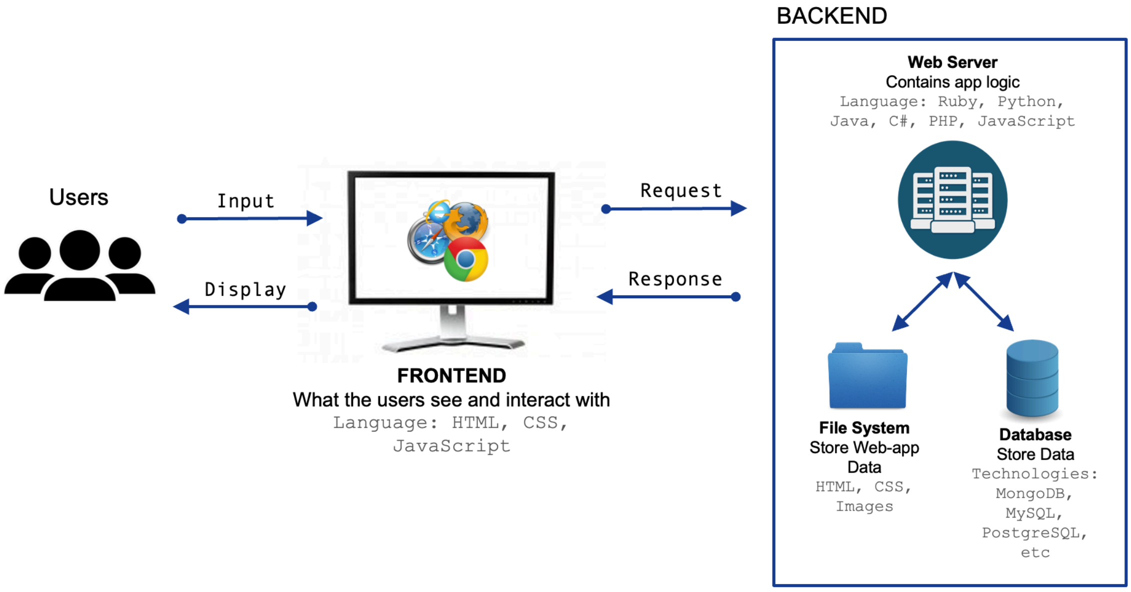
By the end of this unit, you should be able to

1. Name the differences between frontend and backend development for web application
2. Describe the use of a package/project manager for node.js
3. Describe the run-time system of node.js
4. Compare the difference between event callbacks and promises.
5. Analyse the run-time behavior of an asynchronous node.js program

Frontend vs Backend

In many web applications, it is insufficient to run the application in the browser which is a run-time system hosted in the user's device, such as desktop, laptop or mobile phone. An obvious reason is that certain computation must not be executed on the user's device due to security and integrity, for instance, transferring balance from one account to another. These highly sensitive operations should be executed at the server ends, which is known as the "backend".

In practice, frontend deals with the end users' input and interaction, while backend deals with the data storage, data analysis, data retrieval, business logic, authentication, and scalability



Node.js

There are many options of implementing the backend applications, e.g. Spring with Java, Django with Python, Flask with Python, Node.js with JavaScript.

Node.js is a run-time system to run JavaScript applications without the browser (and its event APIs), for instance `document` is no longer a predefined reference in Node.js.

Hello World

The hello world program in Node.js is not too far apart from the one in the browser.

Suppose we have a JavaScript file name `hello.js` with the following content,

```
console.log("hello");
```

To execute it, we need the node.js run-time to be installed, for installation, please refer to

```
https://nodejs.org/en/download
```

Then in the terminal,

```
node hello.js
```

We see the message `hello`, being printed in the terminal.

Since it is using the same language as the browser run-time, we will skip those common language features and focus on the difference.

Node.js Project/Package Manager

In most of the cases, we develop projects based on existing libraries, modules and packages. To better manage all these dependencies, we need a project management tool. `npm` is the mostly commonly used too in the node.js community. Its role is similar to `pip` for python and `gradle` for java.

To start a Node.js project,

```
mkdir myproj
cd myproj
npm init
```

To add a dependency, we type `npm i package_name`. For example,

```
npm i xhr2
```

where `xhr2` is the library that we would like to install as a dependency for our current project. In this example, we install xhr2 library, which is useful to emulate XMLHttpRequest, an API that provides functionality for transferring data between client and server.

After executing the above command, we observe that the `xhr2` library is downloaded to a temporary forder `node_modules` and the following

```
  "dependencies": {
    "xhr2": "^0.2.1"
  }
```

is added to the project definition file `package.json`.

