4.5.4 Save Tasks to an Array

Currently, all the data associated with a task is kept with its respective DOM element on the page. This means that if we tried to save this data to `localStorage`, we'd have to find every task item and pull the important data from it. We can't use a `querySelector()` method to save the entire task item element and save it to `localStorage`, because `localStorage` can only save data as a string and DOM elements are difficult to convert to strings.

All the task data is set up in a way that's difficult for us to organize and store, so we'll need a different storage solution. This solution will have to package each task's ID, name, type, and status together, like this:

```
var taskDataObj = {
  id: 1,
  name: "Add localStorage persistence",
  type: "Web",
  status: "in progress"
}
```

The good news is that we've already started this process of organizing the task data into an object. We did it in the second lesson when we sent the

task's data from `taskFormHandler()` to `createTaskEl()` as an argument. Now we just need to update that object to hold more information.

We'll have more than one task to store, obviously, so creating a variable for each one isn't maintainable or practical.

PAUSE

How might we keep a list of the task objects in one variable?

We could use an array of objects.

Hide Answer

We'll create a new variable in `script.js` to hold an array of task objects that looks like this:

```
var tasks = [
  {
    id: 1,
    name: "Add localStorage persistence",
    type: "Web",
    status: "in progress"
  },
  {
    id: 2,
    name: "Learn JavaScript",
    type: "Web",
    status: "in progress"
  },
  {
    id: 3,
    name: "Refactor code",
    type: "Web",
    status: "to do"
```

```
  }
];
```

The `tasks` array will look like that when we start adding tasks to it, but we can start by creating an empty array variable. Let's add the following code where we've declared the other variables, towards the top of the `script.js` file:

```
var tasks = [];
```

We've created an empty `tasks` array. This way, when we create a new task, the object holding all its data can be added to the array. We just need to update the object holding the task's data to also include its ID and status, both of which are only written to the associated DOM element.

In the `taskFormHandler()` function, let's update the `taskDataObj` variable to include a `status` property. Because this data gets sent to `createTaskEl()`, we can safely assume that the status will always have a value of `to do`. A task that has just been created can't possibly be `in progress` or `complete` yet.

Update the `taskDataObj` variable to look like this:

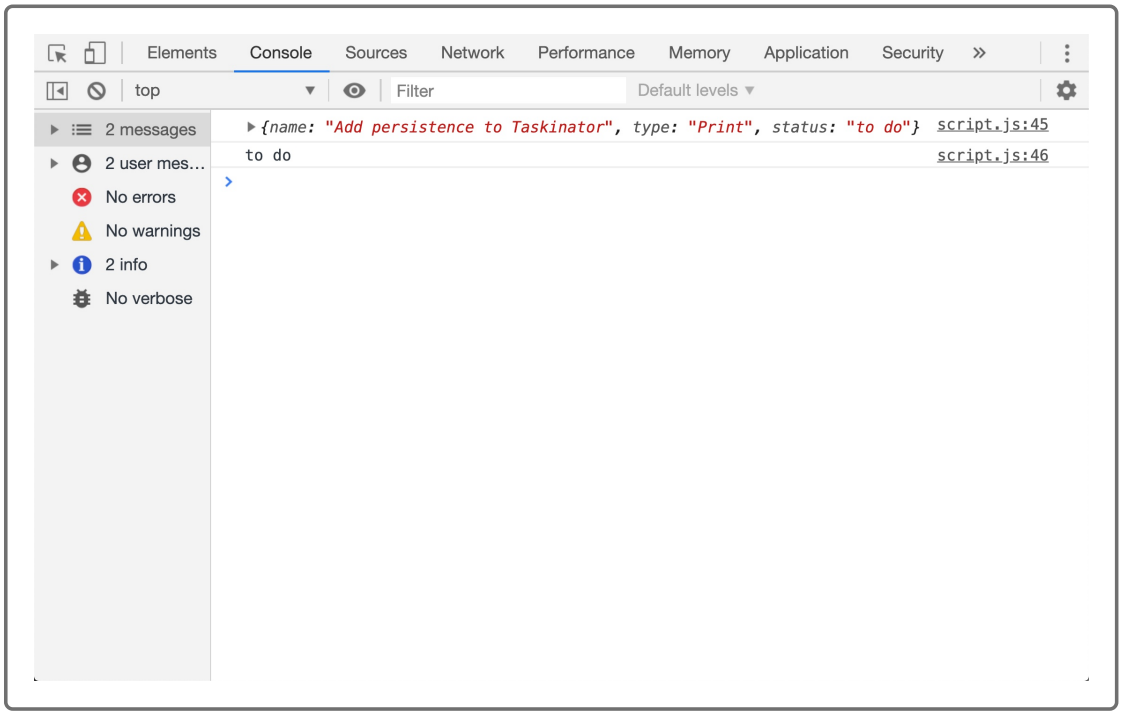
```
var taskDataObj = {
  name: taskNameInput,
  type: taskTypeInput,
  status: "to do"
}
```

