Write Your Own Express.js From Scratch

Over the last 8 years, Express has become the de facto web server framework for Node.js. I wouldn't be surprised if the MEAN stack had something to do with it, because when I first evaluated Express in mid 2012, I compared it against its then biggest competitors like Geddy and Tower. Geddy and Tower have since faded into dim memory, but Express has exploded in popularity.

I immediately loved Express' simplicity and elegance: middleware is the one fundamental concept for adding logic to Express. Plugins are just middleware, so dropping in a plugin is always as simple as `app.use(plugin)`. Once I understood middleware and the APIs available for requests and responses, apps mostly wrote themselves.

I find the best way to really master a framework is to write your own. Really diving into the internals of a framework gives you a deeper understanding than years of reading the documentation. I love open source because documentation often stretches the truth or omits important details, but the source code never lies.

In this article, I'll walk you through building a simplified Express clone called Espresso (you can find the code on GitHub at https://github.com/vkarpov15/espresso-example) in 4 steps. First, you'll see how to implement a rudimentary middleware pipeline. Then you'll see how Express implements routing. Next, you'll see how Express' recursive routing structure is implemented with a separate router class. Finally, you'll see some limitations of using Express with async/await and how Express might improve its support for async/await.

Each step has an associated GitHub commit in the espresso-sample GitHub repo. I recommend looking at the GitHub diff for each step before and after reading each step, so you can see the full scope of the changes in each step.

Step 1: Getting Started With Middleware (Diff)

First, let's implement rudimentary support for Express middleware, without any routing. Modulo error handling middleware, all Express middleware is a function that takes 3 parameters: the request `req`, the response `res`, and the `next()` function. Middleware is executed sequentially on each request, and `next()` is how you tell Express to kick off the next middleware. If a middleware doesn't call `next()`, it should make sure to call `res.end()` to send the HTTP response to the client.

```
app.use(function myMiddleware(req, res, next) {
  res.end('Hello, world');
  next();
});
```

Everything is middleware, even routes are just sugar for pushing middleware onto the stack. In the Express internals, the `use()` function converts the middleware function into a 'Layer', which is just an object wrapper around the middleware function. The use() function then pushes the layer onto the 'stack', which is just an array of layers. With that in mind, let's create the Espresso class and create a `use()` function:

```
const http = require('http');

class Espresso {
  constructor() {
    this._stack = [];
  }

  use(middleware) {
    if (typeof middleware !== 'function') {
      throw new Error('Middleware must be a function!');
    }
    this._stack.push(middleware);
  }
}
```

Espresso also needs to be able to create an HTTP server that executes all middleware on every request. Express has the `listen()` function for this, which just wraps Node.js' built-in `http.createServer()` function. Here's how Espresso implements this `listen()` function:

```
listen(port, callback) {
  const handler = (req, res) => {
    // `this.handle()` executes all middleware defined on this Espresso
    // app instance, will implement this method next!
    this.handle(req, res, err => {
      if (err) {
        res.writeHead(500);
        res.end('Internal Server Error');
      }
    });
  };
  return http.createServer(handler).listen({ port }, callback);
}
```

The `handle()` method used above is responsible for executing every middleware. Internally, Express routers have a similar `handle()` method that executes middleware that matches a request. For this first step, Espresso won't implement routing, it will just execute all middleware

that was passed in to `app.use()`. Here's how you can implement the `handle()` method by calling `next()` recursively.

```
handle(req, res, callback) {
  let idx = 0;

  const next = (err) => {
    // If an error occurred, bypass the rest of the pipeline. In Express,
    // you would still need to look for error handling middleware, but
    // this example does not support that.
    if (err != null) {
      return setImmediate(() => callback(err));
    }
    if (idx >= this._stack.length) {
      return setImmediate(() => callback());
    }

    // Not the same as an internal Express layer, which is an object
    // wrapper around a middleware function. Using the same nomenclature
    // for consistency.
    const layer = this._stack[idx++];
    setImmediate(() => {
      try {
        // Execute the layer and rely on it to call `next()`
        layer(req, res, next);
      } catch(error) {
        next(error);
      }
    });
  };

  next();
}
```

You can find the whole Espresso class on GitHub. You can also find associated mocha tests here. Let's see the Espresso class in action with a simple "Hello, World" example in a mocha test using axios as an HTTP client.

```
const Espresso = require('../lib/step1');
const assert = require('assert');
const axios = require('axios');

describe('Espresso', function() {
  let server;
  afterEach(() => server && server.close());

  it('works in the basic Hello, World case', async function() {
    const app = new Espresso();
    // Add some very simple middleware
    app.use((req, res, next) => {
      res.end('Hello, world!');
      next();
    });
    server = app.listen(3000);

    const res = await axios.get('http://localhost:3000');
    assert.equal(res.data, 'Hello, world!');
  });
});
```

