UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Master's Degree in Artificial Intelligence and Data Engineering

Large Scale and Multistructured Databases Course

EnPassant

https://github.com/team102largedb/ChessTournamentMgmt.git

Students:

Jonathan Frattacci

Mattia Segreto

Federico Calzolari

ACADEMYC YEAR 2024/2025

Contents

Chapter 1

Introduction

Chess tournaments are one of the oldest and most popular entertainment activities ever. While the game itself consits in a battle of logic, the organization of a tournament is a battle against time, resources, and the complexity of managing a large number of people and events.
During a single match there are many things each of the main actors of a chess tournament must keep track of and it's not always easy to do so:

- The players, during a match, usually keep a notebook on their side to keep track of the moves. This is helpful, not only to remember and reanalyze the game, but also to avoid disputes.
- The arbiters must know which games are being played, which games are finished, and which games need their attention.
- The tournament organizers must know how many players are participating, how many games are being played and how the main draws in the different categories are proceding.
- The spectators want to know which games are being played (and watch them), which games are finished, and each player statistics.

EnPassant is a chess management system with the purpose of offering a 360-degree solution for chess players, tournament organizers, and chess enthusiasts. It aims to make each actor's activity easier and digitalize the whole process.

Chapter 2

Design

2.1 Actors

EnPassant considers three roles as main actors:

- Spectators
- Players
- Managers

The application is designed to be used by all three actors, with different functionalities and interfaces for each of them, as a matter of fact, we expect each user to be registered since no functionalities were implemented for unregistered ones.
A Spectator is the kind of user that is just interested in watching the games, following the tournament, and checking the players' statistics.
The Players are users that, in addition to all the operations that a Spectator can do, also play the games.
The Managers, working both as referees and tournament organizers, are able to manage the games, the players, and the tournaments. They also have the authority to review the special requestes sent by players (like re-evalutaion of a game, or checking for misconduct) and to make extraordinary operations like modify a match or disqualify a certain player.

