



Express/Node introduction

In this first Express article we answer the questions "What is Node?" and "What is Express?", and give you an overview of what makes the Express web framework special. We'll outline the main features, and show you some of the main building blocks of an Express application (although at this point you won't yet have a development environment in which to test it).

Prerequisites: Basic computer literacy. A general understanding of [server-side website programming](#), and in particular the mechanics of [client-server interactions in websites](#).

Objective: To gain familiarity with what Express is and how it fits in with Node, what functionality it provides, and the main building blocks of an Express application.

Introducing Node

[Node](#) (or more formally *Node.js*) is an open-source, cross-platform runtime environment that allows developers to create all kinds of server-side tools and applications in [JavaScript](#). The runtime is intended for use outside of a browser context (i.e. running directly on a computer or server OS). As such, the environment omits browser-specific JavaScript APIs and adds support for more traditional OS APIs including HTTP and file system libraries.

From a web server development perspective Node has a number of benefits:

- Great performance! Node was designed to optimize throughput and scalability in web applications and is a good solution for many common web-development problems (e.g. real-time web applications).
- Code is written in "plain old JavaScript", which means that less time is spent dealing with "context shift" between languages when you're writing both client-side and server-side code.
- JavaScript is a relatively new programming language and benefits from improvements in language design when compared to other traditional web-server languages (e.g. Python, PHP, etc.) Many other new and popular languages compile/convert into JavaScript so

you can also use TypeScript, CoffeeScript, ClojureScript, Scala, LiveScript, etc.

- The node package manager (NPM) provides access to hundreds of thousands of reusable packages. It also has best-in-class dependency resolution and can also be used to automate most of the build toolchain.
- Node.js is portable. It is available on Microsoft Windows, macOS, Linux, Solaris, FreeBSD, OpenBSD, WebOS, and NonStop OS. Furthermore, it is well-supported by many web hosting providers, that often provide specific infrastructure and documentation for hosting Node sites.
- It has a very active third party ecosystem and developer community, with lots of people who are willing to help.

You can use Node.js to create a simple web server using the Node HTTP package.

Hello Node.js

The following example creates a web server that listens for any kind of HTTP request on the URL `http://127.0.0.1:8000/` — when a request is received, the script will respond with the string: "Hello World". If you have already installed node, you can follow these steps to try out the example:

1. Open Terminal (on Windows, open the command line utility)
2. Create the folder where you want to save the program, for example, `test-node` and then enter it by entering the following command into your terminal:

```
cd test-node
```

3. Using your favorite text editor, create a file called `hello.js` and paste the following code into it:

```
// Load HTTP module
const http = require("http");

const hostname = "127.0.0.1";
const port = 8000;

// Create HTTP server
const server = http.createServer((req, res) => {

  // Set the response HTTP header with HTTP status and Content type
  res.writeHead(200, {'Content-Type': 'text/plain'});

  // Send the response body "Hello World"
```

```
res.end('Hello World\n');
});

// Prints a log once the server starts listening
server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
})
```

4. Save the file in the folder you created above.
5. Go back to the terminal and type the following command:

```
node hello.js
```

Finally, navigate to `http://localhost:8000` in your web browser; you should see the text **"Hello World"** in the upper left of an otherwise empty web page.

Web Frameworks

Other common web-development tasks are not directly supported by Node itself. If you want to add specific handling for different HTTP verbs (e.g. GET , POST , DELETE , etc.), separately handle requests at different URL paths ("routes"), serve static files, or use templates to dynamically create the response, Node won't be of much use on its own. You will either need to write the code yourself, or you can avoid reinventing the wheel and use a web framework!

Introducing Express

[Express](#) is the most popular *Node* web framework, and is the underlying library for a number of other popular [Node web frameworks](#) . It provides mechanisms to:

- Write handlers for requests with different HTTP verbs at different URL paths (routes).
- Integrate with "view" rendering engines in order to generate responses by inserting data into templates.
- Set common web application settings like the port to use for connecting, and the location of templates that are used for rendering the response.
- Add additional request processing "middleware" at any point within the request handling pipeline.

While *Express* itself is fairly minimalist, developers have created compatible middleware packages to address almost any web development problem. There are libraries to work with cookies, sessions, user logins, URL parameters, POST data, security headers, and *many* more. You can find a list of middleware packages maintained by the Express team at [Express](#)

[Middleware](#) (along with a list of some popular 3rd party packages)

[middleware](#) (along with a list of some popular 3rd party packages).

