Part II — NodeJs

Based on lectures by Alex Lau

Notes taken by Brian Chow

Lent 2017-2018

Contents

1 Introduction to Node

1.1 Modules

Using Modules

Let’s start by creating two files app.js, greet.js. Write the following:

```
// greet.js

console.log('Hello there');

// app.js

require('./greet');
```

in your terminal run node app.js

1.2 Events

1.3 Asynchronous code and callbacks

1.4 fs

1.4.1 read files

To handle files we can use the node module fs which stands for file system. Let?s look at some code and then try to understand what is going on. Let?s create two files, app.js and greet.txt.

```
// greet.text

Hello there.
```

```
// app.js
var fs = require('fs');

var greeting = fs.readFileSync(__dirname + '/greet.txt');

console.log(greeting);
```

__dirname stands for current directory the node file is running in.

Run the code, what does your console log? If you did everything correctly you will see a log of ?¡Buffer 48 65 6c 6c 6f 20 74 68 65 72 65 2e 0a¿?. That is the buffer with it?s binary encoded data. To make it readable we have to decode it in a suitable format. A list of all formats can be found in the documentation for buffer, for our use case we are interested in utf8.

```
// app.js
var fs = require('fs');

var greeting = fs.readFileSync(__dirname + '/greet.txt', 'utf8');

console.log(greeting);
```

As you can see now we get everything in human readable format. However our code could still be improved. We are currently using fs.readFileSync(). As the name suggests method is synchronous, which we generally should avoid if possible. Imagine you are trying to read a very large file, the code wouldn?t continue until finished reading the file, which would result in a poor user experience. Therefore let?s refactor the code to make it asynchronous.

```
// app.js
var fs = require('fs');

var greeting = fs.readFile(__dirname + '/greet.txt', 'utf8',
    function(err, data) {
    console.log(data)
});
```

1.4.2 write files

We don?t just want to read files, but also write to files. Let?s look at some code to figure out how we could do that. First we create three files text.txt, textcopy.txt, app.js. Fill the text.txt file with a large amount of lorem ipsum. Generate around 70000 bytes of text.

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

var readable = fs.createReadStream(__dirname + '/text.txt');

readable.on('data', function(chunk){
    console.log(chunk.length);
});
```

createReadStream emits an event called data, which we can listen for. The stream fills the buffer with data and emits the event once the buffer is full or all data is read. It then passes on the chunk of data from the buffer and begins to refill the buffer. So we are listening for buffers.

If you run the above code you will notice that your console actually displays two buffers. The reason for that is that default buffer size is around 64000 bytes, however our document is 70000 bytes, so it couldn?t be all placed in the buffer at one time.

Imagine you have a lot of traffic, than 64000 bytes buffer can be very memory intensive and slow down everything. We can therefore reduce the size of the buffer, using highWaterMark to let?s say 32kb. Also let?s make it human readable by changing the encoding.

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

var readable = fs.createReadStream(__dirname + '/text.txt', {
    encoding: 'utf8', highWaterMark: 32*1024 });

readable.on('data', function(chunk){
    console.log(chunk.length);
});
```

Let?s actually start writing to a new file.

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

var readable = fs.createReadStream(__dirname + '/text.txt', {
    encoding: 'utf8', highWaterMark: 32*1024 });

var writeable = fs.createWriteStream(__dirname + '/textcopy.txt');
```

```
readable.on('data', function(chunk){
    writeable.write(chunk);
});
```

Run the code above and check out textcopy.txt.

1.4.3 pipes

Connecting two streams by writing from one stream to another. In node you can pipe from a readable stream to a writeable stream. The createReadStream object includes a pipe method. So we can refactor above?s code a bit.

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

var readable = fs.createReadStream(__dirname + '/text.txt', {
    encoding: 'utf8', highWaterMark: 32*1024 });

var writeable = fs.createWriteStream(__dirname + '/textcopy.txt');

readable.pipe(writeable);
```

If you run this code you?ll notice it achieves exactly the same as the previous code, it is just a lot easier and cleaner to write.

Quick note regarding the video below, doing a server will be covered in the next section, so feel free to skip the end.

1.5 Web Server

1.5.1 API

We have already come across APIs before, let?s quickly remind ourself what APIs are. APIs can be imagined as access to a set of tools that help us write software. These tools are other software. In terms of the web APIs are just URLs that send and accept data, through http.

An API consists out of endpoints, the different URLs of the API.

Keeping that in mind let?s put some code down to create our own API:

```
var http = require('http');

http.createServer(function(req, res){

    // respond to the request

    res.writeHead(200, { 'Content-Type':'text/plain' });
    res.end('Hello world');

}).listen(8080, '127.0.0.1');
```

var http = require('http');
http.createServer(function(req, res)
// respond to the request
res.writeHead(200, 'Content-Type':'text/plain'); res.end('Hello world');
).listen(8080, '127.0.0.1');

1.5.2 Serving files

Sending some plain text isn?t really that handy if we want to display complex websites to the user. So let?s serve some HTML. Create two files index.html and app.js

```
<!DOCTYPE html>
<html>
    <head>
        <title>Hello World</title>
    </head>
    <body>
        <h1>Hello World</h1>
    </body>
</html>
```

```
var http = require('http');
var fs = require('fs');

http.createServer(function(req, res){

    res.writeHead(200, { 'Content-Type':'text/html' });
    fs.createReadStream(__dirname + '/index.html').pipe(res);

}).listen(8080, '127.0.0.1');
```

Serve the page via node and check it out in the browser. If you want to you can make changes to the HTML file and refresh the browser window, you should see those changes implemented, without having to restart node. That is because node only runs the JavaScript file and pipes your HTML file anew every time the page is loaded.

