Database Systems

BLG 317E

Project Report

Kanan Bandaliyev
150210909
bandaliyev21@itu.edu.tr
Ramazan Taş
tasr21@itu.edu.tr
Dana Al Jasem
150210922
jasem21@itu.edu.tr
Ragib Guliyev
150210920
guliyev21@itu.edu.tr
Abdullah Jafar Manosur Shamout
150200919
shamout21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 3.1.2024

CONTENTS

1. Introduction

In the following report, we provide vital details about our project. We start by listing the responsibilities of each member and briefly explaining our dataset. Then we explain how our website can be navigated, and how the functionalities can be utilized. We then discuss the technologies used, like the database management system, programming language, frontend, and docker. Moreover, We briefly explain the APIs and then dive into the details of our backend implementations. Finally, we discuss the challenges we have faced throughout the implementation phases of our project.

1.1. Responsibilities

- Kenan: Tournaments and Awards Page
- Dana: Matches and Match ScoreBoard pages. Matches, Goals, and Bookings Tables.
- Ramazan: Groups and Standings, Groupstandings and Managers Page
- Abdullah: Players, Player Appearances, and Squads
- : Ragib: Teams and Team Stats

2. Dataset

Our Database is a comprehensive database about the FIFA World Cup that covers all 22 men's tournaments (1930-2022) and all 8 women's tournaments (1991-2019). The database includes 27 datasets (over 1.58 million data points) that cover all aspects of the World Cup. We didn't use all tables from our database.

This is the ER Diagram of our database:

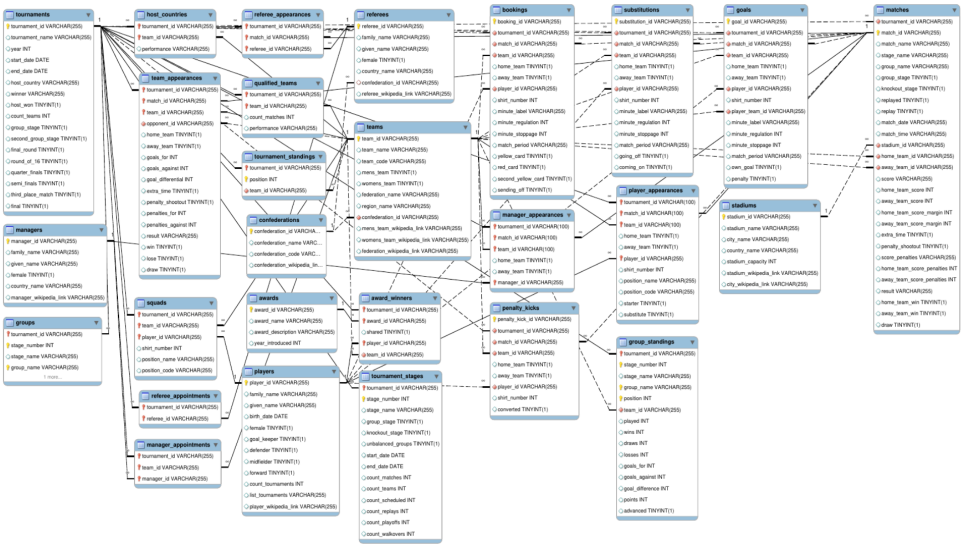

Figure 2.1: ER Diagram

3. Navigation

3.1. Tournaments

The tournaments page facilitates the adddition, deletion and edit functionalities of the database. The frontend for each of the given operation is given in the following figures:

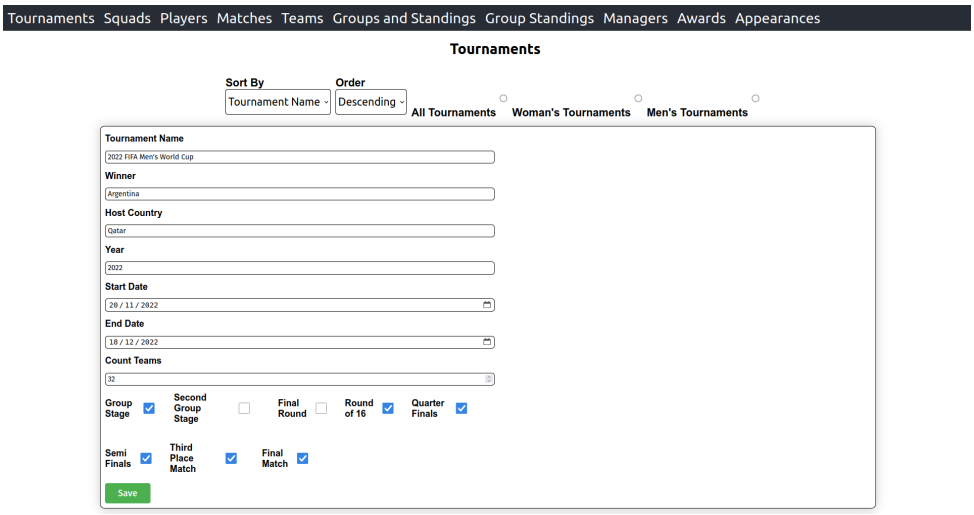

Figure 3.1: Edit operation on Tournament

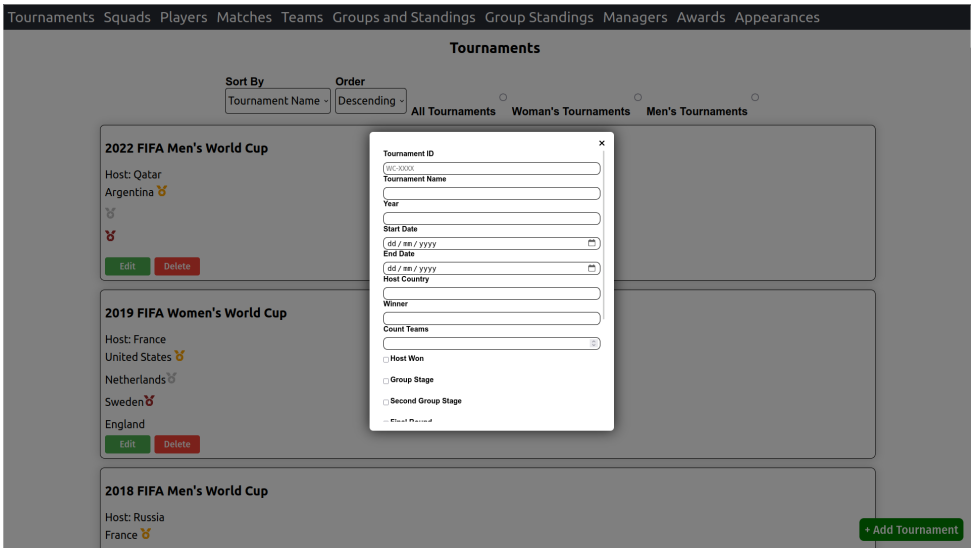

Figure 3.2: Create operation on Tournaments

We can get to the tournament details by clicking on the tournament title. The tournament details for the 2022 FIFA Men's World Cup is shown as below:

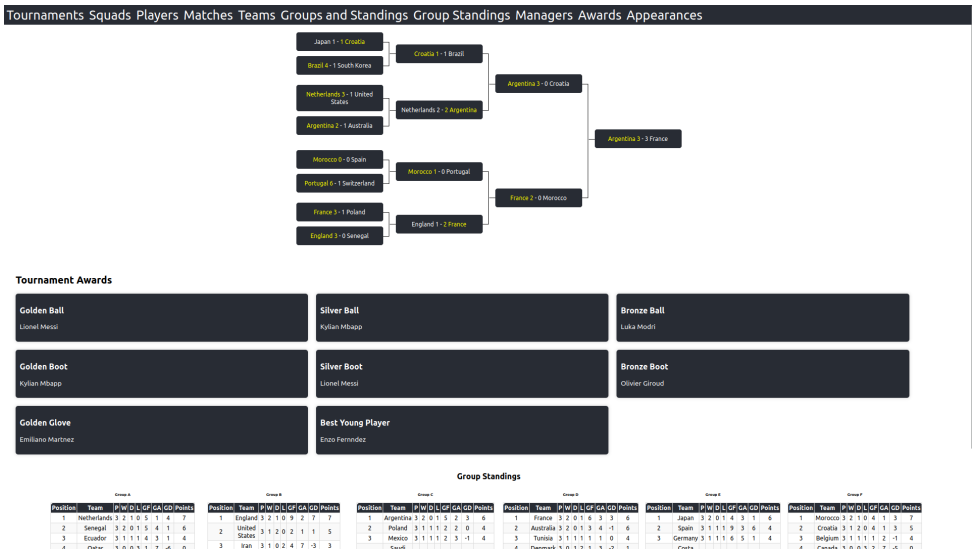

Figure 3.3: Tournament Details for the 2022 Men's World Cup

As we can see this page displays the brackets for the knockout stage of the tournament. In the brackets the winner of the matches are marked with yellow. After the brackets we display the awards for the respective tournament which uses the APIs from the awards page. Finally we can see the group standings and the qualified teams in the tables below for each group. The matches in the brackets are clickable which redirects the page to the match scoreboard for that specific match. The players in the awards are also clickable which redirects to the player details page.

3.2. Squads

The squads page facilitates the addition, deletion and edit functionalities of the squads table. The frontend for each of the given operations is given in the following figures:

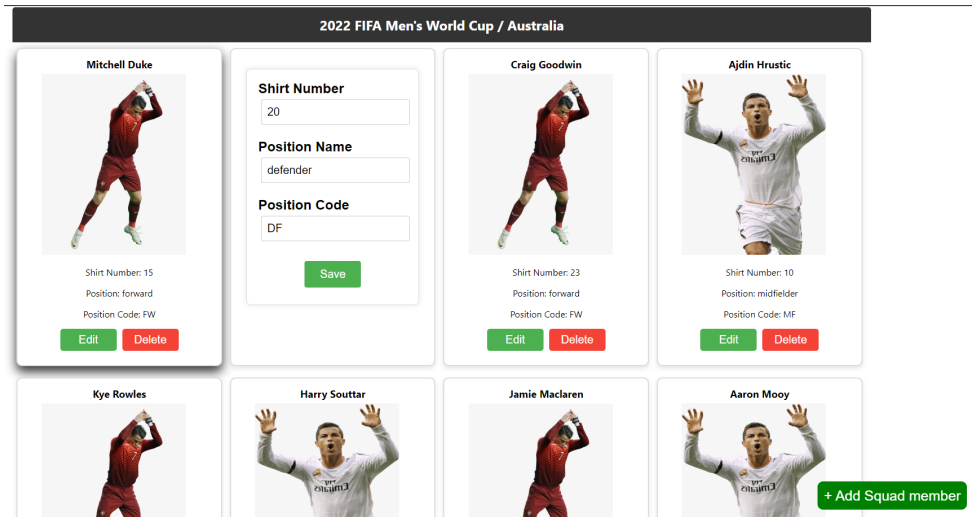

Figure 3.4: edit operation on squads

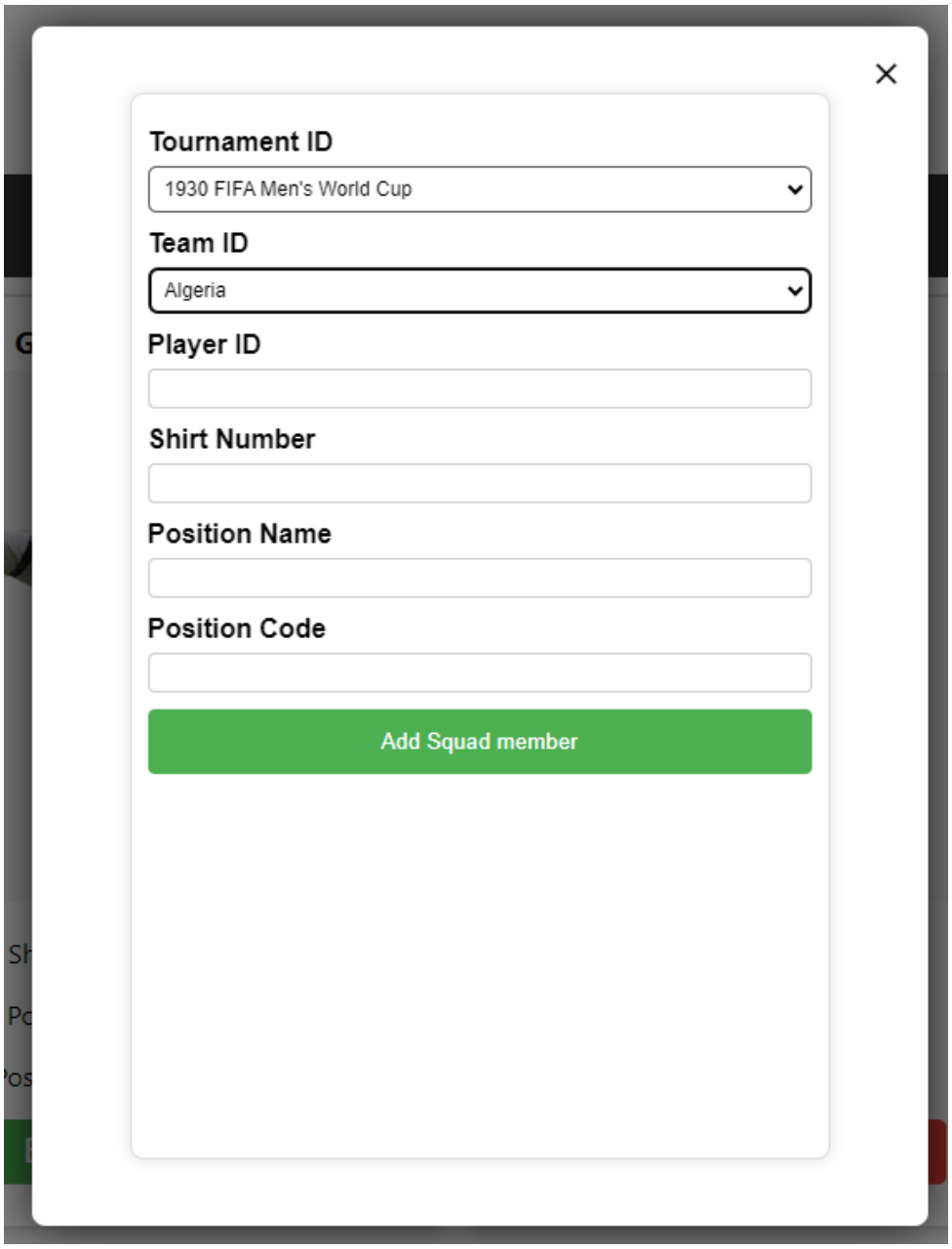

Figure 3.5: insert operation on squads

By clicking on the <tournament name> / <team name> Header template for any of the squads in the main page, we go to a page only for that single squad as shown in the figure below.

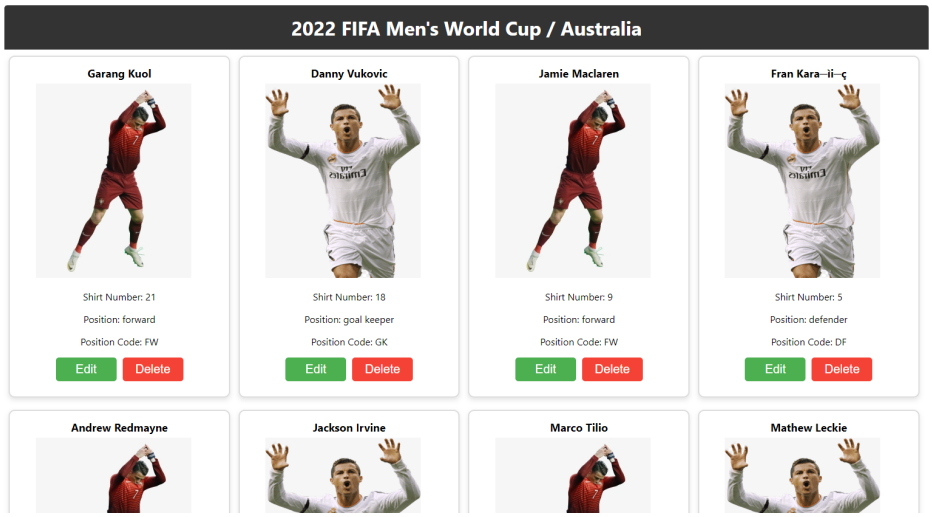

Figure 3.6: single squad page

Also by clicking on any of the player names, we can go to that player details page as shown by the figure below

Figure 3.7: player details

The page squads page is scrollable fetching more new data as you go down.

