Exercise on Ajax

Last update on August 29, 2025.

If you need help with this exercise, you must show up at the lab sessions. There will be no private help unless you also come to the lab.

This exercise consists of two parts. In part one you must create a user interface for viewing tasks. In part two you must use Ajax to retrieve, store and update a database with tasks stored on the server. Your solution to part one should be used in part two to display the tasks stored on the server.

Requirements for the assignment

The following requirements must be fulfilled to get the assignment approved:

- Deadline is Monday, September 22.
  - A missing delivery on that date will count as one failed attempt.
- You have to work and deliver in groups of 3 to 5.
  - You must sign up for a group no later than Wednesday September 17.
- The group must deliver together on Canvas.
- Submission is compulsory.
- File names should not contain national letters. In JavaScript code, national can letters only be used in comments.
- The code must be easily readable. Use indentation to show the structure of the code.
- Use objects, components and class constructs. Use of global variables and functions are not approved.
  - The directory structure should be that of the supplied Eclipse archive *apitemplate.zip*.
- Submission is done by delivering on Canvas either a *tar* (can be compressed) or a *zip* archive containing:
  - an Eclipse project for a working solution, or
  - the the client side only for a working solution.
- Name the archive *TaskList_group_<your_group_number>*, and replace *<your_group_number>* with the number of your group on Canvas.
- You must be able to solve all tasks of this exercises. Answers like "It did not work" will not be accepted. At the lab you will get guidance, and you should then be able to solve all tasks.
- A working solution must be demonstrated to the lecturer.
  - Date and time for each group will be given by the lecturer.

    - All group members must be able to explain every detail of the solution.
- You must not use frameworks, i.e. no use of jQuery, Prototype, Angular(JS), Vue.js or React. If you use code snippets that you copy from others, you must be able to understand and explain the code.

If your application fail on the lecturer's setup with Spring Boot in Eclipse, the delivery will not be approved. You can use e.g. IntelliJ to solve the tasks as the source code should not depend on the IDE.

The client side JavaScript code can be delivered as an Eclipse project.

- Eclipse projects have file extension as *zip*, *tar* or *tar.gz*. War files and rar archives are **not** Eclipse projects. You must use the export capability of Eclipse.

The supplied Eclipse archive *apitemplate.zip* uses the software versions below.

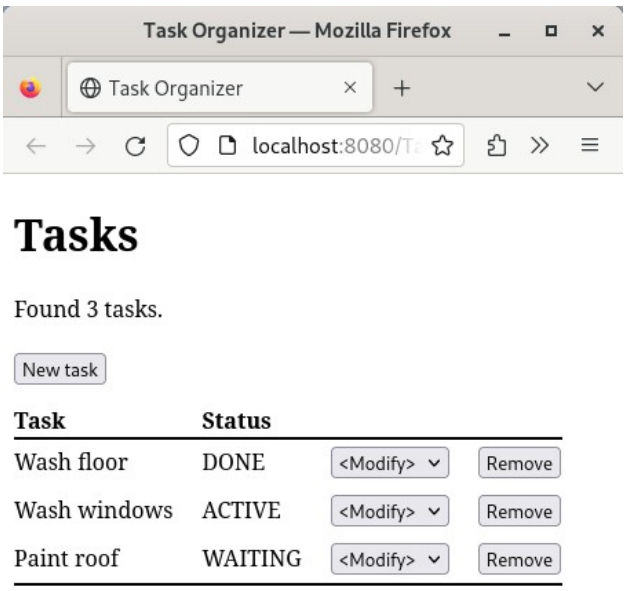
- Eclipse, version "2025-06" for Java Developers.
- Java 21 (OpenJDK).

Part one: User interface for tasks

Here in part one you will create a web application that displays tasks and their statuses. You can add your code to project of the supplied archive *apitemplate.zip* and then rename the project as specified above.

You should not implement functionality to authenticate users. You can assume that users are authenticated prior to using the *TaskList* application.

In this exercise you will create a JavaScript component **TaskView**. A possible view of the **TaskView** component is seen in the illustration below.



