

5.3.5 Convert Lists into Sortables


The jQuery UI documentation lists three drag-related widgets:

- Draggable
- Droppable
- Sortable

If you click through the different pages on the [jQuery UI website \(https://jqueryui.com/\)](https://jqueryui.com/), you'll see some great examples of each of these methods in action. Sortable in particular has a demo of connecting two "sortable" lists together, as seen in the following screenshot:

Sortable

Reorder elements in a list or grid using the mouse.



Sort items from one list into another and vice versa, by passing a selector into the `connectWith` option. The simplest way to do this is to group all related lists with a CSS class, and then pass that class into the sortable function (i.e., `connectWith: '.myClass'`).

[view source](#)

Want to learn more about the sortable interaction? Check out the [API documentation](#).

The Sortable widget looks like it could knock out two features at once: allowing elements to be dragged within the same column to re-order them and allowing elements to be dragged across columns.

To see exactly how you would use this in your own project, click on the "view source" link on the jQuery UI website to unmask this specific example, or click the "API documentation" link to see all of the options that come with Sortable.

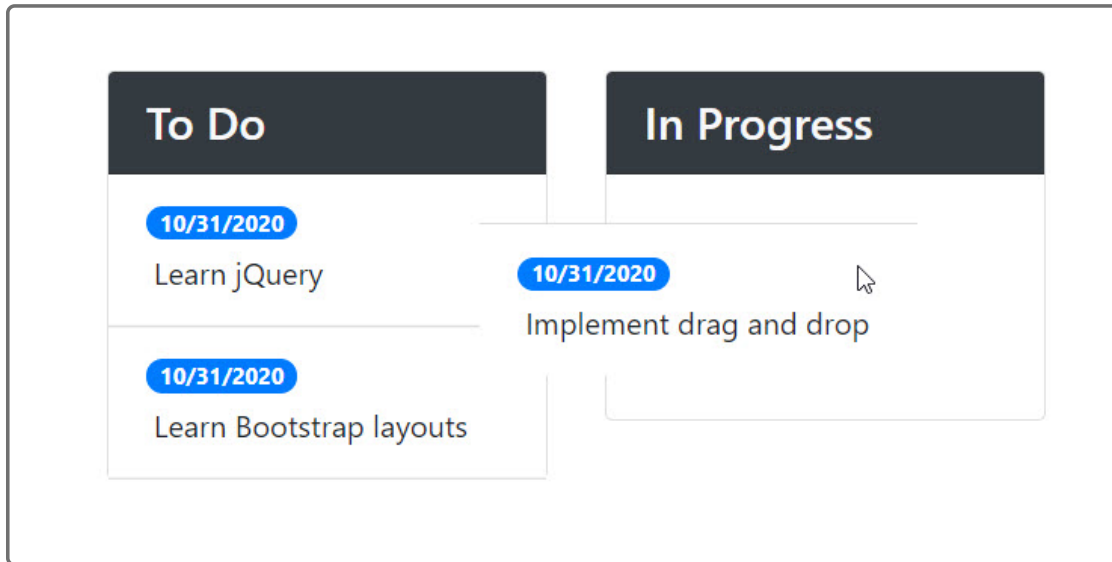
Let's try turning the columns in Taskmaster into sortables. Keep in mind that the actual lists we want to become sortable are the `` elements with the class `list-group`. We'll use a jQuery selector to find all `list-group` elements and then call a new jQuery UI method on them.

Add the following block of code to the `script.js` file:

```
$(".card .list-group").sortable({
  connectWith: $(".card .list-group")
});
```

```
});
```

Save and test out the app in the browser. Tasks can now be dragged within the same column and across other columns. The following screenshot demonstrates what you should see:



The jQuery UI method, `sortable()`, turned every element with the class `list-group` into a sortable list. The `connectWith` property then linked these sortable lists with any other lists that have the same class.

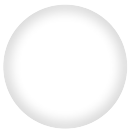
Visit the [Sortable widget documentation](https://api.jqueryui.com/sortable/) (<https://api.jqueryui.com/sortable/>) and look over some of the other options that can be applied, as well as some of the events you can listen in on. You won't need to use most of these for now, but it's good to know that they're there.

Update the `sortable()` method in `script.js` to look like this now:

```
$(".card .list-group").sortable({
  connectWith: $(".card .list-group"),
  scroll: false,
  tolerance: "pointer",
  helper: "clone",
```

```
activate: function(event) {
  console.log("activate", this);
},
deactivate: function(event) {
  console.log("deactivate", this);
},
over: function(event) {
  console.log("over", event.target);
},
out: function(event) {
  console.log("out", event.target);
},
update: function(event) {
  console.log("update", this);
}
});
```

We've added a few more options like `helper: "clone"` that tells jQuery to create a copy of the dragged element and move the copy instead of the original. This is necessary to prevent click events from accidentally triggering on the original element. We also added several event listeners like `activate`, `over`, and `out`.



