LEARN LOCALSTORAGE BY BUILDING A TODO APP

Introduction:

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 1:

In this project, you will learn how `localStorage` works in JavaScript by building a Todo app. LocalStorage is a web storage feature of JavaScript that lets you persist data by storing the data as a **key:value** pair.

The HTML and CSS for this project have been provided for you. Take a look at the files to get yourself familiarized with them.

Begin by accessing the `task-form`, `confirm-close-dialog`, and `open-task-form-btn` elements with the `getElementById()` method. Save them in the variables `taskForm`, `confirmCloseDialog`, and `openTaskFormBtn`.

Step 2:

You need to access more elements with the `getElementById()` method. This time you need the `close-task-form-btn`, `add-or-update-task-btn`, and `cancel-btn` elements. Save them in the variables `closeTaskFormBtn`, `addOrUpdateTaskBtn`, and `cancelBtn`.

Step 3:

Next, access the `discard-btn`, `tasks-container`, and `title-input` elements using the `getElementById()` method. Save them in variables named `discardBtn`, `tasksContainer`, and `titleInput`, respectively.

Step 4:

The last set of elements you need to get from the HTML file are the `date-input` and `description-input` elements. Save them in the variables `dateInput` and `descriptionInput` respectively.

Step 5:

Create a `taskData` constant and set it to an empty array. This array will store all the tasks along with their associated data, including title, due date, and description. This storage will enable you to keep track of tasks, display them on the page, and save them to `localStorage`.

Use `let` to create a `currentTask` variable and set it to an empty object. This variable will be used to track the state when editing and discarding tasks.

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.

Step 7:

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 9:

If the user clicks the `Discard` button, you want to close the modal showing the `Cancel` and `Discard` buttons, then hide the form modal.

Add a click event listener to `discardBtn`, then use the `close()` method on the `confirmCloseDialog` variable. Also, use `classList` to toggle the class `hidden` on `taskForm` so the form modal will close too.

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 callback function. If no such element is found, the method returns -1. The callback should return a truthy value to indicate a matching element has been found, and a falsy value otherwise.

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 12:

When a user creates a task, it should be saved in an object.

Create a const variable called taskObj and assign it the value of an empty object.

Then below that, add a console statement that logs the value of `taskObj` to the console.

Open up the console to see the empty object.

Step 13:

Inside your `taskObj`, add an `id` property name. For the value use the value of the `titleInput`.

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

Step 14:

The user should be able to input a title with upper and lowercase letters. But for the `id`, the value should be all lowercase.

Update your `titleInput.value` to be all lowercase. You can use the `toLowerCase()` method to do this.

Example Code:

```
str.toLowerCase();
```

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 of `"walk dog"`.

Step 15:

Right now, your `id` value is a lowercase string. But the final result should be a hyphenated string.

Start by chaining the `split` method to the `titleInput.value` to split the string into an array of words. For the separator, use a space character(`" "`).

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 of `['walk', 'dog']`.

Step 16:

Now that your `id` is an array of words, you can use the `join` method to turn the result back into a string. For the separator, use a hyphen(`-`).

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 of `"walk-dog"`.

Step 17:

There is one last thing you will need to do to make this `id` unique.

But first, place the entire value below inside an embedded expression `${}`.

Example Code:

```
titleInput.value.toLowerCase().split(" ").join("-")
```

Then wrap that value in template strings.

In the next step, you will add a unique number to the end of the `id` value to make it truly unique.

Step 18:

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()); // 162858680000
```

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 19:

Now it is time to add the remaining properties to the taskObj object.

Retrieve the values from the titleInput, dateInput, and descriptionInput fields, and then save them in the properties title, date, and description of the taskObj object.

Add a new task and open up the console to see the taskObj object with the new properties.

Step 20:

Now that you have finished testing your taskObj, you can remove the console.log(taskObj) statement.

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);
```

Step 22:

Now that you have saved the task in the `taskData` array, you should display the task on the page by looping through it.

Use `forEach()` on `taskData`, then destructure `id`, `title`, `date`, `description` as the parameters. Don't return anything yet.

Step 23:

Using arrow syntax complete the `forEach` callback function. Inside the callback function body use an addition assignment to set the `innerHTML` of `tasksContainer` to empty backticks.

Step 24:

Create a `div` element with the class of `task`. Utilize template strings to set the `id` attribute of the `div` to the `id` you destructured from the task data.

Step 25:

Create a `p` element and use template strings to set its content to the `title` you destructured. Right before the content of the `p` element, create a `strong` element with the text `Title:`.

Step 26:

Similarly to the previous step, create another `p` element, and interpolate the `date` you destructured as the text content. Inside this paragraph, create a `strong` element with the text `Date:`.

