Minesweeper Tech Design

Overview

Minesweeper is a logic-based puzzle game where players uncover tiles on a generated grid while trying to avoid mines. A key feature of this implementation is to keep the the game state and business logic server side in order to prevent the player from cheating. Apart from that, this Minesweeper implementation should be completely functional and bug-free (following all the rules of Minesweeper), along with having a great user interface and intuitive user experience.

Glossary

- **Tile:** a single cell in the Minesweeper game board that can be hidden, flagged, or opened
- **Board:** a grid of tiles that make up the playing field in Minesweeper
- **Mine:** explosive tile that then opened, the game ends and the player looses
- **Flag:** a marker placed on a tile to indicate a suspected mine
- **Game State:** progress of the game: (won, lost, running)
- P → feature is a priority (critical)
- NTH → feature is a nice to have (can be done once P items have been successfully done)

Player Requirements

- As a player, I want to start a new Minesweeper game with based on a set difficulty (P) or with custom dimensions (NTH)
- As a player, I want to reveal tiles and see whether they contain a number or a mine (in that case, game over) (P)
- As a player, I want to reveal tiles I believe are mines (P)

- As a player, I want to resume a Minesweeper game that I previously left based on a provided URL (P)
- As a player, I want to be the only player who can access my Minesweeper game (based on provided URL) (P)
    - As a player, I want to sign up to an account (P)
    - As a player, I want to log in so that I can access my Minesweeper games (P)
    - As a player, I want to sign out of my account successfully (P)

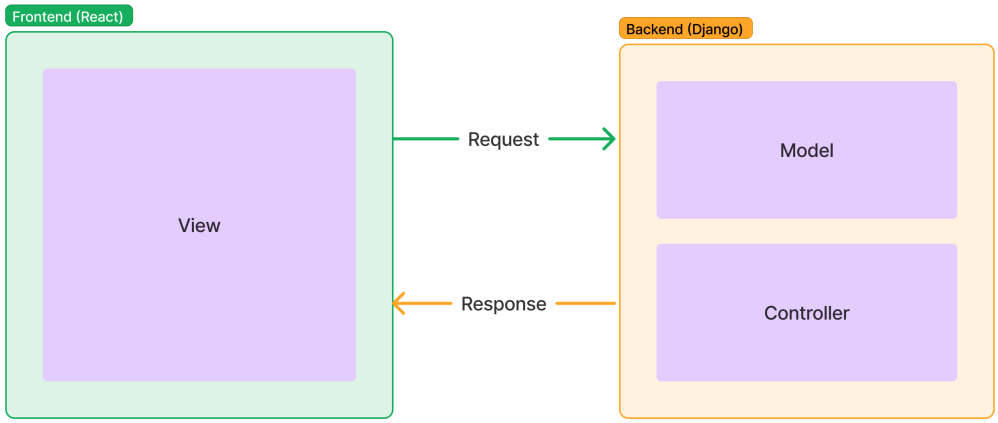
Technical Requirements

- Implement backend service using Python and Django
    - Generate random game board based on difficulty
    - Store game state on the backend
    - Validate all user moves on tile click (left and right) and send response to frontend
    - Handle user authentication (signing up, logging in/out)
- Implement frontend service using React
    - Render game board
    - Send API request to backend to update game state and react accordingly
    - Provide user interface to handle user authentication (signing up, logging in/out)

Solution

- Based on a modified MVC model/architecture:
    - Backend handles the Model and Controller (game state and logic, authentication logic)

- Frontend handles the View only (renders state provided by the backend/interactive elements of the game)



Backend

- Use `Django` and `Python` for backend API **(required)**
- Implement API using `REST` (`djangorestframework`)
  - Allows for speed in development compared to `GraphQL`
- Use `JWT` for user authentication
  - secure and easy to use, don't require database queries or lookups
  - Use `djangorestframework-simplejwt` library to help implement `JWT`
  - Use token blacklisting for sign out feature
- Enable CORS for now
  - Frontend will be communicating from a different origin
  - Need everything to work
  - If deploying, change these settings
  - Use `django-cors-headers`
- Authentication