3.3. Players

The Players page facilitates the addition, deletion and edit functionalities of the players table. The page is paginated, also on this page there are extra capabilities like the ability to search a single player, filter by an attribute and sort ascendingly or descendingly by a header. The frontend for each of the given operations is given in the following figures:

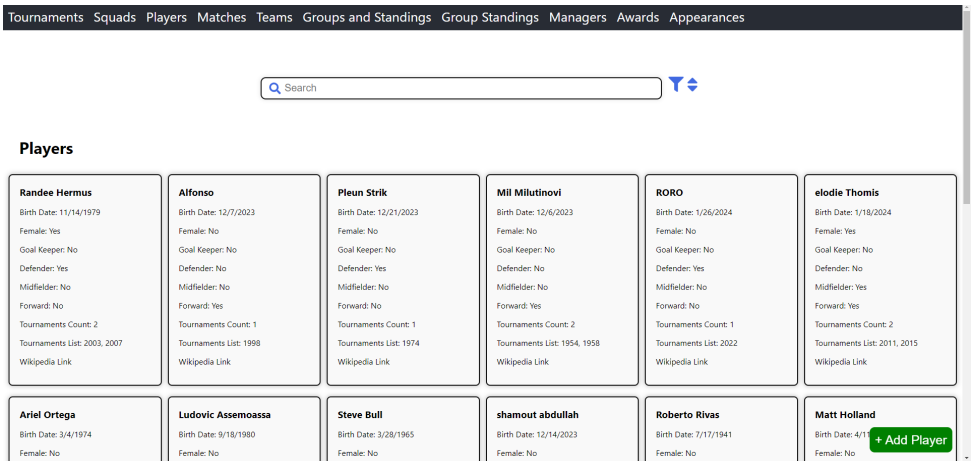

Figure 3.8: player page

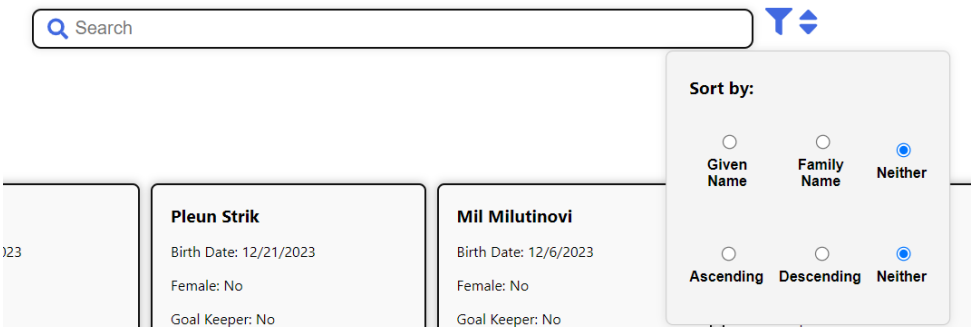

Figure 3.9: players sorting capabilities

Figure 3.10: players filters

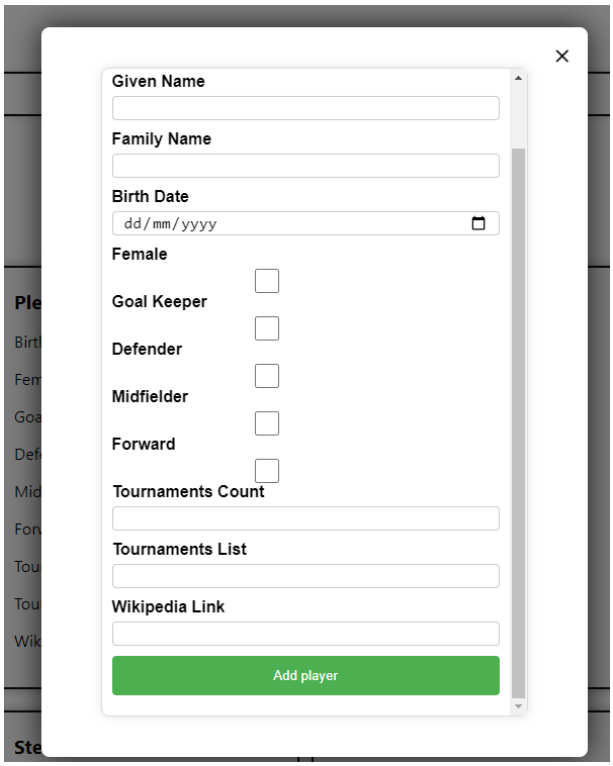

Figure 3.11: player insert

As for deleting and editing a player, that is done from the player's details page. as shown above in the squads section.

The editing makes the page look as follows:

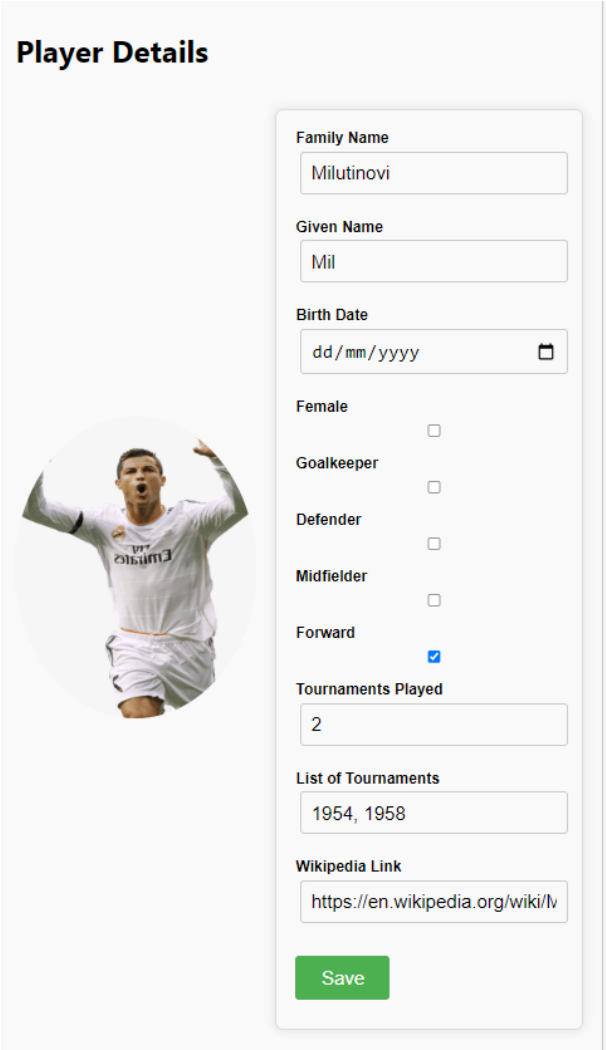

Figure 3.12: player details editing

3.4. Matches

This page facilitates the addition, deletion, and updating functionalities of matches table. Morevover, this page supports certain functionalities like filtering and sorting and is linked to several relevant pages.

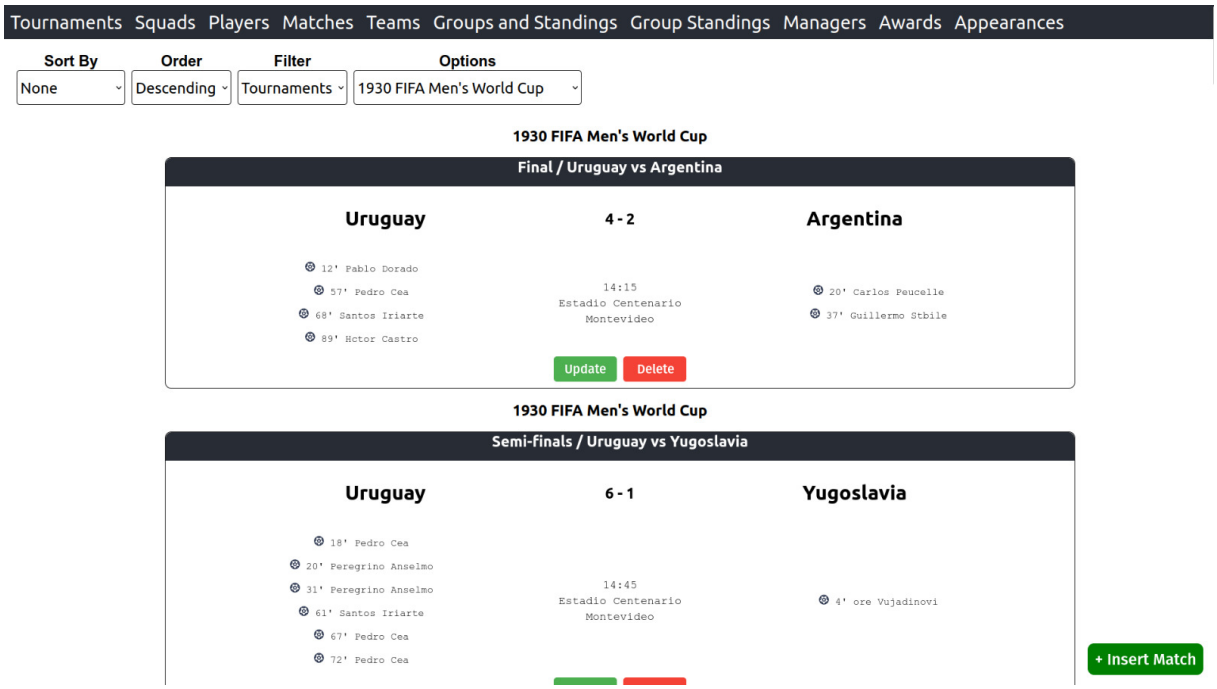

Figure 3.13: Matches Page

As can be seen in the figure above, the matches page displays all matches for a specific tournament or team. It includes most of the attributes in the goals and matches tables like the names of the teams, scores, match names, match stage, stadium, time, city, players' names, and so on. In addition, It has buttons for the update, delete, and insert operations.

If the filter is changed to filter by team, for example, Bolivia, the view will be as shown in the figure below.

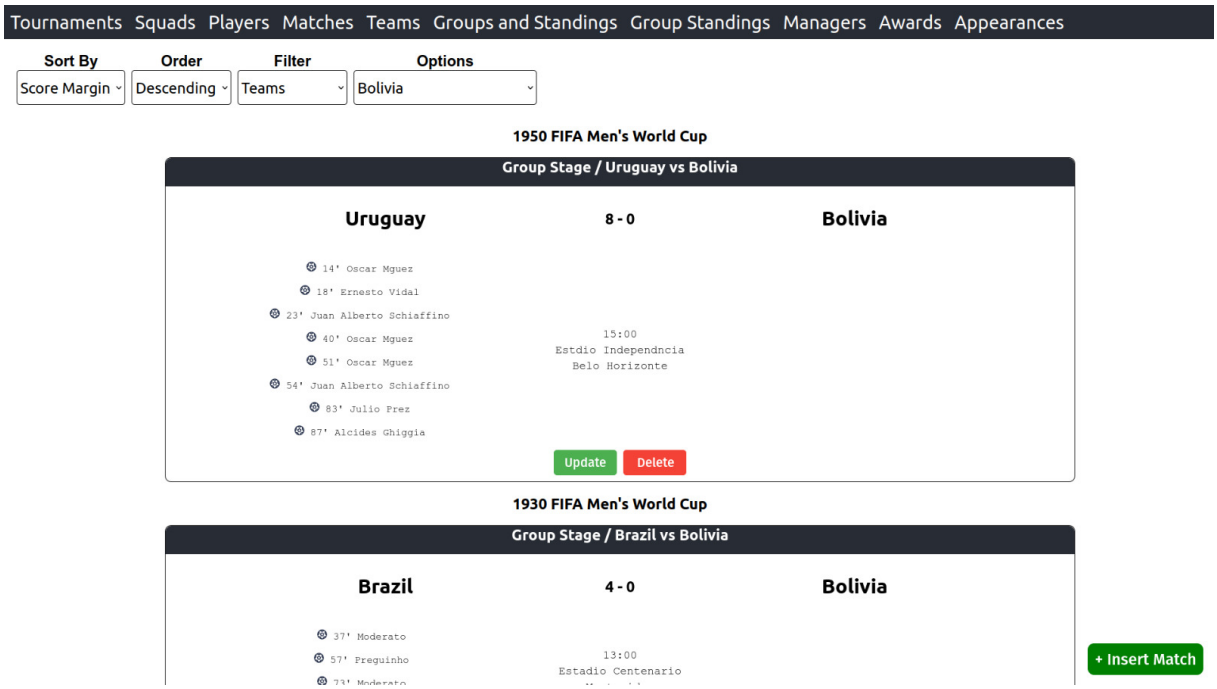

Figure 3.14: Filtering by Team

Here you can also see that sorting was done by score margin in descending order. The first match has a score margin of 8 and the second of 4.

If the update button is pressed, the following form will pop up in place of the match as shown in the figure below.

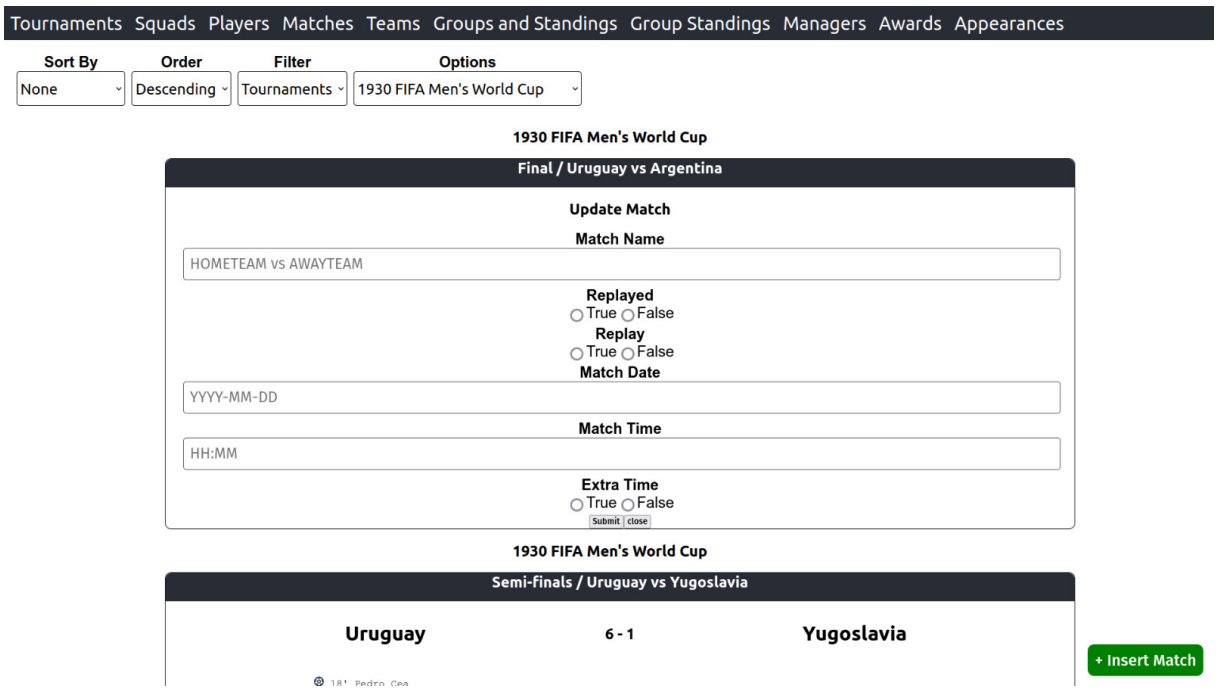

Figure 3.15: Update Form

Only certain attributes can be modified as the remaining are either foreign keys or things like the score of the game which if modified can change the logic of the whole

tournament and effect my teammate's pages. Hence, we agreed that allowing the editing of all fields is not reasonable for this table.

If the insert button is pressed, the form shown in the image below will pop up.

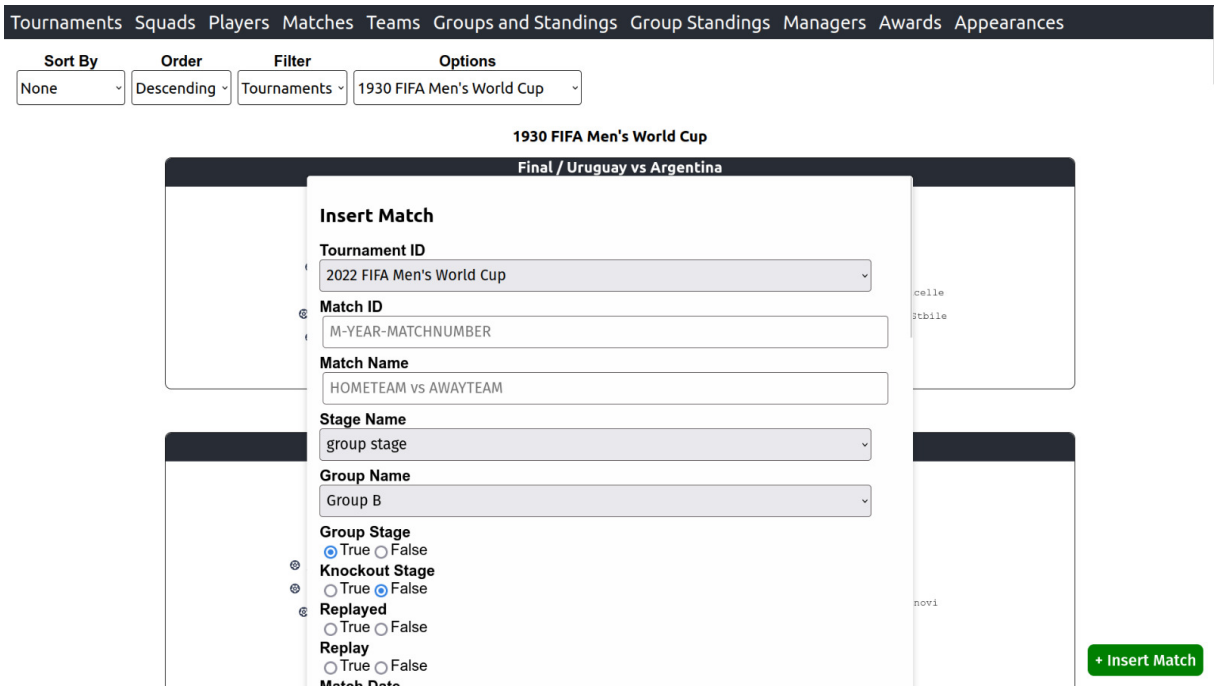

Figure 3.16: Insert Form

In this form, the selects are fetched according to your choices in previous fields. For example here, when you choose the tournament, the stages will be fetched according to the stages that were played in that tournament. Also, when you select a stage, if its a group stage the group names are fetched, if not, it'll be not applicable. Most of the true and false fields are also automatically checked according to the fields you have set before so the data of each tuple makes sense.

Finally, this page is linked to several others. Pressing on a team's name takes you to that team's squad. Pressing on a player takes you to the player's details page. Pressing on the match's name takes you to the match scoreboard page which is a different view of the match with more details as will be explained in the next section.

3.5. Match ScoreBoard

This page is a different more detailed view of the match and can be accessed by pressing on the match name in the Matches page. The view is shown below.

