Event Delegation in a Pomodoro App

This is your first extensive exam task. The concepts in the following lessons will come in handy during interviews. It's a long exercise, but trust me, it's worth the hassle.

We will now build a simple Pomodoro App. If you don’t know what the Pomodoro technique is, you can read about it here.

Exercise

Create a client-side application that displays a table of tasks with the following columns:

- Task name (string);
- Status: number of pomodori done, a slash, the number of pomodori planned, then a space, then the word `pomodori`;
- Controls: contains three buttons for each row, `Done`, `Increase Pomodoro Count`, and `Delete`.

When pressing the `Done` button, the `Done` and the `Increase Pomodoro Count` buttons are replaced by the static text `Finished`.

When pressing `Increase Pomodoro Count`, the number of pomodori done is increased by `1` in the Status column. The initial value of the number of pomodori done is zero.

When pressing `Delete`, the corresponding row is removed from the table.

Create a form that allows you to add a new task. The task name can be any string, and the number of pomodori planned can be an integer between `1` and `4`.

Unblock Yourself

This task may be an exercise that you can either solve during an interview, or as a homework exercise.

Always respect the requirements, and never invent anything on your own. It is fine to clarify questions on the spot if you have time to ask them. If you solve this task as a homework assignment, it is also okay to send your questions to your contacts via email.

Pointing out the flaws in the specification is a great asset to have as long as you demonstrate that you can cooperate with your interviewers.

My usual way of cooperation is that I ask my questions along with the first submitted version of my task. This means I implement everything I can, without getting blocked, and then I enumerate my assumptions and improvement suggestions attached to the first version.

This sends the message that you are aware of a flaw in the specification, and you are willing to fix it, in case it is needed. You also signal that you were proactive in implementing everything you could, based on the information available to you.

Most of the time your interviewers will accept your solution. Sometimes they may ask you to go ahead and make changes to your application based on their feedback.

Remember, don't block yourself just because the task is underspecified. You can implement only what's needed, and tackle the improvement suggestions later.

What is not specified?

In this example, there are quite a few unusual elements.

First, the task name may be an empty string. There is no validation specified in the task description. You may point this out as an improvement suggestion. Implementing validation on your own would mean that you don't respect the specification.

Second, different rows may contain the same task.

Third, there is no way to undo pressing the `Done` button. When a task is finished, it will stay finished.

Fourth, the Pomodoro counter may increase above the planned number of pomodori.

Fifth, `pomodoro` is singular, `pomodori` is plural, but we always display `pomodori` in the status column.

You can point all these anomalies out as improvement suggestions. Improvising and implementing these features without asking for permission would imply that you don't respect the specification. Some hiring crews will not care about it, while others may even reward you for improvising. However, chances are, if you continuously improvise, your interviewers will ask themselves the question if they can cooperate with you smoothly.

This is why I suggest putting all your improvement suggestions in the documentation attached to your solution. You even save yourself time.

Solution

Let's start with the markup:

```
<!doctype html>
<html>
    <head>
        <title>Pomodoro Timer - zsoltnagy.eu</title>
        <link rel="stylesheet"
              href="node_modules/normalize.css/normalize.css">
        <link rel="stylesheet" href="styles/styles.css">
    </head>
    <body>
        <table>
            <thead>
                <tr>
                    <th>Task name</th>
                    <th>Status (done / planned)</th>
                    <th>Controls</th>
                </tr>
            </thead>
            <tbody class="js-task-table-body">
            </tbody>
        </table>

        <form class="js-add-task"
              action="javascript:void(0)">
            <input type="text"
                   name="task-name"
                   class="js-task-name"
                   placeholder="Task Name" />
            <select name="pomodoro-count"
                    class="js-pomodoro-count">
                <option value="1">1</option>
                <option value="2">2</option>
                <option value="3">3</option>
                <option value="4">4</option>
```

```
            </select>
            <input type="submit" />
        </form>
    </body>
    <script src="js/pomodoro.js"></script>
</html>
```

Note the following DOM nodes:

- `js-task-table-body`: this is where we will insert the tasks one by one as table rows
- `js-add-task`: the form to add new tasks. We will listen to the submit event of this form
- `js-task-name`: a text field containing the task name inside the `js-add-task` form
- `js-pomodoro-count`: a dropdown list containing the number of planned pomodori inside the `js-add-task` form. Usually, more than four pomodori would mean that we haven't broken down the task well enough, this is why we only allow up to four pomodori as a limit.

Let's write some JavaScript in the `js/pomodoro.js` file.

```
let tasks = [];
const pomodoroForm = document.querySelector( '.js-add-task' );
const pomodoroTableBody =
    document.querySelector( '.js-task-table-body' );
```

First, notice we need an array of tasks to store the contents of the table. We also need a reference to the pomodoro form and the table body.