Note: This flexibility is a double edged sword. There are middleware packages to address almost any problem or requirement, but working out the right packages to use can sometimes be a challenge. There is also no "right way" to structure an application, and many examples you might find on the Internet are not optimal, or only show a small part of what you need to do in order to develop a web application.

Where did Node and Express come from?

Node was initially released, for Linux only, in 2009. The NPM package manager was released in 2010, and native Windows support was added in 2012. At time of writing the current LTS release is Node v12.18.4 while the latest release is Node 14.13.0. This is a tiny snapshot of a rich history; delve into [Wikipedia](#) if you want to know more.

Express was initially released in November 2010 and is currently on version 4.17.1 of the API (with 5.0 in "alpha"). You can check out the [changelog](#) for information about changes in the current release, and [GitHub](#) for more detailed historical release notes.

How popular are Node and Express?

The popularity of a web framework is important because it is an indicator of whether it will continue to be maintained, and what resources are likely to be available in terms of documentation, add-on libraries, and technical support.

There isn't any readily-available and definitive measure of the popularity of server-side frameworks (although you can estimate popularity using mechanisms like counting the number of GitHub projects and StackOverflow questions for each platform). A better question is whether Node and Express are "popular enough" to avoid the problems of unpopular platforms. Are they continuing to evolve? Can you get help if you need it? Is there an opportunity for you to get paid work if you learn Express?

Based on the number of [high profile companies](#) that use Express, the number of people contributing to the codebase, and the number of people providing both free and paid for support, then yes, *Express* is a popular framework!

Is Express opinionated?

Web frameworks often refer to themselves as "opinionated" or "unopinionated"

Web frameworks often refer to themselves as "opinionated" or "unopinionated".

Opinionated frameworks are those with opinions about the "right way" to handle any particular task. They often support rapid development *in a particular domain* (solving problems of a particular type) because the right way to do anything is usually well-understood and well-documented. However they can be less flexible at solving problems outside their main domain, and tend to offer fewer choices for what components and approaches they can use.

Unopinionated frameworks, by contrast, have far fewer restrictions on the best way to glue components together to achieve a goal, or even what components should be used. They make it easier for developers to use the most suitable tools to complete a particular task, albeit at the cost that you need to find those components yourself.

Express is unopinionated. You can insert almost any compatible middleware you like into the request handling chain, in almost any order you like. You can structure the app in one file or multiple files, and using any directory structure. You may sometimes feel that you have too many choices!

What does Express code look like?

In a traditional data-driven website, a web application waits for HTTP requests from the web browser (or other client). When a request is received the application works out what action is needed based on the URL pattern and possibly associated information contained in POST data or GET data. Depending on what is required it may then read or write information from a database or perform other tasks required to satisfy the request. The application will then return a response to the web browser, often dynamically creating an HTML page for the browser to display by inserting the retrieved data into placeholders in an HTML template.

Express provides methods to specify what function is called for a particular HTTP verb (GET, POST, SET, etc.) and URL pattern ("Route"), and methods to specify what template ("view") engine is used, where template files are located, and what template to use to render a response. You can use Express middleware to add support for cookies, sessions, and users, getting POST/GET parameters, etc. You can use any database mechanism supported by Node (Express does not define any database-related behavior).

The following sections explain some of the common things you'll see when working with *Express* and *Node* code.

Hello world Express

First let's consider the standard Express [Hello World](#) example (we discuss each part of this below, and in the following sections).

