ChessBoxingDB

Relazione Tecnica


Vitaliy Didyk


Juri Manili


Riccardo Nicolardi

Chessboxing Analytics Web Application Overview

The Chessboxing Analytics Web Application is a comprehensive platform designed to provide insights, statistics, and analysis for the emerging sport of chessboxing. Combining the strategic gameplay of chess with the physical intensity of boxing, chessboxing requires athletes to excel in both mental acuity and physical prowess. The web application serves as a central hub for enthusiasts, athletes, and analysts to explore the world of chessboxing, track events, and delve into detailed fighter statistics.

Key Features:

1. **Event Tracking**:
- The platform offers a detailed listing of upcoming and past chessboxing events and tournaments.
- Users can explore event details, including dates, venues, participants, and match outcomes.
2. **Fighter Profiles**:
- Comprehensive profiles of chessboxing fighters provide users with insights into athlete backgrounds, statistics, and match histories.
- Users can view fighter statistics, including win-loss records, Elo ratings, height, weight, and more.
3. **Fight Results**:
- Detailed results of chess boxing matches are available for analysis.
- Users can explore match outcomes, including winners, methods of victory (e.g., TKO or chess), and match descriptions.
4. **Comparison Tool**:
- A premium feature allows users to compare two fighters side-by-side, analyzing their statistics and estimating win probabilities.
- The comparison tool provides visual indicators of potential outcomes based on statistical analysis.
5. **Interactive Visualizations**:
- The web application includes interactive visualizations, such as 3D models of chess boards within boxing rings, providing unique and engaging content for users.

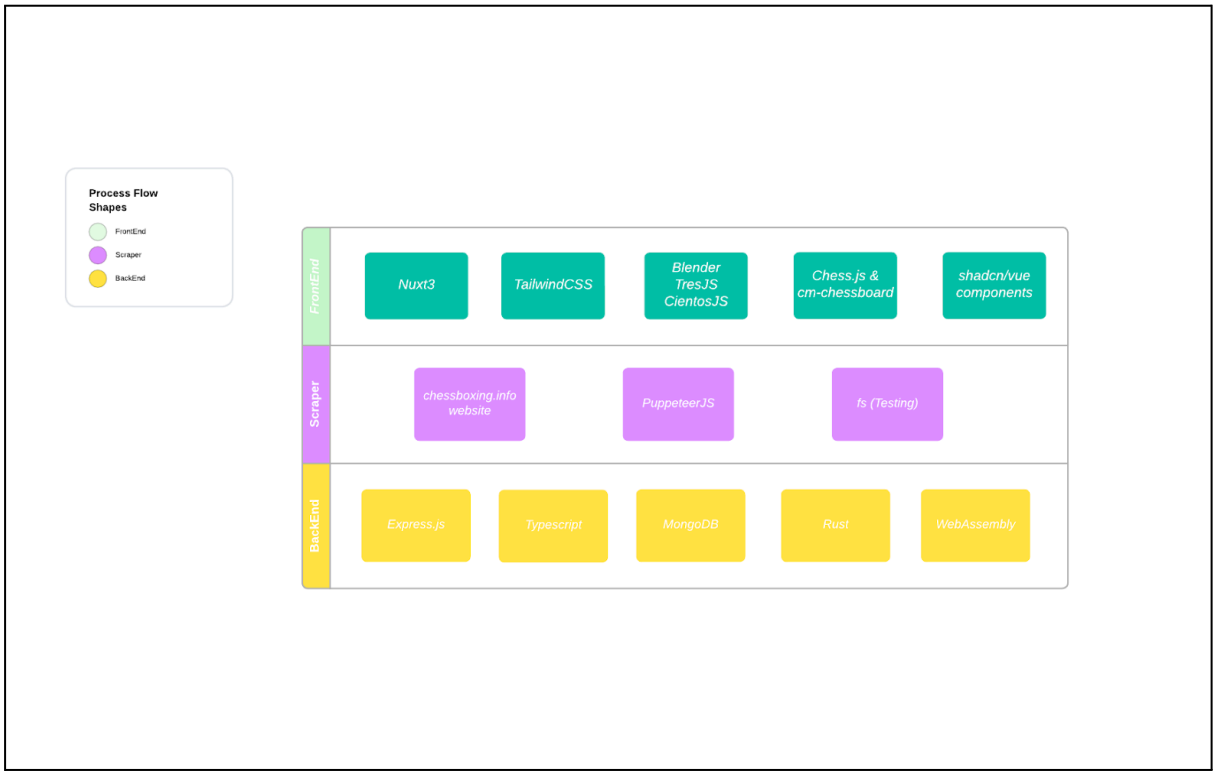
6. **User Authentication and Authorization**:
- User authentication ensures that only logged-in users can access premium features, such as the comparison tool.

- Authorization mechanisms protect sensitive endpoints and data, ensuring user privacy and security.

Technology Stack:

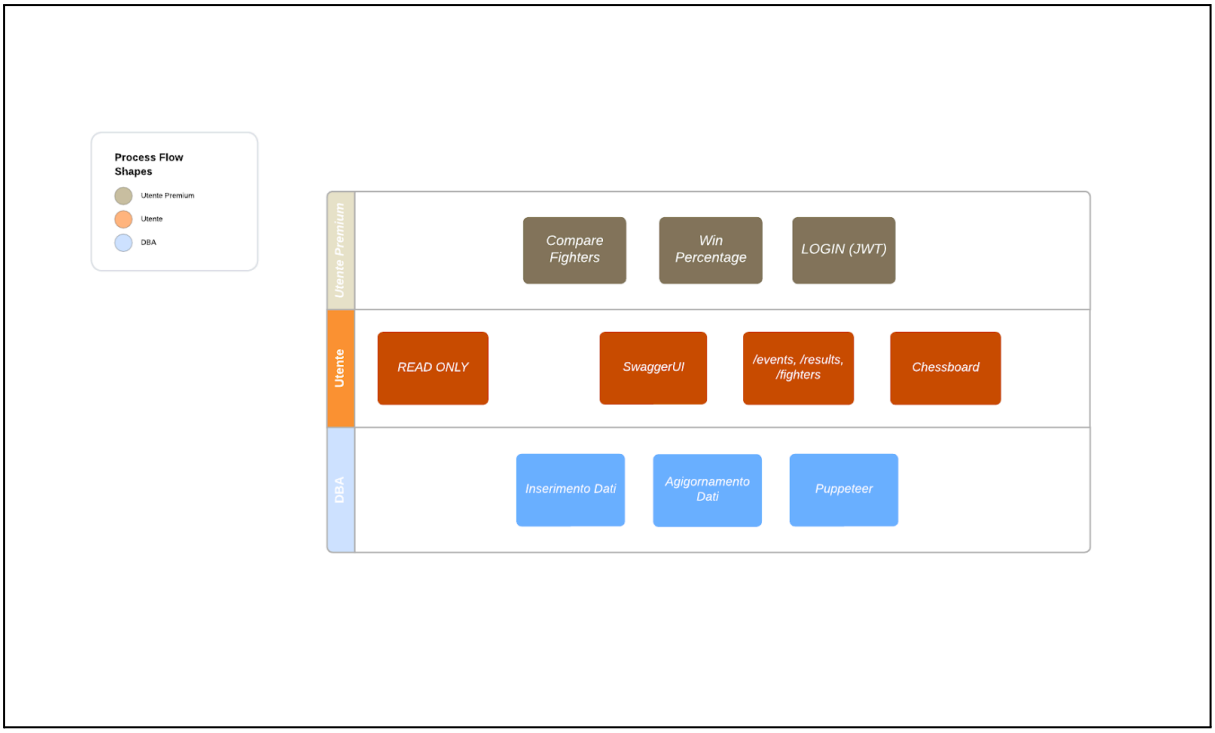
Frontend:

- The frontend of the web application is built using Nuxt.js, a progressive Vue.js framework for building server-side rendered applications.
- Interactive components and visualizations are implemented using libraries such as Three.js for 3D modeling and cm-chessboard.js for interactive chess boards.
- User interface elements are styled using CSS frameworks like Tailwind CSS for rapid development.



Backend:

- The backend is developed using Node.js, providing a scalable and efficient server environment.
- Data storage and retrieval are managed using a database system such as MongoDB, storing information about events, fighters, and match results.
- Authentication and authorization are implemented using JWT (JSON Web Tokens) for secure user authentication and access control.



APIs:

- RESTful APIs are utilized for communication between the frontend and backend components.
- APIs provide endpoints for fetching event details, fighter information, match results, and performing comparisons.

Use Cases:

Enthusiasts:

- Chessboxing enthusiasts can use the web application to stay updated on upcoming events, explore fighter profiles, and analyze match results.

Athletes:

Chessboxing athletes can access detailed statistics about their performance, track their progress over time, and compare their stats with other fighters.

Analysts:

- Analysts and coaches can utilize the comparison tool to assess the strengths and weaknesses of different fighters, strategize match-ups, and predict match outcomes based on statistical analysis.

Organizers:

- Event organizers can use the platform to promote upcoming events, manage participant registrations, and showcase past event highlights.
- The Chessboxing Analytics Web Application serves as a comprehensive resource for anyone interested in the dynamic and multifaceted sport of chessboxing, providing valuable insights, statistics, and analysis to enhance the chessboxing experience for enthusiasts, athletes, analysts, and organizers alike.

Backend in Dettaglio

Introduzione

il backend è una parte fondamentale di un'applicazione web, fornendo l'infrastruttura e la logica necessarie per gestire le richieste dei client, accedere e manipolare i dati nel database e autenticare gli utenti.

Architettura Generale

L'architettura del backend è basata su un'architettura basata su un microframework custom modulare, dove diverse funzionalità sono suddivise in moduli separati, ciascuno responsabile di un aspetto specifico dell'applicazione. I principali moduli del backend includono:

1. **Controller**: Gestisce le richieste HTTP dai client, elaborando le richieste e restituendo le risposte appropriate.
2. **Servizi**: Implementa la logica di business dell'applicazione, interagendo direttamente con il database per recuperare o salvare dati.
3. **Helper**: Fornisce funzionalità di supporto come l'autenticazione degli utenti e la gestione delle operazioni crittografiche.
4. **Database**: Fornisce l'accesso ai dati utilizzando un database MongoDB.

