

11/20/2015 • 7 minutes to read

## In this article

[Routing, Redux](#)

[The "E" in MEAN](#)

[Hello, in JSON](#)

[Wrapping Up](#)

November 2015

Volume 30 Number 12

# The Working Programmer - How To Be MEAN: Express Routing

By [Ted Neward](#) | November 2015



Welcome back, "Nodeists." (I have no idea if that's the official term for those who use Node.js on a regular basis, but Nodeists sounds better to me than "Nodeheads" or "Noderati" or "Nodeferatu.")


In the last installment ([msdn.com/magazine/mt573719](https://msdn.com/magazine/mt573719)), the application's stack had gone from being an "N" stack (just Node) to an "EN" stack by virtue of installing Express to go along with the Node.js. As tempting as it would be to jump directly onto other things, there are a few more things about Express—and its supporting packages and libraries—that deserve exploration and further discussion. You previously got a taste of one of these, Express routing, when the code set up a function to display "Hello World" in response to HTTP requests to the "/" relative URL path. Now, I'll go a little deeper into the Express world and show you how to use it more effectively.

By the way, those who are interested in seeing the latest-and-greatest code being written as part of this series can visit the Microsoft Azure site that holds the most recent of this series' code ([msdn-mean.azurewebsites.net](https://msdn-mean.azurewebsites.net)). It's likely that the information in this column is out of sync with what's on the site, given publication schedules, and the site gives readers a look at what's to come.

# Routing, Redux

A recap of the app.js file from the last installment shows the single-endpoint nature of the application that's been built so far, as shown in **Figure 1**, the simple-yet-necessary homage to the Gods of Computer Science.

Figure 1 Code for the Express "Hello World"

JavaScript	 Copy
<pre>// Load modules var express = require('express'); var debug = require('debug')('app'); // Create express instance var app = express(); // Set up a simple route app.get('/', function (req, res) {   debug("/ requested");   res.send('Hello World!'); }); // Start the server var port = process.env.PORT    3000; debug("We picked up",port,"for the port"); var server = app.listen(port, function () {   var host = server.address().address;   var port = server.address().port;   console.log('Example app listening at http://%s:%s', host, port); });</pre>	

The part in question is the section of code labeled "Set up a simple route"; here, you're establishing a single endpoint, mapped by the HTTP verb ("get") and relative URL endpoint ("/," passed as the first argument to the "get" method).

It's fairly easy to infer the pattern for the other HTTP verbs—for a "POST" request, you use the post method; for a "PUT," put; and "DELETE," you use delete. Express supports the other verbs, too, but for fairly obvious reasons, these are the four you most care about. Each also then takes as its second argument a function, which in the example in **Figure 1** is a function literal that handles the incoming HTTP request.

## The "E" in MEAN

Often, when Nodeists write Express-based applications, they do so in the same manner that we ".NETers" write ASP.NET applications. The server generates an HTML document containing the presentation (HTML) intermixed with the data, and sends that back to the browser, after which the user fills out a form and POSTs the entered data back to Express; or, the user clicks on a link and generates a GET back at Express to do the complete server-side cycle again. And, because handwriting HTML in Node.js is just as

much fun as it is in Visual Basic or C#, a number of tools emerged out of the Node.js world designed to serve the same purpose the Razor syntax does in a classic ASP.NET application. It makes it easier to write the presentation layer without co-mingling the data and code too much.

However, in a MEAN-based application, AngularJS will form the complete client-side experience, so Express takes on the same role as ASP.NET MVC—it's simply a transport layer, taking raw data (usually in the form of JSON) from the client, acting on that data (usually either storing it, modifying it, or finding associated or related data) and sending raw data (again, usually in the form of JSON) back to the client tier. Toward that end, our sojourn in Express will avoid the subject of templating frameworks (of which there are several in the Node.js world, "handlebars" and "jade" being two of the more popular), and I'll focus explicitly on simply shipping JSON back and forth. Some will call this a RESTful endpoint, but, frankly, REST involves a lot more than just HTTP and JSON, and building a Fielding-approved RESTful system is well beyond the scope of this series.

So, for now, I'll talk about standing up a couple of simple read-only endpoints for any JSON-consuming client to use.

