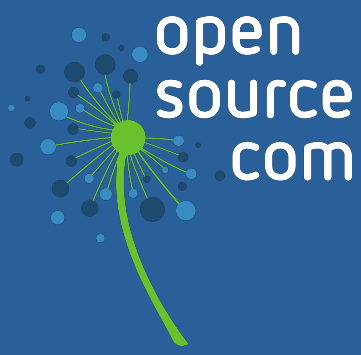


Learn JavaScript

Opensource.com

We are Opensource.com

Opensource.com is a community website publishing stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Do you have an open source story to tell? Submit a story idea at opensource.com/story

Email us at open@opensource.com

Supported by
Red Hat

Table of Contents

Learn JavaScript by writing a guessing game

By Mandy Kendall

It's pretty safe to say that most of the modern web would not exist without JavaScript. It's one of the three standard web technologies (along with HTML and CSS) and allows anyone to create much of the interactive, dynamic content we have come to expect in our experiences with the World Wide Web. From frameworks like React to data visualization libraries like D3, it's hard to imagine the web without it.

There's a lot to learn, and a great way to begin learning this popular language is by writing a simple application to become familiar with some concepts. Recently, some Opensource.com correspondents have written about how to learn their favorite language by writing a simple guessing game, so that's a great place to start!

Getting started

JavaScript comes in many flavors, but I'll start with the basic version, commonly called "Vanilla JavaScript." JavaScript is primarily a client-side scripting language, so it can run in any standard browser without installing anything. All you need is a code editor (Brackets is a great one to try) and the web browser of your choice.

HTML user interface

JavaScript runs in a web browser and interacts with the other standard web technologies, HTML and CSS. To create this game, you'll first use HTML (Hypertext Markup Language) to create a simple interface for your players to use. In case you aren't familiar, HTML is a markup language used to provide structure to content on the web.

To start, create an HTML file for your code. The file should have the `.html` extension to let the browser know that it is an HTML document. You can call your file `guessingGame.html`.

Use a few basic HTML tags in this file to display the game's title, instructions for how to play, interactive elements for the player to use to enter and submit their guesses, and a placeholder for providing feedback to the player:

```
<!DOCTYPE>
  <html>
    <head>
      <meta charset="UTF-8" />
      <title> JavaScript Guessing Game </title>
    </head>
    <body>
      <h1>Guess the Number!</h1>
      <p>I am thinking of a number between 1 and 100. Can you guess what it is?
</p>
       <label for="guess">My Guess</label>
      <input type="number" id="guess">
      <input type="submit" id="submitGuess" value="Check My Guess">
       <p id="feedback"></p>
    </body>
  </html>
```

The `<h1>` and `<p>` elements let the browser know what type of text to display on the page. The set of `<h1>` tags signifies that the text between those two tags (`Guess the Number!`) is a heading. The set of `<p>` tags that follow signify that the short block of text with the instructions is a paragraph. The empty set of `<p>` tags at the end of this code block serve as a placeholder for the feedback the game will give the player based on their guess.

The <script> tag

There are many ways to include JavaScript in a web page, but for a short script like this one, you can use a set of `<script>` tags and write the JavaScript directly in the HTML file. Those `<script>` tags should go right before the closing `</body>` tag near the end of the HTML file.

Now, you can start to write your JavaScript between these two script tags. The final file looks like this:

```
<!DOCTYPE>
<html>
<head>
```

```
  <meta charset="UTF-8" />
  <title> JavaScript Guessing Game </title>
</head>
<body>
  <h1>Guess the Number!</h1>
  <p>I am thinking of a number between 1 and 100. Can you guess what it is?</p>
  <form>
    <label for="guess">My Guess</label>
    <input type="number" id="guess">
    <input type="submit" id="submitGuess" value="Check My Guess">
  </form>
  <p id="feedback"></p>
  <script>
    const randomNumber = Math.floor(Math.random() * 100) + 1
    console.log('Random Number', randomNumber)
    function checkGuess() {
      let myGuess = guess.value
      if (myGuess === randomNumber) {
        feedback.textContent = "You got it right!"
      } else if (myGuess > randomNumber) {
        feedback.textContent = "Your guess was " + myGuess + ". That's too high.
Try Again!"
      } else if (myGuess < randomNumber) {
        feedback.textContent = "Your guess was " + myGuess + ". That's too low.
Try Again!"
      }
    }
    submitGuess.addEventListener('click', checkGuess)
  </script>
</body>
</html>
```

To run this in the browser, either double-click on the file or go to the menu in your favorite web browser and choose **File > Open File**. (If you are using Brackets, you can also use the lightning-bolt symbol in the corner to open the file in the browser).

Pseudo-random number generation

The first step in the guessing game is to generate a number for the player to guess. JavaScript includes several built-in global objects that help you write code. To generate your random number, use the Math object.

Math has properties and functions for working with mathematical concepts in JavaScript. You will use two Math functions to generate the random number for your player to guess.

Start with Math.random(), which generates a pseudo-random number between 0 and 1. (Math.random is inclusive of 0 but exclusive of 1. This means that the function could generate a zero, but it will never generate a 1.)

For this game, set the random number between 1 and 100 to narrow down the player's options. Take the decimal you just generated and multiply it by 100 to produce a decimal between 0 and...not quite 100. But you'll take care of that in a few more steps.

Right now, your number is still a decimal, and you want it to be a whole number. For that, you can use another function that is part of the Math object, Math.floor(). Math.floor()'s purpose is to return the largest integer that is less than or equal to the number you give it as an argument—which means it rounds down to the nearest whole number:

```
Math.floor(Math.random() * 100)
```

That leaves you with a whole number between 0 and 99, which isn't quite the range you want. You can fix that with your last step, which is to add 1 to the result. Voila! Now you have a (somewhat) randomly generated number between 1 and 100:

```
Math.floor(Math.random() * 100) + 1
```

Variables

Now you need to store the randomly generated number so that you can compare it to your player's guesses. To do that, you can assign it to a **variable**.

JavaScript has different types of variables you can choose, depending on how you want to use the variable. For this game, use const and let.

- **let** is used for variables if their value can change throughout the code.
- **const** is used for variables if their value should not be changed.

There's a little more to const and let, but this is all you need to know for now.

The random number is generated only once in the game, so you will use a const variable to hold the value. You want to give the variable a name that makes it clear what value is being stored, so name it `randomNumber`:

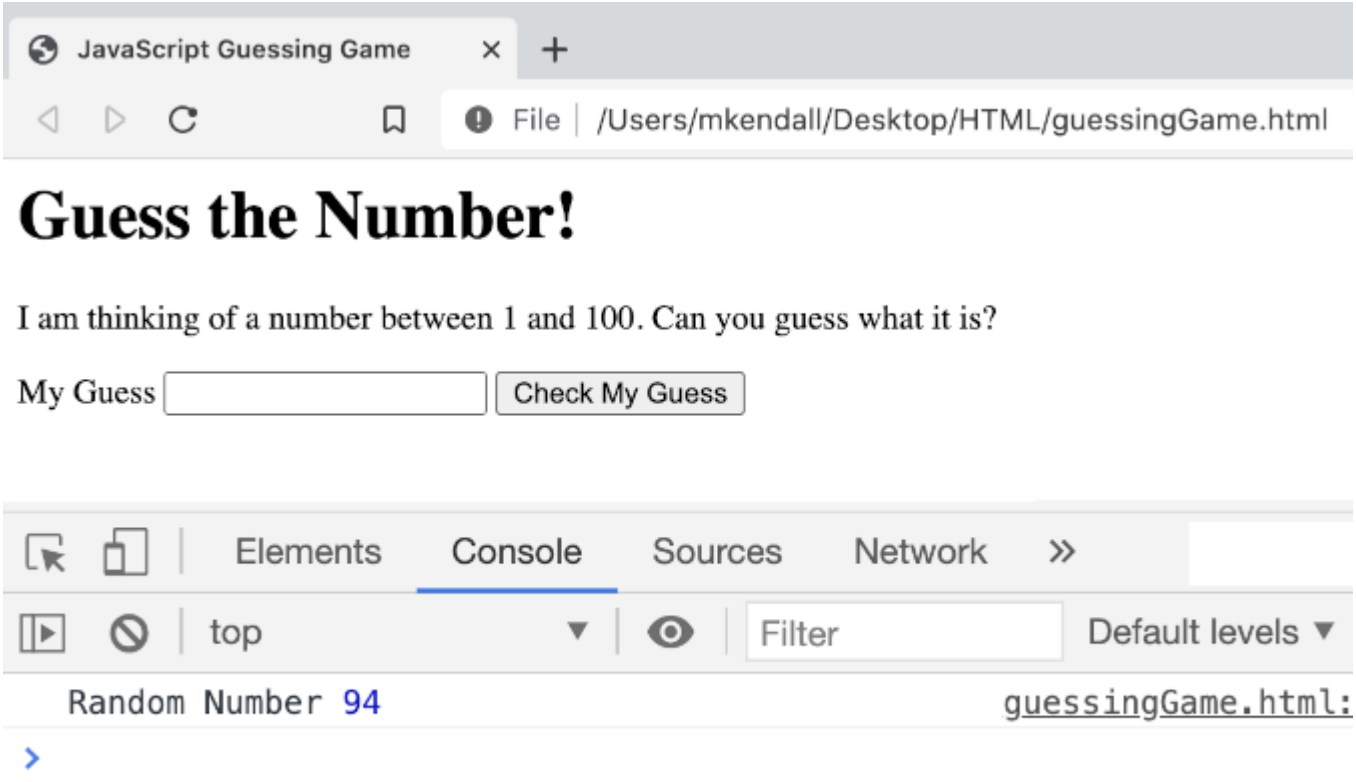
```
const randomNumber
```

A note on naming: Variables and function names in JavaScript are written in camel case. If there is just one word, it is written in all lower case. If there is more than one word, the first word is all lower case, and any additional words start with a capital letter with no spaces between the words.

