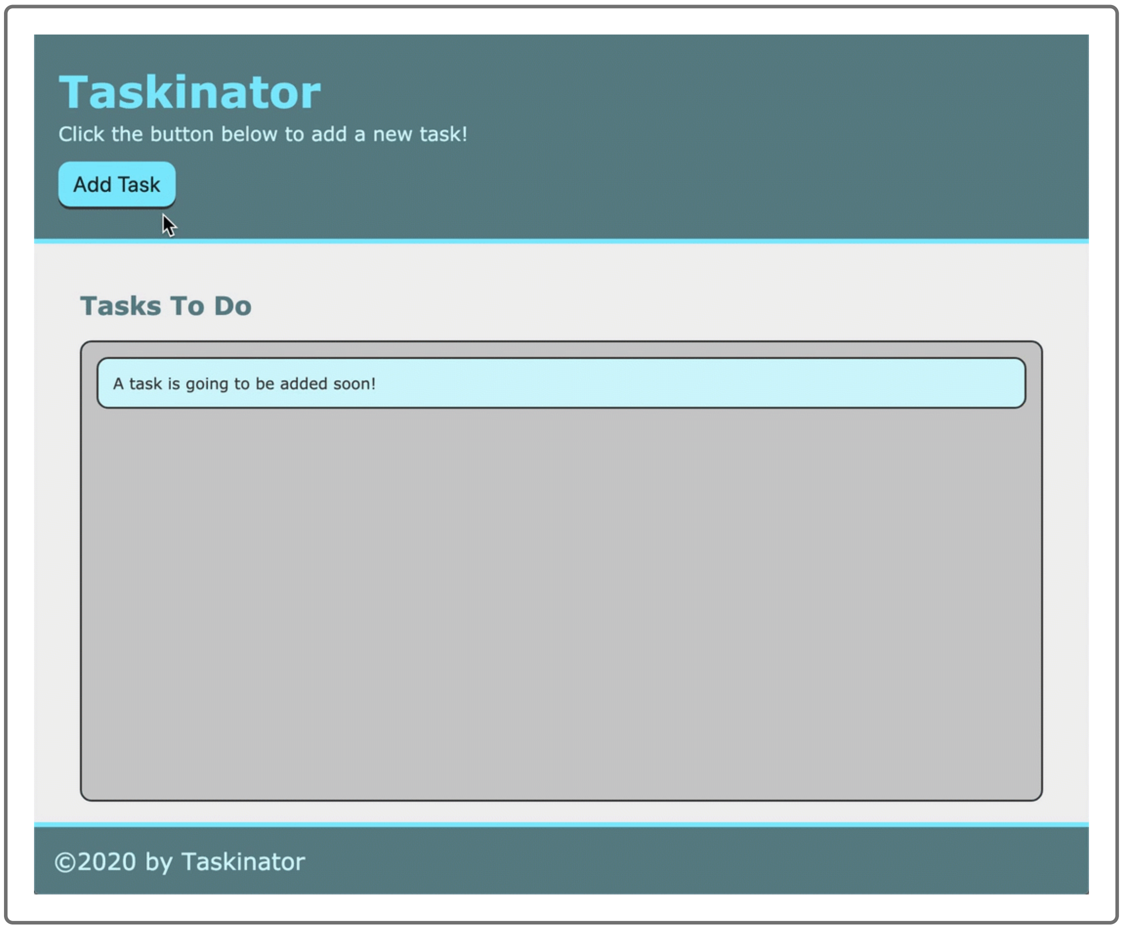
4.1.6 Create a DOM Element

The app looks great so far, but nothing happens if you click the button. This button should add a task to the task list, as shown here in this animation:


Taskinator
Click the button below to add a new task!
Add Task
Tasks To Do
A task is going to be added soon!
©2020 by Taskinator

So how do we get the button to work? Let's answer the following questions:

- What's executing the script file? How does JavaScript know how to run?
- What allows us to use the browser's built-in methods?