Step 27:

Create one more `p` element and interpolate the `description` you destructured as the text. Also, create a `strong` element inside the paragraph with the text `Description:`.

Step 28:

To allow for task management, you need to include both a delete and an edit button for each task.

Create two `button` elements with the `type` attribute set to `button` and the `class` attribute set to `btn`. Set the text of the first button to `Edit` and the text of the second button to `Delete`.

Step 29:

After adding the task to the page, you should close the form modal to view the task. To do this, utilize classList to toggle the hidden class on the taskForm element.

Step 30:

If you attempt to add another task now, you'll notice that the input fields retain the values you entered for the previous task. To resolve this, you need to clear the input fields after adding a task.

Instead of clearing the input fields one by one, it's a good practice to create a function that handles clearing those fields. You can then call this function whenever you need to clear the input fields again.

Use arrow syntax to create a reset function and set it to a pair of curly braces.

Step 31:

Inside the reset function, set each value of titleInput, dateInput, descriptionInput to an empty string.

Also, use classList to toggle the class hidden on the taskForm and set currentTask to an empty object. That's because at this point, currentTask will be filled with the task the user might have added.

Step 32:

Remove the existing code toggling the class of hidden on taskForm and call the reset function instead. This would clear the input fields and also hide the form modal for the user to see the added task.

Step 33:

Also, remove the existing code toggling the class `hidden` on `taskForm` inside the `discardBtn` event listener and call the `reset` function instead. That's because when you click the `Discard` button, everything in the input fields should go away.

Step 34:

You should display the `Cancel` and `Discard` buttons to the user only if there is some text present in the input fields.

To begin, within the `closeTaskFormBtn` event listener, create a `formInputsContainValues` variable to check if there is a value in the `titleInput` field **or** the `dateInput` field **or** the `descriptionInput` field.

Step 35:

Create an `if` statement to check if `formInputsContainValues` is true. If `formInputsContainValues` is true, indicating that there are changes, use the `showModal()` method on `confirmCloseDialog`. Otherwise, if there are no changes, call the `reset()` function to clear the input fields and hide the form modal.

Step 36:

You can enhance code readability and maintainability by refactoring the `submit` event listener into two separate functions. The first function can be used to add the input values to `taskData`, while the second function can be responsible for adding the tasks to the DOM.

Use arrow syntax to create an `addOrUpdateTask` function. Then move the `dataArrIndex` variable, the `taskObj` object, and the `if` statement into the `addOrUpdateTask` function.

Step 37:

Use arrow syntax to create an `updateTaskContainer` function. Then move the `taskData.forEach()` and its content from the `taskForm`'s `submit` event listener into the newly created function.

Step 38:

Inside the `addOrUpdateTask` function, call the `updateTaskContainer` and `reset` functions.

Step 39:

Now remove the `reset()` call inside the `taskForm` submit event listener and call the `addOrUpdateTask` function instead.

Step 40:

There's a problem. If you add a task, and then add another, the previous task gets duplicated. This means you need to clear out the existing contents of `tasksContainer` before adding a new task.

Set the `innerHTML` of `tasksContainer` back to an empty string.

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 42:

Create a `deleteTask` function using arrow syntax. Pass `buttonEl` as the parameter and define an empty set of curly braces for the function body.

Step 43:

You need to find the index of the task you want to delete first.

Create a `dataArrIndex` variable and set its value using the `findIndex()` method on the `taskData` array. Pass `item` as the parameter for the arrow callback function, and within the callback, check if the `id` of `item` is equal to the `id` of the `parentElement` of `buttonEl`.

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.

Step 45:

Use arrow syntax to create an `editTask` function. Pass in `buttonEl` as the parameter and add empty curly braces for the body.

Step 46:

As you did in the `deleteTask` function, you need to find the index of the task to be edited.

Create a `dataArrIndex` variable. For its value, utilize the `findIndex()` method on `taskData`. Pass `item` as the parameter to its callback function and check if the `id` of `item` is equal to the `id` of the `parentElement` of `buttonEl`.

Step 47:

Use square bracket notation to retrieve the task to be edited from the `taskData` array using the `dataArrIndex`. Then, assign it to the `currentTask` object to keep track of it.

Step 48:

The task to be edited is now in the currentTask object. Stage it for editing inside the input fields by setting the value of titleInput to currentTask.title, dateInput to currentTask.date, and descriptionInput to currentTask.description.

Step 49:

Set the innerText of the addOrUpdateTaskBtn button to Update Task.

Step 50:

Finally, display the form modal with the values of the input fields by using classList to toggle the hidden class on taskForm.