Componenti Principali

1. Controller

I controller gestiscono le richieste HTTP dai client e dirigono il flusso di dati tra il client e il servizio appropriato. Nel progetto, ci sono tre controller principali:

- **DataController**: Gestisce le richieste relative ai dati dell'applicazione, come risultati di combattimento, elenchi di combattenti e eventi. Estrae i parametri dalle richieste e chiama i servizi appropriati per ottenere i dati richiesti.
- **UserController**: Gestisce le operazioni relative agli utenti, come registrazione, accesso, aggiornamento e cancellazione dell'account.
- **ValidationController**: Gestisce le richieste di convalida dell'autenticazione, verificando se l'utente è autorizzato a eseguire determinate azioni.

2. Servizi

I servizi implementano la logica di business dell'applicazione e interagiscono direttamente con il database per recuperare o salvare i dati. Nel progetto, il DataService è il principale servizio responsabile della gestione dei dati. Fornisce metodi per recuperare i dati dai diversi tipi di documenti nel database e per confrontare i combattenti.

3. Helper

Gli helper forniscono funzionalità di supporto come l'autenticazione degli utenti e la gestione delle operazioni crittografiche. Nel progetto, l'AuthHelper gestisce l'autenticazione degli utenti utilizzando token JWT e fornisce metodi per registrare, accedere, aggiornare e eliminare gli utenti.

4. Database

Il database utilizzato è MongoDB, un database NoSQL scalabile e flessibile. Il modulo Db fornisce l'accesso al database e gestisce la connessione al server MongoDB.

Tecnologie Utilizzate

Il progetto backend utilizza diverse tecnologie per implementare le diverse funzionalità:

- **Node.js**: È il runtime JavaScript utilizzato per eseguire il codice JavaScript lato server.
- **Express.js**: È un framework web per Node.js che semplifica la gestione delle richieste HTTP.
- **MongoDB**: È un database NoSQL utilizzato per memorizzare i dati dell'applicazione.
- **JWT (JSON Web Token)**: È uno standard per la creazione di token di accesso che consentono l'autenticazione e l'autorizzazione degli utenti.
- **WASM (WebAssembly)**: È un formato binario per codice eseguibile web, utilizzato per eseguire operazioni crittografiche e altri calcoli complessi.

Decisioni di Progettazione

Le principali decisioni di progettazione includono:

- **Architettura a Microservizi**: La suddivisione delle funzionalità in moduli separati consente una maggiore scalabilità, manutenibilità e riusabilità del codice.
- **Utilizzo di MongoDB**: La scelta di MongoDB come database principale consente una maggiore flessibilità nella gestione dei dati non strutturati e semi-strutturati.
- **Autenticazione JWT**: L'uso di token JWT per l'autenticazione offre un'alternativa leggera e sicura alle sessioni utente tradizionali basate su cookie.
- **Utilizzo di WebAssembly**: L'integrazione di WebAssembly è stata scelta per due motivi principali
    - Alta scalabilità
    - Alta estensibilità

Alta Scalabilità

WebAssembly consente l'esecuzione di operazioni ad alte prestazioni direttamente nel browser o sul server. Questo significa che le operazioni complesse, come calcoli crittografici o algoritmi di elaborazione dati, possono essere eseguite in modo efficiente senza dover dipendere interamente dalle capacità del linguaggio JavaScript. Ciò aumenta la scalabilità dell'applicazione, consentendo di gestire un maggior carico di lavoro senza compromettere le prestazioni.

Alta Estensibilità

Utilizzando WebAssembly, è possibile scrivere moduli ad alte performance in linguaggi come C o Rust, che offrono un maggiore controllo sulle risorse di sistema e una maggiore flessibilità nella gestione della memoria. Questo permette di estendere le funzionalità dell'applicazione in modo efficiente e affidabile, aggiungendo nuove funzionalità o ottimizzando quelle esistenti senza dover riscrivere il codice JavaScript. Inoltre, i moduli WebAssembly possono essere riutilizzati in diverse parti dell'applicazione, migliorando la coerenza e la manutenibilità del codice.

Endpoint api

- **Endpoints API**
  Results
    - **GET /results**
        - Descrizione: Recupera i risultati
        - Parametri:
            - limit (query): Limita il numero di risultati
            - orderBy (query): Campo per ordinare i risultati
            - order (query): Ordine di ordinamento (asc o desc)
            - page (query): Numero di pagina per la paginazione
            - fighterName (query): Nome del combattente
            - resultId (query): ID del risultato
            - eventName (query): Nome dell'evento
            - fightWinDetail (query): Dettaglio della vittoria del combattimento
            - date (query): Data dell'evento
            - fighterWhite (query): Nome del combattente bianco
            - fighterBlack (query): Nome del combattente nero
        - Risposte:
            - 200: Risposta di successo
- **Fighters**
    - **GET /fighterslist**
        - Descrizione: Recupera l'elenco dei combattenti
        - Parametri:
            - limit (query): Limita il numero di risultati
            - orderBy (query): Campo per ordinare i risultati
            - order (query): Ordine di ordinamento (asc o desc)
            - page (query): Numero di pagina per la paginazione
            - fighterName (query): Nome del combattente
        - Risposte:
            - 200: Risposta di successo
- **Events**
    - **GET /events**
        - Descrizione: Recupera l'elenco degli eventi
        - Parametri:
            - limit (query): Limita il numero di risultati
            - orderBy (query): Campo per ordinare i risultati
            - order (query): Ordine di ordinamento (asc o desc)
            - page (query): Numero di pagina per la paginazione
            - eventName (query): Nome dell'evento
        - Risposte:

        - 200: Risposta di successo
- **Detailed Fighter Information**
    - **GET /detailedfighters/{fighterName}**
        - Descrizione: Recupera informazioni dettagliate su un combattente
        - Parametri:
            - fighterName (path): Nome del combattente
        - Risposte:
            - 200: Risposta di successo
- **Fight Details**
    - **GET /fightdetails**
        - Descrizione: Recupera i dettagli dei combattimenti
        - Parametri:
            - limit (query): Limita il numero di risultati
            - orderBy (query): Campo per ordinare i risultati
            - order (query): Ordine di ordinamento (asc o desc)
            - page (query): Numero di pagina per la paginazione
        - Risposte:
            - 200: Risposta di successo
- **Compare Fighters**
    - **POST /compare**
        - Descrizione: Confronta due combattenti
        - Parametri:
            - Authorization (header): Token Bearer per l'autenticazione
        - Corpo della richiesta:
          json

```
{

 "fighterOne": "NomeCombattenteUno",

 "fighterTwo": "NomeCombattenteDue"

}
```

-
- Risposte:
    - 200: Risposta di successo
    - 401: Non autorizzato
    - 500: Errore interno del server
- **POST /user/register**
    - Descrizione: Registra un nuovo utente
    - Corpo della richiesta:
      json

```
{
  "username": "string",
  "password": "string"
}
```

        -
        - Risposte:
            - 200: Registrazione riuscita
            - 400: Richiesta non valida
            - 500: Errore interno del server
- **GET /user/{id}**
    - Descrizione: Ottiene un utente per ID
    - Parametri:
        - id (path): ID dell'utente
    - Risposte:
        - 200: Risposta di successo
        - 404: Utente non trovato
        - 500: Errore interno del server
- **DELETE /user/{id}**
    - Descrizione: Elimina un utente per ID
    - Parametri:
        - id (path): ID dell'utente
    - Risposte:
        - 200: Eliminazione riuscita
        - 404: Utente non trovato
        - 500: Errore interno del server
- **PUT /user/{id}**
    - Descrizione: Aggiorna un utente per ID
    - Parametri:
        - id (path): ID dell'utente
    - Corpo della richiesta:
      json

```
{
  "username": "string",
  "password": "string"
}
```

        -
        - Risposte:

        - 200: Aggiornamento riuscito
        - 404: Utente non trovato
        - 500: Errore interno del server
- **GET /users**
    - Descrizione: Ottiene tutti gli utenti
    - Risposte:
        - 200: Risposta di successo
        - 500: Errore interno del server
- **POST /user/login**
    - Descrizione: Effettua il login
    - Corpo della richiesta:
      json

```
{
  "username": "string",
  "password": "string"
}
```

-
- Risposte:
    - 200: Login riuscito
    - 401: Non autorizzato
    - 500: Errore interno del server

Algoritmo di Confronto dei Combattenti:

Il calcolo della percentuale di vittoria viene calcolato basandosi su tutti i risultati passati dell'atleta interessato, nello specifico viene calcolato nella funzione calculateWinPercentage() con i seguenti step:

1. Parametro della Funzione: La funzione calculateWinPercentage accetta un parametro fighter, che rappresenta i dati del combattente di cui vogliamo calcolare la percentuale di vittorie.
2. Calcolo del Numero Totale di Combattimenti: Utilizzando il metodo reduce(), la funzione somma tutti i risultati dei combattimenti del

combattente. Questi risultati sono memorizzati nell'array fighter.results, dove ogni elemento rappresenta il numero di combattimenti disputati in una certa categoria. Questo ci fornisce il totale dei combattimenti in cui il combattente ha partecipato.

3. Calcolo del Numero di Vittorie: L'elemento all'indice 0 dell'array fighter.results contiene il numero di vittorie del combattente. Lo estraiamo utilizzando parseInt() per assicurarci che sia trattato come un numero intero.
4. Calcolo della Percentuale di Vittorie: La percentuale di vittorie è calcolata dividendo il numero di vittorie totali per il numero totale di combattimenti e moltiplicando il risultato per 100. Questo ci fornisce la percentuale di vittorie del combattente.
5. Restituzione del Risultato: La funzione restituisce la percentuale di vittorie calcolata.

