4.4.5 Capture the Task Item ID in the Dragstart Event

So why do we need the `dragstart` event? We already have a draggable element. Couldn't we simply listen for the `drop` event and attach our element to the drop zone (in our case, the task list)?

Unfortunately, although this operation might seem basic, there are quite a few events and functions that need to execute in order for it to succeed. It's a bit like a magician performing a trick with a sleight of hand. Although it might appear like the element is physically moving to a new place, what's happening behind the scenes is that we're recording the `data-task-id` of the `<li>` during the event. When the item is dropped, we have to use that value to locate the element in the DOM so that we can move it within the DOM to its final destination.

Because you created a task item and appended it to the task list in a previous lesson, you know that this is not magic—it's just basic DOM manipulation. The `dragstart` event is the key link that contains the information about the dragged element. Let's learn how to use it.

Use Event Delegation

First let's examine the `dragstart` event. We'll need to attach the `dragstart` event listener to each task item so we can capture each unique task item id from the `data-task-id` attribute. Depending on how many task items we have, that could mean a lot of event listeners. Can you think of a different way to achieve this goal?

Let's use event delegation. This will allow us to attach our event listener to the ancestor element that can listen for the event on all its descendant elements. But which ancestor element should we use?

We could choose `<body>` or `<html>`, but having too broad of an approach could lead to events being handled for elements not in the scope of our operation as well as performance lags due to the increased time for the event to propagate up the DOM. It's safer to choose the most direct ancestor element in order to limit the event listener's scope to only the necessary elements. In this case, it's the `<main>` element. This is because it is the parent element of the task list elements, which will contain all the task item elements, hence making the `<main>` element the grandparent element of the task item elements.

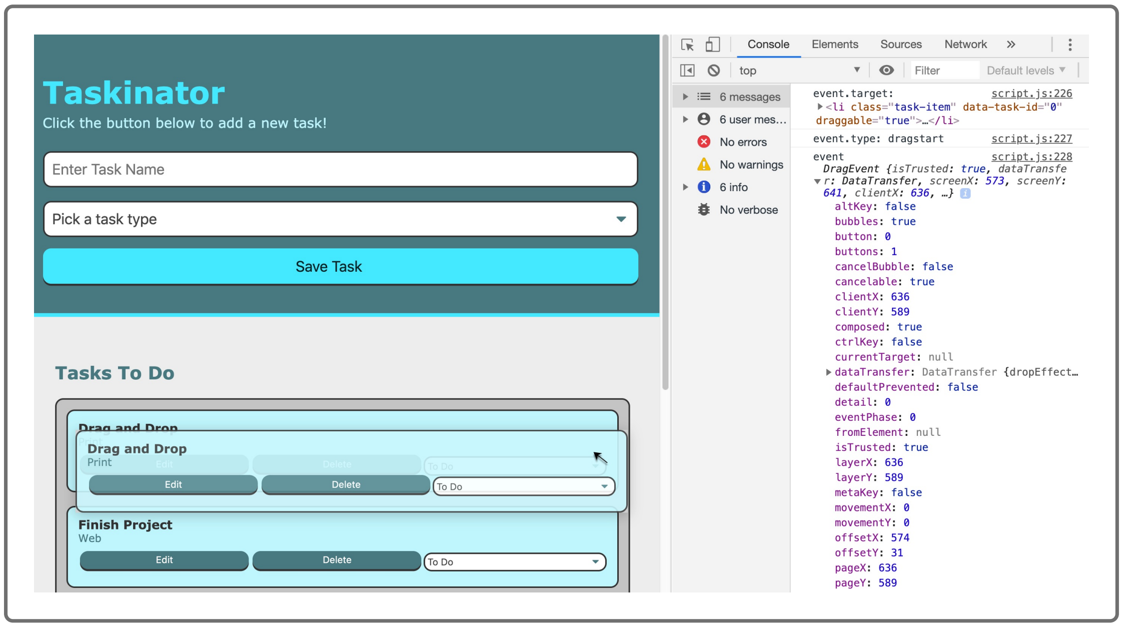
Use the `pageContentEl` DOM element to reference the `<main>` element and delegate the `dragstart` listener to it. Add the following expression to the bottom of the `script.js` file:

```
pageContentEl.addEventListener("dragstart", dragTaskHandler);
```

