<epam>

Introduction to backend development with Node.js and Express.js

November 21, 2019

AGENDA

- Introduction
- Modules
  - Modules Types
  - Importing and exporting
  - Module Context
- Event loop
  - Single-threaded
  - Asynchronous non-blocking operations
  - Libuv
  - Callback hell
- Express framework
  - Middleware concept
  - express() application
  - Request/Response
  - Router

What is Node.js?

What is Node.js?

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine.

- Event-driven
- Non-blocking I/O
- Single threaded
- Open-source & cross-platform

Node.js Use cases

1. Package management through npm, bower, jspm, etc.
2. Development tooling (module management with webpack, task running and automation through grunt or gulp, linters like eslint or jslint, etc)
3. Command line tools like rimraf
4. Data Streaming Applications
5. Data Intensive Real-time Applications (DIRT)
6. JSON APIs based Applications

There are plenty other special use cases, like building neural networks, or chat bots, or really anything you can thing of.

Why Node.js?

- Non-blocking code
- Fast processing
- Concurrent request handling
- One environment
- Easy to learn
- Popularity and community

Node.js is not suited for CPU-intensive tasks.
It is suited for I/O stuff.

Installation and first run

1. Go to https://nodejs.org/
2. Download the latest version
3. Run Installation
4. Open CLI(terminal)
5. Print “node -v”
6. Enjoy

AGENDA

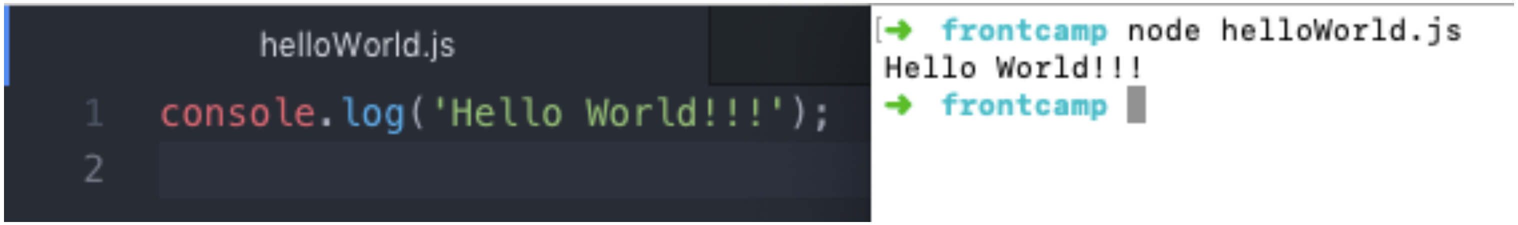
- ~~Introduction~~
- Modules
  - Modules Types
  - Importing and exporting
  - Module Context
- Event loop
  - Single-threaded
  - Asynchronous non-blocking operations
  - Libuv
  - Callback hell
- Express framework
  - Middleware concept
  - express() application
  - Request/Response
  - Router

Modules System

Consider modules to be the same as JavaScript libraries. A set of functions you want to include in your application.

- Each file is a module
- Three types of modules
    1. Core modules
    2. Local modules
    3. Third party modules
- Another three types of modules
    1. JS
    2. JSON
    3. NODE(C++)
- Has its own context
- Implements CommonJS modules standard (*ES6 modules - experimental)
- Cache initialized modules

Hello World in Node.js

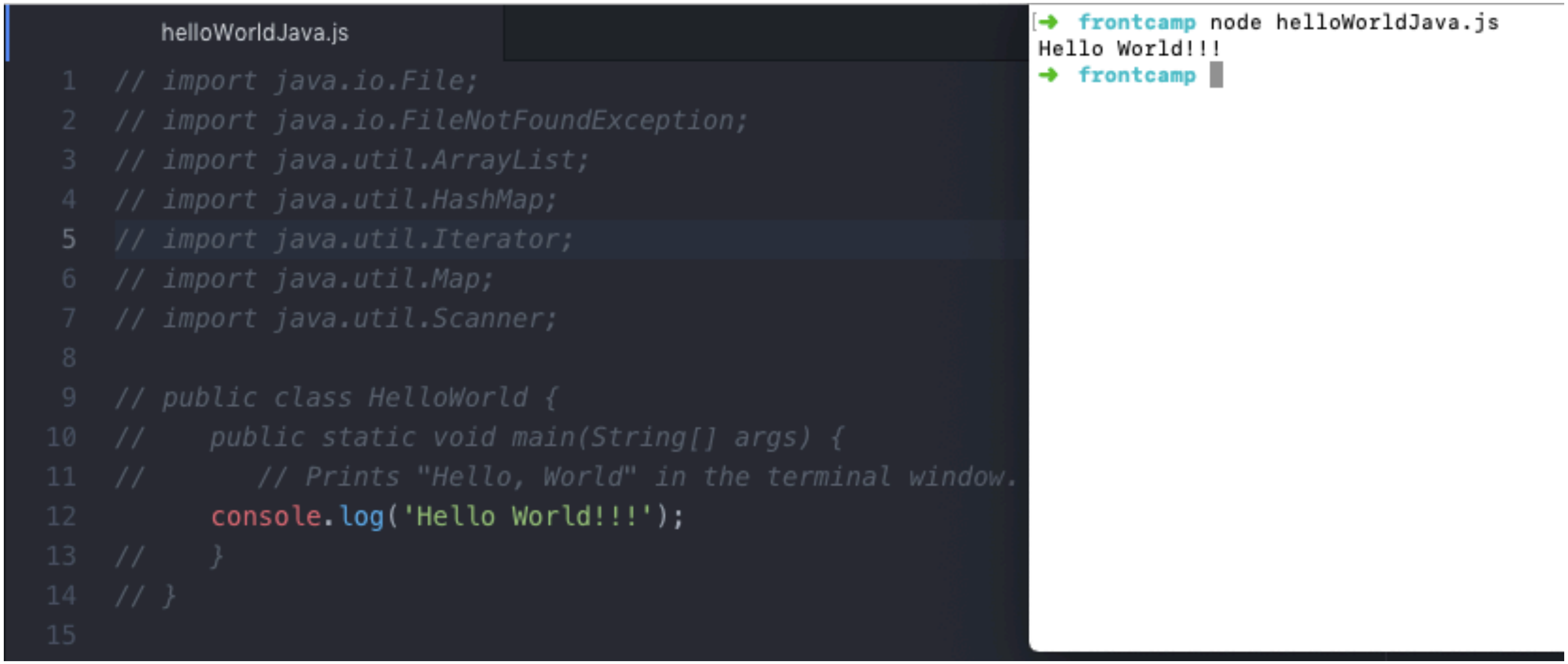


helloWorld.js

```
console.log('Hello World!!!');

```

```
➜  frontcamp node helloWorld.js
Hello World!!!
➜  frontcamp 
```

Hello World in Node.js



```
// import java.io.File;
// import java.io.FileNotFoundException;
// import java.util.ArrayList;
// import java.util.HashMap;
// import java.util.Iterator;
// import java.util.Map;
// import java.util.Scanner;

// public class HelloWorld {
//    public static void main(String[] args) {
//       // Prints "Hello, World" in the terminal window.
      console.log('Hello World!!!');
//    }
// }

```

```
[➜  frontcamp node helloWorldJava.js
Hello World!!!
➜  frontcamp █
```

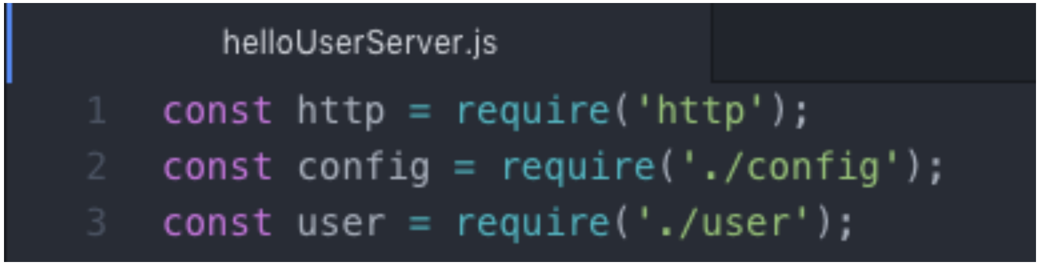
Built-in modules

Node.js has a set of built-in modules which you can use without any further installation.

Name	Description
events	To handle events
fs	To handle the file system
http	To make Node.js act as an HTTP server
https	To make Node.js act as an HTTPS server
child_process	To run a child process
buffer	To handle binary data
os	Provides information about the operation system
path	To handle file paths
… others	

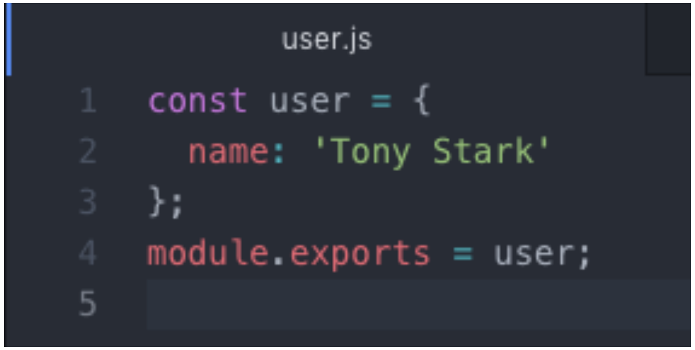
Exporting and requiring

To include a module, use the require() function with the name of the module.

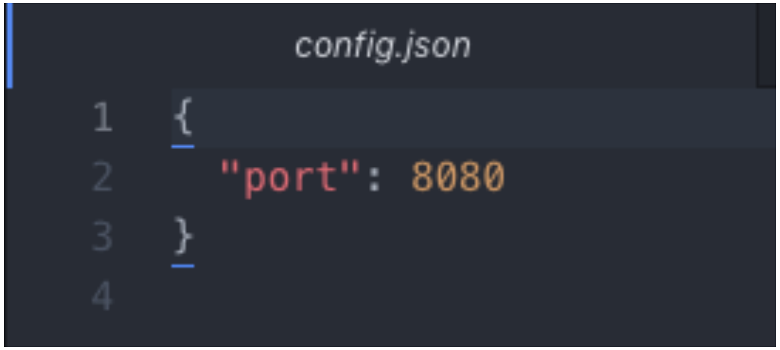


```
const http = require('http');
const config = require('./config');
const user = require('./user');
```

Use the module.exports to make properties and methods available outside the module file.

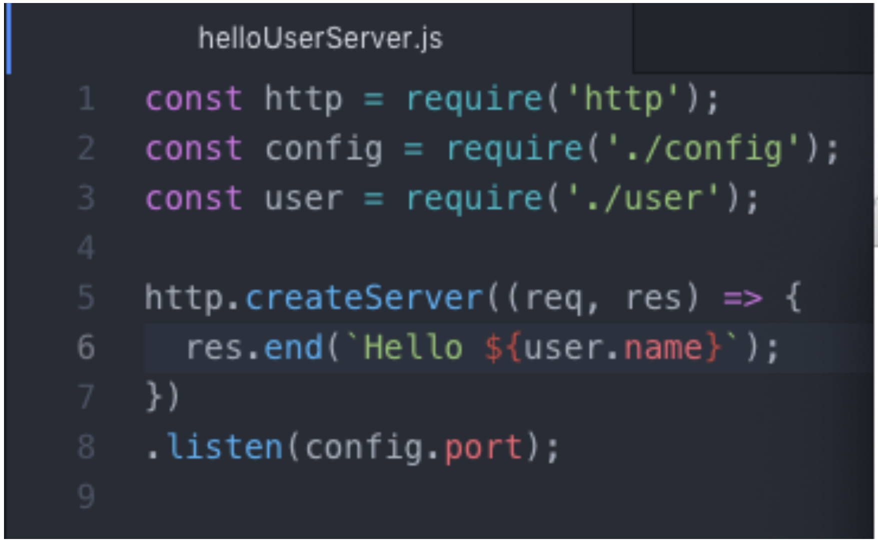


```
const user = {
  name: 'Tony Stark'
};
module.exports = user;
```

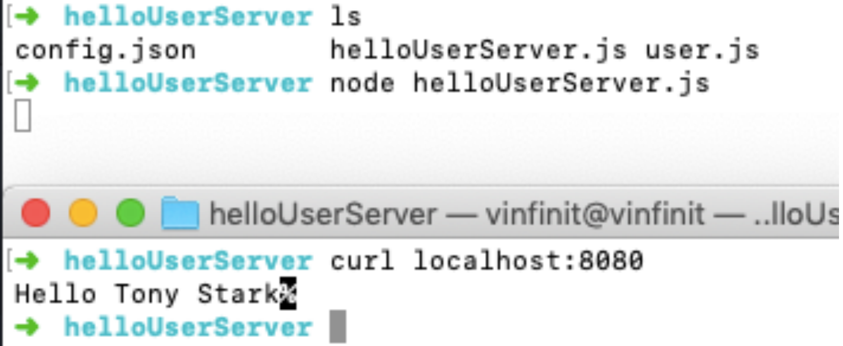


```
{
  "port": 8080
}
```

Hello User server



```
const http = require('http');
const config = require('./config');
const user = require('./user');

http.createServer((req, res) => {
  res.end(`Hello ${user.name}`);
})
.listen(config.port);

```

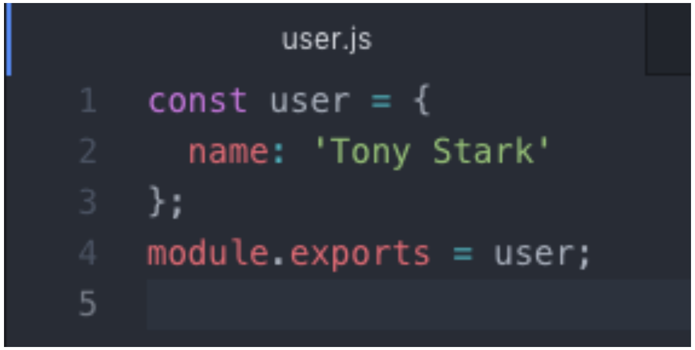


```
[➜  helloUserServer ls
config.json          helloUserServer.js user.js
[➜  helloUserServer node helloUserServer.js
```

```
[➜  helloUserServer curl localhost:8080
Hello Tony Stark%
➜  helloUserServer
```

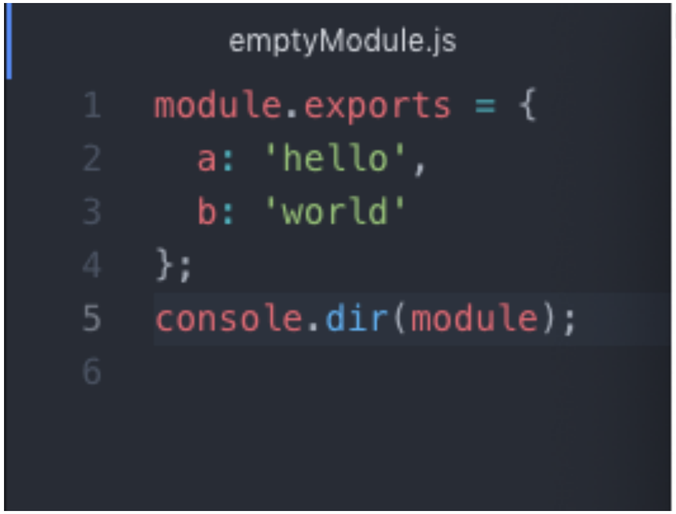
Module scope variables

Variable	Description
__dirname	The directory name of the current module.
__filename	The file name of the current module.
exports	A reference to the module.exports that is shorter to type.
module	A reference to the current module.
require()	To require modules.



