**Information Management II MySQL Project | Conor Evans | 16322861**

**Project Outline**

The database defined in this project aims to model a Fantasy Soccer League (FSL). It is not a novel concept, with many public FSLs, including an [official Barclays Premier League](https://fantasy.premierleague.com/) FSL. As the name would imply, these are not real soccer teams. However, they are based on real teams made up of real players and, by extension, fantasy leagues are based on real leagues.

As noted above, we will have to model real **League**sand **Team**sfrom which we will model **Fantasy Leagues** and **Fantasy Teams**. Every **Team** will have many **Player**s. However, we need not create a Fantasy Player table as fantasy players will always be a one-to-one match with real players and share all of their attributes (i.e. represent the same information). This is not the case for Fantasy Teams, which may be made up of Players from many Teams.

Each Fantasy Team will be managed by a **User** of the product and each User may have many fantasy teams. They can also enter each and any of these Fantasy Team in one/many Fantasy Leagues, with the proviso that the User does not already have a Fantasy Team participating in the given Fantasy League. Each Fantasy League will be administrated by a **User** who will have access and update certain attributes of the Fantasy League, a privilege that other users who may have a

Thus far we have not mentioned the model on which the product will ultimately be based: **Gameweek.** Leagues are ultimately competitions and Gameweek represents the metric that we will use to determine the League ranking. A Gameweek will reference a Team and will have many **Events**, which have an **Event Type** (e.g. goal, assist, red card, yellow card, minutes played). The event will be associated with a player of the Team associated with the Event’s Gameweek. These Events have a given points total as a function of the player’s **Position** – this is to reward rarer events, such as defenders scoring.

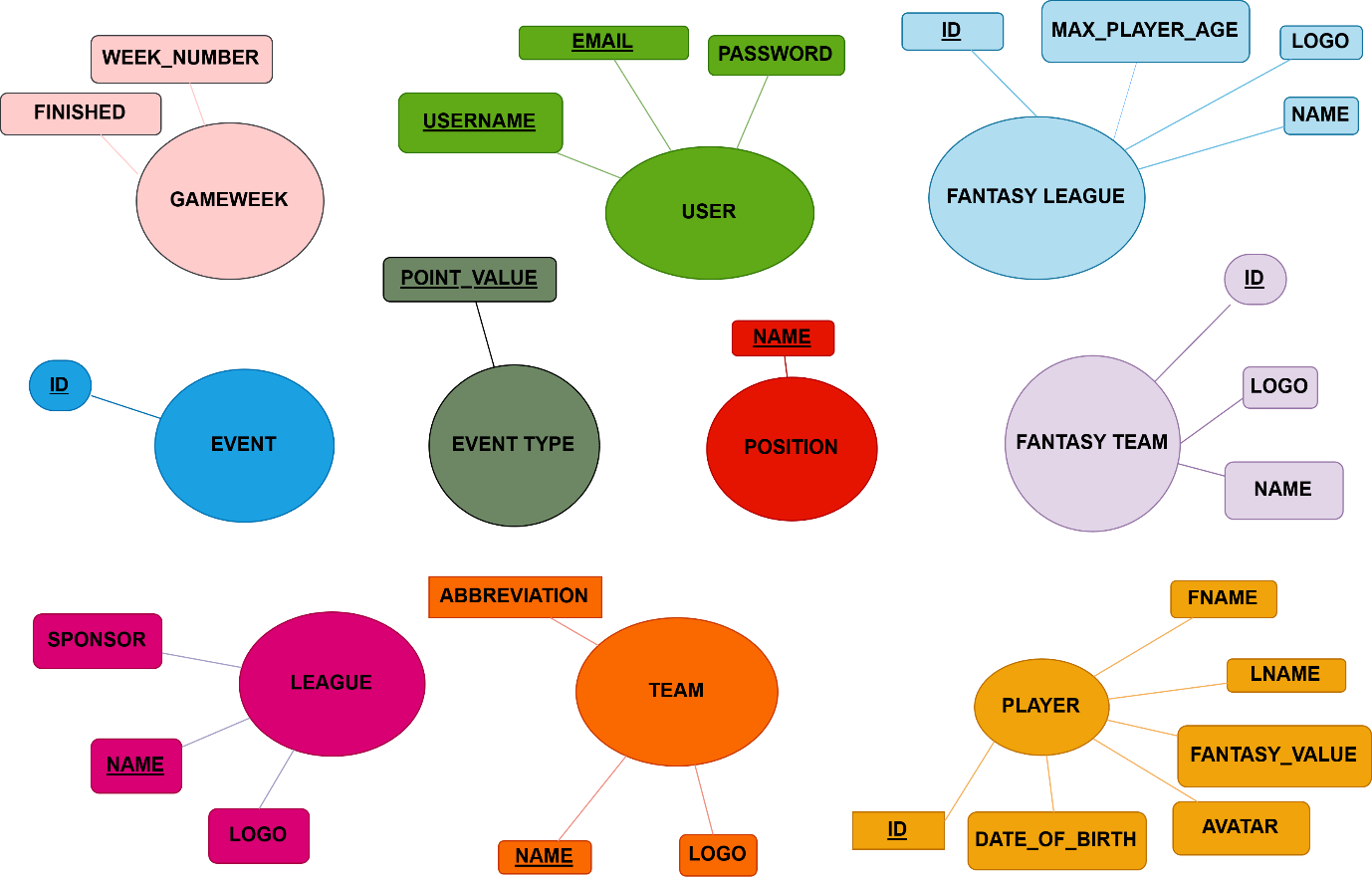
When the Gameweek is updated to *finished,* it will trigger SQL that will tally up all of the points related to Events provoked by Players of the Team associated with the Gameweek.

