Department of Information Engineering and
Computer Science
University of Trento, Italy

Web Architectures - A.Y. 2022/2023

Assignment 3

Authors:
De Menego Luca

Oct 29, 2022

Contents

1 Introduction

The third assignment involves the development of a web application implementing a simple *spreadsheet*.

The **backend**, based on `Java Servlets`, only handles one spreadsheet, so it will be shared among all users connecting to the application. The backend's core with the business logic needed to handle the spreadsheet is already provided to us, as two `Java classes` abstracting away the details and exposing few useful functions to interact with it. Hence, the amount of work we need to spend on the backend is quite small: focus should be put, instead, on the client side.

The **client part** is a single-page web application presenting a spreadsheet with a fixed number of columns and rows. Both the initial configuration and any user action on each cell must trigger an `HTTP request` via `AJAX` to the backend's *API*. The backend's response should be formatted in `JSON`. Each cell is defined by:

- an identifier, defined by the column and the row index (e.g. B2);
- a formula;
- a value.

The cells are not directly editable. When a user clicks on a cell, what should happen is the following:

- the cell's border is highlighted;
- a formula text field put on top of the spreadsheet gets focus;
- both the formula text field and the selected cell are populated with the cell's formula;
- any update on the formula text field is propagated to the cell;
- when the user types *"Enter"* or clicks outside the cell, the content should be automatically evaluated, and the user interface consequently updated.

Formulas should not be validated or evaluated client-side: this aspect is completely handled by the provided spreadsheets core backend.

Since there is one only spreadsheet shared among everyone accessing the application, we need to ensure users always see the most recent version of the spreadsheet. Cell updates must be propagated to all users via a pull approach and the `setInterval` *Javascript* function. In detail, the browser should send a request containing the current state timestamp that it has. The backend will answer with a new state only if its timestamp is more recent.

2 Solution

The project has been built with `Jakarta EE`, exploiting `Java Servlets` and `JSPs`. All client-side code has been written in Javascript, and the main components are exposed as `ES modules`.

2.1 Backend

2.1.1 Spreadsheets core

As explained in the introduction, a ready-to-use solution for spreadsheets management has been given to us as two Java classes easily integratable within the Servlets ecosystem. Only few changes have been made to the actual implementation, mainly to support all features developed client-side:

- a `toJson` function has been added into the `Cell` object, returning a `JSON` string containing the most important properties of the cell;
- getters for the number of rows, the columns and the current state of the spreadsheet have been added to the `SSEngine`;
- a `latestUpdateTimestamp` has been integrated as a property of the `SSEngine`, updated each time a cell is modified with `new Timestamp(System.currentTimeMillis());`
- thanks to a discussion posted on Moodle's forum, a bug has been fixed, that caused the system not to work properly after a circular dependency submission.

2.1.2 Java Servlets

As a good practice to follow when developing `JSPs`, the main `index.jsp` file has been put within the `WEB-INF` folder, to keep it private and only accessible through *Java Servlets*. Hence, the first implemented servlet is a `HomeServlet` which simply forwards the request to the mentioned *JSP*. The path of the servlet is `/home`, so this servlet has been added as a `<welcome-file></welcome-file>` in the `web.xml` file.

The other servlets are called by client-side scripts:

- `SpreadsheetsOptionsServlet`: it exposes information about the rows and the columns of the spreadsheet offered by the application.
- `SpreadsheetsStateServlet`: it exposes the current state of the spreadsheet. It also takes an optional parameter as input: a timestamp. The complete state is returned to the user only if the parameter is not given or if the timestamp the client has provided is not the most recent one. The timestamp must be given in *Unix Timestamp*, i.e. seconds since Jan 01 1970 (UTC). Otherwise, an error with status `400` is returned to the caller.
- `SpreadsheetsServlet`: the main servlet that needs to be called when modifying the spreadsheet. It takes as input the selected cell's ID and the new formula. The formula is then evaluated and the spreadsheet eventually updated. In case of circular dependencies, a `400` error is returned to the client. Otherwise, a *JSON* string is constructed containing all the affected cells with their new values.

