4.1.8 Add Items with the Click of a Button

Currently the task list has a single task item. We need to figure out how to create the task item and add it to the list. Any idea how we can proceed? Try to imagine what steps will be needed. What file needs to be changed? Where in this file do we add the task item? Solving a problem starts with asking the right questions, then finding the answers.

Let's take another look at the `index.html` file to see where the task list is located:

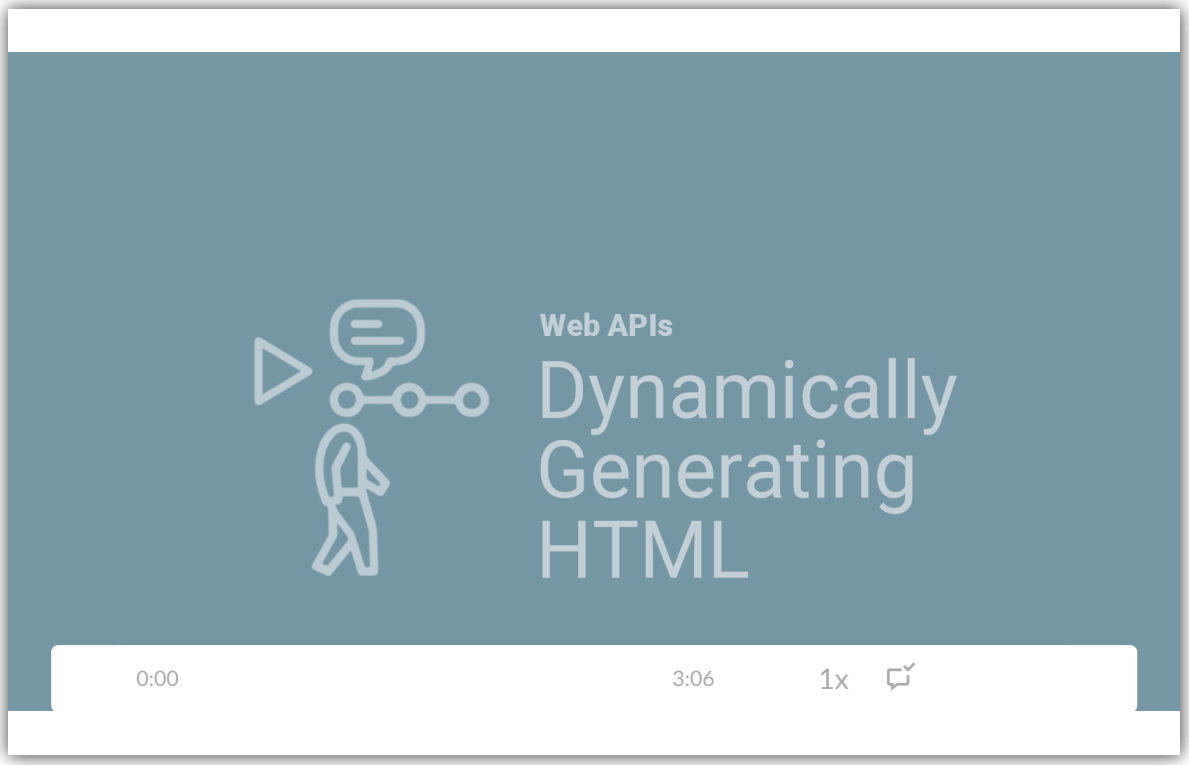
```
<main class="page-content">
  <section class="task-list-wrapper">
    <h2 class="list-title">Tasks To Do</h2>
    <ul class="task-list">
      <li class="task-item">A task is going to be added soon!</li>
    </ul>
  </section>
</main>
```

HIDE HINT

Read the text content of the element for the hint.

We'll need to add an `<li>` element to the existing `<ul>` element, but we can't do this manually every time a user clicks the button. We need to figure out how to do this programmatically in JavaScript.

