Assignment 3

Web Architectures - 06/11/2022

Sabin Andone

Part 1

1.1 Introduction

For this third assignment it has been necessary to develop a web application that behaves like a simple spreadsheet. In this case, the client interacts with only one page that contains an editable field and a matrix of non editable cells. When the user clicks on a cell its borders are highlighted and also the text field above the matrix shows the formula of the selected cell. The user can edit this text field and every key typed is also reflected in the selected cell which shows the formula instead of its value. In order to submit the formula to the server the user must either:

- type the Enter key
- click elsewhere, outside the text field

The submition of the formula triggers an HTTP request. The server receives the request, evaluates the formula and responds by returning all the cells which have been affected by the modifications in JSON format. The browser then updates the cells in the view (without reloading the page).

Regarding the client side, a cell has the following properties:

- An identifier, composed by an alphabetic char which identifies the column and a number which identifies the row (e.g A1)
- A formula
- A value

A formula, can be:

- A string that is parseable as a number
- A string that starts with '=' and contains a reference
- A string that starts with '=' and contains numbers and or references separated by operators (+-*/)

Formulas with syntax error or circular dependencies evaluate with 0.

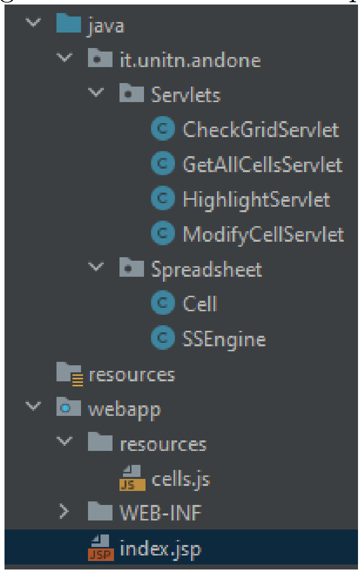
This webapp must be contained in a single page and it is not allowed to refresh that page, thus it has been necessary to use Javascript and AJAX.

Moreover, an usable logic on server side which implements the business logic has been provided by the professor. This means that the part of the code that takes the formula and evaluates it has been already done. In other words, it has been necessary to implement the client side of the webapp and to complete the server side in a such way that it can handle the requests sent by the client using the provided engine.

1.2 Solution

The solution for this assignment is a webapp that is structured as follows:

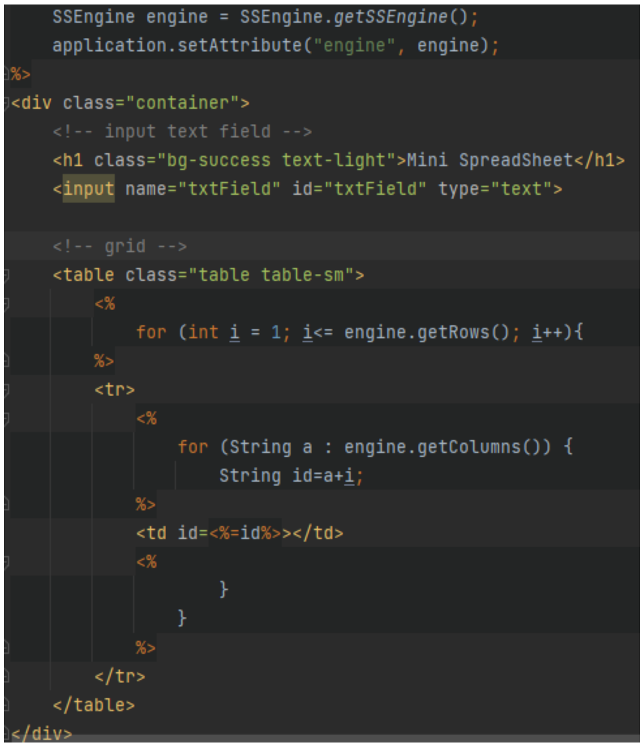
Figure 1.1: Structure of webapp.



```
java
  it.unitn.andone
    Servlets
      CheckGridServlet
      GetAllCellsServlet
      HighlightServlet
      ModifyCellServlet
    Spreadsheet
      Cell
      SSEngine
resources
webapp
  resources
    cells.js
  WEB-INF
  index.jsp
```

Regarding the client side, since this solution must be contained in a single page, there is only the index.jsp page which contains the text field and the grid with all the cells [Figure 1.2], together with some functions written using jQuery (the methods located in index.jsp are all functions that are triggered when the user does a certain action). Inside cells.js there are other methods that generally come in support to the methods inside the index.jsp file. In cells.js there is also defined the Cell class which is used for defining the current cell in the session (i.e. the one selected by an user in a certain moment).

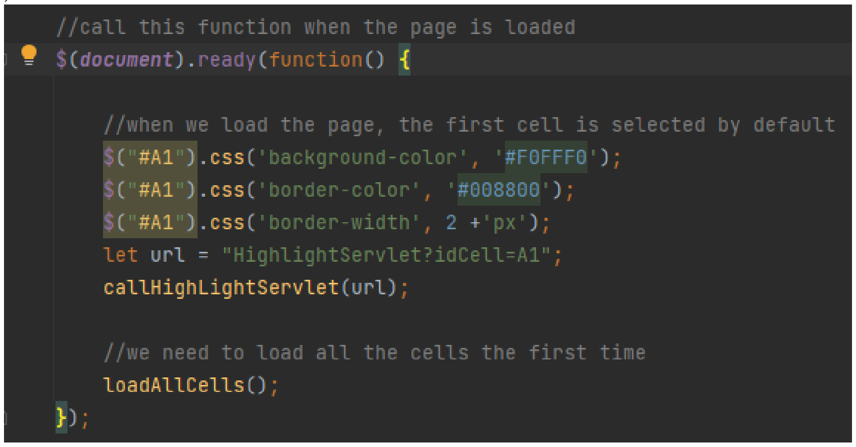
Figure 1.2: The body of index.jsp. The engine is initiated and saved inside the servletContext (with application.setAttribute() method)



```
    SSEngine engine = SSEngine.getSSEngine();
    application.setAttribute("engine", engine);
%>
<div class="container">
    <!-- input text field -->
    <h1 class="bg-success text-light">Mini SpreadSheet</h1>
    <input name="txtField" id="txtField" type="text">

    <!-- grid -->
    <table class="table table-sm">
        <%
            for (int i = 1; i<= engine.getRows(); i++){
        %>
        <tr>
            <%
                for (String a : engine.getColumns()) {
                    String id=a+i;
            %>
            <td id=<%=id%>></td>
            <%
                    }
                }
            %>
        </tr>
    </table>
</div>
```