2.2 Frontend

2.2.1 User Interface

Before constructing the user interface with *HTML+CSS* from scratch, the *Figma* tool has been used to design the web-application. The design file can be viewed at this link.
The actual spreadsheet is created as a simple *HTML* table in which each cell contains an *input* field that, however, is only populated programmatically when the *formula* field is updated. Before constructing the table, a request to `/spreadsheets-options` is performed to get the current number of rows and columns. This allows us to dynamically generate the table by exploiting *Javascript DOM manipulation*. The default number of rows and columns is 4, but thanks to this feature they can be changed server-side, and the user interface will adapt automatically (Figure 2). Each cell maintains information about its ID, formula and value using the `setAttribute` function. Hence, everything is stored in the actual HTML, and easily accessible from *Javascript* with `getAttribute`.
To change the style of the selected cell, event handlers have been exploited to add an `active` class to the element. More details about it can be found in Subsection 2.2.2.

2.2.2 Event Handlers

Most of the client-side business logic can be found within event handlers. In detail, we have:

- an event handler on each cell that is called on `focus-within` - i.e. when the cell or any of its descendants are focused - that:
  - if another cell was previously selected, evaluates it and updates the UI, by also removing the class `active`;
  - updates the currently selected cell, by adding to it the class `active`;
  - puts the cell's ID within an *HTML element* in the user interface, near the formula text field (Figure 3);
  - sets the cell's formula in the formula text field and in the cell;
  - gives focus to the formula text field.
- an event handler on the formula text field, called on `focus-out` - i.e. when it loses focus - that evaluates the previously selected cell and updates the UI accordingly.
- an event handler on the formula text field, called on `input` - i.e. when the user writes something on it - that updates the current cell's content. This only happens if a cell is currently selected. If no cell is selected and the user tries to write on the field, nothing will happen and the inserted text will be automatically discarded.
- an event handler on the formula text field, called on `keypress Enter` - i.e. when the user types *Enter* - that makes the formula text field lose focus. By losing focus, the previously mentioned event handler will be fired.

2.2.3 Get the spreadsheet's state

As explained in the introduction, the server only handles a single spreadsheet. Each user connecting to it should always be able to see and manage its latest version, and this should be implemented with a pull approach. After having created the spreadsheet table, a set of instructions is defined that, thanks to the *Javascript*'s `setInterval` function, is continuously run every `UPDATE_INTERVAL` milliseconds, where `UPDATE_INTERVAL` is a specific property of the main class exposing the spreadsheet, described in Subsection 2.2.5. These instructions simply call the `SpreadsheetsStateServlet` to get the spreadsheet's state and update its local version if necessary. The first time the client calls the Servlet, no timestamp will be passed, since the client does not have one. Afterwards the timestamp will be saved, and passed to the backend on every request. This enables:

- the server to answer with spreadsheet data only if the client's version actually differs from the backend's one;
- the client to perform updates on the local version of the spreadsheet only when necessary.

Since these instructions are continuously updating the user interface, it is vital for us to make sure that the client experience will not suffer from this. That is why in the provided solution the user has complete control over the currently selected cell, thanks to the development choice of not updating the cell the user is modifying. It may seem this could lead the application to an inconsistent state, but that's not the case: a cell's formula can never have circular dependencies, so any update on it will not depend on its previous value.

2.2.4 Connection with the Backend

To communicate with the backend, a class `SpreadsheetsService` has been crafted that exposes three static functions:

- `getSpreadsheetsOptions`: it fetches `/spreadsheets-options` to get the rows and columns of the spreadsheet.
- `getSpreadsheetsState`: it takes as input the timestamp and fetches `/spreadsheets-state` to get the updated state, if available.
- `evaluateCell`: it takes as input the currently selected cell, constructs the body of the `POST` request so that it conforms to the content type `x-www-form-urlencoded` (containing ID and formula) and sends the request to `/spreadsheets`.

All these functions return a promise to the response object, already parsed thanks to the `.json()` function.

