**Installing**

Assuming you’ve already installed [Node.js](https://nodejs.org/), create a directory to hold your application, and make that your working directory.

* $ mkdir myapp
* $ cd myapp

Use the npm init command to create a package.json file for your application. For more information on how package.json works, see [Specifics of npm’s package.json handling](https://docs.npmjs.com/files/package.json).

* $ npm init

This command prompts you for a number of things, such as the name and version of your application. For now, you can simply hit RETURN to accept the defaults for most of them, with the following exception:

entry point: (index.js)

Enter app.js, or whatever you want the name of the main file to be. If you want it to be index.js, hit RETURN to accept the suggested default file name.

Now install Express in the myapp directory and save it in the dependencies list. For example:

* $ npm install express --save

To install Express temporarily and not add it to the dependencies list:

* $ npm install express --no-save

# Hello world example



### Running Locally

$ node app.js

# Express application generator

Use the application generator tool, express-generator, to quickly create an application skeleton.

You can run the application generator with the npx command (available in Node.js 8.2.0).

$ npx express-generator

For earlier Node versions, install the application generator as a global npm package and then launch it:

$ npm install -g express-generator

$ express

Display the command options with the -h option:

$ express -h

Usage: express [options] [dir]

Options:

-h, --help output usage information

--version output the version number

-e, --ejs add ejs engine support

--hbs add handlebars engine support

--pug add pug engine support

-H, --hogan add hogan.js engine support

--no-view generate without view engine

-v, --view <engine> add view <engine> support (ejs|hbs|hjs|jade|pug|twig|vash) (defaults to jade)

-c, --css <engine> add stylesheet <engine> support (less|stylus|compass|sass) (defaults to plain css)

--git add .gitignore

-f, --force force on non-empty directory

For example, the following creates an Express app named myapp. The app will be created in a folder named myapp in the current working directory and the view engine will be set to [Pug](https://pugjs.org/):

$ express --view=pug myapp

create : myapp

create : myapp/package.json

create : myapp/app.js

create : myapp/public

create : myapp/public/javascripts

create : myapp/public/images

create : myapp/routes

create : myapp/routes/index.js

create : myapp/routes/users.js

create : myapp/public/stylesheets

create : myapp/public/stylesheets/style.css

create : myapp/views

create : myapp/views/index.pug

create : myapp/views/layout.pug

create : myapp/views/error.pug

create : myapp/bin

create : myapp/bin/www

Then install dependencies:

$ cd myapp

$ npm install

On MacOS or Linux, run the app with this command:

$ DEBUG=myapp:\* npm start

On Windows Command Prompt, use this command:

> set DEBUG=myapp:\* & npm start

On Windows PowerShell, use this command:

PS> $env:DEBUG='myapp:\*'; npm start

Then load http://localhost:3000/ in your browser to access the app.

The generated app has the following directory structure:

.

├── app.js

├── bin

│ └── www

├── package.json

├── public

│ ├── images

│ ├── javascripts

│ └── stylesheets

│ └── style.css

├── routes

│ ├── index.js

│ └── users.js

└── views

├── error.pug

├── index.pug

└── layout.pug

7 directories, 9 files

# Basic routing

Routing refers to determining how an application responds to a client request to a particular endpoint, which is a URI (or path) and a specific HTTP request method (GET, POST, and so on).

Each route can have one or more handler functions, which are executed when the route is matched.

Route definition takes the following structure:

app.METHOD(PATH, HANDLER)

Where:

* app is an instance of express.
* METHOD is an [HTTP request method](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol" \l "Request_methods), in lowercase.
* PATH is a path on the server.
* HANDLER is the function executed when the route is matched.

## Route handlers

You can provide multiple callback functions that behave like [middleware](https://expressjs.com/en/guide/using-middleware.html) to handle a request. The only exception is that these callbacks might invoke next('route') to bypass the remaining route callbacks. You can use this mechanism to impose pre-conditions on a route, then pass control to subsequent routes if there’s no reason to proceed with the current route.

Route handlers can be in the form of a function, an array of functions

# Using middleware

Express is a routing and middleware web framework that has minimal functionality of its own: An Express application is essentially a series of middleware function calls.

Middleware functions are functions that have access to the [request object](https://expressjs.com/en/4x/api.html" \l "req) (req), the [response object](https://expressjs.com/en/4x/api.html" \l "res) (res), and the next middleware function in the application’s request-response cycle. The next middleware function is commonly denoted by a variable named next.

Middleware functions can perform the following tasks:

* Execute any code.
* Make changes to the request and the response objects.
* End the request-response cycle.
* Call the next middleware function in the stack.

If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging.

An Express application can use the following types of middleware:

* [Application-level middleware](https://expressjs.com/en/guide/using-middleware.html" \l "middleware.application)
* [Router-level middleware](https://expressjs.com/en/guide/using-middleware.html" \l "middleware.router)
* [Error-handling middleware](https://expressjs.com/en/guide/using-middleware.html" \l "middleware.error-handling)
* [Built-in middleware](https://expressjs.com/en/guide/using-middleware.html" \l "middleware.built-in)
* [Third-party middleware](https://expressjs.com/en/guide/using-middleware.html" \l "middleware.third-party)

You can load application-level and router-level middleware with an optional mount path. You can also load a series of middleware functions together, which creates a sub-stack of the middleware system at a mount point.

## Application-level middleware

Bind application-level middleware to an instance of the [app object](https://expressjs.com/en/4x/api.html" \l "app) by using the app.use() and app.METHOD() functions, where METHOD is the HTTP method of the request that the middleware function handles (such as GET, PUT, or POST) in lowercase.