Tip: If you have Node and Express already installed (or if you install them as shown in the [next article](#)), you can save this code in a text file called **app.js** and run it in a bash command prompt by calling:

```
node ./app.js
```

```
const express = require('express');
const app = express();
const port = 3000;

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

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

The first two lines `require()` (import) the express module and create an [Express application](#). This object, which is traditionally named `app`, has methods for routing HTTP requests, configuring middleware, rendering HTML views, registering a template engine, and modifying [application settings](#) that control how the application behaves (e.g. the environment mode, whether route definitions are case sensitive, etc.)

The middle part of the code (the three lines starting with `app.get`) shows a *route definition*. The `app.get()` method specifies a callback function that will be invoked whenever there is an HTTP GET request with a path (`'/'`) relative to the site root. The callback function takes a request and a response object as arguments, and calls [send\(\)](#) on the response to return the string "Hello World!"

The final block starts up the server on a specified port ('3000') and prints a log comment to the console. With the server running, you could go to `localhost:3000` in your browser to see the example response returned.

Importing and creating modules

A module is a JavaScript library/file that you can import into other code using Node's `require()` function. *Express* itself is a module, as are the middleware and database libraries that we use in our *Express* applications.

The code below shows how we import a module by name, using the *Express* framework as an example. First we invoke the `require()` function, specifying the name of the module as a string (`'express'`), and calling the returned object to create an [Express application](#) . We can then access the properties and functions of the application object.

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

You can also create your own modules that can be imported in the same way.

Tip: You will *want* to create your own modules, because this allows you to organize your code into manageable parts — a monolithic single-file application is hard to understand and maintain. Using modules also helps you manage your namespace, because only the variables you explicitly export are imported when you use a module.

To make objects available outside of a module you just need to expose them as additional properties on the `exports` object. For example, the **square.js** module below is a file that exports `area()` and `perimeter()` methods:

```
exports.area = function(width) { return width * width; };
exports.perimeter = function(width) { return 4 * width; };
```

We can import this module using `require()` , and then call the exported method(s) as shown:

```
const square = require('./square'); // Here we require() the name of the file
console.log('The area of a square with a width of 4 is ' + square.area(4));
```

Note: You can also specify an absolute path to the module (or a name, as we did initially).

If you want to export a complete object in one assignment instead of building it one property at a time, assign it to `module.exports` as shown below (you can also do this to make the root of the `exports` object a constructor or other function):

```
module.exports = {  
  area: function(width) {  
    return width * width;  
  },  
  
  perimeter: function(width) {  
    return 4 * width;  
  }  
};
```

Note: You can think of `exports` as a shortcut to `module.exports` within a given module. In fact, `exports` is just a variable that gets initialized to the value of `module.exports` before the module is evaluated. That value is a reference to an object (empty object in this case). This means that `exports` holds a reference to the same object referenced by `module.exports`. It also means that by assigning another value to `exports` it's no longer bound to `module.exports`.

For a lot more information about modules see [Modules](#) (Node API docs).

Using asynchronous APIs

JavaScript code frequently uses asynchronous rather than synchronous APIs for operations that may take some time to complete. A synchronous API is one in which each operation must complete before the next operation can start. For example, the following log functions are synchronous, and will print the text to the console in order (First, Second).

```
console.log('First');  
console.log('Second');
```

By contrast, an asynchronous API is one in which the API will start an operation and immediately return (before the operation is complete). Once the operation finishes, the API will use some mechanism to perform additional operations. For example, the code below will print out "Second, First" because even though `setTimeout()` method is called first, and returns immediately, the operation doesn't complete for several seconds.

```
setTimeout(function() {  
  console.log('First');  
}, 3000);  
console.log('Second');
```

Using non-blocking asynchronous APIs is even more important on Node than in the browser because *Node* is a single-threaded event-driven execution environment. "Single threaded" means that all requests to the server are run on the same thread (rather than being spawned off into separate processes). This model is extremely efficient in terms of speed and server resources, but it does mean that if any of your functions call synchronous methods that take a long time to complete, they will block not just the current request, but every other request being handled by your web application.

There are a number of ways for an asynchronous API to notify your application that it has completed. The most common way is to register a callback function when you invoke the asynchronous API, that will be called back when the operation completes. This is the approach used above.

Tip: Using callbacks can be quite "messy" if you have a sequence of dependent asynchronous operations that must be performed in order because this results in multiple levels of nested callbacks. This problem is commonly known as "callback hell". This problem can be reduced by good coding practices (see <http://callbackhell.com/>), using a module like `async`, or even moving to ES6 features like `Promises`.

Note: A common convention for Node and Express is to use error-first callbacks. In this convention, the first value in your *callback functions* is an error value, while subsequent arguments contain success data. There is a good explanation of why this approach is useful in this blog: [The Node.js Way - Understanding Error-First Callbacks](http://fredkschott.com/blog/the-node-js-way-understanding-error-first-callbacks/) (fredkschott.com).

Creating route handlers

In our *Hello World* Express example (see above), we defined a (callback) route handler function for HTTP GET requests to the site root (`'/'`).

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

The callback function takes a request and a response object as arguments. In this case, the method calls `send()` on the response to return the string "Hello World!" There are a [number of other response methods](#) for ending the request/response cycle, for example, you

could call `res.json()` to send a JSON response or `res.sendFile()` to send a file.

JavaScript tip: You can use any argument names you like in the callback functions; when the callback is invoked the first argument will always be the request and the second will always be the response. It makes sense to name them such that you can identify the object you're working with in the body of the callback.

The *Express application* object also provides methods to define route handlers for all the other HTTP verbs, which are mostly used in exactly the same way:

`checkout()`, `copy()`, `delete()`, `get()`, `head()`, `lock()`, `merge()`, `mkactivity()`, `mkcol()`, `move()`, `m-search()`, `notify()`, `options()`, `patch()`, `post()`, `purge()`, `put()`, `report()`, `search()`, `subscribe()`, `trace()`, `unlock()`, `unsubscribe()`.

There is a special routing method, `app.all()`, which will be called in response to any HTTP method. This is used for loading middleware functions at a particular path for all request methods. The following example (from the Express documentation) shows a handler that will be executed for requests to `/secret` irrespective of the HTTP verb used (provided it is supported by the [http module](#)).