Figur 1: View of TaskView

The HTML **BODY** of the file *index.html* that produces the above view is:

```
<body>
    <task-view data-serviceurl="./api"></task-view>
</body>
```

Code snippet 1: HTML body of the file index.html

The component will fetch a list of tasks from the server, and the attribute *data-serviceurl* is the URL to the web services, as explained in part two of the assignment.

This view of the component should be based on the following HTMLTemplateElement, created by the module of the component:

```
const template = document.createElement("template");
template.innerHTML = `
    <link rel="stylesheet" type="text/css"
        href="${import.meta.url.match(/.*\//)[0]}/taskview.css"/>

    <h1>Tasks</h1>

    <div id="message"><p>Waiting for server data.</p></div>
    <div id="newtask">
        <button type="button" disabled>New task</button>
    </div>

    <!-- The task list -->
    <task-list></task-list>

    <!-- The Modal -->
    <task-box></task-box>
`;
```

Code snippet 2: HTML template of TaskView

The template references a CSS-file *taskview.css* that must exist in the folder of the component JS file.

Tag *task-list* used by the template creates an instance of the **TaskList** component of the previous JavaScript assignment. Tag *task-box* creates and instance of a **TaskBox** component that lets the user add a new task to the task list. Observe that the **TaskBox** component is not visible in the illustration shown in figure 1 on page 2.

All components **TaskView**, **TaskList** and **TaskBox**,should be created as JavaScript modules.

Neither **TaskList** or **TaskBox** should have no knowledge about Ajax. The assignment will not be approved if there is Ajax functionality in any of these components. All Ajax functionality should be put in the methods of **TaskView**, and the base of the URL for the Ajax requests must be taken from the attribute *data-serviceurl* used with the task-view tag.

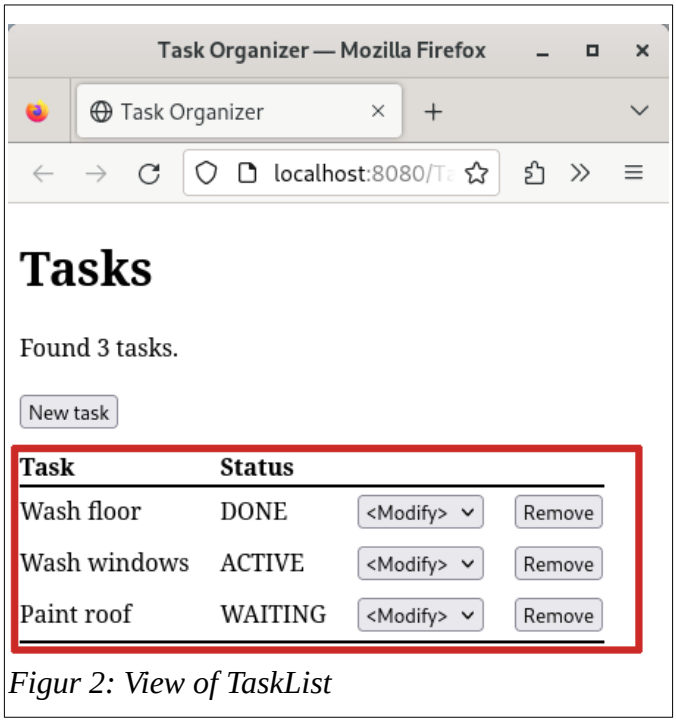
TaskView graphical elements

TaskView is made up of several graphical elements, some of which will be explained here.

TaskList

The component **TaskList** was the topic of the previous lab assignment. See that assignment for details on **TaskList**.

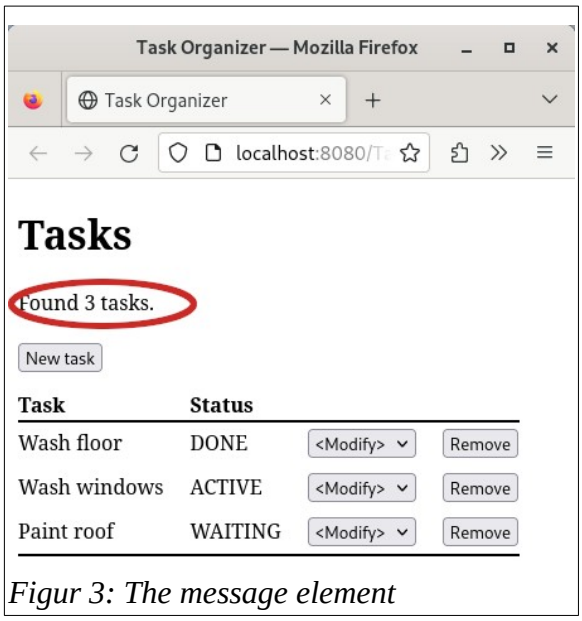
The illustration below outlines the view of the **TaskList** as a component used by **TaskView**.



