4.2.6 Capture Form Field Values

At this point, the same canned tasks keep getting added to the page when we submit the form. What's the point of having form input elements that a user can interact with if we don't do anything with them?

Let's outline what we'll need to do for the event handler to retrieve the form's values upon submission:

1. Target the HTML elements with the pertinent data.

2. Read and store the content that those elements hold.

3. Use that content to create a new task.

We'll start off simple and just worry about reading the task's name first, then we'll retrieve the task's type when we know we're heading in the right direction.

In `createTaskHandler()`, add the following code below the `event.preventDefault();`:

```
var taskNameInput = document.querySelector("input[name='task-name']");
console.log(taskNameInput);
```

Before saving and testing the code, take a look at the selector used as an argument in the `querySelector()` method. What do you think this selector syntax means?

When we use square brackets `[ ]` in a selector, we're trying to select an HTML element by one of its attributes. In this case, we're selecting the `<input>` element on the page that has a `name` attribute set to a value of "task-name".

PAUSE

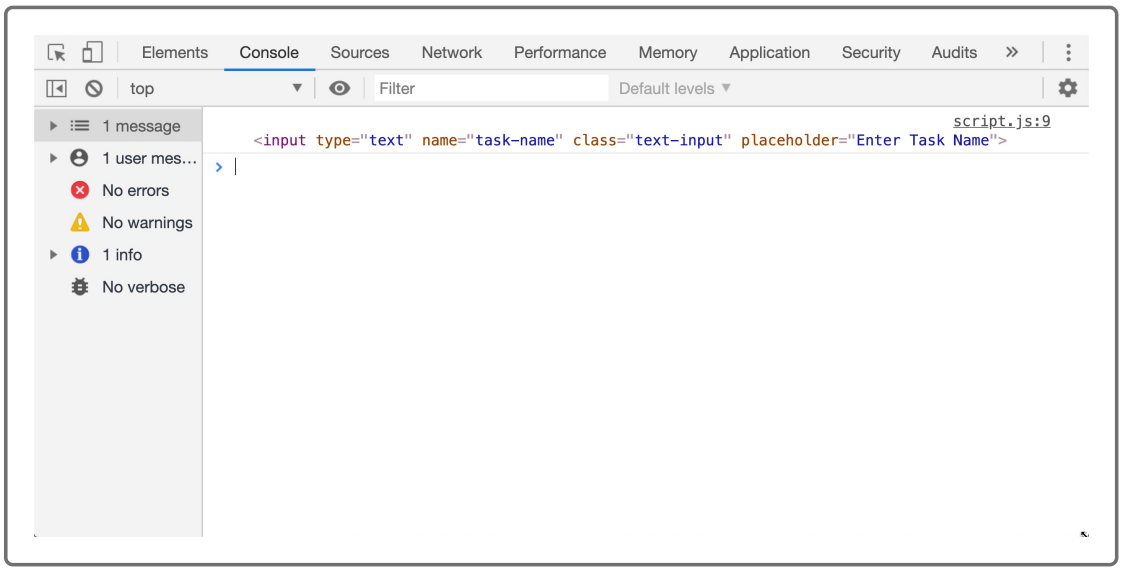
In the following selector, why are single quotes wrapping the attribute's value and double quotes wrapping the entire selector?

```
document.querySelector("input[name='task-name']");
```

If we used another set of double quotes to wrap the attribute's value, the entire string would break; it would assume we ended the string at `"[name="`, and anything after would break the query selector.

