**Sudoku Web Application with Java**

Rae Gainer

24386365

CSC 22100: Software Design Laboratory

Professor Riddhi Kadia

May 9, 2025

**Abstract**

The purpose of this project was to develop a web-based Sudoku application using Java as the primary backend language, along with Spring Boot as the framework. The backend outlined the logic of the game, including generating the puzzle and verifying solutions. The frontend was developed using HTML, Javascript, and CSS, and Thymeleaf was used to connect it with the backend. This allowed for dynamic rendering of the puzzle grid, with user-input values automatically being added to the page. The project included random generation of original puzzle layouts, and a method that checks whether or not the puzzle is completed, as well as whether or not the solution is correct. The key outcomes of the project include greater proficiency with Java programming, as well as a better understanding of the development process, especially using a framework and using a templating engine to integrate the frontend and the backend.

## Introduction

This project involved developing a Sudoku web application. The game is presented as a 9 x 9 grid, divided into nine 3 x 3 subgrids. The goal is to fill the board such that each row, column, and 3 x 3 box contains the numbers 1 through 9, with no number being repeated in the same row, column, or box.

The objective of the project was to create a fully functional, intuitive Sudoku application with a simple and clean user interface. The application generates a new puzzle solution on loading the page, as well as when a “New Game” button is clicked. Originally, the method by which I was generating new solutions seemed a little time-consuming and over-simplistic. I was going cell by cell, randomly picking a number between 1 and 9, and checking if it was a valid placement. If it was, the program moved on to the next and so on for 81 cells. It turns out, there is a much more efficient way to do this: if the three 3 x 3 boxes on the diagonal are filled in first, they are independent of each other and we can skip the row and column checking for these 27 cells. We can randomly fill those boxes with the numbers 1 to 9, then fill the remaining cells recursively by checking each number 1 to 9 to see if it is a valid puzzle.

Once the solution has been generated, the application removes a set number of cells from the board to make it into a solvable puzzle. A user can then select a blank cell, which will be highlighted in blue to show that it has been selected. If the user inputs a number, it will be automatically added to the puzzle. The application also performs real-time input validation, where only the numbers 1 through 9 will be accepted as input, and any other key press will have no effect. Once every cell has been filled in, the application will either display a message that the puzzle has been solved, or if the solution is incorrect, that the puzzle is incomplete.

## Tools & Technologies Used

The primary programming language used for the backend was Java, which formed the majority of the logic in the application. I also used Spring Boot as a framework to help configure and set up the

application, and Maven as a build tool for compiling and running the application. The Java files are SudokuApp.java, PuzzleGenerator.java, and SudokuController.java. SudokuApp.java calls Spring Boot to automatically configure the application based on a specific classpath, and marks the main class of the application. PuzzleGenerator.java creates a new unique solution, and then removes a set number of cells to leave the blanks that must be filled in. SudokuController.java handles GET and POST requests, and maintains/manages the actual game state, including processing move inputs, checking for completion, and starting a new game.

For the frontend, I used HTML and CSS to build and format the application page. I also used Thymeleaf as a template engine to render the board dynamically. Additionally, I used Javascript for one function in the index.html file which would run when a key was pressed with an input-cell selected. The IDE used was VSCode. When testing and presenting the project, I compiled and ran the application with the command `mvn spring-boot:run`, and accessed the page in a browser at http://localhost:8080/.

## Methodology/Implementation

Originally, the intention was to create a more detailed version of the finished application. I had initially intended on implementing multiple difficulty levels, which could be accessed via a dropdown menu and would affect the number of cells removed from the puzzle (a harder difficulty setting would mean more cells removed). I also considered an error checking feature, which would cause some visual change if a value input into a cell was not correct, such as turning the cell text red. These additions would not be very difficult to implement, but it ended up taking me longer than expected to figure out Thymeleaf and how to properly render the puzzle with HTML, and I unfortunately ran out of time.

