NodeJS

Building your first web server

What is NodeJS

- Node.js is an open source server environment
- Node.js is free
- Node.js runs on various platforms (Windows, Linux, Unix, Mac OS X, etc.)
- Simply NodeJS is a way to run javascript outside the browser
- NodeJS can be used to run desktop apps, servers,..
- Our focus here is on creating a web server

How Node.js handles a file request

- A common task for a web server can be to open a file on the server and return the content to the client.
- Here is how Node.js handles a file request:
  - Sends the task to the computer's file system.
  - Ready to handle the next request.
  - When the file system has opened and read the file, the server returns the content to the client
- Node.js eliminates the waiting, and simply continues with the next request.
- Node.js runs single-threaded, non-blocking, asynchronous programming, which is very memory efficient.

What Can Node.js Do?

- Node.js can generate dynamic page content
- Node.js can create, open, read, write, delete, and close files on the server
- Node.js can collect form data
- Node.js can add, delete, modify data in your database

What is a Node.js File?

- Node.js files contain tasks that will be executed on certain events
- A typical event is someone trying to access a port on the server
- Node.js files must be initiated on the server before having any effect
- Node.js files have extension ".js"

Getting started

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

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.end('Hello World!');
}).listen(8080);
```

NodeJS modules

- A module is a set of functions you want to include in your application.
- Node.js has a set of built-in modules which you can use without any further installation.
- To include a module, use the require() function with the name of the module.
- For example: const http=require(‘http’)
- Now your application has access to the HTTP module, and is able to create a server:

Create Your Own Modules

- You can create your own modules, and easily include them in your applications.
- The following example creates a module that returns a date and time object:

```
exports.myDateTime = function () {
  return Date();
};
```

- Use the exports keyword to make properties and methods available outside the module file.
- Save the code above in a file called "**myfirstmodule.js**“
- Now you can **include** and use the module in any of your Node.js files.

Node.js HTTP Module

- Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP)
- The HTTP module can create an HTTP server that listens to server ports and gives a response back to the client.

Node.js http.createServer() Method

- The createServer method creates a server on your computer, it turns your computer into an HTTP server.
- The http.createServer() method creates an HTTP Server object.
- The HTTP Server object can listen to ports on your computer and execute a function, a requestListener, each time a request is made.
- Syntax

```
http.createServer(requestListener);
```

- The *requestListener* is Optional. Specifies a function to be executed every time the server gets a request. This function is called a requestListener, and handles request from the user, as well as response back to the user.

Add an HTTP Header

- If the response from the HTTP server is supposed to be displayed as HTML, you should include an HTTP header with the correct content type:

```
.............
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  ..................
}).listen(8080);
```

The first argument of the res.writeHead() method is the status code, 200 means that all is OK, the second argument is an object containing the response headers.

Read the Query String

- The function passed into the http.createServer() has a req argument that represents the request from the client, as an object.
- This object has a property called "url" which holds the part of the url that comes after the domain name (ie the query string)

```
……
http.createServer(function (req, res) {
  …..
res.write(req.url);
  ……
}).listen(8080);
```

Node.js File System Module

- The Node.js file system module allows you to work with the file system on your computer.

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

- Common use for the File System module:
  - Read files
  - Create files
  - Update files
  - Delete files
  - Rename files

Read Files

- The fs.readFile() method is used to read files on your computer.
- Assume we have the demofile1.html file (located in the same folder as Node.js), to read the HTML file, and return the content:

```
var http = require('http');
var fs = require('fs');
http.createServer(function (req, res) {
  //Open a file on the server and return its content:
  fs.readFile('demofile1.html', function(err, data) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(data);
    return res.end();
  });
}).listen(8080);
```

Create Files

- The File System module has methods for creating new files:
  - fs.appendFile()
  - fs.open()
  - fs.writeFile()

fs.appendFile()

- The fs.appendFile() method appends specified content to a file. If the file does not exist, the file will be created:

```
fs.appendFile('mynewfile1.txt', 'Hello content!', function (err) {
    console.log('Saved!');
});
```

fs.open()

- The fs.open() method takes a "flag" as the second argument, if the flag is "w" for "writing", the specified file is opened for writing. If the file does not exist, an empty file is created:

```
var fs = require('fs');
fs.open('mynewfile2.txt', 'w', function () {

  console.log('Saved!');
});
```

Fs.writeFile()

- The fs.writeFile() method replaces the specified file and content if it exists. If the file does not exist, a new file, containing the specified content, will be created:

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

fs.writeFile('mynewfile3.txt', 'Hello content!', function () {

  console.log('Saved!');
});
```

Update Files

- The File System module has methods for updating files:
  - fs.appendFile(): The fs.appendFile() method appends the specified content at the end of the specified file:
  - fs.writeFile(): The fs.writeFile() method replaces the specified file and content:

Fs.writeFile() flags

- r+ open the file for reading and writing
- w+ open the file for reading and writing, positioning the stream at the beginning of the file. The file is created if it does not exist
- a open the file for writing, positioning the stream at the end of the file. The file is created if it does not exist
- a+ open the file for reading and writing, positioning the stream at the end of the file. The file is created if it does not exist

Delete Files

- To delete a file with the File System module, use the fs.unlink() method.

- The fs.unlink() method deletes the specified file:

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

fs.unlink('mynewfile2.txt', function () {

  console.log('File deleted!');
});
```

Rename Files

- To rename a file with the File System module, use the fs.rename() method.

- The fs.rename() method renames the specified file:

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

fs.rename('mynewfile1.txt', 'myrenamedfile.txt', function (err) {
  if (err) throw err;
  console.log('File Renamed!');
});
```

Node.js URL Module

- The URL module splits up a web address into readable parts.

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

url.parse() Split the Query String

- There are built-in modules to easily split the query string into readable parts, such as the URL module.
-

```
.....
var url = require('url');

http.createServer(function (req, res) {
  ........
  var q = url.parse(req.url, true).query;
  var txt = q.year + " " + q.month;
 ..........
}).listen(8080);
```

The **url.parse()** method takes a URL string, parses it, and it will return a URL object with each part of the address as properties.

Parse an address with the url.parse() method, and it will return a URL object with each part of the address as properties:

Another method to parse

- Another way to access query string parameters is parsing them using the **querystring** builtin Node.js module.
- This method, however, must be passed just a querystring portion of a url. Passing it the whole url, like you did in the url.parse example, won't parse the querystrings.

```
const querystring = require('querystring');
const url = "http://example.com/index.html?code=string&key=12&id=false";
const qs = "code=string&key=12&id=false";
console.log(querystring.parse(qs)); //  { code: 'string', key: '12', id: 'false' }
console.log(querystring.parse(url)); //  { 'http://example.com/index.html?code': 'string', key: '12', id: 'false' }
```

See File Server example

- Create a Node.js file that opens the requested file and returns the content to the client. If anything goes wrong, throw a 404 error:

How to build your server

- Step1: create a js file inside your project
- Step2: we need to create the server and tell it to listen on a specific port. This will include the http library into our code, inside the http variable we created

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

- Step3: create a variable to specify the port the server is going to listen to. We will use this later

```
const port=3000
```

- Step4: create server

```
const server=http.createServer(function(req,res){
})
```

- Step5: setup server to listen to the port we are listening to

```
server.listen(port,function(error){
 if(error){
            console.log(“something went wrong”,error)
}else {
            console.log(“server is listening on port” + port)

}}
)
```

- Step6: Run server

```
Node app.js
```

- To end server hit Ctrl+c

Implement the createServer

- If you want to return some response to the user , then use the response object that is passed into the function

```
Res.write(“”)
Res.end()
```

- Most probably you would want to return an file (create a new file called index.html and put anything in the body)
- Now back inside createServer function, which we will be calling with each request to tell it to send html instead of plain text

```
Res.writeHead(200,{‘Content-type’:’text/html’})
```

Parse index.html

- We then need to parse the html file index.html which we created later. To do that we need to import another library

```
Const fs=import('fs')
```

- Then you can use the readFile function of fs that takes the name of the file we want to read as parameter and a function that has an error property if an error happens and a data argument which is going to be all the data from inside that file

Events

- Much of the Node.js core API is built around an idiomatic asynchronous event-driven architecture in which certain kinds of objects (called "emitters") emit named events that cause Function objects ("listeners") to be called.
- All objects that emit events are instances of the EventEmitter class. These objects expose an eventEmitter.on() function that allows one or more functions to be attached to named events emitted by the object.
- When the EventEmitter object emits an event, all of the functions attached to that specific event are called synchronously. Any values returned by the called listeners are ignored and discarded.

