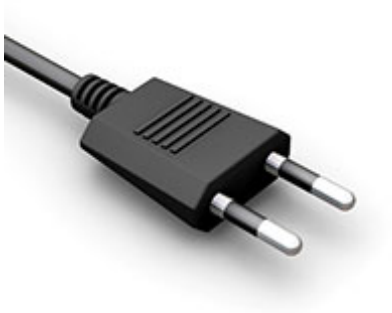


Meet the Connect Framework

Andrew Burgess on Apr 22nd 2013 with 19 Comments

Tutorial Details

-
- **Difficulty:** Intermediate
- **Completion Time:** 1 Hour

View post on Tuts+ Beta**Tuts+ Beta** is an optimized, mobile-friendly and easy-to-read version of the Tuts+ network.

Newcomers to NodeJS typically find its API difficult to grasp. Luckily, many developers have created frameworks that make it easier to work with Node. Connect is one such framework. It sits on top of Node's API and draws the line between comfort and control.

Think of Connect as a stack of middleware. With every request, Connect filters through the layers of middleware, each having the opportunity to process the HTTP request. When T.J. Holowaychuk announced Connect, he said there were two types of middleware. The first is a *filter*.

Filters process the request, but they do not respond to it (think of server logging).

The other type is a *provider*, which responds to the request. You can incorporate as many layers of middleware as you want; the request passes through each layer until one of the middleware responds to the request.

Basic Syntax

First, you need to install the Connect package through npm:

```
npm install connect
```

Now create a `server.js` file, and add the following code:

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

The `connect` variable is a function that returns a new Connect application. So, our next step is to create that app:

```
var app = connect();
```

You don't need to create an `app` variable for most of your applications. The functions involved in creating an application (`connect()` and `use()`) are chainable:

```
connect()
    .use(/* middleware */)
    .use(/* middleware */)
    .listen(3000);
```

The `use()` function adds a layer of middleware to the application, and the `listen()` function tells our application to begin accepting connections on the specified port (`3000` in this example).

Let's start with something simple: logging. The code for a Connect application that uses only the logging middleware is rather simple:

```
connect()
    .use(connect.logger())
    .listen(3000);
```

By default, Node parses very little of the incoming request.

Add that code to your file, and start the server by running `node server.js`. Navigate to any path in your browser, and ignore the "Cannot GET ..." results. We're not interested in what the server sent back to the browser; we're interested in the server's log. Look at the terminal, and you'll see the log of your requests. Be sure to check out the logger documentation for information on its other features and customization.

That was a filter; now let's look at a provider. The simplest provider is the static provider; it serves static files from a specified folder. Here's its syntax:

```
1 .use(connect.static(__dirname + "/public")
```

You can probably guess the purpose of Node's `__dirname` variable: it's the path to the current directory. This middleware statically serves anything from a `public` folder in the current directory. So, create `public/page.html` and add an `<h1>` element. Restart the server (`node server.js`), and navigate to `localhost:3000/page.html` in your browser. You should `page.html` rendered in the browser.

Let's now take a quick look at some of Connect's other middleware options.

Parsing Request Bodies

By default, Node parses very little of the incoming request, but you can incorporate several different filters to parse the request if you need to handle more complexity. There are four filters:

- `connect.json()` parses JSON request bodies (where the `content-type` is `application/json`).
- `connect.urlencoded()` parses `x-ww-form-urlencoded` request bodies.
- `connect.multipart()` parses `multipart/form-data` request bodies.
- `connect.bodyParser()` is a shortcut for enabling all of the three above.

Using any of these filters gives you the ability to access your parsed body via

`request.body` (we'll talk about how to get that `request` object soon).

I think these filters are a good example of how to fine-grain your control with Connect. You can use very little processing in order to streamline your application.

Parsing Cookies and Sessions

Cookies and sessions are an important part of any web application, and there are several pieces of middleware that help manage them. The `connect.cookieParser()` parses cookies for you, and you can retrieve the cookies and their values, via the `request.cookies` object. This is more useful if you add the `connect.session()` filter to your app. This filter requires the cookie parser to already be in place. Here's a small example:

```
connect()
    .use(connect.cookieParser())
    .use(connect.session({ secret: 'some secret text', cookie: { m
    .use(function(req, res) {
        var sess = req.session,
            url = req.url.split("/");

    if (url[1] == "name" && url[2]) {
        sess.name = url[2];
        res.end("name saved: " + url[2]);
    } else if (sess.name) {
        res.write("session-stored name: " + sess.name);
        res.end("stored for another: " + (sess.cookie.maxAge / 100
    } else {
        res.end("no stored name; go to /name/{name} to save a name
    }
}).listen(3000);
```

> Every middleware function you write needs to either pass the request to the `next` layer or respond to the request.

After the `cookieParser`, we include the `session` filter and pass it a two options:

- The `secret` creates a signed cookie which keeps track of the session.
- The `cookie.maxAge` defines its life-span in milliseconds; the `30000` in this code is `30` seconds.

