The Express Application Framework

In Chapter 5, I covered a small subset of the functionality you need to implement a Node web application. The task to create a Node web app is daunting at best. That's why an application framework like Express has become so popular: it provides most of the functionality with minimal effort.

Express is almost ubiquitous in the Node world, so you'll need to become familiar with it. We're going to look at the most bare-bones Express application we can in this chapter, but you will need additional training once you're finished.

EXPRESS NOW PART OF NODE.JS FOUNDATION

Express has had a rocky start but is now part of the Node.js Foundation. Future development should be more consistent and its support more reliable.

Express provides good documentation, including how to start an application. We’ll follow the steps the documentation outlines and then expand on the basic application. To start, create a new subdirectory for the application and name it whatever you want. Use npm to create a *package.json* file, using *app.js* as the entry point for the application. Lastly, install Express, saving it to your dependencies in the *package.json* file by using the following command:

```
npm install express --save
```

The Express documentation contains a minimal Hello World Express application, typed into the *app.js* file:

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

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

The `app.get()` function handles all GET web requests, passing in the request and response objects we’re familiar with from our work in earlier chapters. By convention, Express applications use the abbreviated forms of `req` and `res`. These objects have the same functionality as the default request and response objects, with the addition of new functionality provided by Express. For instance, you can use `res.write()` and `res.end()` to respond to the web request, which we’ve used in past chapters, but you can also use `res.send()`, an Express enhancement, to do the same in one line.

Instead of manually creating the application, we can also use the Express application generator to generate the application skeleton. We’ll use that next, as it provides a more detailed and comprehensive Express application.

First, install the Express application generator globally:

```
sudo npm install express-generator -g
```

Next, run the application with the name you want to call your application. For demonstration purposes, I’ll use `bookapp`:

```
express bookapp
```

The Express application generator creates the necessary subdirectories. Change into the *bookapp* subdirectory and install the dependencies:

```
npm install
```

That’s it, you’ve created your first skeleton Express application. You can run it using the following if you’re using an OS X or Linux environment:

```
DEBUG=bookapp:* npm start
```

Run the following if you’re in a Windows Command window:

```
set DEBUG=bookapp:* & npm start
```

You could also start the application with just `npm start` and forgo the debugging.

The application is started and listens for requests on the default Express port of 3000. Accessing the application via the Web returns a simple web page with the “Welcome to Express” greeting.

Several subdirectories and files are generated by the application:

```
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.jade
    ├── index.jade
    └── layout.jade
```

We’ll get into more detail about many of the components, but the public-facing static files are located in the *public* subdirectory. As you’ll note, the graphics and CSS files are placed in this location. The dynamic content template files are located in *views*. The *routes* subdirectory contains the web endpoint applications that listen for web requests and render web pages.

JADE IS NOW PUG

Because of trademark violation, the creators of Jade can no longer use "Jade" as the name for the template engine used with Express and other applications. However, the process of converting Jade to Pug is still ongoing. At the time this went to production, the Express Generator still generates Jade, but attempting to install Jade as a dependency generates the following error:

```
Jade has been renamed to pug, please install the latest
version of pug instead of jade
```

The web site for Pug is still named Jade, but the documentation and functionality remains the same. Hopefully this will all resolve itself sooner, rather than later.

The *www* file in the *bin* subdirectory is a startup script for the application. It's a Node file that's converted into a command-line application. When you look in the generated *package.json* file, you'll see it listed as the application's start script.

```
{
  "name": "bookapp",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "start": "node ./bin/www"
  },
  "dependencies": {
    "body-parser": "~1.13.2",
    "cookie-parser": "~1.3.5",
    "debug": "~2.2.0",
    "express": "~4.13.1",
    "jade": "~1.11.0",
    "morgan": "~1.6.1",
    "serve-favicon": "~2.3.0"
  }
}
```

You install other scripts to test, restart, or otherwise control your application in the *bin* subdirectory.

To begin a more in-depth view of the application we'll look at the application's entry point, the *app.js* file.