REWIND

Some of these browser events are the same ones you've used in the past. The names just happen to be a little different with jQuery versus regular JavaScript. The regular JavaScript syntax to capture the start of a drag would look like this:

```
pageContentEl.addEventListener("dragstart", function(event)
});
```

Thankfully, jQuery (and jQuery UI) will handle all of the `event.dataTransfer.setData()` logic behind the scenes, so you don't have to worry about the minute details of drag-and-drop.

Save and test the app in the browser. Watch the DevTools console for the corresponding `console.log()` statements to see when each event triggers. To summarize:

- The `activate` and `deactivate` events trigger once for all connected lists as soon as dragging starts and stops.
- The `over` and `out` events trigger when a dragged item enters or leaves a connected list.
- The `update` event triggers when the contents of a list have changed (e.g., the items were re-ordered, an item was removed, or an item was added).

Events like `activate` and `over` would be great for styling. We could change the color of elements at each step to let the user know dragging is working correctly. We'll save those UI enhancements for later, though. For now, we're only concerned with the `update` event, because an updated list signifies the need to re-save tasks in `localStorage`. Remember that an update can happen to two lists at once if a task is dragged from one column to another.

PAUSE

In the event callback, how do you know which list is currently being affected by the update? What JavaScript keyword would refer to the list at hand?

The `this` keyword.

[Hide Answer](#)

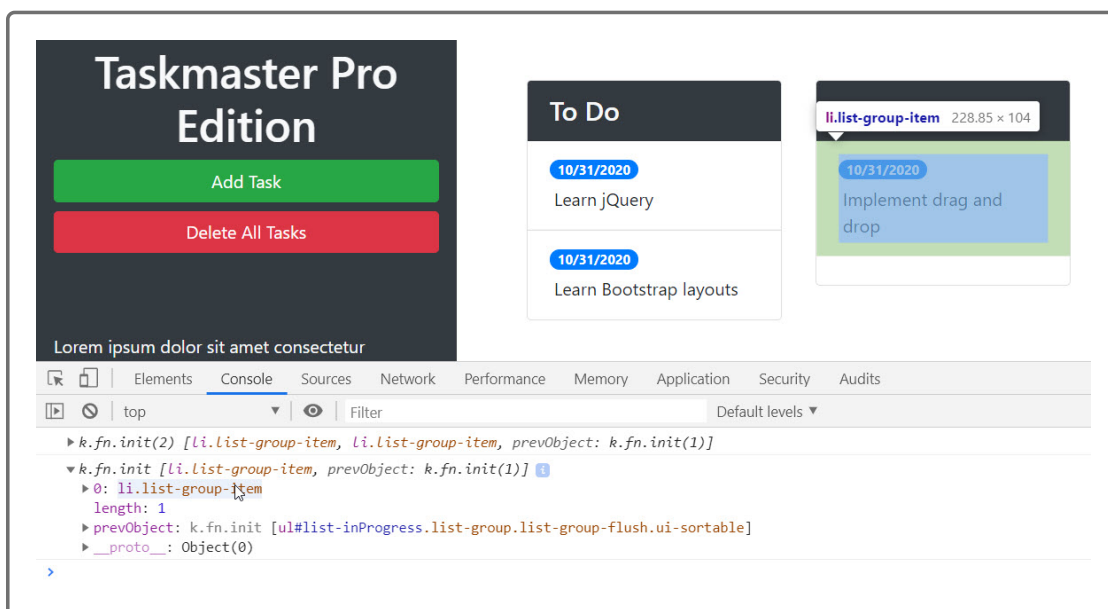
We're already console logging `this` from our previous tests, but that's a regular JavaScript `this`. Let's turn it into a jQuery `this` by wrapping it in `$()` so we can start using jQuery methods.

Change the sortable's `update()` method to the following:

```
update: function(event) {  
  console.log($(this).children());  
}
```

Save and test again by dragging a task to update one of the columns. Note that console logging a jQuery variable will display more information than what you're used to seeing. Expand the logged arrays/objects, though, and you'll find that they do include references to the original DOM elements.

The following screenshot demonstrates what you should see:

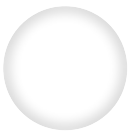


The `children()` method returns an array of the list element's children (the `` elements, labeled as `li.list-group-item`). No coincidence, tasks are saved in `localStorage` as an array. The order of these `` elements and the indexes in the task arrays should match one-to-one. This means we just need to loop over the elements, pushing their text values into a new tasks array.

Update the `update()` method to look like this:

```
update: function(event) {  
  // loop over current set of children in sortable list  
  $(this).children().each(function() {  
    console.log($(this));  
  });  
}
```

jQuery's `each()` method will run a callback function for every item/element in the array. It's another form of looping, except that a function is now called on each loop iteration. The potentially confusing part of this code is the second use of `$(this)`. Inside the callback function, `$(this)` actually refers to the child element at that index.