2.2.5 Modularization

To keep the project's architecture organized and easy-to-maintain, a modularization approach has been used, exploiting *ECMAScript modules* and *HTML templates*.

The developed template, available in the file `unitn-spreadsheets-template.jsp`, provides the main structure of the spreadsheet and all its *CSS styles*, included from an external file. However, as stated in Subsection 2.2.1, the template doesn't directly provide the actual table representing the spreadsheet because it is dynamically generated using *Javascript*.

In the publicly accessible directory `scripts` there are two files:

- `spreadsheets-service.js`: it exports by default the class `SpreadsheetsService`, defined in Subsection 2.2.4.
- `unitn-spreadsheets.js`: it imports the `SpreadsheetsService` and it exports by default the class `UnitnSpreadsheets`, containing most of the client-side business logic.

`UnitnSpreadsheets` has a constructor that takes as input the parent HTML element in which the spreadsheet should be appended and, starting from the previously defined cloned template, it constructs the table based on the information offered by the *Servlets*, it adds all event listeners discussed in Subsection 2.2.2 and it invokes the `setInterval` function outlined in Subsection 2.2.3. The class has the following properties:

- `UPDATE_INTERVAL`: time frame, expressed in milliseconds, between each call to the `SpreadsheetsStateServlet`;
- `document`: cloned template in which we need to append the spreadsheet table;
- `timestamp`: latest timestamp referencing the spreadsheet's state, initialized with `0`;
- `rows` and `columns` of the spreadsheet;
- `currentCell`: the currently selected cell of the spreadsheet; it evaluates to `null` if no cell is selected.

Thanks to this modularization approach, the `index.jsp` file is quite minimal: it contains the template, a background image, a `div` with `id=app-container` that will become the parent of the spreadsheet component and a footer. In order to include the spreadsheet, a `<script type="module">` has been appended at the end of the `body`, that imports `UnitnSpreadsheets` and instantiates it by giving to its constructor the `app-container div`.

In this way, any *HTML* page can easily integrate the developed spreadsheet by simply importing the *Javascript module* and instantiating it, providing a parent container.

3 The application running

The main page with the default number of rows and columns can be seen in Figure 1. As requested, it offers a spreadsheet table with non-editable cells and an upper formula text field. If the number of rows and/or columns is modified in the backend, the user interface is consequently updated, as can be seen in Figure 2.