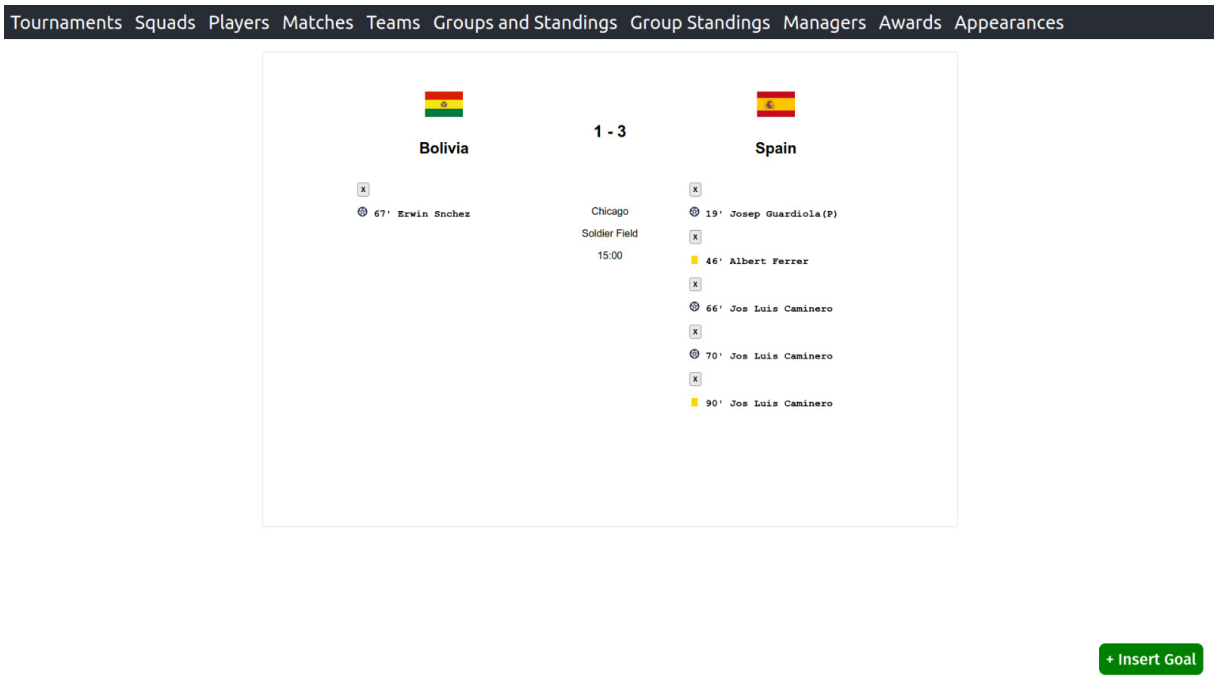

Figure 3.17: Match ScoreBoard

In this page, you can delete a goal by pressing the x button and you can insert a goal by pressing on the insert goal button. If the insert goal button is pressed the form shown in the image below appears.

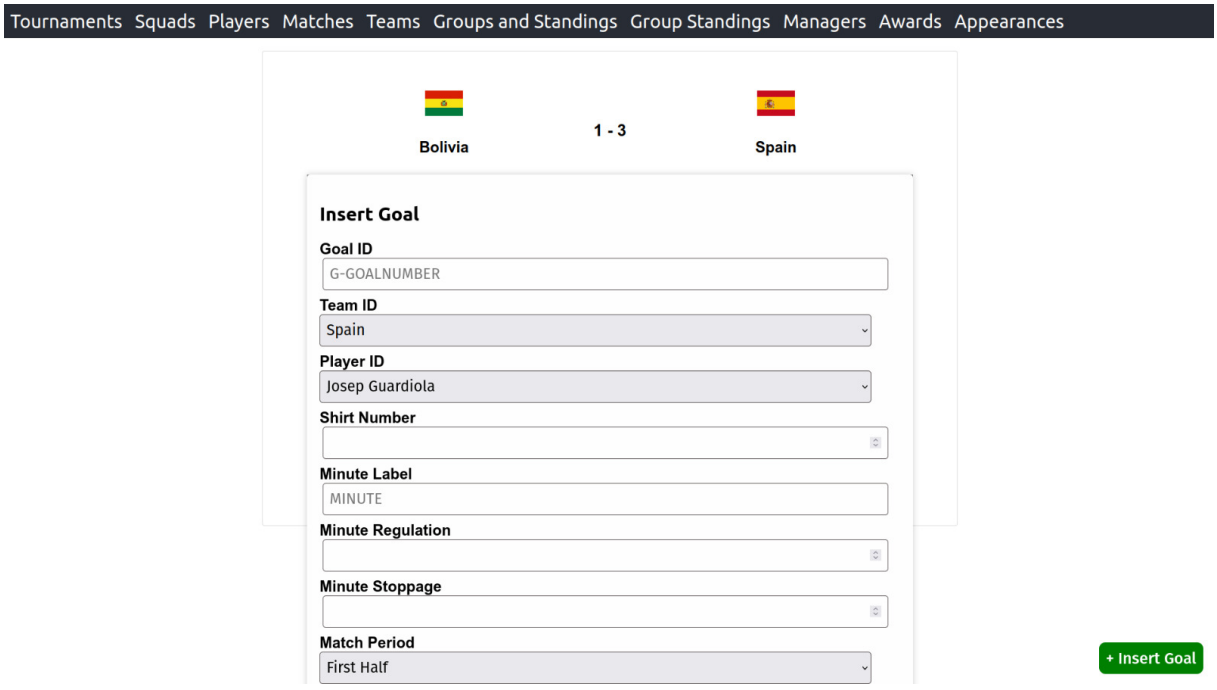

Figure 3.18: Goal Insert Form

Here also validation is applied. The fields of the player ID select bar is fetched after the team is chosen so you can only select a player that actually played in this match.

Also, you can only select one of the two teams in this match. Other fields have pattern and logic validations.

3.6. Groupstandings

For groupstandings there is 2 page. First one is Groups and Standings, second one is Groupstandings. In Group and Standings page it shows groups and groupstandings as ta tables. Create, delete and Update Operations in Groupstandings page.

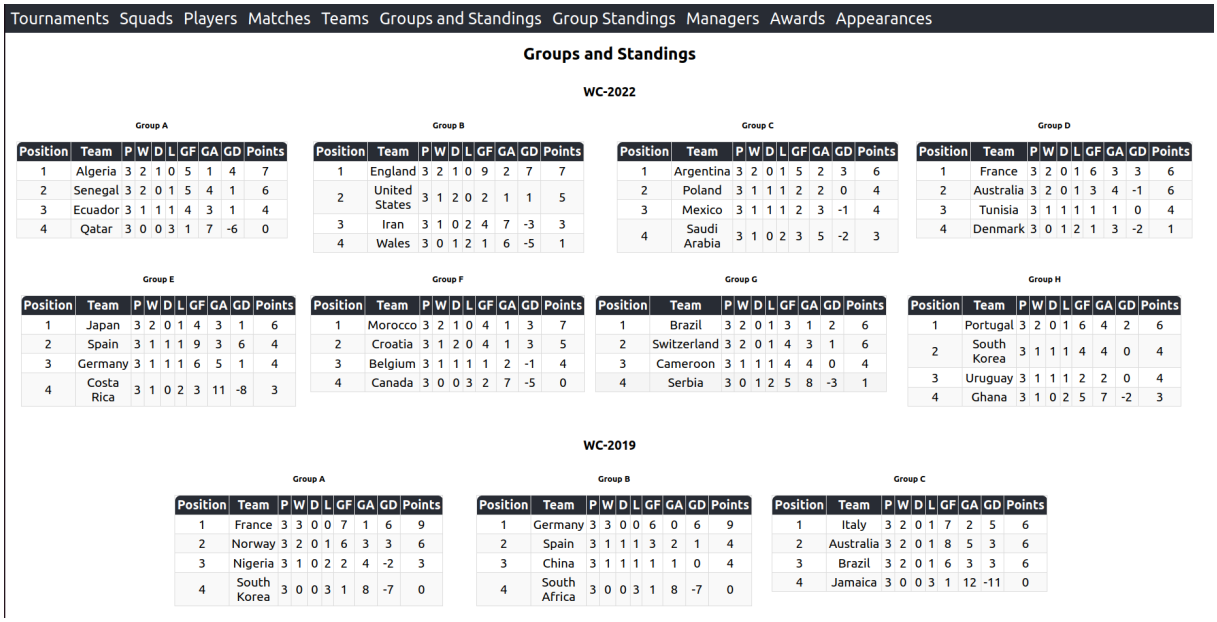

Figure 3.19: Groups And Standings Page

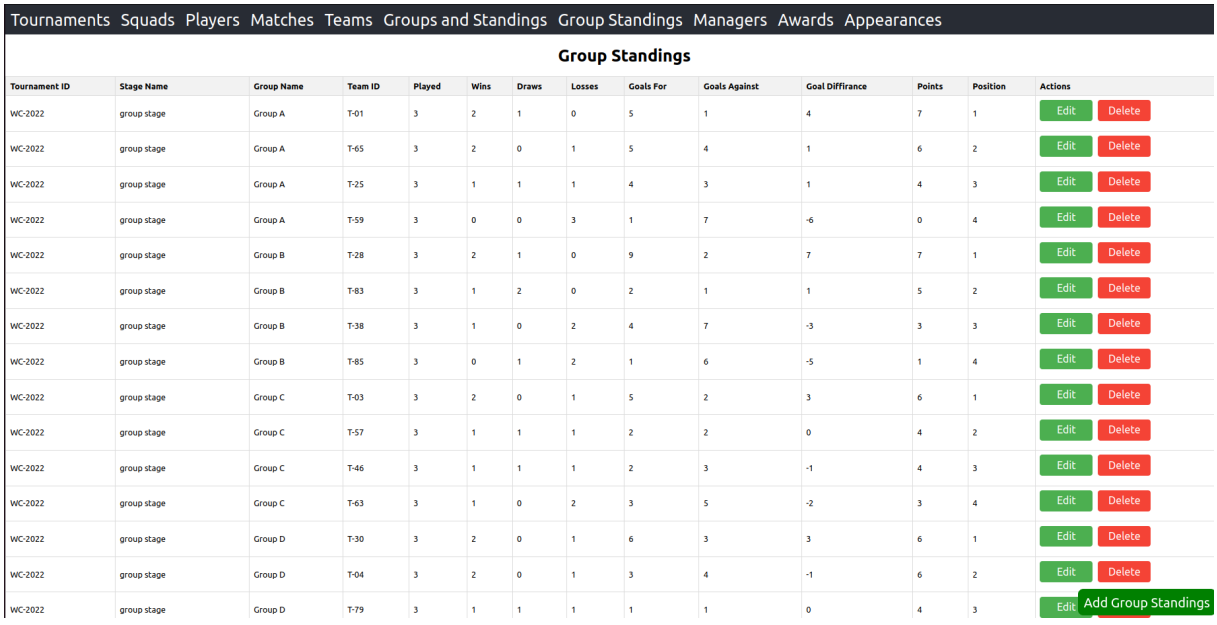

Figure 3.20: Groupstandings Page

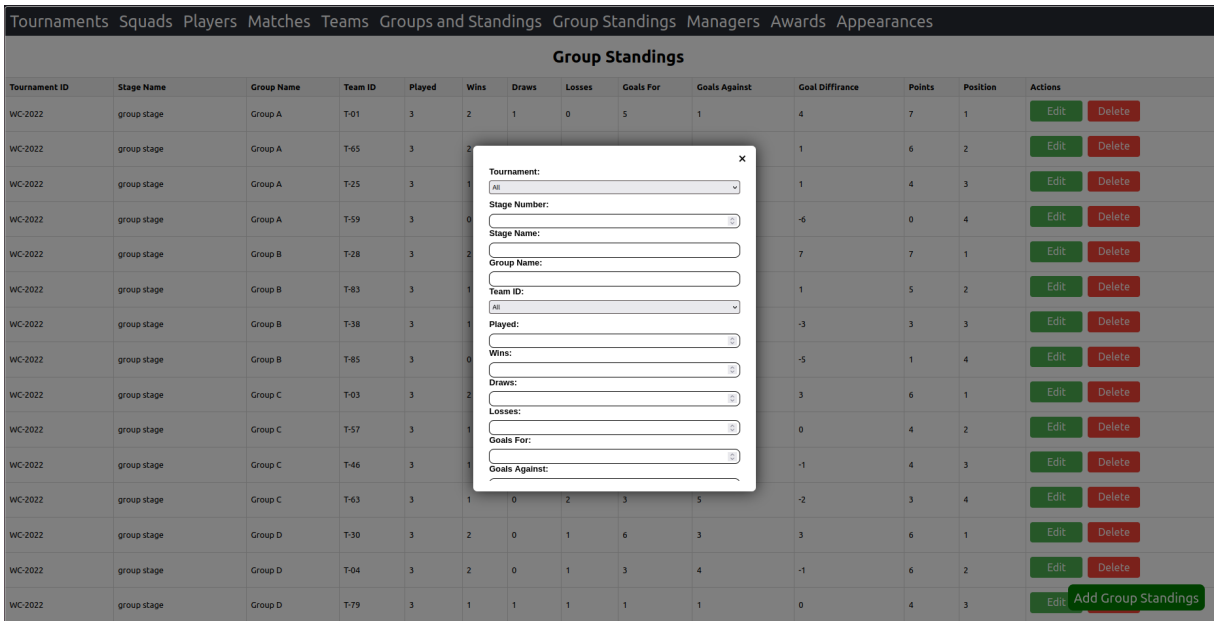

Figure 3.21: Add Operation

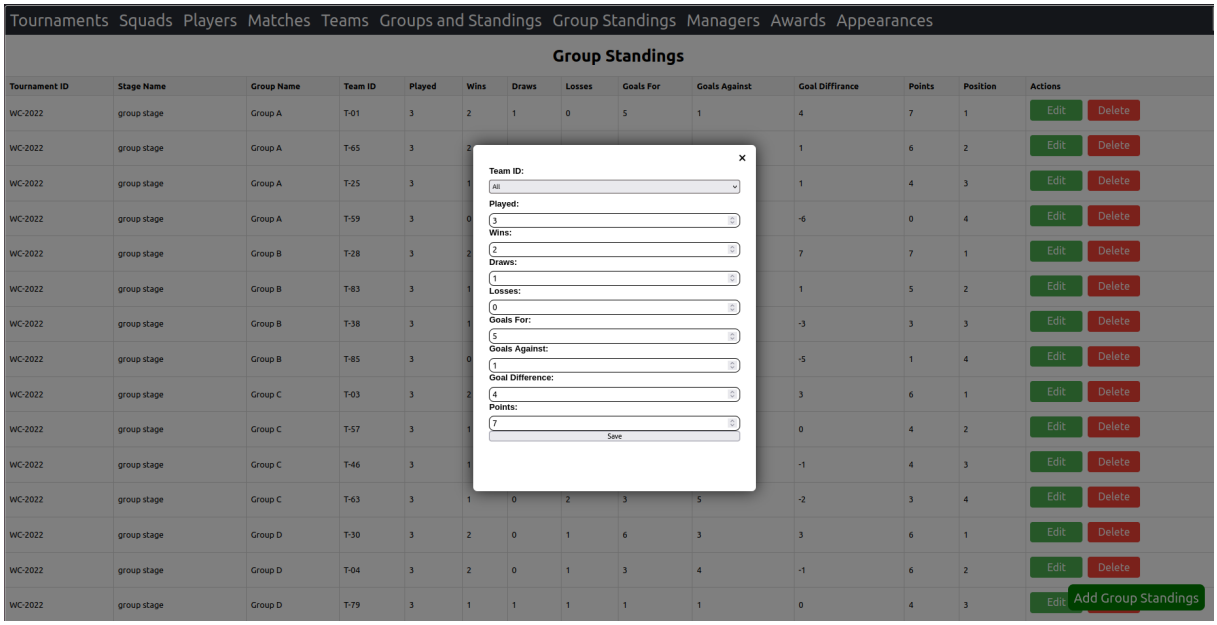

Figure 3.22: Edit Operation

3.7. Managers

The managers page lists all the managers and there are edit and delete buttons for all of them. If you click on the Wikipedia link text in the manager boxes, it directs you to the Wikipedia page of the manager. The photos of the add and edit sections:

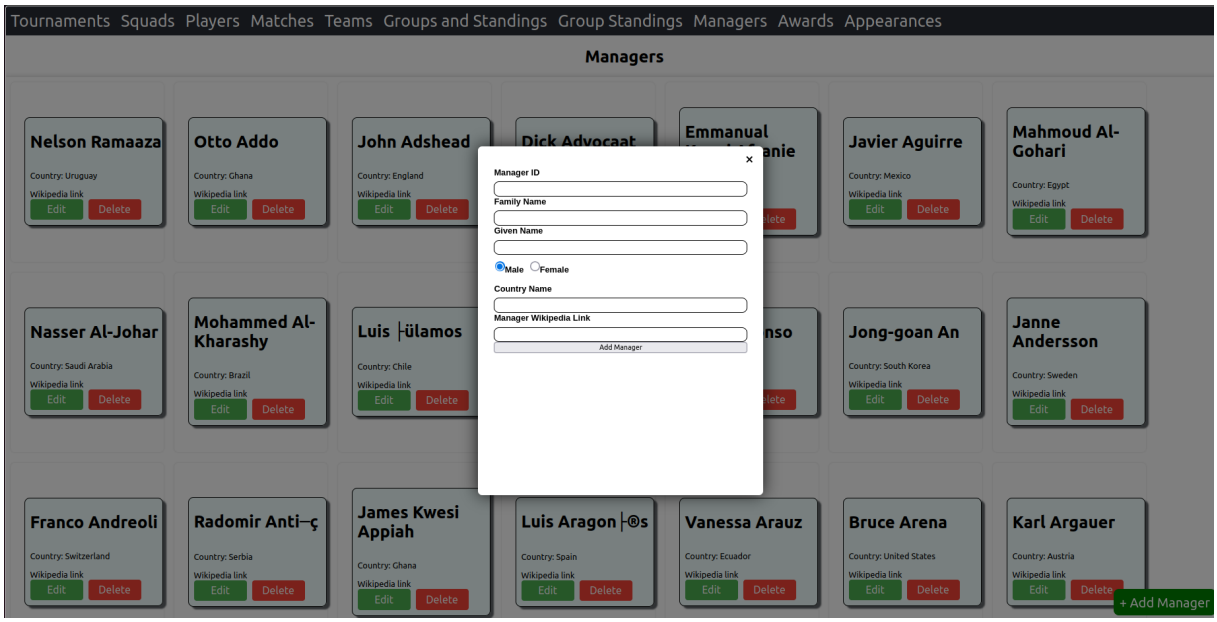

Figure 3.23: Add Operation for Managers

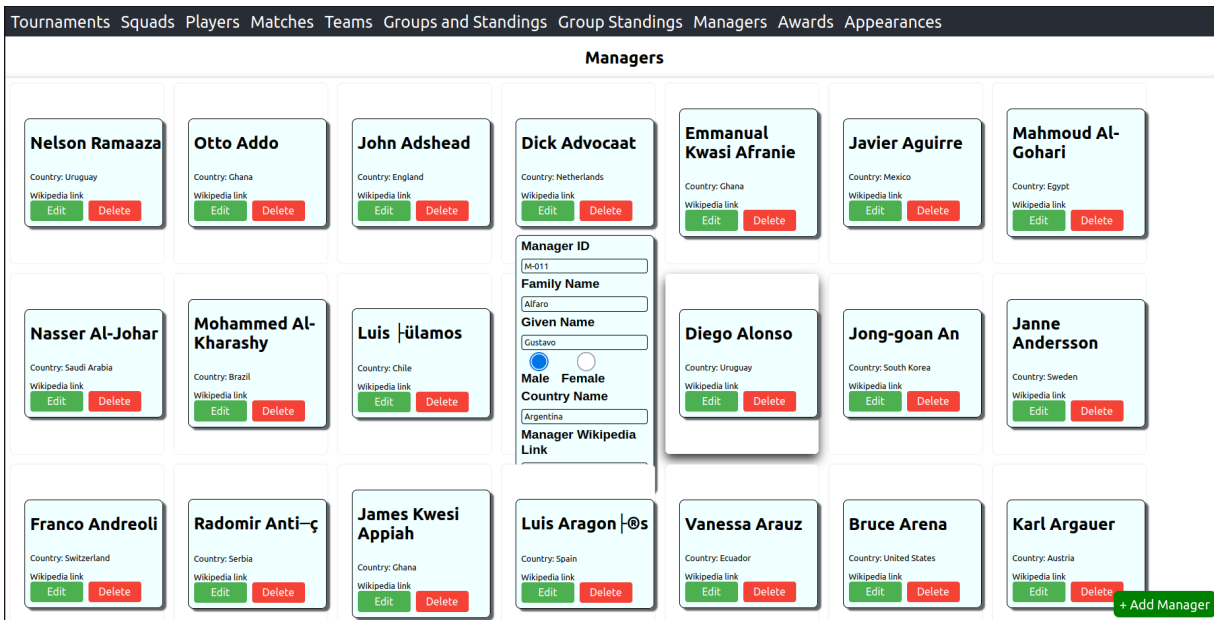

Figure 3.24: Edit Operation for Managers

3.8. Teams

Upon accessing the "Teams" page, users are greeted with a grid layout displaying cards for different football teams. Each card includes the team's national flag, the country code, and a label indicating whether it is a Men's or Women's National Team. Below each team's details, users have the options to 'Edit' or 'Delete' the corresponding team's information or add a completely new team using 'Add Team' button, aforementioned features offer straightforward data management directly from the main page.

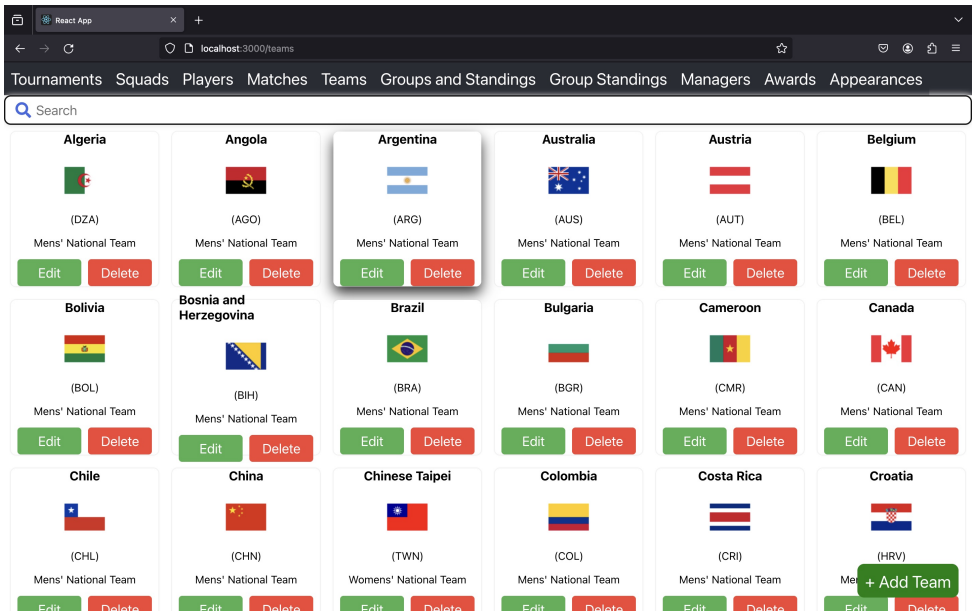

Figure 3.25: Teams page

At the top of the page, there is an interactive search bar that enables users to quickly find a specific team by typing in keywords. This search functionality enhances the user experience by making the retrieval of data both fast and intuitive.

When a user selects (press on any team) a particular team, the interface redirects to a "Team Stats" page. This page provides detailed statistics about the team's performance in various tournaments. To aid in analyzing the team's history and performance, there is a filter dropdown that allows users to narrow down the displayed statistics to specific tournaments in which the team has participated.

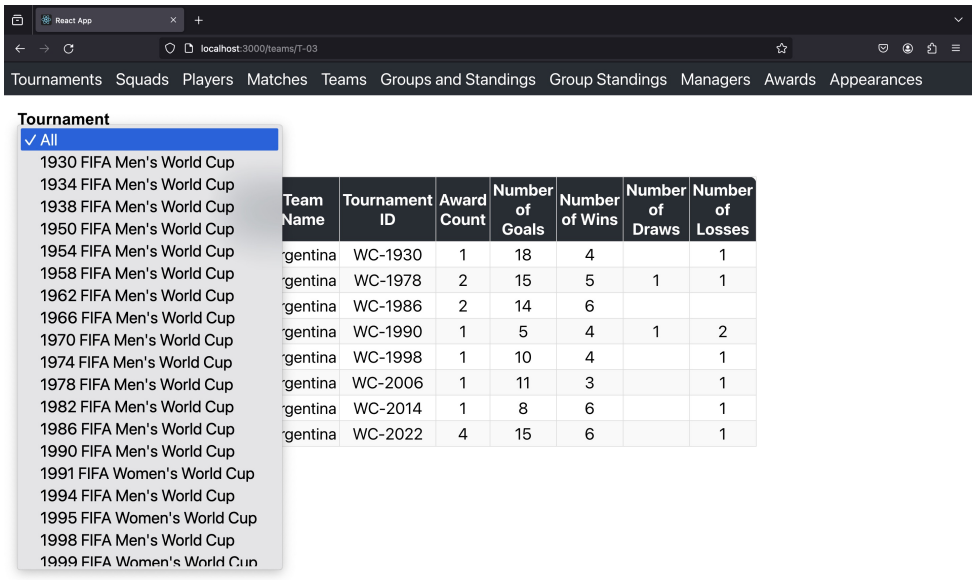

Figure 3.26: Team Stats page

Additionally, each team card on the "Teams" page includes a link to the team's Wikipedia page. Clicking on "Men's National Team" or "Women's National Team" takes the user directly to the respective Wikipedia page, providing in-depth information about the team's history, achievements, and other relevant details.

Figure 3.27: Redirection from "Teams" page to Wikipedia

3.9. Awards

Awards page displays the awards for the respective tournaments with player names. It has sorting, filtering, and searching capabilities. The overall structure of the page is given in the figure below:

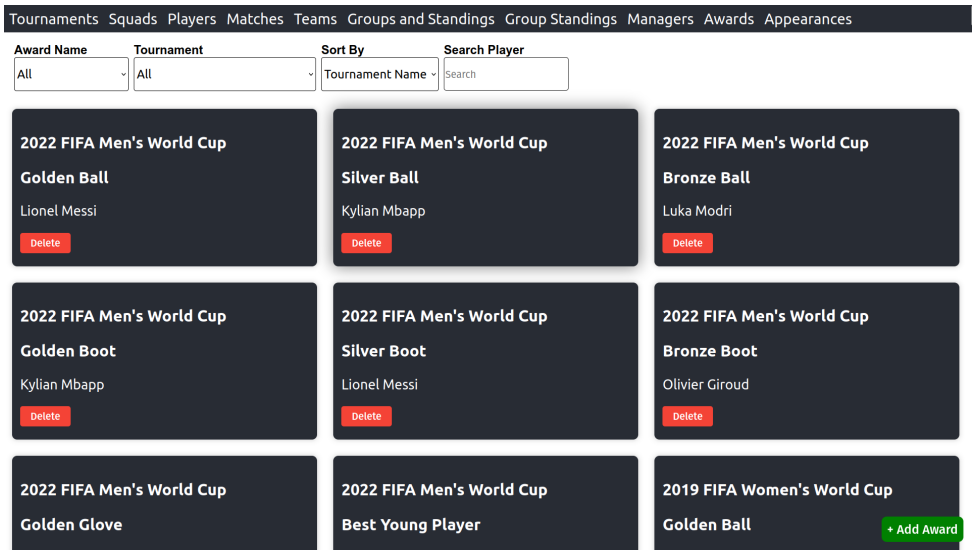

Figure 3.28: Awards page overview

We can add a new award winner using the add award button in the right bottom part of the page. Here is the modal that is displayed when clicked on that button:

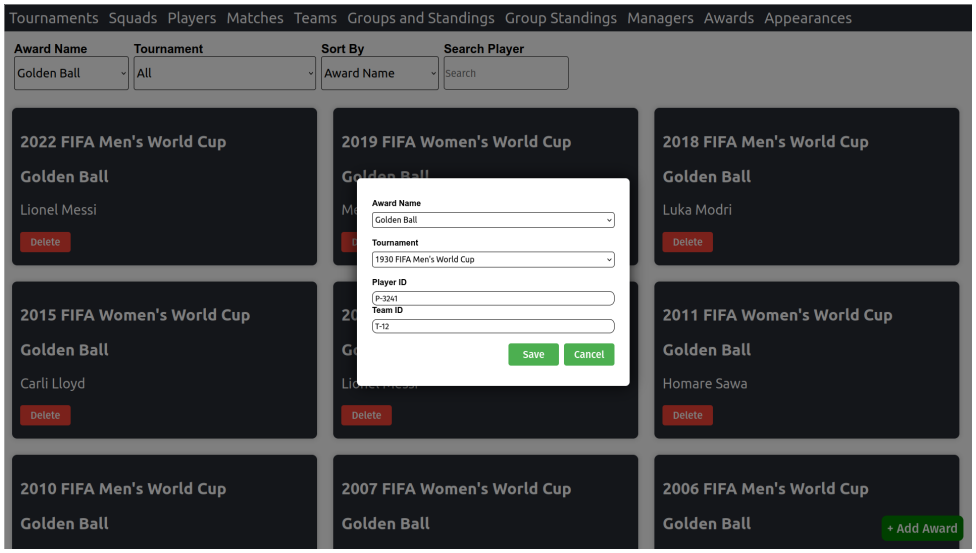

Figure 3.29: Caption

We can also sort the view of the awards using the selection given at the top. Here are all the Golden Boot owners for all tournaments:

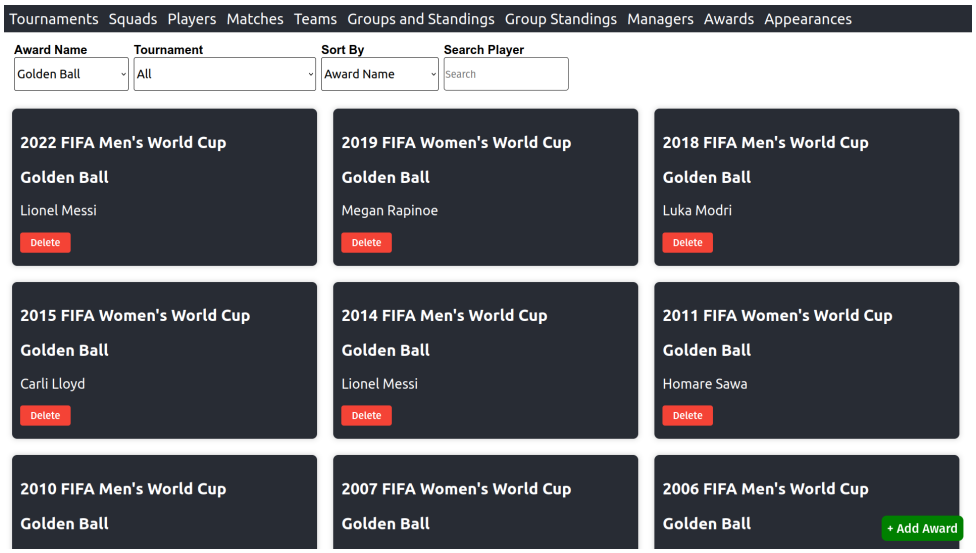

Figure 3.30: Caption

3.10. Player Appearances

This page facilitates the addition, deletion, and updating functionalities of the player appearances table. Morevover, this page supports certain functionalities like sorting and pagination. The page can be seen below

Figure 3.31: The player appearances page

The table headers are interactive as they allow you to sort the table entries based on the header that you press, if you press on the same header again you switch the sorting by that header from ascending to descending. That functionality is shown by the small icon as shown in the figure above.

As for editing, we get prompted with a pop up as it can be seen in the figure below:

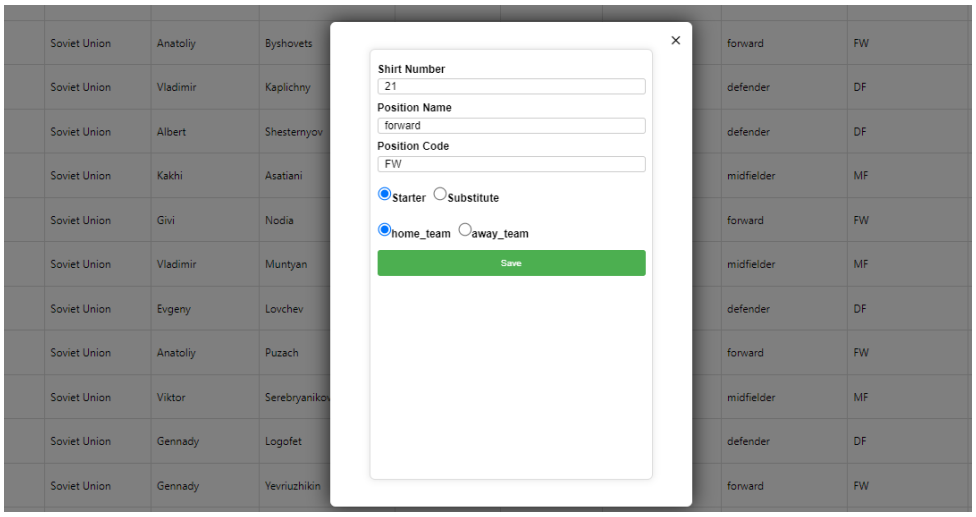

Figure 3.32: The player appearances page insert pop up

Also an appearance can be added as shown below:

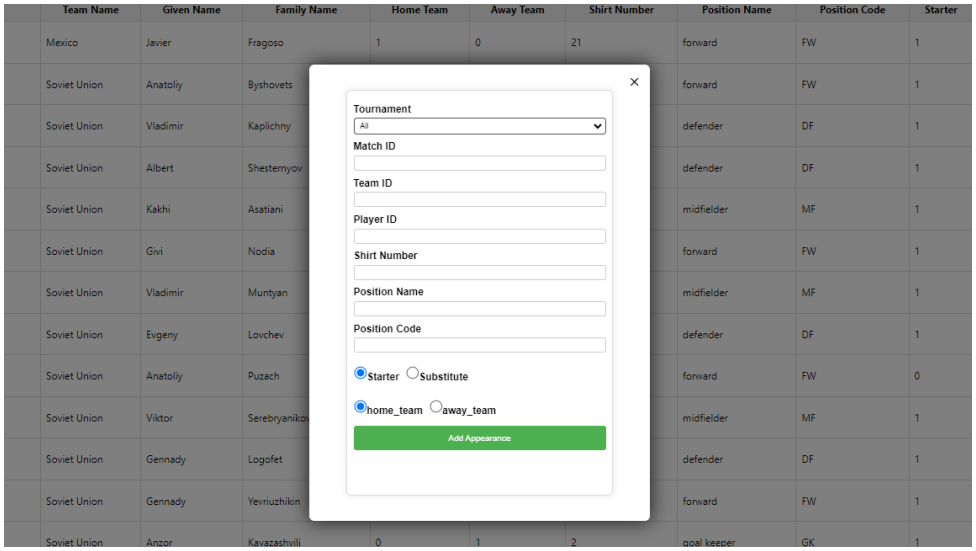

Figure 3.33: The player appearances page update pop up

4. Technologies

4.1. Database

For this project we used MySql database management system. We used the given SQLite dataset given in the Github repository of the dataset[1]. We translated SQLite dataset using the MySql Workbench application. We generated the .sql file that can be fed into a mysql console to generate data. This file is already included in the project files we uploaded. This part of the project was taken care of by Abdullah Shamout. The database was set up using docker container. The implementation details for the container is given in the later subsections.

The code that sets up the database is wrapped inside a db class that is defined as below:

```
# Class: db
# Description: This class contains the database object
class db:
    def __init__(self, db_name):
        self.db_name = db_name
        self.db_username = "root"
        self.db_password = "root"

    def get_connection(self):
        try:
            connection = mysql.connector.connect(
                host="127.0.0.1",
                port="3306",
                user=self.db_username,
                password=self.db_password,
                database=self.db_name,
                auth_plugin='mysql_native_password'
            )
            return connection
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            return None

    @staticmethod
    def disconnect(connection):
        if connection:
            connection.close()
```

Figure 4.1: Database class that handling connections to database

As we can see from the code above we tried to abstract away some of the database connection details in this class. We connect to the local database running on 3306 port using `get_connection` method. This method is called for each request to the server. This of course is a poor implementation since creating a new connection for each request creates some computational overhead. However, this implementation was fast enough for our purposes.

