4.5.6 Load Tasks from localStorage

There's only one more function to build before we're ready to use Taskinator in daily life! In the last step, we saved data because we can't load data that hasn't been saved yet. Now we'll build a `loadTasks()` function that does the following:

1. Gets task items from `localStorage`
2. Converts tasks from the stringified format back into an array of objects
3. Iterates through tasks array and creates task elements on the page from it

That last part should seem pretty familiar to us, as we've already created task elements for the page. We can probably look to the `createTaskEl()` function code for help in making this `loadTasks()` function happen.

With that said, less code will be provided to us. We're in the home stretch now, so it's time to use our knowledge!

HIDE HINT

A lot of the code we'll need to write is already in `createTaskEl()`. We'll just need to apply it in a different context.

Let's create a function called `loadTasks` right below the `saveTasks()` function, to keep them together. Once we've created that function, let's add some pseudocode to keep us on track. Add the three pseudocode steps from the list above to the function we just created. Remember, save them as JavaScript comments, to inform us of what the code does.

HIDE PRO TIP

To comment out code or text in JavaScript (or any other language), highlight all the content to be commented and use `Command+/` (Mac) or `Ctrl+/` (Windows).

The loop we create will have a lot going on. We'll pseudocode those steps when we get to it, but let's make sure the three main pieces work first. We can't print tasks to the page if we haven't retrieved them from `localStorage` yet, so let's do that first.

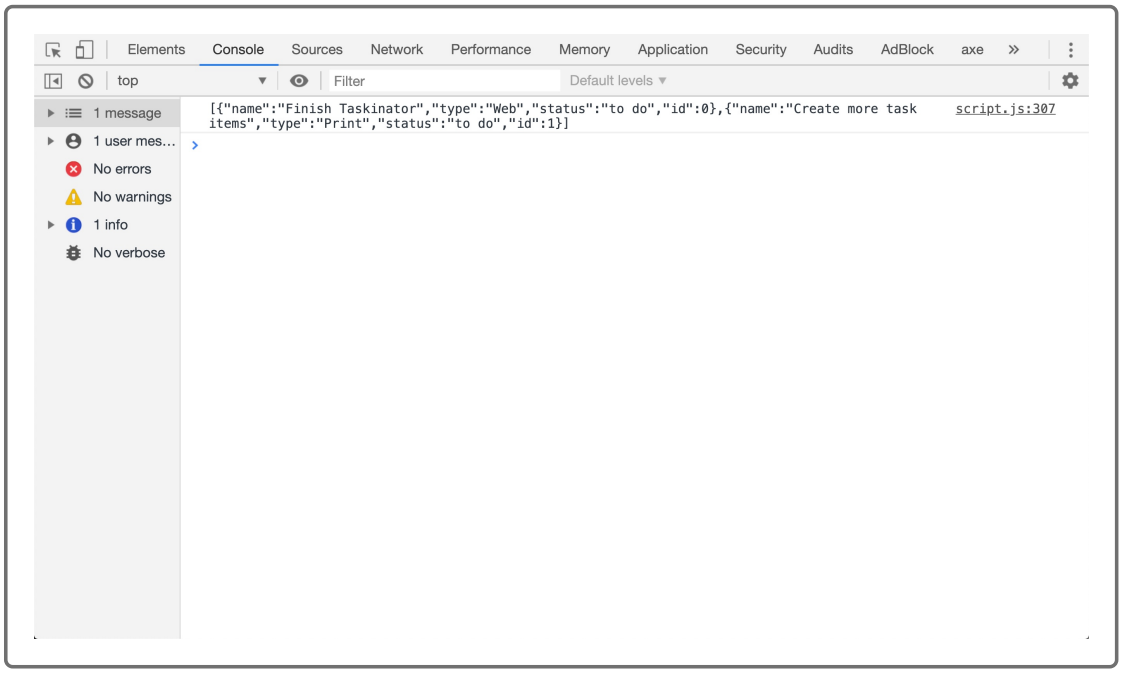
Because we saved these tasks from the `tasks` variable previously, let's load them back into that variable. The first line of real code in `loadTasks()` should reassign the `tasks` variable to whatever `localStorage` returns. We need to ask ourselves a couple of things to do this:

- Should we use the `var` keyword if we're just reassigning the `tasks` variable we created at the top of the page?