2.2 Mockups

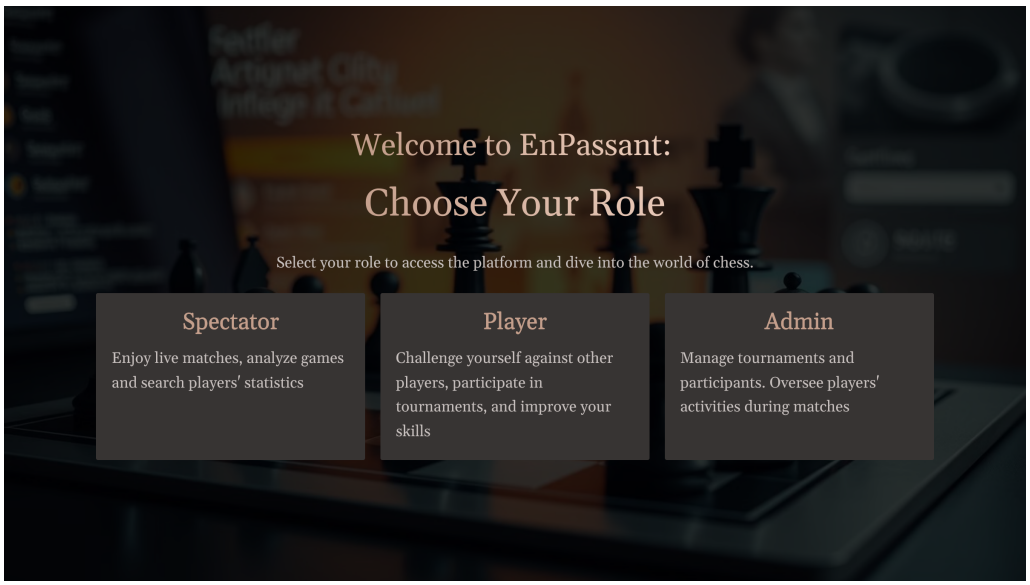


Figure 2.1: Homepage

2.2.1 Login

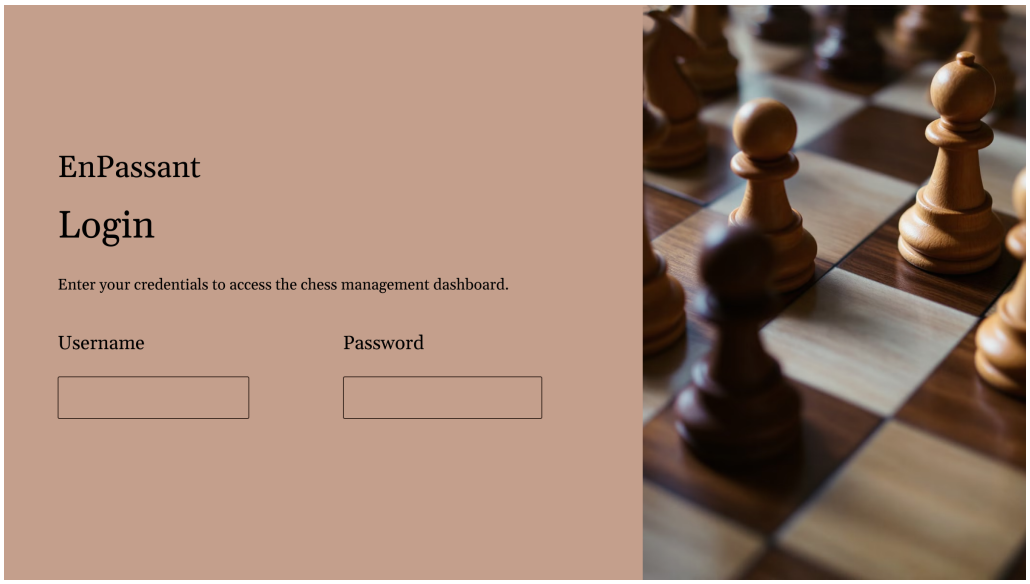


Figure 2.2: Login

2.2.2 Spectator

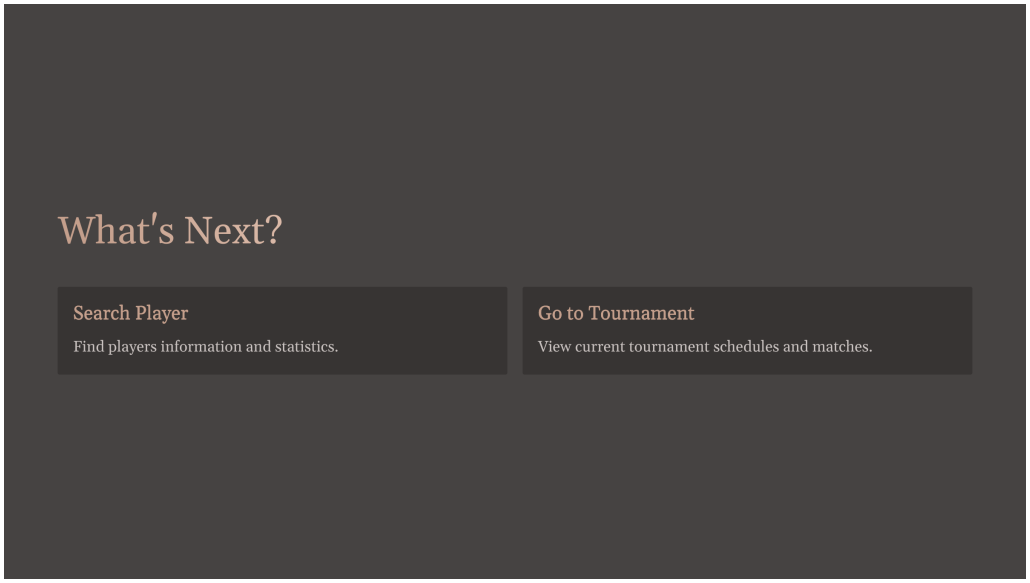


Figure 2.3: Spectator Area

Tournament Section

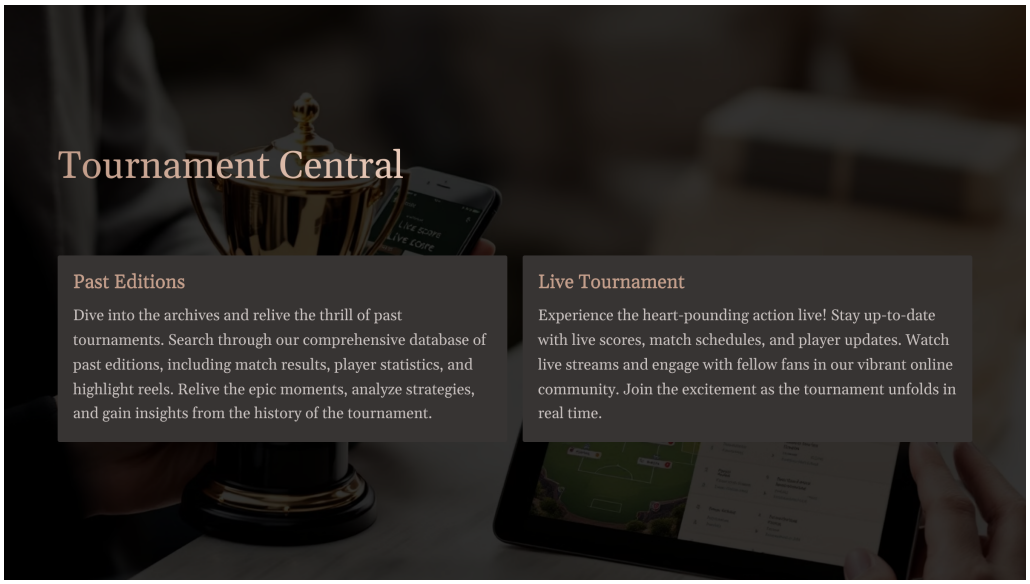


Figure 2.4: Tournament Area

Figure 2.5: Past Tournament Editions

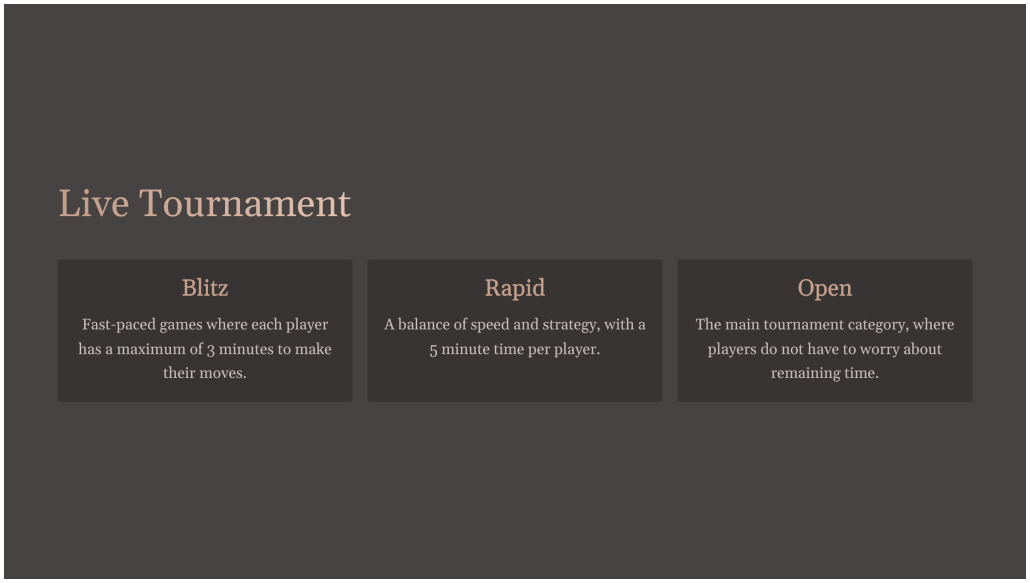


Figure 2.6: Live Tournament

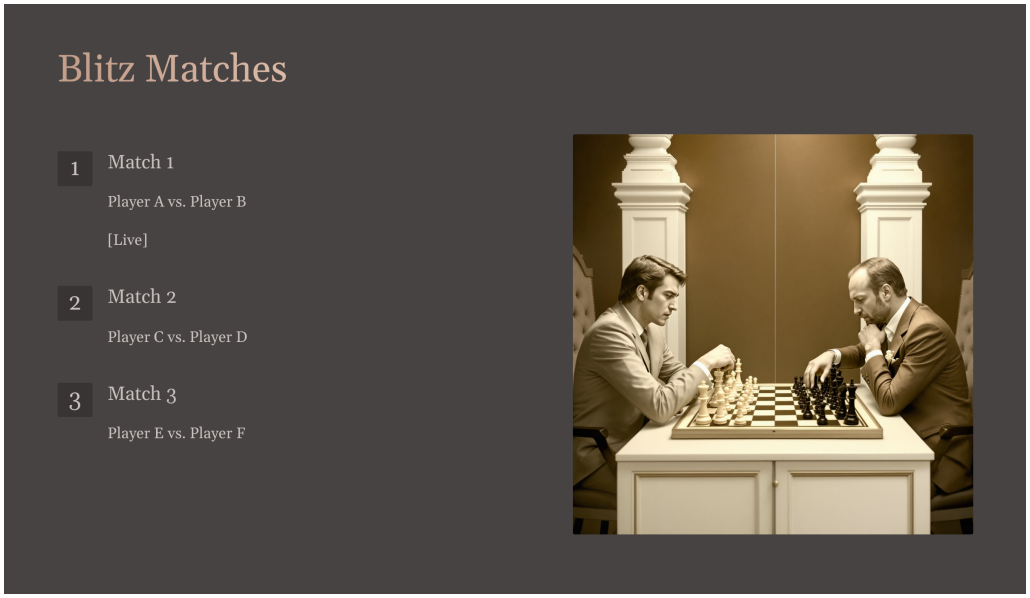


Figure 2.7: Today's Matches

Figure 2.8: Live Match

2.2.3 Player

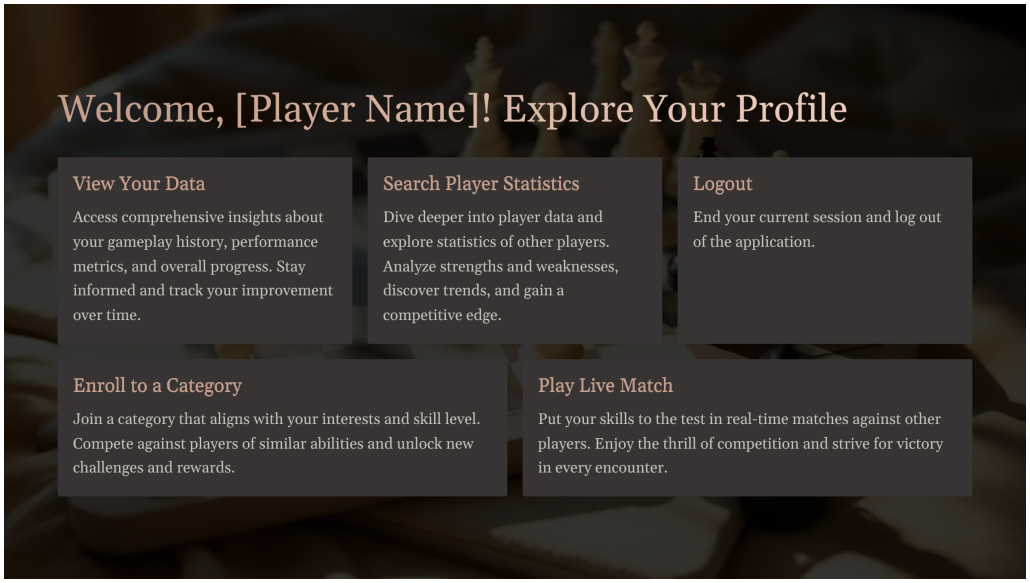


Figure 2.9: Player Dashboard

Figure 2.10: Play Match

2.2.4 Manager

Figure 2.11: Admin Dashboard

2.3 Requirements

2.3.1 Functional Requirements

- The application must ask to users to login and choose a particular role

Spectator

The functionalities that a Spectator can access are also accessible by any other kind of user (Player and Manager) and are the following:

- **Browse through users**
  - The application must allow the user to visualize a user's profile.
  - The application must allow the user to visualize the list of all the users.
  - The application must allow the user look at a specific player's statistics.
  - The application must allow the user to access a set of general statistics retrieval operations
- **Browse through tournaments**
  - The application must allow the user to visualize the list of all the tournaments.
  - The application must allow the user to visualize the list of all the matches of a specific tournament.
  - The application must allow the user to find the match details of a specific match.
  - The application must allow the user to access a set of general statistics retrieval operations
- **Browse through current day matches**
  - The application must allow the user to retrieve the list of all the matches of the current day.
  - The application must allow the user to obtain the details of a specific match of the current day.
  - The application must allow the user to visualize the list of the players disqualified in the current day.
  - The application must allow the user to search a player by id in the disqualified list not notified yet.
  - The application must allow the user to visualize the list of the players enrolled in the current edition of the tournament but have not been notified yet.
  - The application must allow the user to watch a live match.

Manager

- **Routine operations**
  - The application must allow a manager to create a new tournament.
  - The application must allow a manager to delete a tournament.
  - The application must allow a manager to modify a tournament.
  - The application must allow a manager to update the winner of a tournament.
  - The application must allow a manager to create a user account.
  - The application must allow a manager to remove a user account.
  - The application must allow a manager to modify a user account.
  - The application must allow a manager to generate the main draw of a tournament.
  - The application must allow a manager to generate the list of the matches for each day.
  - The application must allow a manager to withdraw a player from a tournament category.
  - The application must allow a manager to disqualify players.
  - The application must allow a manager to insert the result of a match.
  - The application must allow a manager to review a player's request.
  - The application must allow a manager to visualize how many requests are left in the queue.
  - The application must allow a manager to start the synchronization of the application (Redis-to-Mongo step).
- **Extraordinary operations**
  - The application must allow a manager to add a match to a tournament.
  - The application must allow a manager to remove a match from a tournament.
  - The application must allow a manager to add one or more live matches to the list of the current day.
  - The application must allow a manager to remove a match from the list of the current day.
  - The application must allow a manager to remove players from disqualified list.
  - The application must allow a manager to reset the list of the requests.

Player

- The application must allow a player to enroll in a tournament category.
- The application must allow a player to add a move to his own occurring match.
- The application must allow a player to send a request to the manager.

