EXPRESS 201

We're going to start to explore some of the deeper features of express.

TRAJECTORY

- **Handling URL params**

- **`app.use` and `next`**

- **Serving static resources**

URL PARAMETERS
REQ.PARAMS

Let's discuss one more feature of express - the ability to have wildcard parameters in our urls.

GET /PUPPIES/1

You may have seen urls that look like this before. Something like this seems to suggest that we want to go to, say, the web page for the puppy with the id of 1 (or, if this is an API serving up data instead of html, we might want to get the JSON data for the puppy with the id of 1 from our database). Either way, the idea is that we don't want all the puppies, we just want the first one.

```
app.get('/puppies/:puppyId', (req, res, next) ⇒ {
  console.log(req.params.puppyId)
  // use req.params.puppyId to find the puppy with the requested id
  // and then send that puppy as the response!
})
```

Here's that better way. We define a route, and in the place where we would put the 1, or 2, or 3, we place some identifier preceded by a colon. Express interprets the colon as a wildcard. So express will say: "Okay, if I get a request for /puppies/something, I will execute this handler, AND I will take that something, and I'll put it someplace you can get it on the request object. That place is req.params, and the name that you give to the wildcard matters. So say we have /puppies/:puppyId. If someone makes a GET request to /puppies/1, this means that req.params.puppyId will be 1. If someone makes a request to /puppies/2, req.params.puppyId will bw 2. If we change our code so that :puppyId is :pupId or :id, then we'd need to look in req.params.pupId or req.params.id, respectively

APP.USE AND NEXT

For starters, we're going to learn a new "app dot" function called "app.use", and learn about what that "next" thing is.

To understand what these are for though, we need to get a bit of a better understanding of how express evaluates an incoming request.
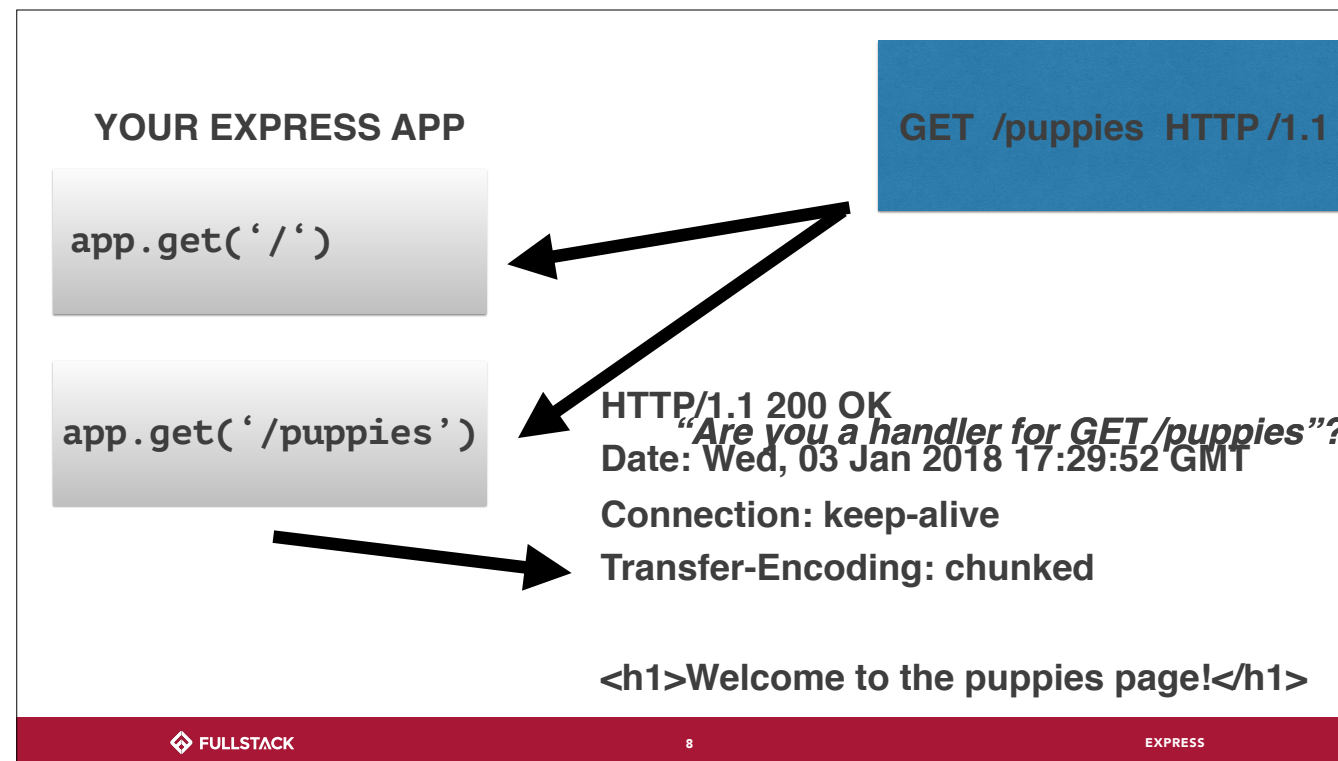
```
const express = require('express')

const app = express()

app.get('/', (req, res, next) ⇒ {
  res.send(`<h1>Welcome to the main page</h1>`)
})

app.get('/puppies', (req, res, next) ⇒ {
  res.send(`<h1>Welcome to the puppies page</h1>`)
})

app.listen(3000)
```

