

Task list web application: using `localStorage`

In this assignment, you will continue working on the todo list application that you started after the midterm exam. The objective is to load and save data to the "local storage" of the browser instead of using variables.

You will have to remove the `data.js` script from your page since we now use dynamic values.

Understand `localStorage`

The `localStorage` object is available in the browser. It allows applications to store data in the visitor's browser. Each domain / subdomain has its own `localStorage` namespace that applications can access. `localStorage` data is persistent and will remain available in the browser even if you close and restart it.

The data is stored using key-value pairs. However, **the keys AND the values** are stored as strings.

Understand the use of JSON

Because `localStorage` values can only be strings, it has limitations if you want to store other types (such as arrays, or objects). `JSON` solves that problem, by transforming Javascript objects and variables into strings and vice versa.


The following object: `{ abc: 1, b: ["hello", "no"] }` has the following JSON representation: `'{"abc":1,"b":["hello","no"]}'`. Notice the `'` and `"`: the JSON representation is a **STRING** that contains all the information found in the original object.

- You can transform from Javascript to JSON by using: `JSON.stringify` (returns a string)
- You can transform from JSON to Javascript by using: `JSON.parse` (returns a Javascript object)

We are going to use JSON to store all the "notes" data in the `localStorage` of the browser: it will replace the `taskData` from the `data.js` file.

Using `localStorage`

- `localStorage.getItem("myKey")` will return the value of the `myKey` key in the local storage.
- `localStorage.setItem("myKey", "value")` will set the `myKey` to the string value `"value"`.

 Some browsers don't support `localStorage` with a local `file:` URL. Chrome does.

Refactor the code to use `localStorage`

Step 1: make sure `localStorage` has data to use

In order to use `localStorage`, we need to have data to use. Make changes to your script to make sure `localStorage` has initial data you can use. Select a key that you want to use to store all your tasks, and fill it with initial data (you may reuse example data from `data.js`).

For example:

```
const taskData = [
  {
    title: "First task",
    description: "Just an example task. The description contains text.",
    dueDate: "2024-01-01",
  }
]

localStorage.setItem("youDidChangeThatValue", JSON.stringify(taskData))
```

You can double check the data has been set in the browser. Open the inspector, and browse to the **Application** tab. On the left hand side, you will find the **Storage** section, and most specifically the **Local storage**.

Step 2: use data from **localStorage** in your existing code

Refactor your existing code to make use of **localStorage** to load the tasks and generate the DOM elements.

```
const allTasks = JSON.parse(localStorage.getItem("didYouReadThat"))

// you can also use forEach or for(... of ...)
// be careful with variable scoping!
allTasks.map(
  (task, index) => {
    const element = document.createElement("p")
    element.id = `task-${index}`
    element.textContent = task.description
    document.getElementById("something").appendChild(element)
  }
)
```



It is probably a good idea to create a function that loads the tasks from localStorage.

Step 3: Save data to **localStorage**

In these steps, you will refactor your code so that the "Save" button actually saves the data to the local storage.

Step 3.1: create a new task

- Make sure the modal box opens when you click the "New task" button, and that it shows an empty form.
- Collect all the values of the form, and create a new **Task** object from it in Javascript.
- Append this new task to the list of existing tasks. Make sure you fetch the tasks from **localStorage** first, and you don't lose any data when saving a new task!

- You will need to convert the local storage data into JSON using `JSON.parse`.
- Convert this list to JSON (using `JSON.stringify`) and save it in the local storage.
- The local storage data has changed. You will need to refresh the list of tasks (see previous step).
- You should have a function dedicated to "refreshing the tasks" that you can call whenever you need.



You can add the element `element` to the array `example` by using `example.push(element)`. This is the equivalent to `my_list.append` in Python.

Step 3.2: edit an existing task

The save process will be the same as for a new task. But you will need to:

- find the ID of the task that you are editing (the "index" of the task in the list of tasks). You can set the value as a hidden `<input>` in the modal box, in order to be able to retrieve it with Javascript.
- replace the values in the `taskList[taskId]` element with the values that are set in the modal dialog box.
- save all your tasks to the local storage, including the updated task
- refresh the list of tasks from local storage
- you can also delete the existing task from the list, and append a new updated task at the end. Choose the option you feel most comfortable with!

You will need to save all tasks data and refresh the list of tasks from local storage quite often. Write functions dedicated to these tasks!

Step 3.3: delete an existing task

The deletion process happens in the same way: find the index of the task you want to delete, and remove the corresponding element from the array (you may want to use `splice` - see below).

Check your application and make sure it works as expected

Hints: array manipulation

Let's consider the array: `const array = ["a", "b", 3, 4, 5, "end"]`

- `array.splice(a, b)` will extract `b` elements from the array starting from index `a`. This method **CHANGES THE ARRAY ITSELF**.
 - so, `array.splice(3, 2)` will give `[4,5]`. `array` is now `["a", "b", 3, "end"]`
 - `array.splice(1, 1)` will give `["b"]`. `array` is now `["a", 3, 4, 5, "end"]`

`.filter` is a powerful method on arrays. It will return a copy of the list, only keeping elements for which the function passed in argument returns `True`. For example:

```
array.filter((element) => (element <= 4 || element === "a"))
```

The code above will return a new list which contains elements from `array` matching the conditions:

- the element is less than 4
- *OR* the element is the letter "a"

Hints: string manipulation

It can be useful to `slice` strings. For example, `"abcdef".slice(3)` will return `def` (all elements starting from index 3). This can be useful to extract ID number from ID strings.

If your task has a `<div id="task-12345">`, then you can:

```
const divId = document.querySelector("div").id
const taskId = divId.slice(5)
```

`taskId` is now a **STRING** that contains the index of the task in the list.