To prepare for this section, watch the following video on dynamically generating HTML elements with JavaScript:

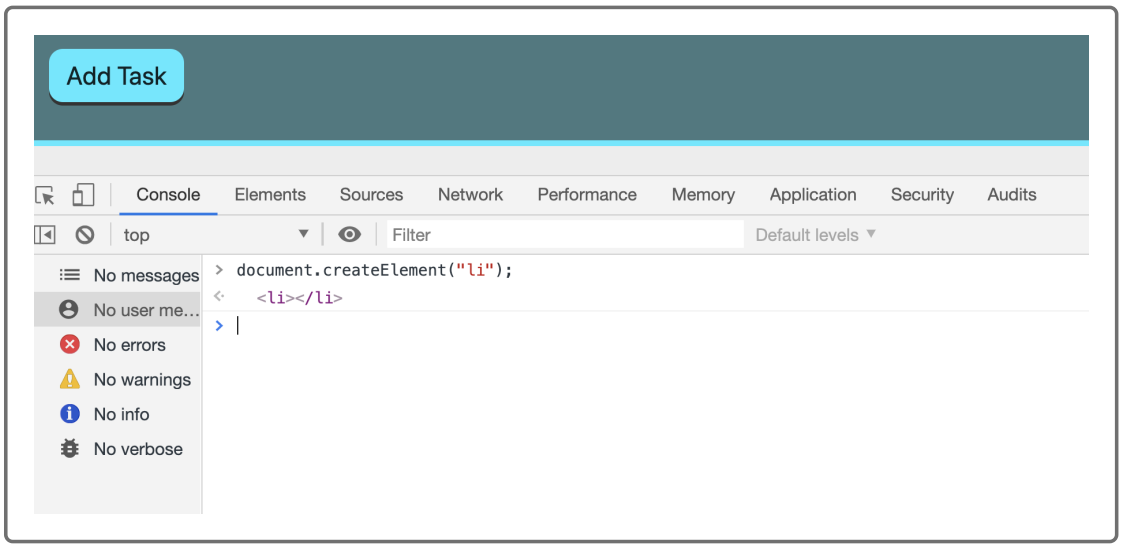


As you can see, one method we can take advantage of is `createElement()`. What does this method do again?

Yes, it creates an element—specifically a DOM element object. We'd like to create a specific element, however, such as an `<li>`. Let's create an element with this method by typing the following into the browser's console:

```
document.createElement("li");
```

The response in the console should look like the following image:



Congrats! You just dynamically created your first element with JavaScript. It doesn't do much now, but we'll soon change that. First let's create a variable to store the reference to this new object.

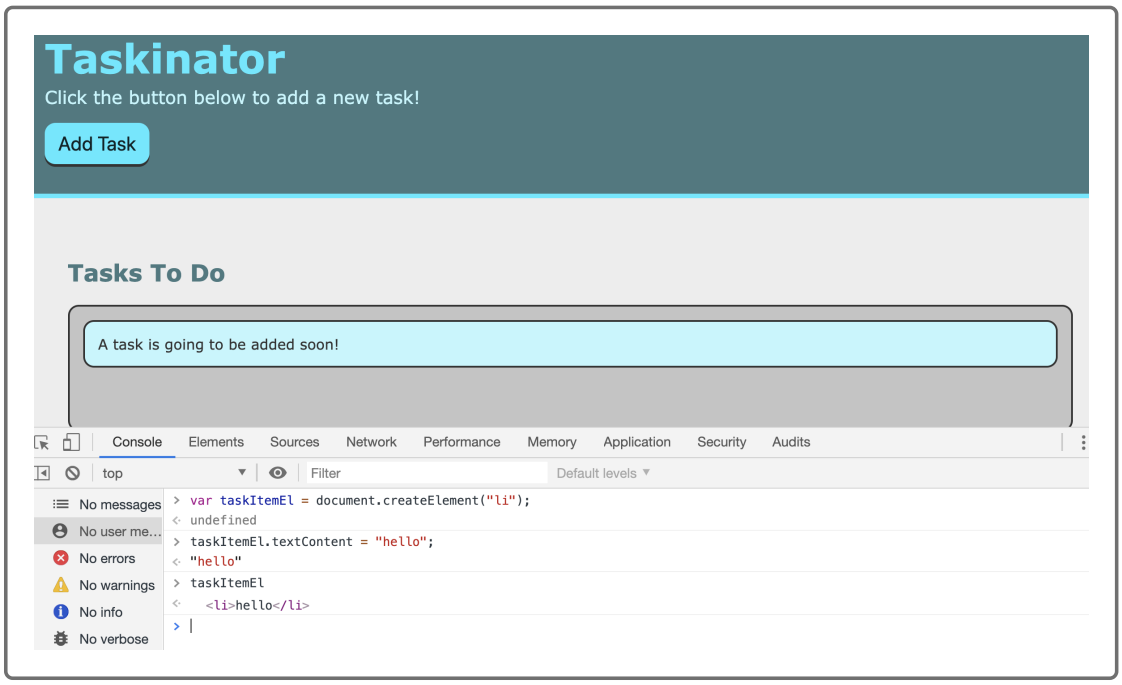
Type the following expression in the console:

```
var taskItemEl = document.createElement("li");
```

