4.4.7 Attach the Task Item in the Drop Event

Now we are ready for the `drop` event!

But first, does the drag-and-drop operation seem more elaborate than originally imagined? This is a good lesson to remember: estimating how long it takes to complete a feature with new technology should always take into account some time to learn and troubleshoot—you'll find this to be the case even when you're an experienced developer!

Let's review what we need to do in the `drop` event handler:

1. Retrieve the original dragged task item from the `dataTransfer` property.

2. Reference the destination of the drop as a DOM element.

3. Update the task status of the task item to match the task list.

4. Append the dragged task item to the destination drop zone.

As we foretold in the previous steps, we'll use the `drop` event handler to retrieve the `data-task-id`, which is our unique task item identifier, to select the dragged element from the DOM and append it to the drop zone.

How do we determine which task list the user wants to move the task item to? Just like in the `dragover` event, the `drop` event's `target` property contains the receiving element, or the element that is being dropped upon.

Let's demonstrate this with the following event listener delegated to the `pageContentEl` element. Place it at the bottom of the `script.js` file:

```
pageContentEl.addEventListener("drop", dropTaskHandler);
```

It is customary to delegate the event listener to the parent element, and narrow the drop zone in the `dragover` event handler. It is also possible to define droppable areas in the `dropTaskHandler()`, but in order to separate our concerns, it is a better practice to leave this responsibility to the `dropZoneDragHandler()`.

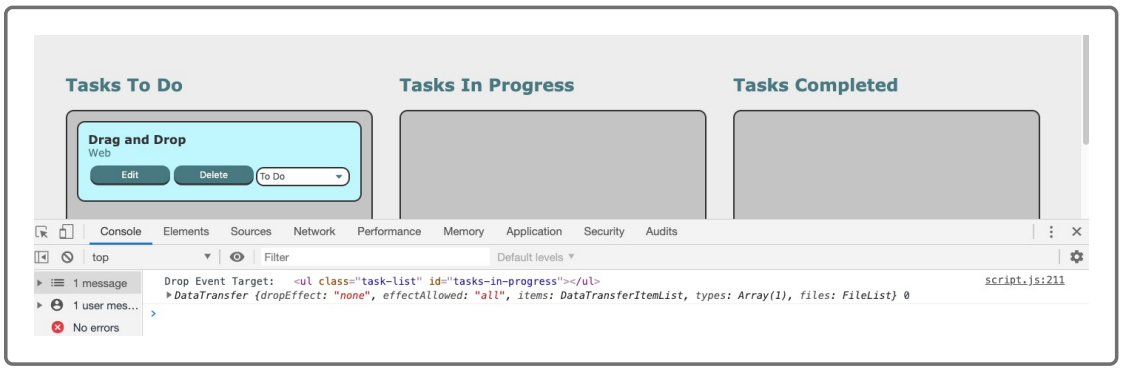
Now let's define our `dropTaskHandler` function by adding this to the `script.js` file in the section for event handlers, above the event listener expressions:

```
var dropTaskHandler = function(event) {
  var id = event.dataTransfer.getData("text/plain");
  console.log("Drop Event Target:", event.target, event.dataTransfer,
};
```

Here we are creating a reference to the `data-task-id` value. We stored it previously in the `dataTransfer` property of the event object by using the `setData()` method. Here we're retrieving it using the corresponding `getData()` method.

Let's save our script and refresh in the browser. Then add a task and drag it to the Tasks in Progress task list. It won't drop yet, but we can verify several of the key values by seeing what's in the console after we try to drop it. Note that we passed multiple arguments into the `console.log()` function using commas, two of them from the event object.

Here's how this looks:



Notice that in the console image, the event.target is `<ul id="tasks-in-progress" ... >`, which is the element the task was dropped on. This confirms that we dropped it on Tasks In Progress in the GUI.

Next in the console is displayed the DataTransfer object. Last, just after the DataTransfer object, we see "0", which is the task id that was returned from the getData() method.

Now try to drag this task item to a different list to ensure that the `event.target` element is changing as it should. Although the drop feature isn't working quite yet, we are making progress by creating the references to the values and objects we'll need to accomplish this step.

