Documentation of Searcher Board | Advanced Databases Project

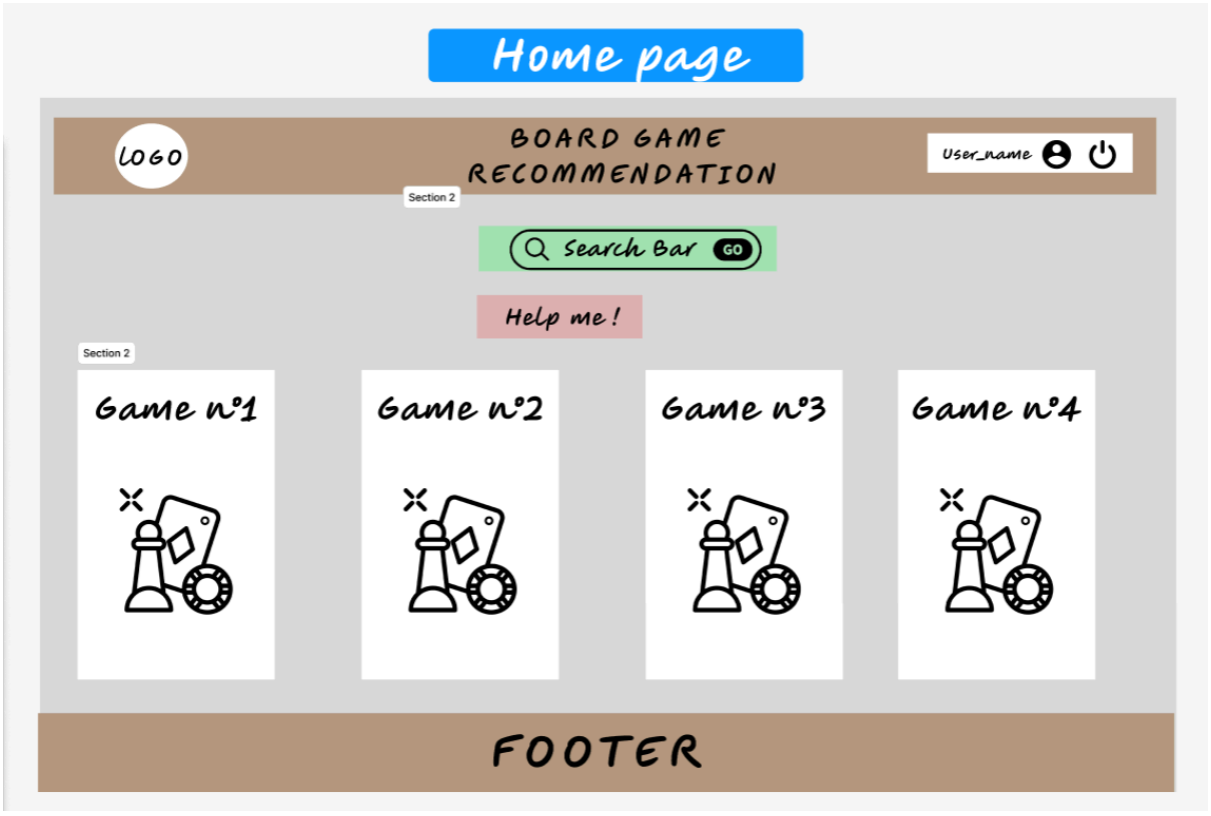
1. Application functionalities and UI mockups

