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 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**s and **Team**s from which we will model **Fantasy League**s and **Fantasy Team**s. 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. 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 **Event**s, 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 Fantasy 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.

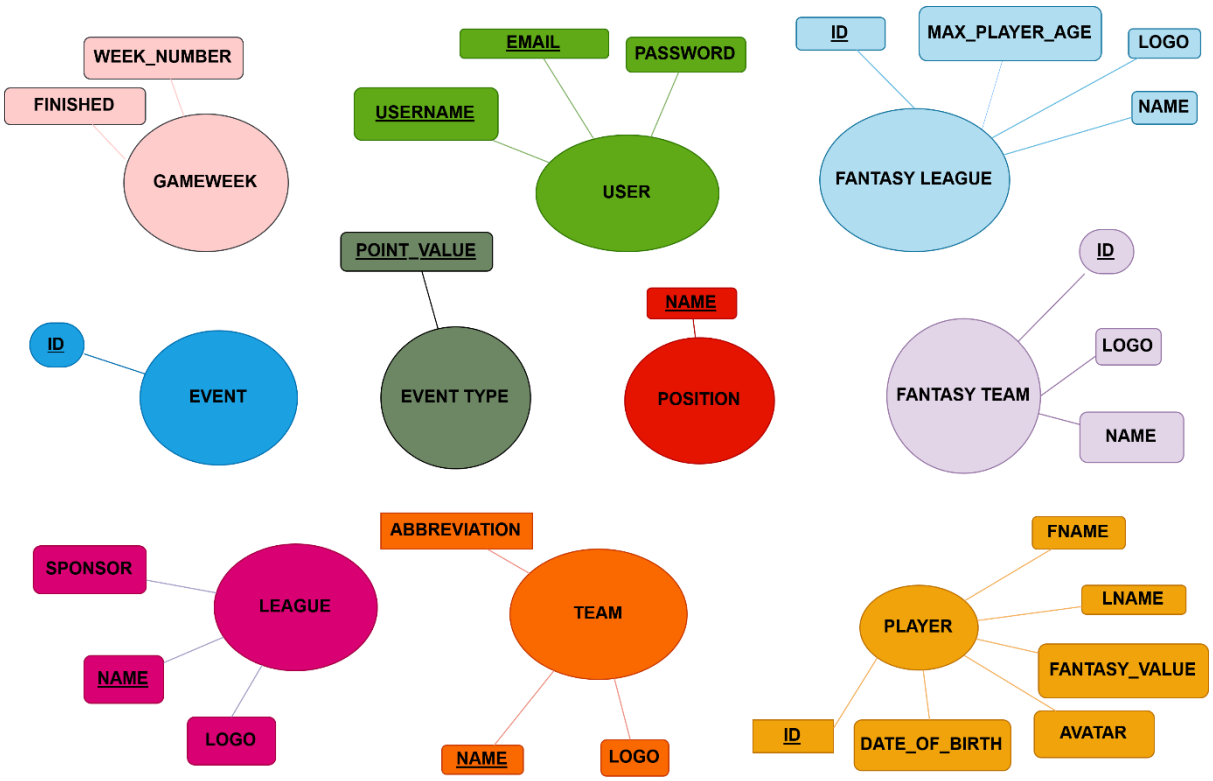
When the Gameweek is updated to *finished,* it will trigger SQL that will tally up all of the points related to Events provoked by Players of the Team associated with the Gameweek.

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:

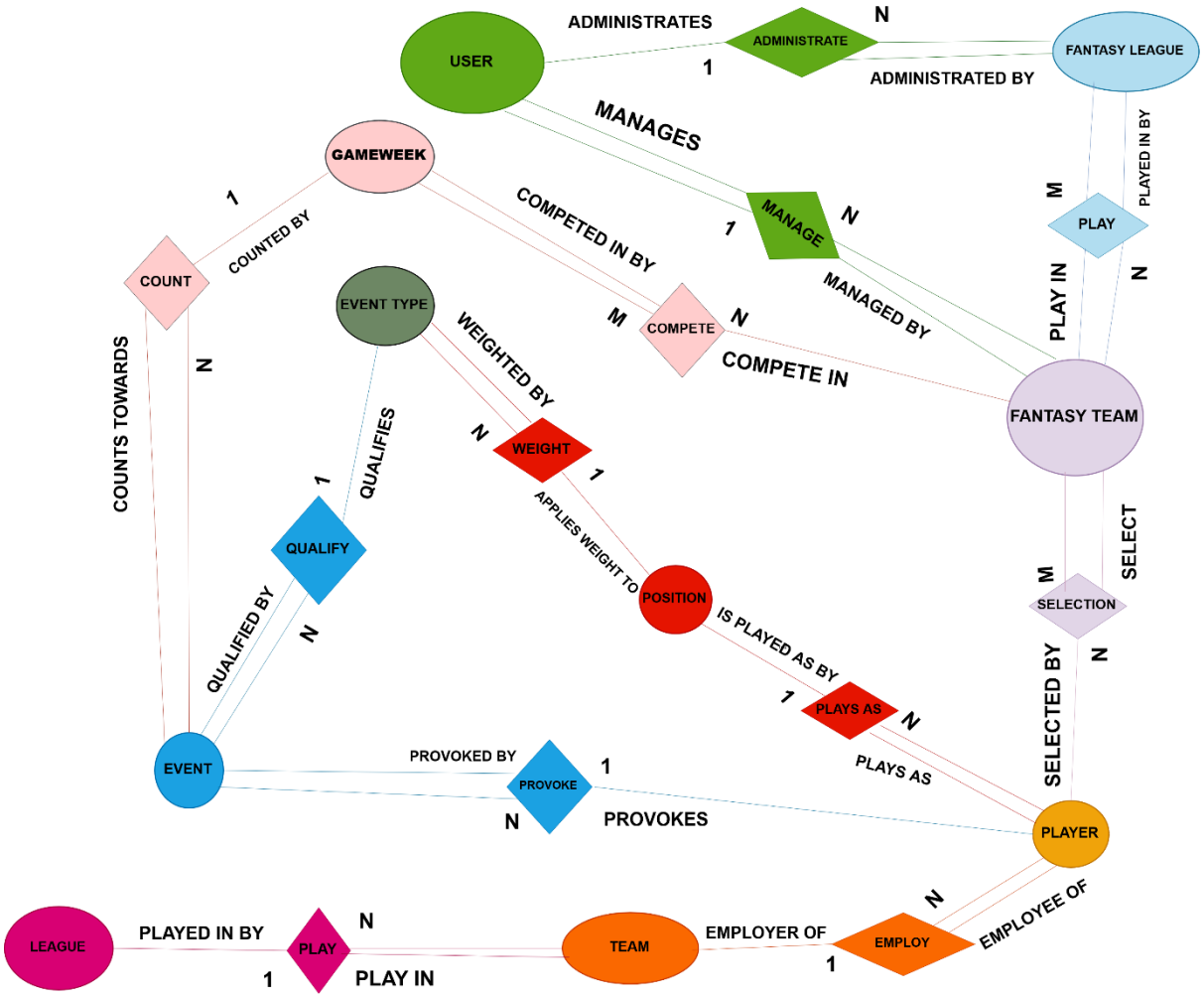


Each entity's candidate key(s) are underlined. It may be noted that **Gameweek** does not have a candidate key among its attributes. Its primary key will ultimately be a composite key made up of its *week_number* attribute and the *id* attribute from Fantasy Team (i.e. a foreign key). This means that 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);
);
```

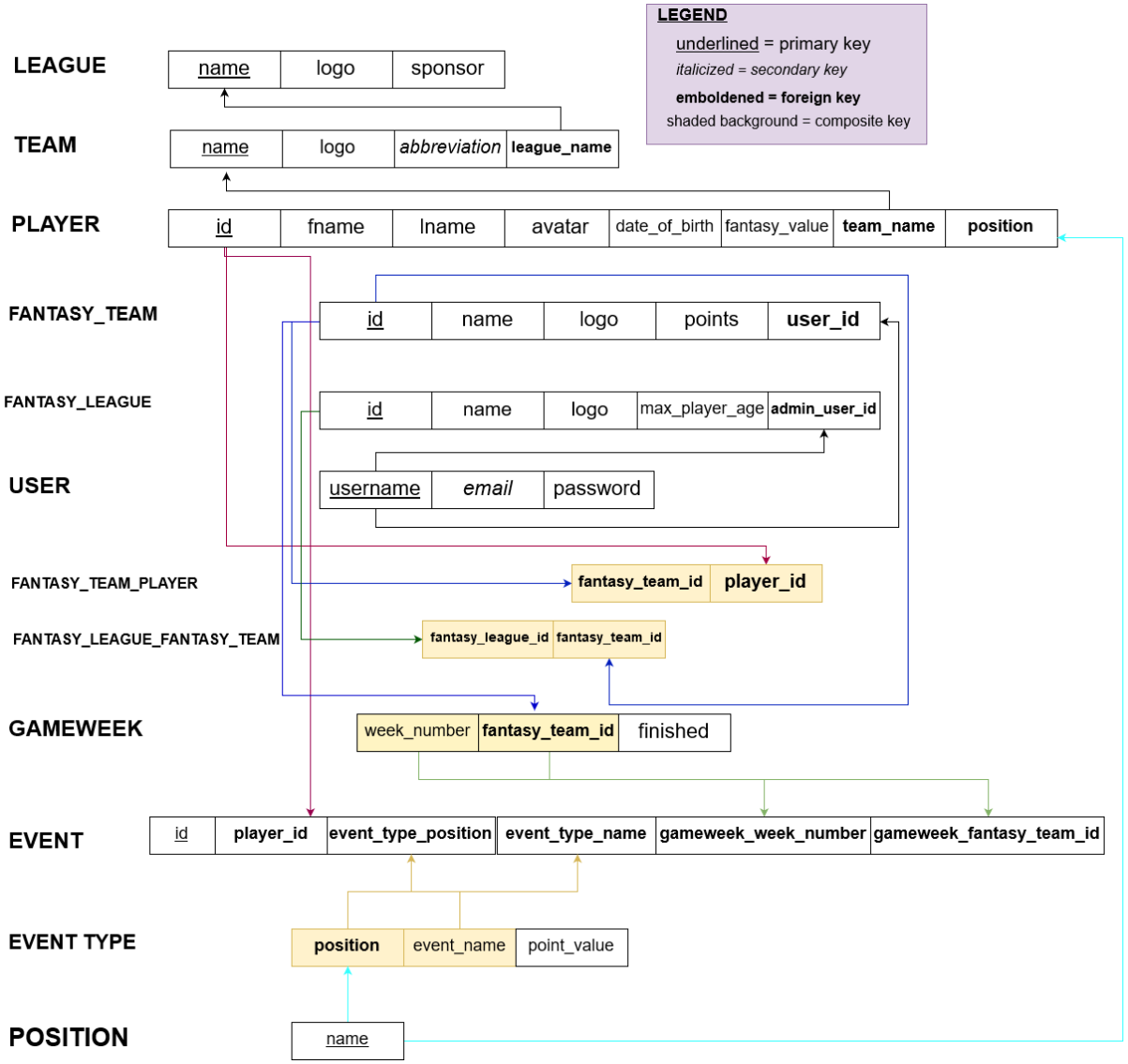
However, I decided against this as I felt it would be extraction for the sake of extraction.

Figure 2:



Certain relationships do not lend themselves particularly well to a natural-language verb, but generally the relationships should form a natural sentence.

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* attribute of 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.

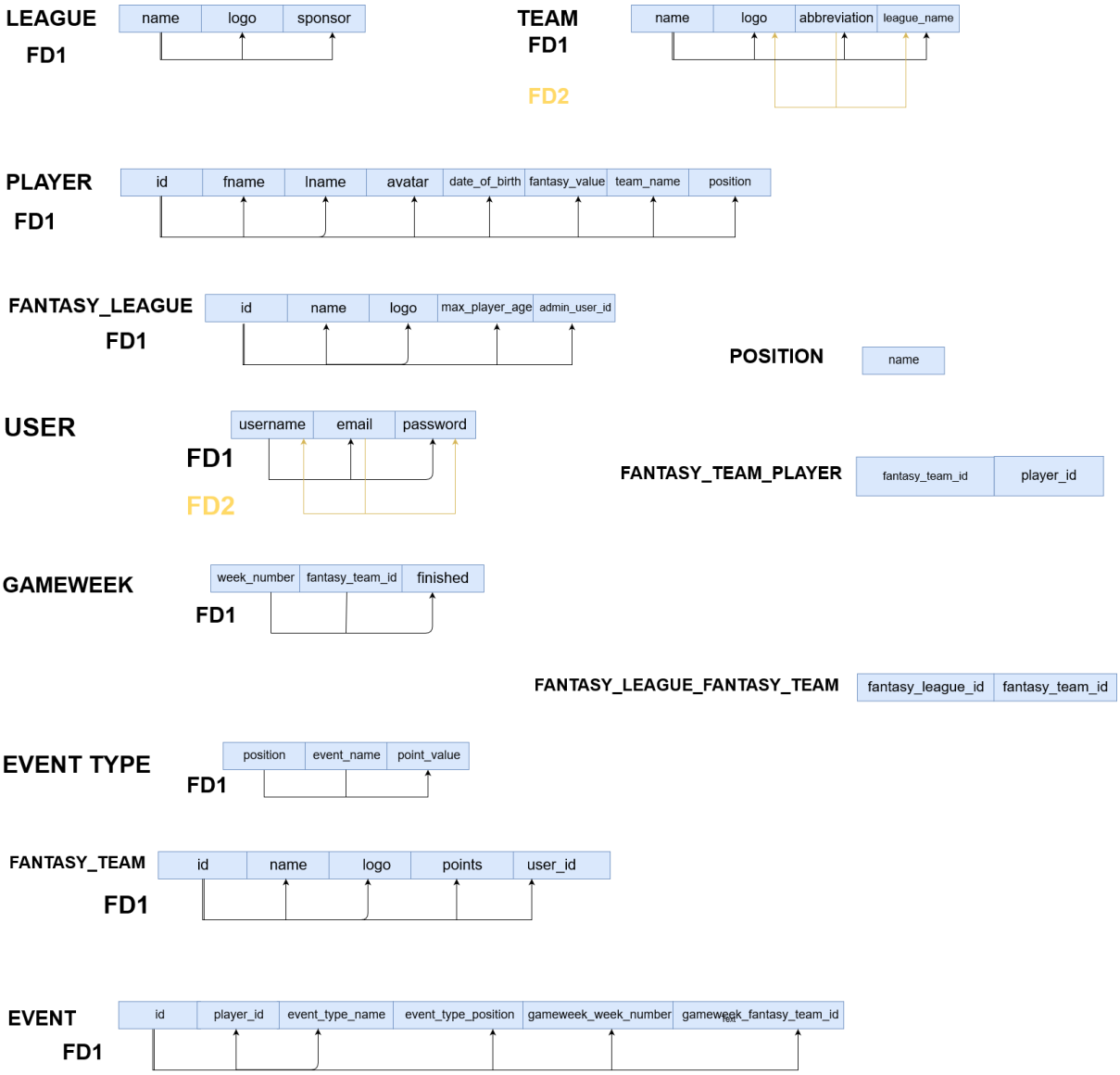
With regard to attribute names, I personally disagree with avoiding generic attribute names like `name`. Unless there is an alternative that is preferable and is naturally linguistically sound, I do not think it is preferable to call the *name* attribute of Position, for example,

position_name. There is no point in reinventing the wheel. Furthermore, using similar variable names for similar ideas, e.g. *id* or *name*, means that a developer does not have to be consistently trying to recall what unique slightly-unusual name was chosen as the primary key attribute for the sake of avoiding typical variable names. One could also argue that you should always qualify your attributes, be they in the project part of a select statement or in the 'lower' part of the statement i.e:

