

02/18/2016 • 8 minutes to read

## In this article

[Another One Bites the Dust ...](#)

[Get into My Database](#)

[How Do I POST Thee?](#)

[Keeping Up-to-Date with the Neighbors](#)

[Wrapping Up](#)

---

December 2015

Volume 30 Number 13

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

By [Ted Neward](#) | December 2015



Welcome back, “Nodeists.” (I established that as the semi-official term of endearment for those who use Node.js on a regular basis. If you don’t care for it, drop an e-mail or a tweet with a better suggestion, bearing in mind my other two ideas were “Noderati” or “Nodeferatu.”)

In the previous installment, the application had grown to include some output capabilities, in the form of Web API endpoints for obtaining either the collection of persons (my resource for this application; I seem to be building some kind of people database) or the individual person via an arbitrary “id” given as part of the URL. It’s time to start processing input—the ability to put a new person into the system, remove a person from the system and update an existing person. In some ways, these are “just” new URL endpoints to the application, but there are a few new tricks I want to talk about along the way.


As I mentioned last time, those who are interested in seeing the latest-and-greatest of the code being written as part of this series can visit the Microsoft Azure site that holds the latest of this series’ code ([msdn-mean.azurewebsites.net](https://msdn-mean.azurewebsites.net)). It’s likely that the text

here is out of sync with what's on the site, given publication schedules, but if anything, the site will be ahead of what's here, giving readers a look ahead at what's to come next.

Speaking of the last column, as of the last installment, the code can display existing people in the database, but there's no modification of them whatsoever yet. Because that's usually a critical part of any online system, let's add the "CUD" to the "R" to finish out the CRUD.

## Another One Bites the Dust ...

The easiest one to implement first is the "D" in CRUD: Delete. Setting up a route in Express requires only that the code use delete instead of the get used last time:

JavaScript	 Copy
<pre>app.delete('/persons/:personId', deletePerson);</pre>	

Recall from my last column that the ":personId" is a parameter that will be available on the Express "req" (request) object in the associated function (deletePerson), and picked up by the personId middleware function also described last time, so that you can know which person to remove from the system.

Speaking of deletePerson, the implementation is relatively straightforward, using the lodash remove method to search the in-memory database and remove the individual described (see **Figure 1**).

Figure 1 Using the Lodash Remove Method

JavaScript	 Copy
<pre>var deletePerson = function(req, res) {   if (req.person) {     debug("Removing", req.person.firstName, req.person.lastName);     _.remove(personData, function(it) {       it.id === req.person.id;     });     debug("personData=", personData);     var response = { message: "Deleted successfully" };     res.status(200).jsonp(response);   }   else {     var response = { message: "Unrecognized person identifier" };     res.status(404).jsonp(response);   } };</pre>	

The `personId` middleware will pick up `:personId` (whatever its value), find the appropriate person object in the database and put that as a `person` property on the incoming request (`"req"`) object, so if there's no `req.person`, it means nobody by that ID was found in the database. When you send the response, however, instead of converting the result to JSON via the built-in `JSON.stringify` method from the previous articles, use the `jsonp` method of converting the data to a JSON format—or, to be more accurate, a JSONP (JSON with padding) format, which is widely considered to be the superior (and safer) way of sending JSON back to browsers. Notice how the method calls `"chain"`; that means you can use one fluent line of code to send back either a 200 with the JSON response consisting of a message that the person was deleted successfully, or a 404 with the message that the person's ID wasn't recognized.

## Get into My Database

Next, you should probably support putting people into the database. Again, this is pretty straightforward routing: Conventionally, Web API advocates suggest using a POST to the resource-collection URL (`/persons`) as the means to insert into the database, so do that:

JavaScript

 Copy

```
app.post('/persons', insertPerson);
```

Not particularly exciting. But a new wrinkle emerges: In order to insert the person into the system, you're going to need to pick out the JSON for the person data out of the incoming request. One way to do that would be to grab the entire request body using the `req.body` parameter, but then you're left with the onerous (and slightly dangerous) task of parsing the JSON into an object suitable for storage. Instead, this is where the Node.js community leans on its extensive collection of libraries, and sure enough, there's a good library out there that handles that, called `body-parser`. Install it (`"npm install body-parser"` in the same directory as `package.json`) and then you can reference it and tell the Express app object to use it:


JavaScript

 Copy

```
// Load modules
var express = require('express'),
    bodyParser = require('body-parser'),
    debug = require('debug')('app'),
    _ = require("lodash");
// Create express instance
var app = express();
app.use(bodyParser.json());
```

Recall that discussion earlier around `:personId`, about how Express allows you to create middleware functions that do a little bit of work silently as part of the pipeline? That's exactly what the `body-parser` library does—it installs a number of “hooks” (for lack of a better term) to process the body of the request in a variety of forms. Here, you're asking it to parse JSON, but it can also support URL encoding, parsing everything as a giant string (making it easier to parse CSV, for example), or grabbing everything “raw” into a Node.js Buffer (presumably because the incoming data is binary of some form). Because most of what we care about right now is handled with JSON, just using that is sufficient.

