Ignacio Abrams and Christopher Brown
CPS 430 - Database Management Systems
Term Project - Fall 2022
12/7/2022

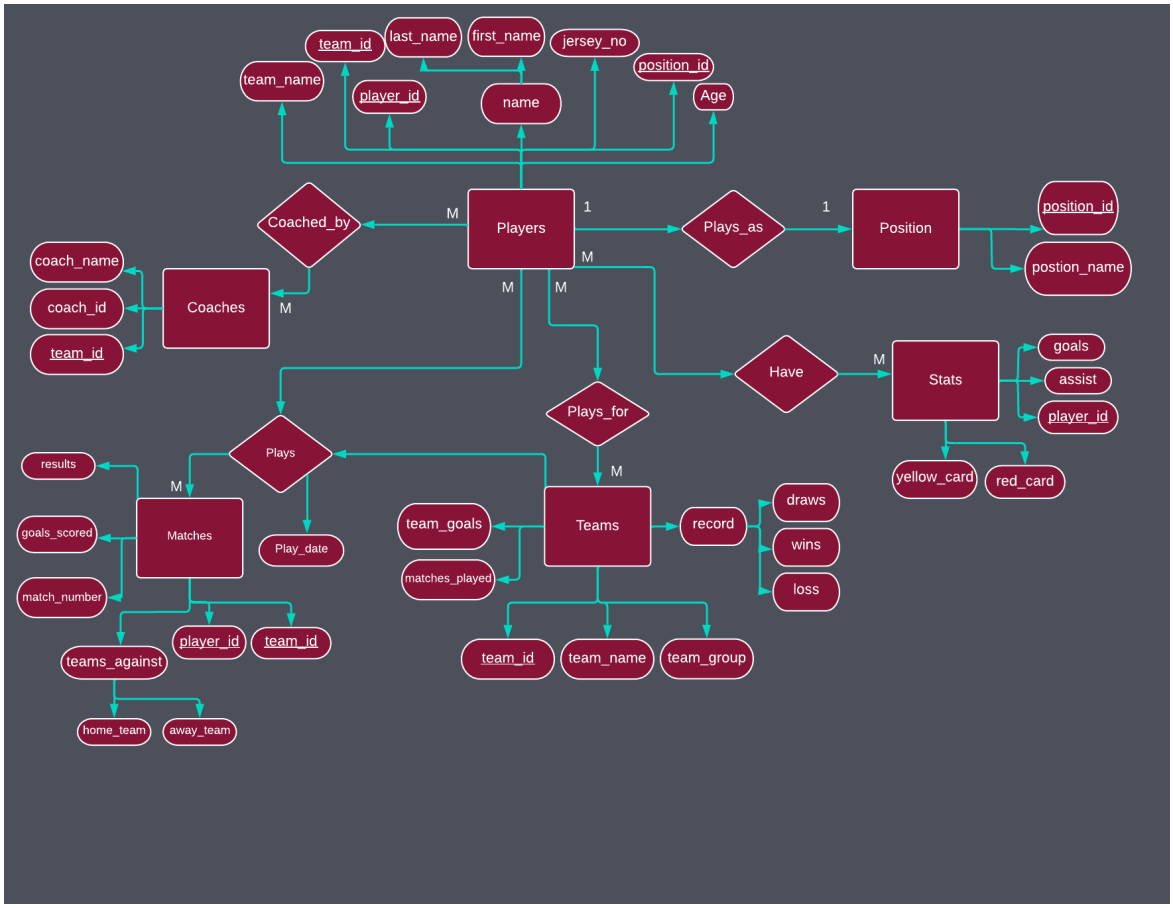
World Cup 2022 Stats Database

Overview:

I. Application Description

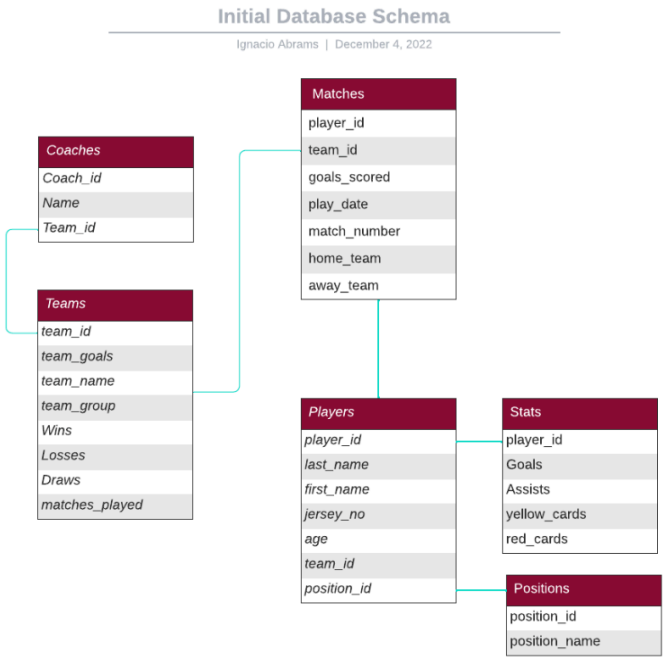
- We decided to create a simple database of the World Cup 2022 in Qatar. Because the World Cup is on going, we thought it was ideal to create a database that could help us keep track of the most important stats from players and teams. Also, we chose to do the World Cup because we thought it would be fun to apply different database concepts to futbol.

II. Conceptual Model



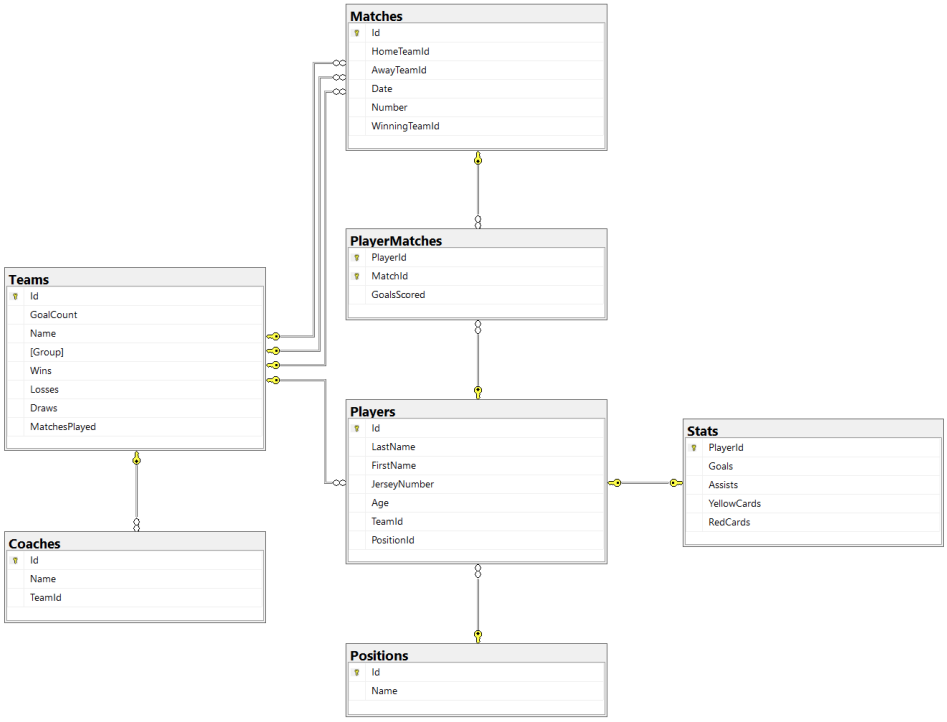
- Entities: Players, Matches, Position, Teams, Coaches, and Stats.
- Relationships: Plays_for, Have, Plays_as, Coached_by, and Plays.
- Weak entities: Stats is a weak entity because it solely depends on the player entity. The identifying relationship is: have.
- Composite attributes: Name. The name attribute is composed of the player's first name and last name. The record of a team is composed by their wins, draws, and losses. The attribute “teams_against” is composed of the teams who will go against each other.
- Relationship attributes: play_date. Is determined by Matches and Teams entity.
- Derived attribute: The record attribute can be derived by the results attribute in the Matches entity.