```
app.all('/secret', function(req, res, next) {  
  console.log('Accessing the secret section ...');  
  next(); // pass control to the next handler  
});
```

Routes allow you to match particular patterns of characters in a URL, and extract some values from the URL and pass them as parameters to the route handler (as attributes of the request object passed as a parameter).

Often it is useful to group route handlers for a particular part of a site together and access them using a common route-prefix (e.g. a site with a Wiki might have all wiki-related routes in one file and have them accessed with a route prefix of `/wiki/`). In *Express* this is achieved by using the [express.Router](#) object. For example, we can create our wiki route in a module named **wiki.js**, and then export the `Router` object, as shown below:

```
// wiki.js - Wiki route module
```

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

// Home page route
router.get('/', function(req, res) {
  res.send('Wiki home page');
});

// About page route
router.get('/about', function(req, res) {
  res.send('About this wiki');
});

module.exports = router;
```

Note: Adding routes to the `Router` object is just like adding routes to the `app` object (as shown previously).

To use the router in our main app file we would then `require()` the route module (**wiki.js**), then call `use()` on the *Express* application to add the `Router` to the middleware handling path. The two routes will then be accessible from `/wiki/` and `/wiki/about/`.

```
const wiki = require('./wiki.js');
// ...
app.use('/wiki', wiki);
```

We'll show you a lot more about working with routes, and in particular about using the `Router`, later on in the linked section [Routes and controllers](#).

Using middleware

Middleware is used extensively in Express apps, for tasks from serving static files to error handling, to compressing HTTP responses. Whereas route functions end the HTTP request-response cycle by returning some response to the HTTP client, middleware functions *typically* perform some operation on the request or response and then call the next function in the "stack", which might be more middleware or a route handler. The order in which middleware is called is up to the app developer.

Note: The middleware can perform any operation, execute any code, make changes to the request and response object, and it can also end the request-response cycle. If it does not

request and response object, and it can *also end the request-response cycle*. If it does not end the cycle then it must call `next()` to pass control to the next middleware function (or the request will be left hanging).

Most apps will use *third-party* middleware in order to simplify common web development tasks like working with cookies, sessions, user authentication, accessing request POST and JSON data, logging, etc. You can find a [list of middleware packages maintained by the Express team](#) (which also includes other popular 3rd party packages). Other Express packages are available on the NPM package manager.

To use third party middleware you first need to install it into your app using NPM. For example, to install the [morgan](#) HTTP request logger middleware, you'd do this:

```
$ npm install morgan
```

You could then call `use()` on the *Express application object* to add the middleware to the stack:

```
const express = require('express');
const logger = require('morgan');
const app = express();
app.use(logger('dev'));
...
```

Note: Middleware and routing functions are called in the order that they are declared. For some middleware the order is important (for example if session middleware depends on cookie middleware, then the cookie handler must be added first). It is almost always the case that middleware is called before setting routes, or your route handlers will not have access to functionality added by your middleware.