2.3.2 Non-Functional Requirements

- **Product Requirements**
  - The system must be able to provide easy scalability to allow the management of an even larger number of tournaments.
  - The system must be able to provide a high level of availability.
  - The system must be able to provide a high level of performance.
- **Implementation Requirements**
  - The system must be implemented using Object-Oriented Programming languages.
  - The system must be implemented using a distributed database.
  - The system must be implemented using RESTful APIs.
- **Security Requirements**
  - The system must use an authentication system to verify users' identity.
  - The system must encrypt users' passwords.
  - The system must use an authorization system to verify users' permissions.

2.4 UML Class Diagram

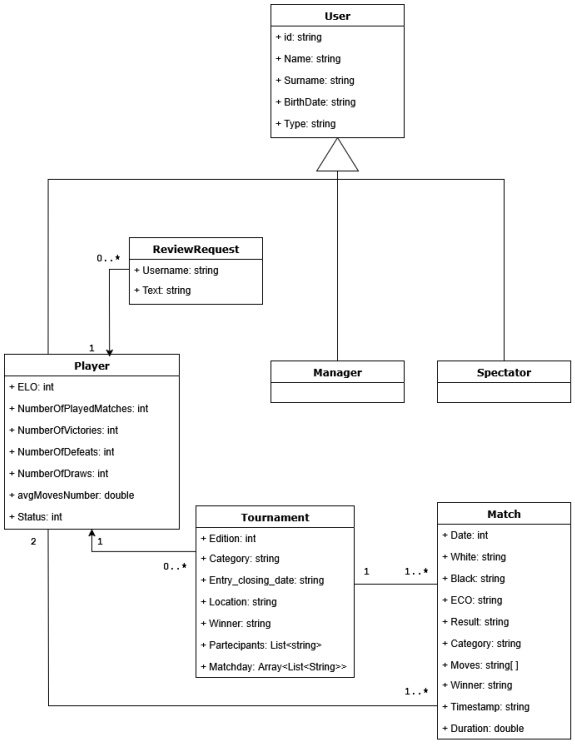


Figure 2.12: Caption

2.5 Data Models

The database was designed to include two main parts: one working as an OLTP database (implemented through a Key-Value DB) and another one workings as an OLAP database (implemented through Document DB). The reason behind this choice is that Redis, being an in-memory database, is perfect for OLTP purposes, while MongoDB, thanks to its versatility, is perfect to handle the kind of queries we considered. It is important to keep this in mind in order to understand the design choices explained in the following paragraphs.

2.5.1 Document Database

The Document Database is designed to store two main collections:

- user
- tournaments

The **tournaments** collection holds all the information concerning past, present and future editions of the competition. Each document in this collection contains details such as the edition, the category, the location, the closing date of the registrations and even the winner (for already concluded tournaments). Let's focus on a fundamental field of the tournaments collection: RawMatches. This field embeds all the

matches of a specific tournament the related information: White, Black, Winner, Result, Category, Date, Duration, ECO (Encyclopedia of Chess Openings) code and Moves (list of all the moves). There are two additional fields that might appear in certain documents: Participants and MatchDays. While the first one is a list of all the players that are participating in the tournament and is used in tournaments that haven't started yet but already accept enrollments, the second one is a list of all the days in which the matches are played. The MatchDays field is actually a list of arrays, each of which contains the matches of a specific day (in the form: white_player-black_player). This kind of implementation with attributes that have a limited Time To Live, was possible thanks to the possibility of denormalization offered by NoSQL databases.
Let's now move to the **user** collection. This collection contains the information of all the registered users, which, as already discussed, are divided into Spectators, Players and Managers. All these users share the data section such as username, name, surname, password and date of birth and are distinguished based on the Type attribute. Players also have a reserved statistics section that provides values such as:

- the number of played matches
- the number of won, drawn and lost matches
- the average number of Moves per match

Another field for this type of user is Match: it consists in an embedding of a series of documents containing the information extracted by the RawMatches list of tournaments.

User Collection

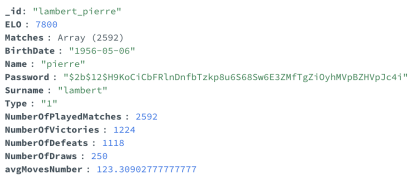


Figure 2.13: User Collection

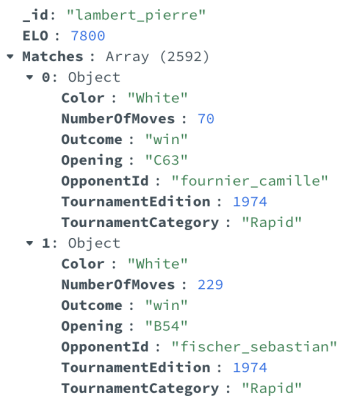


Figure 2.14: User Collection Matches

Tournaments Collection


```
_id: ObjectId('67b06f6c32c14fefb06243d2')
Edition : 1973
Category : "Open"
Entry_closing_date : "1973-02-11"
Location : "Chennai"
Winner : "dupont_antoine"
▸ RawMatches : Array (755)
```


Figure 2.15: Tournaments Collection


```
_id: ObjectId('67b06f6c32c14fefb06243d2')
Edition : 1973
Category : "Open"
Entry_closing_date : "1973-02-11"
Location : "Chennai"
Winner : "dupont_antoine"
▾ RawMatches : Array (755)
  ▾ 0: Object
      Date : 1973
      White : "mehta_anjali"
      ECO : "A04"
      Black : "kim_hana"
      Result : "0-1"
      Category : "Open"
    ▸ Moves : Array (206)
      Timestamp : "1973-03-28 05:22:30"
      Duration : 29.3
      Winner : "kim_hana"
  ▾ 1: Object
      Date : 1973
      White : "mehta_anjali"
      ECO : "B06"
      Black : "wagner_tobias"
      Result : "1-0"
      Category : "Open"
    ▸ Moves : Array (130)
      Timestamp : "1973-03-30 14:06:47"
      Duration : 487.55
      Winner : "mehta_anjali"
```


Figure 2.16: RawMatches


```
_id: ObjectId('67b06f6c32c14fefb06243d2')
Edition : 1973
Category : "Open"
Entry_closing_date : "1973-02-11"
Location : "Chennai"
Winner : "dupont_antoine"
▾ RawMatches : Array (755)
  ▾ 0: Object
      Date : 1973
      White : "mehta_anjali"
      ECO : "A04"
      Black : "kim_hana"
      Result : "0-1"
      Category : "Open"
    ▾ Moves : Array (206)
        0: "Nf3"
        1: "d6"
        2: "2"
        3: "d4"
        4: "g6"
        5: "3"
        6: "e4"
        7: "Bg7"
        8: "4"
        9: "Bd3"
        10: "e5"
        11: "5"
        12: "c3"
        13: "exd4"
```


Figure 2.17: RawMatches Moves

2.5.2 Key-Value Database

Let's now analyze the key-value database, which is designed to store the following buckets: **Request, EnrolledPlayer, DisqualifiedPlayer e LiveMatches.**.
The Request bucket contains a list of keys in the form:
$Request : progressive\_number : \{progressive\}$ each of which contains a progressive number in order to keep track of the order of the requests and a value of JSON type with two fields: username and text. There are also two other keys:

- $Request : min\_progressive$
- $Request : max\_progressive$

which are used to manage the queue so that it has a FIFO policy.
The EnrolledPlayer and DisqualifiedPlayer buckets have relatively simple keys and structure, in fact they only have one key which is respectively $disqualified\_players$ and $registered\_players$ which stores an array of strings.
The last bucket in the key-value database is LiveMatch which contains the following keys:

- $Live : matches$: contains the list of all the matches that are currently being played
- $Live : \{matchId\}$: is associated with a JSON type value that holds a set of data of this format: $category, startingTime, winner, endTime, ECO$.. The choice of grouping all this information in a single key was made since these data are always used together made exeption for the initialization phase.
- $Live : \{matchId\} : progressive$: makes it possible to keep track of the number of moves played. By making the 2-modulus we can find out whose turn is the current one.
- $Live : \{matchId\} : moveList$: holds a list of the moves in the form: $progressive.move$

This buckets are fundamental in order to keep the Document DB as much read-heavy as possible. Thanks to the fast nature of Redis, adding some write request is not a such a problem in terms of request loading (we also have to consider that, in this context, the sharding helps us deal with this)

```
//REQUEST BUCKET
private static final String MIN_PROGRESSIVE_KEY = "Request:min_progressive";
private static final String MAX_PROGRESSIVE_KEY = "Request:max_progressive";
private static final String REQUEST_PREFIX = "Request:progressive_number:" + progressive + ":";

//DISQUALIFIED PLAYER BUCKET
private static final String DISQUALIFIED_PLAYERS_KEY = "disqualified_players";

//REGISTERED PLAYER BUCKET
private static final String REGISTERED_PLAYERS_KEY = "registered_players";

//LIVE MATCH BUCKET
 private static final String LIVE_MATCHES_KEY = "Live:matches";
 private static final String LIVE_MATCHES_KEY = "Live:matches:" + matchId + ":";
 private static final String LIVE_MATCHES_KEY = "Live:matches:" + matchId + ":progressive:";
 private static final String LIVE_MATCHES_KEY = "Live:matches:" + matchId + ":moveList:";
```

Figure 2.18: Key-Value Buckets

2.6 Distributed Database Design

Given the requirements of the application, in order to provide high availability, scalability and performance, we distributed the database on the three-nodes-cluster we were assigned, as explained in the following paragraphs. MongoDB and the LiveMatches Redis bucket implement the CA configuration of the CAP theorem, while the other Redis buckets implement the AP one.

