Custom Middleware | NPM Libs

By : Parveen & Abhishek

Agenda

- Custom Middleware
- Promises
- Async/await
- Child-process
- Buffer/Stream
- Request

Middleware : Overview

Middleware functions are functions that have access to the request object (req), the response object (res), and the next function in the application's request-response cycle. The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.

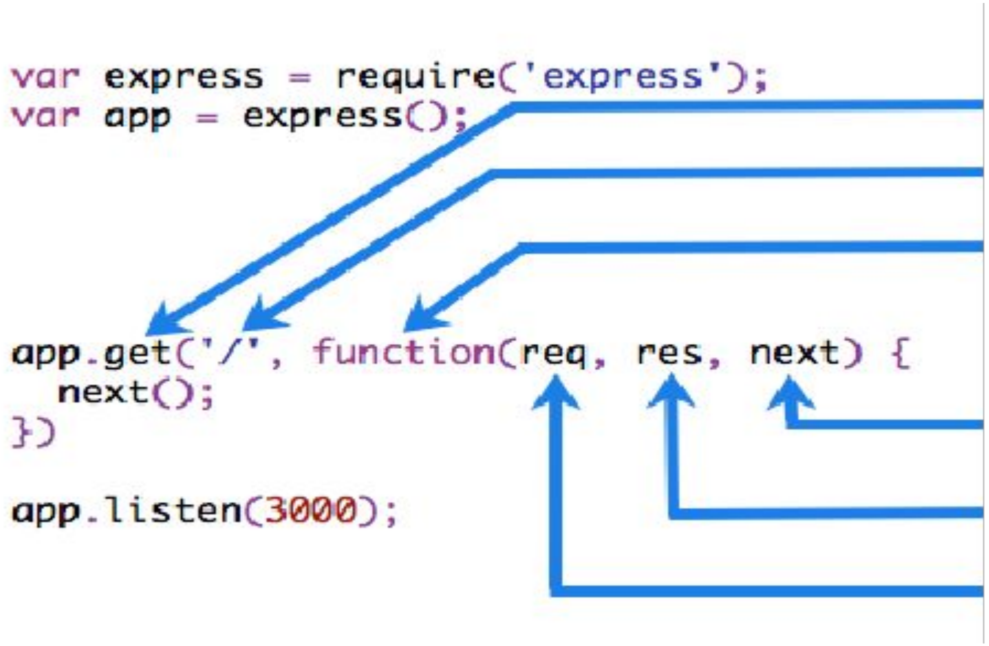
Middleware functions can perform the following tasks:

Execute any code.
Make changes to the request and the response objects.
End the request-response cycle.
Call the next middleware in the stack.

Understanding Middleware

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

The following figure shows the elements of a middleware function call:



```
var express = require('express');
var app = express();

app.get('/', function(req, res, next) {
  next();
})

app.listen(3000);
```

HTTP method for which the middleware function applies.
Path (route) for which the middleware function applies.
The middleware function.

Callback argument to the middleware function, called "next" by convention.
HTTP response argument to the middleware function, called "res" by convention.
HTTP request argument to the middleware function, called "req" by convention.

Middleware

```
//server.js
//logging middleware
function middleware(req, res, next) {
    console.info('inside middleware');
    //do something with req, call next once done
    next();
}
//how to use it
app.use(middleware);
//add static server
app.use('/static', Express.static('views'));
```

Custom Middleware function

Example of checking the request method and if we the method is “GET” we returns the response with GET method not allowed on this server.

```
module.exports = function(req,res,next) {
   if(req.method === 'GET') {
       res.end('GET method not supported');
   } else {
       next();
   }
};
```

Middleware function requestChecker

Every time the app receives a request, it checks the request method if it is GET if yes the return response from middleware itself.
The order of middleware loading is important: middleware functions that are loaded first are also executed first.

If “requestChecker” is loaded after the route to the root path, the request never reaches it and the app will not able to check the method of “req”, because the route handler of the root path terminates the request-response cycle.

The middleware function requestChecker simply checks the method of request, then passes on the request to the next middleware function in the stack by calling the next() function.

Middleware functions request time.

