中國象棋大師

CHINESE CHESS BRACKET MASTER

INFORMATION AND COMMUNICATION TECHNOLOGY
SCHOOL-BASED ASSIGNMENT

SUN HO CHING
C.C.C. MMWC

2024-2025-
HKDSE-ICT-SBA-
CCCMMWC-
Sun-Ho-Ching

Table of Contents.

Introduction

Background:

Chinese Chess is gaining significant momentum. To harness this energy and advance traditional culture, the Hong Kong government is launching a territory-wide Chinese Chess competition for secondary school students. To achieve this, the project delivers a comprehensive knockout system that effectively records, manages, and analyzes the tournament's progress and results.

As the project manager, I will be developing a system with the following key aims.

Aim:

The aims of the project will be divided into three part.

1. To Automate and Simplify Tournament Setup: Create a tool for organizers to easily set up tournaments, manage player lists (including details like their school and seeded status), and automatically generate fair and logical brackets. The system is designed to intelligently place seeded players and separate contestants from the same school.
2. To Provide a Real-Time Visual Bracket: Display the entire tournament structure in a clear, interactive, and visual format on a webpage. This allows participants, organizers, and spectators to easily track the progress of the competition at any time.

3. To Streamline Match Result Tracking: Enable organizers to update match outcomes instantly and accurately. The system is designed for users to simply click on a match winner, and the bracket will automatically reflect the result and advance the player to the next round.

Requirement:

1. Setting Up the Tournament

First, the system needs to let organizers create a tournament and add all the players. For each player, we'll need to save their name and the school they're from.

2. Creating the Bracket

Once the players are in, the system will automatically generate the pairings in a knockout format. A key feature is handling 'byes' correctly—if there is an odd number of players, the system will figure out who gets a free pass to the second round.

3. Showing the Bracket

The system must display the entire tournament structure in a simple and clear chart. This way, everyone involved can easily see who is playing against whom at any stage of the competition.

4.Updating Match Results

Organizers need an effortless way to record who won each match. The system should allow them to just click on the

winner, and the bracket will update automatically, moving that player to the next round.

Intended users:

Since the tournament system is designed for a Chinese Chess competition, two major groups of users will be involved.

Primary Users:

Tournament Organizers & Staff who run the competition.

Secondary Users:

Spectators (like parents and other students) who want to follow the results.

Considerations for future improvements:

1. A More Flexible Check-In System

- We could add a "check-in" page for the day of the tournament. If a player doesn't show up, the system could re-generate the bracket with only the present players, making the tournament run smoother.

2. Expanded Ranking and Award Features

- We could add more ways to rank players, such as a match for 3rd place (Consolation Match) or a points system to rank schools for a "Group Award."

3. Smarter Seeding for Future Events

- The system could save the final results each year. This data could then be used to automatically suggest seeded players for the next competition, saving organizers time.

4. Live Results for Spectators

- We could create a public-facing view of the bracket that updates automatically in real-time as results are entered. This would allow spectators and players to follow the action live on their phones or other devices.

5. Multi-Language Support

- Add an option in the interface to switch between different languages (like Traditional Chinese and English) to make the system more accessible for everyone involved.

Design and Implementation

System Operation Guide

In order to use the system, you have to follow the steps below:

1.First,open the main page of the system

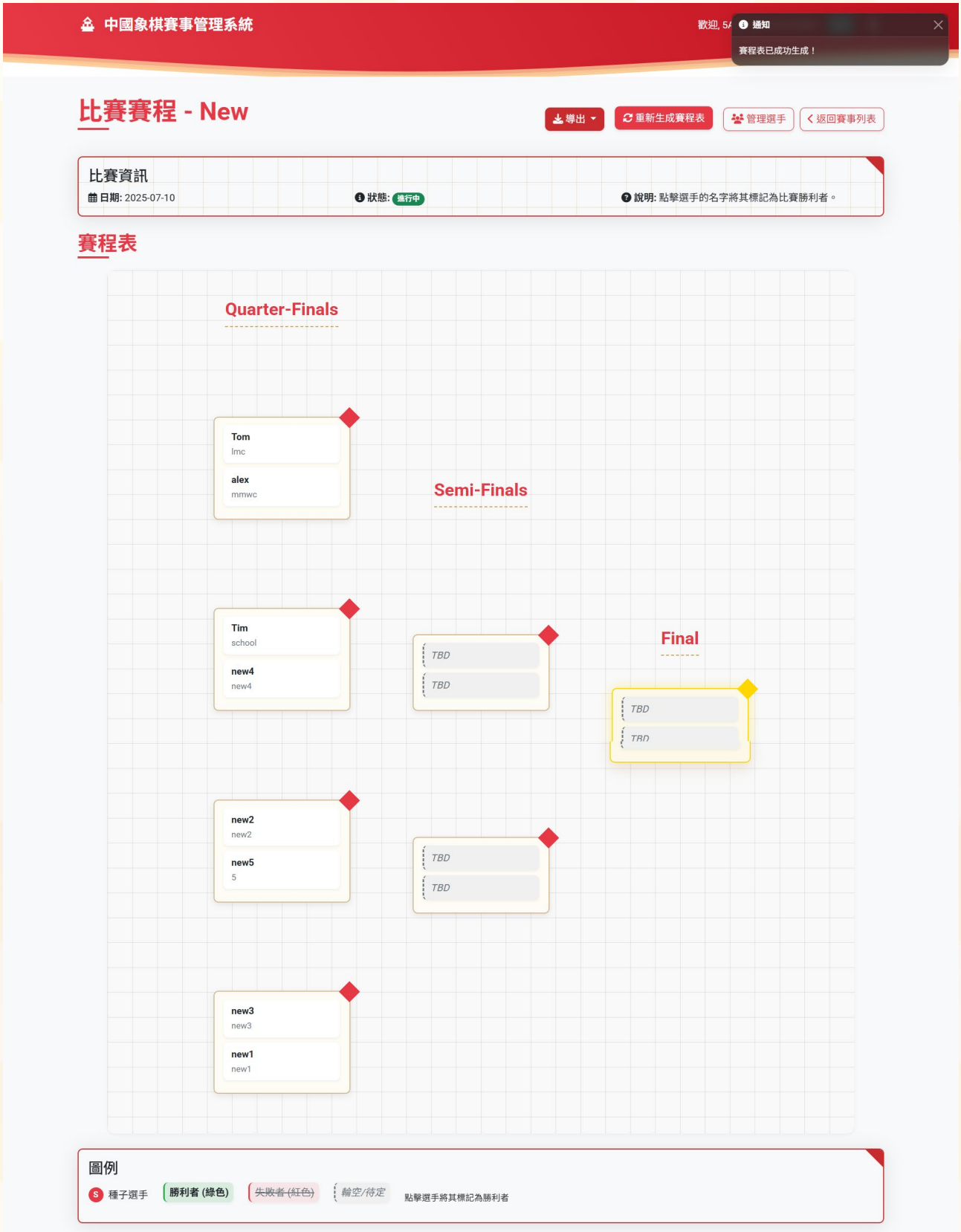
2. Login the system with your admin google account.

3. You may opt to initiate a new competition by entering the competition name, date, and competition format, or alternatively, edit an existing one.

4. You are able to input the player's name and school to incorporate the player information into the database. Additionally, you can select players to be seeded players so that strong players will not encounter other strong players too soon.