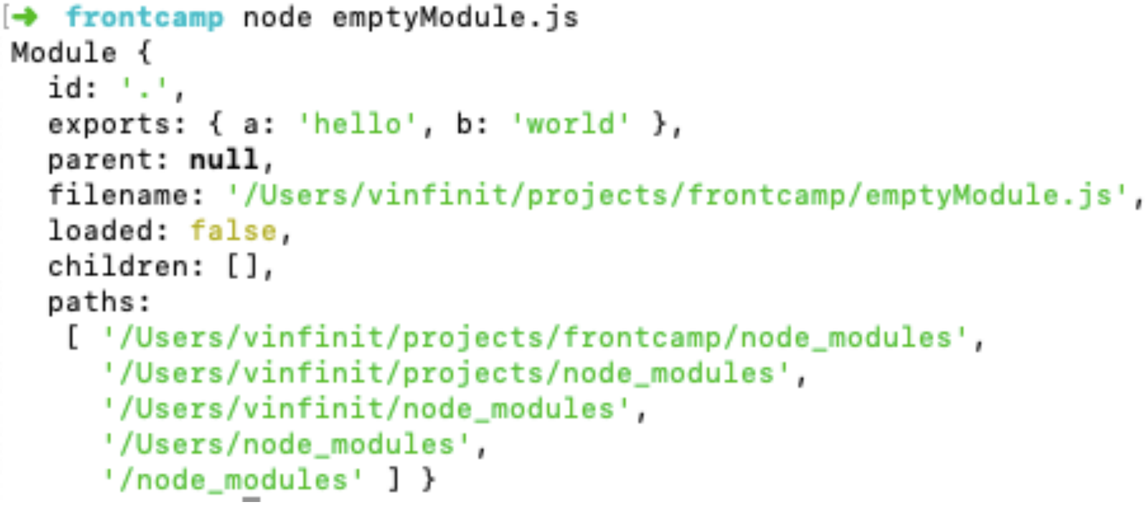
```
const user = {
  name: 'Tony Stark'
};
module.exports = user;
```

Module object



emptyModule.js

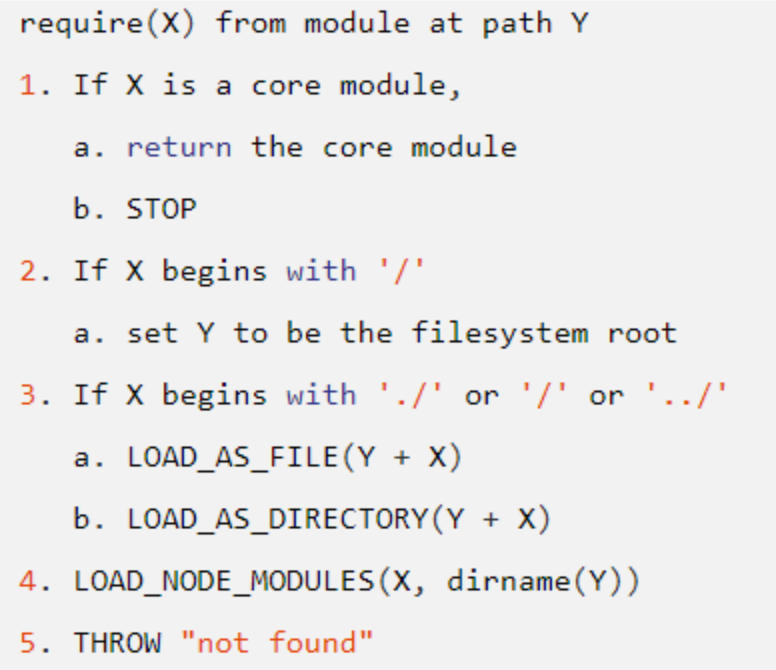
```
module.exports = {
  a: 'hello',
  b: 'world'
};
console.dir(module);
```



```
[➜  frontcamp node emptyModule.js
Module {
  id: '.',
  exports: { a: 'hello', b: 'world' },
  parent: null,
  filename: '/Users/vinfinit/projects/frontcamp/emptyModule.js',
  loaded: false,
  children: [],
  paths:
   [ '/Users/vinfinit/projects/frontcamp/node_modules',
     '/Users/vinfinit/projects/node_modules',
     '/Users/vinfinit/node_modules',
     '/Users/node_modules',
     '/node_modules' ] }
```

Modules resolving

High-level* algorithm:



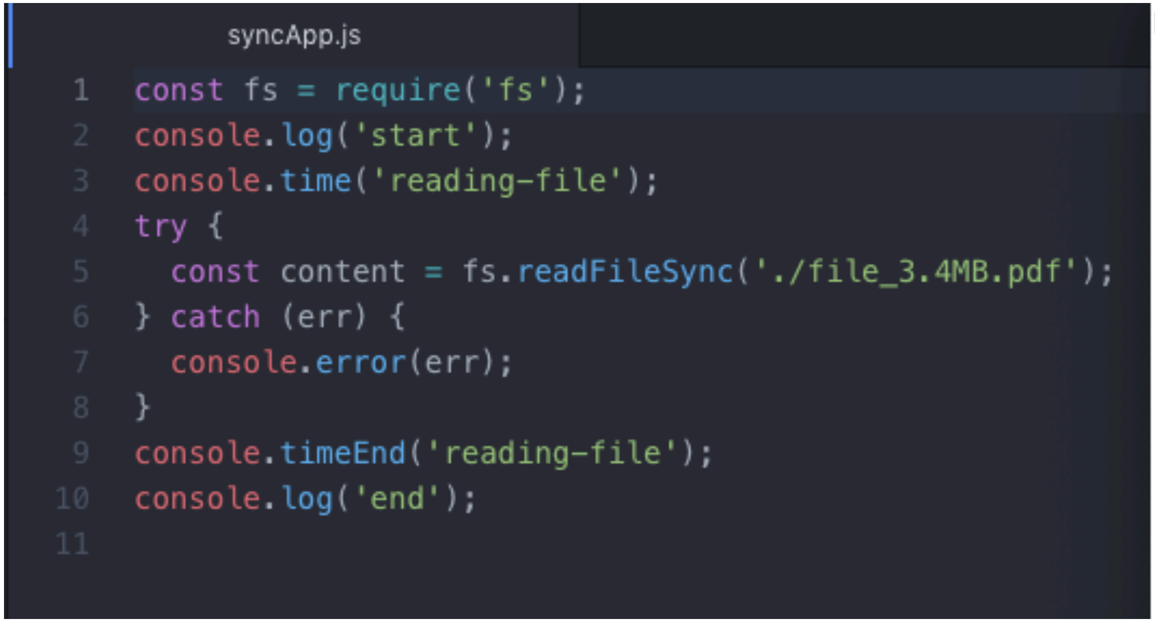
```
require(X) from module at path Y
1. If X is a core module,
   a. return the core module
   b. STOP
2. If X begins with '/'
   a. set Y to be the filesystem root
3. If X begins with './' or '/' or '../'
   a. LOAD_AS_FILE(Y + X)
   b. LOAD_AS_DIRECTORY(Y + X)
4. LOAD_NODE_MODULES(X, dirname(Y))
5. THROW "not found"
```

AGENDA

- ~~Introduction~~
- ~~Modules~~
  - ~~Modules Types~~
  - ~~Importing and exporting~~
  - ~~Module Context~~
- Event loop
  - Single-threaded
  - Asynchronous non-blocking operations
  - Libuv
  - Callback hell
- Express framework
  - Middleware concept
  - express() application
  - Request/Response
  - Router

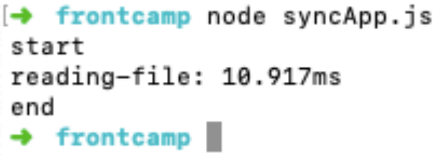
Node.js Single Threaded

Your code in Node.js is single-threaded.



syncApp.js

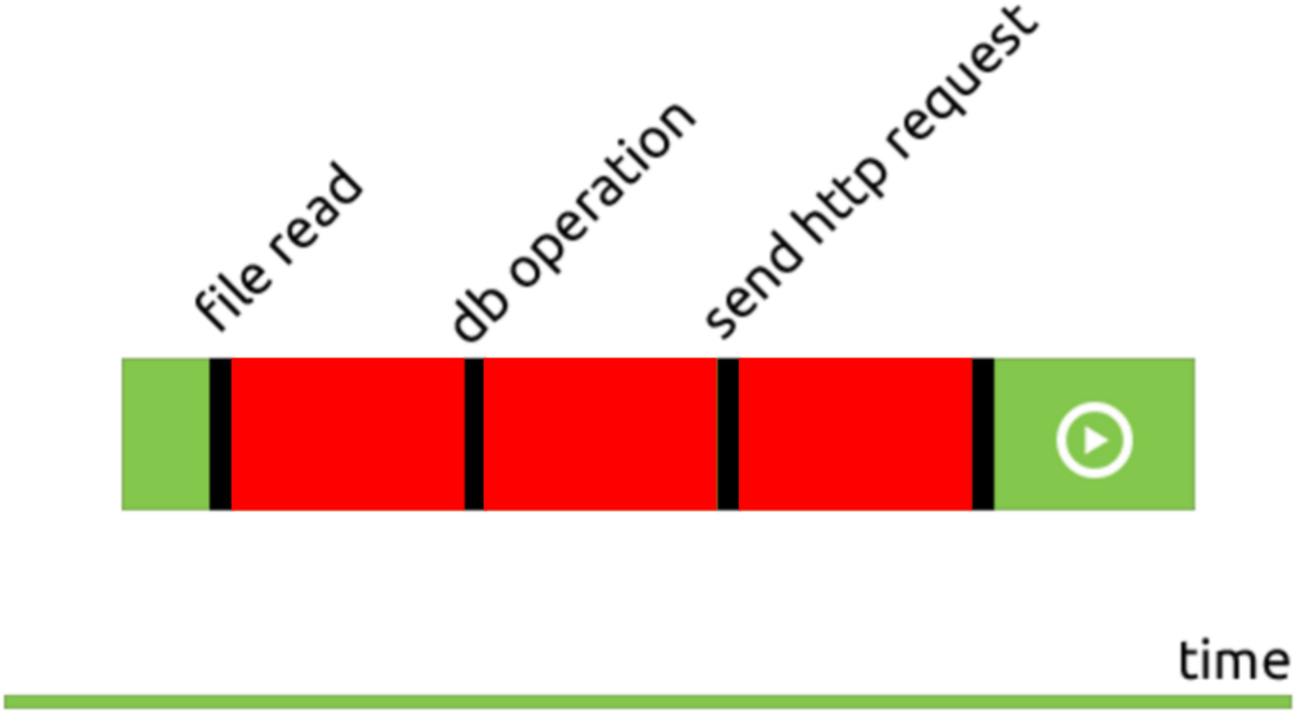
```
const fs = require('fs');
console.log('start');
console.time('reading-file');
try {
  const content = fs.readFileSync('./file_3.4MB.pdf');
} catch (err) {
  console.error(err);
}
console.timeEnd('reading-file');
console.log('end');
```



```
[➜  frontcamp node syncApp.js
start
reading-file: 10.917ms
end
➜  frontcamp █
```

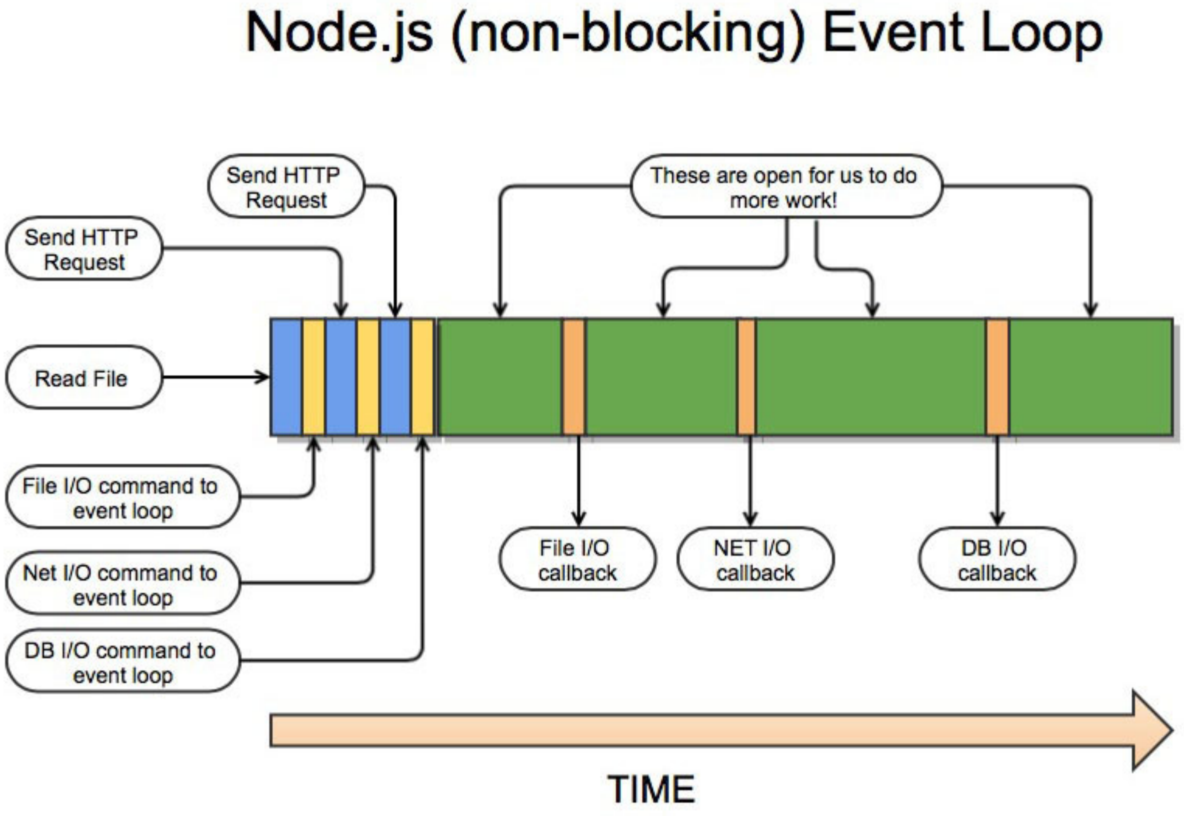
Node.js Single Threaded

How Node JS handles concurrent requests with Single-Threaded model?

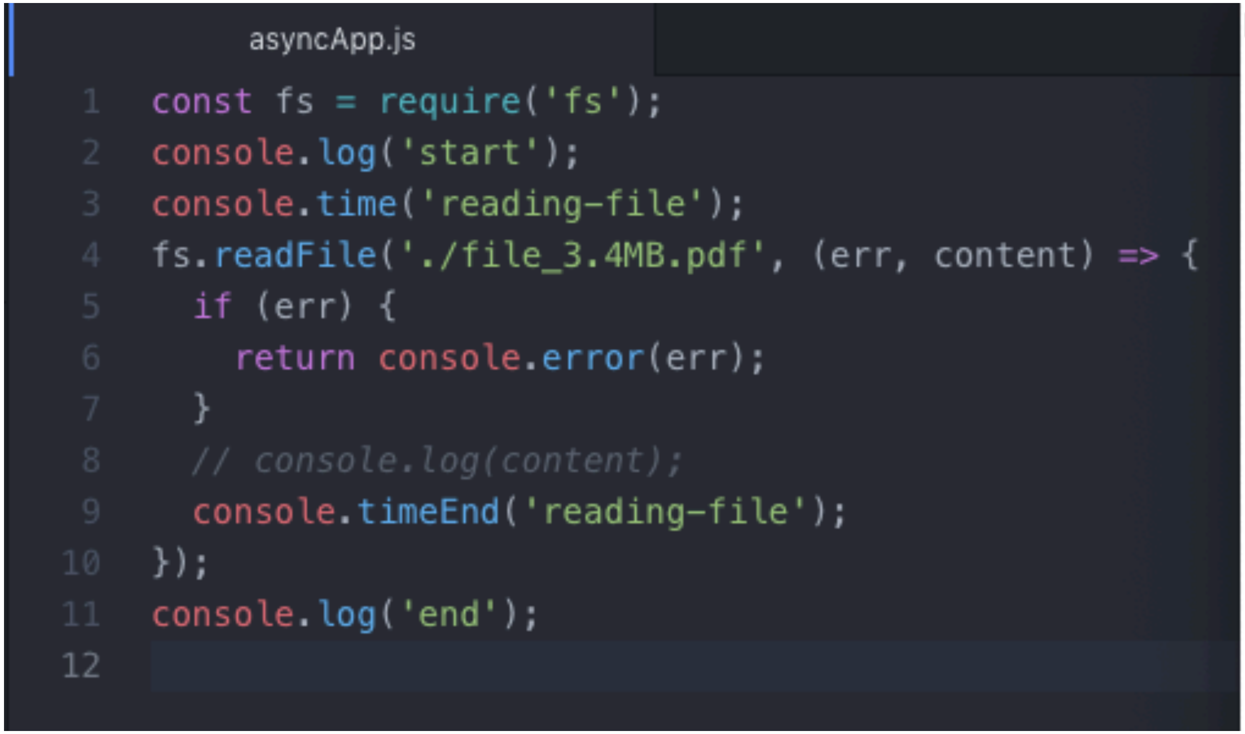


Node.js Event Loop

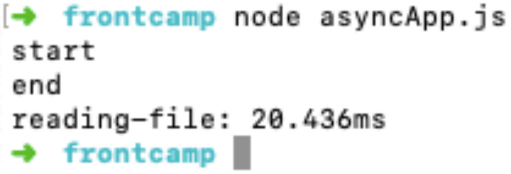
Node JS Processing model mainly based on Javascript Event based model with Javascript callback mechanism.


