
itucsd1507 Documentation

Release 1.0

itucsd1507

December 29, 2016

1	User Guide	3
2	Developer Guide	33
3	Installation Guide	71

Team ITUCSDB1507**Members**

- Alparslan Tozan
- İlay Köksal
- Kubilay Karpat
- Seda Yıldırım
- Sefa Eren Şahin

We designed a database to hold all the information that we need to know about American Football. Our database contains Player & Team & Coach informations. Leagues, Matches and much more! This web application allows users to change existing informations, add new data and see existing tables and statistics. You can see all tables from navigation bar above and start to explore our database and application.

Contents:

User Guide

Our application works as a main website for American Football. Here, users can find various information about the said sport. Users can add, view and edit all the data as they wish. The representation of the site map can be found below.

The site opens up to the welcome page, with the names of the contributors. From here, users can navigate to different pages via the navbar. The navigation bar is alphabetically ordered to ease the user experience.

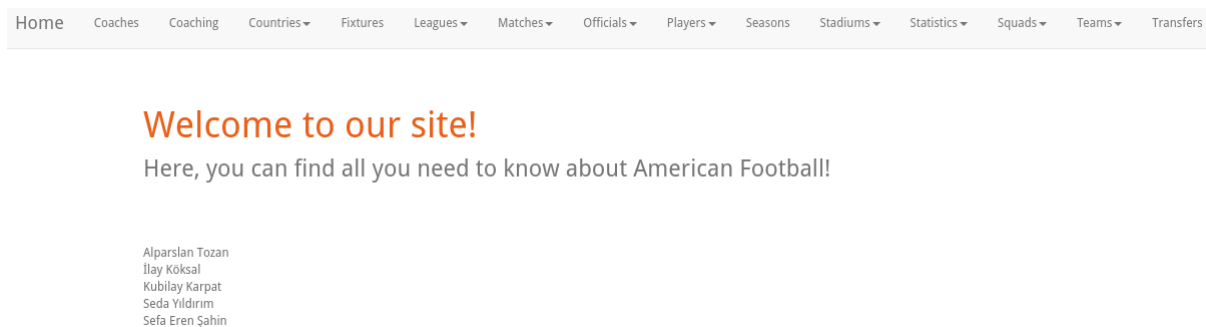


Fig. 1.1: Figure: The welcome page

The site contains various information about American Football, and this user guide further explains the site as it helps the users navigate seamlessly.

The user guides for the parts implemented by said team members can be found below.

1.1 Parts Implemented by Alparslan Tozan

Basic operations of there entities which are Officials, Matches and Transfers can be performed within user interface. All these operations could be reached from related main menu item's dropdown menu.

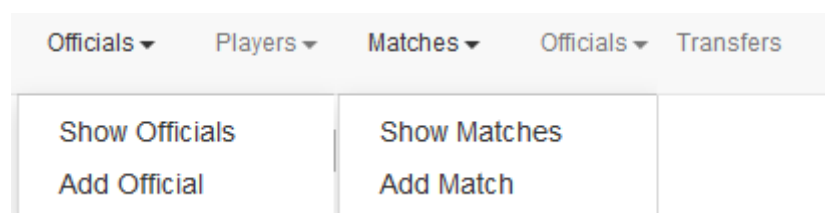


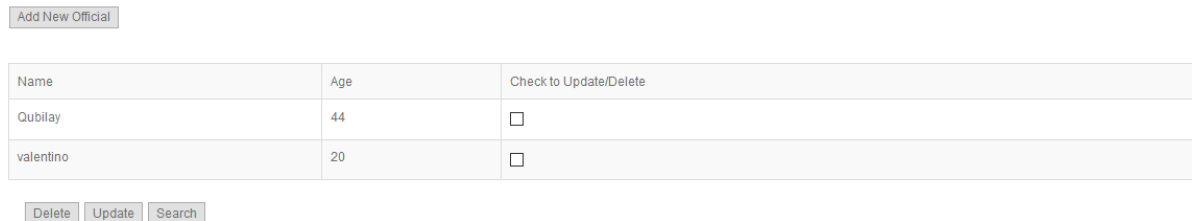
Fig. 1.2: Navbars of Officials, Matches and Transfers

1.1.1 Officials

Officials table is a core table, its only entries are name and age of the official.

Showing Officials

Officials can be listed by selecting “Show Officials” from dropdown menu.



Name	Age	Check to Update/Delete
Qubilay	44	<input type="checkbox"/>
valentino	20	<input type="checkbox"/>

Fig. 1.3: List of officials

Adding Official

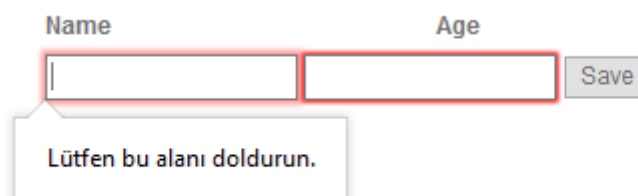
Players can be added by selecting “Add Official” from dropdown menu or clicking “Add New Official” button from Show Officials page.



Name **Age**

Fig. 1.4: Official addition can be completed by clicking Save button after filling required fields.

All required fields must be filled otherwise there will occur an error message.



Name **Age**

Lütfen bu alanı doldurun.

Fig. 1.5: Official addition error message

Updating and Deleting

At the “Show Officials” page after checking one of the check boxes user can select update/delete operations.

Add New Official

Name	Age	Check to Update/Delete
Qubilay	44	<input type="checkbox"/>
valentino	20	<input type="checkbox"/>

Delete Update Search

Fig. 1.6: List of officials

If delete button is clicked, selected official will be deleted and user will be redirected to “Show Officials” page again. If update operation is selected, user will be redirected to Update page, which looks similar to Add page.

Name Age

Update

Fig. 1.7: Update page

After making the desired changes, clicking Update button will update the official.

Searching

Users can search officials either by official name or by official age or by both by using the search form in the Players page.

Name Age

Search

Search Results

Name	Age
------	-----

Fig. 1.8: Search Page

1.1.2 Matches

Matches table is an entity that is connected to many other tables. It has connections with tables Teams, Seasons and Officials. Also it has its own feature result.

Showing Matches

Matches can be listed by selecting “Show Matches” from dropdown menu.

Add New Match

Home Team	Away Team	Result	Season	Official	Check to Update/Delete
Michigan Wolverines	Ohio State Buckeyes	Home Win	2014	Qubilay	<input type="checkbox"/>
Penn State Nittany Lions	Ohio State Buckeyes	Away Win	2015	valentino	<input type="checkbox"/>
Michigan Wolverines	Michigan Wolverines	Home Win	2015	Qubilay	<input type="checkbox"/>
Penn State Nittany Lions	Ohio State Buckeyes	Draw	2015	Qubilay	<input type="checkbox"/>

Delete Update

Fig. 1.9: List of Matches

Adding Matches

Matches can be added by selecting “Add Match” from dropdown menu or clicking “Add New Match” button from Show Matches page. As everything is fixed for Matches table its addition only consists of drop-down selections.

Home Team: Michigan Wolverines

Away Team: Michigan Wolverines

Season: 2015

Official: Qubilay

Result: Draw

Save

Fig. 1.10: Matches addition can be completed by clicking Save button after selecting required fields.

Updating and Deleting

At the “Show Matches” page after checking one of the check boxes user can select update/delete operations.

Add New Match

Home Team	Away Team	Result	Season	Official	Check to Update/Delete
Michigan Wolverines	Ohio State Buckeyes	Home Win	2014	Qubilay	<input type="checkbox"/>
Penn State Nittany Lions	Ohio State Buckeyes	Away Win	2015	valentino	<input type="checkbox"/>
Michigan Wolverines	Michigan Wolverines	Home Win	2015	Qubilay	<input type="checkbox"/>
Penn State Nittany Lions	Ohio State Buckeyes	Draw	2015	Qubilay	<input type="checkbox"/>

Delete Update

Fig. 1.11: List of Matches

If delete button is clicked, selected match will be deleted and user will be redirected to “Show Matches” page again. If update operation is selected, user will be redirected to Update page, which looks similar to Add page.

Home Team:	Michigan Wolverines	▼
Away Team:	Michigan Wolverines	▼
Season:	2015	▼
Official:	Qubilay	▼
Result:	Draw	▼

Fig. 1.12: Update page

After making the desired changes, clicking Update button will update the match.

1.1.3 Transfers

Transfers table is an entity that is connected to many other tables. It has connections with tables Teams, Seasons and Players. Also it has its own feature fee.

Showing Transfers

Transfers can be listed by selecting “Transfers” from navigation bar.

Player Name	Old Team	New Team	Season	Fee	Check to Update/Delete
Jerry Rice	Penn State Nittany Lions	Ohio State Buckeyes	2014	12	<input type="checkbox"/>
Jerry Rice	Michigan Wolverines	Michigan Wolverines	2015	2	<input type="checkbox"/>

Fig. 1.13: List of Transfers

Adding Matches

Transfers can be added by clicking “Add New Transfer” button from “Transfers” page. As almost everything is fixed for Transfers table its addition only consists of drop-down selections and one integer input for fee.

If fee field is filled with something different from integer value it will give an error message.

Player:	<input type="text" value="Jerry Rice"/>	▼
Old Team:	<input type="text" value="Michigan Wolverines"/>	▼
New Team:	<input type="text" value="Michigan Wolverines"/>	▼
Season:	<input type="text" value="2015"/>	▼
Fee:	<input type="text"/>	▲▼
<input type="button" value="Save"/>		

Fig. 1.14: Transfer addition can be completed by clicking Save button after filing and selecting required fields.

Player:	<input type="text" value="Jerry Rice"/>	▼
Old Team:	<input type="text" value="Michigan Wolverines"/>	▼
New Team:	<input type="text" value="Michigan Wolverines"/>	▼
Season:	<input type="text" value="2015"/>	▼
Fee:	<input type="text" value="MONEY"/>	▲▼
<div>Lütfen bir sayı girin.</div> <input type="button" value="Save"/>		

Fig. 1.15: Transfer addition error message

Updating and Deleting

At the “Transfers” page after checking one of the check boxes user can select update/delete operations.

Add New Transfer

Player Name	Old Team	New Team	Season	Fee	Check to Update/Delete
Jerry Rice	Penn State Nittany Lions	Ohio State Buckeyes	2014	12	<input type="checkbox"/>
Jerry Rice	Michigan Wolverines	Michigan Wolverines	2015	2	<input type="checkbox"/>

Delete Update

Fig. 1.16: List of Transfers

If delete button is clicked, selected match will be deleted and user will be redirected to “Transfers” page again. If update operation is selected, user will be redirected to Update page, which looks similar to Add page.

Player:

Old Team:

New Team:

Season:

Fee:

Update

Fig. 1.17: Update page

After making the desired changes, clicking Update button will update the match.

1.2 Parts Implemented by İlay Köksal

Add, Search, Update and Delete operations of tables Coaches, Seasons and Coaching can be done within user interface. Table that user wants to see or change can be chosen from navigation bar.

Home	Coaches	Coaching	Countries ▾	Fixtures	Leagues ▾	Matches ▾	Officials ▾	Players ▾	Seasons	Stadiums ▾	Statistics ▾	Squads ▾	Teams ▾	Transfers
------	---------	----------	-------------	----------	-----------	-----------	-------------	-----------	---------	------------	--------------	----------	---------	-----------

Fig. 1.18: Each operation for Coaches, Coaching and Seasons tables can be done in one single page.

1.2.1 Coaches

Coaches table is one of the core tables of our database. It has Name and BirthYear columns.

New coach can be add from textbox from top of the page.

Name:

BirthYear:

Add

Fig. 1.19: Add operation can be done by filling name and birthday field.

Under the add section, there is a textbox for searching coaches. When search button clicked, table below filled with items that requires search condition.

Search Coach:

Name	BirthYear
ilay	5991
Ahmet	1951

Fig. 1.20: Search operation is a case sensitive operation.

Delete and Update buttons can be seen in table that shows coaches. Every row have Update text boxes to fill when user wants to update related row. Delete button deletes the item in selected row.

1.2.2 Seasons

Seasons table is another core table in our database. It only keeps SeasonYear value for other tables usage. Seasons operations are in one single page as well.

New seasons can be added by filling Season year box that is located at top the page.

Below adding field, user can search season by typing season year that he/she wants to find.

Name	BirthYear	Delete	Update		
ilay	5991	<input type="button" value="Delete"/>	<input type="button" value="Update"/>	<input type="text" value="name value"/>	<input type="text" value="birthday value"/>
Ahmet	1951	<input type="button" value="Delete"/>	<input type="button" value="Update"/>	<input type="text" value="name value"/>	<input type="text" value="birthday value"/>

Fig. 1.21: To update an item, every update box must be filled.

Season Year:

Fig. 1.22: Season add field.

Search Season:

Year	Delete
2015	<input type="button" value="Delete"/>

Delete and Update buttons are table elements as well to affect related row. Update text box filled when user wants to update a season.

