
DBall Documentation

Release 0.5.7

ITUCSDB1515

January 14, 2016

1	How to Install?	3
1.1	User Guide	3
1.2	Developer Guide	36

Team ITUCSDB1515**Members**

- Oğuz Kerem Tural (150130125)
- Umut Can Özyar (150130022)
- Mert Şeker (150130119)
- Furkan Akgün (150130106)

DBall Database Application is prepared for baseball, a branch of sport especially popular in American culture with more than 300 hundred years of history. It can hold many of the statistical data that represents baseball as whole. It is easy to use, simple yet also give much more flexibility than any other application. In other terms, it directly responds to user. If user wants it complex it become like one. And more importantly it is multi functional and open source.

How to Install?

Just follow the following steps in order to install application.

- First go to the `www.python.org` and grab python (preferably version 3.4.3).
- Then install `flask`, `psycopg2`, `passlib` and `requests` packages through `pip`.
 - **PS.** You can use `pip install flask psycopg2 passlib requests` if `pip` is declared in your environment path.
- Then install PostgreSQL through `www.postgresql.org`
- Setup database, then import `init.sql` file into database through recovery.
- Fire up `server.py` and you are ready to roll!

Contents:

1.1 User Guide

DBall Application is designed to become user friendly, simple and clean. Any type of user no matter what level of their computer skills is targeted for this application. Addition to its simple design, it is designed to be multi-functional. More of its functions such as altering and registering new record available through registration. Still all of the record can be accessible through front view. DBall also provides a abstract interface for developers. With its robust REST API, developers can use our services in their programs easily. For further information about this topic please advance to *Developer Guide*.

1.1.1 Parts Implemented by Oğuz Kerem Tural

Main Area

Upon entering the application, user faces with this screen. It contains a navigation bar on top, a search box and two columns. Search box is not yet active. Still user can search each table from their singular views. From top navigation bar user can move across table views and login if it is not yet logged in. Also from right side column, user can be able to see latest changes on records have done by registered users.

Navigation Bar

From this area users can move thorough table views of front area. Also from right corner, where a door symbol seen, user can login to the application. If user already logged in, it can enter management area using drop down menu which replaces login button after log in operation. Using drop down menu user can advance between management and front pages and sign out when it is needed.

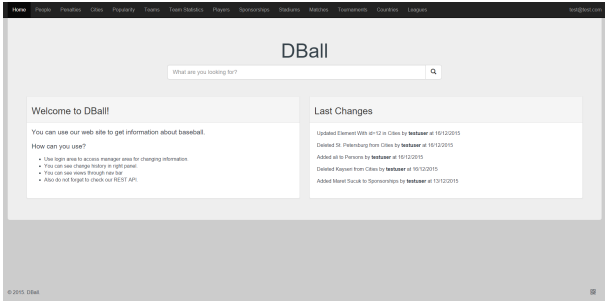


Fig. 1.1: Main screen of the application.

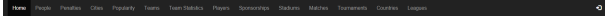


Fig. 1.2: Navigation bar before user logs in.

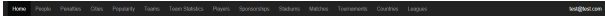


Fig. 1.3: Navigation bar after user logs in

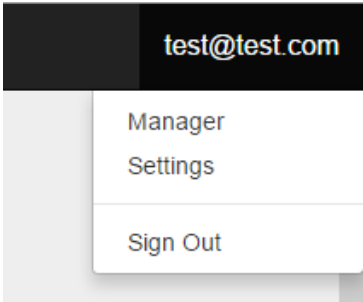


Fig. 1.4: This menu will appears when user logs in, instead of login button.

User System

User system in application is very basic and an abstract system that aims to prevent anonymous changes could have been done to database records. Every registered user has right to add, update or delete records where as unregistered users can only view, search and filter the records. Both user login and register operations are done using an Auth API service that has been provided by application itself. For further information about API please reference to *Developer Guide*.

Login Using Interface

To login using interface, user should click the button provided in navigation bar's top right corner with the door symbol on it. After click, a modal window will be shown which provides user name and password fields to user for log in operation.

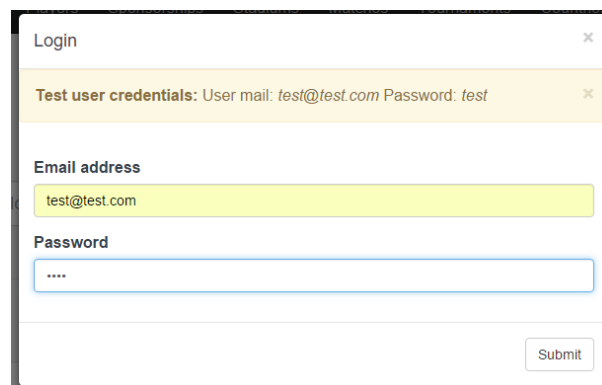
A screenshot of a 'Login' modal window. At the top, it says 'Login' with a close button. Below that is a yellow banner with the text 'Test user credentials: User mail: test@test.com Password: test'. Underneath, there are two input fields: 'Email address' containing 'test@test.com' and 'Password' containing four asterisks. A 'Submit' button is at the bottom right.

Fig. 1.5: Login modal screen.

If user enters wrong credentials, an error message will appear and warns user about wrong credentials.

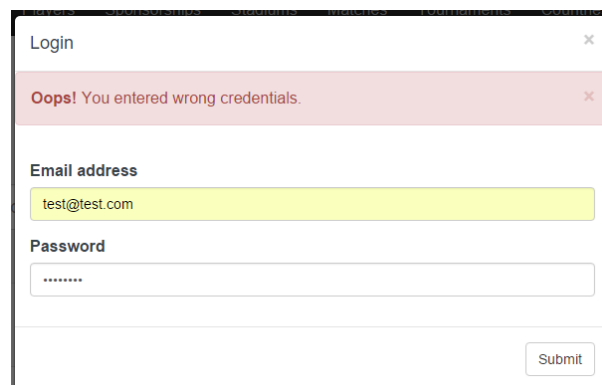
A screenshot of the 'Login' modal window showing an error. A red banner at the top says 'Oops! You entered wrong credentials.' Below it, the 'Email address' field still contains 'test@test.com' and the 'Password' field contains asterisks. The 'Submit' button remains at the bottom right.

Fig. 1.6: The message that appears when user enters wrong credentials.

Management Area

Registered users have privileges to change the records that stored in database. After user logged in, it can redirect here using drop down user menu in navigation bar. In same way, it can go back to front area using drop down menu in navigation bar. In here user greeted with change history again. But difference between the main screen change log and manager screen change log is in manager screen user can be able to see all changes has been done from beginning of the application. User can move to the management areas for different tables from sidebar.

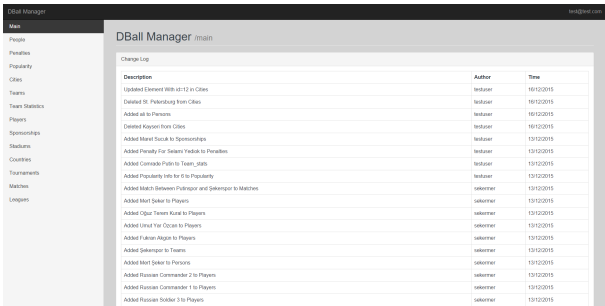


Fig. 1.7: Manager main screen.

Sidebar

From this section, user can navigate through different tables easily. Active page will be highlighted.

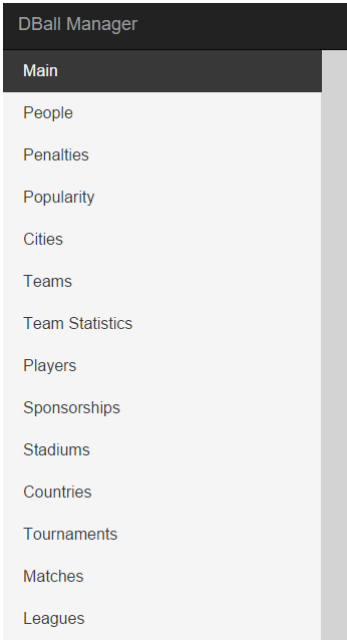


Fig. 1.8: Side navigation bar in management area.

People Records

In application each person stored in people table. From front view both unregistered and registered user can see the view front page.

User can search records that are listed in table. To search user should just type keywords into search box in right corner of the table. Also user can order tables by clicking the header of column whose elements would order the table accordingly. User can order table in ascending or descending order.

Also user can change number of elements that are shown in pages.

Person Name	Birth Date	Birth Place	Person Type
Ali	29/05/1995	Adana	Ali
Adana	10/01/1990	Adana	Coach
Sahin Yedig	01/02/2005	Adana	Normal Citizen
Levent Karaman	12/02/1990	Kayseri	Coach
Sahin Ayar	01/05/1988	Adana	Normal Citizen
Yahya Sahin	11/06/1972	Paris	Player
Buca Fikri	01/05/1982	London	Coach
Emirhan Ozbek	02/01/1987	Kayseri	Normal Citizen
Huseyin Sag	05/01/2011	Adana	Coach
Vahide Pinar	20/12/2021	Moscow	Coach

Fig. 1.9: Front view for people table.

Person Name	Birth Date	Birth Place	Person Type
Ali	29/05/1995	Adana	Ali
Adana	10/01/1990	Adana	Coach

Fig. 1.10: Searching in people table.

From top button right next to title user can advance into management area. If user not logged in it would give an error and asks user to login.

When user advances into management area, three button would appear in the bottom of the table. First of them is for adding operation, second of them is for update and the last one is for delete operation.

If operations are successful a success message will appear on top of the table, if not then an error message will appear.

Add Operation

User can add both person information and person type. Still be warned, person types cannot be deleted from database so add them wisely and only when its necessary.

From “Add New Data” button, open drop down menu. After that user can select either to add new person or person type. When clicked the selected button, a modal which would provide inputs will appear.

- **PS.** If you are not using Chromium-based browser please enter the date in ISO format (YYYY-mm-dd).

User should fill all necessary inputs. If it skips any of them a warning will appear and prevent user to send data.

Update Operation

User can update records easily first selecting which record will be updated and then clicking “Update Selected Row” button. Still, only one record can be updated at time. If user selects more record and hits the update button

Fig. 1.11: Number of elements that are going to shown in page.

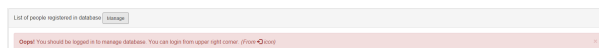


Fig. 1.12: Error that occurs when unregistered user tries to advance in manager area.

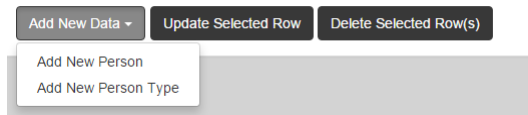


Fig. 1.13: Buttons that appear in management area.

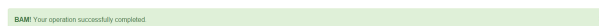


Fig. 1.14: Success message.



Fig. 1.15: Error message.

Fig. 1.16: Add person modal.

Fig. 1.17: Add person type modal.

Fig. 1.18: User warning.

an error message different from other will be appear.



Fig. 1.19: Error which appears when user select many records to update.

1	Al	20/05/1995	Alabama	Al
2	Alabama	10/01/1900	Alabama	Coach
5	Selami Yedik	01/02/2005	Alabama	Normal Citizen

Fig. 1.20: Selecting a row.

After selecting one record, user can hit update button. When user clicks the update button a modal which provides pre-filled inputs would appear. After that user can change any value as it would like.

 A modal titled "Update Person" with a close button (X). It contains four input fields: a text field with "Alibama", a date field with "01/10/1900", a dropdown menu with "Alabama", and another dropdown menu with "Coach". A "Submit" button is at the bottom right.

Fig. 1.21: Person update modal.

Delete Operation

User can delete multiple records at one time. User only needs to select which records to be deleted and hit the delete button. If operation successful the success message will appear and page will reload.

Penalty Records

In penalty records most of the table functionality are the same as people table since all tables derived from a generic table design. Hence, user can search, filter and move across table pages in same way. For those operations please refer to [People Records](#).

Add Operation

When user clicks the "Add New Data" button a drop down similar in people records will appear. From there user can add either a new penalty record or penalty type record.

- **PS.** *Beware penalty type records cannot be deleted*
- **PPS.** *If user not using Chromium-based browser, it should enter the date in ISO format (YYYY-mm-dd).*

DBall Manager /penalties

List of penalty records registered in database

Show 10 entries

Search

Penalty ID	Person Name	Given Date	Penalty Type
11	Alabama	20/04/2015	Red Card
13	Alabama	01/01/1995	Red Card
17	Selam Yedok	11/10/2015	Red Card

Showing 1 to 3 of 3 entries

Add New Data

Update Selected Data

Delete Selected Rows

Previous1Next

Fig. 1.22: Penalty records table.

Add New Person

Natalia Poklonskaya

01/20/1985

Red Card

Submit

Fig. 1.23: Penalty add modal.

Add New Penalty

Fault

Submit

Fig. 1.24: Penalty type add modal.

Update Operation

User can update one record at a time. If more rows selected, user will encounter with an error same as in people records. Again user should click “Update Selected Row” button to reveal update modal which provides necessary inputs for operation.