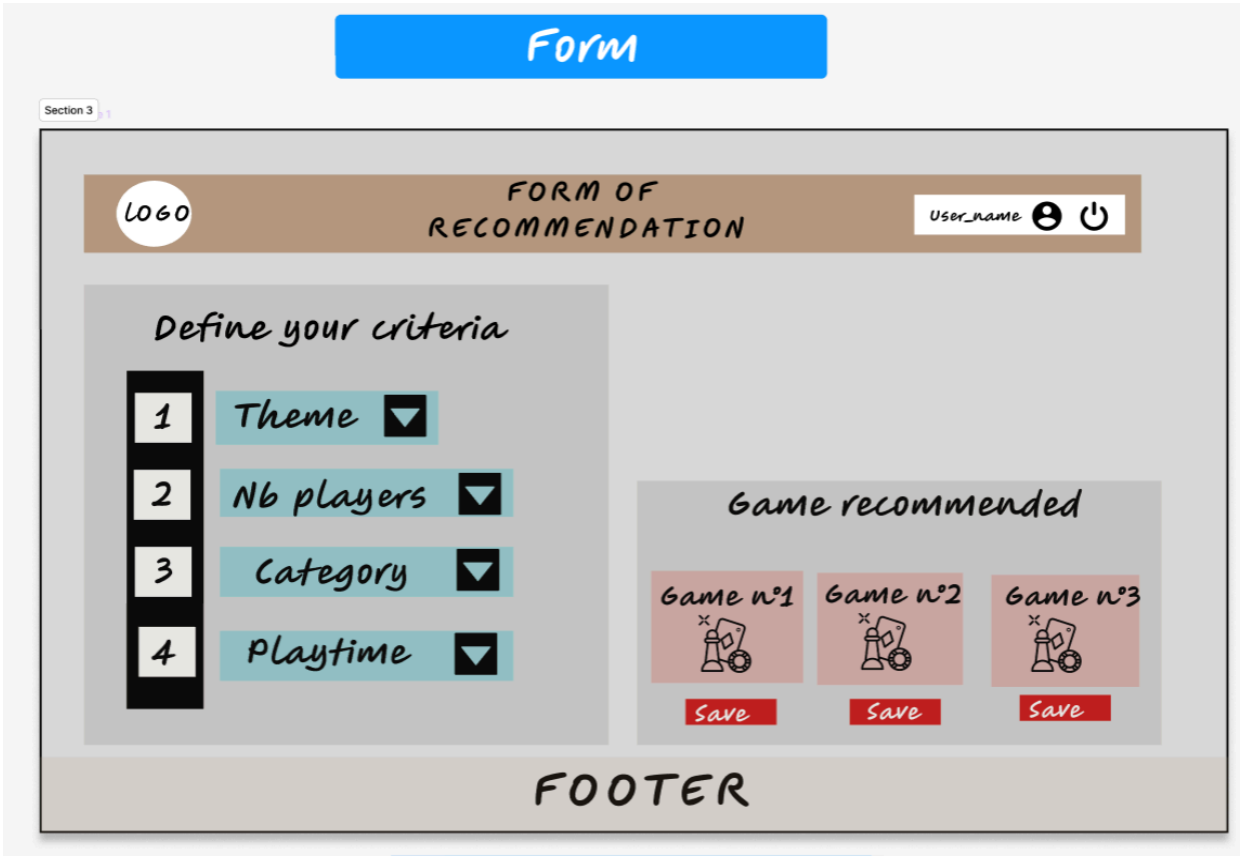
The application that we are designing is here to advise users on board games based on the criterias of their choice. So it is intended for board game enjoyers, families, collectors…

It has several functionalities, such as searching for board games, getting recommended board games by our system via a recommendation system, and having a personal account to store user past search and save games.

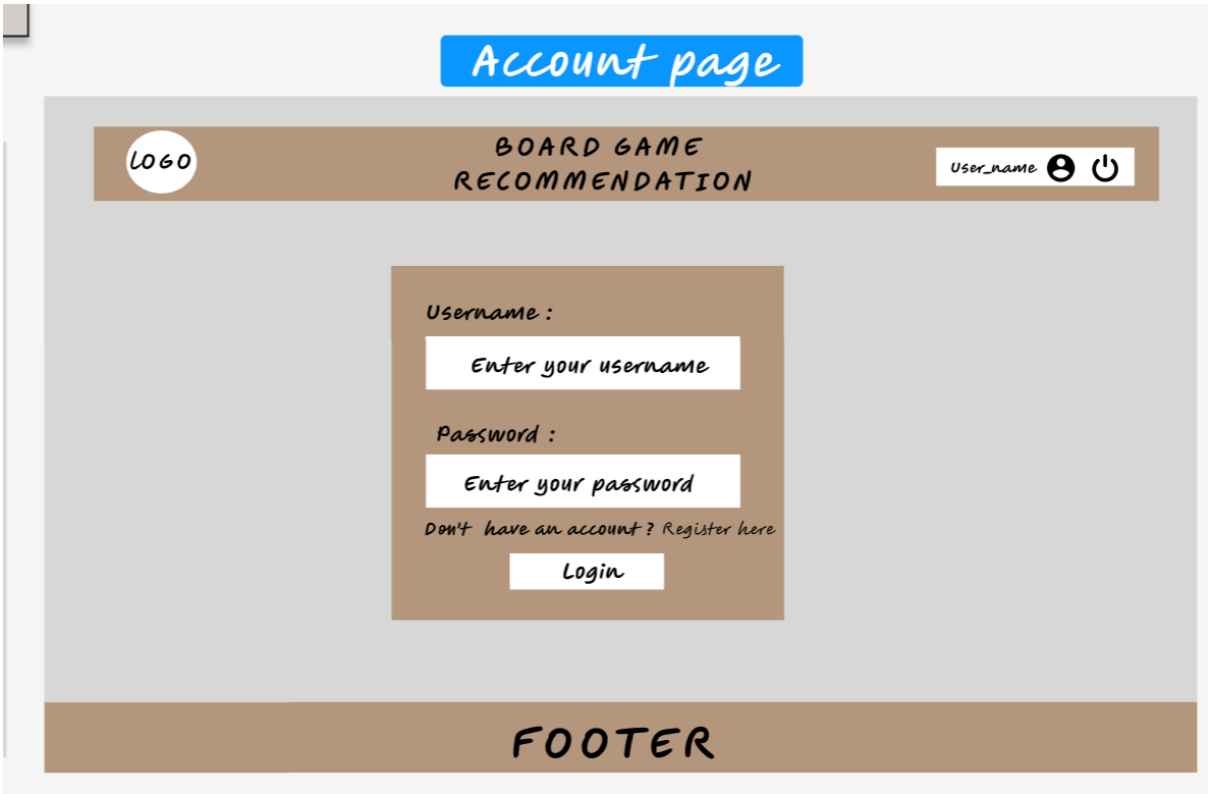
Our target audience is board game enthusiasts and people who want to search some board games with criterias.