4.2. Programming language

The programming language we used for this project was python. We used flask framework to setup our backend server. We used the official mysql-connector package

to access the database. The application assumes that the database is running at the time of initiation.

We made use of data classes in python to model our tables in the programming language. For each table we used, there exist a data class model that the connecter maps its rows to. We access the database using static methods that belong to a separate class, which is conveniently named after the table it utilizes. An example for the standard we used is given in the picture below:

```python
@dataclass
class Tournament:
    tournament_id: str
    tournament_name: str
    year: int
    start_date: datetime.date
    end_date: datetime.date
    host_country: str
    winner: str
    host_won: bool
    count_teams: int
    group_stage: bool
    second_group_stage: bool
    final_round: bool
    round_of_16: bool
    quarter_finals: bool
    semi_finals: bool
    third_place_match: bool
    final: bool
```

```python
class TournamentDAO():
    @staticmethod
+--- 48 lines: def create_tournament(db: db, tournament: Tournament) -> None:·····

    @staticmethod
    def get_tournament_by_id(db: db, tournament_id: str) -> Tournament:
        try:
            conn = db.get_connection()
            query = "SELECT * FROM tournaments WHERE tournament_id = %s LIMIT 1"
            cursor = conn.cursor()
            cursor.execute(query, (tournament_id,))
            rows = cursor.fetchone()
            cursor.close()
            conn.close()
            if rows is None:
                return None
            return Tournament(*rows)
        except mysql.connector.Error as err:
            print(f"Error: {err}")
```

Figure 4.2: Example Data and Data Access Object class

The handling of the errors was done using try catch block. If there is any error in the execution of the sql statement, we roll back any changes made to the database. For the cases where the error doesn't affect the database, as given in the example above, we log the error message to the console.

The flask framework makes it easier for us to map the endpoints to the functions we want to execute. Below is an example of `matches` endpoint that includes sorting and ordering:

```python
@app.route('/matches/<sort>/<order>/<filter>/<filter_value>', methods=['GET'])
def get_all_matches(sort : str, order : str, filter : str, filter_value : str):
    match = MatchDAO.get_all_matches(db, sort, order, filter, filter_value)
    if match:
        return flask.jsonify(match), 200
    else:
        return flask.jsonify({'message': 'Match not found'}), 404
```

Figure 4.3: Example Data and Data Access Object class

4.3. Frontend

Although we could use the static template functionality provided in the flask framework, we chose to only exchange JSON information. The reason we needed raw data is that we wanted to process the data in the front-end. That is why we chose React as our front-end library. React made the front-end development easier to deploy and debug.

We had easier time when adding additional functionalities that looked nice such as create, update, and delete forms.

4.4. Docker

To make the development more complete, we wrote custom docker builds for each part of our project: mysql server, front-end, and back-end. These docker containers and images can be managed using the Makefile provided in the project repository. These containers facilitated our development by isolating the servers from our local systems. This advantage of the docker containers meant that no matter what OS we used, which includes all of three major operating systems, we had unified results across the board. The docker images and Makefile was set up by the assistance of Kanan Bandaliyev.

5. API

We created a server.py file that is run from the main.py page. There we defined our API endpoints, took in any arguments that we need from the frontend and called on the related backend function for it. a couple offollows:

```
1   @app.route('/matches/<sort>/<order>/<filter>/<filter_value>',
    methods=['GET'])
2   def get_all_matches(sort : str, order : str, filter : str,
    filter_value : str):
3       match = MatchDAO.get_all_matches(db, sort, order, filter,
    filter_value)
4       if match:
5           return flask.jsonify(match), 200
6       else:
7           return flask.jsonify({'message': 'Match not found'}), 404
8
9   @app.route('/squadsJOINED', methods=['GET'])
10  def get_all_squads_joined():
11      page = flask.request.args.get('page', default=0, type=int)
12      items_per_page = flask.request.args.get('items_per_page',
    default=22, type=int)
13      squads = SquadAppearancePlayerDAO.get_squads_paginated(db,
    page, items_per_page)
14      return flask.jsonify(squads)
15
16  @app.route('/tournaments', methods=['DELETE'])
17  def delete_tournament():
18      tournament_id = flask.request.get_json()
19      TournamentDAO.delete_tournament(db, tournament_id)
20      return flask.jsonify({})
21
22  @app.route('/tournaments/teams', methods=['PUT'])
23  def update_teams():
24      teams = flask.request.get_json()["newTeam"]
25      teams = make_dataclass('Team', teams.keys())(**teams)
26      teams = TeamsDAO.update_team(db, teams)
27      return flask.jsonify(teams)
28
29  @app.route('/group_standings', methods=['POST'])
30  def create_group_standings():
31      group_standing = flask.request.get_json()['newGroup_standings
    ']
32      group_standing = make_dataclass('GroupStanding',
    group_standing.keys())(**group_standing)
33      GroupStandingDAO.insert_group_standing(db, group_standing)
34      return flask.jsonify({})
```

5.1. Tournament

The tournaments page deals with the general overview of each project, listing winners, second place. and runner up information. The page fetches data from the database using TournamentDAO class in the backend. The implementation details for each CRUD operation is given in the sections below.

5.1.1. Create

The create operation is executed when we save the tournament we want to add in the modal form of the tournaments page. This operation is mapped to `create_tournament` function of the TournamentDAO class that is defined as shown below:

```
class TournamentDAO():
    @staticmethod
    def create_tournament(db: db, tournament: Tournament) -> None:
        try:
            conn = db.get_connection()
            query = """INSERT INTO tournaments (
                        tournament_id,
                        tournament_name,
                        year,
                        start_date,
                        end_date,
                        host_country,
                        winner,
                        host_won,
                        count_teams,
                        group_stage,
                        second_group_stage,
                        final_round,
                        round_of_16,
                        quarter_finals,
                        semi_finals,
                        third_place_match,
                        final
                    ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s,
    %s, %s, %s, %s, %s, %s, %s)
                """
            cursor = conn.cursor()
            cursor.execute(query, (
                tournament.tournament_id,
                tournament.tournament_name,
                tournament.year,
                tournament.start_date,
                tournament.end_date,
                tournament.host_country,
                tournament.winner,
```

```
                tournament.host_won,
                tournament.count_teams,
                tournament.group_stage,
                tournament.second_group_stage,
                tournament.final_round,
                tournament.round_of_16,
                tournament.quarter_finals,
                tournament.semi_finals,
                tournament.third_place_match,
                tournament.final
            ))
            cursor.close()
            conn.commit()
            db.disconnect(conn)
        except mysql.connector.Error as err:
            print(f"Error: {err}")
```

5.1.2. Read

The read operation is handled by `get_all_tournaments` function for the same class. This function is invoked when we try to fetch all the data that we see on the main page. The implementation details for the function is given below:

```
class TournamentDAO():
    @staticmethod
    def get_all_tournaments(db: db, sort: str, order: str, gender:
    str) -> list[TournamentWithPlaces]:
        try:
            conn = db.get_connection()
            query = f"""WITH second AS (SELECT tournament_id,
    team_name AS second from tournament_standings LEFT JOIN teams
    using(team_id) WHERE position = 2),
                        third AS (SELECT tournament_id, team_name AS
    third from tournament_standings LEFT JOIN teams using(team_id)
    WHERE position = 3),
                        fourth AS (SELECT tournament_id, team_name AS
     fourth from tournament_standings LEFT JOIN teams using(team_id)
    WHERE position = 4)
                        select * from tournaments
                            LEFT JOIN second using(tournament_id)
                            LEFT JOIN third using(tournament_id)
                            LEFT JOIN fourth using(tournament_id)
                        WHERE tournament_name
                        LIKE %s
                        ORDER BY {sort} {order}
                    """
            cursor = conn.cursor()
            cursor.execute(query, (gender,))
```

```
            rows = cursor.fetchall()
            cursor.close()
            db.disconnect(conn)
            tournaments = [TournamentWithPlaces(*row) for row in rows
    ]
            return tournaments
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            conn.rollback()
        finally:
            cursor.close()
            conn.close()
```

As we can see to get the second, runner up and the forth place teams, we need to do the subquerries using with statements. We couldn't use subquerries because we can only have one value in the subquery. The subquerries deal with `tournament_standings` table which has the required information in its tables. Then the `team_id` is joined with the `teams` table to get the team name.

We can sort and filter the given tournaments using the dropdown menu for each filter item. The filtering triggers a new read operation into database that uses the same function to determine the given order using `sort` and `order` arguments passed into the function. We can also filter the tournaments based on the gender which is a radio button in the database that works the same way as sorting and ordering. This filter works by checking the `tournament_name` field of the tournament. If it includes the keyword "FIFA Woman" then it concludes that this is a woman's tournament.

5.1.3. Update

The update operation can be handled using the edit button on each tournament. Clicking on the edit button replaces the information about the tournament with the edit fields for each column of the tournament. Then the frontend constructs a JSON object that contains the updated tournament object. This object is then mapped to the `Tournament` dataclass and inserted into the database using this funcition:

```
class TournamentDAO:
    @staticmethod
    def update_tournament(db, tournament: Tournament) -> Tournament:
        try:
            conn = db.get_connection()
            query = """
                    UPDATE tournaments SET
                        tournament_name = %s,
                        year = %s,
                        start_date = %s,
                        end_date = %s,
                        host_country = %s,
```

```
                        winner = %s,
                        host_won = %s,
                        count_teams = %s,
                        group_stage = %s,
                        second_group_stage = %s,
                        final_round = %s,
                        round_of_16 = %s,
                        quarter_finals = %s,
                        semi_finals = %s,
                        third_place_match = %s,
                        final = %s
                    WHERE tournament_id = %s
                """
            cursor = conn.cursor()
            cursor.execute(query, (
                tournament.tournament_name,
                tournament.year,
                tournament.start_date,
                tournament.end_date,
                tournament.host_country,
                tournament.winner,
                tournament.host_won,
                tournament.count_teams,
                tournament.group_stage,
                tournament.second_group_stage,
                tournament.final_round,
                tournament.round_of_16,
                tournament.quarter_finals,
                tournament.semi_finals,
                tournament.third_place_match,
                tournament.final,
                tournament.tournament_id
            ))
            cursor.close()
            conn.commit()
            conn.close()
            return TournamentDAO.get_tournament_by_id(db, tournament.
tournament_id)
        except mysql.connector.Error as err:
            print(f"Error: {err}")
```

The function returns back the updated Tournament object to the frontend so that the information can be updated if the operation was successful.

5.1.4. Delete

Delete operation is, similar to the operations above, handled using a delete button. When the button is clicked the corresponding function is called. The endpoint accepts a

`tournament_id` as a string which is then passed to delete function for `TournamentDAO` class delete function:

```
class TournamentDAO:
    @staticmethod
    def delete_tournament(db, tournament_id: int) -> None:
        try:
            conn = db.get_connection()
            query = "DELETE FROM tournaments WHERE tournament_id = %s
    "
            cursor = conn.cursor()
            cursor.execute(query, (tournament_id,))
            cursor.close()
            conn.commit()
            conn.close()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
```

As we can see this operation is very simple. The delete operation fails when there is a foreign key constraint on the `tournament_id` key.

5.2. Awards

Awards page displays the awards for the tournaments. This page is using the similar principles as the Tournaments page. This page deals with two tables: `awards` and `award_winners`. The `awards` table shows the awards that are existing throughout all of the tournaments while the `award_winners` is a table with the details about who won which award in that specific tournament. The CRUD operations for this page is listed in the subsections below.

5.2.1. Create

Create operation of the awards page can be utilized using the "Add Award" button. Upon clicking the frontend will make a call to the database to query the name of the awards together with the tournament names to keep the database consistent and make the experience easier. Upon filling the shown form one can save the award which will trigger the given function in the AwardsDAO class:

```
class AwardDAO:
    @staticmethod
    def create_award(db: db, award: AwardWinner):
        try:
            conn = db.get_connection()
            cursor = conn.cursor()
            query = """
                    INSERT INTO award_winners (tournament_id,
    award_id, shared, player_id, team_id)
                    VALUES (%s, %s, %s, %s, %s)
                    """
```

```
            cursor.execute(query, (award.tournament_id, award.
award_id, award.shared, award.player_id, award.team_id))
            conn.commit()

        except mysql.connector.Error as err:
            print(f"Error: {err}")
        finally:
            cursor.close()
            conn.close()
```

This simple query will create a row in the `award_winners` table.

5.2.2. Read

The `get_all_awards` function will be called from the frontend when we need to see the awards in the page. The implementation of this function is described as below:

```
    class AwardDAO:
    @staticmethod
    def get_all_awards(db: db, tournament_filter: str, award_filter:
str, sort: str) -> list[AwardWithTournament]:
        try:
            conn = db.get_connection()
            cursor = conn.cursor()
            if award_filter == 'all':
                award_filter = '%'

            if tournament_filter == 'all':
                tournament_filter = '%'

            if sort == 'tournament_name':
                sort = 'tournament_id DESC, award_id ASC'
            elif sort == 'award_name':
                sort = 'award_id ASC, tournament_id DESC'
            elif sort == 'player_name':
                sort = 'player_id DESC'

            query = f"""
                SELECT tournament_id, player_id, award_id,
                       CONCAT(given_name, CONCAT( ' ', family_name))
AS player_name,
                       tournament_name, award_name, team_id
                FROM award_winners
                LEFT JOIN tournaments USING (tournament_id)
                LEFT JOIN awards USING (award_id)
                LEFT JOIN players USING (player_id)
                WHERE award_name LIKE %s AND tournaments.
tournament_id LIKE %s
                ORDER BY {sort}
```

```
                """
            cursor.execute(query, (award_filter, tournament_filter))

            rows = cursor.fetchall()
            return [AwardWithTournament(*row) for row in rows]

        except mysql.connector.Error as err:
            print(f"Error: {err}")
        finally:
            cursor.close()
            conn.close()
```

As we can see this function takes in a couple of arguments as input which were determined from the frontend. These arguments determine the sorting, ordering, and filtering of the awards. The `tournament_filter` argument is a `tournament_id` reference which tells us the tournament we want to see the awards of. Similarly, we have the `award_name` filter to get the award category we want to be displayed.

We can also search for the name of the players to see if they got any awards. This can be done using the search bar in the page. When there is an update in the search bar, a new query is made into the database that is implemented as given:

```
class AwardDAO:
    @staticmethod
    def search_all_awards(db: db, tournament_filter: str,
    award_filter: str, sort: str, search: str):
        try:
            conn = db.get_connection()
            cursor = conn.cursor()
            if award_filter == 'all':
                award_filter = '%'

            if tournament_filter == 'all':
                tournament_filter = '%'

            if sort == 'tournament_name':
                sort = 'tournament_id DESC, award_id ASC'
            elif sort == 'award_name':
                sort = 'award_id ASC, tournament_id DESC'
            elif sort == 'player_name':
                sort = 'player_id DESC'

            query = f"""
                    SELECT tournament_id, player_id, award_id,
                    CONCAT(given_name, CONCAT( ' ', family_name)) AS
    player_name,
                    tournament_name, award_name, team_id
                    FROM award_winners
                    LEFT JOIN tournaments USING (tournament_id)
```

```
                    LEFT JOIN awards USING (award_id)
                    LEFT JOIN players USING (player_id)
                    WHERE award_name LIKE %s AND tournaments.
    tournament_id LIKE %s AND CONCAT(given_name, CONCAT( ' ',
    family_name)) LIKE CONCAT('%', %s, '%')
                    ORDER BY {sort}
                    """
            cursor.execute(query, (award_filter, tournament_filter,
    search))

            rows = cursor.fetchall()
            return [AwardWithTournament(*row) for row in rows]

        except mysql.connector.Error as err:
            print(f"Error: {err}")
        finally:
            cursor.close()
            conn.close()
```

As we can see this function is very similar to the one given above. The only difference is that we want to search for the player name. We do this by concatenating `given_name` and `family_name` of the player and see if it matches the new `search` argument in the function.

5.2.3. Update

For this specific table update operation is not needed. The reason is that there is no non key column that is worth updating the award for. If we need to update a given award to refer to another player, we can just delete and add a new one instead of updating the row.

5.2.4. Delete

For the delete operation, we can perform a simple query that is given as below:

```
class AwardDAO:
    @staticmethod
    def delete_award(db: db, tournament_id: str, award_id: str,
    player_id: str):
        try:
            conn = db.get_connection()
            cursor = conn.cursor()
            query = """
                    DELETE FROM award_winners
                    WHERE tournament_id = %s AND award_id = %s AND
    player_id = %s
                    """
```

```
            cursor.execute(query, (tournament_id, award_id, player_id
    ))
            conn.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
        finally:
            cursor.close()
            conn.close()
```

This operation requires `tournament_id`, `award_id`, and `player_id` fields to identify the specific award winner. These arguments are sent to the backend from frontend using the JSON body.

5.3. Matches

This page of the web app displays details about the matches played in the World Cup tournaments in the dataset. It does so by utilizing two tables, matches table and goals table. The CRUD operations on the matches table can all be done from this page. It also has some features like filtering and sorting in real-time, as well as validations for the insert and update operations. Finally, it's linked to all other relevant pages. More details about the backend implementation of the matches table will be provided below. Other details about the operations on the goals table and bookings table will be explained in another section.