Logging to the console

Normally, you don't want to show anyone the random number, but developers may want to know the number that was generated to use it to help debug the code. With JavaScript, you can use another built-in function, console.log(), to output the number to the console in your browser.

Most browsers include Developer Tools that you can open by pressing the **F12** key on your keyboard. From there, you should see a tab labeled **Console**. Any information logged to the console will appear here. Since the code you have written so far will run as soon as the browser loads, if you look at the console, you should see the random number that you just generated! Hooray!



Functions

Next, you need a way to get the player's guess from the number input field, compare it to the random number you just generated, and give the player feedback to let them know if they guessed correctly. To do that, write a function. A **function** is code that is grouped to perform a task. Functions are reusable, which means if you need to run the same code multiple times, you can call the function instead of rewriting all of the steps needed to perform the task.

Depending on the JavaScript version you are using, there are many different ways to write, or declare, a function. Since this is an introduction to the language, declare your function using the basic function syntax.

Start with the keyword `function` and then give the function a name. It's good practice to use a name that is an action that describes what the function does. In this case, you are checking the player's guess, so an appropriate name for this function would be `checkGuess`. After the function name, write a set of parentheses and then a set of curly braces. You will write the body of the function between these curly braces:

```
function checkGuess() {}
```

Access the DOM

One of the purposes of JavaScript is to interact with HTML on a webpage. It does this through the Document Object Model (DOM), which is an object JavaScript uses to access and change the information on a web page. Right now, you need to get the player's guess from the number input field you set up in the HTML. You can do that using the `id` attribute you assigned to the HTML elements, which in this case is `guess`:

```
<input type="number" id="guess">
```

JavaScript can get the number the player enters into the number input field by accessing its value. You can do this by referring to the element's id and adding `.value` to the end. This time, use a `let` variable to hold the value of the user's guess:

```
let myGuess = guess.value
```

Whatever number the player enters into the number input field will be assigned to the `myGuess` variable in the `checkGuess` function.

Conditional statements

The next step is to compare the player's guess with the random number the game generates. You also want to give the player feedback to let them know if their guess was too high, too low, or correct.

You can decide what feedback the player will receive by using a series of conditional statements. A **conditional statement** checks to see if a condition is met before running a code block. If the condition is not met, the code stops, moves on to check the next condition, or continues with the rest of the code without running the code in the conditional block:

```
if (myGuess === randomNumber){
  feedback.textContent = "You got it right!"
}
else if(myGuess > randomNumber) {
  feedback.textContent = "Your guess was " + myGuess + ". That's too high. Try
Again!"
}
else if(myGuess < randomNumber) {
  feedback.textContent = "Your guess was " + myGuess + ". That's too low. Try
Again!"
}
```

The first conditional block compares the player's guess to the random number the game generates using a comparison operator `===`. The comparison operator checks the value on the right, compares it to the value on the left, and returns the boolean `true` if they match and `false` if they don't.

If the number matches (yay!), make sure the player knows. To do this, manipulate the DOM by adding text to the `<p>` tag that has the id attribute "feedback." This works just like `guess.value` above, except instead of getting information from the DOM, it changes the information in it. `<p>` elements don't have a value like `<input>` elements—they have text instead, so use `.textContent` to access the element and set the text you want to display:

```
feedback.textContent = "You got it right!"
```

Of course, there is a good chance that the player didn't guess right on the first try, so if `myGuess` and `randomNumber` don't match, give the player a clue to help them narrow down their guesses. If the first conditional fails, the code will skip the code block in that `if` statement and check to see if the next condition is true. That brings you to your `else if` blocks:

```
else if(myGuess > randomNumber) {
  feedback.textContent = "Your guess was " + myGuess + ". That's too high. Try
Again!"
}
```

If you were to read this as a sentence, it might be something like this: "If the player's guess is equal to the random number, let them know they got it right. Otherwise (else), check if the player's guess is greater than `randomNumber`, and if it is, display the player's guess and tell them it was too high."

The last possibility is that the player's guess was lower than the random number. To check that, add one more `else if` block:

```
else if(myGuess < randomNumber) {
  feedback.textContent = "Your guess was " + myGuess + ". That's too low. Try
Again!"
}
```

User events and event listeners

If you look at your script, you'll see that some of the code runs automatically when the page loads, but some of it does not. You want to generate the random number before the game is played, but you don't want to check the player's guess until they have entered it into the number input field and are ready to check it.

The code to generate the random number and log it to the console is outside of a function, so it will run automatically when the browser loads your script. However, for the code inside your function to run, you have to call it.

There are several ways to call a function. Here, you want the function to run when the player clicks on the "Check My Guess" button. Clicking a button creates a user event, which the JavaScript code can then "listen" for so that it knows when it needs to run a function.

The last line of code adds an event listener to the button to "listen" for when the button is clicked. When it "hears" that event, it will run the function assigned to the event listener:

```
submitGuess.addEventListener('click', checkGuess)
```

Just like the other instances where you access DOM elements, you can use the button's id to tell JavaScript which element to interact with. Then you can use the built-in `addEventListener` function to tell JavaScript what event to listen for.

You have already seen a function that takes parameters, but take a moment to look at how this works. Parameters are information that a function needs to perform its task. Not all functions need parameters, but the `addEventListener` function needs two. The first parameter it takes is the name of the user event for which it will listen. The user can interact with the DOM in many ways, like typing, moving the mouse, tabbing with the keyboard, or copying and pasting text. In this case, the user event you are listening for is a button click, so the first parameter will be `click`.

The second piece of information `addEventListener` needs is the name of the function to run when the user clicks the button. In this case, it's the `checkGuess` function.

Now, when the player presses the "Check My Guess" button, the `checkGuess` function will get the value they entered in the number input field, compare it to the random number, and display feedback in the browser to let the player know how they did. Awesome! Your game is ready to play.

Learn JavaScript for fun and profit

This bit of Vanilla JavaScript is just a small taste of what this vast ecosystem has to offer. It's a language well worth investing time into learning, and I encourage you to continue to dig in and learn more.

Write your first JavaScript code

By Seth Kenlon

JavaScript is a programming language full of pleasant surprises. Many people first encounter JavaScript as a language for the web. There's a JavaScript engine in all the major browsers, there are popular frameworks such as JQuery, Cash, and Bootstrap to help make web design easier, and there are even programming environments written in JavaScript. It seems to be everywhere on the internet, but it turns out that it's also a useful language for projects like Electron, an open source toolkit for building cross-platform desktop apps with JavaScript.

JavaScript is a surprisingly multipurpose language with a wide assortment of libraries for much more than just making websites. Learning the basics of the language is easy, and it's a gateway to building whatever you imagine.

Install JavaScript

As you progress with JavaScript, you may find yourself wanting advanced JavaScript libraries and runtimes. When you're just starting, though, you don't have to install JavaScript at all. All major web browsers include a JavaScript engine to run the code. You can write JavaScript using your favorite text editor, load it into your web browser, and see what your code does.

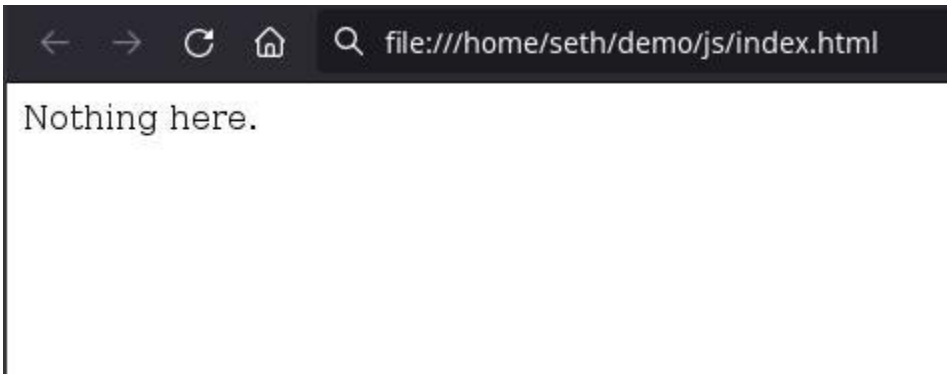
Get started with JavaScript

To write your first JavaScript code, open your favorite text editor, such as Notepad++, Atom, or VSCode. Because it was developed for the web, JavaScript works well with HTML, so first, just try some basic HTML:

```
<html>
  <head>
    <title>JS</title>
  </head>
```

```
  <body>
    <p id="example">Nothing here.</p>
  </body>
</html>
```

Save the file, and then open it in a web browser.