We thus have the following entities: League, Team, Player, Fantasy League, Fantasy Team, User, Gameweek, Event, Event Type, and Position. Given the N-M relations between Fantasy League and Fantasy Team, as well as between Fantasy Team and Player, we will have two additional tables with references to each table within the pair, i.e. **Fantasy League Fantasy Team** and **Fantasy Team Player**.

**Entity-Relation Model**

For the sake of readability, I have split my Entity-Relation Model into two sections. In Figure 1, each entity is depicted with all attributes related to the entity itself. In the second diagram, the relation between entities is outlined.

**Figure 1:**



Each entity’s candidate key(s) are underlined. It may be noted that **Gameweek** does not have a candidate key among its attributes. Its primary key will ultimately be a composite key made up of its *week\_number* attribute and the *id* attribute from Fantasy Team (i.e. a foreign key). This means that no Fantasy Team can have more than one Gameweek for a given *week\_number*, though many Fantasy Teams may have a Gameweek for a given *week\_number*. This could be avoided by extracting week\_number to a **Week** entity, with a Gameweek then representing this M to N relation with each element of the composite key being a foreign key, defined as follow:

*CREATE TABLE GAMEWEEK(*

*week\_number INT,*

*fantasy\_team\_id INT,*

*FOREIGN KEY(week\_number) REFERENCES week(number)*

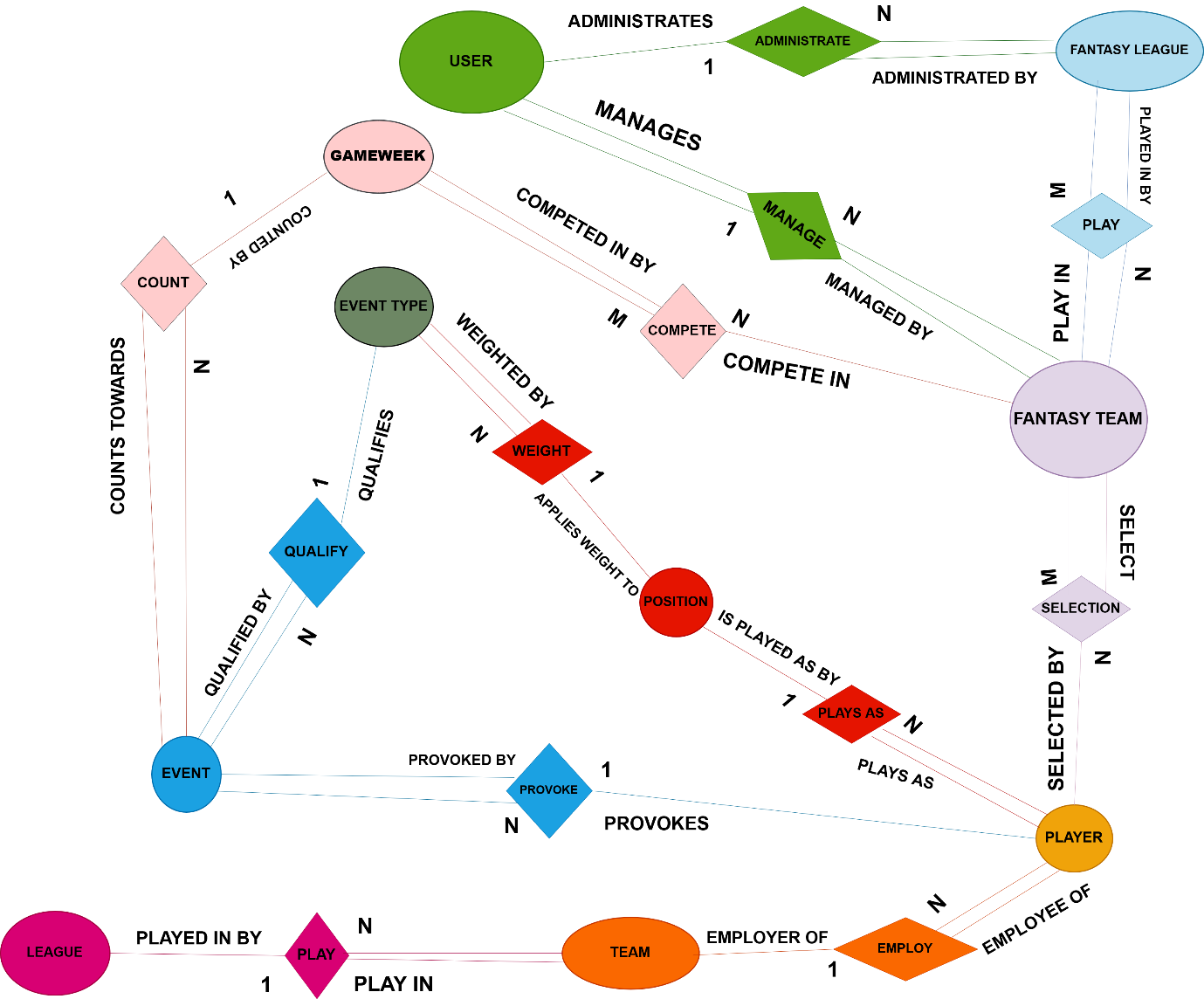
*FOREIGN KEY(fantasy\_team\_id) REFERENCES fantasy\_team(id),*

