- ⏱️ 6 minutes
- ✅ Completed

Moving Into the Express Lane

In an earlier lesson, you created a simple HTTP server using JavaScript and Node.js. That HTTP server, or web application, returned a simple response based upon the incoming request's URL (and HTTP method in one case). For example, a request to the URL `http://localhost:300/OK` returned a `200 OK` HTTP response status code.

Overall, this was easy to do using Node's native APIs, though the requirements were relatively straightforward. Using Node to create a web application with features commonly found in websites unfortunately requires a fair amount of boilerplate code (i.e. verbose, repetitive code). This can slow down and distract developers from working on more important tasks.

Enter Express, a popular Node.js framework for building web applications. Express aims to make common web development tasks easier to implement by reducing the amount of boilerplate code you need to write. This allows you to focus on the things that makes your web application special. At the same time, Express is, in its own words, unopinionated and minimalistic, giving you the flexibility to decide what's best for your situation.

As an introduction to Express, let's create a simple web application. Your application will return a plain text response containing "Hello from Express!" for any request to `http://localhost:8081/`.

When you finish this article, you should be able to:

- Use the `express()` function to create an Express application;
- Recall that routing is determining how an application responds to a client request to a specific URI (or path) and HTTP method combination;
- Use the Application `get()` method to define a route that handles `GET` requests;
- Use the Response object `res.send()` method to send a plain text response to a client; and
- Use the `app.listen()` method to start a server listening for HTTP connections on a specific port.

Installing Express

Before you can use Express to create a web application, you need to install it using npm. Open a terminal or command prompt window, browse to your project's folder, and initialize npm by running the following command:

```
npm init -y
```

You'll now have `package.json` and `package-lock.json` files in the root of your project. The `package.json` file keeps track of your application's dependencies—npm packages that your application needs to successfully start and run.

Run the following command to install Express 4.0:

```
npm install express@^4.0.0
```

The `package.json` file will now list Express as a dependency:

```
{
 "name": "my-project-folder-name",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
   "test": "echo \"Error: no test specified\" && exit 1"
 },
 "keywords": [],
 "author": "",
 "license": "ISC",
 "dependencies": {
   "express": "^4.17.1"
 }
}
```

At the time of this writing, the latest version of Express 4.0 is `4.17.1`. While newer minor or patch versions of Express 4.0 should work fine, newer major versions (5.0+) might not work as expected. The caret character (`^`) the precedes the version number in the `package.json` file (`^4.17.1`) instructs npm to allow versions greater than `4.17.1` and less than `5.0.0`.

Git and the `node_modules` folder

In an earlier lesson, you learned that when using `npm install` to install an npm package locally into your project, npm downloads and installs the specified package to the `node_modules` folder. Over time, as you install dependencies, the `node_modules` folder tends to grow to be very large, containing many folders and files.

If you're using Git for source control, it's important to add a `.gitignore` file to the root of your project and add the entry `node_modules/` so that the `node_modules` folder won't be tracked by Git.

As alternative to creating your own `.gitignore` file, you can use GitHub's comprehensive .gitignore file for Node.js projects.

Creating an Express application

Now you're ready to create your Express application!

Add a file named `app.js` to your project folder and open it in your code editor. Use the `require` directive to import the `express` module and assign it to a variable named `express`. The `express` variable references a function (exported by the `express` module) that you can call to create an Express application. Assign the return value from the `express` function call to a variable named `app`:

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

// Create the Express app.
const app = express();
```

The `app` variable holds a reference to an Express Application (`app`) object. You'll call methods on the `app` object as you build out your web application.

Handling requests

Next, you need to configure the routing for your application.

The process of configuring routing is determining how an application should respond to a client request to an endpoint—a specific URI (or path) and HTTP method combination. For example, when a client makes a `GET` request to your application by browsing to the URL `http://localhost:8081/`, it should return the plain text response "Hello from Express!".

Do you remember the parts of a URL? *In the URL `http://localhost:8081/`, the protocol is `http`, the domain is `localhost`, the port is `8081` (we'll see in a bit how to configure the port for your application), and the path is `/`.*

The Express Application (`app`) object contains a collection of methods for defining an application's routes:

- `get()` - to handle `GET` requests
- `post()` - to handle `POST` requests
- `put()` - to handle `PUT` requests
- `delete()` - to handle `DELETE` requests

`GET` and `POST` are two of the most commonly used HTTP methods, followed by `PUT` and `DELETE`.

See the Express documentation for a complete list of the available routing methods.

To define a route to handle `GET` requests, call the `app.get()` method passing in the route path and a route handler function:

```
app.get('/', (req, res) => {
  // TODO Send a response back to the client.
});
```

Express provides a lot of flexibility with the format of the route path. A route path can be a string, string pattern, regular expression, or an array containing any combination of those. For now, you'll just use a string, but in later articles you'll see how to use the other options.

The route handler function is called by Express whenever an incoming request matches the route. The function defines two parameters, `req` and `res`, giving you access respectively to the Request and Response objects. The Request (`req`) object is used to get information about the client request that's currently being processed. The Response (`res`) object is used to prepare a response to return to the client.
To send a plain text response to the client, call the `res.send()` method passing in the desired content:

```
app.get('/', (req, res) => {
  res.send('Hello from Express!');
});
```

Here's the code for your application so far:

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

// Create the Express app.
const app = express();

// Define routes.

app.get('/', (req, res) => {
  res.send('Hello from Express!');
});
```

Listening for HTTP connections

Great job so far! Now you need to start the server listening for HTTP connections from clients. To do that, call the `app.listen()` method passing in the desired port to use and an optional callback function:

```
const port = 8081;

app.listen(port, () => console.log(`Listening on port
${port}...`));
```

The callback function will be called when the server has started listening for connections. Logging a message to the console gives you an easy way to see when the server is ready for testing.

Here's the complete code for your application:

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

// Create the Express app.
const app = express();

// Define routes.

app.get('/', (req, res) => {
  res.send('Hello from Express!');
});

// Define a port and start listening for connections.

const port = 8081;

app.listen(port, () => console.log(`Listening on port
${port}...`));
```

Testing your application

To test your application, open a terminal or command prompt window, browse to your project's folder, and run the following command:

```
node app.js
```

If your application starts successfully, you'll see the text "Listening on port 8081..." displayed in the terminal or command prompt window. Next, open a web browser and browse to the address `http://localhost:8081/`. You should see the text "Hello from Express!" displayed in the browser.

If you see the expected text, congrats! If you don't, double check the following:

- Make sure that you started your application by running the command `node app.js`.
- Double check that the URL you entered into your browser's address bar is `http://localhost:8081/`.
- Check the terminal or command prompt window to see if an error is occurring.

What you learned

In this article, you learned

- how to use the `express()` function to create an Express application;
- that routing is determining how an application responds to a client request to a specific URI (or path) and HTTP method combination;
- how to use the Application `get()` method to define a route that handles `GET` requests;
- how to use the Response object `res.send()` method to send a plain text response to a client; and
- how to use the `app.listen()` method to start a server listening for HTTP connections on a specific port.

See also…

As you learn about Express, you'll find it helpful to explore Express' official documentation at expressjs.com.