Software Engineering, CISC-6100

SUDOKU KING Using MERN STACK

http://141.155.170.66:5173

By

Prarthana | Gurpreet | Akash

Team Fordham (F)

Guided by

Professor William Lord
Fordham University

December 2024

TABLE OF CONTENTS

Chapter 1 : Requirement Gathering and Analysis

[I] Introduction

This project focuses on developing a dynamic and engaging Sudoku application using the MERN stack framework (MongoDB, Express.js, React, and Node.js). The primary objective of the application is to provide users with an interactive platform for solving 2-dimensional Sudoku puzzles of various sizes (e.g., 4x4, 9x9).
The Sudoku app aims to cater to users of all skill levels by offering features such as customizable difficulty settings, a notation area, and helpful hints to enhance the gaming experience. Additionally, it incorporates a timing mechanism to encourage competitive play and skill improvement.
By leveraging the MERN stack, we have built a robust and user-friendly application that ensures efficient backend operations and seamless frontend interactions.

[II] Functional and Non-Functional Requirements

The key functionalities identified during the requirement gathering phase are:

Functional Requirements

1. **Puzzle Library and Generation:** Users can either select puzzles from a preloaded library or generate them on demand.
2. **Difficulty Levels:** Users can choose puzzles with varying difficulty levels to match their skill level.
3. **Notation Area:** Provides an area where users can make notes to assist with solving the puzzle.
4. **Solution Checking:** Users can check the correctness of their solution up to the current point.
5. **Undo Functionality:**
    a. Undo the last move.
    b. Undo all moves to remove any mistakes made.
6. **Hints:** Users can request random or specific hints for guidance.
7. **Timing Mechanism:** Tracks the time taken to solve the puzzle, encouraging competitive play and personal improvement.

Non-Functional Requirements

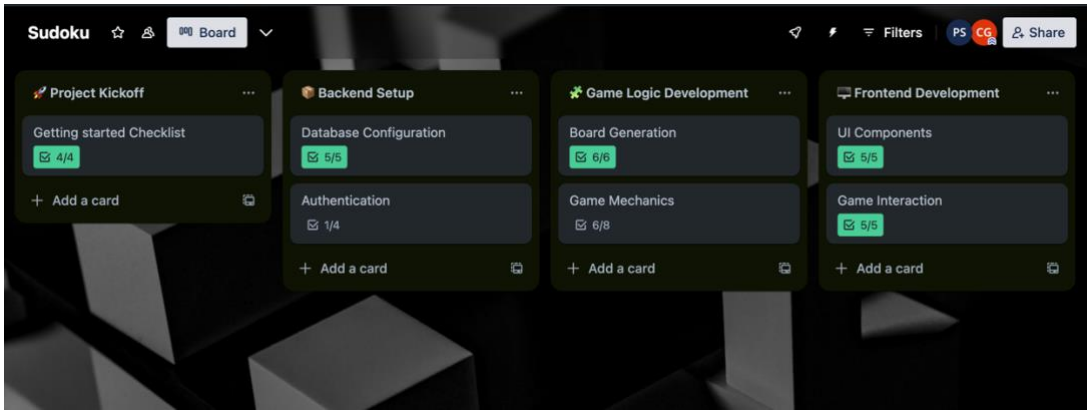
1. **Performance:**
    a. The app should load puzzles and provide hints within seconds.
    b. Ensure smooth transitions between different puzzle sizes and difficulty levels.
2. **Scalability:**
    a. The application should handle multiple concurrent users without performance degradation.
3. **Usability:**
    a. Provide an intuitive user interface that makes navigation and gameplay seamless.

Chapter 2 : Project Planning

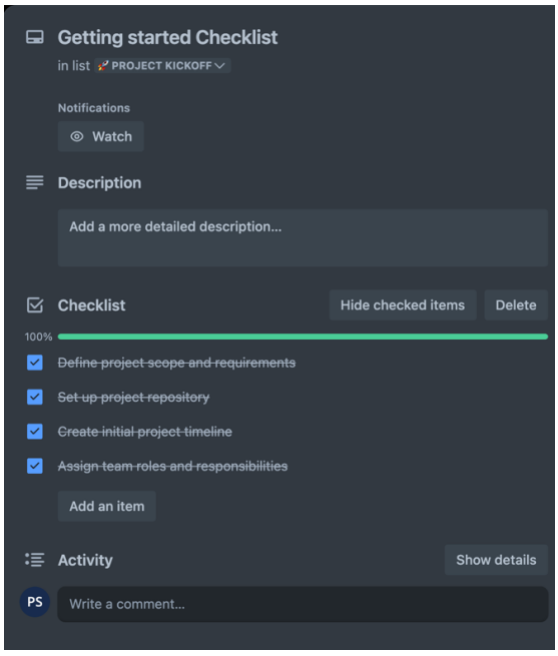
[I] Project Plan Initial Gantt chart

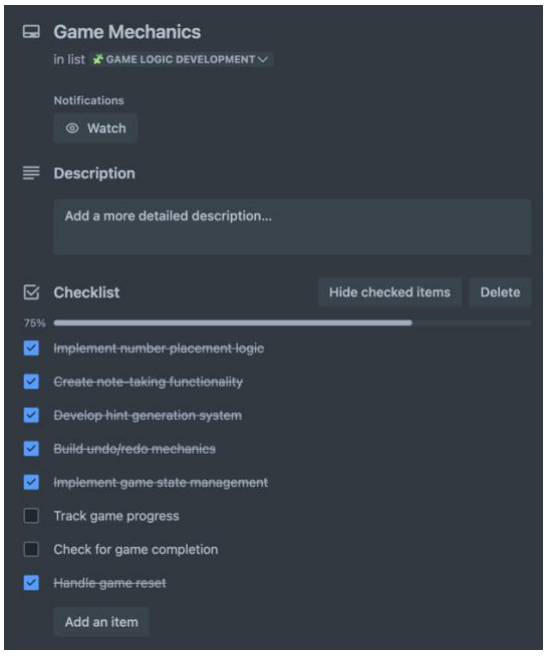
Project	Activity	Detail Description	Project Members	Timeline									
				7-Oct	14-Oct	21-Oct	28-Oct	4-Nov	11-Nov	18-Nov	25-Nov	2-Dec	9-Dec	14-Dec
1	Gathering user requirements	Gathering basic requirement	All members											
		Storyboard	Prarthana											
		Use Case Diagram	Akash											
		Class Diagram	Gurpreet											
2	Generating technical requirements	Decide the tech stack (MERN Stack)	All members											
		Algorithm Design	All members											
		Start the UI/UX Design	All members											
3	Implementation	Write Frontend Code	Prarthana											
		Setup the server	Gurpreet											
		Develop Backend functionality	Akash											
		Design Database Schema	Akash											
		Design API Endpoints	Prarthana/Gurpreet											
4	Testing	Implement Unit Testing	Akash											
		Implement Integration Testing	Prathana											
		Implement System Testing	Gurpreet											
5	Deployment	Build and Optimization	Akash											
		Domain and DNS Configuration	Gurpreet											
		Decide Hosting Platform	All members											
		Final Testing and QA	Prarthana											

[II] Tasks tracking using Trello



For the Sub-tasks :

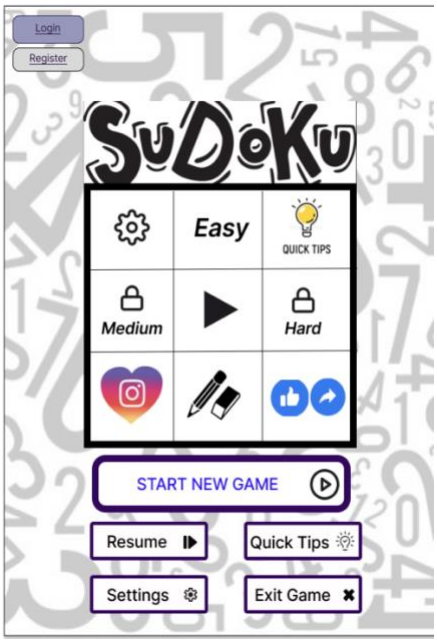


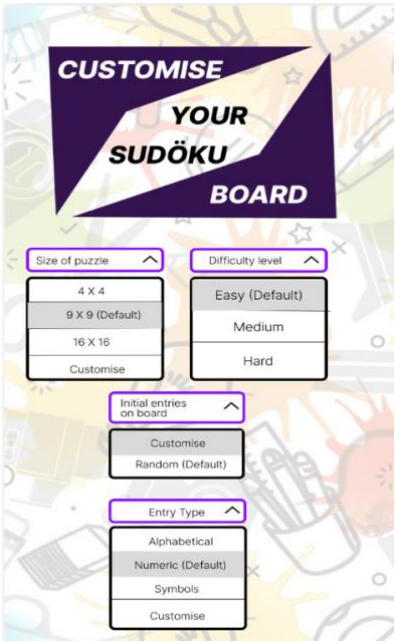


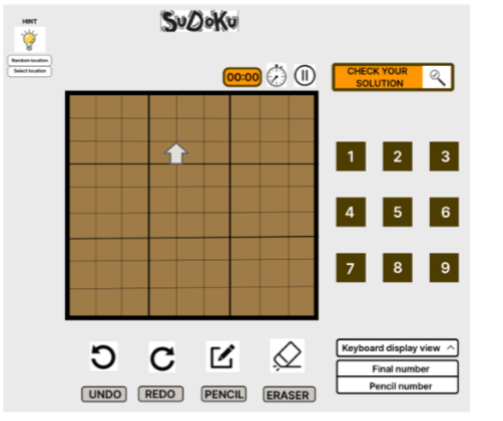
Chapter 3 : System Design

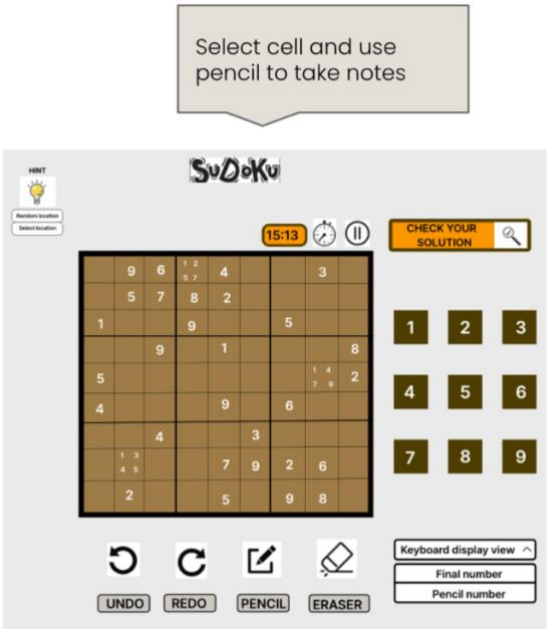
[I] Initial Story Board

We took the reference from Prarthana's initial story board for our project. Although we couldn't replicate the UI but we were able to almost achieve all the functionalities working or the sudoku board.









[II] Use Case Diagrams

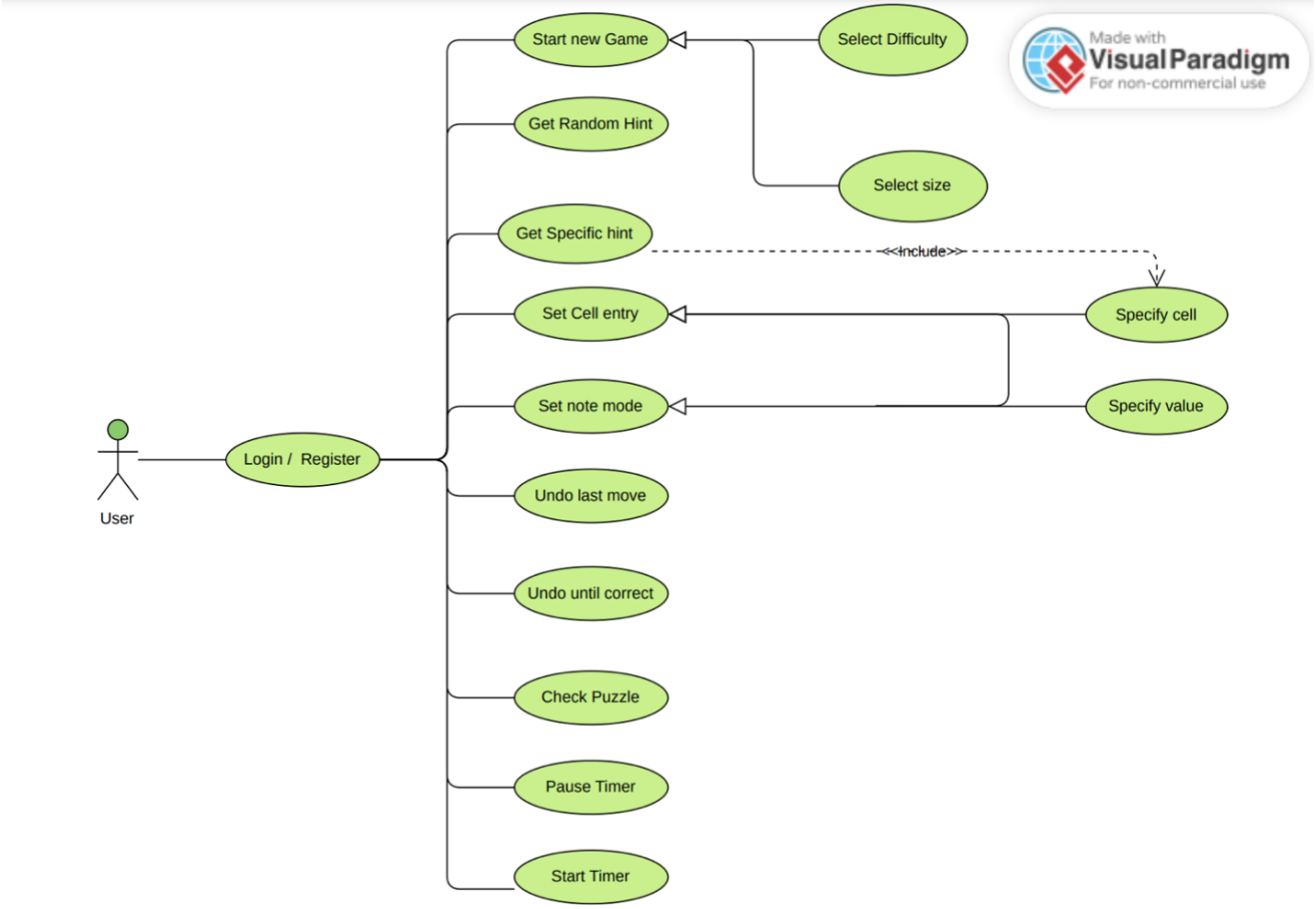


Fig : - High level use case diagram for our Sudoku game

[III] Class Diagram :