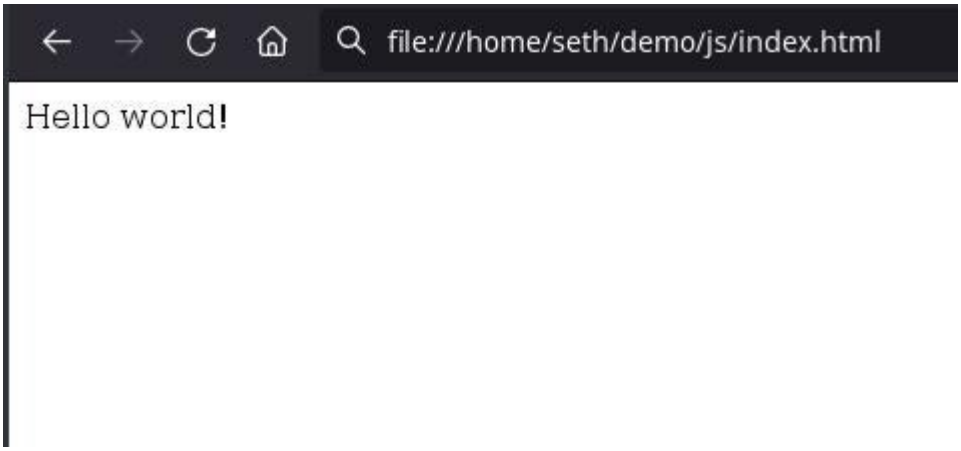


(Seth Kenlon, CC BY-SA 4.0)

To add JavaScript to this simple HTML page, you can either create a JavaScript file and refer to it in the page's `head` or just embed your JavaScript code in the HTML using the `<script>` tag. In this example, I embed the code:

```
<html>
  <head>
    <title>JS</title>
  </head>
  <body>
    <p id="example">Nothing here.</p>
    <script>
      let myvariable = "Hello world!";
      document.getElementById("example").innerHTML = myvariable;
    </script>
  </body>
</html>
```

Reload the page in your browser.



(Seth Kenlon, CC BY-SA 4.0)

As you can see, the <p> tag as written still contains the string "Nothing here," but when it's rendered, JavaScript alters it so that it contains "Hello world" instead. Yes, JavaScript has the power to rebuild (or just help build) a webpage.

The JavaScript in this simple script does two things. First, it creates a variable called `myvariable` and places the string "Hello world!" into it. Finally, it searches the current document (the web page as the browser is rendering it) for any HTML element with the ID `example`. When it locates `example`, it uses the `innerHTML` function to replace the contents of the HTML element with the contents of `myvariable`.

Of course, using a custom variable isn't necessary. It's just as easy to populate the HTML element with something being dynamically created. For instance, you could populate it with a timestamp:

```
<html>
  <head>
    <title>JS</title>
  </head>
  <body>
    <p id="example">Date and time appears here.</p>
    <script>
      document.getElementById("example").innerHTML = Date();
    </script>
  </body>
</html>
```

Reload the page to see a timestamp generated at the moment the page is rendered. Reload a few times to watch the seconds increment.

JavaScript syntax

In programming, **syntax** refers to the rules of how sentences (or "lines") are written. In JavaScript, each line of code must end in a semicolon (`;`) so that the JavaScript engine running your code understands when to stop reading.

Words (or "strings") must be enclosed in quotation marks (`"`), while numbers (or "integers") go without.

Almost everything else is a convention of the JavaScript language, such as variables, arrays, conditional statements, objects, functions, and so on.

Creating variables in JavaScript

Variables are containers for data. You can think of a variable as a box where you can put data to share with your program. Creating a variable in JavaScript is done with two keywords you choose based on how you intend to use the variable: `let` and `var`. The `var` keyword denotes a variable intended for your entire program to use, while `let` creates variables for specific purposes, usually inside functions or loops.

JavaScript's built-in `typeof` function can help you identify what kind of data a variable contains. Using the first example, you can find out what kind of data `myvariable` contains by modifying the displayed text to:

```
<string>
let myvariable = "Hello world!";
document.getElementById("example").innerHTML = typeof(myvariable);
</string>
```

This renders "string" in your web browser because the variable contains "Hello world!" Storing different kinds of data (such as an integer) in `myvariable` would cause a different data type to be printed to your sample web page. Try changing the contents of `myvariable` to your favorite number and then reloading the page.

Creating functions in JavaScript

Functions in programming are self-contained data processors. They're what makes programming *modular*. It's because functions exist that programmers can write generic libraries that, for instance, resize images or keep track of the passage of time for other programmers (like you) to use in their own code.

You create a function by providing a custom name for your function followed by any amount of code enclosed within braces.

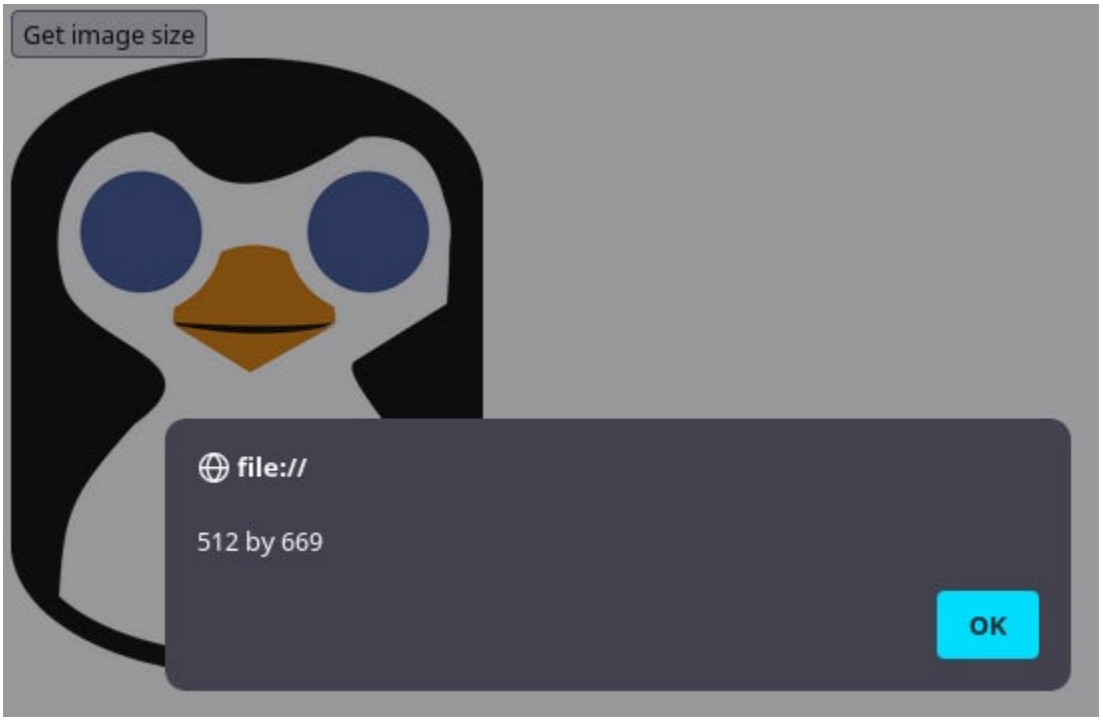
Here's a simple web page featuring a resized image and a button that analyzes the image and returns the true image dimensions. In this example code, the `<button>` HTML element uses the built-in JavaScript function `onclick` to detect user interaction, which triggers a custom function called `get_size`:

```
<html>
  <head>
    <title>Imager</title>
  </head>
  <body>
    <div>
      <button onclick="get_size(document.getElementById('myimg'))">
        Get image size
    </button>
    </div>

    <div>
      <img style="width: 15%" id="myimg" src="penguin.png" />
    </div>
    <script>
      function get_size(i) {
        let w = i.naturalWidth;
        let h = i.naturalHeight;
        alert(w + " by " + h);
      }
    </script>

  </body>
</html>
```

Save the file and load it into your web browser to try the code.



(Seth Kenlon, CC BY-SA 4.0)

Cross-platform apps with JavaScript

You can see from the code sample how JavaScript and HTML work closely together to create a cohesive user experience. This is one of the great strengths of JavaScript. When you write code in JavaScript, you inherit one of the most common user interfaces of modern computing regardless of platform: the web browser. Your code is cross-platform by nature, so your application, whether it's just a humble image size analyzer or a complex image editor, video game, or whatever else you dream up, can be used by everyone with a web browser (or a desktop, if you deliver an Electron app).

Learning JavaScript is easy and fun. There are lots of websites with tutorials available. There are also over a million JavaScript libraries to help you interface with devices, peripherals, the Internet of Things, servers, file systems, and lots more. And as you're learning, keep our **JavaScript cheat sheet** close by so you remember the fine details of syntax and structure.

Create a JavaScript API in 6 minutes

By Jessica Cherry

This article demonstrates creating a base API with Express and JavaScript. Express is a NodeJS minimalist web framework. This combination allows for minimal effort to get an API up and running at the speed of light. If you have six minutes of free time, you can get this API working to do something useful.

Get started with NodeJS

What you need for this project is the NodeJS version of your choice. In this example, I use NodeJS and HTTPie for testing, a web browser, and a terminal. Once you have those available, you're ready to start. Let's get this show on the road!