III. Initial Database Schema



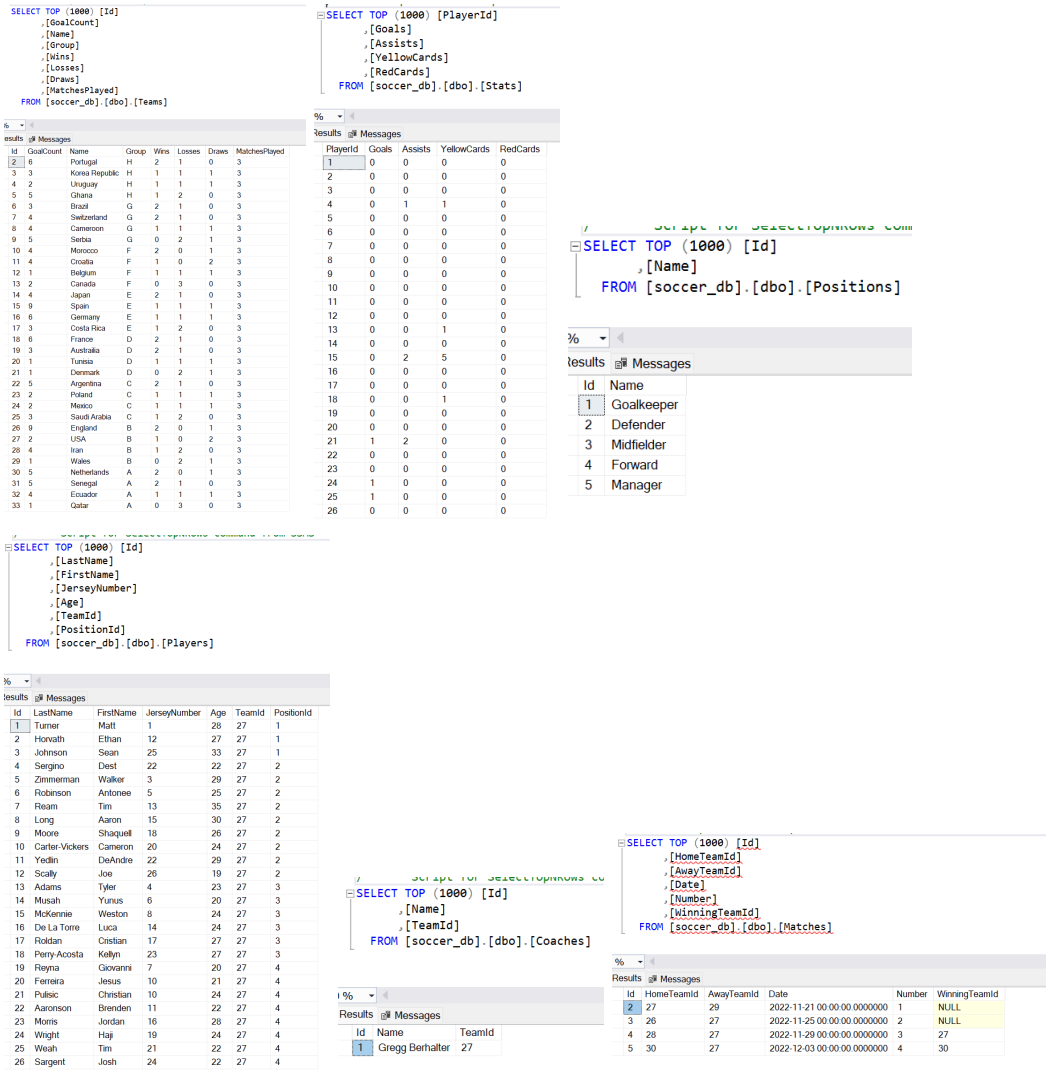
- This schema was taken directly from our ER diagram and resembles the entities and attributes. Our schema is made up of tables that represent our entities, relationships and their attributes.
- Our tables were Coaches, Matches, Teams, Players, Stats, and Positions.
- Most of our relationships were many-to-many relationships, which meant that most of our data could be represented in our schema.