2.6.1 Mongo Replicas

For the Document Database three replicas were implemented: one for each node in the cluster. The replica set is made of one primary node and two secondary nodes. The nodes are configured with different priorities, one with priority 5, making it the most likely to become the primary node when the election starts and two with a lower priority of 2 and 1, which will be the secondary nodes. Since it was designed to not receive writes when working correctly, if not in sporadic cases or in any case that exceeds the normal course of the application's use, Strong Consistency was implemented. This led to a heavy penalization of the writes but in return, the readings were highly improved. This choice is in line with the policies illustrated so far and does not represent a problem.

Write Concern

The write concern was set to 3, which means that the write operation is considered successful only when the data has been written to the primary node and to both the secondary nodes. This policy ensures that the data is replicated on all the nodes, guaranteeing consistency and durability. This idea is in line with the Strong Consistency policy adopted for the Document Database, and is supported by the fact that the application is not write-intensive. As a matter of fact, during the normal course of the application, the writes only happen once a day when the data gets transferred between the two databases, so it does not represent a bottleneck (even in case of extraordinary operations, the write requests that were implemented are of the kind that doesn't need an immediate response).

Read Preference

The read preference was set to nearest, which means that the read operation is executed on the node with the lowest latency. This policy ensures that the reads are performed as quickly as possible, providing a high level of performance. As for the writing, the reading is not penalized by the Strong Consistency policy, since the application is read-intensive and the data is not updated in real-time.

2.6.2 Redis Replicas

The key value database, was also distributed on three nodes of a cluster but this time there was a difference in the policies to be implemented. We had to deal with buckets that required different needs and for this reason the Key-Value DB was

managed in two different ways in order to speed up as much as possible what we could, and to provide consistency where required:

- As far as the following buckets are concerned: **Request, EnrolledPlayer, DisqualifiedPlayer**, eventual consistency was implemented since not seeing the updated data in real time does not harm any of the users as long as in the end each of the servers has the updated copy. This is the reason why we decided to implement the writings in such a way they work on the master node, which then propagates the updates to the replicas (which are Read-Only from our point of view).

- In the **Live match** bucket, if we had implemented the same policy as before, we would have risked problems in the insertion of the moves, for this reason the solution we thought about is writing to this specific bucket to be considered completed only once replicated three nodes.
  Implementing eventual consistency in this case might have led to a failure in the inserting of a new move. This failure would make it necessary to retry the upload which would mean more waste of time. On the opposite, our solution takes advantage of the fact that Key-value databases are in-memory DBs and consequently much faster than the document ones, to sacrifice a little bit of time to wait the writings to be propagated to all the replicas, in order to make sure to have consistent data on whatever node the application tries to access.

In conclusion we handled the Key-Value Database as if there were two different Key-Value DataBases since different subsets of buckets required were different needs.

2.7 Sharding

2.7.1 Document Database

Sharding strategy for User collection

In order to guarantee an efficient distribution of data and optimize queries, we propose a hybrid sharding strategy that uses a composite key. The fragmentation is mainly based on a hash of the player's _id (hashed _id), ensuring an even distribution of the documents across the shards and preventing load imbalances. This solution ensures an efficient distribution of documents, while maintaining flexibility in database query operations.

Match handling through Sharding by hand

In addition to the key-based sharding strategy, in the presence of a high number of matches recorded for a single user, it may be useful to adopt a "sharding by hand" approach. Instead of memorizing all the matches in a single document, the user's profile can be divided into multiple instances, keeping the main data (such as ELO

and player type) unchanged but distributing the array of Matches among multiple documents. For example, a user document with a high number of embedded matches could be divided into multiple documents based on the year of the tournaments played (TournamentEdition), assigning each of them to a specific shard.
This approach allows to distribute the load of the user's profile across multiple shards, improving the efficiency of read and write operations and avoiding the document growth problem. Thanks to this, the queries involving the user's matches can be executed on a smaller portion of the dataset, reducing the computational load and ensuring greater efficiency in read and update operations. It is however necessary, to adopt mechanisms that avoid the dispersion of user information across shards. This is fundamental in order to preserve the coherence of the statistics and to allow a correct data retrieval when necessary. This solution makes it possible to balance the load on the shards improving scalability and the overall system performances.

Sharding Strategy for the Tournaments collection

For the tournaments collection it is possible to adopt a combined sharding strategy on `_id` and `Edition` in order to improve query performance. Since every tournament is uniquely and distinctly identified, the hashed `_id`-based, ensures an even distribution of the tournaments on the cluster, avoiding load unbalances across shards. However, it is also useful to include the (`Edition`) field as secondary sharding criteria; this is because it is common for our queries to filter tournaments based on the year of edition.

Embedded Documents handling and why to avoid Sharding by hand

Differently from the User collection, where the number of saved matches for each player can vary in an unpredictable way, in the tournaments collection it's possible to estimate the number of embedded documents (`RawMatches`) so that it is possible to handle them a priori. Thanks to this we can structure the documents in such a way they don't grow too big, making it unnecessary to shard them by hand. The reason behind the decision of embed the matches inside the tournament document is to allow the queries to work more efficiently and not to have to handle complex aggregations.

2.7.2 Key-value Database

JedisCluster Sharding Approach

When an operation is executed using JedisCluster, it goes trough four main stages:

1. **Slot calculation**: JedisCluster internally calculates the correct hash slot using the same algorithm used by Redis Cluster.

2. **Node individuation**: once the slot is determined, JedisCluster consults the cluster state.

3. **Operation execution**: the request is sent to the correct node.

4. **Dynamic Reconfiguration**: if the cluster gets modified, JedisCluster automatically updates the slots map.

Difference between Modular Hashing and Slot Hash in Redis Cluster

In the classical modular hashing technique, the key is transformed into a numerical value using a hash function (e.g., `hash(key) % num_nodes`), and the result directly determines the node that will handle the key. However, this method has a significant issue: if the number of nodes changes, almost all keys need to be reassigned, causing a high rebalancing cost. Redis Cluster solves this problem with the concept of 16384 fixed hash slots, which act as intermediaries between keys and nodes. The assignment process works as follows:

1. **Slot hash calculation**: Redis Cluster applies the function `CRC16(key) % 16384` to obtain one of the 16384 available slots.

2. **Mapping slots to nodes**: Redis nodes are not chosen directly using a modulo operation, but each node is responsible for a certain number of slots. If a node is added or removed, only a portion of the slots is transferred.

3. **Automatic redirection**: If a key is sent to the wrong node, Redis Cluster responds with a `MOVED` message, indicating the correct node, and the client (e.g., `JedisCluster`) updates its internal map.

Difference between Consistent Hashing and Slot Hash in Redis Cluster

Redis Cluster does not implement traditional Consistent Hashing but instead uses a fixed-slot hash system that allows for more predictable and less costly data rebalancing across nodes.

- **Consistent Hashing**, as used in Memcached, is based on a hash ring where nodes are mapped onto a continuous hash interval.

- **Redis Cluster** uses slot hashing, a fixed number of 16384 predefined buckets. Each node is responsible for a subset of these slots.

Advantages of Slot Hashing over Modular Hashing and Consistent Hashing

Aspect	Modular Hashing	Consistent Hashing	Slot Hash Redis Cluster
Distribution	`hash(key) % num_nodes`	Hash on ring	`CRC16(key) % 16384`
Scalability	Total rebalancing	Partial	Only some slots transferred
Failover	Complex	Dynamic redistribution	Redis Cluster manages slot migration
Efficiency	Fast but not scalable	Good distribution	High scalability

Table 2.1: Comparison between Modular Hashing, Consistent Hashing, and Slot Hashing in Redis Cluster

Advantages and Disadvantages of JedisCluster Sharding

- **Advantages**
  - Horizontal scalability.
  - High Availability.
  - Transparent management of the distribution of the keys.
- **Disadvantages**
  - Limitations on distributed operations.
  - Possible latency in case of load redistributions.
  - Overhead introduced by the handling of slot map.

2.8 Intra-Database Consistency

Our intra-database consistency needs to be managed at two distinct moments:

1. **End of the Day:** At the end of the day, data is taken from the key-value store, and the document database is updated accordingly. The various buckets are handled as follows:
   (a) **Request:** The list of pending requests is emptied since there is no corresponding structure in our document database. However, requests are not lost; all pending requests are returned to the manager, who is then free to handle them as they see fit.

(b) **Disqualified List:** For each player in the disqualified list, a new field called `status` is added to the corresponding user document, and it is set to 1. This field is the key used to search for disqualified users. Until this moment, players are considered to be awaiting disqualification notification.

(c) **Enrolled List:** Players are initially sorted based on their respective categories. Existence checks for the user are performed both at the time of enrollment (which must be logged) and at the time of insertion to recover any errors caused by an extraordinary flow of operations. The created lists are then inserted into the appropriate tournament, creating a temporary field called `Participants`, which will later be used to generate the tournament Main Draw. Until this moment, players are considered to be awaiting enrollment notification.