Now, to assign some text content to this `<li>`, let's use the `textContent` property we used previously and then view the element object:

```
taskItemEl.textContent = "hello";
taskItemEl;
```

The browser looks the same, but our DevTools console looks something like this image:



As the image above shows, even though we've created the task item element and added the text content, the task item still doesn't appear on the task list. Although we created a DOM element, we still haven't attached it to the existing page. Next we'll add this element to the `<ul>` or the task list.

Before we can attach the task item, we'll need an object representation of the task list or `<ul>` element. We could use the element type as the selector, but let's be more exact and sidestep a potential gotcha: add an `id` attribute to the `<ul>` element in case another `<ul>` element is created in this `document`.

Return to the `index.html` file and add the following `id` attribute "tasks-to-do" to the task list or `<ul>`:

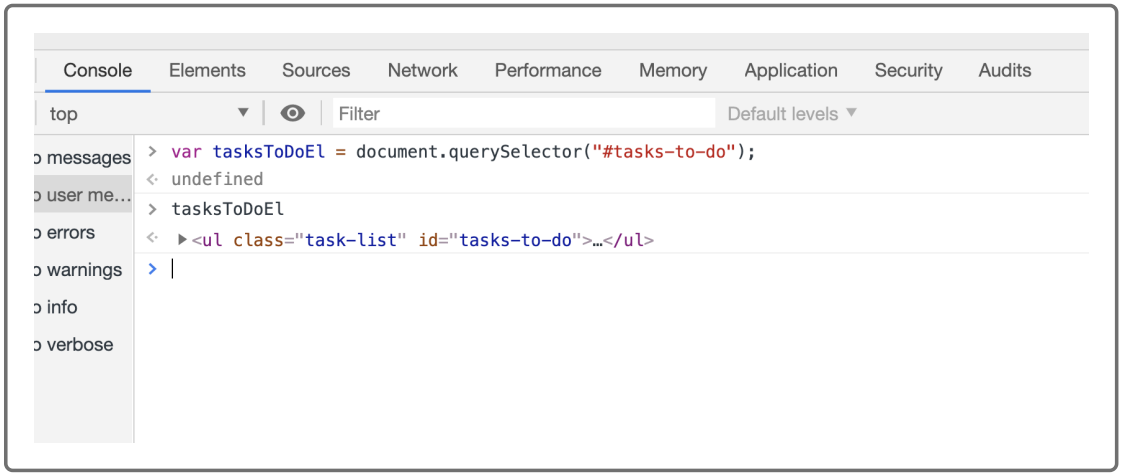
```
<h2 class="list-title">Tasks To Do</h2>
<ul class="task-list" id="tasks-to-do">
  <li class="task-item">A task is going to be added soon!</li>
```

Now we're ready to target the `<ul>` element in the DOM so that we can programmatically manipulate an element object in JavaScript. Let's save

this file and refresh the browser with the `index.html` file. Now create a variable reference to the task list by typing the following expression into the browser's console:

```
var tasksToDoEl = document.querySelector("#tasks-to-do");
tasksToDoEl;
```

To verify that we've targeted the correct element in the DOM, type `tasksToDoEl`. Your console should look something like this image:



As you can see, you have the task list object, the `<ul>`.