Year	Delete	Update
2015	<input type="button" value="Delete"/>	<input type="button" value="Update"/> <input type="text" value="year value"/>
2014	<input type="button" value="Delete"/>	<input type="button" value="Update"/> <input type="text" value="year value"/>
1991	<input type="button" value="Delete"/>	<input type="button" value="Update"/> <input type="text" value="year value"/>

Fig. 1.23: Seasons table rows consists Delete and Update buttons.

1.2.3 Coaching

Coaching table shows when a coach choached a team. So every column in coaching table related another table. Table consists Coach Name, Team Name and Season columns.

To add a coaching relation, user should select the values that he/she wants to add from dropdown lists.

The figure displays two versions of a web form for adding coaching relations. The form includes three dropdown menus: 'Team:', 'Coach:', and 'Year:'. In the left version, the 'Team' dropdown is set to 'Michigan Wolveri', 'Coach' to 'ilay', and 'Year' to '2015'. In the right version, the 'Team' dropdown is open, showing a list of teams with 'Michigan Wolverines' selected. The 'Coach' dropdown is partially visible, showing 'Co'. Both versions have an 'Add' button at the bottom.

Fig. 1.24: Coaching adding fields

Search field can be used to search both Coach name and Team name.

Update and Delete operations are located in table rows. To update user should select new values for item from dropdown lists in selected row. Delete button deletes related row from table.

Search Coach or Team:

Search

Team	Coach	Year
Michigan Wolverines	Ahmet	2014
Penn State Nittany Lions	Ahmet	2014

Fig. 1.25: Search field is case sensitive.

Search Coach or Team:

Search

Team	Coach	Year					
Michigan Wolverines	Ahmet	2014	Delete	Michigan Wolverines	İlay	2015	Update
Penn State Nittany Lions	Ahmet	2014	Delete	Michigan Wolverines	İlay	2015	Update

Fig. 1.26: Delete and Update buttons have their own columns.

1.3 Parts Implemented by Kubilay Karpal

Basic operations of there entities which are countries, leagues and stadiums could be performed within user inter-face. All these operations could be reached from related main menu item's dropdown menu.

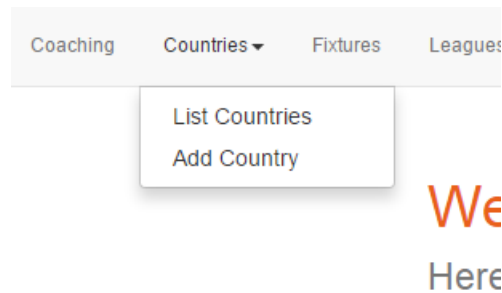


Fig. 1.27: eg. Countries' functions could be reached from main menu.

In the forms, leaving neccessary fields blank is not allowed so users prevented from making mistakes.

Add Country Add a new country

Name:

Abbreviation:


 Please fill out this field.

Fig. 1.28: eg. An error message displayed when the users leaves one of the neccessary fieds blank.

1.3.1 Countires

Country entity serves as a core data which only includes the country name and the abbreviation of it.

New countries can be added by selecting 'Add Country' from dropdown menu.

Another option in dropdown menu opens List Countries page where users can perform many operations related with countries.

Add Country

Add a new country

Name:

Abbreviation:

Submit

Fig. 1.29: Users can add new countries by filling two necessary fields.

Countries

List of all countries

Name	Abbreviation	Operations
Canada	CA	 
Türkiye	TR	 
United States	US	 

Enter search terms

Search

Fig. 1.30: List Countries page where users can list countries and also reach delete, update and search operations.

Users can delete a country with clicking the trash can icon. Also users can update the country with clicking wrench icon next to it. This will led them to update page. In update page users can change the information about the country with using fields which come prefilled with the current data.

Edit Country Update country information

Name:

Abbreviation:

Fig. 1.31: Edit Country page allow users to update information of the countries

Also users may search for countries by using the search field in the Countries List page. This options search for the keyword in the names of Countries.

Countries List of all countries



Name	Abbreviation	Operations
Canada	CA	 

Fig. 1.32: Seaarch result page

1.3.2 Leagues

Leagues are the entities which belogns the countries. They have a league name and a abbreviation also they have to connected with a country.

New leagues can be added by selecting ‘Add League’ option from dropdown menu. In this page there are two form fields and also a dropdown selection. In this dropdown all the countries that recorded at database are shown. User have to chose one of them. By applying this selection, connecting leagues with counties become easir and errorless for users.

Add League Add a new league

Name:

Abbreviation:

Country:

Submit

Fig. 1.33: Users can add new leagues by filling two fields (name is required) and selecting a country from dropdown menu.

User can list leagues like listing countries and maintain basic operations from list page. Delete operation can be done by clicking the trash icon.

When user clicks the wrench icon update page belonging to that entry will be opened.

Search function also works in a similar fashion and could be done by using the search field in list page.

1.3.3 Stadiums

Stadium is an entity that represents stadiums all around the world and as in the real life it is a part of the matches. A stadium must have a name, a country and a team. Also users can specify the capacity of stadium but it is not necessary.

Stadiums could be added by giving 3 necessary and 1 optional information. In these informations team and the country selection made by dropdown menu in order to prevent errors.

Stadiums have also a listing page with basic operations.

Edit page of stadiums is very similar to add page and it comes with the current entry's data.

Stadiums have also a listing page with basic operations.

Users can search the stadiums with their names

Leagues List of all leagues

League Name	Abbreviation	Country	Operations
National Football League	NFL	United States	 
Canadian Football League	CFL	Canada	 
Champions Indoor Football	CIF	United States	 

Fig. 1.34: List League page where users can list leagues and also reach delete, update and search operations.

Edit League Update league information

Name:

Abbreviation:

Country:

Fig. 1.35: Edit League page allow users to update information of the leagues. The page comes with prefilled data belonging to entry that going to be edited.

Leagues Search results: League





League Name	Abbreviation	Country	Operations
Canadian Football League	CFL	Canada	 
National Football League	NFL	United States	 

Fig. 1.36: League eaarch result page with the keyword in the header

Add Stadium Add a new stadium

Name:

Capacity:

Country:

CA

Team:

Michigan Wolverines

Submit

Fig. 1.37: Stadium adding page

Stadiums

 List of all stadiums

Name	Country	Team	Capacity	Operations
Michigan Stadium	United States	Michigan Wolverines	107601	 
Beaver Stadium	United States	Penn State Nittany Lions	106572	 
Ohio Stadium	United States	Ohio State Buckeyes	105944	 
Kyle Field	United States	Texas A&M Aggies	102733	 
Neyland Stadium	United States	Tennessee Volunteers	102455	 

Fig. 1.38: Stadiums listed and the delete / update operation buttons related to the entries

Edit Stadium

 Update stadium information

Name:

Capacity:

Country:

Team:

Fig. 1.39: Stadiums edit page

Stadiums

 Search results: Field



Name	Country	Team	Capacity	Operations
Kyle Field	United States	Texas A&M Aggies	102733	 

Fig. 1.40: Search results page with the given keyword shown in header

1.4 Parts Implemented by Seda Yıldırım

The following three tables were implemented: **Fixtures**, **Player Statistics**, and **Team Statistics**. The tabs Fixtures and Statistics can be seen on the navigation bar above the site interface. The **Player** and **Team Statistics** pages were grouped to one tab to provide better navigation since the navigation bar is in alphabetical order. Both pages can be seen on the drop down menu.

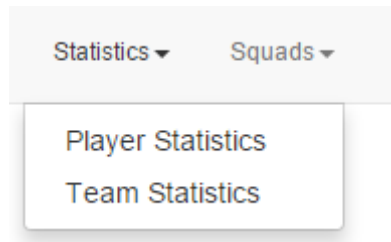


Fig. 1.41: Figure 1: The drop down for the Statistics pages

All of the fields in the forms are necessary to fill except the search form. If the user does not enter a required data, a warning is shown.

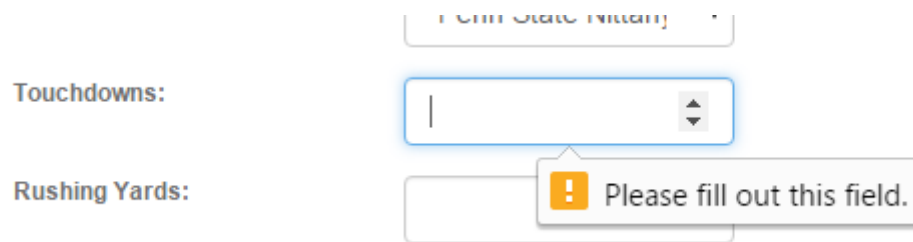


Fig. 1.42: Figure 2: The empty field warning

1.4.1 Fixtures

Fixtures page shows the seasonal scores of the teams in the database. From the main page, the user can navigate to either the **search page** or the **edit page**.

Fixtures

Season	Team	Points
2015	Michigan Wolverines	87
2015	Penn State Nittany Lions	67

Fig. 1.43: Figure 3: Overlay of the fixtures page

The **search page** features a form to search the database for a team name. The exact input of the team name is not required, though the search function is case sensitive.

Team

Click Search to view all records.

Search Results

Season	Team	Points
2015	Michigan Wolverines	87

Fig. 1.44: Figure 4: Search page

Note: If the form is left blank, the page displays the complete fixtures table.

The **edit page** displays two features: add and delete. From the edit page, users can also navigate to the update page, where they can change any fixture data. Users can enter new a fixture data with the add form. The form requires the selection of season and team data from the respective drop down menus. The points of the team can be entered manually.

Add a Fixture

Season:

Team:

Points:

Fig. 1.45: Figure 5: Add feature

Users can also delete any fixture data on the edit page. The delete feature is displayed right below the add feature. The feature displays a checklist, from which the user selects the fixture they want to delete. Only one fixture can be deleted at a time.

Delete a Fixture

	Season	Team	Points
<input type="checkbox"/>	2015	Michigan Wolverines	87
<input checked="" type="checkbox"/>	2015	Penn State Nittany Lions	67
<input type="checkbox"/>	2015	Ohio State Buckeyes	7

Fig. 1.46: Figure 6: Delete feature

The **update page** displays a checklist of all the fixtures. From here, the user can select the team they want to update, and can enter the respective values. The page redirects to itself, showing the current list of fixtures.

Update a Fixture

	Season	Team	Points
<input type="checkbox"/>	2015	Michigan Wolverines	87
<input type="checkbox"/>	2015	Penn State Nittany Lions	67
<input checked="" type="checkbox"/>	2015	Ohio State Buckeyes	7
	2015	Ohio State Buckeyes	10

Fig. 1.47: Figure 7: Update page

1.4.2 Player Statistics

Player statistics page displays the various seasonal statistics regarding the players in the database. Here, from the main page, the user can navigate to either the **search page** or the **edit page**.

Player Statistics

Season	Player	Tackles	Penalties
2014	Peija Stojakovic	98	76
2015	Barry Sanders	6	5
2015	Peija Stojakovic	7	8

Fig. 1.48: Figure 8: Overlay of the Player Statistics page

The **search page** features a form to search the database for a player's statistics, with the players name as the search query. The whole input of the player's name is not required, though the search function is case sensitive.

Player

Click Search to view all records.

Search Results

Season	Player	Tackles	Penalties
2014	Peija Stojakovic	98	76

Fig. 1.49: Figure 9: Search page

Note: If the form is left blank, the page displays the current statistics of all the players.

The **edit page** displays two features: add and delete. From the edit page, users can also navigate to the update page, where they can change any player's statistics. Users can enter a new player's statistics with the add form. The form requires the selection of the season and the player data from the respective drop down menus. The statistics values of the player is entered manually.

Users can also delete any statistics on the edit page as they wish. The delete feature is displayed right below the add feature. The feature displays a checklist, from which the user selects the player whose statistics data they want

Add a Player Statistic

Season:

2015

Player:

Barry Sanders

Tackles:

Penalties:

Add

Fig. 1.50: Figure 10: Add page

to delete. Only one player statistics data can be deleted at a time.

Delete a Player Statistic

	Season	Player Name	Tackles	Penalties
<input checked="" type="checkbox"/>	2014	Peija Stojakovic	98	76
<input type="checkbox"/>	2014	Jerry Rice	4	4

Delete

Fig. 1.51: Figure 11: Delete feature

The **update** page displays a checklist of all the player statistics. From here, the user can select the player whose statistics they want to update, and can enter the respective values. The page redirects to itself, showing the current list of all players and their statistics.

Update a Player Statistic

	Season	Player	Tackles	Penalties
<input checked="" type="checkbox"/>	2014	Peija Stojakovic	98	76
<input type="checkbox"/>	2014	Jerry Rice	4	4

2014

Peija Stojakovic

100

80

Update

Fig. 1.52: Figure 12: Update page

1.4.3 Team Statistics

Team statistics page displays the various seasonal statistics regarding the teams in the database. Here, from the main page, the user can navigate to either the **search** page or the **edit** page.

The **search** page features a form to search the database for a team’s statistics, with the team name as the search query. The whole input of the team’s name is not required, though the search function is case sensitive.