In the final `use()` call, we pass a function that responds to the request. We use two properties from the `request` object: `req.session` for session data, and `req.url` for the request URL.

If the application receives a request for `/name/some_name`, then it stores the value `some_name` in `req.session.name`. Anything stored within a session can be retrieved in subsequent requests for the length of our session. Any requests made for `/name/other` replaces the session variable, and any requests to other URLs output the session variable's value and the time left for the session.

So, you can navigate to `localhost:3000/name/your_name`, and then go to `localhost:3000` to see `your_name`. Refresh the page a few times and watch the seconds count down. When the session expires, you'll see the default "no stored name" message.

I mentioned that the `cookieParser` filter must come before `session`.

> The order of inclusion is important with middleware because the request is passed, in order, from layer to layer.

Because `session` needs the parsed cookie data, the request must go through `cookieParser` before `session`.

I could explain every other built–in piece of middleware, but I'll just mention a few more before we write our own code to interface with Connect.

- compress: Gzip compression middleware
- basicAuth: basic http authentication
- directory: directory listing middleware
- errorHandler: flexible error handler

Writing Your Own Middleware

You just learned how to write your own code with Connect. Here's the basic syntax once again:

```
.use(function (req, res, next) {

})
```

The function's three parameters are important; they provide access to the outside world. The `req` parameter is, of course, the request object, and `res` is the response. The third parameter, `next`, is the key to making functions that work well in the middleware stack. It's a function that passes the request to the next middleware in the stack. See this example:

```
connect()
    .use(function (req, res, next) {
        if (req.method === 'POST') {
            res.end("This is a POST request");
        } else {
            next();
        }
    })
    .use(function (req, res) {
        res.end("This is not a POST request (probably a GET reques
    }).listen(3000);
```

This code uses two middleware functions. The first function checks the request method to see if it's a POST request. If it is, it responds by saying so. Otherwise, we call `next()` and pass the request to the next function, which responds no matter what. Use `curl` to test both layers in the terminal:

```
$ curl http://localhost:3000
This is not a POST request (probably a GET request)

$ curl -X POST http://localhost:3000
This is a POST request
```

If you don't like the terminal, try this useful Chrome plugin.

It's important to remember that every middleware function you write needs to either pass the request to the `next` layer or respond to the request. If your function branches (via if statements or other conditionals), you must ensure that every branch passes the request or responds to it. If your app hangs in the browser, it's probably because you forgot to call `next()` at some point.

Now, what about those `request` and `response` parameters? These are the very same request and response objects you receive when using a "raw" Node server:

```
require("http").createServer(function (req, res) {
    // ...
}).listen(3000);
```

If you haven't used Node's server API before, let me show you what you can do with it.

The Request Object

The `request` object is actually an `http.IncomingMessage` object, and its important properties are listed below::

- `req.method` tells you which HTTP method was used.
- `req.url` tells you which URL was requested.
- `req.headers` is an object with the header names and values.
- `req.query` is an object with any data in a query string (to parse that, you'll need

the `connect.query()` middleware in place).

- `req.body` is an object of the form data (you'll need some body parsing middleware in place).
- `req.cookies` is an object of the cookie data (requires cookie parsing).
- `req.session` is an object of the session data (again, you'll need cookie parsing and session middleware in place)

You can see all of this at work with the following code:

```
connect()
    .use(connect.query()) // gives us req.query
    .use(connect.bodyParser())  // gives us req.body
    .use(connect.cookieParser()) // for session
    .use(connect.session({ secret: "asdf" }))     // gives us req.
    .use(function (req, res) {
        res.write("req.url: " + req.url + "\n\n");
        res.write("req.method: " + req.method + "\n\n");
        res.write("req.headers: " + JSON.stringify(req.headers) +
        res.write("req.query: " + JSON.stringify(req.query) + "\n\
        res.write("req.body: " + JSON.stringify(req.body) + "\n\n"
        res.write("req.cookies: " + JSON.stringify(req.cookies) +
        res.write("req.session: " + JSON.stringify(req.session));
        res.end();
    }).listen(3000);
```

To see something for each one of these values, you need to post some data to a URL with a query string. The following should be enough:

```
curl -X POST -d "name=YourName" "http://localhost:3000/some/url?som
```

With those seven properties, you can manage just about any request you'll receive. I don't think trailers are used often (I've never seen them in my experience), but you can use `req.trailers` if you expect them in your requests (trailers are just like headers, but after the body).

So, what about your response?

The Response Object

The raw response object doesn't provide the luxuries that libraries (like Express)

gives you. For example, you can't respond with a simple render call to a premade template—at least, not by default. Very little is assumed in the response, so you need to fill in all the little details.