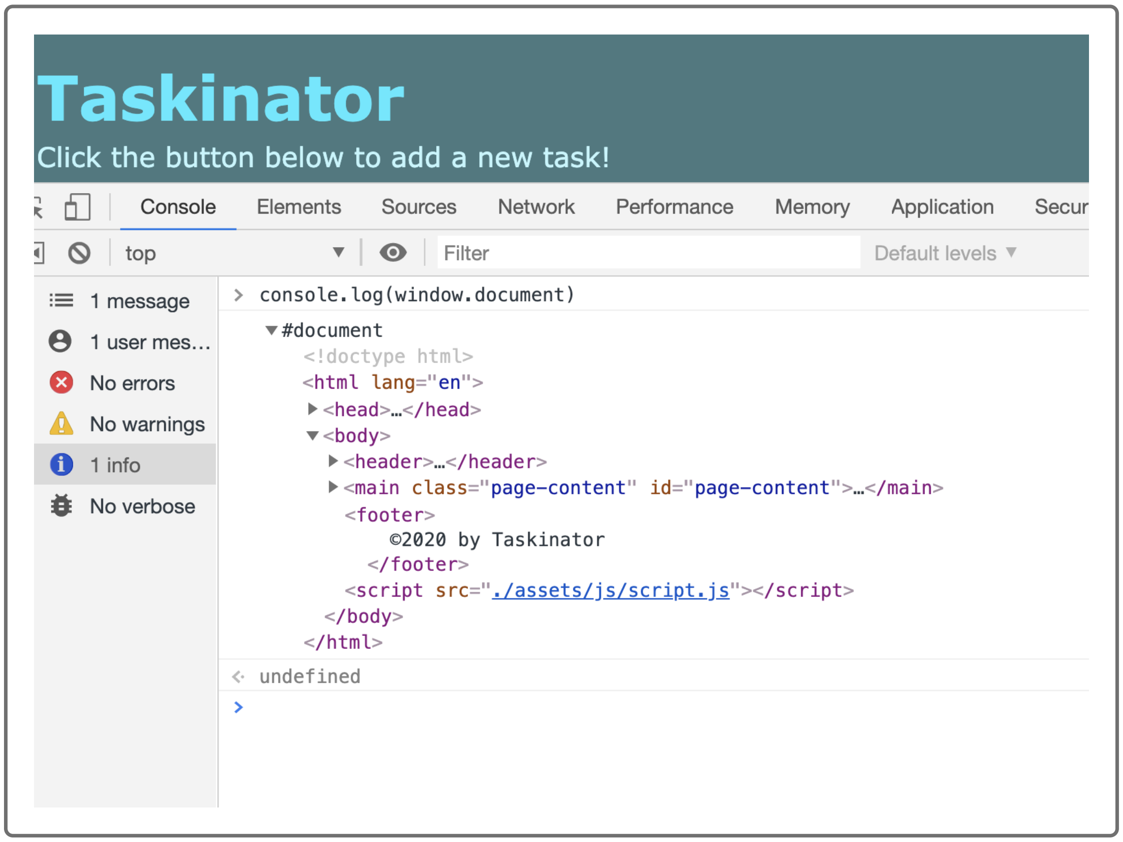
The script runs in the browser, which can interpret JavaScript and execute the code. We've used the browser APIs to access the browser's built-in methods, specifically the `window` object.

We've used functions provided by the browser's `window` object previously, like `alert()` and `prompt()`, but now we need to use some properties that the `window` provides for us, particularly pertaining to the HTML of the application.

Browsers refer to the HTML code that makes up a webpage as a **document**, so let's use our Chrome DevTools Console detective skills to investigate the `document` object. Type the following expression into the browser's console:

```
console.log(window.document);
```

Running this command in the DevTools console tab should display something like the following image:



```
> console.log(window.document)
  ▼#document
     <!doctype html>
     <html lang="en">
      ▶<head>…</head>
      ▼<body>
        ▶<header>…</header>
        ▶<main class="page-content" id="page-content">…</main>
         <footer>
             ©2020 by Taskinator
           </footer>
         <script src="./assets/js/script.js"></script>
       </body>
     </html>
<· undefined
>
```