    - Simple email and password authentication
    - based on an access and refresh token
    - Games are linked to a user and can only be accessed by that user
        - a user can only make game moves on their own games
- Game Mechanics
    - Used this video to help understand the game rules: https://www.youtube.com/watch?v=dvvrOeITzG8
        - Also Wikipedia: https://en.wikipedia.org/wiki/Minesweeper_(video_game)
    - Games are created based on difficulty:
        - Beginner → 9 × 9 with 10 mines
        - Intermediate → 16 × 16 with 40 mines
        - Expert → 30 × 16 with 99 mines
        - Custom game dimensions (up to 30 × 24, with a minimum of 10 mines and a max of `(num_of_rows - 1)(num_of_cols - 1)` mines) (NTH)
        - Based on information provided here: https://minesweepergame.com/strategy/how-to-play-minesweeper.php
    - Game board is represented using a 2D array of `Tile` objects (`Tile[][]`)
    - Needs to store all the information/state so that the game can be successfully resumed at any point

```
type Tile {
    value: str | int # either "m" for mine or a number from (
    is_flagged: bool, # whether the tile is flaged
    is_revealed: bool, # whether the tile is revealed
}
```

    - Functions are then used to edit the board as appropriate (revealing tiles, flagging them, etc)

    - The results of those edits are then sent to the frontend with a “sanitized” version of the board that is just necessary for rendering (does not reveal intimate details about the board like the positions of the mines)

- Important Functions
    - `genarate_board(difficulty: DifficultyEnum) -> (Tile[][], int[][])`
        - `enum DifficultyEnum = BEGINNER | INTERMEDIATE | EXPERT`
        - use a dictionary to store the related row, column and number of mines with their difficulty

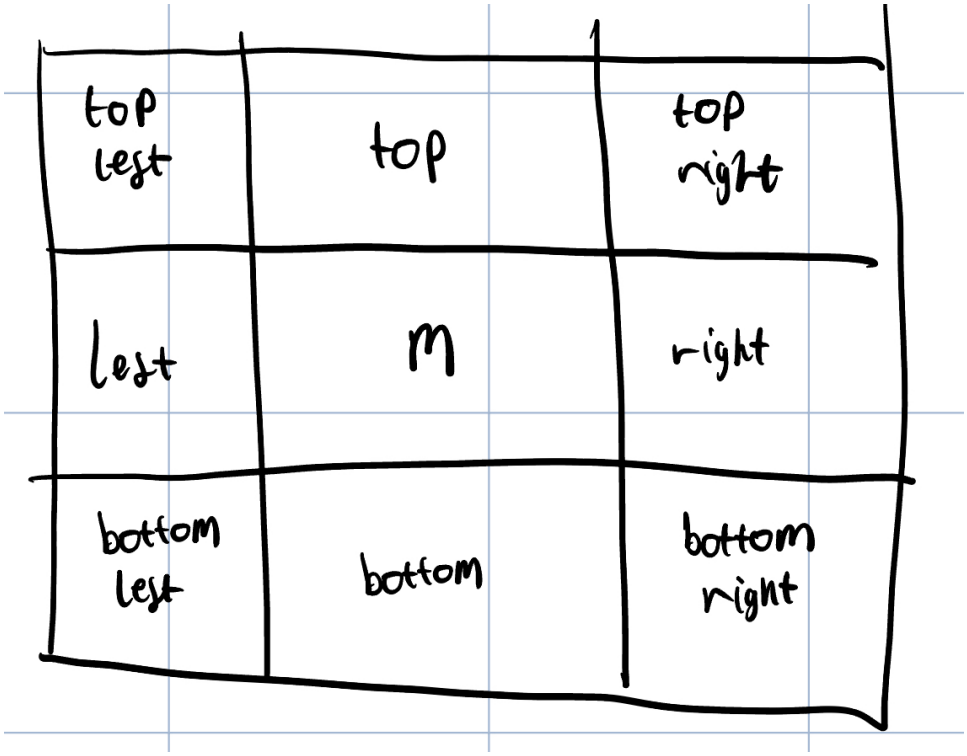
```
difficulty_map = {
    DifficultyEnum.BEGINNER: (9, 9, 10),
    DifficultyEnum.INTERMEDIATE: (16, 16, 40),
    DifficultyEnum.EXPERT: (30, 16, 99),
}

...

(row, colunm, mines) = difficulty_map.get(DifficultyEnum.E
```

