Exercise on Ajax

Last update on September 5, 2022.

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 19 at 15:00.
- You have to work and deliver in groups of 3 to 5.
- 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 methods are not approved.
- Submission is done by delivering an archive with the client side JavaScript code on Canvas.
    - You should only deliver the client-side part of the solution and the solution must work with the lecturer's server side installation of *TaskServices*.
    - Only *tar* and *zip* archives will be accepted. A tar archive can be compressed.
- If using Eclipse, keep the web server name as suggested by Eclipse.
- 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 be able to solve all tasks.
- 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 Eclipse or IntelliJ, the delivery will not be approved.

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.
- Name the Eclipse project *TaskOrganizer_group_<your_group_number>*, and replace *<your_group_number>* with the number of your group on Canvas.
- The archive you provide must not contain the server side application *TaskServices* that is delivered with this exercise.

Software requirements for Eclispe

The application must use the software versions below.

- Eclipse, version "2022-06" for Enterprise Java Developers.
- Apache TomEE Microprofile version 8.0.12.
    - Built upon Apache Tomcat 9.0.63.
- Java 11 (OpenJDK) or newer.
    - Observe that the last stable version of Eclipse with TomEE do not work well with the newest versions of Java.

Software requirements for IntelliJ

The application must use the software versions below.

- IntelliJ, version 2022.2.1 or newer.
- Apache TomEE Plume version 8.0.12.
    - Built upon Apache Tomcat 9.0.63.
- Java 11 (OpenJDK) or newer.

Preparations

You can choose to deliver a solution for either Eclispe or IntelliJ. Instructions on how to configure and install Eclipse was covered in the first lecture, and you find detailed instructions on Canvas.

Instructions for IntelliJ are found at https://eple.hib.no/fag/dat152/h2022/intellij/

Part one: User interface for tasks

Here in part one you will create a web application that displays tasks and their statuses. Create a new *Dynamic Web Project in Eclipse*, and name the project *TaskOrganizer_group_<your_group_number>* where you replace *<your_group_number>* with the number of your group on Canvas. If you use IntelliJ, you answer the assignment by delivering an archive, zip or tar, with the content of the webapp directory of module *TaskOrganizer*.

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

The user interface is made up of two components, **TaskList** and **TaskBox**. The first of these, **TaskList** was the solution of the first JavaScript exercise. The **TaskBox** component lets the user add a new task.

The previous assignment on **TaskList** was missing one method, *setStatuseslist*. The exercise text of assignment two has been updated on Canvas and now also includes this method.

All components, **TaskList** and **TaskBox**, and eventual a common parent component should be created as JavaScript modules.

When the user clicks the button *New task* of **TaskList**, the controller code, or parent component should open a modal box where the user can add details of a new task. Create a JavaScript component **TaskBox** where the user can fill inn the necessary details. Use the HTML element **DIALOG** to create the modal box. Use the method *showModal* of the **DIALOG** element to display the modal box.

As for **TaskList**, **TaskBox** should have no knowledge about Ajax.

The illustration below shows a possible HTML **DIALOG** element for adding a new task.

