Understanding Express.js

5 MARCH 2014

This is aimed at people who have some familiarity with Node.js. They know how to run Node scripts and can install packages with npm. You don't have to be an expert, though -- I promise. This guide was last updated for Express 4.0.0. It's an introduction and mostly deals with concepts.

This guide is for Express 4. If you're looking to learn about Express 3.x, take a look at my guide for the previous version.

If this isn't in-depth enough for you, I've written a book about Express. Go take a look at Express.js In Action if you thirst for more!

Express.js describes itself better than I can: "a minimal and flexible node.js web application framework". It helps you build web apps. If you've used Sinatra in the Ruby world, a lot of this will be familiar.

Like any abstraction, Express hides difficult bits and says "don't worry, you don't need to understand this part". It does things for you so that you don't have to bother. In other words, it's magic.

It's good magic, too. Express catalogs some people using it, and there are some big names: MySpace, Klout, and even some stuff *I've* made. *Me*. I'm a *huge* deal. I've got a *blog*.

But all magic comes at a price: you might not understand the inner workings of Express. This is like driving a car; I drive a car just fine without intimate knowledge of its workings, but I'd be better off with that knowledge. What if things break? What if you want to get all the performance you can out of the car? What if you have an *insatiable thirst for knowledge*?

So let's understand Express from the bottom, with Node.

Bottom layer: Node's HTTP server

Node has an HTTP module which makes a pretty simple abstraction for making a webserver. Here's what that might look like:

```
// Require what we need
var http = require("http");

// Build the server
var app = http.createServer(function(request, response) {
  response.writeHead(200, {
    "Content-Type": "text/plain"
  });
  response.end("Hello world!\n");
});

// Start that server, baby
app.listen(1337, "localhost");
console.log("Server running at http://localhost:1337/");
```

And if you run that app (if that file is called `app.js`, you'd run `node app.js`), you'll get a response of "Hello world!" if you visit `localhost:1337` in your browser. You'll get the same response no matter what, too. You can try visiting `localhost:1337/anime_currency` or `localhost:1337/?onlyfriend=anime`, and it's like talking to a brick wall: "Hello world!"

Let's break this down a bit.

The first line uses the `require` function to load a built-in Node module called `http`. It puts this lovely module inside of a variable called `http`. For more about the require function, check out Nodejitsu's docs.

Then we put a server in a variable called `app` by using `http.createServer`. This takes a function that listens for requests. We'll get back to this in a minute because they're Super Duper Important. Skip over it for the next two sentences.

The last thing we do is tell the server to listen for requests coming in on port 1337, and then we just log that out. And then we're in business.

Okay, back to the request handler function. That thing is *important*.

The request handler

Before I start this section, I should say that there's a bunch of cool HTTP stuff in here that I don't think is relevant to learning Express. If you're interested, you can look at the docs for the HTTP module because they have a bunch of stuff.

Whenever we make a request to the server, that request handler function is called. If you don't believe me, try putting a `console.log` in there. You'll see that it logs out every time you load a page.

`request` is a request that comes from the client. In many apps, you'll see this shortened to `req`. Let's look at it. To do that, we'll modify the above request handler a bit:

```
var app = http.createServer(function(request, response) {
  // Build the answer
  var answer = "";
  answer += "Request URL: " + request.url + "\n";
  answer += "Request type: " + request.method + "\n";
  answer += "Request headers: " + JSON.stringify(request.headers) +
"\n";

  // Send answer
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end(answer);
});
```

Restart the server and reload `localhost:1337`. You'll see what URL you're requesting, that it's a GET request, and that you've sent a number of cool headers like the user-agent and more complicated HTTP stuff! If you visit `localhost:1337/what_is_anime`, you'll see the request URL change. If you visit it with a different browser, the user-agent will change. If you send it a POST request, you'll see the method change.