When we clone the project to a new machine, (for development installation or deployment purpose), we can download all the dependencies defined by `package.json` by running

```
npm i
```

Next we would like to enable the ES6 module mode in this project. We will explain what is ES6 module mode shortly.

Use an editor to add the following entry to the `package.json` file.

```
{
  "name": "myproj",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "type": "module", // enable module type
  "author": "",
  "license": "ISC",
  "dependencies": {
    "xhr2": "^0.2.1"
  }
}
```

CommonJS vs ES6 modules

Module system allows one to put common codes in a module which can be reused by many different use sites.

For instance, consider the following JavaScript program `mymath.js`

```
const pi = 3.14159;
const e = 2.71828;
```

```
module.exports = { pi, e };
```

in which we define two constant variables `pi` and `e`, and export them, so that when `mymath.js` is being imported in another JavaScript program, these two constant variables can be reused.

Traditionally, In Common JavaScript (dubbed as CJS), we import predefined references from another JS file, via the `require()` function.

```
const mymath = require("./mymath.js");
console.log(mymath.pi);
```

In ES6 onwards, the following "better" syntax was instroduced,

Exporting

```
const pi = 3.14159;
const e = 2.71828;

export { pi, e };
```

Importing

```
import { pi } from "./mymath.js";
console.log(pi);
```

For the rest of this unit, we will stick to the ES6 import syntax.

Let's consider the following JavaScript program `circle.js` that makes use of `mymath.js`.

```
import { pi } from "./mymath.js";

class Circle {
  constructor(r) {
    this.r = r;
  }
  area() {
    return this.r ** 2 * pi;
  }
}

export default Circle;
```

In the above, we make use of the `pi` defined in `mymath.js` to compute the area of a `Circle` object. And the end of the file, we export the class definition `Circle` with a `default` modifier. Note that there can only one name to be exported if `default` is used. Being a `default` export, we do not need to surround it with `{}` when importing. In CommonJS (pre ES6), we write `modules.export = Circle` instead.

Consider the following program `moduletest.js`

```
import Circle from "./circle.js";
var circle = new Circle(10);
console.log(circle.area());
```

Node.js Event-loop

Like JavaScript run-time in the browser, when the main program ended, Node.js run-time goes into an event loop. Remember that events like timer, http request, data transfer, or button click can trigger callbacks. The callbacks are asynchronous and might be executed by the other threads or the nodeJS main thread itself.

The Node.js event-loop consists of the following steps, (AKA phases.)

```
   ┌───────────────────────────┐
┌─>│           timers          │
│  └─────────────┬─────────────┘
│  ┌─────────────┴─────────────┐
│  │     pending callbacks     │
│  └─────────────┬─────────────┘
│  ┌─────────────┴─────────────┐
│  │       idle, prepare       │
│  └─────────────┬─────────────┘      ┌───────────────┐
│  ┌─────────────┴─────────────┐      │   incoming:   │
│  │           poll            │<─────┤  connections, │
│  └─────────────┬─────────────┘      │   data, etc.  │
│  ┌─────────────┴─────────────┐      └───────────────┘
│  │           check           │
│  └─────────────┬─────────────┘
│  ┌─────────────┴─────────────┐
└──┤      close callbacks      │
   └───────────────────────────┘
```

1. timers: this phase executes callbacks scheduled by setTimeout() and setInterval().
2. pending callbacks: executes I/O callbacks deferred to the next loop iteration.
3. idle, prepare: only used internally.
4. poll: retrieve new I/O events; execute I/O related callbacks (almost all with the exception of close callbacks, the ones scheduled by timers, and setImmediate()); node will block here when appropriate.
5. check: setImmediate() callbacks are invoked here.
6. close callbacks: some close callbacks, e.g. socket.on('close', ...).

Between each run of the event loop, Node.js checks if it is waiting for any asynchronous I/O or timers and shuts down cleanly if there are not any.

Important points to note in this section are:

1. There are several commonly used APIs that allow NodeJS to execute instructions asynchronously such as `setTimeout()`, `setInterval()`, `setImmediate()`, and I/O related tasks (for example, `fs.writeFile()`, HTTP Request and Response, or `fetch()`)
2. The JS in browser's callback queue that we know before is called **macrotask** queue in nodeJS runtime. Except `fetch()`, all the APIs mentioned in the previous points enqueue callbacks into macrotask.
3. Another type of callback queue are called **microtask**. Microtask has a higher priority than macrotask, meaning that all existing microtasks will be cleared before event-loop dequeue the next macrotask.
4. Callbacks that are enqueued into microtask are `resolve()` or `reject()` functions from `promise` and API `queueMicrotask()`.
5. Now we will discuss I/O APIs callbacks (the 2nd Phase, and 4th Phase), `fetch()`, and `promise` in the next sections. You can explore the other APIs on your own.