Team Statistics

Search Team
Edit Team Statistics

Season	Team	Touchdowns	Receiving Yards
2014	Michigan Wolverines	7051	1555
2014	Ohio State Buckeyes	7	88
2015	Michigan Wolverines	67	6
2015	Penn State Nittany Lions	7	8

Fig. 1.53: Figure 13: Overlay of the Team Statistics page

Team

Click Search to view all records.

Search Results

Season	Team	Touchdowns	Receiving Yards
2014	Michigan Wolverines	7051	1555
2015	Michigan Wolverines	67	6

Fig. 1.54: Figure 14: Search page

Note: If the form is left blank, the page displays the current statistics of all the teams.

The **edit page** displays two features: add and delete. From the edit page, users can also navigate to the update page, where they can change any team's statistics. Users can enter a new team's statistics with the add form. The form requires the selection of the season and the team data from the respective drop down menus. The statistics values of the team is entered manually.

Add a Team Statistic

Season:

2015 ▼

Team:

Michigan Wolver ▼

Touchdowns:

Rushing Yards:

Add

Fig. 1.55: Figure 15: Add feature

Users can also delete any statistics on the edit page as they wish. The delete feature is displayed right below the add feature. The feature displays a checklist, from which the user selects the team whose statistics data they want to delete. Only one team's statistics data can be deleted at a time.

Delete a Player Statistic

	Season	Player Name	Touchdowns	Receiving Yards
<input checked="" type="checkbox"/>	2014	Michigan Wolverines	7051	1555
<input type="checkbox"/>	2014	Tennessee Volunteers	2	3
<input type="checkbox"/>	2015	Michigan Wolverines	67	6

Fig. 1.56: Figure 16: Delete feature

The **update page** displays a checklist of all the team statistics. From here, the user can select the team whose statistics they want to update, and can enter the respective values. The page redirects to itself, showing the current list of all teams and their statistics.

Update a Team Statistic

	Season	Team	Touchdowns	Receiving Yards
<input checked="" type="checkbox"/>	2014	Michigan Wolverines	7051	1555
<input type="checkbox"/>	2014	Tennessee Volunteers	2	3
<input type="checkbox"/>	2015	Michigan Wolverines	67	6
	<input type="text" value="2014"/>	<input type="text" value="Michigan Wolverines"/>	<input type="text" value="751"/>	<input type="text" value="15"/>

Fig. 1.57: Figure 17: Update page

1.5 Parts Implemented by Sefa Eren Şahin

Basic operations of there entities which are Players, Teams and Squads can be performed within user interface. All these operations could be reached from related main menu item’s dropdown menu.

1.5.1 Players

Players table is a core table, including player_id, name, birthday and position datas.

Inserting

Players can be added by selecting “Add Player” from dropdown menu.

All required fields must be filled otherwise there will occur an error message.

Listing and Deleting

Players can be listed by selecting “Show Player” from dropdown menu.

Player Name:

Birthday:

Position:

Fig. 1.58: Player addition can be completed by clicking Add Player button after filling and selecting required fields.

Player Name:

Birthday:

Position:


 Lütfen bu alanı doldurun.

Fig. 1.59: Player addition error message

Players List of all players

Enter the player name to search					Search
PLAYER ID	NAME	BIRTHDAY	POSITION	DELETE/UPDATE	
2	Jerry Rice	1953-10-13	Wide Receiver	 UPDATE	 DELETE
3	Peija Stojakovic	1970-09-10	Cornerback	 UPDATE	 DELETE

Fig. 1.60: List of players

Players can be deleted by clicking Delete button related with the corresponding row. Clicking Update button redirects user to player update page.

Updating



A form for updating player data. It consists of three input fields and an 'Update' button. The first field is labeled 'Player Name:' and contains the text 'Peija Stojakovic'. The second field is labeled 'Birthday:' and contains the text '10.09.1970'. The third field is labeled 'Position:' and contains the text 'Center' with a dropdown arrow on the right. Below these fields is a button labeled 'Update'.

Fig. 1.61: *Player data is prefilled into update form.*

After making the desired changes, clicking Update button will update the player.

Searching

Users can search players by player name by using the search form in the Players page.

Search Results

PLAYER ID	NAME	BIRTHDAY	POSITION	DELETE/UPDATE
2	Jerry Rice	1953-10-13	Wide Receiver	↗ UPDATE ↗ DELETE

Fig. 1.62: *Search results*

1.5.2 Teams

Teams table contains team_id, name and league_id references to leagues table.

Inserting

Teams can be added by selecting “Add Team” from dropdown menu.

All required fields must be filled otherwise there will occur an error message.

Team Name:

League ID:

Fig. 1.63: Team addition can be completed by clicking Add Team button after filling and selecting required fields.

Team Name:

League ID:

! Lütfen bu alanı doldurun.

Fig. 1.64: Team addition error message

Teams List of all teams

Enter the team name to se <input type="button" value="Search"/>			
TEAM ID	NAME	LEAGUE ID	UPDATE/DELETE
4	Michigan Wolverines	1	<input type="button" value="UPDATE"/> <input type="button" value="DELETE"/>
5	Penn State Nittany Lions	1	<input type="button" value="UPDATE"/> <input type="button" value="DELETE"/>
6	Ohio State Buckeyes	1	<input type="button" value="UPDATE"/> <input type="button" value="DELETE"/>
7	Texas A&M Aggies	1	<input type="button" value="UPDATE"/> <input type="button" value="DELETE"/>
8	Tennessee Volunteers	1	<input type="button" value="UPDATE"/> <input type="button" value="DELETE"/>

Fig. 1.65: List of teams

Listing and Deleting

Teams can be listed by selecting “Show Teams” from dropdown menu.

Teams can be deleted by clicking Delete button related with the corresponding row. Clicking Update button redirects user to team update page.

Updating

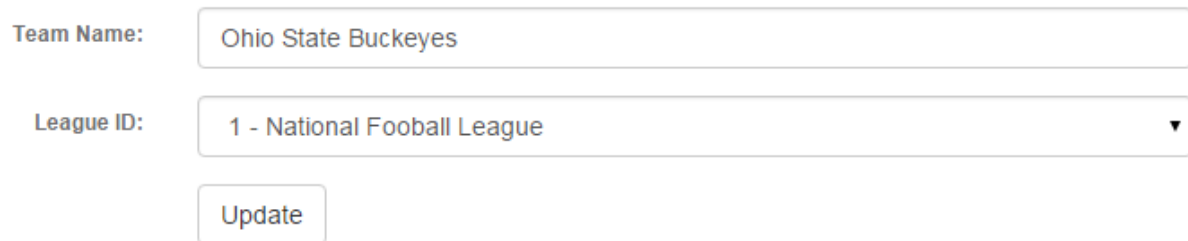
A form for updating team data. It consists of two input fields and one button. The first field is labeled 'Team Name:' and contains the text 'Ohio State Buckeyes'. The second field is labeled 'League ID:' and contains the text '1 - National Football League' with a downward arrow on the right. Below these fields is a button labeled 'Update'.

Fig. 1.66: Team data is prefilled into update form.

After making the desired changes, clicking Update button will update the team.

Searching

Users can search teams by team name by using the search form in the Teams page.

Search Results

TEAM ID	NAME	LEAGUE ID	UPDATE/DELETE
7	Texas A&M Aggies	1	↗ UPDATE ✕ DELETE
8	Tennessee Volunteers	1	↗ UPDATE ✕ DELETE

Fig. 1.67: Search results

1.5.3 Squads

Squads table contains `squad_id`, `team_id` references to Teams table, `player_id` references to Players table and `kit_no`.

Inserting

Squads can be added by selecting “Add Squad” from dropdown menu.

All required fields must be filled otherwise there will occur an error message.

Team ID: 4 - Michigan Wolverines ▼

Player ID: 2 - Jerry Rice ▼

Kit Number:

Add Squad

Fig. 1.68: Squad addition can be completed by clicking Add Squad button after filling and selecting required fields.

Team ID: 4 - Michigan Wolverines ▼

Player ID: 2 - Jerry Rice ▼

Kit Number:

Add Squad

! Lütfen bu alanı doldurun.

Fig. 1.69: Squad addition error message

Listing and Deleting

Squads can be listed by selecting “Show Squads” from dropdown menu.

Squads List of all squads

Team Name Ohio State Buckeyes ▼ Filter				
SQUAD ID	TEAM NAME	PLAYER NAME	KIT NO	UPDATE/DELETE
5	Ohio State Buckeyes	Jerry Rice	34	UPDATE DELETE
6	Tennessee Volunteers	Peija Stojakovic	9	UPDATE DELETE

Fig. 1.70: List of squads

Squads can be deleted by clicking Delete button related with the corresponding row. Clicking Update button redirects user to squad update page.

Updating

After making the desired changes, clicking Update button will update the squad.

Team ID:

8 - Tennessee Volunteers

Player ID:

3 - Peija Stojakovic

Kit Number:

9

Update

Fig. 1.71: Squad data is prefilled into update form.

Searching

Users can filter squads by team name by selecting the team name from the search form in the Teams page.

Squads List of all squads

Team Name

Ohio State Buckeyes

Filter

SQUAD ID	TEAM NAME	PLAYER NAME	KIT NO	
5	Ohio State Buckeyes	Jerry Rice	34	<div><div>UPDATE</div><div>DELETE</div></div>
6	Tennessee Volunteers	Peija Stojakovic	9	<div><div>UPDATE</div><div>DELETE</div></div>

Fig. 1.72: Squad filtering form

After filtering, squads related with selected team are listed.

Search Results

SQUAD ID	TEAM NAME	PLAYER NAME	KIT NO	UPDATE/DELETE
5	Ohio State Buckeyes	Jerry Rice	34	<div><div>UPDATE</div><div>DELETE</div></div>

Fig. 1.73: Search results

Developer Guide

2.1 Database Design

Main purpose of this database is creating a web application to hold basic information about American Football.

Our database mainly contains information about Teams, Players, Coaches, Matches, Leagues etc.

Teams and Players tables are the most active tables. Tables like Player Statistics, Team Statistics, Transfers, Fixture are designed to keep the relations between Teams and Players. Teams and Players tables are referenced by other tables alot.

Countries, Players, Officials, and Seasons tables are core tables. These entities do not reference any other table.

PostgreSQL is the relational database management system used in Database Design.

Psycopg2 is used as database adapter.

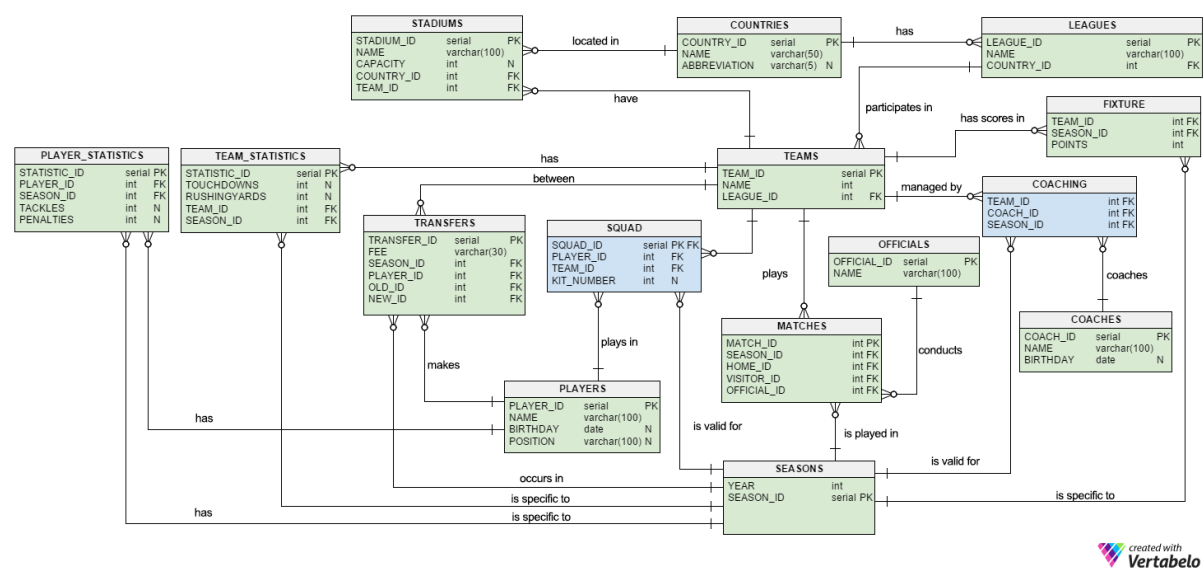


Fig. 2.1: ER Relation Diagram of American Football Database

2.2 Code

Server file has been contributed by all group members.