When you open the *app.js* file, you're going to see considerably more code than the simple application we looked at earlier. There are several more modules imported, most of which provide the *middleware* support we'd expect for a web-facing application. The imported modules also include application-specific imports, given under the *routes* subdirectory:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');

var routes = require('./routes/index');
var users = require('./routes/users');

var app = express();
```

The modules and their purposes are:

`express`

The Express application

`path`

Node core module for working with file paths

`serve-favicon`

Middleware to serve the *favicon.ico* file from a given path or buffer

`morgon`

An HTTP request logger

`cookie-parser`

Parses cookie header and populates `req.cookies`

`body-parser`

Provides four different types of request body parsers (but does not handle multipart bodies)

Each of the middleware modules works with a vanilla HTTP server as well as Express.

WHAT IS MIDDLEWARE?

Middleware is the intermediary between the system/operating system/database and the application. With Express, the middleware is part of a chain of applications, each of which does a certain function related to an HTTP request — either processing it, or performing some manipulation on the request for future middleware applications. The set of middleware that works with Express is quite comprehensive.

The next section of code in *app.js* mounts the middleware (makes it available in the application) at a given path via the `app.use()` function. The order in which the middleware is mounted is important so if you add additional middleware functionality, be sure that it is in relation to other middleware according to the developer recommendations.

The code snippet also includes code that defines the view engine setup, which I'll get to in a moment.

```
// view engine setup
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// uncomment after placing your favicon in /public
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

The last call to `app.use()` references one of the few built-in Express middleware, `express.static`, which is used to handle all static files. If a web user requests an HTML, JPEG, or other static file, `express.static` processes the request. All static files are served relative to the path specified when the middleware is mounted, in this case, in the `public` subdirectory.

Returning to the `app.set()` function calls defining the views engine in the code snippet, you'll use a *template engine* that helps map the data to the delivery. One of the most popular, Jade, is integrated by default, but others such as Mustache and EJS can be used just as easily. The engine setup defines the subdirectory where the template files (*views*) are located and which view engine to use (Jade).

REMINDER: JADE IS NOW PUG

As mentioned earlier in the chapter, Jade is now Pug. Check with both the Express documentation, and Pug, in order to update examples to work with the newly named template engine

As this book went to production, I modified the generated package.json file to replace the Jade module with Pug:

```
<p>"pug": "2.0.0-alpha8",</p>
```

And in the `app.js` file, replace the `jade` reference with `pug`:

```
app.set('view engine', 'pug');
```

And the application seemed to work without problem.

In the *views* subdirectory, you'll find three files: *error.jade*, *index.jade*, and *layout.jade*. These will get you started, though you'll need to provide much more when you start integrating data into the application. The content for the generated *index.jade* file is given here:

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

The line that reads `extends layout` incorporates the Jade syntax from the *layout.jade* file. You'll recognize the HTML header (`h1`) and paragraph (`p`) elements. The `h1` header is assigned the value passed to the template as `title`, which is also used in the paragraph element. How these values get rendered in the template requires us to return to the *app.js* file for the next bit of code:

```
app.use('/', routes);
app.use('/users', users);
```

These are the application-specific endpoints, which is the functionality that responds to client requests. The top-level request (`'/'`) is satisfied by the *index.js* file in the *routes* subdirectory, the users, by the *users.js* file.

In the *index.js* file, we're introduced to the Express *router*, which provides the response-handling functionality. As the Express documentation notes, the router behavior fits the following pattern:

```
app.METHOD(PATH, HANDLER)
```

The method is the HTTP method, and Express supports several, including the familiar `get`, `post`, `put`, and `delete`, as well as the possibly less familiar `merge`, `search`, `head`, `options`, and so on. That path is the web path, and the handler is the function that processes the request. In *index.js*, the method is `get`, the path is the application root, and the handler is a callback function passing request and response:

```
var express = require('express');
var router = express.Router();

/* GET home page. */
```

```
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

The data (local variables) and view meet in the `res.render()` function call. The view used is the *index.jade* file we looked at earlier, and you can see that the value for the `title` attribute in the template is passed as data to the function. In your copy, try changing "Express" to whatever you'd like and reloading the page to see the modification.

The rest of the *app.js* file is error handling, and I'll leave that for you to explore on your own. This is a quick and very abbreviated introduction to Express, but hopefully even with this simple example, you're getting a feel for the structure of an Express application.

INCORPORATING DATA

I’ll toot my horn and recommend my book, the *JavaScript Cookbook* (O’Reilly, 2010) if you want to learn more about incorporating data into an Express application. Chapter 14 demonstrates extending an existing Express application to incorporate a MongoDB store, as well as incorporating the use of controllers, for a full model-view-controller (MVC) architecture.