Node.js (non-blocking) Event Loop
Send HTTP Request
Send HTTP Request
These are open for us to do more work!
Read File
File I/O command to event loop
Net I/O command to event loop
DB I/O command to event loop
File I/O callback
NET I/O callback
DB I/O callback
TIME

Node.js Asynchronous Programming



```javascript
const fs = require('fs');
console.log('start');
console.time('reading-file');
fs.readFile('./file_3.4MB.pdf', (err, content) => {
  if (err) {
    return console.error(err);
  }
  // console.log(content);
  console.timeEnd('reading-file');
});
console.log('end');
```

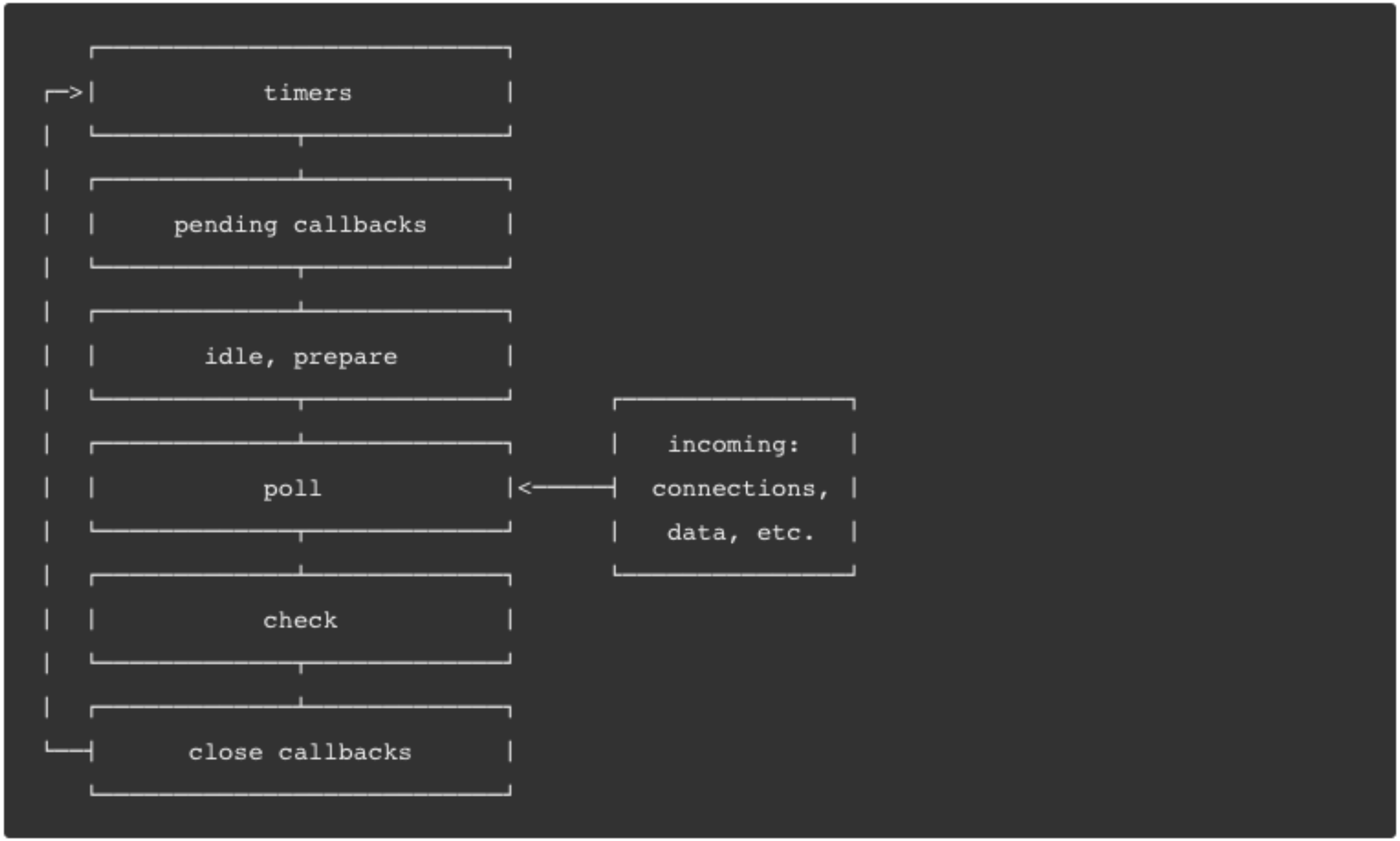


```
➜ frontcamp node asyncApp.js
start
end
reading-file: 20.436ms
➜ frontcamp
```

AGENDA

- ~~Introduction~~
- ~~Modules~~
  - ~~Modules Types~~
  - ~~Importing and exporting~~
  - ~~Module Context~~
- ~~Event loop~~
  - ~~Single-threaded~~
  - ~~Asynchronous non-blocking operations~~
  - Libuv
  - Callback hell
- Express framework
  - Middleware concept
  - express() application
  - Request/Response
  - Router

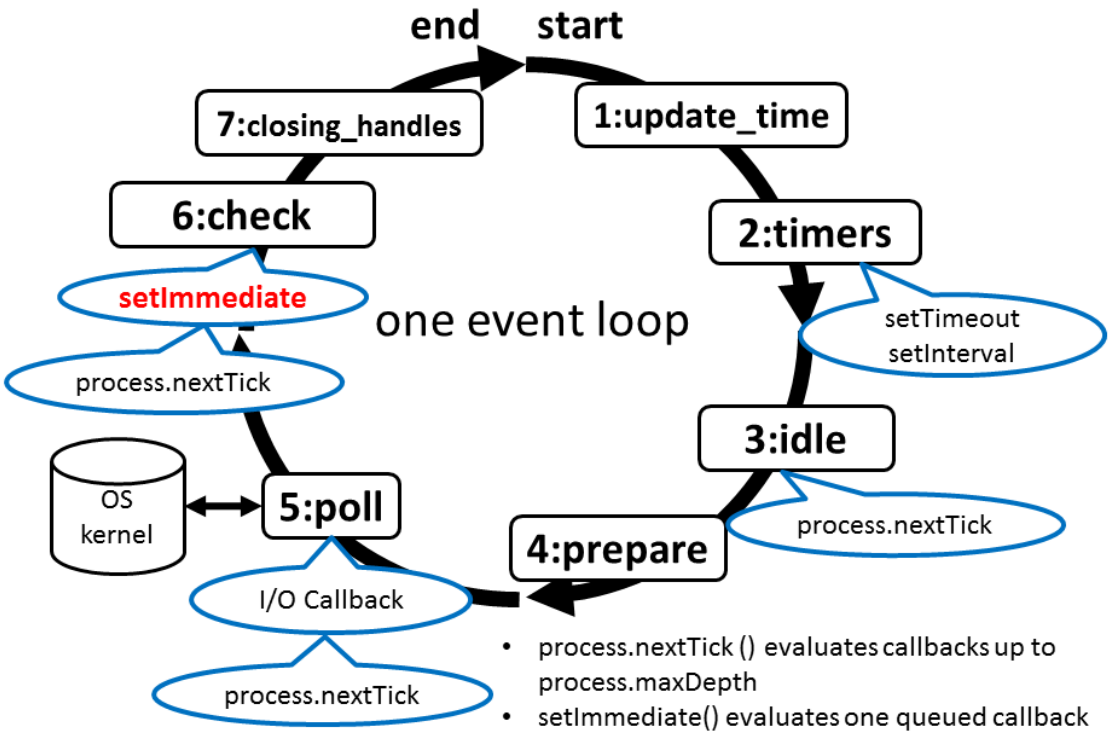
Node.js Event Loop (libuv)



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

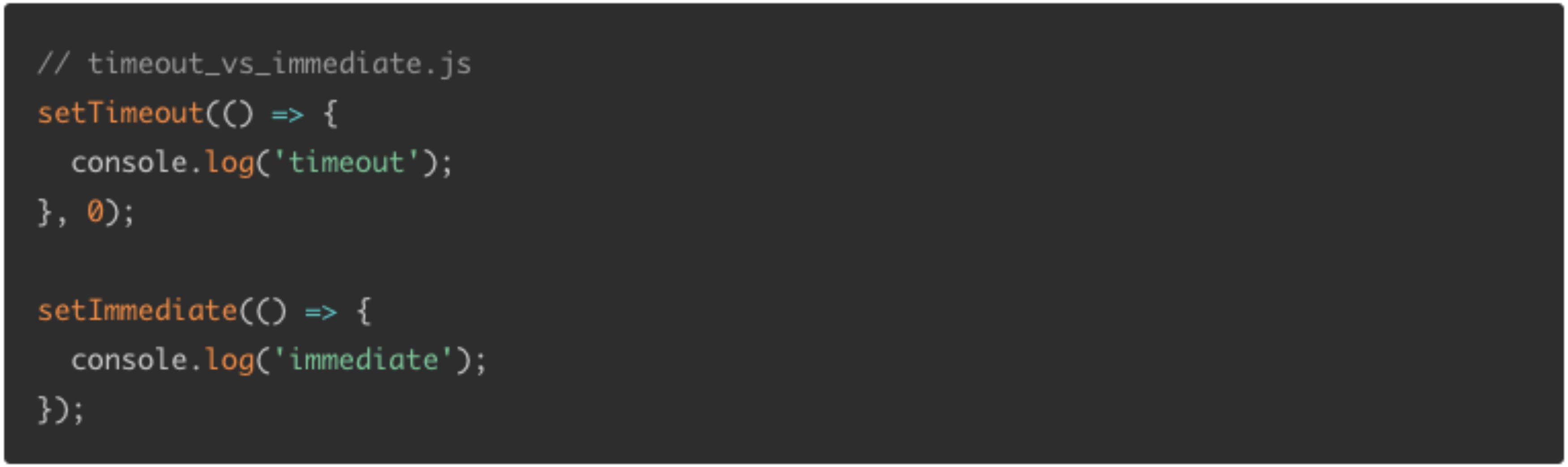
process.nextTick(), setImmediate(), setTimeout()

- *process.nextTick()* - fires immediately on the same phase
- *setImmediate()* - fires on the following operation on 'check' phase of the event loop
- *setTimeout()* - schedules a script to be run after a minimum threshold in ms



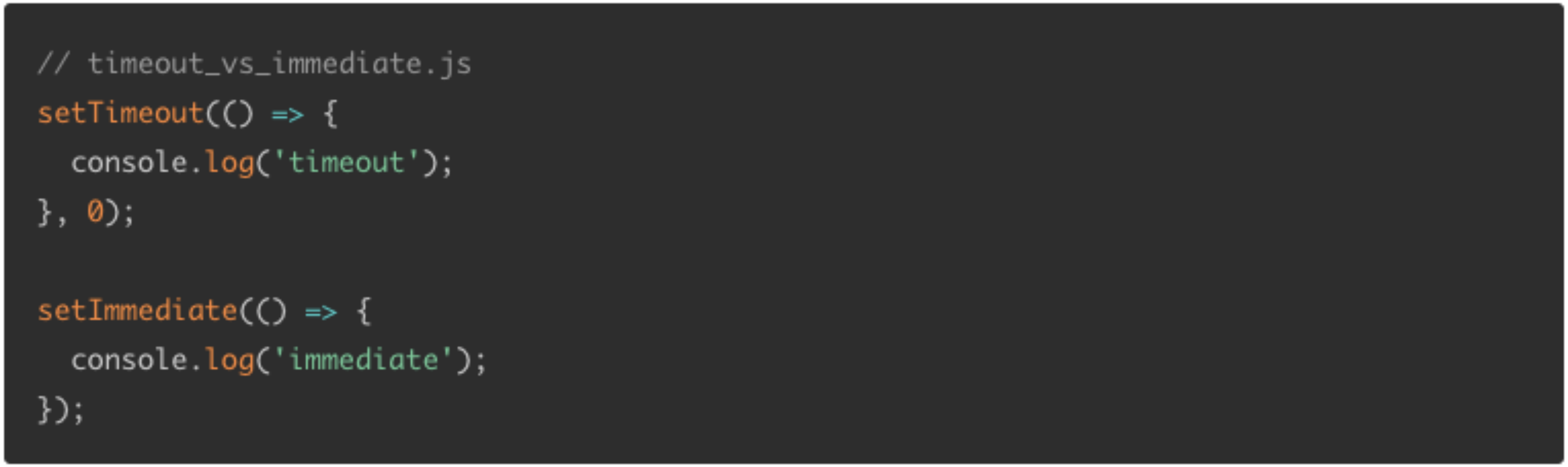
- process.nextTick () evaluates callbacks up to process.maxDepth
- setImmediate() evaluates one queued callback

setImmediate() vs setTimeout()

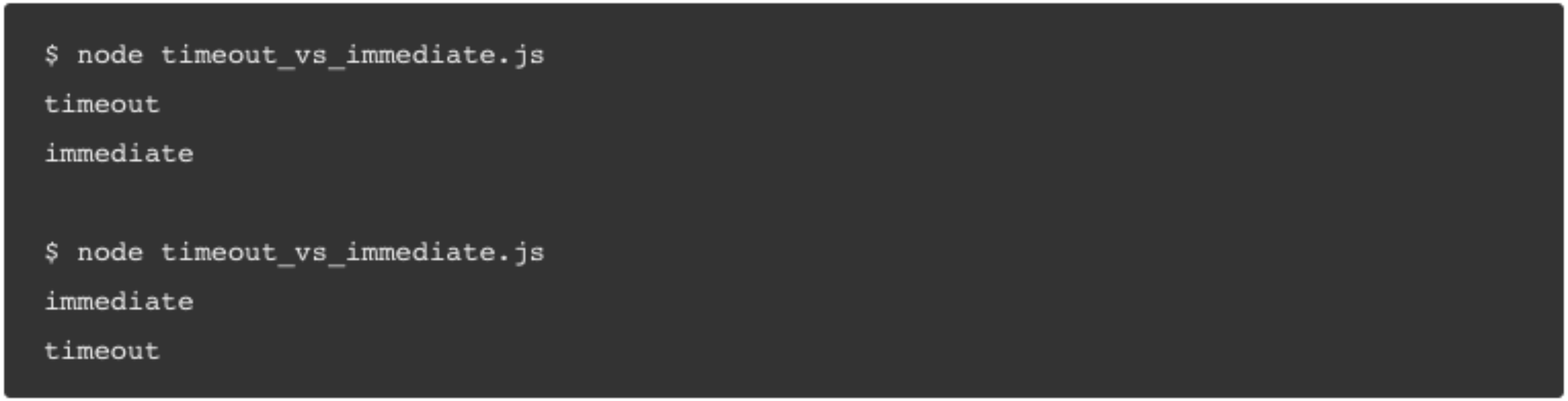


```
// timeout_vs_immediate.js
setTimeout(() => {
  console.log('timeout');
}, 0);

setImmediate(() => {
  console.log('immediate');
});
```

setImmediate() vs setTimeout()

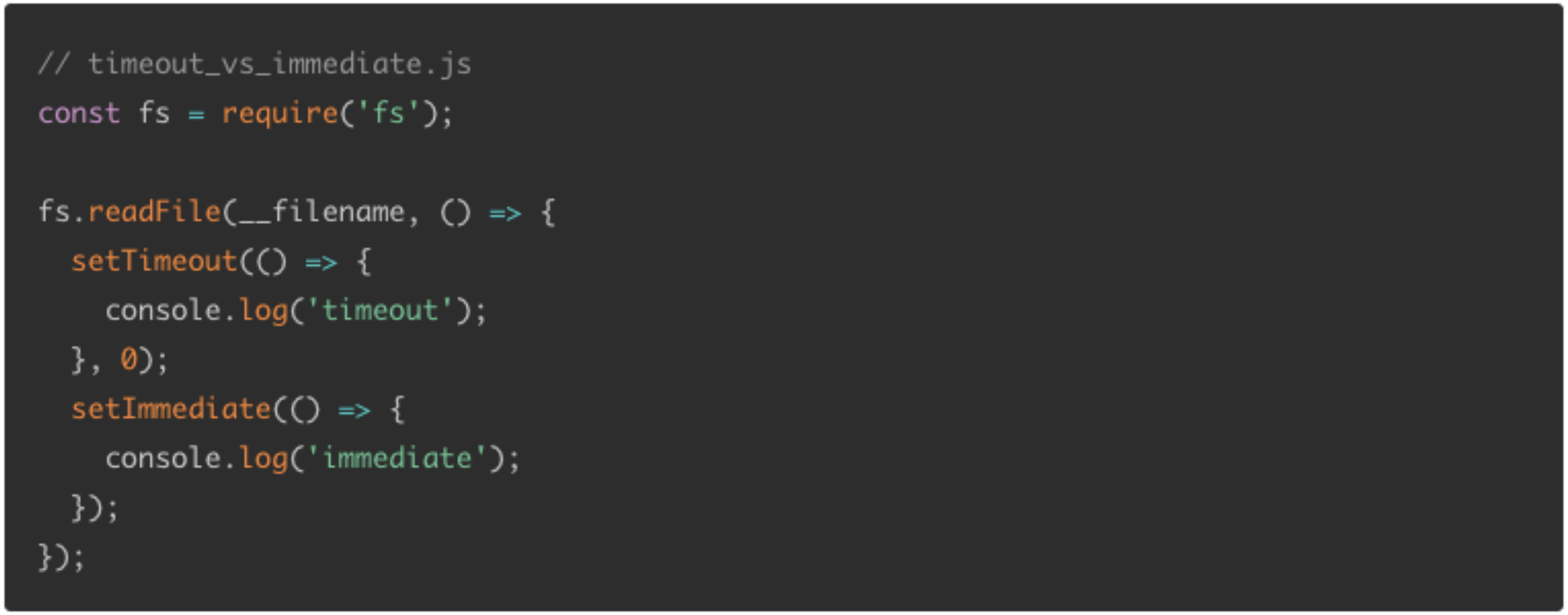


```
// timeout_vs_immediate.js
setTimeout(() => {
  console.log('timeout');
}, 0);

setImmediate(() => {
  console.log('immediate');
});
```



