CS5610 — Project 3 Writeup

Xihao Liu — CS5610 Web Development

1. What were some challenges you faced while making this app?

One main challenge was handling different data types between the frontend and backend. The frontend uses null for empty cells, but MongoDB needs numbers, so I had to convert between null and 0 when sending data. Another issue was when many users use the app at the same time, they might try to create guest users with the same username. I fixed this by checking for duplicate errors and trying again with a new username. The random game name generator could also create duplicate names, so I added code to check if a name already exists and pick a new one if needed. For custom puzzles, checking that they have exactly one solution takes a lot of work, so I made the code stop early when it finds more than one solution instead of checking everything. There was also a timing problem where the app might try to create a game before the user is ready, so I added code to wait for the user to be set up first. Finally, showing which cells are wrong in real-time while the user is typing required careful code to check rows, columns, and boxes without slowing down the app.

2. Given more time, what additional features, functional or design changes would you make?

If I had more time, I would add complete registration and login features. Right now the login and register pages don't actually work - they're just for show. I would add real password checking, ways to keep users logged in, password reset, email checking for new accounts, and rules for strong passwords. I would also make the database better for managing accounts by adding more user information like email and profile pictures, tracking if accounts are active or blocked, keeping records of what users do, and setting up different user types like admin and regular users. For the game itself, I would add ways to save progress, undo and redo moves, let people rate how hard custom games are, share games with links, make better leaderboards with filters, and show game statistics.

3. What assumptions did you make while working on this assignment?

I assumed that the guest user system with auto-generated names would be good enough for the main features, and that real login could be added later. I also assumed that MongoDB would always be connected when the server starts, even though I check the connection in some places. For user setup, I assumed people would wait up to 2 seconds for their guest account to be ready before trying to make games, which isn't the best experience but the code handles it. When people make custom puzzles, I assumed they would try to make valid Sudoku puzzles and that error messages would help them fix problems, but the system doesn't stop obviously wrong inputs right away. I assumed custom games must have exactly one solution, which is normal for Sudoku, but this might be too strict for some cases. I also assumed that network calls would work or fail clearly, without adding complex retry logic or offline support. Finally, I assumed people would use modern browsers that support newer JavaScript features, without making it work on older browsers.

4. How long did this assignment take to complete?

This assignment took about 40 hours to complete.

5. What bonus points did you accomplish? Please link to code where relevant and add any required details.

I completed Custom Games bonus (10pts). This feature allows users to create their own Sudoku puzzles by filling in a 9x9 board, which are then validated and saved to the database.

Core Functionality

1. Frontend: Custom Game Creation Interface

The custom game page (`frontend/src/pages/CustomGame.jsx`) provides an interactive 9x9 Sudoku board where users can input numbers. It includes real-time validation to highlight invalid cells (duplicates in rows, columns, or 3x3 subgrids):

```
// Real-time validation of cell placements
useEffect(() => {
    const invalid = [];
    for (let row = 0; row < SIZE; row++) {
        for (let col = 0; col < SIZE; col++) {
            const value = board[row][col];
            if (value !== null && value !== 0) {
                // Check row, column, and subgrid for conflicts
                // ... validation logic
            }
        }
    }
    setInvalidCells(invalid);
}, [board]);
```

2. API Upload: Submitting Custom Puzzle

When the user clicks "Submit", the board is converted to the backend format (null → 0) and sent via POST request:

```
// Convert board to format expected by backend (null -> 0)
const boardForAPI = board.map(row => row.map(cell => cell === null ? 0 : cell));

// Submit to backend for validation and creation
const response = await fetch(`${API_BASE_URL}/api/sudoku/custom`, {
    method: "POST",
    headers: {
        "Content-Type": "application/json",
    },
    body: JSON.stringify({
        board: boardForAPI,
        createdByUserId: userId,
    }),
});
```

3. Backend Validation: Puzzle Reasonableness Check

The backend endpoint (`backend/apis/sudoku.js`) performs comprehensive validation:

```
// POST /api/sudoku/custom
router.post("/custom", async (req, res) => {
    // 1. Validate board dimensions (must be 9x9)
    if (board.length !== SIZE) {
        return res.status(400).json({error: `Board must be ${SIZE}x${SIZE}`});
    }

    // 2. Check for obvious conflicts (duplicate numbers)
    for (let row = 0; row < SIZE; row++) {
        for (let col = 0; col < SIZE; col++) {
            const value = puzzleBoard[row][col];
            if (value !== null && value !== 0) {
                const tempBoard = puzzleBoard.map(r => [...r]);
                tempBoard[row][col] = null;
                if (!isValidPlacement(tempBoard, row, col, value, SIZE)) {
                    return res.status(400).json({
                        error: "Invalid Sudoku: The board contains conflicts"
                    });
                }
            }
        }
    }

    // 3. Verify puzzle has exactly one solution
    if (!verifyUniqueSolution(puzzleBoard, SIZE)) {
        return res.status(400).json({
            error: "Invalid Sudoku: The puzzle must have exactly one valid solution"
        });
    }

    // 4. Solve the puzzle to get the solution
    const solution = solveSudoku(puzzleBoard, SIZE);
    if (!solution) {
        return res.status(400).json({
            error: "Invalid Sudoku: The puzzle has no solution"
        });
    }

    // 5. Create and save the game
    const gameData = {
        name: gameName,
        difficulty: "NORMAL",
        size: SIZE,
        boardInitial: boardInitial,
        boardSolution: solution,
        createdBy: createdBy,
    };
    const newGame = await createGame(gameData);
});
```

4. Solution Uniqueness Verification

The core validation logic (`backend/utils/sudokuGenerator.js`) uses a backtracking algorithm to count solutions:

```
// Verify that a puzzle has exactly one solution
function verifyUniqueSolution(puzzle, size) {
    const solutions = countSolutions(deepCopy2DArray(puzzle), size, 2);
    return solutions === 1;
}

// Count solutions using backtracking (stops at 2 for optimization)
function countSolutions(board, size, maxSolutions = 2) {
    let solutionCount = 0;

    function solve() {
        // Find first empty cell
        for (let row = 0; row < size; row++) {
            for (let col = 0; col < size; col++) {
                if (board[row][col] === null || board[row][col] === 0) {
                    // Try each possible number
                    for (let num = 1; num <= size; num++) {
                        if (isValidPlacement(board, row, col, num, size)) {
                            board[row][col] = num;
                            solve();
                            board[row][col] = null; // Backtrack

                            // Early exit: stop if multiple solutions found
                            if (solutionCount >= maxSolutions) {
                                return;
                            }
                        }
                    }
                    return;
                }
            }
        }
        solutionCount++; // Found a solution
    }

    solve();
    return solutionCount;
}
```

The validation ensures that:

- The board has no immediate conflicts (duplicate numbers in rows, columns, or subgrids)
- The puzzle has exactly one valid solution (standard Sudoku requirement)
- The puzzle is solvable (has at least one solution)
- The solution is computed and stored for game completion checking

Once validated, the custom game is saved with a unique name and the user is redirected to play it.