The `response` is the next part. Just like the prior argument is often shortened, this is often shortened to the three-letter `res`. With each response, you get the response all ready to send, and then you call `response.end`. Eventually, you *must* call this method; even the Node docs say so. This method does the actual sending of data. You can try making a server where you don't call it, and it just hangs forever.

Before you send it out, you'll want to write some headers. In our example, we do this:

```
response.writeHead(200, { "Content-Type": "text/plain" });
```

This does two things. First, it sends HTTP status code 200, which means "OK, everything is good". Then, it sets some response headers. In this case, it's saying that we're sending back the plaintext content-type. We could send other things like JSON or HTML.

I thirst for more

You want more? Okay. You asked nicely.

One could imagine taking these APIs and turning them into something cool. You could do something (sorta) like this:

```
var http = require("http");

http.createServer(function(req, res) {
  // Homepage
  if (req.url === "/") {
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end("Welcome to the homepage!");
  }

  // About page
  else if (req.url === "/about") {
    res.writeHead(200, { "Content-Type": "text/html" });
    res.end("Welcome to the about page!");
  }

  // 404'd!
  else {
    res.writeHead(404, { "Content-Type": "text/plain" });
    res.end("404 error! File not found.");
  }
}).listen(1337, "localhost");
```

You could clean this up and make it pretty, or you could be hardcore like the npm.org folks and tough it out with vanilla Node. But you could also build a framework...a framework like Express.

Middleware, the middle layer

The middle layer of this JavaScript cake is conveniently called "middleware". Don't go searching "what is middleware" just yet -- I'm about to explain it.

A little bit of Express code

Let's say we wanted to write the "hello world" app that we had above, but with Express this time. Don't forget to install Express (`npm install`, baby). Once you've done that, the app is pretty similar.

```
// Require the stuff we need
var express = require("express");
var http = require("http");

// Build the app
var app = express();

// Add some middleware
app.use(function(request, response) {
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end("Hello world!\n");
});

// Start it up!
http.createServer(app).listen(1337);
```

So let's step through this.

First, we require Express. We then require Node's HTTP module just like we did before. We're ready.

Then we make a variable called `app` like we did before, but instead of creating the server, we call `express()`. What's going on? What is this madness?

We then add some middleware -- it's just a function. We pass this to `app.use`, and this function looks *an awful lot like* the request handlers from above. In fact, *I copy-pasted it*.

Then we create the server and start listening. `http.createServer` took a function before, so guess what -- `app` is just a function. It's an Express-made function that starts going through all the middleware until the end. But it's just a request handler like before.

(Worth noting that you might see people using `app.listen(1337)`, which just defers to `http.createServer`. That's just a shorthand.)

Okay, now I'm going to explain middleware.

What is middleware?

I want to start by saying that Stephen Sugden's description of Connect middleware is really good and does a better job than I can. (Don't worry that it's for something called "Connect" -- replace "Connect" with "Express" and you're good!) If you don't like my explanation, read his.

Remember the request handlers from a few sections earlier? Each piece of middleware is just another request handler. You start by looking at the first request handler, then you look at the next one, then the next, and so on.

Here's what middleware basically looks like:

```
function myFunMiddleware(request, response, next) {
   // Do stuff with the request and response.
   // When we're all done, call next() to defer to the next middleware.
   next();
}
```

When we start a server, we start at the topmost middleware and work our way to the bottom. So if we wanted to add simple logging to our app, we could do it!

```
var express = require("express");
var http = require("http");
var app = express();

// Logging middleware
app.use(function(request, response, next) {
  console.log("In comes a " + request.method + " to " + request.url);
  next();
});

// Send "hello world"
app.use(function(request, response) {
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end("Hello world!\n");
});

http.createServer(app).listen(1337);
```

If you run this app and visit `localhost:1337`, you'll see that your server is logging some stuff and you'll see your page.

It's important to note that anything that works in the vanilla Node.js server also works in middleware. For example, if you want to inspect `req.method`, it's right there.