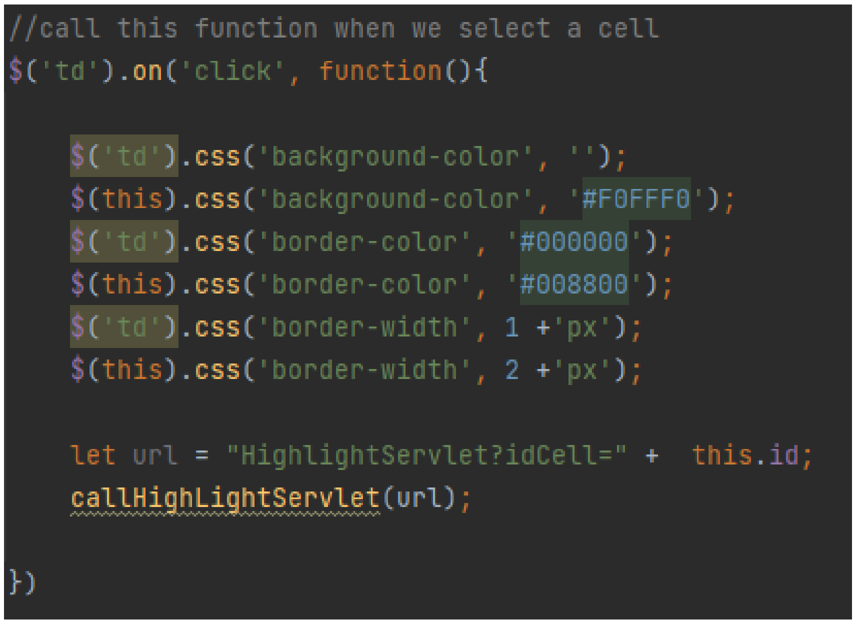
The Javascript functions inside index.jsp are all functions that are triggered when the user does a certain action on cells or on the text field. jQuery has been used in this case. The functions are the following:

Figure 1.3: This function is triggered when the page is loaded. The first thing it does is changing the css style of the first cell (which is always the one with id A1) in such a way is highlighted in the same way as cells are highlighted when clicked on them. After that we call the callHighLightServlet() method which wants a url as parameter. The url is passed in input in this case since in this case the url is always fixed with idCell=A1 while usually the idCell in the url is the id of the selected cell [see Figure 1.4 and Figure 1.7]. Then we load all the cells with loadAllCells() function. This function is usefull also when we close the page and load it again, since it loads all the cells from the last state of the engine. Same thing when another user loads the page (see part 2 for this last statement)



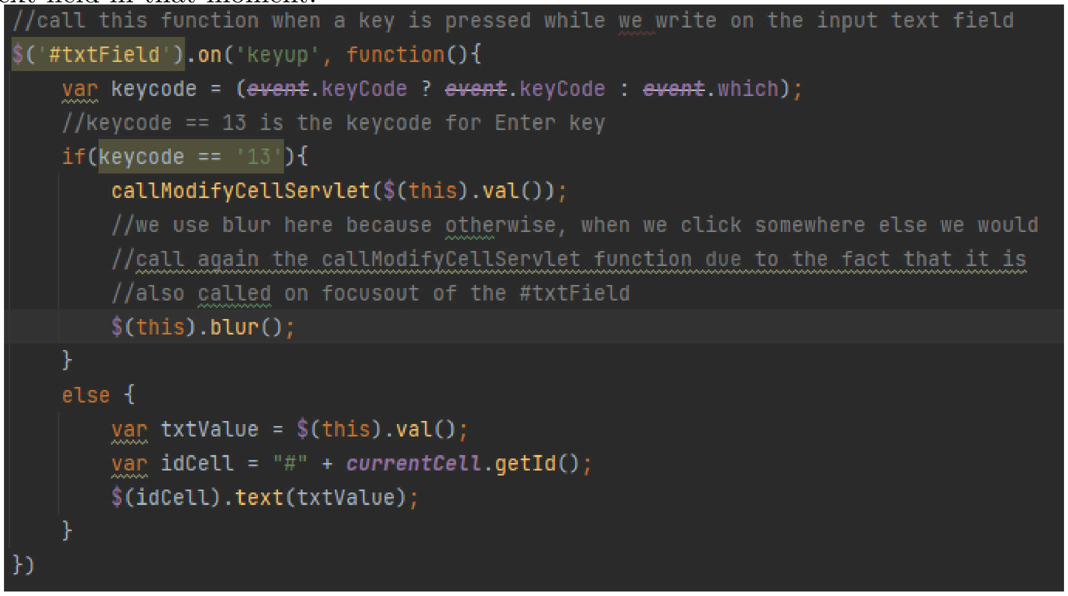
```
//call this function when the page is loaded
$(document).ready(function() {

    //when we load the page, the first cell is selected by default
    $("#A1").css('background-color', '#F0FFF0');
    $("#A1").css('border-color', '#008800');
    $("#A1").css('border-width', 2 +'px');
    let url = "HighlightServlet?idCell=A1";
    callHighLightServlet(url);

    //we need to load all the cells the first time
    loadAllCells();
});
```

Figure 1.4: This function is triggered when the user clicks on a cell. In this case we set the css of the selected cell in the same way defined in Figure 1.3 whereas we set the css of all other cells back to default values. The callHighLightServlet() is then called.



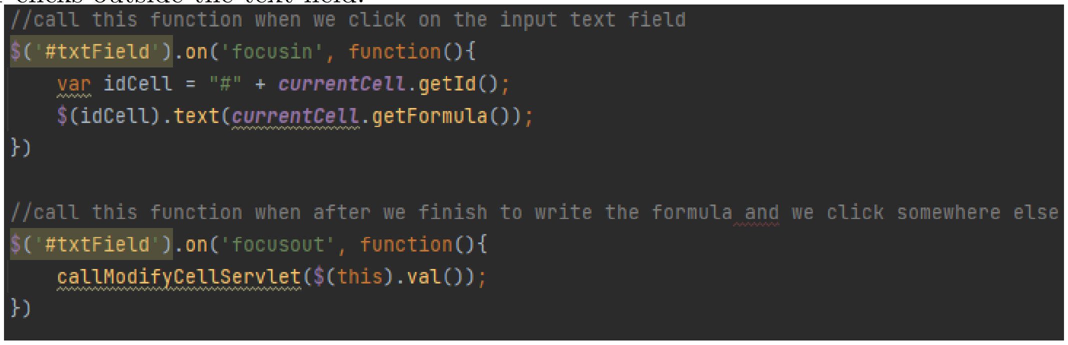
```
//call this function when we select a cell
$('td').on('click', function(){

    $('td').css('background-color', '');
    $(this).css('background-color', '#F0FFF0');
    $('td').css('border-color', '#000000');
    $(this).css('border-color', '#008800');
    $('td').css('border-width', 1 +'px');
    $(this).css('border-width', 2 +'px');

    let url = "HighlightServlet?idCell=" +  this.id;
    callHighLightServlet(url);

})
```