Let's take a closer look at what express does with this code. Actually, first let's talk about what express DOESN'T DO.

YOUR EXPRESS APP

app.get('/')

app.get('/puppies')

GET /puppies HTTP/1.1

HTTP/1.1 200 OK
"Are you a handler for GET /puppies"?
Date: Wed, 03 Jan 2018 17:29:52 GMT

Connection: keep-alive

Transfer-Encoding: chunked

<h1>Welcome to the puppies page!</h1>

Here's express. This list right here represents those request handlers that we wrote. Express is just holding these in a big array.

Now here comes a request. GET for /puppies. Now it seems reasonable to think that Express sees this request and is like, well, you are a get request to puppies, so straightaway here you go to the /puppies handler. But this isn't what happens. Express will check each request handler we've written, in the order we've written them.

So first it goes to our GET / handler and asks, "Are you a handler for GET /puppies"? And express realizes it isn't so we move down to the next one, in order. And it does the same thing, it asks, "Are you a handler for GET /puppies". And this time it is, so it invokes the request handler.

Now this seems silly. Why would we do this? It seems ridiculous, but wait - what if we want to do something *every* GET request that comes in. For example, wouldn't it be useful to log something to the console every time someone makes a request? That would really help us.

```
app.get('*', (req, res, next) ⇒ {
  console.log(req.method, req.url)
})

app.get('/', (req, res, next) ⇒ {
  res.send(`<h1>Welcome to the main page</h1>`)
})

app.get('/puppies', (req, res, next) ⇒ {
  res.send(`<h1>Welcome to the puppies page</h1>`)
})
```

Okay, so here's what we might do, knowing what we know now. We would write a new handler for GET requests, that handles ANY url. We can do this using the star, or glob. It acts as a catch all. In the handler, we're going to console log the method, that is the verb, and the url.

And what we expect, and we'll see that this is wrong, is that Express will just go merrily on its way down to the next handler. Here's what happens…

The same process occurs. We go through each handler in order. We say "Are you a handler for GET /puppies"? And the star handler is for everything, so we execute it's callback and then…nothing happens. We don't just automatically go on to the next handler. Does anyone have any idea why this might be?