Define this event handler `dragTaskHandler()` to verify that the event listener is operating correctly. Remember to place the `dragTaskHandler()` above the event listener. Otherwise, we will get an `Uncaught ReferenceError` on the `dragTaskHandler()` callback in the event listener. So, let's place it right above all of the event listeners at the bottom of the file:

```
var dragTaskHandler = function(event) {
  console.log("event.target:", event.target);
  console.log("event.type:", event.type);
  console.log("event", event);
}
```

Let's save the file and refresh the browser. Create three tasks, then open the console. Drag the first task away from the Tasks To Do list. Look in the console to determine when the `dragstart` event is triggered. Expand the `event` object to display the following in the console:



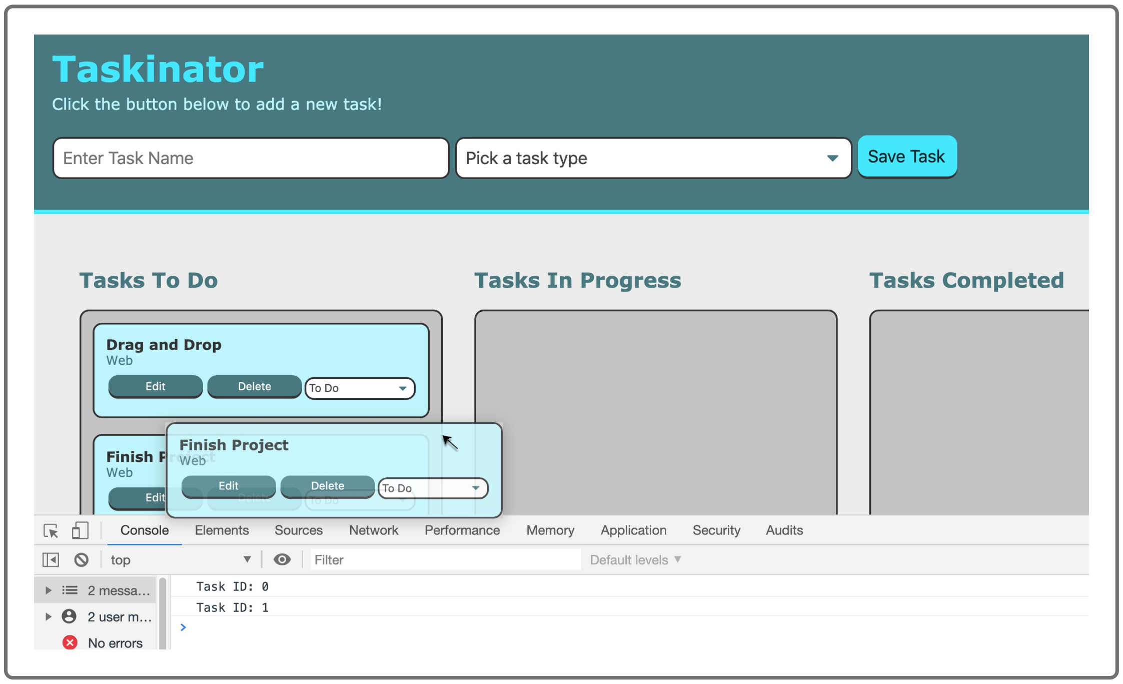
```
event.target:                                  script.js:226
 ▶<li class="task-item" data-task-id="0"
 draggable="true">…</li>
event.type: dragstart                          script.js:227
event                                          script.js:228
  DragEvent {isTrusted: true, dataTransfe
 ▼r: DataTransfer, screenX: 573, screenY:
  641, clientX: 636, …}
    altKey: false
    bubbles: true
    button: 0
    buttons: 1
    cancelBubble: false
    cancelable: true
    clientX: 636
    clientY: 589
    composed: true
    ctrlKey: false
    currentTarget: null
   ▶dataTransfer: DataTransfer {dropEffect…
    defaultPrevented: false
    detail: 0
    eventPhase: 0
    fromElement: null
    isTrusted: true
    layerX: 636
    layerY: 589
    metaKey: false
    movementX: 0
    movementY: 0
    offsetX: 574
    offsetY: 31
    pageX: 636
    pageY: 589
```

As you can see in the image above, the `event.target` DOM element is the task item element that has the `data-task-id` attribute with the value "0". This value is unique to this task item element and will be different each time we drag a different task item. Because the `event` object is connected to the DOM element, we can capture the unique task id from this `data` attribute. The `target` property is critical in every event because it contains the information about the element affected by the event. We also displayed the `event.type` property, which verified this was a `dragstart` event.