You can write your own middleware functions, and you are likely to have to do so (if only to create error handling code). The **only** difference between a middleware function and a route handler callback is that middleware functions have a third argument `next`, which middleware functions are expected to call if they are not that which completes the request cycle (when the middleware function is called, this contains the *next* function that must be called).

You can add a middleware function to the processing chain for *all responses* with `app.use()`, or for a specific HTTP verb using the associated method: `app.get()`, `app.post()`,

etc. Routes are specified in the same way for both cases, though the route is optional when calling `app.use()`.

The example below shows how you can add the middleware function using both approaches, and with/without a route.

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

// An example middleware function
let a_middleware_function = function(req, res, next) {
  // ... perform some operations
  next(); // Call next() so Express will call the next middleware function i
}

// Function added with use() for all routes and verbs
app.use(a_middleware_function);

// Function added with use() for a specific route
app.use('/someroute', a_middleware_function);

// A middleware function added for a specific HTTP verb and route
app.get('/', a_middleware_function);

app.listen(3000);
```

JavaScript Tip: Above we declare the middleware function separately and then set it as the callback. In our previous route handler function we declared the callback function when it was used. In JavaScript, either approach is valid.

The Express documentation has a lot more excellent documentation about [using](#) and [writing](#) Express middleware.

Serving static files

You can use the [express.static](#) middleware to serve static files, including your images, CSS and JavaScript (`static()` is the only middleware function that is actually **part** of *Express*). For example, you would use the line below to serve images, CSS files, and JavaScript files from a directory named **'public'** at the same level as where you call node:

```
app.use(express.static('public'));
```

Any files in the public directory are served by adding their filename (*relative* to the base "public" directory) to the base URL. So for example:

```
http://localhost:3000/images/dog.jpg
http://localhost:3000/css/style.css
http://localhost:3000/js/app.js
http://localhost:3000/about.html
```

You can call `static()` multiple times to serve multiple directories. If a file cannot be found by one middleware function then it will be passed on to the subsequent middleware (the order that middleware is called is based on your declaration order).

```
app.use(express.static('public'));
app.use(express.static('media'));
```

You can also create a virtual prefix for your static URLs, rather than having the files added to the base URL. For example, here we [specify a mount path](#) so that the files are loaded with the prefix `/media`:

```
app.use('/media', express.static('public'));
```

Now, you can load the files that are in the `public` directory from the `/media` path prefix.

```
http://localhost:3000/media/images/dog.jpg
http://localhost:3000/media/video/cat.mp4
http://localhost:3000/media/cry.mp3
```

Note: See also [Serving static files in Express](#) .

Handling errors

Errors are handled by one or more special middleware functions that have four arguments, instead of the usual three: `(err, req, res, next)` . For example:

```
app.use(function(err, req, res, next) {
  console.error(err.stack);
```

```
res.status(500).send('Something broke!');  
});
```

These can return any content required, but must be called after all other `app.use()` and routes calls so that they are the last middleware in the request handling process!

Express comes with a built-in error handler, which takes care of any remaining errors that might be encountered in the app. This default error-handling middleware function is added at the end of the middleware function stack. If you pass an error to `next()` and you do not handle it in an error handler, it will be handled by the built-in error handler; the error will be written to the client with the stack trace.

Note: The stack trace is not included in the production environment. To run it in production mode you need to set the environment variable `NODE_ENV` to `'production'`.

Note: HTTP404 and other "error" status codes are not treated as errors. If you want to handle these, you can add a middleware function to do so. For more information see the [FAQ](#).

For more information see [Error handling](#) (Express docs).

Using databases

Express apps can use any database mechanism supported by *Node* (Express itself doesn't define any specific additional behavior/requirements for database management). There are many options, including PostgreSQL, MySQL, Redis, SQLite, MongoDB, etc.

In order to use these you have to first install the database driver using NPM. For example, to install the driver for the popular NoSQL MongoDB you would use the command:

```
$ npm install mongodb
```

The database itself can be installed locally or on a cloud server. In your Express code you require the driver, connect to the database, and then perform create, read, update, and delete (CRUD) operations. The example below (from the Express documentation) shows how you can find "mammal" records using MongoDB.