Figure 1.5: Everytime the user presses on a key, this function is triggered. First we check if the user pressed on the Enter key by checking if the keycode is equal to 13 (the Enter key code). In case of Enter key, we call the callModifyCellServlet() since it means that the user has submitted a formula. We pass the value of the text field as input. After the modification we call $(this).blur() to make the text field lose the focus. If, instead, the user pressed on any other key than the Enter key, then we just set the text content of the current cell to the value of the text field in that moment.



```
//call this function when a key is pressed while we write on the input text field
$('#txtField').on('keyup', function(){
    var keycode = (~~event~~.keyCode ? ~~event~~.keyCode : ~~event~~.which);
    //keycode == 13 is the keycode for Enter key
    if(keycode == '13'){
        callModifyCellServlet($(this).val());
        //we use blur here because otherwise, when we click somewhere else we would
        //call again the callModifyCellServlet function due to the fact that it is
        //also called on focusout of the #txtField
        $(this).blur();
    }
    else {
        var txtValue = $(this).val();
        var idCell = "#" + currentCell.getId();
        $(idCell).text(txtValue);
    }
})
```

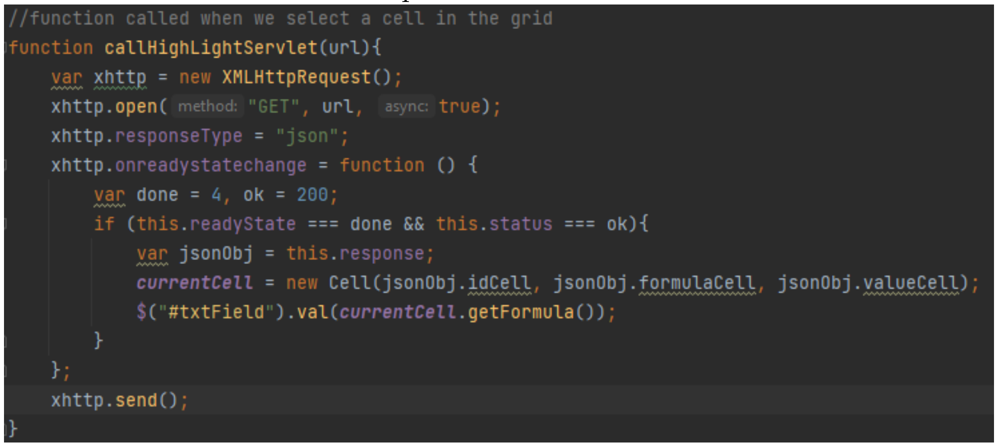
Figure 1.6: These are the two functions that are triggered when the user respectively clicks on the text field and clicks outside it. The first function makes in such way that the content of the current cell is equal to its formula (since we are entering the editing mode). The second method calls the callModifyCellServlet() method when the user clicks outside the text field.



```
//call this function when we click on the input text field
$('#txtField').on('focusin', function(){
    var idCell = "#" + currentCell.getId();
    $(idCell).text(currentCell.getFormula());
})

//call this function when after we finish to write the formula and we click somewhere else
$('#txtField').on('focusout', function(){
    callModifyCellServlet($(this).val());
})
```

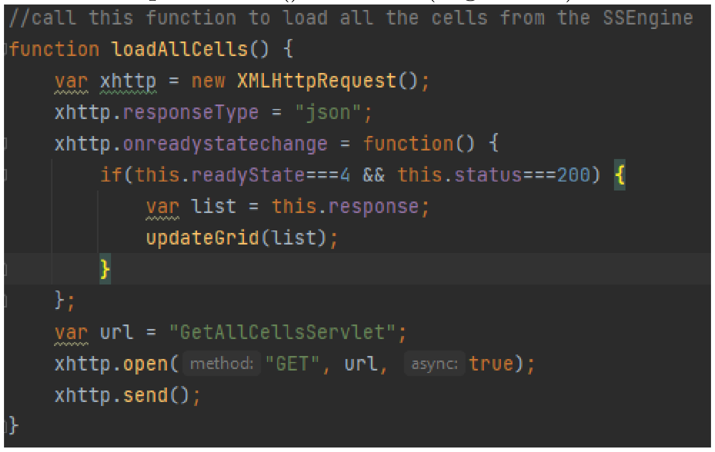
Inside the cells.js file there are all the other functions used for this assignment. It has been decided to separate all the functions in two groups since some functions from the index.jsp file use the same code more than once, thus the repeated code snippets have been implemented inside functions. All these functions are located inside the cells.js file to avoid having too much javascript code inside the jsp page. These functions are the following:

Figure 1.7: This function does an AJAX request to the HighlightServlet and passes the id of the selected cell. After the servlet sends back the response, the function creates the new currentCell with the values sent from the servlet and sets the value of the text field equal to the formula of that cell.



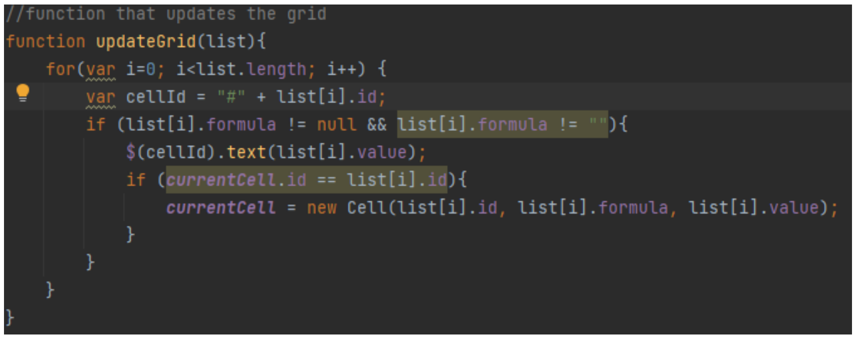
```
//function called when we select a cell in the grid
function callHighLightServlet(url){
    var xhttp = new XMLHttpRequest();
    xhttp.open( method: "GET", url,  async: true);
    xhttp.responseType = "json";
    xhttp.onreadystatechange = function () {
        var done = 4, ok = 200;
        if (this.readyState === done && this.status === ok){
            var jsonObj = this.response;
            currentCell = new Cell(jsonObj.idCell, jsonObj.formulaCell, jsonObj.valueCell);
            $("#txtField").val(currentCell.getFormula());
        }
    };
    xhttp.send();
}
```

Figure 1.8: This function is called when the page is loaded (see Figure 1.3). It does an AJAX request to the GetAllCellsServlet which sends as response the list of all cells in the engine. Such list is put in a variable (named "list") and used as parameter for the updateGrid() function (Figure 1.9)



