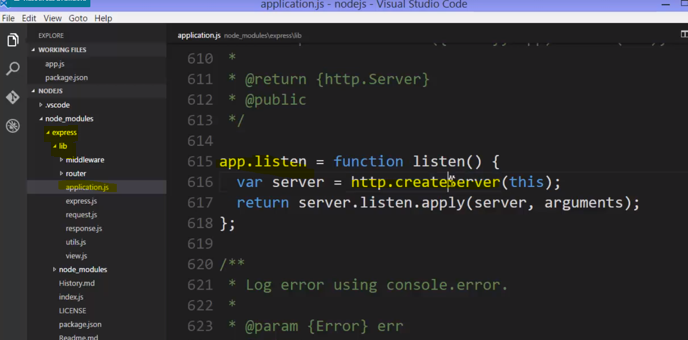
**Express is a minimal and flexible Node.js web application framework**

**Express isn't adding anything to the C++ side of the Node core. Everything Express is doing are things that you could have coded yourself. It's just a lot of JavaScript code taking care of a lot of very common things in a really great way, which is why it is very popular.**

1) What is the purpose of it with Node.js?

That you don't have to repeat same code over and over again. Node.js is a low-level [I/O](http://en.wikipedia.org/wiki/Input/output) mechanism which has an HTTP module. If you just use an HTTP module, a lot of work like parsing the payload, cookies, storing sessions (in memory or in [Redis](http://en.wikipedia.org/wiki/Redis_%28data_store%29)), selecting the right route pattern based on [regular expressions](http://en.wikipedia.org/wiki/Regular_expression) will **have** to be re-implemented. With Express.js it there for you to use.

For example - 

**inside Express, it adds a listen function. And see what it does? It just does what we did before. It calls the standard core Node HTTP object and the createServer method**.



**what about responding to the URL, the way we did with those if statements, originally?**

Well, Express comes with a much, much better way to deal with that. We'll say app, that is, that function that came back from invoking Express, that came back from actually requiring it, and it has on it various methods that map to HTTP methods, or verbs.



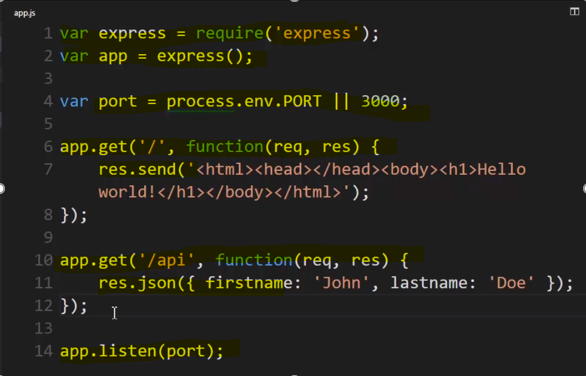
**Now, they don't actually do anything in themselves, because it's just some text, a piece of data, that is coming along for the ride as part of the HTTP request. However, it's up to the server, then, to do what is expected to be done with one of those HTTP methods.**

**Express built in a way to then map what you see is the verb in the HTTP request to a particular function that you want to run when that request comes in.**

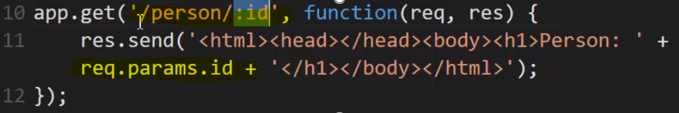
**An HTTP request comes in. This is the URL. And it's a GET method. It's a GET verb, in the HTTP request.Then this function will fire. And it will give me a request and a response. Just like we did before, but this will be an Express request and response, which wraps up the standard HTTP Node request and response, and provides a bit more functionality to make things easier for us.**

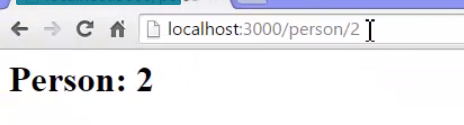
**Notice that I'm not putting a content type of text slash HTML or something like that.**

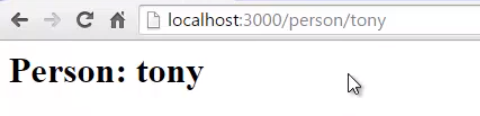
**The Express methods and the response object that wraps up the standard response object actually looks at what I'm sending and takes care of that for me to the best it can.**