The image shows a modal window titled "Update Penalty" with a close button (X) in the top right corner. Inside the modal, there are three input fields: a dropdown menu with "Alabama" selected, a date field with "04/20/2015" selected, and a dropdown menu with "Yellow" selected. A "Submit" button is located at the bottom right of the modal.

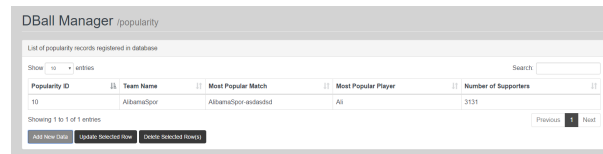
Fig. 1.25: Penalty update modal.

Delete Operation

User can delete selected rows. First it should select every rows that need to be deleted then it should hit “Delete Selected Row(s)” button. If operation successful, success message will appear and page will be reloaded.

Popularity Records

Again in same fashion, popularity records also uses generic table view for user end. User can do all operations that can be done in people record. For further information please refer to *People Records*.



The image shows the "DBall Manager (popularity)" main screen. It features a table with the following columns: Popularity ID, Team Name, Most Popular Match, Most Popular Player, and Number of Supporters. The table contains one row with the following data: 10, AlabamaSpur, AlabamaSpur-vs-Berchid, Ali, 3531. Below the table, there are buttons for "Add New Data", "Update Selected Row", and "Delete Selected Row(s)".

Popularity ID	Team Name	Most Popular Match	Most Popular Player	Number of Supporters
10	AlabamaSpur	AlabamaSpur-vs-Berchid	Ali	3531

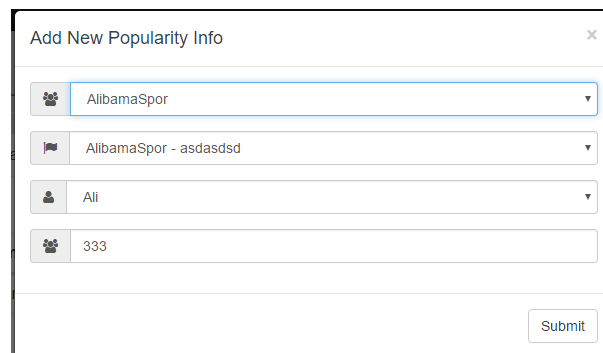
Fig. 1.26: Popularity main screen.

Add Operation

When user clicks the “Add New Data” button this time add modal directly appears and provides input for record. User should fill all necessary input or a warning will warn the user and prevent submitting info.

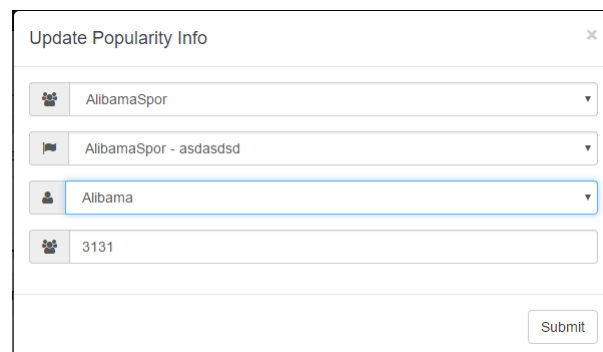
Update Operation

Again in here, user can update one record at a time. If more rows selected, user will encounter with an error same as in people records. Again user should click “Update Selected Row” button to reveal update modal which provides necessary inputs for operation.



A modal window titled "Add New Popularity Info" with a close button (X) in the top right corner. It contains four input fields, each with a small icon on the left: a group of people icon for the first field containing "AlibamaSpor", a flag icon for the second field containing "AlibamaSpor - asdasdsd", a person icon for the third field containing "Ali", and a group of people icon for the fourth field containing "333". A "Submit" button is located at the bottom right.

Fig. 1.27: Popularity add modal.



A modal window titled "Update Popularity Info" with a close button (X) in the top right corner. It contains four input fields, each with a small icon on the left: a group of people icon for the first field containing "AlibamaSpor", a flag icon for the second field containing "AlibamaSpor - asdasdsd", a person icon for the third field containing "Alibama", and a group of people icon for the fourth field containing "3131". A "Submit" button is located at the bottom right.

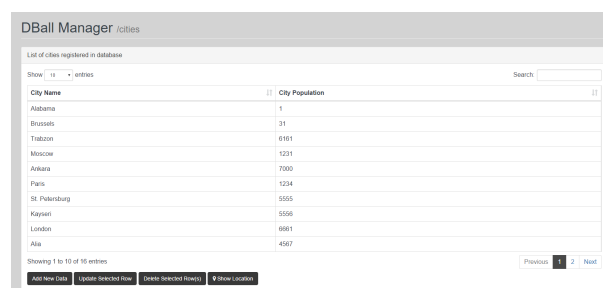
Fig. 1.28: Popularity update modal.

Delete Operation

User can delete selected rows. First it should select every rows that need to be deleted then it should hit “Delete Selected Row(s)” button. If operation successful, success message will appear and page will be reloaded.

City Records

In city records, user again can do the same operations as described in people records section. For more information about that operations please refer to [People Records](#). Additionally, user can see the location of city on map using “Show Location” button. When user hits this button after selecting a city record, a extra modal which contains a map and a marker that show location will appear. Still, user can only see one location at a time. If it selects more an error will appear.



DBall Manager Cities

List of cities registered in database

Show: 18 entries Search: []

City Name	City Population
Alibama	1
Broussels	31
Tribouren	6161
Moscow	1231
Ankara	7000
Paris	1234
St Petersburg	5565
Kayseri	5566
London	6661
Alia	4567

Showing 1 to 10 of 16 entries

Previous 1 2 Next

Buttons: Add New Data, Update Selected Item, Delete Selected Item(s), Show Location

Fig. 1.29: City main screen.

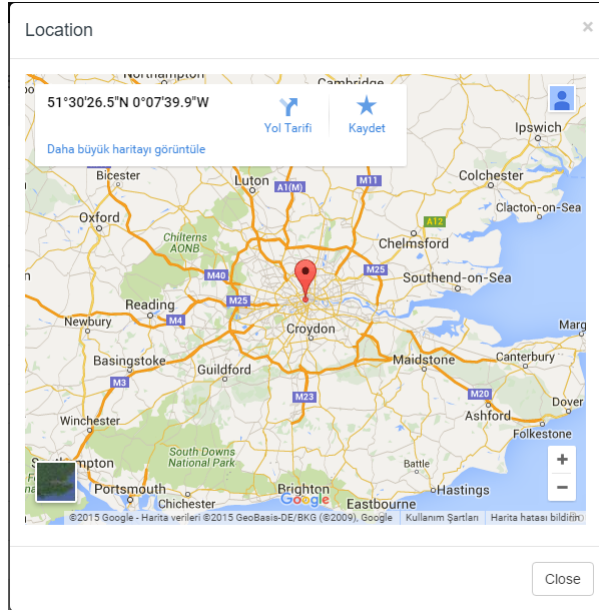


Fig. 1.30: City location modal.

Add Operation

Again as it before, when user clicks “Add New Data” button, a modal which provides necessary inputs for record will appear.

Fig. 1.31: City add modal.

Update Operation

User can update one record at a time. If more rows selected, user will encounter with an error same as in people records. Again user should click “Update Selected Row” button to reveal update modal which provides necessary inputs for operation.

Fig. 1.32: City update modal.

Delete Operation

User can delete selected rows. First it should select every rows that need to be deleted then it should hit “Delete Selected Row(s)” button. If operation successful, success message will appear and page will be reloaded.

1.1.2 Parts Implemented by Umut Can Ozyar

Sponsorships

The sponsorships data is stored in the database. Using the navigation bar located at the top of the front page sponsorships table can be accessed.

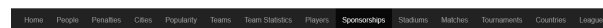


Fig. 1.33: Navigation Bar For Selecting Pages

This table displays the sponsorships data in the database.

 A screenshot of a web application showing a table titled "List of sponsors registered in database: Sponsorships". The table has five columns: Sponsor Name, Sponsorship Start Date, Sponsored League, Sponsored Team, and Sponsored Person. There are two rows of data. The first row shows "Mersin SSK" as the sponsor, starting on "05/01/2011", for the "Rize-Corona" league, sponsored to the "Dallenger" team, by "None". The second row shows "Redbull" as the sponsor, starting on "10/10/2010", for "None" league, sponsored to "None" team, by "Arkanio Tugut". At the bottom, it says "Showing 1 to 2 of 2 entries" and has "Previous" and "Next" buttons.

Fig. 1.34: Front Page For Sponsorships

Several alterations can be made by the user to change the way the data is displayed on the table. The amount of entries desired to be shown can be changed from the drop down list located at the top left of the table. The selected number corresponds to the amount of rows displayed by the table. In case the selected number exceeds the amount of sponsorships data, only the existing data will be displayed with no empty rows.

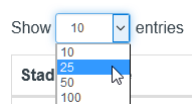


Fig. 1.35: Menu for Shown Entity Amount Selection

There are also page control buttons located at the bottom right side of the table. These buttons are used to navigate through different table pages if perchance there are more data in the database than the amount selected to be shown.

The ordering of the data throughout the table can be changed by clicking on the sort buttons located at each table header. This feature allows user to sort the data depending on various attributes of the table in descending or ascending order.

If there is no data in the database about sponsorships, “No data available in table” message is displayed on the table to notify the user.

Fig. 1.36: Buttons For Table Navigation

Sponsor Name	Sponsorship Start Date	Sponsored League	Sponsored Team	Sponsored Person
John Doe	01/01/2011	Rugby Canada	None	None
Marcel Suck	10/10/2010	La League	Waldenport	Queen Elizabeth

Fig. 1.37: Sorting Table

The manage button located on top of the table directs to user to the manager of the sponsorship table. This page is limited for registered users only. Guest users will be notified to login using the login button located at the top right side of the page.

Manager page allows user to add new data, update existing data or delete existing data.

Add Sponsorship

“Add New Data” button allows the user to add a new sponsorship for league, team and person entities in any combination. Then a modal for adding new data will appear. This modal contains several fields corresponding to different attributes of the table.

First input field is for the name of the sponsor. The second field brings out a calendar for sponsorship start date selection. Third field is for selecting the sponsored league. Fourth field is for selecting the sponsored team and the last field is for the sponsored person. Some of the last three fields can be left blank as a sponsor doesn’t have to sponsor a league, a team and a person at the same time. After the necessary fields are filled submit button is used to add the data to the table.

Some of these fields like the name and the start date cannot be left blank and will warn the user if submit button is clicked without filling these fields.

Alerts will appear on top of the table to notify the user about the outcome of the add operation. This can either be a success message with a green background which means that data is added to the database successfully or it can be a failure message with a red background which means that a problem has occurred and the operation is unsuccessful.

Update Sponsorship

“Update Selected Row” button allows the user to update a sponsorship entity on the table. If a row is not selected or multiple rows are selected, an error message notifies the user to select a single row.

If a single row is selected a modal for updating data will appear. This modal contains several fields corresponding to different attributes of the table filled with the existing data.

Sponsor Name	Sponsorship Start Date	Sponsored League	Sponsored Team	Sponsored Person
No data available in table				

Fig. 1.38: Empty Table

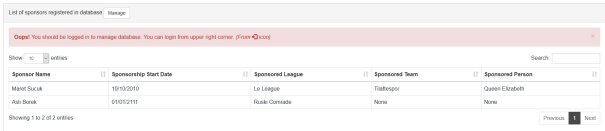


Fig. 1.39: Login Alert

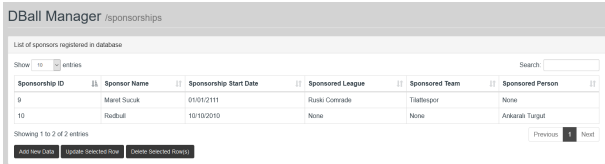


Fig. 1.40: Manager For Sponsorships

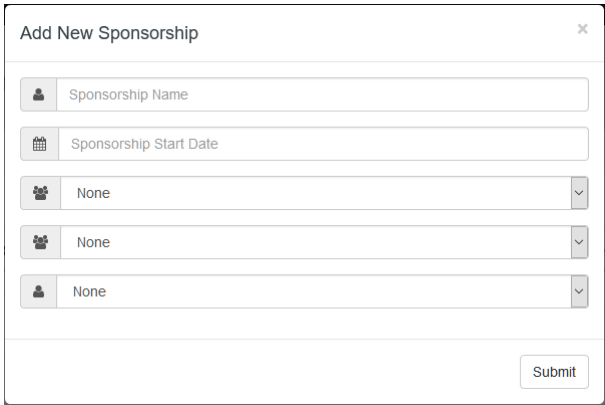


Fig. 1.41: Modal For Adding Sponsorships

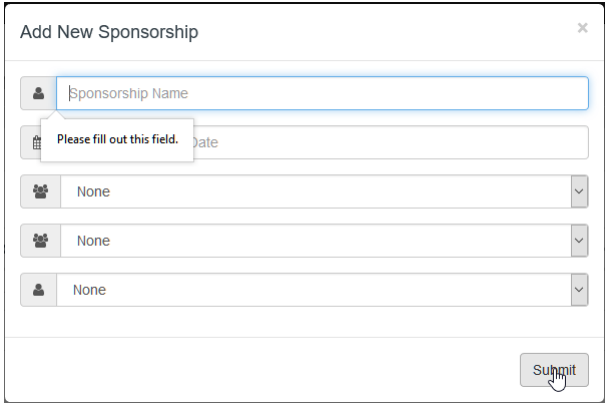


Fig. 1.42: Validation For Required Fields

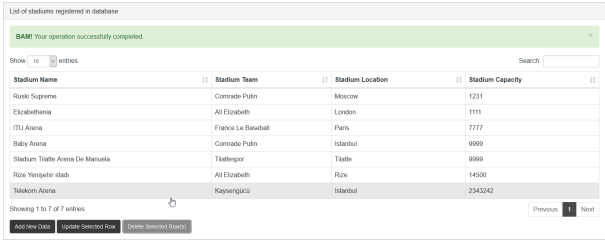


Fig. 1.43: Success Alert

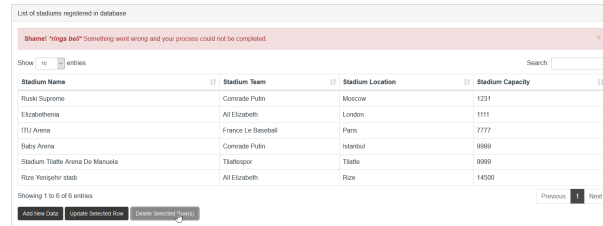


Fig. 1.44: Failure Alert

Fig. 1.45: Modal For Updating Sponsorships

Several attributes can be updated using this modal at the same time. Some fields like the name and start date will still be required to be filled. Submit button will update the data on the database.