```
//this works with older versions of mongodb version ~ 2.2.33
const MongoClient = require('mongodb').MongoClient;

MongoClient.connect('mongodb://localhost:27017/animals', function(err, db) {
  if (err) throw err;

  db.collection('mammals').find().toArray(function (err, result) {
    if (err) throw err;

    console.log(result);
  });
});

//for mongodb version 3.0 and up
const MongoClient = require('mongodb').MongoClient;
MongoClient.connect('mongodb://localhost:27017/animals', function(err, client) {
  if(err) throw err;

  let db = client.db('animals');
  db.collection('mammals').find().toArray(function(err, result){
    if(err) throw err;
    console.log(result);
    client.close();
  });
});
```

Another popular approach is to access your database indirectly, via an Object Relational Mapper ("ORM"). In this approach you define your data as "objects" or "models" and the ORM maps these through to the underlying database format. This approach has the benefit that as a developer you can continue to think in terms of JavaScript objects rather than database semantics, and that there is an obvious place to perform validation and checking of incoming data. We'll talk more about databases in a later article.

For more information see [Database integration](#) (Express docs).

Rendering data (views)

Template engines (referred to as "view engines" by *Express*) allow you to specify the *structure* of an output document in a template, using placeholders for data that will be filled in when a page is generated. Templates are often used to create HTML, but can also create other types of documents. Express has support for [a number of template engines](#), and there is a useful

comparison of the more popular engines here: [Comparing JavaScript Templating Engines: Jade, Mustache, Dust and More](#)

In your application settings code you set the template engine to use and the location where Express should look for templates using the 'views' and 'view engines' settings, as shown below (you will also have to install the package containing your template library too!)

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

// Set directory to contain the templates ('views')
app.set('views', path.join(__dirname, 'views'));

// Set view engine to use, in this case 'some_template_engine_name'
app.set('view engine', 'some_template_engine_name');
```

The appearance of the template will depend on what engine you use. Assuming that you have a template file named "index.<template_extension>" that contains placeholders for data variables named 'title' and "message", you would call [Response.render\(\)](#) in a route handler function to create and send the HTML response:

```
app.get('/', function(req, res) {
  res.render('index', { title: 'About dogs', message: 'Dogs rock!' });
});
```

For more information see [Using template engines with Express](#) (Express docs).

File structure

Express makes no assumptions in terms of structure or what components you use. Routes, views, static files, and other application-specific logic can live in any number of files with any directory structure. While it is perfectly possible to have the whole *Express* application in one file, typically it makes sense to split your application into files based on function (e.g. account management, blogs, discussion boards) and architectural problem domain (e.g. model, view or controller if you happen to be using an [MVC architecture](#)).

In a later topic we'll use the *Express Application Generator*, which creates a modular app skeleton that we can easily extend for creating web applications.

Summary

Congratulations, you've completed the first step in your Express/Node journey! You should now understand Express and Node's main benefits, and roughly what the main parts of an Express app might look like (routes, middleware, error handling, and template code). You should also understand that with Express being an unopinionated framework, the way you pull these parts together and the libraries that you use are largely up to you!

Of course Express is deliberately a very lightweight web application framework, so much of its benefit and potential comes from third party libraries and features. We'll look at those in more detail in the following articles. In our next article we're going to look at setting up a Node development environment, so that you can start seeing some Express code in action.

See also

- [Venkat.R - Manage Multiple Node versions](#)
- [Modules](#) (Node API docs)
- [Express](#) (home page)
- [Basic routing](#) (Express docs)
- [Routing guide](#) (Express docs)
- [Using template engines with Express](#) (Express docs)
- [Using middleware](#) (Express docs)
- [Writing middleware for use in Express apps](#) (Express docs)
- [Database integration](#) (Express docs)
- [Serving static files in Express](#) (Express docs)
- [Error handling](#) (Express docs)

In this module

- [Express/Node introduction](#)
- [Setting up a Node \(Express\) development environment](#)
- [Express Tutorial: The Local Library website](#)
- [Express Tutorial Part 2: Creating a skeleton website](#)
- [Express Tutorial Part 3: Using a Database \(with Mongoose\)](#)
- [Express Tutorial Part 4: Routes and controllers](#)
- [Express Tutorial Part 5: Displaying library data](#)
- [Express Tutorial Part 6: Working with forms](#)
- [Express Tutorial Part 7: Deploying to production](#)

Last modified: Feb 16, 2021, [by MDN contributors](#)

Change your language

English (US) ▼

Change language