Next, we’ll create a middleware function called “requestTime” and add it as a property called requestTime to the request object.

```
const express = require('express')
const app = express()

const requestTime = (req, res, next) => {
  req.requestTime = Date.now();
  next();
}

app.use(requestTime);
app.get('/', (req, res) => {
  var responseText = 'Hello World!<br>'
  responseText += '<small>Requested at: ' + req.requestTime + '</small>'
  res.send(responseText);
});

app.listen(3000);
```

Types of Express Middleware

- Application level middleware `app.use`
- Router level middleware `router.use`
- Built-in middleware `express.static,express.json,express.urlencoded`
- Error handling middleware `app.use(err,req,res,next)`
- Third party middleware `bodyparser,cookieparser`

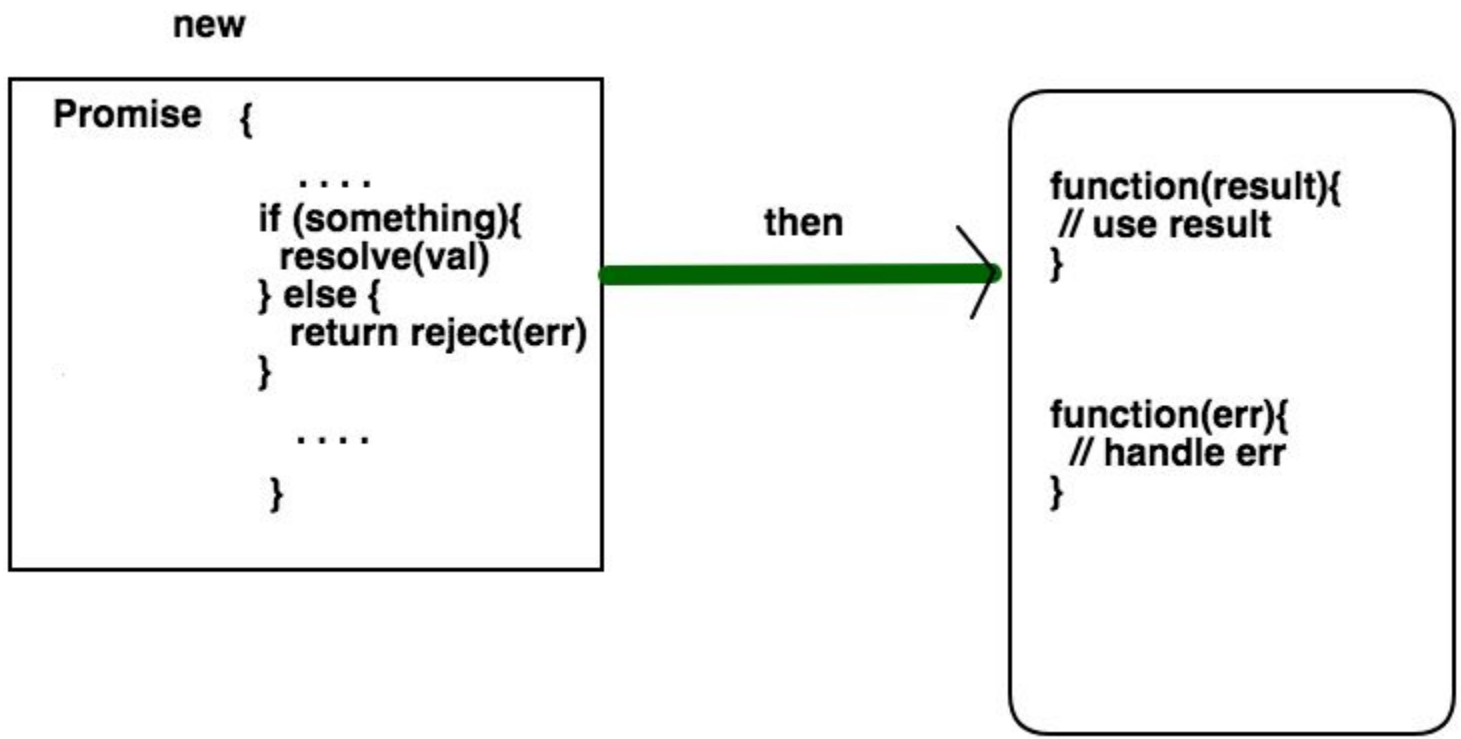
Promises

What is a Promise?

According to the official website:

- A `Promise` is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers to an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of the final value, the asynchronous method returns a *promise* for the value at some point in the future.
- In simple words “A promise is a word taken for some action, the other party who gave the promise might fulfill it or deny it”. In the case of fulfilling, the promise gets resolved, and in another case, it gets rejected.
- Promises are kind of design patterns to remove the usage of unintuitive callbacks.