I/O Asynchronous Callbacks in NodeJS

Let's consider an example of using network I/O,

```
import XMLHttpRequest from "xhr2";
var xhr = new XMLHttpRequest();
var args = process.argv;
if (args.length == 3) {
  var input = args[2];
  xhr.onreadystatechange = function () {
    if (xhr.readyState == 4) {
      var res = xhr.responseText;
      console.log(res);
    }
  };
  xhr.open("GET", `https://postman-echo.com/get?x=${input}`);
  xhr.send();
} else {
  console.log("USAGE: node index.js input");
}
```

Explanation:

- The first statement, we import the `XMLHttpRequest` class from the package `xhr2`. (We don't need to import it when the code is executed in the browser as all browser has `XMLHttpRequest` class builtin in the run-time, but this is not the case for node.js).
- In the second statement, we instantiate an `XMLHttpRequest` request object.
- In the third line, we read the command line arguments into a variable `args`. Remember that we can pass the argument when executing the file. For example, if the filename is `index.js`, then we can execute `node index.js abcd` in the terminal. In this example, `abcd` is the command line argument.

- The following `if-else` statement checks whether the user supplies an argument. The `node` and `index.js` are counted, so the length must be 3.
- In the `if` body, we extract the third argument, which will be used as the input parameters in the API call.
- Before calling the API, we set-up a **callback** in the **event of the HTTP Request state is changed**. As we discussed earlier, the node.js run-time has an event loop, which periodically checks for events (Phase 4). In this case, the change of state in the `XMLHttpRequest` object is one of the events it is checking. When the **event** occurs, i.e. the state of the object changes, the **callback function** will be placed in the **callback queue**. When the main call stack is empty, the callback will be put into the call stack for execution. Read about `.onreadystatechange` here and `readyState` property here. Basically, HTTP request has several states, and the last state is state 4 (DONE) which means that the data transfer has been completed (response received) or failed.
- In the callback function, we check whether the state is `4`, which stands for `DONE`. When the call is done, we extract the response text and print it out.
- After setting up the callback function, we `open` the request and submit it. In this example, we send a request to postman-echo which allows us to test sending a request as a client. The API will return a JSON response.
- If we execute this script, we will observe the API request's response being printed on the command line.

Next let's consider another example that interacts with file system via the `node:fs` library (which is a builtin module).

```
import { writeFile } from "node:fs";
const txt = "hello";
writeFile("save.txt", txt, (err) => {
  if (err) throw err;
  console.log("The file has been saved!");
});
```

Explanation:

- At line 2, we define the text data to be written to the file.
- At line 3, we write the result to a file named `save.txt`, note that `writeFile()` takes three arguments. The last argument is a callback function, whose argument `err` is potentially an error. In this case, when the error is present, we propagate the error by throwing it as an exception, otherwise, we print a message.

Callback Pyramid

Let's say now we would like to combine the two examples, first we would like to make an API call, when the call is returned successfully, we save the result of the API call into a file.

```
import XMLHttpRequest from "xhr2";
import { writeFile } from "node:fs";

var xhr = new XMLHttpRequest();
var args = process.argv;
if (args.length > 2) {
  var input = args[2];
  xhr.onreadystatechange = function () {
    if (xhr.readyState == 4) {
      var res = xhr.responseText;
      writeFile("api_result.txt", res, (err) => {
        if (err) throw err;
        console.log("The file has been saved!");
      });
    }
  };
  xhr.open("GET", `https://postman-echo.com/get?x=${input}`);
  xhr.send();
} else {
  console.log("USAGE: node index.js input");
}
```

Note that we have to "embed" the routine of writing the result `res` into a file `api_result.txt` deep inside the callback of `xhr` request, the `writeFile()` call itself has another callback. If we have many asynchronous steps following one another, the code will become complicated and hard to read.

Promise

Promise is a builtin class in JavaScript, which allows us to build a sequence of asynchronous tasks without nesting call-backs.

