Countdown Timer Integration in the Pomodoro App

The final step of the Pomodoro App: Add a timer to the selected task.

Exercise:

Create a form displaying a timer counting down from `25` minutes in the format of `mm:ss`. Display the name of the task above the timer. Once the timer reaches `00:00`, revert to `25:00`, increase the pomodoro count by `1`, and continue counting down.

Place three links below the timer: `Start`, `Pause`, and `Reset`.

When a card is selected, `25:00` appears on the timer, and the timer stops. You can start the timer by clicking the `Start` link. When the `Pause` link is clicked, the timer is suspended, keeping its current value. `Reset` moves the timer back to `25:00`.

Source code: Use the PomodoroTracker8 folder as a starting point. The result is in PomodoroTracker9.

Solution: We are in the finish line with the Pomodoro App. We only need a timer.

First, let's integrate a static timer in the app that stays visible even if we scroll. This is an HTML/CSS task. Add this markup to the `index.html` file:

```
<div class="js-timer  timer">
    <div class="js-selected-task-label  timer__task">
        Select a Task
    </div>
    <div class="js-time-remaining  timer__time">25:00</div>
    <div class="js-links  timer__links">
        <span class="js-start-timer  timer__control">
            &#x25B6;
        </span>
        <span class="js-pause-timer  timer__control">
            &#x23F8;
        </span>
        <span class="js-stop-timer  timer__control">
```

```
            &#x23F9;
        </span>
    </div>
</div>
```

We can position the div on the bottom-left part of our viewport and make it stick by applying `position:fixed` in the stylesheet and adding the corresponding coordinates. Add these rules to the `styles.css` file:

```
.timer {
    position: fixed;
    bottom: 0px;
    height: 150px;
    width: 300px;
    background-color: #ddd;
    color: #333;
    font-size: 2rem;
}
```

Also add styles for the play, pause, and stop buttons:

```
.timer__control:hover {
    cursor: pointer;
    color: red;
}
```

We are keeping the styles minimalistic for now.

Let's integrate the clock. We will instantiate the `Timer` class and set its default to `25` minutes. We have to pass the timer a callback function that updates the countdown in the DOM. Due to the separation of concerns, we have to use the corresponding `js-` prefixed class as a handle to get the DOM element containing the remaining time.

```
const updateTime = newTime => {
    document.querySelector( '.js-time-remaining' ).innerHTML = newTime;
}
const timer = new Timer( 25 * 60, updateTime );
```

To play around with the clock, let's define some event handlers for the buttons:

```
document.querySelector( '.js-start-timer' )
        .addEventListener( 'click', () => timer.start() );
document.querySelector( '.js-pause-timer' )
        .addEventListener( 'click', () => timer.pause() );
```

```
document.querySelector( '.js-stop-timer' )
        .addEventListener( 'click', () => timer.reset() );
```

If you test the application, you can see that the buttons work as expected.

Our next task is to handle the card selection. Once we select or deselect a card, we do the following:

- Stop the timer if it is running,
- Update the name of the task above the timer or display `Select a Task` in case no cards are selected.

We will perform these updates in two places: during initialization and after selecting a task.

First, we will add a `setupTimer` call after the instantiation of the timer and the corresponding event handlers.

```
const updateTime = newTime => {
    console.log( newTime );
    document.querySelector( '.js-time-remaining' ).innerHTML = newTime;
}
const timer = new Timer( 25 * 60, updateTime );
document.querySelector( '.js-start-timer' )
        .addEventListener( 'click', () => timer.start() );
document.querySelector( '.js-pause-timer' )
        .addEventListener( 'click', () => timer.pause() );
document.querySelector( '.js-stop-timer' )
        .addEventListener( 'click', () => timer.reset() );
setupTimer();
```

Second, we will also place a `setupTimer` call in the `selectTask` function.

```
const selectTask = (taskId, columnIndex, target) => {
    deselectAllTasks();
    board[ columnIndex ].tasks[ taskId ].selected = true;
    setupTimer();
}
```