Figure 1: The application's main page, default number of rows and columns

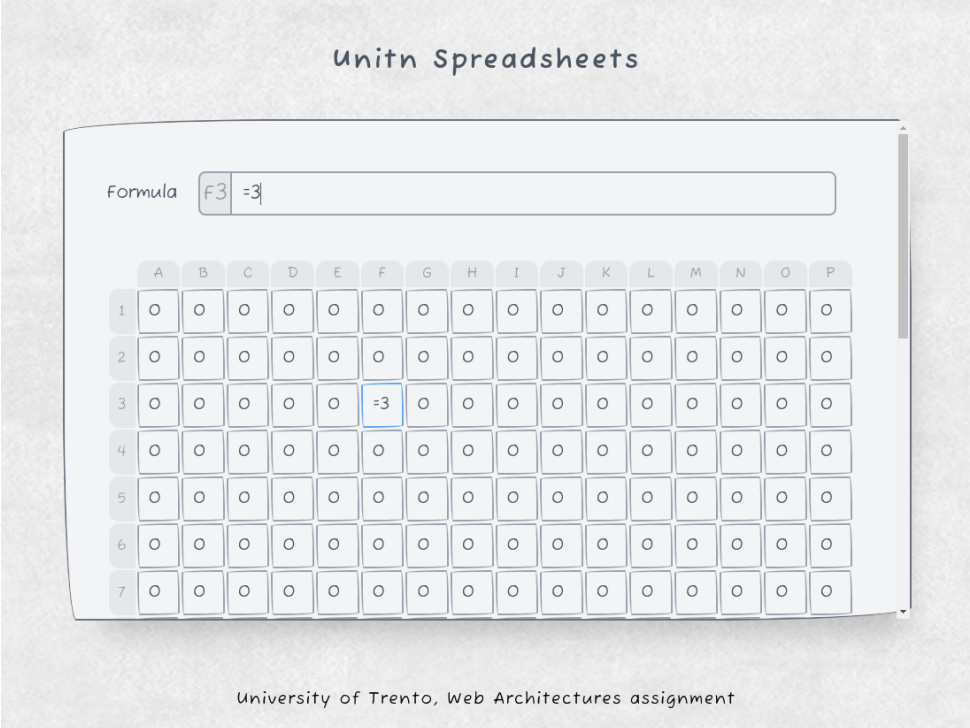

Figure 2: The application's main page, more rows and columns

When a user selects a cell, the cell is highlighted and the focus is brought to the formula text field, in which the user can start writing (Figure 3). After pressing *"Enter"* or clicking outside the cell or within another cell, the provided formula is evaluated with an *AJAX* request (Figure 4).