Set up a project directory and install the tools to get started:

```
$ mkdir test-api
```

The `npm init` command creates the package JSON for our project below. Type `npm init` and press enter several times. The output is shown below:

```
$ npm init
Press ^C at any time to quit.
package name: (test-api)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
```

```
license: (ISC)
About to write to /Users/cherrybomb/test-api/package.json:
{
  "name": "test-api",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
Is this OK? (yes)
```

This utility walks you through creating a `package.json` file. It only covers the most common items, and tries to guess sensible defaults. See `npm help init` for definitive documentation on these fields and exactly what they do.

Use `npm install {pkg}` afterward to install a package and save it as a dependency in the `package.json` file.

Next, install Express using the npm CLI:

```
$ npm install express
npm WARN cherrybomb No description
npm WARN cherrybomb No repository field.
npm WARN cherrybomb No license field.
+ express@4.18.1
added 60 packages from 39 contributors and audited 136 packages in 4.863s
16 packages are looking for funding
  run `npm fund` for details
found 0 vulnerabilities
```

Finally, create your source directory and your `index.js` file, which is where the application code lives:

```
$ mkdir src
$ touch src/index.js
```

Time to code!

Code an API

For your first act of coding, make a simple "hello world" API call. In your `index.js` file, add the code snippet below:

```
const express = require('express')
const app = express()
const port = 5000
app.get('/', (req, res) => {
  res.send('Hello World!')
})
app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

Each of these constant variables is available in the scopes below. Because you're not using the following scopes within the code, these constants are used without too much extra thought.

When you call `app.get`, you define the `GET{rest article needed}` endpoint to a forward slash. This also sets the "hello world" response.

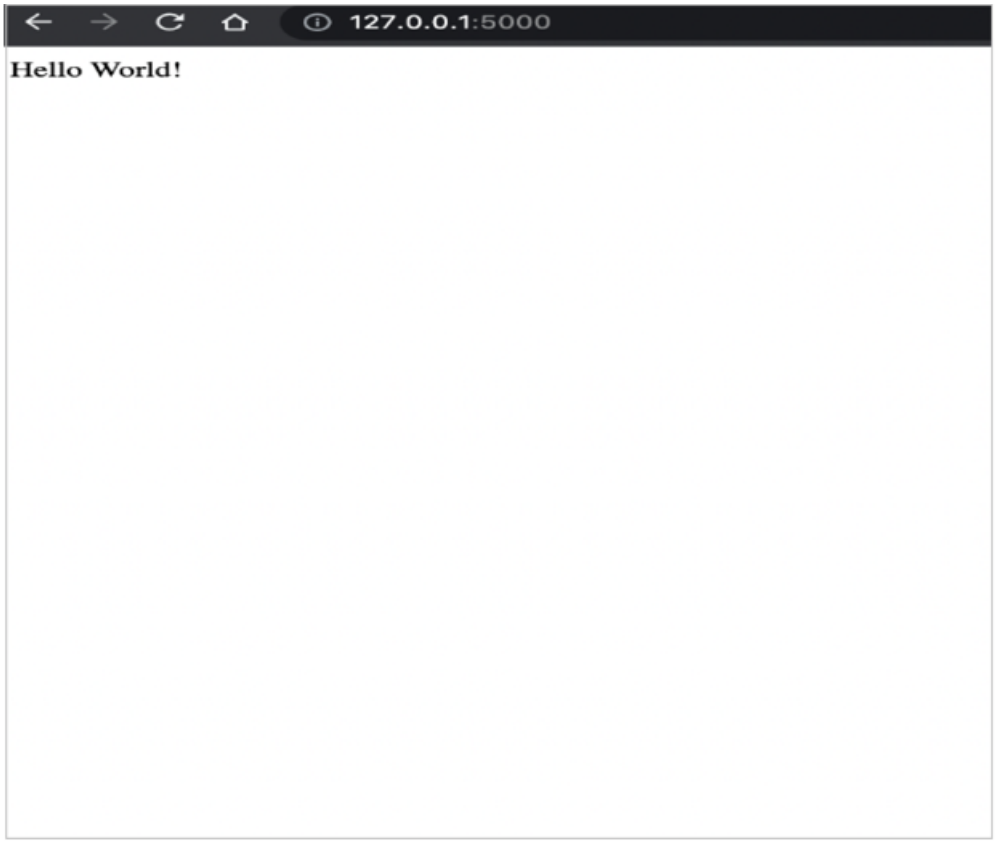
Finally, in the last section, you will start your app on port 5000. The output on your terminal shows your defined message in a file called `console.log`.

To start your application, run the following command, and see the output as shown:

```
$ node ./src/index.js
Example app listening on port 5000
```

Test the API

Now that everything is up and running, make a simple call to ensure your API works. For the first test, just open a browser window and navigate to `localhost:5000`.



(Jessica Cherry, CC BY-SA 4.0)

Next, check out what HTTPie says about the API call:

```
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 12
Content-Type: text/html; charset=utf-8
Date: Tue, 21 Jun 2022 14:31:06 GMT
ETag: W/"c-Lve95gjOVATpfV8EL5X4nxwjKHE"
Keep-Alive: timeout=5
X-Powered-By: Express
Hello World!
```

And there you have it! One whole working API call. So what's next? Well, you could try some changes to make it more interesting.

Make your API fun

The "hello world" piece is now done, so it's time to do some cool math. You'll do some counts instead of just "hello world."

Change your code to look like this:

```
const express = require('express')
const app = express()
const port = 5000
let count = 0;
app.get('/api', (req, res) => {
res.json({count})
})
app.post('/api', (req, res) => {
++count;
res.json({count});
});
app.listen(port, () => {
console.log(`Example app listening on port ${port}`)
})
```

Aside from a `GET` command in your code, you now have a `POST` to make some changes to your count. With count defined as **`0`**, the `LET` command allows changes to the `COUNT` variable.

In `app.get`, you get the count, and in `app.post`, you **`++count`**, which counts upwards in increments of 1. When you rerun the `GET`, you receive the new number.

Try out the changes:

```
test-api → node ./src/index.js
Example app listening on port 5000
```

Next, use HTTPie to run the `GET` and `POST` operations for a test to confirm it works. Starting with `GET`, you can grab the count:

```
test-api → http GET 127.0.0.1:5000/api
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 11
Content-Type: application/json; charset=utf-8
Date: Tue, 21 Jun 2022 15:23:06 GMT
ETag: W/"b-ch7MNww9+xUYoTgutbGr6VU0GaU"
Keep-Alive: timeout=5
X-Powered-By: Express
{
```

```
    "count": 0
}
```

Then do a `POST` a couple of times, and watch the changes:

```
test-api → http POST 127.0.0.1:5000/api
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 11
Content-Type: application/json; charset=utf-8
Date: Tue, 21 Jun 2022 15:28:28 GMT
ETag: W/"b-qA97yBec1rrOyf2eVsYdWwFPOso"
Keep-Alive: timeout=5
X-Powered-By: Express
{ "count": 1 }
test-api → http POST 127.0.0.1:5000/api
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 11
Content-Type: application/json; charset=utf-8
Date: Tue, 21 Jun 2022 15:28:34 GMT
ETag: W/"b-hRuIfkAGnfwKvpTzajm4bAWdKxE"
Keep-Alive: timeout=5
X-Powered-By: Express
{ "count": 2 }
```

As you can see, the count goes up! Run one more `GET` operation and see what the output is:

```
test-api → http GET 127.0.0.1:5000/api
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 11
Content-Type: application/json; charset=utf-8
Date: Tue, 21 Jun 2022 15:29:41 GMT
ETag: W/"b-hRuIfkAGnfwKvpTzajm4bAWdKxE"
Keep-Alive: timeout=5
X-Powered-By: Express
{ "count": 2 }
```

The end and the beginning

I specialize in infrastructure and Terraform, so this was a really fun way to learn and build something quickly in a language I'd never used before. JavaScript moves fast, and it can be annoying to see errors that seem obscure or obtuse. I can see where some personal opinions have judged it harshly as a language, but it's a strong and useful tool. I hope you enjoyed this walkthrough and learned something new and cool along the way.

How much JavaScript do you need to know before learning ReactJS?

By Sachin Samal

React is a UI framework built on top of HTML, CSS, and JavaScript, where JavaScript (JS) is responsible for most of the logic. If you have knowledge of variables, data types, array functions, callbacks, scopes, string methods, loops, and other JS DOM manipulation-related topics, these will tremendously speed up the pace of learning ReactJS.

Your concept of modern JavaScript will dictate the pace of how soon you can get going with ReactJS. You don't need to be a JavaScript expert to start your ReactJS journey, but just as knowledge of ingredients is a must for any chef hoping to master cooking, the same is true for learning ReactJS. It's a modern JavaScript UI library, so you need to know some JavaScript. The question is, how much?

Example explanation

Suppose I'm asked to write an essay about a "cow" in English, but that I know nothing about the language. In this case, for me to successfully complete the task, I should not only have an idea about the topic but also the specified language.

Assuming that I acquire some knowledge about the topic (a cow), how can I calculate the amount of English I need to know to be able to write about the proscribed topic? What if I have to write an essay on some other complex topics in English?

It’s difficult to figure that out, isn’t it? I don’t know what things I'm going to write about the topic, but it could be anything. So to get started, I have to have a proper knowledge of the English language, but it doesn't end there.

Extreme reality

The same is true for the amount of JavaScript required before getting started with ReactJS. According to my example scenario, ReactJS is the topic "cow" while JavaScript is the English language. It's important to have a strong grasp of JavaScript to be successful in ReactJS. One is very unlikely to master ReactJS professionally without having the proper foundation of JavaScript. No matter how much knowledge I might have about the topic, I won't be able to express myself properly if I don't know the fundamentals of the language.

How much is enough?

In my experience, when you start your ReactJS journey, you should already be familiar with:

- variables
- data types
- string methods
- loops
- conditionals

You should be familiar with these specifically in JavaScript. But these are just the bare minimum prerequisites. When you try to create a simple React app, you'll inevitably need to handle events. So, the concept of normal functions, function expressions, statements, arrow function, the difference between an arrow function and a regular function, and the lexical scoping of `this` keyword in both types of function is really important.

But the question is, what if I have to create a complex app using ReactJS?

Get inspired

Handling events, spread operators, destructuring, named imports, and default imports in JavaScript will help you understand the working mechanism of React code.

Most importantly, you must understand the core concepts behind JavaScript itself. JavaScript is asynchronous by design. Don't be surprised when code appearing at the bottom of a file executes before code at the top of the file does. Constructs like promises, callbacks, async-await, map, filter, and reduce, are the most common methods and concepts in ReactJS, especially when developing complex applications.

The main idea is to be good in JavaScript so you can reduce the complexity of your ReactJS journey.

Getting good

It's easy for me to say what you need to know, but it's something else entirely for you to go learn it. Practicing a lot of JavaScript is essential, but you might be surprised that I don't think it means you necessarily have to wait until you master it. There are certain concepts that are important beforehand, but there's a lot you can learn as you go. Part of practicing is learning, so you get started with JavaScript and even with some of the basics of React, as long as you move at a comfortable pace and understand that doing your "homework" is a requirement before you attempt anything serious.

Get started with JavaScript now

Don't bother waiting until you cover all aspects of JavaScript. That's never going to happen. If you do that, you'll get trapped in that forever-loop of learning JavaScript. And you all know how constantly evolving and rapidly changing the tech field is. If you want to start learning JavaScript, try reading Mandy Kendall's introductory article Learn JavaScript by writing a guessing game. It's a great way to get started quickly, and once you see what's possible I think you're likely to find it difficult to stop.

Code your first React UI app

By Jessica Cherry

Who wants to create their first UI app? I do, and if you're reading this article, I assume you do, too. In today's example, I'll use some JavaScript and the API with Express I demonstrated in my previous article. First, let me explain some of the tech you're about to use.

What is React?

React is a JavaScript library for building a user interface (UI). However, you need more than just the UI library for a functional UI. Here are the important components of the JavaScript web app you're about to create:

- npx: This package is for executing npm packages.
- axios: A promise-based HTTP client for the browser and node.js. A *promise* is a value that an API endpoint will provide.
- http-proxy-middleware: Configures proxy middleware with ease. A proxy is middleware that helps deal with messaging back and forth from the application endpoint to the requester.

Preconfiguration

If you haven't already, look at my previous article. You'll use that code as part of this React app. In this case, you'll add a service to use as part of the app. As part of this application, you have to use the npx package to create the new folder structure and application:

```
$ npx create-react-app count-ui
npx: installed 67 in 7.295s
Creating a new React app in /Users/cherrybomb/count-ui.
Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...
[...]
```

```
Installing template dependencies using npm...
+ @testing-library/jest-dom@5.16.4
+ @testing-library/user-event@13.5.0
+ web-vitals@2.1.4
+ @testing-library/react@13.3.0
added 52 packages from 109 contributors in 9.858s
[...]
Success! Created count-ui at /Users/cherrybomb/count-ui
[...]
We suggest that you begin by typing:
  cd count-ui
  npm start
```

As you can see, the `npx` command has created a new template with a folder structure, an awesome `README` file, and a Git repository. Here's the structure:

```
$ cd count-ui/
/Users/cherrybomb/count-ui
$ ls -A -1
.git
.gitignore
README.md
node_modules
package-lock.json
package.json
public
src
```

This process also initialized the Git repo and set the branch to master, which is a pretty cool trick. Next, install the `npm` packages:

```
$ npm install axios http-proxy-middleware
[...]
npm WARN @apideck/better-ajv-errors@0.3.4 requires a peer of ajv@>=8 but none is
installed. You must install peer dependencies yourself.
+ http-proxy-middleware@2.0.6
+ axios@0.27.2
added 2 packages from 2 contributors, updated 1 package and audited 1449 packages
in 5.886s
```

Now that those are set up, add your `services`, and `main.js` file:

```
$ mkdir src/services
src/services
$ touch src/services/main.js
```

Preconfiguration is now complete, so you can now work on coding.

Code a UI from start to finish

Now that you have everything preconfigured, you can put together the service for your application. Add the following code to the `main.js` file:

```
import axios from 'axios';
const baseURL = 'http://localhost:5001/api';
export const get = async () => await axios.get(`${baseURL}/`);
export const increment = async () => await axios.post(`${baseURL}/`);
export default {
    get,
    increment
}
```

This process creates a JavaScript file that interacts with the API you created in my previous article.

Set up the proxy

Next, you must set up a proxy middleware by creating a new file in the `src` directory.

```
$ touch src/setupProxy.js
```

Configure the proxy with this code in `setupProxy.js`:

```
const { createProxyMiddleware } = require('http-proxy-middleware');
module.exports = function(app) {
  app.use(
    '/api',
    createProxyMiddleware({
      target: 'http://localhost:5000',
      changeOrigin: true,
    })
  );
};
```

In this code, the `app.use` function specifies the service to use as `/api` when connecting to the existing API project. However, nothing defines `api` in the code. This is where a proxy comes in. With a proxy, you can define the `api` function on the proxy level to interact with your Express API. This middleware registers requests between both applications because the

UI and API use the same host with different ports. They require a proxy to transfer internal traffic.

JavaScript imports

In your base `src` directory, you see that the original template created an `App.js`, and you must add `main.js` (in the `services` directory) to your imports in the `App.js` file. You also need to import React on the very first line, as it is external to the project:

```
import React from 'react'
import main from './services/main';
```

Add the rendering function

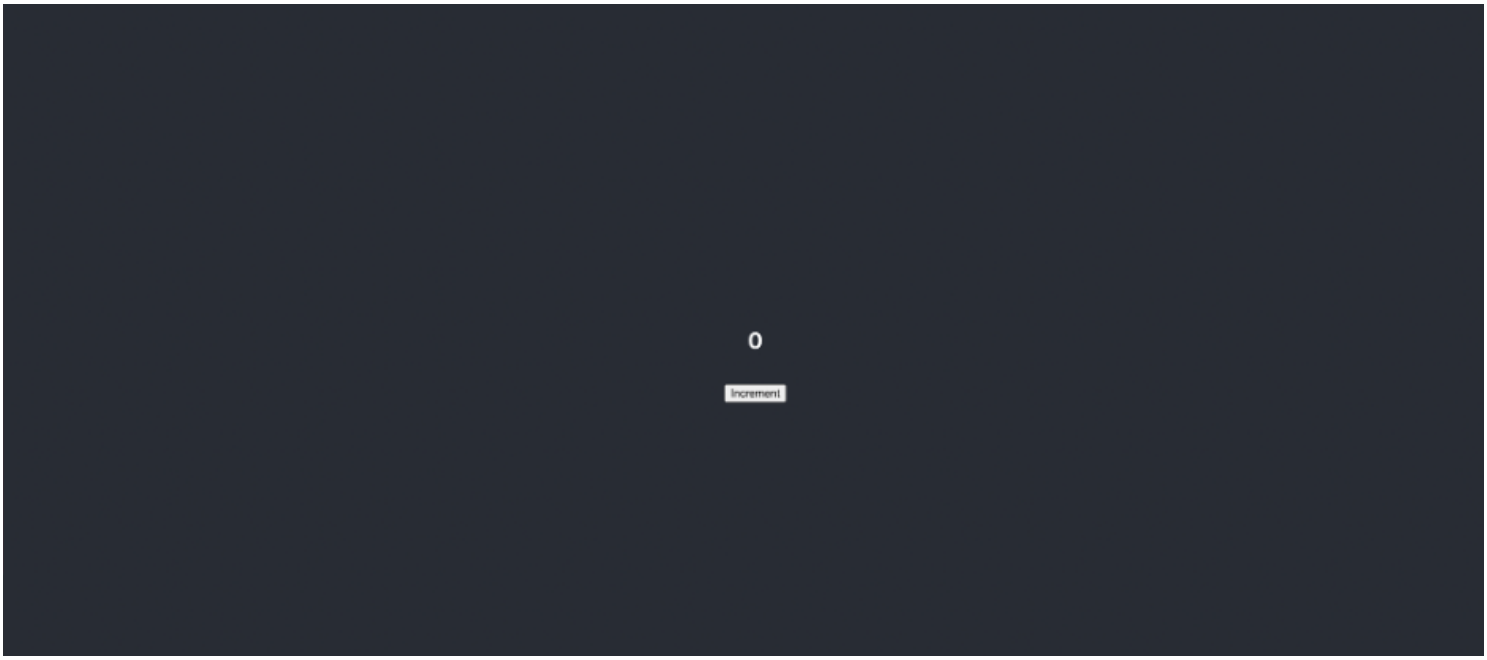
Now that you have your imports, you must add a render function. In the **App()** function of `App.js`, add the first section of definitions for **react** and **count** before the **return** section. This section gets the **count** from the API and puts it on the screen. In the **return** function, a button provides the ability to increment the count.