The answer is what if, in this handler, we wanted to do something asynchronous before moving on to the next handler. We wouldn't be able to do that. So what express provides for us is the *next* function, which is a lot like our *done* callback from node shell.

```
app.get('*', (req, res, next) ⇒ {
  console.log(req.method, req.url)
  next()
})

app.get('/', (req, res, next) ⇒ {
  res.send(`<h1>Welcome to the main page</h1>`)
})

app.get('/puppies', (req, res, next) ⇒ {
  res.send(`<h1>Welcome to the puppies page</h1>`)
})
```

Now we have *next* in the mix.
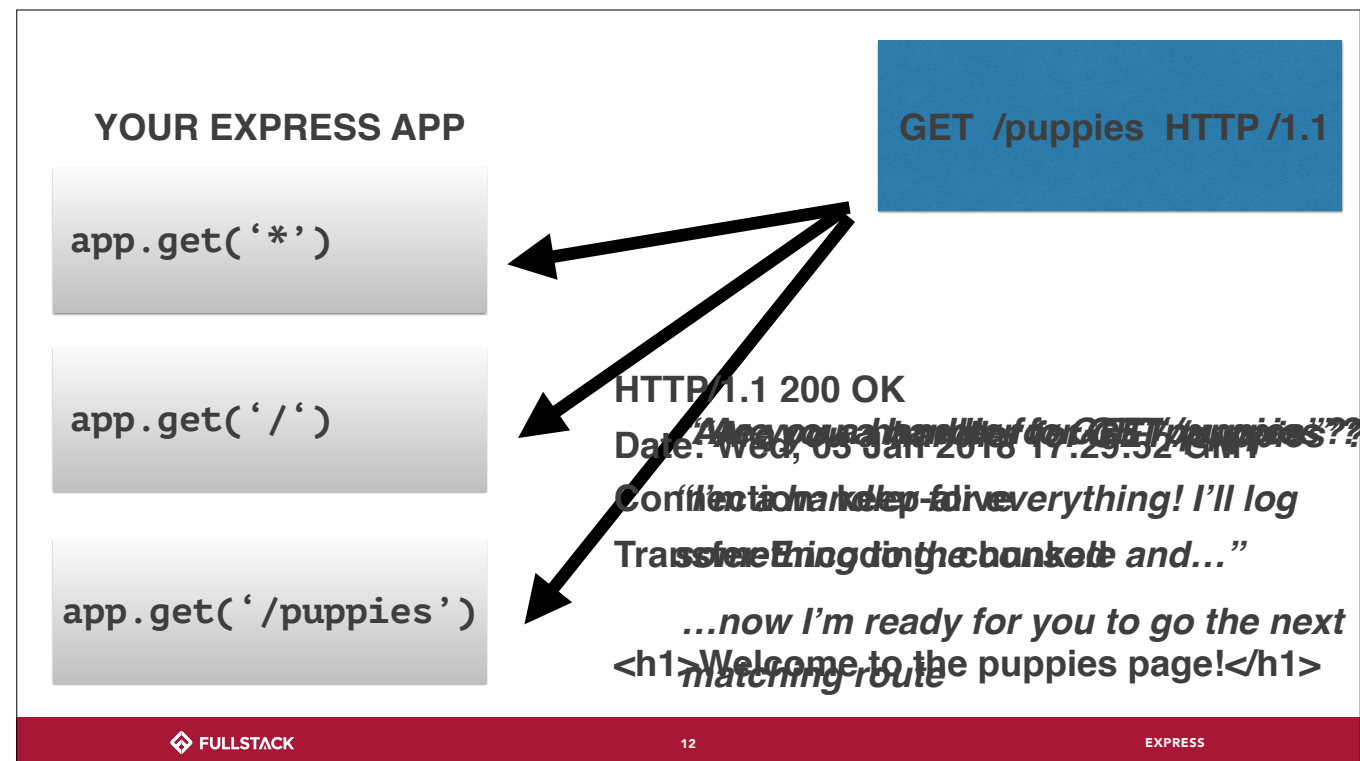
Now this process is going to proceed as we expect.

So now we have a nice logger, but it's only logging for GET requests. This kind of stinks - we're only using GET requests right now, but eventually we want to also deal with POST, PUT and DELETE requests.

Fortunately, there's a method for that - app.use

```
app.get('*', (req, res, next) ⇒ {
  console.log(req.method, req.url)
  next()
})

app.get('/', (req, res, next) ⇒ {
  res.send(`<h1>Welcome to the main page</h1>`)
})

app.get('/puppies', (req, res, next) ⇒ {
  res.send(`<h1>Welcome to the puppies page</h1>`)
})
```

Now we're going to do something for every request, regardless of verb.