Hide Answer

Save `script.js`, refresh the page, and try submitting a task (don't forget to add text in the input field). After you submit a task, turn your attention to the console. The following image shows a glimpse of what you might see:

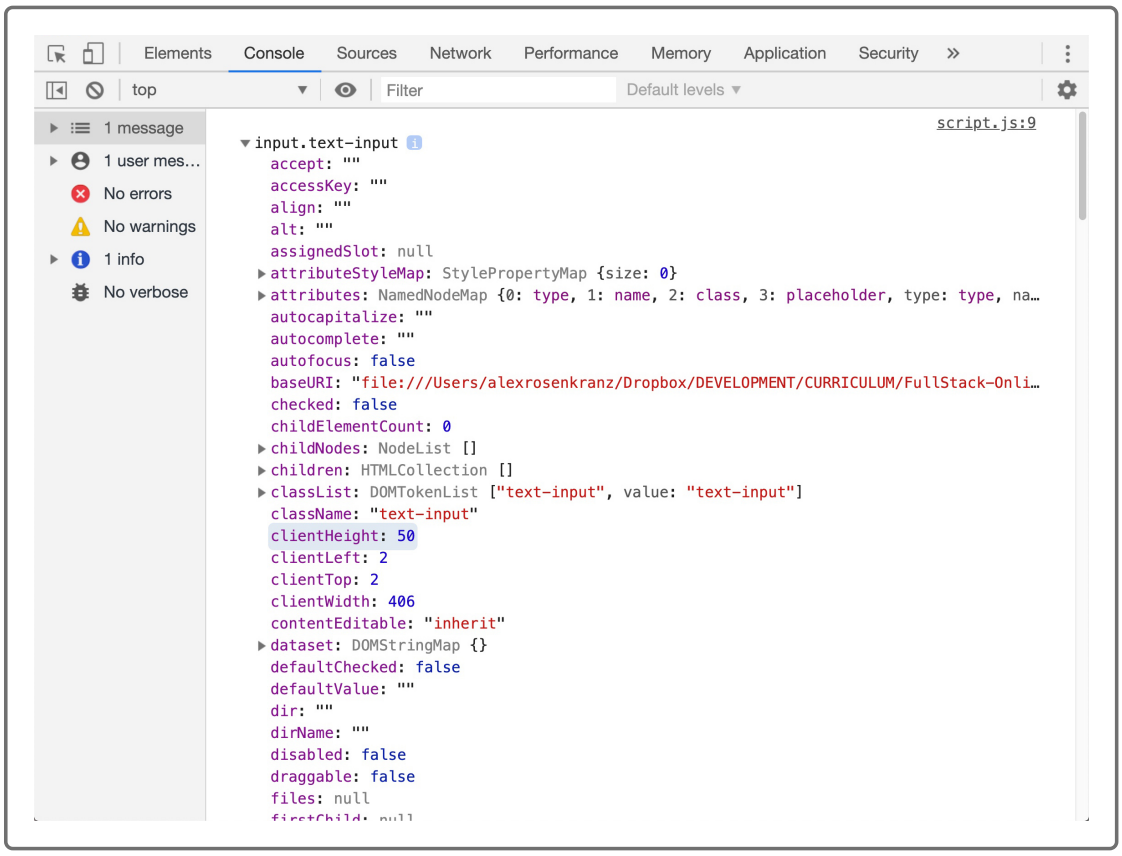


We should see the form element's HTML tag get logged, which isn't too helpful in this case. We need to get more information from the form element, but how? If `console.log()` can't help us see more of the information the HTML element holds, such as the content in the `<input>` field, what can?

Let's try this again, but instead of using `console.log()`, let's change it to look like this:

```
console.dir(taskNameInput);
```

Save `script.js`, refresh the browser, and submit a new task. The console should now display not the HTML tag for that element but rather an object that looks something like this image:



Take a minute to examine all of the different properties. The browser keeps tabs on all of this underlying data for a single HTML element. It knows virtually everything about this element: its height and width on the page, its parent elements, its child elements (if applicable), and even what the user has typed in the input box.

Navigate down to the `value` property for this element, and we'll see what we typed into the form before submitting it. We need to retrieve and create a new task item from this data!

Notice how we had to use `console.dir()` to get the result we needed? While `console.log()` can get us the information we need most of the time, we can also use `console.dir()` to make the console display data as a JavaScript object.

DEEP DIVE ▲

DEEP DIVE

To learn more, see the **MDN web docs on console.dir() (https://developer.mozilla.org/en-US/docs/Web/API/Console/dir)** and the **MDN web docs on the HTMLElement (https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement)** .

