Persistence with Local Storage

We will keep track of the state in our Pomodoro app by storing it locally.

Exercise:

Store the state of the application in local storage. Make sure the application state is reloaded once you refresh the page.

Source code:

Use the PomodoroTracker2 folder as a starting point. The end result is in PomodoroTracker3.

Solution:

Clone PomodoroTracker2 from my GitHub repository as a starting point. Alternatively, you can use your own solution too. We will only modify the JavaScript code, `js/pomodoro.js`.

Local storage is straightforward. There is a `localStorage` variable in the global scope. `localStorage` may contain keys with string values that persist in your browser. This is client-side persistence, so your changes do not carry over to a different browser or computer.

As you can only use strings as values in the local storage, you have to stringify your object or array using `JSON.stringify`.

Let's write a function to save the application state to the local storage:

```
function saveState( tasks ) {
    localStorage.setItem( 'tasks', JSON.stringify( tasks ) );
}
```

Once we retrieve the application state from the local storage, we have to parse it as an array. We will use `JSON.parse`:

```
function loadState() {
    return JSON.parse( localStorage.getItem( 'tasks' ) ) || [];

}
```

If the application state was not saved before, we fall back to an empty array as a default value.

Try out the code a bit. First, add some tasks, finish a few of them, and complete some pomodori. Then save by executing this line in the console:

```
saveState (tasks)
```

Refresh your browser. You should see an empty tasks column. Now load your tasks and render your application:

```
loadState()
renderTasks(pomodoroColumn, tasks)
```

How do we know when to load the state? The answer is surprisingly simple. You load the state when you initialize your `tasks` variable. Replace the `[]` initial value with `loadState()`:

```
let tasks = loadState();
```

Don’t forget to render your application after initialization:

```
let tasks = loadState();
const pomodoroForm = document.querySelector( '.js-add-task' );
const pomodoroColumn = document.querySelector( '.js-task-column-body' );

renderTasks( pomodoroColumn, tasks );
```

Make sure you avoid temporal dead zone issues with the `renderTask` function. If you declared it as

```
const renderTasks = (...) => {...}
```

make it

```
function renderTasks(...) {...}
```

instead.

Our last task is saving the state. When does it make sense to save our state? In theory, we could figure out where we call `renderTasks` and place the saving there.

The problem with this approach is that no-one guarantees that you won't forget saving if there was another occurrence of changing the `tasks` array and rendering it.

Therefore, I would rather bundle this responsibility with `renderTasks` to remind me of persistently saving the state whenever we render:

```
function renderTasks( tBodyNode, tasks = [] ) {
    tBodyNode.innerHTML = // ...
    saveState( tasks );
}
```

If you test the solution, you can see that everything appears correct. Are we done? Hell no! Our solution is very dangerous.

Why doesn't it make sense to place `saveState` inside `renderTasks`? Think about it.

It doesn't make sense because we violate the *single responsibility principle*. We bundle the hidden responsibility of saving the state into the responsibility of rendering tasks. This does not make sense.

Let's change this experience by packaging `renderTasks` and `saveState` inside another function. Without a better idea, I called it `saveAndRenderState`.

```
function saveAndRenderState( tBodyNode, tasks ) {
    renderTasks( tBodyNode, tasks );
    saveState( tasks );
}

function renderTasks( tBodyNode, tasks = [] ) {
    tBodyNode.innerHTML = // ...
}
```

Our last task is to replace the two `renderTask` occurrences with saveAndRenderState:

```
const addTask = function( event ) {
    event.preventDefault();
    const taskName = this.querySelector( '.js-task-name' ).value;
    const pomodoroCount = this.querySelector( '.js-pomodoro-count' ).value;
    this.reset();
    tasks.push( {
        taskName,
        pomodoroDone: 0,
        pomodoroCount,
        finished: false
    } );
    saveAndRenderState( pomodoroColumn, tasks );
}

// ...

const handleTaskButtonClick = function( event ) {
    const classList = event.target.className;
    const taskId = event.target.dataset.id;

    /js-task-done/.test( classList ) ?
        finishTask( tasks, taskId ) :
    /js-increase-pomodoro/.test( classList ) ?
        increasePomodoroDone( tasks, taskId ) :
    /js-delete-task/.test( classList ) ?
        deleteTask( tasks, taskId ) :
    null;

    saveAndRenderState( pomodoroColumn, tasks );
}
```

We are now done with exercise 11.