Data Scraping

1. Puppeteer

Puppeteer is a powerful tool for web scraping and automation, developed by Google. It provides a high-level API over the Chrome DevTools Protocol, which enables you to control Chrome or Chromium using JavaScript. Puppeteer can be used for a variety of tasks, including scraping website data, automating form submissions, generating screenshots, and performing UI testing.

One of the main advantages of Puppeteer is its ease of use and flexibility. With Puppeteer, you can programmatically interact with web pages just like a user would in a web browser. This includes navigating to URLs, clicking on elements, filling out forms, and extracting data from the page's HTML structure.

Puppeteer utilizes a headless browser (can be used also with a graphical interface by inserting { headless: false } into the launch of the browser), which means it can run without a graphical user interface, making it suitable for running in server environments or automated workflows.

In web scraping, Puppeteer excels at handling dynamic content and JavaScript-rendered pages. Traditional scraping tools may struggle with sites that heavily rely on client-side rendering, but Puppeteer can fully render and interact with these pages, allowing you to scrape data that would otherwise be inaccessible.

Resource : https://pptr.dev/

2. Chessboxing Database

The "https://www.chessboxing.info/" website serves as a comprehensive platform for enthusiasts, competitors, and organizers interested in the unique sport of chessboxing. Combining elements of chess and boxing, chessboxing competitions require participants to alternate between rounds of chess and rounds of boxing, testing both mental acuity and physical prowess.

This documentation provides an overview of the structure and content of the "https://www.chessboxing.info/" website, along with guidelines for scraping and utilizing the data available on the site.

Website Sections

The website is structured into three main sections, each containing valuable information related to chessboxing matches and events:

1. Events

The "events" section provides details about past and upcoming chessboxing events. Information available in this section may include event names, dates, locations, and event summaries or descriptions.

2. Results

In the "results" section, you can find the outcomes of individual chessboxing matches. This includes match summaries, scores, winners, and possibly detailed breakdowns of each bout.

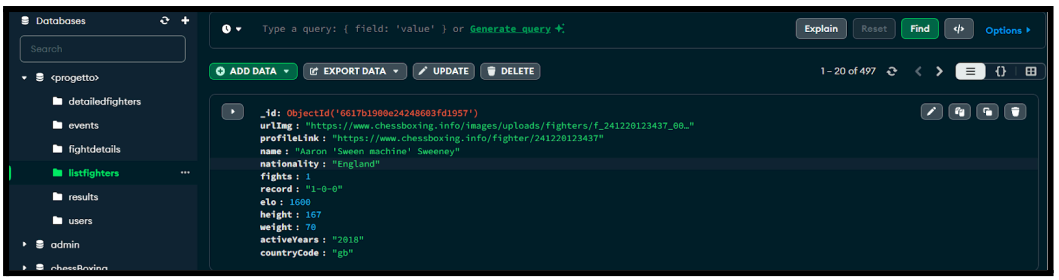
3. Fighters

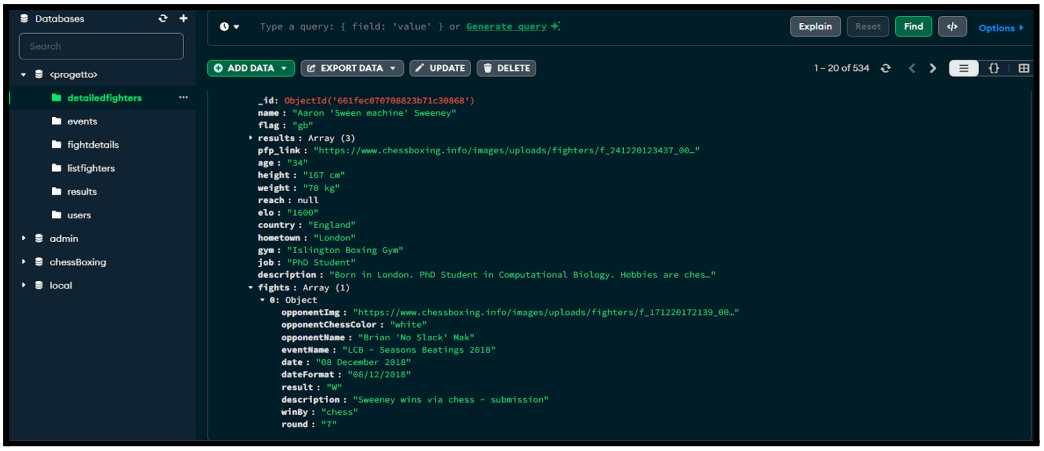
The "fighters" section contains profiles or listings of chessboxers. Information such as names, ages, weights, rankings, and possibly biographical details or career statistics for each fighter may be available.

Data Scraping Process

The provided code utilizes Puppeteer, a Node.js library, to automate web browsing tasks and extract data from the "https://www.chessboxing.info/fighters/" page. The scraping process can be broken down into the following steps:

1. Launch Puppeteer and Navigate to the Page: Puppeteer is used to launch a headless Chromium browser, and a new page is opened, navigating to the target URL, "https://www.chessboxing.info/fighters/".
2. Iterate Through Paginated Results: The code initiates a while loop to iterate through each page of the paginated results on the website. This loop ensures that all available fighter data is scraped, not just the data from the first page.
3. Scrape Fighter Data: Within the loop, Puppeteer's page.evaluate() function is employed to execute JavaScript code within the context of the web page. This function selects relevant elements from the page's HTML structure and extracts the desired information about each fighter.
4. Extract Fighter Details: Information such as the fighter's name, nationality, fight record, Elo rating, and other attributes is extracted from the HTML elements representing individual fighters on the page. These details are stored in an array (listFighters) for further processing.
5. Navigate to Individual Fighter Pages: For each fighter, the script navigates to their individual profile page to gather additional details that are not available on the main fighters page.
6. Extract Detailed Fighter Information: On each individual fighter's page, Puppeteer is again utilized to extract more specific details, such as age, height, weight, fight history, and other biographical information. This nested scraping process enriches the dataset with comprehensive fighter profiles.
7. Store Data: Finally, the scraped data is stored in a MongoDB database. The insertFightDetails() function utilizes the MongoClient library to establish a connection to the MongoDB database and insert the detailed fighter information into the "detailedfighters" collection.





Event Data Extraction:

- Navigate to the chessboxing events page ("https://www.chessboxing.info/events/") and iterate through each paginated page.
- Extract relevant information such as event name, date, venue, and the number of fights for each event.
- Collect event images if available.

2. Nested Scraping for Fight Details:
   - For each event, navigate to its detail page to extract further information about individual fights.
   - Extract details such as fighter names, images, results, and chess moves.
   - Store fight details in a separate array.
3. Data Formatting and Storage:
   - Format extracted data and prepare it for storage.
   - Match country codes with their respective countries using a JSON lookup table.
   - Store formatted event and fight data in separate arrays.
4. Database Insertion:
   - Insert extracted data into a MongoDB database using the insertFightDetails function.
   - Read the MongoDB connection string from environment variables using dotenv.

Result Data Extraction:

- Navigate to the chessboxing results page ("https://www.chessboxing.info/results/") and iterate through each paginated page.
- Extract relevant information such as fighter names, images, match outcomes, event names, and dates for each result entry.

2. Data Formatting:

- Format the extracted date string into a more readable format using the formatDateString function.

3. Storage:
   - Store the extracted result data in the finalResult array.
4. Database Insertion:
   - Insert the finalResult array into a MongoDB database using the insertMany method of the MongoDB collection.

Most important Functions

1. truncateUrl Function:

```
const truncateUrl = (url) => {
    if (url && url.length > 0) {
      const lastIndex = url.lastIndexOf('_');
      return lastIndex !== -1 ? url.substring(0, lastIndex) : url;
    } else {
      return url; // Return the original URL if it's empty
    }
  };
```

- This function takes a URL as input and truncates it by removing the substring from the last underscore (_) character to the end.
- If the URL is empty or null, it returns the original URL.
- The truncateUrl function is designed to remove the trailing portion of a URL string, specifically the part after the last underscore character. This operation is useful for modifying URLs obtained during web scraping to customize the image resolution or size based on frontend requirements.
- Example:

```
const truncatedUrl = truncateUrl("https://example.com/image_123.jpg");
console.log(truncatedUrl); // Output: "https://example.com/image"
```

2. formatDateString Function:

```
function formatDateString(dateString) {
    // Create a new Date object directly from the date string
    let dateObject = new Date(dateString);

    // Format the date components and store in a variable
    let formattedDate = ('0' + dateObject.getDate()).slice(-2) + '/' +
      ('0' + (dateObject.getMonth() + 1)).slice(-2) + '/' +
```

```
        dateObject.getFullYear();

    return formattedDate;
}
```

- This function takes a date string as input and formats it into the "dd/mm/yyyy" format.
- It creates a new Date object from the input date string and extracts the day, month, and year components.
- This conversion is crucial for compatibility with database queries or display requirements that necessitate a standardized date format.
- Example:

```
const formattedDate = formatDateString("2024-04-20");
console.log(formattedDate); // Output: "20/04/2024"
```

3. insertFightDetails Function:

- This asynchronous function inserts fight details data into a MongoDB collection.
- It establishes a connection to the MongoDB database using the provided connection string.
- Inserts the fight details into the "fightdetails" collection.
- Example:

```
const fightDetails = [
  { fighter: "John Doe", result: "Win" },
  { fighter: "Jane Smith", result: "Loss" }
];

insertFightDetails(fightDetails);
```

4. if (urlImgWhite && urlImgWhite.includes('unknown.jpg')):

- This conditional statement checks if the urlImgWhite variable contains the substring "unknown.jpg".
- If it does, it sets urlImgWhite to null.
- Example

```
let urlImgWhite = "https://example.com/unknown.jpg";
if (urlImgWhite && urlImgWhite.includes('unknown.jpg')) {
   urlImgWhite = null;
}
console.log(urlImgWhite); // Output: null
```

5. if (imgFlag):

- This conditional statement checks if the imgFlag variable is truthy (not null, undefined, 0, or false).
- If it is, it further checks if it contains the substring "Unknown.png".
- If it does, it sets countryCode to null; otherwise, it extracts the country code from the URL.
- The countryCode variable is then used to provide the frontend with a shortened representation of the country name, facilitating easier API integration for fetching the corresponding flag image.
- Example

```
let imgFlag = "https://example.com/flags/USA.png";
let countryCode = null;
if (imgFlag) {
   if (imgFlag.includes('Unknown.png')) {
      countryCode = null;
   } else {
      countryCode = imgFlag.split('/').pop().replace('.png', '');
   }
}
console.log(countryCode); // Output: "USA"
```

6. checkNull Function:

```
function checkNull(value) {
        if (value === "?" || value === "" || value === undefined) {
           return null;
        }
        return value;
     }
```

- This function checks if a value is equal to "?" or an empty string ("") or undefined.
- If it matches any of these conditions, it returns null; otherwise, it returns the original value.
- Example:

```
const value = "?";
const result = checkNull(value);
console.log(result); // Output: null
```

Test File Overview:

This test (trialEvent.js) file serves to verify and test the scraping functionality on a single event page before scaling it to iterate over multiple events. It utilizes Puppeteer for browser automation and MongoDB for data storage.