We'll start with the status code and the response headers. You can set these all at once using the `writeHead()` method. Here's an example from the Node docs:

```
var body = 'hello world';
response.writeHead(200, {
    'Content-Length': body.length,
    'Content-Type': 'text/plain'
});
```

If you need to individually set headers, you can use the `setHeader()` method:

```
connect()
    .use(function (req, res) {
        var accept = req.headers.accept.split(","),
            body, type;
            console.log(accept);
        if (accept.indexOf("application/json") &gt; -1) {
            type = "application/json";
            body = JSON.stringify({ message: "hello" });
        } else if (accept.indexOf("text/html") &gt; -1) {
            type = "text/html";
            body = "<h1> Hello! </h1>";
        } else {
            type = "text/plain";
            body = "hello!";
        }
        res.statusCode = 200;
        res.setHeader("Content-Type", type);
        res.end(body);
    }).listen(3000);
```

Add this code to a file, start up the server and request it from the browser. You got HTML! Now run:

```
curl http://localhost:3000
```

And you'll receive plain text. For JSON, try this:

```
curl -H "accept:application/json" http://localhost:3000
```

All from the same URL!

Use `res.getHeader(name)` if you need to know what headers have already been set.

Tou can also use `res.removeHeader(name)` to remove a header.

Of course, a response is useless without a body. As you've seen throughout this tutorial, you can write chunks of data to the body with the `res.write()` method. This accepts a string or buffer object as an argument. If it's a string, the second parameter is the encoding type (it defaults to `utf8`).

The `res.end()` method closes the body, but you can pass data to it to write to the response stream. This is useful in situations where you only need to output a single line.

Third-Party Middleware

It's somewhat difficult to respond with larger HTML bodies in plain old Node and Connect. This is a good place to throw third–party middleware into the mix. You can find a list of third–party middleware on the Connect Github wiki. As an example, we're going to use the connect–jade package, which allows us to render jade views.

First, install `connect-jade`:

```
npm install connect-jade
```

Next, require and add it as middleware. You'll want to set a few default values:

```
var connect = require("connect"),
    connectJade = require("connect-jade");

connect()
    .use(connectJade({
        root: __dirname + "/views",
        defaults: {
            title: "MyApp"
        }
    }))
    .use(function (req, res) {
        res.render("index", { heading: "Welcome to My App" });
    }).listen(3000);
```

Set the root as the directory that contains the view files. You can also set `defaults`; these are variables that are available inside every view, unless we override them

later when calling `render()`.

The final function in this code makes a call to `res.render()`. This method is provided by the `connect-jade` package.

> The first argument it accepts is the name of the view to render.

It's the path to the view, *sans* the path that we defined when adding the middleware, *sans* the jade file extension. For this code, we need a `views/index.jade` template to render. We'll keep it simple:

```
1 html
2   head
3     title= title
4   body
5     h1= heading
```

If you're not familiar with jade, we indent tag names to create an HTML structure. The equal sign retrieves the value of a JavaScript variable. Those variables come from the `defaults` we set up, plus the (optional) second parameter object passed to `res.render()`.

There are many other third-party middlewares, but they work similar to each other. You install them via npm, require them and put them into action.

Modules as Middleware

If you dig into how Connect works, you'll find that each layer is actually a Node module—a very intelligent design. If you use Connect for sizeable applications, it would be ideal to write your code in Node module format. You might have an `app.js` file like this:

```
1 // app.js
2 module.exports = function (req, res, next) {
3     res.end("this comes from a module");
4 };
```

And in your `server.js`:

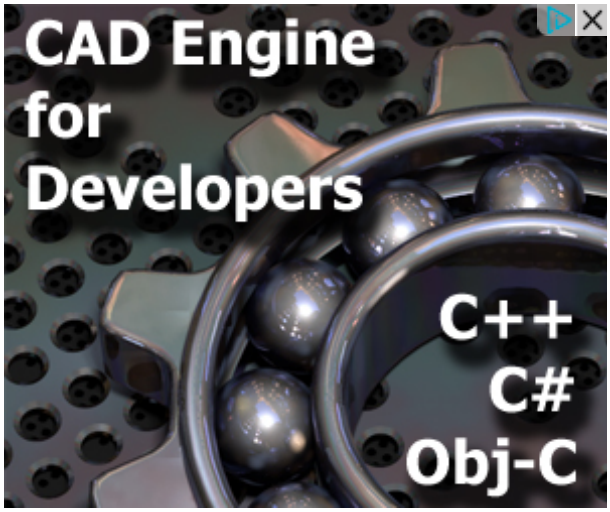
```
1 var connect = require("connect"),
```

```
    app = require("./app");

connect()
    .use(app)
    .listen(3000);
```

Conclusion

If you want a beginner–friendly library that makes it easy to build large web apps, then Connect isn't your solution. Connect is meant to be a thin layer on top of the raw Node API that gives you complete control over your server application. If you want a bit more, I recommend Express (by the same folks, incidentally). Otherwise, Connect is a fantastic, extensible library for Node web applications.



Tags: connectnode

By Andrew Burgess

I'm a Canadian web developer, a staff writer at Nettuts+, and the author of the Rockable Press eBook Getting Good With Git. I prefer JavaScript and Ruby. Soon, I'll be doing some writing on my personal site; of course, you can follow me on Twitter.

Note: Want to add some source code? Type <pre><code> before it and </code></pre> after it. Find out more