A `Promise` object can be instantiated by passing in a function as argument. This function argument is called the **executor**. An executor function is a higher order function that takes two functions as arguments, commonly named as `resolve` and `reject`. `resolve` is applied to the result of the promise during normal execution, `reject` is applied when some error occurs. The function body of executor is executed synchronously while `resolve` and `reject` are **asynchronous and queued into microtask**. Look at the simple `Promise` example below which is not using `reject`, but only `resolve`.

```
// executor is executed synchronously when promise object is initialized with
"new" keyword.
// Console will print "Hello" when this code below is run
let alwaysWork = new Promise(function (resolve, reject) {
  console.log("Hello");
  resolve();
});
```

```
// resolve() and reject() functions are the argument passed to .then() or
.catch()
alwaysWork.then(() => console.log("Comes later"));
```

Explanations:

- `function (resolve, reject) { console.log("Hello"); resolve();` is the **executor**.
- We can think of `Promise` in JS as this way: "I want to run a piece of code (everything defined inside executor, synchronously). At the end of the code execution, I will either reach a success or failed state. If success, I would like to execute the `resolve()` function, otherwise I will execute `reject()`. I don't know yet what is the `resolve()` function. It will be provided when a function is passed as an argument to `.then()`. Same thing for `reject()`, which depends on function passed to `.catch()`".
- In the example above, we use the anonymous function for `resolve()`, which is `() => console.log("Comes later")`. This function is queued into microtask.
- However, it is logical that `resolve()` or `reject()` can be a code that queues task to macrotask (e.g. you can have `setTimeout()` inside the function argument for `.then()`)

Let's consider a more complicated example using `Promise`.

```
function asyncCounter() {
  var count = 0;
  return new Promise((resolve, reject) => {
    resolve(count);
  });
}

function incr(count) {
  console.log(count);
  return new Promise((resolve, reject) => {
    resolve(++count);
  });
}

let counter = asyncCounter();

counter.then(incr).then(incr).then(incr);
```

Explanation:

- In the above we instantiate an asynchronous counter by calling the function `asyncCounter()`, which initializes a local variable `count` and return a promise object that immediately `resolve` with the variable.
- Do you notice that we use closure here?
- Next we invoke `.then` method of this counter (promise) three times. We can chain the invocation because, `.then` method takes a method and **returns a new promise**. In this

case, we use the `incr` function as the argument of `.then`. In the `incr` function, we print the `count` variable and return the new promise while incrementing `count` by 1.

- **Fun-but-important fact**: you can simply write the return statement of `incr()` function as `return ++count;`. JavaScript intrisically turn the returned value as a promise object resolved with returned value. We use this approach sometimes in the notes.

When we execute the above we will see the following printed on the terminal.

```
0
1
2
```

Now let's make some changes so that the we will stop incrementing the counter when a limit is reached. We modify the `incr()` function as follows,

```
// definition of asyncCounter() remains unchanged.
function incr(count) {
  console.log(count);
  return new Promise((resolve, reject) => {
    if (count > 1) {
      reject("limit reached");
    } else {
      resolve(++count);
    }
  });
}
```

In the above, `incr` returns a new promise object which does not always resolve with `++count`. In the event that the current `count` value is greater than 1, it will reject the rest of the computation, by applying `reject`.

Now coming back to the use of the `counter`, we chain it with `incr` 4 times, followed by a call to a `catch()` method invocation and consumes the error raised by any of the `reject` function call.

```
let counter = asyncCounter();

counter
  .then(incr)
  .then(incr)
  .then(incr)
  .then(incr)
  .catch((reason) => console.log(`rejected: ${reason}.`));
```

calling the above, we have

```
0
1
```

```
2
rejected: limit reached.
```

You might argue that we don't experience any asynchronousness here.

Let's modify the example by attaching a prefix to the message printed by the `incr` function and creating a second counter.

```
function asyncCounter() {
  var count = 0;
  return new Promise((resolve, reject) => {
    resolve(count);
  });
}

function incr(id) {
  return function (count) {
    console.log(`${id}:${count}`);
    return new Promise((resolve, reject) => {
      if (count > 1) {
        reject("limit reached");
      } else {
        resolve(++count);
      }
    });
  };
}

let counter1 = asyncCounter();

counter1
  .then(incr("c1"))
  .then(incr("c1"))
  .then(incr("c1"))
  .catch((reason) => console.log(`rejected: ${reason}.`));

let counter2 = asyncCounter();

counter2
  .then(incr("c2"))
  .then(incr("c2"))
  .then(incr("c2"))
  .catch((reason) => console.log(`rejected: ${reason}.`));

setImmediate(() => console.log("tick!"));
console.log("main done!");
```

In the 2nd last statement, we call `setImmediate` which prints a `tick!` message at the next event loop cycle. Note that `setImmediate()` queue the callback to macrotask. When executing the above example, we observe

```
main done!
c1:0
c2:0
```

```
c1:1
c2:1
c1:2
c2:2
rejected: limit reached.
rejected: limit reached.
tick!
```

It reveals that the execution of the `incr` method calls for both promise objects are interleaved. The `tick` message shows that all these happens in one single event loop cycle because all the microtask from `promise` are executed before dequeueing macro task from the `setImmediate()`. We will discuss what happen behind the scene in the next section.