Scraping Event Data:

1. Navigation:
    - The page navigates to a specific event page (eventLink).
2. Event Data Extraction:
    - Utilizing Puppeteer's evaluate method, event data is extracted.
    - Extracted data includes fighter names, images, results, and round information.
3. Nested Page Evaluation:
    - For each event, a nested page is opened to extract additional fight details.
    - Details like fight win details, nationalities, and chess moves are retrieved.
4. Data Formatting:
    - Extracted data is organized into arrays and objects.
5. Data Cleaning:
    - Unnecessary elements like nested links are removed.
6. Data Storage:
    - The extracted event data is written to a JSON file for verification and testing (trialEvent.json).
7. Browser Closure:
    - Once scraping and data storage operations are complete, the browser instance is closed.

Data Folder:

The data folder houses JSON files containing scraped data, serving

primarily for verification before database insertion.

- Purpose:
    - Verification: Ensures accuracy and completeness of scraped data before database entry.
    - Debugging: Offers reference for resolving errors or inconsistencies in scraped data.
    - Documentation: Provides structured documentation of scraped data for future reference and analysis.
- Workflow:
    - Scraping: Data is extracted from the source website and temporarily stored.
    - Verification: Extracted data is compared against the source for validation.
    - Storage: Validated data is stored in JSON files within the data folder.
    - Analysis: Files may be analyzed or compared with source data for quality assurance.
- Benefits:
    - Data Integrity: Ensures insertion of accurate data into the database.
    - Versioning: Facilitates tracking of changes in scraped data over time.
    - Backup: Acts as a backup in case of database data loss or corruption.
- Example:
    - The `trialEvent.json` file contains data from a single event, serving as a sample representation of scraped data.

Country Code Mapping JSON:

The provided JSON (countryCode.json) maps country names to their corresponding ISO 3166-1 alpha-2 country codes.

- Purpose:
    - Standardization: Enables uniform representation of country information across systems.
    - Localization: Facilitates localization by providing standardized

country codes for various regions.
    - Data Enrichment: Enhances data with additional contextual information about countries.
- Usage:
    - Data Transformation: Used to convert country names to country codes for data processing or analysis.
    - Integration: Incorporated into scripts or applications to enrich data with country code information.
    - Normalization: Ensures consistency in country representation across datasets or databases.
    - 

Pages

/default.vue

The "/default.vue" component serves as the template for all pages of the Chessboxing Analytics Platform, except for the login page. It includes the following elements:

1. **Navbar**: The navbar contains the ChessboardDB logo, providing branding for the platform. It also includes a login button for users to access their accounts.
2. **Navigation Menu**: The navigation menu lists the available pages of the platform, allowing users to easily navigate between different sections such as events, results, fighters, and comparison.

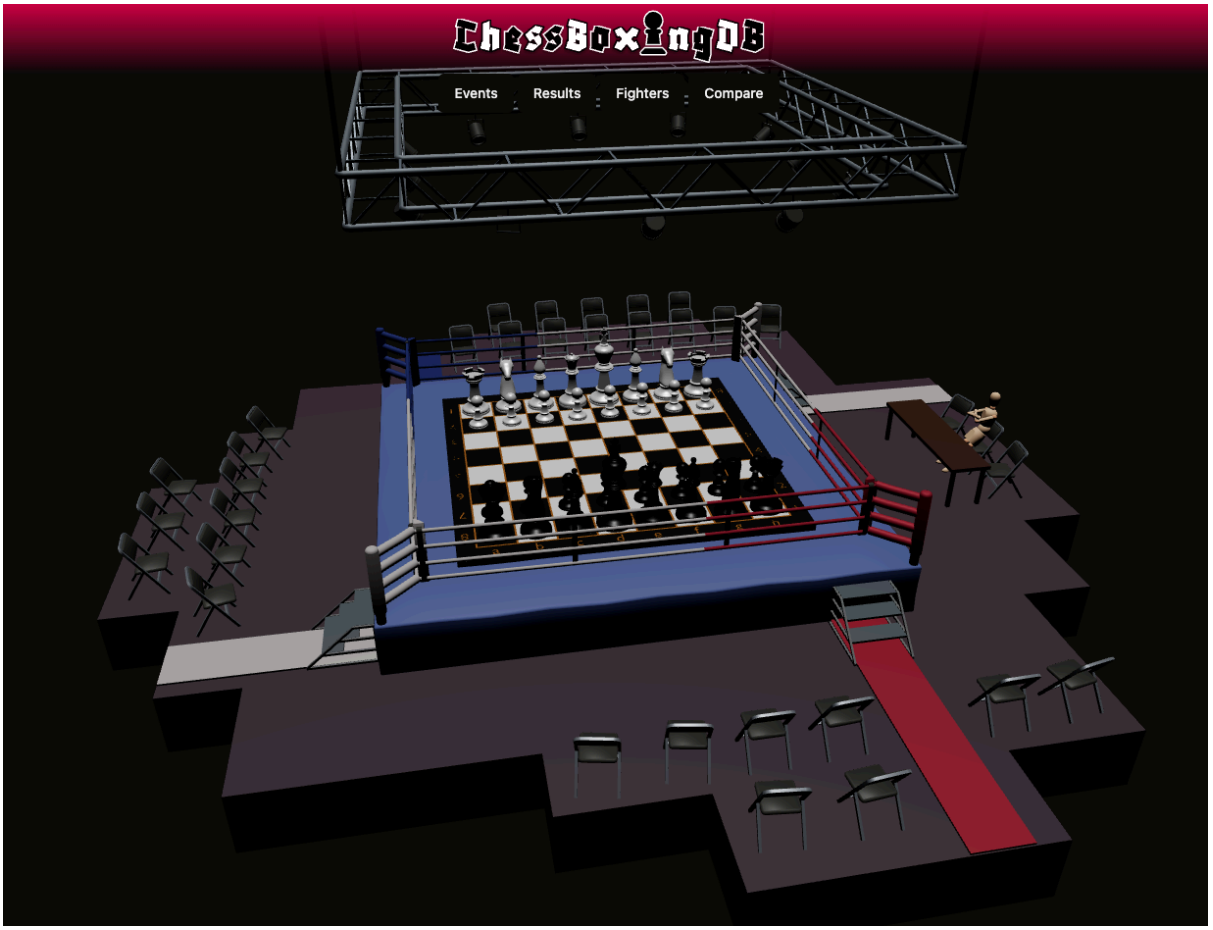
This component ensures consistency across all pages by providing a common layout and navigation structure.

/index.vue

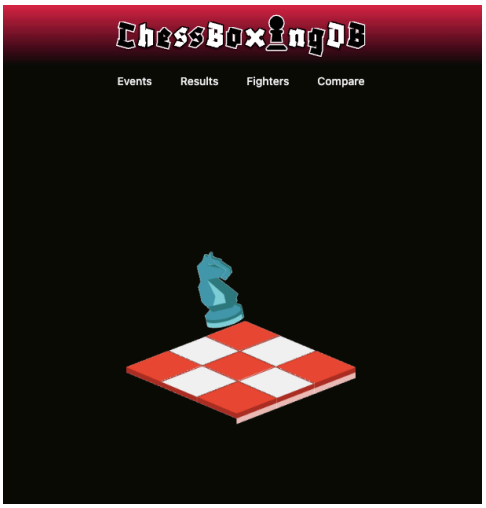
The "**/index.vue**" component represents the homepage of the Chessboxing Analytics Platform. It features a visually engaging 3D model created using Blender, showcasing a chessboard placed atop a boxing ring. Key features of the homepage include:

1. **3D Model**: The centerpiece of the homepage is a 3D model rendered in real-time, depicting a chessboard with all pieces positioned in their starting positions. The model is created using Blender and exported in the "**chessboxing.glb**" format.
2. **Interactive Features**: Users can interact with the 3D model by rotating and scaling it. This interaction enhances user engagement and provides an immersive experience.
3. **Animation**: The 3D model spins continuously on the homepage, capturing the attention of visitors and adding dynamism to the interface.

4. **Technology Integration**: The integration of the 3D model into the Nuxt.js framework is facilitated by the [Tres.js] and Cientos.js modules. These modules enable seamless import and display of 3D models within Nuxt.js applications.