- What method do we use to retrieve data from `localStorage`, and what item in `localStorage` are we retrieving?

Once we've reassigned the `tasks` variable to the data we've retrieved from `localStorage`, we should test it. Add a `console.log()` right below the line of code we just wrote, and call `loadTasks()` at the very bottom of the `script.js` file. When we refresh the page, that function will run and try to find what's in `localStorage`.

When we run the function, the `console.log()` we added should look like this image:



If data comes back into `tasks` from `localStorage`, we should see the stringified version of the task array. If nothing returns for the tasks from `localStorage`, then `tasks` will have a value of `null`. Both outcomes would cause an issue, so let's consider how to fix both of them.

If nothing returns from `localStorage`, then `tasks` has a value of `null`. So when we try adding another task and use `tasks.push()`, we get an error because the `push()` method only works on arrays. To fix this issue, we should check if the value of `tasks` is `null`, and if it is, reassign `tasks` yet

again to an empty array. Let's do that now; then we'll worry about turning the string back into an array.

In `loadTasks()`, after we retrieve and `console.log()` the `tasks` variable, do the following:

1. Check if `tasks` is equal to `null` with an `if` statement.
2. If it is, set `tasks` back to an empty array by reassigning it to `[]` and adding a `return false`. We don't want this function to keep running with no tasks to load onto the page.
3. If it's not `null`, we don't have to worry about it and we can skip the `if` statement's code block.

HIDE HINT

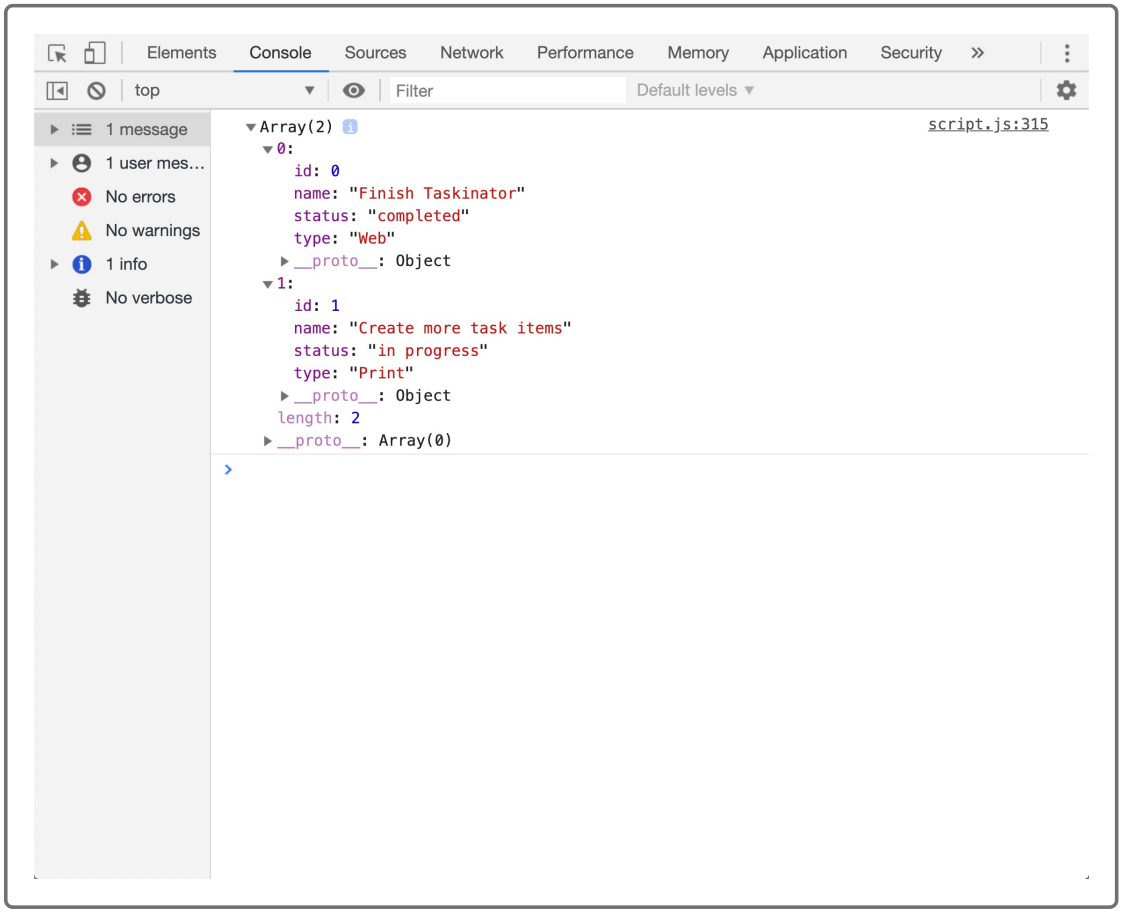
Remember that we can check for a null value in an `if` statement by using `(variableName === null)` or `(!variableName)` as the condition.