Node.js run-time model

Browser vs NodeJS runtime (revisit)

Recall from the previous class, we studied the JavaScript run-time in the browser, which has a call stack, a heap, an event registry and a callback queue.

1. The run-time executes the JavaScript main program in the call stack until no more stack frame left, and goes into the event loop.
2. When there is an event triggered, the callback function associated with the event (in the event registery) will be added to the callback queue.
3. In an event loop cycle, when there is no more frame in the call stack but there is some item in the call back queue, the run-time will dequeue the callback from the queue and add it to the call stack.

In node.js runtime model, there are two callback queues:

1. The macro task queue, which is same as the callback queue that stores callback associated with events.
2. The micro task queue, which stores callbacks associated with promises.

Given that promises are special builtin of node.js run-time, they are treated "differently" when executed. When a promise is instantiated, its executor function is executed upto the `resolve(...)` (or `reject(...)`) statement. The call of `resolve(...)` (or `reject(...)`) is enqueued into the microtask queue. In an event loop cycle, when the call stack is empty, the run-time checks whether the microtask queue contains any item before checking the macrotask queue.

Callstack & Callback Queue Analysis of code execution in Node.js

We illustrate the execution of the last example in the following table, for simplicity, we omit the event registry table, and the macrotask queue. We are interested in the program counter, the call stack, the microtask queue, and the list of promises. We also paste the code here with line numbers to help us in the analysis.

```
1: function asyncCounter()  {
2:     var count = 0;
3:     return new Promise( (resolve, reject) => {
4:         resolve(count);
5:     });
6: }
7:
8: function incr(id) {
9:     return function (count) {
10:         console.log(`${id}:${count}`);
11:         return new Promise((resolve, reject) => {
12:             if (count > 1) {
13:                 reject("limit reached");
14:             } else {
15:                 resolve(++count);
16:             }
17:         });
18:     };
19:}
20:
21: let counter1 = asyncCounter();
22:
23: counter1
24:    .then( incr("c1") )
25:    .then( incr("c1") )
26:    .then( incr("c1") )
27:    .catch( (reason) => console.log(`rejected: ${reason}.`))
28:
29: let counter2 = asyncCounter();
30:
31: counter2
32:    .then( incr("c2") )
33:    .then( incr("c2") )
34:    .then( incr("c2") )
35:    .catch( (reason) => console.log(`rejected: ${reason}.`))
36:
37: // setImmediate(() => console.log("tick!"));
38: console.log("main done!")
```

line num	call stack	micro queue	promises
21	[main]	[]	{}
1	[main, asyncCounter]	[]	{}
3	[main, asyncCounter]	[]	{promise@21}

line num	call stack	micro queue	promises
23	[main]	[]	{promise@21}
8-9	[main, incr]	[promise@21.resolve(0)=incr("c1")(0)]	{promise@21}
23	[main]	[promise@21.resolve(0)=incr("c1")(0)]	{promise@21}

In the above we show the execution of the program from line 1 to line 27.

1. Line 1 defines function `aysncCounter`
2. Line 8 defines function `incr`.
3. Line 21, `asyncCounter()` is invoked, the program counter moves back to line 2 then line 3,
4. Line 3, a promise object `promise@21` is instantiated, its body is executed upto the call to `resolve`, since at this stage, we are not sure what the `resolve` function could be. We add `promise@21` to the set of promises. We added the suffix `@21` to indicate that the promise object was instantiated and with reference at line 21. This helps us to reason about the execution.
5. Line 23, `promise@21` is being chained with `.then(incr("c1"))`, we first move the program pointer back to line 8-9 and compute `incr("c1")`, which add the function call to call stack
6. Line 9, we return a function, this function will be the `resolve` function of the `promise@21` object. One may ask what about the `reject` function of the same promise object, since in this case the `reject` is never used, we could omit it. To be precise, we can say that it can be an identity reject function. `js (err) =>  new Promise((resolve, reject) => reject(err));` We are done with the call `incr("c1")` and return to the call site Line 24. It is another chain with `.then`. However the value to be produced here is the result of a task from the microtask queue, which is not executed yet so we skip the rest of the `.then()`s. If we continue to execute the rest of the program (ignoring lines 37-38), we will end up with

