Pomodoro APP to Kanban Board

Using Kanban to categorize the tasks in our app according to days of the week.

Exercise:

Make a Kanban board from the Pomodoro App with the following fixed columns:

- Done,
- Monday,
- Tuesday,
- Wednesday,
- Thursday,
- Friday,
- Later.

Source code:

Use the PomodoroTracker3 folder as a starting point. The end result is in PomodoroTracker4.

Solution:

Our app is becoming more and more useful after each exercise. These exercises symbolize what kinds of tasks you may get in an interview, but you also get the convenience of building on top of your existing code. If you go through all exercises at once, you may get the experience of solving a homework exercise.

First, let's add a div for the kanban board:

```
<div class="kanban-board  js-kanban-board">
</div>
```

This div will contain our columns.

Microtemplates

The column template needs to be reusable. We could copy-paste it in the html markup 6 times, but we prefer a **DRY** (Don't Repeat Yourself) solution.

Therefore, we will move the column markup to the JavaScript code in the form of a *microtemplate* function:

```
const columnTemplate = ( { header } ) => `
    <div class="task-column">
        <div class="task-column__header">${ header }</div>
        <div class="task-column__body  js-${ header }-column-body"
             data-name="${ header }">
        </div>
    </div>
`;
```

`{ header }` is the shorthand of `{ header: header }` in ES6. This `header` variable is inserted into the template. Notice that we just pass one object to the template, and if we need to add more properties, we just add it to the object. This is convenient because the order of the properties inside the object does not matter. Destructuring takes care of unwrapping these top-level properties.

We also changed the class of the column body to `js-${ header }-column-body`.

Let's test drive the template:

```
document.querySelector( '.js-kanban-board' ).innerHTML =
    columnTemplate( { header: 'task' } );
```

Currently, the existing code only works with the `task` header, because the class of the task column body should match its previous fixed value.

Microtemplates make your code more readable. If you check the `renderTasks` function, it also contains a microtemplate inside a `map` function. There is room for improvement. Let's extract the template out:

```
const cardTemplate = ( { task, id } ) => `
    <div class="task  js-task" data-id="${id}">
        <span class="task__name">${task.taskName}</span>
        <span class="task__pomodori">
            ${task.pomodoroDone} / ${task.pomodoroCount} pomodori
        </span>

        <div class="task__controls">
        ${ task.finished ? 'Finished' : `
            <span class="task-controls__icon  js-task-done"
                  data-id="${id}">\u{2714}</span>
            <span class="task-controls__icon  js-increase-pomodoro"
                  data-id="${id}">\u{2795}</span>`
        }
            <span class="task-controls__icon  js-delete-task"
                  data-id="${id}">\u{1f5d1}</span>
        </div>
    </div>
`;
```

We can call the `cardTemplate` microtemplate function inside `renderTasks`:

```
function renderTasks( tBodyNode, tasks = [] ) {
    tBodyNode.innerHTML = tasks.map( ( task, id ) =>
        cardTemplate( { task, id } )
    ).join( '' );
    saveState( tasks );
}
```

If you reload the application, it is still working.

Kanban data structure

Earlier, there was only one task list. Now we have a list of tasks for seven columns. Beyond the column names, we also need to store the column header name. The initial value of a suggested data structure is as follows:

```
const getEmptyBoard = columnNames => columnNames.map( header => {
    return {
        header,
        tasks: []
    };
} );
```

As our state changed, we will have to temporarily disable loading data from the local storage and generate static data instead:

```
let columns = [
    'Done',
    'Monday',
    'Tuesday',
    'Wednesday',
    'Thursday',

    'Friday',
    'Later'
];

// let tasks = loadState();
let board = getEmptyBoard( columns );
```

Don’t worry about renaming your variables even if your code breaks. Theoretically, you should have covered your code by tests to have a higher level of confidence with changes. This time, we will omit this feature to save space.

Rendering the empty board

The old mechanism of rendering the board is not working anymore, therefore, it’s time to remove it:

```
document.querySelector( '.js-kanban-board' ).innerHTML =
    columnTemplate( { header: 'task' } );
```

Let’s create a function to render the template of the board without tasks:

```
const renderEmptyBoard = board => {
    document.querySelector( '.js-kanban-board' ).innerHTML =
        board.map( columnTemplate ).join( '' );
}
```

Once the value of the board is returned by `getEmptyBoard`, we can render it:

```
renderEmptyBoard( board );
```

We have to slightly modify the CSS to make sure the columns appear next to each other, regardless of the size of the window:

```
.task-column {
    width: 20rem;
    background-color: #ccc;
    border: 1px #333 solid;
    display: inline-block;
    vertical-align:top;
}
```

```
.kanban-board {
    width: calc( 140rem + 40px );
}
```

Yes, as a frontend developer, it is worth knowing some CSS, including the `inline-block` display style and the `calc` function in CSS.

If you open the developer tools, you can see a `TypeError`. This is because event listener administration is not working anymore. We originally tried to listen to events in one column, and now we have seven.