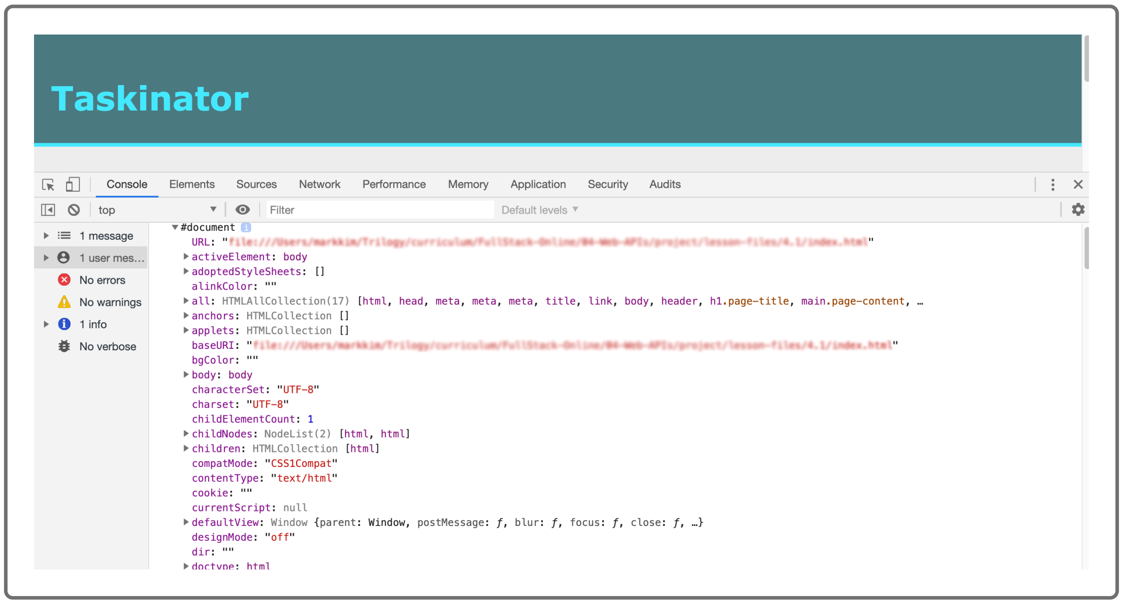
Although this result may look like HTML, it's actually the object representation of the webpage. We call this the **Document Object Model**, or the **DOM**. The `document` object represents the root element or the highest-level element of the webpage, which is the `<html>` element. The rest of the elements are the descendant elements (i.e., children, grandchildren, and so on) of the `document`.

Use the DOM

We can better illustrate the object representation of the element by typing the following in the `script.js` file:

```
console.dir(window.document);
```

Now save the `script.js` file and refresh the `index.html` file in the browser. You can expand the `document` object to see the following in the console window, as this image shows:



```
▼#document
   URL: "[redacted]"
  ▶activeElement: body
  ▶adoptedStyleSheets: []
   alinkColor: ""
  ▶all: HTMLAllCollection(17) [html, head, meta, meta, meta, title, link, body, header, h1.page-title, main.page-content, …
  ▶anchors: HTMLCollection []
  ▶applets: HTMLCollection []
   baseURI: "[redacted]"
   bgColor: ""
  ▶body: body
   characterSet: "UTF-8"
   charset: "UTF-8"
   childElementCount: 1
  ▶childNodes: NodeList(2) [html, html]
  ▶children: HTMLCollection [html]
   compatMode: "CSS1Compat"
   contentType: "text/html"
   cookie: ""
   currentScript: null
  ▶defaultView: Window {parent: Window, postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, …}
   designMode: "off"
   dir: ""
  ▶doctype: html
```

Unlike `console.log`, which displays the element's HTML, `console.dir()` displays the HTML element as an object known as a DOM element. Like with any object in JavaScript, we can access its properties and methods using dot notation. Through the `document` object we can access everything on the webpage, including all the elements and their attributes, text content, metadata, and much more.

DEEP DIVE ▼

Now that we have an object model of the webpage, we can select existing DOM elements, create new elements, use various built-in DOM methods, or create our own functions to provide rich interactive features like drop-down menus, slideshows, drag-and-drop elements, and more.