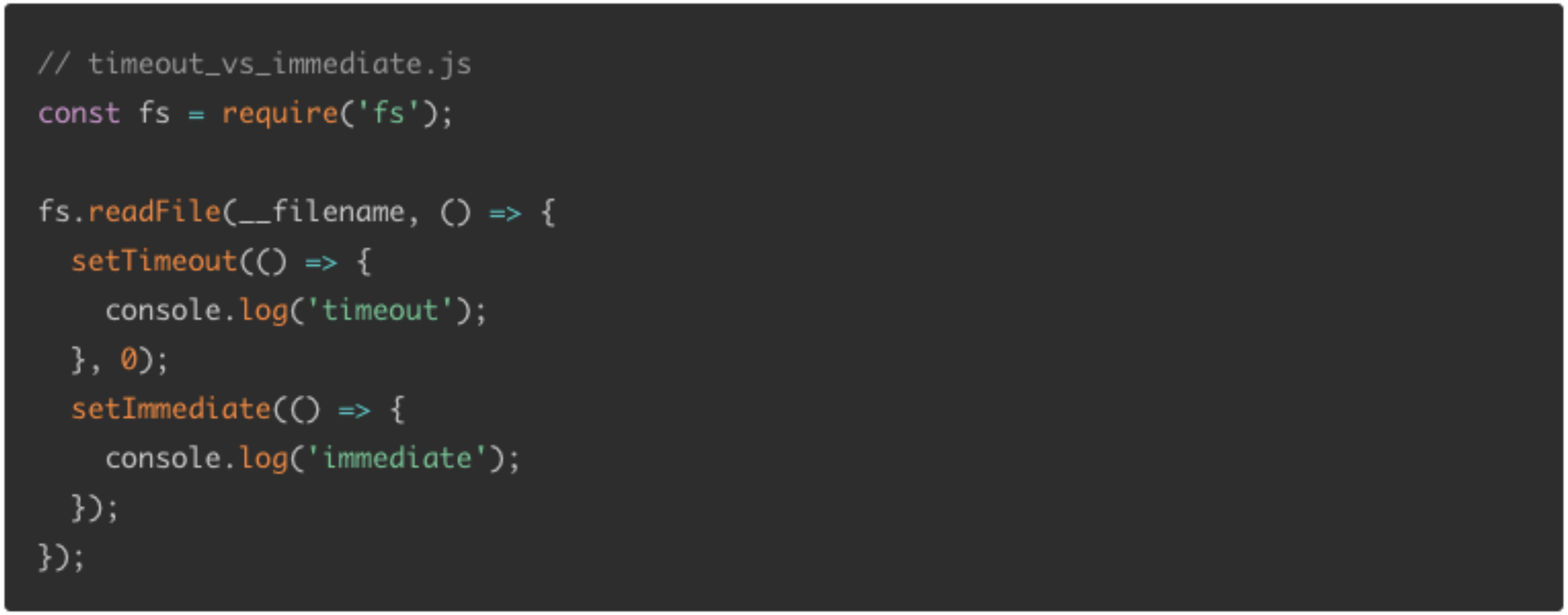
```
$ node timeout_vs_immediate.js
timeout
immediate

$ node timeout_vs_immediate.js
immediate
timeout
```

setImmediate() vs setTimeout()

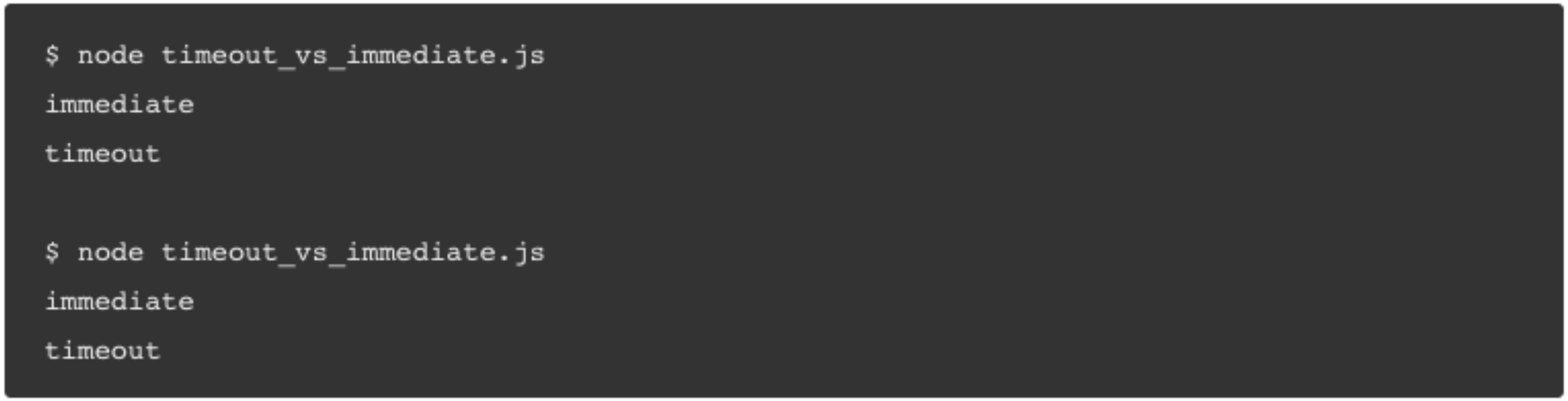


```
// timeout_vs_immediate.js
const fs = require('fs');

fs.readFile(__filename, () => {
  setTimeout(() => {
    console.log('timeout');
  }, 0);
  setImmediate(() => {
    console.log('immediate');
  });
});
```

setImmediate() vs setTimeout()

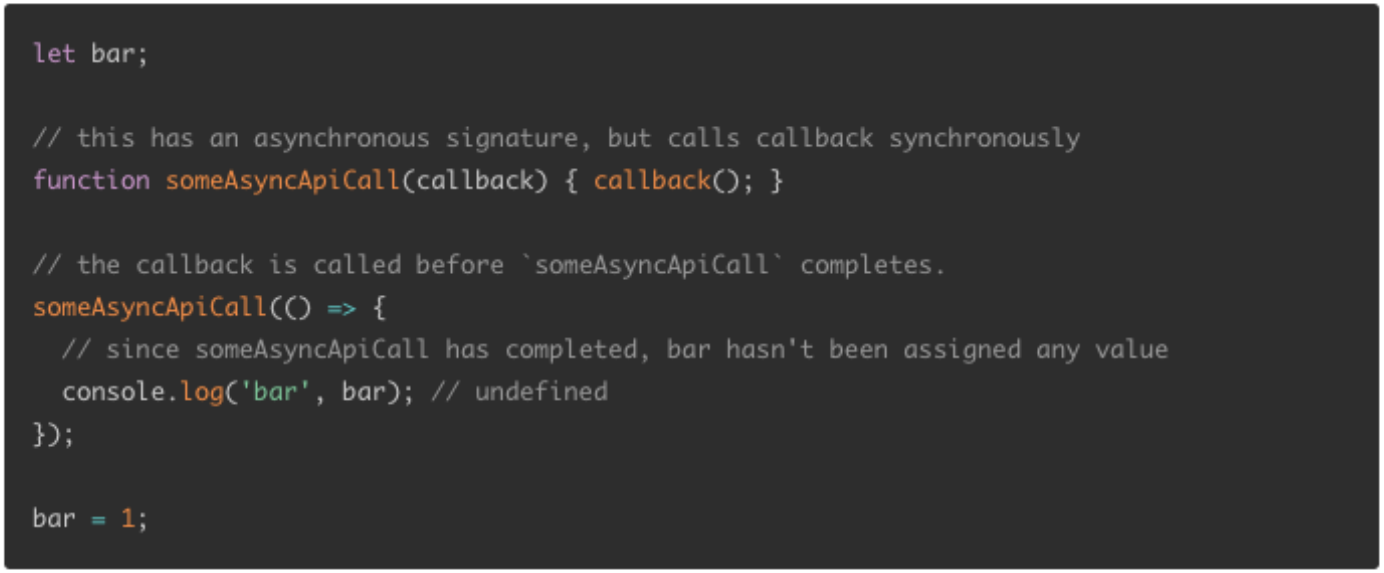


```
// timeout_vs_immediate.js
const fs = require('fs');

fs.readFile(__filename, () => {
  setTimeout(() => {
    console.log('timeout');
  }, 0);
  setImmediate(() => {
    console.log('immediate');
  });
});
```

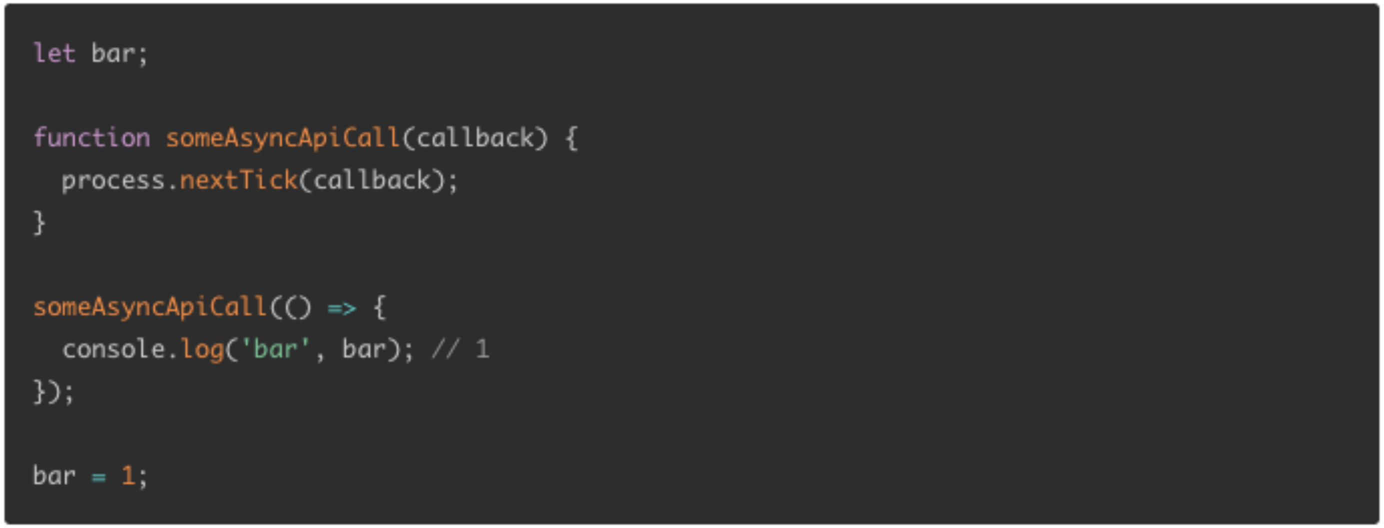


```
$ node timeout_vs_immediate.js
immediate
timeout

$ node timeout_vs_immediate.js
immediate
timeout
```

process.nextTick()



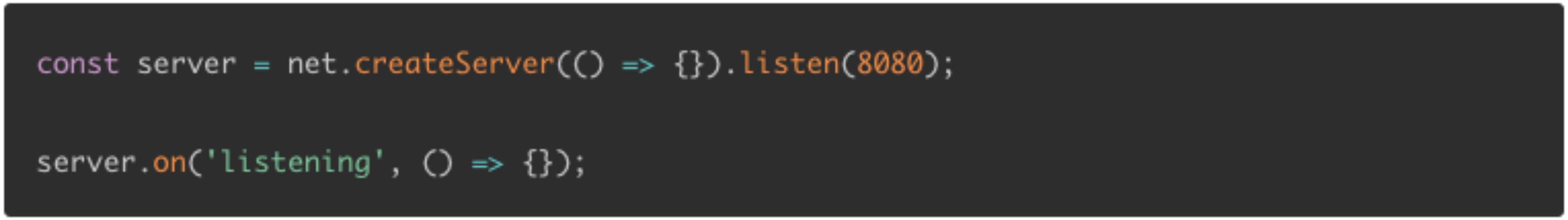
```
let bar;

// this has an asynchronous signature, but calls callback synchronously
function someAsyncApiCall(callback) { callback(); }

// the callback is called before `someAsyncApiCall` completes.
someAsyncApiCall(() => {
  // since someAsyncApiCall has completed, bar hasn't been assigned any value
  console.log('bar', bar); // undefined
});

bar = 1;
```



```
let bar;

function someAsyncApiCall(callback) {
  process.nextTick(callback);
}

someAsyncApiCall(() => {
  console.log('bar', bar); // 1
});

bar = 1;
```

process.nextTick()

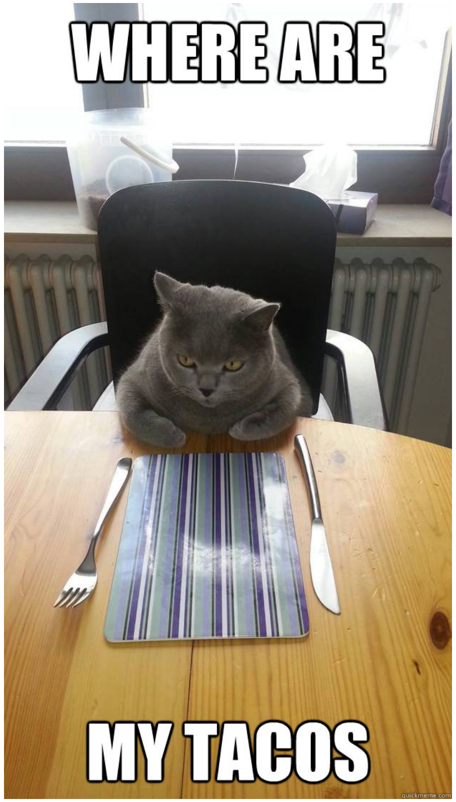
Real world example of using *process.nextTick()*



```
const server = net.createServer(() => {}).listen(8080);

server.on('listening', () => {});
```

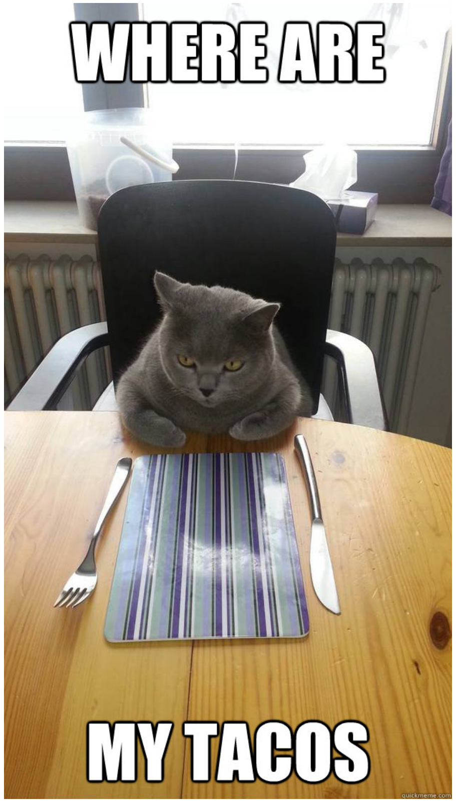
process.nextTick()

What can happen if you exaggerate with using of *process.nextTick()*



process.nextTick()

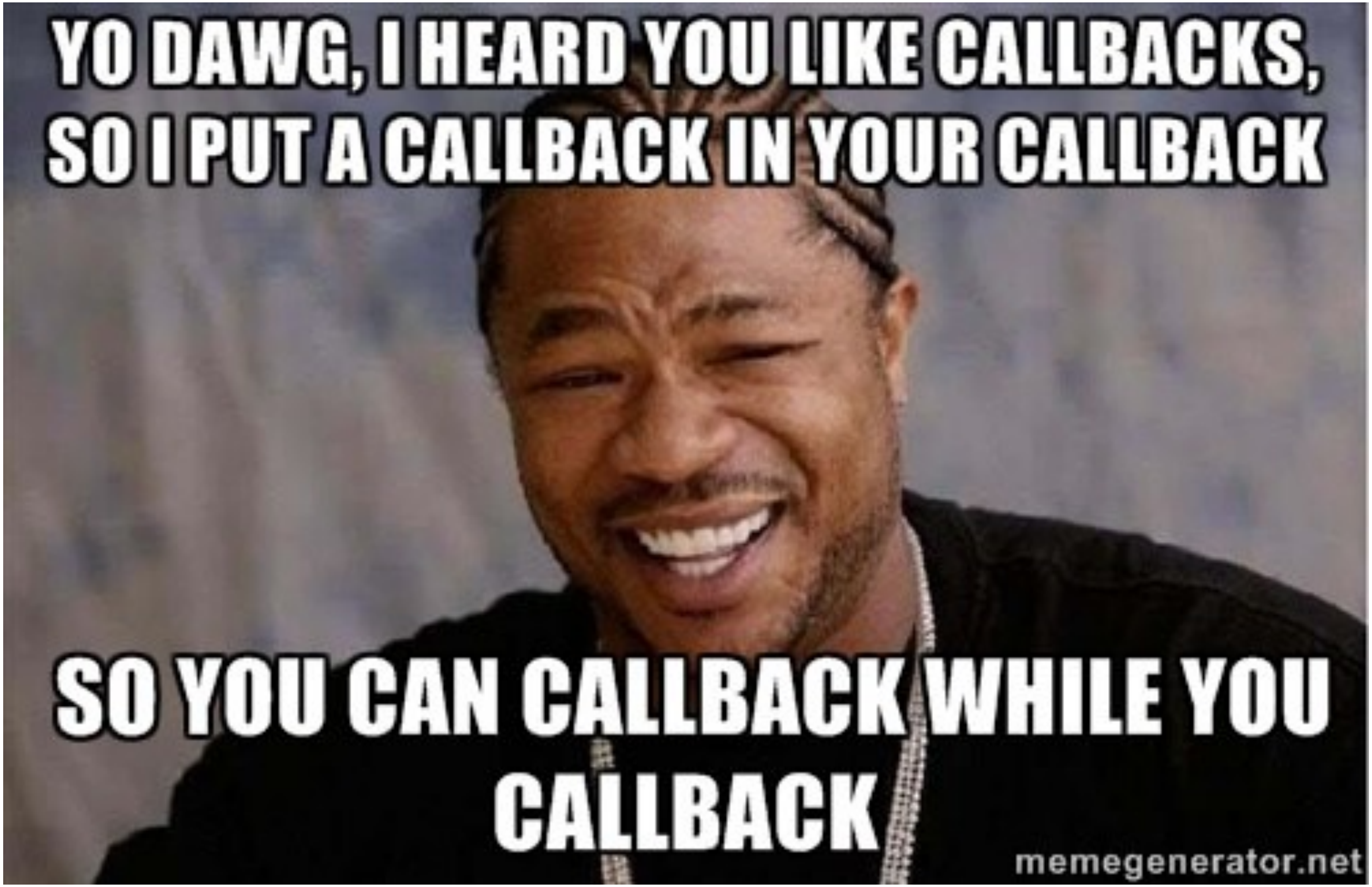
What can happen if you exaggerate with using of *process.nextTick()*