program counter (line num)	call stack	micro queue	promises
29	[main()]	[promise@21.resolve(0)=incr("c1")(0)]	{promise@21}

program counter (line num)	call stack	micro queue	promises
1	[main(), asyncCounter()]	[promise@21.resolve(0)=incr("c1")(0)]	{promise@21}
3	[main(), asyncCounter()]	[promise@21.resolve(0)=incr("c1")(0)]	{promise@21, promise@29}
31	[main()]	[promise@21.resolve(0)=incr("c1")(0)]	{promise@21, promise@29}
8-9	[main(), incr()]	[promise@21.resolve(0)=incr("c1")(0), promise@29.resolve(0)=incr("c2")(0)]	{promise@21, promise@29}
31	[main()]	[promise@21.resolve(0)=incr("c1")(0), promise@29.resolve(0)=incr("c2")(0)]	{promise@21, promise@29}
eof	[]	[promise@21.resolve(0)=incr("c1")(0), promise@29.resolve(0)=incr("c2")(0)]	{promise@21, promise@29}

At this stage, the call stack is empty, the node.js run-time dequeues the first task from the micro task queue, i.e. `promise@21.resolve(0)`. `incr("c1")(0)` prints `c1:0`, generates a new promise `promise@24` (because the promise object will be returned to the chaining at line 24).

program counter (line num)	call stack	micro queue	promises
9-18	[function@9(0)]	[promise@29.resolve(0)=incr("c2")(0)]	{promise@21, promise@29, promise@24}

Since the promise at line 24 become known, the `.then(incr("c1"))` at line 25 can provide the function to resolve the promise of line 24 and enqueue the function to the micro queue.

program counter (line num)	call stack	micro queue	promises
25	[]	[promise@29.resolve(0)=incr("c2")(0), promise@24.resolve(1)=incr("c1")(1)]	{promise@21, promise@29, promise@24}

Since the call stack is empty, we dequeue the next item from micro task queue, which is

program counter (line num)	call stack	micro queue	promises
9-18	[function@9(0)]	[promise@24.resolve(1)=incr("c1")(1)]	{promise@21, promise@29, promise@24, promise@32}

Since the promise at line 32 become known, thanks to the `.then(incr("c2"))`, we resolve the promise at line 32 and enqueue to the micro queue.

program counter (line num)	call stack	micro queue	promises
33	[]	[promise@24.resolve(1)=incr("c1")(1), promise@32.resolve(1)=incr("c2")(1)]	{promise@21, promise@29, promise@24, promise@32}

By repeating the similar steps, we get

```
c1:0
c2:0
c1:1
c2:1
c1:2
```

```
c2:2
rejected: limit reached.
rejected: limit reached.
```

Mixing callbacks and promises

Returning to the earlier example with API call and file write operations, we now can rewrite the example as follows by introducing promises.

```
import XMLHttpRequest from 'xhr2';
import { writeFile } from 'node:fs';

var xhr = new XMLHttpRequest();
var args = process.argv;
if (args.length > 2) {
    var input = args[2];
    let apiPromise = new Promise( (resolve, reject) => {
        xhr.onreadystatechange = () => {
            if (xhr.readyState == 4) {
                var res = xhr.responseText;
                resolve(res);
            }
        };
        xhr.open('GET', `https://postman-echo.com/get?x=${input}`);
        xhr.send();
    });

    function feedResultToFile(result) {
        return new Promise(  (resolve, reject) => {
            writeFile('api_result.txt', result, (err) => {
                if (err) {
                    reject(err);
                } else {
                    resolve('The file has been saved!');
                }
            });
        });
    }
    apiPromise
        .then(feedResultToFile)
        .then( (res) => console.log(res))
        .catch((err) => console.log(err)) ;
} else {
    console.log("USAGE: node api_fs_callback_2nd_attempt input");
}
```