        - function to generate a game board based on the provided difficulty.
        - Will internally use the `generate_custom_board()` function and return its result
    - `generate_custom_board(row: int, cols: int, mines: int) -> (Tile[][], int[][])`
        - function to generate a board based on the provided number of columns, rows, and mines
        - returns the board, along with an array of the mine coordinates
        - Algorithm steps
            - Validate provided dimensions and number of mines
            - initialize the board (2D array with “blank” tile objects)
            - place the mines, ensuring no collision

- as each mine is placed, update the adjacent tiles values




```
# top_left (-1, -1)     |   top (-1, 0) | top_right (-1, 1)
# ____________________________________________________
# left (0, -1)          |    m  (0, 0)  | right (0, 1)
# ____________________________________________________
# bottom_left (1, -1) | bottom (1, 0) | bottom_right (1, 1)
```

- `reveal_tile(row: int, col: int) -> bool`
  - function to reveal a tile (and its neighbours if empty)
  - returns a `bool` of whether the tile is a mine or not
  - if the tile is empty (0), then all of its neighbours will then reveal themselves
- `flag_tile(board: Tile[][], row: int, col: int) -> void`
  - function to toggle the flag of a tile in the board
  - `board[row][col]["is_flagged"] = not board[row][col]["is_flagged"]`
- `get_sanitized_board(board: Tile[][]) -> string[][]`

        - This function takes the provided board and sanitizes it be an 2D array of strings instead. This would be the variant of the board that the front end will receive
    - `has_won(board: Tile[][]) -> bool`
        - function to check if the user has won after making a move based on the state of the board

- File Structure
    - `/api` → app to hangle Django API routes
    - `/authentication` → app for all things authentication and models
    - `/game` → app relating to all the game mechanics/logic and models
    - `/core` → main app where everything is wrapped together (project settings and other things)
- Models
    - `User` (based on provided model by Django)
        - use `uuid4` for id
    - `Game`
        - `id: uuid4`
        - `difficulty: "beginner" | "intermedate" | "expert"`
        - `num_of_rows: int` → determined by the difficulty but can be used for custom boards (NTH)
        - `num_of_colunms: int` → determined by the difficulty but can be used for custom boards (NTH)
        - `num_of_mines: int` → determined by the difficulty but can be used for custom boards (NTH)
        - `board: JSON`
            - Using JSON for the board for multiple reasons

        - Allows for performant querying and writing to the database (better than querying multiple other tables for tiles and performing joins
        - Allows for easy management in business logic (2D Array)
        - Encapsulates the board structure/state with its appropriate `Game` record (a separate table of tiles stores tiles from different games together)
        - Can still query the JSON if needed
    - `status = "won" | "lost" | "running"`
    - `created_at: DateTime`
    - `last_saved: DateTime`
    - `user: uuid4`
        - a user can have many games, but a game can only belong to one user (1:N / one-to-many relationship)
- API Routes
    - Game
        - `POST /api/new_game` → create a new game
        - `GET /api/game/<game_id>` → get the current state of the game
        - `POST /api/game/<game_id>/reveal` → reveal a tile
        - `POST /api/game/<game_id>/flag` → flag a tile
    - Authentication
        - `POST /api/auth/signup` → sign up as a new user (and logs them in)
        - `POST /api/auth/signout` → sign out as a user
        - `HPOST /api/auth/token/` → Get the authentication token (log in)
        - `POST /api/auth/token/refresh` → refresh the authentication token to keep the user logged in