Events Module

- Node.js has a built-in module, called "Events", where you can create-, fire-, and listen for- your own events.
- To include the built-in Events module use the require() method.
- In addition, all event properties and methods are an instance of an EventEmitter object. To be able to access these properties and methods, create an EventEmitter object:

```
var events = require('events');
var eventEmitter = new events.EventEmitter();
```

The EventEmitter Object

- You can assign event handlers to your own events with the EventEmitter object.
- In the example below we have created a function that will be executed when a "scream" event is fired.
- To fire an event, use the emit() method.

```
var events = require('events');
var eventEmitter = new events.EventEmitter();

//Create an event handler:
var myEventHandler = function () {
  console.log('I hear a scream!');
}

//Assign the event handler to an event:
eventEmitter.on('scream', myEventHandler);

//Fire the 'scream' event:
eventEmitter.emit('scream');
```

Passing arguments and this to listeners

- The eventEmitter.emit() method allows an arbitrary set of arguments to be passed to the listener functions. Keep in mind that when an ordinary listener function is called, the standard this keyword is intentionally set to reference the EventEmitter instance to which the listener is attached.

```
const myEmitter = new MyEmitter();
myEmitter.on('event', function(a, b) {
  console.log(a, b, this, this === myEmitter);
  // Prints:
  //   a b MyEmitter {
  //     domain: null,
  //     _events: { event: [Function] },
  //     _eventsCount: 1,
  //     _maxListeners: undefined } true
});
myEmitter.emit('event', 'a', 'b');
```

```
myEmitter.on('event', (a, b) => {
  console.log(a, b, this);
  // Prints: a b {}
});
myEmitter.emit('event', 'a', 'b');
```

When you use arrow function as listeners, however, when doing so, the this keyword will no longer reference the EventEmitter instance:

Node.js Upload Files

- The Formidable Module: a good module for working with file uploads, called "Formidable".
- The Formidable module can be downloaded and installed using NPM:

```
npm install formidable
```

- After you have downloaded the Formidable module, you can include the module in any application:

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

Upload Files

- To make a web page in Node.js that lets the user upload files to your computer:

- Step 1: Create an Upload Form
- Step 2: Parse the Uploaded File
- Step 3: Save the File

Step 1: Create an Upload Form

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

http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/html'});
  res.write('<form action="fileupload" method="post"
enctype="multipart/form-data">');
  res.write('<input type="file" name="filetoupload"><br>');
  res.write('<input type="submit">');
  res.write('</form>');
  return res.end();
}).listen(8080);
```

Form enctype attribute values

Value	Description
application/x-www-form-urlencoded	Default. All characters are encoded before sent (spaces are converted to "+" symbols, and special characters are converted to ASCII HEX values)
multipart/form-data	This value is necessary if the user will upload a file through the form
text/plain	Sends data without any encoding at all. Not recommended

Step 2: Parse the Uploaded File

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

http.createServer(function (req, res) {
  if (req.url == '/fileupload') {
    var form = new formidable.IncomingForm();
    form.parse(req, function (err, fields, files) {  //parses json to object
      res.write('File uploaded');
      res.end();
    });
  } else {
    res.writeHead(200, {'Content-Type': 'text/html'});
      ……..
  }
}).listen(8080);
```

When the file is uploaded and parsed, it gets placed on a temporary folder on your computer.

Step 3: Save the File

```
..... var fs = require('fs');
   http.createServer(function (req, res) {
  if (req.url == '/fileupload') {
  var form = new formidable.IncomingForm();
  form.parse(req, function (err, fields, files) {
    var oldpath = files.filetoupload.filepath;
    var newpath = 'C:/Users/Your Name/' + files.filetoupload.originalFilename;
    fs.rename(oldpath, newpath, function (err) {
if (err) throw err;
      res.write('File uploaded and moved!');
      res.end();
    });      });        }.....
```

Web Frameworks

- Other common web-development tasks are not directly supported by Node itself.
- A web framework makes it easier to build common backend logic. This includes mapping different URLs to content, dealing with databases, and templating for dynamic content.
- If you want to add specific handling for different HTTP verbs (e.g. GET, POST, DELETE, etc.), separately handle requests at different URL paths ("routes"), serve static files, or use templates to dynamically create the response, Node won't be of much use on its own. You will either need to write the code yourself, or you can avoid reinventing the wheel and use a web framework!