While you can totally write your own, there's a *ton* of middleware out there. Let's remove our logger and use Morgan, a nice logger for Express. `npm install morgan` and give this a try:

```
var express = require("express");
var logger = require("morgan");
var http = require("http");
var app = express();

app.use(logger());
// Fun fact: logger() returns a function.

app.use(function(request, response) {
  response.writeHead(200, { "Content-Type": "text/plain" });
  response.end("Hello world!\n");
});

http.createServer(app).listen(1337);
```

Visit `localhost:1337` and you'll see some logging! Thanks, Morgan.

I thirst for more

One could imagine stringing together some middleware to build an app. Maybe you'd do it like this:

```
var express = require("express");
var logger = require("morgan");
var http = require("http");
var app = express();

app.use(logger());

// Homepage
app.use(function(request, response, next) {
  if (request.url === "/") {
    response.writeHead(200, { "Content-Type": "text/plain" });
    response.end("Welcome to the homepage!\n");
    // The middleware stops here.
  } else {
    next();
  }
});

// About page
app.use(function(request, response, next) {
  if (request.url === "/about") {
    response.writeHead(200, { "Content-Type": "text/plain" });
    response.end("Welcome to the about page!\n");
    // The middleware stops here.
  } else {
    next();
  }
});

// 404'd!
app.use(function(request, response) {
  response.writeHead(404, { "Content-Type": "text/plain" });
  response.end("404 error!\n");
});

http.createServer(app).listen(1337);
```

"This is ugly! I don't like it," you say. You *scum*. You're never satisfied, are you? *Will there ever be enough?*

The Express folks are smart. They know that this ugliness won't do. They're smart people.

Top layer: routing

We've finally arrived at the third act of our nerdy quest. We're at the peak of our abstraction mountain. There is a beautiful sunset. Your long, golden locks wave in the cool breeze.

Routing is a way to map different requests to specific handlers. In many of the above examples, we had a homepage and an about page and a 404 page. We'd basically do this with a bunch of `if` statements in the examples.

But Express is smarter than that. Express gives us something called "routing" which I think is better explained with code than with English:

```
var express = require("express");
var http = require("http");
var app = express();

app.all("*", function(request, response, next) {
  response.writeHead(200, { "Content-Type": "text/plain" });
  next();
});

app.get("/", function(request, response) {
  response.end("Welcome to the homepage!");
});

app.get("/about", function(request, response) {
  response.end("Welcome to the about page!");
});

app.get("*", function(request, response) {
  response.end("404!");
});

http.createServer(app).listen(1337);
```

Ooh. That's hot.

After the basic requires, we say "every request goes through this function" with `app.all`. And that function looks an awful lot like middleware, don't it?

The three calls to `app.get` are Express's routing system. They could also be `app.post`, which respond to POST requests, or PUT, or any of the HTTP verbs. The first argument is a path, like `/about` or `/`. The second argument is a request handler similar to what we've seen before. To quote the Express documentation:

> [These request handlers] behave just like middleware, with the one exception that these callbacks may invoke `next('route')` to bypass the remaining route callback(s). This mechanism can be used to perform pre-conditions on a route then pass control to subsequent routes when there is no reason to proceed with the route matched.

In short: they're basically middleware like we've seen before. They're just functions, just like before.

These routes can get smarter, with things like this:

```
app.get("/hello/:who", function(req, res) {
  res.end("Hello, " + req.params.who + ".");
  // Fun fact: this has security issues
});
```

Restart your server and visit `localhost:1337/hello/animelover69` for the following message:

> Hello, animelover69.

The docs also show an example that uses regular expressions, and you can do lots of other stuff with this routing. For a conceptual understanding, I've said enough.

But it gets cooler.

Cool Express features

Routing would be enough, but Express is absolutely ruthless.

Request handling

Express augments the request and response objects that you're passed in every request handler. The old stuff is still there, but they add some new stuff too! The API docs explain everything, but let's look at a couple of examples.