Frontend

- use `React` for the frontend **(required)**
  - TypeScript for type safely
- Use plain CSS for styles
  - Not a UI library due to set up time
  - Use shadcn/ui for inspiration and headlessui for UI components
- Structure
  - `src/` → source folder
    - `pages` → directory for all the pages of the application
      - `Page` → page directory (example)
        - `index.tsx` → index file where page is imported
        - `Page.tsx` → main page component file
        - `Page.css` → style file
        - `types.ts` → type definitions if page needs them
        - `components` → child components the page uses (see below for basic structure)
    - `ui` → folder for all UI components (inputs, modals, buttons)
      - `Component` → component directory (example
        - `index.tsx` → index file where the component is exported
        - `Component.tsx` → main component file
        - `Component.css` → component styles
    - `services` → services that the project can use
      - `api` → module for general `axios` calls
      - `auth` → module for managing authentication calls
        - `index.ts` → module implementation
        - `types.ts` → types for auth module
      - `game` → module for making game API calls

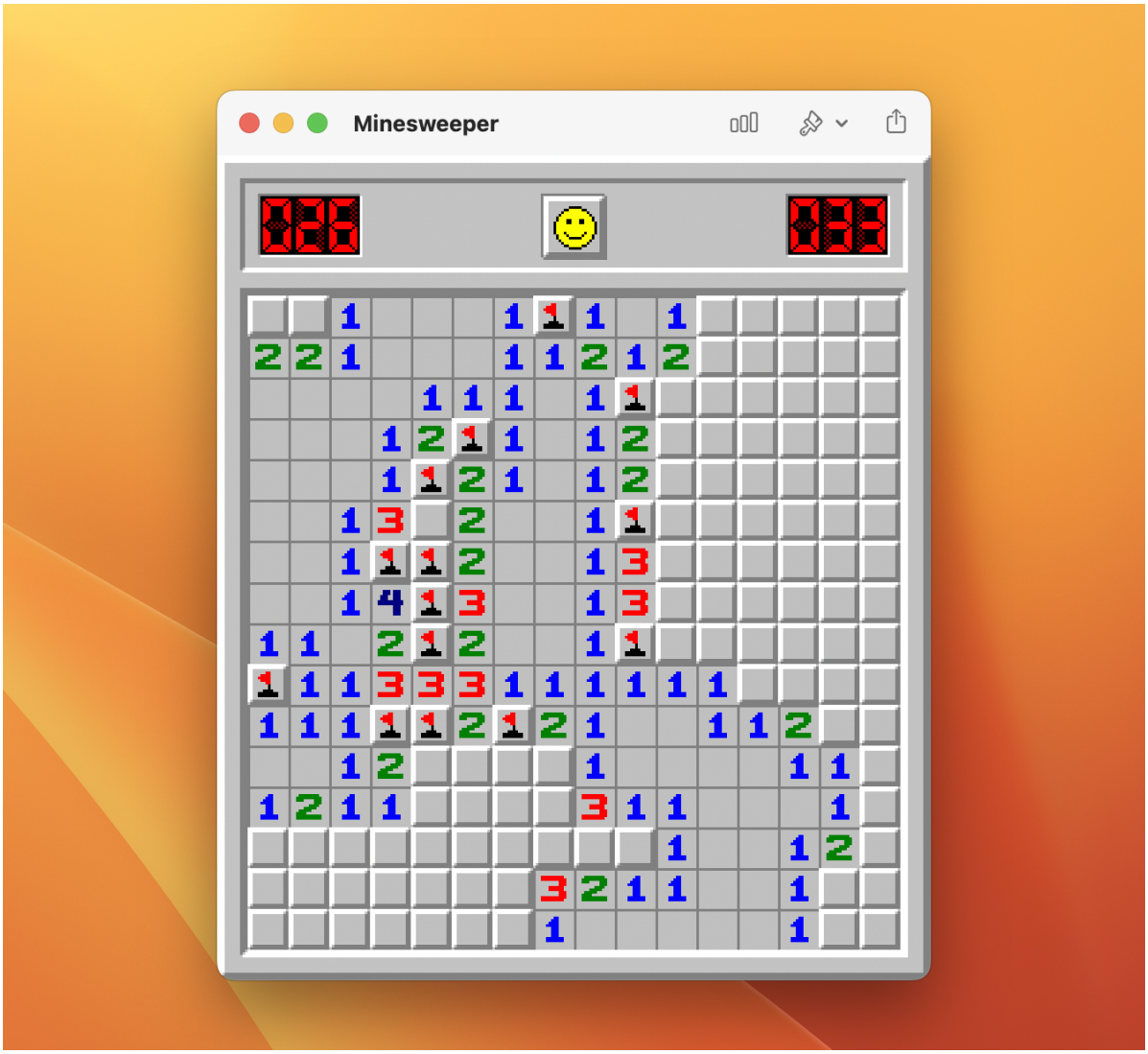
        - `index.ts` → module implementation
            - ex: `game.revealTile(options: RevealTypeOptions): Promise`
        - `types.ts` → types for game module
    - `hooks` → folder for all the hooks
    - `App.tsx` → root component file
    - `index.tsx` → root index file (renders root component)
    - `App.css` → root style file (css overrides/resets, basic styling)