ExpressJS

- ExpressJS is the most popular Node.js web server framework and is the basis of thousands of sites.
- Express.js is specifically designed for building web applications.
- The Express.js framework makes it very easy to develop an application which can be used to handle multiple types of requests like the GET, PUT, and POST and DELETE requests

Install express server using npm

To start off, and because we need an HTTP server, let's install Express using npm. Installing Express is as simple as:

Command	Output
npm install <package>	Installs <package> using npm
npm install express	Installs express
npm uninstall express	Uninstalls express
npm install express@4	Installs a specific version of express

Other packages that express depends on are also installed.

- After you install express a new folder modules is added which includes all installed packages..

Express

- Express is the best way to run an HTTP server in the Node.js environment.
- We will start with using Express to serve only static files.
- To start using Express, let's import the module and use the top-level function that the module exports, in order to instantiate an application. This can be done using the following code:

```
const express = require('express');   //loading up a module called express
```

- In the case of Express**, the module exports a function that can be used to instantiate an application**. We just assigned this function to the variable express

An express application

- An Express application is web server that listens on a specific IP address and port.
- Instantiate the express application by calling the express() function:

...

```
const app = express();  // we have a handle to the application
//Creates an Express application.
//The express() function is a top-level function exported by the express module
```

The app object returned from this function is one that we use in our application code

...

What is express?

- Express is a framework that does minimal work by itself; instead, it gets most of the job done by functions called *middleware*.
- A middleware is a function that takes in an HTTP request and response object, plus the next middleware function in the chain.
- This middleware function can look at and *modify the request and response objects, respond to requests, or decide to continue with middleware chain by calling the next middleware function*.

Express.static() function

- To serve static files such as images, CSS files, and JavaScript files, use the express.static built-in middleware function in Express.
- express.static function responds to a request by trying to match the request URL with a file under a directory specified by the parameter to the generator function.
- If a file exists, it returns the contents of the file as the response, if not, it chains to the next middleware function. This is how we can create the middleware:

```
...
const fileServerMiddleware = express.static('public');
/*public is the root argument which is the name of the root folder that you have created and where
 the middleware should look for the files relative to where the app is being run */
...
```

The app use() method

- For the application to use the static middleware, we need to *mount* it on the application.
- The **app.use()** is an application method is used to mount the specified middleware function(s) in an express application at the path which is being specified. It is mostly used to set up middleware for your application.

```
/* syntax   app.use(path (optional defaults to /), callback (fileserverMiddleware)
```

The app.listen() method

- Finally, now that the application is set up, we'll need to start the server and let it serve HTTP requests.
- The app.listen() method of the application starts the server and waits for requests.
- The **app.listen() function** is used to bind and listen the connections on the specified host and port.
- It takes in a port number as the first argument and an optional callback that can be called when the server has been successfully started.

```
/* syntax   app.listen([port[, host[, backlog]]][, callback (optional)])
...
app.listen(3000, function () {
console.log('App started on port 3000');
});
```

server.js: Express Server (at root level)