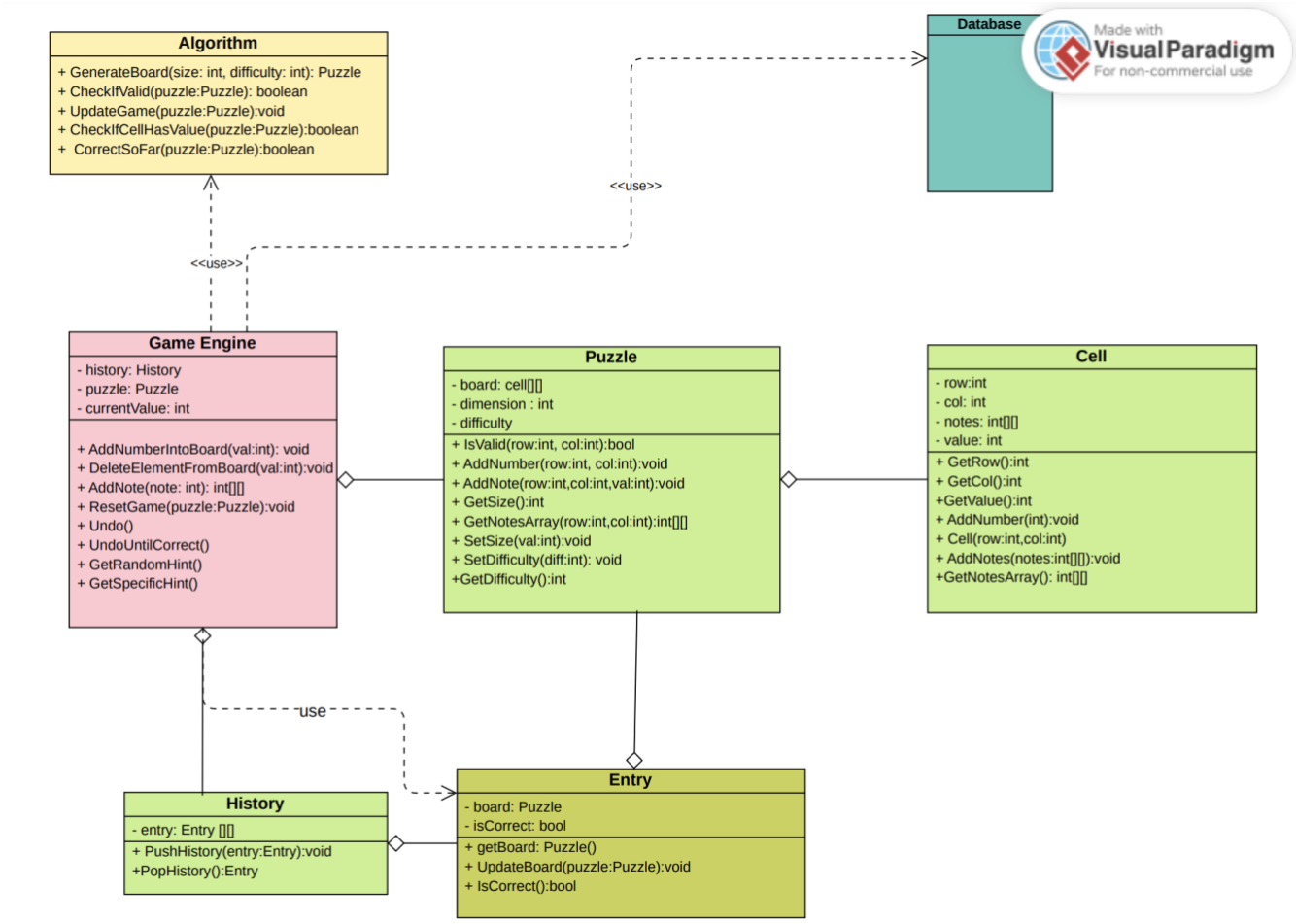
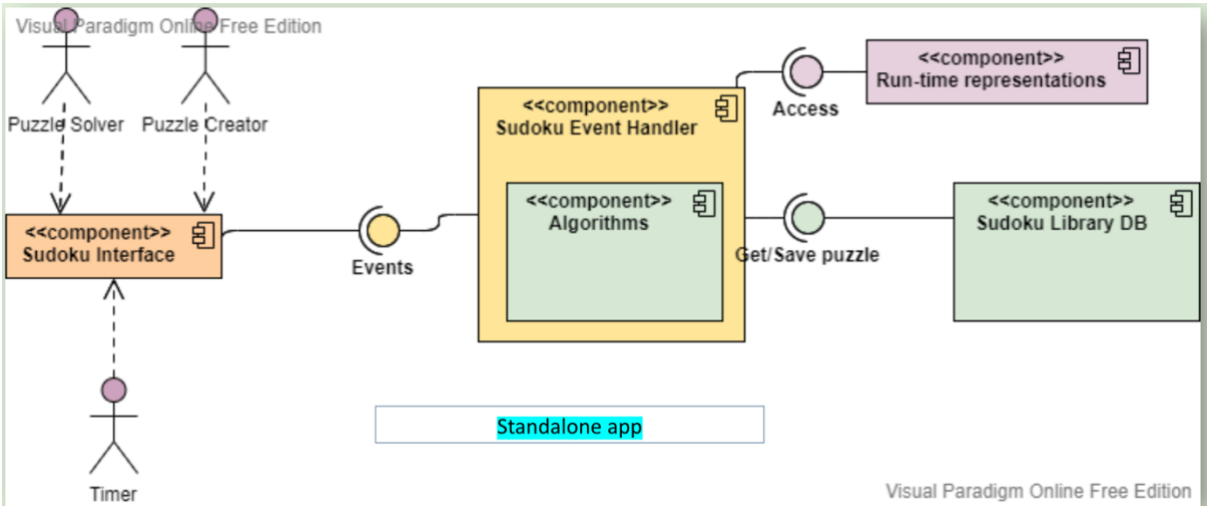