```
function App() {
const [count, setCount] = React.useState(0);
React.useEffect(()=>{
  async function fetchCount(){
    const newCount = (await main.get()).data.count;
    setCount(newCount);
  }
  fetchCount();
}, [setCount]);
return (
    <div className="App">
      <header className="App-header">
        <h4>
          {count}
        </h4>
        <button onClick={async ()=>{
          setCount((await main.increment()).data.count);
        }}>
          Increment
        </button>
      </header>
    </div>
  );
}
```

To start and test the app, run `npm run start`. You should see the output below. Before running the application, confirm your API is running from the Express app by running node `./src/index.js`.

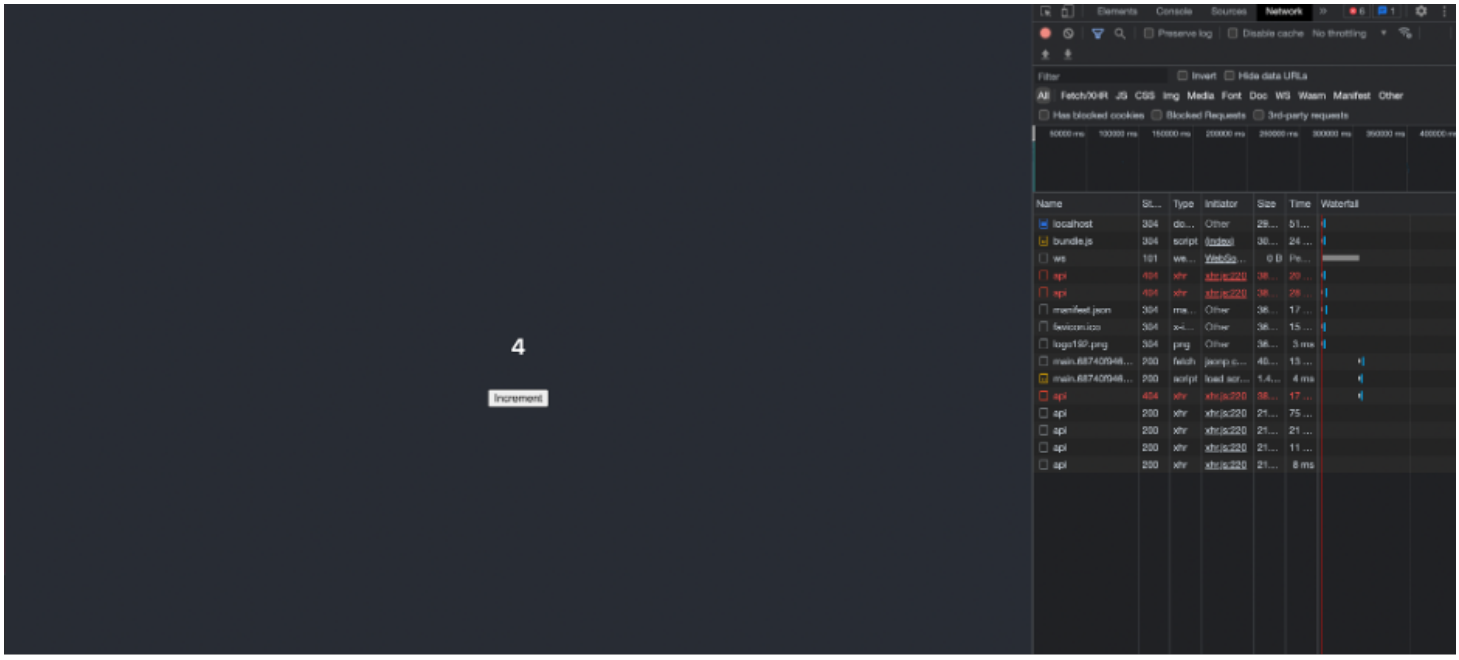
```
$ npm run start
> count-ui@0.1.0 start /Users/cherrybomb/count-ui
> react-scripts start
[HPM] Proxy created: /  -> http://localhost:5000
(node:71729) [DEP_WEBPACK_DEV_SERVER_ON_AFTER_SETUP_MIDDLEWARE]
DeprecationWarning: 'onAfterSetupMiddleware' option is deprecated. Please use the
'setupMiddlewares' option.
(Use `node --trace-deprecation ...` to show where the warning was created)
(node:71729) [DEP_WEBPACK_DEV_SERVER_ON_BEFORE_SETUP_MIDDLEWARE]
DeprecationWarning: 'onBeforeSetupMiddleware' option is deprecated. Please use
the 'setupMiddlewares' option.
Starting the development server...
```

Once everything is running, open your browser to `localhost:5000` to see the front end has a nice, admittedly minimal, page with a button:



(Jessica Cherry, CC BY-SA 4.0)

What happens when you press the button? (Or, in my case, press the button several times.)



(Jessica Cherry, CC BY-SA 4.0)

The counter goes up!

Congratulations, you now have a React app that uses your new API.

Web apps and APIs

This exercise is a great way to learn how to make a back end and a front end work together. It's noteworthy to say that if you're using two hosts, you don't need the proxy section of this article. Either way, JavaScript and React are a quick, templated way to get a front end up and running with minimal steps. Hopefully, you enjoyed this walk-through. Tell us your thoughts on learning how to code in JavaScript.

4 steps to set up global modals in React

By Ajay Pratap

A modal dialog is a window that appears on top of a web page and requires a user's interaction before it disappears. React has a couple of ways to help you generate and manage modals with minimal coding.

If you create them within a **local scope**, you must import modals into each component and then create a state to manage each modal's opening and closing status.

By using a **global state**, you don't need to import modals into each component, nor do you have to create a state for each. You can import all the modals in one place and use them anywhere.

In my opinion, the best way to manage modal dialogs in your React application is globally by using a React context rather than a local state.

How to create global modals

Here are the steps (and code) to set up global modals in React. I'm using Patternfly as my foundation, but the principles apply to any project.

1. Create a global modal component

In a file called **GlobalModal.tsx**, create your modal definition:

```
import React, { useState, createContext, useContext } from 'react';
import { CreateModal, DeleteModal,UpdateModal } from './components';
export const MODAL_TYPES = {
CREATE_MODAL:"CREATE_MODAL",
 DELETE_MODAL: "DELETE_MODAL",
 UPDATE_MODAL: "UPDATE_MODAL"
```

```
};
const MODAL_COMPONENTS: any = {
 [MODAL_TYPES.CREATE_MODAL]: CreateModal,
 [MODAL_TYPES.DELETE_MODAL]: DeleteModal,
 [MODAL_TYPES.UPDATE_MODAL]: UpdateModal
};
type GlobalModalContext = {
 showModal: (modalType: string, modalProps?: any) => void;
 hideModal: () => void;
 store: any;
};
const initalState: GlobalModalContext = {
 showModal: () => {},
 hideModal: () => {},
 store: {},
};
const GlobalModalContext = createContext(initalState);
export const useGlobalModalContext = () => useContext(GlobalModalContext);
export const GlobalModal: React.FC<{}> = ({ children }) => {
 const [store, setStore] = useState();
 const { modalType, modalProps } = store || {};
 const showModal = (modalType: string, modalProps: any = {}) => {
   setStore({
     ...store,
     modalType,
     modalProps,
   });
 };
 const hideModal = () => {
   setStore({
     ...store,
     modalType: null,
     modalProps: {},
   });
 };
 const renderComponent = () => {
   const ModalComponent = MODAL_COMPONENTS[modalType];
   if (!modalType || !ModalComponent) {
     return null;
   }
   return <ModalComponent id="global-modal" {...modalProps} />;
 };
 return (
   <GlobalModalContext.Provider value={{ store, showModal, hideModal }}>
     {renderComponent()}
     {children}
   </GlobalModalContext.Provider>
 );
```

```
};
```

In this code, all dialog components are mapped with the modal type. The `showModal` and `hideModal` functions are used to open and close dialog boxes, respectively.

The `showModal` function takes two parameters: `modalType` and `modalProps`. The `modalProps` parameter is optional; it is used to pass any type of data to the modal as a prop.

The `hideModal` function doesn't have any parameters; calling it causes the current open modal to close.

2. Create modal dialog components

In a file called **CreateModal.tsx**, create a modal:

```
import React from "react";
import { Modal, ModalVariant, Button } from "@patternfly/react-core";
import { useGlobalModalContext } from "../GlobalModal";
export const CreateModal = () => {
 const { hideModal, store } = useGlobalModalContext();
 const { modalProps } = store || {};
 const { title, confirmBtn } = modalProps || {};
 const handleModalToggle = () => {
   hideModal();
 };
 return (
   <Modal
     variant={ModalVariant.medium}
     title={title || "Create Modal"}
     isOpen={true}
     onClose={handleModalToggle}
     actions={[
       <Button key="confirm" variant="primary" onClick={handleModalToggle}>
         {confirmBtn || "Confirm button"}
       </Button>,
       <Button key="cancel" variant="link" onClick={handleModalToggle}>
         Cancel
       </Button>
     ]}
   >
     Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
     tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
     veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
     commodo consequat. Duis aute irure dolor in reprehenderit in voluptate
     velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
     cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id
     est laborum.
```