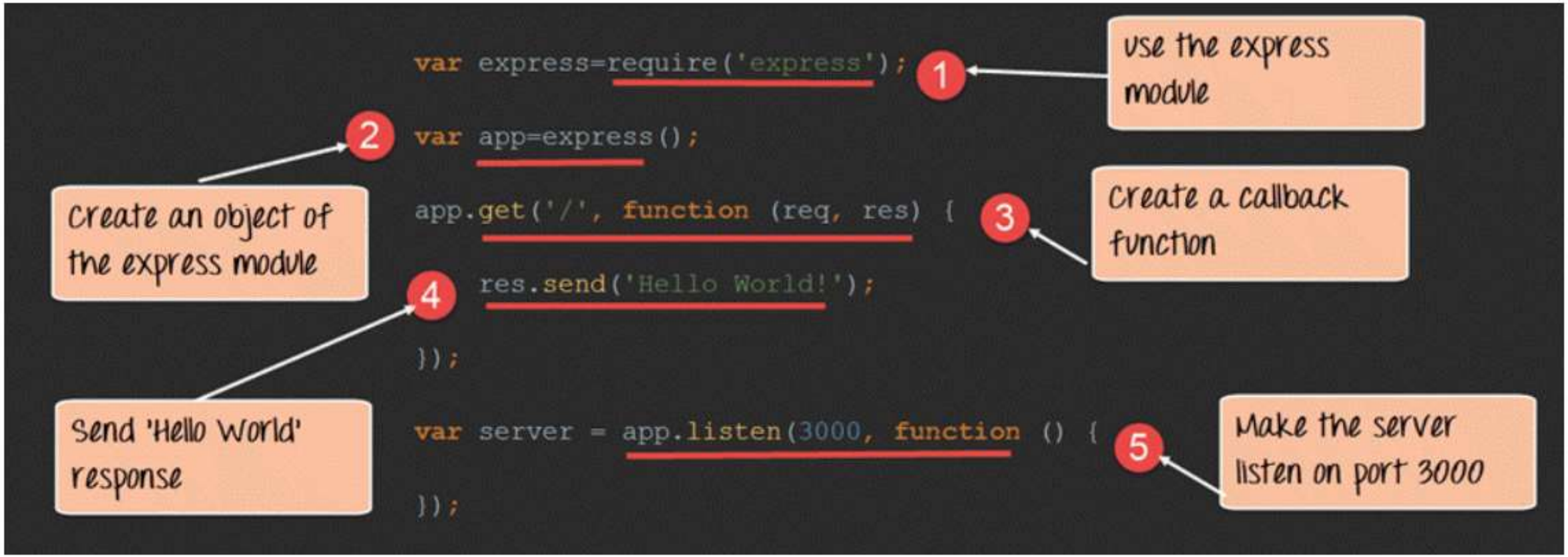
```
const express = require('express');

const app = express();

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

app.listen(3000, function () {
console.log('App started on port 3000');
});
```

Now we are ready to start the web sever and serve index.js

Command to run the server	Output
node server.js	Runs server.js
npm start	Runs the express server

stop the server (using Ctrl+C on the command shell) and restart the server using npm start

Let’s use our newly installed Express framework and create a simple “Hello World” application.



Routing

- Routing **defines the way in which the client requests are handled by the application endpoints**.
- There are two ways to implement routing in node.js
  1. By Using Framework
  2. Without using Framework

ExpressJS - Routing

- Express.js has an “app” object which has methods that are used to define routes. This app object specifies a callback function, which is called when a request is received. We have different methods in app object for a different type of request.

For GET request use app.get() method:

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


app.get('/', function(req, res) {
    res.send('Hello Sir')
})
```

For POST request use app.post() method:

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


app.post('/', function(req, res) {
    res.send('Hello Sir')
})
```

For PUT request use app.put() method:

```
app.put('/user', function (req, res) {
  res.send('Got a PUT request at /user')
})
```

For DELETE request use app.delete() method:

```
app.delete('/user', function (req, res) {
  res.send('Got a DELETE request at /user')
})
```

For handling all HTTP methods (i.e. GET, POST, PUT, DELETE etc.) use app.all() method:

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


app.all('/', function(req, res) {
    console.log('Hello Sir')
    next()   // Pass the control to the next handler
})
```

So, to perform routing with the Express.js you have only to load the express and then use the app object to handle the callbacks according to the requirement.

Routing without Framework:

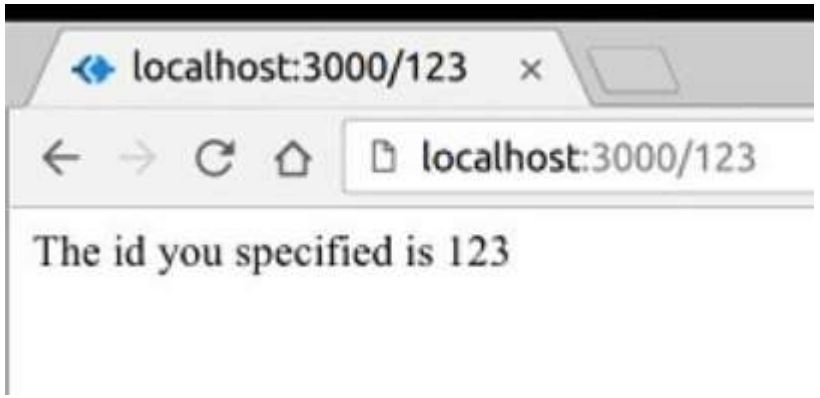
```
var http = require('http');
// Create a server object
http.createServer(function (req, res) {
// http header
res.writeHead(200, {'Content-Type': 'text/html'});
 var url = req.url;
 if(url ==='/about') {
        res.write(' Welcome to about us page');
res.end();
    }
else if(url ==='/contact') {
res.write(' Welcome to contact us page');
        res.end();   }
    else {
 res.write('Hello World!');
  res.end();  }
}).listen(3000, function() {
       // The server object listens on port 3000
    console.log("server start at port 3000");
});
```

Dynamic Routes

- Using dynamic routes allows us to pass parameters and process based on them.

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


app.get('/:id', function(req, res){
  res.send('The id you specified is ' +
req.params.id);
});
app.listen(3000);
```

http://localhost:3000/123.



Pattern Matched Routes

- You can also use **regex** to restrict URL parameter matching.

```
app.get('/things/:id([0-9]{5})', function(req, res){
  res.send('id: ' + req.params.id);
});
```

If none of your routes match the request, you'll get a ***"Cannot GET <your-request-route>"*** message as response. This can be replaced using

```
app.get('*', function(req, res){
  res.send('Sorry, this is an invalid URL.');
});
```

Middleware

- Middleware functions are functions that have access to the **request object (req)**, the **response object (res)**, and the next middleware function in the application's request-response cycle. These functions are used to modify **req** and **res** objects for tasks like parsing request bodies, adding response headers, etc.

Example of a middleware that gets called for every request

```
//Simple request time logger
app.use(function(req, res, next){
  console.log("A new request received at " + Date.now());

  //This function call is very important. It tells that more processing is
  //required for the current request and is in the next middleware
  function route handler.
  next();
});
```

To restrict it to a specific route (and all its subroutes), provide that route as the first argument of ***app.use()***.

```
//Middleware function to log request protocol
app.use('/things', function(req, res, next){
  console.log("A request for things received at " + Date.now());
  next();
});

// Route handler that sends the response
app.get('/things', function(req, res){
  res.send('Things');
});
```

Now whenever you request any subroute of '/things', only then it will log the time.

Order of Middleware Calls

- The order in which middleware calls are written/included in your file is very important; the order in which they are executed, given that the route matches also needs to be considered.

Example

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

//First middleware before response is sent
app.use(function(req, res, next){
  console.log("Start");
  next();
});
```

```
//Route handler
app.get('/', function(req, res, next){
  res.send("Middle");
  next();
});

app.use('/', function(req, res){
  console.log('End');
});

app.listen(3000);
```

Third Party Middleware

- http://expressjs.com/en/resources/middleware.html

ExpressJS - Templating

- A ***template engine*** enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.
- Pug is a very powerful templating engine which has a variety of features including **filters, includes, inheritance, interpolation**, etc
- To use Pug with Express, we need to install it,

- npm install --save pug

../

Set PUG as the templating engine for your app.

- Create a directory called views to place .pug files inside it
- Add the following to your server file

```
app.set('view engine', 'pug');
app.set('views','./views');
```

first_view.pug

```
doctype html
html
  head
    title = "Hello Pug"
  body
    p.greetings#people Hello World!
```

Important Features of Pug

To put text inside of a tag, we have 3 methods −

- **Space seperated**

```
h1 Welcome to Pug
```

- **Piped text**

```
div
   | To insert multiline text,
   | You can use the pipe operator.
```

- **Block of text**

```
div.
   But that gets tedious if you have a lot of text.
   You can use "." at the end of tag to denote block of te
   To put tags inside this block, simply enter tag in a ne
   indent it accordingly.
```

Attributes, classes, and ids

```
div.container.column.main#division(width = "100", height = "100")
```

This line of code, gets converted to the following.