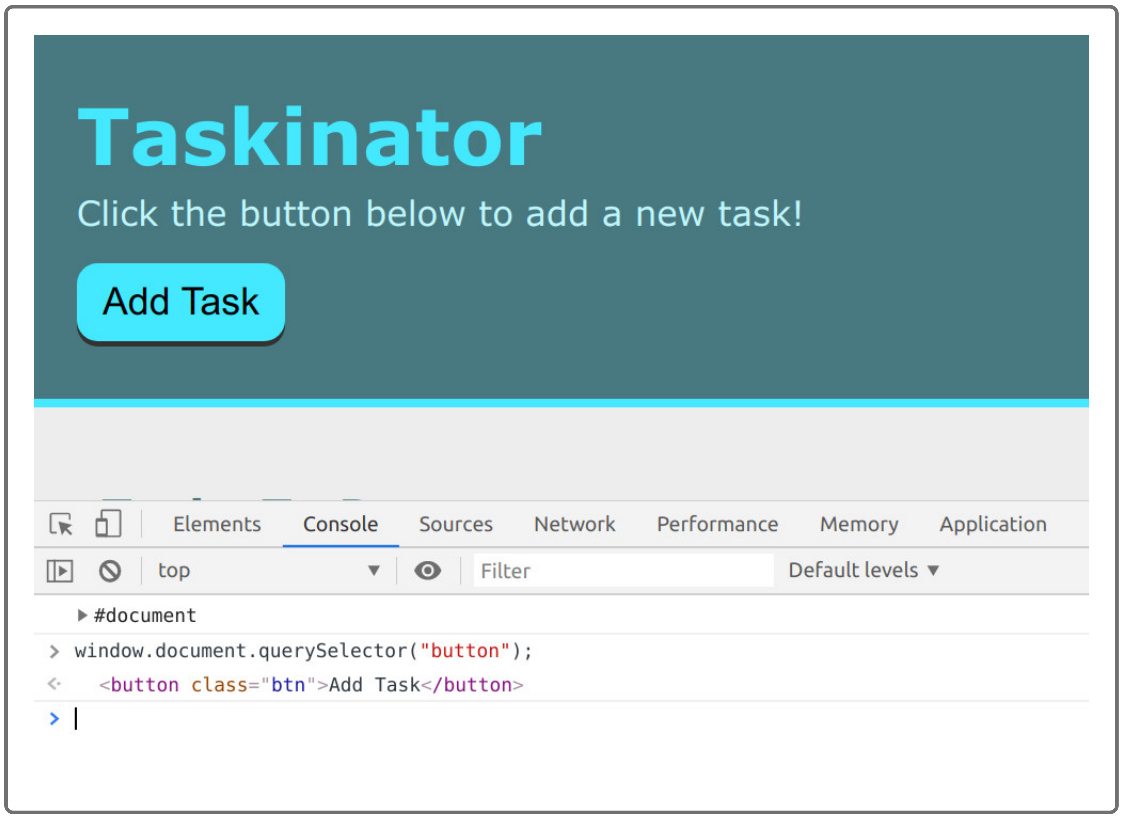
DEEP DIVE ▼

There's not much we can do with the `document` object currently, but the DOM offers various methods for using JavaScript to find specific elements within this `document`.

One method we'll use to find the target element is the `querySelector()`. Let's use this to find the `<button>` and to `console.log` the results. Type the following directly into the console window in the browser of the `index.html` file:

```
window.document.querySelector("button");
```

You should see something like the following image in the console:

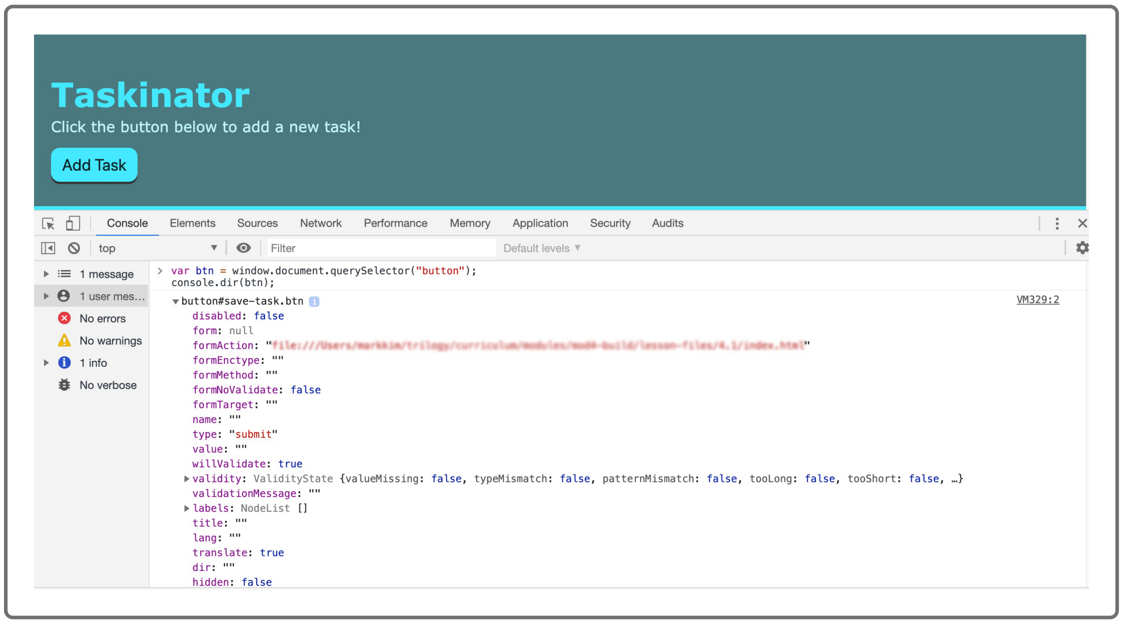


```
▶ #document
> window.document.querySelector("button");
<·   <button class="btn">Add Task</button>
> |
```

The image looks like HTML, but it's an object representation of the `<button>` element. We can verify that this is actually a DOM element by typing the following into the console window of the browser:

```
var btn = window.document.querySelector("button");
console.dir(btn);
```

This will display an object in the console that looks like the following image:



```
> var btn = window.document.querySelector("button");
  console.dir(btn);
  ▼button#save-task.btn                                                    VM329:2
     disabled: false
     form: null
     formAction: "[redacted]"
     formEnctype: ""
     formMethod: ""
     formNoValidate: false
     formTarget: ""
     name: ""
     type: "submit"
     value: ""
     willValidate: true
    ▶validity: ValidityState {valueMissing: false, typeMismatch: false, patternMismatch: false, tooLong: false, tooShort: false, …}
     validationMessage: ""
    ▶labels: NodeList []
     title: ""
     lang: ""
     translate: true
     dir: ""
     hidden: false
```

Why is it important to prefix `document` to the `querySelector()` method? The `document` is the root DOM element that represents the `index.html` file we opened in the browser. Using the method `querySelector()`, we can select any element in the HTML, such as the `<button>`. The `querySelector()` method searches down from the `document` object through all the descendant elements. The versatile `querySelector()` method can search the descendant elements of any DOM element, and can search not just an element type as shown above but also selectors, including any attribute.

Experiment in the console by selecting different elements. Try to target the `<body>` or `<main>`. You'll find that all the elements in HTML are available to target.

Can you select an HTML element by its class attribute value? Doing so requires a little trick.

SHOW HINT

To select a class attribute, you need to add a dot (`.`) prefix, like this:

```
document.querySelector(".btn");
```

Here we chose the class attribute `.btn` on the `<button>` attribute.

REWIND

This is the same syntax we used for CSS class selectors!

The same object will be displayed in the console for the `<button>` element, though we chose a different selector.

Notice that we didn't add `window` before `document` in the expression above. In the `script.js` file and in the browser console, `window` is unnecessary because we opened `index.html` in the browser, where `window` is a global object.

