Exercise on JavaScript components

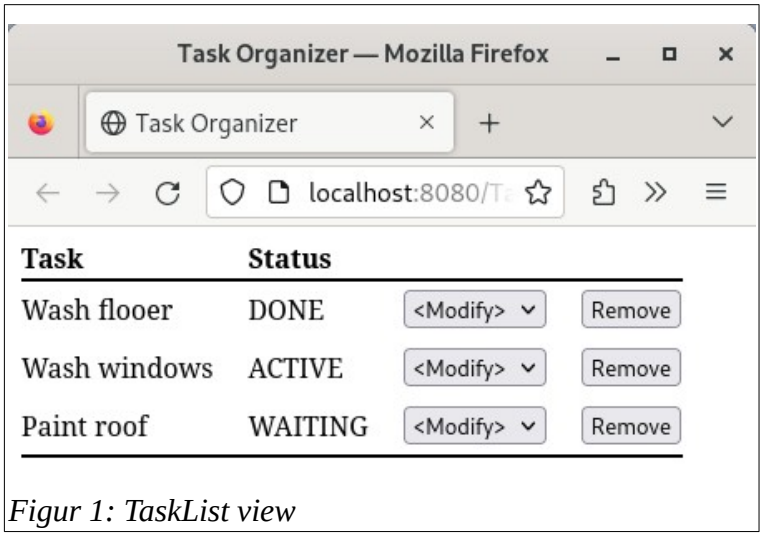
In this exercise you will create a JavaScript component **TaskList**. There is no delivery of this assignment, but the component **TaskList** will be used in the later mandatory JavaScript assignment.

You will find an Eclipse archive on Canvas, *apitemplate.zip* with a template for the **TaskList** component. The archive outlines all the public methods that you must implement.

You should not need to create any additional files to solve this task. You solve this task by filling in your code into to the file *components/tasklist/tasklist.js*.

The task of this exercise is to create the **TaskList** component according to the specifications below. You must create **TaskList** as a JavaScript module.

The **TaskList** component is used to manage tasks, and a possible view is shown in the illustration below.



Figur 1: TaskList view

The HTML **BODY** of the above view is:

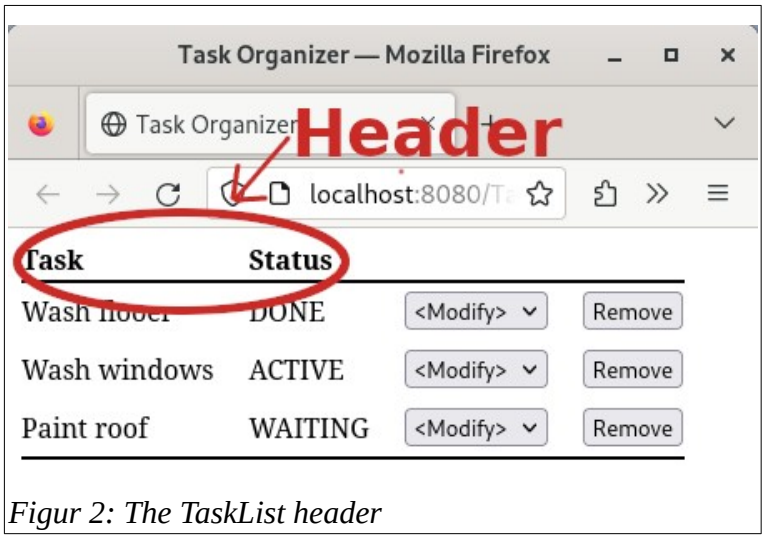
```
<BODY>
    <!-- The task list -->
    <TASK-LIST></TASK-LIST>
</BODY>
```

The **TASK-LIST** tag should create an instance of a JavaScript class **TaskList** in the web document. The view above is produced by the following JavaScript code:

```
const tasklist = document.querySelector("task-list");

const allstatuses = ["WAITING", "ACTIVE", "DONE"];
tasklist.setStatuseslist(allstatuses);

const tasks = [
    {
        id: 1,
```