(d) **Live Match:** This phase involves the most consistency checks and is the most delicate part of updating the document database, as it requires updating both collections. The process proceeds as follows:

i. The validity of the match is checked (including outcomes, for example).
ii. The match is appended to the `RawMatches` field of the appropriate tournament.
iii. The `Tournaments` collection is updated, and the two players who participated in the match need to be updated.
iv. The statistical fields of the players, such as the number of matches played, won, lost, drawn, and the average number of moves, are updated.
v. The information of the matches are extracted from the newly inserted match and adapted to the required fields in the `match` (embedded in the user), followed by an append operation.

At this point, the databases are flushed to clean them in preparation for the next day.

2. **Beginning of the Day:** The manager can decide to load the day's matches. However, since the key-value store is empty and the data in the document database is consistent, no particular checks are required, making this a relatively safe operation. At this point, the day can begin with reads and writes, and at the end of the day, step one is repeated.

2.9 Key Eviction Policy

Our choice is not to directly manage this eventuality. The reasoning behind this is that the high memory usage of our key-value database is primarily given by the number of live matches. However, this number can be easily managed to prevent reaching the memory limit, as it is possible, knowing the dimension occupied by a single match, to estimate in advance how much space the whole bucket will occupy.

If, for any reason, we still wanted to load more matches into the database, the manager could simply:

- Load the morning matches.
- Synchronize during the lunch break.
- Load the afternoon matches.
- Synchronize at the end of the day.

Chapter 3

Dataset

The focus of this application is handling a large scale of chess data and managing it. For this reason, we opted to retrieve the necessary information from two sources both coming from kaggle datasets.
From the first dataset (Dataset1) we retrieved the majority of our data which was used to fill the MongoDB collections.
From the second one (Dataset2), we retrieved some real life world tournament games that were mainly used to test insert operations with Redis.
To construct a suitable dataset for our application, we developed several script to clean, process, and organize raw data. Given the high volume of data and the time required to process it, we retrieved only a fraction of the entire available databases. We extracted game records from Kaggle datasets, which contain hundreds of thousands of chess matches played on online platforms. To refine our dataset, we filtered out games that lacked critical metadata, ensuring each match had valid names, openings, timestamps, results and move sequences.

For players, we assigned a unique identifier (surname_name by convention) to each of them in the matches dataset and then developed a script (javascript) which would extract players from our dataset and recollect their information in order to create the user collection. The script maps usernames to unique players' name, surname, birthdate and password. In parallel, we processed the world tournament game records, substituting usernames with our standard player identifiers.

The processed data was stored in separate CSV files for easy integration into our database system and then, for the final update of the tournaments, we used a json file that would recreate the exact structure of the collection. Since the Redis DB is allocated in memory, we didn't need to allocate data inside of it since it would all be lost after the server was shut down. What we did instead was developing a Python script that would simulate a live match and update the moves in real time and at the same time it makes it possible for a spectator to watch the game live.

At the end of the elaboration, we obtained the following files for MongoDB filling:

- filtered_tournaments.json (263.6 MB)
- insert_tournaments.json (3 MB)
- User_steps.js

Chapter 4

Implementation

The EnPassant software is built using Java and the Spring framework to elaborate the requests. The organization of the codebase is divided in the *controller*, *service* and *repository* structure.

4.1 Spring

The Spring framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure with the use of beans and the dependency injection pattern. Another useful feature of the Spring framework is the Spring Security token system, which is used to have a persistent login without the need of a session. This JWT token (JSON Web token) is created during the login process and must be included in the header of the following requests to ensure a quick and secure authentication. This token allows to protect the endpoints of the application, ensuring that only the authenticated users with the correct role can access them.

```
.authorizeHttpRequests( AuthorizationManagerRequestMat… auth -> auth
        // Login is permitted to unauthenticated users
          .requestMatchers(@"/api/player/login").permitAll()
          .requestMatchers(@"/api/manager/login").permitAll()
          .requestMatchers(@"/api/spectator/login").permitAll()
          .requestMatchers(@"/api/player/register").permitAll()
          .requestMatchers(@"/api/manager/register").permitAll()
          .requestMatchers(@"/api/spectator/register").permitAll()
          .requestMatchers(@"/api-docs.html", @"/swagger-ui/**", @"/v3/api-docs/**").permitAll()
        // All other endpoints require authentication
          .requestMatchers(@"/api/tournaments-analytics").authenticated()
          .requestMatchers(@"/api/user-analytics").authenticated()
          .requestMatchers(@"/api/userCRUD/read/**").authenticated()
          .requestMatchers(@"/api/tournamentsCRUD/read/**").authenticated()
          .requestMatchers(@"/api/LiveMatch/getLiveMatches").authenticated()
          .requestMatchers(@"/api/LiveMatch/matchDetails/**").authenticated()
          .requestMatchers(@"/api/LiveMatch/retrieveMoveList/**").authenticated()
          .requestMatchers(@"/api/managePlayer/disqualifiedList").authenticated()
          .requestMatchers(@"/api/managePlayer/disqualifiedList").authenticated()
          .requestMatchers(@"/api/managePlayer/registerList").authenticated()
          .requestMatchers(@"/api/managePlayer/registerList").authenticated()
          .requestMatchers(@"/api/managePlayer/disqualified/{playerId}").authenticated()
```

```
// Manager endpoints require manager role ("0")
  .requestMatchers(@"/api/manager/**").hasAuthority("0")
  .requestMatchers(@"/api/tournament/matchMaking/**").hasAuthority("0")
  .requestMatchers(@"/api/requests/next").hasAuthority("0")
  .requestMatchers(@"/api/requests/reset").hasAuthority("0")
  .requestMatchers(@"/api/requests/size").hasAuthority("0")
  .requestMatchers(@"/api/appUpdate").hasAuthority("0")
  .requestMatchers(@"/api/userCRUD/**").hasAuthority("0")
  .requestMatchers(@"/api/tournamentsCRUD/**").hasAuthority("0")
  .requestMatchers(@"/api/LiveMatch/addLiveMatch/**").hasAuthority("0")
  .requestMatchers(@"/api/LiveMatch/removeLiveMatch/**").hasAuthority("0")
  .requestMatchers(@"/api/LiveMatch/addLiveMatches").hasAuthority("0")
  .requestMatchers(@"/api/managePlayer/disqualify/**").hasAuthority("0")
  .requestMatchers(@"/api/LiveMatch/insertMatchResult/**").hasAuthority("0")
// Player endpoints require player role ("1")
  .requestMatchers(@"/api/player/**").hasAuthority("1")
  .requestMatchers(@"/api/LiveMatch/insertMoves/**").hasAuthority("1")
  .requestMatchers(@"/api/requests/insert").hasAuthority("1")
  .requestMatchers(@"/api/managePlayer/register/{playerId}/{category}").hasAuthority("1")
// Spectator endpoints require spectator role ("2")
  .requestMatchers(@"/api/spectator/**").hasAuthority("2")
// Register endpoints require manager or player role
  .requestMatchers(@"/api/managePlayer/register/**").hasAnyAuthority( …authorities: "0", "1")
  .requestMatchers(@"/api/managePlayer/register/{playerId}").hasAnyAuthority( …authorities: "0", "1")
  .anyRequest().authenticated()
```

Figure 4.1: Endpoints protection

4.2 Packages

4.2.1 Controller

The endpoints are protected not only by the JWT token, but also by the role of the user to ensure the maximum security.

Mongo controllers

This series of controllers are used to manage the requests that involve the MongoDB database. The controllers handles the login and registration requests, the creation of new tournament and matches, the retrieval of statistics and much more.

Redis controllers

This controlles are used to manage the requests that involve the Redis database. The controllers handle the creation of new live matches, the retrieval of their information, and the update of the moves. They also handle the creation of new review requests.

Redis and Mongo controllers

This controller is used to manage requests that involve both the Redis and the MongoDB databases. The requests are related to the management of scheduled matches and the saving of games from Redis to MongoDB.

4.2.2 Service

The service package contains the classes that handle the business logic of the application. It is used in between the controller and the repository to manage the requests and the responses.

Authentication services

This services implements the main functionalities of the authentication process. It handles the logic of the login, the generation of the JWT token and the management of the roles.

Mongo services

This services implements the main functionalities of the MongoDB database. It handles all the functionalities that use MongoDB, such as the creation of new tournaments and matches, the retrieval of statistics and the management of the users.

Redis services

This services implements the main functionalities of the Redis database. It handles all the functionalities that use Redis, such as the creation of new live matches, the retrieval of their information and the update of the moves. It also manage the server replicas to ensure a working synchronization.

4.2.3 Repository

This package contains the classes that handle the communication with the databases, simplifying the implementation of the API. The repository classes are used to manage the queries and the updates of the databases.

