**REPORT OF THE TESTING TASK IN INDEAL PRO COMPANY**

Autor: Bohdan Kmet

Lviv-2020

TEST REQUIRMENTS: This Program confirms or refutes the possibility to create some wall configuration using some set of bricks.

PROGRAMING LANGUAGUE: JavaScript.

According to requirments that task has some limit on count of files, I wrote styles and front-end js-code in html.file in tags <script>, <style>. It isn’t my normal code. As a usual I create new files .css, .js and write code there.

BACK-END: The server was created using Node.js and express- library. The server is listening localhost:3000. You should write on console “npm start” - to run the server (I’m using nodemon).

VIEW: When the client is Calling ‘get’ on <http://localhost:3000> the server has rendered html.file with bootstrap – the form where client submitted output information and received output code.

The form has 6 inputs:

the result of the program calculations. If true – the wall can be constructed from these types of bricks. If false – it can’t.

This input is invisible until user submits the form.

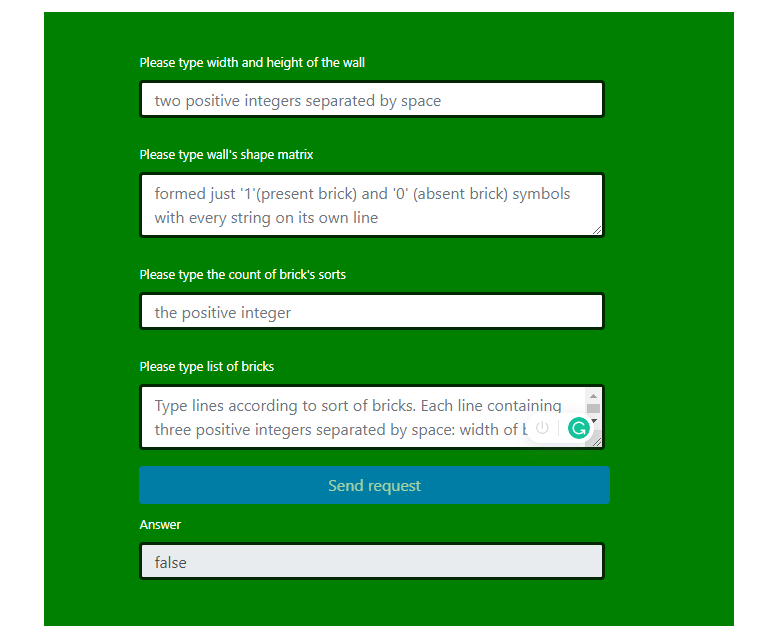
The submitted form button. It is disabled while the form is invalid.

Input intended to add the size and count of each type of brick’s.

Input intended to type count of brick’s sorts

Input intended to create the construction of the wall. If client types ‘1’ – the bricked should be there. If `0` - there is a hole.

Input intended to add width and height of the wall



Main validation of the form is created in front-end. These inputs have comply patterns validation and they are required. If input invalid - its border is red, else – it is dark- green.

Pattern to with and height of the wall is

/^([1-9][0-9]\*)(\s)([1-9][0-9]\*)$/

Only two numbers and space between them.

Wall shape matrix pattern is:

/^[0-1\n]+$/

Only 0 and 1 and enter – when the client wants to create new row of the wall.

The wall should have as count of rows and columns as it was declarated in previous input.

Pattern to the count of brick’s sort is:

/^[1-9][0-9]\*$/

Only number

Pattern to the list of bricks is:

/^(([1-8])(\s)([1-8])(\s)([1-9][0-9]\*)(\n)\*)+$/

With is the first number from 1 to eight, then space then height – second number from 1 to eight, then space, then another number – count of that sort of bricks.

If there are more then one sort of bricks, the client should type enter and repeat typing of with, height and count of another brick’s sort.

Count of rows should be equal to the value in previous input.

When client submitted the form – he called on ‘post’ <http://localhost:3000>, so back-end received request information with output data.

The server first of all is checking is the square of the wall bigger than bricks square. If it is, response is returning ‘false’:

if (wallsShapeMatrixResult > bricksCountResult) {

        return false

    };

Then, served used methods of selection and recursion in function checkMutchBetweenBrickAndPartOfWall(). This function received output arguments:

* bricksArr – array with size of every brick, that not used in wall building.
* wallshapeMatrix – array with rows of wall construction.
* previousFunctionResult – this argument has used when wall construction not matched with some kinds of bricks and server should return to previous calculations and should continue the selection of bricks but with different kind of brick.

All previous calculations server received in outside arrays:

let bricksArrContainer = [];

let wallsShapeMatrixContainer = [];

let bricksArrCountCountainer = [];

bricksArrContainer has included all bricksArr to remember the structure of bricks, when some piece of wall and brick are matched. wallShapeMatrix Container has included all wallShapeMatrix. bricksArrCountCountainer – has included all counts of brick in bricksArr when selection of some brick was successful.