We first connected our tables' python classes to server:

```
if __name__ == '__main__':

    '''Container objects'''

    app.coaches = Coaches2(app)
    app.coaching = Coaching2(app)
    app.teams = Teams(app)
    app.players = Players(app)
    app.countries = Countries(app)
    app.leagues = Leagues(app)
    app.stadiums = Stadiums(app)
    app.officials = Officials(app)
    app.seasons = Seasons2(app)
    app.matches = Matches(app)
    app.statisticsTeam = StatisticsT(app)
    app.statisticsPlayer = StatisticsP(app)
    app.fixtures = Fixtures(app)
    app.squads = Squads(app)
    app.transfers = Transfers(app)
```

Initialization of all tables is controled by a function in the server.py. At the creation of tables we looked at the relations of tables with each other and created them according to their priority. We first created core tables, then other tables were created. As we did not wanted to make it reachable to every user we did not put a button that goes to this address. To reach it user need to write “/init_db” to the end of the home page link. Here is the python code of this function:

```
def create_tables():

    '''Reference order in DB should be preserved'''
    app.coaches.initialize_tables()
    app.seasons.initialize_tables()
    app.countries.initialize_tables()
    app.players.initialize_tables()
    app.leagues.initialize_tables()
    app.teams.initialize_tables()
    app.stadiums.initialize_tables()
    app.coaching.initialize_tables()
    app.squads.initialize_tables()

    app.officials.initialize_tables()
    app.matches.initialize_tables()
    app.transfers.initialize_tables()

    app.statisticsTeam.initialize_tables()
    app.statisticsPlayer.initialize_tables()
    app.fixtures.initialize_tables()

    return redirect(url_for('home_page'))
```

To drop the tables we again write a function at server.py, but this time it does not call any other class. It just scans the database and finds table names in out schema. Then it drops tables according to those names. Here is the python code:

```
def drop_tables():
    with dbapi2.connect(app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""SELECT table_schema,table_name FROM information_schema.tables
                           WHERE table_schema = 'public' ORDER BY table_schema,table_name""")
        rows = cursor.fetchall()
        for row in rows:
            if row[1]!="pg_stat_statements":
                cursor.execute("drop table " + row[1] + " cascade")
```

```
connection.commit()
return redirect(url_for('create_tables'))
```

Note: As ElephantSQL creates a table on its own and we do not have right to delete this table, we skip “pg_stat_statements” table.

2.2.1 Parts Implemented by Alparslan Tozan

I implemented official, match and transfer entities and belonging operations. In order to do that I created officials, matches and transfers classes to implement demanded operations.

All these classes contains same basic methods which listed below.

Operations

- **Initialize table methods** This operations basically run a query to create related table.
- **Add methods** This methods take variables that represent columns in table and perform a insert operation.
- **Delete methods** This method takes the entity’s primary key and delete it from database. Users reach delete function from listing pages and they do not naturally interact with the primary keys, this information kept but hidden.
- **Update methods** Similar with add methods this methods also take entity’s fields as parameters but also an entity ID which corresponds to the primary key in the table is given. Same as in the delete operation, these keys are invisible to users.
- **Get Entity Method**¹ These methods take an entity ID and returns the entity’s all columns. These are actually helper functions that used by another entities since some table have foreign keys and had to reach related name or etc with these keys.
- **Get Entities Methods** Mostly used in list pages these methods simply returns all entries belong to a entity. Also these functions used in to Add / Delete operations of entities that have foreign keys since they need to list all options to the users as a dropdown etc. These methods do not take parameters.
- **Search Method**² Search method designed to search by name or age in official entity as case sensitive. It basically takes a string or a number, which will later be changed into a string, that represents search words and returns all related entities.

Delete and Update Operations and Their Form

I want to implement delete and update functions in list page in a such way that users can easily reach. In order to archive this I placed delete and update buttons following the entries in list page.

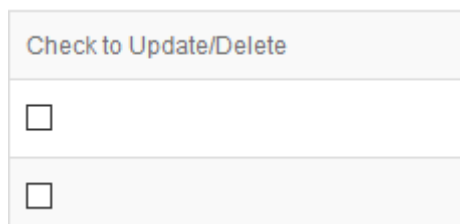


Fig. 2.2: User need to select one of the check boxes.

¹Only in Officials

²Only in Officials



Fig. 2.3: After selecting one of the check boxes user can determine between update and delete operations

In check boxes I used a hidden form value to send the primary key with POST request. Forms action is posting these values to a determination page, which will determine the process. After check boxes i putted another hidden value. If none of the check boxes is selected but update/delete button is still selected, with this value we make sure there will not be an error from web site. In this way I achieved the function that I want, users were able to delete / update entries by just checking the corresponding check box without entering a key value or an attribute like name etc.

HTML Part

```
<form action="{ { url_for('transfer_add') } }" method="post" role="form">
...
    {% for key, transfer in transfers %}
    ...
        <td><input type="checkbox" name="id" value="{ { key } }"/></td>
    </tr>
    {% endfor %}
</table>
<input type="hidden" value="0" name="id">
<div class="col-sm-10">
    <input type="submit" value="Delete" name ="submit">
    <input type="submit" value="Update" name ="submit">
</div>
</form>
```

Python Part

```
def transfer_determine():
    if request.method=='GET':
        return redirect(url_for('transfers'))
    if request.form['id']=="0":
        return redirect(url_for('transfers'))
    if request.form['submit'] == "Delete":
        id = request.form['id']
        form = request.form
        form_data={id: form['id']}
        return redirect(url_for('transfer_delete'), code=307 )
    elif request.form['submit'] == "Update":
        return render_template('transfer_update.html', id = request.form['id'],
                               teams=app.teams.select_teams(),
                               season=app.seasons.select_seasons(),
                               players=app.players.select_players())
    else:
        return redirect(url_for('transfers'))
```

Official Implementation

I designed officials class in order to perform operations in my officials table. Official is a core entity in our database and used in matches tables as a foreing key. Also I designed a official class to represent a row data of a country except for primary key.

Note: After implementing official entity and some of matches functions I realized that using a class for holding

entity information and using it as a parameter in functions is not a good way to maintain the operations. In other entities I did not use classes for entities / methods instead I used column variables as separate parameters.

Officials Table

In our database countries table has following columns

- **OFFICIAL_ID** as serial type and primary key *This is the primary key of the table*
- **NAME** as varchar(100) and not null *This column holds the full name of the official and it can't be null*
- **AGE** as INT and not null *This column holds the age of the official*

As python/SQL code:

```
def initialize_tables(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS OFFICIALS (
                OFFICIAL_ID SERIAL NOT NULL PRIMARY KEY,
                NAME varchar(100) NOT NULL,
                AGE INT NOT NULL
            );""")

        connection.commit()
```

Since this is a core entity, it does not have a foreign key.

add_official Method

This method takes an official object as a parameter and inserts it into the database.

Here is the code block that does the add operation in the database using **INSERT** command:

```
def add_official(self, name, age):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            INSERT INTO OFFICIALS (NAME, AGE)
            VALUES (%s, %s) """,
            (name, age))
        connection.commit()
```

delete_official Method

This method takes an *official_id* (which is a primary key of the officials table) and deletes it from the database. To match the country on the database **WHERE** statement is used on *official_id* column.

Here is the code block that performs the delete operation on the officials table using **DELETE** command:

```
def delete_official(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            DELETE FROM OFFICIALS
            WHERE OFFICIAL_ID = %s""",
            id)
        connection.commit()
```

update_official Method

This method works in a similar fashion with add function, it takes one more argument which is the *official_id*. The given *Official* object is parsed and the row that related with *official_id* argument is updated with this parsed information.

Here is the code block that perform update operation on officials table using **UPDATE** command:

```
def update_official(self, id, official):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE OFFICIALS
                     SET NAME = %s, AGE = %s
                     WHERE OFFICIAL_ID = %s"""
        cursor.execute(query, (official.name, official.age, id))
        connection.commit()
```

get_official Method

This method is used by matches class. Its main function is to provide all columns related with a foreign key which consists a *official_id*. It does simply run *SELECT* query with *WHERE* statement to match *official_id*. It just returns the *name* of the matching id.

```
def get_official(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """SELECT * FROM OFFICIALS WHERE OFFICIAL_ID = %s"""
        cursor.execute(query, [id])
        key, name, age = cursor.fetchone()
        return name
```

get_officials Method

Similar to *get_country* methods runs a *SELECT* on countries table but this time without a specific ID. Simply it returns all officials in database without taking a parameter.

```
def get_officials(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """SELECT * FROM OFFICIALS
                   ORDER BY OFFICIAL_ID ASC"""
        cursor.execute(query)
        connection.commit()

        officials = [(key, Official(name, age))
                     for key, name, age in cursor]

    return officials
```

search_officials Method

This method takes two string values to search in officials table by matching these strings which is the search phrase actually on the name and age columns and returns a list of matching officials.

```
def search_officials(self, name, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """SELECT * FROM OFFICIALS
                   WHERE NAME LIKE '%s' AND CAST(AGE as VARCHAR(30)) LIKE '%s'"""
```

```

        ORDER BY OFFICIAL_ID ASC""" % (('%' + name + '%', id + '%'))
    cursor.execute(query)
    connection.commit()
    print(name)

    officials = [(key, Official(name, age))
                 for key, name, age in cursor]

    return officials

```

Match Implementation

Match is an important entity in American Football Database project that stores all matches that had been played.

Matches Table

Matches table consists of following columns:

- **MATCH_ID** as serial type and primary key *This is the primary key of the table*
- **SEASON_ID** as integer type, not null and references to seasons table *This is foreing key to seasons table, represent the season that the match has been played at*
- **HOME_ID** as integer type, not null and references to teams table *This is foreing key to teams table, represent the home team that played match*
- **VISITOR_ID** as integer type, not null and references to countries table *This is foreing key to teams table, represent the away team that played match*
- **OFFICIAL_ID** as integer type and references to officials table *This is foreing key to officials table, represent the official that monitored the match*
- **RESULT** as varchar(30) and not null *This column holds the result of the match and it cannot be null*

As python/SQL code:

```

def initialize_tables(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS MATCHES (
                MATCH_ID SERIAL NOT NULL PRIMARY KEY,
                SEASON_ID int NOT NULL REFERENCES SEASONS(SEASON_ID),
                HOME_ID int NOT NULL REFERENCES TEAMS(Team_ID),
                VISITOR_ID int NOT NULL REFERENCES TEAMS(Team_ID),
                OFFICIAL_ID int REFERENCES OFFICIALS(OFFICIAL_ID),
                RESULT VARCHAR(30) NOT NULL
            );""")

        connection.commit()

```

add_match Method

This method takes a match object and performs *INSERT* operation onto database.

```

def add_match(self, match):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            INSERT INTO MATCHES (SEASON_ID, HOME_ID,
                VISITOR_ID, OFFICIAL_ID, RESULT)

```

```
VALUES (%s, %s, %s, %s, %s) """ ,
        (match.season_id, match.home_id, match.away_id,
         match.official_id, match.result))
connection.commit()
```

delete_match Method

This method takes a *match_id* and deletes corresponding row from database using *DELETE* operation.

```
def delete_match(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            DELETE FROM MATCHES
            WHERE MATCH_ID = %s""" ,
            id)
        connection.commit()
```

update_match Method

Takes an *match_id* and match the row in database then updates all columns with given parameters.

```
def update_match(self, id, match):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE MATCHES
            SET SEASON_ID = %s, HOME_ID = %s,
            VISITOR_ID = %s, OFFICIAL_ID = %s,
            RESULT = %s
            WHERE MATCH_ID = %s """
        cursor.execute(query, (match.season_id, match.home_id,
                               match.away_id, match.official_id,
                               match.result, id))
        connection.commit()
```

get_matches Method

This method used to fetch all matches from the database. It does not take a parameter and as a return value it returns the list of matches information in the database.

```
def get_matches(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="\"SELECT * FROM MATCHES
            ORDER BY MATCH_ID ASC\""
        cursor.execute(query)
        connection.commit()

        matches = [(key, Match(season_id, official_id, home_id, away_id, result,
                                season=self.app.seasons.get_season(season_id),
                                official_name=self.app.officials.get_official(official_id),
                                home_team=self.app.teams.get_team_name(home_id),
                                away_team=self.app.teams.get_team_name(away_id)))
                    for key, season_id, home_id, away_id, official_id, result in cursor]

    return matches
```

Note: *MATCHES* table holds various informations where these informations located by referencing other tables. To make our listing more understandable we get names of these informations using basic get functions using ID's of these tables.

Transfer Implementation

Transfer is a small entity that used to store records of transfers.