```
SELECT p.name, pos.name

FROM player AS p

JOIN position AS pos

ON p.position = pos.name;
```

Functional Dependency

I then drew up my functional dependency diagram, to ensure that I had not missed anything which does not adhere to Boyce-Codd 3rd Normal Form:



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.

With these explicit CHECK constraints in mind, it was time to define the entities which would make up the database. Other implicit semantic constraints are present, but these will be defined later in the *Triggers* section as we need to create tables before we can look into implementing triggers.

Table Creation

Due to foreign key constraints, the tables in the database must be defined in an order that allows these dependencies to be defined on table creation. Thus, a table with a foreign key referring to the primary key of another table cannot be defined before the other table has been defined.

Only three tables in the database do not have a foreign key: League, User, and Position. The latter two need not be defined before we get further along, so League will be the first table created.

```
CREATE TABLE league(
    name varchar(255) NOT NULL,
    sponsor varchar(255),
    logo BLOB,
    PRIMARY KEY(name)
);
```

Due to personal preference, I will always define the PRIMARY KEY after the attributes and FOREIGN KEY references even if the primary key is not composite and thus could be placed after the primary key attribute ('name', in this case). Additionally, the use of the attribute 'name' will be touched on in the final overview of the project.

Next up was the Team entity:

```
CREATE TABLE team(
  name varchar(255) NOT NULL,
  logo BLOB,
  league_name varchar(255) NOT NULL,
  FOREIGN KEY (league_name) REFERENCES league(name),
  PRIMARY KEY(name)
);
```

As stated earlier, the extraction of position from being an attribute of player to being its own entity was done after defining the relational schema. However, it was done before creating tables so I was aware that Position would need to be defined before Player:

```
CREATE TABLE `position`(
    `name` varchar(255) NOT NULL PRIMARY KEY
);
```

Player table is defined as follows:

```
CREATE TABLE player(
  id INT NOT NULL AUTO_INCREMENT,
  date_of_birth DATE NOT NULL,
  avatar BLOB,
  fantasy_value FLOAT NOT NULL,
  fname VARCHAR(255) NOT NULL,
  lname VARCHAR(255) NOT NULL,
  position VARCHAR(255) NOT NULL,
  team_name VARCHAR(255) NOT NULL,
  FOREIGN KEY(team_name) REFERENCES team(name),
  FOREIGN KEY(position) REFERENCES `position`(name),
  PRIMARY KEY(id)
);
```

Once we had the core real-life entities on which our fantasy soccer entities would be based, it became possible to define the latter. First, however, we would need to define the Users who would be associated to these Fantasy Teams and Fantasy Leagues in a manager/administrator role respectively.

My expectation for defining user passwords, an extremely sensitive data item, was validated on research. The password attribute stored in the database will simply be a string. The encryption/salting takes place between intaking a user password and storing it in the database. Thus, the password attribute is a hashed-string, but a string nonetheless, so we will define it as an attribute of type *varchar*:

```
CREATE TABLE `user`(
   username varchar(255) NOT NULL,
   email varchar(255) NOT NULL UNIQUE,
   `password` varchar(255) NOT NULL,
   PRIMARY KEY(username)
);
```

We see the explicit UNIQUE semantic constraint for the secondary-key *email*. It's also worth noting that defining a *user* table within a database does not overwrite the global *mysql.user* table. Similarly, running *SELECT * FROM `user`;* will return all users defined within the user table in the schema of the database in which the query is executed. This will be touched on more in the *Views* section of the report.

With our user defined, we could now build the fantasy entities.

Fantasy League:

```
CREATE TABLE fantasy_league(
    id INT NOT NULL AUTO_INCREMENT,
    `name` varchar(255) NOT NULL,
    max_player_age INT,
    logo BLOB,
    admin_user_id varchar(255) NOT NULL,
    FOREIGN KEY(admin_user_id) REFERENCES `user`(username),
    PRIMARY KEY(id)
);
```

Fantasy Team:

```
CREATE TABLE fantasy_team(
    id INT NOT NULL AUTO_INCREMENT,
    `name` varchar(255) NOT NULL,
    logo BLOB,
    user_id varchar(255) NOT NULL,
    FOREIGN KEY(user_id) REFERENCES user(username),
    PRIMARY KEY(id)
);
```

With our fantasy teams defined, we can define the core Gameweek model which will be used to rank fantasy teams within their fantasy league, and globally across fantasy leagues:

```
CREATE TABLE gameweek(
    week_number INT NOT NULL,
    fantasy_team_id INT NOT NULL,
    CONSTRAINT valid_week_number CHECK(week_number BETWEEN 1 AND 38),
    FOREIGN KEY (fantasy_team_id) REFERENCES fantasy_team(id),
    PRIMARY KEY (week_number, fantasy_team_id)
);
```

The explicit week number constraint is based off the English Barclays Premier League.

We could then associate Events to a given Gameweek, but as these Events are qualified by an Event Type, it is the latter that we define next:

```
CREATE TABLE event_type(
    event_name varchar(255) NOT NULL,
    position varchar(255) NOT NULL,
    point_value INT NOT NULL,
    FOREIGN KEY (position) REFERENCES `position`(name),
    PRIMARY KEY (event_name, position)
);
```

Event:

```
CREATE TABLE event(
  id INT NOT NULL AUTO_INCREMENT,
   event_type_name varchar(255) NOT NULL,
   event_type_position varchar(255) NOT NULL,
   player_id INT NOT NULL,
   gameweek_id INT NOT NULL,
   FOREIGN KEY(event_type_name, event_type_position)
   REFERENCES `event_type`(event_name, position),
   FOREIGN KEY(player_id) REFERENCES player(id),
   PRIMARY KEY(id)
);
```

Now that all of the tables have been defined, we must take care to define the tables that will retain information relating to our M-N relations, of which there are two:

```
CREATE TABLE fantasy_team_player(
   fantasy_team_id INT NOT NULL,
   player_id INT NOT NULL,
   FOREIGN KEY(fantasy_team_id) REFERENCES fantasy_team(id),
   FOREIGN KEY(player_id) REFERENCES player(id),
   PRIMARY KEY(fantasy_team_id, player_id)
);


CREATE TABLE fantasy_league_fantasy_team(
   fantasy_league_id INT NOT NULL,
   fantasy_team_id INT NOT NULL,
   FOREIGN KEY(fantasy_team_id) REFERENCES fantasy_team(id),
   FOREIGN KEY(fantasy_league_id) REFERENCES fantasy_league(id),
   PRIMARY KEY(fantasy_league_id, fantasy_team_id)
);
```

Inserting values:

With our tables defined, we now insert several tuples for each relational entity. Due to referential integrity constraints, we must insert values in a way that does not involve us inserting NON-NULL foreign keys where the tuple of the relation identified by this key in the referenced relation already exists. Simply put, a foreign key must exist as a primary key in its table before it can be referenced by a tuple in another relation.

League:

Values = {Name, Sponsor, Logo}

```
INSERT INTO league VALUES('Premier League','Barclays',NULL);
```

```
INSERT INTO league VALUES('Ligue 1', 'Conforama', NULL);
```

```
INSERT INTO league VALUES('Bundesliga', NULL, NULL);
```

```
INSERT INTO league VALUES('EFL Championship', 'Skybet', NULL);
```

```
INSERT INTO league VALUES('Serie A', 'TIM', NULL);
```

```
INSERT INTO league VALUES('League of Ireland', NULL, NULL);
```

You might notice the logo (BLOB) attribute is always NULL. This will be dealt with in the *Update* section as I was unsure of inserting images into MySQL at the time of definition.

Team:

Values = {Name, Logo, League_Name, Abbreviation}

```
INSERT INTO team VALUES('Arsenal', NULL, 'Premier League');
```

```
INSERT INTO team VALUES('Liverpool', NULL, 'Premier League', 'LIV');
```

```
INSERT INTO team VALUES('Leicester City', NULL, 'Premier League', 'LEI');
```

```
INSERT INTO team VALUES('Chelsea', NULL, 'Premier League','CHE');
```

```
INSERT INTO team VALUES('Manchester City', NULL, 'Premier League', 'MCI');
```

```
INSERT INTO team VALUES('Sheffield United', NULL, 'Premier League', 'SHU');
```

```
INSERT INTO team VALUES('Manchester United', NULL, 'Premier League', 'MUN');
```

```
INSERT INTO team VALUES('Wolverhampton Wanderers', NULL, 'Premier League', 'WOL');
```

```
INSERT INTO team VALUES('Bournemouth', NULL, 'Premier League', 'BOU'); WHITESPACE
```