REWIND

This should look familiar—we dropped the `window` prefix with the `Math` object when we used `Math.random()`. Incidentally, `alert()`

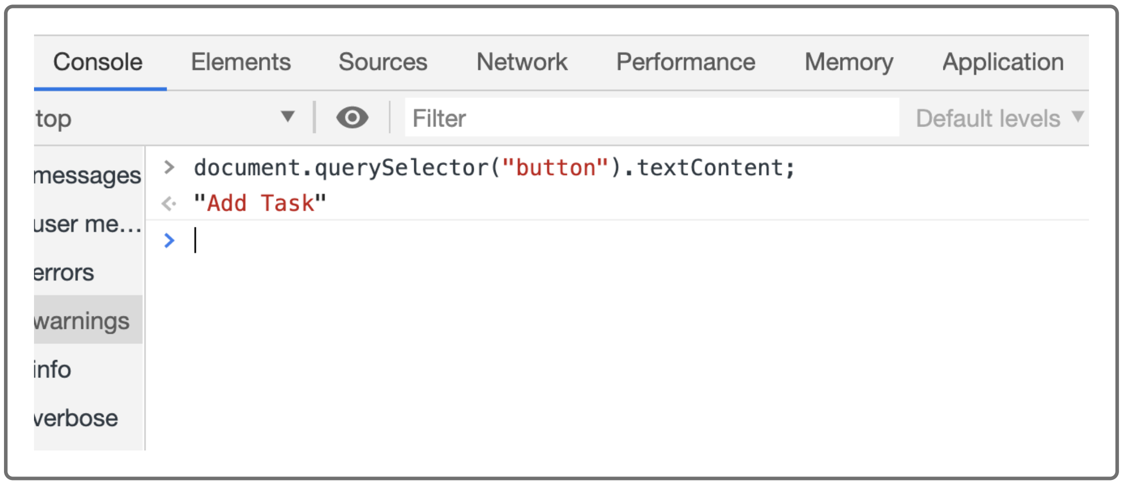
and `prompt()` can also function without the `window` prefix. We'll drop it from here going forward because shortcuts make a developer's life that much sweeter!

Although the display in the console for the `<button>` query looks like raw HTML, it's actually an object representation of this element. This means we have access to built-in properties and methods. One such property is called `textContent`.

Let's type the following into the console:

```
document.querySelector("button").textContent;
```

Here's an image of how this looks in the console:



```
> document.querySelector("button").textContent;
<· "Add Task"
>
```

The result shows that we can use a built-in property of a DOM element, so clearly we're dealing with objects here. The aptly named `textContent` is the property that returns the text content of the element.

Great job! You were able to select the button, but what happens if you start adding more buttons to the page? The `querySelector()` would only be able to find the first button in the document.

So how do you distinguish this button from the rest? You can use a familiar attribute called the `id`.

Let's add the `id` "save-task" attribute to the `button` element in the `index.html` file so that the element looks like this:

```
<button class="btn" id="save-task">Add Task</button>
```

Refresh the browser so that the DOM has the `id` attribute on the element. Then let's update the `querySelector()` call to look for the `id` attribute instead of the generic element. Type the following into the console to see how the result looks:

```
document.querySelector("#save-task");
```

The result should look exactly like the previous result in the console.

Having successfully targeted the HTML element, now we can add this code into the JavaScript file, `script.js`. Delete the `console.dir(window.document);` code that's there from our previous work. Then assign the button element object representation to a variable in the file:

```
var buttonEl = document.querySelector("#save-task");
console.log(buttonEl);
```

The name of the button element is `buttonEl`. Camelcase marks the element as a JavaScript variable. Also, the `El` suffix identifies this as a DOM element. This is a naming convention that will help us keep track of which variables store DOM elements.

SHOW PRO TIP

To see the expression in action, we need to save the script file and refresh the `index.html` in the browser.

IMPORTANT

> Have you tried opening the `script.js` file in the browser and noticed that you can't? Why can't you? Think about the HTML file as the subject matter and the style sheet and script files as the modifiers or enhancements. The HTML file provides a canvas to apply styles and behaviors and will always be the connection to the web browser. The HTML file acts as the hub, connecting the supplemental files with relative file paths.

As we can see in the console, the `buttonEl` variable now represents the same `button` element we displayed earlier. Now that we've selected the correct element in the DOM and preserved the element reference in the script, how can we add a task to the task list with a button click?