To show that this is actually sufficient to reproduce the basics of Express middleware, let's plug in an actual Express plugin into Espresso and see it in action. The CORS module is an Express plugin that sets the AccessControlAllowOrigin header for enabling cross-origin resource sharing. Here's a test that shows using `cors()` with Espresso works as intended with no additional changes.

```
it('works with real Express middleware (CORS)', async function() {
  const app = new Espresso();
  app.use(cors());
  app.use((req, res, next) => {
    res.end('Hello with CORS');
    next();
  });
  server = app.listen(3000);

  const res = await axios.get('http://localhost:3000');

  // This is the header that `cors()` should set
  assert.equal(res.headers['access-control-allow-origin'], '*');
  assert.equal(res.data, 'Hello with CORS');
});
```

That's it for producing a simplified middleware step. Next up is decorating `req` and `res` with Express-specific helper functions.

Step 2: Routing and Layers (Diff)

Express is a little more sophisticated than just executing all middleware in sequence. Routing enables you to only execute certain middleware on requests that meet certain criteria, like matching a given URL or a given HTTP method (GET vs POST vs PUT).

The fundamental routing construct in Express is called a 'layer'. The middleware pipeline in Express is just an array of layers. The layer class has the distinct responsibility of determining whether this middleware should execute on a given request via the `match()` method. Express then skips the middleware if `match()` returns false.

Most of the time you only want to match layers based on two properties: the HTTP method and the URL. This way, you can have distinct handlers for `GET /hello/world`, `POST /hello/world`, and `GET /goodbye/everyone`. With that in mind, let's create a `Layer` class that will enable Espresso to match against `req.method` and `req.url`.

In order to do implement the `match()` method, we'll need to include the `path-to-regexp` npm module, which converts paths like `/hello/:name` to regular expressions. The `path-to-regexp` module makes it easier to pull out URL parameters, this is the module that Express uses to populate `req.params`. So if you have the route `/hello/:name` and your API gets a request for `/hello/world`, `req.params.name` will equal 'world'.

```
const pathToRegexp = require('path-to-regexp');

class Layer {
  constructor(method, url, middleware) {
    this.method = method;
    if (url != null) {
      // For example, `/hello/:id` -> `[{ name: 'id' }]`
      this.keys = [];
      // `this.url` is a regexp, `this.keys` contains the params.
      this.url = pathToRegexp(url, this.keys);
    }
    this.middleware = middleware;
  }
}
```

In Express, parameters are defined on the individual layer, not on the request, so `req.params` might be different for different layers in the middleware pipeline. For example, suppose you hit `GET http://localhost:3000/hello` on the below Express API:

```
const express = require('express');

const app = express();

// Suppose you visit `GET http://localhost:3000/hello

app.use(function(req, res, next) {
  // "Hello, {}", no params here!
  console.log(`Hello, ${JSON.stringify(req.params)}`);
  next();
});

app.get('/:id', function(req, res, next) {
  // "Hello, world" as you might expect
  console.log(`Hello, ${req.params.id}`);
  next();
});

app.get('/:name', function(req, res) {
  // "Hello, world" as well, despite having a different param name
  console.log(`Hello, ${req.params.name}`);
  res.end('N/A');
});

app.listen(3000);
```

Oddly enough, the layer's `match()` method also sets the `params` that get attached to `req` in Express. Espresso's simplified `match()` method is shown below:

```
match(method, url) {
  // Matching method is easy: if specified, check to see if it matches
  if (this.method != null && this.method !== method) {
    return false;
  }
  // Matching URL is harder: need to check if the regexp matches, and
  // then pull out the URL params.
  if (this.url != null) {
    const match = this.url.exec(url);
    // If the URL doesn't match, this layer doesn't match
    if (match == null) return false;

    const params = this.params = {};
    for (let i = 1; i < match.length; ++i) {
      // First element of the `match` array is always the whole URL
      params[this.keys[i - 1].name] = decodeURIComponent(match[i]);
    }
  }

  return true;
}
```

Now that Espresso has a layer class, all that remains is to modify the Espresso class itself to use layers. First, let's modify `use()` to add layers, and add `route()` and `get()` helpers to make adding routes easier.

```
use(middleware) {
  if (typeof middleware !== 'function') {
    throw new Error('Middleware must be a function!');
  }
  this._stack.push(new Layer(null, null, middleware));
}

route(method, url, handler) {
  this._stack.push(new Layer(method, url, handler));
  return this;
}

get(url, handler) {
  return this.route('GET', url, handler);
}
```