- Components
    - `PublicRoute` → `Route` wrapper for public routes (explained below)
    - `PrivateRoute` → `Route` wrapper for public routes (explained below)
    - `ui`
        - `Button` → generic button
        - `Input` → generic input (can also use a label if needed)
        - `NavBar` → navbar present throughout the UI
            - Link to minesweeper tutorial
            - link back to the landing page
            - Renders a different button based on the page (NTH)
        - `NewGameModal` → modal component to create a new game
    - `pages`
        - `LandingPage` → general landing page
        - `HomePage` → home page for authenticated users
        - `LoginPage` → page to log in
        - `SignUpPage` → page to sign up
        - `GamePage` → page that renders the game
            - parses the `game_id` from the URL

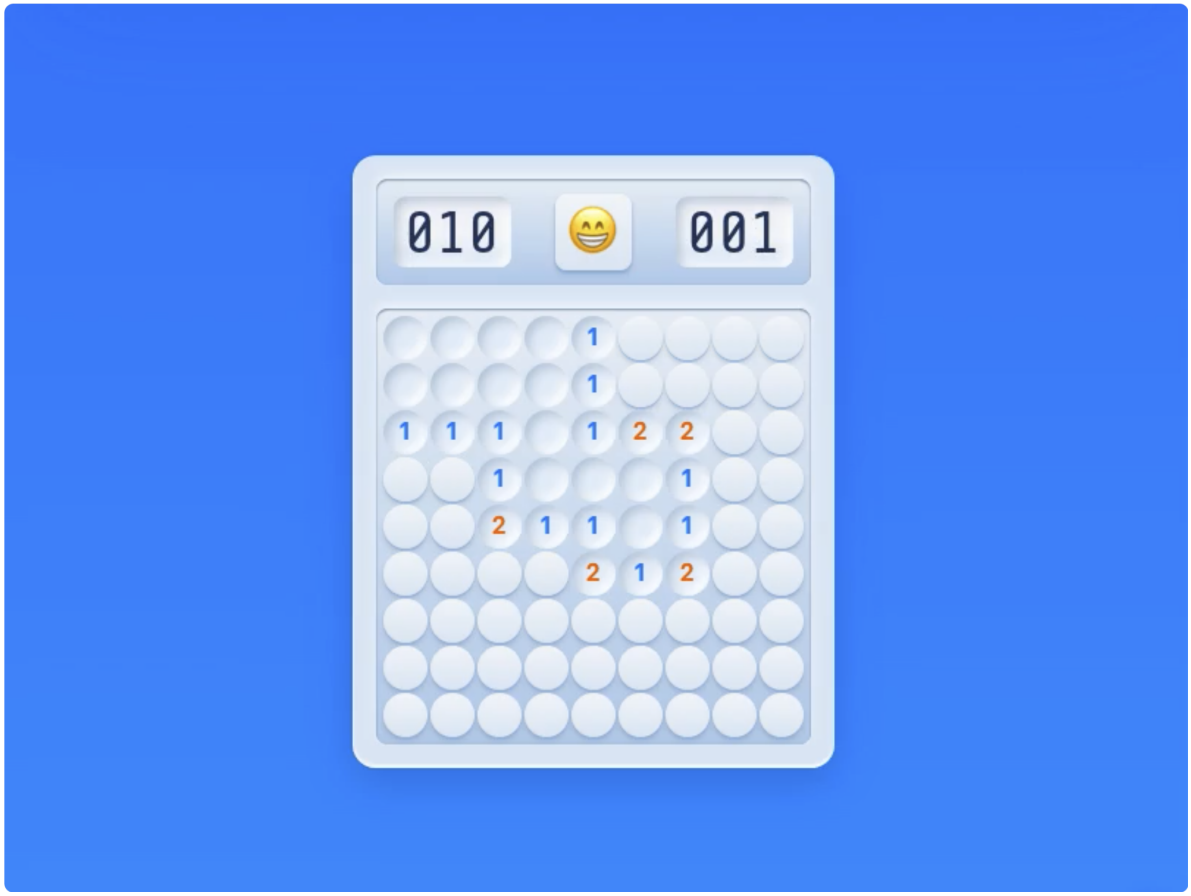
        - `MineSweeper` → actual minesweeper component
            - `Board` → component to render the board grid
                - Use CSS Grid to layout the tiles
            - `Tile` → component for each tile in board
                - will accept a `onRightClick` and `onLeftClick` prop
                - also a `value` prop
- Hooks
    - `useIsAuthenticated` → hook to check if the user is authenticated
- Routes
    - 2 types of routes:
        - Private → can only be accessed when authenticated and should redirect to `/login`
