Forum Donate

Learn to code — free 3,000-hour curriculum

NOVEMBER 3, 2017 / #WEB DEVELOPMENT

Going out to eat and understanding the basics of Express.js

Forum Donate

Learn to code — free 3,000-hour curriculum

Forum Donate

Learn to code — free 3,000-hour curriculum

If you have ever visited a sit-down restaurant, then you can understand the basics of Express. But if you are just starting to build your first Node.js back end…you might be in for a bumpy ride.

Yes — it is certainly easier to learn Node if you have past experience with JavaScript. But the challenges you will face while building a back end are completely different than the ones you face while using JavaScript on the front end.

When I learned Node, I chose the hard way. I studied eBooks, written tutorials, and videos over and over until I finally understood **why** I was doing what I was doing.

There is an easier way. I am going to use a restaurant analogy to explain four key parts of your first Express app. Express.js is a popular framework for organizing your code, and I would recommend it for any beginner. I'll explain further in a moment.

Here are the four key parts we will cover:

1. The require statements
2. Middleware
3. Routing
4. App.listen()/ Starting the server

Forum Donate

Learn to code — free 3,000-hour curriculum

who creates all the processes and manages the place so that it runs smoothly and customers leave happy.

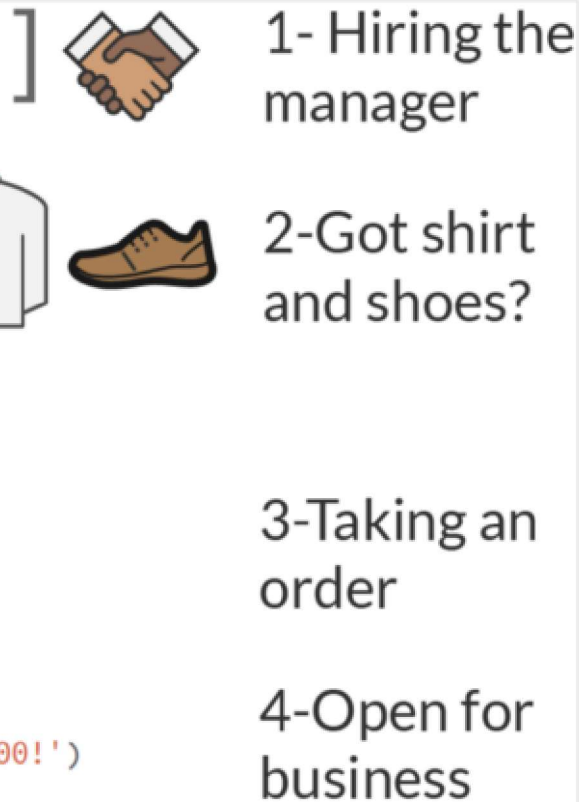
Here is a preview of what is next:

```
const express = require('express' 4.16.2 )
const app = express()

app.use(function (req, res, next) {
  console.log('Request: ', req)
  console.log('Response: ', res)
  next()
})

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

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



Forum Donate

Learn to code — free 3,000-hour curriculum

By the end, you will understand the functionality of every part of a basic Express app.

Step 1: hiring the manager (require statements)

In this example, you are the restaurant owner. And you need to hire an expert to run the day-to-day operations of your new restaurant. You certainly aren't an expert, and you can't leave it to the waitstaff and kitchen to figure out.

If you want to run an efficient and safe restaurant, you need someone to keep your staff working at maximum efficiency. Express is the new manager.

The first part is pretty straightforward. Like with any other NPM package, you need to npm install the express module and then use a **require** statement to load the module.