REWIND

This is another example of scoped variables. Remember, if an inner function declares a variable with the same name as a variable declared outside, the inner function will use the closer of the two declarations. A similar situation would look like this:

```
var name = "Bob";  
  
var sayName = function() {  
  var name = "Alice";  
  console.log(name);  
}
```

```
};  
  
sayName(); //prints "Alice"  
  
console.log(name); // prints "Bob"
```

However, if the nested use of `$(this)` is overwhelming, the [jQuery documentation for each\(\)](https://api.jquery.com/each/) provides some examples of how you could avoid having to use `$(this)` when using `each()`.

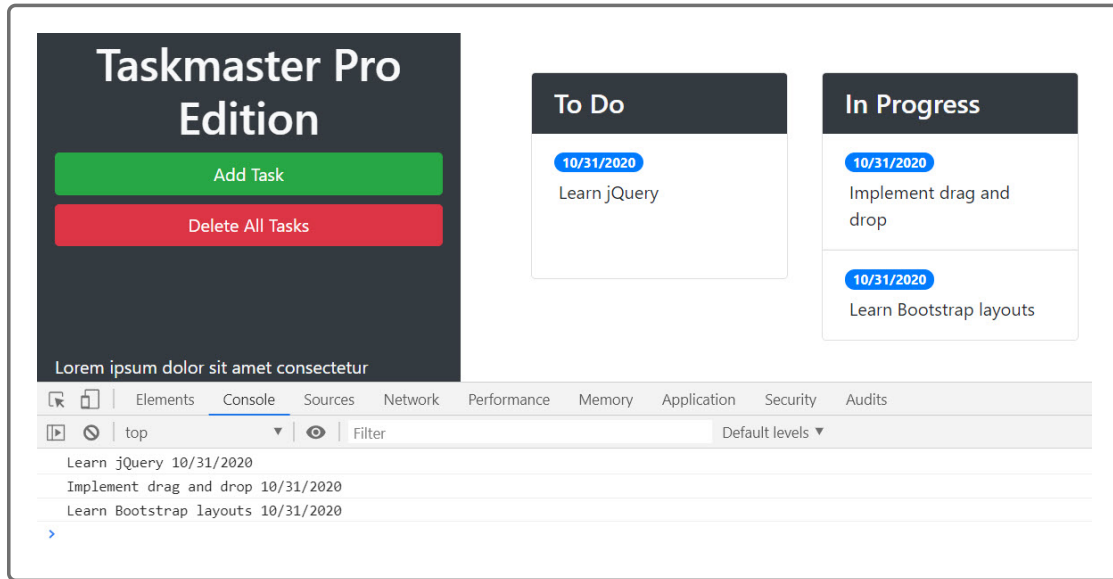
Because the nested `$(this)` refers to the task `` element, you can use additional jQuery methods to strip out the task's description and due date. jQuery's `find()` method is perfect for traversing through child DOM elements.

Update the `children().each()` loop to look like the following:

```
// loop over current set of children in sortable list  
$(this).children().each(function() {  
  var text = $(this)  
    .find("p")  
    .text()  
    .trim();  
  
  var date = $(this)  
    .find("span")  
    .text()  
    .trim();  
  
  console.log(text, date);  
});
```

Test the app again to ensure that the correct `text` and `date` values are being console logged. The following screenshot demonstrates what you

should see:



These values ultimately need to go in an array. Let's declare a new array before the looping starts. Then instead of console logging the values in the loop, we'll combine them into an object and push this object into the array.

The entire code inside the `update()` method should now look like this:

```
// array to store the task data in
var tempArr = [];

// loop over current set of children in sortable list
$(this).children().each(function() {
  var text = $(this)
    .find("p")
    .text()
    .trim();

  var date = $(this)
    .find("span")
    .text()
    .trim();

  // add task data to the temp array as an object
  tempArr.push({
```

```
    text: text,  
    date: date  
  });  
});  
  
console.log(tempArr);
```

The next step is to use the `tempArr` array to overwrite what's currently saved in the `tasks` object. We're dealing with multiple lists, though, not just `todo`. How do we know when to update `tasks.todo` versus `tasks.inReview` or `tasks.done`?

We solved a similar problem earlier in the project when dealing with blur events. Each list has an `id` attribute that matches a property on `tasks`. For instance: `id="list-inReview"`. We'll use the same approach here to get the list's ID and update the corresponding array on `tasks`.

Below the `children()` loop, add the following:

```
// trim down list's ID to match object property  
var arrName = $(this)  
  .attr("id")  
  .replace("list-", "");  
  
// update array on tasks object and save  
tasks[arrName] = tempArr;  
saveTasks();
```

Save and test the app again in the browser. Drag and drop a few tasks, then refresh the browser. The tasks should now stay in their respective columns!

You just added some valuable interactions to the Taskmaster app. Thanks to jQuery UI, it didn't take nearly as long to implement as it would have with regular JavaScript.

Next, we'll continue to leverage jQuery UI to create an easier way to delete tasks.

© 2020 Trilogy Education Services, a 2U, Inc. brand. All Rights Reserved.