Web Architectures
Assignment 3

Antonio Pio Padalino

November 6, 2022

Contents

1 Introduction **2**
1.1 The project goal . . . 2
1.2 The project structure . . . 2

2 Methods **2**
2.1 The spreadsheet . . . 3
2.2 Modifying cells . . . 4
2.3 Checking for updates . . . 4
2.4 The server logic . . . 5

3 Comments **6**
3.1 Modifications to `SSEngine` . . . 6
3.2 Possible improvements . . . 6

1 Introduction

1.1 The project goal

The assignment concerns the development of a web application that implements a simple spreadsheet. Client-side, a *Single Page Application* allows the user to make modifications to the spreadsheet. Any action that modifies a cell triggers an HTTP request to which the server in turn responds computing the actual variations, updating all the cells which have been modified in the view. Data about the updated cells content (their value and formula) are passed back to the client using JSON format. Other users may connect to the web application as well, obtaining the most updated view of the table. They can also do further modification to the sheet, and edits made by anyone are propagated to all connected users.

1.2 The project structure

The project makes use of a number of Java classes.

- `Cell` and `SSEngine` deal with the logic behind the modifications made to the spreadsheet;
- `SheetState` and `CellState` are used to represent the current state of the spreadsheet, and are structured so that using GSON[3] a `sheetState` object can be turn into a JSON and passed to the front-end, but more on that will come later;
- `SheetServlet` is the actual servlet that implements part of the logic behind the functioning of the spreadsheet, thanks to its methods `doPost()`, `doGet()` and `initState()`;

The logic of the spreadsheet is also actuated thanks to `spreadsheet.jsp`, which contains JavaScript code both for interacting with the sheet and communicating with the server to save new data and retrieve updated states.

More on each component and their specific functioning will be discussed in the Methods section.

2 Methods

Before discussing of each component, here is a brief overview of the functioning of the web app.

The user can interact with the spreadsheet through `spreadsheet.jsp`: when the editing is done, they can save the changes by pressing Enter or clicking away: in this way `saveCell()` is triggered, sending a POST[4] request to the server with the id of the cell edited by the user, the formula typed and the timestamp of the modifications. The server in turn uses `SSEngine` and in particular the method `engine.modifyCell()` to apply the changes. Such method returns a `Set` of `Cell` objects, containing the cells which were modified (be it directly or in cascade). Finally, the representation of the state of the spreadsheet `sheetState` - declared at the beginning of the Servlet code - is updated with the modified cells.

On the other hand, the role of `checkUpdates()` is to check if a newer version of `sheetState` is available, pull it in JSON format and update the view of the spreadsheet. To accomplish this in a push fashion, `checkUpdates()` is executed every half a second. It is also executed right after the function `saveCells()` sends an HTTP request, to immediately apply visually the newest changes made by the user. The timestamp of the client's current sheet state is the only parameter passed in the request made by `checkUpdates()`, since we don't have any content to transmit. The server compare the timestamp sent with that of the `sheetState` and - if outdated - pass it as a JSON back to the `.jsp`. The timestamp of the last state of the spreadsheet on client-side is stored at the beginning of file in a paragraph (which also serves as a way to display the last update time).

Summing up, `saveCells()` communicates with server to execute and write changes to `sheetState`, the server-side representation of the state of the table, while `checkUpdates()` checks if the client-side representation is outdated, and if so pulls the last server sheet state and updates the view.

2.1 The spreadsheet

The key part of the web application is the possibility to interact with a spreadsheet presented to the user. To this account, `spreadsheet.jsp` deals with the presentation part of the application, while the JavaScript code which follows deals with interaction and the editing / update of the sheet.

Presentation The overall `spreadsheet.jsp` page is pretty simple. It displays a simple title and a line with the time of the last update at the top of the page, followed by a form and the actual table. In particular, the table HTML is not static, and some java scriplets have been used to generate a table featuring the exact number of columns and rows specified in the `SSEngine`, to address the possibility the user might want to change the default values. This was also possible thanks to some slight modifications to the code of `SSEngine`, which will be discussed in the Comments section. Some style options have also been specified at the beginning of the `spreadsheet.jsp` to make the table look better even with no values inside.

Interaction The JavaScript code inside `spreadsheet.jsp` accounts for the interaction with the spreadsheet. The function `editCell()` is called whenever a particular cell is clicked, passing the `id` of said cell. The function allows for switching the focus to the form above the sheet, changing the background color of the selected cell and adding a soft frame to it. Moreover, the function is also responsible for displaying a `=` symbol before the actual content of the cell and for displaying a formula instead of the actual value in case any formula for that id has been previously inserted. This also allows to achieve a result that is most similar to a traditional spreadsheet application such as Microsoft Excel or Google Spreadsheets. Two other snippets, duly commented in the code, are responsible for

- avoiding page refresh[1] after submitting the form on top of the sheet
  (since we only want to trigger `saveCell()`);
- displaying the content that is being written by the user simultaneously in both the specific cell and form (again, in a Google Spreadsheets fashion).