Finally, let's tweak the top-level Espresso `handle()` method to skip layers that don't `match()` the given request, and decorate `req.params`.

```
handle(req, res, callback) {
  let idx = 0;

  const next = (err) => {
    // If an error occurred, bypass the rest of the pipeline. In Express,
    // you would still need to look for error handling middleware, but
    // this example does not support that.
    if (err != null) {
      return setImmediate(() => callback(err));
    }
    if (idx >= this._stack.length) {
      return setImmediate(() => callback());
    }

    let layer = this._stack[idx++];
    // Find the next layer that matches
    while (idx <= this._stack.length && !layer.match(req.method, req.url)
      layer = this._stack[idx++];
    }
    // If no more layers, we're done.
    if (layer == null) {
      return setImmediate(() => callback());
    }

    // Decorate `req` with the layer's `params`. Make sure to do it
    // **outside** `setImmediate()` because of concurrency concerns.
    req.params = Object.assign({}, layer.params);

    setImmediate(() => {
      try {
        // Execute the layer and rely on it to call `next()`
        layer.middleware(req, res, next);
      } catch(error) {
        next(error);
      }
    });
  };

  next();
}
```

Here's a test showing the new `app.get()` helper in action, creating 2 layers with URL params. You might wonder why it is ok to not call `next()` in the below route handlers. That's because, once you call `res.end()`, Node.js finalizes the HTTP response and sends it back to the client, so no other middleware needs to execute.

```
it('basic routing', async function() {
  const app = new Espresso();
  app.get('/hello/:id', (req, res) => res.end(`Hello, ${req.params.id}`))
  app.get('/bye/:id', (req, res) => res.end(`Bye, ${req.params.id}`));
  server = app.listen(3000);

  let res = await axios.get('http://localhost:3000/hello/world');
  assert.equal(res.data, 'Hello, world');

  res = await axios.get('http://localhost:3000/bye/everyone');
  assert.equal(res.data, 'Goodbye, everyone');
});
```

Step 3: Routers (Diff)

Routers were one of the exciting new features in Express 4.0.0, replacing the cumbersome sub-app concept from Express 3. Routers are essentially middleware that contains other middleware, so you can create a sub-router with a distinct middleware chain for a separate set of URLs.

In order to reduce code duplication between routers and apps, this example will create a `MiddlewarePipeline` base class. Express does not do this currently: `Router.handle()` is completely separate from `Application.handle()`, which seems like a candidate for future refactoring. The `MiddlewarePipeline` class will contain 4 methods:

```
class MiddlewarePipeline {
  constructor() {
    this._stack = [];
  }

  use(url, middleware) { /* ... */ }

  route(method, url, handler) {
    this._stack.push(new Layer(method, url, handler));
    return this;
  }

  get(url, handler) {
    return this.route('GET', url, handler);
  }

  handle(req, res, callback) {
    // ...
  }
}
```

Now, instead of defining the above methods in the `Espresso` class, `Espresso` will inherit from `MiddlewarePipelin`. In order to facilitate routers, there are a couple necessary changes to `use()` and `handle()`, as well as the layer class's `match()` method. First, `use()` needs to support a url parameter. The intuition of how `use(url, fn)` works is easy to demonstrate in code, below is the test case for using nested routers:

```
it('using router', async function() {
  const app = new Espresso();

  const nested = Espresso.Router();
  nested.get('/own', (req, res) => res.end('Wrote your own Express!'));

  const router = Espresso.Router();
  router.use('/your', nested);
  app.use('/write', router);

  server = app.listen(3000);
  let res = await axios.get('http://localhost:3000/write/your/own');
  assert.equal(res.data, 'Wrote your own Express!');
});
```

The `/write/your/own` endpoint ends up going through 3 different middleware chains, each nested in the other like Russian stacking dolls. The `app.use('/write', router)` line gives control to router, which is just a separate middleware. The tricky part with router above is with the layer class' `match()` function. By default, req.url will always be `/write/your/own`, how do `router` and `nested` know to match `/your` and `/own`?

Express actually changes `req.url` internally for routing purposes. This is why Express has req.originalUrl, so it can still expose the original URL while rewriting req.url. The below script shows Express changing `req.url`:

```
const assert = require('assert');
const axios = require('axios');
const express = require('express');

const app = express();
const nested = express.Router();
nested.get('/own', (req, res) => res.end(req.url));
const router = express.Router();
router.use('/your', nested);
app.use('/write', router);

(async function() {
  await app.listen(3000);
  let res = await axios.get('http://localhost:3000/write/your/own');
  // Prints "/own", **not** "/write/your/own"
  console.log(res.data);
})();
```

To rewrite `req.url` correctly, you'll need to first tweak the `MiddlewarePipeline` class's `use()` method to use the `end` option for pathToRegExp, which creates a regular expression that matches a string which starts with `/hello` rather than is exactly `/hello`.

```
use(url, middleware) {
  if (arguments.length === 1) {
    middleware = url;
    url = null;
  }
  if (typeof middleware !== 'function') {
    throw new Error('Middleware must be a function!');
  }
  // Explicitly use `end: false` so `/hello/world` matches `/hello`
  this._stack.push(new Layer(null, url, middleware, { end: false }));
}
```

