Assignment 2.B (7%) - Complete Client Site JavaScript Application

T-213-VEFF, Web programming I, 2026-1

Reykjavik University - Department of Computer Science, Menntavegi 1, IS-101 Reykjavík, Iceland

Deadline: Friday, February 27th 2026, 23:59

This is Part B of the second assignment in Web Programming I. In this part, you will complete the JavaScript functionality for the rest of the dashboard by building on your Part A submission.

This assignment has to be done individually, no group work is permitted.

1. Tasks and Quick Notes Widgets

In Part B, you will extend the dashboard by implementing the functionality of the *Today's tasks* and *Quick notes* widgets. This part builds directly on your solution from Part A, where you implemented the *Daily quote* widget.

Your task is to write JavaScript code that communicates with a local REST API to retrieve, display, create, and update data for both widgets. The *Today's tasks* widget should allow the user to view a list of tasks, add new tasks, and mark tasks as finished or unfinished. The *Quick notes* widget should allow the user to view and update a block of text-based notes.

The focus of this part is correct communication with a REST API using different HTTP methods, proper handling of asynchronous operations, and correct manipulation of the DOM using JavaScript. You are expected to keep the UI in sync with the backend state.

As in Part A, visual styling and layout are **not** part of this assignment. You should not modify the provided HTML or CSS.

Finally, as part of this assignment, you are required to deploy the frontend to a publicly accessible website and submit the deployment URL.

You may need to use JavaScript features and API concepts that have not yet been fully covered in the lectures. Helpful resources include developer.mozilla.org.

2. Starter Pack

Part B builds directly on your solution from Part A.

- If you completed Part A, you must continue working in the same project.
- If you did not complete Part A, you must use the provided Part A starter pack before implementing Part B.

All implementation must be done in `app.js`. The HTML and CSS files are provided and must **not** be modified.

The structure of `app.js` is required for the Gradescope setup and must not be changed. In particular:

- Do not rename or remove any existing functions
- Do not remove the `export` statement
- Do not remove the `init()` call at the bottom of the file

The project is configured as a JavaScript module. This is necessary for the Gradescope setup and requires the project to be run using the included Vite development server:

- Run `npm install`
- Run `npm run dev`
- Open the local development URL shown in the terminal

Do not open the `index.html` file directly in the browser. Since the project uses JavaScript modules, it must be served through the development server.

3. Requirements

In Part B, you must implement the functionality for the *Today's tasks* and *Quick notes* widgets using the provided local REST API.

1. **General requirements**
   - All implementation must be done in `app.js`.
   - The provided HTML and CSS must **not** be modified.
   - The predefined structure of `app.js` must be preserved to ensure compatibility with the Gradescope setup.
   - All communication with the backend must be done using HTTP requests.
2. **Today's tasks: loading tasks**
   - When the page loads, tasks must be fetched from the backend and displayed as a list.
   - Each task must show its text and completion state, represented by a checkbox reflecting the `finished` status.
3. **Today's tasks: updating task status**
   - Toggling a task checkbox must correctly update the backend without breaking the application.

4. **Today's tasks: adding new tasks**
    - The user must be able to add a new task using the input field, either by clicking the Add task button or pressing Enter.
    - Empty or invalid tasks must not be added.
    - After a successful addition, the task list must update to include the new task.
5. **Quick notes: loading notes**
    - When the page loads, the current notes must be fetched from the backend.
    - The notes text must be displayed inside the notes text area.
6. **Quick notes: updating notes**
    - The Save changes button must be disabled by default and enabled only when the notes text is modified.
    - Notes must be saved to the backend only when the user clicks the Save changes button, using the correct HTTP method.
    - After a successful save, the Save changes button must be disabled again until further changes are made.
7. **Error handling**
    - All failed backend requests must be handled.
    - In case of an error, it is sufficient to log the error using `console.log` or `console.error`.
    - The application must not crash or stop working due to an error.
    - You are **not** required to display error messages in the user interface.
8. **JavaScript usage**
    - You must use `const` and `let` instead of `var`.
    - You must use arrow functions when defining functions.
    - Asynchronous code must be written using `async/await`.
9. **Deployment**
    - The frontend must be deployed to a publicly accessible website (e.g., using Netlify).
    - Only the frontend needs to be deployed; the local backend should not be deployed (we will use a local backend to grade your assignment)
    - The deployment URL must be added to a file called `myDeploymentUrl.txt`, the only content of the file should be your url.
    - The deployed version must match the submitted source code.
    - Instructions for the deployment process are available on Canvas and will also be demonstrated during the Friday lecture on February 20th (recording available after the class).

3. Starting up the backend (REST API)

In Part B of this assignment, you will use a **local REST API** to implement the functionality for the *Today's tasks* and *Quick notes* widgets. This backend is provided as part of the assignment and must be run locally on your machine.

Note that this is different from Part A, where the *Daily quote* widget communicates with a remote API hosted on a server. In Part B, all task and note related functionality must communicate with the local backend described below.

Below you have all the steps necessary to run the backend. I will also show you how to start up the backend in the Friday lecture on February 20th, which will also be available on recording after the class.