1.5.3 sending data

The most common way to transmit data from a server to another server or frontend is through the JSON format. Since JSON is very similar to a JS object, sending JSON through node is an extremely easy process.

```
var http = require('http');

http.createServer(function(req, res){

    res.writeHead(200, { 'Content-Type':'application/json' });
    var obj = {
        name: 'John',
        surname: 'Doe'
    }
    res.end(JSON.stringify(obj));

}).listen(8080, '127.0.0.1');
```

JSON.stringify() transforms the JS object into JSON. Now your JS on the front end will know how to deal with the object.

1.5.4 routing

So far we only accepted requests on one route. However that doesn?t really help us serve more than one file. Let?s go back to our html example and change the html, below the heading include an image.

```
<img src="/assets/image.png" />
```

```
var http = require('http');
var fs = require('fs');

http.createServer(function(req, res){
    fs.createReadStream(__dirname + '/index.html').pipe(res);
}).listen(8080, '127.0.0.1');
```

run the code and try to find the image on the website. It won?t appear, the problem is that we every request to node will return a the html page. When the browser tries to load the image, it will make a request for the image to the node server, but node will just return the same html page.

In order to change that, we need to actually look at the request the node server is receiving.

```
var http = require('http');
var fs = require('fs');

http.createServer(function(req, res){
    if(req.url === '/') {
        fs.createReadStream(__dirname + '/index.html').pipe(res);
    } else if(req.url === '/assets/image.png') {
        fs.createReadStream(__dirname + '/assets/image.png').pipe(
            res);
    } else {
        res.writeHead(404);
        res.end();
    }

}).listen(8080, '127.0.0.1');
```

The request object includes a url property which contains the path the request was made to. So we can use that path and determine which document to return. If we receive the image path we return the image, and so on.

It could also happen that a request was made to a path we haven?t defined yet, in that case we need to return something otherwise node won?t know what to do and stop working. So we just return a header containing 404.

2 Node packages

2.1 npm

We already came across npm before, however we never really talked about how to use it. npm hosts the world wide largest collection of open source code. The code there is available in form of packages. A package is simply a collection of code that you can use. To maintain (install, update, delete) those packages we use a package manager, in our case npm. While you might say installing some code is easy, updating it is pretty hard, especially if the code depends on other packages, because then you need to keep track of all the packages to update. That is where npm comes in as a really useful helper.
npm comes installed with node. You can find packages on the official npm website.

2.1.1 npm init

2.1.2 package.json

2.1.3 installing global packages

2.1.4 dev dependencies

2.1.5 GitHub and .gitignore

2.2 Express

Express is a npm package that allows us to set up servers on node in a more straightforward manner.

Let's get started, create a new folder and npm init a new module in that folder. Add a 'app.js' file and let?s install express.

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

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

Express the module returns a function constructor, so in order to make use of it we need to call express() and save that constructor in a variable, called app by convention.
Let?s add a listen function to our app

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

app.listen(8080);
```

That's it, we have a running server.
Let's start listening for requests

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

app.get('/', function(req, res) {
    res.send('hello');
});
```

```
app.post('/login', function(req, res) {
    console.log(req.path);
    res.send('post received');
});

app.listen(8080);
```

2.2.1 Routing

When our app grows it will have many different routes, we don't want to manually define the route for every single page. Luckily Express makes that a painless process.

```
app.get('/users/:id', function(req, res){
    console.log('the user id is ' + req.params.id);
});
```

Express tells us the parameters (in this case id) that the dynamic url contains. This we can dynamically load content without having to define every route manually.

2.2.2 Serving static files

Middleware is code that sits between two layers of software. So in the case of Express those two layers would be request and response. Express makes it very easy to plugin some middleware between the two and therefore make it extensible. That allows us to handle certain requests and automate things. We could even chain several pieces of middleware together.

Imagine you have a website with many static files that you want to serve, like CSS files, HTML files or image files. We don?t want to make a route for every file, so instead we use middleware that handles those kind of requests. We usually put those kind of files in a folder called public.

To serve them just use

```
app.use(express.static('public'));
```

2.2.3 Creating middleware

It is very straightforward to create your own middleware. Let's look at some code

```
app.use('/', function(req, res, next){
    console.log('Request Url: '+ req.url);
    next();
});

app.get('/', function(req, res){
    res.send('hello');
});
```

As you see in the example above the middleware has a third parameter called next(). It tells express to run the next middleware. In the example above express would run tapp.get() next, as it is the next middleware. app.get() is a middleware because it sits between the request and response.

2.2.4 Form Submission

There are multiple ways to post data from client to server. The most fundamental way is form submission. You can look at the example below.

Submitting normal data

In index.html

```
<html>
    <head>
            <link href="https://maxcdn.bootstrapcdn.com/bootstrap
                /3.3.7/css/bootstrap.min.css"
                    rel="stylesheet"
                    integrity="sha384-
                        BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va
                        +PmSTsz/K68vbdEjh4u"
                    crossorigin="anonymous">
    </head>
    <body>
        <form action="/login" method="POST">
            <div class="form-group">
                <label> Email:
                    <input type='text' name="email" class="form-
                        control" placeholder="Please input Email
                        Here"/>
                </label>
            </div>
            <div class="form-group">
                <label>
                    Password:
                    <input type='password' name="password" class="
                        form-control"/>
                </label>
            </div>
            <input type='submit' value="Submit" class="btn btn-
                primary"/>
        </form>
        <script
        src="https://code.jquery.com/jquery-3.2.1.min.js"
        integrity="sha256-
            hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
        crossorigin="anonymous"></script>
        <script src="https://maxcdn.bootstrapcdn.com/bootstrap
            /3.3.7/js/bootstrap.min.js"
            integrity="sha384-
                Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA712mCWNIpG9mGCD8wGNIcPD7Txa
                "
            crossorigin="anonymous"></script>
    </body>
</html>
```