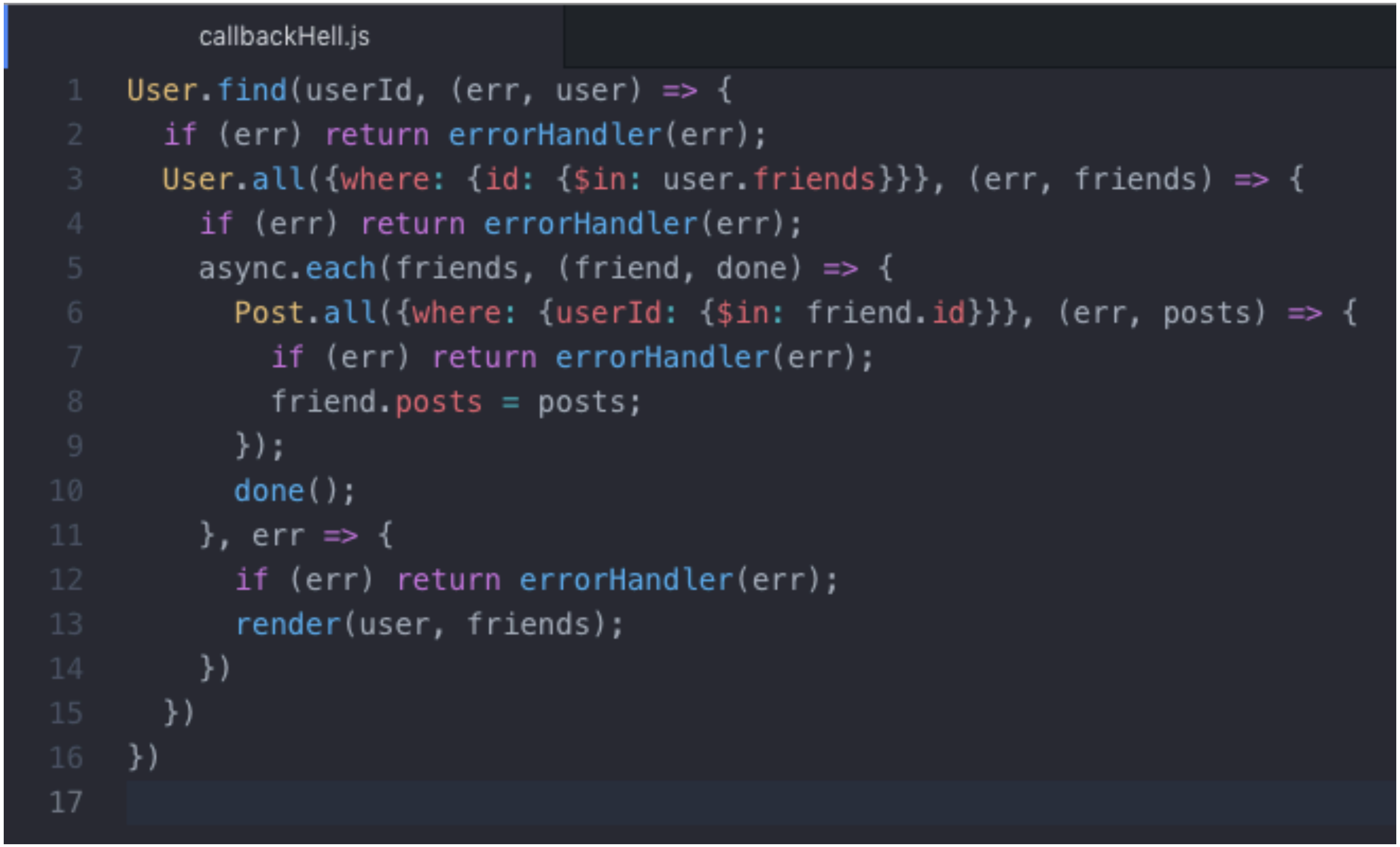
Event loop starving

AGENDA

- ~~Introduction~~
- ~~Modules~~
  - ~~Modules Types~~
  - ~~Importing and exporting~~
  - ~~Module Context~~
- ~~Event loop~~
  - ~~Single-threaded~~
  - ~~Asynchronous non-blocking operations~~
  - ~~Libuv~~
  - Callback hell
- Express framework
  - Middleware concept
  - express() application
  - Request/Response
  - Router

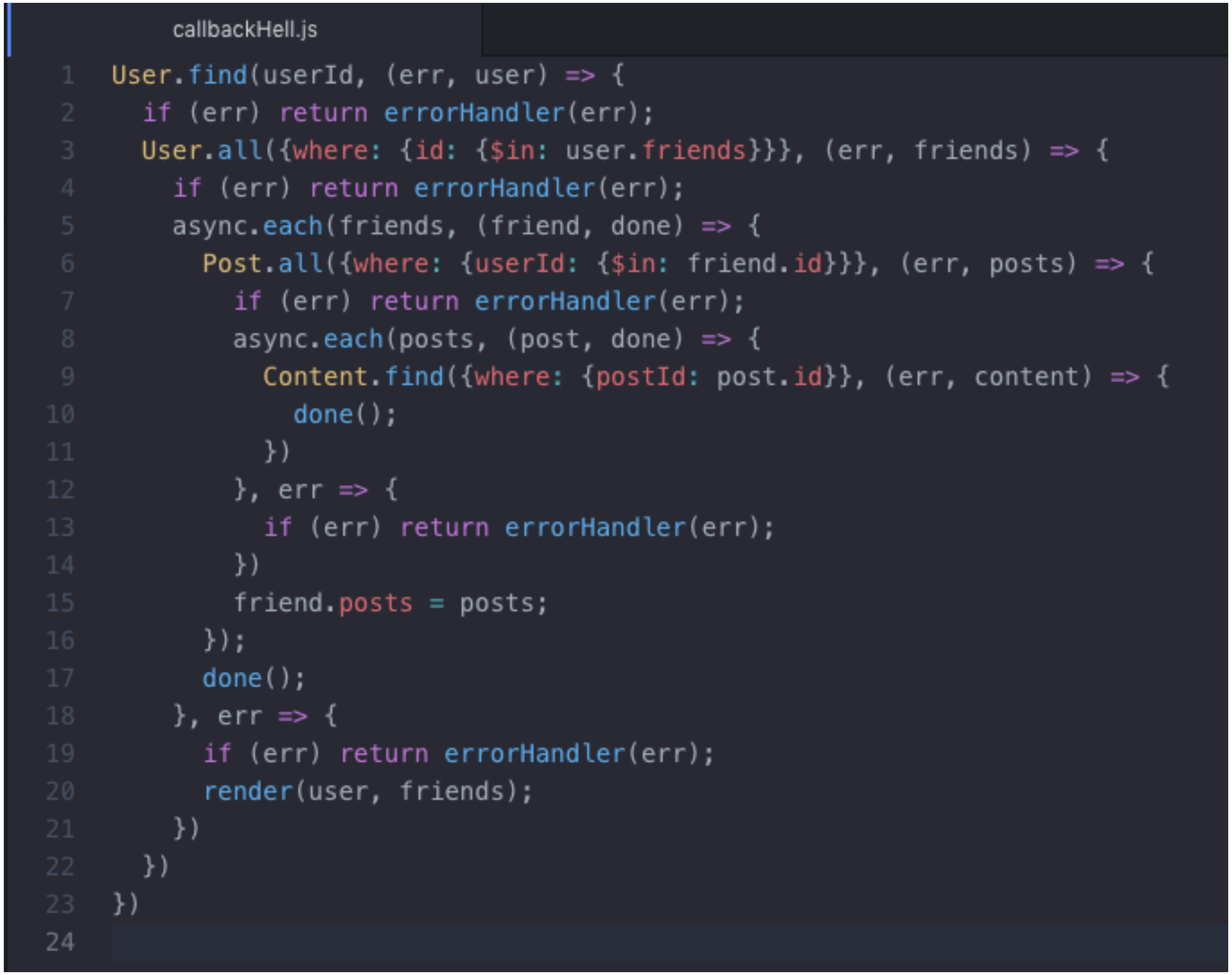

YO DAWG, I HEARD YOU LIKE CALLBACKS,
SO I PUT A CALLBACK IN YOUR CALLBACK
SO YOU CAN CALLBACK WHILE YOU CALLBACK
memegenerator.net

Callback Hell



```javascript
User.find(userId, (err, user) => {
  if (err) return errorHandler(err);
  User.all({where: {id: {$in: user.friends}}}, (err, friends) => {
    if (err) return errorHandler(err);
    async.each(friends, (friend, done) => {
      Post.all({where: {userId: {$in: friend.id}}}, (err, posts) => {
        if (err) return errorHandler(err);
        friend.posts = posts;
      });
      done();
    }, err => {
      if (err) return errorHandler(err);
      render(user, friends);
    })
  })
})
```

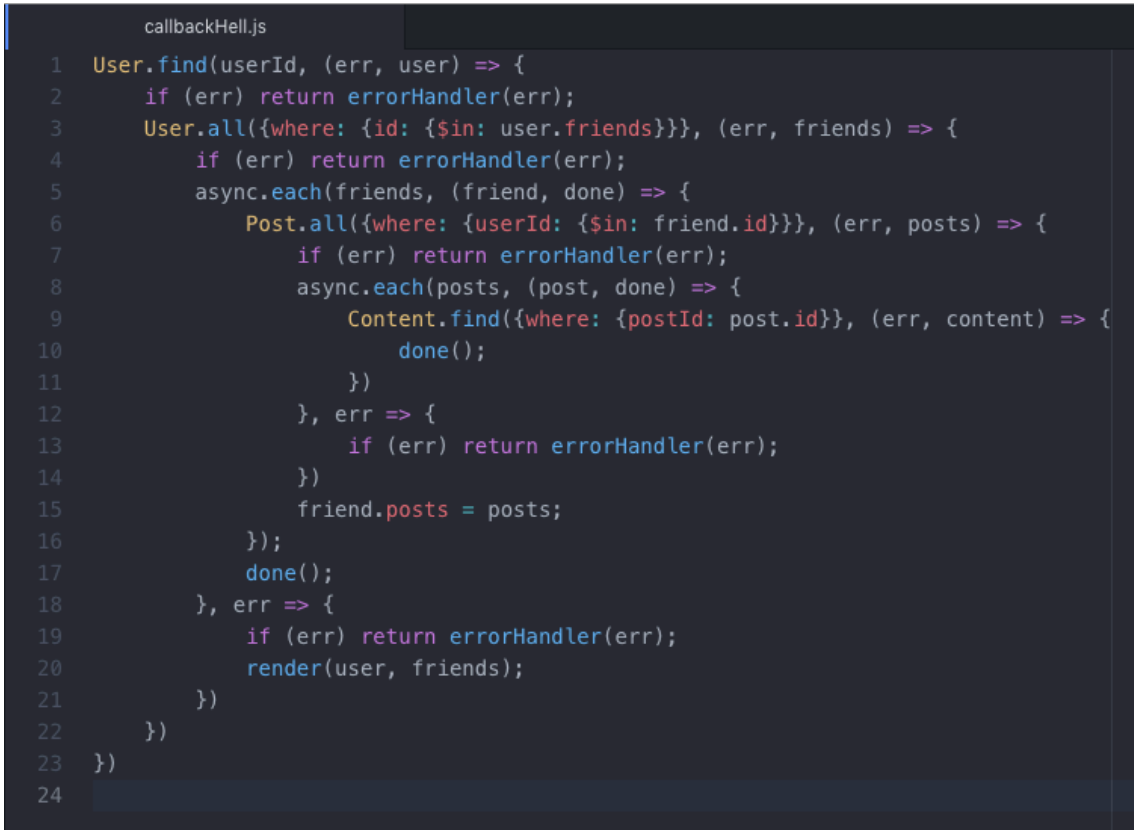
Callback Hell



```javascript
User.find(userId, (err, user) => {
  if (err) return errorHandler(err);
  User.all({where: {id: {$in: user.friends}}}, (err, friends) => {
    if (err) return errorHandler(err);
    async.each(friends, (friend, done) => {
      Post.all({where: {userId: {$in: friend.id}}}, (err, posts) => {
        if (err) return errorHandler(err);
        async.each(posts, (post, done) => {
          Content.find({where: {postId: post.id}}, (err, content) => {
            done();
          })
        }, err => {
          if (err) return errorHandler(err);
        })
        friend.posts = posts;
      });
      done();
    }, err => {
      if (err) return errorHandler(err);
      render(user, friends);
    })
  })
})

```

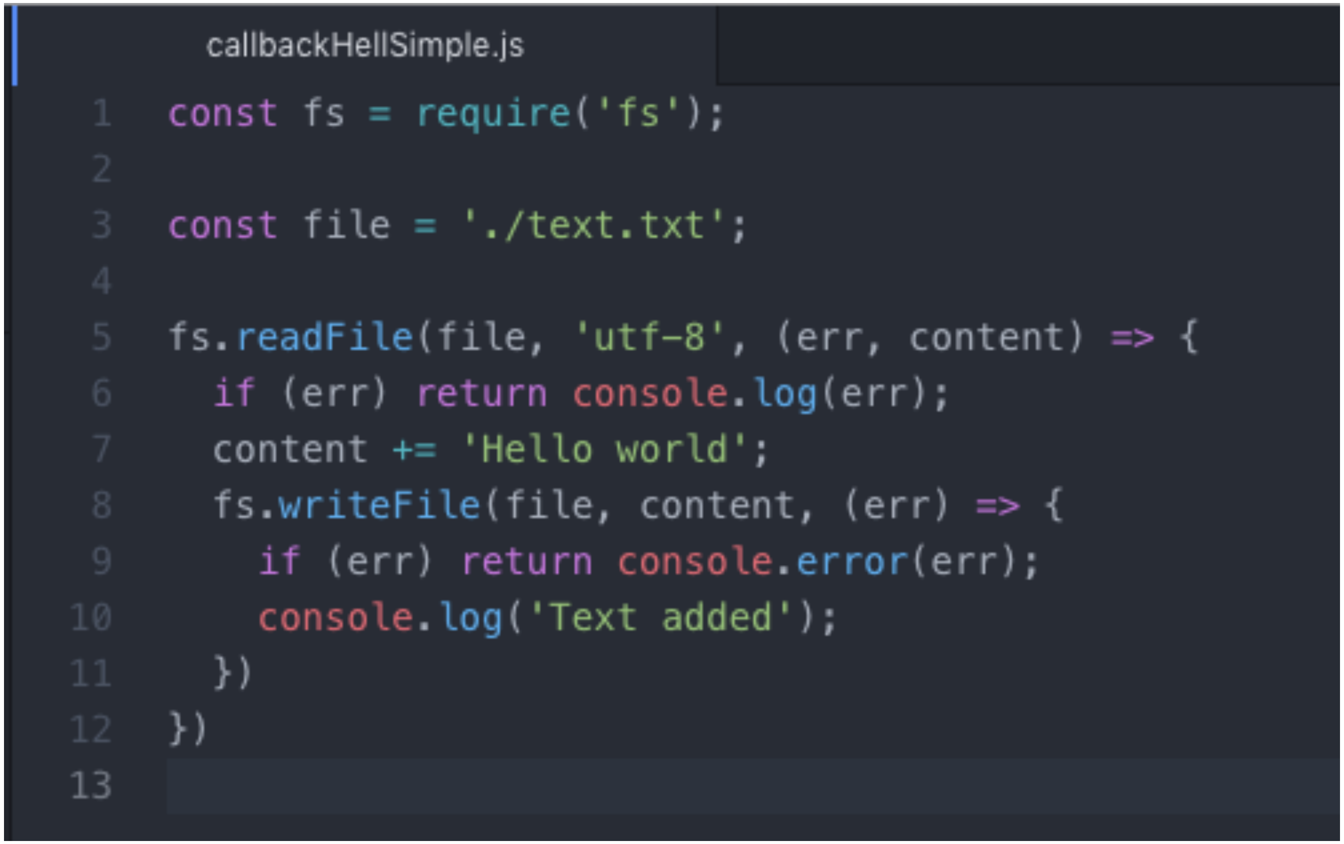
Avoiding Callback Hell in Node.js

Its easy to lose track of the logic flow, or even syntax, when small spaces are congested with so many nested callbacks.



```
User.find(userId, (err, user) => {
    if (err) return errorHandler(err);
    User.all({where: {id: {$in: user.friends}}}, (err, friends) => {
        if (err) return errorHandler(err);
        async.each(friends, (friend, done) => {
            Post.all({where: {userId: {$in: friend.id}}}, (err, posts) => {
                if (err) return errorHandler(err);
                async.each(posts, (post, done) => {
                    Content.find({where: {postId: post.id}}, (err, content) => {
                        done();
                    })
                }, err => {
                    if (err) return errorHandler(err);
                })
                friend.posts = posts;
            });
            done();
        }, err => {
            if (err) return errorHandler(err);
            render(user, friends);
        })
    })
})
```