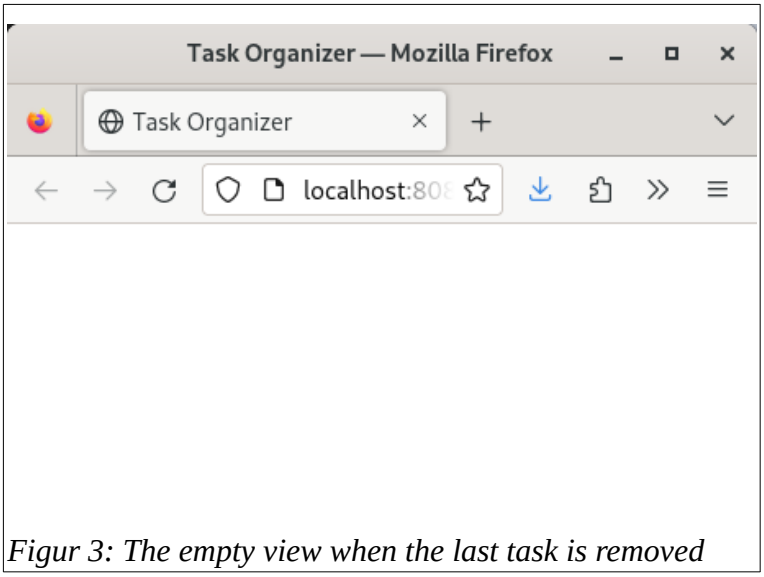
```
        status: "WAITING",
        title: "Paint roof"
    },
    {
        id: 2,
        status: "ACTIVE",
        title: "Wash windows"
    },
    {
        id: 3,
        status: "DONE",
        title: "Wash flooer"
    }
];

for (let t of tasks) {
    tasklist.showTask(t);
}
```

The **TaskList** view should include a header, as outlined below.



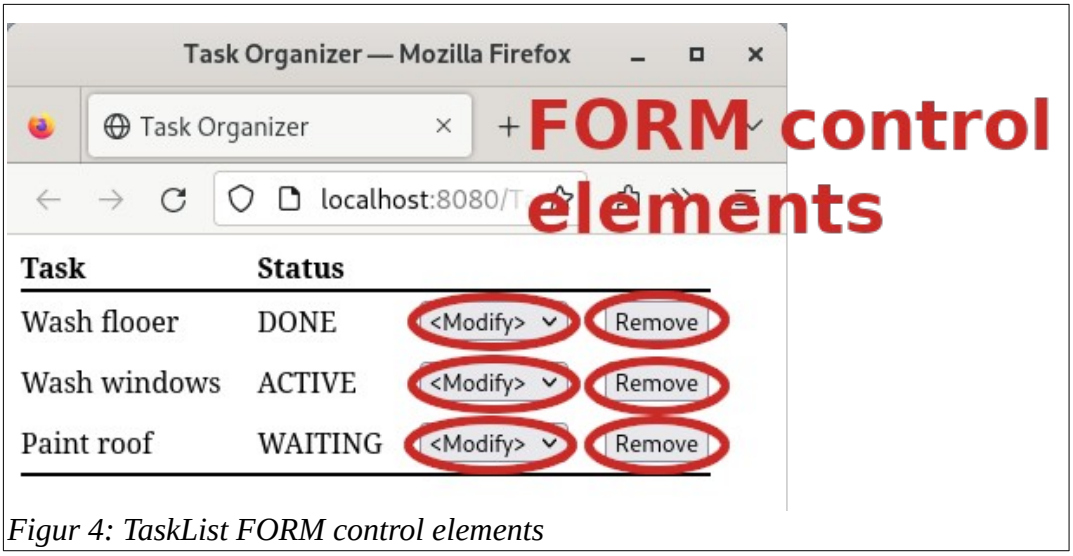
Figur 2: The TaskList header

The header should only be displayed if there are tasks in the list. When the last task is removed from the list, the header should be removed together with the last task. The illustration below shows the empty view if no tasks.



Figur 3: The empty view when the last task is removed

The tasks will be stored by a back end system that will be addressed in the mandatory assignment. The FORM control elements of the **TaskList** view do not by themselves modify the view. The view should be modified only after the task is updated on the back end system. The FORM control elements only informs the controlling code about the action by running callbacks supplied by the controlling code.



Figur 4: TaskList FORM control elements

The Tasklist API

The **TaskList** class has methods that are intended for the controlling code to set up the view and manage the tasks of the view.

Method *showTask*

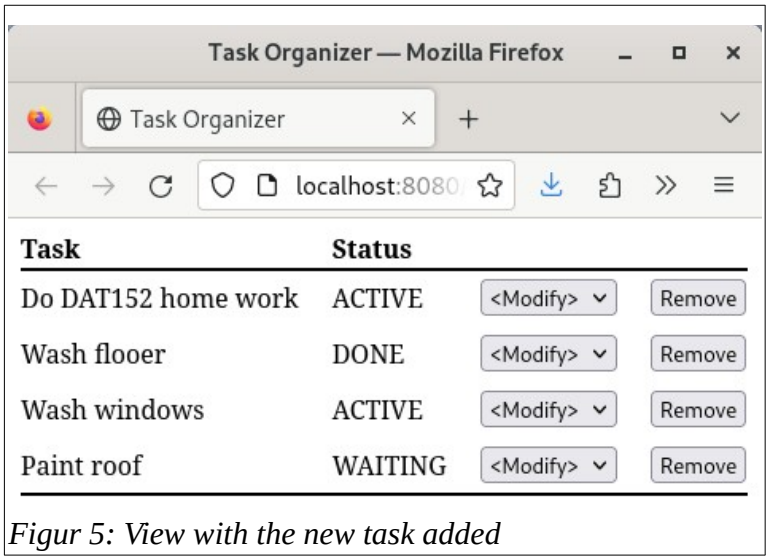
This method will update the view.