Then in your app.js

```
const express = require('express');
const bodyParser = require('body-parser');

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

app.use(bodyParser.urlencoded({ extended: false }))


app.get('/',(req,res)=>{
  res.sendFile(path.join(__dirname,'index.html'));
});

app.post('/login',(req,res)=>{
  console.log(req.body);
  res.send("Login infomation received.");
});
```

Uploading files

Let's add a extra file input profile to our login form.

```
<html>
    <head>
            <link href="https://maxcdn.bootstrapcdn.com/bootstrap
                /3.3.7/css/bootstrap.min.css"
                    rel="stylesheet"
                    integrity="sha384-
                        BVYiiSIFeK1dGmJRAkycuHAHRg32OmUcww7on3RYdg4Va
                        +PmSTsz/K68vbdEjh4u"
                    crossorigin="anonymous">
    </head>
    <body>
        <form action="/login" method="POST" encType="multipart/form
            -data">
            <div class="form-group">
                <label> Email:
                    <input type='text' name="email" class="form-
                        control" placeholder="Please input Email
                        Here"/>
                </label>
            </div>
            <div class="form-group">
                <label>
                    Password:
                    <input type='password' name="password" class="
                        form-control"/>
                </label>
            </div>
            <div class="form-group">
                <label>
                    Profile:
                    <input type='file' name="profile" class="form-
                        control"/>
                </label>
            </div>
            <input type='submit' value="Submit" class="btn btn-
                primary"/>
        </form>
        <script
        src="https://code.jquery.com/jquery-3.2.1.min.js"
        integrity="sha256-
            hwg4gsxgFZhOsEEamdOYGBf13FyQuiTwlAQgxVSNgt4="
        crossorigin="anonymous"></script>
        <script src="https://maxcdn.bootstrapcdn.com/bootstrap
            /3.3.7/js/bootstrap.min.js"
            integrity="sha384-
                Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGNIcPD7Txa
                "
            crossorigin="anonymous"></script>
    </body>
</html>
```

Please note that the input has to be of the type file. On the server side, you have to install the package express-fileupload using npm.

```
npm install --save express-fileupload
```

And the new app.js is as follow,

```
const express = require('express');
const bodyParser = require('body-parser');
const fileUpload = require('express-fileupload');

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

app.use(bodyParser.urlencoded({ extended: false }))
app.use(fileUpload());

app.get('/',(req,res)=>{
  res.sendFile(path.join(__dirname,'index.html'));
});

app.post('/login',(req,res)=>{
  console.log(req.body);
  console.log(req.files);
  res.send("Login Information Received.");
});

app.listen(8080)
```

Please make sure you import the express-fileupload and setup-express to use it as a middleware by the following line.

```
app.use(fileupload());
```

Then the file content is accessible through req.file. In this case, the file is accessible through req.file.profile

Example. Serve the flowershop website via express.

2.3 Express middleware

So far we have learned how to set up our own API and receive all sorts of requests. However, there is one small problem. If we receive a POST request we cannot read the body of that request. If you console.log(req.body) you?ll see a undefined.

The reason for that is that the creators of Express try to keep the code as lightweight as possible. That means that all code that is not required to run the most basic functions is excluded from the express package itself. Any additional code has to be added manually through separate npm modules.

If you think about it, we don?t need the code to read the body of a POST request, if we only want to set up an API that handles GET requests. In the case we need it, we can always add it to Express later, which makes Express so powerful and at the same time lightweight.

2.3.1 Body parser

To be able to read the body of a POST request we need a module called body-parser.

Let's npm install it

```
npm install --save body-parser
```

To use it we make use of app.use

```
var express = require('express')
var bodyParser = require('body-parser')

var app = express()

// parse application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: false }))

// parse application/json
app.use(bodyParser.json())
```

The difference between urlencoded and json is the format in which we receive the body. Either you receive the post request encoded within the URL (more about that below) or it is passed along the request body in a JSON format. body-parser allows you to read both requests.

So now we can read the request body.

```
app.post('/test', (req, res) => {
  console.log(req.body);
  res.send('post successful');
});
```

here are a lot of other useful npm modules which we can use as middleware. You can find a complete list here

2.3.2 Url query string

A URL consist of several parts:

- http:// defines the protocol, you will usually come across this version or the secured https
- www.website.com the domain of the website. The domain again consist of a few parts, the www is considered the subdomain. Whereas website.com is called the domain name. .com itself is called a domain. With domains we differentiate between Top Level Domains (.com, .net),
- /path/to/somewhere after the domain we find the path, which leads to a specific file or API route
- play?vidID=8372395 sometimes we also have parameters in the domain, they contain a ?. These are used to pass along information, like the this example which video to play.

Let?s take a better look at the parameters which form something called a query strings. They always start with a ? which is used to differentiate them from the path, since the server needs to know where does the path end and the query string begin.

We can stuff all sorts of information in these query strings, for example imagine we have a database of workers and want to retrieve all males above height 175cm. A query string could look like this

Example.

Example. Make an app that has a POST route. If we call the POST route with an array of numbers it should return the sum of all the numbers in the array. Test it out with Postman. Receive the array via the body.

```
var arr = [1,2,3,4] // -> sending that to the API returns 10
```

2.4 Request Response Cycle

Every http action consists of two parts, a **request** and a **response**. Every time somebody makes a request to our server we should respond with an appropriate answer. Imagine somebody greets you, it would be extremely rude if you wouldn?t greet back, you wouldn?t follow common social etiquette. The same can be said for servers, if you receive a request it is extremely rude to the users browser sending the request, if you don?t answer back.
Therefore express is structured around the two fundamental components, request and response, or req and res as we call them in our code.

2.4.1 Request

2.4.2 Response

2.4.3 Visualisation

2.5 Cookies and Session

You already learned Cookies Before With Express we have two options to deal with cookies. We can either store a large part with the user or keep most of the information on our servers. The difference is using cookies or cookie sessions. Lets look at each one separately.