What does this mean for the layer class's `match()` function? It needs to expose the part of the URL that matched the regular expression as `layer.path`:

```
match(method, url) {
  // Matching method is easy: if specified, check to see if it matches
  if (this.method != null && this.method !== method) {
    return false;
  }
  // Matching URL is harder: need to check if the regexp matches, and
  // then pull out the URL params.
  if (this.url != null) {
    const match = this.url.exec(url);
    // If the URL doesn't match, this layer doesn't match
    if (match == null) {
      return false;
    }
    // Store the part of the URL that matched, so `this.path` will
    // contain `/hello` if we do `app.use('/hello', fn)` and
    // get `/hello/world`
    this.path = match[0];

    // Copy over params
    const params = this.params = {};
    for (let i = 1; i < match.length; ++i) {
      // First element of the `match` array is always the part of
      // the URL that matched.
      params[this.keys[i - 1].name] = decodeURIComponent(match[i]);
    }
  }

  return true;
}
```

And, finally, the `MiddlewarePipeline` class's `handle()` method needs to rewrite `req.url` to omit `layer.path`, so the nested routers don't see the leading part of the URL.

```
const originalUrl = req.url;
req.path = layer.path;
req.url = req.url.substr(req.path.length);

try {
  // Switch to using `setImmediate()` in the callback, because `req.url`
  // needs to be reset synchronously before calling `next()`
  layer.middleware(req, res, err => setImmediate(() => next(err)));
  req.url = originalUrl;
} catch(error) {
  req.url = originalUrl;
  next(error);
}
```

And that's all you need to make recursive routers work. Do `git checkout step3 && npm test` and you should see the tests succeed.

Step 4: Async/Await Integration (Diff)

Express has some serious limitations with using async/await. Specifically, if your async function throws an error, Express will never respond to the request.

```
const app = require('express')();

// This does **not** currently work with Express, the request will just
// hang forever.
app.get('/', async function(req, res) {
  throw new Error('Oops!');
});
```

There have been pull requests to fix this issue, but they were never merged because of some valid concerns regarding potential double `next()` calls. First, let's see how to replicate this PR in Espresso. In theory, integrating with async/await is easy, you just need to check if your layer function returned a promise and `.catch()` any errors. Here's the diff on GitHub.

```
// Modify how `MiddlewarePipeline.handle()` calls the layer middleware
try {
  // Switch to using `setImmediate()` in the callback, because
  // `req.url` needs to be reset synchronously before calling `next()`
  const retVal = layer.middleware(req, res, err => {
    setImmediate(() => next(err));
  });
  req.url = originalUrl;
  if (retVal instanceof Promise) {
    retVal.catch(error => next(error));
  }
} catch(error) {
  req.url = originalUrl;
  next(error);
}
```

With this simple change, the below test now passes:

```
it('using async/await', async function() {
  const app = new Espresso();

  // This does **not** currently work with Express, the request will just
  // hang forever.
  app.get('/', async function(req, res) {
    throw new Error('woops!');
  });

  server = app.listen(3000);

  let threw = false;
  try {
    await axios.get('http://localhost:3000/');
  } catch (error) {
    assert.equal(error.response.status, 500);
    assert.equal(error.response.data, 'Internal Server Error');
    threw = true;
  }
  assert.ok(threw);
});
```

So what's the problem with this change? The argument against it is the possibility of double `next()` calls. For example, the below code will not report the error that was thrown.

```
it('using async/await badly', async function() {
  const app = new Espresso();

  app.use(async function(req, res, next) {
    next();

    // Wait for next middleware to finish sending the response
    await new Promise(resolve => setTimeout(() => resolve(), 100));

    // Double `next()`, this error won't get reported!
    throw new Error('woops!');
  });

  app.use(function(req, res, next) {
    res.end('done');
    next();
  });

  server = app.listen(3000);

  const res = await axios.get('http://localhost:3000/');
  assert.equal(res.data, 'done');
});
```

Double `next()` calls are possible even with synchronous errors, but they're easier to run into with async/await. If you're willing to accept that risk, there's libraries like `@awaitjs/express` that let you use async/await with Express.

Moving On

I hope you found this tutorial as enlightening and enjoyable as I did, Express is a beautiful framework and digging through its internals is a pleasure. There are a few concepts in the Express code base that this article didn't touch upon, like template engines and response helpers, but those concepts are far simpler than routing. Hopefully this guided tour of the Express internals will enable you to dig into the Express internals next time you run into something unexpected.