Transfer Table

Transfer table consists of following columns:

- ***TRANSFER_ID* as serial type and primary key** *This is the primary key of the table*
- ***SEASON_ID* as integer type, not null and references to seasons table** *This is foreing key to seasons table, represent the season that the transfer has took place*
- ***OLD_ID* as integer type, not null and references to teams table** *This is foreing key to teams table, represent the team that player played before transfer*
- ***NEW_ID* as integer type, not null and references to countries table** *This is foreing key to teams table, represent the team that player is played after transfer*
- ***PLAYER_ID* as integer type and references to players table** *This is foreing key to players table, represent the player that transfered*
- ***FEE* as varchar(30) and not null** *This column holds the fee of the transfer and it cannot be null*

As python/SQL code:

```
def initialize_tables(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS TRANSFERS (
                TRANSFER_ID SERIAL NOT NULL PRIMARY KEY,
                SEASON_ID int NOT NULL REFERENCES SEASONS(SEASON_ID),
                OLD_ID int NOT NULL REFERENCES TEAMS(Team_ID),
                NEW_ID int NOT NULL REFERENCES TEAMS(Team_ID),
                PLAYER_ID int REFERENCES PLAYERS(PLAYER_ID),
                FEE VARCHAR(30) NOT NULL
            ); """)

        connection.commit()
```

add_transfer Method

This method takes a transfer object and performs *INSERT* operation on *TRANSFERS* table.

```
def add_transfer(self, transfer):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            INSERT INTO TRANSFERS (SEASON_ID, OLD_ID,
            NEW_ID, PLAYER_ID, FEE)
            VALUES (%s, %s, %s, %s, %s) """,
            (transfer.season_id, transfer.old_id, transfer.new_id,
            transfer.player_id, transfer.fee))

        connection.commit()
```

delete_transfer Method

This method takes an *transfer_id* and deletes corresponding row from database.

```
def delete_transfer(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            DELETE FROM TRANSFERS
            WHERE TRANSFER_ID = %s""",
                id)
        connection.commit()
```

update_transfer Method

This method takes an *transfer_id* and new information that belongs to this entry as transfer object.

```
def update_transfer(self, id, transfer):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE TRANSFERS
            SET SEASON_ID = %s, OLD_ID = %s,
            NEW_ID = %s, PLAYER_ID = %s,
            FEE = %s
            WHERE TRANSFER_ID = %s """
        cursor.execute(query, (transfer.season_id, transfer.old_id,
                                transfer.new_id, transfer.player_id,
                                transfer.fee, id))
        connection.commit()
```

get_transfers Method

This method returns all transfers and information belongs to that transfers by using fetchall function and **JOIN** operations.

```
def get_transfers(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT T.TRANSFER_ID, T1.NAME, T2.NAME, S.YEAR, P.NAME, T.FEE
            FROM TRANSFERS T
            JOIN TEAMS T1 ON T1.TEAM_ID=T.OLD_ID
            JOIN TEAMS T2 ON T2.TEAM_ID=T.NEW_ID
            JOIN SEASONS S ON S.SEASON_ID=T.SEASON_ID
            JOIN PLAYERS P ON P.PLAYER_ID=T.PLAYER_ID"""
        cursor.execute(query)
        connection.commit()

        transfers = [(key, Transfer("1", "1", "1", "1", fee, season,
                                    player_name, old_team, new_team))
                    for key, old_team, new_team, season, player_name, fee in cursor]

    return transfers
```

Note: Even though get functions of *MATCHES* and *TRANSFERS* tables works same, one gets its entries names from basic functions, other gets these values from join operations. As join operation returns only the result that we want, we can say it is more effective.

2.2.2 Parts Implemented by İlay Köksal

I created coaches, seasons and coaching tables and their operations. All these tables contains same operations like Add, Delete, Update and Search.

- **Initialize Table** Creation of the table.
- **Select** Returns all elements of table
- **Get** Makes inner join to select wanted columns from other tables. Basically used in tables in which consists foreign key.
- **Add** Adding new row to table
- **Delete** Deleting row from table
- **Update** Updating selected row
- **Search** Searching table with given condition and returning rows which verify search condition.

Coaches Table and Operations

First i created a coaches class to implement all related operations for Coaches table.

Coaches table has the following columns

- **COACH_ID as serial primary key** This is the primary key of the table
- **NAME as varchar(50) and not null** Holds the name of the coach and can not be null
- **BIRTHDAY as integer and not null** Birthyear of coach.

Coaches table is a core table so it does not have any foreign key.

initialize_tables

First we create table with *CREATE* sql statement.

```
def initialize_tables(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""CREATE TABLE IF NOT EXISTS COACHES
        (
            COACH_ID SERIAL PRIMARY KEY,
            NAME VARCHAR(50) NOT NULL,
            BIRTHDAY INTEGER NOT NULL
        ) """)
        connection.commit()
```

select_coaches

With this method, we can see every coach item in table in ascending order.

```
def select_coaches(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT * FROM COACHES ORDER BY COACH_ID ASC"""
        cursor.execute(query)
        result = cursor.fetchall()
        return result
```

`add_coach`

This function takes name and birthday and add them to database with *INSERT* statement.

```
def add_coach(self, name, birthday):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ INSERT INTO COACHES (NAME, BIRTHDAY) VALUES (%s, %s) """
        cursor.execute(query, (name, birthday))
        connection.commit()
```

`search_coach`

This method returns the matching coaches to given string with *WHERE* and *SELECT* statements.

```
def search_coach(self, name):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT * FROM COACHES c WHERE c.NAME LIKE '%s'"""% ((''+name+'%'))
        cursor.execute(query)
        connection.commit()
        result = [(key, name,birth)
                   for key, name,birth in cursor]
    return result
```

`delete_coach`

Deleting done with taking the id of item that we want to delete and using it in *DELETE* and *WHERE* query.

```
def delete_coach(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ DELETE FROM COACHES WHERE COACH_ID =%s """
        cursor.execute(query, [id])
        connection.commit()
```

`update_coach`

Works similar to add function but in addition takes id argument of the item that we want to update.

```
def update_coach(self, coach_id, name, birthday):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE COACHES SET NAME = %s, BIRTHDAY= %s WHERE COACH_ID = %s """
        cursor.execute(query, (name,birthday,coach_id))
        connection.commit()
```

Seasons Table and Operations

Seasons table class created first to write its operations.

This table has columns below.

- **SEASON_ID as serial primary key** This is the primary key of the table
- **YEAR as integer and not null** Year value of season.

Seasons table is a core table as well so it does not have any foreign key too.

initialize_tables

First we create table with *CREATE* sql statement.

```
def initialize_tables(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""CREATE TABLE IF NOT EXISTS SEASONS
        (
            SEASON_ID SERIAL PRIMARY KEY,
            YEAR INTEGER NOT NULL
        ) """)
        connection.commit()
```

select_seasons

With this method, we can see every season value in ascending order.

```
def select_seasons(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT * FROM SEASONS ORDER BY SEASON_ID ASC"""
        cursor.execute(query)
        result = cursor.fetchall()
        return result
```

get_season

This method used by other classes and tables. They use this to select season with season id.

```
def get_season(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT * FROM SEASONS
        WHERE SEASON_ID = %s"""
        cursor.execute(query, [id])
        season_id, year = cursor.fetchone()
        return year
```

add_season

This function takes year value and add it to database with *INSERT* sql statement.

```
def add_season(self, year):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ INSERT INTO SEASONS (year) VALUES (%s) """
        cursor.execute(query, [year])
        connection.commit()
```

seach_coach

This method returns the matching season with *WHERE* and *SELECT* statements.

```
def search_season(self, year1):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
```

```
query="""SELECT * FROM SEASONS WHERE YEAR = %s"""
cursor.execute(query,[year1])
connection.commit()
result = [(key, year)
          for key, year in cursor]
return result
```

delete_season

Method takes id of the item as parameter. With *WHERE* statement, we can delete related item.

```
def delete_season(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ DELETE FROM SEASONS WHERE SEASON_ID =%s """
        cursor.execute(query, [id])
        connection.commit()
```

update_coach

Similar to add function but in addition takes id value of the item to be updated.

```
def update_season(self, season_id, year):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE SEASONS SET YEAR = %s WHERE SEASON_ID = %s """
        cursor.execute(query, (year,season_id))
        connection.commit()
```

Coaching Table and Operations

Coaching table class created and its operations implemented.

This table has columns below.

- **COACHING_ID as serial primary key** This is the primary key of the table
- **TEAM_ID** as integer and not null and references TEAM table
- **COACH_ID** as integer and not null and references COACHES table
- **SEASON_ID** as integer and not null and references SEASONS table

Coaching table is a relation table. It has three foreign keys and one serial primary key.

initialize_tables

First we create table with *CREATE* sql statement.

```
def initialize_tables(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""CREATE TABLE IF NOT EXISTS COACHING
(
    COACHING_ID SERIAL NOT NULL PRIMARY KEY,
    TEAM_ID INT NOT NULL REFERENCES TEAMS(TEAM_ID),
    COACH_ID INT NOT NULL REFERENCES COACHES(COACH_ID),
    SEASON_ID INT NOT NULL REFERENCES SEASONS(SEASON_ID)
) """)
        connection.commit()
```

select_coaching

This method helps us to see every coaching relation we have in our database.

```
def select_coaching(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """SELECT * FROM COACHING"""
        cursor.execute(query)
        result = cursor.fetchall()
        return result
```

get_coaching

With this method we call the values from other tables to show.

```
def get_coaching(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ select coaching_id, teams.name, coaches.name, seasons.year
                    from coaching
                    inner join teams on teams.team_id=coaching.team_id
                    inner join coaches on coaches.coach_id=coaching.coach_id
                    inner join seasons on seasons.season_id=coaching.season_id"""
        cursor.execute(query)
        result = cursor.fetchall()
        return result
```

add_coaching

This function takes Team id, Season id and Coach id and add them to database with *INSERT* sql statement.

```
def add_coaching(self, team_id, coach_id, season_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ INSERT INTO COACHING (TEAM_ID, COACH_ID, SEASON_ID) VALUES (%s, %s, %s) """
        cursor.execute(query, (team_id, coach_id, season_id))
        connection.commit()
```

seach_coaching

This method returns the matching coaching row with *WHERE* and *SELECT* statements.

```
def search_coaching(self, term):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""select coaching_id, teams.name, coaches.name, seasons.year
                    from coaching
                    inner join teams on teams.team_id=coaching.team_id
                    inner join coaches on coaches.coach_id=coaching.coach_id
                    inner join seasons on seasons.season_id=coaching.season_id
                    WHERE coaches.name LIKE '%s' OR teams.name LIKE '%s'""" % (('s'+term+'s'), ('s'+term+'s'))
        cursor.execute(query)
        connection.commit()
        coachlist = [(key, team, name, year)
                      for key, team, name, year in cursor]

    return coachlist
```

delete_coaching

Method takes id of the item as parameter. With *WHERE* statement it finds item that we want to delete.

```
def delete_coaching(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor=connection.cursor()
        query = """
            DELETE FROM COACHING
            WHERE COACHING_ID = %s"""
        cursor.execute(query, [id])
        connection.commit()
```

update_coaching

Like add function but in addition takes id value of the row to update.

```
def update_coaching(self, coaching_id, team_id, coach_id, season_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE COACHING
            SET TEAM_ID = %s,
            COACH_ID = %s,
            SEASON_ID = %s
            WHERE COACHING_ID = %s"""
        cursor.execute(query, [team_id, coach_id, season_id, coaching_id])
        connection.commit()
```

2.2.3 Parts Implemented by Kubilay Karpal

I implemented country, league and stadium entities and belonging operations. I created countries, leagues and stadiums classes in order to implement demanded operations. All these classes contains same basic methods which listed below.

- **Initialize table methods** This operations basically run a query to create related table.
- **Add methods** This methods take variables that represent columns in table and perform a insert operation.
- **Delete methods** This method takes the entity's primary key and delete it from database. Users reach delete function from listing pages and they do not naturally interact with the primary keys, this information kept but hidden.
- **Update methods** Similar with add methods this methods also take entity's fields as parameters but also an entity ID which corresponds to the primary key in the table is given. Same as in the delete operation, these keys are invisible to users.
- **Get Entity Methods** These methods take an entity ID and returns the entities all columns. These are actually helper functions that used by another entities since some table have foreign keys and had to reach related name or etc with these keys.
- **Get Entities Methods** Mostly used in list pages these methods simply returns all entries belong to a entity. Also these functions used in to Add / Delete operations of entities that have foreign keys since they need to list all options to the users as a dropdown etc. These methods do not take parameters.
- **Search Methods** Search methods designed to search by name in all three entities as case sensitive. They basically takes a string that represents search words and returns all related entries.

Note: The difference between single and all get functions is not limited with the number of elements returned. There is also difference between returned element's properties. Single get functions returns values as in the table but get entities methods change foreign key IDs with more understandable variables. (*country_id* changed with

country_name in *LEAGUES* table for example.) Because get entities methods used to list entities to users while get entity methods used by another back-end functions.

Delete and Update Operations and Their Form

I want to implement delete and update functions in list page in a such way that users can easily reach. In order to archive this I placed delete and update buttons following the entries in list page.







	Abbreviation	Operations
	CA	 
	TR	 
	US	 

Fig. 2.4: Users can directly delete entries or reach their update pages.

In delete button I used a hidden form value to send the primary key with POST request. But in update button I just put the ID of element that want to deleted in to URL. In this way I achieved the function that I want, users were able to delete / update entries by just clicking the corresponding button without entering a key value or an attribute like name etc.

```
<form action="{ { url_for('countries') } }" method="post"
      role="form" style="display: inline">
  <input value="{ {key} }" name="id" type="hidden" />
  <button class="btn btn-primary btn-sm" name="Delete" type="submit">
    <span class="glyphicon glyphicon-trash" >
  </button>
</form>
<form action="{ {url_for('country_edit', country_id=key)} }" method="get"
      role="form" style="display:inline">
  <button class="btn btn-primary btn-sm" name="Update" type="submit">
    <span class="glyphicon glyphicon-wrench" ><
  </button>
</form>
```

Country Implementation

I designed countries class in order to perform operations in my countries table. Country is a core entity in our database and used in some tables as a foreign key. Also I designed a country class to represent a row data of a country except for primary key.