Figur 2: View of TaskList

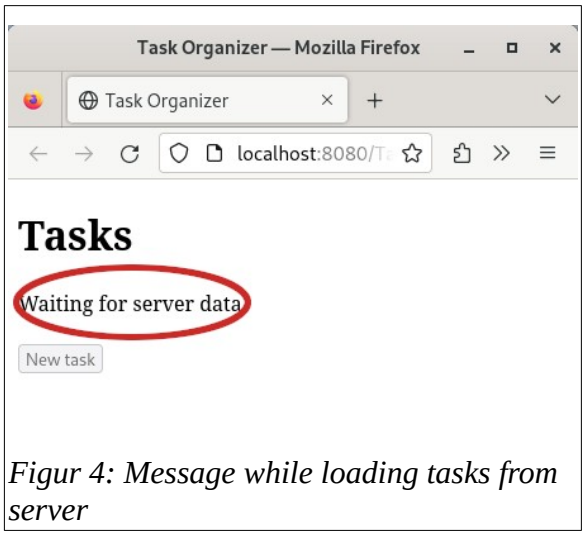
Message element

TaskView contains a message elements, seen in code snippet 2 as the *DIV* element with *ID* equal to *message*. The illustration below outlines the view of the the message element with three tasks in the task list.



Figur 3: The message element

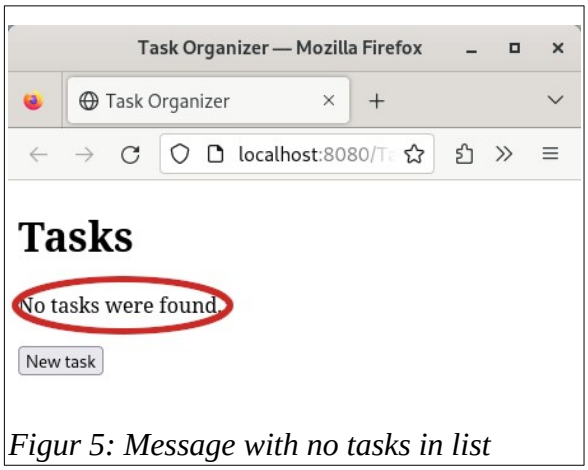
The illustration below shows the message element while the application is waiting for the tasks to be fetched with Ajax from the web server.



Figur 4: Message while loading tasks from server

Observe that the button *New task* is greyed out while waiting for the tasks.

The illustration below shows the message element if there are no tasks in the list.

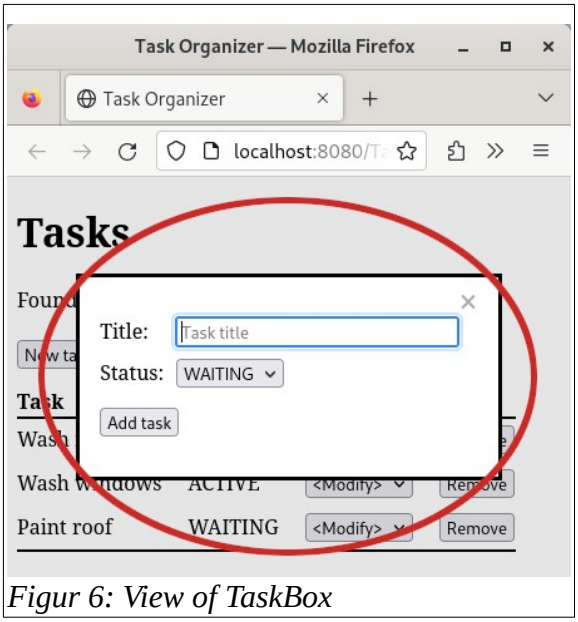


Figur 5: Message with no tasks in list

TaskBox component

The tag *task-box* creates an instance of **TaskBox**, a component that manages a modal box where the user can add details of a new task.