Note that passport.js also makes use of cookies and handles these for us if we use it. Using cookies usually isn?t required if we use passport. However passport isn?t always the solution, so we need to be able to set cookies ourselves.

2.5.1 cookie parser

If we want to make use of cookies we need a module called cookie-parser It allows us to read cookies on the request with req.cookies as well as set cookies using res.cookie

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

var app = express();
app.use(cookieParser());

app.get('/', (req, res) => {
  console.log(req.cookies);
  res.cookie('name', 'tobi', { expires: new Date(Date.now() + 1000
      * 60 * 60 * 24 * 14), httpOnly: true });
})
```

In the above example we set the cookie for that person calling our root route. We give the cookie a property name and a value of tobi (should be unique name that allows identification). Then we set the cookie options, like the expiration date, when the cookie will expire.
From now on we can uniquely identify this user, as long as the user doesn?t delete the cookie.
There are a lot more properties for res.cookie check them out in the official documentation.

2.5.2 cookie session

The other way to identify a user is using session. Sessions are information stored about the user on the server side, usually in a redis db, but could also be a postgres.
To use sessions we can use a package called cookie-session. Cookie Session stores an identifier (which we define) in a cookie on the client side. However since the user could manually manipulate the cookie to gain access to other peoples accounts, cookie session also stores a hash of that identifier with the identifier itself.
The hash works just like the hashing of user passwords. The identifier is hashed with a secret key to result in an impossible to guess combination. With every request from the user cookie sessions will look at the identifier and hash it together with the secret. If the resulting hash matches the one stored in the cookie, we can be certain that the cookie hasn?t been tempered with.

```
var express = require('express');
var cookieSession = require('cookie-session');

var app = express();
app.use(cookieSession({
  name: 'session',
  secret: 'a hard to guess secret' ,

  // Cookie Options
  maxAge: 24 * 60 * 60 * 1000 // 24 hours
}))

app.get('/', (req, res) => {
  console.log(req.session);

  if(!req.session.id) {
    req.session.id = 'unique identifier'
  }

  findUser(req.session.id) // return information about the user
})
```

In the above example we add unique id to the user?s session if it doesn?t exist. We then call a function to retrieve information about the user based on the unique ID so we can respond dynamically with content based on the user. Please note we can store more information in the session that simply an ID, it?s pretty much up to us what we store there. However we should try to keep it to a minimum if possible, as to don?t make the user store too much information in his browser. On top of that is the maximum cookie size 4093 bytes.

3 Promises

3.0.1 Promises

So far we have mainly relied on callbacks for our asynchronous code, which has long been the standard. However you will have noticed that error handling can quickly become very messy as we chain several callbacks together. Luckily with the new ES6 standard a new feature called Promise got introduced.

You can imagine a Promise like the word suggest. You give it some JavaScript code and it promises to return you something. Either everything went well or an error occurred, either way you will get an answer from the Promise once it knows what happened.

A Promise consist of three states:

– unresolved→waiting for something to finish

– resolved→something finished and all went well

– rejected→something finished and an error occurred

By default a Promise is the unresolved state and waiting for something to occur. That something can be any kind of JavaScript function, for example a Get request.

Let?s declare a promise to see what is happening

```
var promise = new Promise((resolve, reject) => {
    resolve();
});
console.log(promise);
```

Try running the above code in your browser console. Afterwards replace the resolve() with reject(). What do you see?

You will notice that first of all you get back an object, which indicates whether the Promise was resolved or rejected. If you look at promise˙__proto__ you will see the object has two methods, catch and then.

You will further notice that you get an error message if you run the code with reject(). That error message basically tells you, you haven?t handled the error. So let?s solve that by make use of the two methods.

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

promise.then(() => {
    console.log('Promise resolved.');
});

promise.catch(() => {
    console.log('An error occurred');
});
```

Run the above code, then replace the resolve() with reject(). Notice how textcolorredthen() only runs if the the promise was resolved and catch() only runs when it was rejected.

The real power of promises comes to light when we start chaining different .then() together. Let?s refactor to the code to some ES6 standard and chain everything together.

```
const promise = new Promise((resolve, reject) => {
    resolve();
});

promise
    .then(() => console.log('I ran'))
    .then(() => console.log('I ran afterwards'))
    .then(() => console.log('Then I ran'))
    .catch((err) => console.log('uh oh error', err));
```

If you run the code above you will notice that you can do some pretty powerful chaining with promises, without having to worry about callbacks and error handling. You only need to handle the error once. To demonstrate that let?s change our code

```
const promise = new Promise((resolve, reject) => {
    resolve();
});

promise
    .then(() => console.log('I ran'))
    .then(() => { throw new Error('Help I am an error'); })
    .then(() => console.log('I did not run'))
    .catch((err) => console.log('uh oh error', err));
```

Run the code above to test it.

Why did we change the arrow function to one with curly brackets? That is because without the curly brackets the arrow function will automatically return the inner content. However we don?t want to return anything, but instead throw an error. Therefore we need to wrap it in curly brackets to disable the return.

(Notice if you run the above code and get an error variable already declared, refresh the page. That error occurs because of the const declaration, promise can only be declared once, replace it with var if you want or omit);

3.0.2 fetch

Promises are often used for AJAX requests, as these are processes that are asynchronous and a Promise makes life a lot easier handling their responses. Let?s demonstrate it using a new ES6 feature fetch() which makes simple AJAX requests and returns a Promise.

```
const url = 'https://jsonplaceholder.typicode.com/posts';

fetch(url)
    .then(response => response.json())
    .then(data => console.log(data))
    .catch(err => console.log(err))
```

With fetch we have to make two requests, in the first one we receive an object, however without a body, so we have to fetch the body using response.json(). Chaining the requests together makes this all a breeze to write. Try manipulating the domain to something like typicode123.com to see how the errors are handled. The implementation of fetch isn?t ideal, especially it?s error handling. While it is nice to have a native solution, other solutions like axios make a lot better use of Promises for API requests.