//////

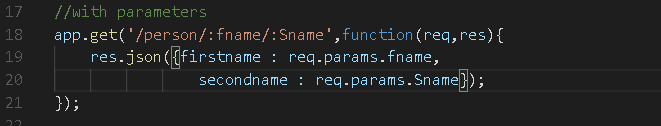






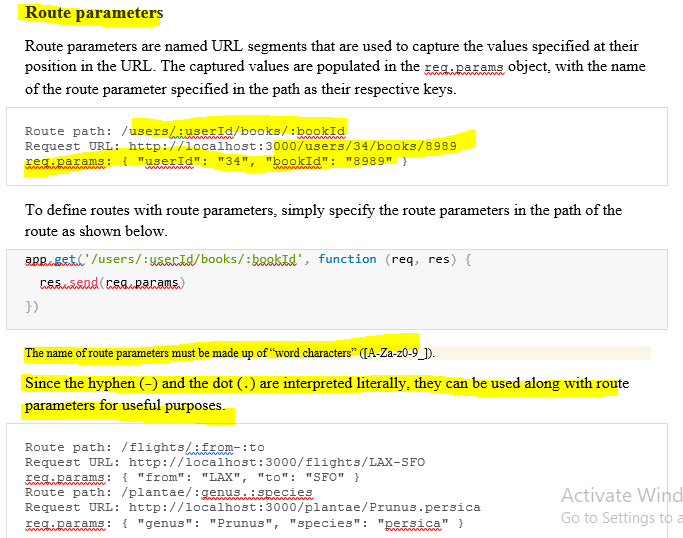
**you could have multiple parameters in url.**

**You could have person/:page/:id and these would be two different variables.**



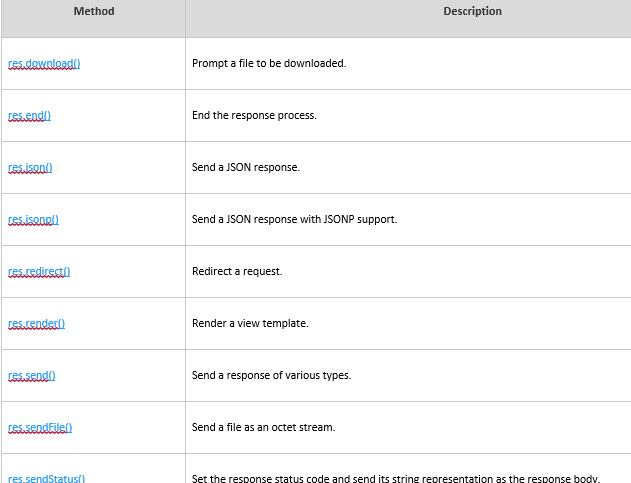


It does pattern matching and it does a variety of pattern matching and let's me pull out specific values in the pattern. And so i have full flexibility on dealing with my routes and I don't have to do too much work, in order to make this happen.



# Response methods

The methods on the response object (res) in the following table can send a response to the client, and terminate the request-response cycle. If none of these methods are called from a route handler, the client request will be left hanging.