We plan to handle the form submission with an event handler.

```
const addTask = function( event ) {
    event.preventDefault();
    // ...
    this.reset();
    // ...
}
pomodoroForm.addEventListener( 'submit', addTask );
```

Notice the `preventDefault` call. When we submit a form, a redirection is made. The default action defined in HTML forms is that a server handles our submitted form data, and renders new markup for us. In a client side

application, we rarely need this default action from the end of the server.

Therefore, we can prevent this default action by calling the `preventDefault` method of the submit event.

Technically, this is not mandatory, because the action of the `form` is `javascript:void(0)`, which does not make any redirections. I showed you this option in the markup, but I still recommend using `preventDefault` from JavaScript's end to avoid the consequences someone accidentally removing `javascript:void(0)` from the markup.

The context inside the event handler is the form element itself. Calling the `reset` method of the form resets all form fields to their default values. We can safely reset the form once we are done processing the values.

Let's do this processing now:

```
const addTask = function( event ) {

    // 1. Prevent default action
    event.preventDefault();

    // 2. Extract form field values
    const taskName = this.querySelector( '.js-task-name' ).value;
    const pomodoroCount =
        this.querySelector( '.js-pomodoro-count' ).value;

    // 3. Create a new task item by updating the global state
    tasks.push( {
        taskName,
        pomodoroDone: 0,
        pomodoroCount,
        finished: false
    } );

    // 4. Reset the form
    this.reset();

    // 5. Render the global state
    renderTasks( pomodoroTableBody, tasks );
}
```

We have already covered steps 1 and 4.

Step 2 is about extracting the values the user entered. Notice the `this.querySelector` construct. Remember? The value of `this` is the DOM node of the form. Therefore, we can use the `querySelector` method of this DOM node to search for the corresponding form fields and take their `value`

attribute.

In Step 3, we create a new object. Notice the object shorthand notation. Remember, in ES6, `{ x }` is equivalent to `{ x: x }`.

I decided on implementing rendering in a separate function, because this feature will likely be needed later once we update the form. Let's finish Step 5 by implementing the `renderTasks` function. I will use the ES6 Template Literal format.

We can conveniently include newline characters in the template without terminating it. We can also evaluate JavaScript expressions in the form `${expression}`:

```
const renderTasks = function( tBodyNode, tasks = [] ) {
    tBodyNode.innerHTML = tasks.map( ( task, id ) => `
        Template for task[${id}] with name ${task.taskName}
    ` ).join( '' );
}
```

We will set the `innerHTML` property of `tBodyNode` to a text node containing the string that we assemble.

The assembly is made using the `map` method of the `tasks` array. A map is a *higher order function*, because it expects a function as an argument. This function is executed on each element of the `tasks` array one by one, transforming `tasks[id]` also accessible as `task` onto string return values. The template is assembled by joining these string values.

If you are not familiar with `map`, the code is almost the same as the below `for` loop equivalent. The only difference is the whitespacing inside the template literal.

```
const renderTasks = function( tBodyNode, tasks = [] ) {
    let template;
    for ( let i = 0; i < tasks.length; ++i ) {
        template += `task[${i}] with name ${tasks[i].taskName}`;
    }
    tBodyNode.innerHTML = template;
}
```

As a loose tangent, technically, we don't need the `template` variable, because we could simply append each template row to `tBodyNode.innerHTML`. Right?

We could simply append each template row to `tBodyNode.innerHTML`. Right?

Well, right and wrong. Technically, you could do this, and your code would look shorter. In practice, always bear in mind that DOM operations are more expensive than JavaScript operations. So much so, that once I managed to dig inside the jQuery UI Autocomplete code to shove off more than 95% of the execution time of opening the autocomplete by assembling the `$node.innerHTML +=` type of DOM manipulations in memory, and making just one DOM insertion at the end.

Back to business. Let's assemble our task table row:

```
const renderTasks = function( tBodyNode, tasks = [] ) {
    tBodyNode.innerHTML = tasks.map( ( task, id ) => `
        <tr>
            <td class="cell-task-name">${task.taskName}</td>
            <td class="cell-pom-count">
                ${task.pomodoroDone} / ${task.pomodoroCount} pomodori
            </td>
            <td class="cell-pom-controls">
            ${ task.finished ? 'Finished' : `
                <button class="js-task-done"
                        data-id="${id}">
                    Done
                </button>
                <button class="js-increase-pomodoro"
                        data-id="${id}">
                    Increase Pomodoro Count
                </button>`
            }
                <button class="js-delete-task"
                        data-id="${id}">
                    Delete Task
                </button>
            </td>
        </tr>
    ` ).join( '' );
}
```

The `td` classes are there for styling. I won't bother you with the details; you can check out the CSS code on my GitHub repository.

The `button` classes are there for event handling. After all, we will have to handle the button clicks later. To make our life easier, we can also add the `id` data attribute to the button.

