**Express**

**Review:**

* + **Express is a framework (library of functions which are designed to work tgether) for building the basic web applications . It makes our life much easier.  
    In particular, it handles requests which come from different pages (routes) on your site and it allows you to interface with a templating engine (called the view) so that you don't have to re-code each response from scratch. Express supports pug (used to be called jade), handlebars, ejs and hogan.js Express also makes it easy to handle the query parameters, whether from a get or a post.  
    Like Node, Express makes heavy use of callbacks, and like Node, the callback often follows the function(err, data){} approach, which you have already seen.  
    Finally, Express is small and fast. It is *unopinionated* which means it doesn't make you do things its way.**
  + Express is maintained by the Nodejs Foundation --- so it will be well maintained and stable. It is, no surprise, open source.
  + The name of the module is **express** and you could install it globally b/c once you start to use it, you will use it all the time.  
    **npm install express**but most people install it locally each time. (If you have already used npm init then you need to use the –save option so that package.json will be updated.)
  + In the folder you wish to use type **npm init**  and then **npm install express**

Here is the world's simplest express app ---- save the code in index.js and then run it with node:  
//Simplest possible app with Express

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

app.get('/', function(req,res) {  
 res.send("Hello World");|  
});

app.listen(3000);

On your browser go to localhost:3000  
  
The 4th line of code here is used to handle a get request, and is of the form  
 **app. get('**someRoute**',** callback);  
Our routes our *relative to the URL for the route*, and they begin as /   
The simplest route is just /  
The callback is the usual function with parameters req and res.  
  
You can figure out what

**app. post('**someRoute**',** callback);

will do.  
   
You can see that Express sent back to your browser the Hello World message.  
Notice that Express has taken care of the header (the writeHead() in the http module) for us.

* + We can also use Express to access the values sent to us in a request, either from a get or a post. *For the get the name-value pairs are in the query (string) and for a post the name-value pairs are in the body.* Accordingly:  
      
    If the request was a **get** from /someRoute?name1=value1&name2=value2  
    then we would code:  
     app. **get**('/someRoute', (req, res) =>{  
     let ourValue1 = req.**query**.name1;   
     let ourValue2 = req.**query**.name2;  
     //do stuff with them  
     });  
      
    If the request was a **post** from /someRoute with name1=value1&name2=value2  
    posted, then we would code:  
     app. **post**('/someRoute', (req, res) =>{  
     let ourValue1 = req.**body**.name1;   
     let ourValue2 = req.**body**..name2;  
     //do stuff with them  
     });  
       
    An aside, whether you use post or get, if you also query on name1, so that your route is  
    '/someRoute:name1' then you can set ourValue1 to be req.params.name1 See [https://www.geeksforgeeks.org/express-js-req-params-property/](about:blank) or [https://www.geeksforgeeks.org/express-js-req-params-property/](about:blank) for examples.  
      
    **Material from the HW:**
  + Now let's do something a little more complex.  
    Often Express does this with **middleware**. Middleware is a module which has access to the request and response objects.  
    We let a particular piece of middleware do this with the **app.use()** method.  
    We will see two very common uses of middleware – first a module to manage all our routes, and then (later on) a template engine.   
    Middleware is also commonly used to sanitize data, make authentication checks, log in transactions, etc. You can think of middleware as building blocks. You take the pieces you need, in the order you want. Each piece of middleware will end with next(), which calls the next piece of middleware.  
    And, typically the middleware does its processing before the app.post() or app. get handles a particular route.
  + Typical code which *uses* middlesware will look like:  
      
     **app.use(***someRoute,* ***function (req, res,next){*** *//some code* ***next()*** //this passes control to the next piece of middleware. ***});***or  
     **app.use(***myMW); //Here myMW is the name of a piece of middleware.*When you *write* your own middleware function it will typically look like  
      
     **var myMW = function(req, res, next)( ….};**There are a number of clear examples at https://expressjs.com/en/guide/writing-middleware.html#writing-middleware-for-use-in-express-apps   
    In your Express script you may define functions  
     function F1(req, res, next) {……} //these function can modify the properties   
     function F2(req, res, next) {……} //of req and res!function F3(req, res, next) {……}  
      
    The last line in each of F1 F2, F3 will be next().  
      