Get the Task to Display

Now that we've pinpointed the data we care about, we can grab it and put it on the page. Let's delete the `console.dir(taskNameInput);` from `createTaskHandler()` and update the query selector to look like this:

```
var taskNameInput = document.querySelector("input[name='task-name']").
```

Previously we selected and stored the entire HTML element for the task name form input. We don't need to worry about any of that element's other properties—just the `value` property. We can get to that property by adding a `.value` to the end. Now the value of the `taskNameInput` variable will be the text we entered in the `<input>` element.

NERD NOTE

The common verb used for retrieving or reading data from an object's property is **getting.** When we provide and store data in an object's property, it's called **setting**. These two terms are used often in web development.

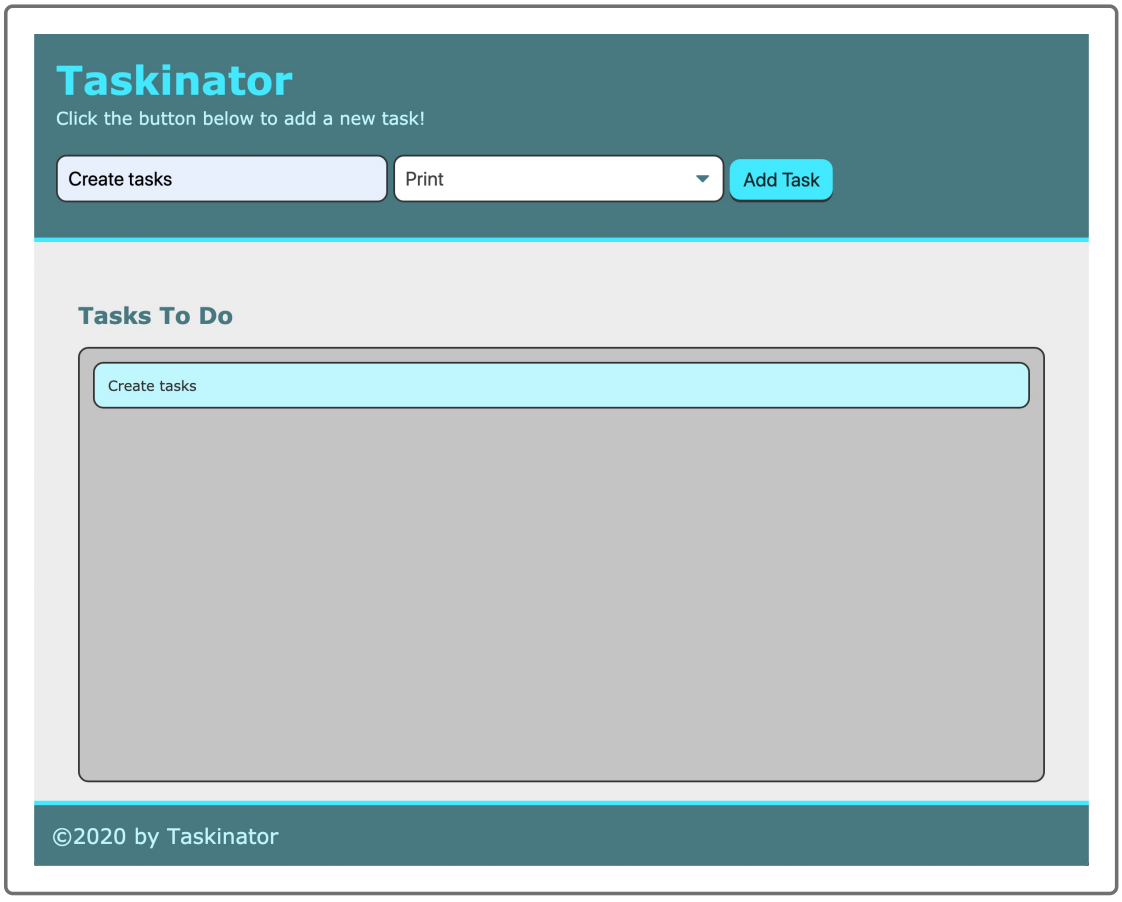
Next we want to get the task name we just stored in `taskNameInput` and add it to the `listItemEl` variable. Let's update the `listItemEl.textContent` property to look like this instead:

```
listItemEl.textContent = taskNameInput;
```