Promises



```
Promise  {
            ....
          if (something){
            resolve(val)
          } else {
             return reject(err)
          }
            ....
          }
```

```
function(result){
 // use result
}

function(err){
  // handle err
}
```

Promises

- A promise can be created in our JavaScript code. Or else it can be returned from an external node package
- Any promise that performs async operations should call any one of the two methods **resolve** or **reject.** The code logic should take care of when and where to call these functions. If the operation is successful, pass that data to the code that uses that promise, otherwise pass error
- The code which uses a promise should call **then** function on that promise. It takes two anonymous functions as parameters. The first function executes if the promise is resolved and the second function executes if promise is rejected

Promises

```
 var promise = new Promise((resolve, reject)=>{

 ...

});
```

Promises

`Promise.all[]`

We can make a sequence of promises for doing things in a particular order. We can use **Promise.all** function which takes a list of promises in the given order and returns another promise which we can use a **then** method to conclude the logic.

*Note: **Promise.all** fails if any one of the Promise got rejected. It is an **and** operation between promise fulfillments*

Async/Await

What is Async/Await?

- The newest way to write asynchronous code in JavaScript.
- It is non blocking (just like promises and callbacks).
- Async/Await was created to simplify the process of working with and writing chained promises.
- Async functions return a Promise. If the function throws an error, the Promise will be rejected. If the function returns a value, the Promise will be resolved.

Async/Await

Syntax

Writing an async function is quite simple. You just need to add the `async` keyword prior to `function`:

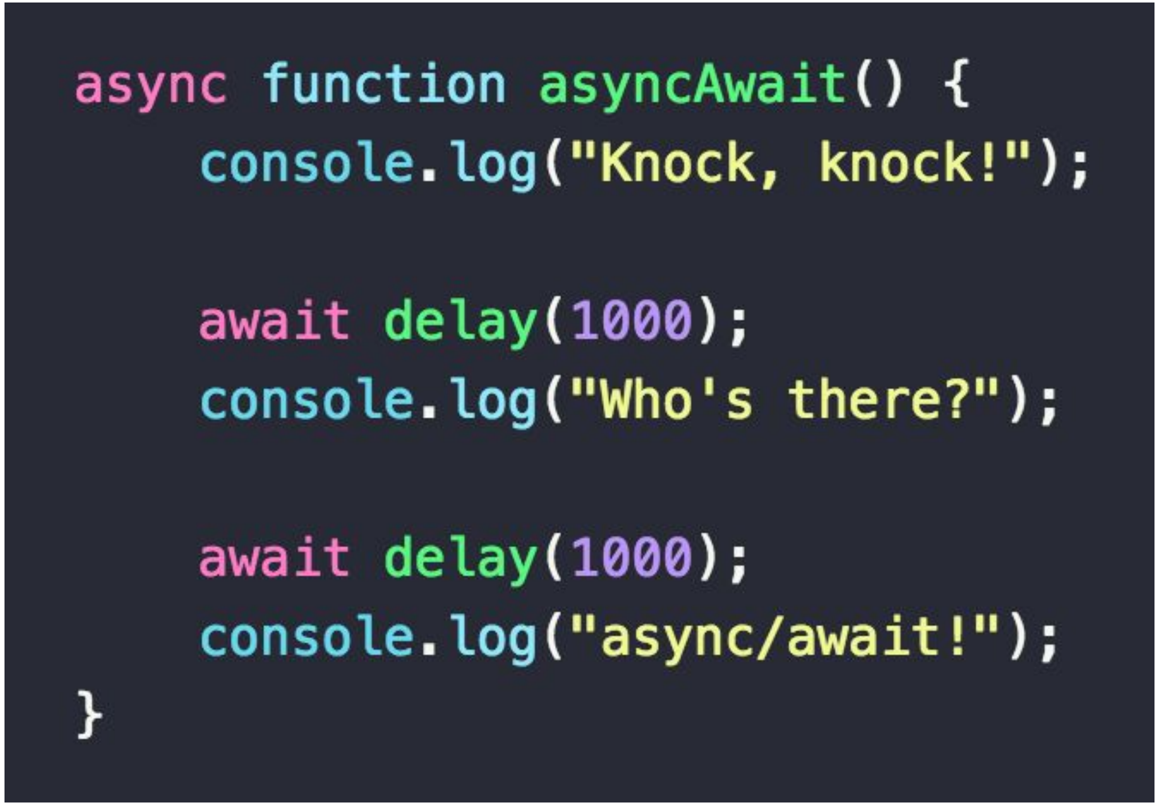
`Normal Function`

```
function add(x,y){
 return x + y;
}
// Async Function
async function add(x,y){
 return x + y;
}
```

Await

Async functions can make use of the `await` expression. This will pause the `async` function and wait for the Promise to resolve prior to moving on.

