4.4.6 Define the Drop Zone

Although this API is called the Drag and Drop API, there are actually quite a few events that we can use to fire off functions at critical points in the process. One useful event is the `dragover`. This event is triggered when an element is hovered over another element. We'll use the `dragover` event handler to define the **drop zone**, or where the draggable element can be dropped.

In our Taskinator app, we want to drop the task items onto one of the three task lists. We therefore want to restrict the drop zone to just the task lists. To begin, let's add the event listener to the task lists for the `dragover` event and then examine the event in the console.

To begin to define our drop zone, we'll need a `dragover` event on all three task lists. Let's use event delegation to add an event listener for the `dragover` event to the parent element of the task lists, the `<main>` element, just as we used event delegation for the `dragstart` event.

Add the following expression to the bottom of the `script.js` file:

```
pageContentEl.addEventListener("dragover", dropZoneDragHandler);
```

PAUSE

Why did we leave out the parentheses for the `dropZoneDragHandler` in the `addEventListener()` argument?

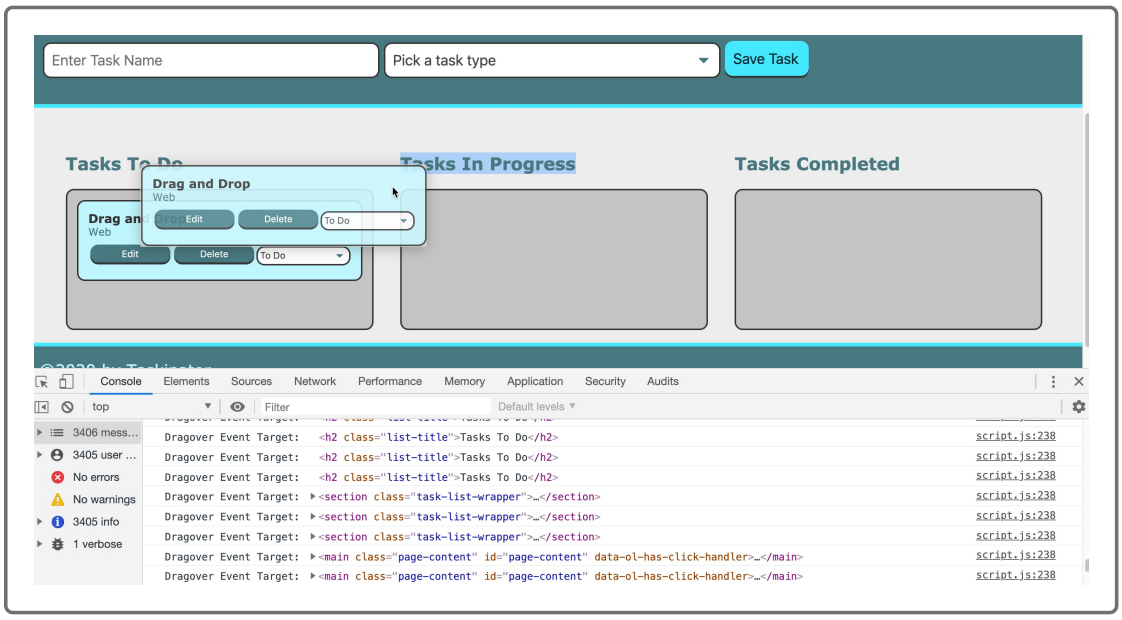
Adding the parentheses will call this function immediately, so we pass the reference to the function as a callback, which is triggered by the event.

Hide Answer

For the `dropZoneDragHandler`, let's verify that our event listener is working and see which element is being targeted with the following expression placed above the event listeners and beneath the `dragTaskHandler` function:

```
var dropZoneDragHandler = function(event) {
  console.log("Dragover Event Target:", event.target);
};
```

Save the file and refresh the browser. Then add a task and drag it, and check the console, as shown here:



As we can see in the console, the `dragover` event continuously fires when an element is dragged over another element. In the course of a few seconds, the event handler is executed thousands of times. This is quite different than other events we've used that fired once, such as the `click` or `dragstart` events.