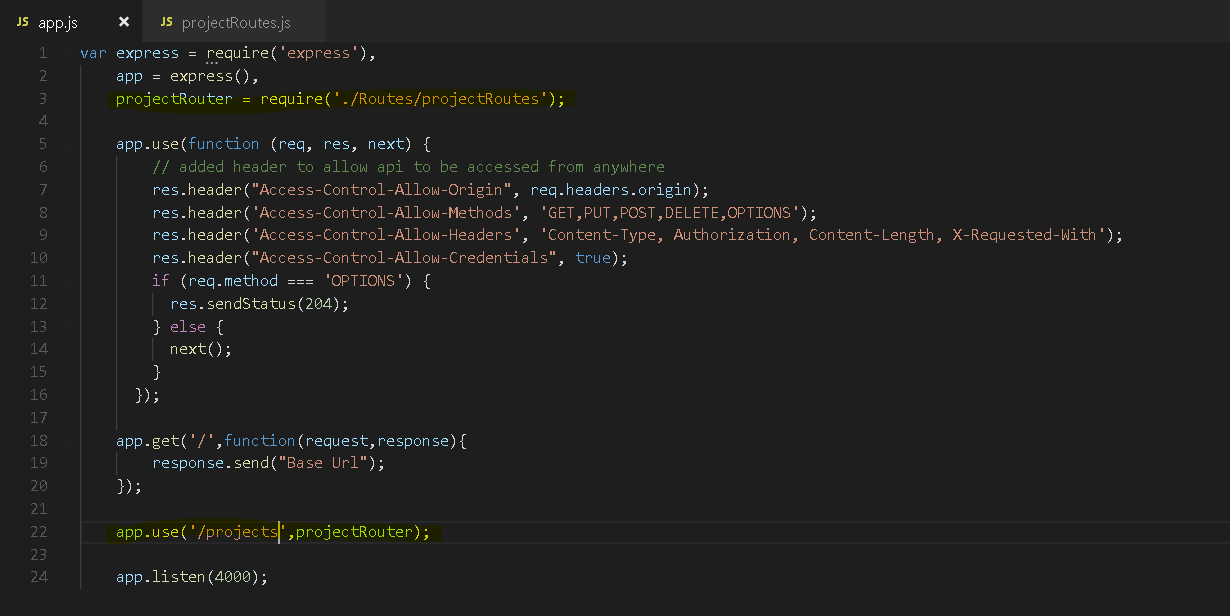


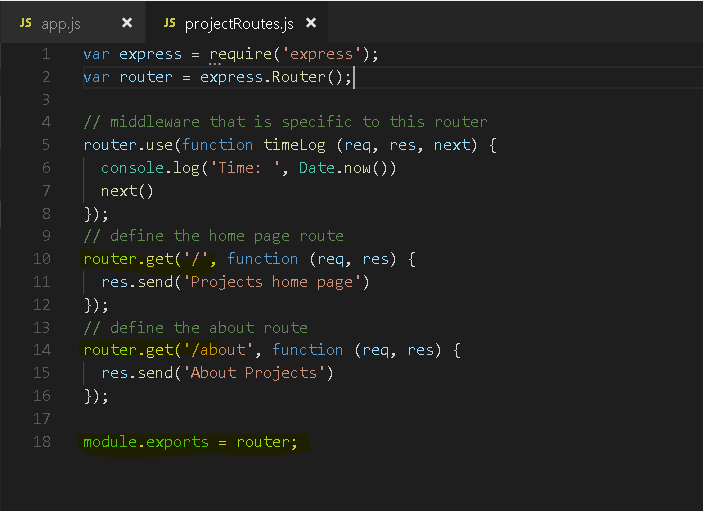
# app.route()

# **express.Router**

**Use the express.Router() class to create modular, mountable route handlers. A Router instance is a complete middleware and routing system; for this reason, it is often referred to as a “mini-app”.**

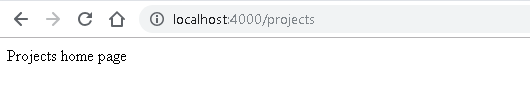
**Basic example of express.Router() –**

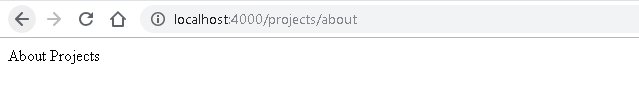




**Output –**







# **middleware**

***Middleware* functions are functions that have access to the**[**request object**](https://expressjs.com/en/4x/api.html#req)**(req), the**[**response object**](https://expressjs.com/en/4x/api.html#res)**(res), and the nextfunction in the application’s request-response cycle. The next function is a function in the Express router which, when invoked, executes the middleware succeeding the current middleware.**

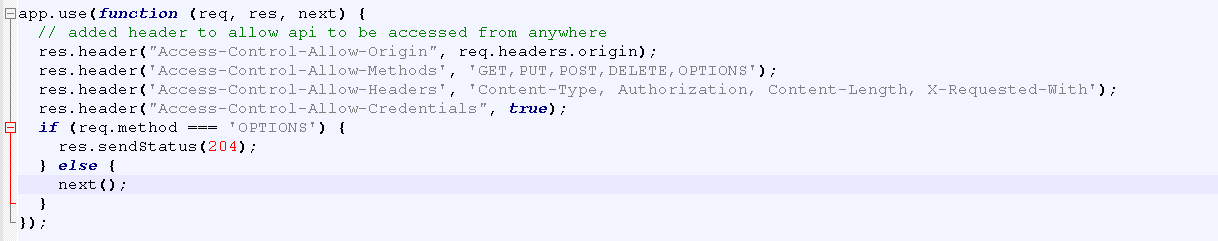
**Middleware functions can perform the following tasks:**

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

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

Example of using a **developer written middleware** .This middleware function add different header to response.

This middleware will execute **Every time the app receives a request.**



To load the middleware function, call **app.use(),** specifying the middleware function. For example, the **following code loads the myLogger middleware function before the route to the root path (/).**



**Every time the app receives a request, it prints the message “LOGGED” to the terminal.**

**The order of middleware loading is important: middleware functions that are loaded first are also executed first.**

**If myLogger is loaded after the route to the root path, the request never reaches it and the app doesn’t print “LOGGED”, because the route handler of the root path terminates the request-response cycle.**