Despite these issues, the project functions as intended as a fully operational Sudoku game. The application automatically generates a new puzzle when the page is loaded, and a new puzzle can also be

generated by clicking the “New Game” button. This button sends a GET request to the controller in SudokuController.java, which runs the function below.

```
77      @GetMapping("/newGame")
78      public String newGame() {
79          //creates new solution and puzzle
80          solution = generator.generateSolved();
81          current = generator.createPuzzle(solution);
82          original = new int[9][9];
83          for (int i = 0; i < 9; i++) {
84              for (int j = 0; j < 9; j++) {
85                  original[i][j] = current[i][j];
86              }
87          }
88      complete = false;
89      return "redirect:/"; //redirects to root
90      }
```

This calls the generator defined in SudokuGenerator.java, which generates a puzzle solution and redirects back to the root, which adds attributes to the model (which are then used to display the puzzle on the main page, index.html.

```
63      @GetMapping("/")
64      public String home(Model model) {
65          //starts new game on load
66          if (current == null) {
67              newGame();
68          }
69          model.addAttribute("current", current);
70          model.addAttribute("original", original);
71          model.addAttribute("complete", complete);
72          model.addAttribute("incorrect", incorrect());
73
74          return "index";
75      }
```

When a move is made, a POST request is sent along with the row, column, and value information as parameters. This function adds the input value to the board if the selected cell is available, and checks if the puzzle is complete.

```java
@PostMapping("/makemove")
public String makemove(@RequestParam("row") int row,
                       @RequestParam("col") int col,
                       @RequestParam("value") int value) {

    if (original[row][col] == 0) { //if empty in original board
        current[row][col] = value;
    }
    checkComplete();
    return "redirect:/";
}
```

When the puzzle is complete, the following functions verify if it is both complete and correct, or if it is completely filled in but incorrect. The output of these functions are passed as model attributes to the frontend, where they are used in conditional divs which display information on whether or not the puzzle is complete and whether or not the puzzle has been solved.

```java
//complete and correct
private void checkComplete() {
    boolean solved = true;
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (current[i][j] != solution[i][j]) {
                solved = false;
                break;
            }
        }
        if (solved == false) { break; }
    }
    this.complete = solved;
}

//complete but incorrect
private boolean incorrect() {
    boolean filled = true;
    boolean solved = true;
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            if (current[i][j] == 0) {
                filled = false;
                break;
            }
            if (current[i][j] != solution[i][j]) {
                solved = false;
            }
        }
        if (filled == false) { break; }
    }
    return filled && !solved;
}
```

The file SudokuGenerator.java handles the generation of the puzzle solution. The `generateSolved` function creates an array, then calls functions which fill the three diagonal 3 x 3 boxes and then the rest of the board to create the puzzle solution (according to the process described earlier).

```java
public int[][] generateSolved() {
    int[][] solution = new int[9][9];
    fillDiagonalBoxes(solution);
    solve(solution);
    return solution;
}
//3x3 regions in diagonal
private void fillDiagonalBoxes(int[][] board) {
    for (int i = 0; i < 9; i += 3) {
        fillBox(board, i, i);
    }
}
private void fillBox(int[][] board, int row, int col) {
    ArrayList<Integer>numbers = new ArrayList<>();
    for (int i = 1; i <= 9; i++) {
        numbers.add(i);
    }

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (numbers.isEmpty()) {
                return;
            }
            int index = random.nextInt(numbers.size());
            board[row + i][col + j] = numbers.get(index);
            numbers.remove(index);
        }
    }
}
```

```java
 private boolean solve(int[][] board) {
     for (int i = 0; i < 9; i++) { //rows
         for (int j = 0; j < 9; j++) { //cols
             if (board[i][j] == 0) {
                 for (int k = 1; k <= 9; k++) {
                     if (isPuzzle(board, i, j, k)) {
                         board[i][j] = k;
                         if (solve(board) == true) {
                             return true;
                         }
                         //if not true
                         board[i][j] = 0;
                     }
                 }
                 return false;
             }
         }
     }
     return true;
 }
```

```
93      private boolean isPuzzle(int[][] board, int row, int col, int value) {
94          //checking if number exists anywhere in the row
95          for (int i = 0; i < 9; i++) {
96              if (board[row][i] == value) {
97                  return false;
98              }
99          }
100         //for cols
101         for (int i = 0; i < 9; i++) {
102             if (board[i][col] == value) {
103                 return false;
104             }
105         }
106
107         //indexes for 3x3 box regions
108         int boxrow = row - row % 3;
109         int boxcol = col - col % 3;
110         for (int i = 0; i < 3; i++) {
111             for (int j = 0; j < 3; j++) {
112                 //check if num appears in the same box already
113                 if (board[boxrow + i][boxcol + j] == value) {
114                     return false;
115                 }
116             }
117         }
118         return true;
119     }
120 }
```