```
const express = require('express' 4.16.2 )
const app = express()
```

1- Hiring the manager

Forum Donate

Learn to code — free 3,000-hour curriculum

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

This is because you need a variable to hold your new Express application. Express is not a default part of Node.

Step 2: making decisions at the restaurant (middleware)

Let's take a step back here. What are some common routines that happen at restaurants? There are three that immediately jump into my head:

1. Seating new customers
2. Taking food orders
3. Presenting the check at the end of the meal

For each one, there are a series of checks that you need to run before you can execute the action. For example, before you seat customers you need to know:

1. Are they wearing a shirt and shoes (and pants)? Otherwise, they cannot be seated.

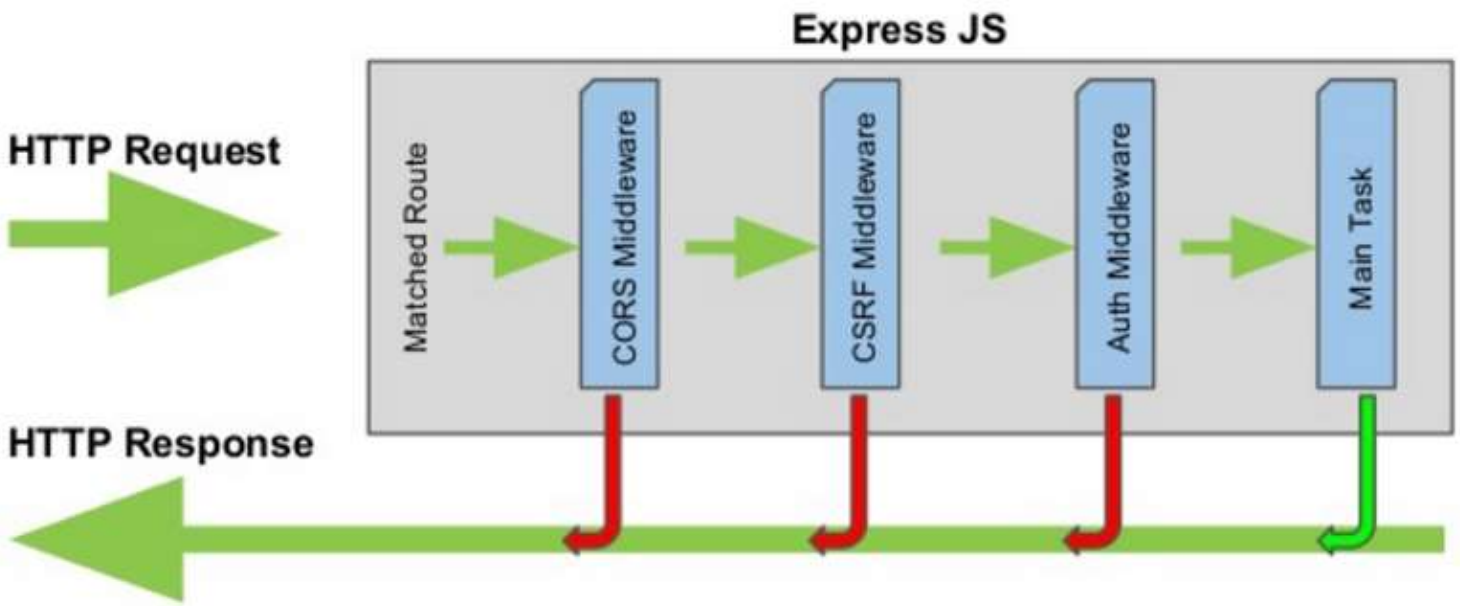
Forum Donate

Learn to code — free 3,000-hour curriculum

This ain't a beach bar! Similarly, in your code, you will need to validate that requests have certain criteria before they can continue. For example, if a person tries to log in to your site:

1. Do they have an account?
2. Did they enter the correct password?

This is where the concept of **middleware** comes in. Middleware functions allow you to take action on any incoming request and modify it before sending back a response.



Forum Donate

Learn to code — free 3,000-hour curriculum

In your restaurant, you need a series of rules to decide if you should seat incoming people or not. Let's say a couple walks through your door. You have one rule before giving them a table: are they wearing a shirt and shoes?

```
4  app.use(function (req, res, next) {
5    var shirt = req.shirt;
6    var shoes = req.shoes;
7    if(shirt && shoes){
8      next()
9    }
10 })
```

First, you start with app.use(). This means that these are simply rules that need to be applied for the routes coming up next. They are not a GET, POST, PUT or DELETE.

In line 4, you have an anonymous function with the parameters req, res and next. For the purposes of this code block, you are just inspecting the request (req) to see if it has shirt and

Forum Donate

Learn to code — free 3,000-hour curriculum

You also need to use the next() function at the end because you are simply validating clothing here. Later, in the routes, you will allow the guests to get an actual table.

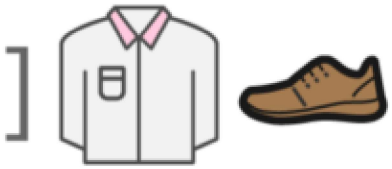
In lines 5 and 6, you check if they have a shirt and shoes.

And in lines 7–9, you only proceed if they have both.

The code block above is missing one important thing: A **path**. This is the specific string included with the request. And since it is missing a path, it will run on every single request.

Can you imagine? When customers entered the restaurant… ordered food… asked for the check… employees would be forced to look up and down at them to make sure they were clothed! That is a quick way to go out of business.