Note: After implementing country entity and some of league functions I realized that using a class for holding entity information and using it as a parameter in functions is not a good way to maintain the operations. In other entities I did not use classes for entities / methods instead I used column variables as separate parameters.

Countires Table

In our database countries table has following columns

- ***COUNTRY_ID* as serial type and primary key** This is the primary key of the table
- ***NAME* as varchar(50) and not null** This column holds the name of the country and it can't be null
- ***ABBREVIATION* a varchar(5)** This column holds the abbrevitaion of the country (like US, UK etc.)

Since this is a core entity, it does not has a foreing key.

add_country Method

This method takes a country object as a parameter and insert it into database.

Here is the code block that does the add operation in database using INSERT command:

```
def add_country(self, country):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            INSERT INTO COUNTRIES (NAME, ABBREVIATION)
            VALUES (%s, %s) """ , (country.name, country. abbreviation))
        connection.commit()
```

delete_country Method

This method takes a country id (which is a primary key of countries table actually) and deletes if from database. To match the country on database *WHERE* statement used on country id column.

Here is the code block that perform delete operation on countries table.

```
def delete_country(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ DELETE FROM COUNTRIES WHERE COUNTRY_ID =%s """
        cursor.execute(query, [id])
        connection.commit()
```

update_country Method

This method works in a similar fashion with add function, it takes one more argument which is the *country id*. The given *Country* object is parsed and the row that related with country id argument is updated with tihs parsed information.

```
def update_country(self, country_id, country):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE COUNTRIES
            SET NAME = %s, ABBREVIATION = %s
            WHERE COUNTRY_ID = %s """
        cursor.execute(query, (country.name, country. abbreviation, country_id ))
        connection.commit()
```

get_country Method

This method is used by another classes. It is main function is the provide all columns related with a foreing key which consists a *country id*. It does simply run *SELECT* query with *WHERE* statement

to match *country id*.

```
def get_country(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT * FROM COUNTRIES WHERE COUNTRY_ID =%s """
        cursor.execute(query, [id])

        connection.commit()
        result = cursor.fetchone()
        country = Country(result[1], result[2])
    return country
```

get_countries Method

Similar to *get_country* methods runs a *SELECT* on countries table but this time without a specific ID. Simply it returns all countries in database without taking a parameter.

```
def get_countries(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT * FROM COUNTRIES ORDER BY NAME"""
        cursor.execute(query)
        connection.commit()
        countries = [(key, Country(name, abbreviation))
                     for key, name, abbreviation in cursor]

    return countries
```

search_countries Method

This method takes a string and search in countries table by matching this string which is the search phrase actually on the name column and returns a list of matching countries.

```
def search_countries(self, search_terms):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT * FROM COUNTRIES WHERE NAME LIKE '%s' ORDER BY NAME"""
                                                % (('%' + search_terms + '%'))
        cursor.execute(query)
        connection.commit()
        countries = [(key, Country(name, abbreviation))
                     for key, name, abbreviation in cursor]
    return countries
```

League Implementation

League is an important entity in American Football Database project because all the teams, matches, coaches, officials are specific for a league.

Leagues Table

Leagues table consists of following columns:

- **LEAGUE_ID** as serial type and primary key This is the primary key of the table
- **NAME** as varchar(100) and not null This column holds the name of the league and it can't be null
- **ABBREVIATION** a varchar(10) This column holds the abbreviation of the league (like NFL)

- ***COUNTRY_ID* as integer type, not null and references to countries table** This is foreign key to countries table, represent the country that the league belongs to

add_league Method

This method takes a league object and performs *INSERT* operation onto database.

```
def add_league(self, league):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            INSERT INTO LEAGUES (NAME, ABBREVIATION, COUNTRY_ID)
            VALUES (%s, %s, %s) """
            (league.name, league.abbreviation, league.countryID))
        connection.commit()
```

delete_league Method

This method takes a *league_id* and deletes corresponding row from database using *DELETE* operation.

```
def delete_league(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ DELETE FROM LEAGUES WHERE LEAGUE_ID = %s """
        cursor.execute(query, [id])
        connection.commit()
```

update_league Method

Takes an *league_id* and match the row in database then updates all columns with given parameters.

```
def update_league(self, league_id, name, abbreviation, country_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE LEAGUES
            SET NAME = %s, ABBREVIATION = %s, COUNTRY_ID = %s
            WHERE LEAGUE_ID = %s """
        cursor.execute(query, (name, abbreviation, country_id, league_id))
        connection.commit()
```

get_league Method

This method is an helper function to other entities which hold *league_id* as a foreign key. It simply takes an *league_id* and returns corresponding league information.

```
def get_league(self, league_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """SELECT * FROM LEAGUES
            WHERE LEAGUE_ID = %s """
        cursor.execute(query, [league_id])
        connection.commit()

    league_id, name, abbreviation, country_id = cursor.fetchone()
    return league_id, name, abbreviation, country_id
```


get_leagues Method

This method used to fetch all leagues from the database. It does not take a parameter and as a return value it returns the list of leagues information in the database.

```
def get_leagues(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT L.LEAGUE_ID, L.NAME, L.ABBREVIATION, C.NAME
                FROM LEAGUES L
                LEFT JOIN COUNTRIES C ON (L.COUNTRY_ID = C.COUNTRY_ID)
                """

        cursor.execute(query)
        connection.commit()

        leagues = [(league_id, name, abbreviation, country_name)
                    for league_id, name, abbreviation, country_name in cursor]

    return leagues
```

search_leagues Method

Search countries method runs a *SELECT* argument with *WHERE* argument which compare the given input parameter with leagues' names with *LIKE* option. The results returned as a list.

```
def search_leagues(self, search_terms):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT L.LEAGUE_ID, L.NAME, L.ABBREVIATION, C.NAME
                FROM LEAGUES L
                LEFT JOIN COUNTRIES C ON (L.COUNTRY_ID = C.COUNTRY_ID)
                WHERE L.NAME LIKE '%s' ORDER BY L.NAME"""
                % ((''+search_terms+'%'))

        cursor.execute(query)
        connection.commit()

        leagues = [(league_id, name, abbreviation, country_name)
                    for league_id, name, abbreviation, country_name in cursor]

    return leagues
```

Note: *LEAGUES* table holds the countries where stadiums located by referencing *COUNTRIES* table. This information established with storing *country_id* as a foreign key. But this ID number is meaningless to users. In order to properly show country information with country name **LEFT JOIN** method used and countries table joined on stadiums table with *country_id* in common.

Stadium Implementation

Stadium is a small entity that used to store records of stadiums.

Note: We first planned to give a reference to stadium in *MATCHES* table but we could not able to implement time due to lack of time.

Stadium Table

Stadium table consists of following columns:

- ***STADIUM_ID* as serial type and primary key** This is the primary key of the table
- ***NAME* as varchar(100) and not null** This column holds the name of the stadium and it can't be null
- ***CAPACITY* as integer** This column used to store capacity of stadium if given.
- ***COUNTRY_ID* as integer type, not null and references to countries table** This is foreign key to COUNTRIES table, represent the country where stadium placed.
- ***TEAM_ID* as integer type, not null and references to teams table** This is foreign key to TEAMS table, represent the owner team of the stadium.

add_stadium Method

This method takes variables corresponds to columns of *STADIUMS* and insert new row to the table.

```
def add_stadium(self, name, capacity, country_id, team_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """
            INSERT INTO STADIUMS (NAME, CAPACITY, COUNTRY_ID, TEAM_ID)
            VALUES (%s, %s, %s, %s) """
        cursor.execute(query, (name, capacity, country_id, team_id))
        connection.commit()
```

delete_stadium Method

This method takes an *stadium_id* and deletes corresponding row from database.

```
def delete_stadium(self, stadium_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ DELETE FROM STADIUMS WHERE STADIUM_ID = %s """
        cursor.execute(query, [stadium_id])
        connection.commit()
```

update_stadium Method

This method takes an *stadium_id* and new information that belongs to this entry.

```
def update_stadium(self, stadium_id, name, capacity, country_id, team_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE STADIUMS
            SET NAME=%s, CAPACITY=%s, COUNTRY_ID=%s, TEAM_ID=%s
            WHERE STADIUM_ID = %s """
        cursor.execute(query, (name, capacity, country_id, team_id, stadium_id))
        connection.commit()
```

get_stadium Method

Using fetchone function, this method returns information of an stadium whose *stadium_id* given as parameter.

```
def get_stadium(self, stadium_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT * FROM STADIUMS
                WHERE STADIUM_ID =%s """
        cursor.execute(query, (stadium_id))
        connection.commit()

        stadium_id, name, capacity, country_id, team_id = cursor.fetchone()
    return stadium_id, name, capacity, country_id, team_id
```

get_stadiums Method

Without an input parameter this method returns all stadiums and information belongs to that stadiums by using fetchall function. **LEFT JOIN** used in order to get league's and country's name.

```
def get_stadiums(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT S.STADIUM_ID, S.NAME, S.CAPACITY, C.NAME, T.NAME
                FROM STADIUMS S
                LEFT JOIN COUNTRIES C ON (S.COUNTRY_ID = C.COUNTRY_ID)
                LEFT JOIN TEAMS T ON (S.TEAM_ID = T.TEAM_ID)
                """
        cursor.execute(query)
        connection.commit()

        stadiums = [(key, name, capacity, country, team)
                    for key, name, capacity, country, team in cursor]

    return stadiums
```

search_stadiums Method

This method searches stadiums with stadium name and return results in a same fashion with *get_stadiums* method. Again **LEFT JOIN** used in order to get league's and country's name.

```
def search_stadiums(self, search_terms):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT S.STADIUM_ID, S.NAME, S.CAPACITY, C.NAME, T.NAME
                FROM STADIUMS S
                LEFT JOIN COUNTRIES C ON (S.COUNTRY_ID = C.COUNTRY_ID)
                LEFT JOIN TEAMS T ON (S.TEAM_ID = T.TEAM_ID)
                WHERE S.NAME LIKE '%s' ORDER BY S.NAME"""
                % ((''+search_terms+'%'))

        cursor.execute(query)
        connection.commit()
        stadiums = [(key, name, capacity, country, team)
                    for key, name, capacity, country, team in cursor]

    return stadiums
```

2.2.4 Parts Implemented by Seda Yıldırım

The following three tables were implemented: **Fixtures**, **Player Statistics**, and **Team Statistics**. The tabs Fixtures and Statistics can be seen on the navigation bar above the site interface. The classes for the respective tables were created with the same method in mind. All classes include the methods below.

- *Initialize Table*: Run a query to create the table.
- *Add methods*: Add a new value to the respective tables.
- *Delete methods*: Delete the selected entry from a table.
- *Update methods*: Update the selected entry.
- *Get Single Entity Methods*: Take an entity ID and return the whole row.
- *Get Multiple Entities Methods*: Return all entries of an entity. Does not take parameters.
- *Search Methods*: Search methods by name. Case sensitive.

Fixtures Table

Fixtures table was implemented to hold the fixture data of the teams. It has *Fixture_ID* as a **primary key**, and *Season_ID* and *Team_ID* as a **foreign key**. It also has *points* data as local data.

```
def initialize_tables(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS FIXTURES
            (
                FIXTURE_ID SERIAL NOT NULL PRIMARY KEY,
                SEASON_ID INTEGER NOT NULL REFERENCES SEASONS(SEASON_ID),
                TEAM_ID INTEGER NOT NULL REFERENCES TEAMS(TEAM_ID),
                POINTS INTEGER NOT NULL
            )
            """)
        connection.commit()
```

add_fixture Method

This method takes the respective queries and adds the resulting fixture to the database. This operation is done by INSERT INTO feature in SQL. The said code is shown below.

```
def add_fixture(self, season_id, team_id, points):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ INSERT INTO FIXTURES (SEASON_ID, TEAM_ID, POINTS) VALUES
            (%s, %s, %s) """
        cursor.execute(query, (season_id, team_id, points))
        connection.commit()
```

delete_fixture Method

This method takes the *Fixture_ID* of a query and deletes the resulting fixture from the database. This operation is done by DELETE FROM feature in SQL. The said code is shown below.

```
def delete_fixture(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            DELETE FROM FIXTURES
            WHERE FIXTURE_ID = %s""",
            id)
        connection.commit()
```

update_fixture Method

This method takes the Fixture_ID of a query and updates the said entry by simply calling the UPDATE feature in SQL.

```
def update_fixture(self, fixture_id, season_id, team_id, points):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE FIXTURES
                     SET SEASON_ID = %s,
                     TEAM_ID = %s,
                     POINTS = %s
                     WHERE FIXTURE_ID = %s"""
        cursor.execute(query, (season_id, team_id, points, fixture_id))
        connection.commit()
```

search_fixture Method

This method provides the user with all the columns related to the search query. It runs a SELECT query with a WHERE statement to match *Fixture_ID*. It uses JOIN feature of SQL to display the proper results.

```
def search_fixture(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT FIXTURE_ID, SEASONS.YEAR, TEAMS.NAME, POINTS
                 FROM FIXTURES
                 INNER JOIN SEASONS ON SEASONS.SEASON_ID=FIXTURES.SEASON_ID
                 INNER JOIN TEAMS ON TEAMS.TEAM_ID=FIXTURES.TEAM_ID
                 WHERE TEAMS.NAME LIKE '%s'""" % ('%'+id+'%')
        cursor.execute(query)
        connection.commit()

        result = cursor.fetchall()
        return result
```

get_fixtures Method

This method simply returns all the fixtures in the database. It uses LEFT JOIN feature of SQL to get **season** and **team name** data from the foreign keys.

```
def get_fixtures(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT F.FIXTURE_ID, S.YEAR, T.NAME, F.POINTS
                 FROM FIXTURES F
                 LEFT JOIN SEASONS S ON (F.SEASON_ID = S.SEASON_ID)
                 LEFT JOIN TEAMS T ON (F.TEAM_ID = T.TEAM_ID)
                 ORDER BY S.YEAR ASC"""
        cursor.execute(query)
        connection.commit()

        fixtures = [(key, season, team, points)
                    for key, season, team, points in cursor]
        return fixtures
```