Please refer to [Add Sponsorship](#) for more detail about the fields and all encountered alerts.

Delete Sponsorship

“Delete Selected Row(s)” button allows the user to delete sponsorship entities from the table. At least one row has to be selected to perform this operation.

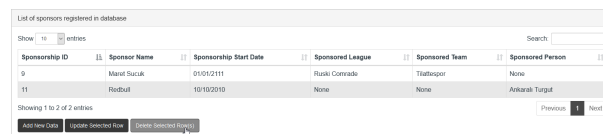


Fig. 1.46: Delete Operation For Sponsorships

Team Statistics

The team statistics data is stored in the database. Using the navigation bar located at the top of the front page team statistics table can be accessed. This table displays the sponsorships data in the database.

The manage button located on top of the table directs to user to the manager of the team statistics table. This page is limited for registered users only. Manager page allows user to add new data, update existing data or delete existing data.

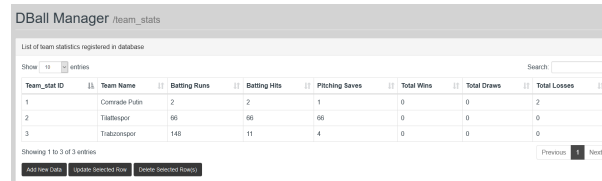


List of team statistics registered in database: Manager

Team Name	Batting Runs	Batting Hits	Pitching Saves	Total Wins	Total Draws	Total Losses
Comrade Putin	2	2	1	0	0	2
Tiathespor	66	66	66	0	0	0
Trabzonspor	148	11	4	0	0	0

Showing 1 to 3 of 3 entries

Fig. 1.47: Front Page For Team Statistics



DBall Manager /team_stats

List of team statistics registered in database

Team_stat ID	Team Name	Batting Runs	Batting Hits	Pitching Saves	Total Wins	Total Draws	Total Losses
1	Comrade Putin	2	2	1	0	0	2
2	Tiathespor	66	66	66	0	0	0
3	Trabzonspor	148	11	4	0	0	0

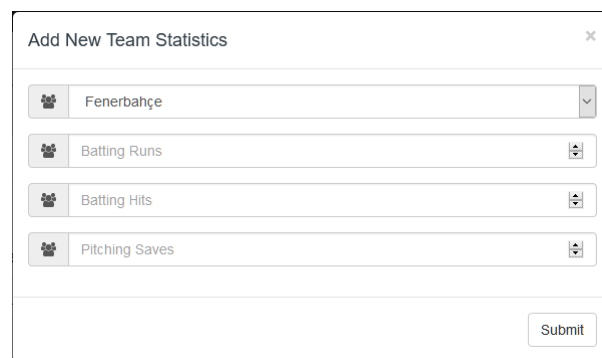
Showing 1 to 3 of 3 entries

[Add New Data](#)
[Update Selected Row](#)
[Delete Selected Row\(s\)](#)

Fig. 1.48: Manager For Team Statistics

Add Team Statistics

“Add New Data” button allows the user to add team statistics for an existing team. Then a modal for adding new data will appear. This modal contains several fields corresponding to different attributes of the table. Wins, draws and losses are automatically calculated according to the matches data.



Add New Team Statistics

List of team statistics registered in database

Team: Fenerbahçe

Batting Runs:

Batting Hits:

Pitching Saves:

Submit

Fig. 1.49: Modal For Adding Team Statistics

First input field is a drop down menu for team selection. The rest of the fields are inputs for batting runs, batting hits, pitching saves respectively. After the necessary fields are filled submit button is used to add the data to the table.

Please refer to [Add Sponsorship](#) for instructions about validation or alerts, and [Sponsorships](#) for navigation.

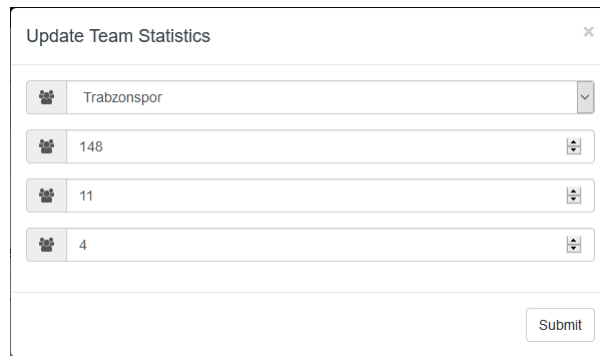
Update Team Statistics

“Update Selected Row” button allows the user to update a team statistics entity on the table. If a row is not selected or multiple rows are selected, an error message notifies the user to select a single row.

If a single row is selected a modal for updating data will appear. This modal contains several fields corresponding to different attributes of the table filled with the existing data.

Several attributes can be updated using this modal at the same time. Some fields like hits, runs and saves date will still be required to be filled. Submit button will update the data on the database.

Please refer to [Add Team Statistics](#) for more detail about the fields and [Add Sponsorship](#) for all encountered alerts.



A modal window titled "Update Team Statistics" with a close button (X) in the top right corner. It contains four input fields, each with a team icon on the left and a "v" dropdown arrow on the right. The first field is labeled "Trabzonspor". The second field contains the number "148". The third field contains the number "11". The fourth field contains the number "4". A "Submit" button is located at the bottom right of the modal.

Fig. 1.50: Modal For Updating Team Statistics

Delete Team Statistics

“Delete Selected Row(s)” button allows the user to delete team statistics entities from the table. At least one row has to be selected to perform this operation.



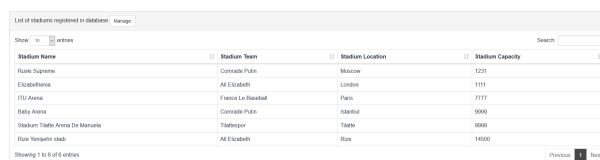
A screenshot of a web application showing a table titled "List of team statistics registered in database". The table has columns: Team_id, Team Name, Betting Runs, Betting Hits, Pitching Saves, Total Wins, Total Draws, and Total Losses. There are three rows of data. Below the table, there are buttons for "Delete Selected Row(s)", "Delete All Rows", and "Delete All Rows".

Team_id	Team Name	Betting Runs	Betting Hits	Pitching Saves	Total Wins	Total Draws	Total Losses
1	Comrade Putin	2	2	1	0	0	2
2	Trabzonspor	66	66	66	0	0	0
3	Trabzonspor	148	11	4	0	0	0

Fig. 1.51: Delete Operation For Team Statistics

Stadiums

The stadium data is stored in the database. Using the navigation bar located at the top of the front page stadiums table can be accessed. This table displays the stadiums data in the database.



A screenshot of a web application showing a table titled "List of stadiums registered in database". The table has columns: Stadium Name, Stadium Team, Stadium Location, and Stadium Capacity. There are six rows of data. Below the table, there are buttons for "Delete Selected Row(s)", "Delete All Rows", and "Delete All Rows".

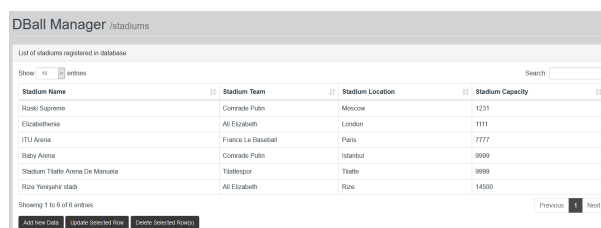
Stadium Name	Stadium Team	Stadium Location	Stadium Capacity
Rock Supreme	Comrade Putin	Moscow	1231
Elizabethtown	Al Elzabeth	London	1111
ITU Arena	France Le Roubert	Paris	7777
Baby Arena	Comrade Putin	Istanbul	9999
Stadium Trade Arena De Manville	Trabzonspor	Turkey	8888
Rice Newspaper Hall	Al Elzabeth	Rice	14555

Fig. 1.52: Front Page For Stadiums

The manage button located on top of the table directs to user to the manager of the stadium table. This page is limited for registered users only. Manager page allows user to add new data, update existing data or delete existing data.

Add Stadium

“Add New Data” button allows the user to add a new stadium for an existing team. Then a modal for adding new data will appear. This modal contains several fields corresponding to different attributes of the table.



DBall Manager /stadiums

List of stadiums registered in database

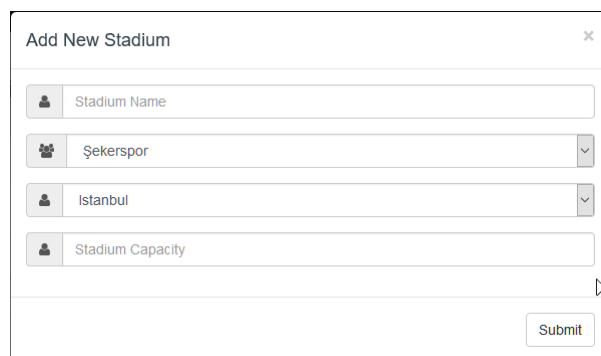
Show 12 entries

Stadium Name	Stadium Team	Stadium Location	Stadium Capacity
Ruebe Supremee	Comrade Putin	Moscow	1231
Elizabethterra	Ali Elizabeth	London	1111
ITU Arena	France La Baseball	Paris	7777
Baby Arena	Comrade Putin	Istanbul	9999
Stadium Tlatte Arena De Manuela	Tlattespor	Tlatte	9999
Rize Yorguldu stadı	Ali Elizabeth	Rize	14500

Showing 1 to 6 of 6 entries

[Add New Data](#)
[Update Selected Data](#)
[Delete Selected Data](#)

Fig. 1.53: Manager For Stadiums



Add New Stadium

Stadium Name

Şekerspor

Istanbul

Stadium Capacity

Submit

Fig. 1.54: Modal For Adding Stadiums

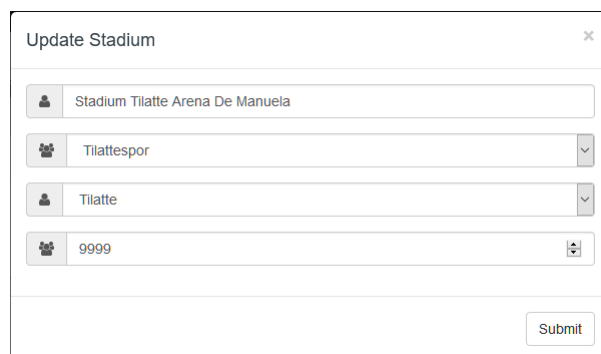
First input field is the name of the stadium. Second input field is a drop down menu for team selection. Third input field is another drop down menu for location selection which indicates the city the stadium is located in. The last field is a numerical value representing the capacity of the stadium. After the necessary fields are filled submit button is used to add the data to the table.

Please refer to [Add Sponsorship](#) for instructions about validation or alerts, and [Sponsorships](#) for navigation.

Update Stadium

“Update Selected Row” button allows the user to update a stadium entity on the table. If a row is not selected or multiple rows are selected, an error message notifies the user to select a single row.

If a single row is selected a modal for updating data will appear. This modal contains several fields corresponding to different attributes of the table filled with the existing data.