Very good! Now we'll want to append the `taskItemEl` list item to the `<ul>` list. But there's a problem: we refreshed the browser after we added the `id` attribute to the `<ul>`, so we lost all of our work in the console before the refresh. So, recreate the `taskItemEl` with these steps, as before:

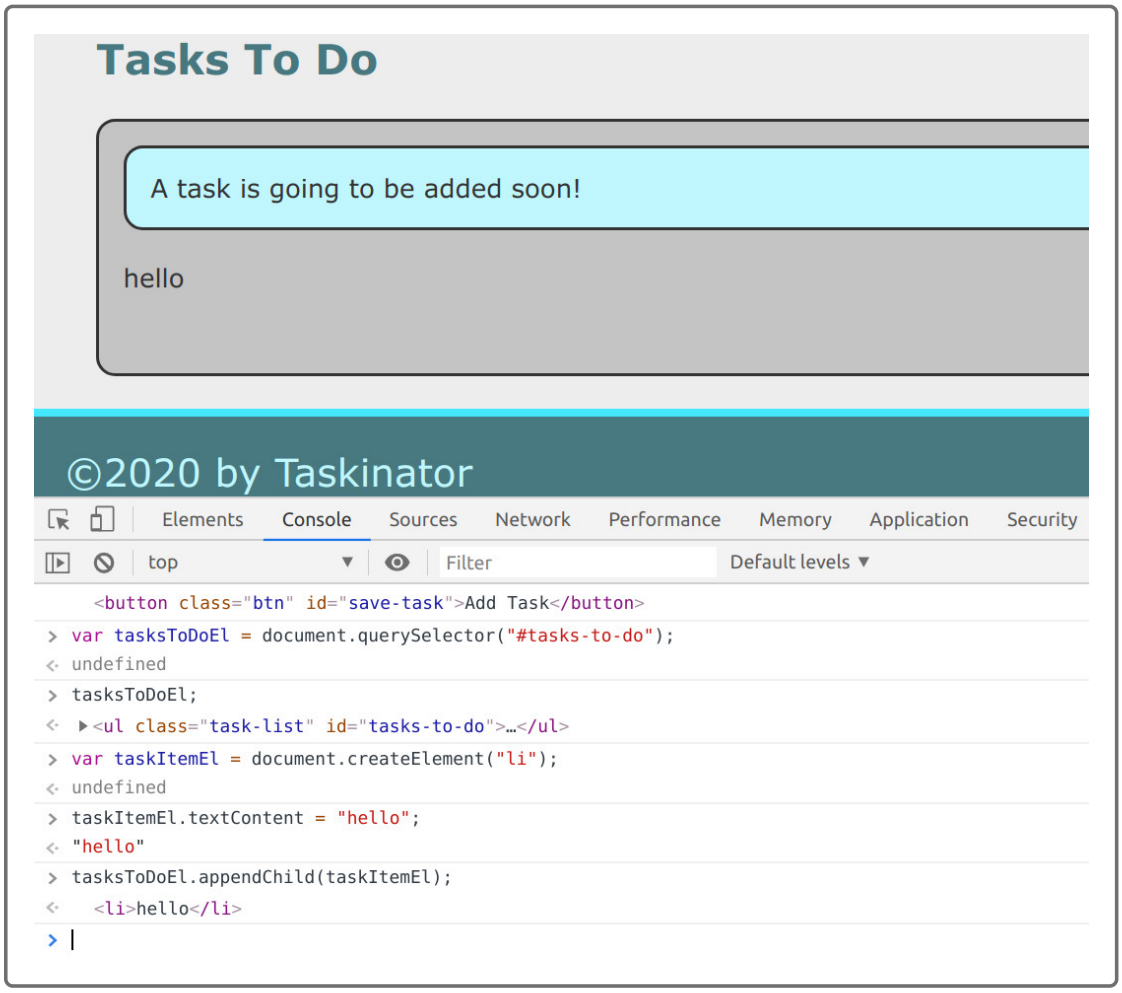
```
var taskItemEl = document.createElement("li");
taskItemEl.textContent = "hello";
```