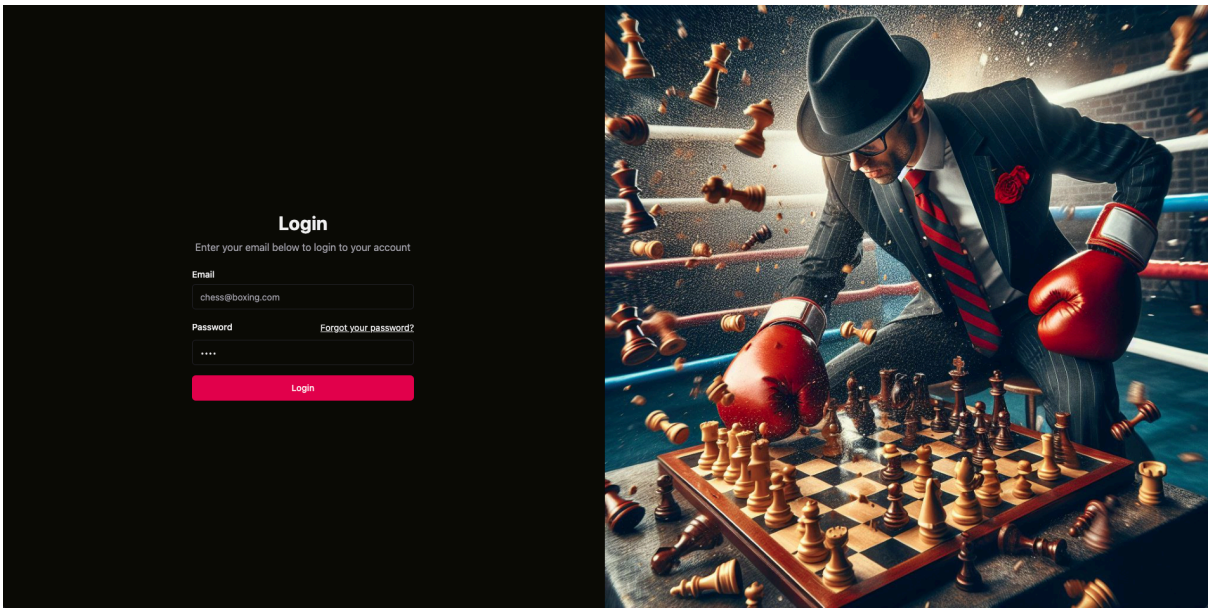
The homepage serves as an introduction to the platform, combining elements of chess and boxing to reflect the unique nature of chessboxing while showcasing the platform's innovative use of technology.



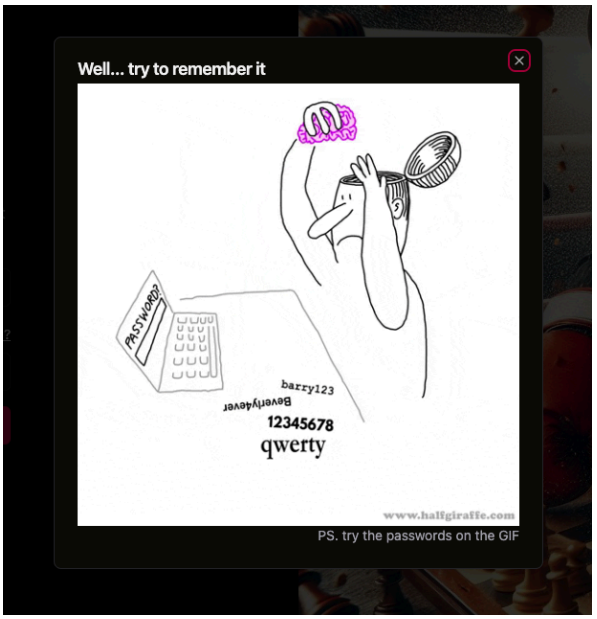
In addition to the 3D model and interactive features described earlier, the homepage ("/index.vue") incorporates a chess-themed loading GIF. This loading GIF serves as a visual indicator to users while the 3D model is being loaded onto the page.

/login.vue

The "/login.vue" page allows users to log in to the Chessboxing Analytics Platform by entering their email and password. Key features of the login page include:



1. **Login Form**: The login form prompts users to input their email address and password to access their accounts.
2. **Forgot Password**: If users forget their password, they can click on the "Forgot Password" button. This action opens a modal window featuring an entertaining GIF animation.
    - **GIF Animation**: The GIF animation depicts a humorous scenario where a character attempts to recall their password by visualizing it and physically interacting with it. The character comically retrieves various password fragments from their mind and tosses them onto a laptop screen.
    - **Interactive Element**: Users are encouraged to engage with the amusing GIF animation while waiting for password retrieval instructions.



3. **Authentication Process**:

- Upon completing the login form, users submit their credentials.

A POST request is sent to the platform's API endpoint at **http://localhost:8000/user/login**, with the following request body:

```
{
  "username": "chess@boxing.com",
  "password": "tryToGuessIt"
}
```

If the API request is successful, the platform responds with a JWT token, indicating successful authentication.
If the API request fails, a toaster UI component appears, notifying the user that the credentials provided may be incorrect.

The "/login.vue" page provides a user-friendly interface for accessing the platform, incorporating entertaining elements like the GIF animation to enhance user engagement. Additionally, it seamlessly handles the authentication process, ensuring secure access to user accounts.

/events.vue

The "/events.vue" page presents users with a list of the latest chess boxing events and tournaments, displayed in a convenient list layout. Key features of this page include:

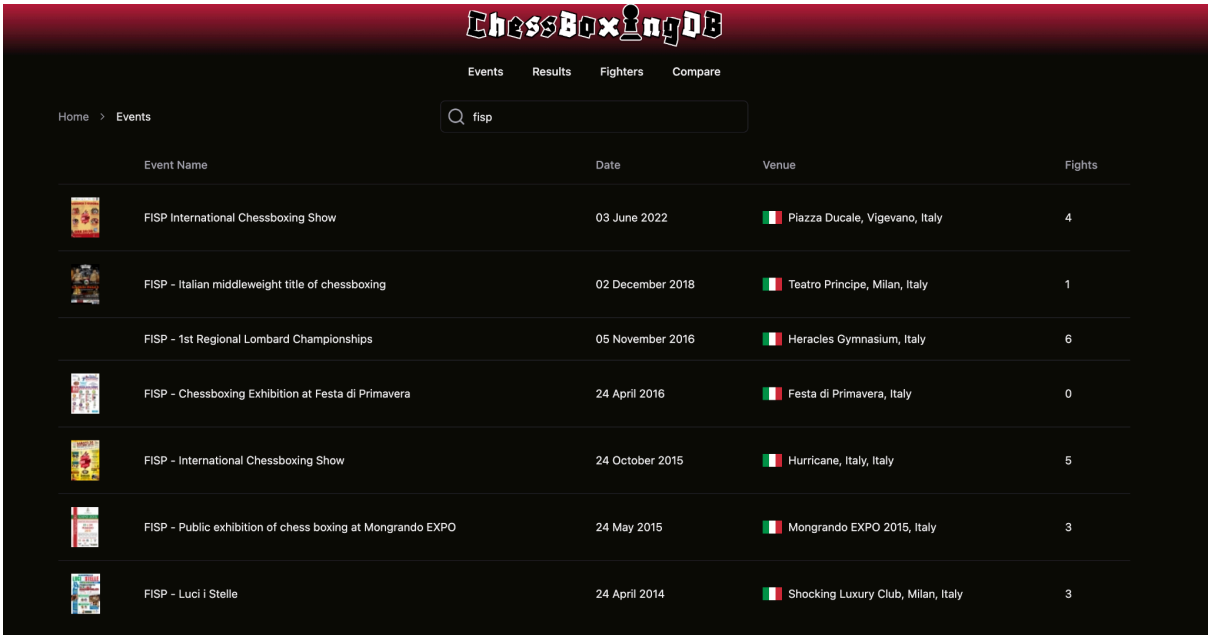
1. **Event List Layout**:
- Each event in the list is accompanied by essential details such as:
    - **Manifesto**: A brief description or manifesto of the event, providing context or highlights.
    - **Name**: The name or title of the event.
    - **Date**: The date on which the event took place.
    - **Venue/Gym**: The location or venue where the event was held.
    - **Number of Fights**: The total number of fights conducted during the event.

This layout enables users to quickly scan and identify events of interest.

2. **Search Functionality**:
- Users can search for specific events using the provided search bar located at the top of the page.
- The search functionality triggers a GET request to the platform's API endpoint:

```
GET http://localhost:8000/events?search=eventName:FISP
```

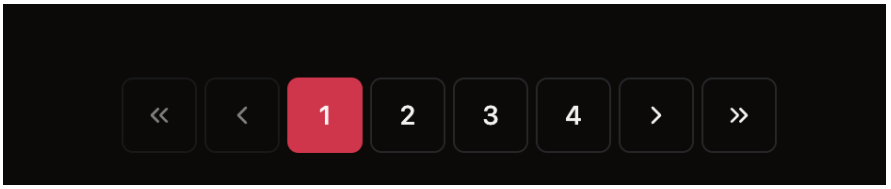
This endpoint filters events based on the specified search criteria, such as event name.