Any data in `localStorage` for `tasks` to retrieve is still in a string format, and we have to get it back into an array of objects. To do that, add the following code after the new `if` statement:

```
tasks = JSON.parse(tasks);
```

We used `JSON.stringify()` previously to convert an array of objects to a string, so what might `JSON.parse()` do? Add a `console.log()` after that

code to see what happens to the `tasks` variable. It should look like this image in the console:



It turns it back into a real array of objects! That's great; with the data back to normal now, we can actually use it like an array of objects.

DEEP DIVE

DEEP DIVE

To learn more, see the **MDN web docs for JSON.parse() (https://developer.mozilla.org/en-**

US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse)_.

With the data back in its array form, we can iterate over it. Because each element in the array is a task's object, with those values we can create DOM elements and print them to the page.

Create a `for` loop with a condition of `i < tasks.length`. Test it by using `console.log(tasks[i])` inside the loop. The console should print all the task objects we've created, one by one. Study the properties of each task object.

Once you've created and tested the loop, follow these instructions to print the task items to the page:

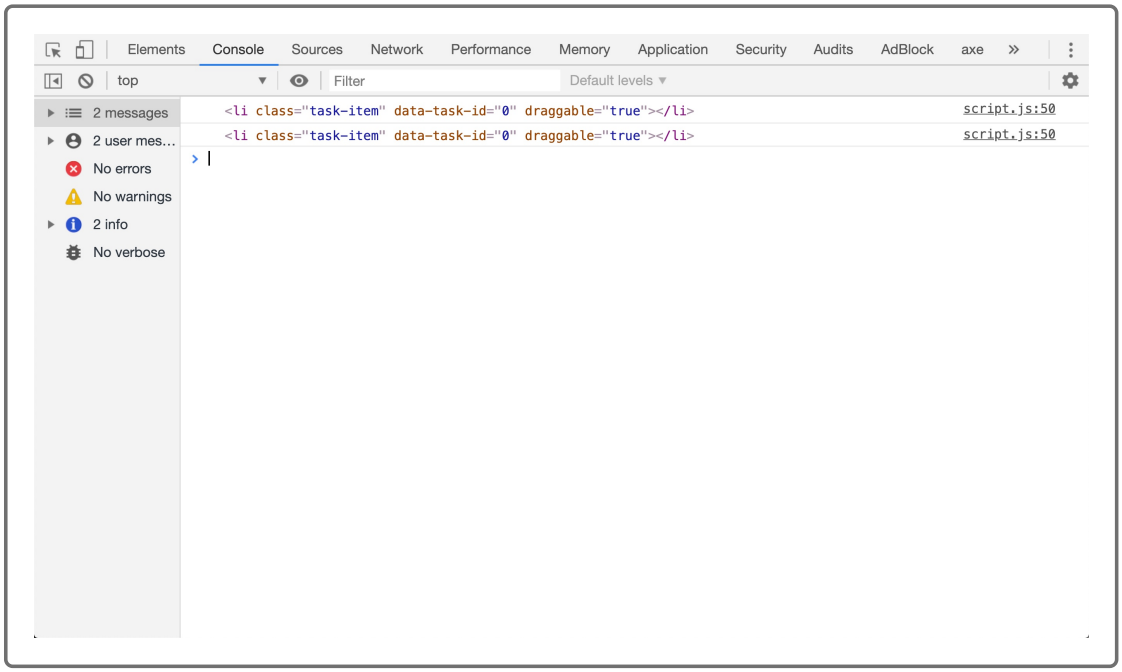
HIDE HINT

After completing each step, add a `console.log()` or `debugger;` statement in the loop to see if that task works!