3.0.3 Axios

Axios is another package that makes a lot of uses of Promises. Here is an example of using axios.

```
// GET method
axios.get("/users")
    .then(res=> console.log(res))
    .catch(err=> console.log(err));

// POST method
axios.post("/users",{"id":2,"username":"newUserName"})
    .then(res=> console.log(res) )
    .catch(err=> console.log(err) );

// DELETE method
axios.delete("/users",{id:3})
    .then(res=> console.log(res))
    .catch(err=> console.log(err));

// PUT method
axios.put("/users",{"email":"user@user.com","password":"1234"})
    .then(res=> console.log(res))
    .catch(err=> console.log(res));
```

3.0.4 Applying Promise to asynchronous code

A lot of code we use in node is asynchronous, Promises can help us clean that up.

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

function readdir(path) {
    return new Promise((resolve, reject) => {
        fs.readdir(path, (err, files) => {
            if (err) {
                reject(err);
            } else {
                resolve(files);
            }
        });
    });
}

module.exports = readdir();
```

Now if you require readdir() in another module you can chain .then() and .catch to it.

3.0.5 Promise.all

If you have several Promises you can rely on Promise.all to determine once they finished running. Promise.all itself is a promise that resolves when all promises resolved, if one fails it rejects. All you need to pass it is an array of promises.

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, 'one');
});
const p2 = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'two');
});
```

```
Promise.all([p1, p2])
    .then(values => console.log(values))
    .catch(err => console.log(err));
```

3.0.6 Async & Await

ES7 supports new keyword called async and await to further simplify your promise method call.

```
// Async Version
async function callAPIs(){
    try{
        let res1 = await axios.get("http://example.com")
        let res2 = await axios.get(`http://example2.com?key1=${res1
            .key2}`)
        return res2.key2;
    }catch(err){
        console.log(err);
        throw err;
    }
}

// Promise version
function callAPIs(){
    return new Promise((resolve,reject)=>{
        axios.get("http://example.com")
            .then((res1)=>axios.get(`http://example2.com?key1=${
                res1.key2}`))
            .then((res2)=> resolve(res2.key2))
            .catch((err)=>{
                console.log(err);
                reject(err);
            });
    });
}
```

As shown above, the async function is correspondent to a new Promise object.

```
// Async Version
async function callAPIs(){
    return doSomething();
}

//Promise Version
function callAPIs(){
    return new Promise((resolve,reject)=>{
        resolve(doSomething());
    })
}
```

Each catch block is correspondent to the catch method call in the promised version.

```
// Async version
async function callAPIs(){
    try{
        let res1 = await axios.get('http://example.com');
        return res1;
    }catch(err){
        console.log(err);
        throw err;
    }
}

// Promised version
function callAPIs(){
    return new Promise((resolve,reject)=>{
        axios.get("http://example.com")
        .then((res1)=>resolve(res1))
        .catch((err)=> reject(err))
    });
}
```

Remark.

- Each return in the async version is correspondent to the resolve function call in the promised version.
- Each throw err in the async version is correspondent to the reject function call in the promised version.

3.0.7 Async version $\Longleftrightarrow$ Promise version Correspondence

Here is a quick comparison between the twos,

- async function $\Longleftrightarrow$ new Promise Object
- await keyword $\Longleftrightarrow$ then() method call
- catch block $\Longleftrightarrow$ resolve() method call
- return keyword $\Longleftrightarrow$ resolve() function call
- throw error $\Longleftrightarrow$ reject() function call

3.0.8 Promise.all for Async

You can also use Promise.all in async function when you want to wait for multiple promises before proceeding.

```
async function callAPIs(){
    let combinedPromise = Promise.all([axios.get('http://example1.
        com'),
                                                                axios.get('http://example2.
                                                                    com')]);
    let responses = await combinedPromise;
    return responses;
}
```

Example. Create a module that contains a promised version of fs.readdir and fs.stat. Similar to the example above.

- Then download this file structure
- Use the modules to determine whether each path leads to a directory or file. Read the documentation for fs properly you?ll find all the necessary knowledge in there.

– Output should look like this

```
/files/project_A/ is a directory
/files/project_A/README.md is a file
/files/project_A/images/ is a directory
...
```

4 Assignment 1 - dropbox with cache

4.1 Introduction

4.2 Questions and solutions

5 Jasmine

5.1 Introduction

Basics & Adv.

5.2 Expectation and Matcher

5.3 Setup and Teardown

5.4 Jasmine Spies

5.5 Async and Clock

5.6 Supertest

6 API and Handlebars

6.1 Introduction

6.2 Restful APIs

6.3 API Methods

6.3.1 API Structure

6.3.2 HTTP Methods

Now it?s time to introduce the verbs to the equation. The verbs are going to be the four main http methods.

- GET Ask the server to retrieve some resources
- POST Ask the server to create a new resource
- PUT Ask the server to edit/update an existing resource
- DELETE Ask the server to remove an existing resource

With these four verbs we can basically do anything with our API. In case of our pizza parlor it would work as follows:

Request	**endpoint**	**action**
GET	/orders	List all the orders
POST	/orders	Add new order
GET	/orders/1	List order with ID 1
GET	/orders/2	List order with ID 2
PUT	/orders/1	Update order with ID 1
DELETE	/orders/1	Delete order with ID 1

6.3.3 Express example

The Restful API can thus be translated to the Express application as follow,

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

// Remember to put necessary plugin here.