Once the solution is created, the following function copies the solution to another array, and removes a set number of cells from the puzzle.

```
39      public int[][] createPuzzle(int[][] solution) {
40          int[][] puzzle = new int[9][9];
41          //copy solution
42          for (int i = 0; i < 9; i++) {
43              for (int j = 0; j < 9; j++) {
44                  puzzle[i][j] = solution[i][j];
45              }
46          }
47          //removing 30 numbers from puzzle
48          int numBlanks = 30;
49          boolean blanks[][] = new boolean[9][9];
50          for (int i = 0; i < 9; i++) {
51              for (int j = 0; j < 9; j++) {
52                  blanks[i][j] = true;
53              }
54          }
55          while (numBlanks > 0) {
56              int row = random.nextInt(9);
57              int col = random.nextInt(9);
58              if (blanks[row][col]) {
59                  blanks[row][col] = false;
60                  puzzle[row][col] = 0;
61                  numBlanks--;
62              }
63          }
64          return puzzle;
65      }
```

The main HTML file, index.html, contains the code that dynamically renders the board based on user input. The code below generates the image of the sudoku board, formatted as a 9 x 9 HTML table. Iterating through each row and column, depending on if the cell is filled or empty in the original puzzle, the cell is either given the class ‘editable,’ which allows a user to edit its value, or ‘original,’ which does not allow the value to be changed. If the cell is editable (i.e, not filled in the original puzzle), a form submits the value entered via a POST request (handled in SudokuController.java, as seen above).

```
<table id="sudoku-table">
    <tr th:each="row : ${#numbers.sequence(0, 8)}">
        <td th:each="col : ${#numbers.sequence(0, 8)}"
            th:class="${original[row][col] > 0} ? 'original' : 'editable'">
            <span th:if="${original[row][col] > 0}"
                  th:text="${current[row][col]}"></span>

            <form th:if="${original[row][col] == 0}"
                action="/makemove" method="post">
                <input type="hidden" name="row" th:value="${row}">
                <input type="hidden" name="col" th:value="${col}">
                <input type="hidden" name="value" value="0">

                <input type="text" maxlength="1" class="input-cell"
                       onkeydown="cellInput(event, this.form)"
                       th:value="${current[row][col] > 0 ? current[row][col] : ''}"></span>
            </form>
        </td>
    </tr>
</table>
```

When the user enters a value, the `onkeydown` event calls the Javascript function `cellInput`, which submits whatever value was entered, and sets the value to 0 if the backspace key is pressed. This function also handles input validation, using the `preventDefault` function to suppress any input other than the numbers 1 through 9.

```
<script>
    //to handle input
    function cellInput(event, form) {
        event.preventDefault(); //only character 1-9 can be input
        const num = event.key;
        if (num >= '1' && num <= '9') {
            event.target.textContent = num;
            form.querySelector('input[name="value"]').value = num;
            form.submit();
        } else if (num == 'Backspace' || num == 'Delete') {
            event.target.textContent = '';
            form.querySelector('input[name="value"]').value = '0';
            form.submit();
        }
    }
</script>
```

When all empty cells are filled in, the following uses the output from the `checkComplete` and `incorrect` functions (included above) to display either a success message or an error message, depending on whether or not the solution is correct.

```
49          <div th:if="${complete}" class="congrats">
50              <h2>Congratulations! You finished the puzzle :)</h2>
51          </div>
52          <div th:if="${incorrect}" class="incorrect">
53              <h2>The solution is not correct :(</h2>
54          </div>
```

**Results/Output**

When the application is first loaded, a new, automatically generated puzzle is displayed.