Callback Hell (Simple example)

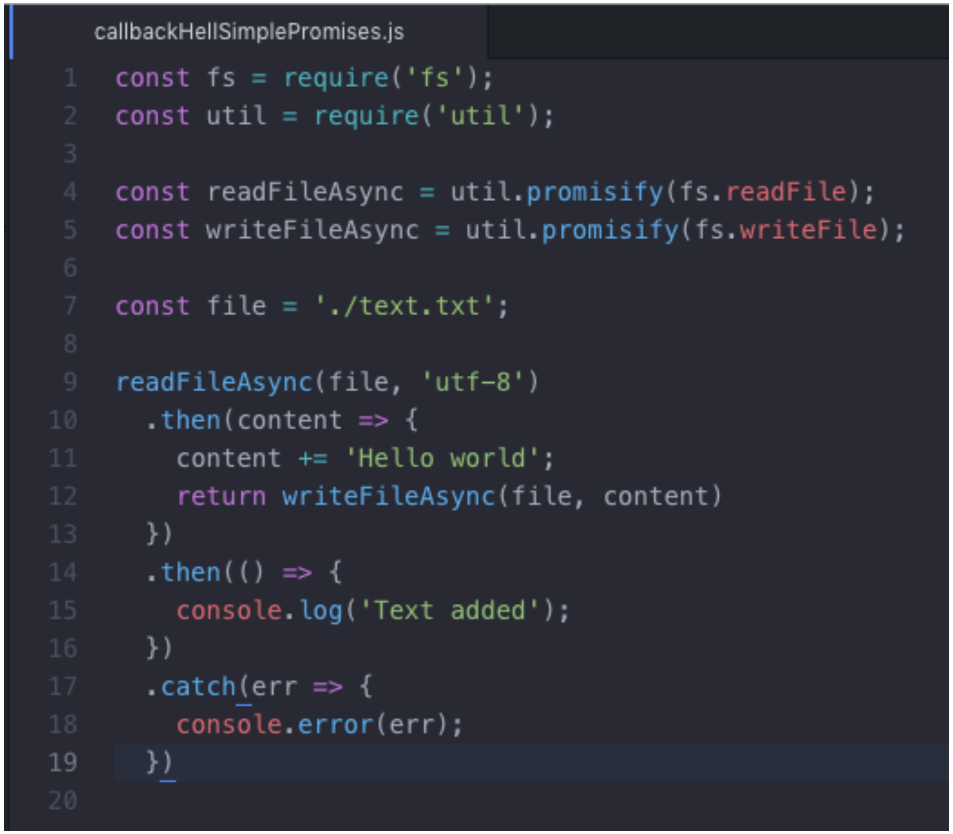


callbackHellSimple.js

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

const file = './text.txt';

fs.readFile(file, 'utf-8', (err, content) => {
  if (err) return console.log(err);
  content += 'Hello world';
  fs.writeFile(file, content, (err) => {
    if (err) return console.error(err);
    console.log('Text added');
  })
})

```

Using Promise

The **Promise** object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

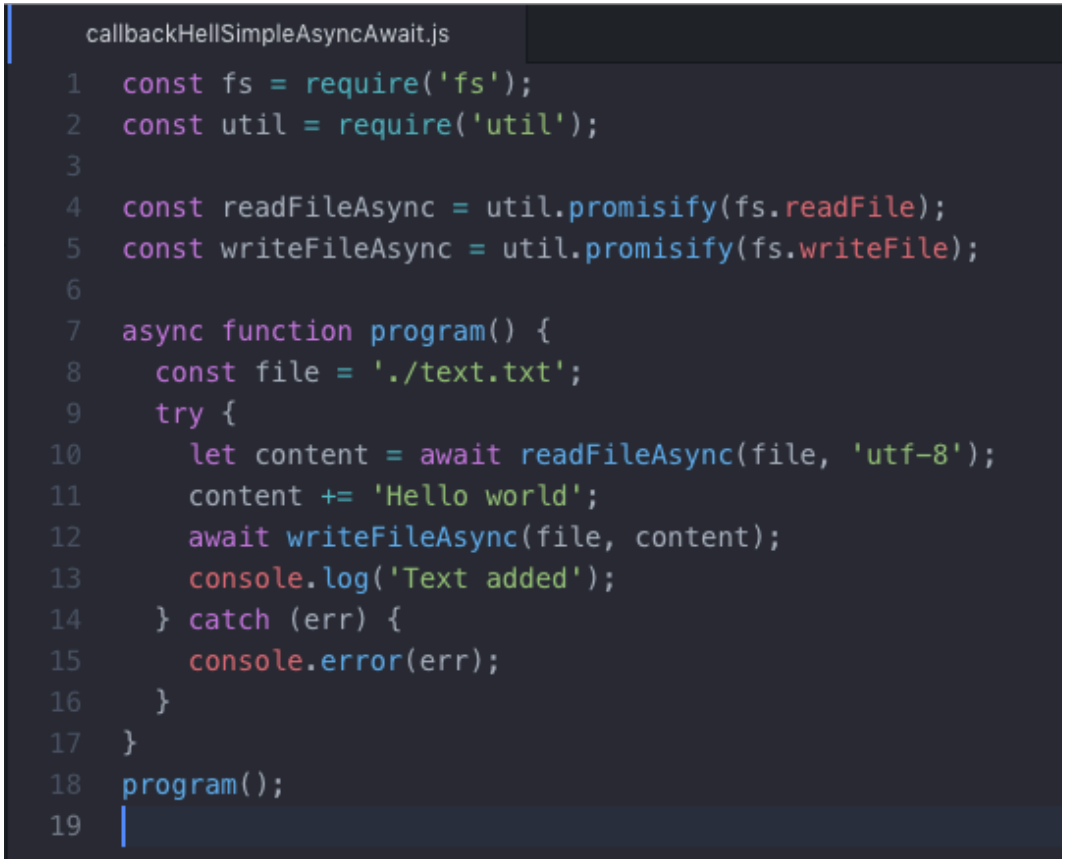


callbackHellSimplePromises.js

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

const readFileAsync = util.promisify(fs.readFile);
const writeFileAsync = util.promisify(fs.writeFile);

const file = './text.txt';

readFileAsync(file, 'utf-8')
  .then(content => {
    content += 'Hello world';
    return writeFileAsync(file, content)
  })
  .then(() => {
    console.log('Text added');
  })
  .catch(err => {
    console.error(err);
  })

```

Using Async/Await

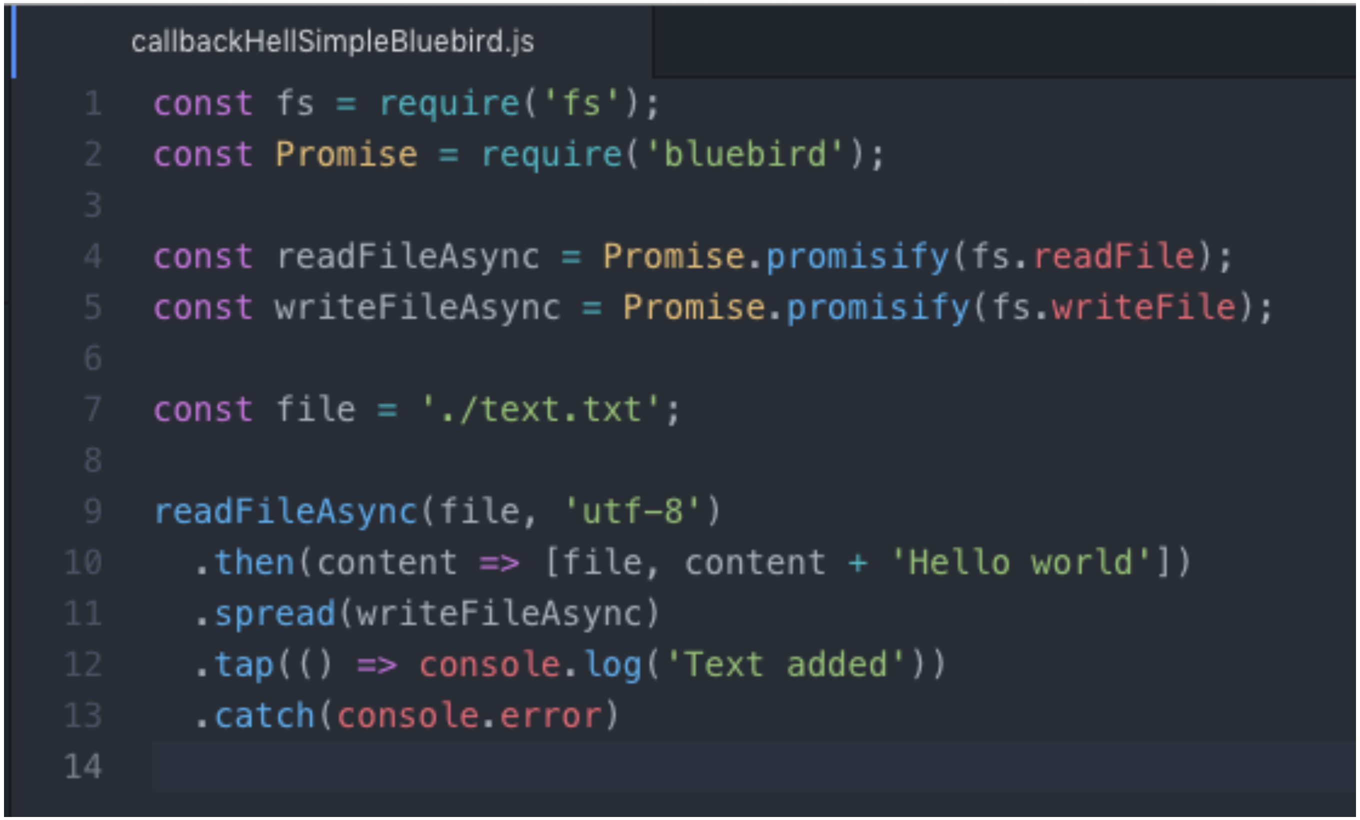
An **async** function can contain an **await** expression, that pauses the execution of the **async** function and waits for the passed Promise's resolution.



```javascript
const fs = require('fs');
const util = require('util');

const readFileAsync = util.promisify(fs.readFile);
const writeFileAsync = util.promisify(fs.writeFile);

async function program() {
  const file = './text.txt';
  try {
    let content = await readFileAsync(file, 'utf-8');
    content += 'Hello world';
    await writeFileAsync(file, content);
    console.log('Text added');
  } catch (err) {
    console.error(err);
  }
}
program();
```

Using Promise (bluebird)

The **Promise** object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.



callbackHellSimpleBluebird.js

```
const fs = require('fs');
const Promise = require('bluebird');

const readFileAsync = Promise.promisify(fs.readFile);
const writeFileAsync = Promise.promisify(fs.writeFile);

const file = './text.txt';

readFileAsync(file, 'utf-8')
  .then(content => [file, content + 'Hello world'])
  .spread(writeFileAsync)
  .tap(() => console.log('Text added'))
  .catch(console.error)

```

Using Promise (bluebird)

Advantages

- Error pattern matching in catch
- Bluebird promises can be cancelled
- Unhandled errors are not silently swallowed by default
- promisifyAll(), spread()
- Bluebird can work as a drop-in replacement for native promises for an instant performance boost

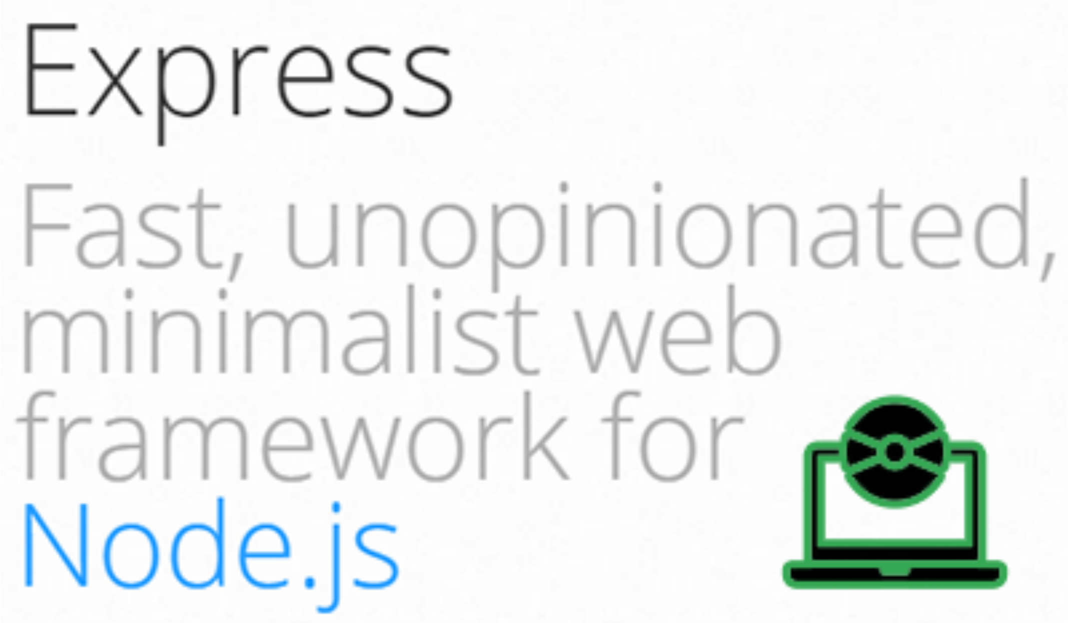
Disadvantages

- Bluebird weighs 17K gzipped
- Bluebird extra features may confuse some at first

AGENDA

- ~~Introduction~~
- ~~Modules~~
  - ~~Modules Types~~
  - ~~Importing and exporting~~
  - ~~Module Context~~
- ~~Event loop~~
  - ~~Single-threaded~~
  - ~~Asynchronous non-blocking operations~~
  - ~~Libuv~~
  - ~~Callback hell~~
- Express framework
  - Middleware concept
  - express() application
  - Request/Response
  - Router

What is Express



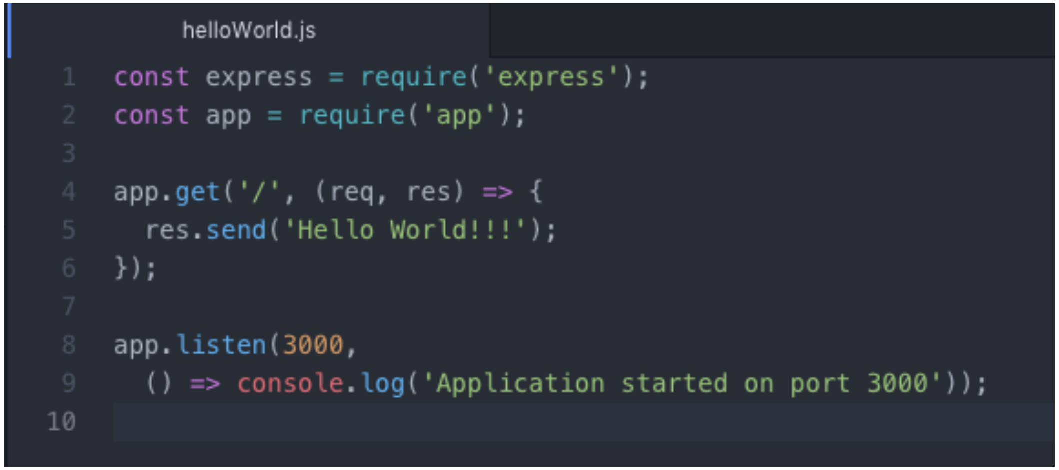
Express

Fast, unopinionated, minimalist web framework for Node.js

```
$ npm install express --save
```

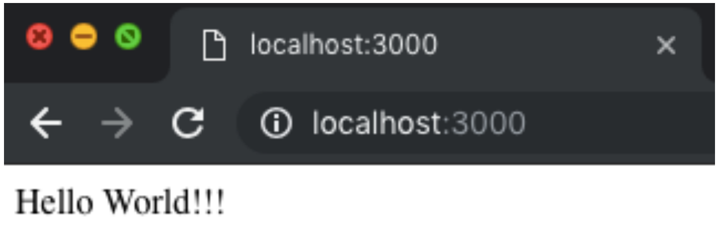
Hello World example

This app starts a server and listens on port 3000 for connections.