A possible view of the **TaskBox** component is outlined in the illustration below.



Figur 6: View of TaskBox

When the user clicks the button *New task* of **TaskView**, the component should open the modal box of **TaskBox** where the user can add details of a new task, as seen in the illustration above

This view of the **TaskBox** component should be based on the following HTMLTemplateElement, created by the module of the component:

```
const template = document.createElement("template");
template.innerHTML = `
    <link rel="stylesheet" type="text/css"
        href="${import.meta.url.match(/.*\//)[0]}/taskbox.css"/>
    <dialog>
       <!-- Modal content -->
        <span>&times;</span>
        <div>
            <div>Title:</div>
            <div>
                <input type="text" size="25" maxlength="80"
                    placeholder="Task title" autofocus/>
            </div>
            <div>Status:</div><div><select></select></div>
        </div>
        <p><button type="submit">Add task</button></p>
     </dialog>
`;
```

Code snippet 3: HTML template of TaskBox

The template references a CSS-file *taskbox.css* that must exist in the folder of the component JS file.

TaskView should call the method *showModal* of the HTML *DIALOG* element to display the modal box.

The **TaskBox** component should have the following public methods:

- *show()* - Opens (shows) the modal box in the browser window.
- *setStatuseslist(list)* – Sets the list of possible task statuses.

- *newtaskCallback(callback)* - Adds a callback to run at click on the *Add task* button.
- *close()* - Removes the modal box from the view.

When **TaskBox** runs a method set with *newtaskCallback*, the method must be run with the new task as parameter.

The modal box should close if the user clicks the close symbol or press the Escape button, or if the *close()* method is called.

The JavaScript code below demonstrates how to use the **TaskBox** JavaScript API.

```
const taskbox = document.querySelector("TASK-BOX");
taskbox.newtaskCallback(
    (task) => {
        console.log(`Have '${task.title}' with status ${task.status}.`);
        taskbox.close();
    }
);
taskbox.setStatuseslist(["WATING","ACTIVE","DONE"]);
taskbox.show();
```

Code snippet 4: Demonstrating the TaskBox JavaScript API

The **TaskView** component should use code similar to that of code snippet 4 above to initialize a **TaskBox** component.

In part two of this assignment, the component **TaskView** should use Ajax to fetch the list of possible statuses from the web server.

Each task will get a unique id that is set by the web server and returned to the application. In part two of this exercise, tasks are stored in a database on the server side, and the task id will be equal to the primary key that is chosen by the database.

Part two: Interacting with the server through Ajax

In this part of the exercise you will use the API of **TaskList** and **TaskBox** to update the task database on the server through the use of Ajax. The parent component **TaskView** will need to add callbacks to **TaskList** through the methods *changestatusCallback* and *deletetaskCallback*, and to **TaskBox** through method *newtaskCallback*.

Implement all Ajax functionality described below using the Fetch API. You should not do any changes to **TaskList**, nor **TaskBox** from part one. All Ajax functionality must be added to **TaskView**. **TaskView** then uses the APIs of **TaskList** and **TaskBox** to create and updated the view of the application.

The server side application does not include functionality required to update concurrent clients on changes to the database. You can therefore assume that only one client at any time is working with the server database. In a real scenario with concurrent access to data, clients must be informed about changes made by others:

- Clients can regularly pull changes from the server with Ajax, or

- the server can push changes to the clients using websockets or Server-Sent Events.

The server side part of the application has been made ready for this exercise. In the first assignment you set up this application in Eclipse. In the Eclipse project

The response documents from the server side application is sent with the following content type:

```
application/json; charset=utf-8
```

The following services are available from the server side application:

- GET api/allstatuses
- GET api/tasklist
- GET api/task/{id}
- POST api/task
- PUT api/task/{id}
- DELETE api/task/{id}

Note: Do not modify the service contacts of the of the provided application. The lecturer will test your solution and the test will fail if the server side API has been changed.