Mockups of the UI



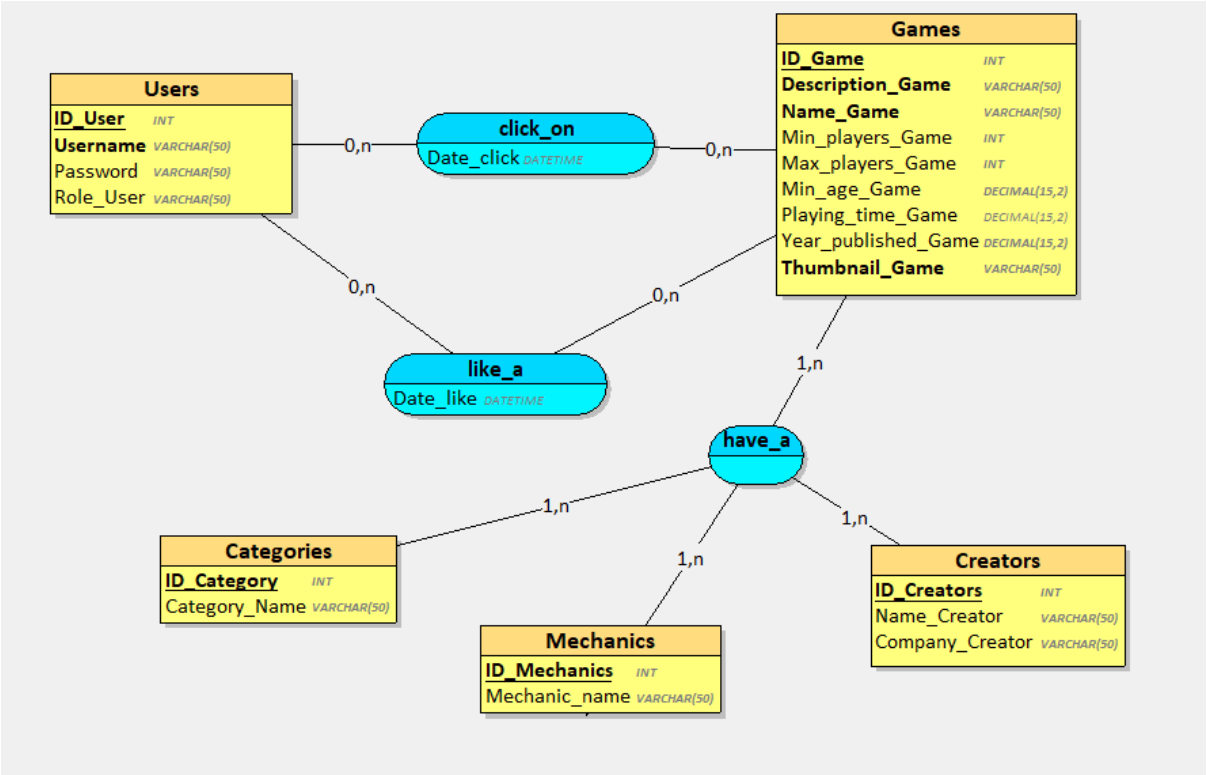



Game Description
GAME
1
Game Name
Description
Year
Rating
Min-mAx Player
Category
Playing time
Publisher
...