Fig: Class diagram for Sudoku King

[IV] Component diagram



[V] Activity Diagram for Key Algorithms

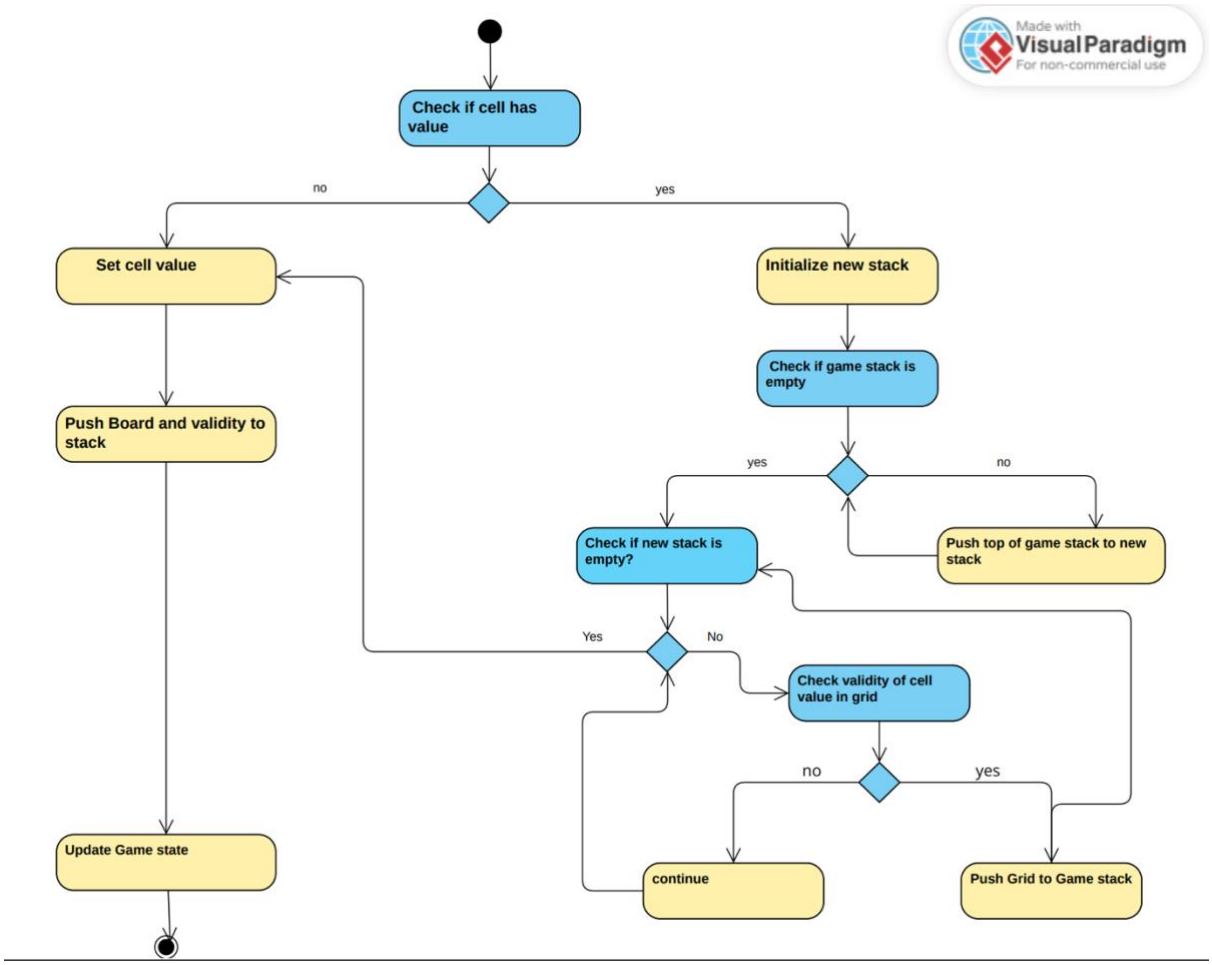


Fig : Activity Diagram for fixing history stack

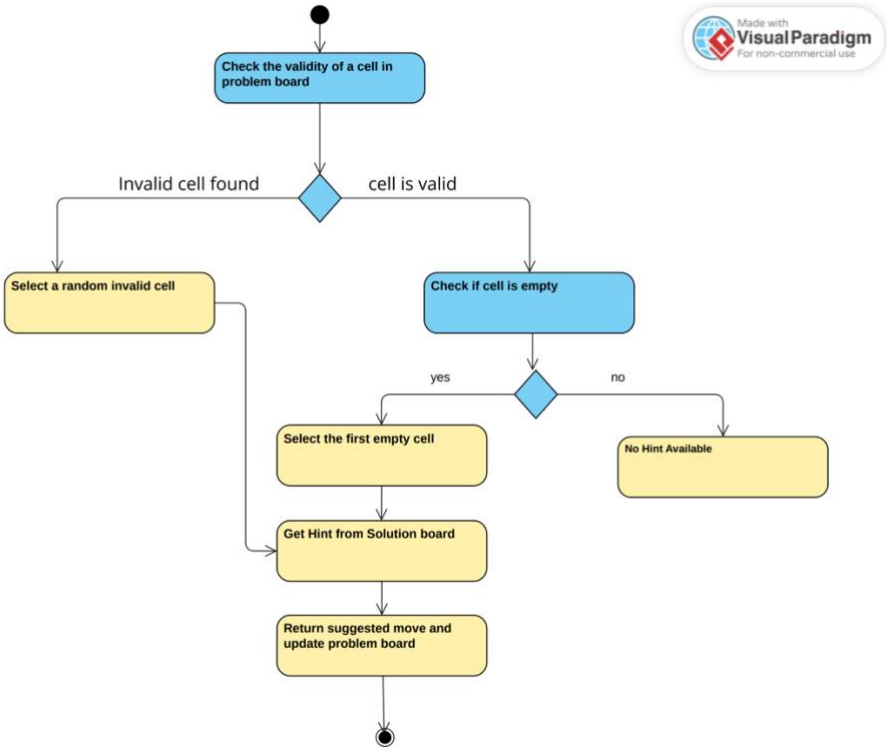


Fig : Activity Diagram for random hint

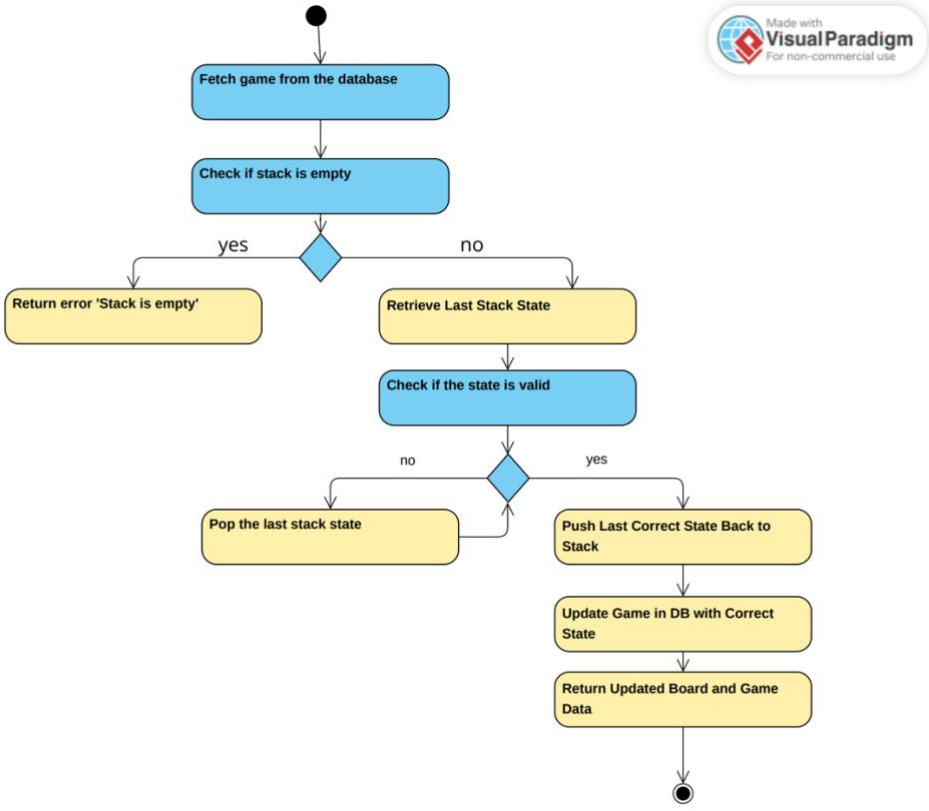


Fig : Activity Diagram for Undo until correct

[VI] Sudoku King Development Process

For our Sudoku game project, we followed an **iterative development process with Agile principles** because it allowed us to build the project incrementally while continuously refining features.

Iterative approach was ideal as it enabled us to focus on delivering small, manageable increments, such as the UI, backend, and API, while ensuring each part was functional and tested before moving to the next.

Agile principles were ideal for this project because they emphasized flexibility, frequent testing, and incorporating feedback, allowing us to adapt to evolving requirements, such as improving the UI design or adding backend features.

Chapter 4: Implementation

[I] Front end Development

Overview:

The frontend of the application is built using **React**, a powerful JavaScript library for building interactive and dynamic user interfaces. To enhance the visual appeal and maintain consistency in styling, we employed **Tailwind CSS**, a utility-first CSS framework. Tailwind CSS simplifies the styling process by offering pre-defined classes that can be directly applied to HTML elements, reducing the need for writing custom CSS. Additionally, **React Context** and custom hooks are extensively used to manage the application's state efficiently, ensuring that shared data flows seamlessly across components.

1. App Structure and Routing:

The App component is the main entry point of the frontend. It orchestrates routing, persists critical game states across sessions, and sets the stage for all Sudoku-related components. By separating concerns—routing at the top level, state management in providers, and logic within each component—this structure ensures scalability, maintainability, and a consistent user experience.

Key Responsibilities:

- ⇒ Routing: Uses react-router-dom to define and manage different application views.
- ⇒ Persistent State: Stores and retrieves currentGameId from localStorage via the useLocalStorage hook, ensuring game continuity across browser sessions.
- ⇒ Game Configuration: Maintains puzzle dimension, difficulty, and note mode states, passing them down to the Sudoku-related components.
- ⇒ Providers and Layout: Wraps the /play route in a SudokuBoardProvider to make the puzzle state available throughout the child components. Includes Navbar, Content, Toolbar, and integrates background music with MusicPlayer.

2. Hooks :

In React, **custom hooks** are functions that start with use and allow to extract component logic into reusable pieces. They help keep code organized, maintainable, and scalable. For this Sudoku project, custom hooks are used to abstract state management and storage details, making it easier to maintain and update the puzzle's state across different components.

i. useLocalStorage

Purpose:

This hook simplifies reading and writing data to localStorage, so user can persist certain parts of the application state—like the current puzzle or user preferences—across browser sessions. This store user settings (e.g., difficulty level, or last played board state) so the user can continue where they left off. It initializes state from localStorage if available; otherwise, uses defaultValue. It also updates both the component state and localStorage whenever the value changes.

ii. useSudokuGrid

Purpose:
This hook manages the state of the Sudoku board and handles modifications such as adding numbers, adding/removing notes, and resetting cells. It centralizes the logic for updating the puzzle grid and ensures the UI remains consistent with the backend game data. It interacts with the backend via addNumber, addNote, and deleteElementFromBoard API calls to ensure the state is in sync with the server.

This hook is crucial for the core gameplay functionality. Any cell interaction—such as entering a number or toggling a note—is handled here. It updates the state of the board, reflecting those changes both visually and in the stored game state on the server.

3. Context Provider:

SudokuBoardProvider

Purpose:

It provides a central context to share the Sudoku board state and its related methods across multiple components. It uses useSudokuGrid hook for managing the board's state (numbers, notes, selected cell). It also makes it easy for child components to read and update the Sudoku board without manually passing props down multiple levels.

It mainly interacts with :
SudokuBoardContext: The React context holding board state and actions.
useSudokuBoard(): Hook to access the context's values (e.g., sudokuGrid, handleCellChange).
SudokuBoardProvider: Wraps children components so they can consume the Sudoku board context.

4. Key Frontend Components:

4.1 Toolbar Component

- ⇒ Provides controls for user actions such as undoing moves, switching note mode, getting hints, and checking the board.
- ⇒ Interacts directly with the useSudokuBoard() context to access selectedCell and sudokuGrid.
- ⇒ Calls backend APIs (e.g., undo, undoUntilCorrect, getRandomHint, getSpecificHint, CheckSolution, Note Mode) to adjust the puzzle and updates the UI accordingly.

4.2 Navbar Component

- ⇒ Provides user interface controls for starting a new game, choosing puzzle dimension (4x4 or 9x9), and difficulty (Easy, Medium, Hard).
- ⇒ Updates global game states like setBoardDimension and setDifficulty.

⇒ Displays dialogs for confirming puzzle switches.
⇒ Calls setCurrentGameId to reset or load new puzzles.

4.3 Puzzle Board

⇒ Renders the Sudoku grid, subdivided into subgrids (2x2 for 4x4 puzzles, 3x3 for 9x9 puzzles).
⇒ Interfaces with the backend to fetch a new board or load an existing one by currentGameId.
⇒ Manages keyboard and mouse interactions like Arrow keys to navigate cells and Number keys and backspace/delete to modify cells.
⇒ Displays a GameTimer and a numeric Keypad for user inputs.

Key Props involved:

⇒ currentGameId, addNoteMode, boardDimension
⇒ setCurrentGameId, setBoardDimension

Key Internal State/Handlers:

⇒ isLoading to track board data fetching.
⇒ handleCellChange from the Sudoku context to apply changes to the board.

4.4 Cell Component

- Represents a single Sudoku cell.
- Shows either a fixed number, a user-entered number, or candidate notes.
- Highlights the selected cell, as well as related rows/columns/subgrids.
- Accepts styling for special states (e.g., incorrect cells highlighted in red).

Key Props involved:

- row, col, cell data (including value and notes).
- Handlers for cell selection (onCellClick) and changes (onChange).

Notes Handling:

- Notes are displayed in a 3x3 grid inside the cell for 9x9 puzzles (adjusted for 4x4 puzzles), showing possible candidate values.

4.5 GameTimer Component

- Tracks the time the user spends on the current game.
- Resets whenever a new game is started (identified by a new currentGameId).
- Allows pausing and resuming the timer.

4.6 Keypad Component:

- Provides a clickable UI element for entering numbers into the Sudoku puzzle.
- Adjusts the keypad numbers based on the board dimension (1-4 for 4x4 puzzles, 1-9 for 9x9 puzzles).

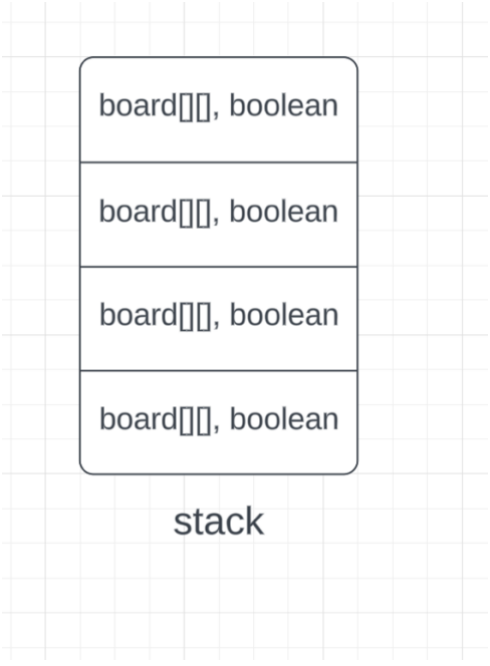
[II] Back-end Development

Overview:

The backend uses **Node.js** and **Express** to define RESTful API endpoints for a Sudoku application. It interacts with **MongoDB** via Mongoose to store and retrieve Sudoku game states. Each game document captures the puzzle boards (problem and solution), user moves (stack), and note mode status. By separating concerns—controllers for handling requests, helper functions for validation and updates, and a schema for data modeling—the backend maintains a clean, modular structure.

History stack :

Our history stack data structure records complete snapshots of the Sudoku board's state over time. Each entry stores the entire grid configuration along with a boolean flag indicating whether it was valid at that point. The most recent state sits at the top of the stack, allowing quick retrieval or rollback of the puzzle's progression. By pushing and popping these snapshots, our application can easily implement undo features and maintain a reliable history of changes as the user solves the puzzle.



1. Documentation of Backend API endpoints:

addNotes.js:
This controller handles adding candidate notes (possible numbers) to a specific Sudoku cell when the note mode is active. It validates user input, checks if note mode is enabled, and then updates the targeted cell's notes array. The logic implemented involves parsing row, column, and element values, inserting the note into the correct sub-array, and pushing this updated state onto the stack for future undo operations. By maintaining these state changes and validations, the controller ensures that notes are accurately reflected in the game's data model and can be easily reverted if necessary.

addNumberIntoBoard.js:
This controller is responsible for placing a user-selected number into a chosen cell of the Sudoku puzzle. It implements logic to parse the provided coordinates and element, update that cell on the board, and then verify if the insertion is currently valid by checking the corresponding row, column, and subgrid. If needed, the state is saved onto a stack for undo operations. Through these validations and stack manipulations, the logic ensures that each new number adheres to Sudoku rules and can be undone or reviewed later.

checkIfSolved.js:
This file's controller checks whether the Sudoku puzzle is fully solved. The logic involves iterating through every cell to confirm that no cell is empty and then verifying the entire board's correctness through a validation function. If all cells are filled and correct, it returns that the puzzle is solved. This ensures that the application can provide immediate feedback on the puzzle's completion status, helping confirm that the user's solution is correct and final.

correctSoFar.js:
This controller determines if the partially filled Sudoku board remains valid up to the current point. The implemented logic iterates through the board, comparing each cell's current value against the known correct solution. If any mismatches occur, their coordinates are collected and returned. This gives players a way to check their progress without revealing the entire solution, offering a helpful tool for detecting mistakes early and guiding corrective actions.

deleteElementFromBoard.js:
This controller allows the user to remove a previously placed number from a cell, effectively resetting it. The logic checks for valid cell coordinates and sets the cell's value back to -1, indicating an empty state. It updates the game's stack of states, ensuring that this removal action can be reversed if needed. By maintaining the board state and allowing for edits, this logic supports user-driven puzzle management and encourages iterative problem-solving.

deleteNotes.js:
This file's controller deals with removing a specific candidate note from a cell's notes array. The logic filters out the unwanted note from the cell's notes, saves the modified board state, and updates the stack to track this change. By structuring the logic around array manipulation and state updating, it preserves the puzzle's consistency and ensures easy reversibility of any note modifications.

callRandomHint.js:
This controller provides the user with a random hint by identifying either incorrect cells or empty cells and suggesting the correct number for one of them. The implemented logic checks the board to find either a wrong cell or the next empty cell and retrieves the correct solution number from the solution board. Using doubleStack() or a similar helper, the hint and updated state are applied to the game's stack, ensuring changes are tracked. This assists players who need help without immediately solving the entire puzzle.

callSpecificHint.js:
Here, the logic focuses on providing a hint for a specific cell chosen by the user. It verifies the cell coordinates, retrieves the correct number from the solution board, and then applies this suggestion to the puzzle's current state. By coupling logic for coordinate validation and solution retrieval with updates to the stack, the controller ensures that the user can receive targeted assistance without losing track of their game's state.

listAllGamesInDb.js:
This controller lists all stored Sudoku games in the database. The logic is straightforward: it queries the database for all game documents and returns them. By implementing this logic, the file supports administrative or debugging tasks, helping developers or players to examine available games or resume saved puzzles.

resetGame.js:
This controller resets the current Sudoku puzzle to its initial state by clearing all values and notes. The logic loops through the entire problem board, resetting each cell's value and notes, then updates the game record in the database. By implementing logic that clears cells and recalculates the board's stored state, this controller gives users the option to start over without losing puzzle structure.

undo.js:
The undo controller reverts the board to a previous state by manipulating the stack of recorded states. The logic involves popping the latest entry from the stack and restoring the previous one. This ensures that each user action can be rolled back if they realize they've made an incorrect move. By preserving a history of states and implementing logic that cycles through them, the file provides the fundamental undo functionality crucial to user-friendly puzzle-solving.

undoUntilCorrect.js:
This controller repeatedly undoes moves until the board returns to a correct state. The implemented logic checks the validity of the current top stack state and continues popping states until it finds a valid configuration. By applying iterative logic over the stack to restore correctness, it helps players backtrack from potentially complex errors to a point where they were still on the right track, promoting a more guided problem-solving experience.