Copy the code without the line number here

```
import XMLHttpRequest from "xhr2";
import { writeFile } from "node:fs";

var xhr = new XMLHttpRequest();
var args = process.argv;
if (args.length > 2) {
  var input = args[2];
  let apiPromise = new Promise((resolve, reject) => {
    xhr.onreadystatechange = () => {
      if (xhr.readyState == 4) {
        var res = xhr.responseText;
        resolve(res);
      }
    };
    xhr.open("GET", `https://postman-echo.com/get?x=${input}`);
    xhr.send();
  });

  function feedResultToFile(result) {
    return new Promise((resolve, reject) => {
      writeFile("api_result.txt", result, (err) => {
        if (err) {
          reject(err);
        } else {
          resolve("The file has been saved!");
        }
      });
    });
  }
  apiPromise
    .then(feedResultToFile)
    .then((res) => console.log(res))
    .catch((err) => console.log(err));
} else {
  console.log("USAGE: node api_fs_callback_2nd_attempt input");
}
```

First we wrap the API call into a promise, `apiPromise`, which is chained with a resolve function `feedResultToFile`, in which we wrap the operation of writing into the a file into another promise. When the above is executed,

line num	call stack	micro queue	promises	macro queue	event reg
8	[main()]	[]	{promise@8}	[]	
9	[main()]	[]	{promise@8}	[]	{xhr.readyst : function@

line num	call stack	micro queue	promises	macro queue	event reg
15,16	[main()]	[]	{promise@8}	[]	{xhr.readyst : function@'
30	[main()]	[]	{promise@8}	[]	{xhr.readyst : function@'
eof	[]	[]	{promise@8}	[]	{xhr.readyst : function@'

When the program counter is at line 8, we instantiate a promise, which is unresolved. At line 9, we register the `xhr.headystatechange` event with a callback function `function@9`. At lines 15-16, we are setting up the API calls and send the request, which does not affect the micro nor macro queues. At line 30, though we see a `.then()` chaining with promise@8, however, promise@8 is not yet resovled until function@9 is called. Hence we skip the rest. At the end of the JavaScript program, the call stack is empty and the microtask queue is also empty. Suppose at this moment, `xhr`'s `readystatechange` event is triggered, and `function@9` will be added to the call stack. Suppose `xhr.readyState == 4` and the `responseText` is `hello`.

line num	call stack	micro queue	promises	macro queue
eof	[]	[]	{promise@8}	[function@9
9	[function@9()]	[]	{promise@8}	[]
11,12	[function@9()]	[]	{promise@8}	[]
31	[]	[promise@8.resolve("hello") = feedResultToFile("hello")]	{promise@8}	[]

We put the function@9 to call stack and execute it, which in term resolves `promise@8`. Then we proceed to line 31 `.then(feedResultToFile)` as `promise@8` is resolved, in which `promise@8.resolve` is enqueued to the microtask queue. When `function@9` finishes, the run-time will dequeue `promise@8.resolve` and put `feedResultToFile("hello")` into the call stack to execute, which generate a promise `promise@31`. The executor calls `writeFile()` API which queues a callback into macrotask. Assume that all promise will resolve, we will get the desired behavior.

line num	call stack	micro queue	promises	m
19	[feedResultToFile("hello")]	[]	{promise@8}	[]
20	[feedResultToFile("hello")]	[]	{promise@8, promise@31}	[]
21	[feedResultToFile("hello")]	[]	{promise@8, promise@31}	[f
eof	[]	[]	{promise@8, promise@31}	[f
21	[function@21]	[]	{promise@8, promise@31}	[]
32	[]	[promise@31.resolve(res) = (res) => console.log(res)]	{promise@8, promise@31}	[]
32	[(res) => console.log(res)]	[]	{promise@8, promise@31}	[]
eof	[]	[]	{promise@8, promise@31}	[]

In this example, we see how promises (micro tasks) and event callbacks (macro tasks) being executed together. Observe that `o.then(f)` chaining is pending until the promise object `o` is resolved, then `o.resolve=f` is enqeueued into the microtask queue. When the call stack is empty, the run-time tries to look into the microtask queue before checking the macrotask queue.

Full of promises

For ease of use, many node.js libraries provides both callback (lower level) and promise (higher level) APIs, some even provides synchronous APIs. For instance, `node:fs` library offers all three types of APIs.

Unfortunataely, `xhr2` is one of those that do not provide promise APIs. To rewrite the earlier example using promise API only, we replace `xhr2` with `node-fetch`. The fetch() API returns a promise.

```
import fetch from "node-fetch";
import { promises } from "node:fs";

var args = process.argv;
if (args.length > 2) {
  var input = args[2];
  let apiPromise = fetch(`https://postman-echo.com/get?x=${input}`);
  apiPromise
    .then((response) => response.text())
    .then((text) => promises.writeFile("api_result.txt", text))
    .then((res) => {
      console.log("The file has been saved!");
      return res;
    })
    .catch((err) => console.log(err));
} else {
  console.log("USAGE: node api_fs_call_back_3rd_attempt input");
}
```

Some notes to help you understand the code above:

- `response` is an HTTP response object as the result of resolved promise of `fetch()
- `response.text()` actually returns a promise, reading the data streams that resolves with a String. Therefore, `text` is a String
- `fs.writeFile()` also returns a promise that resolves with `undefined`.