1. It is necessary to install **node.js**, and it is also good to install an software to test and use the backend, e.g. **Insomnia**. Links for node.js and Imsonia can be found in Canvas on the "Useful Links and Tools" page.

2. Use the backend shared with you **assignment_2_localBackend**, and navigate to the folder in a terminal window (can be done e.g. in terminal via VSCode).

3. Install the NPM packages listed in package.json by running the `npm install` command (`npm i` does the same thing). This will fetch all the required packages needed to run the backend.

4. To start up the process, you can run the `npm start` command. The process will now print out a message telling you it has started:

```
> my_dashboard@1.0.0 start
> node ./index.js
Server running on http://localhost:3000/api/v1
```

5. Your backend is now running http://localhost:3000 and can be accessed via Insomnia, Postman, cURL or your website.

The backend supports a number of endpoints ("actions") that are relevant to this assignment. These are:

1. **GET** /api/v1/tasks
   A GET request to this URL returns a list of all tasks. Each task object contains:
   - **id**: a unique numeric identifier
   - **task**: the task description (string)
   - **finished**: completion status (0 = not finished, 1 = finished)

2. **POST** /api/v1/tasks
   A POST request to this URL creates a new task. The request body must contain:
   - **task**: a non-empty string describing the task

   If successful, the response returns the newly created task object with a generated **id** and **finished** set to 0.

3. **PATCH** /api/v1/tasks/{id}
   A PATCH request to this URL updates the completion status of an existing task. The **id** parameter in the URL specifies which task to update. The request body must contain:

    - **finished**: either 0 (not finished) or 1 (finished)

    If the task does not exist or the request body is invalid, an error response is returned.

4. **GET** /api/v1/notes
   A GET request to this URL returns the current notes text as a single string.

5. **PUT** /api/v1/notes
   A PUT request to this URL updates the notes text. The request body must contain:

    - **notes**: a string representing the updated notes

   If successful, the updated notes are returned in the response.

In general, the different endpoints return 2xx status codes if successful, and 4xx/5xx status codes if unsuccessful. The POST method returns the created object, if successful. To call any of the endpoints, you send an HTTP request with the right HTTP method (verb) and the right request body to "your" backend url with the added endpoint path . Note that GET requests do not have a request body, while POST requires it. Additionally, it might be a good idea to start trying out the different requests using Insomnia, Postman or cUrl.

4. Submission

The project is submitted via Gradescope. Submit the following files:

- index.html
- app.js
- style.css
- oops.jpg
- man_3728711.png
- package.json
- myDeploymentUrl.txt

Submissions will not be accepted after the deadline.

5. Grading and point deduction

Assignment 2.B is graded primarily manually after the submission deadline. The total score for Assignment 2.B is **70 points**. Submissions will open on Monday, February 23rd 2026.

Before manual grading, a small set of automatic checks will be performed in Gradescope. If any of the conditions listed below are violated, the assignment will receive a score of **0 points** and will not be graded further. If this happens, you will receive feedback from the system explaining the issue, and you are expected to fix it and submit again.

The table below gives an overview of how the assignment will be graded. While it covers the main grading criteria, it is not an exhaustive list, and minor adjustments may be made during grading.

Only submissions that pass the automatic checks will be graded manually.

Criteria	**Point deduction**
Required files and file names: All required files are present and correctly named.	Missing or incorrectly named required files: **-70 points (score reduced to 0; grading stops)**.
No extra files: Only the expected files are submitted.	Any extra/unexpected file(s) included: **-70 points (score reduced to 0; grading stops)**.
Restrictions on provided files: The CSS file must remain completely unchanged. The HTML file must remain unchanged.	Any modification to the CSS file, or any change to the HTML file other than adding the script reference: **-70 points (score reduced to 0; grading stops)**.
Load tasks from backend (8 points): Tasks are fetched from the local backend and displayed when the page loads.	Tasks not loaded, incorrectly fetched, or not displayed: up to **-8 points**.
Render task list correctly (8 points): Each task is rendered with the correct text and completion state.	Incorrect or incomplete task rendering: up to **-8 points**.
Update task completion status (8 points): Toggling a task checkbox updates the task status using the backend.	Missing or incorrect update behavior: up to **-8 points**.
Add new task (8 points): New tasks can be added using the input field, both by clicking the button and pressing Enter.	Missing or incorrect task creation behavior: up to **-8 points**.
Load notes from backend (8 points): The current notes are fetched from the backend and displayed in the text area.	Notes not loaded or displayed incorrectly: up to **-8 points**.
Save notes behavior (8 points): Notes can be edited and saved using the "Save changes" button, with correct enable/disable behavior.	Incorrect save or button behavior: up to **-8 points**.
Deployment (6 points): The frontend is deployed to a publicly accessible website and the deployment URL is correctly provided.	Missing, inaccessible, or incorrect deployment URL: up to **-6 points**.
Error handling (6 points): Backend errors are handled gracefully and logged to the console.	Errors not handled or causing application failure: up to **-6 points**.
JavaScript best practices (10 points): Appropriate use of modern JavaScript features, including `const`/`let`, arrow functions, `async`/`await`, and clear structure.	Significant deviation from required practices or poor code quality: up to **-10 points**.