2. Backend Routes

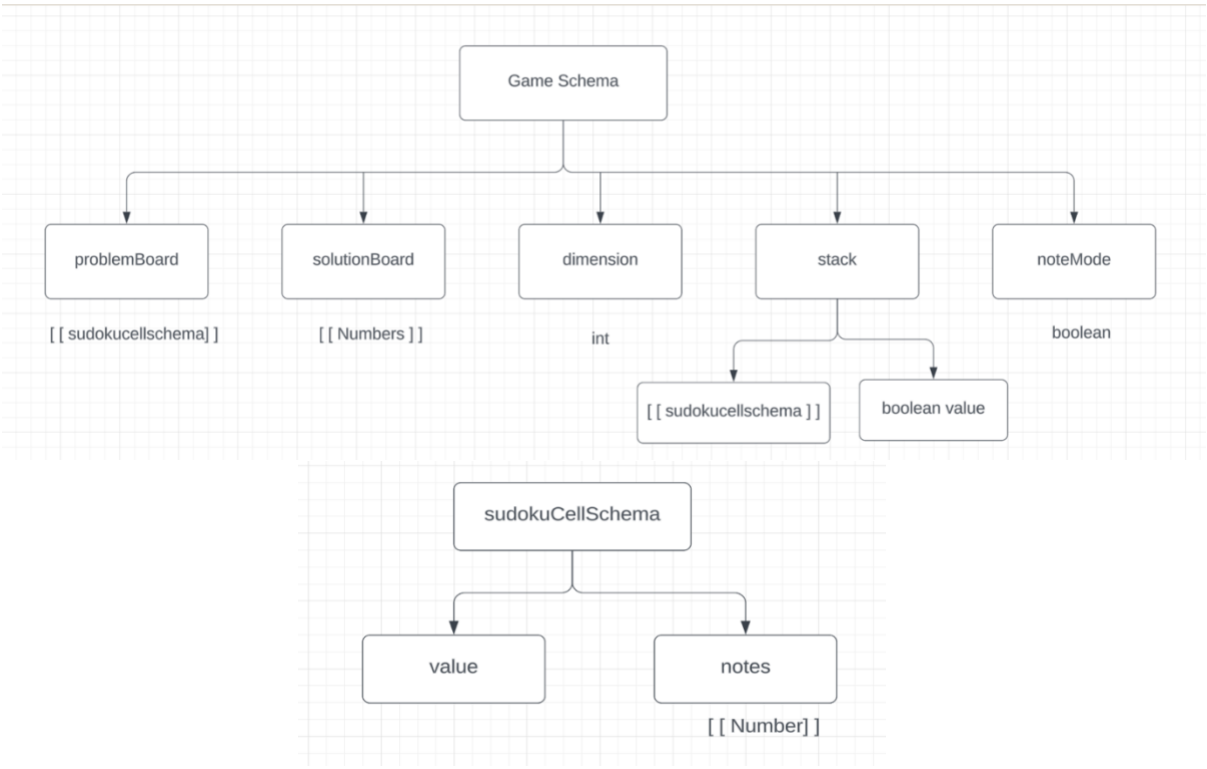
Auth Routes:
Defines endpoints for user registration and login. It integrates with authController.js to handle user credential validation, token generation, and user data storage.

Game Manipulation Routes :
Provides a variety of endpoints related to the Sudoku puzzle, such as adding numbers, deleting elements, checking if the board is solved, verifying correctness, undoing moves, toggling note mode, and fetching hints. These routes rely on corresponding controller functions to update the puzzle state, interact with the database via the Game model, and return the updated board and validation results.

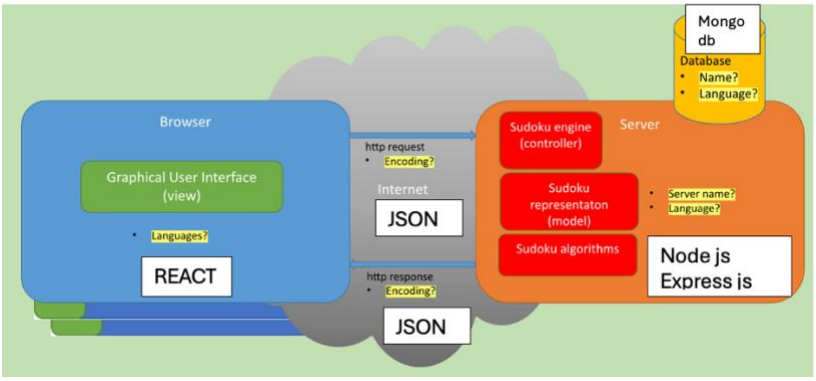
Game Retrieval Routes :
Offers endpoints for fetching individual game data or listing all stored games. By calling controller functions that query the MongoDB database, these routes supply the front end with the current puzzle state or archived game sessions for management or resumption.

Board Retrieval Routes :
Handles requests for new Sudoku boards of different dimensions (e.g., 4x4, 9x9). Calls the appropriate controller functions to generate or fetch puzzle configurations from the database or puzzle generation logic, then returns the newly created game state to the client.

3. MongoDB Database schema representation



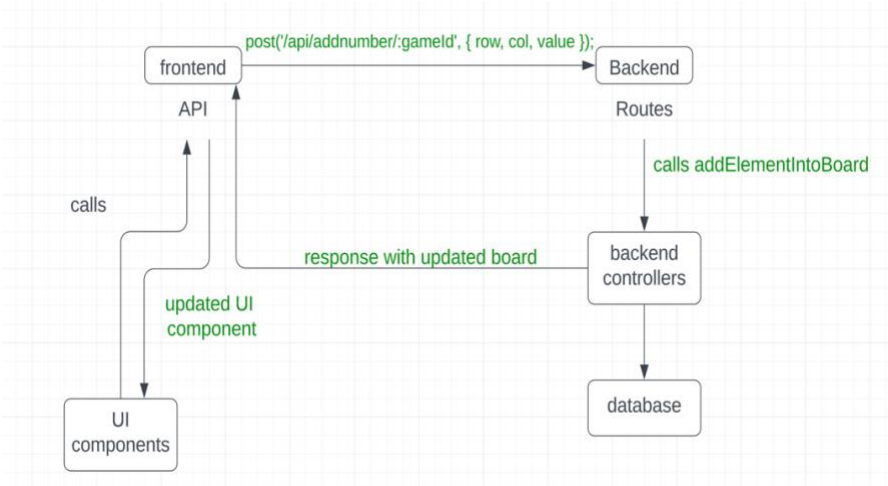
[III] Integration & communication of all components



We connect our React front end to the back end using a REST API and Axios. Axios streamlines the process of making HTTP requests from React components, ensuring smooth and efficient data exchange. By adhering to REST principles, the front end can asynchronously send and receive data, resulting in a dynamic, responsive user interface. This structure also maintains a clean separation between the presentation layer and server logic, enhancing both scalability and maintainability.

When handling URL-encoded data, the body-parser middleware is instrumental. Commonly used with frameworks like Express.js, it automatically parses and decodes incoming URL-encoded data from the front end. This ensures that all special characters and encoded values are accurately interpreted by the back end, allowing for proper processing of request bodies.

JSON encoding converts a JavaScript object into a JSON-formatted string. As a lightweight and human-readable data exchange format, JSON is easily parsed and generated by machines, making it ideal for transferring data between the front end and back end.
The provided code sets up an Express.js router with various API endpoints tailored for a game-oriented application. These routes enable operations like adding or removing items on the game board, validating the puzzle's correctness, requesting hints, undoing moves, and managing notes. Designed to handle parameters passed through the URL, the endpoints return responses detailing the validity of the operations, the current state of the game board, the action history (stack), and any recommended next moves. Collectively, these APIs offer a robust interface, ensuring smooth two-way communication between the React front end and the underlying game logic and data storage.

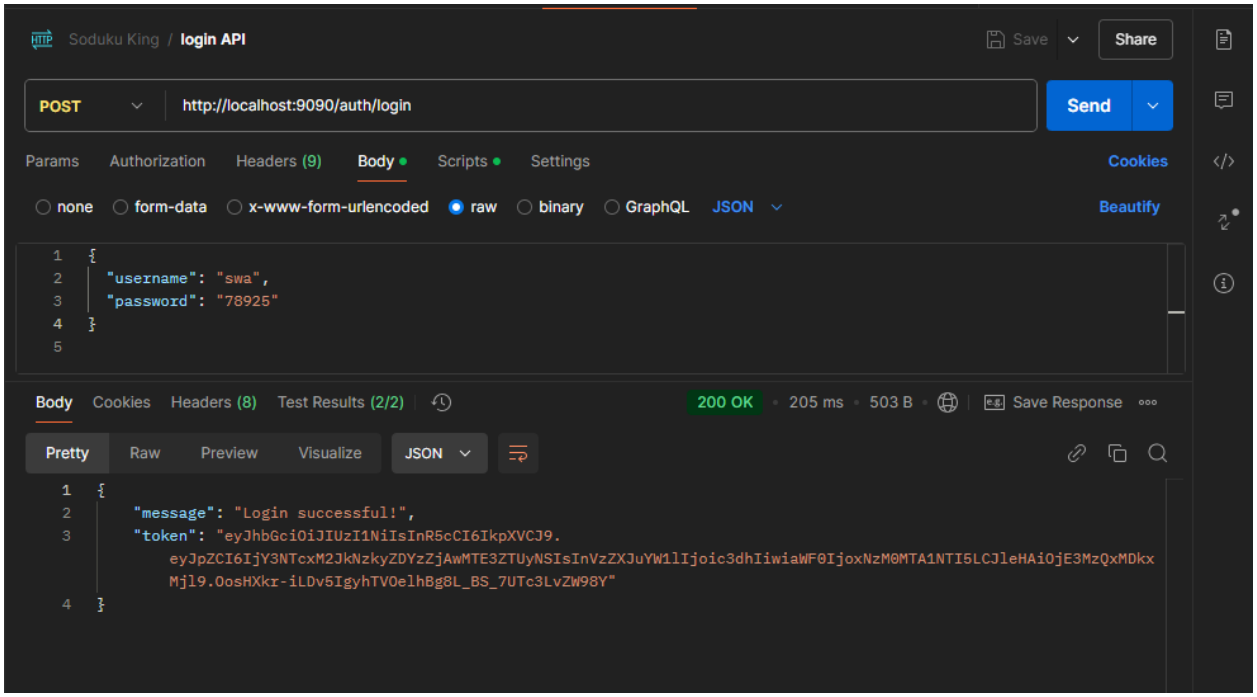


Chapter 5. Testing

Backend API Documentation

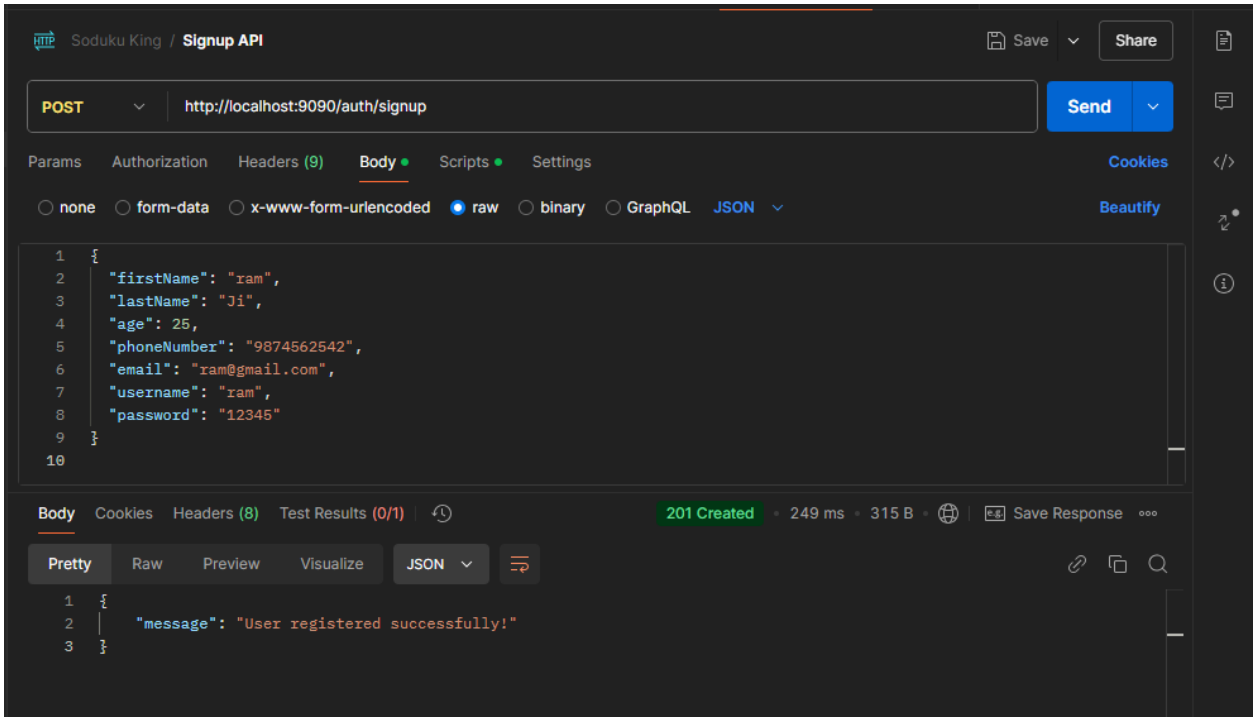
API Name: **login API**
Method: POST
URL: http://localhost:9090/auth/login
Request Body:

```
{
  "username": "swa",
  "password": "78925"
}
```

