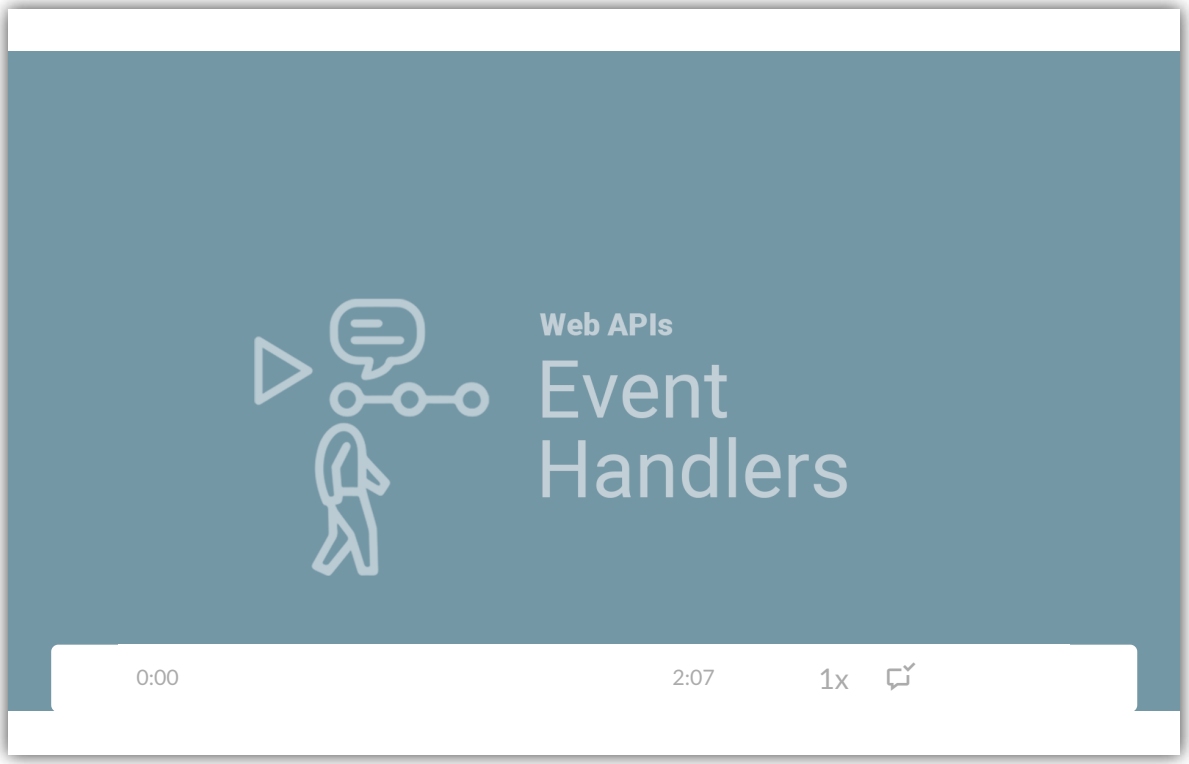
4.2.5 Handle Form Submission

In the last lesson, we attached an event listener to the "Add Task" button so that clicking it triggered a function to create a new task item.

For a refresher on how click events work in JavaScript, watch the following video:



In the Taskinator app, we can keep the click event as-is when it comes to submitting the form. We can set the `createTaskHandler()` function to read the form value inputs after a button click, then use them to create a task on the page.

However, click event listeners can lack usability features that may seem trivial but make forms feel more intuitive for some users, such as allowing both the submit `<button>` element and the Enter key to submit the form. Changing this type of behavior doesn't take too much work on our end, so let's get started!

To begin, we'll move the event listener from the `<button>` element we added in the last lesson and apply it to the `<form>` element itself. Now the browser will be able to listen to an event happening on the whole form rather than just the button.