```
4  app.use('/table', function (req, res, next) {
5    var shirt = req.shirt;
6    var shoes = req.shoes;
7    if(shirt && shoes){
8      next()
9    }
10 })
```



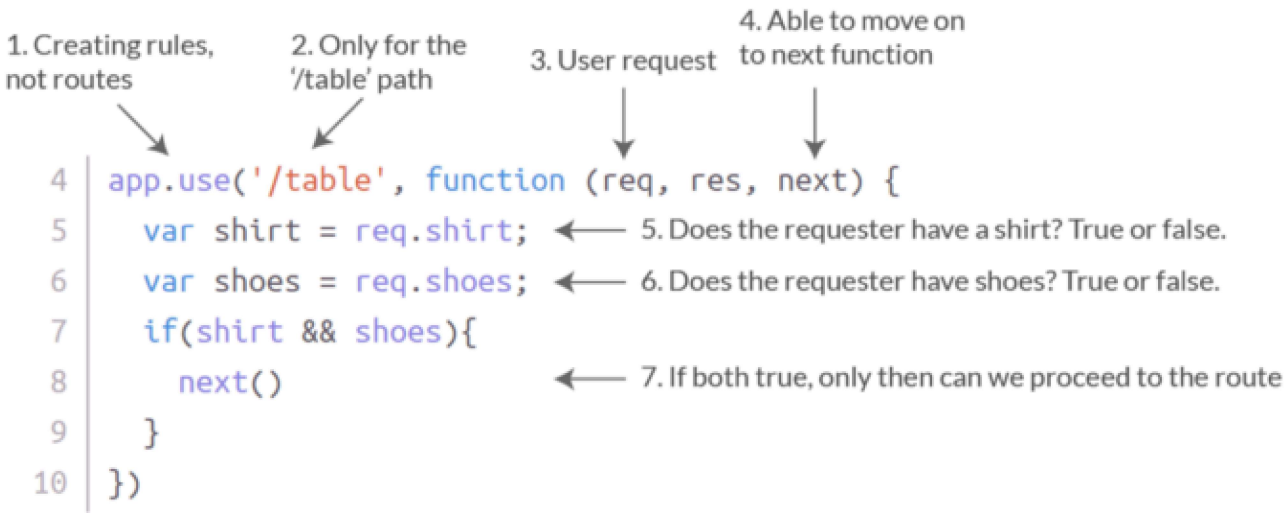
2-Got shirt and shoes?

Forum Donate

Learn to code — free 3,000-hour curriculum

So, we change line 4 in the example above. Now, we will only run this code when a user requests along the '/table' route.

The full explanation:



Step 3: executing common routines (routing)

Let's continue with the seating example. So far, we only know how to validate whether

Forum Donate

Learn to code — free 3,000-hour curriculum

This is where **routes** come in. Routes allow us to script specific actions based on the **path**. The options are GET, POST, PUT and DELETE, but we will focus on GET and POST for now.

In the context of a restaurant, we need to create a GET request in order to choose a specific table and seat the guests. GETs do not modify or add to your database. They just retrieve information based on specific parameters.

In this case, let's say that you need to create a procedure to seat a party of two. The number 2 came from the customer **request.**

```
app.get('/table/:amount', function (req, res) {
  var party = req.params.amount
  res.send('We are searching for your table for '+party+'!');
})
```

Okay, before I explain: Yes, this is only sending a message at the end. It has not actually found a specific table to seat the customer yet. I would need to search an array for an open table,

Forum Donate

Learn to code — free 3,000-hour curriculum

route. Just like the middleware example above, we have request and response parameters available. It also has a **parameter**, amount. This is two, in this example.

In fact, everything after the function declaration in line 12 is technically **middleware** since it modifies a user request. You will see in the diagram at the end.

In line 13, we access the number of people in the party from the **parameters** of the request object. That is not declared anywhere since the request came from the user, and we do not have any front-end code. So here is what the request might look like if this was a real app:

```
req = {  params: {    amount: 2;  }}
```

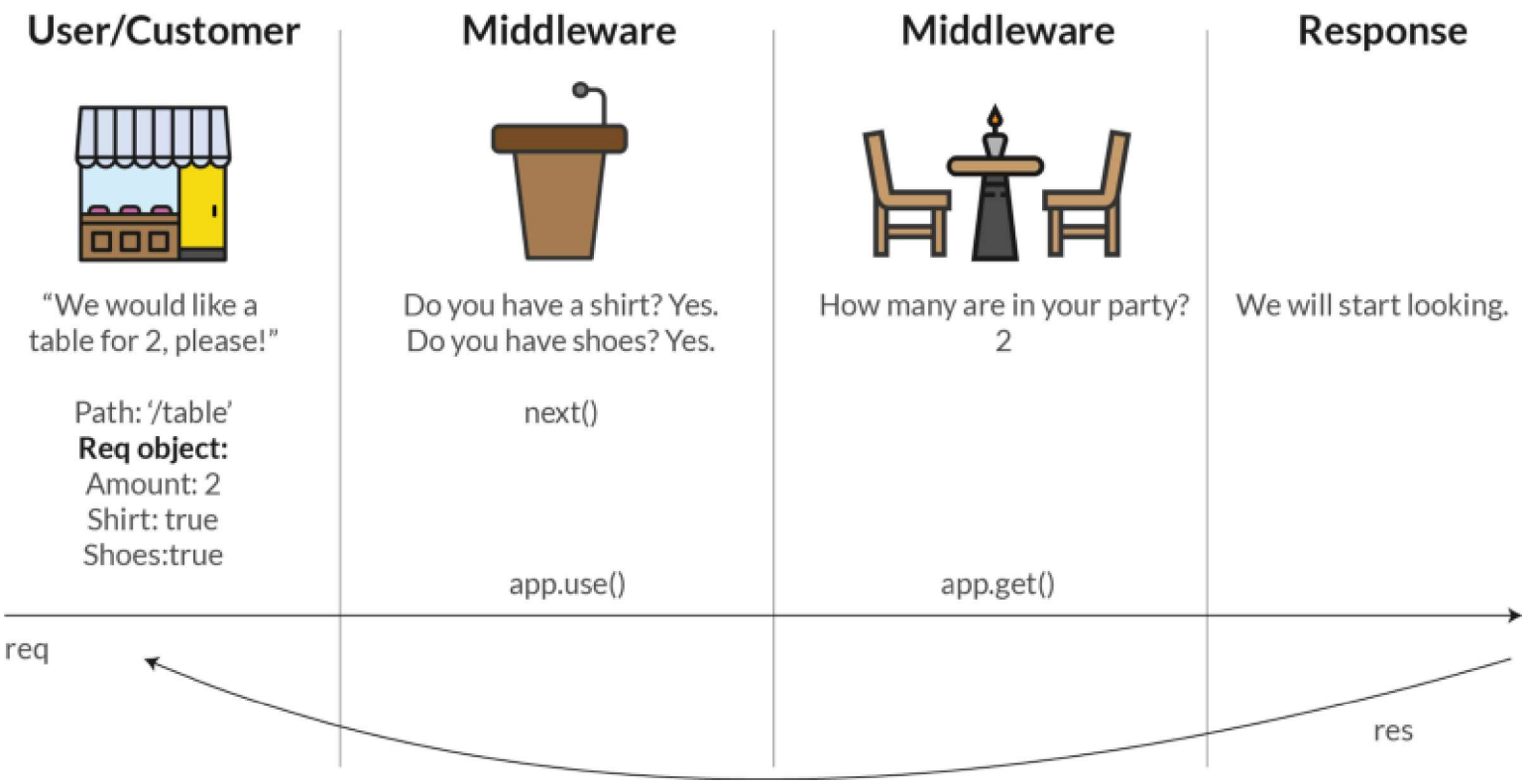
In line 13, our party variable accesses the amount **property** of the params **object** within the **request**.

Finally in line 14, we send a **response** back to the customer: we are looking for the appropriately sized table.

That is a lot at once. Here is a diagram:

Forum Donate

Learn to code — free 3,000-hour curriculum



Step 3.5: making your restaurant efficient (router)

Now you can trace the full path from request to response. But as your app grows in size, you will not want to code the rules for each route individually. You will find that some routes share the same rules, so you need to find a way to apply one set of rules to multiple routes.

Forum Donate

Learn to code — free 3,000-hour curriculum

And, in terms of serving customers, you will need to use a slightly different procedure for serving the appetizer, main course, and dinner. But, those three routes have plenty in common as well.

This is where the **router** comes in. The router lets you group your routes so that you can create common rules.

Seating	Serving Meals
All Routes -Need to have shirt + shoes	**All Routes** -Need to be ready to order
Seating at Bar -Need to be 21+	**Main Meal** -Need to have waited over 5 min to choose a dish

Forum Donate

Learn to code — free 3,000-hour curriculum

Here is the full code snippet:

```
const seatingRouter = express.Router()

seatingRouter.use(function (req, res, next) {
  var shirt = req.shirt;
  var shoes = req.shoes;
  if(shirt && shoes){
    next()
  }
})

seatingRouter.use('/bar', function (req, res, next) {
  var age = req.age;
  if(age > 21){
    next()
  }
})

app.use('/seating', seatingRouter);
```

Forum Donate

Learn to code — free 3,000-hour curriculum

In line 4, we declare our router.

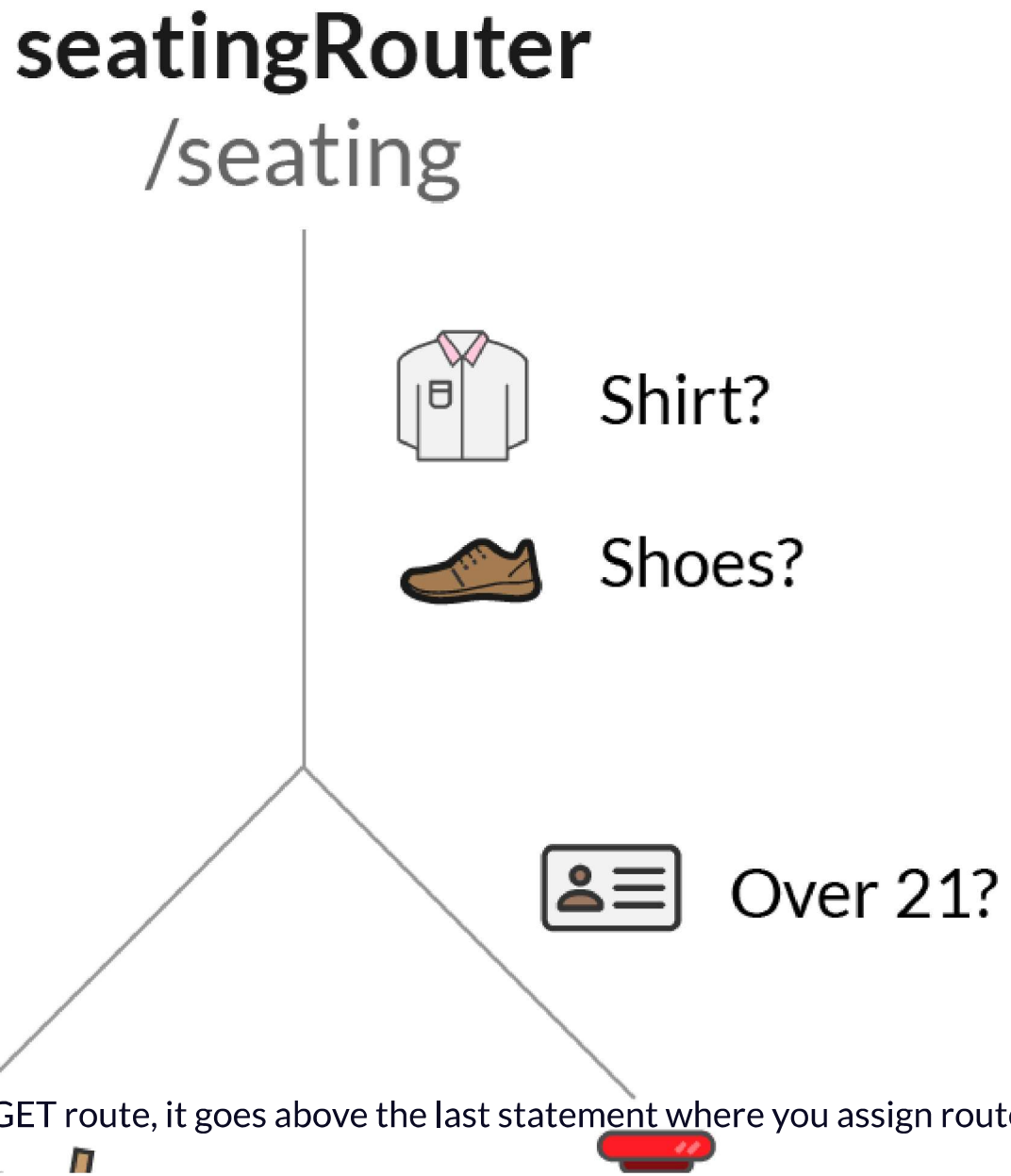
In lines 6 and 14, we now have seatingRouter.use() in place of app.use() to indicate that this **middleware** is only related to seatingRouter routes.

Finally, in line 21, we add more middleware to show that every seatingRouter route begins with '/seating'. So, if someone requested a seat at the bar, the full path would be '/seating/bar.' This may feel a little out of order, since you might expect the path to be defined when you create the router in line 4. That is normal!

Here is that in diagram form:

Forum Donate

Learn to code — free 3,000-hour curriculum



And, when you add a GET route, it goes above the last statement where you assign routes to the router.

Forum Donate

Learn to code — free 3,000-hour curriculum

Seat at a table Seat at bar