Let's test this and add a `console.log()` into the `createTaskEl()` function. This will help us ensure that the new property gets to the function via the `taskDataObj` parameter we set up.

In `createTaskEl()`, add the following anywhere in the function:

```
console.log(taskDataObj);
console.log(taskDataObj.status);
```

Save `script.js`, refresh the app in the browser, and submit a new task. The two `console.log()` functions we added should look like this in the console:



As we can see, `createTaskEl()` now receives this new `status` property in its `taskDataObj` parameter.

The only thing missing from the task's object that we need to save is its ID. Luckily, we already have the value of the ID in the `taskIdCounter` variable and are using it in the `createTaskEl()` function. Now we need to add that value as a property to the `taskDataObj` argument variable and add the entire object to the `tasks` array, but how do we add something to an array?

Below `listItemEl.appendChild(taskInfoEl);` and above `taskIdCounter++;`, let's update `createTaskEl()` with the following code:

```
taskDataObj.id = taskIdCounter;

tasks.push(taskDataObj);
```

Remember, we're now managing two task lists: one in the visible GUI that's represented in the DOM, and one only found in our code in the form of an array of objects. So, when we edit or delete a task in the GUI/DOM, we need to do the same to the object that represents it in our array. That way, we can sync the array data we'll be putting in `localStorage` with the GUI/DOM data.

We did this by adding an `id` property to the `taskDataObj` argument and giving it a value of whatever `taskIdCounter` is at that moment. This way, the ID of the newly created DOM element gets added to the task's object as well. We can use that ID later to identify which task has changed for both the DOM and the `tasks` array.

IMPORTANT

> Just as we can reassign the value of an object property, we can also create new properties as needed.

Once we gave the task its ID value, we had to get that object into the `tasks` array, so we used an array method called `push()`. This method adds any content between the parentheses to the end of the specified array.

Developers use this method a lot, so let's go through a couple of examples:

```
var pushedArr = [1, 2, 3];

pushedArr.push(4);
// pushedArr is now [1,2,3,4]
```

```
pushedArr.push("Taskinator");
// pushedArr is now [1,2,3,4,"Taskinator"]

pushedArr.push(10, "push", false);
// pushedArr is now [1,2,3,4,"Taskinator",10,"push",false]
```