The method will add a task to the view. The new task will be added **to the top of the list**, with the oldest task at the bottom.

The code below demonstrates how to add a new task to the view. The method takes one argument, an object with data for the new task.

```
const newtask = {
    "id": 5,
    "title": "Do DAT152 home work",
    "status": "ACTIVE"
};
tasklist.showTask(newtask);
```

The illustration below shows the view after adding the new task. **Observe that the new task is added at the top of the list.**



Figur 5: View with the new task added

Method *updateTask*

This method will update the view.

The method will modify the status shown for a task.

The code below will set the status of the task with id 1 (*Paint roof*) to *ACTIVE* in the view. The method takes one argument, an object with data for the new status of the task.

```
const status = {
    "id": 1,
    "status": "ACTIVE"
};
tasklist.updateTask(status);
```

Method *removeTask*

This method will update the view.

The method removes a task from the view.

The code below will remove the task with id 1 (Paint roof) from the view. The method takes one argument, the numerical id of the task to remove.

```
tasklist.removeTask(1);
```

Method *setStatuseslist*

This method will update the view and set the list of possible values in the HTML **SELECT** element *Modify*.

This method lets the controller code specify the values to choose from in the HTML **SELECT** element, in addition to the <*Modify*> value.

The code below demonstrates how to use this method:

```
tasklist.setStatuseslist(["WATING","ACTIVE","DONE"]);
```

The method *setStatuseslist* must be run before the first task is added to the view.

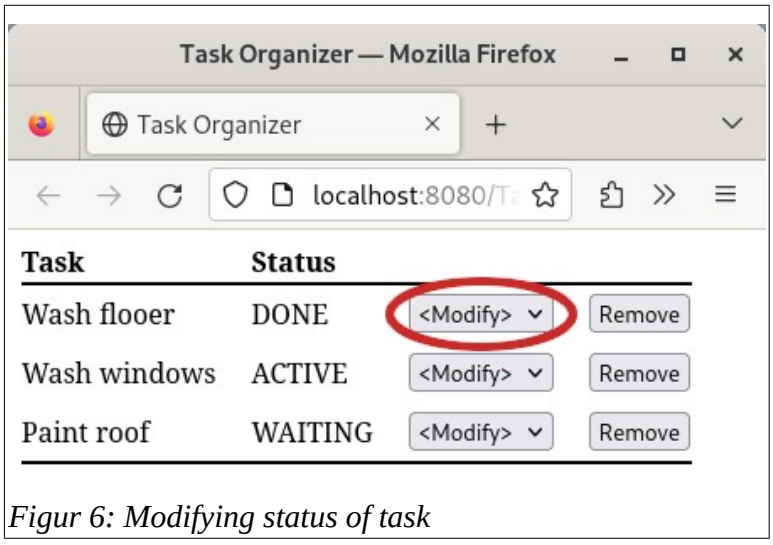
Method *changestatusCallback*

This method does not update the view.

This methods lets the controlling code add a callback, to be run when the user selects a different task status.

Each task has an unique identification that identifies the task, a positive integer.

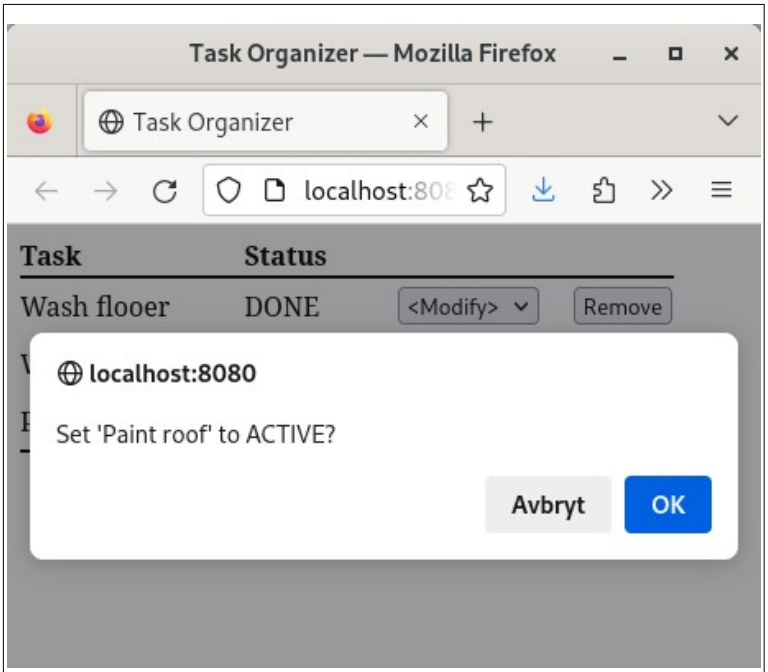
The callback should be run with two parameters, the id of the task to modify and the selected new status to be set for the task.