app.get('/orders/:id',(req,res)=>{
    const id = req.params.id;
    const order = // Logic that read order by id.
    res.json(order);
});

app.get('/orders',(req,res)=>{
    const orders = // Logic that reads orders from somewhere.
    res.json(orders);
});


app.post('/orders',(req,res)=>{
    const newOrder = req.body.order; // Remember to use body-parser
        .
    // Logic that create a order
    res.json({success:true});
});

app.put('/orders/:id',(req,res)=>{
    const id = req.params.id;
    //Logic that updates the order
    res.json({success:true})
});


app.delete('/orders/:id',(req,res)=>{
    const orderIdToBeDeleted = req.params.id;
    // Logic that deletes the order
    res.json({success:true})
});
```

6.3.4 API Response

Every request to our API needs an adequate response. Whereas it is very obvious how to respond to a GET request, what is an adequate response to a POST request?
First of all the software posting wants to know that the POST request was successful, so we need to respond with an adequate header of 200 if the request was properly handled. Further by convention we would respond with some sort of unique identifier of that post request. In the above example of the pizza order, we would respond with the order ID.
For a PUT or DELETE request, a response with a header 200 would be adequate.

6.3.5 Scaling the API

Imagine now our Pizza Parlor is booming and the chief Pizza baker decides to reward good customers for their loyalty. For that we need to keep track of customers and give each customer a unique identifier.

How would we go about that? There are some possible ways. The most obvious one would be to create another resource called /customers. So that we could make GET requests to /customers/customer_id and so on.

However, how do we link a customer to his order? Opinions are split here in the developer community amongst two main solutions and you?ll encounter both. Either we extend our customer endpoint customers/5/orders to get the orders of customer 5 and customer/5/orders/3 to get customer 5?s third order. Or we keep everything as flat as possible and when making a request to orders pass along the customer ID. Both ways are perfectly legit according to REST principles and it is up to you how you?d like to structure your API.

So a request to order a POST pizza would look like

POST /customers/5/orders

```
{
    "type": "napoletana",
    "toppings": "extra cheese"
}
```

POST/orders

```
{
    "type": "napoletana",
    "toppings": "extra cheese",
    "customer_id": "5"
}
```

6.3.6 REST vs RPC vs GraphQL

There are multiple styles of API structures out there. You probably read about it somewhere in internet.

- **REST** is the one we have been discussing in this article.
- **RPC** is the short form of Remote Procedural Call which differs with REST in the format and the semantics of the url.
- **GraphQL** is the newcomer which use a single endpoint with a newly introduced query language GraphQL

Rest vs RPC
Here are two contrasting examples of Rest-style API & RPC-style API

```
//REST
GET   /friends
POST  /friends
PUT   /friends
DELETE /friends

//RPC
GET  /friends
POST /addFriends
POST /updateFriends
POST /deleteFriends
```

In REST, the action is represented by the HTTP method while in RPC, the action is represented by the url route. Because the HTTP verb is well-defined, the developers can understand your API easier.
Rest vs GraphQL
GraphQL is the new contender of REST. It features a flexible query language and type system to make writing complex API call easier and more efficient. You can read more about it here

6.4 HTML Template

6.5 Handlebars

6.5.1 Server side vs Client side rendering

Rendering is the process of picking a template and populating it with the data we needed to create the HTML file. Here are the main difference between server-side rendering and client-side rendering:

- Server Side Rendering:

    - The rendering process is happened on the server side making use of the templates and data.
    - It facilitates form submission and page refresh
    - Examples: Handlebars

- Client Side Rendering:

    - The rendering process is happened on the client side using javascript and HTML templates.
    - It facilitates AJAX Call and DOM manipulations
    - Example: JQuery with HTML Template, React and Anugular

We are going to employ a MIXED approach for this problem:

- We are going to use Server side rendering to render the layout of your pages.
- We are going to use AJAX and Client side rendering to populate your page with the necessary data.

6.5.2 Handlebars

Handlebars is an extremely lightweight template engine. Handlebars is used by some of the biggest websites on the Internet. It is extremely simple to use, we run with res.render and it returns us HTML pages ready to send to the users. Express is build with support for template engines, making it extremely straightforward to plug in Handlebars.

Create a new npm project with npm init. Further install npm install express. To install handlebars we can use the express-handlebars module.

```
npm install express-handlebars
```

6.5.3 Basic folder structure

To use Handlebars we need to save our files in certain structure. Create a views folder and two subfolders layouts and partials.

```
app.js // main file
views
    -layouts
    -partials
```

6.5.4 Main Handlebars

The main page structure is saved in a file called main.handlebars. It contains the HTML code that that sets the page structure. Its actual content can be dynamically replaced, and will be rendered where the {{{body}}} element is. Add it to the layouts folder

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>...</title>
    </head>
    <body>

            {{{body}}}

    </body>
</html>
```

Let's add a simple app.js file

```
const express = require('express');
const app = express();
const hb = require('express-handlebars');

app.engine('handlebars', hb({ defaultLayout: 'main' }));
app.set('view engine', 'handlebars');

app.get('/people', function(req, res){
    res.render('person');
});


app.listen(8080);
```

Notice in the file above how we set handlebars as the app.engine which is used to register a template engine with express. We set the default layout to main, which is the main.handlebars file we created earlier. Afterwards we set the view engine to handlebars.

In the get request below we don?t use res.send but instead use res.render([view]). The view defines which file from the view folder to render. So in this case , you need a file called person.handlebars withing the folder views.

Let?s take a look at layouts, partials and helpers.

6.5.5 Layouts

If you remember the ?views? folder we created had the following structure:

```
views
    - layouts
        - main.handlebars
    - partials
```

We put the main.handlebars file in the ?layouts? folder. You might have wondered why there is a special folder just for main.handlebars. The reason is that we can have several templates in there. Depending on the page section we might want to have a different structure, so we can put that there.

Let?s add a second layout, we will call it otherPage.handlebars:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>...</title>
    </head>
    <body>
        <h1>I am a different template </h1>

            {{{body}}}

    </body>
</html>
```

When we call res.render() we can manually update the default layout in the object we are passing to the function. Look at the object below, we give it a property called layout where we define the new layout page to use for that route.