Therefore, we can extend listening to events in the whole pomodoro table. Replace the line

```
const pomodoroColumn = document.querySelector( '.js-task-column-body' );
```

with

```
const kanbanBoard = document.querySelector( '.js-kanban-board' );
```

I also suggest moving up the DOM references to the top of the file and replacing all occurrences of the `.js-kanban-board` query selector with the variable:

```
const kanbanBoard = document.querySelector( '.js-kanban-board' );
const pomodoroForm = document.querySelector( '.js-add-task' );

// ...

kanbanBoard.innerHTML = columnTemplate( { header: 'task' } );
```

Adding tasks

Let’s take care of adding tasks. We will need to specify the column of the task. We will hard code this information in this lesson. Add a dropdown list inside the form with the handle class `js-add-task`:

```
<select name="column-chooser"
            class="js-column-chooser">
        <option value="1">Monday</option>
        <option value="2">Tuesday</option>
        <option value="3">Wednesday</option>
        <option value="4">Thursday</option>
        <option value="5">Friday</option>
        <option value="4">Later</option>
```

```
</select>
```

We can make use of this value in the `addTask` function. We can read the `dayIndex` from the form, and then insert the created task in its proper slot. Notice that `tasks` is now a two-dimensional array inside the `board` global state. There is one more change in the last line: we removed the parameters of `saveAndRenderState`. This is because we already have a handle of the contents of the board and the container node, therefore, using additional arguments would just complicate things in the code.

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

Let’s also update the `saveAndRenderState` function:

```
function saveAndRenderState() {
    renderTable();
    saveState();
}
```

Once again, we removed the parameters, as they are redundant. Everything is accessible globally.

In the `renderTable` function, all we need to do is call the `renderTasks` function for each column in the Kanban board, once an empty table is rendered.

```
function renderTable() {
    renderEmptyBoard( board );
    board.map( column => {
        renderTasks(
            document.querySelector( `.js-${ column.header }-column-body` ),
            column.tasks
        );
    } );
}
```

If you run the application, you can see that now you can add cards to the proper columns.

Event handling

Unfortunately, the buttons on the cards are not working. Let’s correct event handling. First of all, it comes handy if we added a column index in the card markup:

```
const cardTemplate = ( { task, id, columnIndex } ) => `
    <div class="task  js-task"
         data-id="${id}"
         data-column-index="${columnIndex}">
        <span class="task__name">${task.taskName}</span>
        <span class="task__pomodori">
            ${task.pomodoroDone} / ${task.pomodoroCount} pomodori
        </span>
        <div class="task__controls">
        ${ task.finished ? 'Finished' : `
            <span class="task-controls__icon  js-task-done"
                  data-id="${id}"
                  data-column-index="${columnIndex}">\u{2714}</span>
            <span class="task-controls__icon  js-increase-pomodoro"
                  data-id="${id}"
                  data-column-index="${columnIndex}">\u{2795}</span>`
        }
            <span class="task-controls__icon  js-delete-task"
                  data-id="${id}"
                  data-column-index="${columnIndex}">\u{1f5d1}</span>
        </div>
    </div>
`;
```

Remember, the value of the `data-column-index` attribute can be retrieved using the `dataset.columnIndex` property of the DOM node.

We have to make a few changes in the JavaScript code too:

- now we listen to events on the whole Kanban board,
- we have to remove the original first argument of `finishTask`, `increasePomodoroDone`, and `deleteTask`, because instead of a task list, we now have a full board,
- we have to read the `columnIndex` from the card dataset, and pass it to the above three functions,
- in the implementation of `finishTask`, `increasePomodoroDone`, and

`deleteTask`, the lookup of the clicked task changes,
- the parametrization of `saveAndRenderState` should be removed.

```
const finishTask = (taskId, columnIndex) => {
    board[ columnIndex ].tasks[ taskId ].finished = true;
}

const increasePomodoroDone = (taskId, columnIndex) => {
    board[ columnIndex ].tasks[ taskId ].pomodoroDone += 1;
}

const deleteTask = (taskId, columnIndex) => {
    board[ columnIndex ].tasks.splice( taskId, 1 );
}

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

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

    saveAndRenderState();
}

kanbanBoard.addEventListener( 'click', handleTaskButtonClick );
```

If you test the application, you can see that everything is working as expected.

Repairing the local storage and initializing the application

Our final task is to automatically save and load the state. We have to take care of the whole board, not just an array of states:

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

function loadState() {
    return JSON.parse( localStorage.getItem( 'board' ) ) ||
           getEmptyBoard( columns );
}
```

Then replace the line

```
let board = getEmptyBoard( columns );
renderEmptyBoard( board );
```

with

```
let board = loadState();
renderTable();
```

The state has been successfully repaired.

You can see how many things had to be changed to implement the requirements. If you ever get stuck, it makes sense to modularize your application in your mind, and completely rewrite some parts of the application.

You might have noticed that the form adding new tasks is not styled properly. We will take care of this problem in the next exercise.