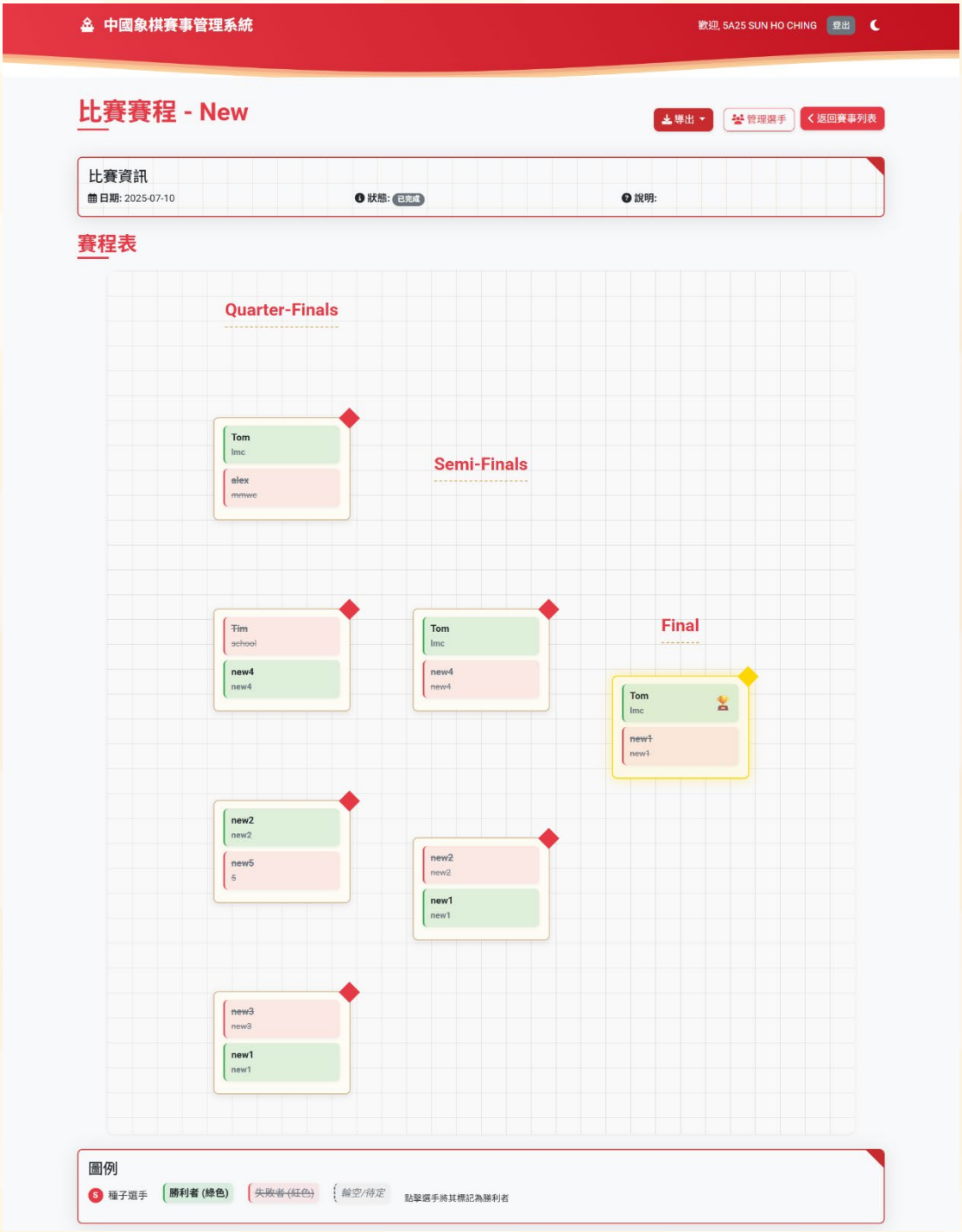
5. After entering all the information of the players,press the ‘生成賽程表’ button to generate the competition chart.

The output result:



You may select the winner of each section when the competition begins and the chart will be updated automatically.

After the competition ends, you can choose to output the chart to pdf to review the result offline.



System Analysis

1. Data Types and Structures

The system is built around a clear data model that separates different concerns.