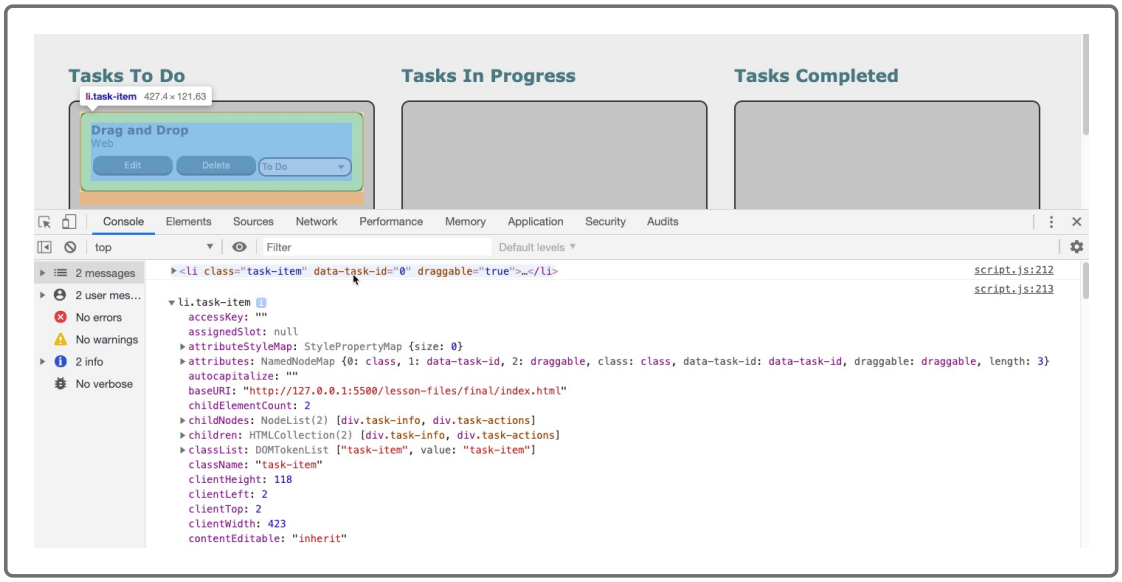
Excellent work! We now have the information necessary to make a successful drop: the unique task id and the destination drop zone element. In the next step, we'll use the `id` to find the element that was initially dragged and store the reference to this DOM element in a variable.

Type the following expression into the `dropTaskHandler()` function beneath the `id` declaration, and delete the previous `console.log()` that was there:

```
var draggableElement = document.querySelector("[data-task-id='" + id +
console.log(draggableElement);
console.dir(draggableElement);
```

If the first expression looks a bit complex, it is because we're using string concatenation to account for the variable task id. We're using the `querySelector()` method on the `data-task-id` attribute to locate the dragged task item with our unique task id. In our case, we're searching for the task item with the `data-task-id` value of "0".

Save the file and refresh the browser. Now add a task, then drag the task to get the following in the console:



If you see what appears in the preceding image in your browser, congrats! We've successfully stored and retrieved our task id in the `dataTransfer` object and used the `data-task-id` attribute to find the correct task item that was initially dragged and then dropped. We also verified that this task item is a DOM element so now we can append it to the drop zone element, which is our task list.

In the next step, we'll identify which task list the `draggableElement` was dropped onto.