APP.GET

- Only for GET requests

- Exact match of verb and URL

APP.USE

- Handles all verbs

- Fuzzy match of url (we'll explore this more later)

So let's look at app.get and app.use side-by-side.

EXPRESS MIDDLEWARE

A function that handles responding to requests.
Of the form: `function(req, res, next){…}`

So let's talk about these handler functions a bit more. We can see the way that express uses them is that it checks each one in turn to see if it needs to be run based on the rules of express. We call these functions "middleware" functions. If you're familiar with systems architecture, you might recognize the term, and it's borrowed from there, but it means something a bit more specific here. A middleware function is any function we give to an express handler like app.use or app.get. The idea is that our app receives the request, and between receiving and responding (that is, in the middle), we have all of these other functions that can do things with that request on its way to it's eventual response.

```
app.use((req, res, next) ⇒ {
  console.log(req.method, req.url)
  next()
})
```

A very common piece of middleware is logging middleware, like the one we saw. Here it is, about as simple as we could write it.

```
const morgan = require('morgan')

app.use(morgan('dev'))
```

Very often, we use libraries that actually give us middleware functions that do these sorts of things for us (and often better than we would have done ourselves). A very common npm package that gives us a logging middleware function is called morgan. Here's how we hook it up. Note that the thing that we get when we require morgan isn't the middleware function - it's a function that we invoke with an argument that tell's the morgan library what kind of logger we want. In this case, we're saying we want the 'dev' logger - a logger that's appropriate for a dev testing stuff out - and there are other kinds that we could get as well.
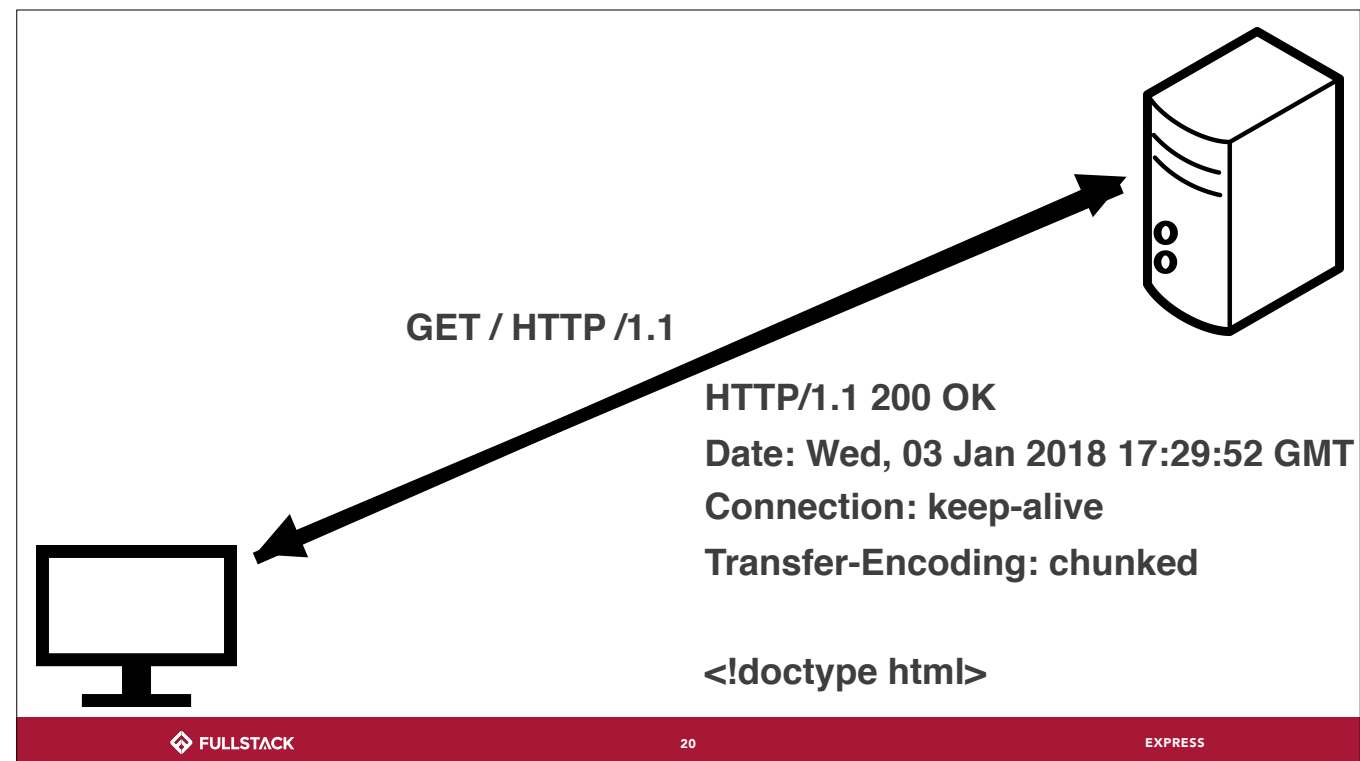
EXPRESS.STATIC