In the Database (Firestore): I use a NoSQL approach. Data is stored in collections of documents, which are like flexible JSON files. The main collections are `tournaments`, `players`, and `matches`. This is efficient for a web app because the data structure closely matches the objects we use in our code. For example, a "player" document holds their name (string), school (string), and seeded status (boolean).

In the Backend (Python): When the backend code fetches data from Firestore, it's typically loaded into Python dictionaries and lists. For instance, the list of all players for a tournament is handled as a list of dictionaries. This data is then serialized into JSON to be sent to the frontend.

In the Frontend (JavaScript): The JavaScript in the browser receives this JSON data. The entire bracket is managed as an array of match objects. The code then intelligently processes this array to visually construct the rounds and connections of the bracket on the webpage.

2. Variable and Constant Declaration

We are careful about how we declare and initialize variables to keep the code clean and predictable.

Key Constants: Things that don't change, like the database connection client (`db_firestore`) or fundamental configuration settings, are initialized once when the application starts. In the JavaScript, the ID of the tournament being viewed is fetched from the URL and stored as a constant for the page's lifetime.

Local Variables: Most variables are declared locally within the functions that need them. For example, when the bracket generation algorithm runs, the list of players is a local variable inside that function. This prevents different parts of the program from accidentally interfering with each other.

3. Modular Approach

The project is intentionally broken down into specialized, independent modules. This is a core design principle that makes the system much easier to manage and debug.

`routes.py` (The Controller): This file acts as the traffic controller. It handles all incoming web requests (like when a user wants to view a bracket or update a winner) and calls the appropriate functions to handle them. It contains the main application logic.

`tournament.py` (The Brains): This file contains the most complex logic: the algorithm for creating the tournament

bracket. By keeping it in a separate file, we can work on this critical feature without touching any other part of the application.

`bracket.js` (The Frontend Logic): All the code that runs in the user's browser to display and interact with the bracket is in this single JavaScript file. It is completely separate from the backend server logic.

Templates: We use a base `layout.html` template for the common parts of the site (like the header and footer), and other pages like `tournament.html` extend it. This means we don't have to repeat the same HTML code on every page.

4. Scope of Variables and Parameters

We limit the scope of variables wherever possible to avoid unexpected behavior.

Global vs. Local: The only truly global variable is the database connection, which is necessary. Almost everything else is passed between functions as parameters.

Parameter Passing: For example, when generating a bracket, the `tournament_id` and the `list of players` are explicitly passed *into* the generation function. The function then returns the result. It doesn't rely on hidden variables. Similarly, when you click a winner in the browser, the `updateMatchWinner()` JavaScript function is called with specific parameters for the

`matchId` and `winnerId`. This makes the code predictable and easy to test.

5. Reusability and Portability

The system is designed to be reusable and reasonably portable.

Reusable Components: The bracket generation logic in `tournament.py` is highly reusable. Because it's a self-contained module, it could theoretically be used in another program to create brackets. The frontend notification system (`showToastNotification()`) is also a generic function that can be reused anywhere in the app to show messages.