```
seatingRouter.get('/bar/:amount', function (req, res) {
  var party = req.params.amount
  res.send('We are searching for a spot at the bar for '+party+'!');
})

app.use('/seating', seatingRouter);
```

Step 4: opening for business (ports)

Okay, last part. So far, you have hired a manager, defined what to do before accepting customer requests, and determined what to do with specific customer requests once they come in. Now, you just need to determine the address for the location where all this will happen.

Your server has **ports** that are kind of like the address for the restaurant itself. Since your server can handle many types of restaurants (or server-side scripts) at once, you need to tell it where each script should run.

Forum Donate

Learn to code — free 3,000-hour curriculum

In the example above, the port is 3000 and it is located on your computer. So if you type:

```
https://localhost:3000/
```

into your browser, and you are running your Node app, the server knows to run the specific script. In this case, as soon as you enter the URL, you will log the message in the console and be able to use any of your **routes**. If the restaurant itself is your entire app, then it is now open for business at the address 3000.

Forum Donate

Learn to code — free 3,000-hour curriculum

Did you enjoy this? Give it a clap so others can discover it as well. And, if you want to get notified when I release future tutorials that use analogies, sign up here:

If this article was helpful, tweet it.

Learn to code for free. freeCodeCamp's open source curriculum has helped more than 40,000 people get jobs as developers. Get started

freeCodeCamp is a donor-supported tax-exempt 501(c)(3) nonprofit organization (United States Federal Tax Identification Number: 82-0779546)

Our mission: to help people learn to code for free. We accomplish this by creating thousands of videos, articles, and interactive coding lessons - all freely available to the public. We also have thousands of freeCodeCamp study groups around the world.

Donations to freeCodeCamp go toward our education initiatives and help pay for servers, services, and staff.

You can make a tax-deductible donation here.

Forum Donate

Learn to code — free 3,000-hour curriculum

Big O Notation
SQL Outer Join
Python For Loop
What is JavaScript?
Learn How To Code
Chrome Bookmarks
Concatenate Excel
C# String to Int
Git Switch Branch
JavaScript Splice
Model View Controller
Git Checkout Remote Branch
Insert Checkbox in Word
Find and Replace in Word
C Programming Language
HTML Link
Bayes Rule
Python Map
HTML Italics
Python SQL
HTML Bold
GraphQL VS Rest
If Function Excel
HTML List
Wav File
Subnet Cheat Sheet
String to Char Array Java
JavaScript Append to Array
Add Page Numbers in Word
JavaScript Projects

Our Nonprofit

About Alumni Network Open Source Shop Support Sponsors Academic Honesty Code of Conduct Privacy Policy

Terms of Service Copyright Policy

Forum Donate

Learn to code — free 3,000-hour curriculum