```
INSERT INTO team VALUES('Burnley', NULL, 'Premier League', 'BUR');
```

```
INSERT INTO team VALUES('Brighton and Hove Albion', NULL, 'Premier League', 'BHA');
```

```
INSERT INTO team VALUES('Crystal Palace', NULL, 'Premier League', 'CRY');
```

```
INSERT INTO team VALUES('Newcastle United', NULL, 'Premier League', 'NEW');
```

INSERT INTO team VALUES('Tottenham Hotspur', NULL, 'Premier League', 'TOT');

INSERT INTO team VALUES('Everton', NULL, 'Premier League', 'EVE');

INSERT INTO team VALUES('West Ham United', NULL, 'Premier League', 'WHU');

INSERT INTO team VALUES('Aston Villa', NULL, 'Premier League', 'AVL');

INSERT INTO team VALUES('Watford', NULL, 'Premier League', 'WAT');

INSERT INTO team VALUES('Southampton', NULL, 'Premier League', 'SOU');

INSERT INTO team VALUES('Norwich City', NULL, 'Premier League', 'NOR');

You may notice that Arsenal has a different arity (it does not have an abbreviation). This row was inserted before the table was altered to add the abbreviation attribute. This will be dealt with in the *Alter Table* and *Update* sections.

Position:

Values = {Name}

INSERT INTO `position` VALUES('Goalkeeper');

INSERT INTO `position` VALUES('Defender');

INSERT INTO `position` VALUES('Midfielder');

INSERT INTO `position` VALUES('Forward');

Player:

Values = {ID, DateOfBirth, Avatar, FantasyValue, FName, LName, Position, TeamName}

INSERT INTO player VALUES(NULL,'1990-04-12', NULL, 4.25, 'Conor', 'Heffernan', 'Goalkeeper', 'Arsenal');

INSERT INTO player VALUES(NULL,'1985-07-04', NULL, 7.45, 'Chris', 'Henderson', 'Goalkeeper', ‘Chelsea');

INSERT INTO player VALUES(NULL,'1996-03-11', NULL, 3.95, 'Jonah', 'Rafferty', 'Defender', 'Arsenal');

INSERT INTO player VALUES(NULL,'1992-04-24', NULL, 5.25, 'Gilberto', 'Da Silva', 'Defender', ‘Chelsea');

INSERT INTO player VALUES(NULL,'1995-09-28', NULL, 4.25, 'Albert', 'Tierney', 'Defender', 'Arsenal');

INSERT INTO player VALUES(NULL,'1983-08-03', NULL, 7.45, 'Alejandro', 'Lopez', 'Defender', 'Arsenal');

INSERT INTO player VALUES(NULL,'1988-09-01', NULL, 3.95, 'Gerard', 'Roe', 'Defender', ‘Bournemouth'); **FAIL**

INSERT INTO player VALUES(NULL,'1981-07-22', NULL, 5.25, 'Christian', 'Bale', 'Midfielder', 'Arsenal');

INSERT INTO player VALUES(NULL,'1987-08-19', NULL, 4.25, 'Manuel', 'Herrera', 'Midfielder', ‘Leicester');

INSERT INTO player VALUES(NULL,'1980-06-10', NULL, 7.45, 'Jay', 'Pritchett', 'Midfielder', 'Arsenal');

INSERT INTO player VALUES(NULL,'1999-05-13', NULL, 3.95, 'James', 'Burke', 'Midfielder', ‘Manchester United');

INSERT INTO player VALUES(NULL,'2000-03-13', NULL, 5.25, 'Vladimir', 'Rubins', 'Midfielder', ‘Sheffield United');

INSERT INTO player VALUES(NULL,'1999-02-22', NULL, 4.25, 'Aleksandr', 'Nemenko', 'Forward', ’Everton’);

INSERT INTO player VALUES(NULL,'1993-01-23', NULL, 7.45, 'Eric', 'Bailly', 'Forward', ‘Tottenham Hotspur');

INSERT INTO player VALUES(NULL,'1992-12-31', NULL, 3.95, 'Malcolm', 'Barry', 'Forward', 'Arsenal');

INSERT INTO player VALUES(NULL,'1992-05-01', NULL, 5.25, 'Kaka', 'McAllister', 'Forward', 'Arsenal');

While inserting my players, one failed (noted by FAIL) due to a referential integrity error:

‘Error 1452: Cannot update or add a child row: a foreign key constraint fails’

This was because of an erroneous whitespace (background color in red and WHITESPACE written to the side of the insertion in *Team*) in a Team insertion. Per MySQL documentation:

`VARCHAR` values are not padded when they are stored. Trailing spaces are retained when values are stored and retrieved, in conformance with standard SQL.

This is fixed in the *Update* section.

Note also that the *id* primary key attribute is always inserted as NULL. This is because MySQL, through the schema, is aware of the AUTO_INCREMENT it is to carry out. Again, the *avatar* (photo) attributes are all NULL. This was to prevent me having to continually download photos or just use one constant value throughout. If we were expecting many values to be NULL, it might be wiser instead to extract it to a *PlayerAvatar* table which we could refer to when needed.

User:

Values = {Username, Email, Password}

INSERT INTO fantasy_soccer.`user` VALUES('johnboy','john@example.com','Hunter123');

INSERT INTO fantasy_soccer.`user` VALUES('jangirl','jan@example.com','Hunter123');

```
INSERT INTO fantasy_soccer.`user` VALUES('bobbo123','bob@example.com','Hunter123');

INSERT INTO fantasy_soccer.`user` VALUES('hotrod','rod@example.com','Hunter123');

INSERT INTO fantasy_soccer.`user` VALUES('gerardo','ger@example.com','Hunter123');

INSERT INTO fantasy_soccer.`user` VALUES('gonzo78','gonzalo@example.com','Hunter123');
```

Fantasy League:

Values = {ID, Name, MaxPlayerAge, Logo, AdminUserID}

```
INSERT INTO fantasy_league VALUES(NULL,'The Lads',NULL, NULL,'johnboy');

INSERT INTO fantasy_league VALUES(NULL,'The Lads: The SQL',NULL, NULL,'jangirl');

INSERT INTO fantasy_league VALUES(NULL,'Das Lads', NULL, NULL, 'bobbo123');