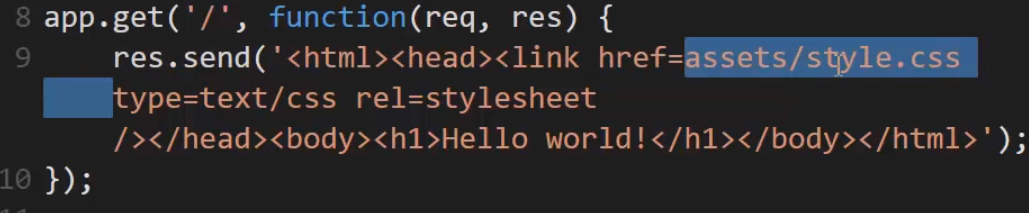
**The middleware function myLogger simply prints a message, then passes on the request to the next middleware function in the stack by calling the next() function.**

A very common scenario might be that I need to handle files being downloaded.

So for example if I have a CSS file, a stylesheet file, that defines what my application looks like on the Internet in the browser, then the browser is going to request it. It's going to make an HTTP request to get that CSS file, and I don't want to manually deal with that. I just want whatever files, CSS files, image files, etc., that are just sitting on the server, if they're requested in that URL then they should just be hooked to and streamed automatically to the response.

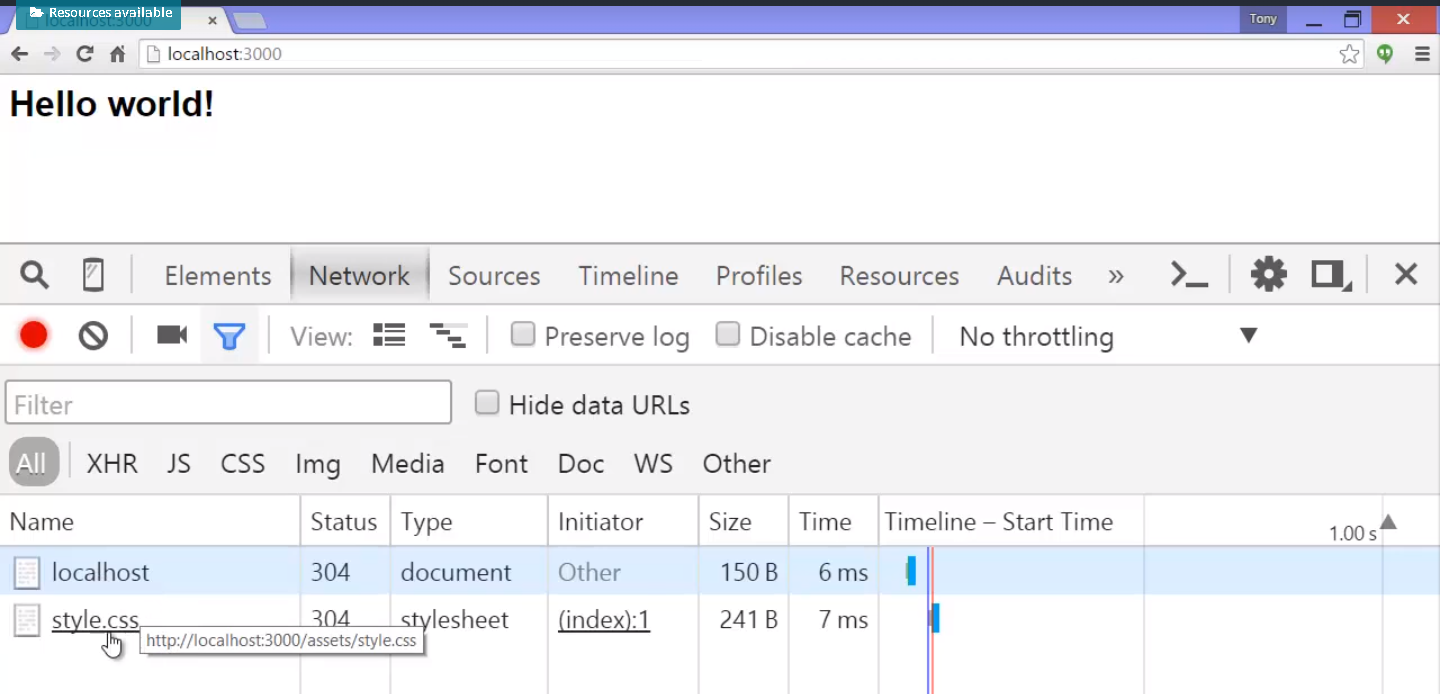
**"Well, every time we see /assets "and then we see slash some file," what I'll actually do is go find this full file path, public, that's where I decided to put them, and then look for that same file and stream the response back, and that's all I have to do. Any static file then I'd put inside the public folder would become available.**