    Now I can handle a route, say to the about route with a post as:  
      
     app.post('/about', F3, F1, F2, {req, res} => { });  
      
    Then the functions F3, F1, FF2 will be called in the order I specified and then finally the callback function (res, req) => { }.  
      
    Obviously, we can re-use the F1 etc. fundtion in handling other routes, we can store them in a different module (either one we created or a third party module) which we require etc. You'll see an example of having the functions in another module when we talk about routes and Router.

**Material you are about to see in the Brad Traversy video, starting at minute 18:**Express makes it easy to serve **static** files.  
It is customary to put all such files in a **public** directory (which is in the same folder as your index.js file)  
In that case you would tell Express about this my saying

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

Here, the 'public' is the relative path from your current directory. If you need to you can write a diferent path or construct an absolute path (using the ***path*** module) as ( see [https://expressjs.com/en/starter/static-files.html](about:blank) for details)

**app.use('/static', express.static(path.join(\_\_dirname, 'public')))**

It is also possible to serve static files from more than one directory --- for example if I had directories public and pix then I would say:  
 **app.use(express-static('public'));**

**app.use(express-static('pix'));**

* How does Express know which static page to serve?   
  If your route is '/' then Express willl look for index.html (The default is always index.html, after all.)  
  If your route is '/contactUs.html' (that is the url is localhost:3000/contactUs.html) then Express will look for contactUs.html etc  
  NOTE: if the browser has the addresss *hocalhost:3000/about* then express will expect to handle the /about route, but if the browser has localhost:3000/about.html then express will expect to serve the about.html paqe
* To see some examples, look at the projects in the withExpress folder (of the simpleExamples folder) at [http://web.simmons.edu/~menzin/CS321/Unit\_8\_Server\_Side\_Node/SimpleProjects/withExpress/](about:blank)
* BTW even the firstExpressApp has over a hundred files --- look in the folder and then look in the node\_modules folder that has all the dependencies. (and you can see that body-parser is already there.)  
  Let's look at the firstExpressApp and the serveStatic app.
* The firstExpressApp has code:

//Simplest possible app with Express

var express = require('express');

var app = express();

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

res.send("Hello World");

});

app.listen(3000);

On your own machine, create a folder firstExpressApp and put the above code in   
 it in a file index.js  
 From that folder, in your terminal, type  
 npm init  
 npm install express -save  
 node index

Now, in your browser type in the url for localhost:3000 and hit enter.

* The serveStatic app has code:

//Simple app serving static pages with Express

const express = require('express');

const app = express();

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

app.listen(3000);

Repeat the steps to set up a project, making sure to also create the public directory and the index.html, about.html and contactUs.html pages in it.  
The about.html page has the code

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>About page</title>

</head>

<body>

<h1>This is my About page</h1>

It is saved as about.html in my public directory

</body>

</html>

And the other pages are the same, except with the obvious changes in the title,   
<h1> and following lines.

Test the program at localhost:3000 and localhost:3000/about.html etc.

* + One of Express' strengths is its ability to handle many routes. (Notice that we don't need a bunch of nested if statements the way we did with basic Node.)  
    In fact express.Router is designed for exactly that purpose.  
    One example is at [https://www.tutorialspoint.com/expressjs/expressjs\_routing.htm](about:blank)

In Express you can actually match the route using regex and (Your code would be app. get(/*someRegex, ….) – that is no quotes around the route.)*

Finally you can pass variables. These were retreived with req.params (see [https://www.tutorialspoint.com/expressjs/expressjs\_url\_building.htm](about:blank) )   
Please note that the params values are *strings, so if you*  want a number you will need to run it thru parseInt or parseFloat.)   
There are changes coming in version 5, currently in alpha, but expect more changes given those in Node itself ( changes which made url.parse() and querystring legacy.)  
  