```
<div class = "container column main" id = "division" width = "100" height = "100"></div>
```

Passing Values to Templates

- When we render a Pug template, we can actually pass it a value from our route handler, which we can then use in our template. Create a new route handler with the following.
- **See: dynamic.pug**

Conditionals and Looping

- We can use conditional statements and looping constructs as well.

```
html
  head
    title Simple template
  body
    if(user)
      h1 Hi, #{user.name}
    else
      a(href = "/sign_up") Sign Up
```

Looping

- Pug supports two primary methods of iteration: each and while.

```
ul
  each val in [1, 2, 3, 4, 5]
    li= val
```

```
ul
  each val, index in ['zero', 'one', 'two']
    li= index + ': ' + val
```

```
ul
  each val, key in {1: 'one', 2: 'two', 3: 'three'}
    li= key + ': ' + val
```

```
- var n = 0;
ul
  while n < 4
    li= n++
```

Multer (a node.js middleware for handling multipart/form-data)

- Multer will not process any form which is not multipart (multipart/form-data).
- Usage
  - Multer adds a body object and a file or files object to the request object. The body object contains the values of the text fields of the form, the file or files object contains the files uploaded via the form.

For single files uploads

```
const express = require('express')
const multer  = require('multer')
const upload = multer({ dest: 'uploads/' })
const app = express()
app.post('/profile', upload.single('avatar'), function (req, res, next) {
  // req.file is the `avatar` file,
// req.body will hold the text fields, if there were any
})
```

For multi-files uploads

```
.....
app.post('/photos/upload', upload.array('photos', 12), function (req, res, next) {
  // req.files is array of `photos` files
  // req.body will contain the text fields, if there were any
})
```

For form fields uploads

```
.....
const cpUpload = upload.fields([{ name: 'avatar', maxCount: 1 }, { name: 'gallery', maxCount: 8 }])
app.post('/cool-profile', cpUpload, function (req, res, next) {
  // req.files is an object (String -> Array) where fieldname is the key, and the value is array of files
//  req.files['avatar'][0] -> File
  //  req.files['gallery'] -> Array
  // req.body will contain the text fields, if there were any
})
```

For text-only multipart form

```
app.post('/profile', upload.none(), function (req, res, next) {
  // req.body contains the text fields
})
```

Form.html example

```
<form action="/stats" enctype="multipart/form-data" method="post">
 <div class="form-group">
  <input type="file" class="form-control-file" name="uploaded_file">
  <input type="text" class="form-control" placeholder="Number of speakers"
name="nspeakers">
  <input type="submit" value="Get me the stats!" class="btn btn-default">
 </div>
</form>
```

.js file

```
const multer  = require('multer')
const upload = multer({ dest: './public/data/uploads/' })
app.post('/stats', upload.single('uploaded_file'), function (req, res) {
  // req.file is the name of your file in the form above, here 'uploaded_file'
  // req.body will hold the text fields, if there were any
  console.log(req.file, req.body)
});
API
```

multer(opts)

- Multer accepts an options object, the most basic of which is the dest property, which tells Multer where to upload the files. In case you omit the options object, the files will be kept in memory and never written to disk.

Key	Description
`dest` or `storage`	Where to store the files
`fileFilter`	Function to control which files are accepted
`limits`	Limits of the uploaded data
`preservePath`	Keep the full path of files instead of just the base name

DiskStorage

- The disk storage engine gives you full control on storing files to disk.
- There are two options available, **destination and filename**. They are both functions that determine where the file should be stored.
- Destination is used to determine within which folder the uploaded files should be stored.
- Filename filename is used to determine what the file should be named inside the folder.
- Each function gets passed both the request (req) and some information about the file (file) to aid with the decision.

fileFilter

- Set this to a function to control which files should be uploaded and which should be skipped. The function should look like this:

```
function fileFilter (req, file, cb) {
  // The function should call `cb` with a Boolean  to indicate if the file should be
// accepted
  // To reject this file pass `false`, like so:
  cb(null, false)
  // To accept the file pass `true`, like so:
  cb(null, true)
  // You can always pass an error if something goes wrong:
  cb(new Error('I don\'t have a clue!'))
```

Form Data

- Forms are an integral part of the web. Almost every website we visit offers us forms that submit or fetch some information for us.
- To get started with forms, we need the body-parser(a Node.js **body parsing** middleware. for parsing JSON and url-encoded data) and multer(for parsing multipart/form data) middleware.

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

Formidable vs Multer

- Formidable is **a Node. js module for parsing form data, including multipart/form-data file upload**.
- **Multer** is a node.js middleware for handling multipart/form-data , which is primarily used for uploading files. It saves intermediate files either in memory or on hard disk and populates req.files object for consuming the files. You can have fine-grained control over which fields allow files and limit the number of uploaded files.

More on Dynamic Content & templates

- We will use templating engines to put dynamic content into our html pages.
- The idea is we have a template (looks like html)..which includes your html structure, markup, styles, and javascript files usually included..but with some plceholders..
- you will use some templating engine which understands certain syntax for which it scans your html looking templates and where it then replaces placeholders or certain snippets depending on the engine you’re using, with real html

EJS (Embedded JavaScript templating)

- EJS is a simple templating language that lets you generate HTML markup with plain JavaScript.
- You need to install it $ npm install ejs
- Ejs Supports conditions
- Uses normal html
- Same as pug you need to set the templating engine and specify the folder in which you are going to place your templates

```
app.set('view engine','ejs')
app.set('views','views');
```

How to use

- Specify your template engine

```
App.set('view engine','ejs')
App.set('views','views');
```

- Now your ejs file is simple html with placeholders

```
<%= 5 + 5 %>
//this outputs 10
```

Example

```
//.ejs file
<h1> Example variable name: <%= exampleVar %> </h1>
//"exampleVar" gets outputted as the value "JavaScript"
```

```
//server.js file
app.get("/", function(req, res){
res.render("index.ejs", {exampleVar: Javascript});
});
//the "exampleVar" being the name of our variable from the EJS file
//the "JavaScript" the value assigned to the variable
```

EJS If Statements:

- Besides adding just variable we could also do other things like control flows (if statements, loops, etc)

```
//.ejs file


<% if( exampleVar === "JavaScript") { %>
    <p> Good Choice </p>
<% } %>
//this will add only the contents of the paragraph to our HTML, if the condition is true
```

ejs loops

```
//.ejs file


<% for(var i = 1; i <= 10; i++) { %>
  <%= i %>  <%# this will output the numbers from 1 - 10 %>
<%  } %>
Note: comments in EJS are written between the <%# %> brackets
```

Creating the EJS Partials

- The idea is to have some blocks that you can reuse in your templates
- Like a lot of the applications you build, there will be a lot of code that is reused. These are considered *partials*. In this example, there will be three partials that will be reused on the Index page and About page..
- Create a new subfolder inside the views and inside it create shared files or shared code blocks (code you can share across your views. Example: the header of your document..some of the body…the nav bar..

Example of partial files

views/partials/head.ejs

```
<meta charset="UTF-8">
<title>EJS Is Fun</title>

<!-- CSS (load bootstrap from a CDN) -->
<link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/twitter-bootstrap/4.5.
<style>
  body { padding-top:50px; }
</style>
```

views/partials/header.ejs

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a class="navbar-brand" href="/">EJS Is Fun</a>
  <ul class="navbar-nav mr-auto">
    <li class="nav-item">
      <a class="nav-link" href="/">Home</a>
    </li>
    <li class="nav-item">
                          ink" href="/about">About</a>
```

views/partials/footer.ejs

```
<p class="text-center text-muted">&copy; Copyright 2020 The Awesome People</p>
```

Adding the EJS Partials to Views

- After you defined the partials. Now you can include them in your views.
- Use <%- include('RELATIVE/PATH/TO/FILE') %> to embed an EJS partial in another file.
- The hyphen <%- instead of just <% to tell EJS to render raw HTML.
- The path to the partial is relative to the current file.

Example

views/pages/index.ejs

```
<!DOCTYPE html>
<html lang="en">
<head>
  <%- include('../partials/head'); %>
</head>
<body class="container">

<header>
  <%- include('../partials/header'); %>
</header>

<main>
  <div class="jumbotron">
    <h1>This is great</h1>
    <p>Welcome to templating using EJS</p>
  </div>
</main>

<footer>
  <%- include('../partials/footer'); %>
</footer>

</body>
</html>
```