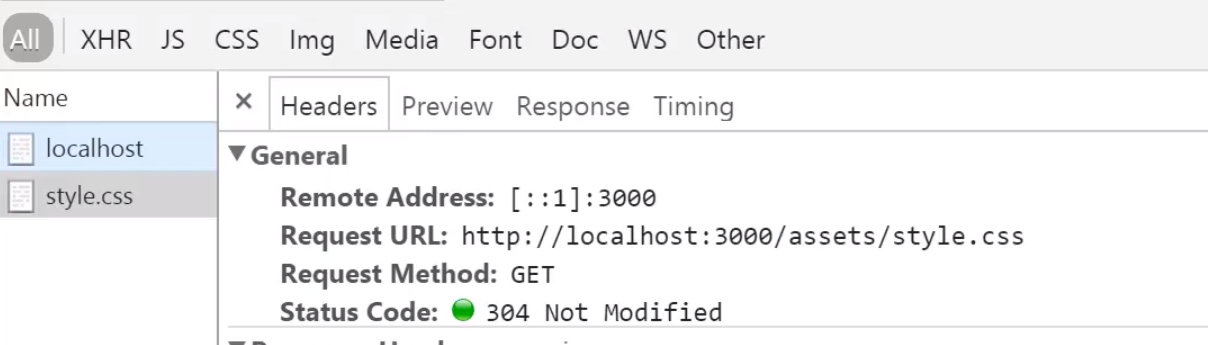


**So, the browser is going to download this HTML, see this link tag, and generate an HTTP request for my site / assets/style.css. That HTTP request will be handled by Node, and when it sees the /assets part it will say, "Okay, anything after that "I will look for it in this folder."**

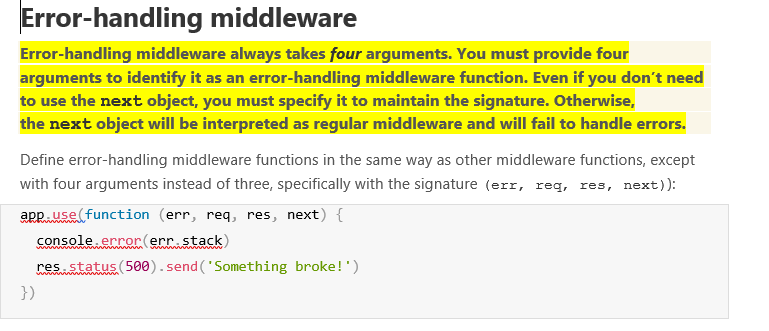
**This is middleware between the request and the response, and it will respond by streaming back this file for me.**

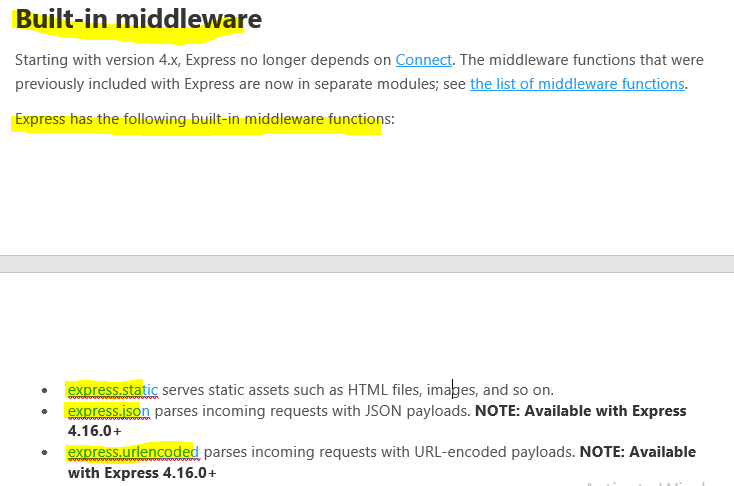


**There was my request for the HTML, and then the browser saw the link tag and generated that second request for the style.css.**









**Third-party middleware**

Use third-party middleware to add functionality to Express apps.

Install the Node.js module for the required functionality, then load it in your app at the application level or at the router level.

The following example illustrates installing and loading the cookie-parsing middleware function cookie-parser.



Using it with Express I use .use, and not specifying a route, so it would use it always,

and so this middleware will take the request, the HTTP request, which incidentally includes cookies as a string separated by semicolons.

cookie - **a little piece of information stored on your browser. Well, that gets sent along for the ride with every HTTP request to a particular site, a cookie for that site.**

**So, this pulls that out of the HTTP request, that string, breaks it apart, and then adds it as a .cookies property to the request, and then calls next , so that when you go to get the route you can just grab the .cookies property and get all the values in the cookie.**

# **Templates and Template Engine**

Express comes with the ability to plug in a whole variety of template engines.