Player Statistics Table

Player Statistics table was implemented to hold the various statistics data of the players in the database. It has *Statistic_ID* as a **primary key**, and *Season_ID* and *Player_ID* as a **foreign key**. It also has *tackles* and *penalties*

data as local data. The following code initializes the Team Statistics table.

```
def initialize_tables(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS STATISTICSP
            (
                STATISTIC_ID SERIAL NOT NULL PRIMARY KEY,
                SEASON_ID INTEGER NOT NULL REFERENCES SEASONS(SEASON_ID),
                PLAYER_ID INTEGER NOT NULL REFERENCES PLAYERS(PLAYER_ID),
                tackles INTEGER NOT NULL,
                penalties INTEGER NOT NULL
            )
        """)
        connection.commit()
```

add_statistic_player Method

This method takes the respective queries and adds the resulting statistics to the database. This operation is done by INSERT INTO feature in SQL. The said code is shown below.

```
def add_statistic_player(self, season_id, player_id, tackles, penalties):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ INSERT INTO STATISTICSP (SEASON_ID, PLAYER_ID, tackles,
            penalties) VALUES (%s, %s, %s, %s) """
        cursor.execute(query, (season_id, player_id, tackles, penalties))
        connection.commit()
```

delete_statistic_player Method

This method takes the Statistic_ID of a query and deletes the resulting statistic from the database. This operation is done by DELETE FROM feature in SQL. The said code is shown below.

```
def delete_statistic_player(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            DELETE FROM STATISTICSP
            WHERE STATISTIC_ID = %s""",
            id)
        connection.commit()
```

update_statistic_player Method

This method takes the Statistic_ID of a query and updates the said entry by simply calling the UPDATE feature in SQL.

```
def update_statistic_player(self, statistic_id, season_id, player_id, tackles,
    penalties):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE STATISTICSP
            SET SEASON_ID = %s,
            PLAYER_ID = %s,
            TACKLES = %s,
            PENALTIES = %s
            WHERE STATISTIC_ID = %s"""
```

```

        cursor.execute(query, (season_id, player_id, tackles, penalties,
                                statistic_id))
        connection.commit()

```

search_statistic_player Method

This method provides the user with all the columns related to the search query. It runs a SELECT query with a WHERE statement to match *Statistic_ID*. It uses JOIN feature of SQL to display the proper results.

```

def search_statistic_player(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT STATISTIC_ID, SEASONS.YEAR, PLAYERS.NAME, TACKLES, PENALTIES
        FROM STATISTICSP
        INNER JOIN SEASONS ON SEASONS.SEASON_ID=STATISTICSP.SEASON_ID
        INNER JOIN PLAYERS ON PLAYERS.PLAYER_ID=STATISTICSP.PLAYER_ID
        WHERE PLAYERS.NAME LIKE '%s'""" % ('%'+id+'%')

        cursor.execute(query)
        connection.commit()

        result = cursor.fetchall()
        return result

```

get_statistics_player Method

This method simply returns all the player statistics in the database. It uses LEFT JOIN feature of SQL to get **season** and **player name** data from the foreign keys.

```

def get_statistics_player(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT S.STATISTIC_ID, SS.YEAR, P.NAME, S.TACKLES, S.PENALTIES
        FROM STATISTICSP S
        LEFT JOIN SEASONS SS ON (S.SEASON_ID = SS.SEASON_ID)
        LEFT JOIN PLAYERS P ON (S.PLAYER_ID = P.PLAYER_ID)
        ORDER BY SS.YEAR ASC"""

        cursor.execute(query)
        connection.commit()

        statisticsp = [(key, season, player, tackles, penalties)
                        for key, season, player, tackles, penalties in cursor]
        return statisticsp

```

Team Statistics Table

Team Statistics table was implemented to hold the various statistics data of the teams in the database. It has *Statistic_ID* as a **primary key**, and *Season_ID* and *Team_ID* as a **foreign key**. It also has *touchdowns* and *rushing yards* data as local data. The following code initializes the Team Statistics table.

```

def initialize_tables(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS STATISTICST
            (
                STATISTIC_ID SERIAL NOT NULL PRIMARY KEY,
                SEASON_ID INTEGER NOT NULL REFERENCES SEASONS(SEASON_ID),
                TEAM_ID INTEGER NOT NULL REFERENCES TEAMS(TEAM_ID),

```

```
        touchdowns INTEGER NOT NULL,
        rushingYards INTEGER NOT NULL
    )
    """
    connection.commit()
```

add_statistic_team Method

This method takes the respective queries and adds the resulting statistics to the database. This operation is done by INSERT INTO feature in SQL. The said code is shown below.

```
def add_statistic_team(self, season_id, team_id, touchdowns, rushingYards):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ INSERT INTO STATISTICST (SEASON_ID, TEAM_ID, touchdowns,
            rushingYards) VALUES (%s, %s, %s, %s) """
        cursor.execute(query, (season_id, team_id, touchdowns, rushingYards))
        connection.commit()
```

delete_statistic_team Method

This method takes the Statistic_ID of a query and deletes the resulting statistic from the database. This operation is done by DELETE FROM feature in SQL. The said code is shown below.

```
def delete_statistic_team(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        cursor.execute("""
            DELETE FROM STATISTICST
            WHERE STATISTIC_ID = %s""",
            id)
        connection.commit()
```

update_statistic_team Method

This method takes the Statistic_ID of a query and updates the said entry by simply calling the UPDATE feature in SQL.

```
def update_statistic_team(self, statistic_id, season_id, team_id, touchdowns,
    rushingYards):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE STATISTICST
            SET SEASON_ID = %s,
            TEAM_ID = %s,
            TOUCHDOWNS = %s,
            RUSHINGYARDS = %s
            WHERE STATISTIC_ID = %s"""
        cursor.execute(query, (season_id, team_id, touchdowns, rushingYards,
            statistic_id))
        connection.commit()
```

search_statistic_team Method

This method provides the user with all the columns related to the search query. It runs a SELECT query with a WHERE statement to match *Statistic_ID*. It uses JOIN feature of SQL to display the proper results.


```
def search_statistic_team(self, id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT STATISTIC_ID, SEASONS.YEAR, TEAMS.NAME, TOUCHDOWNS,
            RUSHINGYARDS
            FROM STATISTICST
            INNER JOIN SEASONS ON SEASONS.SEASON_ID=STATISTICST.SEASON_ID
            INNER JOIN TEAMS ON TEAMS.TEAM_ID=STATISTICST.TEAM_ID
            WHERE TEAMS.NAME LIKE '%s'""" % ('%'+id+'%')
        cursor.execute(query)
        connection.commit()

        result = cursor.fetchall()
    return result
```

get_statistics_team Method

This method simply returns all the team statistics in the database. It uses LEFT JOIN feature of SQL to get **season** and **team name** data from the foreign keys.

```
def get_statistics_team(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query="""SELECT S.STATISTIC_ID, SS.YEAR, T.NAME, S.TOUCHDOWNS,
            S.RUSHINGYARDS
            FROM STATISTICST S
            LEFT JOIN SEASONS SS ON (S.SEASON_ID = SS.SEASON_ID)
            LEFT JOIN TEAMS T ON (S.TEAM_ID = T.TEAM_ID)
            ORDER BY SS.YEAR ASC"""
        cursor.execute(query)
        connection.commit()

        statisticst = [(key, season, team, touchdowns, rushingYards)
            for key, season, team, touchdowns, rushingYards in cursor]
    return statisticst
```

2.2.5 Parts Implemented by Sefa Eren Şahin

Players, Teams and Squad tables are implemented.

Players Table

This table consists of 4 columns

Column Name	Data Type	Key
PLAYER_ID	serial	PRIMARY KEY
NAME	varchar	none
BIRTHDAY	date	none
POSITION	varchar	none

Table Initialization

Table is created by following sql code:

```
CREATE TABLE IF NOT EXISTS PLAYERS
(
    PLAYER_ID serial NOT NULL PRIMARY KEY,
    NAME varchar(100) NOT NULL,
    BIRTHDAY date NOT NULL,
```

```
        POSITION varchar(100) NOT NULL
    )
```

Selection

If “/players” route is loaded by GET method, players are going to be selected and will be printed to players.html:

```
@app.route('/players', methods=['GET', 'POST'])
def players():
    if request.method == 'GET':
        return render_template('players.html', players=app.players.select_players())
    else:
        name = request.form['name']
        birthday = request.form['birthday']
        position = request.form['position']
        app.players.add_player(name, birthday, position)
    return redirect(url_for('players'))
```

Selection operation is done by the following function which is in players.py:

```
def select_players(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT * FROM PLAYERS ORDER BY PLAYER_ID """
        cursor.execute(query)
        players = cursor.fetchall()
    return players
```

Insert Operation

A route is defined in order to use Player Adding html page:

```
@app.route('/players/add', methods=['GET', 'POST'])
def add_players():
    return render_template('players_add.html')
```

After the form is filled and submitted in page, form action directs to the following route:

```
@app.route('/players', methods=['GET', 'POST'])
def players():
    if request.method == 'GET':
        return render_template('players.html', players=app.players.select_players())
    else:
        name = request.form['name']
        birthday = request.form['birthday']
        position = request.form['position']
        app.players.add_player(name, birthday, position)
    return redirect(url_for('players'))
```

If “/players” route is loaded by POST method, which is the player addition form’s method, player will be added and route will redirect to itself again. If route is loaded by GET method, players.html page will be opened up.

Insertion operation is done by the following function which is in players.py:

```
def add_player(self, name, birthday, position):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ INSERT INTO PLAYERS (NAME, BIRTHDAY, POSITION)
                    VALUES (%s, %s, %s) """
        cursor.execute(query, (name, birthday, position))
        connection.commit()
```

Update Operation

In update operation, route is defined uniquely for the corresponding tuple's player_id:

```
@app.route('/players/update/<player_id>', methods=['GET', 'POST'])
def update_players(player_id):
    if request.method == 'GET':
        return render_template('players_edit.html',
                               player = app.players.get_player(player_id))
    else:
        name = request.form['name']
        birthday = request.form['birthday']
        position = request.form['position']
        app.players.update_player(player_id, name, birthday, position)
        return redirect(url_for('players'))
```

If the route is loaded by GET method, player with corresponding player_id will be selected to update and route will be directed to players_edit.html:

```
def get_player(self, player_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT * FROM PLAYERS WHERE PLAYER_ID = %s """
        cursor.execute(query, [player_id])
        player = cursor.fetchall()
        return player
```

The form's action in players_edit.html redirects form to the current route. Since form's method is POST, route is loaded by POST method. Values are requested from form and the update function is called. After that, route redirects to players page. Update operation is done by the following function in players.py:

```
def update_player(self, player_id, name, birthday, position):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE PLAYERS
                     SET NAME = %s,
                     BIRTHDAY = %s,
                     POSITION = %s
                     WHERE
                     PLAYER_ID = %s """
        cursor.execute(query, (name, birthday, position, player_id))
        connection.commit()
```

Delete Operation

Delete operation is very similar to Update operation. Like update, in delete operation, route is defined uniquely for the corresponding tuple's player id.:

```
@app.route('/players/delete/<player_id>', methods=['GET', 'POST'])
def delete_players(player_id):
    app.players.delete_player(player_id)
    return redirect(url_for('players'))
```

After the player is deleted, route redirects to players page. Delete operation is done by the following function in players.py:

```
def delete_player(self, player_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ DELETE FROM PLAYERS
                     WHERE PLAYER_ID = %s """
```

```
cursor.execute(query, [player_id])
connection.commit()
```

Search Operation

A route is defined in order to search players by player name. Search form is in players.html:

```
@app.route('/players/search', methods = ['GET', 'POST'])
def search_players():
    if request.method == 'GET':
        return redirect(url_for('players_search.html'))
    else:
        searchname = request.form['nametosearch']
        return render_template('players_search.html',
                               players = app.players.search_player(searchname))
```

Since the form has POST method, after the submission, search name will be requested from form. After searching, results will be listed in players_search.html.