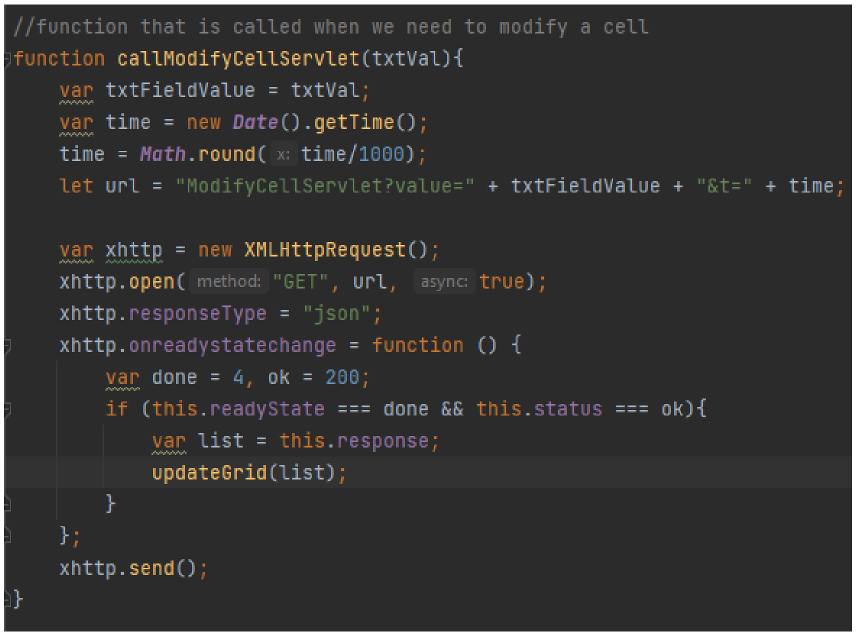
```
//call this function to load all the cells from the SSEngine
function loadAllCells() {
    var xhttp = new XMLHttpRequest();
    xhttp.responseType = "json";
    xhttp.onreadystatechange = function() {
        if(this.readyState===4 && this.status===200) {
            var list = this.response;
            updateGrid(list);
        }
    };
    var url = "GetAllCellsServlet";
    xhttp.open( method: "GET", url,  async: true);
    xhttp.send();
}
```

Figure 1.9: This function takes the list and iterates over all its elements (cells in this case). For every cell, it takes its id and then checks if its formula are not null. In the case it is not null we assign to the cell its value inside the grid.



```
//function that updates the grid
function updateGrid(list){
    for(var i=0; i<list.length; i++) {
        var cellId = "#" + list[i].id;
        if (list[i].formula != null && list[i].formula != ""){
            $(cellId).text(list[i].value);
            if (currentCell.id == list[i].id){
                currentCell = new Cell(list[i].id, list[i].formula, list[i].value);
            }
        }
    }
}
```

Figure 1.10: This is the function which interacts with ModifyCellServlet. First we define the time in which we have called the function in that moment and divide it by 1000 so we can consider it in seconds and not milliseconds. Then the function sends an AJAX request to the ModifyCellServlet passing as parameters the time and the value of the text field (which contains the formula to submit). The response in this case is the list of all cells which have been modified by the submition of the formula. In order to update the view we call again the updateGrid() function.



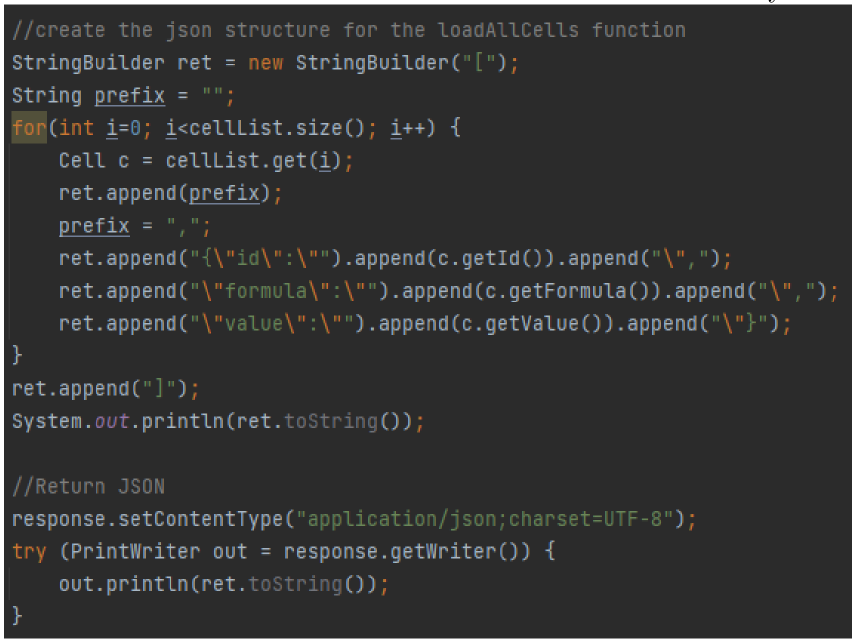
```
//function that is called when we need to modify a cell
function callModifyCellServlet(txtVal){
    var txtFieldValue = txtVal;
    var time = new Date().getTime();
    time = Math.round( x: time/1000);
    let url = "ModifyCellServlet?value=" + txtFieldValue + "&t=" + time;

    var xhttp = new XMLHttpRequest();
    xhttp.open( method: "GET", url,  async: true);
    xhttp.responseType = "json";
    xhttp.onreadystatechange = function () {
        var done = 4, ok = 200;
        if (this.readyState === done && this.status === ok){
            var list = this.response;
            updateGrid(list);
        }
    };
    xhttp.send();
}
```

The engine provided is included inside the it.unitn.andone.Spreadsheet package and includes two java classes: Cell and SSEngine. Such engine is initiated with the command SSEngine engine=SSEngine.getSSEngine() and lives inside the server side.

Inside the it.unitn.andone.Servlets package are located the servlets developed to realize this assignment. In summary:

- CheckGridServlet is a servlet that checks if any modifications to the engine happened or not (see Part 2).
- GetAllCellsSerlvlet is a servlet that retrieves all the cells from the engine and passes them to the client in a JSON format (see figure 1.11). It is called when the page is loaded and also when it is necessary to propagate modifications to other users.