4.2.4 Model

We list the principal models of our system

```java
public class DataTournamentMatchModel
    @JsonProperty("edition") 3 usages
    private int edition;

    @JsonProperty("category") 3 usages
    private String category;

    @JsonProperty("location") 3 usages
    private String location;

    @JsonProperty("matchDate") 3 usages
    private String matchDate;

    @JsonProperty("whitePlayer") 3 usage
    private String whitePlayer;

    @JsonProperty("blackPlayer") 3 usage
    private String blackPlayer;

    @JsonProperty("eco") 3 usages
    private String eco;

    @JsonProperty("result") 3 usages
    private String result;

    @JsonProperty("moveList") 3 usages
    private List<String> moveList;
```

(a) DataTournamentMatchModel

```java
public class TournamentsAnalyticsModel {
    @JsonProperty("userId") 6 usages
    private String userId;

    @JsonProperty("eco") 6 usages
    private String eco;

    @JsonProperty("category") 6 usages
    private String category;

    @JsonProperty("edition") 6 usages
    private Integer edition;

    @JsonProperty("howMany") 6 usages
    private Integer howMany;

    @JsonProperty("averageMatchDuration") 6 usages
    private Double averageMatchDuration;

    @JsonProperty("numberOfMatches") 6 usages
    private Integer numberOfMatches;

    @JsonProperty("whiteWinPercentage") 6 usages
    private Double whiteWinPercentage;

    @JsonProperty("drawPercentage") 6 usages
    private Double drawPercentage;

    @JsonProperty("blackWinPercentage") 6 usages
    private Double blackWinPercentage;
```

(b) TournamentsAnalyticsModel

```java
public class UserAnalyticsModel {

    @JsonProperty("userId")  4 usages
    private String userId;

    @JsonProperty("mostFrequentOpening")  4 usages
    private String mostFrequentOpening;

    @JsonProperty("howMany")  4 usages
    private int howMany;
```

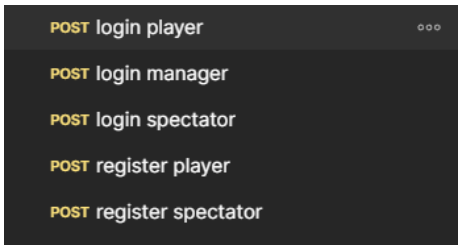
(a) UserAnalyticsModel

```java
public class DocumentUser {

    @Id
    private String id;

    private String BirthDate;  5 usages
    private String Name;  5 usages
    private String Password;  5 usages
    private String Surname;  5 usages
    private String Type;  5 usages
    private Integer ELO;  5 usages
    private Boolean Status;  5 usages
    private List<DocumentMatchUser> Matches;  5 usages


    private Integer NumberOfPlayedMatches;  5 usages
    private Integer NumberOfVictories;  5 usages
    private Integer NumberOfDefeats;  5 usages
    private Integer NumberOfDraws;  5 usages
    private Double avgMovesNumber;  5 usages
```

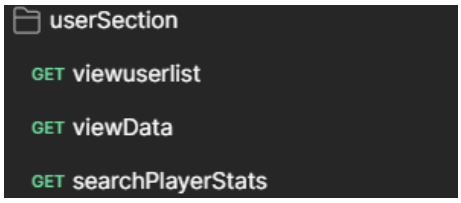
(b) DocumentUser

4.3 RESTful Endpoints

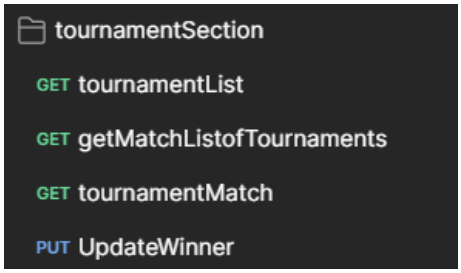
Login APIs



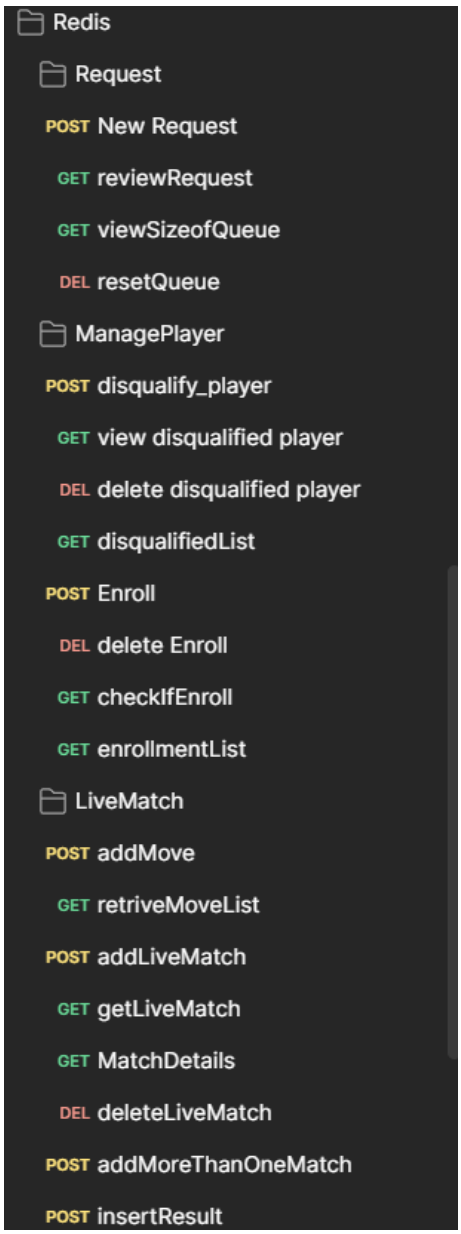
User APIs



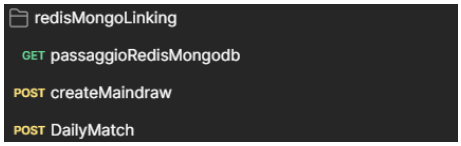
Tournament APIs



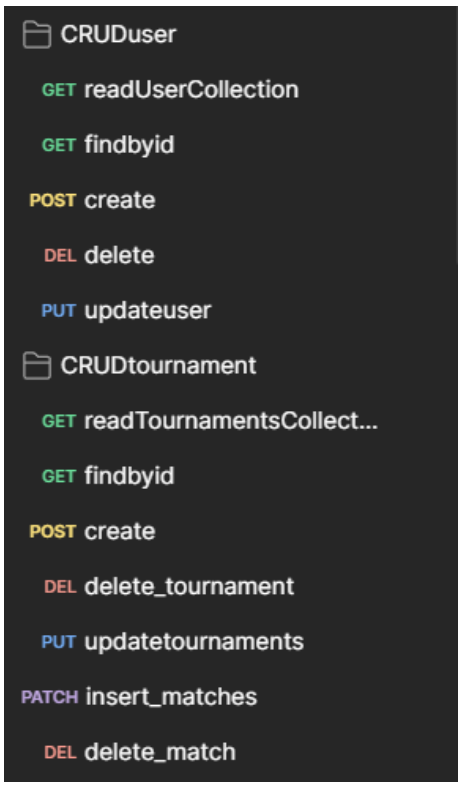
Redis APIs



Redis and Mongo APIs



CRUD APIs


CRUDuser
GET readUserCollection
GET findbyid
POST create
DEL delete
PUT updateuser
CRUDtournament
GET readTournamentsCollect...
GET findbyid
POST create
DEL delete_tournament
PUT updatetournaments
PATCH insert_matches
DEL delete_match

Chapter 5

Queries and Indexes

5.1 Relevant Queries

5.1.1 MongoDB

The most relevant queries that work on the document database use aggregation pipelines. In the following we provide a description of their functionality:

- **Counting Wins for each User enrolled in a given Category of a given Edition.**
  This query aggregates match data to determine how many games each user has won in a specific category of a given edition.
  The output provides a list of users who have won matches, categorized by edition and game category, along with their respective win counts.
  The aggregation follows these steps:

  1. **Unwind**: expands the `RawMatches` array, so each match result is processed individually.
  2. **Match**: filters records to exclude those where the outcome is a draw and the match does not belong to the edition and category specified by the arguments.
  3. **Project**: extracts only relevant fields: `Edition`, `RawMatches.Winner` (as `winner`), and `Category`.
  4. **Group**: groups by edition, category, and winner, counting the total number of games won (`wonGames`).
  5. **Sort**: orders results by `edition`, `category`, and `userId`.

```
@Aggregation(pipeline = {  1 usage   jonnyfratta
        "{ $unwind: '$RawMatches' }",
        "{ $match: { 'RawMatches.Winner': { $exists: true, $ne: 'draw' }, 'Edition': ?0, 'Category': ?1 } }",
        "{ $project: { edition: '$Edition', winner: '$RawMatches.Winner', category: '$Category'} }",
        "{ $group: { _id: { edition: '$edition', category: '$category', winner: '$winner' }, wonGames: { $sum: 1 } } }",
        "{ $project:  {userId:  '$_id.winner', category:  '$_id.category', edition: '$_id.edition', howMany: '$wonGames'}}",
        "{ $sort: { 'edition': 1, 'category': 1, 'userId': 1 } }"
})
List<TournamentsAnalyticsModel> countGamesWonByPlayerPerEdition(int edition, String category);
```

- **Statistical Analysis of a Specific Opening.**
  This query aggregates match data to compute statistics for a specific opening, identified by its **ECO** code. The output displays the number of games played and the win percentages for White, Black, and draws. The aggregation follows these steps:

  1. **Unwind**: expands the `RawMatches`.
  2. **Match**: filters the matches to include only those with:
     - a valid result among:
     - the specific `ECO` code passed through the argument.
  3. **Group**: aggregates matches by `ECO` code, computing:
     - `totalMatches`: total number of matches played with this opening.
     - `whiteWins`: number of games won by White.
     - `draws`: number of drawn games.
     - `blackWins`: number of games won by Black.
  4. **Project**: computes the win and draw percentages for this opening: `whiteWinPercentage`, `drawPercentage`, `blackWinPercentage`.
     If no matches exist for a specific ECO code, percentages by default 0.
  5. **Sort**: orders results by `ECO` code.

```
@Aggregation(pipeline = {  1 usage   jonnyfratta
        "{ $unwind: '$RawMatches' }",
        "{ $match: { 'RawMatches.Result': { $in: ['1-0', '0-1', '1/2-1/2'] }, 'RawMatches.ECO': ?0 } }",
        "{ $group: {"
                + " _id: '$RawMatches.ECO',"
                + " totalMatches: { $sum: 1 },"
                + " whiteWins: { $sum: { $cond: [{ $eq: ['$RawMatches.Result', '1-0'] }, 1, 0] } },"
                + " draws: { $sum: { $cond: [{ $eq: ['$RawMatches.Result', '1/2-1/2'] }, 1, 0] } },"
                + " blackWins: { $sum: { $cond: [{ $eq: ['$RawMatches.Result', '0-1'] }, 1, 0] } }"
                + "} }",
        "{ $project: {"
                + " eco: '$_id',"
                + " numberOfMatches: '$totalMatches',"
                + " whiteWinPercentage: {"
                + "    $cond: { if: { $gt: ['$totalMatches', 0] },"
                + "             then: { $multiply: [{ $divide: ['$whiteWins', '$totalMatches'] }, 100] },"
                + "             else: 0 }"
                + " },"
                + " drawPercentage: {"
                + "    $cond: { if: { $gt: ['$totalMatches', 0] },"
                + "             then: { $multiply: [{ $divide: ['$draws', '$totalMatches'] }, 100] },"
                + "             else: 0 }"
                + " },"
                + " blackWinPercentage: {"
                + "    $cond: { if: { $gt: ['$totalMatches', 0] },"
                + "             then: { $multiply: [{ $divide: ['$blackWins', '$totalMatches'] }, 100] },"
                + "             else: 0 }"
                + " }"
                + "} }",
        "{ $sort: { 'eco': 1 } }"
})
TournamentsAnalyticsModel calculateRatesByOpening(String opening);
```

- **Average Number of Moves per Game by Edition and Category.**
  This query aggregates chess match data to compute the average number of moves per game for a given tournament edition, grouped by game category. The output provides an overview of the average number of moves per game in a given tournament edition, grouped by game category. The aggregation follows these steps:

  1. **Unwind**: expands the `RawMatches` array.
  2. **Match**: filters the matches to include only those from a specific edition given in the argument.
  3. **Project**: extracts relevant fields:
     - `edition`: the tournament edition.
     - `category`: the game category.
     - `moveCount`: the number of moves in each match, computed as the size of the `RawMatches.Moves` array. If no moves are recorded, it defaults to an empty array.
  4. **Group**: aggregates matches by `edition` and `category`, computing: `averageMoves`, the average number of moves per game.
  5. **Sort**: orders the results by `edition` and `category`.
  6. **Project**: renames fields for clarity

```
@Aggregation(pipeline = {  1 usage   jonnyfratta
        "{ $unwind: '$RawMatches' }",
        "{ $match: { 'Edition': ?0 } }",
        "{ $project: {"
                + " edition: '$Edition',"
                + " category: '$Category' ,"
                + " moveCount: { $size: { $ifNull: ['$RawMatches.Moves', []] } }"
                + "} }",
        "{ $group: {"
                + " _id: { edition: '$edition', category: '$category' },"
                + " averageMoves: { $avg: '$moveCount' }"
                + "} }",
        "{ $sort: { '_id.edition': 1, '_id.category': 1 } }",
        "{ $project: {"
                + " edition: '$_id.edition',"
                + " category: '$_id.category',"
                + " howMany: '$averageMoves',"
                + " _id: 0"
                + "} }"
})
List<TournamentsAnalyticsModel> calculateAverageMovesPerTournament(int edition);
```

- **Most Frequent Chess Opening by Edition and Category.**
  This query aggregates match data to determine the most frequently played opening for a given tournament edition, grouped by category. The output displays the opening's `ECO` code and the number of times it was played.
  The aggregation follows these steps:

  1. **Unwind**: expands the `RawMatches` array.
  2. **Match**: filters the matches to include only those that:
     - have a valid `ECO` code.
     - belong to a specific tournament edition specified in the argument.
  3. **Project**: extracts relevant fields: `edition`, `category`, `opening`.
  4. **Group**: aggregates matches by the previously extracted fields, computing: `count`, the number of times each opening was played.
  5. **Sort**: orders the results based on the count of the opening.
  6. **Group**: extracts the most frequently played opening for each `edition` and `category`:
     - `mostFrequentOpening`: The opening played the most times.
     - `maxCount`: The number of times it was played.
  7. **Sort**: Orders the results by `edition` and `category`.

```
@Aggregation(pipeline = {  1 usage  jonnyfratta
        "{ $unwind: '$RawMatches' }",
        "{ $match: { 'RawMatches.ECO': { $exists: true, $ne: '' }, 'Edition': ?0 } }",
        "{ $project: {"
                + " edition: '$Edition',"
                + " category: '$Category',"
                + " opening: '$RawMatches.ECO'"
                + "} }",
        "{ $group: {"
                + " _id: { edition: '$edition', category: '$category', opening: '$opening' },"
                + " count: { $sum: 1 }"
                + "} }",
        "{ $sort: { 'count': -1 } }",
        "{ $group: {"
                + " _id: { edition: '$_id.edition', category: '$_id.category' },"
                + " mostFrequentOpening: { $first: '$_id.opening' },"
                + " maxCount: { $first: '$count' }"
                + "} }",
        "{ $sort: { '_id.edition': 1, '_id.category': 1 } }",
        "{ $project: {"
                + " _id: 0,"
                + " edition: '$_id.edition',"
                + " category: '$_id.category',"
                + " eco: '$mostFrequentOpening',"
                + " howMany: '$maxCount'"
                + "} }"
})
List<TournamentsAnalyticsModel> findMostFrequentOpeningPerTournament(int edition);
```

- **Most Frequent Opening per User Based on Outcome.**
  This query aggregates chess match data to determine the most frequently played opening by a specific user, considering only games that resulted in a particular outcome (win, loss, or draw). The output displays the opening name and the number of times it was played under the given conditions.
  The aggregation follows these steps:

  1. **Match**: filters documents to include only those that belong to `Type:1`.
  2. **Unwind**: expands the `Matches` array.
  3. **Match**: further filters the matches to include only those where:
     - the user ID matches the provided parameter.
     - the match outcome matches the provided parameter.
  4. **Project**: extracts relevant fields: `userId`, `opening`.
  5. **Group**: aggregates matches by `userId` and `opening`, computing: `count`.
  6. **Sort**: sorts the results in descending order based on the number of times an opening was played.
  7. **Group**: extracts the most frequently played opening for each user:
     - `mostFrequentOpening`.
     - `howMany`: The number of times it was played.
  8. **Project**: Renames fields: `userId`, `mostFrequentOpening`, `howMany`.
  9. **Sort**: Orders the results by `userId`.

```
@Aggregation(pipeline = {  1 usage   jonnyfratta
        "{ $match: {'_id': ?0 ,'Type': '1'}}",
        "{ $unwind: '$Matches'}",
        "{ $project: { userId: '$_id', opening: '$Matches.Opening' } }",
        "{ $group: { _id: { userId: '$userId', opening: '$opening' }, count: { $sum: 1 } } }",
        "{ $sort: { count: -1 } }",
        "{ $group: { _id: '$_id.userId', mostFrequentOpening: { $first: '$_id.opening' }, howMany: { $first: '$count' } } }",
        "{ $project: { userId: '$_id', mostFrequentOpening: 1, howMany: 1 } }",
        "{ $sort: { 'userId': 1 } }"
})
UserAnalyticsModel findPlayerMostFrequentOpening(String username);
```

- **Number of Unique Opponents Defeated per User.**
  This query aggregates match data to determine how many unique opponents a specific user has defeated.
  The aggregation follows these steps:

  1. **Match**: filters documents to include only those that belong to match `Type:1`.