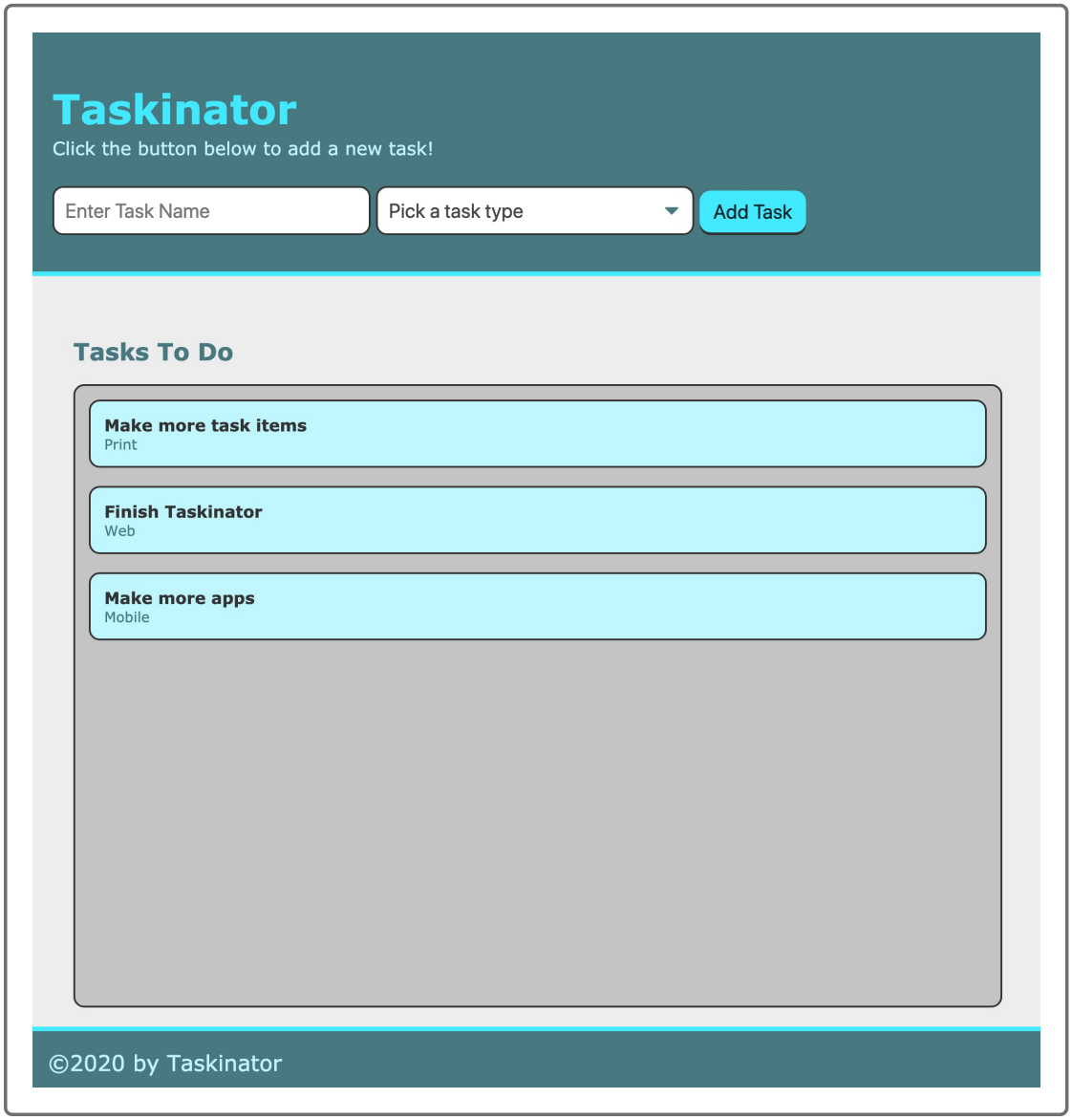
If we save, refresh the page, and submit a new task, we should see whatever we entered into the form appear as a task in the list! We can make any custom task that we need. Now we just need to add the task's type as well.