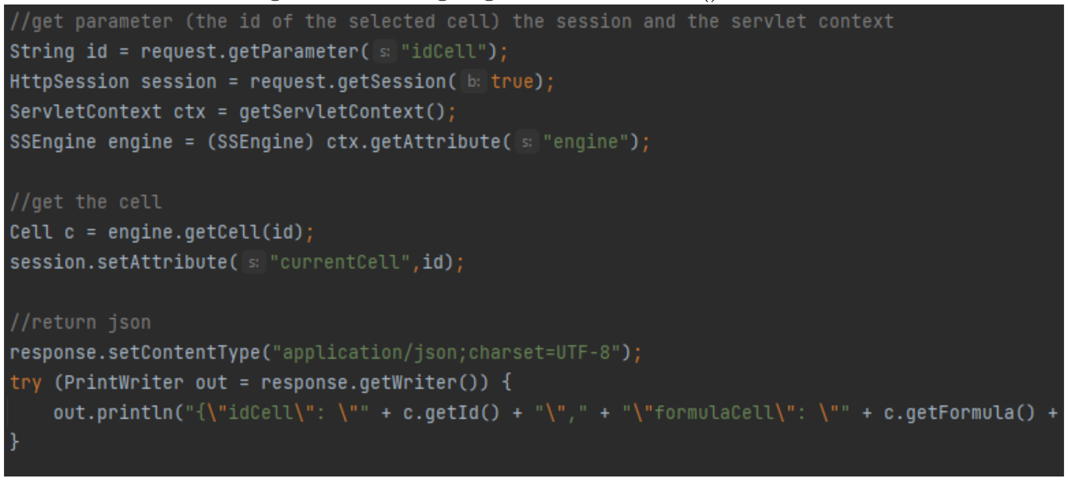
Figure 1.11: The JSON structure for GetAllCellServlet and ModifyCellServlet.



```java
//create the json structure for the loadAllCells function
StringBuilder ret = new StringBuilder("[");
String prefix = "";
for(int i=0; i<cellList.size(); i++) {
    Cell c = cellList.get(i);
    ret.append(prefix);
    prefix = ",";
    ret.append("{\"id\":\"").append(c.getId()).append("\",");
    ret.append("\"formula\":\"").append(c.getFormula()).append("\",");
    ret.append("\"value\":\"").append(c.getValue()).append("\"}");
}
ret.append("]");
System.out.println(ret.toString());

//Return JSON
response.setContentType("application/json;charset=UTF-8");
try (PrintWriter out = response.getWriter()) {
    out.println(ret.toString());
}
```

- HighlightServlet is a servlet used to set the CurrentCell in the session. It is called the first time when the page is loaded and then it is called everytime the user selects a cell. In this case it gets as parameter the id of the selected cell and sets it to the currentCell attribute of the session. It returns in output the id, formula and value of the cell.

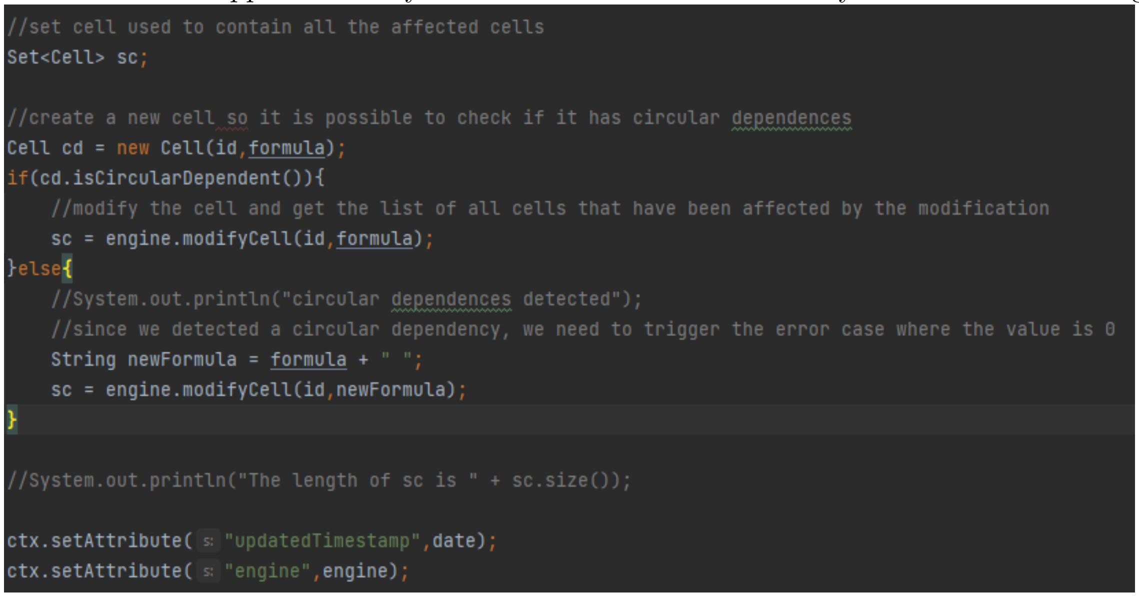
Figure 1.12: HighlightServlet doGet() method



```java
//get parameter (the id of the selected cell) the session and the servlet context
String id = request.getParameter( s: "idCell");
HttpSession session = request.getSession( b: true);
ServletContext ctx = getServletContext();
SSEngine engine = (SSEngine) ctx.getAttribute( s: "engine");

//get the cell
Cell c = engine.getCell(id);
session.setAttribute( s: "currentCell",id);

//return json
response.setContentType("application/json;charset=UTF-8");
try (PrintWriter out = response.getWriter()) {
    out.println("{\"idCell\": \"" + c.getId() + "\"," + "\"formulaCell\": \"" + c.getFormula() +
}
```

- ModifyCellServlet is the servlet which interacts with the engine by modifying the cell with the formula given as parameter by the client. It gets in input the time and the formula, whereas it gets the engine and the id of the cell to modify respectively from the ServletContext and from the session. Then it modifies the cell (see Figure 1.13) and after that it takes the content of the list of affected cells to realize the JSON structure (Figure 1.11) and send it in output.

Figure 1.13: Code snippet of ModifyCellServlet which calls the modifyCell method of the engine



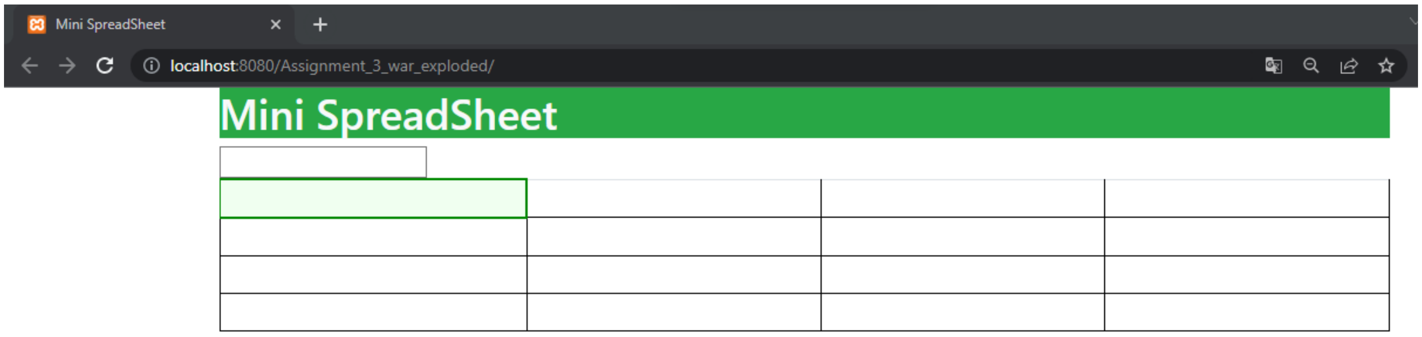
```
//set cell used to contain all the affected cells
Set<Cell> sc;

//create a new cell so it is possible to check if it has circular dependences
Cell cd = new Cell(id,formula);
if(cd.isCircularDependent()){
    //modify the cell and get the list of all cells that have been affected by the modification
    sc = engine.modifyCell(id,formula);
}else{
    //System.out.println("circular dependences detected");
    //since we detected a circular dependency, we need to trigger the error case where the value is 0
    String newFormula = formula + " ";
    sc = engine.modifyCell(id,newFormula);
}

//System.out.println("The length of sc is " + sc.size());

ctx.setAttribute( s: "updatedTimestamp",date);
ctx.setAttribute( s: "engine",engine);
```