2. **Unwind**: expands the `Matches` array.
3. **Match**: further filters the matches to include only those where:
   - the user ID matches the provided parameter.
   - the match outcome is a win.
4. **Project**: extracts relevant fields `userId`, `opponentId`.
5. **Group**: aggregates matches by `userId` and `opponentId`, computing: `totalOpponents`, the number of times the user has defeated a specific opponent.
6. **Group**: aggregates by `userId`, computing: `howMany`, the number of unique opponents the user has defeated.
7. **Project**: renames fields: `userId`, `howMany`.
8. **Sort**: orders the results by `userId`.

```
@Aggregation(pipeline = {  1 usage  jonnyfratta
        "{ $match: { 'Type': '1' } }",
        "{ $unwind: '$Matches' }",
        "{ $match: { '_id': ?0, 'Matches.Outcome': 'win' } }",
        "{ $project: { userId: '$_id', opponentId: '$Matches.OpponentId'}}",
        "{ $group: { _id: { userId: '$userId', opponentId: '$opponentId'}, totalOpponents: { $sum: 1 } } }",
        "{ $group: { _id: '$_id.userId', howMany: { $sum: 1 } } }",
        "{ $project: { userId: '$_id', howMany: 1 } }",
        "{ $sort: { 'userId': 1 } }"
})
```

- **Counting Matches by Outcome for a Specific User.** This query aggregates chess match data to count the number of matches a specific user has played with a given outcome (win, loss, or draw).
  The aggregation follows these steps:

  1. **Match**: filters documents to include only those that belong to match `Type:1`.
  2. **Unwind**: expands the `Matches` array.
  3. **Match**: further filters the matches to include only those where:
     - The user ID matches the provided parameter.
     - The match outcome matches the specified value.
  4. **Group**: aggregates matches by `userId`, computing `howMany`, the total number of matches played with the specified outcome.
  5. **Project**: Renames fields: `userId`, `howMany`.
  6. **Sort**: Orders the results by `userId`.

```
@Aggregation(pipeline = {  1 usage  jonnyfratta
        "{ $match: { 'Type': '1' } }",
        "{ $unwind: '$Matches' }",
        "{ $match: { '_id': ?1, 'Matches.Outcome': ?0 } }",
        "{ $group: { _id: '$_id', howMany: { $sum: 1 } } }",
        "{ $project: { userId: '$_id', howMany: 1 } }",
        "{ $sort: { 'userId': 1 } }"
})
UserAnalyticsModel playerNumberOfWonMatches(String outcome, String username);
```

5.1.2 Redis

The queries that work on the key-value database are the following:

```
private static final String LIVE_MATCHES_KEY = "Live:matches";
private static final String DISQUALIFIED_PLAYERS_KEY = "disqualified_players";  4 usages
private static final String REGISTERED_PLAYERS_KEY = "registered_players";  5 usages
```

- **Retrieve the list of all the matches of the current day.**
  This method retrieves a list of live matches stored in a Redis cluster using Jedis. The Redis set is identified by the key `Live:matches`. The function returns all elements in this set as a list of strings.

```
public List<String> getLiveMatches() {  2 usages  jonnyfratta +1
    return new ArrayList<>(jedisCluster.smembers(LIVE_MATCHES_KEY));
}
```

- **Retrieve the list of all disqualified players not notified yet.**
  This method retrieves a list of all the recently disqualified players, stored in a Redis cluster using Jedis. The Redis set is identified by the key `disqualified_players`. The function returns all elements in a set of strings.

```
public Set<String> getAllDisqualifiedPlayers() {  2 usages  jonnyfratta
    return jedisCluster.smembers(DISQUALIFIED_PLAYERS_KEY);
}
```

- **Retrieve the list of enrolled players not notified yet.**
  This method retrieves a list of all the recently enrolled players, stored in a Redis cluster using Jedis. The Redis set is identified by the key `registered_players`. The function returns all elements in this set as a map of strings.

```
public Map<String, String> getAllRegisteredPlayers() {  2 usages
    return jedisCluster.hgetAll(REGISTERED_PLAYERS_KEY);
}
```

5.2 Indexes Evaluation

This section evaluates the performance of various MongoDB indexes by comparing execution times, documents examined, and keys examined across different queries. The goal is to determine the most efficient indexing strategy for optimizing query performance.

5.2.1 Tournaments Collection Indexes

Average Moves per Winner in a given Edition and Category

Index Used	Execution Time (ms)	Keys Examined	Docs Examined
No Indexes	8	0	144
{ "Edition": 1, "Category": 1 }	5	1	1
{ "Edition": 1, "Category": 1, "Location": 1 }	**4**	**1**	**1**
{ "RawMatches.Winner": 1 }	6	0	144

Table 5.1: Performance Comparison for Query 1

As it's possible to see, the index { "Edition": 1, "Category": 1, "Location": 1 } is the most efficient for this query. Actually the { "Edition": 1, "Category": 1 } index is also almost equally efficient.

Number of Victories per Player in a given Edition and Category

Index Used	Execution Time (ms)	Keys Examined	Docs Examined
No Indexes	7	0	144
{ "Edition": 1, "Category": 1 }	6	1	1
{ "Edition": 1, "Category": 1, "Location": 1 }	**3**	**1**	**1**
{ "RawMatches.Winner": 1 }	9	0	144

Table 5.2: Performance Comparison for Query 2

The index { "Edition": 1, "Category": 1, "Location": 1 } is the most efficient for this query, however the { "Edition": 1, "Category": 1 } index is also almost equally efficient.

Match Statistics for each Opening (ECO)

Index Used	Execution Time (ms)	Keys Examined	Docs Examined
No Indexes	353	0	144
{ "Edition": 1, "Category": 1 }	369	0	144
{ "Edition": 1, "Category": 1, "Location": 1 }	370	0	144
{ "RawMatches:ECO": 1 }	461	0	144
{ "RawMatches.Result": 1 }	**269**	**0**	**144**

Table 5.3: Performance Comparison for Query 3

The index { "RawMatches.Result": 1 } is the most efficient for this query, as a matter of fact even though it doesn't affect the number of examined documents, it returns a result in almost 100ms less then the version without indexes.

Match Statistics for a Specific Opening

Index Used	Execution Time (ms)	Keys Examined	Docs Examined
No Indexes	**231**	**0**	**144**
{ "Edition": 1, "Category": 1 }	246	0	144
{ "Edition": 1, "Category": 1, "Location": 1 }	249	0	144
{ "RawMatches.ECO": 1 }	247	0	144

Table 5.4: Performance Comparison for Query 3VAR

In this query the indexes don't seem to have a significant impact on the performance, as the execution time is almost the same for all the cases.

Retrieve All Matches of a Player

Index Used	Execution Time (ms)	Keys Examined	Docs Examined
No Indexes	287	0	144
{ "Edition": 1, "Category": 1 }	269	0	144
{ "Edition": 1, "Category": 1, "Location": 1 }	222	0	144
{ "RawMatches.White": 1, "RawMatches.Black": 1 }	**192**	**0**	**144**

Table 5.5: Performance Comparison for Query 10

The index { "RawMatches.White": 1, "RawMatches.Black": 1 } is the most efficient for this query, as it returns the result in almost 100ms less than the version without indexes.

Overall Conclusions

- { "_id": 1} (default index implemented by MongoDB)
- { "Edition": 1, "Category": 1, "Location": 1 }
- { "RawMatches.Result": 1 }
- { "RawMatches.White": 1, "RawMatches.Black": 1 }

5.2.2 User Collection Indexes

Most Frequent Opening for Each User

Index Used	Execution Time (ms)	Keys Examined	Docs Examined
No Indexes	327	0	155
{"Type": 1}	**280**	**101**	**101**

Table 5.6: Performance Comparison for Query 1

The index { "Type": 1 } is the most efficient for this query, reducing execution time by nearly 50ms compared to running without indexes and reducing the number of examined documents.

Most Frequent Opening When User Wins

Index Used	Execution Time (ms)	Keys Examined	Docs Examined
No Indexes	174	0	155
{ "Type": 1 }	**185**	**101**	**101**

Table 5.7: Performance Comparison for Query 2

Surprisingly, the indexed version took slightly longer. The index { "Type": 1 } does not provide a significant advantage in this case.

Most Frequent Opening When a Specific User Wins

Index Used	Execution Time (ms)	Keys Examined	Docs Examined
No Indexes	3	1	1
{ "Type": 1 }	3	1	1

Table 5.8: Performance Comparison for Query 2VAR

Using the index does not impact performance for a specific user.

Count of Unique Opponents When a User Wins

Index Used	Execution Time (ms)	Keys Examined	Docs Examined
No Indexes	186	0	155
{ "Type": 1 }	**199**	**101**	**101**

Table 5.9: Performance Comparison for Query 3

The { "Type": 1 } index results in slightly higher execution time, however the number of analyzed documents is inferior when using the index.

Count of Unique Opponents for a Specific User When Winning

Index Used	Execution Time (ms)	Keys Examined	Docs Examined
No Indexes	13	1	1
{ "Type": 1 }	**5**	**1**	**1**

Table 5.10: Performance Comparison for Query 3VAR

For a specific user, using the { "Type": 1 } index results in an improvement in the execution time.

Total Wins per User

Index Used	Execution Time (ms)	Keys Examined	Docs Examined
No Indexes	170	0	155
{ "Type": 1 }	**177**	**101**	**101**

Table 5.11: Performance Comparison for Query 4

The performance difference is small, however the number of examined documents is inferior when using the index.

Sparse Index on "status" Field

The query that allows a user to visualize the list of disqualified players, checks if the field status is set to 1. However, since the status field does not exist in the user collection and is only created and set to 1 when a player is disqualified, the sparse index {"Status": 1} allows the query to only navigate through the list of disqualified players instead of all of them.

Overall Conclusions

- { "_id": 1} (default index implemented by MongoDB)
- { "status": 1 } (sparse)
- { "Type": 1 } (partial). This index might seem not that much convenient given the values read in the previous tables, however we have to consider that right now the DB is filled with 101 players, 50 spectators and 4 managers. In a real life application of course the number of spectators would be much greater than the number of players and in that case, limiting the accesses of the query to just Type:1 users would significantly increase its performance (imagine the case with 1000 users but only 100 players).