Account page
LOGO
BOARD GAME
RECOMMENDATION
User_name
Username :
Enter your username
Password :
Enter your password
Don't have an account ? Register here
Login
FOOTER

2. Database design (CDM, LDM, normalization)

- **Conceptual data model (CDM):**



- **Logical Data Model (LDM):**

Users = (**ID_User** INT, **Username** VARCHAR(50), Password VARCHAR(50), Role_User VARCHAR(50));

Games = (**ID_Game** INT, **Description_Game** VARCHAR(50), **Name_Game** VARCHAR(50), Min_players_Game INT, Max_players_Game INT, Min_age_Game DECIMAL(15,2), Playing_time_Game DECIMAL(15,2), Year_published_Game DECIMAL(15,2), **Thumbnail_Game** VARCHAR(50));

Categories = (**ID_Category** INT, Category_Name VARCHAR(50));

Mechanics = **(ID_Mechanics** INT, Mechanic_name VARCHAR(50));

Creators = (**ID_Creators** INT, Name_Creator VARCHAR(50), Company_creator VARCHAR(50));

click_on = (**#ID_User, #ID_Game**, Date_click DATETIME);

like_a = (**#ID_User, #ID_Game**, Date_like DATETIME);

have_a = (**#ID_Game, #ID_Category, #ID_Mechanics, #ID_Creators**);

- **Normalization**

    - **1st Normal Form**

In the database design, all tables have clear primary keys:

Users uses ID_User.

Games uses ID_Game.

Categories, Mechanics, and Creators each use a distinct ID.

Association tables like click_on, like_a, and have_a also have defined keys.

Each field stores atomic information. For example, Username stores a single username, not multiple, and Mechanic_name stores the name of a single mechanic.

There are no repeating groups or lists inside any attribute.

The database is correctly in First Normal Form.

    - **2nd Normal Form**

Tables like Users, Games, Categories, and Mechanics have simple (single-column) primary keys, so partial dependencies are not possible.

Tables like click_on and like_a have composite primary keys.

For instance, in the like_a table, the primary key is the combination of ID_User and ID_Game. The attribute Date_like depends on both ID_User and ID_Game, because it only makes sense when identifying both the user and the game involved.

Similarly, the click_on table correctly associates a user and a game with a click date, depending on both keys.

There are no partial dependencies. Thus, the database satisfies Second Normal Form.

    - **3rd Normal Form**

The database doesn't have any transitive dependencies. It is fully in Third Normal Form.

3. **Implementation details (tables, triggers, stored procedures, views)**

- **Tables**

- **Users** : ID_User, Username, Password, Role_User => store user credentials when login into the website

- **Games** : ID_Game, Description_Game, Name_Game, Min_players_Game, Max_players_Game, Min_age_Game, Playing_time_Game, Year_published_Game, Thumbnail_Game => retrieve data from Kaggle data set and store it to access it easily from our database
- **Mechanics** : ID_Mechanics, Mechanic_name => retrieve mechanics data from dataset
- **Categories** : ID_Category, Category_Name => retrieve categories data from dataset
- **Creators** : ID_Creators, Name_Creator, Company_Creator => retrieve creators data from dataset
- **click_on** : ID_User, ID_Game, Date_click => store consulted games by the user as history
- **like_a** : ID_User, ID_Game, Date_like => store liked games by the user, either recommendations or general games
- **have_a** : ID_Game, ID_Creators, ID_Category, ID_Mechanics

- **Views**

GameSummaryView:

- Selects basic information about games (ID, name, description, player range, minimum age, playing time, year published, thumbnail) from the `Games` table.
- Renames the columns for better readability.
- Orders the results by year published in descending order.

FamilyFriendlyGamesView:

- Selects games suitable for families (minimum age less than or equal to 10 years) from the `Games` table.
- Includes game ID, name, description, minimum age, player range, playing time, and thumbnail.
- Orders the results by minimum age in ascending order, then by playing time in ascending order.

GamesByYearView:

- Counts the number of games published each year from the `Games` table.
- Returns the year published and the total number of games for that year.
- Orders the results by year published in descending order.

GamesWithCategoriesView:

- Retrieves games and their categories from the `Games`, `have_a`, and `Categories` tables.
- Performs a join to associate games with their categories.
- Returns the game ID, game name, and category name.
- Orders the results by game name in ascending order, then by category in ascending order.

GamesWithMechanicsView:

- Retrieves games and their mechanics from the `Games`, `have_a`, and `Mechanics` tables.
- Performs a join to associate games with their mechanics.
- Returns the game ID, game name, and mechanic name.
- Orders the results by game name in ascending order, then by mechanic in ascending order.