The ternary `? :` operator makes sure that we either display the `Finished` text, or the two buttons described in the specification.

The Twist

So far, the code is straightforward. I mean, you have to know what you are doing to come up with a solution like this. You also have to know the ins and outs of writing basic HTML markup and basic JavaScript.

Progress may even give you a false illusion. You might have perceived that the main adversity is the creation and rendering of the table. Now that you are done, you might think, the rest of the task is a piece of cake.

This is when a surprise knocks you off. In most well-written stories, the hero's journey contains a twist after the hero defeats the enemy. This is when the hero realizes that things are a lot more difficult than he anticipated.

This is also the point when the hero needs to reverse engineer a prophecy to move forward. It is now time to reveal the prophecy: "One Event Handler to Rule Them All".

One Event Handler?!

I bet you were about to consider how you would implement one event handler for each button in the DOM. It is definitely feasible. Once you render the markup, you have to take care of dynamically adding the corresponding event listeners. You have to make sure you don't mess up event handling.

This seems to be a lot of unnecessary work. The stubborn hero could implement it like this:

```
const finishTask = ( e ) => {
    const taskId = e.target.dataset.id;
    tasks[ taskId ].finished = true;
    renderTasks( pomodoroTableBody, tasks );
}

const increasePomodoroDone = ( e ) => {
    const taskId = e.target.dataset.id;
    tasks[ taskId ].pomodoroDone += 1;
    renderTasks( pomodoroTableBody, tasks );
}

const deleteTask = ( e ) => {
    const taskId = e.target.dataset.id;
    tasks.splice( taskId, 1 );
    renderTasks( pomodoroTableBody, tasks );
}

const addTaskEventListeners = () => {
```

```
    const prefix = '.js-task-table-body .js-';
    document.querySelectorAll( prefix + 'increase-pomodoro' )
        .forEach( button =>

            button.addEventListener( 'click', increasePomodoroDone )
        );
    document.querySelectorAll( prefix + 'task-done' )
        .forEach( button =>
            button.addEventListener( 'click', finishTask )
        );
    document.querySelectorAll( prefix + 'delete-task' )
        .forEach( button =>
            button.addEventListener( 'click', deleteTask )
        );
}

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

Just imagine! If you store 100 tasks in your list, you add 300 event listeners each time you make one tiny modification to the table. The code is also very WET (We Enjoy Typing). After all, the first and the third row of the functions `increasePomodoroDone`, `finishTask`, and `deleteTask` are all the same. We also have to do three tedious `forEach` helpers and copy-paste the structure to add the event listeners, let alone adding the event listeners after each render. Who guarantees that we can't manipulate the DOM without calling the `renderTasks` function?

This is a bit too much. The structure is not clean enough, and it requires too much maintenance. Therefore, it is now time to consider our prophecy and start thinking.

Event delegation

One event handler to rule them all. When a click on a DOM node happens, and an event handler is not defined on the node, the event handler defined on the closest parent node captures and handles the event.

To handle event propagation without the need for adding event listeners during runtime, we have to find an ancestor node that contains the buttons. This node is the table body node `.js-task-table-body`. Let's define our event listener there:

```
const handleTaskButtonClick = function( event ) {
```

```
    const classList = event.target.className;
    const taskId = event.target.dataset.id;

    // increase pomodoro count or finish task or delete task

    renderTasks( pomodoroTableBody, tasks );
}

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

The click handler will stay in place throughout the whole lifecycle of the application.

The `event` itself belongs to the button we clicked. Therefore, we can easily extract the class attribute and the `data-id` attribute. `className` contains all classes added to the node.

After performing the requested action, we have to re-render the table.

Let's see how to perform the actions. We have to determine if `classList` contains a class we are looking for. We can simply do it with a regex matching. If you are interested in more details on regexes, check out my post on JavaScript regular expressions.

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

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

const deleteTask = ( tasks, taskId ) => {
    tasks.splice( taskId, 1 );
}

const handleTaskButtonClick = function( event ) {
    const classList = event.target.className;
    const taskId = event.target.dataset.id;
    switch ( true ) {
        case /js-task-done/.test( classList ):
            finishTask( tasks, taskId );
            break;
        case /js-increase-pomodoro/.test( classList ):
            increasePomodoroDone( tasks, taskId );
            break;
        case /js-delete-task/.test( classList ):
            deleteTask( tasks, taskId );
            break;
    }
    renderTasks( pomodoroTableBody, tasks );
}
```

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

Note we could have used `event.target.matches('.js-task-done')` instead of `/js-task-done/.test(classList)`. Both solutions are the same.

Notice that `finishTask`, `increasePomodoroDone`, and `deleteTask` are now DRY compared to their previous version.

The `renderTasks` function was also reverted to its original version because we don't have to add any event listeners after rendering.

Check out the source code on GitHub.