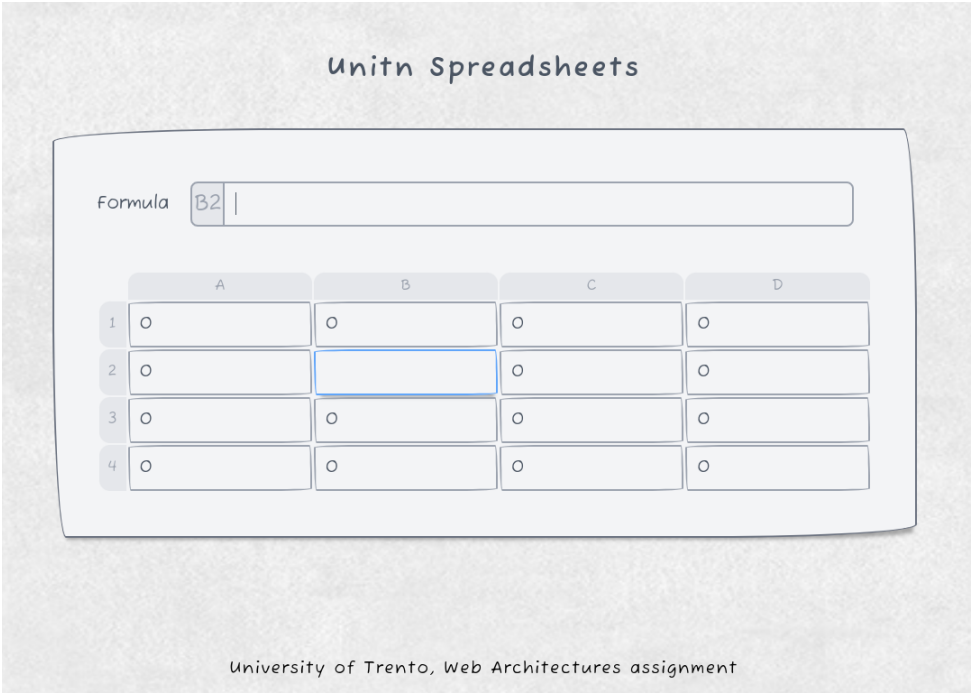

Figure 3: A user has selected the cell B2

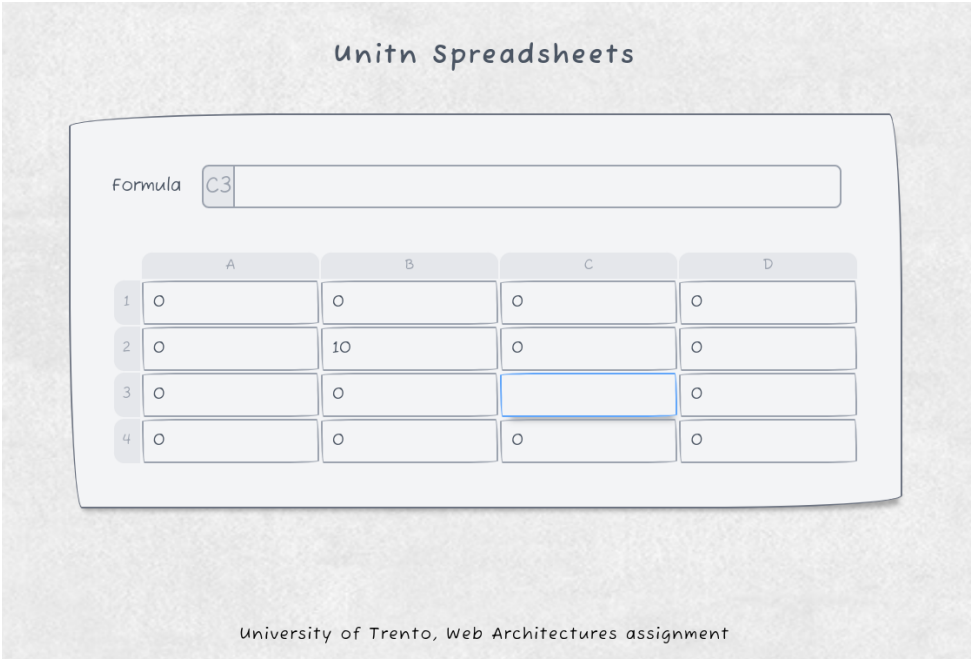

Figure 4: The user has clicked on another cell, so the cell B2 has been evaluated based on the provided formula

At this point, if the user clicks again on the evaluated cell, he will be able to see the provided formula, and update it if necessary (Figure 5).

Figure 5: A user has selected again the cell B2

When another user enters the application, the spreadsheet will automatically update every `UPDATE_INTERVAL` milliseconds (see Subsection 2.2.3 for more details), so the user will always see and perform updates on the latest version of the spreadsheet (Figure 6).

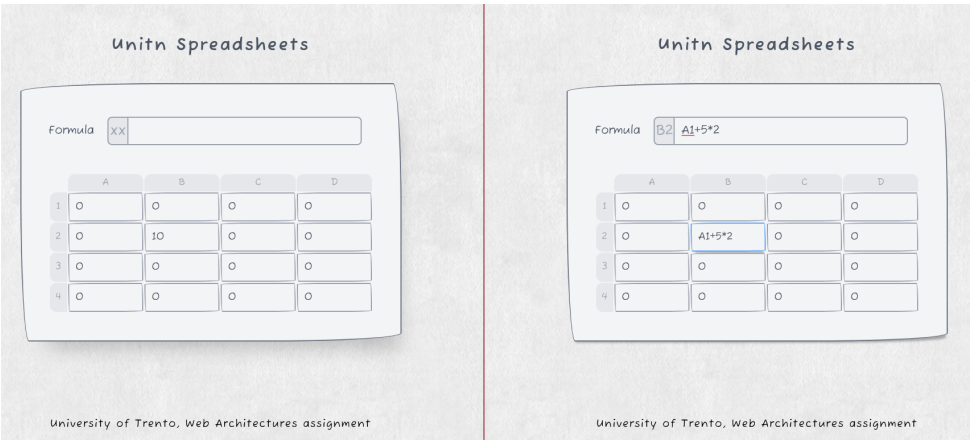

Figure 6: Another user entered the application, he sees the updated spreadsheet

In case of errors, the cell will simply be evaluated to `0`. If there is an error in the formula, it will be shown to the user when clicking on the corresponding cell (Figure 7).

Figure 7: Example of what happens when a formula has an error

4 Comments

The developed solution exports a self-contained class that creates the spreadsheet and appends it to the provided HTML element. Such an architecture can be defined modular, but probably a better alternative would have been to define a custom web component. This would allow us to integrate the spreadsheet into a page by using a custom tag like `<unitn-spreadsheet> </unitn-spreadsheet>`. However, due to the fact that this approach relies on the *shadow DOM*, which is completely separated from the actual page's DOM interface, some event listeners did not work out as expected: for instance, when clicking outside the component to make a cell lose focus, the event would not reach the developed listeners. For this reason, I decided not to spend too much time solving these problems and I preferred the reported solution, which is quite easier to manage.