Update Stadium

Stadium Tlatte Arena De Manuela

Tlattespor

Tlatte

9999

Submit

Fig. 1.55: Modal For Updating Stadiums

Several attributes can be updated using this modal at the same time. None of the fields can be left blank. Submit button will update the data on the database.

Please refer to [Add Stadium](#) for more detail about the fields and [Add Sponsorship](#) for all encountered alerts.

Delete Stadium

“Delete Selected Row(s)” button allows the user to delete stadium entities from the table. At least one row has to be selected to perform this operation.

Stadium Name	Stadium Team	Stadium Location	Stadium Capacity
Ruzki Supreme	Comrade Putin	Moscow	1231
Ekzabthemia	All Elizabeth	London	1111
ITU Arena	France La Baseball	Paris	7777
Baby Arena	Comrade Putin	Istanbul	9999
Stadium Tlatla Arena De Manuela	Tlatlamer	Tlatla	9999
Rice Yangelis stadi	All Elizabeth	Rice	14500

Fig. 1.56: Delete Operation For Stadiums

1.1.3 Parts Implemented by Mert Şeker

Teams

All team data is kept in database. A front page to change or represent this data is used. First page is on /teams route and it represents the data in the database in a simple and understandable way and provides some functionality.

Team Name	Coach
Comrade Putin	Vladimir Putin
All Elizabeth	Queen Elizabeth
France La Baseball	Stephen Olo
KuzangGold	Hayden Yangelis
Sany Styles	Queen Elizabeth
La La La	Stephen Olo
Russian Nuke	Vladimir Putin
ISIS Paris	Stephen Olo
Padmameppor	Hayden Yangelis
Putlamer	Vladimir Putin

As it can be seen in the above figure, data is divided into 2 columns; team name, team's coach.

Second page is for both displaying and editing the data for teams and it is on the /manager/teams route and only users that have authority can access this page. In this manager page, all data is shown in data table structure. Even though the team id column is not shown on the front page, it is shown here.

On the top left side of the screen you can select how many entities are shown in a single page. You can search for a team by using the search bar on the top right side of the screen. You can sort all tuples by clicking on a column with respect to the clicked column.

The three buttons at the bottom of the page are buttons for add, update and delete operations.

Add Operation

By clicking the “Add New Data” button on the bottom of the page, a modal shows up prompting data for new record.

First box is a textbox for entering the team name. Second box is a drop down menu to choose a team coach; it is only possible to choose a person that have the person type as coach. None of these fields can be null. After

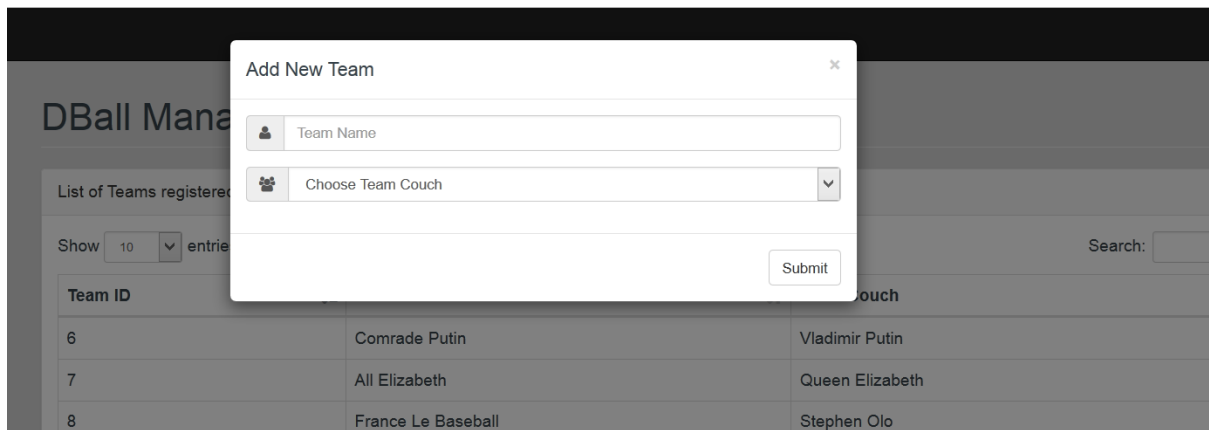


Fig. 1.57: Team Add Screen

entering the data to the fields and clicking the “Submit” button, if there are no problems in the back end, new team data will be added to the database and it can be seen in the front and manager pages.

Update Operation

Clicking on a row will select that team and clicking the “Update Selected Data” button will show up the update screen if only one row have been selected. If more than one row have been selected, an error message will be shown on the screen.

After user selects one row and clicks the update button a modal will show up for updating the team data.

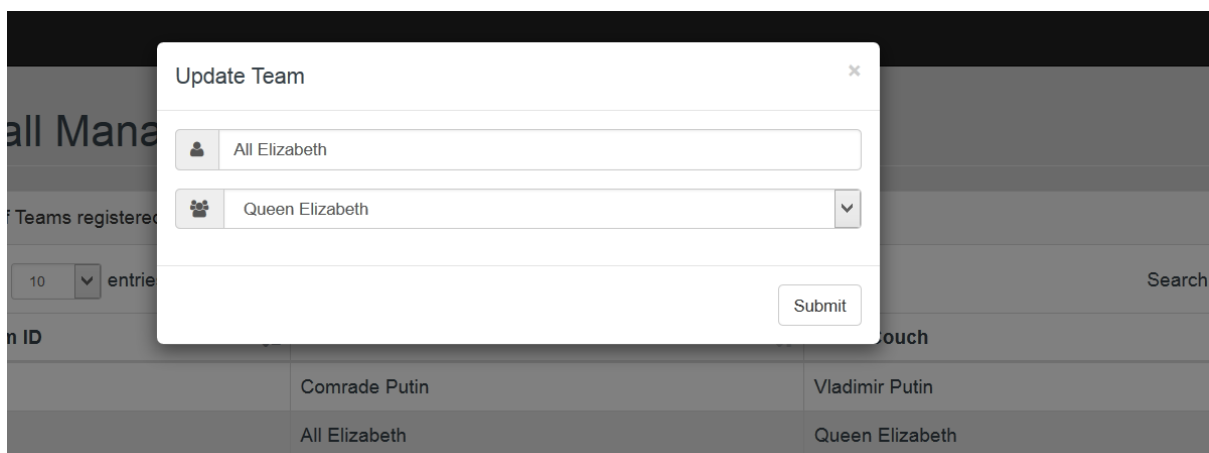


Fig. 1.58: Team Update Screen

After user enters the new data in the fields and submits the form , selected team will be updated accordingly. After the update operation is successful, all references to the previous data will also be changed by the new data.

Delete Operation

By clicking the “Delete Selected Row(s)” button user can select one or more teams. After the user have selected the rows, clicking the button will delete all the chosen rows from the team table.

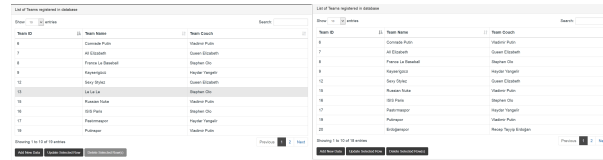
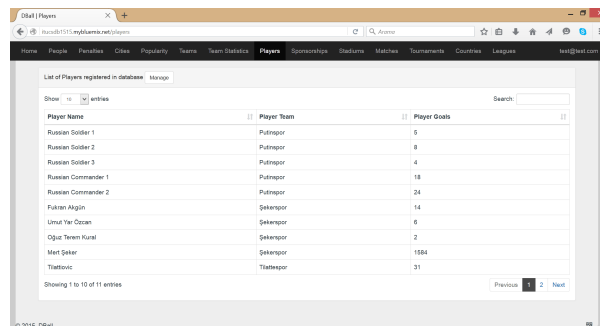


Fig. 1.59: Team Delete

Players

All player data is kept in database. A front page to change or represent this data is used. First page is on /players route and it represents the data in the database in a simple and understandable way and provides some functionality.



As it can be seen in the above figure, data is divided into 3 columns; player name, player's team and number of goals that the player have scored.

Second page is for both displaying and editing the data for players and it is on the /manager/players route and only users that have authority can access this page. In this manager page, all data is shown in data table structure. Even though the player id column is not shown on the front page, it is shown here.

On the top left side of the screen you can select how many entities are shown in a single page. You can search for a player by using the search bar on the top right side of the screen. You can sort all tuples by clicking on a column with respect to the clicked column.

The three buttons at the bottom of the page are buttons for add, update and delete operations.

Add Operation

By clicking the “Add New Data” button on the bottom of the page, a modal shows up prompting data for new record.

First box is a textbox for entering the player's name. Second box is a drop down menu to choose the player's team; it is only possible to choose a team from the teams table. Third box is for entering the number of goals that the player have scored and it is entered as integer. None of these fields can be null. After entering the data to the fields and clicking the “Submit” button, if there are no problems in the back end, new player data will be added to the database and it can be seen in the front and manager pages.

Update Operation

Clicking on a row will select that team and clicking the “Update Selected Data” button will show up the update screen if only one row have been selected. If more than one row have been selected, an error message will be shown on the screen.

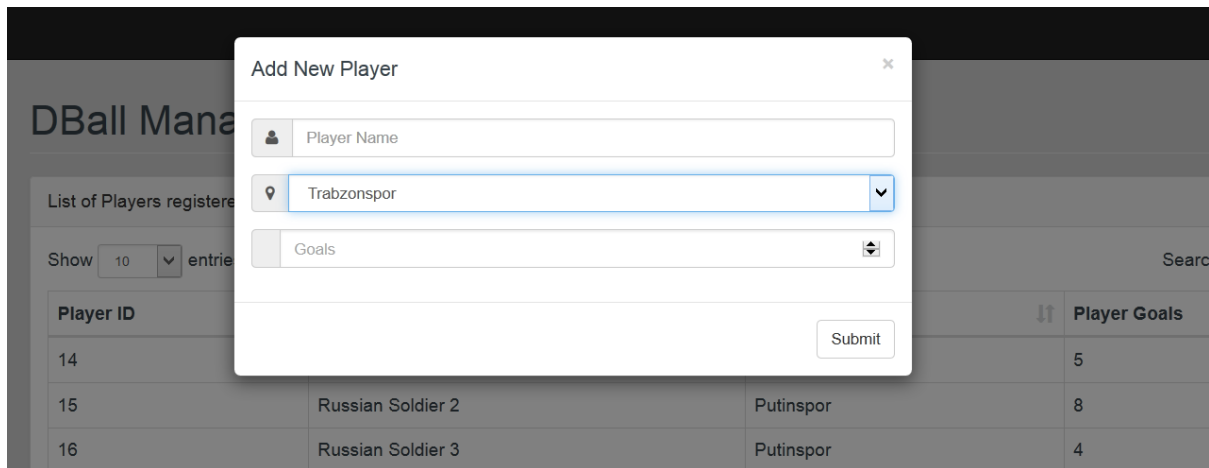


Fig. 1.60: Player Add Screen

After user selects one row and clicks the update button a modal will show up for updating the player data.

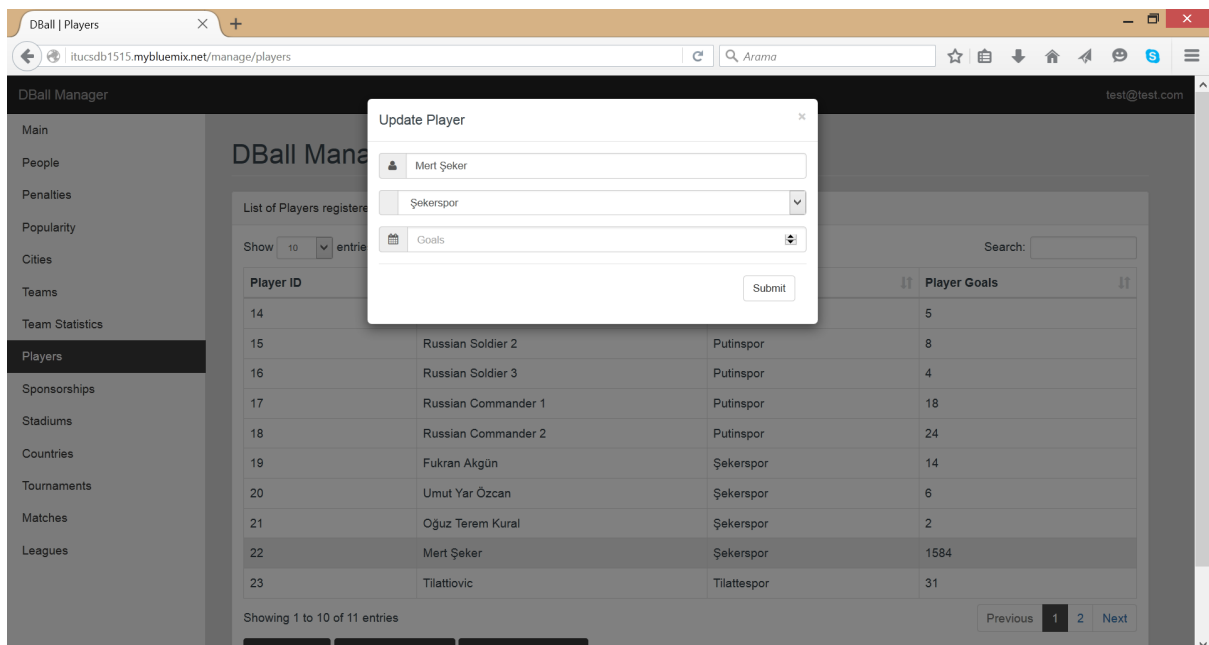


Fig. 1.61: Player Update Screen

After user enters the new data in the fields and submits the form, selected player will be updated accordingly. After the update operation is successful, all references to the previous data will also be changed by the new data.