Nicer syntax

Let `p` be a promise that eventually produces result `r`, let `(r) => e` be a function that takes `r` and produces a new promise.

```
p.then((r) => e);
```

can be rewritten

```
let r = await p;
e;
```

The reverse direction also works. In otherwords, we can rewrite our earlier API and file writing example as follows,

```
import fetch from "node-fetch";
import { promises } from "node:fs";

var args = process.argv;
if (args.length > 2) {
  var input = args[2];
  let response = await fetch(`https://postman-echo.com/get?x=${input}`);
  let text = await response.text();
  let res = await promises.writeFile("api_result.txt", text);
  console.log("The file has been saved!");
} else {
  console.log("USAGE: node index.js input");
}
```

which will be translated to the version with `.then()`.

When we want to use `await` style programming in a function, we should declare the function as `async`. For instance, if we rewrite the above example, by moving the main routine in to a main function.

```
import fetch from "node-fetch";
import { promises } from "node:fs";

async function main(args) {
  if (args.length > 2) {
    var input = args[2];
    let response = await fetch(`https://postman-echo.com/get?x=${input}`);
    let text = await response.text();
    let res = await promises.writeFile("api_result.txt", text);
    console.log("The file has been saved!");
  } else {
    console.log("USAGE: node index.js input");
  }
}

main(process.argv);
```

User Defined Events

We can define customized events to trigger callbacks in Node.js.

For instance

```
import EventEmitter from "events";

const myEvtEmt = new EventEmitter();

myEvtEmt.on("start", (data) => {
  console.log(`data ${data} received`);
});

myEvtEmt.emit("start", 1);
```

we make use of the `EventEmitter` class imported from the `events` library. In the above code, we define an `EventEmitter` object `myEvtEmt`. Then we use `.on()` method to register a customized event `start` with a callback function, in this case the call back is an anonymous function taking the data and printing it out. In the third statement, we trigger the event by calling `.emit()` method.

In case the callback does not take any parameters, the `.emit()` will only be called with one argument, i.e. the event.

```
myEvtEmt.on("end", () => {
  console.log(`bye`);
});

myEvtEmt.emit("end");
```

When will the event loop stop?

For Node.js, some of the events are expecting a closure (closure as in general term, not JS closure), e.g. API call, file operation, the Node.js event loop will keep looping until all expected closures have returned and there is no more pending tasks in the micro nor the macro queues.

For browsers, the event loop will continue as it waits for the user's next input.

Further Readings

- An interactive example showing how promise and callback works.

```
https://jakearchibald.com/2015/tasks-microtasks-queues-and-schedules/
```

- Synchronous vs Asynchronous JavaScript – Call Stack, Promises, and More

```
https://www.freecodecamp.org/news/synchronous-vs-asynchronous-in-
javascript/#:~:text=JavaScript%20is%20a%20single%2Dthreaded,language%20with%20lot
```

- Node.js Event loop behavior

```
https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick
```

- Node.js API for timer functions, `setTimeout`, `setImmediate`, `setInterval`.

```
https://nodejs.org/en/docs/guides/timers-in-node
```

Corhort Exercise (Graded)

Using the callstack-microtask-macrotask table, illustrate the execution of the following JavaScript program

```javascript
import EventEmitter from "events";

const ev1 = new EventEmitter();
const ev2 = new EventEmitter();

let count = 0;

let promise1 = new Promise((resolve, reject) => {
  resolve(count);
});

let promise2 = new Promise((resolve, reject) => {
  resolve(count);
});

function foo(x) {
  return new Promise((resolve, reject) => {
    if (x > 10) {
      resolve();
    } else if (x % 2 == 0) {
      ev1.emit("run", ++x);
    } else {
      ev2.emit("run", ++x);
    }
  });
}

ev1.on("run", (data) => {
  setImmediate(() => {
    console.log(`data ${data} received by ev1`);
    promise2.then(foo(data));
  });
});

ev2.on("run", (data) => {
  setImmediate(() => {
    console.log(`data ${data} received by ev2`);
    promise1.then(foo(data));
  });
});

ev2.emit("run", count);
```