When selection is successful the server is pushing successful results to that’s arrays. And when in the next step selection will be not successful the server is popping previous succesfull results and returned to calculations with next kind of brick. If there no any kind of brick the function returned false:

if (previousFunctionResult) {

                    bricksArrCount = (bricksArrCountCountainer.pop() + 1);

                    if (bricksArrCount >= bricksArr.length) {

                        wallsShapeMatrixContainer.pop();

                        bricksArrContainer.pop();

                        if (bricksArrContainer.length) {

                           result = checkMutchBetweenBrickAndPartOfWall(bricksArrContainer[bricksArrContainer.length - 1], wallsShapeMatrixContainer[wallsShapeMatrixContainer.length - 1], true);

                        } else {

                            return false;

                        }

                    }

                }

First of all function searches what piece of wall program should matched. That’s why server searches where in wallShapeMatrix are ‘1’ – the places where bricks should be presented:

 for (let i = 0; (i < wallsShapeMatrix.length && moreCheck); i++){

        for(let j=0; (j<wallsShapeMatrix[i].length && moreCheck); j++) {

            if (wallsShapeMatrix[i].charAt(j)==='1') {

and then server calculated possibility of the first brick in brickArr to match the wall. When it matches the program searches next ‘1’ in wallsShapeMatrix. And when it not matches the program calculated possibility of the next brick in brickArr to match the wall. If no one matches program returns to the previous steps:

 while (cycleChecker) {

    let brickWidth = +bricksArr[bricksArrCount][0];

    let brickHeight = +bricksArr[bricksArrCount][1];

    let checkWithAndHeightCount = true;

    for (let brickHeightCount = 0; (brickHeightCount < brickHeight &&

checkWithAndHeightCount); brickHeightCount++ ) {

      for (let brickWithCount = 0; (brickWithCount < brickWidth &&

checkWithAndHeightCount); brickWithCount++) {

      if (wallsShapeMatrix[i+brickHeightCount].charAt(j+brickWithCount) === '1') {

let newString = newWallsShapeMatrix[i+brickHeightCount].slice(0, (j + brickWithCount)) + '2' + newWallsShapeMatrix[i+brickHeightCount].slice((j + brickWithCount + 1));

   newWallsShapeMatrix[i + brickHeightCount] = newString;

       } else {

            newWallsShapeMatrix = *arguments*;

            bricksArrCount++;

            newWallsShapeMatrix = *arguments*;

            checkWithAndHeightCount = false;

        if (bricksArrCount === bricksArr.length) {

                   cycleChecker = false;

                   wallsShapeMatrixContainer.pop();

                   bricksArrContainer.pop();

                   if (bricksArrContainer.length) {

result = checkMutchBetweenBrickAndPartOfWall(bricksArrContainer[bricksArrContainer.length - 1], wallsShapeMatrixContainer[wallsShapeMatrixContainer.length - 1], true);

                   } else {

                      return false;

                   }

           }

       }

       }

}

      if (checkWithAndHeightCount) {

           bricksArr.splice(bricksArrCount, 1);

           cycleChecker = false;

           if (newWallsShapeMatrix.filter((elem) => elem.match('1')).length) {

               bricksArrCountCountainer.push(bricksArrCount);

     result = checkMutchBetweenBrickAndPartOfWall(bricksArr, newWallsShapeMatrix);

             }

             if(!newWallsShapeMatrix.filter((elem) => elem.match('1')).length) {

                      return true;

              }

      }

     }

As you can see from code bellow when selection is successful the server has modified wallShapeMatrix. He replaced ‘1’ in wallShapeMatrix by ‘2’. So, In the next step when server will be search what piece of wall program should matched, it doesn’t move this piece of wall:

if (wallsShapeMatrix[i+brickHeightCount].charAt(j+brickWithCount) === '1') {

let newString = newWallsShapeMatrix[i+brickHeightCount].slice(0, (j + brickWithCount)) + '2' + newWallsShapeMatrix[i+brickHeightCount].slice((j + brickWithCount + 1));

   newWallsShapeMatrix[i + brickHeightCount] = newString;

Therefore, due to recursion which allows the function to move forward when a particle of the wall is selected:

if (newWallsShapeMatrix.filter((elem) => elem.match('1')).length) {

               bricksArrCountCountainer.push(bricksArrCount);

     result = checkMutchBetweenBrickAndPartOfWall(bricksArr, newWallsShapeMatrix);

             }

             if(!newWallsShapeMatrix.filter((elem) => elem.match('1')).length) {

                      return true;

              }

      }

or to move back, when a wall and bricks not matched:

if (wallsShapeMatrix[i+brickHeightCount].charAt(j+brickWithCount) === '1') {

       } else {

                   if (bricksArrContainer.length) {

result = checkMutchBetweenBrickAndPartOfWall(bricksArrContainer[bricksArrContainer.length - 1], wallsShapeMatrixContainer[wallsShapeMatrixContainer.length - 1], true);

                   } else {

                      return false;

                   }

           }

the server send response to the view.

On the front-end the form will be rested automatically during 7 sec after view received response.