Add More Content to a Task

At this point, the task list looks like this image:



The mock-up, however, shows that the task type should appear below each task name, as shown here in this image:



To do this, first we'll have to get the value of the `<select>` dropdown's picked `<option>` element, then we'll have to create more HTML to go inside the `<li>` element we created for a task item.

Let's start by getting the value of the `<select>` dropdown. Luckily we can use the same code we used to get the task name's value and update the selector to find the `<select>` element instead.

Add the following code below the variable declaration for `taskNameInput`:

```
var taskTypeInput = document.querySelector("select[name='task-type']")
```

Test it by using `console.log(taskTypeInput);` below the `taskTypeInput` variable declaration to see what the value is. It will display whatever `<option>` element was picked in the `<select>` dropdown.

Great! We can use that value and add it to the task item, but first we'll need to refactor the code a little bit to make the HTML easier to style and maintain.

Let's update the code in `createTaskHandler()` to look like this below the `taskNameInput` and `taskTypeInput` declarations:

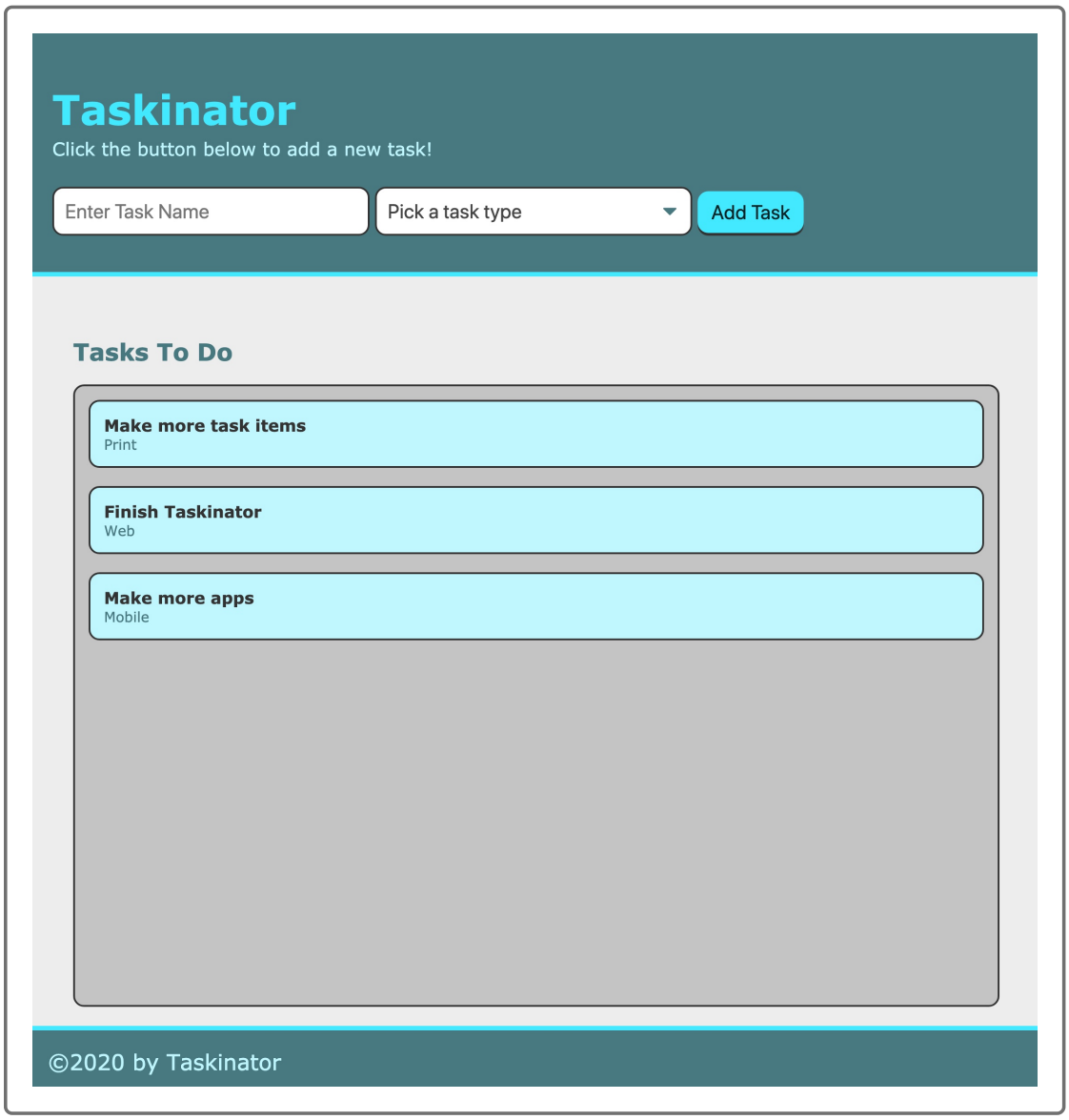
```
// create list item
var listItemEl = document.createElement("li");
listItemEl.className = "task-item";

// create div to hold task info and add to list item
var taskInfoEl = document.createElement("div");
// give it a class name
taskInfoEl.className = "task-info";
// add HTML content to div
taskInfoEl.innerHTML = "<h3 class='task-name'>" + taskNameInput + "</h

listItemEl.appendChild(taskInfoEl);

// add entire list item to list
tasksToDoEl.appendChild(listItemEl);
```

