Task list web application: continued

In this assignment, you will continue working on the todo list application that you started before the midterms. The objective is to add interactivity to the application.

For this first iteration, we will use static data loaded directly in a variable (see the script `data.js`).

Improve / update your HTML structure

Your page must have:

- a responsive design
- a "navigation bar" with buttons (all tasks / pending / completed)
- the main area, showing the list of tasks
- a form that we will use to create new tasks

Make sure you have all these elements in your page. You can also use the `static.html` page provided as a reference / example.

Turn the form into a modal dialog box

Instead of having the form hang at the bottom of the page, we are going to display it in a modal dialog box in order to make the user interaction easier.

- Wrap your form into an HTML element, and give it an ID.
  - this element will be the "background" of the modal box
  - it will cover the whole viewport
  - it is usually a good idea to make it opaque in order to bring the focus to the actual dialog box
- You can then position your form inside this parent element - display it centered on the page.
- You may want to use `z-index` to make sure the elements "stack" properly.

Example

```
#modal_wrapper {
    position: fixed;
    top: 0;
    left: 0;
    right: 0;
    bottom: 0;
    margin: auto;
    padding: 2rem;
    background-color: rgba(0, 0, 0, 0.5);
    z-index: 9998;
}
```

This is the background for the modal: it takes the whole page, is 50% transparent, and is almost "at the top" of the CSS vertical stack (`9998`, the maximum is usually `9999` - see below).

```
#modal_contents {
    position: relative;
    width: 50vw;
    margin: 0 auto;
    background-color: #fff;
    z-index: 9999;
    padding: 1rem;
}
```

This is the actual contents of the dialog box. It takes about 50% of the width of its parent and is centered. The fact that we use `position: relative` allows us to use `position: absolute` for children elements. For example, you can add a button on the top right to close the modal:

```
#close_modal {
    position: absolute;
    top: 1rem;
    right: 1rem;
}
```

Add Javascript for interactivity

Make sure your modal is hidden when the page loads (you can set `display: none` in the CSS). Add a button to your page to show the modal box: add Javascript to change the style of the modal element when the button is clicked. Make sure your "close" button in the modal also has Javascript that hides the modal window when it is clicked.

Hint: use event propagation

It is a good idea to close the modal if the user clicks anywhere **BUT** the modal content. Using a separate element for the modal background allows us to do that by combining it with **event propagation**.

```
document.getElementById("modal_wrapper").addEventListener("click", (event) => {
    if (event.target.id === "modal_wrapper") {
        hideModal()
    }
})
```

In the code above, `#modal_wrapper` is the background of the modal dialog. When the user clicks on the page:

- if they click on the background (= `#modal_wrapper`), then `event.target.id === "modal_wrapper"`: we can hide the modal dialog
- if they click on the actual modal contents, the event will be first captured on this element before being propagated to its parent (the modal background). Then `event.target.id !== "modal_wrapper"`.

Load and render tasks dynamically in Javascript

The file `data.js` contains tasks data that you can use. Add Javascript code to render all the elements in the `taskData` variable into HTML elements when the page loads.

Side note: arrow functions

Javascript uses a lot of functions (callback), and sometimes these functions are very short (a few lines). There is an alternative syntax to define functions, called **arrow functions**. They are essentially the same as regular functions - it is just a more concise syntax.

You already know that the following functions are the same (**function expressions**):

```
function myFunc(a, b, c) {
    // do something with a, b, c
    return myValue
}

const myFunc = function(a, b, c) {
    // do something with a, b, c
    return myValue
}
```

To create an arrow function, you simply remove `function` before the parameter list, and you add an arrow `=>` between the parameter list and the body of the function.

```
const myFunc = (a, b, c) => {
    // do something with a, b, c
    return myValue
}
```

There are other rules:

- if there is a single argument, then the brackets are not required around the argument

```
const myFunc = a => {
    // do something with a
    return myValue
}
```

- if there is a single statement in the body of the function, the `{}` and `return` statement can be omitted

```
const addTwo = myNumber => myNumber + 2
console.log(addTwo(3))
```

Arrow functions can help with readability of your Javascript code.

Side note: looping in Javascript

Aside from the `while` and `for` loops that we have seen, Javascript also has options to loop over an *iterable* (like in Python). There are two ways to do this:

- using a `for ... of` syntax
- using the `forEach` method with a callback function. The callback function receives each element of the iterable, one after the other.

Examples

```
const myList = ["a", "b", "c"]

// Method 1

for (const element of myList) {
    console.log(element)
}

// Method 2
myList.forEach(function(element) { console.log(element) })
// or with an arrow function
myList.forEach(element => console.log(element))
// a more complex example - will add paragraphs in the document
myList.forEach(element => {
    const myParagraph = document.createElement("p")
    myParagraph.textContent = element
    document.getElementById("main").addChildren(myParagraph)
})

// you can also access the index of the current element with forEach:
myList.forEach((index, element) => {
    const myParagraph = document.createElement("p")
    myParagraph.textContent = element
    myParagraph.classList.add(`paragraph_${index}`)
    document.getElementById("main").addChildren(myParagraph)
})
```

Add the edit and delete buttons

When rendering the tasks into the DOM, add two button elements for each task: one to "edit", and one to "delete". We will deal with the "delete" button later. For the time being, make sure that, when the edit button is clicked:

- the modal dialog box is displayed
- the values in the dialog box / form are updated to reflect the values for the task whose "edit" button was clicked