In the supplied Eclipse archive *apitemplate.zip* you will find a demonstration of the service contacts at the URL http://127.0.0.1:8080/TaskList/demo/index.html.

Do not modify the server side part of the application, or your application will fail when the lecturer test your solution.

The H2 database of **TaskList** stores the database in memory, and the database is lost when the server is restarted. The application will create an initial database with three tasks. You can modify the application to store the database on disk, and remove the initial tasks.

You can access and work with the in memory database through the URL http://localhost:8080/TaskList/h2-console. Authentication details are found in the project file *application.properties*.

You can make the database persistent by modifying the file *application.properties* and update the property *spring.datasource.url* to save to a file on disk.

The database is populated with three tasks by the file *data.sql*. If you store the database to disk, you can remove the file that creates the three initial tasks.

Service "GET api/allstatuses"

The service retrieves a list of all possible states that a task can have. This list is used to populate the option elements of the **SELECT** elements *Modify* of **TaskBox**.

```
GET api/allstatuses
```

An example of JSON that is returned by the service is shown below. The text is formatted for improved readability:

```
{
    "allstatuses": ["WAITING","ACTIVE","DONE"],
    "responseStatus":true
}
```

The property *responseStatus* has value **true** if the statuses were found in the database, **false** otherwise. When *responseStatus* has the value **true**, the response will also have a property *allstatuses* with the list of all possible task statuses.

Service "GET api/tasklist"

The service retrieves a list of all task from the server. This list is used to create the list of tasks that is displayed by **TaskList**.

```
GET ../TaskServices/api/services/tasklist
```

An example of JSON that is returned by the service is shown below. The text is formatted for improved readability:

```
{
    "responseStatus":true,
    "tasks":
        [
            {"id":1,"title":"Paint roof","status":"WAITING"},
            {"id":2,"title":"Wash windows","status":"ACTIVE"},
            {"id":3,"title":"Wash floor","status":"DONE"}
        ]
}
```

The property *responseStatus* has value **true** if the statuses were found in the database, **false** otherwise. When *responseStatus* has the value **true**, the response will also have a property *tasks* with the list of all tasks in the database.

Service "POST api/task"

The service adds a task to the database. You add a task to the database with the request below:

```
POST api/task
```

The service expects that the data is sent to the server with the following content type:

```
application/json; charset=utf-8
```

Data for the new task must be sent as JSON with properties *title* and *status*. Below is an example of data that the service will accept:

```
{
    "title": "Something more to do",
    "status": "WAITING"
}
```

An example of JSON that is returned by the service is shown below. The text is formatted for improved readability:

```
{
    "task":
        {
            "id":6,
            "title":"Something more to do",
            "status":"WAITING"
        },
    "responseStatus":true
}
```

The property *responseStatus* has value **true** if the task was added to the database, **false** otherwise. When *responseStatus* has the value **true**, the response will also have a property *task* with properties of the task. The property *id* corresponds to an unique attribute value that the task has in the database, i.e. its primary key.

Service "GET api/task/{id}"

The service retrieves a single ask from the database.

Parameter *id* specifies what task to get and corresponds to the unique attribute *id* that we got from the POST request, i.e. the primary key of the task.

The request below shows how to retrieve the task with *id* equal to 2:

```
GET api/task/2
```

You will probably not need this service in your solution.

Service "PUT api/task/{id}"

The service updates the status for a task that already exists in the database. Parameter *id* specifies what task to update and corresponds to the unique attribute *id* that we got from the POST request, i.e. the primary key of the task.

The request below shows how to update the task with *id* equal to 2:

```
PUT api/task/2
```

The service expects that the data is sent to the server with the following content type:

```
application/json; charset=utf-8
```

Data for the new status must be sent as JSON with a property *status*. Below is an example of data that the service will accept:

```
{
    "status": "DONE"
}
```

An example of JSON that is returned by the service is shown below. The text is formatted for improved readability:

```
{
    "id":2,
    "status":"DONE",
```

```
    "responseStatus":true
}
```

The property *responseStatus* has value **true** if the status of the task was updated in the database, **false** otherwise. When *responseStatus* has the value **true**, the response will also have a property *status* that is the new status of the task. The property *id* corresponds to the unique attribute that identifies the task.