If you have a bunch of routes you can put all the handling in a separate module.  
Our module might start off:  
 const express = require('express');  
 const router = express.Router()  
Obviously, Router is a method of express which is used for handling multiple routes.  
Instead of api.get(someRoute, (req, res)=> { *code* })  
we write   
 router.get(someRoute, (req, res)=>) {code})

And notice that we are constructing the router object --- and our last line will be

module.exports = router;  
You can find a detailed explanation of the syntax at [https://expressjs.com/en/5x/api.html#router](about:blank#router)

* + Another useful piece of middleware is   
    ***app.use(express.urlendcoded( {extended:false} ) );*** //do this up near the top  
    This lets us access the parameters of a post easily as **req.body.**keyName.
  + Express at some point included ***body-parser*** , which as of Expess 4 is named **bodyParser** (you may need to ***require*** it – it's not clear; some newer articles say it is part of express – or you don't need to *require* it, others say do need to require and *use* it.) There is also ***multer*** which is similar to bodyParser, but for forms which have multiple answers. At minute 44 of [https://www.youtube.com/watch?v=L72fhGm1tfE](about:blank) it looks like we just need to use express.json() and express.urlencoded({extended:false}) . It appears to me that bodyParser has been separate middleware, then a separate (had to be require'd) module, then back to Express. Crazy! We'll have to see what happens with the current version. In any case, discussion of this is deferred until we come to middleware.
  + The response object has methods res.send () which returns text, res.json() which returns json, res.redirect() which returns to a different url,res.status() which returns a status code, etc.
  + Summary: Key steps for using Express to handle many routes:

1.. Make a folder (same level as public) named **routes**

**2.**For each subsite or group of routes you want to handle together, make a file

**stuffRoutes.js     *this would be for routes beginning /stuff***

*3*.  In that folder put:

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

**const router = express.Router();     //this returns an object in express**

**//require any files with data that you will use in your handlers**

**//For each route that started '/stuff'  you write your handlers**

   route.get('/',  (req, res) => { *whatever*});  
      [route.post](http://route.post/)('/', (req, res) => {*whatever*})'  
  
    //For routes with params you can say   
 // route.get('/:id',   (req, res) => {*whatever*});

    //For routes which started '/stuff/more'  you can say  
 // route.get('/more',  (req, res) => {*whatever*})');

**module.exports = router**

4. Now we need to instruct our index.js (main entry point) to use this module.

    So in index.js we add the lines:  
    **stuffRoutes = require('./routes/stuffRoutes');**

* + **app.use('/stuff', stuffRoutes);**
* **Template Enignes**
  + **A template engine** is a program ( or middleware) which allows you to write html pages as templates --- that is there are some variables which will need to have their values filled in , and then you can produce an html page.
  + The Ethan Brown book [https://learning.oreilly.com/library/view/web-development-with/9781492053507/ch07.html#ch\_templating](about:blank#ch_templating) has a nice chapter comparing template engines with a lot of detail on handlebars (second half of the chapter.)
  + In Express a template engine is referred to as a **view engine**. Of course, you must require the appropriate package. Then you tell Node what engine you will use by setting the 'view engine' property on your app.   
    For example, handlebars has the name hbs and your code would read:  
     const express = require('expess');  
     const hbars = require('express-handlebars');  
     const app = express();  
     **app.engine('handlebars', hbars())** //Not sure if need this line  
     **app.set('view engine', 'hbs');**
  + After a template has been created, then with then an object needs to be created which holds the values of the variables which will be substituted. (These could be the values which were posted from the previous client page – nothing says you must use all of them) and there is an optional callback.   
    Views are stored in specific folders (handlebars has a folder **views** which has subfolders **layouts** and **partials.** The layouts folder is for whole pages and the partials folder is for sections like headers, nav etc.)  
    Express will **render** the new page it is sending off to the client with   
    **res.render(**view [,locals][,callback]) where locals is the object with the values we will be substituting in the template.
  + For Express there are 3 common, simple ones and which you use is a matter of personal preference:
    - pug (an outgrowth of Jade)
      * Uses indentation (the way python does)
      * Abstracts away much of the html --- a little hard to read for some
    - handlbars
      * It may be used both client side and server side. (In fact the older tutorials emphasize client side use --- and remember they were often written before the introduction of template literals in ES6.
      * To use handlebars with express we need the following code:  
        **exphbs = require('express-handlebars');** //obviously installed  
        **app.engine('handlebars', exphbs({defaultLayout:'main'}))):**//main will be the default layout **app.set('view engine', 'handbars');** //registers handlebars as the template engine
      * There is a mandatory folder structure which is desccribed, along with other documentation, at [https://github.com/ericf/express-handlebars](about:blank) In the same folder as your app.js or index.js there is another folder *views* which has a subfolder *layouts*, and in that folder we put our default layout, which we have just named main.handlbars . (The { } inside the middle line of code is optional.)