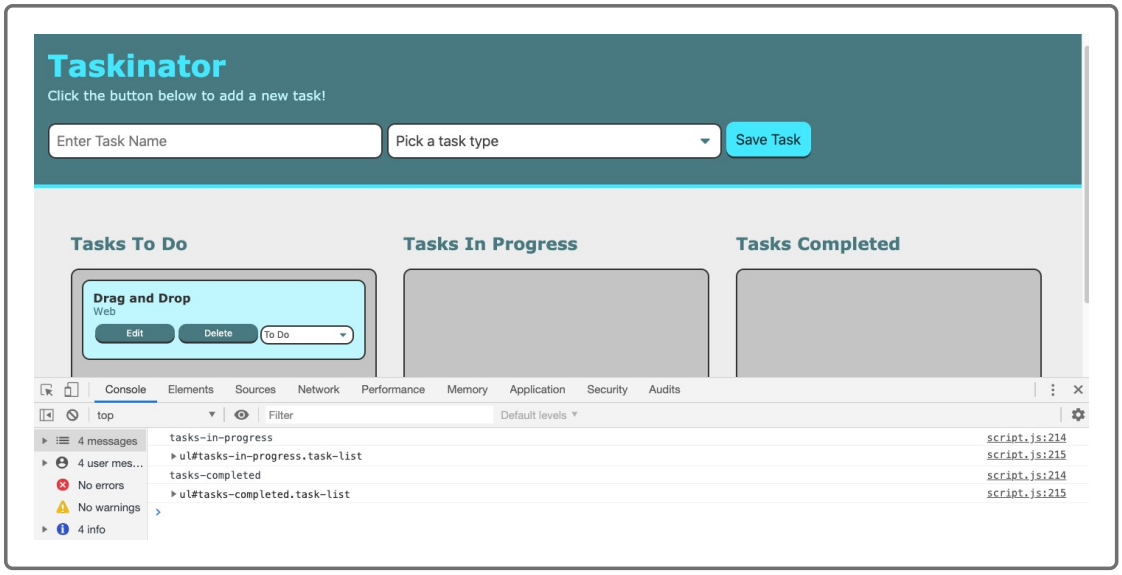
Let's add the following statements to the `dropTaskHandler()` function beneath the `draggableElement` declaration, and delete the previous `console.log()` statements.

```
var dropZoneEl = event.target.closest(".task-list");
var statusType = dropZoneEl.id;
console.log(statusType);
console.dir(dropZoneEl);
```

We've previously verified that the `event.target` property of the `drop` event identified the drop zone. In the above expression, we're using the `closest()` method again to return the corresponding task list element of the drop zone. The `closest()` method is especially well suited for this job because even if we drop an element on a deeply nested child element of the task list, the `closest()` method will then traverse up to each successive parent element until the selector is found.

Once we have the task list element, we'll use the `id` property of the task list element to retrieve the `id` attribute. This will identify which task list was dropped on, which designates the task status. We'll display our results in the console to verify our data is as expected.

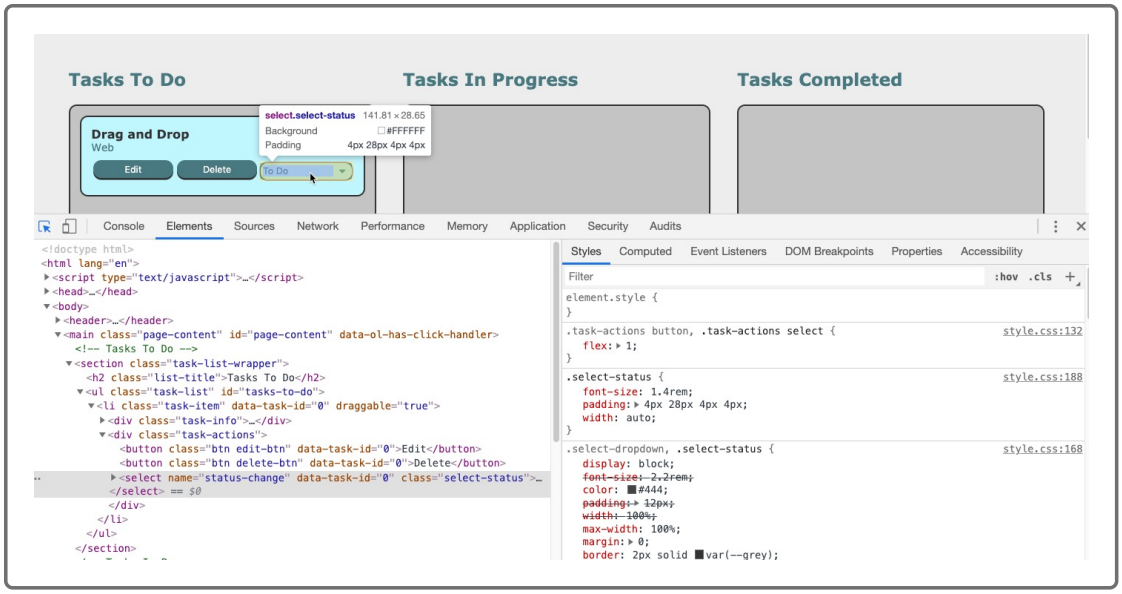
Save, then refresh the browser. Create a task item, then drag and drop it to the Tasks in Progress and the Tasks Completed lists. You should see the following in the browser:



As we can see in the console, the drop zone is different in relation to which list is dropped upon. First in the console, we see the id of the drop zone element. Below it we see an element with a drop-down list carat. If expand the drop-down list, we can verify that the `closest()` method, like the `querySelector()` method, is returning a document object, here created with the `console.dir()` function.