*PRIMARY KEY(week\_number, fantasy\_team\_id);*

*);*

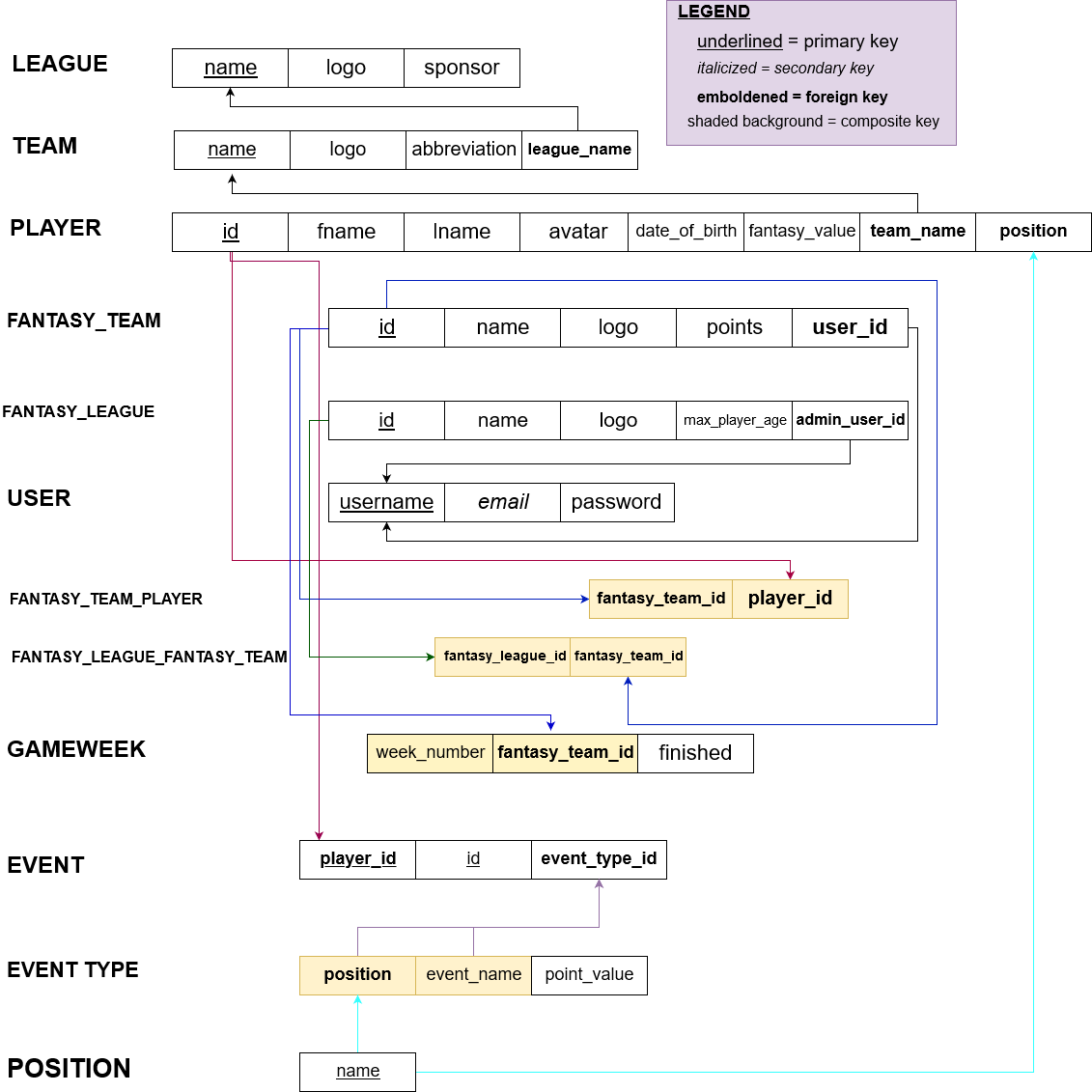
However, I decided against this as I felt it would be extraction for the sake of extraction.

**Figure 2:**



Certain relationships do not lend themselves particularly well to a natural-language verb.

**Relational Schema**



In designing my schema, I normalized Player by removing a *‘Position’* attribute and extracting it to its own table. I mainly did this because I added an event type and this would have caused an issue with data integrity as I would have had to keep a position attribute within Event Type, and I would have had to ensure that the *Position* attributeof the player whose ID attribute is stored in Event were consistent with the *Position* attribute in the Event Type whose ID attribute is stored in Event.

However, even if this were not the case, it would probably have been a good idea to extract this attribute regardless. In our database (and every real Fantasy League), there are only 4 options for Position: { ‘Goalkeeper’, ‘Defender’, ‘Midfielder’, ‘Forward’ }. There may be many thousands of Players stored in our database, and many of these records’ position attribute would have common information.