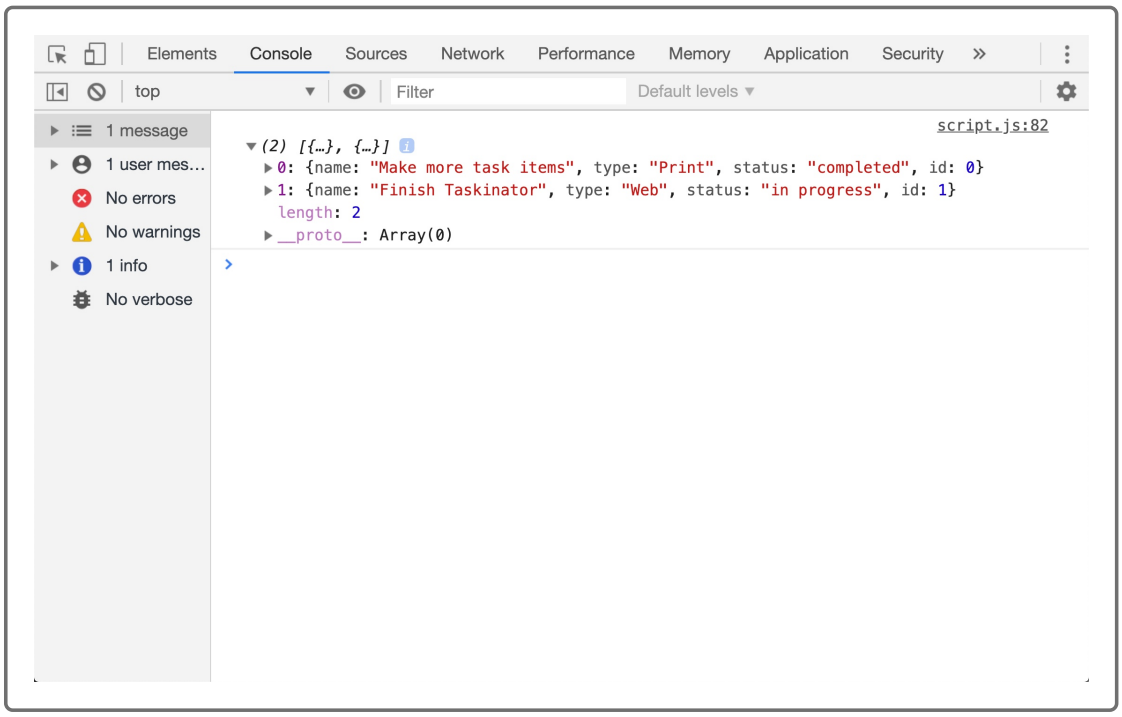
DEEP DIVE

DEEP DIVE

For more information, see the **MDN web docs on the array push method. (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/push)**

This update won't yield any visible differences on the page, but we can still test to make sure it works.

Save `script.js`, refresh the page, and create a task or two. Then, in the Chrome DevTools console, type `console.log(tasks)`. Pressing Enter to run the log function should return a printed list of the tasks in an array of objects, as shown here:



Now the `tasks` array stores any tasks we add to the page, along with all their important information.

So we've added the ability to save a task not only to the page but in the array as well. Because we also update and remove tasks as well as add them, we'll have to update the `completeEditTask()`, `taskStatusChangeHandler()`, `dropTaskHandler()`, and `deleteTask()` functions too.

Find and Edit Array Data

To sync the data presented by the DOM with the data stored in the `tasks` array, we'll change the functions for updating and deleting tasks in the DOM so that they also update and delete tasks in the `tasks` array.

We'll do this by looking up that task in the array by its ID value. Then we'll update its object properties or remove it from the `task` array entirely. How do we find an element in an array by its ID? We'll have to check each task in the array for an `id` property equal to the updated ID of the task.

In the `completeEditTask()` function, add this code beneath the two `querySelector()` methods so that it looks like this:

```
// THIS CODE IS ALREADY IN PLACE
taskSelected.querySelector("h3.task-name").textContent = taskName;
taskSelected.querySelector("span.task-type").textContent = taskType;

// loop through tasks array and task object with new content
for (var i = 0; i < tasks.length; i++) {
  if (tasks[i].id === parseInt(taskId)) {
    tasks[i].name = taskName;
    tasks[i].type = taskType;
  }
};
```

At each iteration of this `for` loop, we are checking to see if that individual task's `id` property matches the `taskId` argument we passed into `completeEditTask()`.