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

app.get('/', (req, res) => {
  res.send('Hello World!!!');
});

app.listen(3000,
  () => console.log('Application started on port 3000'));

```

The app responds with “Hello World!” for requests to the root URL (/) or **route**.



Hello World!!!

What is Express

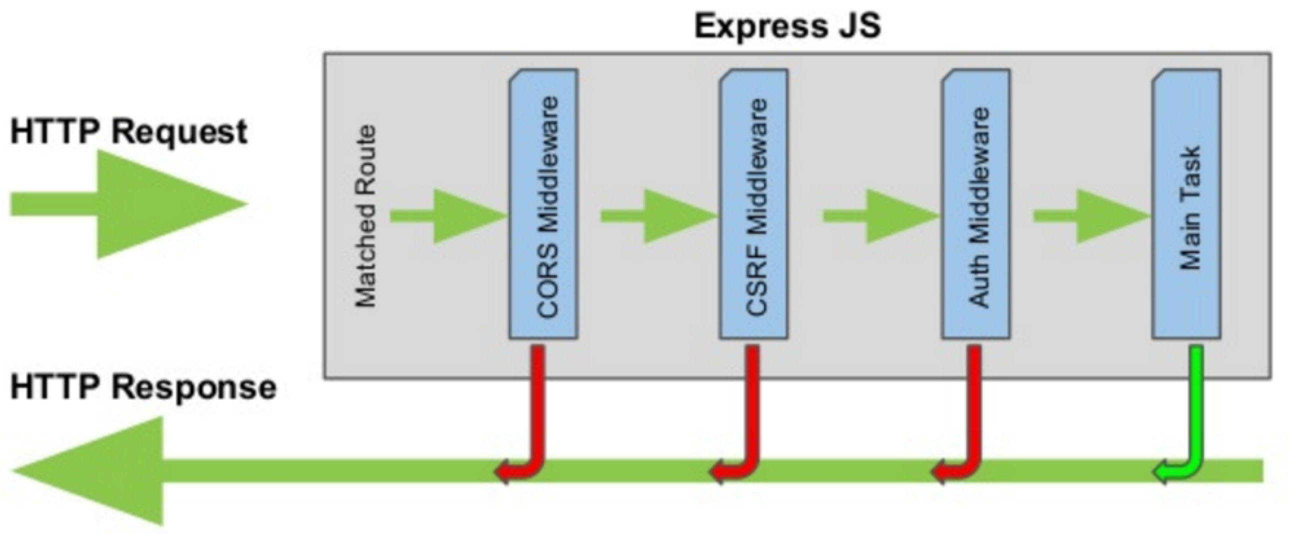
Express is the most popular Node web framework, and is the underlying library for a number of other popular Node web frameworks. It provides mechanisms to:

- Write handlers for requests
- Integrate with "view" rendering engines
- Set common web application settings (port, view template etc.)
- Add additional request processing "**middleware**" at any point within the request handling pipeline

Middleware concept

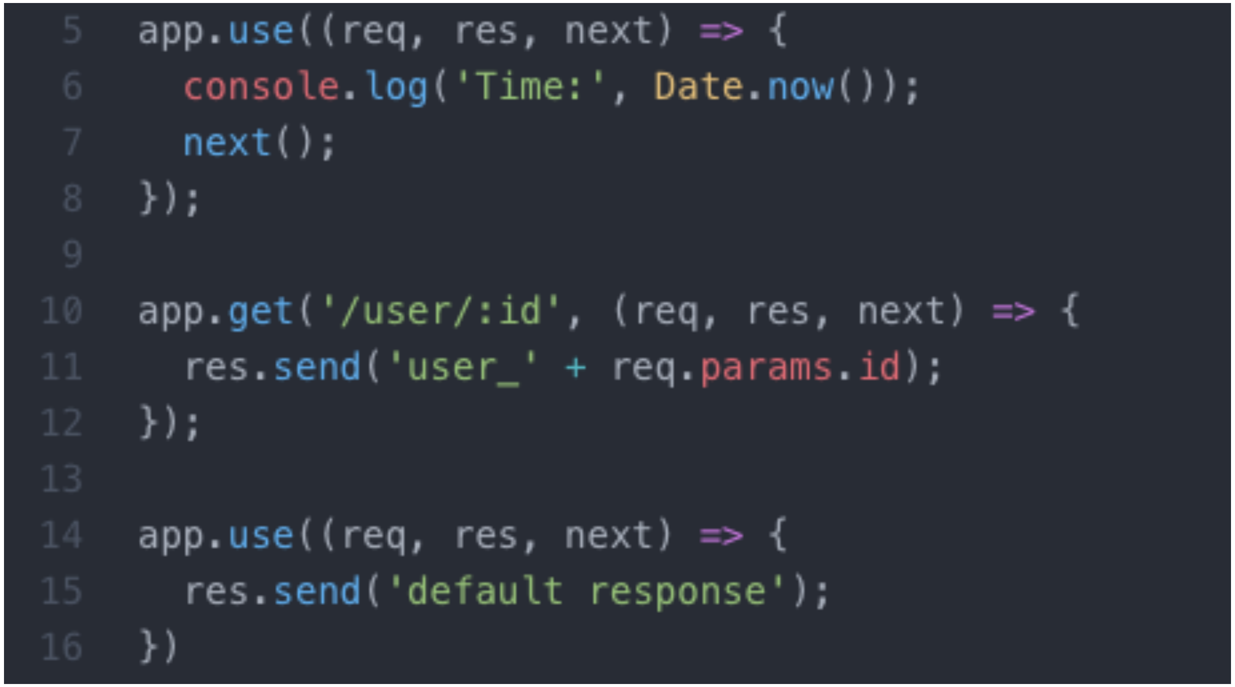
Middleware functions:

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



Application-level middleware

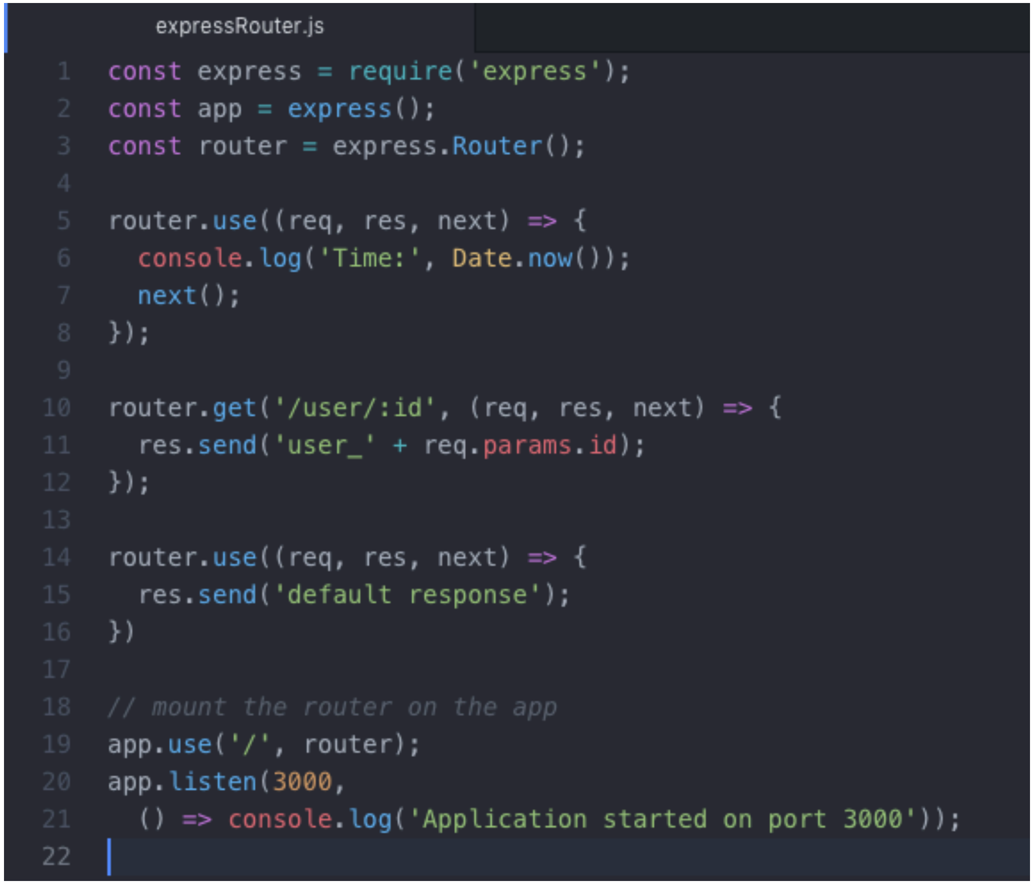
Bind application-level middleware to an instance of the app object by using the *app.use()* and *app.METHOD()* functions.



```
app.use((req, res, next) => {
  console.log('Time:', Date.now());
  next();
});

app.get('/user/:id', (req, res, next) => {
  res.send('user_' + req.params.id);
});

app.use((req, res, next) => {
  res.send('default response');
})
```

Router-level middleware

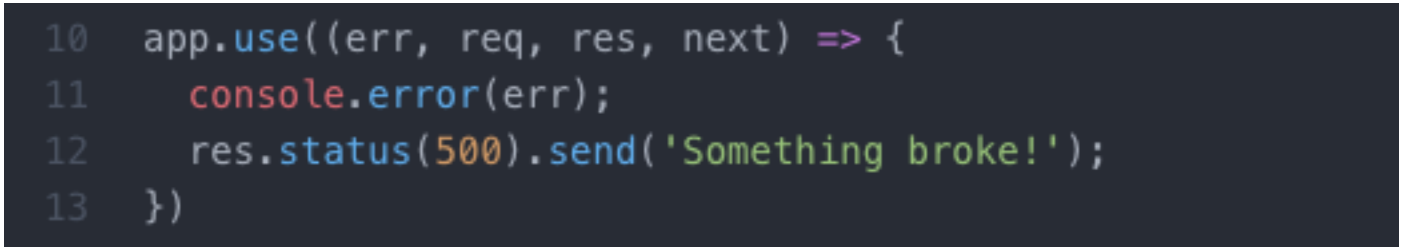
Router-level middleware works in the same way as application-level middleware, except it is bound to an instance of *express.Router()*.



```
const express = require('express');
const app = express();
const router = express.Router();

router.use((req, res, next) => {
  console.log('Time:', Date.now());
  next();
});

router.get('/user/:id', (req, res, next) => {
  res.send('user_' + req.params.id);
});

router.use((req, res, next) => {
  res.send('default response');
})

// mount the router on the app
app.use('/', router);
app.listen(3000,
  () => console.log('Application started on port 3000'));
```

Error-handling middleware

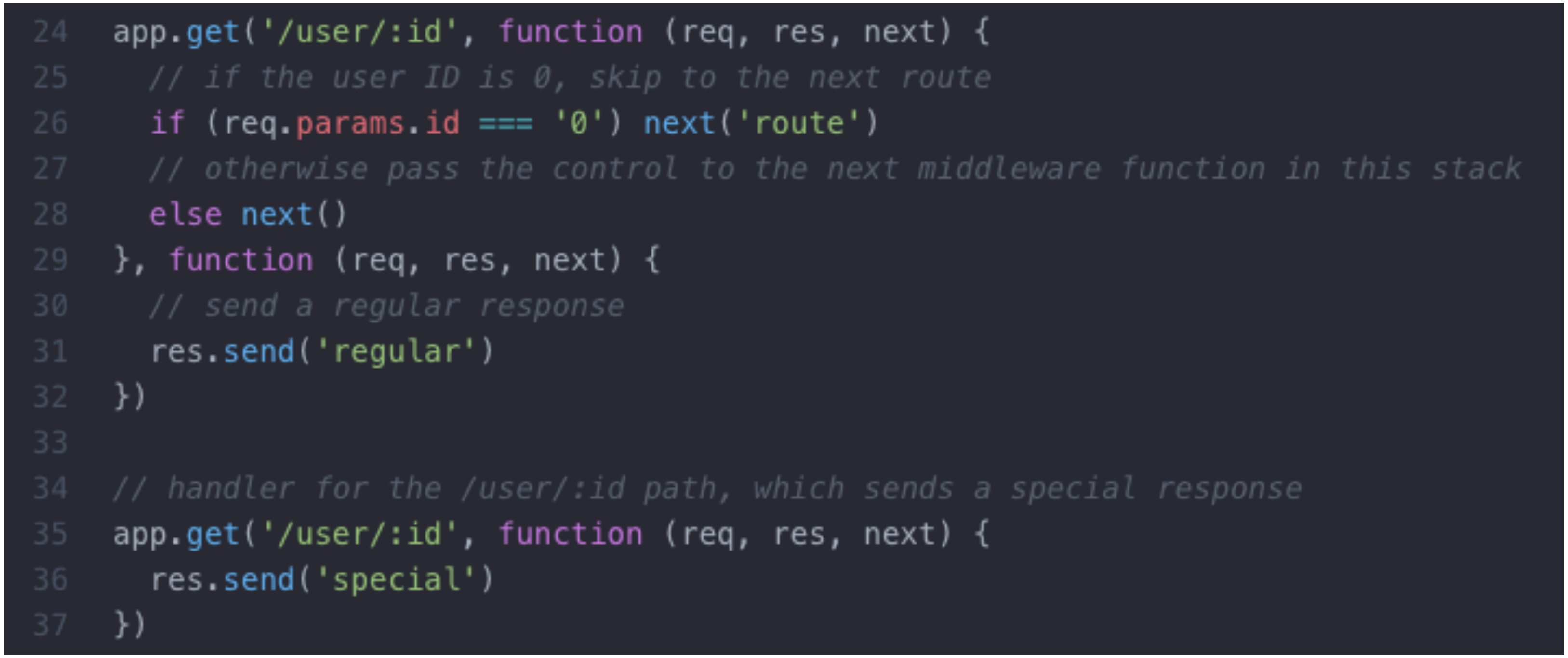
1. **Always** takes **four** arguments and the first is error.
2. You can call it by **next(err)** in any middleware.
3. Once **next(err)** is called, all other non-error middlewares would be skipped.



```
app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).send('Something broke!');
})
```

Express middleware

1. To skip the rest of the middleware functions from a router middleware stack, call next('route') to pass control to the next route.



```
app.get('/user/:id', function (req, res, next) {
  // if the user ID is 0, skip to the next route
  if (req.params.id === '0') next('route')
  // otherwise pass the control to the next middleware function in this stack
  else next()
}, function (req, res, next) {
  // send a regular response
  res.send('regular')
})

// handler for the /user/:id path, which sends a special response
app.get('/user/:id', function (req, res, next) {
  res.send('special')
})
```

Built-in middleware

Express has the following built-in middleware functions:

1. express.static serves static assets such as HTML files, images, and so on.
2. express.json parses incoming requests with JSON payloads.
3. express.urlencoded parses incoming requests with URL-encoded payloads.

AGENDA

- ~~Introduction~~
- ~~Modules~~
  - ~~Modules Types~~
  - ~~Importing and exporting~~
  - ~~Module Context~~
- ~~Event loop~~
  - ~~Single-threaded~~
  - ~~Asynchronous non-blocking operations~~
  - ~~Libuv~~
  - ~~Callback hell~~
- ~~Express framework~~
  - ~~Middleware concept~~
  - express() application
  - Request/Response
  - Router

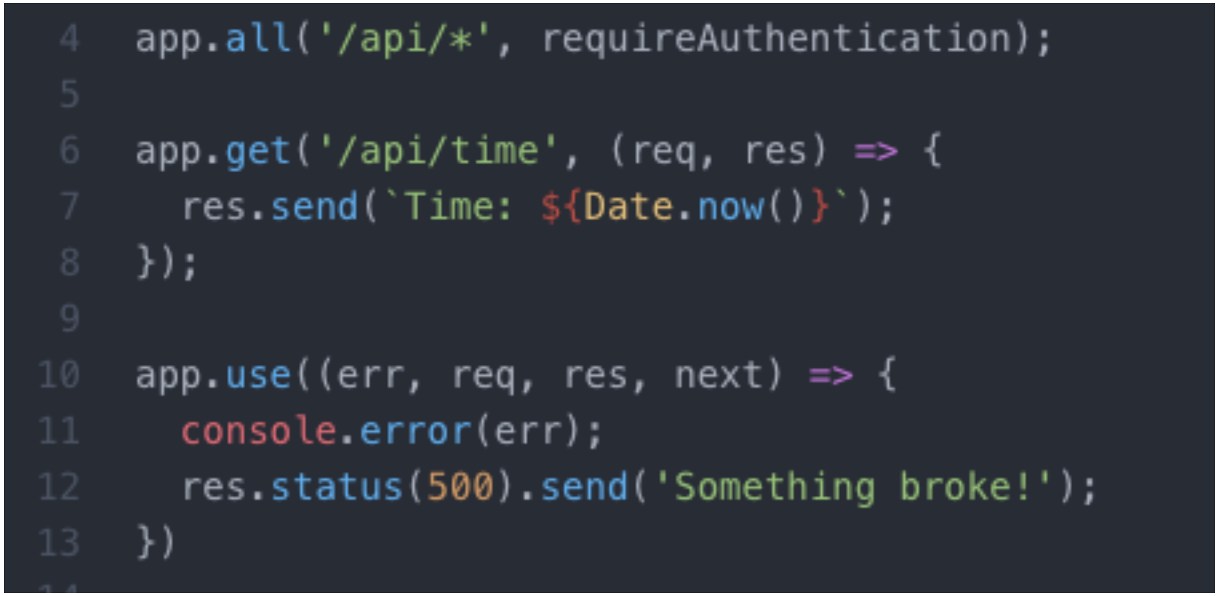
Express Application

The app object has methods for

1. Routing HTTP requests; see for example, app.METHOD and app.param
2. Configuring middleware; see app.route
3. Rendering HTML views; see app.render
4. Registering a template engine; see app.engine
5. General app configuration(app settings table)

Application: Routes

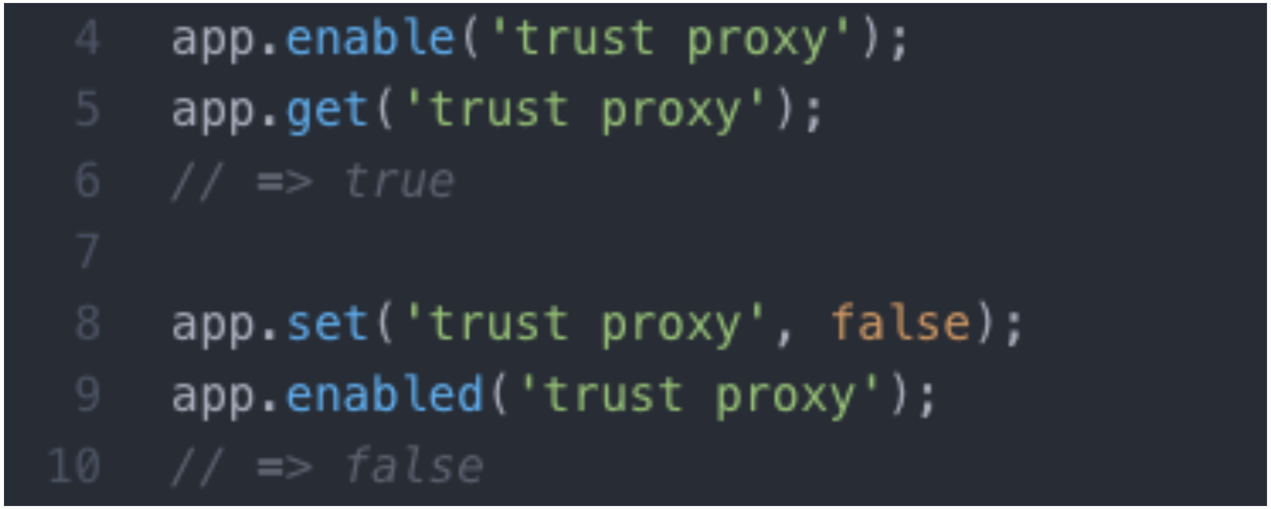
1. app.all(path, callback [, callback ...])
2. app.METHOD(path, callback [, callback ...])
3. app.use([path,] callback [, callback...]) for middleware