        - Public → can be accessed when not authenticated and should redirect `/home` if it is
    - `/` → `Landing` page
    - `/home` → `Home` page (Private route)
    - `/login` → `Login` page (Public route)
    - `/signup` → `SignUp` page (public route)
    - `/game/<game_id>` → `Game` page (Private route)

Design

- Idea: modern, intuitive UI
    - Flat design elements
    - monochromatic
- Modern interpretation of the classic game



- Muted color palete
  - Based off on Tailwind colors
- Light and dark mode (NTH)


010
001

- Mockups


Mine sweeper
Login
Username
Password
Login
Hello, User
Difficulty
Duration
Resume
Hellow User!
You have no game
Start new game
New Game
Rows
Cols

Tech Stack

Design

- `Concepts` (initial mockups)

Frontend

- `React` (requirement)
- `TypeScript` (type safety)
  - `TSX` for React components
- `Prettier` and `ESLint` (for ensure high code quality)
- `CSS` (styling)
  - Could use Chakra UI for rapid UI development)
- `Axios` (library to send REST requests to server)
- `react-router-dom` for page routing

Backend

- `Python` (requirement)
- `Django` (requirement)
  - Django REST Framework (for implementing REST API)
  - `djangorestframework-simplejwt` (for JWT support)
- `Flake8` and `Black` (for high level code quality)
  - Decided to use Flake8 over `Pylint` due to speed