## Hello, in JSON


Usually, a Web API follows a fairly loose structure for obtaining data:

- A GET request to a given resource type (such as "persons") will yield a JSON result that's an array of objects, each one containing at minimum a unique identifier (for individual retrieval) and usually some kind of short descriptive text, suitable for display in a list of options.
- A GET request to a given resource type with an identifier as part of the URL ("persons/1234," where 1234 is the identifier uniquely identifying the person we're interested in) will yield a JSON result that is (usually) a single JSON object describing the resource in some level of detail.

Web APIs will also use PUT, POST and DELETE, but for now, I'll focus on just retrieving data.

So, assuming the resource type is "persons," you'll create two endpoints, one labeled `/persons,` and the other `/persons/<unique identifier>.` For starters, you need a small "database" of persons to work with—first names, last names, and their current "status" (whatever they happen to be doing right now) will suffice (see **Figure 2**).

Figure 2 Creating a Small Database of Persons

JavaScript	 Copy
------------	--

```
var personData = [
  {
    "id": 1,
    "firstName": "Ted",
    "lastName": "Neward",
    "status": "MEANing"
  },
  {
    "id": 2,
    "firstName": "Brian",
    "lastName": "Randell",
    "status": "TFSing"
  }
];
```

Not exactly SQL Server, but it'll do for now.

Next, you need the endpoint for the full collection of persons:

JavaScript

 Copy

```
var getAllPersons = function(req, res) {
  var response = personData;
  res.send(JSON.stringify(response));
};
app.get('/persons', getAllPersons);
```

Notice that in this case, the route mapping is using a standalone function (getAllPersons), which is more common, because it helps keep a separation of concerns a little more clean—the function acts as a controller (in the Model-View-Controller sense). For now, I use JSON.stringify to serialize the array of JavaScript objects into a JSON representation, but I'll use something more elegant later.

Next, you need an endpoint for individual person objects, but this will take a bit more doing because you need to pick up the person identifier as a parameter, and Express has a particular way of doing this. One way (arguably the easier way on the surface of things) is to use the "params" object of the request object (the "req" parameter to the function used in the route map) to fetch the parameter specified in the route, but Node.js can also use a parameter function to do more—it's a form of filter, which will be invoked when a parameter of a particular naming pattern is found:

JavaScript

 Copy

```
app.get('/persons/:personId', getPerson);
app.param('personId', function(req, res, next, personId) {
  debug("personId found:", personId);
  var person = _.find(personData, function(it) {
    return personId == it.id;
  });
  debug("person:", person);
});
```

```
req.person = person;
next();
});
```

When the route is invoked, whatever follows `"/persons"` (as in `"/persons/1"`) will be bound into a parameter of name `"personId,"` just as you might find with ASP.NET MVC. But then when using the `param` function—which will be invoked when any route with `":personId"` is invoked—the associated function is invoked, which will look up (using the `"lodash"` package function `find`, as shown in the previous code snippet) from the tiny `personData` database. Then, however, it's added to the `"req"` object (because JavaScript objects are always dynamically typed, it's trivial to do), so that it will be available to the remainder of what is invoked, which in this case will be the `getPerson` function—this now becomes quite trivial, because the object you want to return is already fetched:

JavaScript

 Copy

```
var getPerson = function(req, res) {
  if (req.person) {
    res.send(200, JSON.stringify(req.person));
  }
  else {
    res.send(400, { message: "Unrecognized identifier: " + identifier });
  }
};
```

See what I mean by "trivial"?

## Wrapping Up

I've got a bit more to do with Express, but despite being on a roll here, I'm out of space for this one, so ... happy coding!

**Ted Neward** is the CTO at iTrellis, a consulting services company. He has written more than 100 articles and authored or co-authored a dozen books, including *"Professional F# 2.0"* (Wrox, 2010). He is an F# MVP and speaks at conferences around the world. He consults and mentors regularly—reach him at [ted@tedneward.com](mailto:ted@tedneward.com) or [ted@itrellis.com](mailto:ted@itrellis.com) if you're interested.

Thanks to the following technical expert for reviewing this article: Shawn Wildermuth