We'll have to change the code in two places:

- At the top of `script.js`, delete the variable declaration for `buttonEl` and add this in its place:

```
var formEl = document.querySelector("#task-form");
```

- At the bottom of `script.js`, remove the code to add an event listener to `buttonEl` and add this in its place:

```
formEl.addEventListener("submit", createTaskHandler);
```

Now the `script.js` file finds the `<form>` element in the page and saves it to the variable `formEl`, so that we can not only interact with the form but also access some of its child HTML elements. The latter will come into play more later, so let's think about interacting with the form.

Because we're targeting the entire form instead of just the button, we can't use the click event listener anymore. If we kept a click listener, then every time we clicked on the form it would run the `createTaskHandler()` function, which wouldn't really help us.

Instead, we're using a form-specific event called **submit** (also called **onsubmit** in certain documentation). This particular listener actually listens for two events within the context of the form:

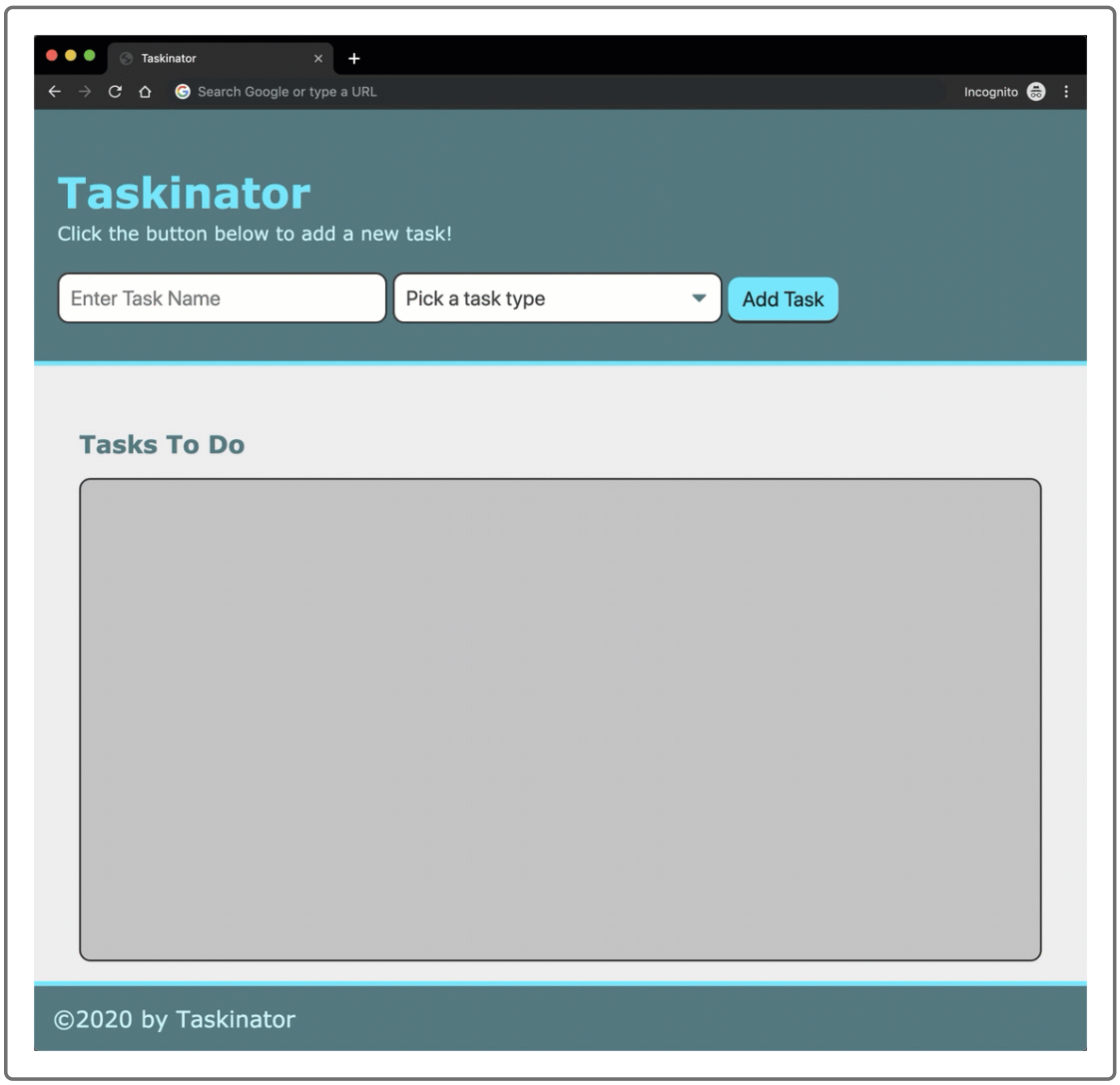
- When the user clicks on a `<button>` element with a `type` attribute that has a value of "submit", like the button we currently have in the form
- When the user presses Enter on their keyboard