Delete Operation

By clicking the “Delete Selected Row(s)” button user can select one or more players. After the user have selected the rows, clicking the button will delete all the chosen rows from the player table.

Fig. 1.62: Player Delete

Tournaments

All tournament data is kept in database. A front page to change or represent this data is used. First page is on /tournaments route and it represents the data in the database in a simple and understandable way and provides some functionality.

As it can be seen in the above figure, data is divided into 6 columns; tournament name, number of matches, start date, end date, country and prize.

Second page is for both displaying and editing the data for tournaments and it is on the /manager/tournaments route and only users that have authority can access this page. In this manager page, all data is shown in data table structure. Even though the tournament id column is not shown on the front page, it is shown here.

On the top left side of the screen you can select how many entities are shown in a single page. You can search for a tournament by using the search bar on the top right side of the screen. You can sort all tuples by clicking on a column with respect to the clicked column.

The three buttons at the bottom of the page are buttons for add, update and delete operations.

Add Operation

By clicking the “Add New Data” button on the bottom of the page, a modal shows up prompting data for new record.

First box is a textbox for entering the tournament’s name. Second box is for entering the number of matches. Third box is for entering the start date. Fourth box is for entering the end date. Fifth box is for choosing a country from the countries table, it is also possible to see the country’s location on the map by clicking the pin icon next to it. Sixth box is for entering the prize that will be given to the winner. None of these fields can be null. After entering the data to the fields and clicking the “Submit” button, if there are no problems in the back end, new tournament data will be added to the database and it can be seen in the front and manager pages.

Update Operation

Clicking on a row will select that team and clicking the “Update Selected Data” button will show up the update screen if only one row have been selected. If more than one row have been selected, an error message will be

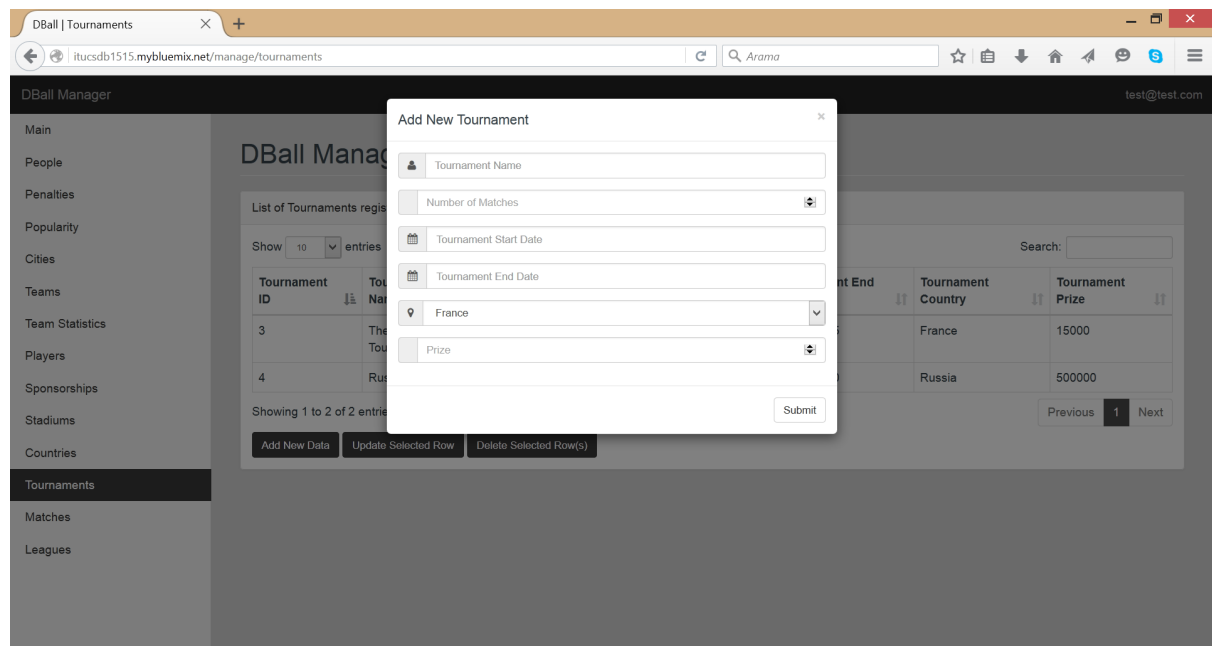


Fig. 1.63: Tournament Add Screen

shown on the screen.

After user selects one row and clicks the update button a modal will show up for updating the tournament data.

After user enters the new data in the fields and submits the form, selected tournament will be updated accordingly. After the update operation is successful, all references to the previous data will also be changed by the new data.

Delete Operation

By clicking the “Delete Selected Row(s)” button user can select one or more tournaments. After the user have selected the rows, clicking the button will delete all the chosen rows from the tournament table.

1.1.4 Parts Implemented by Furkan Akgün

Change Log

When you first enter the site, you will realize that there is a column showing the last five operations done in the site. When an authenticated user perform an operation, last five operations always be showing up in main page. If that is the first time user entered the site, by checking both columns in the home page and examining last changes user can get an idea of the website. On the other hand if it is not user’s first time, then instead of checking all tables to see what changed; user can simply look on the last changes column.

Change Log serves two main ideas; to track down which operations are done and by whom, and by some chance if database operations fails as a means of debugging. In the home page we represent only the last five changes, but in manager screen all logs are stored.

As can seen in the above figure, logs are all divided into 3 different columns; first column to explain what is done, second to tell by whom and the third for date of the operation. In the main change log it is easy to differentiate

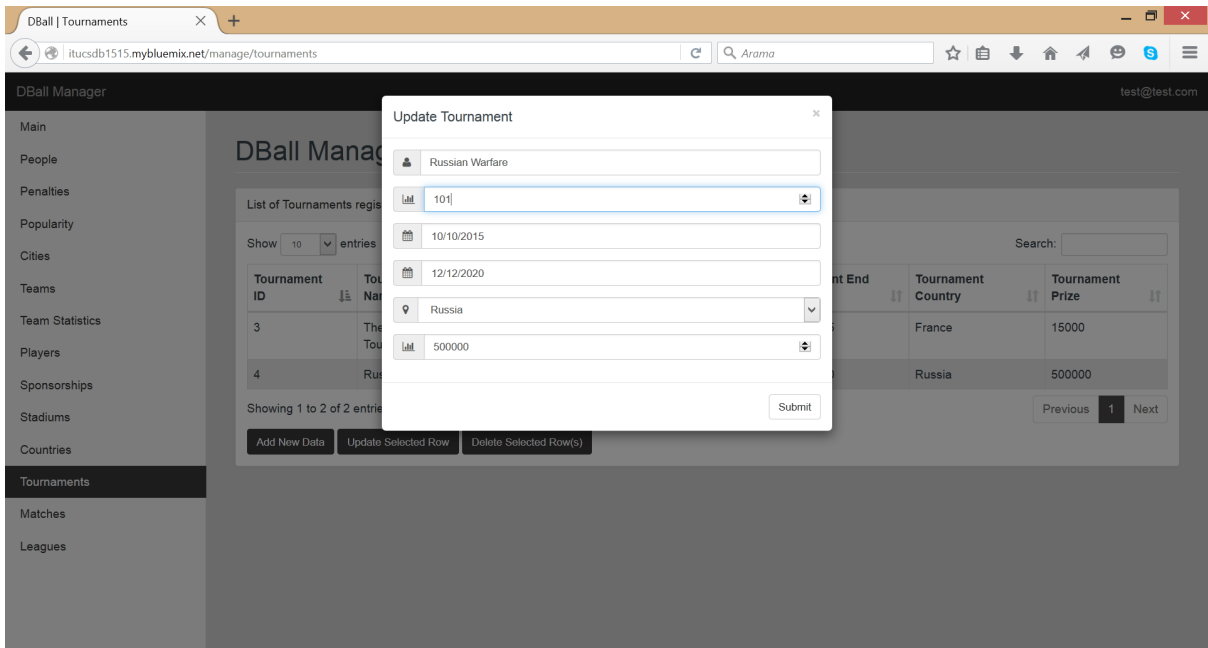


Fig. 1.64: Tournament Update Screen

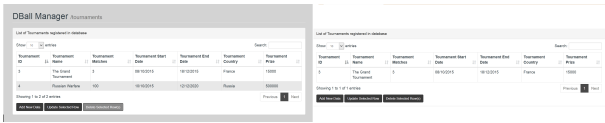


Fig. 1.65: Tournament Delete

Last Changes	
Deleted Tlatteieröglu from Countries by testuser at 20/12/2015	
Added cyka to Teams by akgunfu at 20/12/2015	
Deleted Turkey from Countries by turalog at 16/12/2015	
Deleted Small Tournament from Tournaments by turalog at 16/12/2015	
Updated Element With id=8 in Tournaments by turalog at 16/12/2015	

Fig. 1.66: Last Changes Column in Home Page

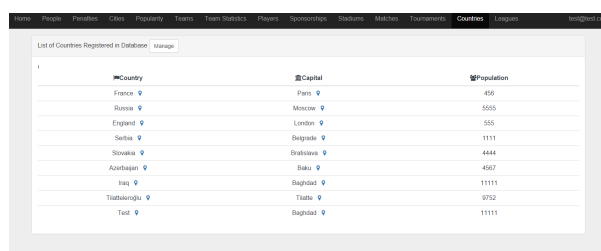
Change Log		
Description	Author	Time
Added cyka to Teams	turalog	20/12/2015
Deleted Turkey from Countries	turalog	16/12/2015
Deleted Small Tournament from Tournaments	turalog	16/12/2015
Updated Element With id=8 in Tournaments	turalog	16/12/2015
Added Medium Tournament to Tournaments	turalog	16/12/2015
Updated Element With id=20 in Players	turalog	16/12/2015
Deleted Nagphen from Players	turalog	16/12/2015
Added Nagphen to Players	turalog	16/12/2015
Added Nagphen to Players	turalog	16/12/2015
Deleted Liffegor from Teams	turalog	16/12/2015
Updated Element With id=30 in Teams	turalog	16/12/2015
Updated Element With id=30 in Teams	turalog	16/12/2015
Updated Element With id=30 in Teams	turalog	16/12/2015
Added Liffegor to Teams	turalog	16/12/2015

Fig. 1.67: All Stored Log Data

users from the description because table structure make their positions clear. But in the home page in last changes column, in some cases it may not be easy to see user in first glance. So to emphasize some keywords in log like user, we used bold font for users.

Country

All country data are stored in database. So we have basically a front page to represent or change this data. First page is simply on /country route and its purpose to represent data we have in an elegant way and providing some functionality.



Country	Capital	Population
France	Paris	456
Russia	Moscow	5555
England	London	555
Serbia	Belgrade	1111
Slovakia	Bratislava	4444
Azerbaijan	Baku	4567
Iraq	Baghdad	11111
Turkmenistan	Tashkent	9752
Turk	Baghdad	11111

Fig. 1.68: Front Page For Countries

As can be seen in the above figure, data is simply divided into 3 columns; country name, country's capital and the population. Also the table is striped, meaning that if you have your cursor over a row, that row will be focused and will be easy to see. There are location markers next to city and country names, as you can guess by clicking those icons, users can see the location of the clicked name on Google Maps API.

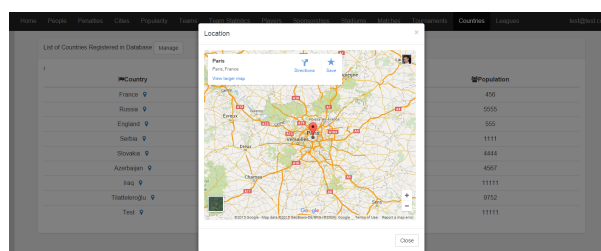


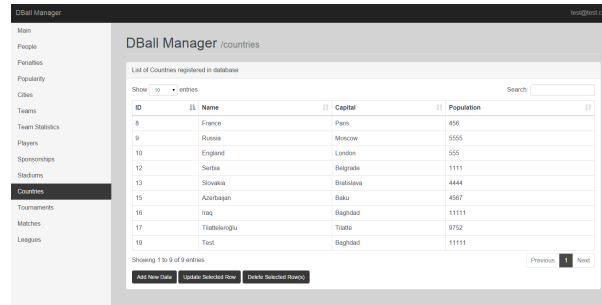
Fig. 1.69: Country Locations

In this example, I have clicked on Paris and the results can be seen as in the figure above. Right after clicking the location marker, a modal with a location map shows up by taking all the focus. Also at the top of the table, you can see "Manage" button. By clicking this button, if a user has sufficient permission, the user will be directed to the manager page for countries where he/she can change data.