For writing the actual JavaScript code, tutorials on GeeksforGeeks.org[5] have been used as support.

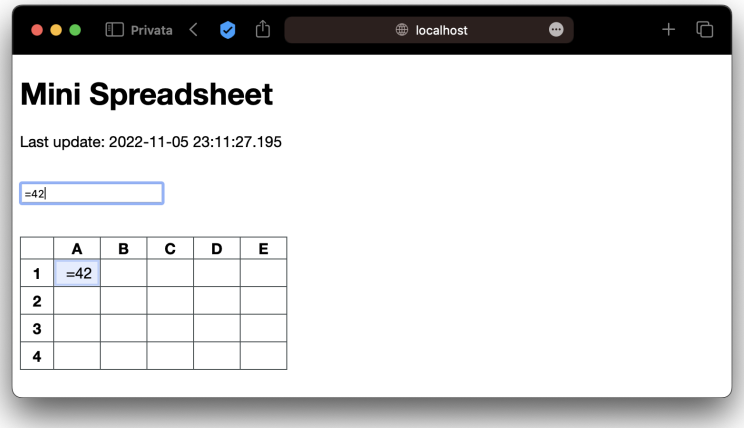

Figure 1: An example of interaction with the spreadsheet.

2.2 Modifying cells

The JavaScript function `saveCell()` is called both when the form above the sheet is submitted or when, after inserting a cell content, a mouse click happens away from any cell or the form. The function prepare and sends an POST request, containing `cell`, `content` and `timestamp` parameters.

- `cell` simply contains the `id` of the edited cell ;
- `content` contains the user-inserted formula, duly encoded[6], of the cell ;
- `timestamp` passes the time of the current modification.

The function does not do anything special after the requests, since its aim is just to transmit the information typed by the user to the server and to update the server-side representation of the spreadsheet state. The only action performed then is to check for updates with `checkUpdates()` (we are sure there will be updates to the sheet state, since the user has just made some modification themselves).

Yet, if the user leaves the cell blank before pressing Enter or clicking away (meaning the cell content is just =), then no request to the server is sent, to avoid being it evaluated to zero, and the cell content is reverted back to an empty string.

In any case, at the end of the function the cell highlight disappears, returning as default, and the focus is moved away from the form.

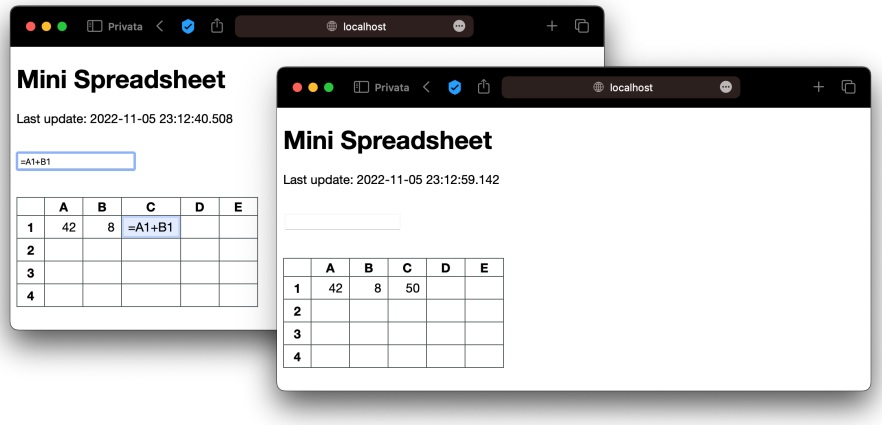

Figure 2: A formula is inserted into cell C1 of the spreadsheet: when Enter is pressed, the formula is a evaluated to a number, and both the focus on the form and the blue cell background disappear.

2.3 Checking for updates

Undoubtedly, the core JS function used here is `checkUpdates()`. It sends a POST to the server, this time only specifying the timestamp of the last state of the client-side spreadsheet (contained in the *"last updated"* line on top of the page).

After receiving back the JSON representation of the most recent sheet state (only if the current one is outdated, of course), `checkUpdates()` proceeds to the update of all cells, cycling through the element of `sheetState` and modifying the actual view. As already mentioned, the application checks for updates every half a second, simulating a push service.

This constant check for updates also makes possible for other users to immediately see all modifications made by others since startup, when the spreadsheet web app is launched. This is also possible since `checkUpdates()` pulls the state of the sheet that is located server-side, unique among different sessions and users.