DEEP DIVE

So instead of running the `createTaskHandler()` function with every button click, either of the two above form submission browser events can now run it. Let's test that out, as the `createTaskHandler()` function should still work.

Save `script.js` and refresh the browser, then try filling out the form and pressing `Enter` on the keyboard or clicking the Add Task button. As you can see, the code kind of works.

The code runs and creates a new task, but it's immediately deleted and the browser window seems to refresh itself. See this animation below for reference:



What is happening here? Why would the code run, put something on the page, and then do nothing? Even if we add a `console.log()` statement to `createTaskHandler()` and monitor the Chrome DevTools console here, we'll notice that the log shows up for a second and then disappears as well.

Depending on how fast your computer is, you may not even notice that the browser itself actually reloads the page every time you submit the form! So the code works just fine here, but the browser is causing problems. This legacy browser behavior used to help webpages communicate with servers, but now JavaScript does most of that work.

Now that we use JavaScript to handle these types of actions, we no longer need to rely on this default browser behavior to complete the task. But the

browser doesn't know that, and it still wants to do what it was designed to do. It's up to us to explicitly instruct the browser to not do that.

Let's fix the little issue right now by making the `createTaskHandler()` function look like this:

```
var createTaskHandler = function(event) {

  event.preventDefault();

  var listItemEl = document.createElement("li");
  listItemEl.className = "task-item";
  listItemEl.textContent = "This is a new task.";
  tasksToDoEl.appendChild(listItemEl);
};
```

Save `script.js` and refresh the page, then try filling out and submitting the form.

It works! The page doesn't refresh at all, and the newly created task remains in the list. So why did the addition of `event.preventDefault()` help us here? Let's find out.

The Event Object

First, a little backstory on the browser event and JavaScript relationship. The browser pays attention to everything that happens on the page. If someone scrolls down the page, the browser knows exactly how far they go. If someone clicks on the unused margins of a page, the browser knows. All of this happens whether we create a JavaScript event listener or not.

When we use JavaScript to listen for an event that occurs on an HTML element, however, the browser collects all of the information for that event and packages it into an object for us to use. This is known as the **event interface**, but that's just a fancy name for a nicely packaged JavaScript

object we get to use in the event handler function. We can use this **event object** by making the function executed by the event have an argument to represent the event object. Once we do that, the browser can fill in the data for that event and pass the argument into the function.

DEEP DIVE

By adding the `event` argument to the `createTaskHandler()` function, we can use the data and functionality that object holds. We did that when we added `event.preventDefault();` to the handler function's code. What might that method's name mean?