IV. Final Database Schema



- The initial and final schemas remained mostly the same. The biggest difference was in nomenclature. Our choice of Entity Framework Core drove our decision to rename primary keys simply to **Id**, to name our attributes in PascalCase, and to remove underscores from names.
- Also, after revising the initial schema we added a new table called PlayerMatches so that we could get the statistics of a given player in a certain match.
- Our primary keys were mainly Id and our foreign keys were PlayerId, PositionId, TeamId, and MatchId.

V. Database Instance



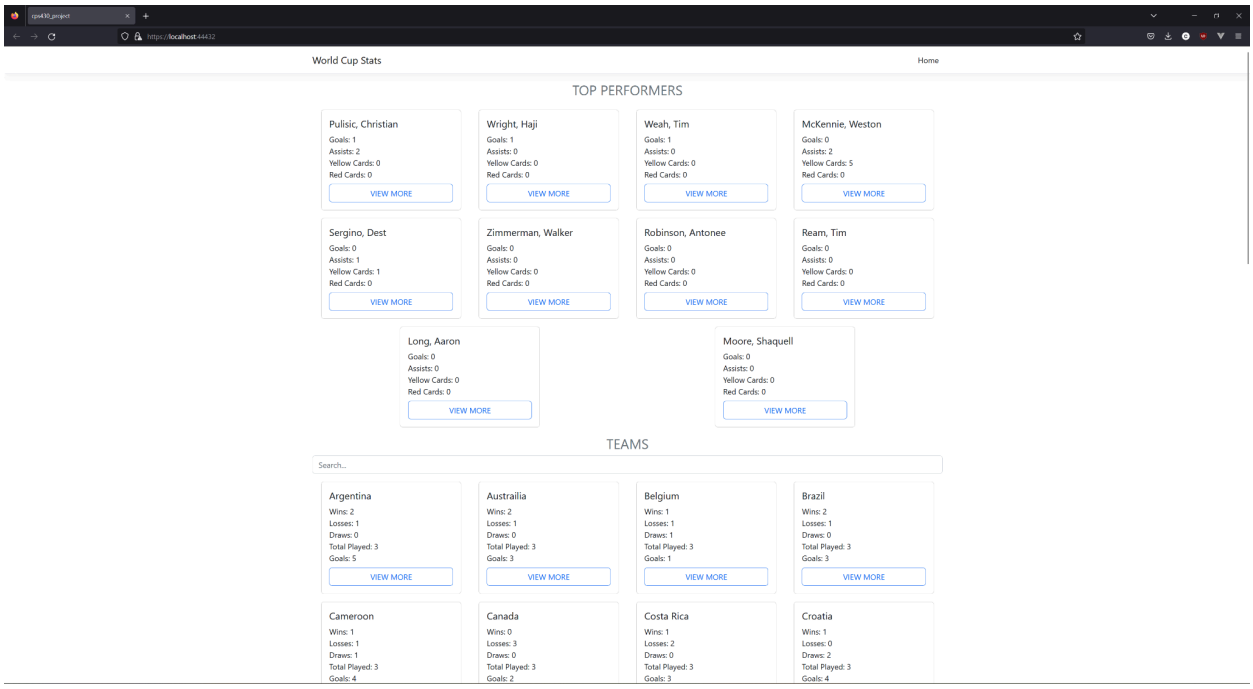
- Our database has all teams entered in, with the most complete information for the United States. This gave us enough data to adequately test our queries and come up with meaningful results.

VI. Data Manipulation

- Data manipulations are discussed in-depth in the Web Interface section of this report.

VII. Web Interface

- If you would like to see the source code for the application, it is available here.
- The website for this application was written with Angular, ASP.NET, and Entity Framework Core. These frameworks drove design-time decisions, especially the schema and naming conventions of the database.



The home page displays several pieces of important information, each with its own, dedicated query. No data manipulation was done on the front end of the website or past the database.