Searching is done by the following function in players.py:

```
def search_player(self, name):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT * FROM PLAYERS WHERE NAME LIKE %s
                    ORDER BY PLAYER_ID """
        cursor.execute(query, ['%'+name+'%'])
        players = cursor.fetchall()
        return players
```

Teams Table

This table consists of 4 columns

Column Name	Data Type	Key
TEAM_ID	serial	PRIMARY KEY
NAME	varchar	none
LEAGUE_ID	date	FK LEAGUES(LEAGUE_ID)

Table Initialization

Table is created by following sql code:

```
CREATE TABLE IF NOT EXISTS TEAMS
(
    TEAM_ID serial NOT NULL PRIMARY KEY,
    NAME varchar(100) NOT NULL,
    LEAGUE_ID int NOT NULL REFERENCES LEAGUES(LEAGUE_ID)
)
```

Selection

If “/teams” route is loaded by GET method, teams are going to be selected and will be printed to teams.html:

```
@app.route('/teams', methods=['GET', 'POST'])
def teams():
    if request.method == 'GET':
        return render_template('teams.html', teams = app.teams.select_teams())
```

```

else:
    name = request.form['name']
    league_id = request.form['league_id']
    app.teams.add_team(name, league_id)
return redirect(url_for('teams'))

```

Selection operation is done by the following function which is in teams.py:

```

def select_teams(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT * FROM TEAMS ORDER BY TEAM_ID """
        cursor.execute(query)
        connection.commit()

        teams = cursor.fetchall()
    return teams

```

Insert Operation

A route is defined in order to use Team Adding html page Leagues are selected and added to Dropdown Menu since League_id is foreign key.:

```

@app.route('/teams/add', methods=['GET', 'POST'])
def add_teams():
    return render_template('teams_add.html', leagues = app.leagues.get_leagues())

```

After the form is filled and submitted in page, form action directs to the following route:

```

@app.route('/teams', methods=['GET', 'POST'])
def teams():
    if request.method == 'GET':
        return render_template('teams.html', teams = app.teams.select_teams())
    else:
        name = request.form['name']
        league_id = request.form['league_id']
        app.teams.add_team(name, league_id)
    return redirect(url_for('teams'))

```

If “/teams” route is loaded by POST method, which is the team addition form’s method, team will be added and route will redirect to itself again. If route is loaded by GET method, teams.html page will be opened up.

Insertion operation is done by the following function which is in teams.py:

```

def add_team(self, name, league_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ INSERT INTO TEAMS (NAME, LEAGUE_ID) VALUES (%s, %s) """
        cursor.execute(query, (name, league_id))
        connection.commit()

```

Update Operation

In update operation, route is defined uniquely for the corresponding tuple’s team_id.:

```

@app.route('/teams/update/<team_id>', methods=['GET', 'POST'])
def update_teams(team_id):
    if request.method == 'GET':
        return render_template('teams_edit.html', team = app.teams.get_team(team_id),
                                leagues = app.leagues.get_leagues())
    else:

```

```
name = request.form['name']
league_id = request.form['league_id']
app.teams.update_team(team_id, name, league_id)
return redirect(url_for('teams'))
```

If the route is loaded by GET method, team with corresponding team_id will be selected to update and route will be directed to teams_edit.html:

```
def get_team(self, team_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT * FROM TEAMS WHERE TEAM_ID = %s """
        cursor.execute(query, [team_id])
        connection.commit()
        team = cursor.fetchall()
    return team
```

The form's action in teams_edit.html redirects form to the current route. Since form's method is POST, route is loaded by POST method. Values are requested from form and the update function is called. After that, route redirects to teams page. Update operation is done by the following function in teams.py:

```
def update_team(self, team_id, name, league_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE TEAMS
                    SET NAME = %s,
                    LEAGUE_ID = %s
                    WHERE
                    TEAM_ID = %s """
        cursor.execute(query, (name, league_id, team_id))
        connection.commit()
```

Delete Operation

Delete operation is very similar to Update operation. Like update, in delete operation, route is defined uniquely for the corresponding tuple's team id.:

```
@app.route('/teams/delete/<team_id>', methods=['GET', 'POST'])
def delete_teams(team_id):
    app.teams.delete_team(team_id)
    return redirect(url_for('teams'))
```

After the team is deleted, route redirects to players page. Delete operation is done by the following function in teams.py:

```
def delete_team(self, team_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ DELETE FROM TEAMS WHERE TEAM_ID = %s """
        cursor.execute(query, [team_id])
        connection.commit()
```

Search Operation

A route is defined in order to search teams by team name. Search form is in teams.html:

```
@app.route('/teams/search', methods = ['GET', 'POST'])
def search_teams():
    if request.method == 'GET':
        return redirect(url_for('teams_search.html'))
    else:
```

```
searchname = request.form['nametosearch']
return render_template('teams_search.html',
teams = app.teams.search_team(searchname))
```

Since the form has POST method, after the submission, search name will be requested from form. After searching, results will be listed in teams_search.html.

Searching is done by the following function in teams.py:

```
def search_team(self, name):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT * FROM TEAMS WHERE NAME LIKE %s ORDER BY TEAM_ID """
        cursor.execute(query, ['%'+name+'%'])
        teams = cursor.fetchall()
    return teams
```

Squads Table

This table consists of 4 columns

Column Name	Data Type	Key
SQUAD_ID	serial	PRIMARY KEY
TEAM_ID	int	FK TEAMS(Team_ID)
PLAYER_ID	int	FK PLAYERS(PLAYER_ID)
KIT_NO	int	none

Table Initialization

Table is created by following sql code:

```
CREATE TABLE IF NOT EXISTS SQUADS
(
    SQUAD_ID serial NOT NULL PRIMARY KEY,
    TEAM_ID int NOT NULL REFERENCES TEAMS(Team_ID),
    PLAYER_ID int NOT NULL UNIQUE REFERENCES PLAYERS(PLAYER_ID),
    KIT_NO int NOT NULL
)
```

Selection

If “/squads” route is loaded by GET method, squads are going to be selected and will be printed to squads.html:

```
@app.route('/squads', methods=['GET', 'POST'])
def squads():
    if request.method == 'GET':
        return render_template('squads.html', teams = app.squads.get_teams(),
squads = app.squads.show_squads())
    else:
        team_id = request.form['team_id']
        player_id = request.form['player_id']
        kit_no = request.form['kit_no']
        app.squads.add_squad(team_id, player_id, kit_no)
    return redirect(url_for('squads'))
```

Selection is made in a way that, instead of using team_id and player_id, team name and player name corresponding to their id's are selected using LEFT JOIN. Selection operation is done by the following function which is in squads.py:

```
def show_squads(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT squad_id, teams.name, players.name, kit_no FROM SQUADS
                     LEFT JOIN TEAMS
                     ON SQUADS.TEAM_ID = TEAMS.TEAM_ID
                     LEFT JOIN PLAYERS
                     ON SQUADS.PLAYER_ID = PLAYERS.PLAYER_ID
                     ORDER BY SQUADS.TEAM_ID """
        cursor.execute(query)
        connection.commit()

        squads = cursor.fetchall()
        return squads
```

Insert Operation

A route is defined in order to use Squad Adding html page. Teams and Players are selected and added to Dropdown Menus since they're foreign keys.:

```
@app.route('/squads/add', methods=['GET', 'POST'])
def add_squads():
    return render_template('squads_add.html', teams = app.teams.select_teams(),
        players = app.squads.get_players())
```

After the form is filled and submitted in page, form action directs to the following route:

```
@app.route('/squads', methods=['GET', 'POST'])
def squads():
    if request.method == 'GET':
        return render_template('squads.html', teams = app.squads.get_teams(),
            squads = app.squads.show_squads())
    else:
        team_id = request.form['team_id']
        player_id = request.form['player_id']
        kit_no = request.form['kit_no']
        app.squads.add_squad(team_id, player_id, kit_no)
    return redirect(url_for('squads'))
```

If “/squads” route is loaded by POST method, which is the squad addition form’s method, team will be added and route will redirect to itself again. If route is loaded by GET method, squads.html page will be opened up.

Insertion operation is done by the following function which is in squads.py:

```
def add_squad(self, team_id, player_id, kit_no):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ INSERT INTO SQUADS (TEAM_ID, PLAYER_ID, KIT_NO)
                     VALUES (%s, %s, %s) """
        cursor.execute(query, (team_id, player_id, kit_no))
        connection.commit()
```

Update Operation

In update operation, route is defined uniquely for the corresponding tuple’s squad_id.:

```
@app.route('/squads/update/<squad_id>', methods=['GET', 'POST'])
def update_squads(squad_id):
    if request.method == 'GET':
        return render_template('squads_edit.html', squad=app.squads.get_squad(squad_id),
            teams = app.teams.select_teams(), players = app.players.select_players())
```



```

else:
    team_id = request.form['team_id']
    player_id = request.form['player_id']
    kit_no = request.form['kit_no']
    app.squads.update_squad(squad_id, team_id, player_id, kit_no)
    return redirect(url_for('squads'))

```

If the route is loaded by GET method, team with corresponding squad_id will be selected to update and route will be directed to squads_edit.html:

```

def get_squad(self, squad_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT * FROM SQUADS WHERE SQUAD_ID = %s """
        cursor.execute(query, [squad_id])
        connection.commit()
        squad = cursor.fetchall()
    return squad

```

The form's action in squads_edit.html redirects form to the current route. Since form's method is POST, route is loaded by POST method. Values are requested from form and the update function is called. After that, route redirects to squads page. Update operation is done by the following function in squads.py:

```

def update_squad(self, squad_id, team_id, player_id, kit_no):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ UPDATE SQUADS
                    SET
                    TEAM_ID = %s,
                    PLAYER_ID = %s,
                    KIT_NO = %s
                    WHERE
                    SQUAD_ID = %s """
        cursor.execute(query, (team_id, player_id, kit_no, squad_id))
        connection.commit()

```

Delete Operation

Delete operation is very similar to Update operation. Like update, in delete operation, route is defined uniquely for the corresponding tuple's squad id.:

```

@app.route('/squads/delete/<squad_id>', methods=['GET', 'POST'])
def delete_squads(squad_id):
    app.squads.delete_squad(squad_id)
    return redirect(url_for('squads'))

```

After the team is deleted, route redirects to squads page. Delete operation is done by the following function in squads.py:

```

def delete_squad(self, squad_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ DELETE FROM SQUADS WHERE SQUAD_ID = %s """
        cursor.execute(query, [squad_id])
        connection.commit()

```

Search Operation

A route is defined in order to search and filter squads by team name. Searching is made in a way that in squads.html, team names are selected and added to a dropdown list. And squads can be filtered by selecting team name. Search form is in squads.html:

```
@app.route('/squads/search', methods = ['GET', 'POST'])
def search_squads():
    if request.method == 'GET':
        return redirect(url_for('squads_search.html'), teams = app.squads.get_teams())
    else:
        team_id = request.form['name']
        return render_template('squads_search.html', teams = app.squads.get_teams(),
                               squads = app.squads.search_squad(team_id))
```

Team names are selected by the following function in `squads.py`. This function selects team names distinctly. To obtain team name corresponding to `team_id`, LEFT JOIN is used.:

```
def get_teams(self):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT DISTINCT teams.team_id, teams.name FROM SQUADS
                     LEFT JOIN TEAMS
                     ON SQUADS.TEAM_ID = TEAMS.TEAM_ID ORDER BY TEAM_ID """
        cursor.execute(query)
        connection.commit()
        teams = cursor.fetchall()
    return teams
```

Since the form has POST method, after the submission, search name will be requested from form. After searching, results will be listed in `squads_search.html`.

Searching is done by the following function in `squads.py`:

```
def search_squad(self, team_id):
    with dbapi2.connect(self.app.config['dsn']) as connection:
        cursor = connection.cursor()
        query = """ SELECT squad_id, teams.name, players.name, kit_no FROM SQUADS
                     LEFT JOIN TEAMS
                     ON SQUADS.TEAM_ID = TEAMS.TEAM_ID
                     LEFT JOIN PLAYERS
                     ON SQUADS.PLAYER_ID = PLAYERS.PLAYER_ID
                     WHERE SQUADS.TEAM_ID = %s
                     ORDER BY SQUADS.TEAM_ID """
        cursor.execute(query, [team_id])
        connection.commit()
        squad = cursor.fetchall()
    return squad
```

Installation Guide

American Football Database Project coded in Python using *Flask* web framework. In this part installation instructions of our project will be given.

3.1 Package Requirements

3.1.1 Python

Python 3.4.x required and it can be installed from <https://www.python.org/downloads/> page.

Note: Many Linux distributions comes with python3 package installed.

3.1.2 Flask

Flask can be downloaded from <http://flask.pocoo.org/> or could be simply installed via pip package manager via following command:

```
pip install Flask
```

3.1.3 Psycopg2

Psycopg2 can be downloaded from <http://initd.org/psycopg/> or could be installed via pip package manager via following command:

```
pip install psycopg2
```

3.1.4 PostgreSQL

PostgreSQL can be downloaded from <http://www.postgresql.org/download/>