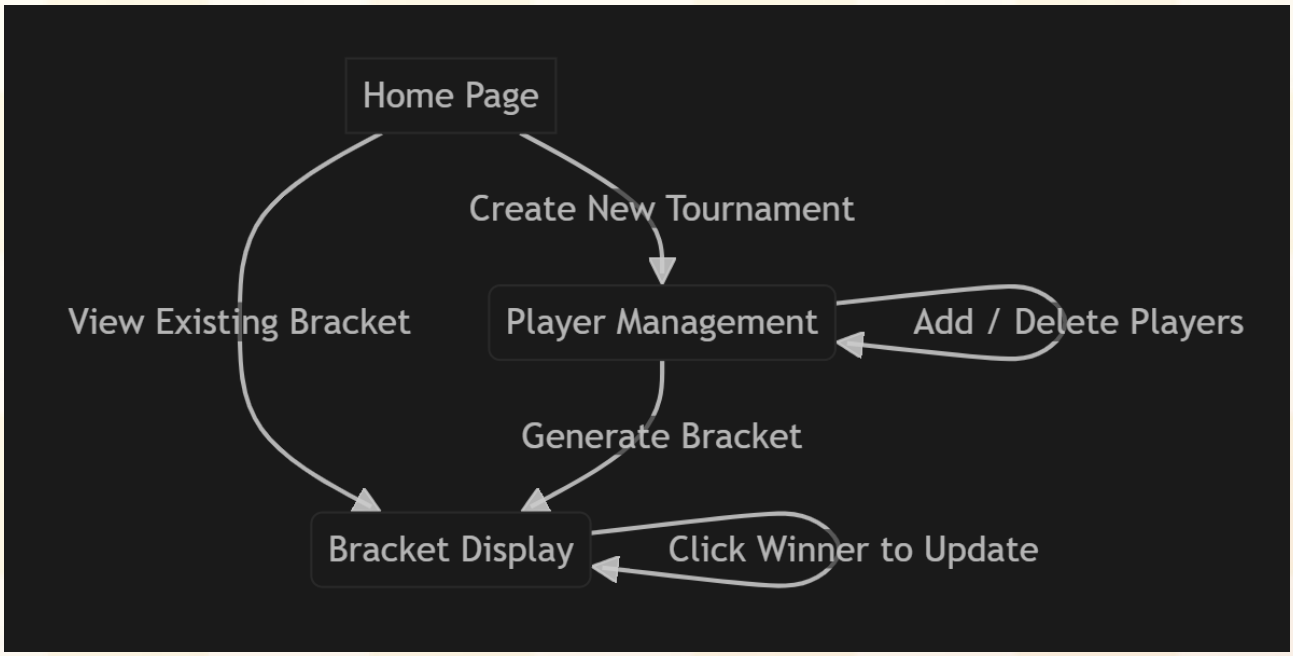
Portability: Because it's a standard Flask (Python) web application, the backend can be deployed on almost any modern cloud server or platform. The frontend uses standard HTML, CSS, and JavaScript, so it works in any modern web browser. The only major dependency is Google Firestore; while the app is portable to different servers, migrating it to a different *database* (like MySQL) would require significant code c

Modules Analysis

1. **Tournament & Player Management Module:** This is the administrative heart of the system. It provides the user interface for organizers to create new tournaments, and for each tournament, to add, edit, and delete players. This module handles all the data entry for player names, their schools, and their seeded status before the competition begins.
2. **Bracket Generation Engine:** This is a purely backend module located in tournament.py. It contains the core algorithm of the project. Its job is to take a finalized list of players from the Player Management module and apply a set of rules to generate a fair single-elimination structure. It intelligently handles placing byes, separating seeded players, and trying to avoid same-school matchups in the first round. It produces the raw data that the competition chart then visualizes.
3. **Match Update API:** This is a small but critical "headless" module. It's a backend API endpoint that listens for requests from the frontend. When an organizer clicks a winner on the competition chart, the browser sends the match ID and winner's ID to this API. The module then performs the necessary database operations in a secure transaction: it updates the winner of the current match and automatically creates the *next* match in the following round with the winner as a participant.

4. **User Notification System:** This is a user-facing module that provides feedback. Using Flask's flash system, it displays small, non-intrusive "toast" notifications in the corner of the screen to inform the user if their actions were successful (e.g., "Winner updated successfully") or if an error occurred. This ensures the user is always aware of the system's status.

User interface analysis



1. Home Page (index.html):

- **Layout:** Very simple. The top of the page has a clear form to **"Create a New Tournament"**. Below this is a list of all existing tournaments.
- **User-Friendliness:** Each tournament in the list has three obvious buttons: **"View Bracket"**, **"Manage Players"**, and **"Delete"**. This makes navigation clear and predictable for the organizer.