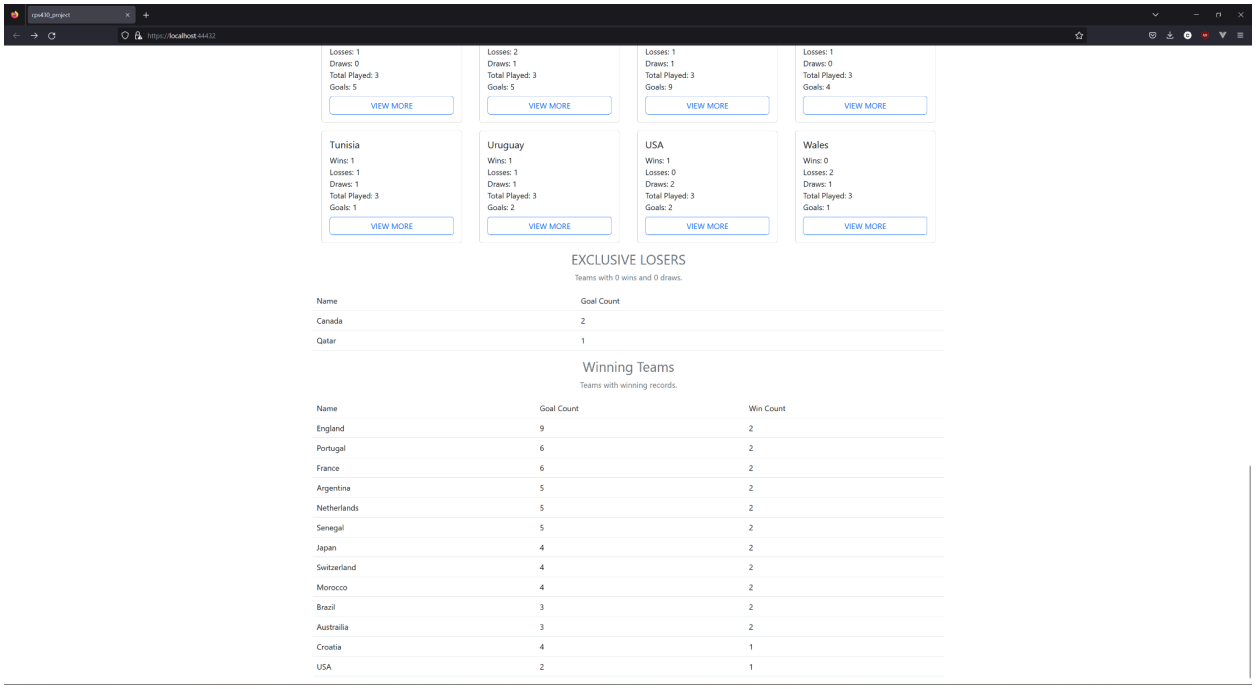
The first query seen is the Top Performers, which aims to show which players, cup wide, have performed the best overall.

```
return _context.Players
    .OrderByDescending(x => x.Stats.Goals)
    .ThenByDescending(x => x.Stats.Assists)
    .ThenBy(x => x.Stats.YellowCards)
    .ThenBy(x => x.Stats.RedCards)
    .Take(4)
    .AsNoTracking()
    .Select(x => new Player
    {
        Id = x.Id,
        FirstName = x.FirstName,
        LastName = x.LastName,
        Stats = x.Stats,
    })
    .ToList();
```

The second and third queries are seen in the above screenshot, under “Teams.” The more basic of these queries simply pulls back basic information about *every* team in the database. The second query takes a filter into account, attempting to match the searched name to the team name. This search is performant, and is therefore a “live” search.

```
return _context.Teams
    .Include(t => t.Players)
    .Where(t => String.IsNullOrEmpty(query) ? true : (t.Name.ToLower().Contains(query.ToLower())))
    .OrderBy(t => t.Name)
    .Select(x => new Team
    {
        Id = x.Id,
        Name = x.Name,
        Wins = x.Wins,
        Losses = x.Losses,
        Draws = x.Draws,
        MatchesPlayed = x.MatchesPlayed,
        GoalCount = x.GoalCount
    })
    .ToList();
```

In this code snippet, **query** is an optional parameter that defines the search term, if provided.