3. **Pagination**:
- The page implements pagination to manage large datasets efficiently.
- Initially, when the page is first rendered, a GET request is made to retrieve the necessary data:

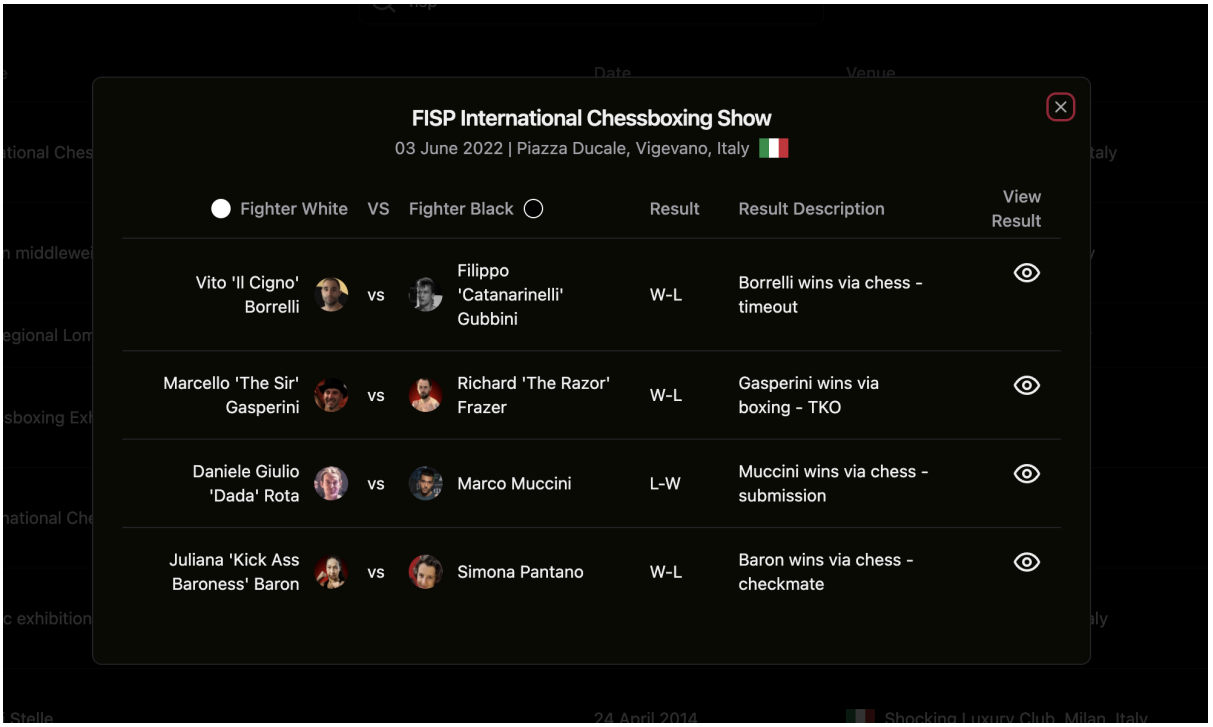
```
GET http://localhost:8000/events?page=0
```

- The "page" parameter in the URL specifies the page number to load, allowing for segmented data retrieval.
- Pagination components are provided at the end of the list layout, enabling users to navigate through multiple pages of events.



4. **Event Details Modal**:

- Clicking on an event name opens a modal component displaying detailed information about the fights conducted during the event.
- The modal presents the following information for each fight:
    - **Fighter White**: The name of the white player or boxer.
    - **Fighter Black**: The name of the black player or boxer.
    - **Results**: Indicates whether the fight resulted in a win or loss.
    - **Result Description**: Provides additional details on the outcome, such as which fighter won and how (e.g., by TKO or chess).
    - **View Detailed Result Button**: Allows users to view additional details of a specific fight



5. **Fighter Statistics**:

- Users can click on a fighter's name within the modal to view detailed statistics for that fighter.
- This feature provides insights into individual fighter performance and history.

The "/events.vue" page offers a comprehensive overview of past chessboxing events, providing users with the ability to explore event details, view fight outcomes, and access fighter statistics.

/results.vue

The "/results.vue" page displays the latest results of chess boxing matches in both **list** and optional **grid layouts**. Key features of this page include:

1. **Result Display**:
- Each result is presented with the following details:
    - **Fighter White**: The name of the white player or boxer.
    - **Fighter Black**: The name of the black player or boxer.
    - **Result**: Indicates the outcome of the fight.
    - **Date**: The date on which the fight took place.
    - **View Detailed Result Button**: Allows users to access a detailed page of the result for more information.
- Users can choose between a list or grid layout for viewing results, providing flexibility in presentation.

2. **Detailed Result Page**:

- Clicking on the "**View Detailed Result**" button redirects users to a detailed page of the result. The URL format for the detailed page is:

```
http://localhost:3000/results/<date>?eventName=<name>&fighterWhite=<name
>&fighterBlack=<name>
```

- The detailed page provides additional information about the specific result, including event name, fighter names, and date.

3. **Result Filtering**:

- Users can filter results by searching for a specific event name.
- The search functionality triggers a GET request to the platform's API endpoint:

```
GET http://localhost:8000/results?search=event:fisp
```

This endpoint filters results based on the specified event name, enhancing user convenience in finding relevant results.

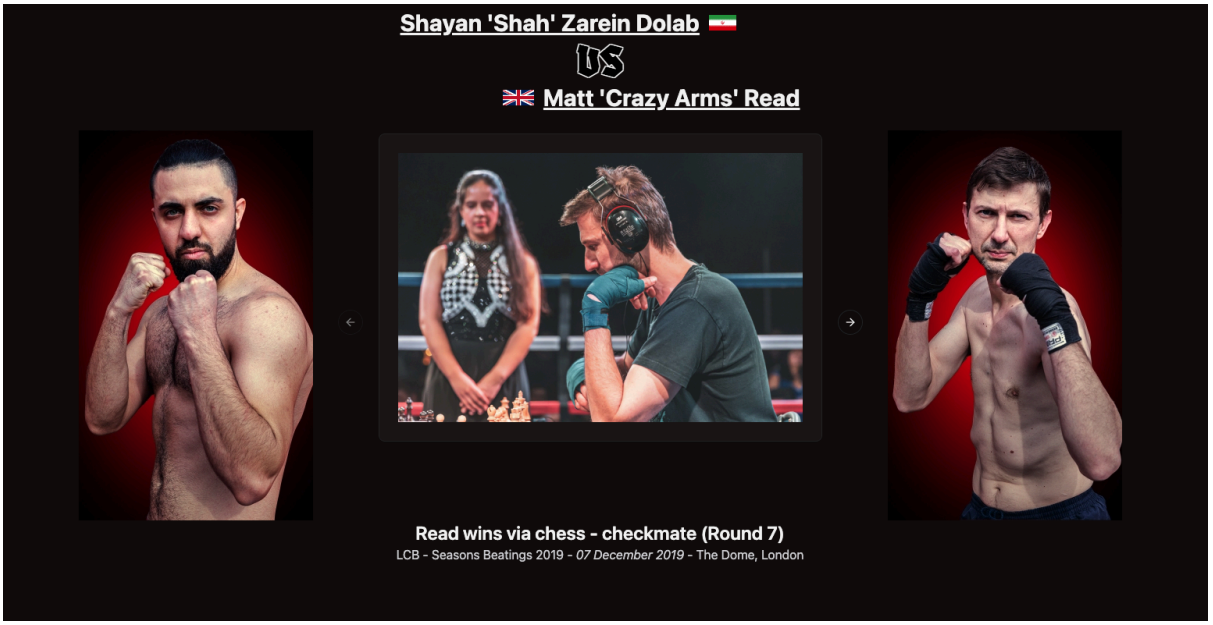
The "/results.vue" page offers users an overview of recent chessboxing match results, with options for customized result display and easy access to detailed result pages.

/results.vue/[id]

The "/results.vue/[id]" page displays detailed information about the results of a chess boxing match between two players. This page requires the following parameters in the URL to function properly:

1. **Date of the Event**: The date of the event in the format "DD/MM/YYYY".
2. **Event Name**: The name of the event.
3. **Name of Fighter White**: The name of the white player or boxer.
4. **Name of Fighter Black**: The name of the black player or boxer.

The page utilizes the useRouter() Nuxt composable to extract these parameters from the URL and sends a request to the platform's API endpoint using the useFetch() composable to fetch detailed fight information.



Components and Features:

1. **Fight Details Display**:
- The page displays detailed information about the match, including the event name, date, and names of the fighters involved.
- It showcases the results of the match, providing insights into the outcome.
2. **Photo Gallery Carousel**:
- A carousel presents a gallery of photos taken from the event, allowing users to view images related to the match.
3. **Fighter Profile Pictures (PFP):**
- Profile pictures (PFP) of the fighters are displayed, providing visual identification.

4. **Interactive Chess Board**:

- An interactive chessboard is implemented using cm-chessboard.js and chess.js libraries.
- Users can navigate through the chess game by selecting moves, which are displayed in algebraic notation, that will be later converted into FEN notation, for making the chessboard work.
- The chess pieces move accordingly on the board, providing a dynamic visualization of the match.

Data Retrieval:

- The page utilizes the useRouter() Nuxt composable to extract parameters from the URL.
- A request is made to the platform's API endpoint using the useFetch() composable to fetch detailed fight information based on the extracted parameters.

The API endpoint URL for fetching fight details is:

```
GET
http://localhost:8000/fightdetails?date=<date>&eventName=<eventName>&fighterWhite=<fighterWhite>&fighterBlack=<fighterBlack>
```

The "/results.vue/[id]" page offers users a comprehensive view of the results of a chess boxing match, complete with detailed information, photo gallery, and an interactive chessboard for dynamic engagement.

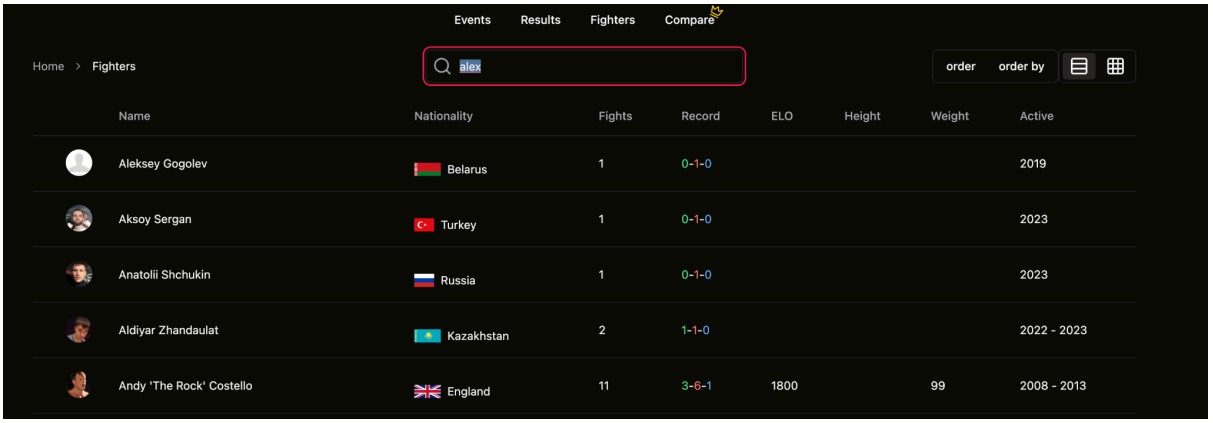
/fighters.vue

The "/fighters.vue" page presents users with a grid layout featuring card components showcasing information about chess boxing fighters. Key features of this page include:

1. **Fighter Cards Grid Layout**:
- The page displays fighter cards arranged in a 5-column grid layout.
- Each card features the fighter's profile picture (PFP) and name on the front.
2. **Hover Interaction**:
- Hovering over a fighter card causes it to **flip**, revealing additional information about the fighter on the back of the card.
- The back of the card displays player information in a table format, including details such as nationality, fights, record, Elo rating, height, weight, and activity.
- The fighter's flag is displayed in the background of the table for aesthetic purposes.
3. **Fighter Search**:
- Users can search for specific fighters using the provided search bar.
- The search functionality triggers a GET request to the platform's API endpoint:

```
GET http://localhost:8000/fighterslist?search=name:alex
```

This endpoint filters fighters based on the specified player name.