2. Player Management Page (players.html):

- **Layout:** The page is titled with the tournament's name. A simple form at the top allows the organizer to add a new player by entering their **Name**, **School**, and checking a box if they are a **Seeded Player**. Below the form is a table listing all players added so far.

- **User-Friendliness:** Each player in the table has an **"Edit"** and **"Delete"** button, making it easy to fix mistakes. A large **"Generate Bracket"** button is prominently displayed, but it remains disabled until at least two players are added, naturally guiding the user through the process.

3. Tournament Bracket Page (tournament.html):

- **Layout:** This page is dominated by the visual bracket itself. It's laid out horizontally, with rounds flowing from left to right.
- **User-Friendliness:** The design focuses on clarity. Matchups are clearly connected. To update a result, the organizer simply **clicks on the winner's name**. The page then reloads to show the updated bracket, providing immediate and reliable feedback that the action was successful. A loading animation prevents confusion while the data is being fetched.

Program analysis

System Overall Architecture

This system utilizes a classic architecture for modern web applications:

- **Backend:** Uses Python's Flask framework, responsible for handling business logic, database operations, and API endpoints.
- **Frontend:** Uses standard HTML, CSS, and JavaScript. bracket.js is the core of the frontend, responsible for fetching data from the backend and dynamically rendering the entire tournament bracket.
- **Database:** Google Firebase Firestore is chosen as the NoSQL database to store all data related to tournaments, players, and matches.

Module One: Tournament Creation and Management (Main Page)

This is the first screen users see upon entering the system. Its main functions are to create new tournaments and to view and manage existing ones.

- **Interface Design:**
    - The layout of the main page (index.html) is clean and simple.

- **Top:** A form for entering the "Tournament Name" and "Tournament Date," along with a "Create Tournament" button.
- **Bottom:** A list displaying all previously created tournaments. Each tournament item shows its name and date, and provides action buttons such as "Manage Players," "View Bracket," and "Delete."

- **Data Collection, Input, and Validation:**
  - **Data Collection:** Users input the Tournament Name and Tournament Date via an HTML form.
  - **Input:**
    - tournament_name: Text type, required.
    - tournament_date: Date type.
  - **Validation:** The backend validates the request upon receipt, ensuring that the "Tournament Name" is not empty. If validation fails, the user is prompted.
- **Data Processing:**
  - When a user clicks "Create Tournament," the browser sends a POST request to the /create_tournament route on the backend.
  - The backend logic (located in routes.py) receives the form data.
  - The system creates a new document in the tournaments collection in the Firestore database. This

document will contain the tournament name, date, and a unique tournament_id.

- Upon successful creation, the user is redirected to the "Player Management" page for that tournament.

- **Program Output:**
  - **On Success:** A new tournament record is added to the database, and a brief success notification ("Tournament created successfully!") is displayed at the top of the screen. The user is then taken to the next step, which is adding players to this tournament.
  - **Error Handling:**
    - If the tournament name is empty, the tournament will not be created, and an error notification ("Tournament name cannot be empty") will be displayed.
    - If a problem occurs while communicating with the Firestore database, an exception will be caught, and a generic error message will be shown to the user.

Module Two: Player Management

After creating a tournament, the next step is to add participating players.

- **Interface Design:**

- The player management page (players.html) is divided into two parts:
    - **Left Side:** An "Add Player" form containing input fields for "Player Name," "Representing School," and a checkbox for "Seeded Player."
    - **Right Side:** A table that instantly displays a list of all players added to the current tournament, including their name, school, and whether they are a seeded player. A "Delete" button is next to each player. At the bottom of the page is the crucial "Generate Bracket" button.