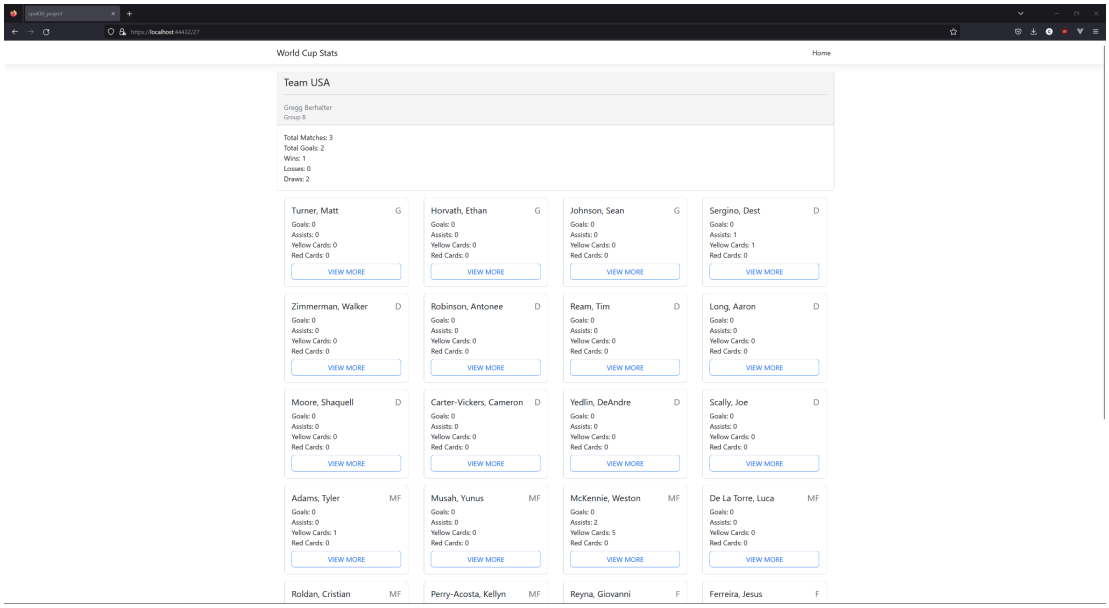


The next query is called “Exclusive Losers.” This pulls back any team that has not won or tied any matches.

```
return _context.Teams
    .Where(t => t.Wins == 0 && t.Draws == 0)
    .Select(x => new Team
    {
        Name = x.Name,
        GoalCount = x.GoalCount
    })
    .ToList();
```

After this query, we see the final query present on the home page, the “Winning Teams” query, which pulls back any team that has more wins than losses (draws can be ignored because they count for half on both sides of the equation). The query also orders the resulting teams first by their number of wins, and then by their goal count, to display the teams with the “best” records first.

```
return _context.Teams
    .Where(t => t.Wins > t.Losses)
    .OrderByDescending(t => t.Wins)
    .ThenByDescending(t => t.GoalCount)
    .Select(x => new Team
    {
        Name = x.Name,
        GoalCount = x.GoalCount,
        Wins = x.Wins
    })
    .ToList();
```