Save `script.js` and try submitting a new task after refreshing the page. The result should look something like this image:



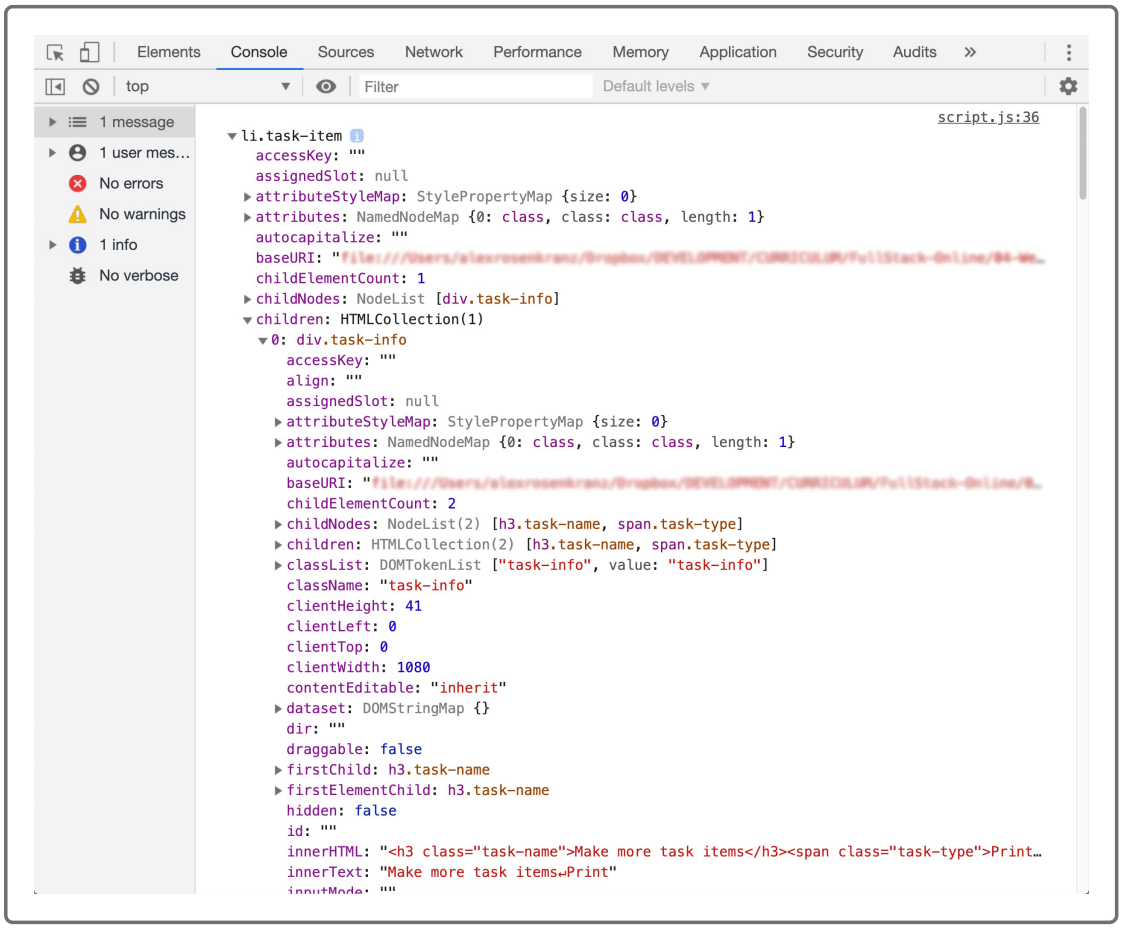
We can now create a new task item with the content submitted through the form. We could've organized this content in a number of ways, but as we've discussed, wrapping content in a container `<div>` element will sync and bundle all that content together.