In the preceding image, we can also see that the `target` property of the `dragover` event is the element that is being dragged over, not the element being dragged. This is different than the `dragstart` event, which identified the element being dragged.

We can also drag over different elements on the document, such as the ancestor and sibling elements of the task lists (e.g., the `<main>` and `<h2>` elements). This isn't actually a good thing because we want our task items to drop onto the task lists and not elsewhere on the page. For now, let's focus on making our element droppable. We can finetune the drop zone later.

If we try to drop the task item now, we can see that it bounces back to its original position. This is because the default behavior of this event prevents elements from being dropped onto one another. But this is the behavior we want, so we need to disable or prevent this action. Luckily, there is a method in the `event` object—aptly named `preventDefault()`—

that we can use to do this. Let's use this method in the `dropZoneDragHandler()` to define our drop zone.

Replace the `console.log()` in the event handler with the following statement:

```
event.preventDefault();
```

Save the file and refresh the browser, then add a task. Try dragging and dropping it, and you'll see dropping still has no effect. The `dragover` event allowed the element to be dropped, but we're still missing something. Ah, yes—the all-important `drop` event! This will be the final step in completing the drag-and-drop operation.

But before we get to that step, let's finetune the drop zone. How can we limit the drop zones to only the task list elements? We need to find out if the `target` property is either a descendant element of the task list or the task list element itself. Luckily, there is a method to help with this: `closest()`.

Let's take a look at the `closest()` method and see how it works:

```
targetElement.closest(selector);
```

Similar to the `querySelector()` method, the `closest()` method will originate a search from the `targetElement` for an element that contains the **selector**. If the element with the selector is found, it's returned as a DOM element, but if it's not found, it will return null.

Here are the major differences between the `querySelector()` method and the `closest()` method:

- The `querySelector()` method searches down from the `targetElement` to descendant elements such as the children,

grandchildren, and so on.

- The `closest()` method searches up through the `targetElement` to ancestor elements, such as the parent and grandparent elements, until is reaches the document root, which is the highest level of the DOM.

- The `closest()` method searches the `targetElement` as well as the ancestor elements, whereas the `querySelector()` does not.

So now that we know how the `closest()` method works, let's see how we can use it to define our drop zone.

Let's employ the following code, and let's study it a bit:

```
event.target.closest(".task-list")
```

The `.closest()` method searches up through the ancestors of the target (the `event.target` here, which is the drop zone). It looks for an element that matches the selector (`.task-list`, in this case). If it can find a matching element among the ancestors of the target, then it returns that element. If the target itself matches the selector, then the target itself is returned. If it can't find a matching element, it returns null.

So, in our case, it will return the `task-list` item when it encounters it in the ancestor tree of the elements we're dragging the task across. This is precisely the behavior we are looking for.

Can you create a conditional statement that will return true if the `target` element is either the task list element or a descendant of the task list element?

If you solved this question with either of the following two statements, you are correct:

```
if (event.target.closest(".task-list") !== null)

if (event.target.closest(".task-list"))
```

The first statement says if the target is not null or not outside the task list element, then return the DOM element, which will evaluate to a truthy value.

The second statement says if the target is within the task list, then return the DOM element, which will also evaluate to a truthy value.

We'll use the second statement because it is concise and lacks a double negative. Now let's incorporate the `preventDefault()` method to create a drop zone as long as the target is within the task list elements. It's important to point out that since the class attribute `task-list` is a part of all three task list elements, we're making all three lists part of the drop zone.