Excellent work! We have our drop destination element and can now append our `draggableElement` to it. Nice! But wait, the drag-and-drop feature's main purpose was to change the status of our task item. Let's proceed with this important step.

To change the status of the task item, let's first take a closer look at the `select` element in the browser, which designates the task status:



As we can see, the `select` element has a `name` attribute that is a unique identifier which we can use to find this specific element in the DOM.

Delete the previous `console.log()` and `console.dir()` statements. Then type the following in the `dropTaskHandler()` function to create a variable to reference the `<select>` element as a document object:

```
// set status of task based on dropZone id
var statusSelectEl = draggableElement.querySelector("select[name='stat
console.dir(statusSelectEl);
console.log(statusSelectEl);
```

Notice in this statement that we used the `draggableElement` and not `document` as the reference point of the `querySelector()` method. Why did we replace this reference point?

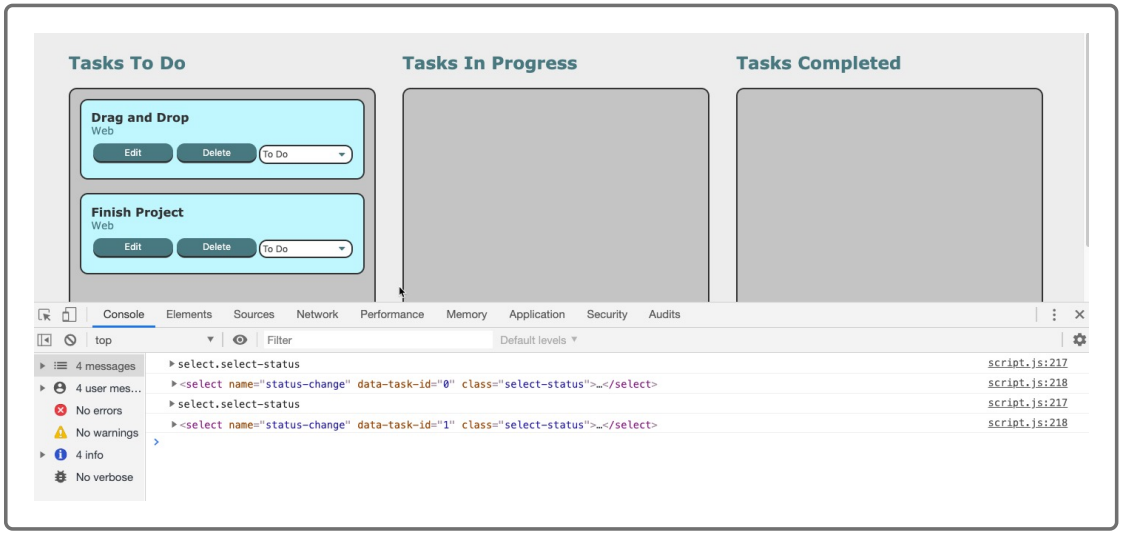
Let's say we used the `document` to query our `<select>` element. If we had multiple task items, this would also mean that we'd have multiple `<select>` elements with the same `name` attribute. Therefore, the `querySelector()` method would simply choose the first one in the DOM tree, not the element we need to change, which would be the `<select>` element of the task item that was dragged.

Luckily, we have a reference to the dragged document object. We'll use the `querySelector()` method on it to traverse down its descendent elements to find the corresponding `<select>` element of the dragged task item.

REWIND

The `querySelector()` method traverses down from the reference point to its children while `closest()` traverses up to ancestor elements to the root document from the reference element.

Let's save and refresh the browser, then add two tasks. Drag both tasks to see the following results in the console:



As we can see in the console, traversing from the dragged task item element, or the `draggableElement`, allowed the correct `<select>` element to be captured. The `console.log()` in this case helped identify the `data-task-id` attribute more clearly.

Now we have the `<select>` DOM element and the destination task list. Next, we'll identify how to update the status of the task. Using the `id` property to identify the new task status, let's use an `if-else` conditional to reassign the task.

Delete the previous `console.log()` and `console.dir()` statements in the function, and then type the following in the `dropTaskHandler()` function beneath the `statusSelectEl` variable initialization.