Okay, so we know that we can use middleware to log stuff to the console. Let's talk about writing static-file serving middleware. We're going to use a utility to write this kind of middleware called "express.static".

```
/server
  app.js
/public
  styleA.css
  styleB.css
  clientSideScript.js
```
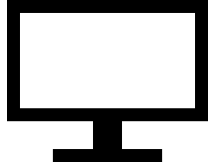
First, what is a "static resource" or a "static asset". That's just a fancy name for a file that our server might send down to the client. This might include html files, css files and client-side javascript files. By convention, these are often stored in a directory called "public" (but purely by convention - there's nothing magic about it). They're called "static" because they're files, they don't change the way that your data (stored in a database) might - you might get likes, or upvotes or change your password, but files are files.

Here's an example of a project that has two sub-directors: one called "server" which contains a file called "app.js", and we might have the code for our express app in here. And then we have a public folder that has a bunch of static assets, like css and some js

Side note: it's important to make sure that you see the difference between the server-side javascript, our app.js, and the client-side javascript (our "clientSideScript.js). One of these is meant to be executed on YOUR computer, the server, and the other is intended to be served up by that server to run in SOMEBODY ELSE's browser.

GET / HTTP /1.1

HTTP/1.1 200 OK
Date: Wed, 03 Jan 2018 17:29:52 GMT
Connection: keep-alive
Transfer-Encoding: chunked

<!doctype html>

Let's take a look at how the client is going to request those files. When visiting your website, most initial requests are going to be GET for /. Typically, this request will server up an "index.html", so we'll go ahead and serve that….

```
<!doctype html>
    <head>
      <link rel="stylesheet" href="/style.css" />
    </head>
    <body>
    </body>
</html>
```

When the client's browser gets this response, it sees that it has some html in it, so it goes through the process of parsing that html line-by-line so that it can render it and make it look nice in the browser. It evaluates it line by line, similar to the way that javascript is evaluated line by line.