Next let's use the `event.target` DOM element to get the unique task id. Revise the `dragTaskHandler()` function to this:

```
var dragTaskHandler = function(event) {
  var taskId = event.target.getAttribute("data-task-id");
  console.log("Task ID:", taskId);
}
```

Save and refresh the browser, then add a task and drag it. Then add another task and drag that one. You should see the following in the console:

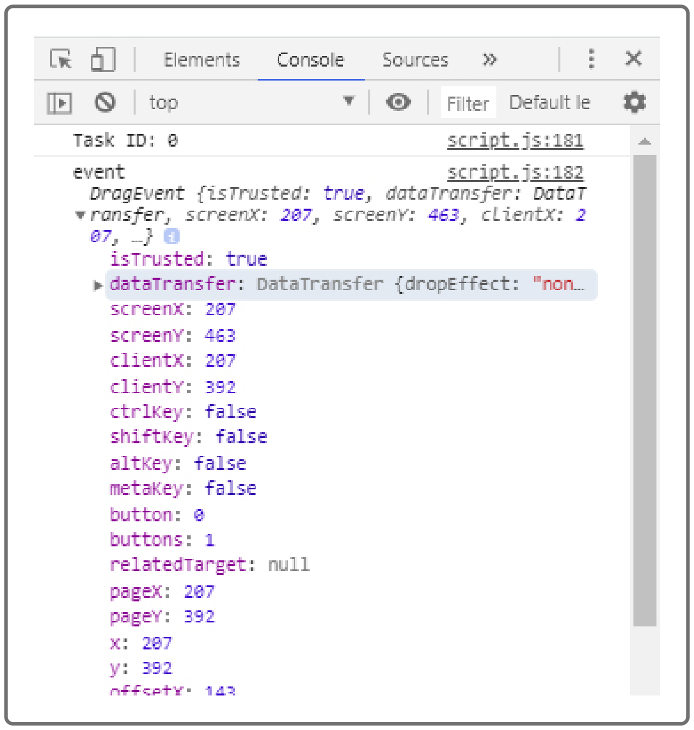


```
Task ID: 0
Task ID: 1
```

Now that we can retrieve the `data-task-id` attribute, how do we store it so it can be retrieved in the drop event? Let's examine the event object for clues. Start by logging the event once again in our function. The function should look like this:

```
var dragTaskHandler = function(event) {
    var taskId = event.target.getAttribute("data-task-id");
    console.log("Task ID:", taskId);
    console.log("event", event);
}
```

Save your code, refresh the browser, and then add and drag a list item. The event should be logged in the console below the `taskId`. Expand the event and find `dataTransfer` in its property list:



```
Task ID: 0                                          script.js:181
event                                               script.js:182
  DragEvent {isTrusted: true, dataTransfer: DataT
 ▾ransfer, screenX: 207, screenY: 463, clientX: 2
  07, …}
    isTrusted: true
  ▸ dataTransfer: DataTransfer {dropEffect: "non…
    screenX: 207
    screenY: 463
    clientX: 207
    clientY: 392
    ctrlKey: false
    shiftKey: false
    altKey: false
    metaKey: false
    button: 0
    buttons: 1
    relatedTarget: null
    pageX: 207
    pageY: 392
    x: 207
    y: 392
```

Examining the `event` object, note the `dataTransfer` property, which is outlined in the image above. This is the data storage property and we use it in a similar way to how we used the Web Storage API in `localStorage`. We'll want to save the `taskId` in the event object itself, and the `dataTransfer` property is the place to put it. Using the `setData()` and `getData()` methods, we're able to store data to and retrieve data from the `dataTransfer` property on the object. We'll use those methods for storing and retrieving our unique task id from the event.

Next, let's delete all of the current `console.log()` statements in the function. Then let's add the following expression directly following the `taskId` expression in the `dragTaskHandler()` function. It will store the `taskId` in the `dataTransfer` property of the event.