```
   </Modal>
 );
};
```

This has a custom hook, `useGlobalModalContext`, that provides store object from where you can access all the props and the functions `showModal` and `hideModal`. You can close the modal by using the `hideModal` function.

To delete a modal, create a file called **DeleteModal.tsx**:

```
import React from "react";
import { Modal, ModalVariant, Button } from "@patternfly/react-core";
import { useGlobalModalContext } from "../GlobalModal";
export const DeleteModal = () => {
 const { hideModal } = useGlobalModalContext();
 const handleModalToggle = () => {
   hideModal();
 };
 return (
   <Modal
     variant={ModalVariant.medium}
     title="Delete Modal"
     isOpen={true}
     onClose={handleModalToggle}
     actions={[
       <Button key="confirm" variant="primary" onClick={handleModalToggle}>
         Confirm
       </Button>,
       <Button key="cancel" variant="link" onClick={handleModalToggle}>
         Cancel
       </Button>
     ]}
   >
     Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod...
     cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id
     est laborum.
   </Modal>
 );
};
```

To update a modal, create a file called **UpdateModal.tsx** and add this code:

```
import React from "react";
import { Modal, ModalVariant, Button } from "@patternfly/react-core";
import { useGlobalModalContext } from "../GlobalModal";
export const UpdateModal = () => {
 const { hideModal } = useGlobalModalContext();
```

```
 const handleModalToggle = () => {
   hideModal();
 };
 return (
   <Modal
     variant={ModalVariant.medium}
     title="Update Modal"
     isOpen={true}
     onClose={handleModalToggle}
     actions={[
       <Button key="confirm" variant="primary" onClick={handleModalToggle}>
         Confirm
       </Button>,
       <Button key="cancel" variant="link" onClick={handleModalToggle}>
         Cancel
       </Button>
     ]}
   >
     Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod
     tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim...
     cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id
     est laborum.
   </Modal>
 );
};
```

3. Integrate GlobalModal into the top-level component in your application

To integrate the new modal structure you've created into your app, you just import the global modal class you've created. Here's my sample **App.tsx** file:

```
import "@patternfly/react-core/dist/styles/base.css";
import "./fonts.css";
import { GlobalModal } from "./components/GlobalModal";
import { AppLayout } from "./AppLayout";
export default function App() {
 return (
   <GlobalModal>
     <AppLayout />
   </GlobalModal>
 );
}
```

App.tsx is the top-level component in your app, but you can add another component according to your application's structure. However, make sure it is one level above where you want to access modals.

`GlobalModal` is the root-level component where all your modal components are imported and mapped with their specific `modalType`.

4. Select the modal's button from the AppLayout component

Adding a button to your modal with **AppLayout.js**:

```
import React from "react";
import { Button, ButtonVariant } from "@patternfly/react-core";
import { useGlobalModalContext, MODAL_TYPES } from "./components/GlobalModal";
export const AppLayout = () => {
 const { showModal } = useGlobalModalContext();
 const createModal = () => {
   showModal(MODAL_TYPES.CREATE_MODAL, {
     title: "Create instance form",
     confirmBtn: "Save"
   });
 };
 const deleteModal = () => {
   showModal(MODAL_TYPES.DELETE_MODAL);
 };
 const updateModal = () => {
   showModal(MODAL_TYPES.UPDATE_MODAL);
 };
 return (
   <>
     <Button variant={ButtonVariant.primary} onClick={createModal}>
       Create Modal
     </Button>
     <br />
     <br />
     <Button variant={ButtonVariant.primary} onClick={deleteModal}>
       Delete Modal
     </Button>
     <br />
     <br />
     <Button variant={ButtonVariant.primary} onClick={updateModal}>
       Update Modal
     </Button>
   </> );
};
```

There are three buttons in the AppLayout component: create modal, delete modal, and update modal. Each modal is mapped with the corresponding `modalType`: `CREATE_MODAL`, `DELETE_MODAL`, or `UPDATE_MODAL`.

Use global dialogs

Global modals are a clean and efficient way to handle dialogs in React. They are also easier to maintain in the long run. The next time you set up a project, keep these tips in mind.

If you'd like to see the code in action, I've included the complete application I created for this article in a sandbox.

How I build command-line apps in JavaScript

By Seth Kenlon and Ramakrishna Pattnaik

JavaScript is a language developed for the web, but its usefulness has gone far beyond just the Internet. Thanks to projects like Node.js and Electron, JavaScript is as much a general-purpose scripting language as a browser component. There are JavaScript libraries specially designed to build command-line interfaces. Yes, you can run JavaScript in your terminal.

Now, when you enter a command into your terminal, there are generally options, also called *switches* or *flags*, that you can use to modify how the command runs. This is a useful convention defined by the POSIX specification, so as a programmer, it's helpful to know how to detect and parse the options. To get this functionality from JavaScript, it's useful to use a library designed to make it easy to build command-line interfaces. My favorite is Commander.js. It's easy, it's flexible, and it's intuitive.

Installing node

To use the Commander.js library, you must have Node.js installed. On Linux, you can install Node using your package manager. For example, on Fedora, CentOS, Mageia, and others:

```
$ sudo dnf install nodejs
```

On Windows and macOS, you can download installers from the nodejs.org website.

Installing Commander.js

To install Commander.js, use the `npm` command:

```
$ npm install commander
```

Adding a library to your JavaScript code

In JavaScript, you can use the `require` keyword to include (or import, if you're used to Python) a library into your code. Create a file called `example.js` and open it in your favorite text editor. Add this line to the top to include the Commander.js library:

```
const { program } = require('commander');
```

Option parsing in JavaScript

The first thing you must do to parse options is to define the valid options your application can accept. The Commander.js library lets you define both short and long options, along with a helpful message to clarify the purpose of each.

```
program
  .description('A sample application to parse options')
  .option('-a, --alpha', 'Alpha')
  .option('-b, --beta <VALUE>', 'Specify a VALUE', 'Foo');
```

The first option, which I've called `--alpha` (`-a` for short), is a Boolean switch: It either exists or it doesn't. It takes no arguments. The second option, which I've called `--beta` (`-b` for short), accepts an argument and even specifies a default value when you've provided nothing.

Accessing the command-line data

Once you've defined valid options, you can reference the values using the long option name:

```
program.parse();
const options = program.opts();
console.log('Options detected:');
if (options.alpha) console.log('alpha');
const beta = !options.beta ? 'no' : options.beta;
console.log('beta is: %s', beta);
```

Run the application

Try running it with the `node` command, first with no options:

```
$ node ./example.js
Options detected:
beta is: Foo
```

The default value for `beta` gets used in the absence of an override from the user.

Run it again, this time using the options:

```
$ node ./example.js --beta hello --alpha
Options detected:
alpha
beta is: hello
```

This time, the test script successfully detected the option `--alpha`, and the `--beta` option with the value provided by the user.

Option parsing

Here's the full demonstration code for your reference:

```
const { program } = require('commander');
program
  .description('A sample application to parse options')
  .option('-a, --alpha', 'Alpha')
    .option('-b, --beta <VALUE>', 'Specify a VALUE', 'Foo');
program.parse();
const options = program.opts();
console.log('Options detected:');
console.log(typeof options);
if (options.alpha) console.log(' * alpha');
const beta = !options.beta ? 'no' : options.beta;
console.log(' * beta is: %s', beta);
```

There are further examples in the project's Git repository.

Including options for your users is an important feature for any application, and Commander.js makes it easy to do. There are other libraries aside from Commander.js, but I find this one easy and quick to use. What's your favorite JavaScript command-line builder?

165+ JavaScript terms you need to know

By Sachin Samal

JavaScript is a rich language, with sometimes a seemingly overwhelming number of libraries and frameworks. With so many options available, it's sometimes useful to just look at the language itself and keep in mind its core components. This glossary covers the core JavaScript language, syntax, and functions.

JavaScript variables

`var`: The most used variable. Can be reassigned but only accessed within a function, meaning function scope. Variables defined with `var` move to the top when code is executed.

`const`: Cannot be reassigned and not accessible before they appear within the code, meaning block scope.

`let`: Similar to `const` with block scope, however, the `let` variable can be reassigned but not re-declared.

Data types

Numbers: `var age = 33`

Variables: `var a`

Text (strings): `var a = "Sachin"`

Operations: `var b = 4 + 5 + 6`

True or false statements: `var a = true`

Constant numbers: `const PI = 3.14`

Objects: `var fullName = {firstName:"Sachin", lastName: "Samal"}`

Objects

This is a simple example of objects in JavaScript. This object describe the variable `car`, and includes *keys* or *properties* such as `make`, `model`, and `year` are the object's property names. Each property has a value, such as `Nissan`, `Altima`, and `2022`. A JavaScript object is a collection of properties with values, and it functions as a method.