The implementation of `setupTimer` depends on a small function that returns the name of the selected task. Although we could have gotten this information more easily by using `taskId` and `columnIndex` in the `selectTask` function, it just adds some code complexity for no reason. Note that we are wasting the resources of the client, and we are talking about microseconds if not nanoseconds. Any code optimization here counts as productive

procrastination or perfectionism. You gain a lot more by keeping the solution simple.

```
const getSelectedTaskName = () => {
    for ( let { tasks } of board ) {
        for ( let i = 0; i < tasks.length; ++i ) {
            if ( tasks[i].selected ) {
                return tasks[i].taskName;
            }
        }
    }

    return '';
}

const setupTimer = () => {
    timer.reset();
    document.querySelector( '.js-selected-task-label' )
            .innerHTML = getSelectedTaskName();
}
```

Regardless of whether the clock is running, we can change tasks. Once we change to a new task, the clock is reset to `25:00`.

Let's now update the pomodoro count of the selected task once the countdown reaches zero. This update is easy because we already have an `updateTime` function, where we get `0:00` in the `newTime` argument once the pomodoro time is over.

```
var updateTime = newTime => {
    console.log( newTime );
    document.querySelector( '.js-time-remaining' ).innerHTML = newTime;
}
```

After getting rid of the console log, we can check if `newTime` is `0:00`, reset the timer to 25 minutes, restart the timer, and add one to the pomodoro count of the task. Once the pomodoro count increases, don't forget to save and render the state.

```
var updateTime = newTime => {
    document.querySelector( '.js-time-remaining' ).innerHTML = newTime;
    if ( newTime == '0:00' ) {
        timer.reset();
        timer.start();
        increaseSelectedTaskPomodoroDone();
        saveAndRenderState();
    }
}

function increaseSelectedTaskPomodoroDone() {
```

```
    // call increasePomodoroDone with correct arguments
}
```

To test this code, it makes sense to temporarily set the pomodoro length to a significantly smaller value than 25 minutes. Don't forget to reset it once you finish testing.

```
const timer = new Timer( /* 25 * 60*/ 5, updateTime );
```

We only have one problem to solve: the original form of `increasePomodoroDone` has two arguments:

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

If we execute this function without arguments, we will get an error. But wait a minute! We can use this problem to make the code more structured. Recall the function `getSelectedTaskName`:

```
function getSelectedTaskName() {
    for ( let { tasks } of board ) {
        for ( let i = 0; i < tasks.length; ++i ) {
            if ( tasks[i].selected ) {
                return tasks[i].taskName;
            }
        }
    }

    return '';
}
```

Beyond the task name, we could also be interested in the task id or the column number. So we could simply return a data structure that gives us access to all these data.

```
function getSelectedTaskInfo() {
    for ( let columnIndex = 0;
          columnIndex < board.length;
          ++columnIndex ) {
        let { tasks } = board[ columnIndex ];
        for ( let taskId = 0; taskId < tasks.length; ++taskId ) {
            if ( tasks[taskId].selected ) {
                return {
                    columnIndex,
                    taskName: tasks[taskId].taskName,
                    taskId
```

```
                    taskId
                };
            }
        }
    }

    return null;
}

function getSelectedTaskName() {
    const info = getSelectedTaskInfo();
    if ( info == null ) return '';
    return info.taskName;
}
```

As we implemented the `getSelectedTaskName` function using `getSelectedTaskInfo`, nothing changes in the interface.

The `getSelectedTaskInfo` function is ready, so it is time to implement the function increasing the pomodoro count of the selected task:

```
function increaseSelectedTaskPomodoroDone() {
    let { taskId, columnIndex } = getSelectedTaskInfo() || {};
    if ( typeof taskId === 'number' && typeof columnIndex === 'number' ) {
        increasePomodoroDone( taskId, columnIndex );
    }
}
```

After putting the last puzzle piece in place, we are done with this exercise.

You can see the code in action if you clone my GitHub repository. The code is in the PomodoroTracker9 folder.

In these nine steps, you have seen how to build a complex exercise from scratch.

The code is simple enough to maintain, but I admit, if I had more plans with it, I would soon run out of mental capacity to track what is going on. This is why it makes sense to improve the code structure by adding a framework around the code or abstracting some functionalities. It is out of scope for us in this book though.