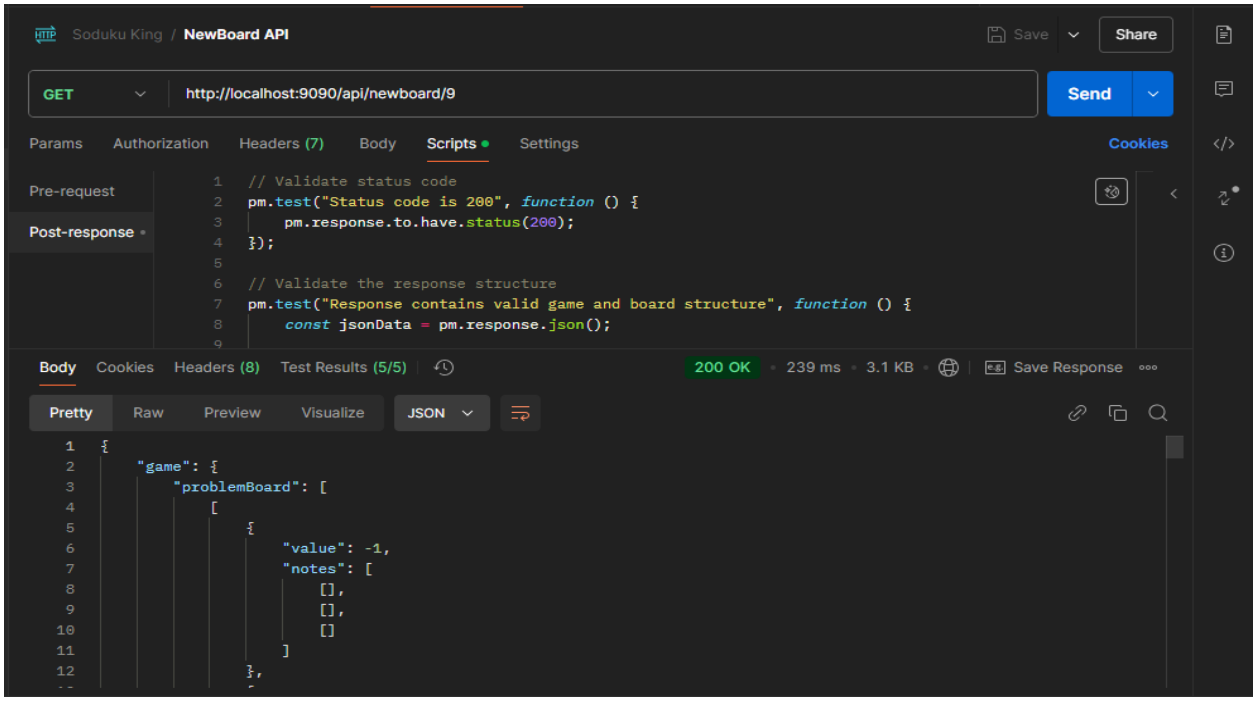


API Name: Signup API
Method: POST
URL: http://localhost:9090/auth/signup
Request Body:

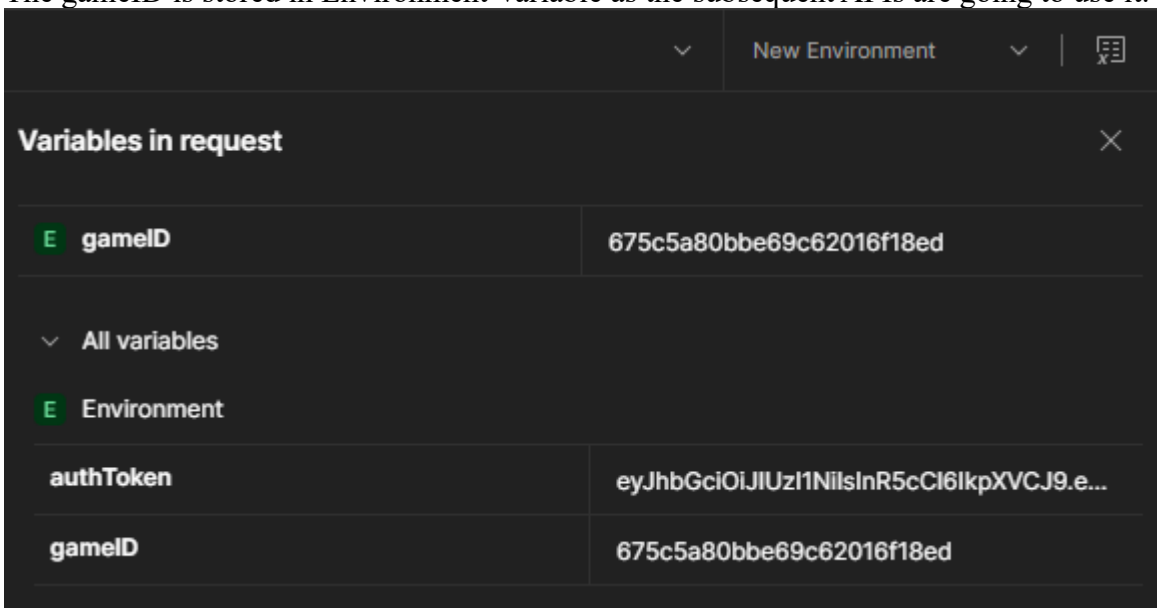
```
{
  "firstName": "ram",
  "lastName": "Ji",
  "age": 25,
  "phoneNumber": "9874562542",
  "email": "ram@gmail.com",
  "username": "ram",
  "password": "12345"
}
```



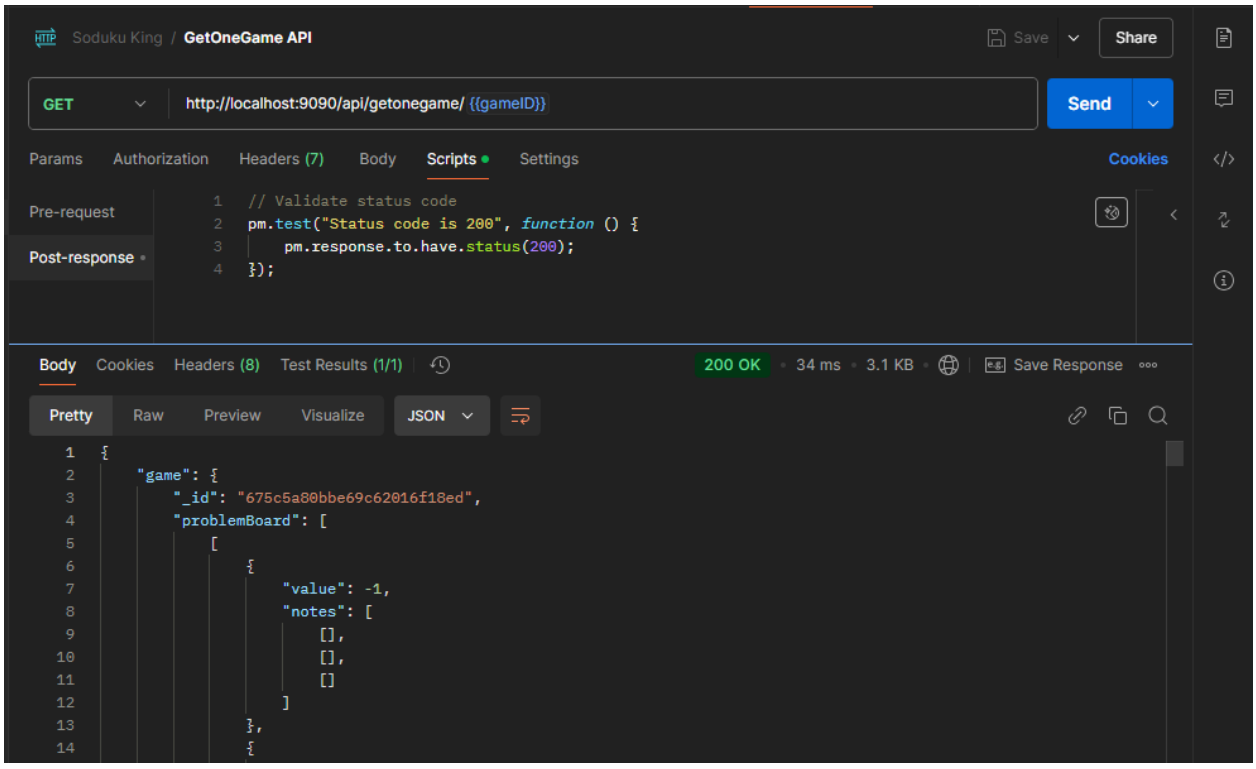
API Name: New Board API
Method: GET
URL: http://localhost:9090/api/newboard/9



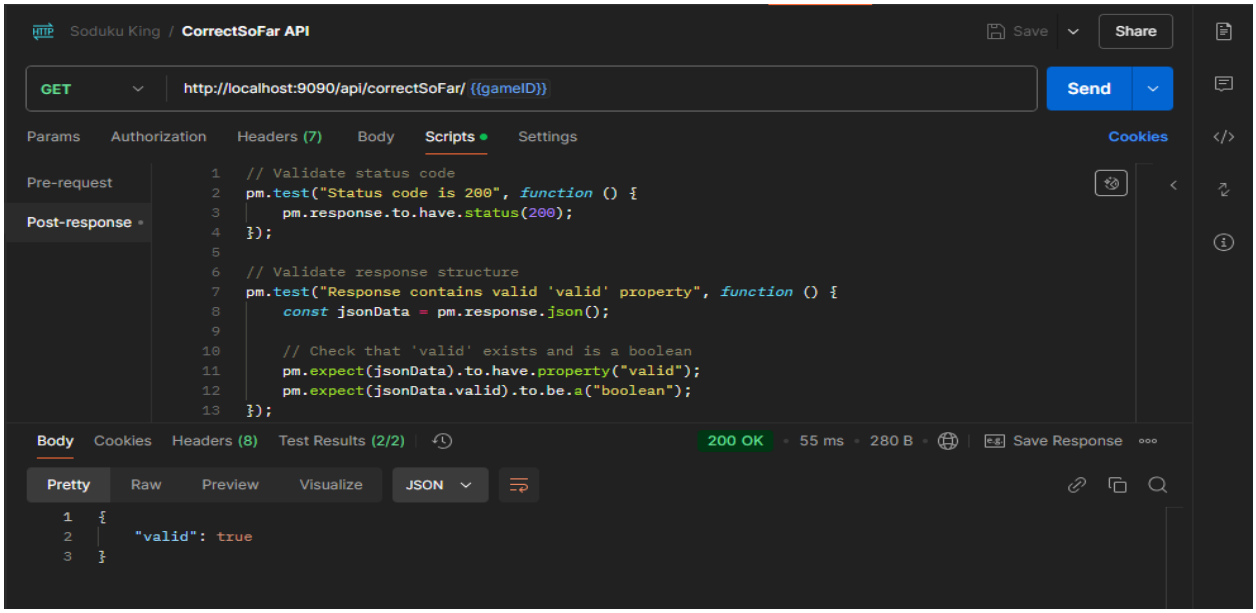
The gameID is stored in Environment Variable as the subsequent APIs are going to use it.



API Name: **GetOneGame API**
Method: GET
URL: http://localhost:9090/api/getonegame/{{gameID}}

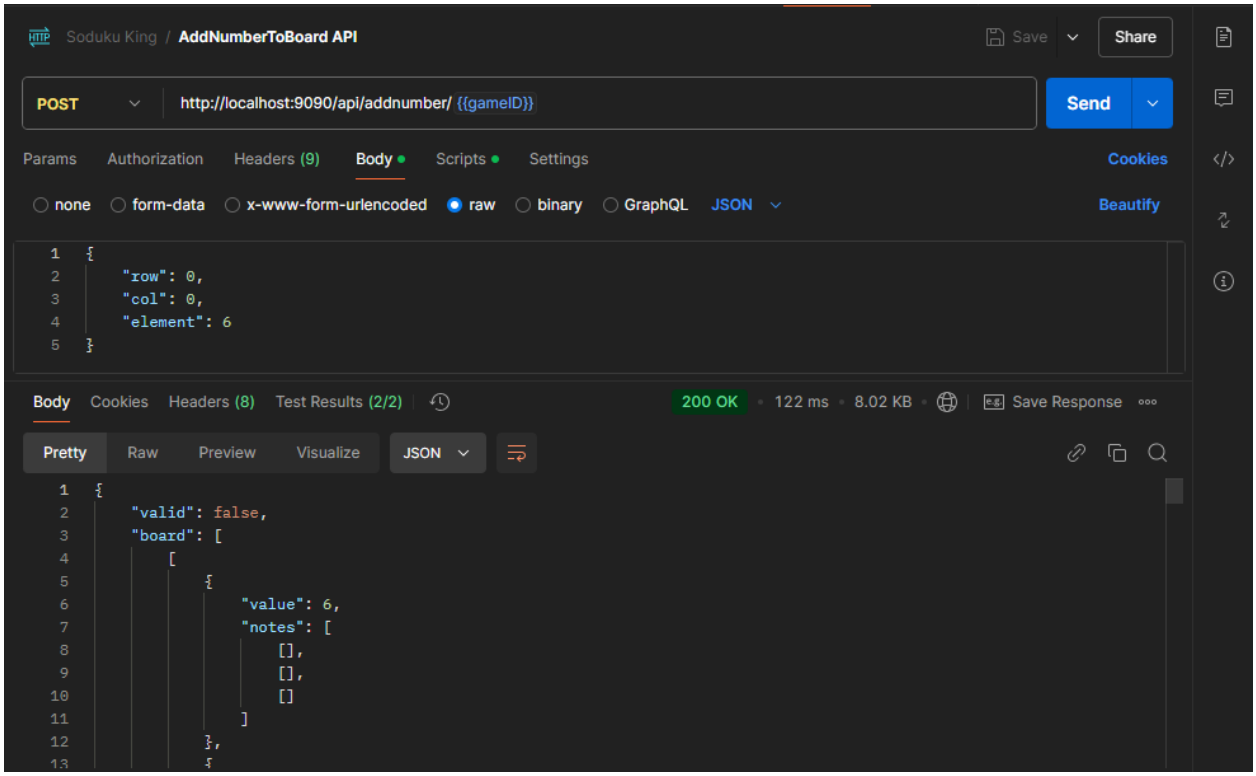


API Name: **CorrectSoFar API**
Method: GET
URL: http://localhost:9090/api/correctSoFar/{{gameID}}