Step 51:

At this point, editing a task won't reflect when you submit the task. To make the editing functional, go back to the if statement inside the addOrUpdateTask function. Create an else block and set taskData[dataArrIndex] to taskObj.

Step 52:

If the user attempts to edit a task but decides not to make any changes before closing the form, there is no need to display the modal with the Cancel and Discard buttons.

Inside the closeTaskFormBtn event listener, use const to create another variable named formInputValuesUpdated. Check if the user made changes while trying to edit a task by verifying that the titleInput value **is not equal to** currentTask.title, or the dateInput value **is not**

equal to `currentTask.date`, or the `descriptionInput` value **is not equal to** `currentTask.description`.

Step 53:

Now add `formInputValuesUpdated` as the second mandatory condition in the `if` statement using the `AND` operator.

This way, the `Cancel` and `Discard` buttons in the modal won't be displayed to the user if they haven't made any changes to the input fields while attempting to edit a task.

Step 54:

`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.

Step 55:

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.

Step 56:

Now that you have the `myTaskArr` array saved in `localStorage` correctly, you can retrieve it with `getItem()` by specifying the key you used to save the item.

Use the `getItem()` method to retrieve the `myTaskArr` array and assign it to the variable `getTaskArr`. Then, log the `getTaskArr` variable to the console to see the result.

Step 57:

The item you retrieve is a string, as you saved it with `JSON.stringify()`. To view it in its original form before saving, you need to use `JSON.parse()`.

Use `getItem()` to retrieve the `myTaskArr` array again. This time, wrap it inside `JSON.parse()`, assign it to the variable `getTaskArrObj` and log the `getTaskArrObj` to the console.

Check the console to see the difference between `getTaskArr` and `getTaskArrObj`.

Step 58:

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`.

Step 59:

Using `localStorage.clear()` won't just delete a single item from local storage but will remove all items.

Remove `localStorage.removeItem()` and use `localStorage.clear()` instead. You don't need to pass in anything. You should also see `null` in the console.

Step 60:

Remove the `myTaskArr` array and all of the code for `localStorage` because you don't need them anymore.

Step 61:

Now you should save the task items to local storage when the user adds, updates, or removes a task.

Inside the `addOrUpdateTask` function, use `setItem()` to save the tasks with a key of `data`, then pass the `taskData` array as its argument. Ensure that you stringify the `taskData`.

This would persist data once the user adds or updates tasks.

Step 62:

You also want a deleted task to be removed from local storage. For this, you don't need the removeItem() or clear() methods. Since you already use splice() to remove the deleted task from taskData, all you need to do now is save taskData to local storage again.

Use setItem() to save the taskData array again. Pass in data as the key and ensure that taskData is stringified before saving.

Step 63:

If you add, update, or remove a task, it should reflect in the UI. However, that's not happening now because you have yet to retrieve the tasks. To do this, you need to modify your initial taskData to be an empty array.

Set taskData to the retrieval of data from local storage **or** an empty array. Make sure you parse the data coming with JSON.parse() because you saved it as a string.

Step 64:

You've retrieved the task item(s) now, but they still don't reflect in the UI when the page loads. However, they appear when you add a new task.

You can check if there's a task inside taskData using the length of the array. Because 0 is a falsy value all you need for the condition is the array length.

Here is an example checking the length of an array:

Example Code:

```
if (arr.length) {

  // do something

}
```

In that example, if `arr` has a length greater than `0`, the code inside the `if` statement block will run. If `arr` has a length of `0`, the code inside the `if` statement block will not run.

Check if there's a task inside `taskData`, then call the `updateTaskContainer()` inside the `if` statement block.

Step 65:

If you try to add a new task, edit that task, and then click on the `Add New Task` button, you will notice a bug.

The form button will display the incorrect text of `"Update Task"` instead of `"Add Task"`. To fix this, you will need to assign the string `"Add Task"` to `addOrUpdateTaskBtn.innerText` inside your `reset` function.

Step 66:

There are two bugs left to fix. First, if you create a task with only spaces for the title, an empty task will be created even though the title is required.

Fix that by adding a check at the beginning of the function for if `!titleInput.value.trim()`. If there's no title, show an alert with the text `Please provide a title` and `return` after that.

Step 67:

It is time to work on the final issue. If there is a to-do task with a special character like a quote inside of the name or description of the item, the application breaks. While it appears otherwise, the correct item will appear missing.

In order to fix this, we need to create a function called `removeSpecialChars` that takes a string as input and removes all special characters.

Step 68:

Call `removeSpecialChars` on the `title`, and `description` properties in your `taskObj`. For the `id` property, only call it on the `titleInput.value` part of the property value.

This will remove issues with broken task data.

With that you have completed the project.