Local storage is a web browser feature that lets web applications store key-value pairs persistently within a user's browser. This allows web apps to save data during one session, then retrieve it in a later page session.

In this TODO application, you'll learn how to handle form inputs, manage local storage, perform CRUD (Create, Read, Update, Delete) operations on tasks, implement event listeners, and toggle UI elements.

Step 6

Now, you will work on opening and closing the form modal.

In earlier projects, you learned how to add and remove classes from an element with `el.classList.add()` and `el.classList.remove()`. Another method to use with the `classList` property is the `toggle` method.

The `toggle` method will add the class if it is not present on the element, and remove the class if it is present on the element.

Example Code

```
element.classList.toggle("class-to-toggle");
```

Add an event listener to the `openTaskFormBtn` element and pass in a `"click"` event for the first argument and an anonymous callback function for the second argument.

Inside the callback function, use the `classList.toggle()` method to toggle the `"hidden"` class on the `taskForm` element.

Now you can `click` on the "Add new Task" button and see the form modal.

A *modal* is an element that prevents all interaction with elements outside it until the modal has been dismissed.

The HTML `dialog` element has a `showModal()` method that can be used to display a modal dialog box on a web page.

Example Code

```
dialogElement.showModal();
```

Add an event listener to the *closeTaskFormBtn* variable and pass in a *click* event for the first argument and a callback function for the second argument.

For the callback function, call the *showModal()* method on the *confirmCloseDialog* element. This will display a modal with the *Discard* and *Cancel* buttons.

Step 8

If the user clicks the *Cancel* button, you want to cancel the process and close the modal so the user can continue editing. The HTML *dialog* element has a *close()* method that can be used to close a modal dialog box on a web page.

Example Code

```
dialogElement.close();
```

Add an event listener to the *cancelBtn* element and pass in a *click* event for the first argument and a callback function for the second argument.

For the callback function, call the *close()* method on the *confirmCloseDialog* element.

Step 10

Now that you've worked on opening and closing the modal, it's time to get the values from the input fields, save them into the *taskData* array, and display them on the page.

To start, add a *submit* event listener to your *taskForm* element and pass in *e* as the parameter of your arrow function. Inside the curly braces, use the *preventDefault()* method to stop the browser from refreshing the page after submitting the form.

Step 11

You will need to determine whether the task being added to the *taskData* array already exists or not. If the task does not exist, you will add it to the array. If it does exist, you will update it. To accomplish this, you can use the *findIndex()* method.

The *findIndex()* array method finds and returns the index of the first element in an array that meets the criteria specified by a provided testing function. If no such element is found, the method returns *-1*.

Here's an example:

Example Code

```
const numbers = [3, 1, 5, 6];
const firstNumLargerThanThree = numbers.findIndex((num) => num
> 3);


console.log(firstNumLargerThanThree); // prints index 2
```

Use *const* to declare a variable called *dataArrIndex* and assign it the value of *taskData.findIndex()*. For the *findIndex()* method, pass in an arrow function with *item* as the parameter.

Within the arrow function, check if the *id* property of *item* is strictly equal to the *id* property of *currentTask*.

Step 18 Passed

To make the *id* more unique, add another hyphen and use *Date.now()*.

Date.now() returns the number of milliseconds elapsed since *January 1, 1970 00:00:00 UTC*.

Example Code

```
console.log(Date.now()); // 1628586800000
```

To see the new result, click on the *"Add New Task"* button. Then add a title of *WALK DOG* and click on the *"Add Task"* button. Open up the console to see the result.

Step 21

Now that you have obtained the values from the input fields and generated an *id*, you want to add them to your *taskData* array to keep track of each task. However, you should only do this if the task is new. If the task already exists, you will set it up for editing. This is why you have the *dataArrIndex* variable, which provides the index of each task.

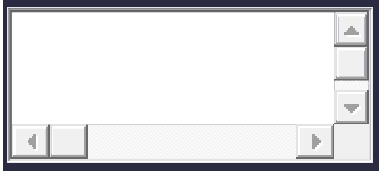
Create an *if* statement with the condition *dataArrIndex === -1*. Within the *if* statement, use the *unshift()* method to add the *taskObj* object to the beginning of the *taskData* array.

unshift() is an array method that is used to add one or more elements to the beginning of an array.

Example Code

```
const arr = [1, 2, 3];
arr.unshift(0);

// [0, 1, 2, 3]
console.log(arr);
```



Check Your Code (Ctrl + Enter)

Reset

Navigated to Step 21

No results found for 'title'

Cursors added: line 41 column 10, line 41 column 16

Step 41

To enable editing and deleting for each task, add an *onclick* attribute to both buttons. Set the value of the *onclick* attribute to *editTask(this)* for the *Edit* button and *deleteTask(this)* for the *Delete* button. The *editTask(this)* function will handle editing, while the *deleteTask(this)* function will handle deletion.

this is a keyword that refers to the current context. In this case, *this* points to the element that triggers the event – the buttons.

Step 44

You need to remove the task from the DOM using *remove()* and from the *taskData* array using *splice()*.

splice() is an array method that modifies arrays by removing, replacing, or adding elements at a specified index, while also returning the removed elements. It can take up to three arguments: the first one is the mandatory index at which to start, the second is the number of items to remove, and the third is an optional replacement element. Here's an example:

Example Code

```
const fruits = ["mango", "date", "cherry", "banana", "apple"];

// Remove date and cherry from the array starting at index 1
const removedFruits = fruits.splice(1, 2);
```

```
console.log(fruits); // [ 'mango', 'banana', 'apple' ]
console.log(removedFruits); // [ 'date', 'cherry' ]
```

Use the *remove()* method to remove the *parentElement* of the *buttonEl* from the DOM. Then use *splice()* to remove the task from the *taskData* array. Pass in *dataArrIndex* and *1* as the arguments of your *splice()*.

dataArrIndex is the index to start and *1* is the number of items to remove.

localStorage offers methods for saving, retrieving, and deleting items. The items you save can be of any JavaScript data type.

For instance, the *setItem()* method is used to save an item, and the *getItem()* method retrieves the item. To delete a specific item, you can utilize the *removeItem()* method, or if you want to delete all items in the storage, you can use *clear()*.

Here's how you can save an item:

Example Code

```
localStorage.setItem("key", value); // value could be string, number, or any other data type
```

A *myTaskArr* array has been provided for you. Use the *setItem()* method to save it with a key of *data*.

After that, open your browser console and go to the *Applications* tab, select *Local Storage*, and the freeCodeCamp domain you see.

If you check the "Application" tab of your browser console, you'll notice a series of *[object Object]*. This is because everything you save in *localStorage* needs to be in string format.

To resolve the issue, wrap the data you're saving in the `JSON.stringify()` method. Then, check local storage again to observe the results.

You can use `localStorage.removeItem()` to remove a specific item and `localStorage.clear()` to clear all items in the local storage.

Remove the `data` item from local storage and open the console to observe the result. You should see `null`.