Let's add this conditional statement so the `dropZoneDragHandler()` now looks like this:

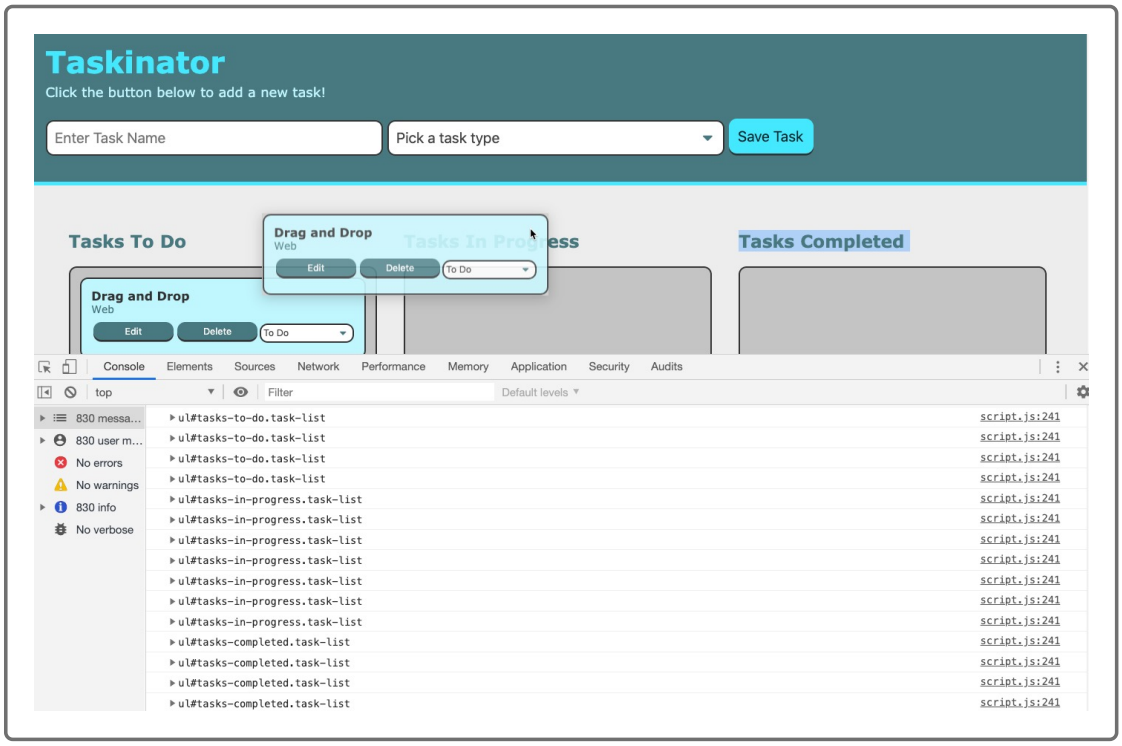
```
var dropZoneDragHandler = function(event) {
  var taskListEl = event.target.closest(".task-list");
  if (taskListEl) {
    event.preventDefault();
    console.dir(taskListEl);
  }
};
```

Verify that the code is working by saving the file and refreshing the browser. Now create a task and initiate the `dragover` event.

In the conditional statement for the `dropZoneDragHandler()` function, we are limiting the droppable area to be the task list or a child element of the task list. If the `target` element isn't a task list element or has an ancestor that is a task list, the conditional statement will return null. This resolves

the condition to false, thus keeping the default behavior which doesn't allow a drop to occur.

Let's save and refresh, then add a task and drag it to a task list. We should see the following:



This image shows the display in the console, which is logging the elements returned by the callback. The elements are represented by strings of concatenated attributes, which means everything is an attribute of the element in question.

In each case, it's an element (`ul` always in our case) followed by an id (`#`) followed by a class (`.`) The class is what we're targeting with `closest()`, and the id corresponds to the section we're dragging the task over. By dragging the task item over the different task lists, we can therefore identify which task list was dragged over by the id attribute.

Go ahead and remove the `console.dir()` now that the drop zone has been verified.

Excellent work! Let's move onto the next step and finish the `drop`.