INSERT INTO fantasy_league VALUES(NULL,'Les Lads', NULL, NULL,'hotrod');

INSERT INTO fantasy_league VALUES(NULL,'Na leaidz', NULL, NULL,'gerardo');

INSERT INTO fantasy_league VALUES(NULL,'fantasy league', 30, NULL,'gonzo78');
```

The same comments with regard to AUTO_INCREMENT and the NULL values for the photo/logo attribute apply here. Additionally, we could once more extract *max_player_age* to a separate entity if we expected there to be a lot of Fantasy Leagues (which would be the case in the real product) but did not expect many of these FLs to have a *max_player_age* attribute. We could then use an EXISTS in any triggers where we may want to see if Players are too old to be on a Fantasy Team of a Fantasy League which is referred to by a tuple in this entity.

Fantasy Team:

Values = {ID, Name, Logo, UserID}

```
INSERT INTO fantasy_team VALUES(NULL,'Queen Park Strangers',NULL,'johnboy');

INSERT INTO fantasy_team VALUES(NULL,'Real Sociopath', NULL,'jangirl');

INSERT INTO fantasy_team VALUES(NULL,'Kings of Lyon', NULL, 'bobbo123');

INSERT INTO fantasy_team VALUES(NULL,'Hangover 96', NULL,'hotrod');

INSERT INTO fantasy_team VALUES(NULL,'Tea and Busquets', NULL,'gerardo');

INSERT INTO fantasy_team VALUES(NULL,'Robben you Blind', NULL,'gonzo78');

INSERT INTO fantasy_team VALUES(NULL,'Fake Madrid', NULL,'johnboy', 0);

INSERT INTO fantasy_team VALUES(NULL,'PSG', NULL,'johnboy', 0);
```

Again, there is an arity mismatch between the first six tuples and the final two. This is because the final two were inserted after altering the table to add a column. This is dealt with in *Altering Tables* and *Update*.

Fantasy League Fantasy Team

Values = {FantasyLeagueID, FantasyTeamID}

```
INSERT INTO fantasy_league_fantasy_team VALUES(6,1);
INSERT INTO fantasy_league_fantasy_team VALUES(6,2);
INSERT INTO fantasy_league_fantasy_team VALUES(6,3);
INSERT INTO fantasy_league_fantasy_team VALUES(6,4);
INSERT INTO fantasy_league_fantasy_team VALUES(6,5);
INSERT INTO fantasy_league_fantasy_team VALUES(5,1);
INSERT INTO fantasy_league_fantasy_team VALUES(5,2);
INSERT INTO fantasy_league_fantasy_team VALUES(5,3);
```

Fantasy Team Player:

Values = {FantasyTeamID, FantasyPlayerID}

```
INSERT INTO fantasy_team_player VALUES(6,1);
INSERT INTO fantasy_team_player VALUES(6,3);
INSERT INTO fantasy_team_player VALUES(6,5);
INSERT INTO fantasy_team_player VALUES(6,8);
INSERT INTO fantasy_team_player VALUES(6,11);
INSERT INTO fantasy_team_player VALUES(6,12);
INSERT INTO fantasy_team_player VALUES(6,14);
```

It is worth noting with respect to the previous two tables that these sort of insertions would likely be dealt in a transaction consisting of X Fantasy Team Players being inserted, before saving this Fantasy Team and assuring persistence of the record. In practice, we would have no interest in storing Fantasy Team Players for a Fantasy Team that is not valid, i.e. we wouldn't insert each 11 players individually. Instead, the 11 insertions would be run in a transaction and if they were all successful, we would then ensure the Fantasy Team to who they would refer had persisted.

Gameweek:

Values = {WeekNumber, FantasyTeamID, Finished}

```
INSERT INTO gameweek VALUES(3,1,'false');
INSERT INTO gameweek VALUES(4,1,'false');
INSERT INTO gameweek VALUES(5,1,'false');
```

```
INSERT INTO gameweek VALUES(6,1,'false');

INSERT INTO gameweek VALUES(7,1,'false');

INSERT INTO gameweek VALUES(8,1,'false');
```

Event Type:

Values = {EventName, Position, PointsValue}

```
INSERT INTO event_type VALUES('clean_sheet','Goalkeeper',4);

INSERT INTO event_type VALUES('clean_sheet','Defender',4);

INSERT INTO event_type VALUES('clean_sheet','Midfielder',1);

INSERT INTO event_type VALUES('red_card','Goalkeeper',-2);

INSERT INTO event_type VALUES('red_card','Defender',-2);

INSERT INTO event_type VALUES('yellow_card','Goalkeeper',-1);

INSERT INTO event_type VALUES('goal','Goalkeeper',8);

INSERT INTO event_type VALUES('goal','Defender',6);

INSERT INTO event_type VALUES('assist','Midfielder', 2);

INSERT INTO event_type VALUES('assist','Forward',2);
```

Event:

Values = {ID, EventTypeEventName, EventTypePosition, PlayerID,

GameweekWeekNumber, GameweekFantasyTeamID}

```
INSERT INTO `event` VALUES(NULL, 'red_card', 'goalkeeper', 14, 1, 1);

INSERT INTO `event` VALUES(NULL, 'red_card', 'goalkeeper', 14, 1, 1);

INSERT INTO `event` VALUES(NULL, 'red_card', 'goalkeeper', 14, 2, 6);

INSERT INTO `event` VALUES(NULL, 'red_card', 'goalkeeper', 14, 1, 6);

INSERT INTO `event` VALUES(NULL, 'goal', 'goalkeeper', 14, 1, 6);

INSERT INTO `event` VALUES(NULL, 'yellow_card', 'goalkeeper', 14, 2, 6);

INSERT INTO `event` VALUES(NULL, 'red_card', 'goalkeeper', 14, 1, 4);

INSERT INTO `event` VALUES(NULL, 'red_card', 'goalkeeper', 14, 1, 1);
```

Altering Tables:

Although preferable to avoid entirely, it may be necessary sometimes to add attributes or constraints to tables, be they related to new features being modelled by the database or constraints that were missed on table creation. Below are some of the additional attributes or constraints that were defined on the above tables after they had been defined in the schema:

```
ALTER TABLE fantasy_team

ADD COLUMN points_total INT DEFAULT(0);
```

I'm not entirely sure how default values are inserted in SQL, or at least in MySQL. As seen below, if we allow NULL values to be inserted (test), we must either insert NULL (which should then set the attribute to its default) or a value. We cannot leave it blank otherwise MySQL throws an error due to a column cardinality mismatch between the schema and the attempted query. However, inserting NULL sets the attribute's value to NULL (allowed), rather than 0, which should be the default on NULL insertion into an attribute whose default value has been defined. If we say that it cannot be NULL (test_2), then evidently we must insert a value and the default is thus useless.

Creation + Insertion:

```
CREATE TABLE test(
    id INT NOT NULL auto_increment PRIMARY KEY,
    num INT DEFAULT(0)
);

CREATE TABLE test_2(
    id INT NOT NULL auto_increment PRIMARY KEY,
    num INT NOT NULL DEFAULT(0)
);

INSERT INTO test VALUES(NULL,NULL);
INSERT INTO test_2 VALUES(NULL,NULL);
```

Error on inserting NULL into NOT NULL attribute (to be expected):

✓	8	15:43:18	CREATE TABLE test(id INT NOT NULL auto_increment PRIMARY KEY, num INT DEFAULT(0))	0 row(s) affected	0.032 sec
✓	9	15:43:18	CREATE TABLE test_2(id INT NOT NULL auto_increment PRIMARY KEY, num INT NOT NUL...	0 row(s) affected	0.015 sec
✓	10	15:43:18	INSERT INTO test VALUES(NULL,NULL)	1 row(s) affected	0.016 sec
✗	11	15:43:18	INSERT INTO test_2 VALUES(NULL,NULL)	Error Code: 1048. Column 'num' cannot be null	0.000 sec

NULL attribute not set to DEFAULT defined in ALTER TABLE ADD COLUMN query (table *test*)

Result Grid

	id	num
▶	1	NULL
*	NULL	NULL

This is not something I have experienced in embedded SQL (through Ruby on Rails), which does not require an arity match between the relational schema and the attributes of the INSERT query.

Moving on:

```
ALTER TABLE team
ADD COLUMN abbreviation char(3) UNIQUE;
```

I then realized that I should ensure a length check because although the x in char(x) defines a maximum, it does not ensure that the values inserted are of the exact length x.

```
ALTER TABLE team
ADD CONSTRAINT abbreviation_length CHECK(char_length(abbreviation) = 3);
```

In the interest of a product that would be used by many children, a constraint is added on the usernames allowed:

```
ALTER TABLE fantasy_soccer.`user`
ADD CONSTRAINT no_rude_username
CHECK(username NOT IN ('rude','list','of','words'));
```

We also add some basic e-mail verification:

```
ALTER TABLE fantasy_soccer.`user`
ADD CONSTRAINT verify_email_format
CHECK(email LIKE '%@%' AND email LIKE '%.%')
```

We limit the values that can fill the *name* attribute in the position Table

```
ALTER TABLE position
ADD CONSTRAINT valid_position
CHECK(position.`name` IN('Goalkeeper','Defender','Midfielder','Forward'));
```

Update:

Having altered several columns with no default value in the column definition statement, I had to update some values to be consistent and to minimise NULL column attributes, particularly where I did not want the attribute to be NULL. As noted earlier, an insertion on Player failed because of a whitespace in the abbreviation attribute.

```
UPDATE team
SET abbreviation = 'ARS'
WHERE name = 'Arsenal';


UPDATE team
SET name = 'Bournemouth'
WHERE name = 'Bournemouth ';


UPDATE league
SET logo =
LOAD_FILE('C:\Users\Conor\Documents\college_programming\sql_project\bpl.png')
WHERE `name` = 'Premier League';
```

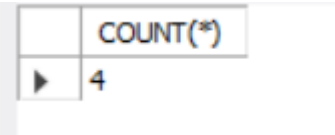
Select:

Any sort of 'interesting' SELECT or project query is found in the *Views* section. Here's an example of a really basic SELECT which I used to keep track of how many tuples I had in each relation.

```
SELECT COUNT(*) FROM event_type;
SELECT COUNT(*) FROM fantasy_soccer.`event`;
SELECT COUNT(*) FROM fantasy_league;
SELECT COUNT(*) FROM fantasy_league_fantasy_team;
SELECT COUNT(*) FROM fantasy_team;
SELECT COUNT(*) FROM fantasy_team_player;
SELECT COUNT(*) FROM gameweek;
SELECT COUNT(*) FROM league;
SELECT COUNT(*) FROM player;
SELECT COUNT(*) FROM position;
```

```
SELECT COUNT(*) FROM team;

SELECT COUNT(*) FROM fantasy_soccer.`user`;
```

In MySQL, this rendered a series of simplistic “result grids”. Here is Position’s result grid, for example:



	COUNT(*)
▶	4

Triggers:

Any semantics constraints that impact several tables cannot be implemented using a CONSTRAINT x CHECK(y) within table definition – this is because we cannot access information from another table within a CREATE TABLE. It may be possible using SQL functions within the CHECK, but given that we may want to specify the action which triggers this constraint and it may not be constant across actions (for example, we may want to check something before INSERT of a record, but we may not be concerned about the same problem before UPDATE of a record), we thus implement these constraints using triggers.

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.

Before insertion of an Event, we must ensure that the Position of the Player to whom we are associating the event matches the Position of the Event Type to which we are associating the Event **(T3)**.

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* **(T4)**. 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 **(T5)**, 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*).

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;//
```

T2: Maximum of three players from any given Team in a Fantasy Team

```
DELIMITER //
CREATE TRIGGER max_num_players_from_team
BEFORE INSERT ON fantasy_team_player
FOR EACH ROW
BEGIN
   DECLARE count INT;
   DECLARE error_msg varchar(255);

   SELECT COUNT(*)
   INTO count
   FROM team AS t
   JOIN player AS p
   JOIN fantasy_team_player AS ftp
   ON ftp.fantasy_team_id = new.fantasy_team_id
   WHERE t.`name` = p.team_name;

   IF(count > 2) THEN
     SET error_msg = "You cannot add this player to your team.
                              You already have three players from this team";
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = error_msg;
   END IF;
END;//
```

