Pomodoro App Markup and Styling Refactoring

You will now improve the appearance and functionality of the app created in the last lesson.

Exercise:

Refactor the Pomodoro App from the previous exercise such that your tasks will be placed on cards, not table rows. Use the block-element-modifier syntax in your CSS and emphasize separation of concerns. Take care of the styling of the application as well as the functionality.

You have a free choice in your design decisions.

Source code:

Use the PomodoroTracker1 folder as a starting point. The end result is in PomodoroTracker2.

Imagination, life is your creation

This time, you are the freelancer, and you are supposed to drive all design decisions. If you don't know what you are doing, check out tools that have solved the same problem. To save your time, I suggest searching for screenshots of Trello and KanbanFlow boards. Don't worry; you don't have to implement a full board yet. We are just focusing on one column.

Let's start with creating the markup for our column:

```
<div class="task-column">
    <div class="task-column__header">Tasks</div>
    <div class="task-column__body  js-task-column">
        <div class="task  js-task" data-id="0">
            <span class="task__name">Write Article</span>
            <span class="task__pomodori">0 / 2 pomodori</span>
            <div class="task__controls">
                <span class="task-controls__icon  js-task-done">
                    &#x2714;
                </span>
                <span class="task-controls__icon  js-increase-pomodoro">
                    &#x2795;
                </span>
                <span class="task-controls__icon  js-delete-task">
                    &#x1f5d1;
                </span>
            </div>
        </div>
    </div>
</div>
```

Notice the block-element-modifier syntax. A container is connected to an element via `__`. We can only use one `__` in a class name, so instead of `task__controls__done`, we just used `task-controls__done`. We could attach modifiers to these classes with `--`. Block-element-modifier classes should not be referenced in our JavaScript code. They are for styling purposes only. This is how we achieve separation of concerns.

Let’s style the elements.

```
.task-column {
    width: 20rem;
    background-color: #ccc;
    border: 1px #333 solid;
}

.task-column__header {
    width: 14rem;
    margin: 1rem 1rem 0 1rem;
    padding: 2rem;
    background-color: #777;
    color: #eee;
    text-align: center;
    font-size: 1.5rem;
}

.task-column__body {
    width: 16rem;
    margin: 0 1rem 1rem 1rem;
    padding: 0.8rem 1rem;
    background-color: #999;
    min-height: 2rem;
}

.task {
    width: 12rem;
    height: 3rem;
    margin: 0.5rem 1rem;
    background-color: #eee;
    padding: 1rem;
    user-select: none;
}

.task__name {
    display: block;
}

.task__pomodori {
    display: inline-block;
    float: left;
}

.task__controls {
    display: inline-block;
    float: right;
}

.task-controls__icon {
    cursor: pointer;
}
```

I will not get into the details of explaining each rule. Understanding CSS is a lot easier than JavaScript. You can reverse engineer each rule with some googling. Some minimal styling knowledge always comes in handy.

Let's connect the markup with the JavaScript code. First of all, let's delete the static tasks from the markup:

```
<div class="task-column">
   <div class="task-column__header">Tasks</div>
   <div class="task-column__body  js-task-column">
   </div>
</div>
```

In the JavaScript code belonging to the previous example, let's replace `js-task-table-body` with `js-task-column` on line 3. Let's also replace the variable name `pomodoroTableBody` with `pomodoroColumn`.

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

Don't forget to replace all occurrences of `pomodoroTableBody` with `pomodoroColumn` in the code.
We have one task left: let's rewrite the `renderTasks` function to generate the new markup structure:

```
const renderTasks = function( tBodyNode, tasks = [] ) {
    tBodyNode.innerHTML = tasks.map( ( task, id ) => `
        <div class="task  js-task" data-id="0">
            <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>
    ` ).join( '' );
}
```

We are done.

There was not much JavaScript in this task. I included it, because it is important to emphasize that in frontend and full stack development, you need to be competent in refactoring markup and add some basic CSS. On some occasions, you also have to showcase your creativity and grit by taking the initiative and designing the interface for your applications.