The only problem is `taskId` is a string and `tasks[i].id` is a number; so when we compare the two, we need to make sure that we are comparing a number to a number. This is why we wrap the `taskId` with a `parseInt()` function and convert it to a number for the comparison.

If the two ID values match, we have confirmed that the task at that iteration of the `for` loop is the one we want to update, so let's go ahead and do that by reassigning that task's `name` and `type` property to the new content submitted by the form when we finished editing it.

HIDE PRO TIP

To see if a task is changed in the `tasks` array, use `debugger;` before and after the `for` loop to check what the data looks like before and after it's updated!

Now that we got the `completeEditTask()` function to update the `task` array, let's turn our attention to the other functions that deal with updating tasks as well, starting with `taskStatusChangeHandler()`.

In `completeEditTask()`, we focused on updating a task's name or type. Here, we only need to worry about updating a task's status. Let's use a `for` loop again, but with a different property to reassign.

At the bottom of the `taskStatusChangeHandler()` function, add the following code:

```
// update task's in tasks array
for (var i = 0; i < tasks.length; i++) {
  if (tasks[i].id === parseInt(taskId)) {
    tasks[i].status = statusValue;
  }
}
```

Add another `console.log(tasks);` after the `for` loop so we can verify that it's working. Save `script.js`, create a task, and update its status through the task's `<select>` dropdown. After this, the console should display an updated `tasks` array reflecting that task's new status. We can't test it via drag and drop yet, as we haven't added this update functionality to its handler function. Let's do that now.

Let's add the following code to the bottom of `dropTaskHandler()`:

```
// loop through tasks array to find and update the updated task's stat
for (var i = 0; i < tasks.length; i++) {
  if (tasks[i].id === parseInt(id)) {
    tasks[i].status = statusSelectEl.value.toLowerCase();
  }
}
```

```
console.log(tasks);
```

Save `script.js`, create a task or two, and test it by dragging and dropping a task item into a different list. The `console.log()` we added should reflect the task's updated object in the `tasks` array.

That wraps it up for taking care of updating tasks; now let's delete them from the `tasks` array!

Delete Task from Array

Just as we used a `for` loop to identify and update a task's information, we're going to use a `for` loop to identify a task we want to remove. Since we're removing a task, however, we'll need to take a couple of extra steps to get there.

Let's find the `deleteTask()` function and add this code to the bottom of it:

```
// create new array to hold updated list of tasks
var updatedTaskArr = [];

// loop through current tasks
for (var i = 0; i < tasks.length; i++) {
  // if tasks[i].id doesn't match the value of taskId, let's keep that
  if (tasks[i].id !== parseInt(taskId)) {
    updatedTaskArr.push(tasks[i]);
  }
}

// reassign tasks array to be the same as updatedTaskArr
tasks = updatedTaskArr;
```

When we updated a task's information, all we had to do was overwrite its data. We didn't have to worry about creating something new because were

just modifying something that already existed. When we delete a task, however, we essentially have to create a new array of tasks that is identical to our current one, except it won't receive the task we're deleting.

What we did here is we create a new empty array variable called `updatedTaskArr`, then when we iterate through the current `tasks` array, we can check to see if the current task in the loop (`tasks[i]`) *does not* have that same ID value as the task we're looking to delete. If it's not the same task, then we know we should keep it, so we use the `.push()` method to add that task to the `updatedTaskArr` array.

When we're done with the `for` loop our `updatedTasksArr` should be almost the exact same as the `tasks` array, except it won't have the task that we want removed. Since the `tasks` array variable is the variable we need to keep up to date at all times, we reassign the `tasks` variable to `updatedTasksArr` so they are now the same and without the deleted task.

We just learned a lot of code for functionality that doesn't affect the page, but remember the end goal. Storing all the task's data as an array of objects in the JavaScript code will make it easier to save that data to `localStorage`.

Don't forget to add, commit, and push the code to the GitHub feature branch!