1. To keep the task IDs in sync, reassign `task[i]`'s `id` property to the value of `taskIdCounter`.
2. Add `console.log(tasks[i])` after reassigning the `id` property. This way we can see them printing in order when we view it in the Console.
3. Create a `<li>` element and store it in a variable called `listItemEl`.
    - Give it a `classname` attribute of `task-item`.

- With `setAttribute()`, give it a `data-task-id` attribute with a value of `tasks[i].id`.
- With `setAttribute()`, give it a `draggable` attribute with a value of `true`.

If we `console.log(listItemEl)`, it should look like this image:



If more than one element has the same `data-task-id` attribute value, don't worry! We'll fix that at the end.

4. Create a `<div>` element and store it in a variable called `taskInfoEl`.
   - Give it a `classname` property of `task-info` to set the HTML `class` attribute.
   - Set its `innerHTML` property to the following:

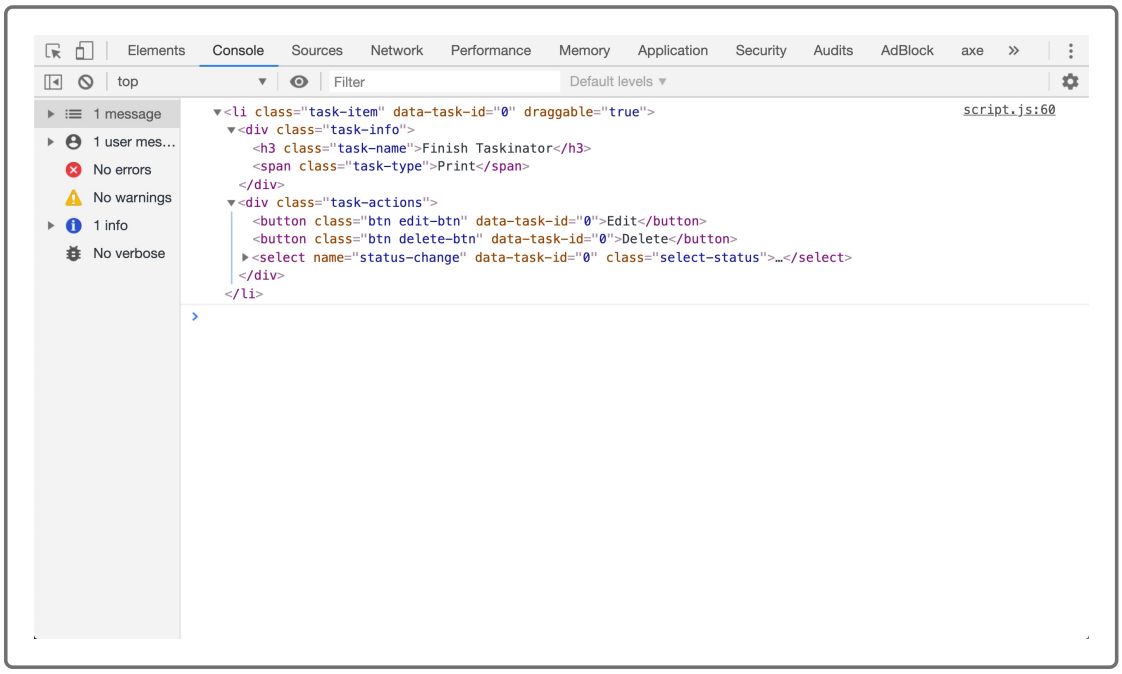
```
taskInfoEl.innerHTML = "<h3 class='task-name'>" + tasks[i].name +
```

5. Append `taskInfoEl` to `listItemEl`.

6. Create the actions for the task by creating a variable called `taskActionsEl` and giving it a value of `createTaskActions()` with `tasks[i].id` as the argument.

7. Append `taskActionsEl` to `listItemEl`.

Check that `taskActionsEl` was appended to `listItemEl` correctly by using a `console.log(listItemEl);`. It should look like this image:



8. With an `if` statement, check if the value of `tasks[i].status` is equal to `to do`.

    - If yes, use `listItemEl.querySelector("select[name='status-change']").selectedIndex` and set it equal to 0.