Now it's recreated in the DOM, and we can proceed. We'll next use the method called `appendChild()`. To append the task item as a child to the

task list, use the following expression pattern:

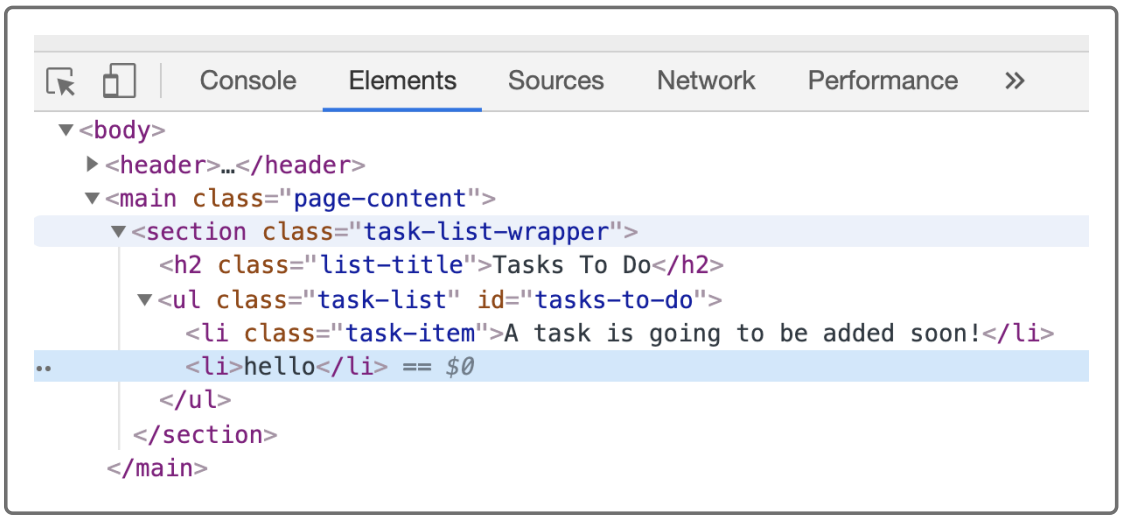
```
tasksToDoEl.appendChild(taskItemEl);
```

Type that expression into the console to render something like the following image in the browser:



Nice, we attached the task item to the task list, as shown in the browser.

The Elements tab in Chrome DevTools shows that the element we created —the task item, or `<li>`—has now attached as a child to the task list, or `<ul>`. Our HTML structure looks something like this image:

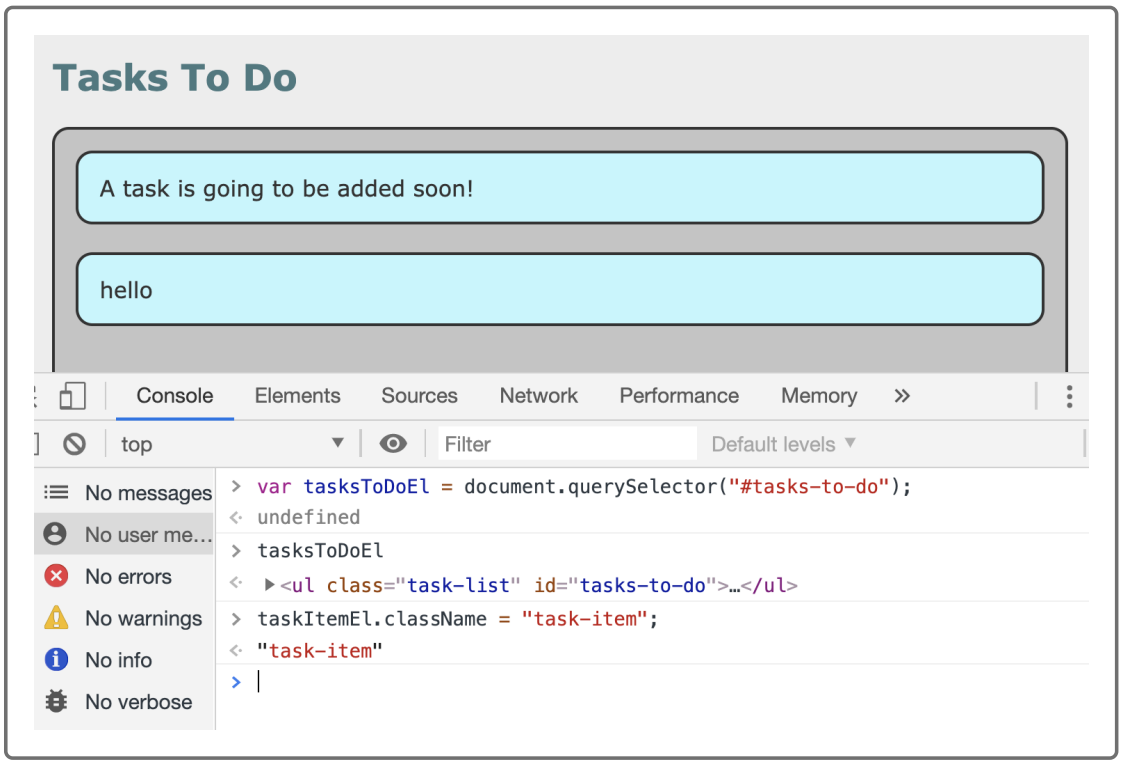


Excellent work; however, this task item needs a bit of style. Looking at the HTML, you can see that the first `<li>` has the class attribute `task-item`. You can programmatically assign this class to the task item with the property `className`. Called **dynamic styling**, this approach is similar to how you previously dynamically created and manipulated the DOM with the task item. It's also similar to how you previously assigned the `textContent` with the `<button>`.

Let's add the following into the console:

```
taskItemEl.className = "task-item";
```

This should render a styled task item, as shown in the following image:



By adding the class attribute, we incorporated the CSS styles. Because each expression has demonstrated the desired results, let's move the code into the `script.js` file.

First we'll add the DOM object reference to the task list at the top of the page, underneath the task item DOM reference. Then we'll add the following event listener and also change the text of the new task item dynamically, using the `textContent` property we used previously to note, "This is a new task."

```
var buttonEl = document.querySelector("#save-task");
var tasksToDoEl = document.querySelector("#tasks-to-do");

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

We exchanged `console.log` for expressions that will do the following:

- Create a new task item
- Style the new task item
- Add the text
- Append this element to the task list

Going forward, these four steps will be key in dynamically creating elements with the DOM.

Save `script.js` and refresh `index.html` in the browser. Click the Add Task button a few times to see this new JavaScript at work!

For a recap of all things DOM, take a moment to watch the following video:

Let's wrap up this lesson with a nice-to-have optimization that will add clarity and organization.

Because event listeners can get very long due to the many steps needed to create more than one element, developers often place this code into

separate functions to delineate the single responsibility principle.

Let's make a `createTaskHandler()` function to dynamically create the task item.

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

Notice that the code in the `createTaskHandler()` function replicates the code block in the anonymous function. We'll place this function expression above the event listener; otherwise we'd receive an error that `createTaskHandler()` isn't defined, since we'd be calling the function before we defined it.

Lastly, let's replace the existing event listener with this one, which uses `createTaskHandler` as the callback function:

```
buttonEl.addEventListener("click", createTaskHandler);
```

The name `createTaskHandler` is easy to understand at a glance, so seeing it in the event listener is preferable to a long anonymous function, which is what we had before.

Now this statement can be read as "on a button click, create a task."

Let's test the code again. Save and refresh the browser, then click the button. You should see something like this image:

Congrats on achieving a milestone!

Now we can add a new task to the task list with a button click. In the next lesson, we'll develop the app further to obtain data from the user for more personalized task items and we'll add states to the task progress. Let's save the progress in Git and push changes to GitHub. We'll merge the `feature` branch into the `develop` branch.

```
git checkout develop
git merge feature/add-task
```

Don't forget to close the GitHub issue.

Before we move on and learn more DOM-based functionality, let's make sure we understand everything we've done so far with a quick assessment:

Final score: 100%
Retake