- **Data Collection, Input, and Validation:**
    - **Data Collection:** Performed via the "Add Player" form.
    - **Input:**
        - player_name: Text type, the player's name, required.
        - player_school: Text type, the school the player belongs to, required.
        - is_seeded: Boolean type (True/False), selected via a checkbox.
    - **Validation:** The backend checks if both the player's name and school have been filled out.
- **Data Processing:**

- After the user fills out the form and clicks "Add Player," the data is POSTed to the /tournament/<tournament_id>/add_player route.
- The backend in routes.py handles this request, validates the data, and then creates a new document in the players collection in Firestore.
- This player document will contain the player's name, school, seed status, and most importantly, a tournament_id field to associate the player with a specific tournament.
- The delete player operation triggers a request to /player/<player_id>/delete, and the backend removes the corresponding player document from the database.

- **Sorting and Searching Algorithms:**
  - In this module, the primary operation involves data retrieval. When the page loads, the system queries the players collection in Firestore for all players belonging to the tournament (based on tournament_id) and displays them in the list. There are no complex sorting algorithms here; players are typically sorted by their join time or name.
- **Program Output:**

- After each successful addition or deletion of a player, the page refreshes, and the player list on the right side is updated in real-time.
- A corresponding success or error notification is displayed at the top of the screen.

- Error Handling:
  - If the name or school is not filled in when adding a player, an error prompt will appear.
  - If deleting a player or interacting with the database fails, an error notification will also be displayed.

Module Three: Bracket Generation and Display

This is the most core and complex module of the system. It is responsible for intelligently generating a fair single-elimination tournament bracket based on the player list.

- Interface Design:
  - The core of the bracket page (tournament.html) is a dynamically generated, visual, tree-like structure diagram.
  - The bracket is divided into multiple rounds from left to right (Round 1, Round 2, ...).
  - Each matchup block displays the names and schools of the two players. Fields for undetermined players are shown as "TBD" (To Be Determined).

- Users can click on the winner's name to advance them.
- When the page first loads, a loading animation is displayed until the bracket data is successfully fetched from the backend.

- Data Processing:
  - **Trigger:** The user clicks the "Generate Bracket" button on the player management page.
  - **Backend Processing (/tournament/<tournament_id>/generate_bracket):**
    1. The system first retrieves all player data for the tournament from the database.
    2. It then calls the core algorithm create_tournament_bracket() located in tournament.py.
  - **Algorithm Execution:**
    1. The algorithm calculates the total number of slots required for a standard bracket based on the total number of players (a power of 2, such as 8, 16, 32).
    2. It calculates the number of players who will receive a "Bye."

3. "Seeded Players" are strategically distributed into different sections of the bracket to ensure they do not meet in the first round.
4. Players from the "same school" are separated as much as possible to avoid them facing each other early on.
5. Regular players and "Bye" slots are randomly filled into the remaining positions.
6. The algorithm ultimately generates a list containing the structure of all first-round and subsequent matches.

- Storage and Display:
    1. The backend saves each "match" generated by the algorithm into the matches collection in Firestore.
    2. Once complete, the user is redirected to the tournament.html page.
    3. The bracket.js script on that page sends a request to the backend API at /api/tournament/<tournament_id>/bracket.
    4. The backend API reads all relevant match and player data from the database, combines it into a JSON data structure, and returns it to the frontend.

5. Upon receiving the JSON data, the frontend JavaScript dynamically draws the complete bracket in the browser.

- **Sorting and Searching Algorithms:**
  - The core of this module is the create_tournament_bracket layout and sorting algorithm. It is not a traditional data sorting algorithm (like quicksort) but rather a strategic placement algorithm that arranges players based on specific rules (seeded players, same-school avoidance).
- **Program Output:**
  - The final output is an interactive, visual bracket on the webpage.
  - Simultaneously, the complete match data is generated in the matches collection in the database.
- **Error Handling:**
  - If the number of players is insufficient (e.g., fewer than 2), the bracket cannot be generated, and the system will display an error message.
  - If any error occurs during algorithm execution or database writing, the exception is caught and the user is notified.
  - If the frontend API request fails, the bracket will fail to load, and an error log will be displayed in the browser's developer console.