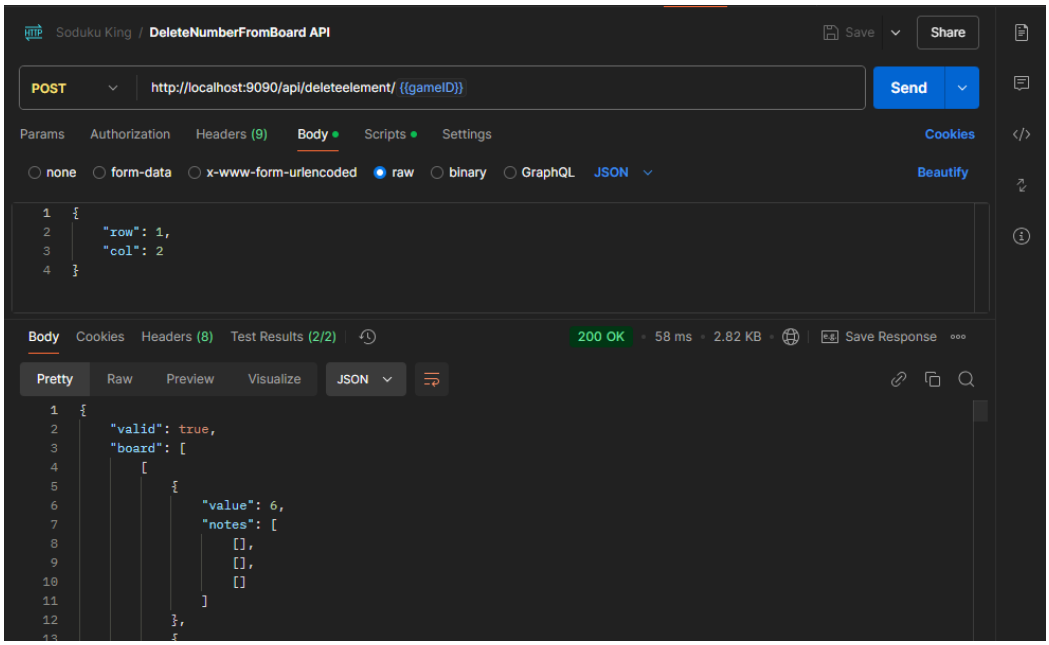
API Name: **AddNumberToBoard API**
Method: POST
URL: http://localhost:9090/api/addnumber/{{gameID}}
Request Body:

```
{
   "row": 0,
   "col": 0,
   "element": 6
}
```

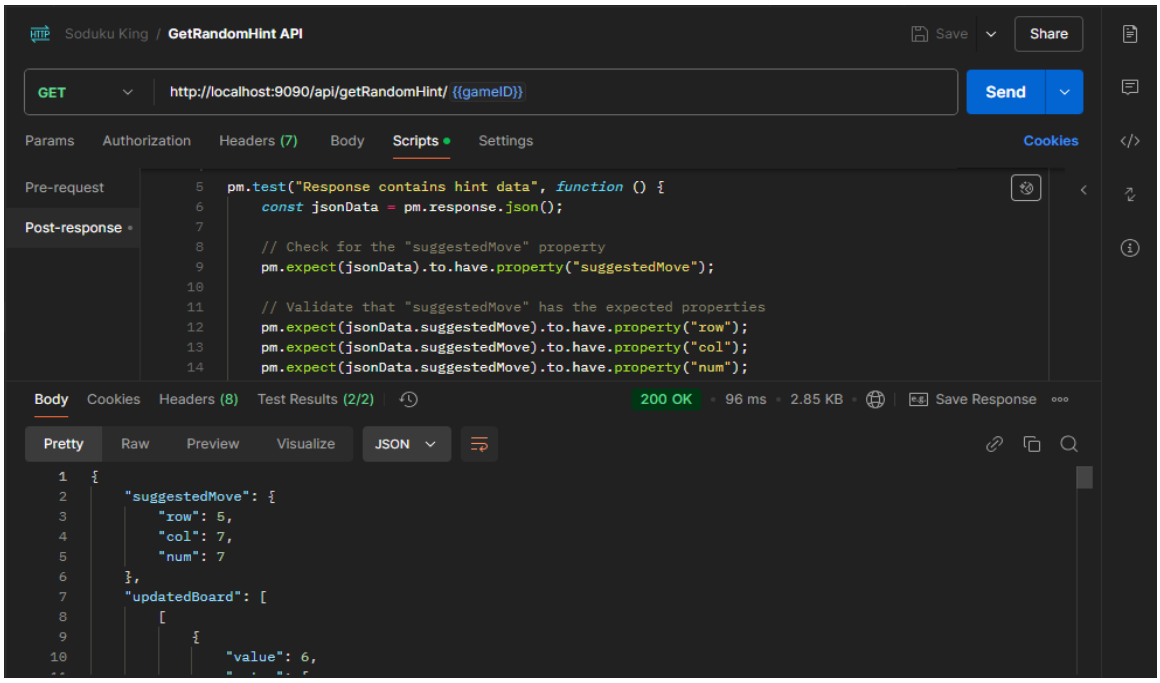


API Name: **DeleteNumberFromBoard**
Method: POST
URL: http://localhost:9090/api/deleteelement/{{gameID}}
Request Body:

```
{
   "row": 1,
   "col": 2
}
```

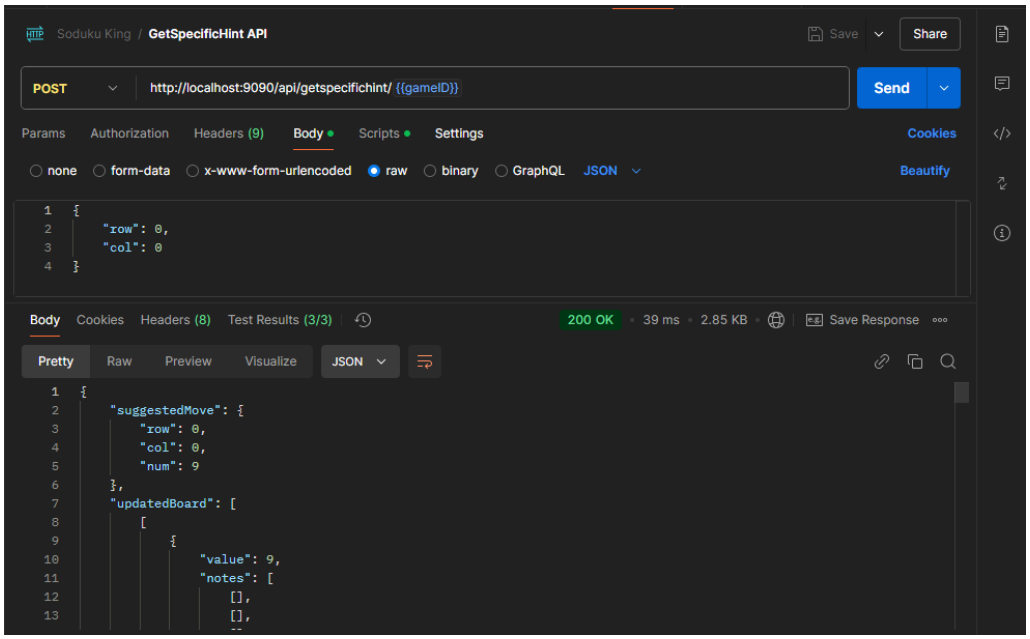


API Name: **GetRandomHint API**
Method: GET
URL: http://localhost:9090/api/getRandomHint/{{gameID}}

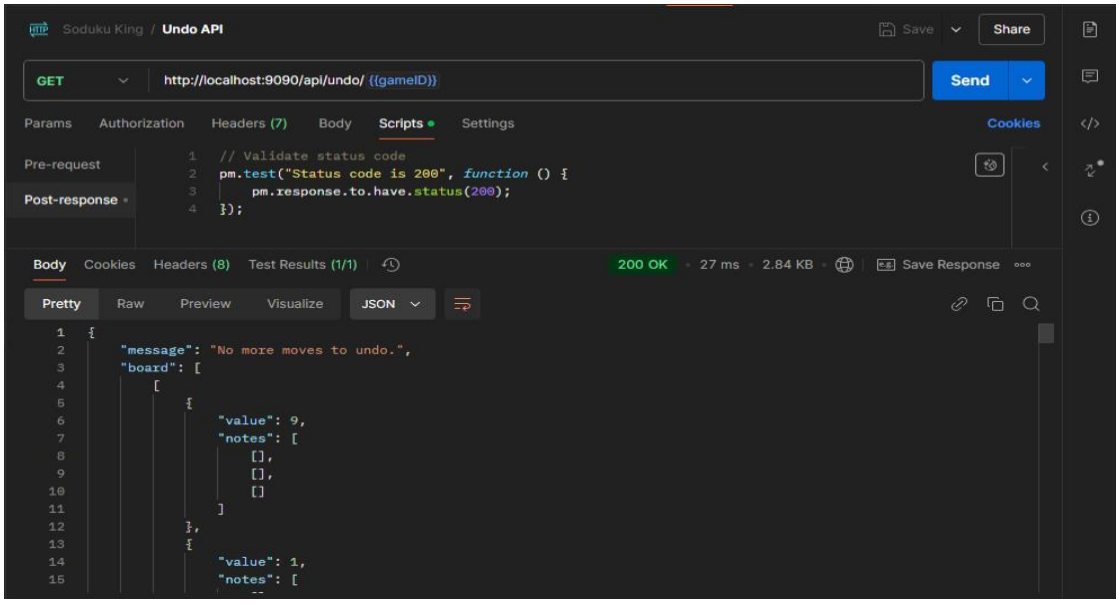


API Name: **GetSpecificHint API**
Method: POST
URL: http://localhost:9090/api/getspecifichint/{{gameID}}
Request Body:

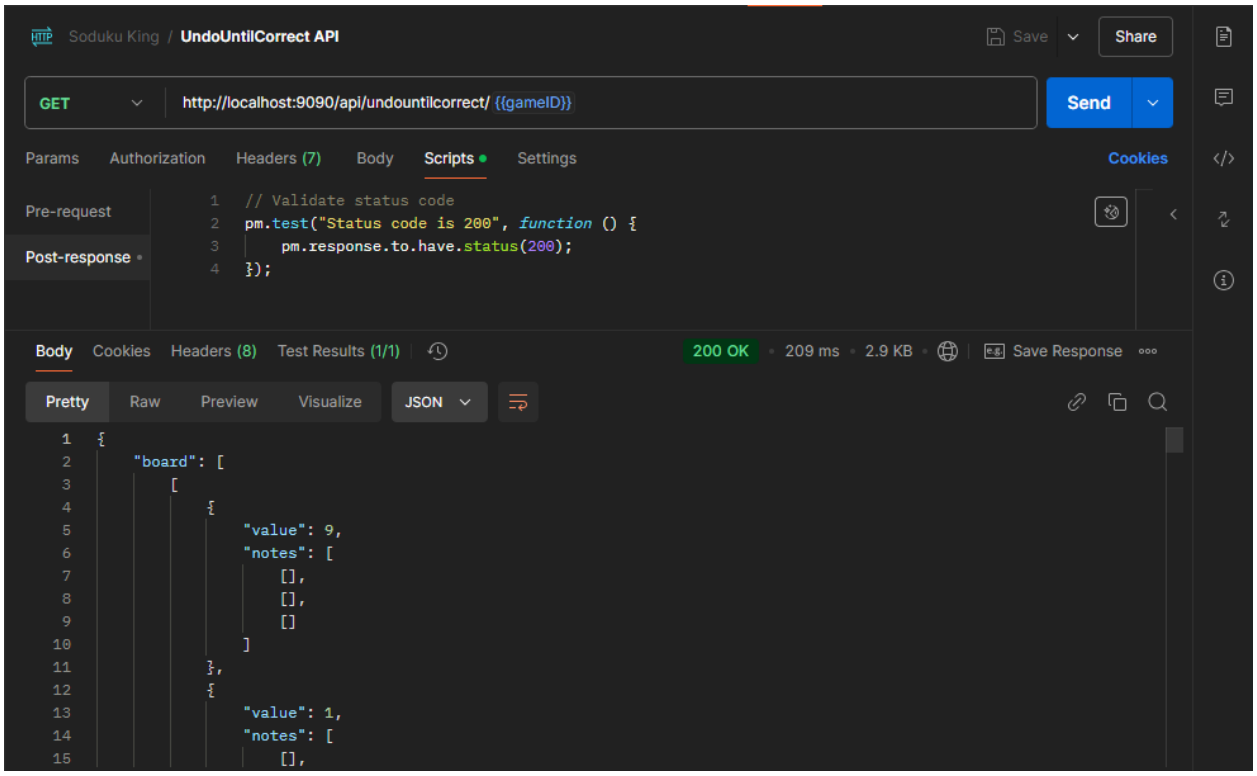
```
{
   "row": 0,
   "col": 0
}
```



API Name: **Undo API**
Method: GET
URL: http://localhost:9090/api/undo/{{gameID}}