```
app.all('/api/*', requireAuthentication);

app.get('/api/time', (req, res) => {
  res.send(`Time: ${Date.now()}`);
});

app.use((err, req, res, next) => {
  console.error(err);
  res.status(500).send('Something broke!');
})
```

Application: Properties

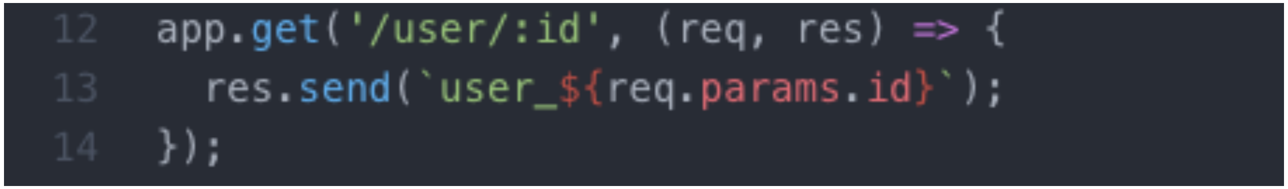
1. app.set(name, value) / app.get(name)
2. app.enable(name) / app.disable(name)
3. app.enabled(name) / app.disabled(name)



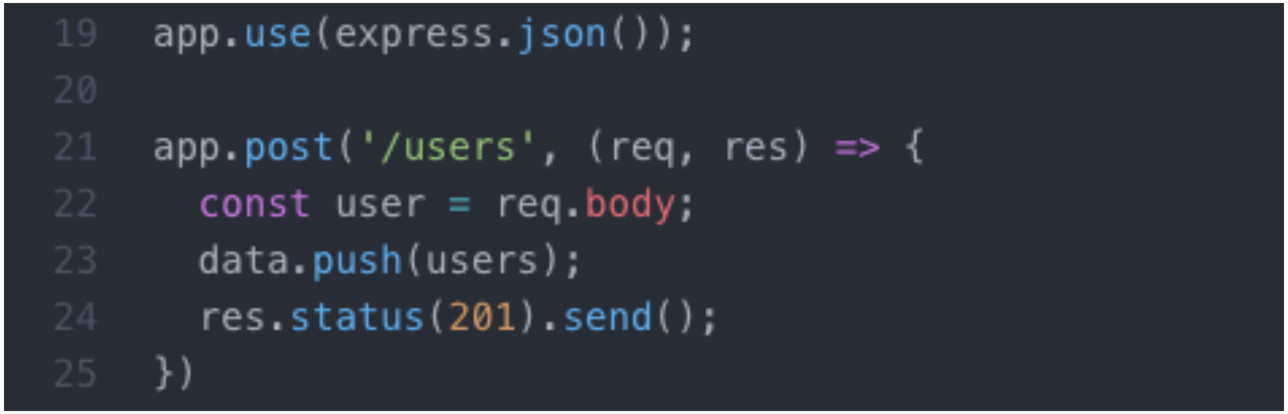
```
app.enable('trust proxy');
app.get('trust proxy');
// => true

app.set('trust proxy', false);
app.enabled('trust proxy');
// => false
```

REQUEST

The req object represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers and so on.



```
app.get('/user/:id', (req, res) => {
  res.send(`user_${req.params.id}`);
});
```



```
app.use(express.json());

app.post('/users', (req, res) => {
  const user = req.body;
  data.push(users);
  res.status(201).send();
})
```

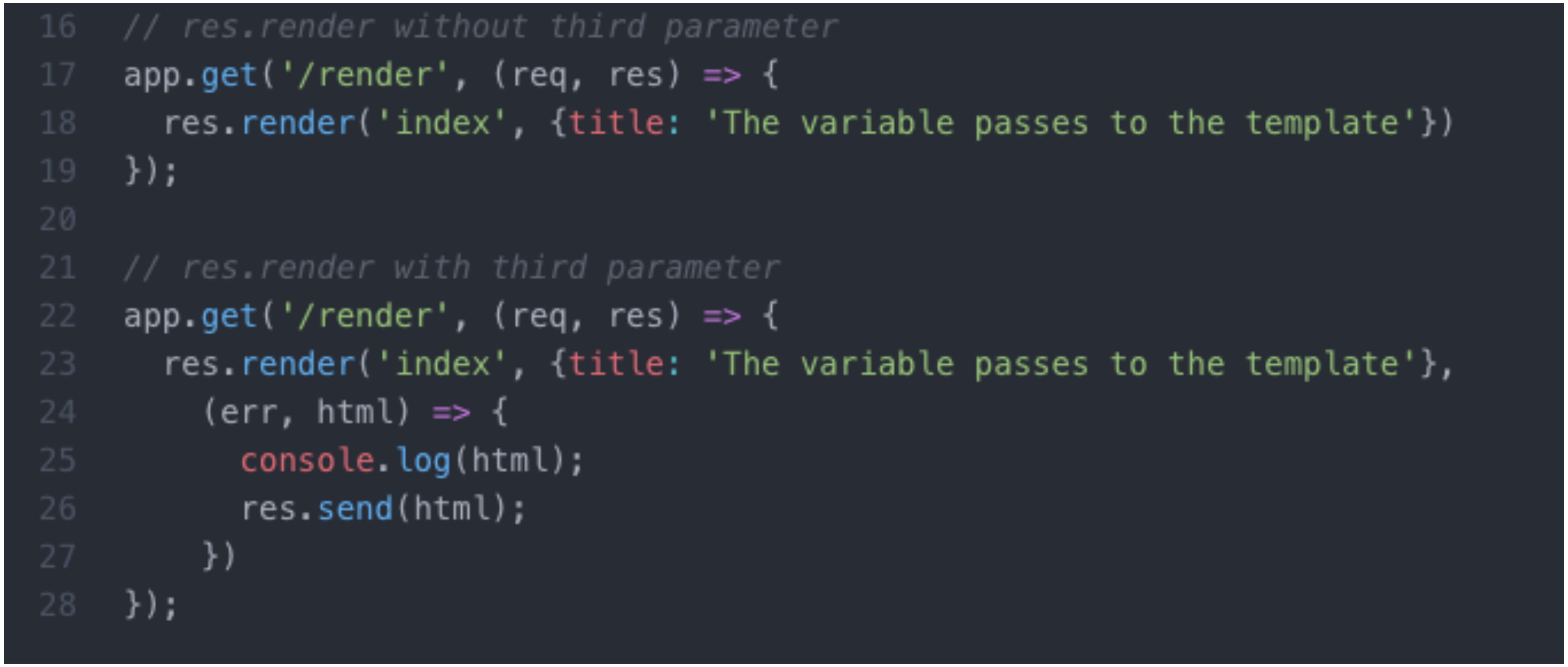
RESPONSE

The res object represents the HTTP response that an Express app sends when it gets an HTTP request.

1. **end** – quickly end the response without any data
2. **sendStatus** – send status code string representation as the response body
3. **send** – default response with data (Buffer, String, object, or Array)
4. **sendFile** – sends the file at the given path to the client
5. **json** – JSON response with propercontent-type
6. **jsonp** – JSON response with JSONP support. Callback called callbackby default
7. **redirect** – redirects to the specified URL (or path). You can set status code
8. **render** – renders a view and sends the rendered HTML string to the client

RESPONSE: Render view

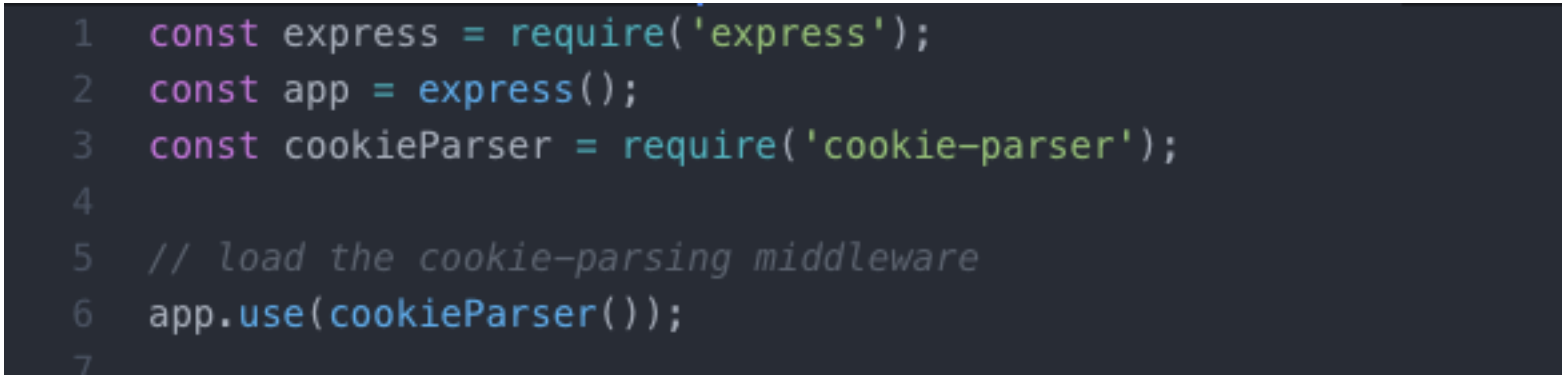
res.render(view [, locals] [, callback]) - renders a view and sends the rendered HTML string to the client.



```
// res.render without third parameter
app.get('/render', (req, res) => {
  res.render('index', {title: 'The variable passes to the template'})
});

// res.render with third parameter
app.get('/render', (req, res) => {
  res.render('index', {title: 'The variable passes to the template'},
    (err, html) => {
      console.log(html);
      res.send(html);
    })
});
```

Express third-party middleware

The usage of the third-party middleware is the same



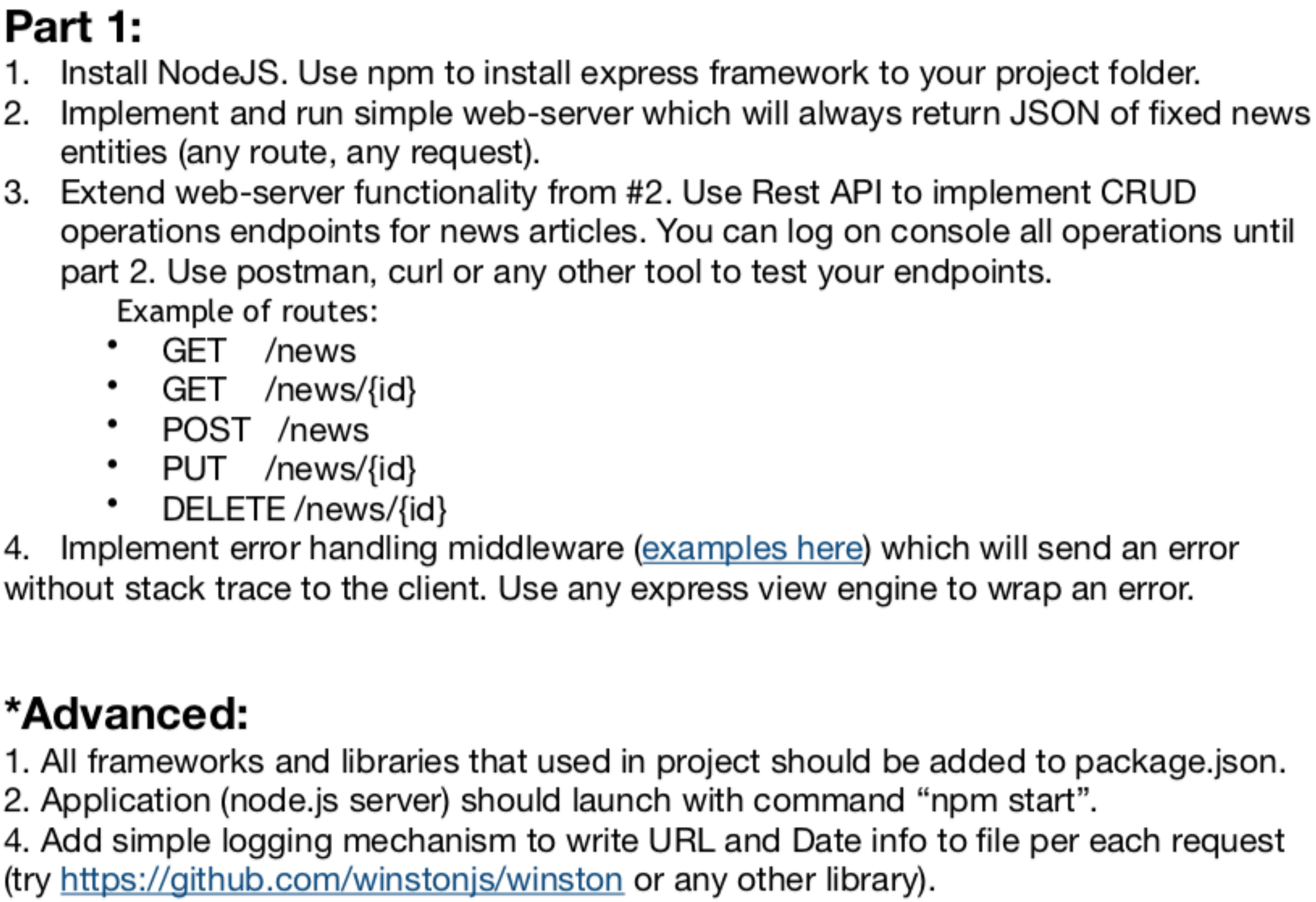
```
const express = require('express');
const app = express();
const cookieParser = require('cookie-parser');

// load the cookie-parsing middleware
app.use(cookieParser());
```

ROUTER

A router object is an isolated instance of middleware and routes.

1. router.all(path, [callback, ...] callback)
2. router.METHOD(path, [callback, ...] callback)
3. router.param(name, callback)
4. router.route(path)
5. router.use([path], [function, ...] function)

Hometask



Part 1:

1. Install NodeJS. Use npm to install express framework to your project folder.
2. Implement and run simple web-server which will always return JSON of fixed news entities (any route, any request).
3. Extend web-server functionality from #2. Use Rest API to implement CRUD operations endpoints for news articles. You can log on console all operations until part 2. Use postman, curl or any other tool to test your endpoints.

   Example of routes:
   - GET /news
   - GET /news/{id}
   - POST /news
   - PUT /news/{id}
   - DELETE /news/{id}

4. Implement error handling middleware (examples here) which will send an error without stack trace to the client. Use any express view engine to wrap an error.

*Advanced:

1. All frameworks and libraries that used in project should be added to package.json.
2. Application (node.js server) should launch with command “npm start”.
4. Add simple logging mechanism to write URL and Date info to file per each request (try https://github.com/winstonjs/winston or any other library).

ARE THERE
ANY QUESTIONS?