Note: some modifications to the engine have been made in order to resolve some issues revealed during the implementation of this solution. Some of them are just added functions such as getter methods but the most important ones have been discussed in the moodle forum since a bug regarding circular dependecies has been found. Thus the following modifications have been made:

- replace inside ”if (! theCell.checkCircularDependencies(id)) {” the line ”theCell.formula = clone.formula;” to ”theCell.setFormula(clone.formula);”
- inside Cell.parseFormula() add this two lines of code after ”if (!formula.startsWith(”=”)) {” :
  - operands.clear();
  - operators.clear();

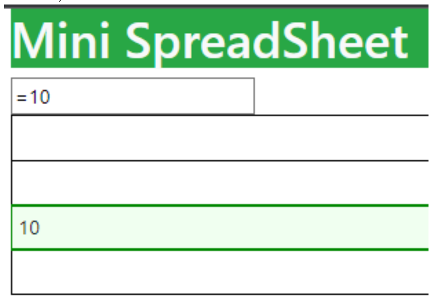
1.3 The webapp running

When the user loads the page for the first time, if it is the first user then the page shown on the browser looks like following:



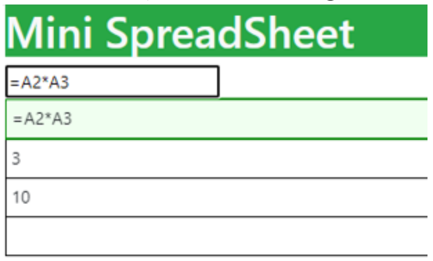
In this case the first cell is the cell that is currently selected, which means that if the user wants to put a formula, that formula would be evaluated on that cell. If the user wants to select another cell, s/he has just to click on the cell s/he wants to put the formula.

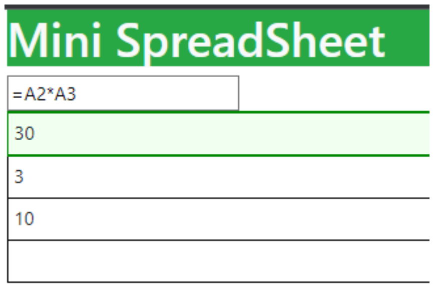
Figure 1.14: In this case the user clicks on cell A3, thus that cell becomes the current cell of the session. Note that the cell inside the grid shows its value, while the text field shows its formula.



To put a formula, the user must click on the text field and put the formula s/he wants to evaluate. While the user types on the text field the formula is reflected on the selected cell, until the user submits the formula to the server, in which case it assumes the new value computed after the modifications.

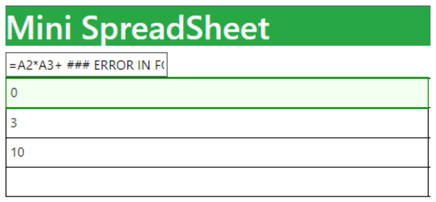
Figure 1.15: In this case the user has put the value 3 in A2 and 10 in A3. Now s/he wants to calculate the product between A2 and A3. While s/he does so, the formula is shown also in the cell currently selected (A1 in this example). After the user submits the formula, the cell changes its content with its value.





If the user puts an illegal formula (either formulas with syntax error or circular dependencies), the computed value for the modified cell is evaluated with 0. Then if the user clicks on the cell with the illegal formula, the text field shows that there is an error in the formula.

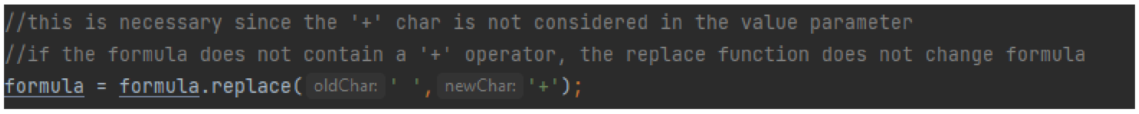
Figure 1.16: Example of illegal formula. In this case the user has put a '+' char after the formula, which made the engine to consider it wrong and to evaluate it as 0. The same thing would happen if the user puts a formula with circular dependency.



1.4 Comments

Overall I managed to realize a solution that would fulfill all the specifications defined for this assignment without too many problems, even toughd during the development of the first part of the assignment I had to deal with some (minor) issues:

- First issue was the fact that the ModifyCellServlet was not able to detect the '+' char which would be present in the formula given as parameter by the browser request, thus it reads it as a ' ' char, which would lead to a syntax error. The solution to this issue was to replace the ' ' char with the '+' char. This has been done easily by using the replace() method which replaces a character (or a substring) with another one inside the ModifyCellServlet.



```
//this is necessary since the '+' char is not considered in the value parameter
//if the formula does not contain a '+' operator, the replace function does not change formula
formula = formula.replace( oldChar: ' ', newChar: '+');
```

- Second issue was the fact that when a circular dependency happens, the cell in which the formula has been write does not show the value 0 but it shows the previous one (i.e. the value before the formula has been submitted). This happens because, inside the modifyCell() method, to check if there are any circular dependencies we call the method checkCircularDependencies() and in the case of any circular dependecy present in the formula the modifyCell() method restores the cell without returing anything (usually this method returns the list of all the affected cells but in this case it returns null). In my opinion this behaviour is correct because it prevents having cells with illegal formulas with circular dependencies, but in the specifications for this assignment it is written that in the case of circular dependecies the formula must evaluate with 0. So, to resolve this issue and to satisfy that specification, first it was necessary to check if any circular dependecy exists in the formula and then if there is at least one circular dependecy, then a ' ' is added to the formula. In this way the engine reads it as a formula with syntax error, thus it returns a cell with the formula given as input and with the value 0. This solution has been made inside ModifyCellServlet.

- Third issue which regards the client side was the fact that initially for the text field it was added a codeline where, once the formula was submitted, clears itself (in other words using the $('#txtField').val() function its value becames ""), but when dealing with cells that have circular dependencies it happens that when we click again on the text field immediatly after submitting the formula, it erases the content of the cell. In order to prevent this bug, I removed the code snippet that implemented such feature to keep things simpler, even though it could have been a nice feature since it would make the spreadsheet a bit more realistic.