Module Four: Tournament Update (Winner Advancement)

After the bracket is generated, a referee or administrator needs to update the bracket based on actual match results.

- Interface Design:
    - This functionality is directly integrated into the bracket interface (tournament.html). In every matchup block that displays two players, the players' names are clickable.
- Data Collection, Input, and Validation:
    - **Data Collection:** The user's click event serves as the data input.
    - **Input:** The player_id of the clicked player and the match_id of that match.
    - **Validation:** The frontend JavaScript ensures that clicks are only registered within a valid matchup block. The backend validates the effectiveness of the match_id and player_id.
- Data Processing:
    - **Frontend Trigger:** When a user clicks on a player's name in the bracket, the updateMatchWinner() function in bracket.js is triggered.

- **API Request:** This function sends a POST request to the backend API /api/match/<match_id>/update, including the winner_id in the request body.
- **Backend Transaction:** This is a critical step. The backend in routes.py uses a database transaction to ensure data integrity, performing the following atomic operations:
    1. Find the corresponding match_id document in the matches collection and update its winner_id field to the winner's ID.
    2. Calculate the next match that the winner should advance to.
    3. Update the document for the next match, filling the winner into the appropriate player slot.
- If the transaction is successful, the API returns a success JSON response.

- **Program Output:**
    - According to the project rules, upon a successful API call, bracket.js will execute window.location.reload(), which reloads the entire page.
    - After reloading, bracket.js will re-fetch the latest bracket data, and the user will see the updated bracket with the winner advanced to the next round.
- **Error Handling:**

- If the API request fails (e.g., due to network issues or a backend error), bracket.js will catch this failure.
- It will not reload the page. Instead, it will call a showToastNotification() function to display a non-blocking error notification at the top of the screen (e.g., "Update failed, please try again"), thus avoiding user interruption.
- If the backend transaction fails, it will automatically roll back, ensuring the database is not left in an inconsistent state.

Special Features

1. Role-Based Access Control via Google Authentication

- **What it is:** The system integrates with Firebase Authentication to provide a secure login system. When a user first visits the site, they see a prominent **"Admin Login with Google"** button.

- **Admin Role:** If the person logging in is a pre-approved administrator (their Google email is on a special list in the system's configuration), they gain full access. They can create tournaments, manage players, delete data, and, most importantly, click on winners to update the bracket.

- **Guest/Viewer Role:** Anyone who is not logged in is considered a guest. Guests can view the tournament brackets, but all administrative controls are hidden. The buttons for "Manage Players," "Delete," and the ability to click on a winner are completely absent from their view.

- Reason for this design:
- **Security:** This is the most critical reason. It creates a secure barrier that prevents unauthorized individuals—be it curious students or malicious users— from altering official tournament results, deleting players, or disrupting the entire event. It ensures the integrity of the competition data.
- **Clean User Experience (UX):** This design provides a tailored experience for each user type. Viewers get a simple, clean, read-only interface without the clutter of admin buttons they can't use. Admins get the powerful tools they need. This separation makes the interface much more intuitive and less confusing for everyone.

2. Printable Bracket Export (Export to PDF/Image)

- **What it is:** On the tournament bracket page, there is an "Export" button. This feature allows an administrator to generate a clean, static, and printable version of the current bracket, either as a PDF file or a high-quality PNG image. This exported version is specifically formatted for printing (e.g., on A4 paper) and removes all interactive web elements like buttons or hover effects.

- Reason for this design:

- **Offline & Physical Use:** Tournament organizers almost always need to post physical copies of the bracket at the venue for players and coaches to see. This feature directly serves that need, saving them the hassle of taking screenshots and trying to format them.
- **Official Archiving:** A PDF serves as a permanent, timestamped record of the tournament's final state. This is invaluable for record-keeping, resolving any future disputes, and for sharing the final results officially with schools or governing bodies. It provides a professional and easily distributable artifact of the competition.