One nicety they give you is a `redirect` method. Here are some examples:

```
response.redirect("/hello/anime");
response.redirect("http://www.myanimelist.net");
```

This isn't in vanilla Node, but Express adds this stuff. It adds things like `sendFile` which lets you just send a whole file:

```
response.sendFile("/path/to/anime.mp4");
```

The request gets a number of cool properties, like `request.ip` to get the IP address and `request.files` to get uploaded files.

Conceptually, there's not much to know, other than the fact that Express extends the request and response. For everything Express gives you, check out the API docs.

Views

More features? *Oh, Express, I'm blushing.*

Express can handle views. It's not too bad. Here's what the setup looks like:

```
// Start Express
var express = require("express");
var app = express();

// Set the view directory to /views
app.set("views", __dirname + "/views");

// Let's use the Pug templating language
app.set("view engine", "pug");
```

The first block is the same as always. Then we say "our views are in a folder called 'views'". Then we say "use Pug". Pug is a templating language. We'll see how it works in just a second!

Now, we've set up these views. How do we use them?

Let's start by making a file called `index.pug` and put it into a directory called `views`. It might look like this:

```
doctype 5
html
  body
    h1 Hello, world!
    p= message
```

This is basically HTML without all the brackets. It should be *fairly* straightforward if you know HTML. The only interesting part is the last line. `message` is a variable! Woah! Where did that come from? *I'll tell you.*

We need to render the view from within Express. Here's what that looks like:

```
app.get("/", function(request, response) {
  response.render("index", { message: "I love anime" });
});
```

Express adds a method to `response`, called `render`. It does a bunch of smart stuff, but it basically looks at the view engine and views directory (the stuff we defined earlier) and renders `index.pug`.

The last step (I suppose it could be the first step) is to install Pug, because it's not bundled with Express. Add it to your `package.json` or `npm install` it.

If you get all of this set up, you'll see this page. Here's all the source code.

Actually building something

Most of the stuff in this post is conceptual, but let me push you in the right direction for building something you want to build. I don't want to delve into specifics.

You can install Express as an executable in your terminal. It spits out boilerplate code that's very helpful for starting your app. Install it globally with npm:

```
# You'll probably need `sudo` for this:
npm install -g express-generator
```

If you need help, use `express --help`. It spits out some options. For example, let's say I want to use EJS templating and LESS for CSS. My app is called "myApp". Here's what I'd type to get started:

```
express --ejs --css less myApp
```

It'll generate a bunch of files and then tell you to go into that directory and `npm install`. If you do that, you'll have a basic app running with `node app`! I'd recommend looking through the generated files to see some boilerplate, and then messing with it a bunch. It's hardly a full app, but I found it very helpful to poke through these files and mess with them when getting started.

Also helpful are the many official examples on GitHub.

Some concluding miscellany

- If you love CoffeeScript like I do, you should know that all of this stuff works with CoffeeScript. You don't even need to compile it! Instead of starting your server with `node app.js`, start it with `coffee app.coffee`. This is what I do in my apps. *Me.* I'm a big deal. I've got a *blog*.
- Express used to be built on a thing called Connect, which is like Express but it's *just* the

middleware layer. Connect middleware is compatible with Express middleware (but not the other way around).

I thirst for more

Is there no satisfying you? You *glutton*. You make me *sick*. Soon you're gonna be sitting in an opium den, eyes half-open, drooling out the last drop of your programming talent.

I have to shamelessly self-promote my book on the topic, *Express.js In Action*. I hope this tutorial gives you a good understanding of how Express works, but the book goes a lot more in depth and talks about lots of other things, too. Go give it a read if you're interested!

I won't go into them here, but people have built things on top of Express. The Express wiki lists them and many of them are Pretty Cool. You can use these frameworks if you'd prefer, or you can stick to the lower-level Express. Either way, go build cool stuff!