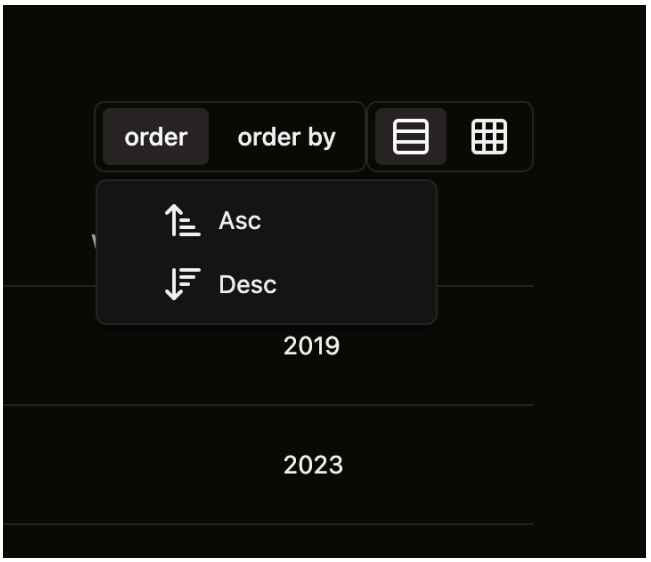
4. **Pagination**:
- Pagination is implemented to display 100 fighters per page, ensuring efficient data presentation.

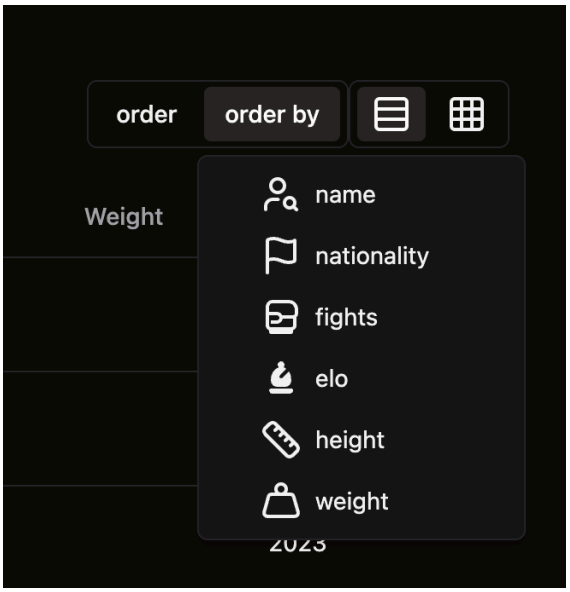
- Users can navigate through multiple pages of fighters using pagination controls.

5. **Filtering**:
- Users can apply filters to sort fighters in ascending or descending order by player name, nationality, number of fights, Elo rating, height, or weight.
- Filters trigger a request to the platform's API endpoint:

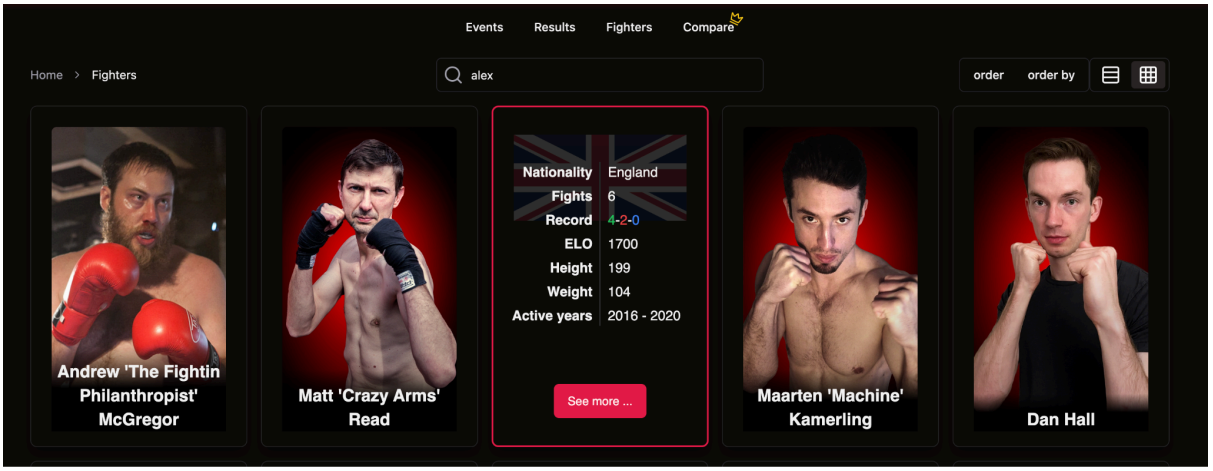
```
GET http://localhost:8000/fighterslist?page=0&orderBy=height&order=asc
```

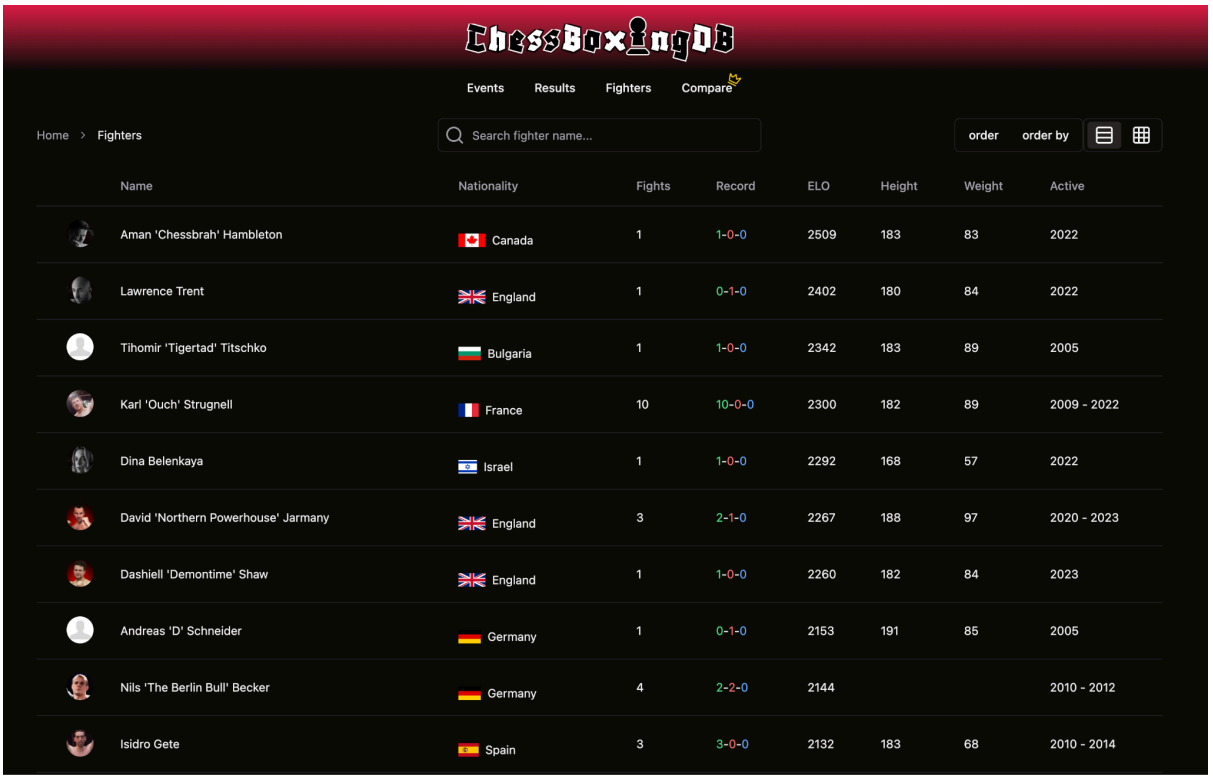
This endpoint retrieves fighter data ordered according to the specified criteria





6. **Detailed Fighter Information**:
- Each fighter card includes a button that redirects users to a detailed page about the particular fighter.
- The detailed page is located at "/fighters/[name].vue" and provides comprehensive information about the selected player.





The "/fighters.vue" page offers users an overview of chessboxing fighters, with interactive card components providing quick access to essential fighter information. Users can search, paginate, and filter fighters to find specific players of interest.