5.3.1. Create

On this page, the create operation of the matches table is implemented. The following INSERT command is used to add a new match to the database:

```
    @staticmethod
    def create_match(db: db, match: Match) -> None:
        try:
            connection = db.get_connection()
            query = """
                INSERT INTO matches(
                    tournament_id,
                    match_id,
                    match_name,
                    stage_name,
                    group_name,
                    group_stage,
                    knockout_stage,
                    replayed,
                    replay,
                    match_date,
                    match_time,
                    stadium_id,
                    home_team_id,
```

```
                    away_team_id,
                    score,
                    home_team_score,
                    away_team_score,
                    home_team_score_margin,
                    away_team_score_margin,
                    extra_time,
                    penalty_shootout,
                    score_penalties,
                    home_team_score_penalties,
                    away_team_score_penalties,
                    result,
                    home_team_win,
                    away_team_win,
                    draw
                ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
            """
            cursor = connection.cursor()
            cursor.execute(query, (
                    match.tournament_id,
                    match.match_id,
                    match.match_name,
                    match.stage_name,
                    match.group_name,
                    match.group_stage,
                    match.knockout_stage,
                    match.replayed,
                    match.replay,
                    match.match_date,
                    match.match_time,
                    match.stadium_id,
                    match.home_team_id,
                    match.away_team_id,
                    match.score,
                    match.home_team_score,
                    match.away_team_score,
                    match.home_team_score_margin,
                    match.away_team_score_margin,
                    match.extra_time,
                    match.penalty_shootout,
                    match.score_penalties,
                    match.home_team_score_penalties,
                    match.away_team_score_penalties,
                    match.result,
                    match.home_team_win,
                    match.away_team_win,
```

```
66                      match.draw
67              ))
68              connection.commit()
69          except mysql.connector.Error as error:
70              connection.rollback()
71              print("MySQL Error:", error)
72          finally:
73              cursor.close()
74              connection.close()
```

Listing 5.1: Insert Matches

This inserts into the match table a new match tuple with values for all the fields of the table. The try, except, and finally blocks are used for exception handling. In the try block, first a connection to the database is established, Line 4. Then in lines 5-36, the MySQL INSERT command is defined, and placeholders are used for SQL injection prevention. After that, a new cursor object is created, and the command is executed by passing the query and the correct arguments into the execute() function. Finally, the changes are committed. In the except block, if an error occurs, a rollback is done to revert the state of the database to the point before the execution of the command. The finally block ensures that the cursor and connection are closed even if an error occurs.

Validation of the data inserted was emphasized and implemented carefully to ensure data integrity and accuracy. The code will not be displayed here as many query were used for this purpose but the application will be briefly explain. All select fields, which are mostly foreign keys in the backend, are fetched and displayed in real-time. This means that for example, fields like stadiums, teams, stages, and group names will not be shown until you choose the option before them so that only stadiums used in this tournament can be chosen, and only teams that actually played in this tournament can be chosen and so on. Also, some of the true or false fields like for the group stage, knockout stage, home team win, away team win, and draw are automatically filled according to the attributed related to them. All text fields are validated through patterns and other logical operations to ensure the inserted strings make sense with the data.

5.3.2. Read

The read operations for both goals and matches tables are used on this page. The read operation for goals is to be explained in a different section. For the matches table, matches were fetched according to one of two filters, by tournament or by team.
The implementation of the read operation for matches is shown below:

```
1     @staticmethod
2     def get_all_matches(db: db, sort: str, order: str, filter:
3     str, filter_value : str) -> list[Match]:
4         try:
```

```
            matches = []
            connection = db.get_connection()
            if sort == 'tournament_name' or sort == 'year':
                sort = f'm.tournament_id {order}'
            elif sort == 'Stage':
                sort = f'''
                        CASE
                            WHEN m.stage_name = 'group stage'
                            THEN 1
                            WHEN m.stage_name = 'first round'
                            THEN 2
                            WHEN m.stage_name = 'first group
                            stage' THEN 3
                            WHEN m.stage_name = 'second group
                            stage' THEN 4
                            WHEN m.stage_name = 'round of 16'
                            THEN 5
                            WHEN m.stage_name = 'quarter-
                            finals' OR m.stage_name = 'quarter-
                            final' THEN 6
                            WHEN m.stage_name = 'semi-finals'
                            OR m.stage_name = 'semi-final' THEN
                            7
                            WHEN m.stage_name = 'third-place
                            match' THEN 8
                            WHEN m.stage_name = 'final round'
                            THEN 9
                            WHEN m.stage_name = 'final' THEN 10
                        END
                        {order}
                        '''
            elif sort == 'Score-margin':
                sort = f'ABS(m.home_team_score -
                m.away_team_score) {order}'
            elif sort == 'Goals-Scored':
                sort = f'm.home_team_score + m.away_team_score
                {order}, m.home_team_score DESC'
            else:
                sort = f'm.tournament_id {order}'


            if filter == 'team' and filter_value != 'All':
                filter = f'thome.team_id = "{filter_value}" OR
                taway.team_id = "{filter_value}"'
            elif filter == 'tournament' and filter_value !=
            'All':
                filter = f'm.tournament_id= "{filter_value}"'
            else:
```

```
53                 filter = '1 = 1'
54
55             query = f"""
56                     SELECT m.*, s.stadium_name, s.city_name,
57                     thome.team_name, taway.team_name,
58                     t.tournament_name
59                     FROM matches m
60                     LEFT JOIN stadiums s ON m.stadium_id
61                     = s.stadium_id
62                     LEFT JOIN teams thome ON m.home_team_id
63                     = thome.team_id
64                     LEFT JOIN teams taway ON m.away_team_id
65                     = taway.team_id
66                     LEFT JOIN tournaments t ON m.tournament_id
67                      = t.tournament_id
68                     WHERE {filter}
69                     ORDER BY {sort},  m.match_id DESC
70                     """
71             cursor = connection.cursor()
72             cursor.execute(query)
73             results = cursor.fetchall()
74             if results:
75                 for result in results:
76                     match = Match(result[0], result[1], result[2],
    result[3],result[4], result[5], result[6], result[7], result[8],
    result[9], result[10], result[11],result[12], result[13], result
    [14], result[15], result[16], result[17], result[18],result[19],
    result[20], result[21], result[22], result[23], result[24], result
    [25], result[26],result[27],result[28], result[29], result[30],
    result[31], result[32])
77                     matches.append(match)
78                 return matches
79             else:
80                 return None
81
82         except mysql.connector.Error as error:
83             connection.rollback()
84             print("MySQL Error:", error)
85         finally:
86             cursor.close()
87             connection.close()
```

Listing 5.2: Read Matches

Again here, exception handling is used. In the try block, first, an array is defined, a connection to the database is established, and then some if statements related to filtering and sorting and filtering are implemented; these will be discussed in detail in the following subsection. The query is then defined in lines 55-70. Here we have

some joins. The matches table has foreign keys that reference stadiums, teams, and tournaments tables; hence, a left join is performed first joining matches with stadiums to get the stadium names, then joining that with teams once using the home team key and once using the away team key to get the home and away team names, and finally with tournaments to get the tournament name. The reason a left join was used was that we did not wanna lose any of the matches tuples even if they were missing some information. After that, a cursor is obtained, the query is executed, and the results are fetched and appended into the array and then returned. As for the except and finally block, they do exactly the same thing as mentioned before.

The matches can also be fetched by match_id since matches are displayed individually in the matchscoreboard page.

```
    @staticmethod
    def get_match_by_id(db : db, match_id : str) ->
    Match:
        try:
            connection = db.get_connection()
            query = """
                    SELECT m.*, s.stadium_name,
                    s.city_name, thome.team_name,
                    taway.team_name,
                    t.tournament_name,
                    thome.team_code, taway.team_code
                    FROM matches m
                    LEFT JOIN stadiums s ON
                    m.stadium_id = s.stadium_id
                    LEFT JOIN teams thome ON
                    m.home_team_id = thome.team_id
                    LEFT JOIN teams taway ON
                    m.away_team_id = taway.team_id
                    LEFT JOIN tournaments t ON
                    m.tournament_id = t.tournament_id
                    WHERE match_id = %s
                    """
            cursor = connection.cursor()
            cursor.execute(query, (match_id,))
            result = cursor.fetchone()
            if result:
                print(result)
                return Match(result[0], result[1],
                result[2], result[3],result[4],
                result[5], result[6], result[7],
                result[8], result[9], result[10],
                result[11],result[12], result[13],
                result[14], result[15], result[16],
                result[17], result[18],result[19],
                result[20], result[21], result[22],
```

```
36                  result[23], result[24], result[25],
37                  result[26],result[27], result[28],
38                  result[29], result[30], result[31],
39                  result[32], result[33], result[34])
40              else:
41                  return None
42
43          except mysql.connector.Error as error:
44              connection.rollback()
45              print("MySQL Error:", error)
46          finally:
47              cursor.close()
48              connection.close()
```

The code does the same thing as mentioned before except that it does not have filtering and sorting and has a WHERE clause for the match id.

5.3.3. Filtering and Sorting

The matches can be filtered in two ways, by tournament and by teams. They can also be sorted in three ways, by stage, score margin, and goals scored, each in ascending and descending order.

In lines 7-43 of the code, the sort logic is implemented. According to value passed by the API, the sort field in the ORDER BY clause is set to either the default case "m.tournament_id" which in the case of filtering by tournament id does nothing, but in the case of filtering by team, sorts that team's matches by tournament. In the stage case, a CASE is used to assign a numerical value to each stage to be able to sort them. In the score margin case, the sorting is done according to the absolute value of the hometeamscore - awayteam; hence, in descending order, the matches with the highest goal difference between the two teams appear first. In the goals scored case, the ordering is done according to the addition of the score of both teams, so matches with the most goals scored appear first. Also, matches with the same number of goals score are then sorted according to the home team score. The order field refers to either descending or ascending.

In lines 46-53, the filtering logic is implemented. Acording to the value passed by the API, the filter field in the WHERE clause is set to either tournament_id, or to team_id. In the case of team_id, the condition should check if the desired team id matches either the home team id or the away team id as the match should be fetched if either is satisfied. The default condition here is used as a precaution only.

5.3.4. Update

The update operation allows certain attributes of the tuple to be modified as changing certain fields like the score and the tournament could mess up the whole structure of

the tournaments. Hence, we decided to not allow such modifications and allowed only attributes like match_name, match date, match time, and certain boolean attributes like replayed and extratime. The code is provided below:

```
    @staticmethod
    def update_match(db : db, match : Match) -> None:
        try:
            connection = db.get_connection()
            query = """
                    UPDATE matches SET
                        match_name = %s,
                        replayed = %s,
                        replay = %s,
                        match_date = %s,
                        match_time = %s,
                        extra_time = %s
                    WHERE match_id = %s
                    """
            cursor = connection.cursor()
            cursor.execute(query, (
                    match.match_name,
                    match.replayed,
                    match.replay,
                    match.match_date,
                    match.match_time,
                    match.extra_time,
                    match.match_id
            ))
            connection.commit()
        except mysql.connector.Error as error:
            connection.rollback()
            print("MySQL Error:", error)
        finally:
            cursor.close()
            connection.close()
```

Listing 5.3: Update Matches

The function does essentially the same things as explained before. The update command sets the aforementioned fields to a certain match.

5.3.5. Delete

The delete operation allows you to delete the matches from the database; however, due to foreign key constraints, a match can only be deleted after all tuples in other tables that reference it are deleted. Unfortunately, since several tables have a foreign key referencing this table it was difficult to implement deleting for all. However, the feature

can be tested by adding a new match and deleting it as was demonstrated in the demo session. The code is provided below:

```
    @staticmethod
    def delete_match(db : db, match_id : str) -> None:
        try:
            connection = db.get_connection()
            query = """
                    DELETE FROM matches WHERE match_id = %s
                    """
            cursor = connection.cursor()
            cursor.execute(query, (match_id,))
            connection.commit()
        except mysql.connector.Error as error:
            connection.rollback()
            print("MySQL Error:", error)
        finally:
            cursor.close()
            connection.close()
```

Listing 5.4: Delete Matches

The function does essentially the same things as explained before. The delete command deletes matches with a certain match_id.

5.4. Goals and Bookings

5.4.1. Create

The create operation was implemented for the goals table but not the bookings. The operation can be done from the matchscoreboad page which can be accessed by clicking on the match name in the matches pages as explained in the navigation section.

```
    @staticmethod
    def create_goal(db: db, goal: Goal) -> None:
        try:
            connection = db.get_connection()
            print(goal)
            query = """
                INSERT INTO goals (
                    goal_id,
                    tournament_id,
                    match_id,
                    team_id,
                    home_team,
                    away_team,
                    player_id,
                    shirt_number,
```

```
                    player_team_id,
                    minute_label,
                    minute_regulation,
                    minute_stoppage,
                    match_period,
                    own_goal,
                    penalty
                ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s,
    %s, %s, %s, %s)
            """
            cursor = connection.cursor()
            cursor.execute(query, (
                goal.goal_id,
                goal.tournament_id,
                goal.match_id,
                goal.team_id,
                goal.home_team,
                goal.away_team,
                goal.player_id,
                goal.shirt_number,
                goal.player_team_id,
                goal.minute_label,
                goal.minute_regulation,
                goal.minute_stoppage,
                goal.match_period,
                goal.own_goal,
                goal.penalty
            ))
            connection.commit()
        except mysql.connector.Error as error:
            connection.rollback()
            print("MySQL Error:", error)
        finally:
            cursor.close()
            connection.close()
```

Listing 5.5: Insert Goals

The function does essentially the same thing as explained before for matches. It inserts into the goals table a new goal tuple with values for all the fields of the table. The try, except, and finally blocks are used for exception handling.

5.4.2. Delete

The delete operation was implemented for the goals table but not the bookings. The operation can be done from the matchscoreboad page which can be accessed by clicking on the match name in the matches pages as explained in the navigation section.

```
    @staticmethod
    def delete_goal(db: db, goal_id: str) -> None:
        try:
            connection = db.get_connection()
            query = """
                    DELETE FROM goals WHERE goal_id = %s
                    """
            cursor = connection.cursor()
            cursor.execute(query, (goal_id,))
            connection.commit()
        except mysql.connector.Error as error:
            connection.rollback()
            print("MySQL Error:", error)
        finally:
            cursor.close()
            connection.close()
```

Listing 5.6: Delete Goals

The function does essentially the same thing as explained before for matches. It deletes from goals table goal tuple by goal_id. The try, except, and finally blocks are used for exception handling.

5.4.3. Read

The read operation was implemented for both goals and bookings. The goals are read in two ways, in the main matches page, it is fetched for all matches and in the matchscoreboard page its fetched by match_id. The code for fetching by match_id is shown below as they're essentially the same thing except for a minor difference in the WHERE clause.

```
    @staticmethod
    def get_match_goals(db: db, match_id: str) ->
    list[Goal]:
        goals = []
        try:
            connection = db.get_connection()
            query = """
                    SELECT g.*,
                    CASE WHEN p.given_name = 'not
                    applicable' THEN ' ' ELSE
                    p.given_name END as given_name,

                    CASE WHEN p.family_name = 'not
                    applicable' THEN ' ' ELSE
                    p.family_name END as family_name
                    FROM goals g
```

```
                    LEFT JOIN players p ON g.player_id
                    = p.player_id
                    WHERE match_id = %s
                    """
            cursor = connection.cursor()
            cursor.execute(query, (match_id,))
            results = cursor.fetchall()
            if results:
                for result in results:
                    goal =  Goal(result[0], result[1],
                    result[2], result[3], result[4],
                    result[5], result[6], result[7],
                    result[8], result[9], result[10],
                    result[11], result[12],result[13],
                    result[14], result[15], result[16])
                    goals.append(goal)
                return goals
            else:
                return None
        except mysql.connector.Error as error:
            connection.rollback()
            print("MySQL Error:", error)
        finally:
            cursor.close()
            connection.close()
```

Listing 5.7: Read Goals

Here the code does essentially the same thing as explained before. As for the CASE used in the query, some of the players' given or family names are set to not applicable. Because we do not want to display "not applicable" in the front end we set it to an empty string when it is the case. The table is joined with the players table to get their given and family names. We chose to use left join to not lose any goal even if it has no information about the player.

The read operation for the bookings table is the same as that for the goal. They are fetched by match_id and only displayed in the matchscoreboard in the frontend. The code is provided below:

```
    @staticmethod
    def get_match_bookings(db : db, match_id : str):
        try:
            connection = db.get_connection()
            query = """ SELECT b.*,
                    CASE WHEN p.given_name = 'not
                    applicable' THEN ' ' ELSE
                    p.given_name END as given_name,
```

```
                    CASE WHEN p.family_name = 'not
                    applicable' THEN ' ' ELSE
                    p.family_name END as family_name
                    FROM bookings b JOIN players p ON
                    b.player_id = p.player_id
                    WHERE match_id = %s """
            cursor = connection.cursor()
            cursor.execute(query, (match_id,))
            results = cursor.fetchall()
            bookings = []
            for result in results:
                booking = Booking(result[0],
                result[1], result[2], result[3],
                result[4], result[5],result[6],
                result[7], result[8], result[9],
                result[10], result[11], result[12],
                result[13], result[14], result[15],
                result[16], result[17])
                bookings.append(booking)
            return bookings
        except mysql.connector.Error as error:
            print(error)
            connection.rollback()
        finally:
            connection.close()
```

Listing 5.8: Read Bookings

5.5. Groupstandings

Groupstanding

The groupstandings page groups groupstandings according to their tournaments, stages and groups and displays them in the form of Group tables. There is a second page for Crud operations.