Part 2

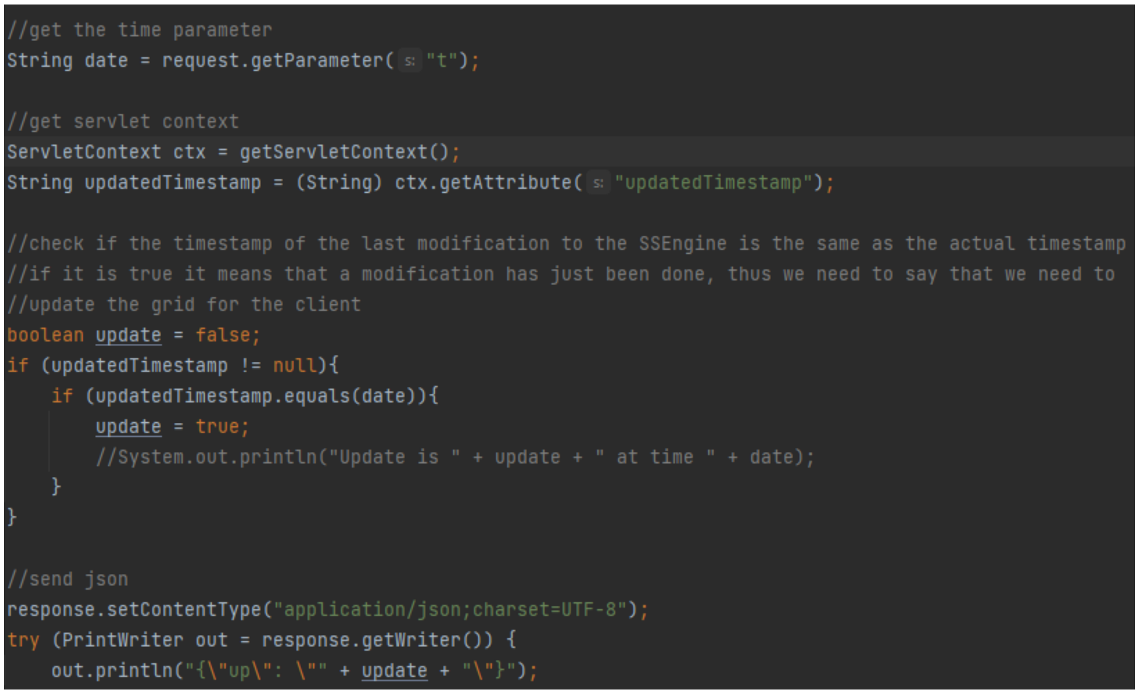
2.1 Introduction

For the second part of this assignment it was necessary to make in such a way that other users may connect to the web site and be able to see all the modifications in real time and also write new formulas which results are then propagated to all other users' browser. In order to achieve this, it has been recommended to use the Javascript setInterval function. In this case the browser sends to the server a request after a certain interval and if any changes have been made to the engine, it notifies the browser to load the new view.

2.2 Solution

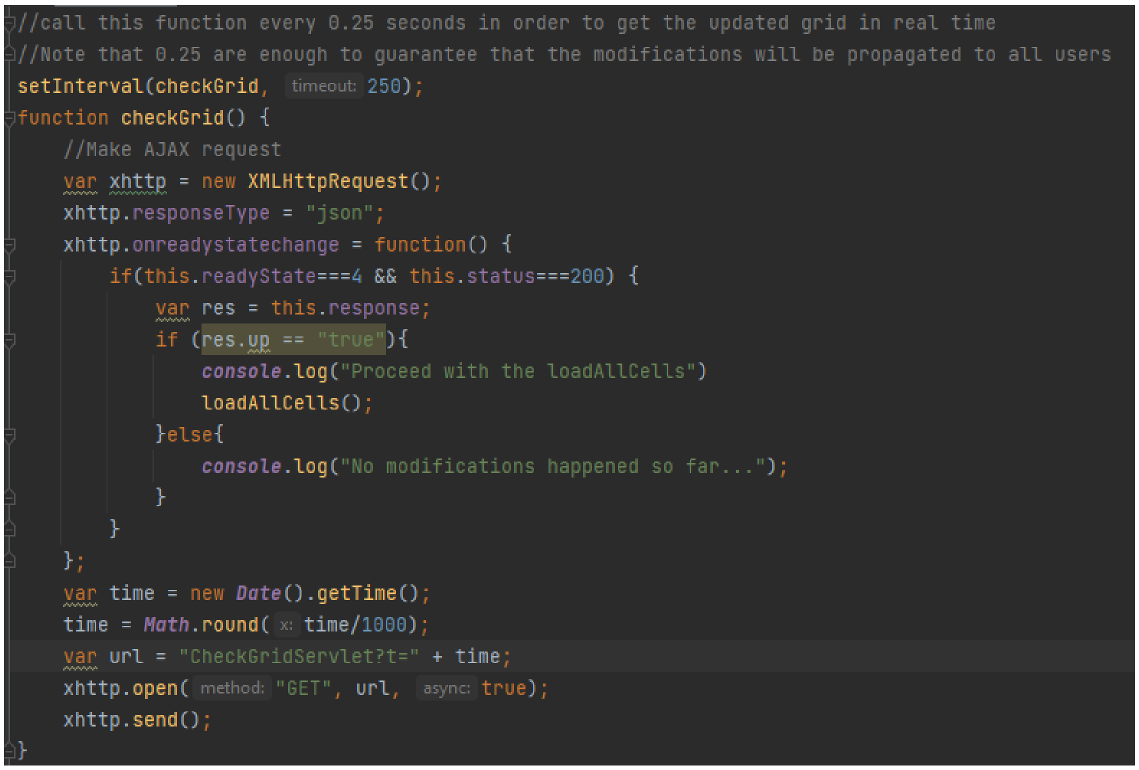
Most part of the solution of this assignment has been discussed in the previous part. In order to realize also the second part, a new servlet and a new javascript function have been implemented and added to the project. These are respectively CheckGridServlet and checkGrid().

CheckGridServlet is a simple servlet that takes the time parameter given in input by the checkGrid() function, compares it with the timestamp of the last modification in the engine and passes a boolean variable as result (Figure 2.1). If the two timestamps are equal it means that a modification has just happened and in order to notify the browser to change its view the servlet return true. Otherwise, if no changes have been made to the engine, it returns false.



```
//get the time parameter
String date = request.getParameter( s: "t");

//get servlet context
ServletContext ctx = getServletContext();
String updatedTimestamp = (String) ctx.getAttribute( s: "updatedTimestamp");

//check if the timestamp of the last modification to the SSEngine is the same as the actual timestamp
//if it is true it means that a modification has just been done, thus we need to say that we need to
//update the grid for the client
boolean update = false;
if (updatedTimestamp != null){
    if (updatedTimestamp.equals(date)){
        update = true;
        //System.out.println("Update is " + update + " at time " + date);
    }
}

//send json
response.setContentType("application/json;charset=UTF-8");
try (PrintWriter out = response.getWriter()) {
    out.println("{\"up\": \"" + update + "\"}");
```

The checkGrid() function is called every 0.25 seconds and everytime it is triggered it sends an AJAX request to CheckGridServlet. Then it gets the response and if the response value is true, then it must update the view of the grid. In order to do so it calls the loadAllCells() function which loads all the cells from the engine. In this way it is guaranteed that the modifications are propagated to all other users.