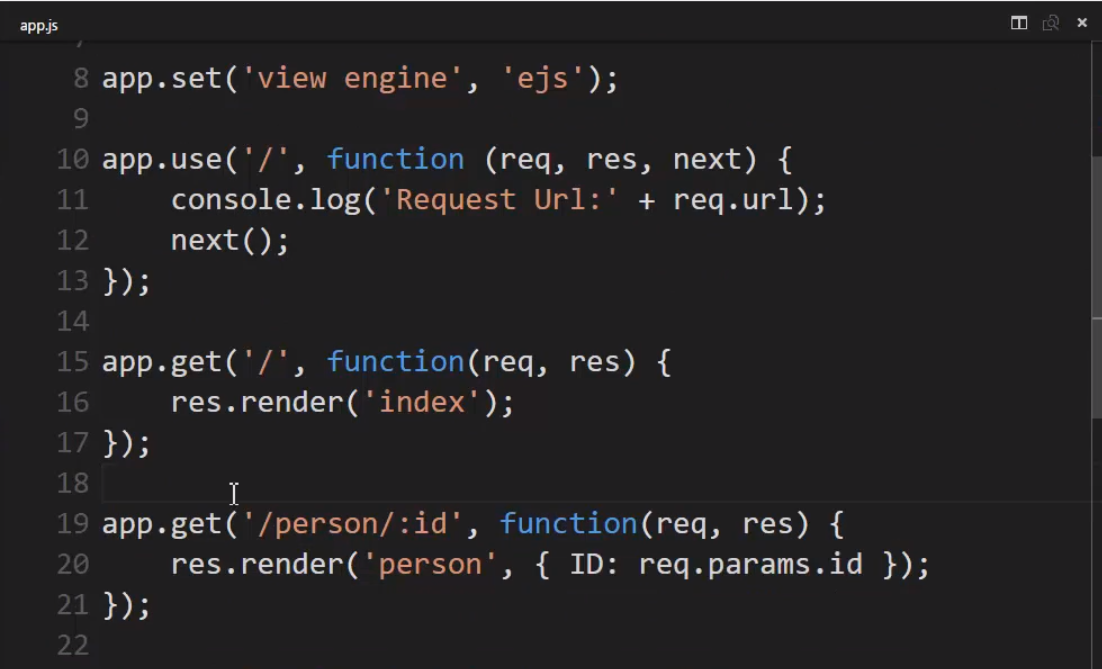
A **template engine** enables you to use static template files in your application. At runtime, the template engine replaces variables in a template file with actual values, and transforms the template into an HTML file sent to the client. This approach makes it easier to design an HTML page.

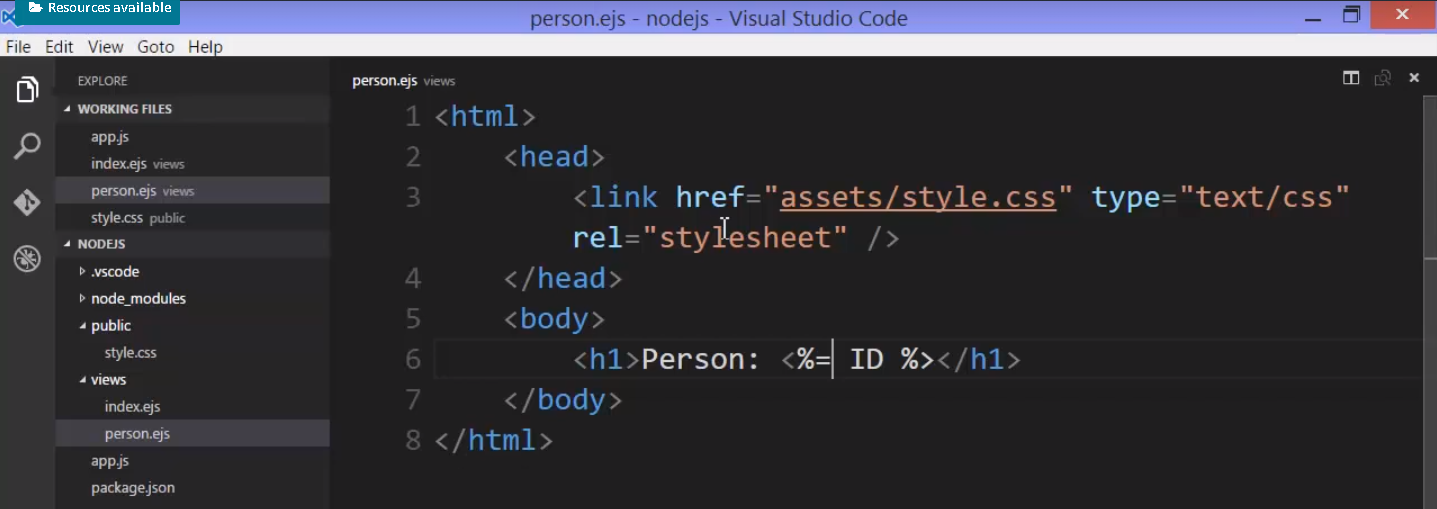
Some popular template engines that work with Express are [Pug](https://pugjs.org/api/getting-started.html), [Mustache](https://www.npmjs.com/package/mustache), and [EJS](https://www.npmjs.com/package/ejs). The [Express application generator](https://expressjs.com/en/starter/generator.html) uses [Jade](https://www.npmjs.com/package/jade)as its default, but it also supports several others.

