Dragging and Dropping Cards

Shift cards around by dragging and dropping.

Exercise:

Make it possible to drag and drop cards.

Source code: Use the PomodoroTracker6 folder as a starting point. The result is in PomodoroTracker7.

Solution: There is a `draggable` attribute in HTML5. When using this attribute, you will be able to automatically move `draggable` elements by holding down the mouse button. Let's make our selected card draggable:

```
const cardTemplate = ( { task, id, columnIndex } ) => `
    <div class="task  js-task ${ task.selected ? 'selected' : '' }"
         data-id="${id}"
         draggable="${ task.selected ? "true" : "false" }"
         data-column-index="${columnIndex}">
...
    </div>
`;
```

So far we have only added one attribute, but as soon as you start dragging the selected card, you can see that a higher opacity version of the selected card follows the mouse cursor as long as you hold the mouse button down.

The question arises, how can you find out about the `draggable` html5 attribute during an interview? The answer is simple: Google is your friend. Remember, in most interview situations you will be able to use Google. Also remember, that with the series of exercises on the Pomodoro App, we are simulating a homework assignment, where you have all the time in the world to research tricks worth using.

Back to the exercise. Our next task is to place the card to the proper column

upon dropping it.

There is only one twist when it comes to dropping an element. The `drop` event only fires if we cancel the `dragover` and `dragenter` events. For more details, check this StackOverflow post. Therefore, when adding a drop listener, make sure you prevent the default action on the corresponding `dragenter` and `dragover` events.

As we re-render the table quite often, but we never re-render the table while we are dragging elements, it makes sense to attach the event listeners in the `renderEmptyBoard` function:

```
const renderEmptyBoard = board => {
    kanbanBoard.innerHTML =
        board.map( columnTemplate ).join( '' );
    addColumnDropListeners();
}

function dropCard( e ) {
    console.log( 'drop', e );
}

function addColumnDropListeners() {
    const columns = document.querySelectorAll( '.task-column' );
    for ( let column of columns ) {
        column.addEventListener( 'dragover', e => {
            e.preventDefault();
        });
        column.addEventListener( 'dragenter', e => {
            e.preventDefault();
        });
        column.addEventListener( 'drop', dropCard );
    }
}
```

Once you start to drag and drop the selected card of the Kanban board, you can see that the drop event is fired. If you dig a bit deeper into the event object, you can find a `path` array, containing the hierarchy of fields the card was dropped onto. For instance, after dropping the card to the first column, you can see the following hierarchy in the array:

```
Array(7)
0: div.task-column__body.js-Done-column-body
1: div.task-column
```

```
2: div.kanban-board.js-kanban-board
3: body
4: HTML
5: document
6: Window {postMessage: ƒ, blur: ƒ, focus: ƒ, close: ƒ, frames: Window, …}
```

Our task is to find the DOM node that has a `.js-XXX-column-body` class, where `XXX` is one of the seven column names.

Alternatively, we could also use the `srcElement` property, which contains the DOM node where we dropped the card. However, this source element can be the column, the column header, a card, or even a label on the card. This method is not reliable to identify where we are, because we cannot always place a label on each `div` or `span` we create.

Therefore, we will stick to the `path` array. Our rule is that an element in the `path` has to be the column body. If we drop the card outside the column body, we ignore it.

```
function getDroppedColumnName( e ) {
    for ( let node of e.path ) {
        // extraction of the column name comes here
    }
}
```

We can get the class list string from a DOM node using `node.classList.value`. In our hierarchy, `document` and the global `Window` object do not have a `class list`, therefore we have to test if the `classList` property exists at all.

```
function getDroppedColumnName( e ) {
    for ( let node of e.path ) {
        if ( typeof node.classList === 'undefined' ) break;
        console.log( node.classList.value );
        // extraction of the column name comes here
    }
}
```

Now we can process the `classList` string. Let's see an example string:

```
task-column__body  js-Done-column-body
```

We need to extract `Done` encapsulated by a `js-` prefix, and a `-column-body` suffix. We cannot rely on the order of the classes, so other classes may or may

not precede `js-Done-column-body`. We will describe the following strategy using a regular expression:

- first, we capture as many characters as possible in a greedy way
- then we capture the `js-` string
- then we do a match of a sequence of at least one arbitrary character. We capture this sequence in a capture group, using parentheses
- then we match `-column-body`

The regular expression is `/.*js-(.+)-column-body/`. Notice `(.+)`. The parentheses make sure the value of `.+` is recorded and returned by the `match` method of the regular expression.

The result of the match is either `null`, or a data structure containing the full match at index `0`, and the captured value of `.+` in the expression at index `1`. Therefore, if `match` is not null, `match[1]` becomes the column name we are looking for:

```
function getDroppedColumnName( e ) {
    for ( let node of e.path ) {
        if ( typeof node.classList === 'undefined' ) break;
        const match =
            node.classList.value.match( /.*js-(.+)-column-body/ );
        if ( match && typeof match[1] === 'string' ) {
            console.log( 'columnName', match[1] );
        }
    }
}
```

You don't have to know regular expressions to solve this exercise. You could also use the JavaScript `split` method. However, regular expressions come in handy in your career; I highly recommend learning them. If you need help, check out my book and video course on JavaScript regular expressions.

Let's now implement the `dropCard` event handler.

```
function dropCard( e ) {
    const newColumn = getDroppedColumnName( e );
    if ( typeof newColumn === 'string' ) {
        moveSelectedCard( newColumn );
    }
}
```

If we get the column name, we can move the selected card to the new column. The only puzzle piece in this task is the implementation of the `moveSelectedCard` function. We have the following tasks to implement:

- Find the selected card.
- If the selected card is in the same column as `toColumn`, we do nothing.
- Otherwise, we remove the selected card from the `tasks` array of the column and add it to the task list of the column described by the `toColumn` header
- if there is a change in state, we have to re-render the application and save the state to the local storage

The implementation looks as follows:

```
function moveSelectedCard( toColumn ) {
    for ( let { header, tasks } of board ) {
        if ( header === toColumn ) continue;
        for ( let i = 0; i < tasks.length; ++i ) {
            if ( tasks[i].selected ) {
                const selectedTask = tasks[i];
                tasks.splice( i, 1 );
                let columnIndex = columns.indexOf( toColumn );
                board[ columnIndex ].tasks.push( selectedTask );
                saveAndRenderState();
            }
        }
    }
}
```

The only surprising element may be the destructuring inside the `for..of` operator:

```
for ( let { header, tasks } of board ) {

}
```

The code snippet above is the same as

```
for ( let column of board ) {
    { header: header, tasks: tasks } = column;
}
```

which is in turn equivalent to

```
for ( let column of board ) {
    let header = column.header;
    let tasks = column.tasks;
}
```

If you execute the application, you can enjoy the brand new experience of drag and dropping the selected card to any column you want.