Select the Elements tab in the Dev Tools and expand the `<li>` elements in the `<ul>`. In each `<li>`, see each nested `<div>` with `<h3>` and `<span>` elements, and then look back at our JavaScript to see how our code created them.

We still created the `<li>` element to hold the whole task (listItemEl), but instead of writing the task's content right to it, we created a `<div>` to hold the content (`taskInfoEl`). Once we'd set the data into the `<div>`, we

appended it to the `<li>`. Lastly, we appended the entire `<li>` to the parent `<ul>` (`tasksToDoEl`).

Notice how we can add a child HTML element to another HTML element in JavaScript before it even gets to the page? We can append the `taskInfoEl` to the `listItemEl`, meaning that all of the `taskInfoEl` content is set inside `listItemEl` as a child HTML element before we add `listItemEl` to the page. We can visualize this in the console by using `console.dir()` to print out the data in `listItemEl`. For example, this image shows the nested element in the `children` property:

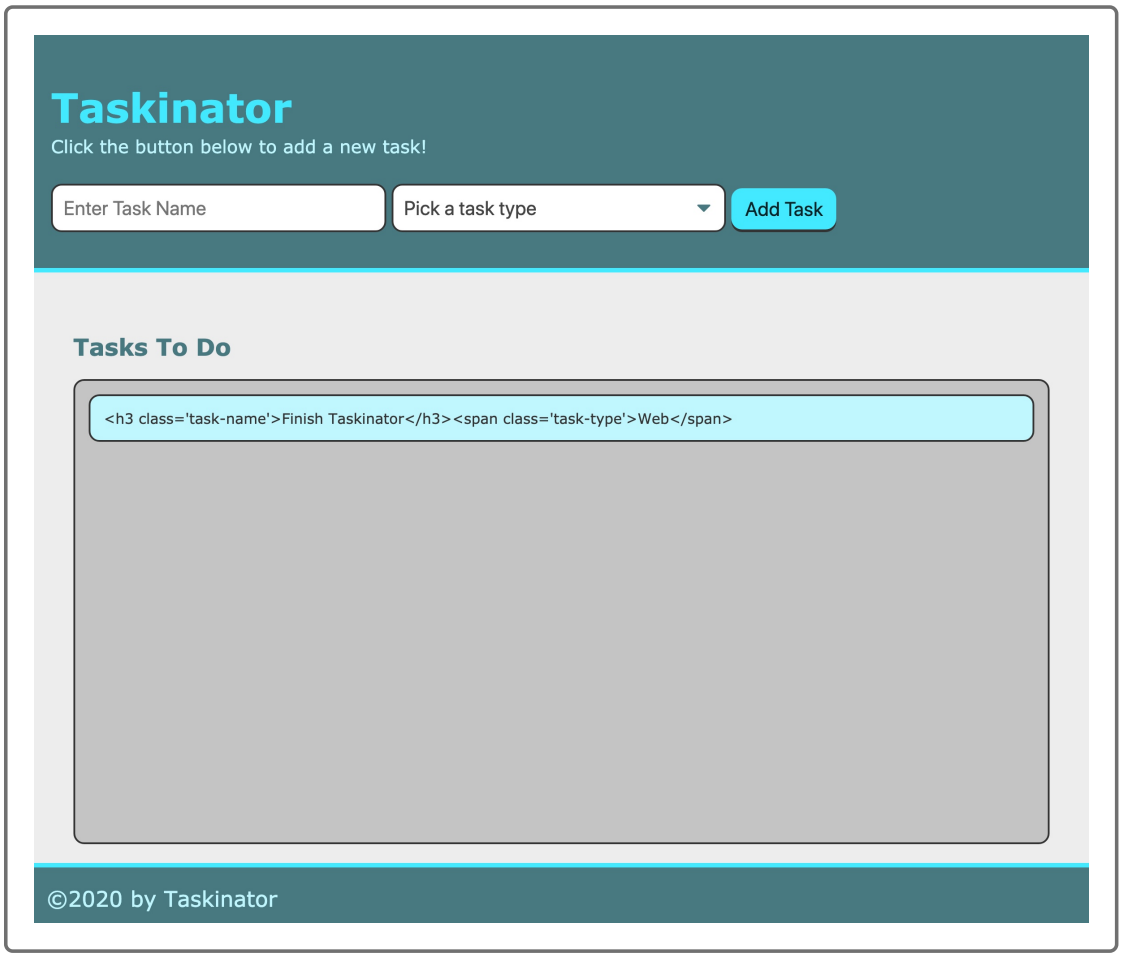


That's not all, though. We also used a new DOM element property called `innerHTML` that works a lot like the `textContent` property, but with one big difference. The `textContent` property only accepts text content values. If it saw an HTML tag written in as a value, it would literally display the markup characters for that HTML tag without interpreting it as the HTML tag.

The `innerHTML` property allows us to write HTML tags inside of the string value we're giving it. When it loads, it reads the content as HTML and renders it to the DOM. So, an `<h3>` tag here will be rendered as an `<h3>` element in the DOM. (Notice here our difference in language: we write *tags* in our HTML in order to create corresponding *elements* in the DOM.)

For `textContent`, it would display something like `"<h3 class='task-name'>"` and not infer that we wanted to use that actual HTML element.

Let's change `innerHTML` to `textContent` for a second and see what displays. It should look something like this image:



Obviously that's not what we want, so `innerHTML` is the better fit here. Be sure to change it back to `innerHTML`, and we can move on!

IMPORTANT

> We didn't have to use the `innerHTML` property here but rather we could have created HTML elements for the title and type separately and then appended them to the container element. While both approaches work, `innerHTML` lets us create fewer variables, but with less readable code.
>
> Remember, there's usually more than one way to complete a task. It's up to you to decide which way to go.

When to Refactor

Right now, the `createTaskHandler()` function does a lot. It reads the form elements on submission, then creates quite a bit of HTML content and adds it to the page.

This works for our needs at the moment, but leaving it like this may lead to a headache when we add more features to the application. This may be a good time to separate `createTaskHandler()` into two different functions:

- One to handle the form submission, get the form values, and pass those values to another function as arguments

- One to accept the form values as arguments and use them to create the new task item's HTML

We'll tackle this type of refactor next. But first, since we know the code is working, let's add, commit, and push to GitHub.