**Note**: Jade has been renamed to [Pug](https://www.npmjs.com/package/pug). You can continue to use Jade in your app, and it will work just fine. However if you want the latest updates to the template engine, you must replace Jade with Pug in your app.

**To render template files, set the following**[**application setting properties**](https://expressjs.com/en/4x/api.html#app.set)**, set in app.js in the default app created by the generator:**

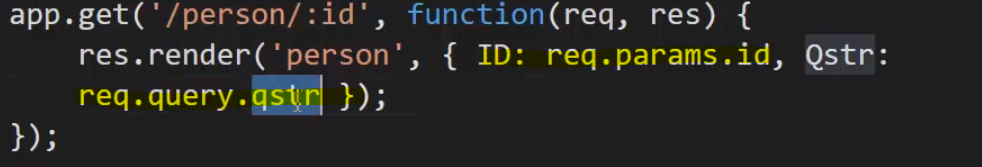
* **views, the directory where the template files are located. Eg: app.set('views', './views'). This defaults to the viewsdirectory in the application root directory.**
* **view engine, the template engine to use. For example, to use the Pug template engine: app.set('view engine', 'pug').**
* Then install the corresponding template engine npm package; for example to install Pug:
* $ npm install pug --save
* After the view engine is set, you don’t have to specify the engine or load the template engine module in your app; Express loads the module internally, as shown below (for the above example).
* app.set('view engine', 'pug')
* Create a Pug template file named index.pug in the views directory, with the following content:
* html
* head
* title= title
* body
* h1= message
* Then create a route to render the index.pug file. If the view engine property is not set, you must specify the extension of the view file. Otherwise, you can omit it.
* app.get('/', function (req, res) {
* res.render('index', { title: 'Hey', message: 'Hello there!' })
* })
* When you make a request to the home page, the index.pug file will be rendered as HTML

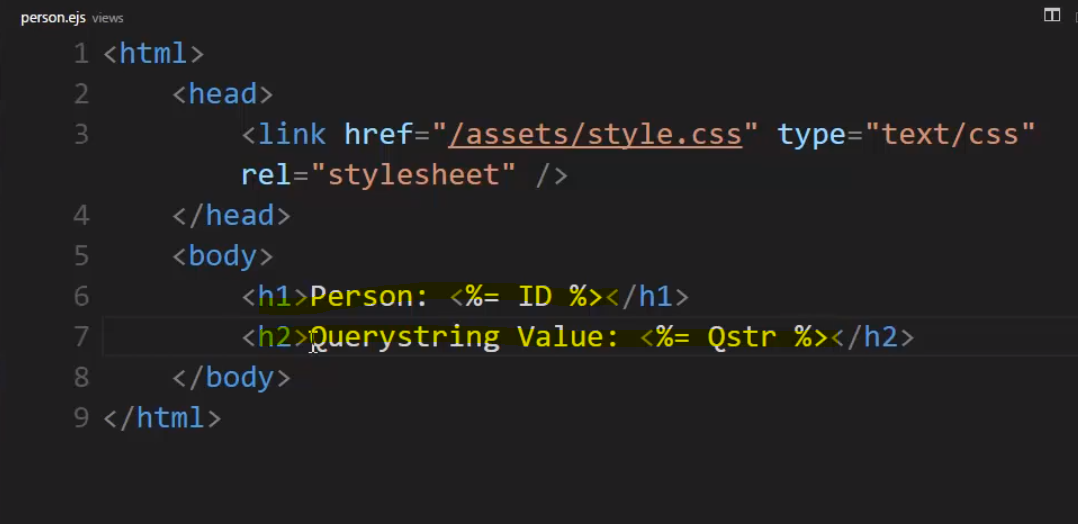
Example with ejs Template engine - 



# **QueryString/Path parameter and Post parameters**

**Get Query String(Parameter) and Path parameter from request**







**Getting POST Parameters**

However if we want to deal with a form being posted or with JSON data being posted,

we'll need to be able to parse the body of the HTTP request and that's not something that Express is doing just out of the box.