T3: Position of Player matches Position of Event Type

```
delimiter //
CREATE TRIGGER event_type_matches_player_position
BEFORE INSERT ON `event`
FOR EACH ROW
BEGIN
   DECLARE player_position varchar(255);
   DECLARE error_msg varchar(255);

   SELECT p.position
   INTO player_position
   FROM player AS p
   WHERE p.id = new.player_id;

   IF (player_position != new.event_type_position) THEN
     SET error_msg = "You cannot create this event as the position for the
                    event does not match its player's position";
      SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = error_msg;
    END IF;
END;//
```

T4: 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;//
```

T5: Tally points for Events associated with Players of a given Fantasy Team's Gameweek

```
delimiter //
CREATE TRIGGER tally_points
AFTER UPDATE ON gameweek
FOR EACH ROW
BEGIN
	DECLARE points INT;
	IF old.finished = 'false' AND new.finished = 'true' THEN
		SELECT SUM(et.point_value)
		INTO points
		FROM event_type AS et
		JOIN `event` AS e
		ON e.event_type_name = et.event_name
			AND e.event_type_position = et.position
		JOIN fantasy_team_player AS ftp
		ON e.player_id = ftp.player_id;

		UPDATE fantasy_team
		SET points_total = points_total + points
		WHERE fantasy_team.id = new.fantasy_team_id;
	END IF;
END;//
```

Data Security:

There are several aspects of this project that look to maximise data security. Views are a tool used in SQL to partition information to prevent the end-user from having access to information to which he should not have access. There are several views which would be common across all users. In **V1**, I define a global leaderboard view. This displays the top 50 Fantasy Teams with respect to their accumulated points, as well as the username of the Users who manage the teams. Given that our user table contains some very sensitive information (e-mail/password), it allows us to prevent this sensitive data from being displayed.

V1: Global Leaderboard

```
CREATE VIEW global_leaderboard
AS
SELECT ft.`name`, ft.logo, u.username, ft.points_total
FROM fantasy_team as ft
JOIN `user` AS u
ON ft.user_id = u.username
ORDER BY ft.points_total DESC
LIMIT 50;
```

Secondly, I define a similar leaderboard for the Players whose associated Events have accumulated the most points. Here, the view does not have any particular security benefits, but it does allow us to store a SELECT that would presumably be used very often in the application this database would support. It also only gets the relevant attributes of the tables whose data we want in the view, which means that the executor of the SQL query does not have to remember the various attributes that a commonly executed SELECT such as this would contain.

```
CREATE VIEW highest_scoring_players
AS
SELECT p.avatar, p.fname, p.lname, pos.`name`,
        p.team_name, p.fantasy_value, SUM(et.point_value)
FROM player AS p
JOIN `event` AS e
ON e.player_id = p.id
JOIN position AS pos
```

```
ON pos.`name` = p.position
JOIN event_type AS et
ON e.event_type_name = et.event_name AND e.event_type_position = et.position
ORDER BY et.point_value DESC
LIMIT 50;
```

Once these views were defined, I turned my focus onto less general views. I wanted to allow a user to update their own team details, and I wanted the admin user of a fantasy league to be able to see the contact details (i.e. e-mail) of any Users who have teams entered in the Fantasy League the admin user administrates. In order to do this, I had to define some manner of access permission.

I chose the in-built MySQL user to create a User. Note that this User is stored in the *mysql.user* table and not in *fantasy_soccer.user*.

```
CREATE USER 'jangirl'@'localhost';
```

I then created a view to represent a Fantasy Team's information of a *fantasy_soccer.user*'s whose username corresponds with our *mysql.user*'s username.

```
CREATE VIEW individual_user_teams
AS
SELECT ft.`name`, ft.logo, ft.points_total
FROM fantasy_team AS ft
WHERE ft.user_id = 'jangirl';
```

I then granted UPDATE permissions to the *mysql.user*:

```
GRANT UPDATE ON individual_user_teams TO 'jangirl'@'localhost';
```

Similarly, I created a view that would represent all of the contact details of the Users who have Fantasy Teams in a league administrated by our *mysql.user*.

```
CREATE VIEW fl_2_personal_details
AS
SELECT fl.`name`, u.username, u.email
FROM fantasy_league AS fl
JOIN fantasy_soccer.`user` AS u
ON fl.admin_user_id = 'jangirl';
```

We then allow this admin user read the data represented in the view.

```
GRANT SELECT ON fl_2_personal_details TO 'jangirl'@'localhost';
```

This data security section led me to several conclusions. Firstly, with regard to GDPR, allowing a User to access contact details (i.e. personal details) of the Users who have entered Fantasy Teams in the league administered by said User would oblige us to inform the users and have them agree to this before registering. Alternatively, on sign up, we could allow them to decide whether these personal details were shared. This could be done by keeping a *share_personal_details* column in our User table which would act as a boolean.

Secondly, I found the concept of views to be positive. I think storing queries we might make very often as a view would help to reduce developer workload, with them not having to remember the SELECT operation on each occasion. Secondly, from a data security standpoint, data partitioning is always a good thing. However, views are very rigid in MySQL and not very customizable. In the *individual_user_team* view, it is not possible to pass a parameter such as *username* which would allow us to dynamically select information. It is possible to use MySQL functions within a CREATE VIEW statement but again, as a given view creation query can only be ran once, it is not very customizable. For example, if we could do:

```
SELECT * FROM individual_user_teams('jangirl');
```

that would be much more useful.

I did create a procedure which allows for parameters and thus is much more customizeable:

```
DELIMITER //

CREATE PROCEDURE individual_user_teams(username varchar(255))

BEGIN

  SELECT `name`, logo, points_total

  FROM fantasy_team

  WHERE user_id = username;

END //

DELIMITER ;
```

And this can simply be called as follows:

```
CALL individual_user_teams('jangirl')
```

This is much more akin to the dynamic embedded SQL that I would have experience with in Ruby on Rails.

With regard to security by implementation on role based security, I would have preferred to implement a User (as in, based on the User entity in my schema) Role based access system. However, this is an extensive task beyond the scope of this project so I elected instead to show using built in SQL functionalities how such a system might work on a basic level.