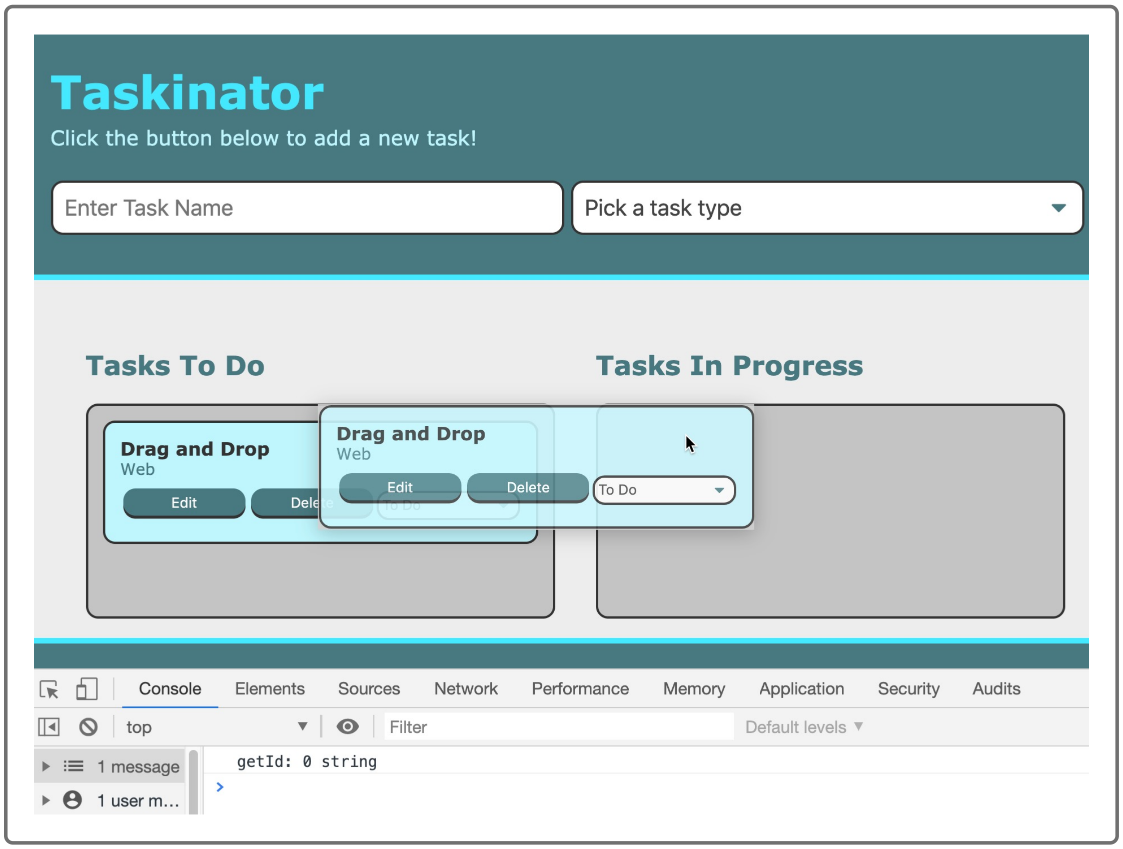
```
event.dataTransfer.setData("text/plain", taskId);
```

Notice how the `setData()` method receives two arguments: the first argument states the data's format and the second argument states the data's value.

To verify that our dataTransfer property stored the data-task-id attribute, we need to use the `getData()` method. Add the following expressions after the `setData()` statement in the `dragTaskHandler()` function. Notice that we're using commas inside the `console.log()` to separate a sequence of values we want to see.

```
var getId = event.dataTransfer.getData("text/plain");
console.log("getId:", getId, typeof getId);
```

Save the file and refresh the browser. Then save a task and drag it to view the following in the console:



```
getId: 0 string
```

You can see in the console that we were able to verify that the `data-task-id` was stored successfully in the `dataTransfer` property object: the `taskId` is reported (0), and the type of the `taskId` is reported (string).

Because `dataTransfer` is a property of the drag event, we can access the `data-task-id` later in the drop event because both drag and drop are of the type `DragEvent`. In other words, since each action shares the same event type, we can access properties set during dragging later during dropping.

LEGACY LORE

The `dataTransfer` property was originally used on the desktop application for file transfer, which is how most of us first became familiar with the drag-and-drop utility.

PAUSE

Why is having the data type format important in the Drag and Drop API?

Think about a scenario when you want the user to be able to drag and drop a link into an input field. We could use a conditional statement that only allows links to be stored by filtering for the format `"text/uri-text"`.

Hide Answer

Now when we're dragging the `<li>` item, we can grab its `data-task-id` and place that value in the `DragEvent` object. When we then drop the `<li>` item, we'll generate a drop event and have access to the same `DragEvent` object again. We'll then be able to retrieve the `data-task-id` from the `DragEvent` object and use it to find our element—the `<li>`

element that we're dropping—in the DOM with the `querySelector()` method. That will then allow us to remove the `<li>` element from its current DOM location and append it to its new DOM location.