```
var car = {
make:"Nissan",
model:"Altima",
year:2022,
};
```

Comparison operators

==: Is equal to

===: Is equal value and equal type

!=: Is not equal

!==: Is not equal value or not equal type

>: Is greater than

<: Is less than

>=: Is greater than or equal to

<=: Is less than or equal to

?: Ternary operator

Logical operators

&&: Logical AND

||: Logical OR

!: Logical NOT

Output data

`alert()`: Output data in an alert box in the browser window

`confirm()`: Open up a yes/no dialog and return true/false depending on user click

`console.log()`: Write information to the browser console. Good for debugging.

`document.write()`: Write directly to the HTML document

`prompt()`: Create a dialog for user input

Array methods

Array: An object that can hold multiple values at once.

`concat()`: Join several arrays into one

`indexOf()`: Return the primitive value of the specified object

`join()`: Combine elements of an array into a single string and return the string

`lastIndexOf()`: Give the last position at which a given element appears in an array

`pop()`: Remove the last element of an array

`push()`: Add a new element at the end

`reverse()`: Sort elements in descending order

`shift()`: Remove the first element of an array

`slice()`: Pull a copy of a portion of an array into a new array

`splice()`: Add positions and elements in a specified way

`toString()`: Convert elements to strings

`unshift()`: Add a new element to the beginning

`valueOf()`: Return the first position at which a given element appears in an array

JavaScript loops

Loops: Perform specific tasks repeatedly under applied conditions.

```
for (before loop; condition for loop; execute after loop) {
// what to do during the loop
}
```

`for`: Creates a conditional loop

`while`: Sets up conditions under which a loop executes at least once, as long as the specified condition is evaluated as true

`do while`: Similar to the `while` loop, it executes at least once and performs a check at the end to see if the condition is met. If it is, then it executes again

`break`: Stop and exit the cycle at certain conditions

`continue`: Skip parts of the cycle if certain conditions are met

if-else statements

An `if` statement executes the code within brackets as long as the condition in parentheses is true. Failing that, an optional `else` statement is executed instead.

```
if (condition) {
// do this if condition is met
} else {
// do this if condition is not met
}
```

Strings

String methods

`charAt()`: Return a character at a specified position inside a string

`charCodeAt()`: Give the Unicode of the character at that position

`concat()`: Concatenate (join) two or more strings into one

`fromCharCode()`: Return a string created from the specified sequence of UTF-16 code units

`indexOf()`: Provide the position of the first occurrence of a specified text within a string

`lastIndexOf()`: Same as `indexOf()` but with the last occurrence, searching backwards

`match()`: Retrieve the matches of a string against a search pattern

`replace()`: Find and replace specified text in a string

`search()`: Execute a search for a matching text and return its position

`slice()`: Extract a section of a string and return it as a new string

`split()`: Split a string object into an array of strings at a specified position

`substr()`: Extract a substring depended on a specified number of characters, similar to `slice()`

`substring()`: Can't accept negative indices, also similar to `slice()`

`toLowerCase()`: Convert strings to lower case

`toUpperCase()`: Convert strings to upper case

`valueOf()`: Return the primitive value (that has no properties or methods) of a string object

Number methods

`toExponential()`: Return a string with a rounded number written as exponential notation

`toFixed()`: Return the string of a number with a specified number of decimals

`toPrecision()`: String of a number written with a specified length

`toString()`: Return a number as a string

`valueOf()`: Return a number as a number

Math methods

`abs(a)`: Return the absolute (positive) value of `a`

`acos(x)`: Arccosine of `x`, in radians

`asin(x)`: Arcsine of `x`, in radians

`atan(x)`: Arctangent of `x` as a numeric value

`atan2(y,x)`: Arctangent of the quotient of its arguments

`ceil(a)`: Value of a rounded up to its nearest integer

`cos(a)`: Cosine of `a` (`x` is in radians)

`exp(a)`: Value of Ex

`floor(a)`: Value of a rounded down to its nearest integer

`log(a)`: Natural logarithm (base E) of `a`

`max(a,b,c…,z)`: Return the number with the highest value

`min(a,b,c…,z)`: Return the number with the lowest value

`pow(a,b)`: `a` to the power of `b`

`random()`: Return a random number between 0 and 1

`round(a)`: Value of `a` rounded to its nearest integer

`sin(a)`: Sine of `a` (`a` is in radians)

`sqrt(a)`: Square root of `a`

`tan(a)`: Tangent of an angle

Dealing with dates in JavaScript

Set dates

`Date()`: Create a new date object with the current date and time

`Date(2022, 6, 22, 4, 22, 11, 0)`: Create a custom date object. The numbers represent year, month, day, hour, minutes, seconds, milliseconds. You can omit anything except for year and month.

`Date("2022-07-29")`: Date declaration as a string

Pull date and time values

`getDate()`: Day of the month as a number (1-31)

`getDay()`: Weekday as a number (0-6)

`getFullYear()`: Year as a four-digit number (yyyy)

`getHours()`: Hour (0-23)

`getMilliseconds()`: Millisecond (0-999)

`getMinutes()`: Minute (0-59)

`getMonth()`: Month as a number (0-11)

`getSeconds()`: Second (0-59)

`getTime()`: Milliseconds since January 1, 1970

`getUTCDate()`: Day (date) of the month in the specified date according to universal time (also available for day, month, full year, hours, minutes, etc.)

`parse`: Parse a string representation of a date and return the number of milliseconds since January 1, 1970

Set part of a date

`setDate()`: Set the day as a number (1-31)

`setFullYear()`: Set the year (optionally month and day)

`setHours()`: Set the hour (0-23)

`setMilliseconds()`: Set milliseconds (0-999)

`setMinutes()`: Set the minutes (0-59)

`setMonth()`: Set the month (0-11)

`setSeconds()`: Set the seconds (0-59)

`setTime()`: Set the time (milliseconds since January 1, 1970)

`setUTCDate()`: Set the day of the month for a specified date according to universal time (also available for day, month, full year, hours, minutes, etc.)

Dom mode

Node methods

`appendChild()`: Add a new child node to an element as the last child node

`cloneNode()`: Clone an HTML element

`compareDocumentPosition()`: Compare the document position of two elements

`getFeature()`: Return an object which implements the APIs of a specified feature

`hasAttributes()`: Return **true** if an element has any attributes, otherwise **false**

`hasChildNodes()`: Return **true** if an element has any child nodes, otherwise **false**

`insertBefore()`: Insert a new child node before a specified, existing child node

`isDefaultNamespace()`: Return **true** if a specified `namespaceURI` is the default, otherwise **false**

`isEqualNode()`: Check if two elements are equal

`isSameNode()`: Check if two elements are the same node

`isSupported()`: Return **true** if a specified feature is supported on the element

`lookupNamespaceURI()`: Return the `namespaceURI` associated with a given node

`normalize()`: Join adjacent text nodes and removes empty text nodes in an element

`removeChild()`: Remove a child node from an element

`replaceChild()`: Replace a child node in an element

Element methods

`getAttribute()`: Return the specified attribute value of an element node

`getAttributeNS()`: Return string value of the attribute with the specified namespace and name

`getAttributeNode()`: Get the specified attribute node

`getAttributeNodeNS()`: Return the attribute node for the attribute with the given namespace and name

`getElementsByTagName()`: Provide a collection of all child elements with the specified tag name

`getElementsByTagNameNS()`: Return a live HTMLCollection of elements with a certain tag name belonging to the given namespace

`hasAttribute()`: Return **true** if an element has any attributes, otherwise **false**

`hasAttributeNS()`: Provide a true/false value indicating whether the current element in a given namespace has the specified attribute

`removeAttribute()`: Remove a specified attribute from an element

`lookupPrefix()`: Return a `DOMString` containing the prefix for a given `namespaceURI`, if present

`removeAttributeNS()`: Remove the specified attribute from an element within a certain namespace

`removeAttributeNode()`: Take away a specified attribute node and return the removed node

`setAttribute()`: Set or change the specified attribute to a specified value

`setAttributeNS()`: Add a new attribute or changes the value of an attribute with the given namespace and name

`setAttributeNode()`: Set or change the specified attribute node

`setAttributeNodeNS()`: Add a new namespaced attribute node to an element

JavaScript events

Mouse

`onclick`: User clicks on an element

`oncontextmenu`: User right-clicks on an element to open a context menu

`ondblclick`: User double-clicks on an element

`onmousedown`: User presses a mouse button over an element

`onmouseenter`: Pointer moves onto an element

`onmouseleave`: Pointer moves out of an element

`onmousemove`: Pointer moves while it is over an element

`onmouseover`: Pointer moves onto an element or one of its children

`setInterval()`: Call a function or evaluates an expression at

`oninput`: User input on an element

`onmouseup`: User releases a mouse button while over an element

`onmouseout`: User moves the mouse pointer out of an element or one of its children

`onerror`: Happens when an error occurs while loading an external file

`onloadeddata`: Media data is loaded

`onloadedmetadata`: Metadata (like dimensions and duration) is loaded

`onloadstart`: Browser starts looking for specified media

`onpause`: Media is paused either by the user or automatically

`onplay`: Media is started or is no longer paused

`onplaying`: Media is playing after having been paused or stopped for buffering

`onprogress`: Browser is in the process of downloading the media

`onratechange`: Media play speed changes

`onseeked`: User finishes moving/skipping to a new position in the media

`onseeking`: User starts moving/skipping

`onstalled`: Browser tries to load the media, but it is not available

`onsuspend` – Browser is intentionally not loading media

`ontimeupdate`: Play position has changed (e.g., because of fast forward)

`onvolumechange`: Media volume has changed (including mute)

`onwaiting`: Media paused but expected to resume (for example, buffering)