5.5.1. Read

We first call groupstandings by joining the team names with the team name table. Then, we use data classes to separate the tournament into stages and groups. We put those in the same tournament in the same list, those in the same stage, those in the same stage and the same group in the same group, and return them as a list.

```
    @staticmethod
    def get_all_group_standings_joined(db: db) -> list:
        try:
            conn = db.get_connection()
            query = """
                SELECT gs.*, t.team_name
```

```
                FROM group_standings gs
                JOIN teams t ON gs.team_id = t.team_id
                ORDER BY gs.tournament_id DESC, gs.stage_number ASC, gs.group_name ASC, gs.position ASC
            """
            cursor = conn.cursor()
            cursor.execute(query)
            results = cursor.fetchall()
            tournament_list = {}
            staged_list = {}
            grouped_list = {}
            for result in results:
                (
                    tournament_id, stage_number, stage_name, group_name, position, team_id,
                    played, wins, draws, losses, goals_for, goals_against, goal_difference,
                    points, advanced, team_name
                ) = result
                key1 = (tournament_id)
                key2 = (tournament_id, stage_number)
                key3 = (tournament_id, stage_number, group_name)
                if key1 not in tournament_list:
                    tournament_list[key1] = joined_Groups_of_Tournament(tournament_id=tournament_id, stagedlist=[])
                if key2 not in staged_list:
                    staged_list[key2] = joined_GroupStandings_of_Stages(stage_number=stage_number, stage_name=stage_name, grouplist=[])
                    tournament_list[key1].stagedlist.append(staged_list[key2])
                if key3 not in grouped_list:
                    grouped_list[key3] = joined_GroupStandings_of_Groups(group_name=group_name, group_standings=[])
                    staged_list[key2].grouplist.append(grouped_list[key3])
                grouped_list[key3].group_standings.append(
                    joined_GroupStanding(
                        tournament_id=tournament_id, stage_number=stage_number, stage_name=stage_name,
                        group_name=group_name, position=position, team_id=team_id, played=played,
                        wins=wins, draws=draws, losses=losses, goals_for=goals_for, goals_against=goals_against,
                        goal_difference=goal_difference, points=points, advanced=advanced, team_name=team_name
```

```
                    )
                )
            return list(tournament_list.values())
        except mysql.connector.Error as err:
            conn.rollback()
        finally:
            cursor.close()
            conn.close()
```

5.5.2. Create

Insert operation get 2 parameters database and GroupStanding dataclasses object. It insert to database. Tournament_ids and team_ids autofill in add form, it fetching all tournaments_id from tournaments table and fetching all team_id from team table, then lsiting them as autofill fro them.

```
    @staticmethod
    def insert_group_standing(db: db, group_standing: GroupStanding)
-> None:
        try:
            conn = db.get_connection()
            query="""
                INSERT INTO group_standings (
                    tournament_id,
                    stage_number,
                    stage_name,
                    group_name,
                    position,
                    team_id,
                    played,
                    wins,
                    draws,
                    losses,
                    goals_for,
                    goals_against,
                    goal_difference,
                    points,
                    advanced
                ) VALUES (%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s,%s
,1)"""

            cursor = conn.cursor()
            cursor.execute(query, (
                group_standing.tournament_id,
                group_standing.stage_number,
                group_standing.stage_name,
                group_standing.group_name,
                group_standing.position,
```

```
                group_standing.team_id,
                group_standing.played,
                group_standing.wins,
                group_standing.draws,
                group_standing.losses,
                group_standing.goals_for,
                group_standing.goals_against,
                group_standing.goal_difference,
                group_standing.points
                ))
            conn.commit()

        except mysql.connector.Error as err:
            conn.rollback()
        finally:
            cursor.close()
            conn.close()
```

5.5.3. Delete

It takes the primary keys of the groupstandings table, tournament_id, stage_number, group_name and position, and deletes this groupstanding from the database.

```
    def delete_group_standing(db: db, tournament_id: str,
stage_number: int, group_name: str, position: str) -> None:
        try:
            conn = db.get_connection()
            query="""
                DELETE FROM group_standings
                WHERE tournament_id=%s AND stage_number=%s AND
group_name=%s AND position=%s
                """
            cursor = conn.cursor()
            cursor.execute(query, (
                tournament_id,
                stage_number,
                group_name,
                position
                ))
            conn.commit()
        except mysql.connector.Error as err:
            conn.rollback()
        finally:
            cursor.close()
            conn.close()
```

5.5.4. Update

It can take all values as input and change them except the primary key.

```
    def update_group_standing(db: db, group_standing: GroupStanding)
-> None:
        try:
            conn = db.get_connection()
            query="""
                UPDATE group_standings SET
                    team_id=%s,
                    played=%s,
                    wins=%s,
                    draws=%s,
                    losses=%s,
                    goals_for=%s,
                    goals_against=%s,
                    goal_difference=%s,
                    points=%s,
                    advanced=%s
                    WHERE tournament_id=%s AND stage_number=%s AND
group_name=%s AND position=%s
                    """
            cursor = conn.cursor()
            cursor.execute(query, (
                group_standing.team_id,
                group_standing.played,
                group_standing.wins,
                group_standing.draws,
                group_standing.losses,
                group_standing.goals_for,
                group_standing.goals_against,
                group_standing.goal_difference,
                group_standing.points,
                group_standing.advanced,
                group_standing.tournament_id,
                group_standing.stage_number,
                group_standing.group_name,
                group_standing.position,
                ))
            conn.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            conn.rollback()
        finally:
            cursor.close()
            conn.close()
```

5.6. Squads

The squads page deals with the general overview of each squad per tournament per team. Since a team might change its squad from tournament to tournament. The page fetches data from the database using SquadDAO and SquadAppearancePlayerDAO classes in the backend. The implementations of the CRUD operations and extra functionalities are shown below

5.6.1. Create

Using the database and the new squad object, a connection is created to the database, an SQL query is written, by filling in the values from the object received from the frontend into the placeholder, the query is executed. There is also some error handling present.

```
    @staticmethod
    def create_squad_member(db: db, squad: Squad) -> None:
        try:
            connection = db.get_connection()
            query = """
                INSERT INTO squads (
                    tournament_id,
                    team_id,
                    player_id,
                    shirt_number,
                    position_name,
                    position_code
                ) VALUES (%s, %s, %s, %s, %s, %s)
            """
            cursor = connection.cursor()
            cursor.execute(query, (
                squad.tournament_id,
                squad.team_id,
                squad.player_id,
                squad.shirt_number,
                squad.position_name,
                squad.position_code
            ))
            connection.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

5.6.2. Read

for reading I have two functions as we might fetch all the squads or a single squad depending on the page that we visit. They are as follows:

```
    @staticmethod
    def get_single_squad(db: db,tournament_id, team_id):
        try:
            connection = db.get_connection()
            query = """
                SELECT s.tournament_id, s.team_id, s.player_id, s.shirt_number, s.position_name, s.position_code,
                    p.family_name, p.given_name, t.team_name, tr.tournament_name
                FROM squads s
                JOIN players p ON s.player_id = p.player_id
                JOIN teams t ON s.team_id = t.team_id
                JOIN tournaments tr ON s.tournament_id = tr.tournament_id
                WHERE s.tournament_id = %s AND s.team_id = %s
                ORDER BY s.tournament_id DESC, s.team_id ASC
            """
            cursor = connection.cursor()
            cursor.execute(query, (tournament_id, team_id))
            results = cursor.fetchall()

            if not results:
                return None

            squad_members = [
                Squad(
                    tournament_id, team_id, player_id, shirt_number, position_name, position_code,
                    family_name=family_name, given_name=given_name, team_name=team_name, tournament_name=tournament_name
                )
                for (
                    _, _, player_id, shirt_number, position_name, position_code,
                    family_name, given_name, team_name, tournament_name
                ) in results
            ]

            return actual_squad(tournament_id, team_id, squad_members[0].team_name, squad_members[0].tournament_name, squad_members)

        except mysql.connector.Error as err:
            print(f"Error: {err}")
```

```
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

```
    @staticmethod
    def get_squads_paginated(db: db, page: int, items_per_page: int) -> list:
        try:
            connection = db.get_connection()
            offset = (page) * items_per_page
            query = """
                SELECT s.tournament_id, s.team_id, s.player_id, s.shirt_number, s.position_name, s.position_code,
                    p.family_name, p.given_name, t.team_name, tr.tournament_name
                FROM squads s
                JOIN players p ON s.player_id = p.player_id
                JOIN teams t ON s.team_id = t.team_id
                JOIN tournaments tr ON s.tournament_id = tr.tournament_id
                ORDER BY s.tournament_id DESC, s.team_id ASC
                LIMIT %s OFFSET %s
            """
            cursor = connection.cursor()
            cursor.execute(query, (items_per_page, offset))
            results = cursor.fetchall()
            if results is None:
                return None

            grouped_squads = {}
            for result in results:
                tournament_id, team_id, player_id, shirt_number, position_name, position_code, family_name, given_name, team_name, tournament_name = result
                key = (tournament_id, team_id, team_name, tournament_name)
                if key not in grouped_squads:
                    grouped_squads[key] = actual_squad(tournament_id, team_id, team_name, tournament_name, [])
                grouped_squads[key].squad.append(
                    Squad(tournament_id, team_id, player_id, shirt_number, position_name, position_code,
                        family_name=family_name, given_name=given_name, team_name=team_name, tournament_name=tournament_name)
                )

            return list(grouped_squads.values())
        except mysql.connector.Error as err:
```

```
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

As it can be seen in the previous code, the get all function for squads utilizes pagination with the limit and offset, the handling of those variables is done at the frontend and are sent to the backend with updated values when the scroll bar reaches its end.

5.6.3. Update

The update operation works by changing values present at the frontend per ID, then the changed data is sent to the backend. The function is shown below:

```
    @staticmethod
def update_squad_member(db: db, squad: Squad) -> None:
    try:
        connection = db.get_connection()
        query = """
            UPDATE squads SET
                shirt_number = %s,
                position_name = %s,
                position_code = %s
            WHERE tournament_id = %s AND team_id = %s AND
player_id = %s
        """
        cursor = connection.cursor()
        cursor.execute(query, (
            squad.shirt_number,
            squad.position_name,
            squad.position_code,
            squad.tournament_id,
            squad.team_id,
            squad.player_id
        ))
        connection.commit()
    except mysql.connector.Error as err:
        print(f"Error: {err}")
        connection.rollback()
    finally:
        cursor.close()
        connection.close()
```

5.6.4. Delete

For the delete function, I simply search for the player to be deleted from the squad by his primary key which is compound in this case. The function is as follows:

```
    @staticmethod
    def delete_squad_member(db: db, tournament_id: str, team_id: str,
    player_id: str) -> None:
        try:
            connection = db.get_connection()
            query = """
                DELETE FROM squads WHERE tournament_id = %s AND
team_id = %s AND player_id = %s
            """
            cursor = connection.cursor()
            cursor.execute(query, (tournament_id, team_id, player_id)
)
            connection.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

5.7. Players

THe players page shows all the players across all the tournaments and matches. From there general information about the players is shown. Since there are a lot of players, the page is paginated with a smart scroll bar. There is also a smart search bar that queries the table with word parts instead of the entire text which brings better results if you dont know the exact words to search for. It autocompletes your typing and shows options for you to pick from that send you to the relative player details page. There is also sorting and filtering capabilities that work in sync with the pagination.

5.7.1. Create

To create a player I use the following function.

```
    @staticmethod
    def create_player(db: db, player: Player) -> None:
        try:
            connection = db.get_connection()
            cursor = connection.cursor()
            query = """
                INSERT INTO players (
                    player_id,
                    family_name,
                    given_name,
                    birth_date,
                    female,
                    goal_keeper,
                    defender,
```

```
                    midfielder,
                    forward,
                    count_tournaments,
                    list_tournaments,
                    player_wikipedia_link
                ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s,
    %s)
            """
            cursor.execute(query, (
                player.player_id,
                player.family_name,
                player.given_name,
                player.birth_date,
                player.female,
                player.goal_keeper,
                player.defender,
                player.midfielder,
                player.forward,
                player.count_tournaments,
                player.list_tournaments,
                player.player_wikipedia_link
            ))
            connection.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

from the frontend I receive the object that I want to create, after creating it from the received JSON, I insert it into the database as shown above.

5.7.2. Read

For the read operation, I used two functions since one is used for the player details page and the other is used for the players page where all the players are displayed. The get single player is a simple get by ID as shown below:

```
        @staticmethod
    def get_player(db: db, player_id: str) -> Player:
        try:
            connection = db.get_connection()
            query = """
                SELECT * FROM players WHERE player_id = %s
            """
            cursor = connection.cursor()
            cursor.execute(query, (player_id,))
            result = cursor.fetchone()
```

```
            if result is None:
                return None
            return Player(*result)
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

while the get all is a bit more complicated since it includes pagination, soritng and filtering. It's implementation can be seen below:

```

    @staticmethod
    def get_all_players_paginated(db: db, page: int, items_per_page: int, female: str, goal_keeper: str, defender: str, midfielder: str, forward: str, sorting_field: str, sorting_order:str) -> list:
        try:
            connection = db.get_connection()
            offset = page * items_per_page

            filters = []
            if female != 'all':
                filters.append(f"female = {female}")
            if goal_keeper != 'all':
                filters.append(f"goal_keeper = {goal_keeper}")
            if defender != 'all':
                filters.append(f"defender = {defender}")
            if midfielder != 'all':
                filters.append(f"midfielder = {midfielder}")
            if forward != 'all':
                filters.append(f"forward = {forward}")

            if sorting_field == 'neither' or sorting_order == 'neither':
                sorting_field = 'player_id'
                sorting_order = 'ASC'

            query = f"""
                SELECT * FROM players
                {"WHERE " + " AND ".join(filters) if filters else ""}
                ORDER BY {sorting_field} {sorting_order}
                LIMIT %s OFFSET %s
            """
            cursor = connection.cursor()
            cursor.execute(query, (items_per_page, offset))
            results = cursor.fetchall()
            if results is None:
```

```
                return None
            return [Player(*result) for result in results]
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

based on what I get from the frontend as filters and sorting parameters, I check the data with if statements, then I add them to the SQL query dynamically. Also there is handling for pagination using the limit and offset part of the query.

5.7.3. Update

The update function is a simple update that updates per ID, it edits the information on the frontend, and the edited object is sent to the backend where its cross searched with its ID and the table is updated with its new values. The implementation can be seen below:

```
    @staticmethod
def update_player(db: db, player: Player) -> None:
    try:
        connection = db.get_connection()
        query = """
            UPDATE players SET
                family_name = %s,
                given_name = %s,
                birth_date = %s,
                female = %s,
                goal_keeper = %s,
                defender = %s,
                midfielder = %s,
                forward = %s,
                count_tournaments = %s,
                list_tournaments = %s,
                player_wikipedia_link = %s
            WHERE player_id = %s
        """
        cursor = connection.cursor()
        cursor.execute(query, (
            player.family_name,
            player.given_name,
            player.birth_date,
            player.female,
            player.goal_keeper,
            player.defender,
            player.midfielder,
            player.forward,
```

```
                player.count_tournaments,
                player.list_tournaments,
                player.player_wikipedia_link,
                player.player_id
            ))
            connection.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

5.7.4. Delete

Same thing for the delete function, it is a simple delete where the ID is found. Its implementation can be seen below:

```
        @staticmethod
    def delete_player(db: db, player_id: str) -> None:
        try:
            connection = db.get_connection()
            query = """
                DELETE FROM players WHERE player_id = %s
            """
            cursor = connection.cursor()
            cursor.execute(query, (player_id,))
            connection.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

5.7.5. search bar

As for the search bar, to have it function in a smart way, we used a couple of expressions to cross search with. Each is a different combination of a concatenation. Also to prevent extra useless loading time, and since its auto complete we limited the completion by 20 entries, thus the implementation is as shown below:

```
        @staticmethod
    def search_all_players(db: db, name: str) -> list:
        try:
            connection = db.get_connection()
            query = """
                SELECT * FROM players
```

```
                WHERE given_name LIKE %s OR family_name LIKE %s
                OR CONCAT(given_name, ' ', family_name) LIKE %s
                OR CONCAT(family_name, ' ', given_name) LIKE %s
                OR CONCAT(given_name, family_name) LIKE %s
                OR CONCAT(family_name, given_name) LIKE %s
                LIMIT 20
            """

            cursor = connection.cursor()
            cursor.execute(query, (f"%{name}%", f"%{name}%", f"%{name}%", f"%{name}%", f"%{name}%", f"%{name}%"))
            results = cursor.fetchall()
            if results is None:
                return None
            return [Player(*result) for result in results]
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

5.7.6. statistics

For each player some statistics are present, to fetch those statistics dynamically, I wrote some queries. The implementation of some of them is shown below:

```

    @staticmethod
    def get_awards_per_player(db: db, player_id) -> int:
        try:
            connection = db.get_connection()
            query = """
            SELECT
                    total_awards_won
                FROM
                    (SELECT
                        player_id,
                        COUNT(*) AS total_awards_won
                    FROM
                        award_winners
                    GROUP BY
                        player_id) AS blah
                WHERE
                    player_id = %s;
            """
            cursor = connection.cursor()
            cursor.execute(query, (player_id,))
            result = cursor.fetchone()
```

```
            if result is None:
                return None
            return result
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()

    @staticmethod
    def get_appearances_per_player(db: db, player_id) -> int:
        try:
            connection = db.get_connection()
            query = """
            SELECT
                    total_appearances
                FROM
                    (SELECT
                        player_id,
                        COUNT(*) AS total_appearances
                    FROM
                        player_appearances
                    GROUP BY
                        player_id) AS blah
                WHERE
                    player_id = %s;
            """
            cursor = connection.cursor()
            cursor.execute(query, (player_id,))
            result = cursor.fetchone()
            if result is None:
                return None
            return result
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

5.8. Player Appearances

The player appearances deals with all the appearances of all the players across all the tournaments and matches. From there you can add appearances to players, delete appearances for them, and update how they appeared "eg. shirt number, starter/substitution ...etc". You can also sort my a lot of attributes. Also since the data in this table is very big we implemented pagination with a smart scroll bar like in every

case we implemented pagination

5.8.1. Create

For creating an appearance, the JSON would be created at the frontend, sent to the backend where it is inserted into the database with a query execution. My implementation can be seen below:

```
    @staticmethod
    def create_player_apperance(db: db, player_apperance:
Player_apperance) -> None:
        try:
            connection = db.get_connection()
            query = """
                INSERT INTO player_appearances (
                    tournament_id,
                    match_id,
                    team_id,
                    home_team,
                    away_team,
                    player_id,
                    shirt_number,
                    position_name,
                    position_code,
                    starter,
                    substitute
                ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
            """
            cursor = connection.cursor()
            cursor.execute(query, (
                player_apperance.tournament_id,
                player_apperance.match_id,
                player_apperance.team_id,
                player_apperance.home_team,
                player_apperance.away_team,
                player_apperance.player_id,
                player_apperance.shirt_number,
                player_apperance.position_name,
                player_apperance.position_code,
                player_apperance.starter,
                player_apperance.substitute
            ))
            connection.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