```
app.get('/something', (req, res) => {
    res.render('view', {layout:"otherPage"})
})
```

6.5.6 Partials

The other folder you probably noticed is the partials folder. Partials are elements that keep on reappearing throughout the app. For example a header or a footer. Since we can have several layouts but might want to share a header across layouts it makes no sense to hardcode the header into a layout.

We instead want to have a dynamic and sharable header as well. Let?s add one.

In the partials folder add a file called header.handlebars:

```
<h1>I am a header </h1>
```

In our otherPage.handlebars file we add the header with {{¿ header }} :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>...</title>
    </head>
    <body>

        {{> header }}


        <h1>I am a different template </h1>


        {{{body}}}

    </body>
</html>
```

Render it and you should see the header appear.

6.6 Basic Authentication

The simplest way to authenticate is using password and username, something we call Basic Authentication (Basic Auth). It is pretty simple and pretty old school. However while ?old school = gold school? holds true for Hip Hop, it doesn?t necessarily hold true for technology. Let?s try to understand why.

We simply pass username and password in the header, in a part called Authorization. So every call we make to an API includes a header with the password and username. Anybody with those two can access the API. However what usually happens is that many people work on an application, including outsiders. Using basic auth gives everybody access to the user credentials and anybody with malicious intends can misuse them and create a mess with your account. You can compare it to giving out the master key to all hotel rooms instead of giving guests only a key to their room. That?s why companies came up with a different solution.

6.6.1 Adding Basic Auth to Express

You can easily add basic authentication to express using the package express-basic-auth. The following code adds a Basic Authentication for your express application.

```
var app = require('express')()
var basicAuth = require('express-basic-auth')

app.use(basicAuth({
    users: { 'admin': 'supersecret' },
    challenge: true,
    realm: 'My Application'
}))
```

The username is this admin and the password is supersecret. The only problem of this authentication is that you cannot keep the login status if you close your browser. And you cannot logout without closing your browser.

6.6.2 Authenticating different users

You can also use the authorizer in this package to authenticate different users.

```
app.use(basicAuth({
    authorizer: myAuthorizer,
    challenge: true,
    realm: 'My Application'
}));

const USERS = [
    {
        "username":"gordon",
        "password":"abc"
    }
]

function myAuthorizer(username, password) {
    return USERS.some((user)=>{
        return user.username == username && user.password ==
            password
    })
}
```

Because by default the express-basic-auth does not challenge the user. We have to add the attribute for challenge to be true. realm defines the area of effectiveness for this authentication which Express and the browser handle it for us automatically.

6.7 Postman

6.7.1 What is Postman?

Postman is a popular tool for testing HTTP request/response, espeically RESTful APIs.

6.7.2 Where can I get Postman

Download it here

7 Assignment 2 - notes taking file

8 SQL

8.1 Introduction

A relational database (RDB) is a collective set of multiple data sets organized by tables, records and columns. RDBs establish a well-defined relationship between database tables. Tables communicate and share information, which facilitates data searchability, organization and reporting. You can imagine them being very similar to an excel spreadsheet, with rows and columns. Whereas a database is more like an excel workbook, which consists out of several spreadsheets.

RDBs use Structured Query Language (SQL) which allows us to program on these databases and query them for data. While there are many SQL databases out there we will be using PostgreSQL.

8.1.1 Setting up

Mac
If you have a Mac the easiest way to install PostgreSQL is via the PostgresApp.
Ubuntu/Windows
If you are using linux (ubuntu) you can run the following code

```
sudo apt-get install postgresql postgresql-contrib
```

You can also consider downloading a graphical interface software like pgAdmin. (linux, windows, mac) . We won?t use it for now, but they can help you with debugging at a later stage.

You should start your database server by running the command below.

```
sudo service postgresql start
```

8.1.2 Creating a database

By default postgresql creates a user called postgres. The postgres user has all of the permission on working on the DB. Therefore in order to perform the operation below, you can run the command below.

```
sudo su postgres
```

You are now logged in to the user postgres in ubuntu. Because of the mechanism of peer authentication. You won?t need a password to access the DB using the user postgres if you are at the same machine as the postgresql server.

Using the createdb command creates a new database. Please note that the following commands are run in **BASH**.

```
createdb fruits
```

Now you can work the database using PostgreSQL. You will notice that your console changes and you are in the database.

```
psql fruits
```

The above command is only a shortform. Here is the full version of a generic command for logging to postgresql server.

```
The above command is only a shortform. Here is the full version of
    a generic command for logging to postgresql server.
```

8.1.3 Setting up user

Example.

8.2 Adding, updating and deleting data

8.3 Querying data

Example.

8.4 Node & PostgreSQL

8.4.1 PostgreSQL

Example.

8.4.2 SQL injection

8.5 SQL joins

When there are more than one table in a database, we have to establish relation between them. We are using primary and foreign key to establish relation among different tables.

8.5.1 Primary Key and Foreign Key

Primary key is a unique identifier for each row of data. You have already encounter it in the previous section. It is the id SERIAL NOT NULL

Foreign key simply means the primary keys of the other table. They are fundamental to linking two SQL tables together. To join two tables together using foreign key, we normally use the SQL Join. We have other ways like sub-queries to achieve the same thing. But SQL Join is the preferable way of doing this.

8.5.2 Join

8.5.3 Different type of JOINs

Example.

8.6 Transaction

8.6.1 Database Transactior

9 Knex

9.1 Knex.JS

9.2 Knex migrations CLI

10 Database Theory

10.1 Introduction

10.2 Associations

Database design should be the first step in your database-driven application development. It is often said that we should always design our code around the data because once the database design is designed well, normally the code to manipulate the data will be trivial.

10.2.1 Different Types of Associations

There are mainly three types of association. To illustrate the differences , we can look at the tables of a hypothetical school.