Second page for both representing and changing data for countries is on the /manager/country route and only users with sufficient permissions can locate the page. In this page, all data represented in data table structure. Also any columns for country such as id are shown here while it was not showing in the front page.

On the top left side of the table, you can select how many records to show in a single page. And on the top right side of the table, you can search for any records. By clicking on the column name, you can sort all records by the clicked column.

And finally, the last three buttons in the bottom of the page are add, update, and delete buttons respectively.



ID	Name	Capital	Population
8	France	Paris	456
9	Russia	Moscow	5555
10	England	London	555
12	Serbia	Belgrade	1111
13	Slovakia	Bratislava	4444
15	Azerbaijan	Baku	4557
16	Iraq	Baghdad	11111
17	Turkmenistan	Ashgabat	9752
19	Turk	Baghdad	11111

Fig. 1.70: Country Manager Page

Add Operation

By clicking the “Add New Data” button on the bottom of the page, a modal shows up prompting data for new record.

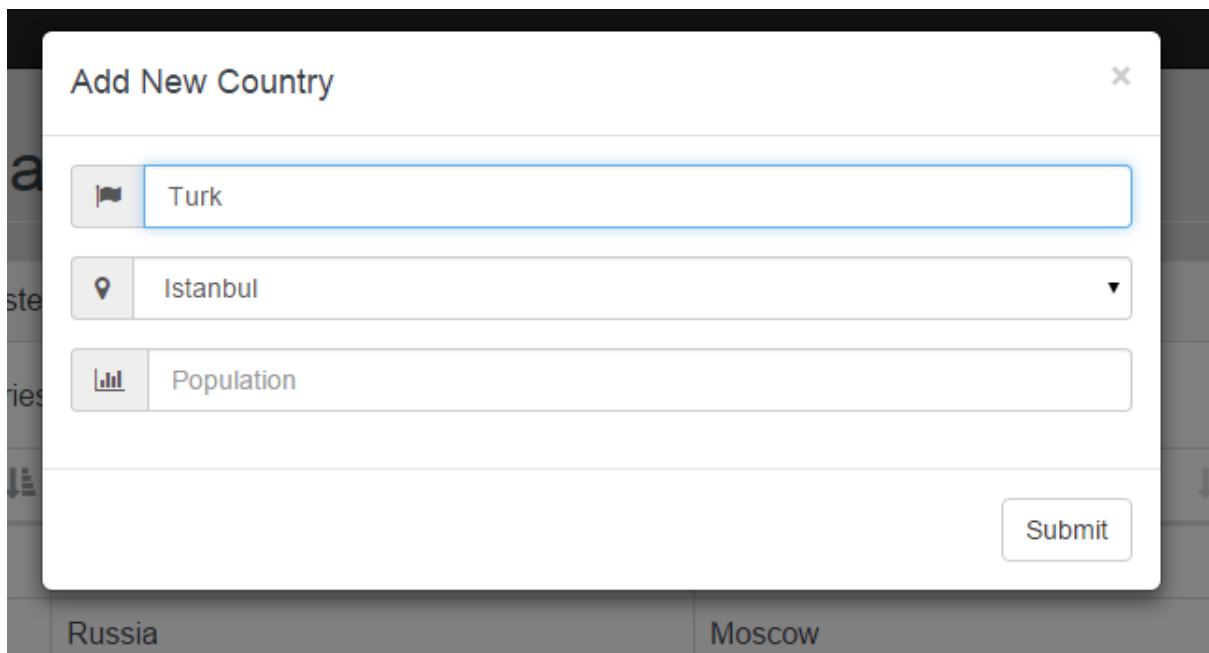


Fig. 1.71: Country Add Screen

First is country name which is simply a textbox and user can enter a country name in mind. Second is city name; users can only select cities currently on the database which are available in the selection. Third is population and users can enter an integer value. Right after completing the input and clicking the “Submit” button at the bottom of page. If there is no problem in backend new country data will be added to database and now can be seen in both front and manager pages.

Update Operation

By clicking the “Update Selected Data” button a modal will show up if the user have selected only one row. If selected row count exceeds one, then right after user clicked update button an error will show up on the top of table warning users about number of selected items.

After user selected only one row and clicked update button a modal for updating data will show up.

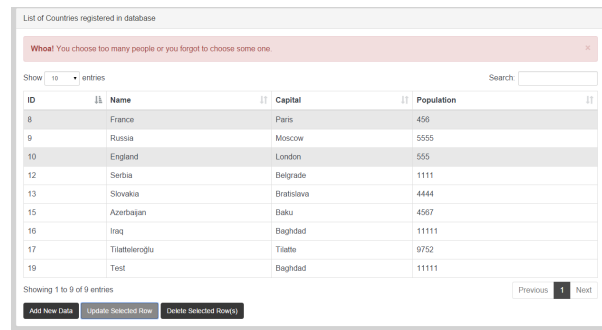


Fig. 1.72: A Warning Appears if User Tries to Update Many Rows in an Operation

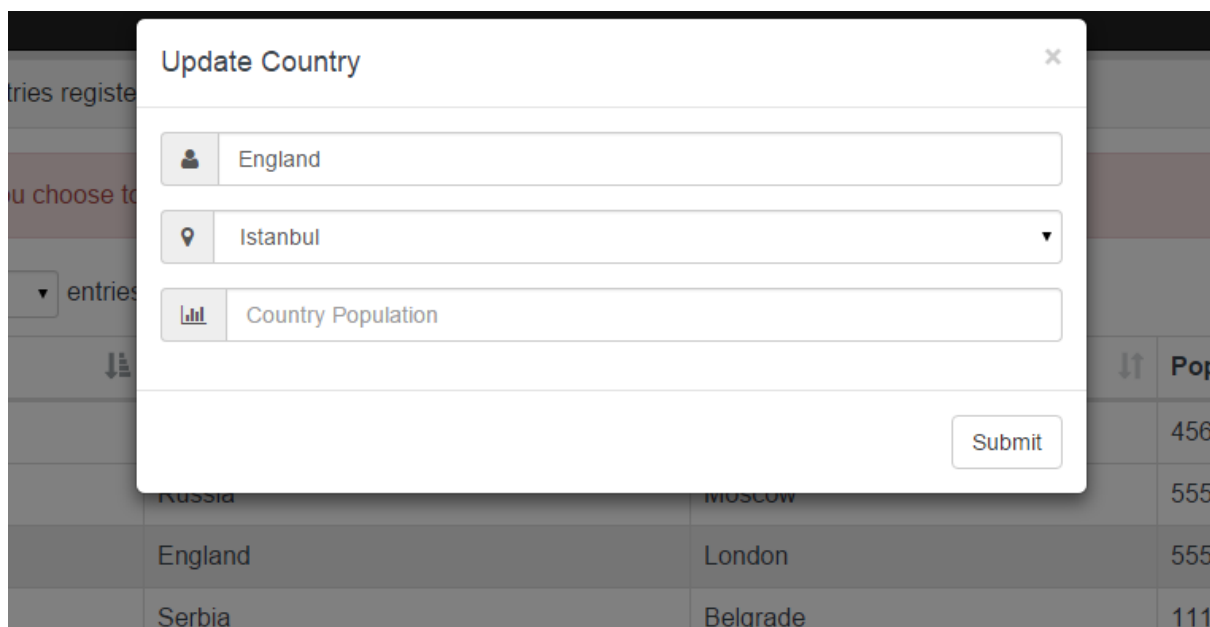


Fig. 1.73: Country Update Screen

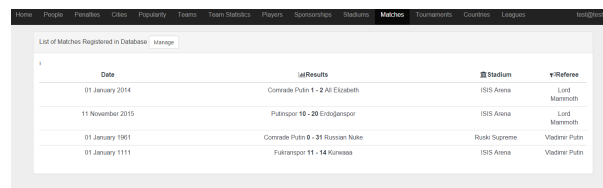
Right after user fill the inputs and submit the form ,if nothing prevents in the backend, selected row of country table will be updated. After update operation all links of previous data also be changed by the new data.

Delete Operation

By clicking the “Delete Selected Row(s)” button user can delete either one entry or multiple entries. After user selected the rows he/she wish to delete, clicking the button will delete all selected rows from the table.

Match

As like the country, match table also have two different pages on purpose. One again for to represent data in an elegant way, the other for changing the data. First page is to represent data and any user can locate this page on route /matches.



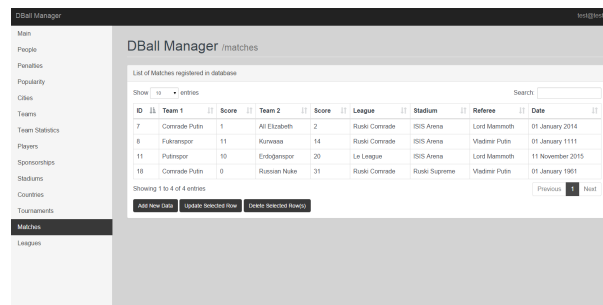
Date	Results	Stadium	Referee
01 January 2014	Comrade Putin 1 - 2 Ali Elizabeth	ISIS Arena	Lord Mammoth
11 November 2015	Putinspor 10 - 20 Endoganspor	ISIS Arena	Lord Mammoth
01 January 1951	Comrade Putin 0 - 31 Russian Nike	Ruski Supreme	Vladimir Putin
01 January 1111	Fukanspor 11 - 14 Kurwasa	ISIS Arena	Vladimir Putin

Fig. 1.74: Front Match Page

As can seen in the above figure, data is represented in a table structure and have several columns which are date, results, referee and stadium. Date, simply as the name says, shows the date when the match took place and formatted as D/M/Y. Results column shows teams and their scores with scores emphasized. And so stadium shows which stadium match took place and referee shows who was the referee in the match.

After user clicked “Manage Button” on the top of table, user will be directed to /manager/matches page if he/she have sufficient permission.

Second page is for both representation and modifying data and can be accessed only by authenticated users.

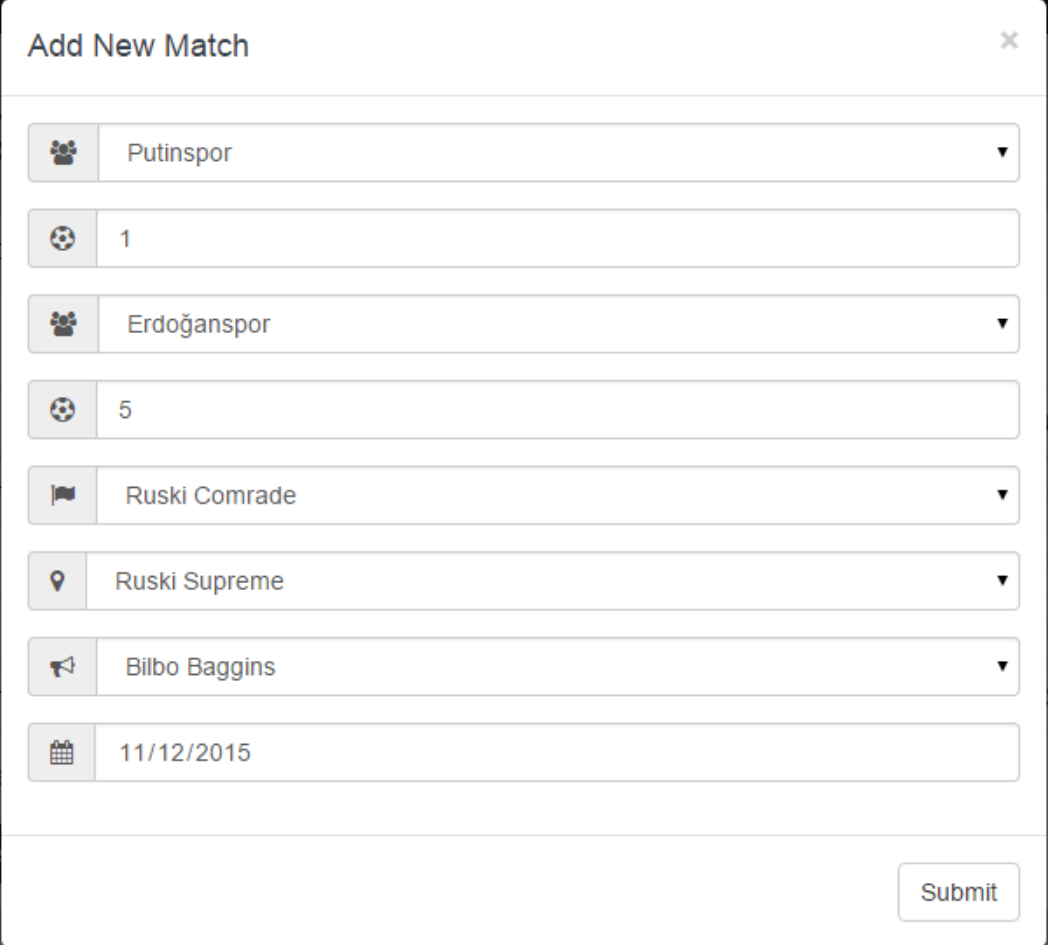


ID	Team 1	Score	Team 2	Score	League	Stadium	Referee	Date
7	Comrade Putin	1	Ali Elizabeth	2	Ruski Comrade	ISIS Arena	Lord Mammoth	01 January 2014
8	Fukanspor	11	Kurwasa	14	Ruski Comrade	ISIS Arena	Vladimir Putin	01 January 1111
11	Putinspor	10	Endoganspor	20	La League	ISIS Arena	Lord Mammoth	11 November 2015
18	Comrade Putin	0	Russian Nike	31	Ruski Comrade	Ruski Supreme	Vladimir Putin	01 January 1951

Fig. 1.75: Manager Page For Matches

Add Operation

Just like in the country page, when clicking “Add New Data” a modal shows up and asks for data for entry to be added.