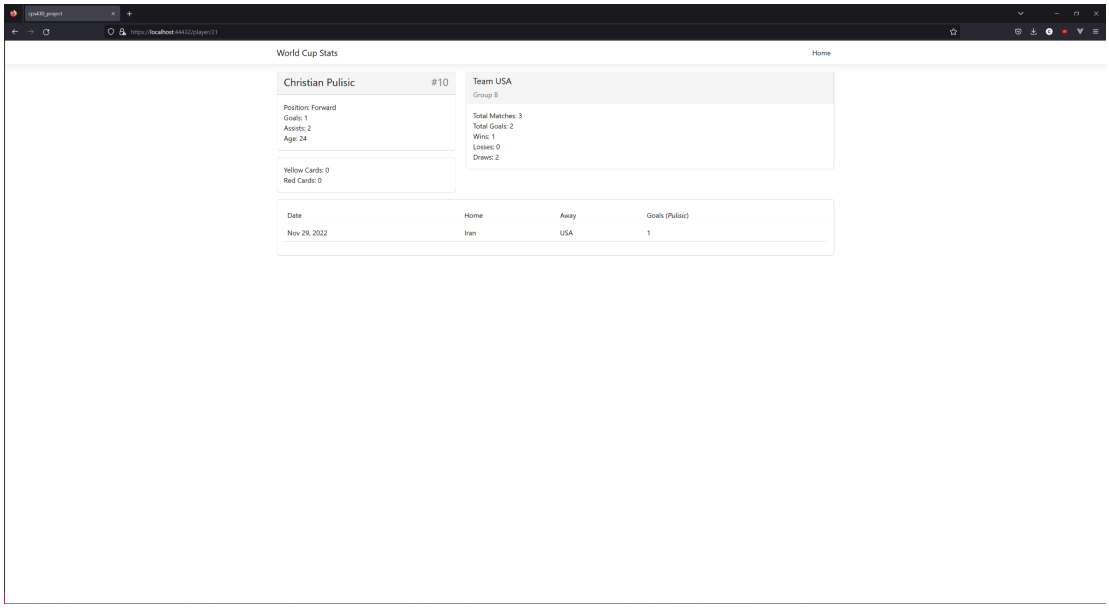
The “Team Dashboard” shows information about the team, and information about its players. There are two main queries here.

```
return _context.Teams
    .Include(x => x.Coaches)
    .FirstOrDefault(x => x.Id == id);
```

The preceding query takes an **id** as input, and finds the team with the matching id to pull back its information. The query also joins in the Coaches table to display information pertaining to that.

```
return _context.Players
    .Where(x => x.TeamId == id)
    .Include(p => p.Position)
    .ToList();
```

This query takes the same team id, and attempts to find all players who are on the specified team, and joins in the Position table to display information about the player’s role on the field.



The final query that appears in the website is the player query. This pulls back all information pertaining to a player, including joins to all relevant tables.

```
return _context.Players
    .Include(p => p.Team)
    .Include(p => p.PlayerMatches)
    .ThenInclude(p => p.Match)
    .Include("PlayerMatches.Match.HomeTeam")
    .Include("PlayerMatches.Match.AwayTeam")
    .Include(p => p.Position)
    .FirstOrDefault(p => p.Id == id);
```

The syntax for this query is a little different for brevity. Using LINQ to capture PlayerMatches.Match.Home/AwayTeam would have added several unnecessary lines to the query.

There were a few queries we did not have time to implement into the site, but that do exist and can be hit from the api.

The first of these is “GetScoringPlayers” which pulls back any player who has ever scored a goal.

```
return _context.Players
      .Include(p => p.PlayerMatches)
      .Where(p => p.PlayerMatches.Sum(x => x.GoalsScored) > 0)
      .ToList();
```

The final query is our most complicated, and finds the team that has the most conceded goals

```
return _context.Teams
     .Include(t => t.AwayMatches)
     .ThenInclude(t => t.PlayerMatches)
     .Include(t => t.HomeMatches)
     .ThenInclude(t => t.PlayerMatches)
     .OrderByDescending(t => t.AwayMatches
            .Sum(x => x.PlayerMatches
                  .Where(p => p.Player.TeamId != t.Id)
                  .Sum(y => y.GoalsScored)))
     .ThenByDescending(t => t.HomeMatches
            .Sum(x => x.PlayerMatches
                  .Where(p => p.Player.TeamId != t.Id)
                  .Sum(y => y.GoalsScored)))
     .FirstOrDefault();
```

VIII. Observations and Conclusions

- Design-time decisions can be heavily influenced by the frameworks chosen to build a project upon. In this case, the database was designed to be used from a website, and uses a Microsoft stack through the entire backend (SQL Server, EF Core, ASP.NET, and C#). These frameworks are opinionated, and while configurable, it is best to keep with their standards. Keeping with EF Core's naming guidelines allowed for it to do most of the "heavy lifting" when determining relationships among tables. The only time intervention was absolutely necessary, was when creating many-to-many relationships. Even then, explicitly defining an XRef table would have resolved this issue.
- Databases inherently store more data than might be needed for a particular view. To keep things clean, and to keep data as secure as possible, it is important to limit the amount of data being returned by a query. In other words, **select *** is not a good practice. It is important to pull out only the columns deemed necessary for the given query and its intended use case.