The handlebars documentation suggests a very basic template for   
 the default:  
  
 <!DOCTYPE html>  
 <html>

<head>

<meta charset="utf-8">

<title>Example App</title>

</head>

<body>

{{{body}}}

</body>

</html>

Notice the triple braces {{{ body }}} which is where our body will   
 go.  
 Then in the *views* folder we create index.handlebars for our main   
 page, and there we fill in the content of our body for the home   
 page (what will turn into index.html)  
 n our index.js (main js file) we can say"  
 app.get('/', (req, res) => res.render('index')  
 The response object will use its render method to look in views for  
 index.handlebars, apply the default template, and produce and   
 send index.html

* + - * Express will start at the top and look for the first match --- so for  
         example, the static page might be at the bottom.
      * You can add an object to res.render as a second parameter, i.e.  
        res.render('index', {var1:value1, var2:value2,..}) and then wherever the index page has {{var1}} (notice 2 braces) that will be replaced by value1.  
        Obviously you need some code to read key-value pairs from the request and then construct the {var1:value1,….} obect.
      * Handlebars also has some basic logic funcstions   
        {{#if someCondition }} {{else}} and you can loop thru an array : *{{#each someArray}}* stuff to do on element *this {{someArray}}*
      * You can write a page that has a form with action = someRoute   
        so that the code at someRoute will be handled as you wish.
    - ejs
      * I think this is an outgrowth of ember.js and it seems to be less popular than some of the others these days.

**Miscellaneous useful information re Express**

**Some modules you may come across:**

**morgan** is a module which logs on the console all the requests to the server.  
**Passport** handles authentication, incl. thru FB, Twitter etc.  
**multer** handles forms with multiple answers  
**body-parser** is now built into Express (as of v 4.16) but not in older versions).   
That is, bodyParser is a dependency of Express. (automatically installed with express,  
but you may still need to require() it. It is used to parse json, url-encoded and raw  
data, as well as text.  
**express-generator** will set up all the folders for an application for you – see <https://expressjs.com/en/starter/generator.html> for more information **session** - for session variable; cookies for cookies etc.  
**lodash** is an older library of many useful JS functions. It’s less in use now that ES6/ESNext  
has included a lot of its functionality.  
The exception is a deep copy (also called a structered clone) but that may come soon to ESNext

The usual advice for a deep copy is   
 let newObject = JSON.parse(JSON.stringify(oldObject))   
which usually works, but will fail under certain circumstances.  
You will recognize lodash functions b/c Lodash and its predecessor Underscore  
use \_ as the abbreviation for the library name (the same way jQuery uses $.)  
Lodash is fairly large, so if you need something from it you may wish to import only  
certain functions. That said, React uses Lodash.

**Remember the changes coming in v5 of Express.**Recall from the v5.x Guide that res.send(status) is replaced by res.sendStatus(statusCode);

**The Router() object**It has a param method which is useful for validating user input. If we have set  
 const router = express.Router();   
then we can use  
 router.param(some\_id, callback)   
to fire the callback based on being sent to a route with a router parameter --- e.g.  
 router.param(‘user\_name’, (req, res, next, id) =&gt; {  
 //validation code and error handling;  
 next();  
 });  
where user\_name is the parameter from ‘/login/:user\_name   
See <https://expressjs.com/en/5x/api.html#router.param> for an example.