LongPlayingGamesView:

- Selects games with a playing time greater than 120 minutes from the `Games` table.
- Includes the game ID, game name, playing time, year published, and thumbnail.
- Orders the results by playing time in descending order.

GamesPublishedBefore2000View:

- Selects games published before the year 2000 from the `Games` table.
- Includes the game ID, game name, year published, minimum players, maximum players, and thumbnail.
- Orders the results by year published in ascending order.

GamesWithThumbnailView:

- Selects games that have a thumbnail (i.e., where the `Thumbnail_Game` field is neither NULL nor empty) from the `Games` table.
- Includes the game ID, game name, and thumbnail.
- Orders the results by game name in ascending order.

GamesByPlayerCountView:

- Counts the number of games for each player range (e.g., 2-4 players) from the `Games` table.
- Returns the player range and the total number of games for that range.
- Orders **the results by the total number of games in descending order.**

- **Stored procedures**

InsertNewUser

- This procedure takes 3 parameters (`Username`, `Password`, `Role`).
- The procedure checks that the username does not already exist.
- It also checks if the role matches one of our 3 roles: `Admin`, `User`, or `Editor`.
- If these 2 tests pass, then it inserts the user into the `Users` table, with the `ID` being auto-incremented.

- This procedure is useful for inserting new users when they register on the website.

UpdateUserPassword

- This procedure takes 3 parameters (`User_ID`, `Previous_Password`, `New_Password`).
- This procedure checks if the user with the provided `ID` exists.
- Then, it checks the user's old password (to double-check if the user is logged in, which should be the case).
- If the old password is correct, it updates the `Password` in the `Users` table with the `New_Password`.
- This procedure is useful when a user wants to change their password.

InsertGame

- This procedure takes 9 parameters, which correspond to the columns of the `Games` table. These parameters include the game's name, description, published year, etc.
- This procedure inserts a game into the `Games` table.
- It checks the role of the user attempting to insert the game.
- If the user has the role `Admin` or `Editor`, the game is inserted.
- If the user has the role `User`, the insertion is not allowed, and an error message is displayed.
- This procedure can be used to insert a game and ensures that only administrators and editors can perform this action.

- **Triggers**

Trigger 1 : TR_Users_BeforeInsert

- Ensure that the Role_User is one of a predefined set of valid roles (e.g., 'Admin', 'Player', 'Editor').
- Hash the Password before storing it in the database.

Trigger 2 : TR_Users_BeforeUpdate

- Prevent direct modification of the Role_User (if role changes should be controlled by a specific procedure).
- Re-hash the password if it is being updated.

Trigger 3 : TR_click_on_BeforeInsert

- Set the Date_click to the current timestamp if it's not provided.
- Prevent a user from clicking on the same game too frequently (click fraud).

Trigger 4 : TR_like_a_BeforeInsert

- Set the Date_like to the current timestamp if it's not provided.
- Prevent duplicate likes.

Trigger 5 : users_results_changes

- Check the insertion limit before a new line in the like_a table is inserted.
- Make tests easier, let's consider that the liked_limit is 50.

4. Challenges faced and solutions

- Transfer user credentials from login page to backend : use local storage console
- Automatic creation of the database :
    - creation of file database.js with executeQuery functions to implement SQL creation of tables, views, triggers and stored procedures
    - initialize database function to create the database if it doesn't exist
    - initialization automated in server.js when node.js launched

```
// MySQL Database Connection
const pool = mysql.createPool({
  host: process.env.DB_HOST,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME,
  port: process.env.DB_PORT,
}).promise();

console.log("Connecté avec succès à la base de données MySQL.");
if (!process.env.DB_USER) {
    console.warn("ATTENTION: La variable d'environnement DB_USER n'est pas définie. Vérifiez votre fichier .env et sa configurati
}

await initializeDatabase();
```

- Issues with MySQL connection to the database through .env file (database host, username, password, name, port db and port) : create locally a new env file with own password of MySQL Workbench instance
- Implement triggers and transactions in a web application using JS : special connection from the pool for transaction

```
// Get a connection from the pool for transaction
connection = await pool.promise().getConnection();
await connection.beginTransaction();
```

- Errors handling for the api endpoints with try and catch to display personalized error messages and handle different types of error (404, 500, 400)
- Extract details from kaggle files : use of python to extract data from excel sheets
- Ensure confidentiality of user credentials in the database, especially the password : hash it with bcrypt before insertion in the table USERS