- One-to-One :
- Many-to One and One-to-Many :
- Many-to-Many :

One-to-One

Many-to-One and One-to-Many

Many-to-Many

10.3 Database Design

11 Assignment 3 - notes taking knex

12 Redis

12.1 Introduction

12.2 Redis and node

13 Websockets

13.1 Introduction

13.2 Socket.io

```
//server.js
const express = require('express');
const app = express();
const http = require('http').Server(app);
const io = require('socket.io')(http);

app.get('/', (req, res) => {
    res.sendFile(__dirname + '/index.html');
});

io.on('connection', (socket) => {
    console.log('a user connected to the socket');
});

http.listen(3030);
```

As you can see above we set up a very simple server using express. The server only serves one file, the index.html file. interestingly we listen using the http module internal to node. That is because socket.io makes use of this module. We then require socket.io and pass it the http module.

To listen out for events using socket.io we use the .on method and specify the event we are listening for, in our case the connection event.

Let?s add the index.html file so the code actually works.

```
<html>
<head>
    <style>
        * { margin: 0; padding: 0; box-sizing: border-box; }
        body { font: 13px Helvetica, Arial; }
        form { background: #000; padding: 3px; position: fixed;
            bottom: 0; width: 100%; }
        form input { border: 0; padding: 10px; width: 90%; margin-
            right: .5%; }
        form button { width: 9%; background: rgb(130, 224, 255);
            border: none; padding: 10px; }
        #messages { list-style-type: none; margin: 0; padding: 0; }
        #messages li { padding: 5px 10px; }
        #messages li:nth-child(odd) { background: #eee; }
    </style>
</head>
<body>
    <ul id="messages"></ul>
    <form action="">
        <input id="m" autocomplete="off" /><button>Send</button>
    </form>

<script src="https://cdnjs.cloudflare.com/ajax/libs/socket.io
    /2.0.3/socket.io.slim.js"></script>
<script src="https://code.jquery.com/jquery-1.11.1.js"></script>
<script>
    var socket = io();
</script>
```

```
</body>
</html>
```

Let?s add another event listener to our server file. This time we will listen on the socket (the parameter passed in the callback) directly. The code should look like this:

```
io.on('connection', (socket) => {
    console.log('a user connected to the socket');
    socket.on('disconnect', () => console.log('a user left us'));
});
```

13.2.1 emitting events

In order to actually listen to events we also need to be able to emit them. Besides a few events like connection few events are predefined in socket.io

In our case we want to create a simple chat, that means the user will add messages, which would be an ideal event to listen for. So let?s edit the client side logic.

```
<script>
  $(function () {
    var socket = io();
    $('form').submit(function(){
      socket.emit('chat message', $('#m').val());
      $('#m').val('');
      return false;
    });
  });
</script>
```

Every time the user adds a new message, socket.io will fire off an event which we can listen out for. So let?s add the relevant code on the server.

```
io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    console.log('message: ' + msg);
  });
});
```

13.2.2 Broadcasting

Now it is time to send something back from the server. Since we want to inform all the participants of the new message. We can broadcast the message back to all listeners. Let?s update our server code.

```
io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});
```

As you can see above all we do is listen for a message and then rebroadcast this message to all listening sockets. We do that by emitting an event which the clients listen for.

Now we need to listen out on the client for the server broadcast and render the message to the user. Let?s update the html file.

```
<script>
  $(function () {
    var socket = io();
    $('form').submit(function(){
      socket.emit('chat message', $('#m').val());
      $('#m').val('');
      return false;
    });
    socket.on('chat message', function(msg){
      $('#messages').append($('<li>').text(msg));
    });
  });
</script>
```

13.2.3 Using SocketIO with express

In order to use SocketIO with express , you have to use the package http to retrieve the server instance from express. Here is the official example from SocketIO:

```
var app = require('express')();
var server = require('http').Server(app);
var io = require('socket.io')(server);


// Call listen method ON SERVER!! Not app
server.listen(80);

app.get('/', function (req, res) {
  res.sendfile(__dirname + '/index.html');
});

io.on('connection', function (socket) {
  socket.emit('news', { hello: 'world' });
  socket.on('my other event', function (data) {
    console.log(data);
  });
});
```

13.2.4 Namespaces

By default, SocketIO is using the namespace /. You can actually switch to other namespace to use it with another endpoint(Not URL!!). You can actually making use of the namespace by the method of on the server-side. On the server side,

```
const namespace1 = io.of('/namespace1');
namespace1.on('connection',(socket)=>{
  console.log(`Socket with id ${socket.id} connected!`);
});
namespace1.emit('hello',"World!");
```

On the client side,you can access this namespace1 by providing the name of the namespace.

```
let socket = io('/namespace1')
```

13.2.5 Rooms

Namespaces are good if you want to completely separate the clients. But if you want to concept of subscribing, which one client can listen to multiple channels at the time. Then you have to use Rooms instead. Here is one simple subscribe event that can hook the respective socket to the room.

```
//Server Side
io.on('subscribe',(room)=>{
  socket.join(room);
});

//Client side
socket.emit('subscribe','my-room');
```

To simply send messages to a specific room, you can use the method to.

```
io.to('my-room').emit('subscription-reply',"Thanks for subscribing
    my-room!");
```

14 User Authentication

14.1 Introduction

14.2 Passport.js

14.2.1 Users Data

14.3 Bcrypt

14.4 Social Login

14.5 Redis Session

15 System Admin

15.1 Introduction

15.2 Digital Ocean

15.3 apt-get

15.4 Bash

15.5 NodeJs Setup

15.6 Process Manager and Nginx

15.7 Git Deployment

16 Assignment 4 - chatroom single file

17 Extension topics

18 Data Structure

18.1 List, Map and Set

18.2 Stack and Queue

19 HTTP Header

20 Algorithm

20.1 Sorting

20.2 Searching

21 Dependency Injection

22 Operating System

23 Hamming

23.1 Introduction

24 Software Licensing

24.1 Introduction