When we get to this link tag, your browser sees it, sees that theres an href on it, and knows - hey, I need some more resources. I'm going to make another request to the same server that gave me this document, and ask it for GET /style.css. It knows to do this because that's just the way we've specified that browsers need to behave when they get this kind of HTML.
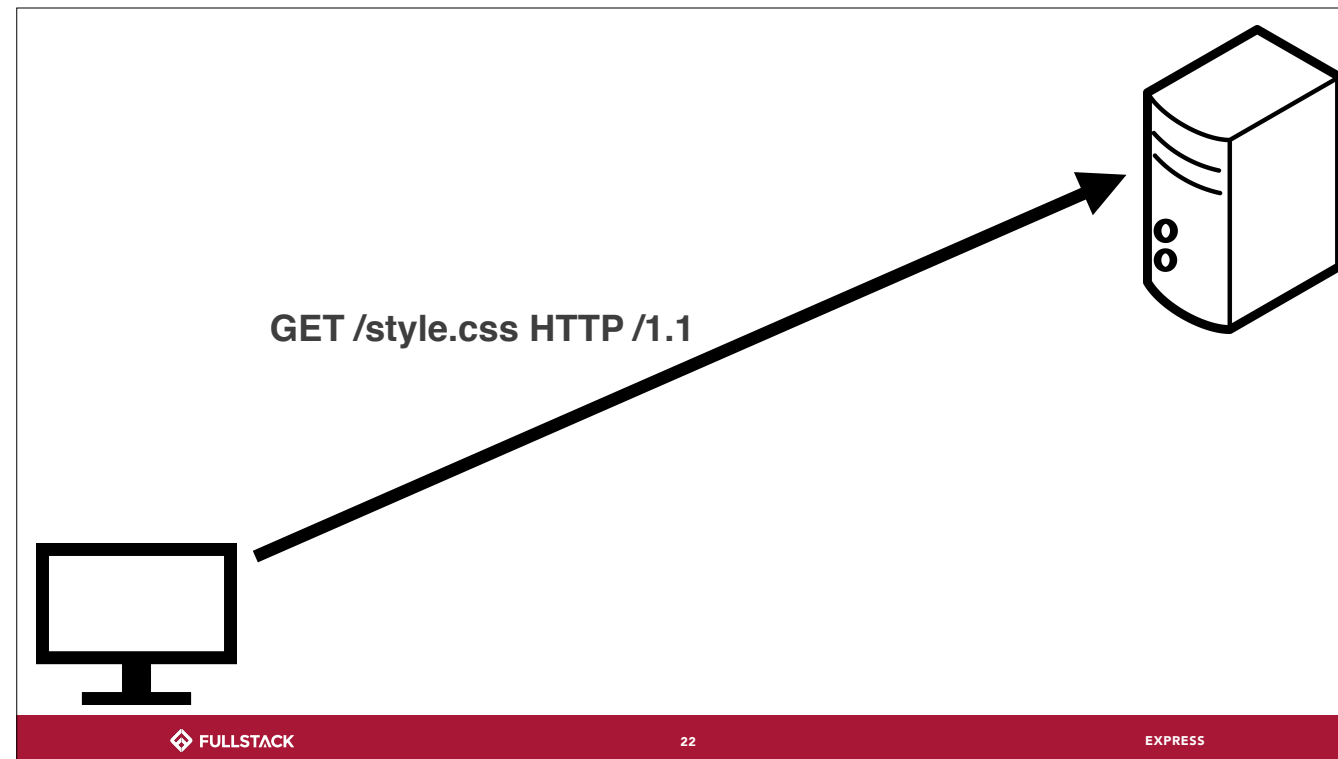
So the browser stops what it's doing (it's blocking, just like in javascript), and it makes another HTTP request for GET /style.css.

Now our server needs to handle this request and send back a file called /style.css (it could of course send back whatever it wanted, but that appears to be what's being asked for).

```
app.get('/style.css', (req, res, next) ⇒ {
  res.sendFile(path.join(__dirname, 'public', 'style.css'))
})
```

Here's a route handler that would do just that.
A new thing here is that we're using the built-in path module to construct an absolute file path to our css file, but don't worry too much about that. The important thing to get here is that this line of code sends the style.css file from our public folder as a response.

Here's the problem. We don't actually have just one style.css file. We have many. And we might have many more. So, do we want to write a new route handler just to serve up the files in our public folder? No way!

```
const staticMiddleware = express.static(path.join(__dirname, 'public'))
app.use(staticMiddleware)
```

Instead, express has this built in utility, called `express.static`. What express.static gives us is a middleware function (that is, a function with the signature (req, res, next)), that will accept any incoming request, and check and see if the name of the url path matches a file in our public directory (note where we specify the path to the public directory). So instead of writing a whole bunch of middleware, we just write the one. Pretty cool. And remember that the term "public" is just convention - you could call it "static", or whatever you want. We just need to tell the express.static where to look for files.