The image shows a modal dialog box titled "Add New Match" with a close button (X) in the top right corner. The dialog contains several input fields, each with a small icon on the left and a dropdown arrow on the right. The fields are: a team name field with a group of people icon and the text "Putinspor"; a score field with a soccer ball icon and the text "1"; another team name field with a group of people icon and the text "Erdoğanspor"; a second score field with a soccer ball icon and the text "5"; a third team name field with a flag icon and the text "Ruski Comrade"; a location field with a location pin icon and the text "Ruski Supreme"; a fourth team name field with a megaphone icon and the text "Bilbo Baggins"; and a date field with a calendar icon and the text "11/12/2015". A "Submit" button is located at the bottom right of the dialog.

Team 1	Score 1	Team 2	Score 2	Team 3	Location	Team 4	Date
Putinspor	1	Erdoğanspor	5	Ruski Comrade	Ruski Supreme	Bilbo Baggins	11/12/2015

Fig. 1.76: Add Screen for Matches

Here you can choose two teams registered in database in dropdown menus and set score values for each of them. Score value must be between 0 and 100. Next choose a stadium from database and assign it to this match. You can also select a referee and specify date of the match in this add screen.

Update Operation

After clicking “Update Data” Button after selection row to be updated, a modal shows up asks for user to enter new data. In every page, just like in country page, user should select only one row to update. If user, by any change, try to update two or more row at the same time, a warning message will be created.

The image shows a modal window titled "Update Match" with a close button (X) in the top right corner. The modal contains the following fields:

- Team 1: Putinspor (dropdown menu)
- Score 1: 12 (text input)
- Team 2: Erdoğanspor (dropdown menu)
- Score 2: 11 (text input, highlighted with a blue border)
- League: Le League (dropdown menu)
- Stadium: Ruski Supreme (dropdown menu)
- Referee: Yahya Selamli (dropdown menu)
- Date: mm/dd/yyyy (text input)
- Submit button (bottom right)

Fig. 1.77: Update Screen for Matches

You can simply change any value of the match without damaging integrity of database.

Delete Operation

Just like in country page, you can select one or multiple entries and then hit delete button to delete them from the database.

League

All league data are stored in database. League data just like the other tables have two pages with different purposes; one for representing the data in a way appropriate to content and the other for editing data.

League	Country	Start Date	Leaderboard
Ruski Comrade	Russia	05/05/1955	Leaderboard
Le League	France	12/05/1977	Leaderboard

Fig. 1.78: League Front Page

In this page, user can see all the leagues registered in database. User can see a league's country and start date. What's more is that by clicking the "Leaderboard" button, user can access leaderboard for that league easily.

Home							Away						
Rank	Team	Played	Win	Draw	Loss	Points	Rank	Team	Played	Win	Draw	Loss	Points
1	Fakampor	1	0	0	1	0	1	Russian Rule	1	1	0	0	3
2	Comrade Putin	2	0	0	2	0	2	Kurassos	1	1	0	0	3
							3	All Elizabeth	1	1	0	0	3

Fig. 1.79: League Leaderboards

Manager page of leagues is also identical to the other class manager pages. All data are in datatable and ready to modify.

League ID	League Name	League Country	League Start Date
4	Ruski Comrade	Russia	05/05/1955
6	Le League	France	12/05/1977

Fig. 1.80: Manager Page for League


Add Operation

Just like previous classes, after clicking add button a modal for league shows up and prompts for entry. After submitting new entry will be added to the database.

Here user can name the league anything he wants and can select a registered country from the database in drop-down menu. Also user can specify start date of the league.

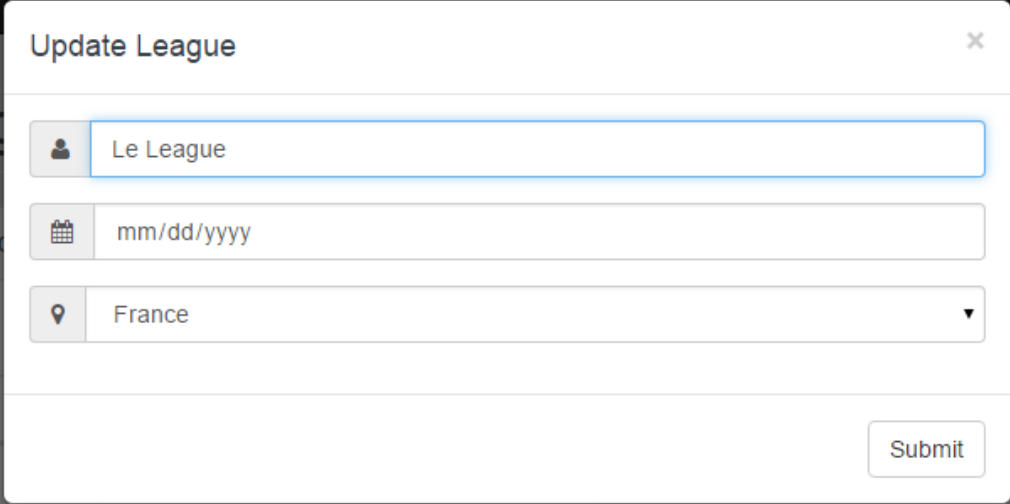
Update Operation

User first select one row to be updated by clicking on rows. However only one row at a time allowed to be updated, meaning if user ever try to update two or more selected items, a warning will appear in top of the table just like in country and match page.



The image shows a modal dialog box titled "Add New League" with a close button (X) in the top right corner. The dialog contains three input fields: a text field for "League Name" with a person icon, a date field for "mm/dd/yyyy" with a calendar icon, and a dropdown menu for "France" with a location pin icon. A "Submit" button is located at the bottom right of the dialog. In the background, a table is visible with columns for "League Name" and "Country", and a row containing "Le League" and "France".

Fig. 1.81: Add Screen for League



The image shows a modal dialog box titled "Update League" with a close button (X) in the top right corner. The dialog contains three input fields: a text field for "Le League" with a person icon, a date field for "mm/dd/yyyy" with a calendar icon, and a dropdown menu for "France" with a location pin icon. A "Submit" button is located at the bottom right of the dialog. In the background, a table is visible with columns for "League Name" and "Country", and a row containing "Le League" and "France".

Fig. 1.82: Update Screen for League

Delete Operation

User must first select the rows he/she wish to delete. After selecting the one or multiple rows to be deleted just hitting delete button will delete all selected data from the database.

1.2 Developer Guide

1.2.1 Database Design

Our database relations has been designated to be use power of relations as much as possible. All possible repeated data amount has been reduced in order to reduce used storage amount. More detailed information has been explained by each group member.

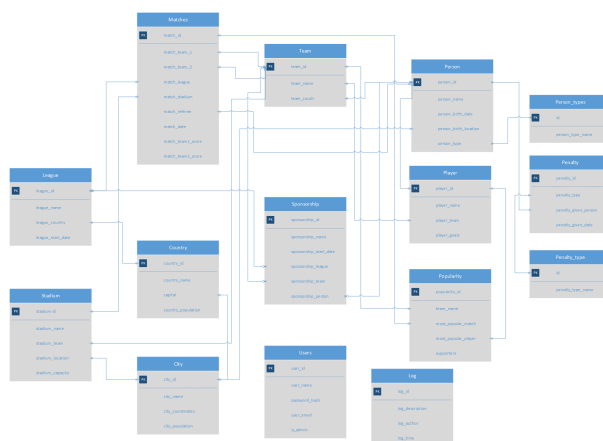
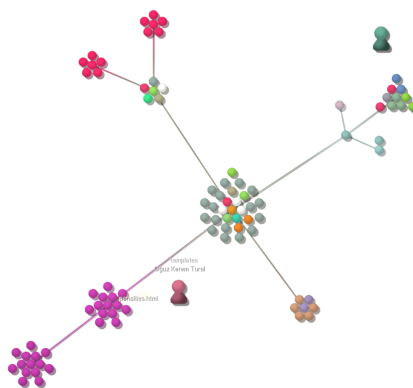


Fig. 1.83: Entity-Relation Diagram for Database

1.2.2 Git Workflow

Git workflow thorough development process has been visualized with open source software called gource.



1.2.3 Code

For code structure, model-view-controller hierarchy has been used. Where model methods and control methods has been separated. For each entity a class has been created. These classes used as models which have done the database operations. Routes has been connected to views and if user enters an input, entered data went through view to controller and then model. Also an API has been created to made possible the abstract operations which is free from user interface. In reality, models has been designed as API, thus it increases technical capabilities of our code. Each group member has been explained their parts in more detail.

Parts Implemented by Oğuz Kerem Tural

Front End Design

Application user interface uses Bootstrap framework for responsive UI, jQuery framework for much more dynamic design and DataTables framework for glorious tables. Main aim for the design was simplicity. Any type of user could easily use the application without losing its way. Thus, color scheme selection and content placement has been done accordingly. On top of the Bootstrap, a hand written CSS file has been added to extend both its responsivity and design.

Different enhancements has been applied on both *front body* and *manager body* classes. *Pagination* has been fixed, columns in front page has been hidden in *smaller screens*. Also **navigation bar** and **sidebar** has been changed in *smaller screens*.

```
.sidebar {
    display: none;
}

@media (min-width: 768px) {
    .sidebar {
        background-color: #f5f5f5;
        position: fixed;
        top: 31px;
        left: 0;
        bottom: 0;
        display: block;
        padding: 20px;
    }
}
```

For show **sidebar** minimum *screen width* has been selected as **768px**. If *screen width* is smaller than this, **sidebar** will be hidden and a **navigation bar** on top would be displayed. Both **navigation bar** and **sidebar** uses Jinja2's variable switching ability. Both front and manager layout contains a Jinja2 block that contains all menu items.

```
{% set navigation_bar = [
    ('/manage', 'main', 'Main'),
    ('/manage/people', 'people', 'People'),
    ('/manage/penalties', 'penalties', 'Penalties'),
    ('/manage/popularity', 'popularity', 'Popularity'),
    ('/manage/cities', 'cities', 'Cities'),
    ('/manage/teams', 'teams', 'Teams'),
    ('/manage/team_stats', 'team_stats', 'Team Statistics'),
    ('/manage/players', 'players', 'Players'),
    ('/manage/sponsorships', 'sponsorships', 'Sponsorships'),
    ('/manage/stadiums', 'stadiums', 'Stadiums'),
    ('/manage/countries', 'countries', 'Countries'),
    ('/manage/tournaments', 'tournaments', 'Tournaments'),
    ('/manage/matches', 'matches', 'Matches'),
    ('/manage/leagues', 'leagues', 'Leagues')] -%}
{% set active_page = active_page|default('main') -%}
```

This code block creates links, names, alternatives and also determines which page is active. Design also gives extreme importance to the dynamism. To create dynamic pages, design utilizes jQuery and JavaScript's AJAX capabilities. All submit operations handled with an AJAX handler that written for operation-specific purposes. This will be discussed in later parts.

Configuration File

Configuration file has been written in order to maintain simplicity when implementing other methods. All configuration methods has been stored in **config.py** file. It contains two methods one for parsing database parameters and another one is for creating a connection to database.

```
def db_connect():
    # Connecting db by checking VCAP credentials. By courtesy of Turgut Hoca. #
    VCAP_SERVICES = os.getenv('VCAP_SERVICES')
    if VCAP_SERVICES is not None:
        dsn = get_elephantsql_dsn(VCAP_SERVICES)
    else:
        # Change this line according to your local db credentials #
        dsn = """user='postgres' password='password'
                host='localhost' port=5432 dbname='itucsd1515'"""

    try:
        db_connection = connect(dsn)
        return db_connection
    except Error as error:
        print(error)
        return None
```

First this method checks for OS environment for environment variable called “VCAP_SERVICES”. If this variable exists then it takes and parses the connection information from deployment server. If it is not exists then it works on **localhost**, thus it takes local information to connect the database.