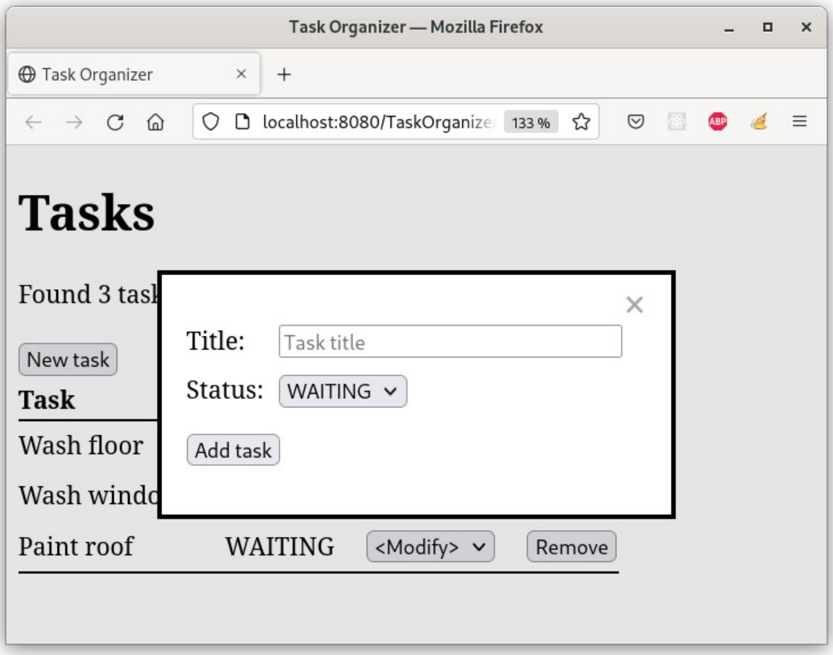

Figure 1: Modal box of DIALOG element for adding a new task

An HTML **BODY** for the above view, using both **TaskList** and **TaskBox** can be:

```
<BODY>
    <H1>Tasks</H1>

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

In the above HTML, the **BODY** element is the parent of the **TASK-LIST** and **TASK-BOX** components. You can chose to put these components into a common parent component, e.g. **TASK-VIEW**.

The **TaskBox** component should have the following 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 **TaskBox**.

```
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();
```

In part two, the list of possible statuses should be fetched by the controller code, or a common parent from the webserver with Ajax.

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 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 controller code, or a parent component will need to add callbacks to **TaskList** through the methods *addtaskCallback*, *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 functionality must be added through the component APIs of **TaskList** and **TaskBox**.

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.

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

The project **TaskDemo** expects that the URL to the services of **TaskServices** is "../TaskServices/broker". Do not modify the URL, or your application will fail when the lecturer test your solution.

The Derby database of **TaskServices** 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 make the database persistent by modifying the file "TaskServices/Java Resources/src/META-INF/resources.xml" and update the property *jdbcUrl* to save to a file on disk.

The database is populated with three tasks in the *setupDB* method of the Java class **hvl.dat152.dataaccess.jdbc.TaskDAOImpl**. If you store the database to disk, you can comment out the code that creates the three initial tasks.

Service "GET broker/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 ../TaskServices/broker/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 broker/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/broker/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 broker/task"

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

```
POST ../TaskServices/broker/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 "PUT broker/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 ../TaskServices/broker/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 broker/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 ../TaskServices/broker/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, **TaskList** and **TaskBox**, and eventual a common parent component must be imported as default JavaScript modules.
- The methods that add callbacks to **TaskList** and **TaskBox** must only be called from controller code or a common parent component.
    - *addtaskCallback*, *changestatusCallback*, *deletetaskCallback* and *newtaskCallback*.
- There must be no references to the controller code, or a common parent 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 handled only by the **TaskList** component.
  - Neither the controller code, a common parent component, 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 handled by the **TaskList** component.
- All HTML elements of the modal window should be handled only by **TaskBox** component.
  - Neither the controller code, a common parent component, 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 handled by **TaskBox**.
- All access to **TaskList**, **TaskBox** and the HTML structures handled by these should be through the component APIs.
  - **TaskList:**
    - *enableaddtask*, *setStatuseslist*, *noTask*, *showTask*, *updateTask*, *removeTask addtaskCallback*, *changestatusCallback* and *deletetaskCallback*.
  - **TaskBox:**
    - *newtaskCallback*, *setStatuseslist*, *show* and *close*.
- No Ajax or use of fetch in **TaskList** or **TaskBox**.
  - All use of Ajax must either be put in a separate class, in the controller code, or in common parent component.
  - If using a separate class for Ajax, its API should only be used by the controller code or a common parent component.
- 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.
- 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*.
- Do not include pure text into the document using *innerHTML*, *outerHTML* or *insertAdjacentHTML*.
    - These methods can be used to build HTML skeletons, but not for HTML code constructed from data.
- 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.
    - Both a JavaScript Array and the HTML table must be updated when tasks are modified.
- 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.

```html
<body>
    <h1>Tasks</h1>

    <!-- First task list -->
    <task-list></task-list>

    <!-- Second task list -->
    <task-list></task-list>

    <!-- First Modal -->
    <task-box></task-box>

    <!-- Second Modal -->
    <task-box></task-box>
</body>
```

Both modal boxes can be used to add tasks to the list, and both task containers should display the same tasks. No modifications should be necessary to neither **TaskList**, nor **TaskBox**. When tasks are modified, added or deleted, both task list containers should be updated and display the current list.

You may have to duplicate or modify some controller code, or a common parent component. It is possible though to have the application work without any changes to the controller code when more **TaskList** or **TaskBox** components are added.