Async/Await



```
async function asyncAwait() {
    console.log("Knock, knock!");

    await delay(1000);
    console.log("Who's there?");

    await delay(1000);
    console.log("async/await!");
}
```

Child Process

Node.js runs in a single-thread mode, but it uses an event-driven paradigm to handle concurrency. It also facilitates creation of child processes to leverage parallel processing on multi-core CPU based systems.

Child processes always have three streams child.stdin, child.stdout, and child.stderr which may be shared with the stdio streams of the parent process.

Node provides child_process module which has the following three major ways to create a child process.

- exec − child_process.exec method runs a command in a shell/console and buffers the output.
- spawn − child_process.spawn launches a new process with a given command.
- fork − The child_process.fork method is a special case of the spawn() to create child processes.

The exec() method

child_process.exec method runs a command in a shell and buffers the output. It has the following signature −

```
child_process.exec(command[, options], callback)
```

Parameters

- command (String) The command to run, with space-separated arguments
- options (Object) may comprise one or more of the following options −
    - cwd (String) Current working directory of the child process
    - env (Object) Environment key-value pairs
    - encoding (String) (Default: 'utf8')
    - shell (String) Shell to execute the command with (Default: '/bin/sh' on UNIX, 'cmd.exe' on Windows)
    - timeout (Number) (Default: 0)
    - maxBuffer (Number) (Default: 200*1024)
    - killSignal (String) (Default: 'SIGTERM')
    - uid (Number) Sets the user identity of the process.
    - gid (Number) Sets the group identity of the process.
- callback The function gets three arguments error, stdout, and stderr which are called with the output when the process terminates.

The spawn() method

```
child_process.spawn(command[, args][, options])
```

Parameters

- command (String) The command to run
- args (Array) List of string arguments
- options (Object) may comprise one or more of the following options –
  - cwd (String) Current working directory of the child process.
  - env (Object) Environment key-value pairs.
  - stdio (Array) String Child's stdio configuration.
  - customFds (Array) Deprecated File descriptors for the child to use for stdio.
  - detached (Boolean) The child will be a process group leader.
  - uid (Number) Sets the user identity of the process.
  - gid (Number) Sets the group identity of the process.
  - The spawn() method returns streams (stdout & stderr) and it should be used when the process returns a volume amount of data. spawn() starts receiving the response as soon as the process starts executing.

The spawn() method

The `spawn` function launches a command in a new process and we can use it to pass that command any arguments.

The fork() method

```
child_process.fork(modulePath[, args][, options])
```

The `fork` function is a variation of the `spawn` function for spawning node processes. The biggest difference between `spawn` and `fork` is that a communication channel is established to the child process when using `fork`, so we can use the `send` function on the forked process along with the global `process` object itself to exchange messages between the parent and forked processes. We do this through the `EventEmitter` module interface.

Buffer/Stream

Buffer

Buffer class provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.

Buffer instance is not resizable.

Buffer/Stream

Methods & Properties

- Constructor / from / alloc / allocUnsafe / allocUnsafeSlow
- .fill
- .write
- .isBuffer
- .concat
- .compare
- .equals
- .includes
- .indexOf

Buffer/Stream

Methods & Properties

- .lastIndexOf
- .keys
- .entries
- .length
- .slice
- .toJSON
- .toString

Buffer/Stream

Stream

Streams are objects that let you read data from a source or write data to a destination in continuous fashion. There are four types of streams:

- Readable
- Writable
- Duplex
- Transform

Buffer/Stream

Events

- data
- end
- error
- finish

Examples:

https://www.tutorialspoint.com/nodejs/nodejs_streams.htm

Requests

Some popular modules to make HTTP requests:

- HTTP
- request
- axios
- superagent
- got

Exercise

1.Create on custom middleware to validate the session of user if valid session allow to access request otherwise send back.

2. Create Example for Buffer, Streams and pipe.

3. Create API using Async/Await or Promises, fetch github profile and followers.
(`https://api.github.com/users/<username>`)

4.Child Process execFile(); example .

Thank You

Any queries? Open to Q&A