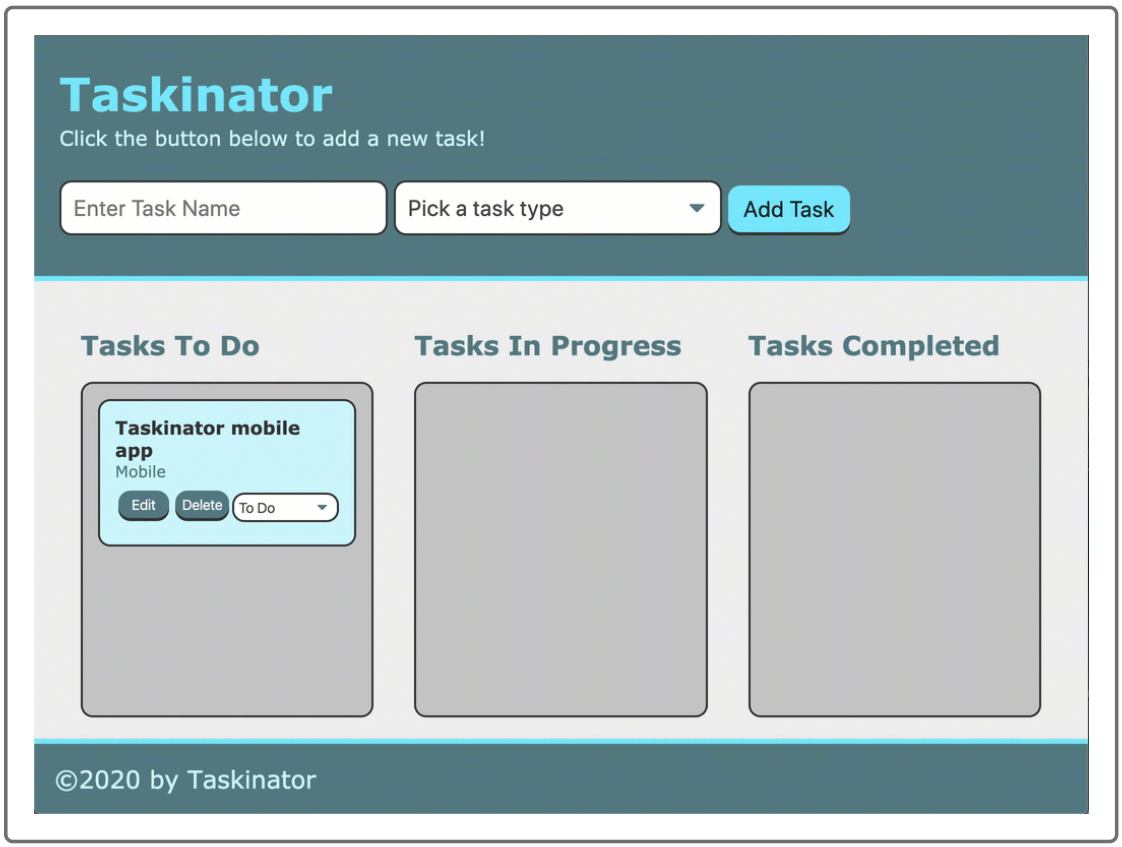
```
if (statusType === "tasks-to-do") {
  statusSelectEl.selectedIndex = 0;
}
else if (statusType === "tasks-in-progress") {
  statusSelectEl.selectedIndex = 1;
}
else if (statusType === "tasks-completed") {
  statusSelectEl.selectedIndex = 2;
}
```

Here we're using a new property, `selectedIndex`. This allows us to set the displayed option in a list by specifying the option's 0-based position in the list (where 0 is the first list option, etc.) By assigning a number to `selectedIndex`, we're selecting the option that we want to display in the `<select>` element. Our code maps the `statusType`'s id string values to option numbers, and sets the `selectedIndex` value appropriatelty.

Now all that's left is to append the `draggableElement` to its new parent element, `dropZoneEl`. Put this at the bottom of the function:

```
dropZoneEl.appendChild(draggableElement);
```

Let's save and refresh our browser and see how our feature looks. Add a few tasks and drag them around. You should see something like this:



;

Great job! The application is functioning very nicely with our new drag-and-drop feature. Let's add, commit, and push our work. Even though the app is working well, we could make a few tweaks that would add some nice polish. We'll go over a few of these in the next step.