5.8.2. Read

For getting all the appearances with pagination and sorting, I get those parameters from the frontend, calculate the limit and offset, set the ordering by and its order, then execute the function.

```
    @staticmethod
    def get_all_appearances_paginated(db: db, page: int,
items_per_page: int, order_by: str, order: str) -> list:
        try:
            connection = db.get_connection()
            offset = (page) * items_per_page
            if order_by == "tournament_name":
                order_by = "tr.tournament_name"
            elif order_by == "match_name":
                order_by = "m.match_name"
            elif order_by == "family_name":
                order_by = "p.family_name"
            elif order_by == "given_name":
                order_by = "p.given_name"
            elif order_by == "team_name":
                order_by = "t.team_name"
            else:
                order_by = "tournament_id"

            if order == "asc":
                order = "asc"
            else:
                order = "desc"

            query = f"""
                SELECT
                pa.tournament_id,
                pa.match_id,
                pa.team_id,
                pa.home_team,
                pa.away_team,
                pa.player_id,
                pa.shirt_number,
                pa.position_name,
                pa.position_code,
                pa.starter,
                pa.substitute,
                m.match_name,
                p.family_name,
                p.given_name,
                t.team_name,
                tr.tournament_name
                FROM player_appearances pa
```

```
                JOIN players p ON pa.player_id = p.player_id
                JOIN teams t ON pa.team_id = t.team_id
                JOIN tournaments tr ON pa.tournament_id = tr.
tournament_id
                JOIN matches m ON pa.match_id = m.match_id
                ORDER BY {order_by} {order}
                LIMIT %s OFFSET %s
            """
            cursor = connection.cursor()
            cursor.execute(query, (items_per_page, offset))
            results = cursor.fetchall()
            if results is None:
                return None
            return [Joined_Player_apperance(*result) for result in
results]
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

5.8.3. Update

As for updating a player, again like mentioned previously in other tables, by cross searching the table with the wanted ID, we update its data with the new parameters given in the input. My implementation can be seen below:

```
    @staticmethod
 def update_player_apperance(db: db, player_apperance:
Player_apperance) -> None:
    try:
        connection = db.get_connection()
        query = """
            UPDATE player_appearances SET
                shirt_number = %s,
                position_name = %s,
                position_code = %s,
                starter = %s,
                substitute = %s,
                home_team = %s,
                away_team = %s
            WHERE tournament_id = %s AND match_id = %s AND
team_id = %s AND player_id = %s
        """
        cursor = connection.cursor()
        cursor.execute(query, (
            player_apperance.shirt_number,
```

```
                player_apperance.position_name,
                player_apperance.position_code,
                player_apperance.starter,
                player_apperance.substitute,
                player_apperance.home_team,
                player_apperance.away_team,
                player_apperance.tournament_id,
                player_apperance.match_id,
                player_apperance.team_id,
                player_apperance.player_id
            ))
            connection.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

5.8.4. Delete

As for deleting, by searching for the instance of the appearance with its primary key that is a compound of multiple attributes, if that instance is found it is deleted from the database per the query written there. My implementation can be seen below:

```
        @staticmethod
    def delete_player_apperance(db: db, tournament_id: str, match_id: str, team_id: str, player_id: str) -> None:
        try:
            connection = db.get_connection()
            query = """
                DELETE FROM player_appearances WHERE tournament_id = %s AND match_id = %s AND team_id = %s AND player_id = %s
            """
            cursor = connection.cursor()
            cursor.execute(query, (tournament_id, match_id, team_id, player_id))
            connection.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
            connection.rollback()
        finally:
            cursor.close()
            connection.close()
```

5.9. Teams

This page of the web app displays details about the teams played in the World Cup tournaments. It is performed by using teams and confederations tables. The CRUD

operations on the teams table can all be done from this page. The page also has some features like for example flag images were fetched from the API, interactive search-bar. Additionally, it's linked to all other relevant pages and to the open-web. More details about the page are provided below.

5.9.1. Create

This method allows inserting new team records into the database.

```
class TeamsDAO():
    @staticmethod
    def create_team(db: db, team: Teams) -> None:
        try:
            conn = db.get_connection()
            query = """INSERT INTO teams (
                        team_id,
                        team_name,
                        team_code,
                        mens_team,
                        womens_team,
                        federation_name,
                        region_name,
                        confederation_id,
                        mens_team_wikipedia_link,
                        womens_team_wikipedia_link,
                        federation_wikipedia_link
                    ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s,
    %s)
            """
            cursor = conn.cursor()
            cursor.execute(query, (
                team.team_id,
                team.team_name,
                team.team_code,
                team.mens_team,
                team.womens_team,
                team.federation_name,
                team.region_name,
                team.confederation_id,
                team.mens_team_wikipedia_link,
                team.womens_team_wikipedia_link,
                team.federation_wikipedia_link
            ))
            cursor.close()
            conn.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
```

```
        finally:
            db.disconnect(conn)
```

Operation in UI: Corresponds to the "Add Team" button.

This method uses an `INSERT INTO` SQL statement to add a new team into the teams table. It executes this statement using the team details provided in the team parameter. For instance, to add a new team with the ID "123", the method would take a Teams instance with "123" as one of its attributes and insert it into the database.

5.9.2. Read

```
    @staticmethod
    def get_team_by_id(db:db, team_id: str) -> Teams:
        try:
            conn  = db.get_connection()
            query = "SELECT * FROM teams WHERE team_id = %s"
            cursor = conn.cursor()
            cursor.execute(query, (team_id,))
            row = cursor.fetchone()
            cursor.close()
            if row is None:
                return None
            return Teams(*row)
        except mysql.connector.Error as err:
            print(f"Error: {err}")
        finally:
            db.disconnect(conn)

    @staticmethod
    def get_all_teams(db:db) -> list[Teams]:
        try:
            conn = db.get_connection()
            query = "SELECT * FROM teams"
            cursor = conn.cursor()
            cursor.execute(query)
            rows = cursor.fetchall()
            cursor.close()
            return [Teams(*row) for row in rows]
        except mysql.connector.Error as err:
            print(f"Error: {err}")
        finally:
            db.disconnect(conn)
```

Operation in UI: Displaying the teams on the page.

The `get_team_by_id` method reads a specific team's details using a `SELECT` statement where `team_id` matches the given parameter. If the user clicks on a team in the UI, this method could be used to fetch the details for editing. The `get_all_teams` method would be used to populate the UI with all teams' details.

5.9.3. Update

```
    @staticmethod
    def update_team(db:db, team: Teams) -> None:
        try:
            conn = db.get_connection()
            query = """
                UPDATE teams SET
                    team_name = %s,
                    team_code = %s,
                    mens_team = %s,
                    womens_team = %s,
                    federation_name = %s,
                    region_name = %s,
                    confederation_id = %s,
                    mens_team_wikipedia_link = %s,
                    womens_team_wikipedia_link = %s,
                    federation_wikipedia_link = %s
                WHERE team_id = %s
            """
            cursor = conn.cursor()
            cursor.execute(query, (
                team.team_name,
                team.team_code,
                team.mens_team,
                team.womens_team,
                team.federation_name,
                team.region_name,
                team.confederation_id,
                team.mens_team_wikipedia_link,
                team.womens_team_wikipedia_link,
                team.federation_wikipedia_link,
                team.team_id
            ))
            cursor.close()
            conn.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
        finally:
            db.disconnect(conn)
```

Operation in UI: The "Edit" button under each team.

When a user clicks "Edit" and modifies a team's details, the `update_team` method is called. It uses an `UPDATE` SQL statement to modify the team's data in the database based on the provided `team_id`. For example, if the user changes the team name for Argentina, this method will update the `team_name` field in the corresponding database record.

5.9.4. Delete

```
    @staticmethod
    def delete_team(db:db, team_id: str) -> None:
        try:
            conn = db.get_connection()
            query = "DELETE FROM teams WHERE team_id = %s"
            cursor = conn.cursor()
            cursor.execute(query, (team_id,))
            cursor.close()
            conn.commit()
        except mysql.connector.Error as err:
            print(f"Error: {err}")
        finally:
            db.disconnect(conn)
```

Operation in UI: The "Delete" button under each team.

If the user wants to remove a team, clicking the "Delete" button will invoke the `delete_team` method. This method uses a `DELETE FROM` SQL statement to remove the team from the database where the `team_id` matches the provided parameter.

5.10. Team Stats

The "Team Stats" page is a crucial component of the soccer database application, providing a comprehensive view of various statistical data points related to a team's performance in tournaments. This report outlines the functionality and technical implementation of the Read operation within this page.

5.10.1. Read

The Read operation on the "Team Stats" page is realized through a complex SQL query that aggregates data from multiple views, representing different statistical categories. The operation is invoked when a user selects a team from the "Teams" page, which triggers a redirection to the "Team Stats" page with detailed statistics for the chosen

team.

The data retrieval is facilitated by the `TeamStatsDAO` class, which defines the `get_team_stats` method. This method accepts a team ID and a tournament filter, executing an SQL query that joins several views to compile the statistics.

Teamstats.py:

```
class TeamStatsDAO():
    @staticmethod
    def get_team_stats(db: db, team_id: str, tournament_filter: str)
   -> list:
        try:
            if tournament_filter == 'all':
                tournament_filter = '%'

            conn = db.get_connection()
            query = """SELECT       tac.team_id,       tac.team_name,
       tac.tournament_id,       tac.award_count,      nog.
   number_of_goals,      nofw.number_of_wins,      nod.number_of_draws,
        nol.number_of_losses
            FROM       team_award_count tac
            LEFT JOIN       number_of_goals nog ON tac.team_id = nog.
   team_id AND tac.tournament_id = nog.tournament_id
            LEFT JOIN       number_of_wins nofw ON tac.team_id = nofw.
   team_id AND tac.tournament_id = nofw.tournament_id
            LEFT JOIN       number_of_draws nod ON tac.team_id = nod.
   team_id AND tac.tournament_id = nod.tournament_id
            LEFT JOIN       number_of_losses nol ON tac.team_id = nol.
   team_id AND tac.tournament_id = nol.tournament_id
            where tac.tournament_id like %s and tac.team_id = %s"""
            cursor = conn.cursor()
            cursor.execute(query, (tournament_filter,team_id))
            records = cursor.fetchall()
            cursor.close()
            return [TeamStats(*record) for record in records]
        except mysql.connector.Error as err:
            print(f"Error: {err}")
        finally:
            db.disconnect(conn)
```

Views.sql:

```
-- Create view for number of awards
create view team_award_count as
select count(team_id) as award_count, team_id, team_name,
   tournament_id
from award_winners
         left join fifa.teams using (team_id)
group by team_id, tournament_id;
```

```
7
8  -- Create view for number of goals
9  create view number_of_goals as
10 select count(teams.team_id) as number_of_goals, teams.team_id, teams.
       team_name, tournament_id
11 from teams
12 left join fifa.goals g on teams.team_id = g.team_id
13 group by teams.team_id, tournament_id;
14
15 -- Create view for number of wins
16 create view number_of_wins as
17 select count(m.away_team_win) as number_of_wins, teams.team_id, teams
       .team_name, tournament_id
18 from teams
19          left join fifa.matches m on case when m.away_team_win = 1
       then m.away_team_id = teams.team_id
20                                          when m.home_team_win = 1
       then m.home_team_id = teams.team_id
21      end
22 group by teams.team_id, tournament_id;
23
24 -- Create view for number of draws
25 create view number_of_draws as
26 select count(m.draw) as number_of_draws, teams.team_id, teams.
       team_name, tournament_id
27 from teams
28          left join fifa.matches m on case when m.draw = 1 then m.
       home_team_id = teams.team_id
29      end
30 group by teams.team_id, tournament_id;
31
32 -- Create view for number of losses
33 create view number_of_losses as
34 select count(m.home_team_win) as number_of_losses, teams.team_id,
       teams.team_name, tournament_id
35 from teams
36          left join fifa.matches m on case when m.home_team_win = 1
       then m.away_team_id = teams.team_id
37                                          when m.away_team_win = 1
       then m.home_team_id = teams.team_id
38      end
39 group by teams.team_id, tournament_id;
```

5.11. Managers

Managers table contains: Given name, family name, country, femela or not and wikipedia link. Page shows all managers in boxes.

5.11.1. Create

This function adding new managers to our database.

```
    def insert_manager(db: db, manager: Manager) -> None:
        try:
            conn = db.get_connection()
            query = """
                INSERT INTO managers (
                    manager_id,
                    family_name,
                    given_name,
                    female,
                    country_name,
                    manager_wikipedia_link
                ) VALUES (%s, %s, %s, %s, %s, %s)
            """
            cursor = conn.cursor()
            cursor.execute(query, (
                manager.manager_id,
                manager.family_name,
                manager.given_name,
                manager.female,
                manager.country_name,
                manager.manager_wikipedia_link
            ))
            cursor.close()
            conn.commit()
        except mysql.connector.Error as err:
            conn.rollback()
        finally:
            cursor.close()
            conn.close()
```

5.11.2. Read

This function returns all managers in the list of Manager dataclass so we can show in all managers in page.

```
    def get_all_managers(db: db) -> list:
        try:
            conn = db.get_connection()
            query = """
                SELECT * FROM managers
            """
            cursor = conn.cursor()
            cursor.execute(query)
            result = cursor.fetchall()
            return [Manager(*row) for row in result]
```

```
        except mysql.connector.Error as err:
            conn.rollback()
        finally:
            cursor.close()
            conn.close()
```

5.11.3. Update

Updating every information about manager except manager id, because we are changing by manager_id.

```
    def update_manager(db: db, manager: Manager) -> None:
        try:
            conn = db.get_connection()
            query = """
                UPDATE managers SET
                    family_name = %s,
                    given_name = %s,
                    female = %s,
                    country_name = %s,
                    manager_wikipedia_link = %s
                WHERE manager_id = %s
            """
            cursor = conn.cursor()
            cursor.execute(query, (
                manager.family_name,
                manager.given_name,
                manager.female,
                manager.country_name,
                manager.manager_wikipedia_link,
                manager.manager_id,
            ))
            conn.commit()
        except mysql.connector.Error as err:
            print("Mysql Error:", err)
            conn.rollback()
        finally:
            cursor.close()
            conn.close()
```

5.11.4. Delete

```
    def delete_manager(db: db, manager_id: str) -> None:
        try:
            conn = db.get_connection()
            query = """
                DELETE FROM managers WHERE manager_id = %s
            """
```

```
            cursor = conn.cursor()
            cursor.execute(query, (manager_id,))
            conn.commit()
        except mysql.connector.Error as err:
            conn.rollback()
        finally:
            cursor.close()
            conn.close()
```

6. Challenges

In the development of our database class project, we encountered several challenges related to pagination, data fetching, and the application of filters and sorts. This chapter discusses the issues faced and the solutions implemented to overcome these challenges.

6.1. Pagination Challenges

Pagination is a crucial aspect of our project, allowing users to navigate through large datasets efficiently. However, as we implemented pagination, we encountered issues with data consistency since our pagination is not changing the page, it is scroll bar dependent so everything is on the same page. When applying update, delete, or add operations, the offset-based pagination led to discrepancies in the displayed data.

6.1.1. Data Inconsistency with Offset Based Pagination

The primary challenge arose from the continuous increase in the offset when fetching more data. As a result, if update, delete, or add operations occurred while fetching additional data, the displayed data became inconsistent. This issue was particularly noticeable when sorting or filtering operations were applied.

6.2. Solution Implementation

To address the challenges related to pagination and data consistency, we implemented the following solutions:

6.2.1. Resetting Data on Operations

To maintain data consistency, we introduced a mechanism to reset the fetched data when update, delete, or add operations were applied. This ensured that the subsequent pagination requests started with the most up-to-date dataset, preventing inconsistencies in the displayed information.

6.2.2. Handling Sorts and Filters

To accommodate sorting and filtering operations without compromising data integrity, we implemented a strategy to preserve the applied sorts and filters during pagination. By storing the sorting and filtering criteria, we could maintain a coherent dataset even when fetching additional records.

6.2.3. Optimizing Fetch Operations

To enhance the efficiency of data fetching, we optimized the backend to retrieve only the necessary data based on the pagination, sort, and filter criteria. This not only improved

the overall performance but also mitigated the risk of data inconsistencies and increased the responsiveness of the website.

7. Conclusion

In conclusion, the completion of our database class project has been a valuable and enriching experience. Throughout this project, we gained insights into various technologies and methodologies, enhancing our skills in both frontend and backend development.

7.1. Technical Learnings

One of the key takeaways from this project is the hands-on experience with Python and the Flask framework for backend development. We learned about building robust and scalable APIs, implementing CRUD operations, filtering, sorting, and incorporating search functionality with smart autocomplete features and word detection.

On the frontend, we explored the dynamic capabilities of JavaScript, particularly in conjunction with the React library. Creating responsive and user-friendly interfaces allowed us to create a seamless user experience.

We utilized MySQL for database management. It provided us with a foundation for handling and organizing data efficiently. We learned how to create complex database schemas, execute CRUD operations, apply filtering and sorting, create responsive pagination, and implement intelligent search bars with them.

7.2. Microservices and Dockerization

A significant highlight of our project was the application of microservices architecture. Creating a modular structure with APIs enabled us to develop an easily scalable and maintainable application. Additionally, the utilization of Docker facilitated a consistent and reproducible environment, ensuring compatibility across different systems.

7.3. Collaborative Development and Version Control

Working collaboratively using Git and GitHub was alo a very vital part of our project. Understanding the principles of version control, branching, and merging not only eased our development process but also taught us valuable lessons in teamwork and collaboration.

7.4. Overall Reflection

All in all, this project was not just about coding; it was a great learning experience of how real life projects are done. We faced challenges, solved problems, and honed our skills in various aspects of software development. From creating smart search bars to implementing validations and conducting thorough testing, each phase of the project contributed to our growth as developers.

As a team, we navigated the complexities of building a functional and cohesive application. This project has equipped us with a diverse skill set and a deeper

understanding of the hardships involved in developing a comprehensive database-driven application.

In conclusion, the journey from project inception to completion has been rewarding, providing us with a solid foundation for future endeavors in software development.

Bibliography

[1] The Fjelstul World Cup Database https://github.com/jfjelstul/worldcup/tree/master