The `insertPerson` function, then, is actually really anticlimactic:

JavaScript	 Copy
<pre>var insertPerson = function(req, res) {   var person = req.body;   debug("Received", person);   person.id = personData.length + 1;   personData.push(person);   res.status(200).jsonp(person); };</pre>	

(OK, the unique-id generation code could use a lot of work, but remember the goal is to eventually end up using MongoDB, so let's not spend a ton of time worrying about that.)

When returning from `insertPerson`, the code sends back the complete `Person` object just inserted, including its new `id` field; this isn't a completely standard convention, but in my own code I've found it helpful. This way the client is aware of any additional validation/correction/server-side-augmentation of a submitted entity, such as the `id` field in this case.

## How Do I POST Thee?

By the way, one of the interesting parts about building a Web API this way (as opposed to a more traditional Web app) is that some of the simple tricks for doing quick-and-dirty testing aren't available to you. For example, how do you quickly test to see if the `insertPerson` function's working? To be sure, there's a ton of support for automating testing in Node.js, and that's coming in a future column, but for now, it's easiest to use either a browser plug-in (like Chrome Postman) or, for the die-hard command-line fan, the `cURL` freeware utility that comes pre-installed on OS X and most Linux images. If you don't like either of those, there's a number of others, all of which stretch across the entire gamut of complexity, from the ad hoc to the automated, including one of my favorites, Runscope ([runscope.com](https://runscope.com)), a cloud-based system for automated testing of your API endpoints, for something running 24x7.

Regardless of what you use, if you don't like it, move on quickly, because there are literally thousands of them out there.

## Keeping Up-to-Date with the Neighbors

Finally, the app needs to support updating a person already in the database; in many ways, this is a combination of the complexity of delete (finding the right person in the database) and insert (parsing the incoming JSON), but the middleware already installed handles most of that already, as shown in **Figure 2**.

Figure 2 Updating a Person Already in the Database

JavaScript	 Copy
<pre>var updatePerson = function(req, res) {   if (req.person) {     var originalPerson = req.person;     var incomingPerson = req.body;     var newPerson = _.merge(originalPerson, incomingPerson);     res.status(200).jsonp(newPerson);   }   else {     res.status(404).jsonp({ message: "Unrecognized person identifier" });   } }; // ... app.put('/persons/:personId', updatePerson);</pre>	

The key thing in **Figure 2** is the use of the lodash method `merge`, which enumerates across the properties of the second parameter and either overwrites the property of the same name on the first parameter, or else adds it in. It's an easy way to copy the attributes of the second onto the first without destroying and recreating the first object (which is important in this case, because the first object is inside the `personData` array that serves as our impromptu database).

For the curious, the true branch of the if statement could be condensed down into one line of code:

JavaScript	 Copy
<pre>var updatePerson = function(req, res) {   if (req.person) {     res.status(200).jsonp(_.merge(req.person, req.body));   }   else {     res.status(404).jsonp({ message: "Unrecognized person identifier" });   } };</pre>	

... but whether that's more or less readable is up to you to decide.

## Wrapping Up

For anybody who's counting, the `app.js` file that began life in part two of this series is now exactly 100 lines of code (which includes about a half-dozen lines of comments), and now supports a full CRUD Web API against an in-memory database. That's not bad for a century's worth in your text editor. But there's more to do: The system needs to move away from using the in-memory database toward using MongoDB, and in order to do that without breaking all of the clients that are already using this highly prized Web API, there needs to be tests. Nobody wants to have to run tests over and over again by hand, so the next steps are to add automated tests to ensure the transition to MongoDB is seamless and simple from the client's perspective. In the meantime ... happy coding!

---

**Ted Neward** is the CTO of iTrellis, a Seattle-based polytechnology consulting firm. He has written more than 100 articles, is an F# MVP, INETA speaker, and has authored or co-authored a dozen books. Reach him at [ted@tedneward.com](mailto:ted@tedneward.com) if you're interested in having him come work with your team, or read his blog at [tedneward.com](http://tedneward.com) .

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