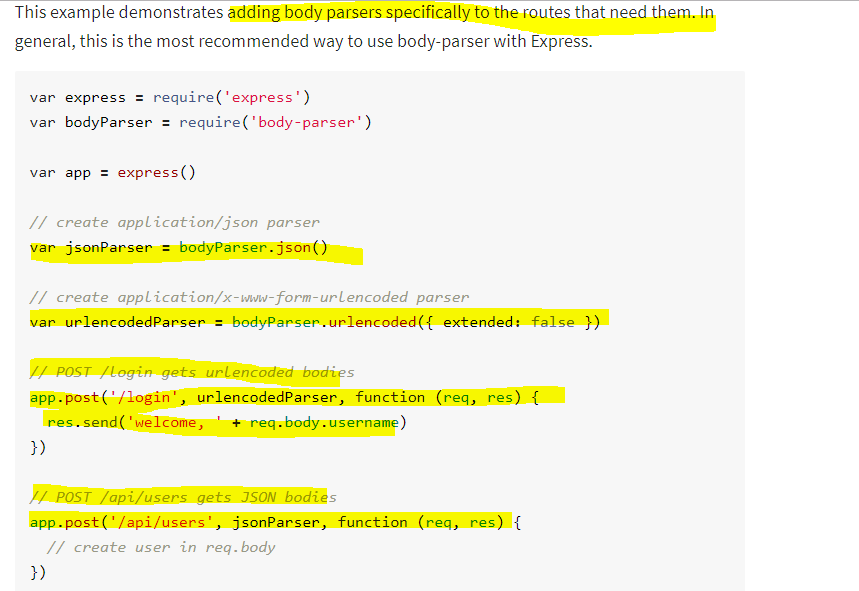
**That means we'll need to go back to our middleware. We need something between the request and the response. Something that will look at the request, give us access to that information, parse that content of that HTTP request for us so we can work with it.**

We could do it ourselves, but there's already stuff available. So we'll look at **body parser**.

Node.js body parsing middleware.

**Parse incoming request bodies in a middleware before your handlers, available under the req.body property.**

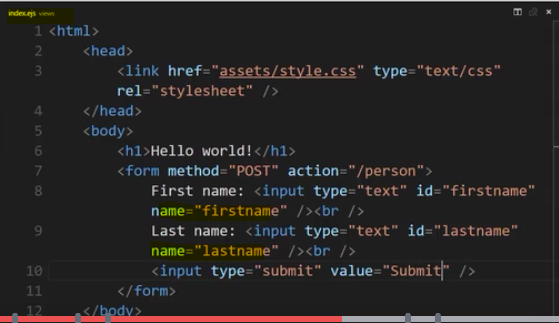
There are actually more than just one of these body parsing middlewares, but the idea is that it will parse the body.



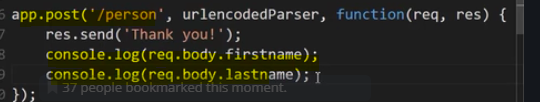
**we have to pass it as a callback to the POST statement. You see, these methods take other callbacks.**

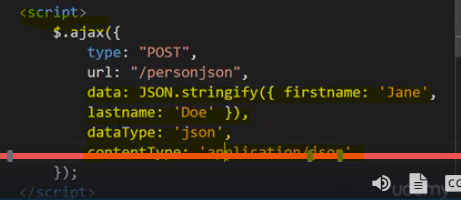
**The good thing about doing it this way as opposed to just .use , is that it makes certain that the HTTP request is formatted properly for what's expected from the parser.**

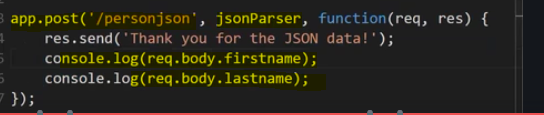
**similar to when we said app.use.**



So what the browser does is when we submit this form, the browser takes the values I've typed into these fields, uses name attribute as the name, and sets up that querystring, name equals value based on what I typed in.



**Example with json parser -** 



//

**having a RESTful API means that**

**you design your API so that it responds to the HTTP request verbs, the methods, in the way that it's expected. And you also look at the URL and understand what it's doing.**

**So if I combine this URL, I see /person and an ID, and I combine that with the HTTP method being used, I can pretty much guess what this is going to do.**

I sent an HTTP request with the delete verb to a /person address, with an ID, I can pretty much guess that this API is going to delete that person out of the database. And that's all the idea of a RESTful API really is.