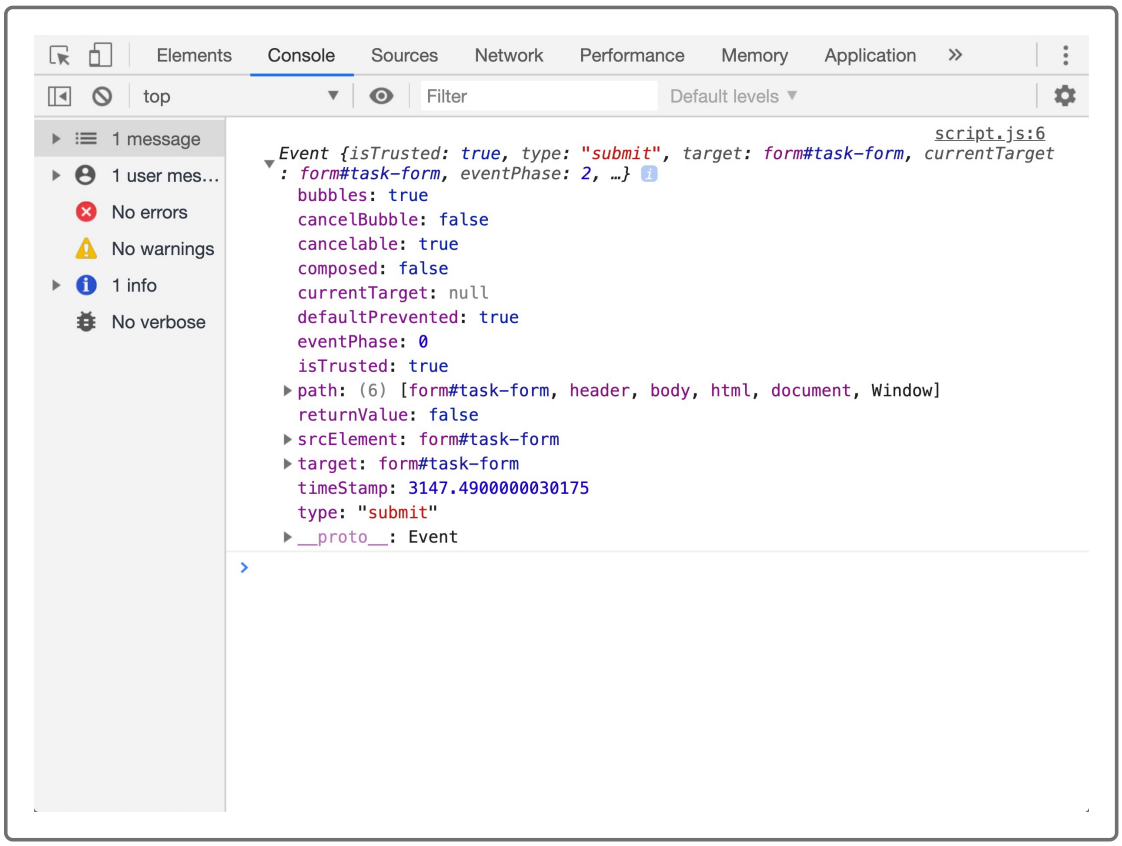
Think about what happened before we added it: the page refreshed every time we submitted the form. That was due to the browser's default behavior when handling a form submission. So if we execute a method named `event.preventDefault();` in the handler, we're instructing the browser to not carry out its default behavior.

We won't need `event.preventDefault();` for all event handlers, but we'll have to use it a few more times for Taskinator. After all, the click handler worked without it this whole time.

One more thing before we move on. Let's explore the event object a little bit by logging it to the console. Add the following to the `createTaskHandler()` function:

```
console.log(event);
```

Save `script.js` and try submitting the form again. If you turn your attention to the DevTools console tab, you'll see the event object. Open it, and you should see something like this image:



As we learned when implementing the click event, the event object holds a fair amount of data. A lot of it's unimportant at the moment, but over time we'll learn how to use some of the properties in our applications.

So the form submission works, and the event handler can work as intended, but we're still creating a task item with preset values. Let's focus on the form's input values and see how we can retrieve the content we enter.

Remove the `console.log(event)` call you added to `createTaskHandler()`, then add, commit, and push the code up to the GitHub feature branch!