/fighters/[name].vue

The "/fighters/[name].vue" page requires a fighter name parameter to retrieve detailed information about a specific chessboxing fighter. Upon loading, the page sends a request to the following API endpoint:

```
http://localhost:8000/detailedFighters/Cameron%20'The%20Hurt%20Locker'%20Little
```

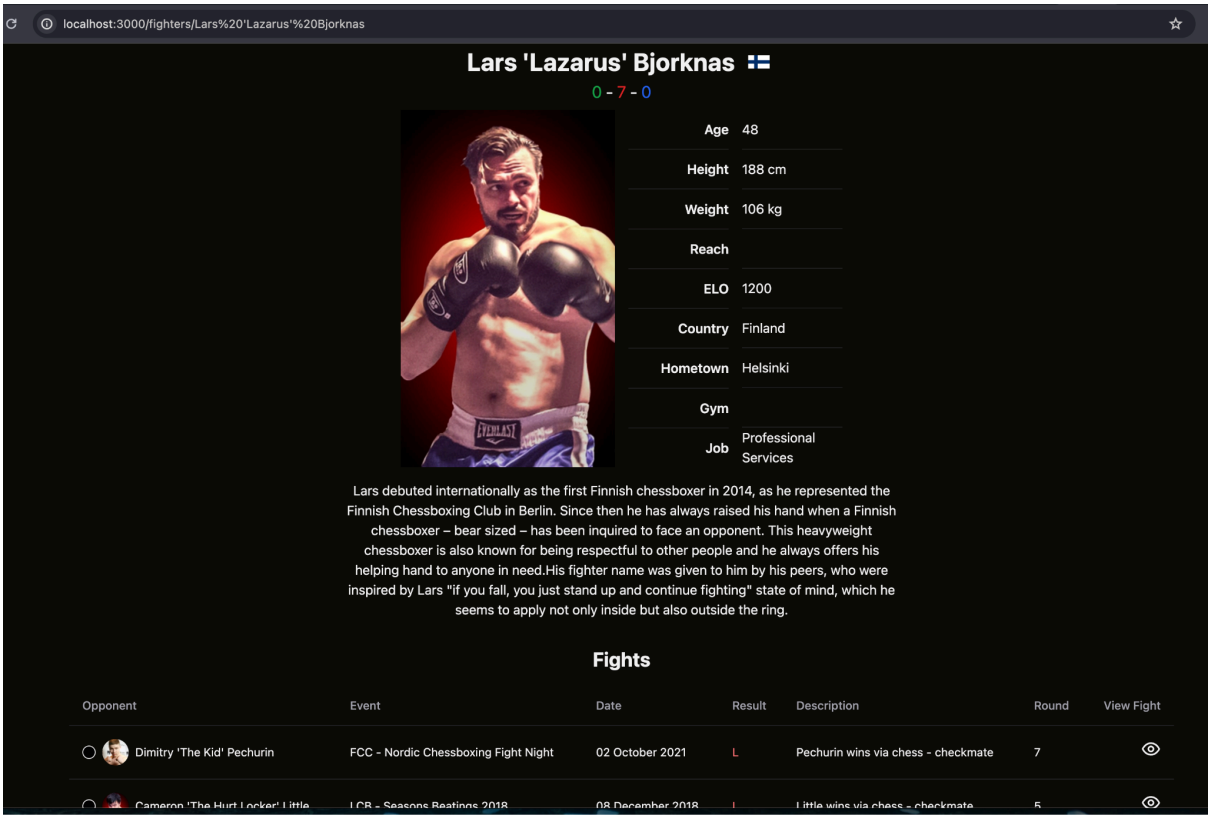
This endpoint fetches detailed information about the fighter named "Cameron 'The Hurt Locker' Little" or any other fighter specified in the URL parameter.

Fighter Information Display:

1. **Basic Information**:
- The page displays essential information about the fighter, including nationality, hometown, gym, job, and a brief description.
2. **Additional Information**:
- Other details such as reach and country of the player are also included to provide comprehensive fighter information.
3. **Fight History**:
- A table lists all the fights that the player has participated in.
- Each row in the table represents a fight and includes details such as:
    - **Opponent name** with chess color and profile picture (PFP)
    - **Event name**
    - **Date** of the fight
    - **Result** (win or loss)
    - **Description** of the outcome (who won and how)
    - Number of **rounds**
    - Button to **view** the fight in detail

Data Retrieval:

The page sends a request to the platform's API endpoint to fetch detailed fighter information based on the provided fighter name parameter.



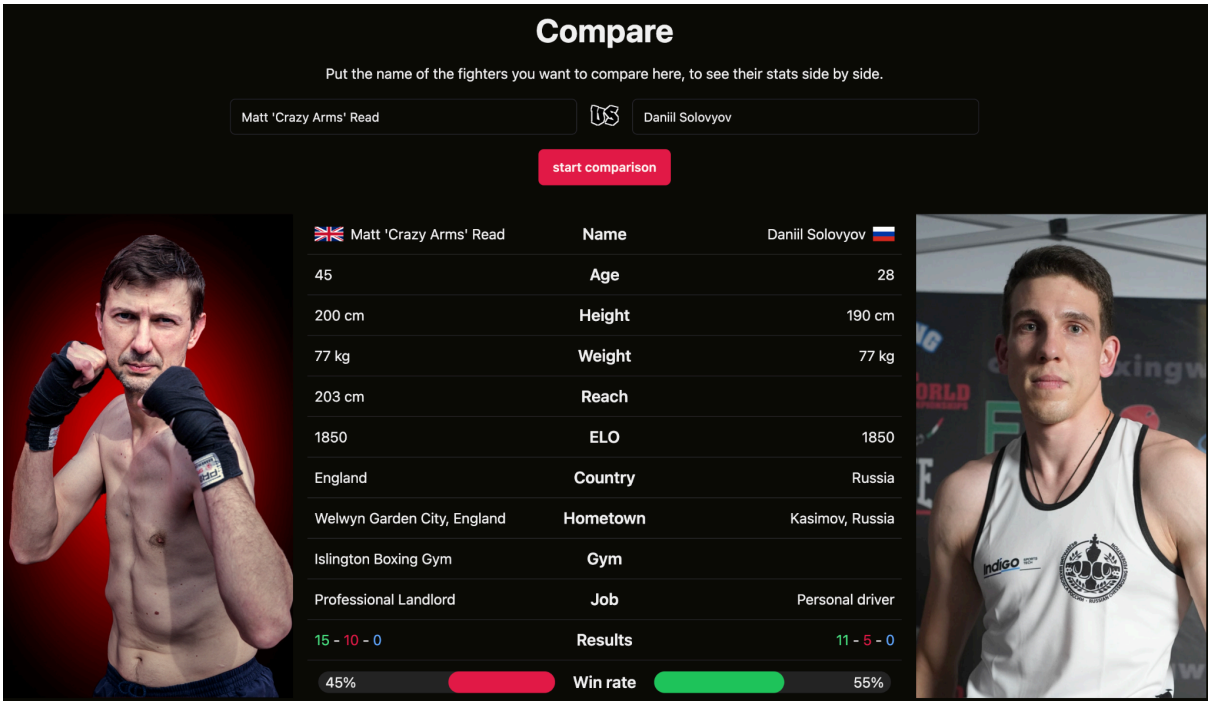
Fight Detail:

- Clicking on the "**View Fight Detail**" button redirects users to a detailed page about the specific fight, providing additional information about the match.

The "/fighters/[name].vue" page offers users comprehensive information about a specific chessboxing fighter, including basic details, fight history, and the ability to view individual fight details.

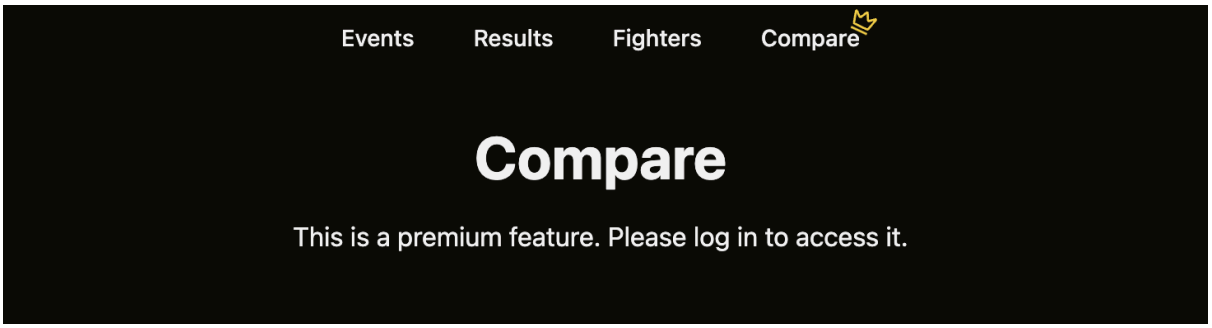
/compare.vue

The "/compare.vue" page is a premium feature accessible only to logged-in users. It allows users to compare two chessboxing fighters based on various statistics and probabilities. Key features of this page include:



1. **Authentication Check**:

- Upon loading the page, a request is sent to the platform's API endpoint /validate to validate the **JWT token** present in the **local storage**.
- If the API returns a **200** status code, indicating a valid token, access to the premium page is granted. Otherwise, the page displays a message prompting users to log in to access the feature.



2. **Comparison Form**:
- Users can input the names of two fighters they want to compare into text inputs on the form.
- After submitting the form by clicking the "Start Comparison" button, a request is made to the protected API endpoint /compare to fetch data for the comparison.
3. **Comparison Display**:
- Once the data is fetched, the page displays a side-by-side comparison of the two fighters.
- The comparison includes:
    - Fighter statistics (similar to those found on the fighterDetails page).
    - Profile pictures (PFP) of the fighters.
    - Win rate percentage with a progression bar indicating the probability of each fighter winning against the other.
    - Color-coded indication of win probability: green for higher win rate and red for lower win rate, providing visual clarity on potential outcomes.
4. **Data Retrieval**:
- The page sends a POST request to the platform's API endpoint /compare with the names of the two fighters provided by the user.
- The API endpoint is protected from unauthorized access by requiring a valid JWT token in the request header.

API Endpoints:

1. **Validation Endpoint**:

```
GET http://localhost:8000/validate
```

2. **Comparison Endpoint**:

```
POST http://localhost:8000/compare
```

Request Headers:

For both the validation and comparison endpoints, the request must include the following authentication header:

```
Authorization: Bearer <JWT Token>
```

Request Body for Comparison Endpoint:

The request body for the comparison endpoint includes the names of the two fighters to be compared:

```
{
  "fighterOne": "Matt 'Crazy Arms' Read",
  "fighterTwo": "Daniil Solovyov"
}
```

The "/compare.vue" page provides users with a powerful tool for comparing chessboxing fighters, offering insights into potential match outcomes based on statistical analysis.

Io dopo aver finito il progetto di TPSI