Figur 6: Modifying status of task

When the user selects a different status for a task, the *TaskList* component must inform the controlling code that the user wishes to modify a task. This method does updated the view. That will be achieved by another method, to be described later.

On status change, *TaskList* should use a confirmation window (*window.confirm*) where the user can confirm the change. If the user confirms, the callback will be run. The illustration below shows the confirmation window when the user tries to change the status of *Paint roof* to *ACTIVE*.



Figur 7: Confirmation window on status change

The code below demonstrates how to add a callback to be run when the user selects a different status and approves the change. The callback function will write a text in the browser console. The text will contain the id of the task and the selected status.

```
tasklist.changestatusCallback(
    (id, newStatus) => {
        console.log(`Status ${newStatus} for task ${id} approved`)
    }
);
```

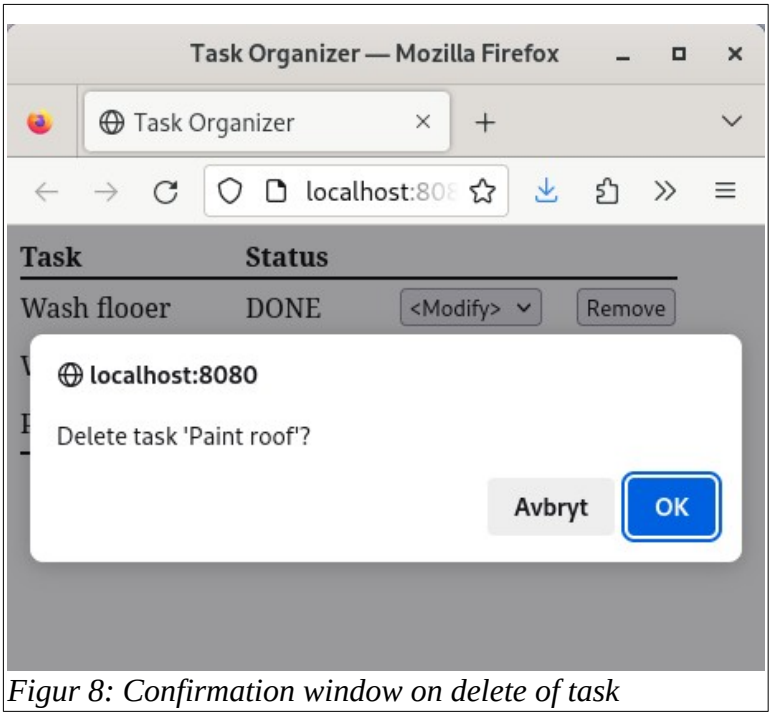
Observe, the above code does not modify the view.

Method *deletetaskCallback*

This method does not update the view.

This methods lets the controlling code add a callback, to be run when the user clicks on the delete button of a task. The callback should be run with one parameters, the id of the task to delete.

When the user wishes to delete a task, *TaskList* should open a confirmation window (*window.confirm*) where the user can confirm the action. The illustration below shows the confirmation window when the user tries to delete the *Paint roof* task.



Figur 8: Confirmation window on delete of task

The code below demonstrates how to add a callback to be run when the user clicks a delete button and approves the delete action in the confirmation window. The callback function will write a text in the browser console The text will contain the id of the task to delete.

```
tasklist.deletetaskCallback(
    (id) => {
        console.log(`Delete of task ${id} approved`)
    }
);
```

Method getNumtasks

This method does not update the view.

The method returns the number of tasks in the **TaskList** view.

Recommendations

You will find on Canvas a template for the **TaskList** component. Use this template. The **TaskList** template includes several HTML templates, and all HTML content should be created from these HTML templates. No additional *innerHTML* or *createElement* should be needed.

You should start implementing the *showTask* method.

The *showTask* method should check if the component has a HTML TABLE element, and if not create it by using the HTML template *tasktable*. When the last task is deleted, set the container DIV element content to an empty string, to delete all its content.