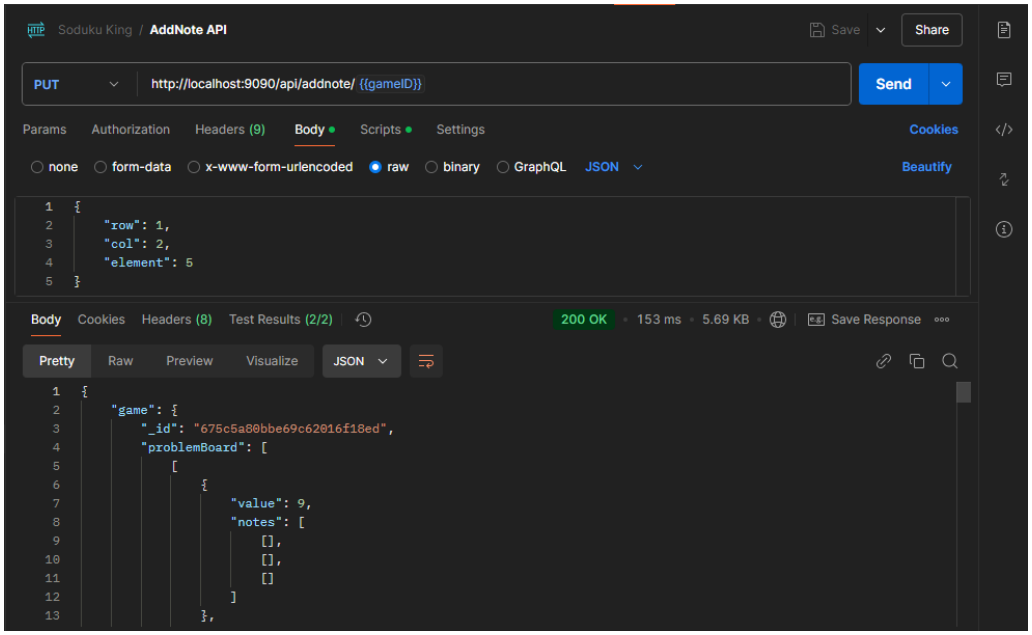
API Name: **UndoUntilCorrect API**
Method: GET
URL: http://localhost:9090/api/undountilcorrect/{{gameID}}



API Name: **NoteMode API**
Method: PUT
URL: http://localhost:9090/api/switchnote/{{gameID}}

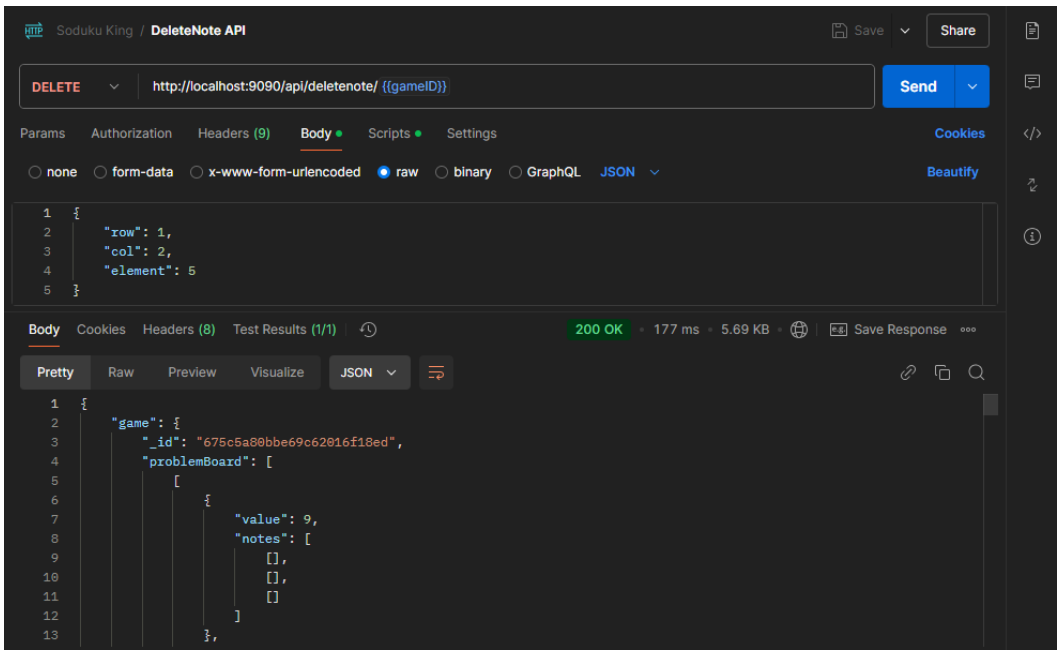
API Name: **AddNote API**
Method: PUT
URL: http://localhost:9090/api/addnote/{{gameID}}
Request Body:
{

```
  "row": 1,
  "col": 2,
  "element": 5
}
```

API Name: **DeleteNote API**
Method: DELETE
URL: http://localhost:9090/api/deletenote/{{gameID}}
Request Body:


```
{
  "row": 1,
  "col": 2,
  "element": 5
}
```

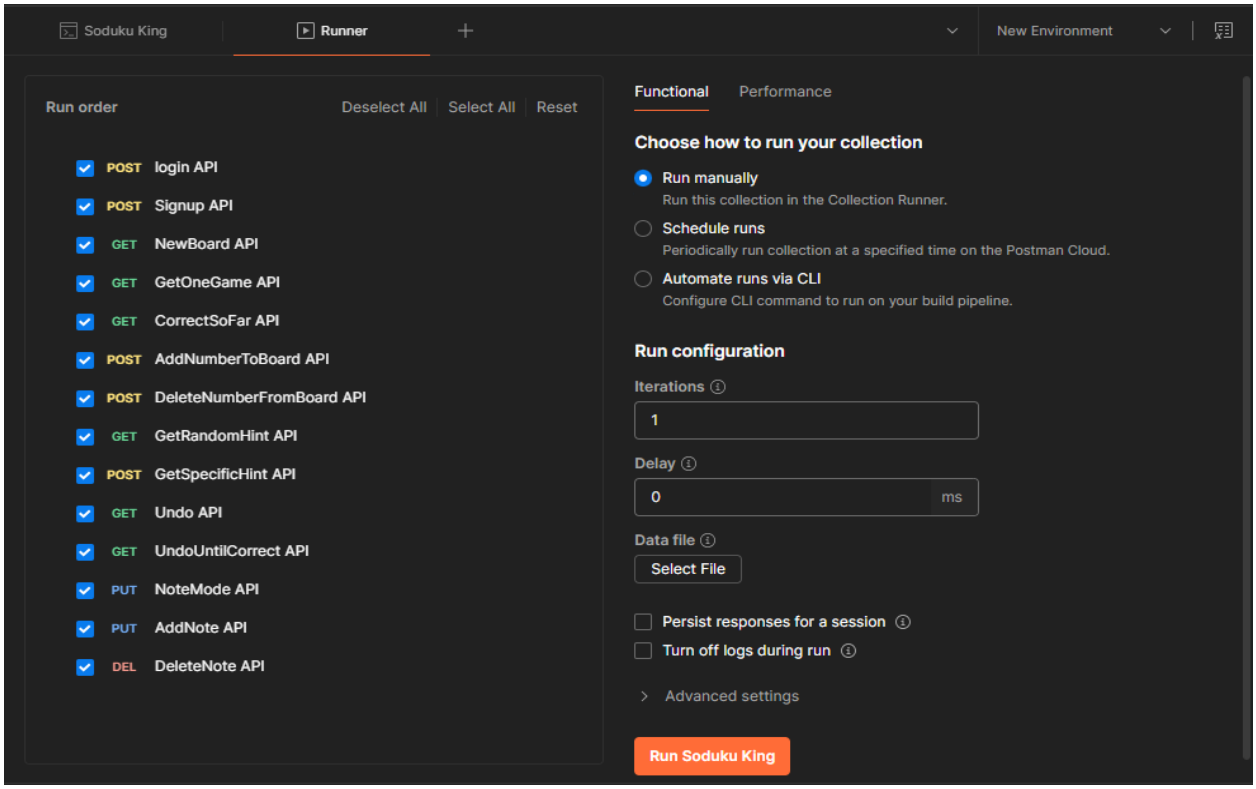



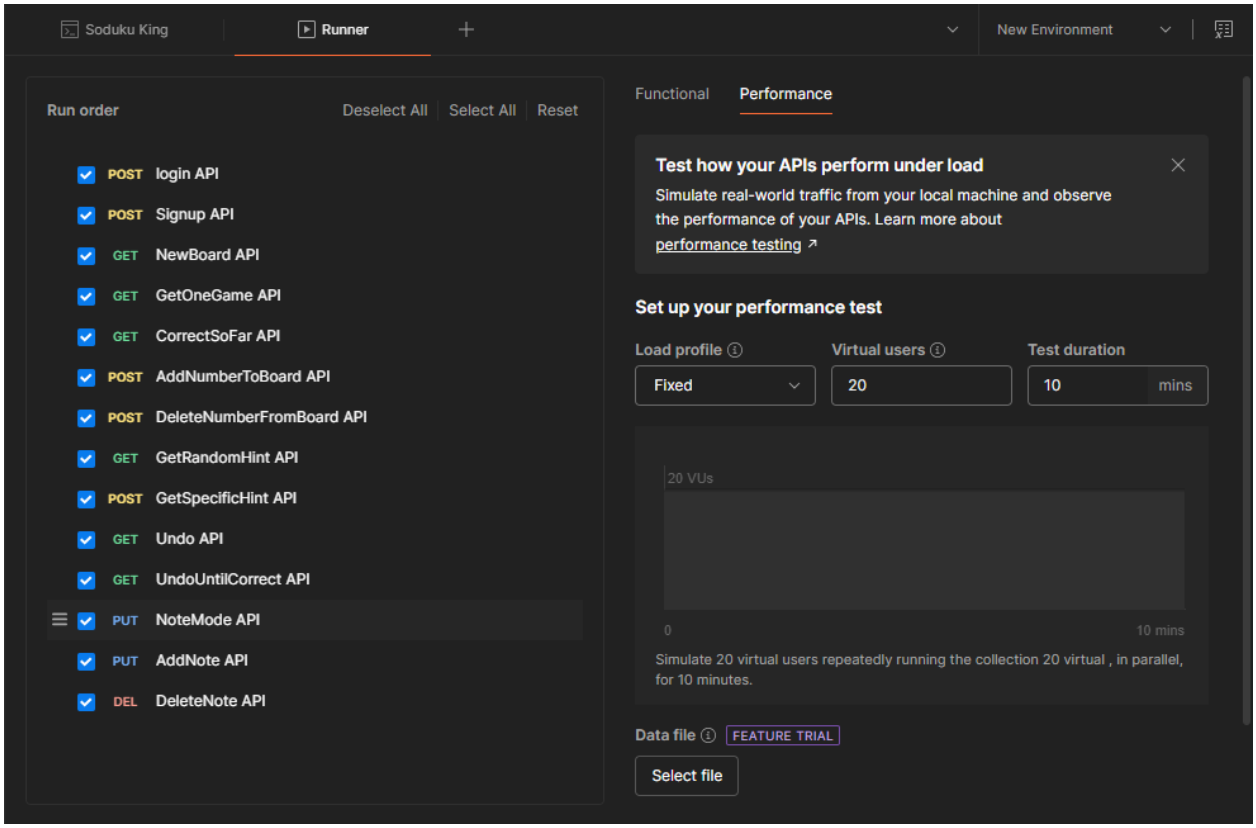
Automatic API Testing:

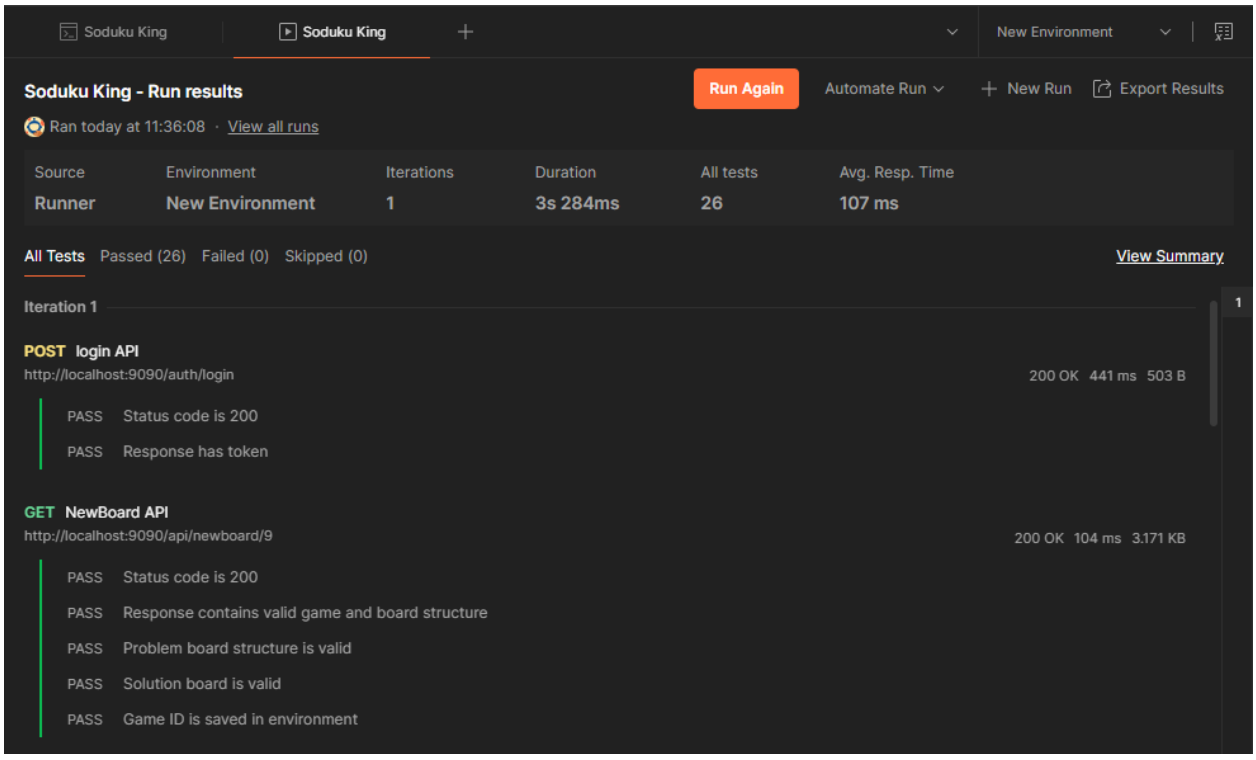
- **Choose APIs for Testing**
  - From the list of APIs (e.g., login API, Signup API, NewBoard API), check the boxes for the ones you want to include in the test run.
  - You can select all APIs or just specific ones, depending on what you want to test.