**Semantic Constraints**

There are many semantic constraints on the entities within the database. Some of these are accounted for through CHECKs and other cross-table constraints are implemented using triggers. For example, a Team’s abbreviation must be unique and exactly three characters. This constraint is imported from ‘industry’ standard across soccer, with all scoreboards following these abbreviations.

A User’s email must also be unique. This is just a characteristic of e-mails generally, though it prevents Users from creating multiple accounts. Rather, if they were no longer able to remember their details, we would instead send them through a standard Password Recovery process. It also means that no User could use another’s email, with a different username (primary key, thus allowed), and then try to access the original User’s password by claiming a password reset to a ‘secondary email’ or something along those lines. Though you would expect your customer service to have better security instincts, it’s always better to ensure data security at the source.

For user experience, there are several implicit constraints on the Fantasy Team Player (FTP) table. Firstly, we validate that for *any* of the Fantasy Leagues in which the given Fantasy Team participates, the maximum allowed player age (defined in Fantasy League as *max\_player\_age*) is less than the age of the Player who we are entering into the FTP table. Thus, as we have to compare data across two tables on inserting values into their relational table (i.e. the table created to store the MxN table containing foreign key references to the primary keys in the original two tables), this is implemented using a trigger *before insert*. As the triggers are, generally, longer than the remaining space on the pages allow, all triggers are stored in the following section titled *Triggers*. This is **T1.**  
Secondly, before insertion into Fantasy Team Player **(T2),** we also verify that the Fantasy Team to which we are associating the Player does not already three players from the Player’s Team. In plain English, a Fantasy Team may contain a maximum of three players from one Team.

Finally, we have several triggers when a Gameweek finishes to tally the points for all Events related to Fantasy Players associated to the Fantasy Team to which the Gameweek is associated. Firstly, we verify the validity *before insert* ***(T3)***. Here, if the *finished* were previously set to true and the user update attempts to set it to false, the user would be presented with the following error message: *You cannot ‘unfinish’ a gameweek*. Given that gameweek is our core model, we want to ensure that data manipulation is airtight. Allowing the gameweek to be ‘unfinished’ would mean that it could be ‘refinished’ which would then recalculate the points for **all** Fantasy Teams who participated in the Gameweek and thus update their point value across their Fantasy Leagues. This would be fairly disastrous.

However, if the *finished* were previously false and is set to true **(T4)**, it triggers a COUNT of all of the Events associated to Players associated to Fantasy Teams who have participated in the given gameweek. The *points\_total* of all of these Fantasy Teams are then updated to reflect the new total (old\_total += EVENT\_POINTS\_COUNT).

**Triggers:**

**T1: verify player age does not exceed max allowed by fantasy league of its fantasy team**

delimiter //

CREATE TRIGGER player\_age

BEFORE INSERT ON fantasy\_team\_player

FOR EACH ROW

BEGIN

DECLARE max\_age INT;

DECLARE player\_dob DATE;

DECLARE error\_msg varchar(255);

SELECT MIN(max\_player\_age)

INTO max\_age

FROM fantasy\_league as fl

JOIN fantasy\_league\_fantasy\_team as flft

ON flft.fantasy\_team\_id = new.fantasy\_team\_id;

SELECT date\_of\_birth

INTO player\_dob

FROM player

WHERE player.id = new.player\_id;

IF ((max\_age \* 365) < DATEDIFF(CURDATE(),player\_dob)) THEN

SET error\_msg = "You cannot add this player to your team because

he is older than the maximum player age the league you are entering allows.";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = error\_msg;

END IF;

END;//

**T3: Gameweek counted at most once**

delimiter //

CREATE TRIGGER gameweek\_counted\_only\_once

BEFORE UPDATE ON gameweek

FOR EACH ROW

BEGIN

DECLARE error\_msg varchar(255);

IF(old.finished = 'true' AND new.finished = 'false') THEN

SET error\_msg = "You cannot 'unfinish' a gameweek.";

SIGNAL SQLSTATE '45000' SET MESSAGE\_TEXT = error\_msg;

END IF;

END;//