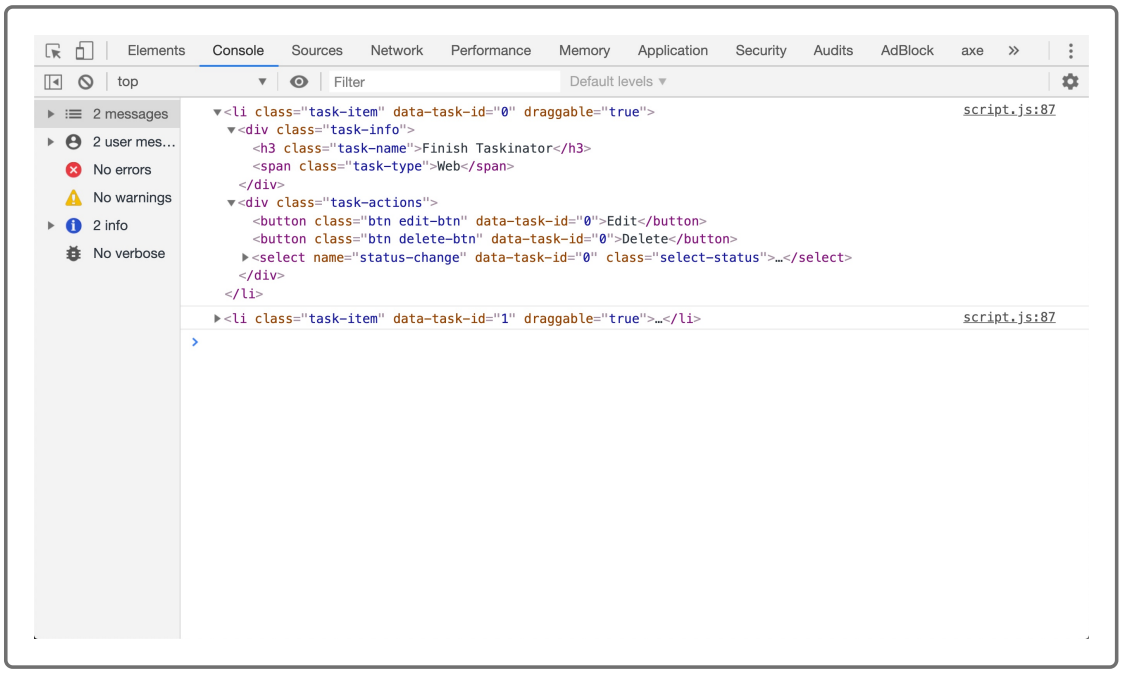
    - Append `listItemEl` to `tasksToDoEl`.

9. With `else if`, check if the value of `tasks[i].status` is equal to `in progress`.

- If yes, use `listItemEl.querySelector("select[name='status-change']").selectedIndex` and set it equal to 1.
- Append `listItemEl` to `tasksInProgressEl`.

10. With `else if`, check if the value of `tasks[i].status` is equal to `complete`.
    - If yes, use `listItemEl.querySelector("select[name='status-change']").selectedIndex` and set it equal to 2.
    - Append `listItemEl` to `tasksCompletedEl`.
11. Increase `taskIdCounter` by 1.

Add one more `console.log(listItemEl)` after incrementing the counter, then test. The elements should look like this image:



As this image shows, now the elements get unique `data-task-id` values since we're incrementing the `taskIdCounter` variable in each iteration of the loop.

This seems like a lot of work to go into a `for` loop, but most of this code is already written in `createTaskEl()`. We can refer to that code to see how it works and replace some values with the values of `tasks[i]`'s properties. Don't be afraid to refer to `createTaskEl()` and other functions we've created throughout the build of Taskinator—this won't all come from memory yet!

Once complete, save and task it again. The tasks now load to their correct lists based on status! However, some tasks in the list appear in a different order than when it was saved. This has to do with the drag-and-drop functionality. It would take some work to fix, but it's not an app-breaking quirk.

You now have a fully working task-tracking application—a project you've done for yourself! As you take on more work this application can help you stay organized by creating, editing, deleting, and saving the tasks.

Note that because the new `for` loop uses a ton of repeated code from `createTaskEl()`, we're entering into technical debt. If we change how something functions or looks, we have to change code in both `createTaskEl()` and `loadTasks()`, potentially causing confusion and errors.

You could refactor `loadTasks()` to use `createTaskEl()` as well when printing the tasks. Feel free to jump to the deployment step to finalize the Git process, but otherwise, get ready to take on this refactor!