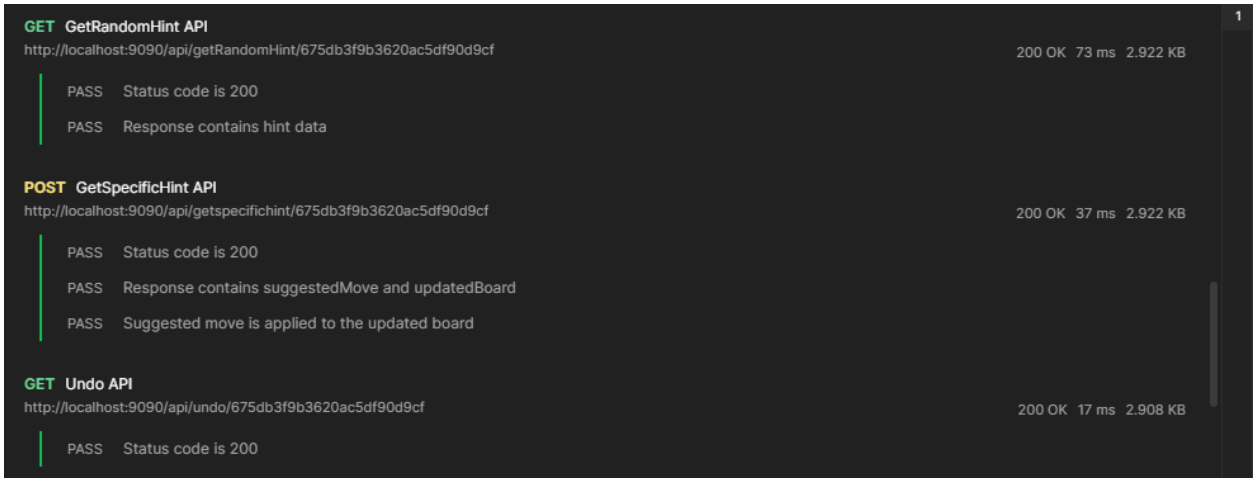
- **Configure Test Settings**
  - **Set Iterations:** Enter the number of times you want to run the selected APIs (e.g., 1 for a single run).
  - **Set Delay:** Specify a delay between API calls (e.g., 0 ms for no delay).
  - **Data File (Optional):** Upload a file if your APIs require dynamic data for testing (e.g., user credentials, game data).

- **Run and Review Results**
  - Click **Run Sudoku King** to start the test.
  - Postman will execute the APIs in the selected order, and results will display:
  - Status codes (e.g., 200 OK for success).
  - Validation checks (e.g., correct token, valid game board structure).
  - Analyze the outcomes to ensure the APIs work as expected.




Soduku King
Runner
New Environment
Run order
Deselect All
Select All
Reset
POST login API
POST Signup API
GET NewBoard API
GET GetOneGame API
GET CorrectSoFar API
POST AddNumberToBoard API
POST DeleteNumberFromBoard API
GET GetRandomHint API
POST GetSpecificHint API
GET Undo API
GET UndoUntilCorrect API
PUT NoteMode API
PUT AddNote API
DEL DeleteNote API
Functional
Performance
Test how your APIs perform under load
Simulate real-world traffic from your local machine and observe the performance of your APIs. Learn more about performance testing
Set up your performance test
Load profile
Virtual users
Test duration
Fixed
20
10
mins
20 VUs
0
10 mins
Simulate 20 virtual users repeatedly running the collection 20 virtual , in parallel, for 10 minutes.
Data file
FEATURE TRIAL
Select file


Soduku King
Soduku King
New Environment
Soduku King - Run results
Run Again
Automate Run
New Run
Export Results
Ran today at 11:36:08 · View all runs
Source
Runner
Environment
New Environment
Iterations
1
Duration
3s 284ms
All tests
26
Avg. Resp. Time
107 ms
All Tests
Passed (26)
Failed (0)
Skipped (0)
View Summary
Iteration 1
POST login API
http://localhost:9090/auth/login
200 OK 441 ms 503 B
PASS Status code is 200
PASS Response has token
GET NewBoard API
http://localhost:9090/api/newboard/9
200 OK 104 ms 3.171 KB
PASS Status code is 200
PASS Response contains valid game and board structure
PASS Problem board structure is valid
PASS Solution board is valid
PASS Game ID is saved in environment



Chapter 6. Deployment:

The application was deployed on Gurpreet's Arch Linux server using Node.js and libraries from npm. The project had several dependencies: Vite, React, and react-icons for the front end, and Nodemon, Express, and MongoDB for the backend.

Server Infrastructure:

⇒ Hosted on a dedicated Arch Linux server with:
2 Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GH, 20 cores total
64 GB memory (either DDR3 or DDR4 forgot which one, but they are with lower-end speeds)
Suitable for up to a few hundred to a thousand concurrent users, though not scalable to millions.

⇒ Configured with PM2 for process management, ensuring continuous application uptime.

⇒ Environment-specific variables were managed using .env files for sensitive configuration.

Security Measures:

⇒ Implemented fail2ban to protect against brute-force attacks.

⇒ Enabled automatic security updates through pacman to ensure the server remained patched against vulnerabilities.

⇒ Restricted SSH access with key-based authentication for enhanced security.

⇒ Configured a firewall to limit exposed ports, reducing attack surface.

⇒ MongoDB connections were secured with authentication and network restrictions.

Continuous Integration/Deployment:

- ⇒ Employed a Git-based deployment workflow, enabling seamless updates.
- ⇒ We wanted to deploy automatic builds being triggered on repository updates to ensure the latest changes are being deployed, but local and server tactics were different. Needed to manually deploy the changes by cloning & pulling from the repositories.
- ⇒ Included version control and rollback capabilities to address potential deployment issues. (These were done manually by creating backups.)

Challenges Faced During Deployment:

Initial Deployment:

- ⇒ **Dependency Compatibility:**
  Managing dependencies across both the frontend and backend was initially challenging due to mismatched versions of npm packages and conflicting requirements. For example, some libraries required Node.js 16+, but the server was initially running an older version. Updating Node.js without breaking existing configurations on Arch Linux required careful testing.

- ⇒ **Environment Setup:**
  Setting up MongoDB on Arch Linux posed challenges, as official guides were more aligned with Debian-based distributions. The absence of pre-configured packages meant manually compiling and configuring certain tools.

- ⇒ **Server Configuration:**
  PM2 configuration was initially problematic, as the application failed to restart properly after crashes. Tweaking PM2's ecosystem file to handle environment-specific variables and auto-restart resolved this issue.

- ⇒ **Firewall and Networking:**
  Opening only the necessary ports for HTTP traffic while securing MongoDB's port for local-only connections caused downtime during testing.
  Misconfigurations in ufw rules temporarily blocked access to the application. ufw is the standard/built in software on arch linux that handles the ports.

Final Deployment of the Finished Copy:

- ⇒ **Scalability Concerns:**
  We were unable to test the full capabilities of the server against volume of connections, even though several software can be used to test it's performance, we decided it would be a security risk to implement them. The server has only a select few things installed, any software can pose a risk, new ones maybe more than old ones.

⇒ **UI Responsiveness Issues:**
After finalizing the application, feedback indicated lag in rendering the Sudoku grid under certain conditions. Optimizations in React's rendering logic and minimizing state updates fixed these performance issues.

⇒ **Environment Management:**
Ensuring all dependencies were correctly installed in both the development and production environments introduced complexity. Some npm packages behaved differently in production due to optimizations enabled by Vite, leading to inconsistencies that required debugging.

Application Accessibility:

The application is now live at http://141.155.170.66:5173 and ready to handle a moderate user base with stability and security measures in place.

You can click on the link to access the game!!

Individual contribution :

Prarthana:
I designed and implemented RESTful API endpoints using Express.js and Node.js. This involved structuring controllers to handle key game operations—such as adding numbers, checking puzzle validity, generating hints, and enabling undo/redo functionality. By creating a clear separation of concerns, I ensured each endpoint focused on a specific part of the puzzle logic and also contributed in routing the frontend API to corresponding backend controller.

In addition, I contributed to designing the Mongoose schema for the Game model, which captures the puzzle's state, history (stack), and notes. This schema provided a strong foundation for data persistence, streamlining the storage and retrieval of previous game states and ensuring the backend remained maintainable and scalable.
I also implemented validation checks, synchronized state updates with the database, and ensured that features like note-taking and the undo stack worked seamlessly.

On the frontend, I improved the user interface by integrating React icons into the toolbar. This not only enhanced the aesthetics but also improved the user experience, making the puzzle tools more intuitive and visually appealing. I also contributed to integrating frontend API calls to efficiently fetch and update game data. This included implementing functionalities like boardManipulation.js for modifying the game board and notes.js for managing in-game notes.

Moreover, I also contributed to the conceptual and documentation aspects of the system. I created different diagrams across the documents including activity and class diagrams to clarify data flow and relationships between components.

Akash:

In the backend, I implemented the logic to add numbers to the Sudoku board, ensuring input validation and adherence to game rules. I developed robust authentication functionality, incorporating JWT-based sessions and password hashing for secure user management. Additionally, I designed a utility for managing undo/redo operations using a double-stack mechanism, and I created an endpoint to generate and fetch 4x4 Sudoku boards with varying difficulty levels. I also helped implementing functionality to provide specific hints for Sudoku board cells. Furthermore, I built logic to toggle between notes mode and number mode, enhancing the player's experience. In terms of managing the game state and data flow, I was involved in the development of the getBoard.js route, which fetched the current game board for users. As part of the backend development process, I also conducted comprehensive testing to ensure all functionalities worked as intended and met performance standards.

In the database, I designed the user schema with fields for username, email, hashed password, and timestamps to track creation and updates, ensuring a well-structured and secure database. I helped setup secure MongoDB connection, managing environment variable and handling connection error.

On the frontend, I created a service for fetching game details from the backend, complete with error handling and retry logic. I also contributed to build hooks to manage Sudoku grid state.I designed and implemented intuitive login and signup forms, including validation and seamless API integration for user authentication. I developed a keypad interface for number input and

note-taking functionality, providing users with a smooth and interactive experience. Through these contributions, I ensured the application was both functional and user-friendly.

Gurpreet:

In this Sudoku project, I had more focus on the frontend as well as deployment and infrastructure setup, as I started to wander off from functionalities to having more visible features on our sudoku game. For the backend as well, I contributed to several features, including the correctSoFar.js and deleteElementFromBoard.js controllers. I also worked on the generateBoard.js helper, which handled logic for generating Sudoku boards of varying difficulty levels, and helped refine features like undo.js and undo until correct to ensure users could easily revert mistakes and return to a correct game state.

On the frontend, I worked on key components that impacted user experience. My goal was trying to balance good looks with performance. I was involved in developing the Board.jsx component, which displayed the game grid, and ensured that it responded well to user interactions. It wasn't perfect, but it was responsive and had really good functionality. I also contributed to the design and functionality of the GameTimer.jsx, keeping track of the time spent on each puzzle, and the Keypad.jsx, which allowed users to input numbers onto the Sudoku grid. I also helped create the HomePage.jsx and MusicPlayer.jsx, enhancing the user interface with smooth navigation and additional interactive features. I worked closely on optimizing the visual and interactive aspects of the game, which required careful attention to detail in handling the game's dynamic nature.

Deployment and Infrastructure:

I was also involved in the deployment process, where I worked on the application's setup on a dedicated Arch Linux server. This included configuring Node.js, handling server dependencies, and resolving compatibility issues between the frontend and backend. I helped deploy the application using PM2 for process management, ensuring that the app remained up and running consistently.
One of the key challenges I faced was configuring MongoDB on Arch Linux, which lacked official support compared to Debian-based systems. But there was unofficial support which helped us successfully navigate through those challenges. I was also responsible for ensuring that MongoDB was configured securely, with proper authentication and network restrictions.
Additionally, I contributed to the server's security setup by implementing fail2ban, restricting SSH access, and configuring a firewall to secure the application against potential threats. These measures were critical in ensuring the stability and safety of the application, particularly in a live environment that is being hosted for the entire world to see.