2.4 The server logic

The functioning of the server has already been hinted when talking of the ways the client communicates the changes made by the user, but here it will be briefly recapped.

doGet() it simply accounts for the initialization of `sheetState` (the server-side representation of the state of the spreadsheet) through `initState()` and for the early displaying of `spreadsheet.js` via the usual request dispatcher.

initState() initializes the state of the spreadsheet, with the an initial timestamp and a map of all cells with value and formula initialized to empty strings.

doPost() deals with the request sent by the JavaScript code inside `spreadsheet.jsp`. Specifically, it collects and parses all parameters received and then adopts one of two possible behaviors:

if the only parameter received is `timestamp`, hence the request has been sent by the JavaScript function `checkUpdates()`, the server compares the timestamp received with `lastTimestmap` and if the client-side representation is outdated the `sheetState` is converted to JSON and passed to the client.

if the the request also contains `cell` and `content` parameters, hence the request has been sent by `saveCells()`, the server uses the `engine.modifyCells()` method to apply changes and get a map of modified cells, then it cycles through such map to update the `sheetState` representation. Finally, the `lastTimestamp` variable is updated with the received timestamp. This section of the code also accounts for the possibility in which illegal formulas were used. The engine is perfectly capable of parsing illegal formulas (which are evaluated to 0), yet for this implementation of the web app the server makes use of the map of modified cells returned by `.modifyCells()` which gives a `null` if illegal formulas are detected. Therefore, if the return content of the method is null, the `CellState` is set with value 0 and a `[!]` in the formula, which will show up later in the view to alert the user.

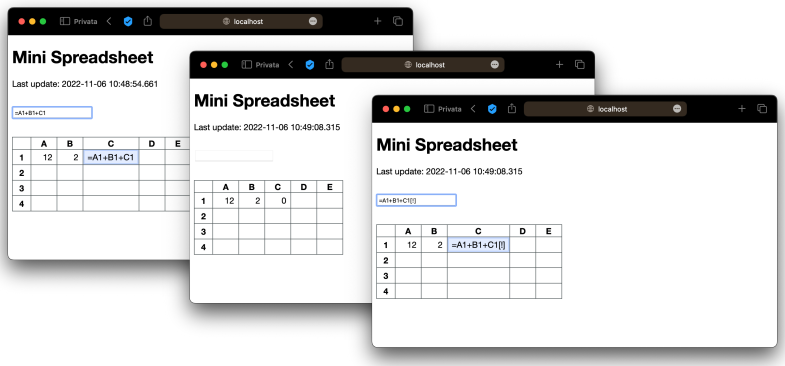

Figure 3: When the user types an illegal formula (in this example, one which causes a circular dependency) it is evaluated to zero on confirmation, but on re-inspection it shows a `[!]` to alert the user of the error.

3 Comments

3.1 Modifications to `SSEngine`

The code accounting for the logic behind the actual operations on the spreadsheet is given and do not require any heavy modification for this implementation of the web app, and for this reason it was not discussed thoroughly in this report. However, slight modifications were made to the code of both `SSEngine` and `Cell`, which will be discussed here shortly.

Retrieving Rows and Columns Two simple methods `getNRows` and `getColumns` were implemented to get the number of rows and the names of the columns specified at the beginning of the `SSEngine` file. This was mainly done to be able to create the spreadsheet table in a more concise way using a for cycle, so that the view can adapt to the number specified "dimensions" of the sheet.

Bug on circular dependencies A bug in `SSEngine` caused the system to not behave properly after a circular dependency is detected. Said bug was discovered and shared on the Moodle forum of the course[2], and the workaround described there was applied to prevent unexpected behavior.

3.2 Possible improvements

Speaking of possible ways to improve the web application, a couple of ideas regards the possibility of changing the dimensions of the sheet, and the possibility to save the state of the sheet.

Change spreadsheet size Another web page could be implemented as some sort of configuration page for the spreadsheet, to give the user the possibility to choose a desired number of rows and columns. These parameters could be treated as parameters of `SSEngine` and used to create an engine with the desired features. The rest of the code would simply remain the same, since the table in the `spreadsheet.jsp` is created dynamically starting from the dimensions specified engine, and the ids for the cells are uniquely tied to the letters of the columns and the row numbers.

Save and load the state One further improvement would be storing the current state of the spreadsheet so that it can be loaded after redeployment. This could be achieved either by serializing the `sheetState` object or by converting it to a JSON file and writing it to some directory. This last approach has been originally tested, but upon better understanding of the assignment a more simple solution has been preferred, since no need for saving the state of the spreadsheet was mentioned.

References

[1] developer.mozilla.org. *Event.preventDefault() documentation.* `https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault`.

[2] didatticaonline.unitn.it. `https://didatticaonline.unitn.it/dol/mod/forum/discuss.php?d=290067`.

[3] jenkov.com. *GSON tutorial.* `https://jenkov.com/tutorials/java-json/gson.html`.

[4] stackoverflow.com. *Send POST data using XHttpRequest.* `https://stackoverflow.com/questions/9713058/send-post-data-using-xmlhttprequest`.

[5] www.geeksforgeeks.org. *JavaScript tutorial.* `https://www.geeksforgeeks.org/javascript/`.

[6] www.w3schools.com. *HTML URL Encoding Reference.* `https://www.w3schools.com/tags/ref_urlencode.ASP`.