# Sudoku

New Game

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 4 | | 2 | | | 6 | 5 | |
| 9 | | 5 | | | 1 | | 2 | 8 |
| 2 | | 8 | 7 | 5 | 9 | 1 | | 4 |
| 3 | 7 | 1 | 8 | | | 4 | 6 | 2 |
| | 8 | 4 | 1 | 2 | 6 | 9 | | 3 |
| 6 | 2 | | 3 | 7 | 4 | 8 | 1 | |
| | | | | 8 | | 5 | | |
| | | 6 | 9 | | 3 | | | 1 |
| 8 | 9 | 2 | 5 | 1 | | 3 | 4 | |

When a cell is clicked, it turns blue to signal that the cell is selected. Once a numeric value is entered, it is automatically saved and entered into the puzzle. This is shown below with a 5 on the left of the board.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 4 | | 2 | | | 6 | 5 | |
| 9 | | 5 | | | 1 | | 2 | 8 |
| 2 | | 8 | 7 | 5 | 9 | 1 | | 4 |
| 3 | 7 | 1 | 8 | | | 4 | 6 | 2 |
| | 8 | 4 | 1 | 2 | 6 | 9 | | 3 |
| 6 | 2 | | 3 | 7 | 4 | 8 | 1 | |
| | | | | 8 | | 5 | | |
| | | 6 | 9 | | 3 | | | 1 |
| 8 | 9 | 2 | 5 | 1 | | 3 | 4 | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 4 | | 2 | | | 6 | 5 | |
| 9 | | 5 | | | 1 | | 2 | 8 |
| 2 | | 8 | 7 | 5 | 9 | 1 | | 4 |
| 3 | 7 | 1 | 8 | | | 4 | 6 | 2 |
| 5 | 8 | 4 | 1 | 2 | 6 | 9 | | 3 |
| 6 | 2 | | 3 | 7 | 4 | 8 | 1 | |
| | | | | 8 | | 5 | | |
| | | 6 | 9 | | 3 | | | 1 |
| 8 | 9 | 2 | 5 | 1 | | 3 | 4 | |

Once the puzzle is complete, the application will display a message, depending on whether or not the user's solution is correct. In the example below, the 1 in the top right corner is incorrect, and when it is changed to the correct value, 9, the message changes to reflect that the solution is correct.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 7 | 2 | 3 | 8 | 6 | 5 | 1 |
| 9 | 3 | 5 | 4 | 6 | 1 | 7 | 2 | 8 |
| 2 | 6 | 8 | 7 | 5 | 9 | 1 | 3 | 4 |
| 3 | 7 | 1 | 8 | 9 | 5 | 4 | 6 | 2 |
| 5 | 8 | 4 | 1 | 2 | 6 | 9 | 7 | 3 |
| 6 | 2 | 9 | 3 | 7 | 4 | 8 | 1 | 5 |
| 4 | 1 | 3 | 6 | 8 | 2 | 5 | 9 | 7 |
| 7 | 5 | 6 | 9 | 4 | 3 | 2 | 8 | 1 |
| 8 | 9 | 2 | 5 | 1 | 7 | 3 | 4 | 6 |

**The solution is not correct :(**

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 7 | 2 | 3 | 8 | 6 | 5 | 9 |
| 9 | 3 | 5 | 4 | 6 | 1 | 7 | 2 | 8 |
| 2 | 6 | 8 | 7 | 5 | 9 | 1 | 3 | 4 |
| 3 | 7 | 1 | 8 | 9 | 5 | 4 | 6 | 2 |
| 5 | 8 | 4 | 1 | 2 | 6 | 9 | 7 | 3 |
| 6 | 2 | 9 | 3 | 7 | 4 | 8 | 1 | 5 |
| 4 | 1 | 3 | 6 | 8 | 2 | 5 | 9 | 7 |
| 7 | 5 | 6 | 9 | 4 | 3 | 2 | 8 | 1 |
| 8 | 9 | 2 | 5 | 1 | 7 | 3 | 4 | 6 |

**Congratulations! You finished the puzzle :)**

## Conclusion

This project successfully completed the initial objective, to create a fully functional web-based Sudoku application. The backend, built using Java and Spring Boot, generates unique puzzles using a more efficient process, and manages the game state and solution checking. The frontend, built using HTML, Javascript, and CSS, conducts real-time user input validation and automatically updates the board display when a value is entered. The frontend and the backend are integrated using Thymeleaf for the template engine, which allows the board to be rendered dynamically as new values are entered.

The primary outcomes of this project were developing proficiency in Java, as well as in designing a more complex program. I had a lot to learn about designing a web application, specifically in regard to integrating Spring Boot and Thymeleaf, both of which I was unfamiliar with when I started working on the project. Now that the project is complete, I feel that I have a much better understanding of how the many moving pieces interact in an application like this one.