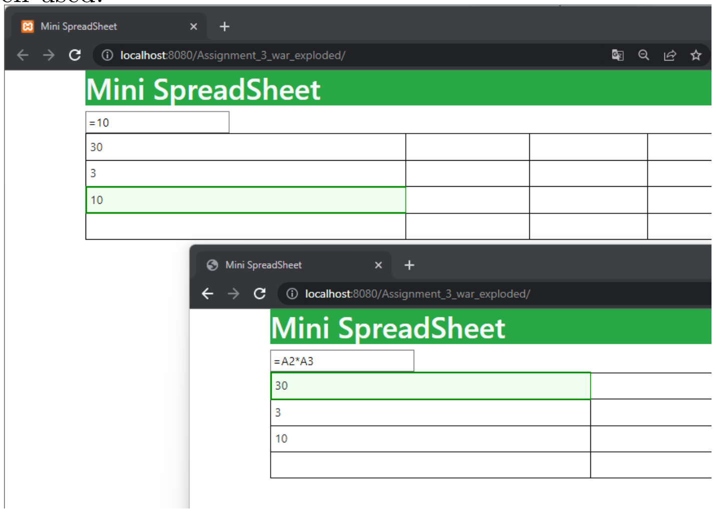


```
//call this function every 0.25 seconds in order to get the updated grid in real time
//Note that 0.25 are enough to guarantee that the modifications will be propagated to all users
setInterval(checkGrid,  timeout: 250);
function checkGrid() {
    //Make AJAX request
    var xhttp = new XMLHttpRequest();
    xhttp.responseType = "json";
    xhttp.onreadystatechange = function() {
        if(this.readyState===4 && this.status===200) {
            var res = this.response;
            if (res.up == "true"){
                console.log("Proceed with the loadAllCells")
                loadAllCells();
            }else{
                console.log("No modifications happened so far...");
            }
        }
    };
    var time = new Date().getTime();
    time = Math.round( x: time/1000);
    var url = "CheckGridServlet?t=" + time;
    xhttp.open( method: "GET", url,  async: true);
    xhttp.send();
}
```

2.3 The webapp running

While the first user works on the spreadsheet, if another user connects to the website, s/he must have the same view as all the other users since there is only one spreadsheet managed by the app. In this case, the new user's browser loads the same spreadsheet as the first user's browser.

Figure 2.1: Note that to test if different users can access and do modifications to the same spreadsheet, incognito windows in chrome have been used.



Mini SpreadSheet
localhost:8080/Assignment_3_war_exploded/

Mini SpreadSheet

=10

30			
3			
10			

Mini SpreadSheet
localhost:8080/Assignment_3_war_exploded/

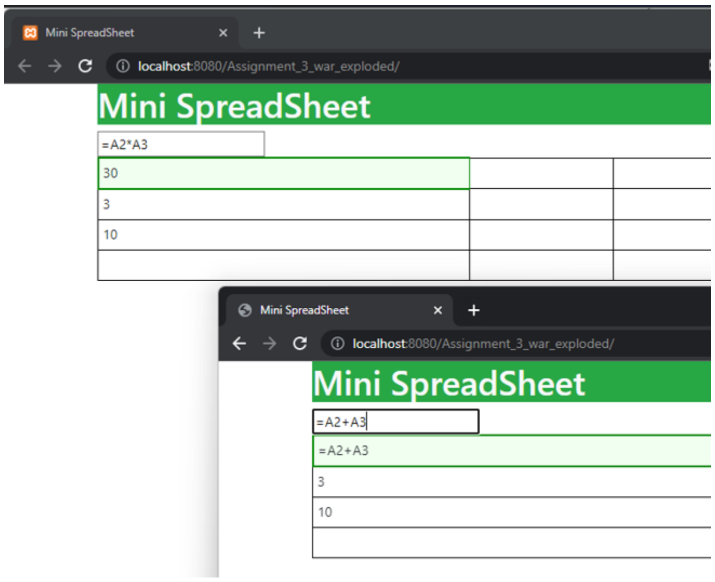
Mini SpreadSheet

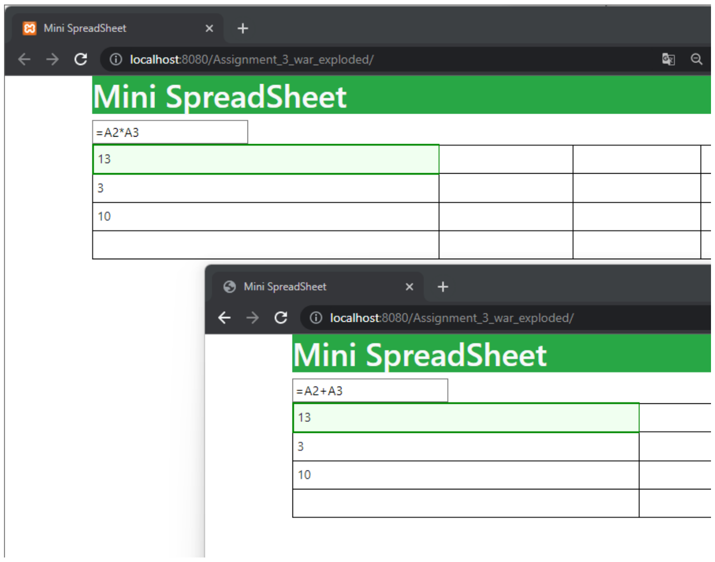
=A2*A3

30	
3	
10	

During the session, each user has a certain current cell on which they work in a certain moment whereas when a user submits a formula for her/his currently selected cell, the modifications are propagated to all users.

Figure 2.2: For example, the second user wants to change the formula and wants to do a sum instead of a product. To do so, it substitutes the '*' with '+'. When s/he submits the formula, the change first reaches her/his browser and then with the checkGrid() function which is called every 0.25 seconds, that change also reaches the other user.


Mini SpreadSheet
localhost:8080/Assignment_3_war_exploded/
Mini SpreadSheet
=A2*A3
30
3
10
Mini SpreadSheet
localhost:8080/Assignment_3_war_exploded/
Mini SpreadSheet
=A2+A3
=A2+A3
3
10


Mini SpreadSheet
localhost:8080/Assignment_3_war_exploded/
Mini SpreadSheet
=A2*A3
13
3
10
Mini SpreadSheet
localhost:8080/Assignment_3_war_exploded/
Mini SpreadSheet
=A2+A3
13
3
10