Service "DELETE api/task/{id}"

The service removes a task from the database. Parameter *id* specifies what task to update and corresponds to the unique attribute *id* that we got from the POST request, i.e. the primary key of the task.

The request below shows how to remove the task with *id* equal to 2:

```
DELETE api/task/2
```

An example of JSON that can be returned by the service is shown below. The text is formatted for improved readability:

```
{
    "id":2,
    "responseStatus":true
}
```

The property *responseStatus* has value **true** if the task was removed from the database, **false** otherwise. The property *id* corresponds to the unique attribute that identified the task.

Modifying the view

The POST, DELETE and PUT HTTP requests can all update the database stored on the server. If the database was updated, the response parameter *responseStatus* will have the value **true**. The view from part one should therefore be modified only after POST, DELETE and PUT requests, and only if *responseStatus* is **true**.

Requirements to the application

All the below requirements must be met, or the assignment will be "fail".

- All functionality of the application must work as specified.
- All components, **TaskView**, **TaskList** and **TaskBox** must be imported as JavaScript modules.
- The methods that add callbacks to **TaskList** and **TaskBox** must only be called from **TaskView**.
    - *changestatusCallback*, *deletetaskCallback* and *newtaskCallback*.
- There must be no references to the **TaskView** from **TaskList** or **TaskBox**.

- There must be no references to **TaskList** from **TaskBox**.
- There must be no references to **TaskBox** from **TaskList**.
- All HTML elements of the task list should be created and managed only by the **TaskList** component.
    - Neither **TaskView**, nor **TaskBox** should refer to any of the HTML elements of the task list.
    - All event listeners on the HTML elements of the task list must be managed by the **TaskList** component.
- All HTML elements of the modal window should be managed only by the **TaskBox** component.
    - Neither **TaskView**, nor **TaskList** should refer to any of the HTML elements of the modal window.
    - All event listeners on the HTML elements of the modal window must be managed by **TaskBox**.
- All access to **TaskList**, **TaskBox** and the HTML structures managed by these should be through the component APIs.
    - **TaskList:**
        - *setStatuseslist, changestatusCallback, deletetaskCallback, showTask, updateTask, removeTask* and *getNumtasks*.
    - **TaskBox:**
        - *show, setStatuseslist, newtaskCallback* and *close*.
- There must be no use of Ajax or *fetch* in **TaskList** nor **TaskBox**.
    - All use of Ajax must be put in the methods of **TaskView**.
    - If using a separate class for Ajax, its API should only be called by the methods of **TaskView**.
- The URLs to the services should be relative paths within the application.
    - No modifications should be necessary if moving the application to a different host or if using a gateway server.
    - The base of the URL to be used for the Ajax requests is given **TaskView** through the attribute *data-serviceurl*.
- There must be no use of data from external sources with *innerHTML*, *outerHTML* or *insertAdjacentHTML*.
    - No user supplied data or data read from the database.
- Any reference to an instance of a class from within the class must use the keyword *this*.

Try also to follow the below advices.

- Do not mix the `await` and `async` syntax with the `.then(…).catch(…)` syntax when using Promises.
  - The syntax with `await` and `async` gives code that is easier to read.
- Do not embed JavaScript code in the HTML code.
  - Do not use *onclick* attributes on HTML elements in the HTML code.
  - Do not put JavaScript inside the **SCRIPT** tags, but use separate JavaScript files.
- All **SCRIPT** tags should be put in the head of the document.
  - Attribute *type="module"* implies *defer.*
- There is no need of a separate JavaScript Array to store the task list.
  - Tasks can be read from the HTML table using DOM methods.
- Document your code using e.g. JSDoc, see e.g. https://jsdoc.app/.

Test the application

In order to test that you solution fulfils the requirements of the exercise, use a web document with the HTML body below.

```
<body>
    <!-- First task view -->
    <task-view></task-view>

    <!-- Second task view -->
    <task-view></task-view>
</body>
```

The modal boxes of either of the **TaskView** instances can be used to add tasks to the list, and both instances should display the same tasks. Since **TaskView** does not poll for changes, the page must be reloaded after a modification though to make the other **TaskView** to see the same task list.