REST API Skeleton

All operations have done through the **REST API** that has written from scratch. The power of **REST API** is flexibility. It creates an abstract layer for all operations that needed to be done. By this way, without using any interface all operations can be completed through API. Application's user interface utilizes this ability and uses **AJAX handlers** for completing operations. API can be accessible through /api route. If user send request to the route `http://localhost/api` the answer will be in **JSON** format. All information in REST APIs are handled in JSON format. This makes it easier for AJAX handlers to understand data.

```
$ curl http://localhost:5000/api

{
    "welcome_message": "Welcome to the DBall API v1.0"
}
```

Example API usage.

Even though application has user interface, it also serves as a REST server. User interface connects API through AJAX handlers which handles the data that came from inputs. It formats the data in JSON and passes data to API. Then API methods does operation from the data which has been taken from request and sends a respond. According to this respond AJAX handler either creates an error message or shows the changes.

```
$('#modal-submit-form').submit(function() {
    var user_data = {
        // User data in dictionary form
    };

    $.ajax({
```

```

url: "/api/login",
contentType: 'application/json',
data: JSON.stringify(user_data),
type: "POST",
dataType : "json",
success: function( json ) {
    if ( json.result ) {
        // Operation Success.
    } else {
        // Operation Failure
    }
    console.log( json );
},
error: function( ) {
    console.log( "TROUBLE!" );
}
});
return false;
});

```

Skeleton for all AJAX handlers which has been used as a template on all AJAX handlers.

Get Operation API can both pull and push information to the application. To pull information, users should use specific routes that has been designed for that record. Users can either pull information for specific ID or they can pull all the records that has been stored in database. All responses will be in JSON format. **GET** routes are only allows *GET* method. Thus if it encounters with a *POST* request it would give a 405 error.

```
$ curl http://localhost:5000/api/<record_name>/<id>
```

Example request for **GET** operation.

Add Operation To complete add operation through API, user must be logged in. In other words, it should have a **session** in computer. This prevents unauthorized users to alter records. After login operation user can add using /api/<record_name>/add route to add new record to the system. It only accepts *POST* method.

```
$ curl -X POST -d "{...}" http://localhost:5000/api/<record_name>/add
```

Example request for **ADD** operation.

Update Operation Again to complete update operation user should be logged in. After logged in, user can use /api/<record_name>/update route to update records that have been stored in database. It only accepts *POST* method.

```
$ curl -X POST -d "{...}" http://localhost:5000/api/<record_name>/update
```

Example request for **UPDATE** operation.

Delete Operation After login operation user can delete records on database from the route /api/<record_name>/delete. It only accepts *POST* method.

```
$ curl -X POST -d "{...}" http://localhost:5000/api/<record_name>/delete
```

Example request for **DELETE** operation.

User Login and Register System

Another ability of API is handling user operations for application. User system something that relies on Auth API a lot. It uses sessions in order to recognize user and store its data. Login operation can be done thorough either

from user interface or through API. Further, add, delete and update operations need authorization to complete thorough API. On the other hand register operations only can be done through API.

```
class User(object):
    def __init__(self, user_alias=None, user_email=None, user_pass=None,
                  is_admin=False, user_id=None):
        self.id = user_id
        self.alias = user_alias
        self.email = user_email

        if user_pass is not None:
            self.password_hash = bcrypt.encrypt(user_pass)
        else:
            self.password_hash = user_pass

        self.user_type = is_admin

    def get_user(self, email=None):
        pass

    def add_user_to_db(self):
        pass
```

Class hierarchy in User class.

User Login User login is secure and critical process for users to alter records that have been stored in database. Since API is open, we had to require users to login before done any operation on records to prevent data persistence. When user tries to login through user interface data which user entered, gathered by AJAX and formatted into JSON notation. From here AJAX handler generates a request to the API. API gets JSON-formatted data and creates a respond again in JSON format. According to respond message AJAX handler either generates an error message or reloads the window.

```
def api_user_login():
    # Get request header #
    json_user_data = request.get_json()

    # Get user object #
    user_info = user.User()
    user_info.get_user(json_user_data['user_email'])

    # Check user credentials #
    if user_info is not None and user_info.password_hash is not None:
        if bcrypt.verify(json_user_data['user_password'],
                        user_info.password_hash) is True:
            # Create session for user #
            session['logged_in'] = True
            session['email'] = json_user_data['user_email']
            session['alias'] = user_info.alias

            status = True
        else:
            status = False
    else:
        status = False

    return jsonify({'result': status})
```

API method for user login.

API is heavily dependent on **User class** which has multiple methods for completing database operations. API method first creates an **User class** object. Then it gets data from database and compares entered password with stored salt. If they match it returns success message, otherwise error message.

```
$ curl -X POST -d '{"user_email":"test@test.com", "user_password":"ali"}' http://localhost:5000/api/register
```

Example request for user login operation through.

User Register User registration has been only implemented in API level. From user interface there is not possible to register a new user. When user creates and sends a request to API path, API generates a new **User class** object. Then it invokes `add_user_to_db()` method to store record in database. Before it stores data to database, it encrypts user password with **bcrypt** key derivation function to increase security.

```
def api_user_register():
    # Get request header #
    json_user_info = request.json

    # Convert it into user #
    user_info = user.User(
        user_alias=json_user_info['alias'],
        user_email=json_user_info['user_email'],
        user_pass=json_user_info['user_password']
    )

    # Add user to database #
    status = user_info.add_user_to_db()

    return jsonify({'result': status})
```

API method for user register.

```
"""INSERT INTO users (user_name, password_hash, user_email, is_admin)
VALUES (%s, %s, %s, %s);"""
```

SQL Query used to store user information to database.

```
$ curl -X POST -d '{"alias":"tester", "user_name":"test", "user_password":"ali"}'
http://localhost:5000/api/register
```

Example request for user register operation through.

People Records

People records are again completed in the same way. Request generated by AJAX handler, comes into API. API parses request gets data, and then it invokes `add_to_db()` method to store record in database.

As in terms of database design, it has a foreign key in `person_birth_place` column which is designated as city. Also it has another foreign key to `person_type` table. This table has only add operation and it makes possible user to add and thus select an type of person such as players, coaches, sponsors etc.

```
class Person(object):
    def __init__(self, name=None, birth_date=None, birth_place=None, user_type=None, user_id=None):
        self.id = user_id
        self.name = name
        self.birth_date = birth_date
        self.birth_place = birth_place
        self.type = user_type

    def get_person_by_id(self, get_id=None):
        pass

    def add_to_db(self):
        pass

    def delete_from_db(self):
        pass
```

```
        pass

    def update_db(self):
        pass

class PersonType(object):
    def __init__(self, type_name=None, type_id=None):
        self.id = type_id
        self.type = type_name

    def get_person_type(self, type_id=None):
        pass

    def add_to_db(self):
        pass
```

Class hierarchy for Person class.

Get Operation Because of foreign keys, when getting person information JOIN SQL operation has been used. Tables has been joined where their keys has been intersect and data derived according to resulted table.

```
# Get person type #
type_obj = people.PersonType()
type_obj.get_person_type(type_id)
# Create a dict #
data = {
    'id': type_obj.id,
    'type': type_obj.type
}

return jsonify(data)
```

API method for get operation

```
"""SELECT * FROM person
      JOIN city ON city.city_id = person.person_birth_location
      JOIN person_types ON person_types.id = person.person_type
      WHERE person_id = %s"""
```

SQL query used for get operation.

Add Operation Since person table has two foreign keys, thus before saving record into database it should have take foreign ids from city_id attribute from City table and id attribute from person type table. After it got the city_id and id it can store data to database. It uses name attribute for both foreign keys as search point because it is *unique*.

```
def api_add_person():
    # Prevent unauthorized access from API #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    # Get json request from AJAX Handler #
    json_post_data = request.get_json()
    # print(json_post_data)
    # Create an person object #
    person_info = people.Person(json_post_data['person_name'], json_post_data['person_birth_c'],
                                json_post_data['person_birth_place'], json_post_data['person_

    # Add it to db and send result #
    result = person_info.add_to_db()

    if result:
```

```

description = "Added " + json_post_data['person_name'] + " to Persons"
log_info = log.Log(description, session['alias'], datetime.datetime.now())
log_status = log_info.add_to_db()

return jsonify({'result': result})

```

API method for person add operation.

```

"""SELECT id FROM person_types WHERE person_type_name = %s"""
"""SELECT city_id FROM city WHERE city_name = %s"""
"""INSERT INTO person(person_name, person_birth_date, person_birth_location, person_type)
VALUES (%s, %s, %s, %s)"""

```

SQL Queries used to store information to database.

Update Operation Update operation is rather similar to add operation. After data passes from AJAX handler, API invokes `update_db()` method.

```

def api_update_person():
    # Get request from AJAX #
    json_data = request.get_json()
    # Get person from db #
    person_obj = people.Person()
    person_obj.get_person_by_id(json_data['person_id'])

    # Update person object's values #
    person_obj.name = json_data['person_name']
    person_obj.birth_date = json_data['person_birth_date']
    person_obj.birth_place = json_data['person_birth_place']
    person_obj.type = json_data['person_type']

    # Update db #
    result = person_obj.update_db()

    # Log operations #

    return jsonify({'result': result})

```

API method for person update operation.

```

"""SELECT city_id FROM city WHERE city_name=%s"""
"""SELECT id FROM person_types WHERE person_type_name=%s"""
"""UPDATE person
SET person_name=%s, person_birth_date=%s, person_birth_location=%s, person_type=%s
WHERE person_id=%s"""

```

SQL Queries used to update stored information on database.

Delete Operation Delete operation is relatively simple when comparing the other operations. API gets a list of ids that wanted to be deleted from request and just invokes `delete_from_db()` method for each.

```

def api_delete_person():
    # Prevent unauthorized access #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    status = False
    # Get request #
    person_id_json = request.get_json()
    # print(person_id_json)
    # Delete every requested id #
    for person_id in person_id_json:

```

```
person_obj = people.Person()
person_obj.get_person_by_id(person_id)
# print(person_id)
status = person_obj.delete_from_db()

if status:
    description = "Deleted " + person_obj.name + " from Persons"
    log_info = log.Log(description, session['alias'], datetime.datetime.now())
    log_status = log_info.add_to_db()

return jsonify({'result': status})
```

API method for person delete operation.

```
"""DELETE FROM person WHERE person_id = %s"""
```

SQL Query used to delete stored information from database.

Penalty Records

Penalty records table is relatively same as person table. It has again two foreign keys one for person and another for penalty type. Again user can add and select which types it wants but cannot delete or update it.

```
class Penalty(object):
    def __init__(self, given_person=None, given_date=None, penalty_type=None, penalty_id=None):
        self.id = penalty_id
        self.person = given_person
        self.given_date = given_date
        self.type = penalty_type

    def get_penalty_by_id(self, get_id=None):
        pass

    def add_to_db(self):
        pass

    def delete_from_db(self):
        pass

    def update_db(self):
        pass

class PenaltyType(object):
    def __init__(self, type_name=None, type_id=None):
        self.id = type_id
        self.type = type_name

    def get_penalty_type(self, type_id=None):
        pass

    def add_to_db(self):
        pass
```

Class hierarchy for Penalty class.

Get Operation Again JOIN operation has been used for getting all data in same manner as people table.

```
def api_get_penalty(data_id):
    # Create empty penalty and fill it from db #
    penalty_obj = penalties.Penalty()
    penalty_obj.get_penalty_by_id(data_id)
```



```
# Create a dict for jsonify #
data = {
    'id': penalty_obj.id,
    'person': penalty_obj.person,
    'given_date': penalty_obj.given_date.strftime('%d/%m/%Y'),
    'penalty_type': penalty_obj.type
}

return jsonify(data)
```

API method for get operation

```
"""SELECT * FROM penalty
    JOIN person ON penalty_given_person = person.person_id
    JOIN penalty_type ON penalty_type = penalty_type.id
    WHERE penalty_id = %s"""
```

SQL query used for get operation.

Add Operation Add operation also in same way as person table. But differently, this time it takes person id directly from user, thus no additional query is needed for penalty add operation.

```
def api_add_penalty():
    # Prevent unauthorized access from API #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    # Get json request from AJAX Handler #
    json_post_data = request.get_json()
    # print(json_post_data)
    # Create an penalty object #
    penalty_info = penalties.Penalty(json_post_data['person_name'], json_post_data['penalty_type'],
                                     json_post_data['penalty_type'])

    # Add it to db and send result #
    result = penalty_info.add_to_db()

    if result:
        log_person = people.Person().get_person_by_id(json_post_data['person_name'])
        description = "Added Penalty For " + log_person.name + " to Penalties"
        log_info = log.Log(description, session['alias'], datetime.datetime.now())
        log_status = log_info.add_to_db()

    return jsonify({'result': result})
```

API method for add operation.

```
"""SELECT id FROM penalty_type WHERE penalty_type_name = %s"""
"""INSERT INTO penalty(penalty_type, penalty_given_person, penalty_given_date)
VALUES (%s, %s, %s)"""
```

SQL Queries used to store information to database.

Update Operation Again it is similar to add operation when updating record.

```
def api_update_penalty():
    # Get request from AJAX #
    json_data = request.get_json()
    # Get penalty from db #
    penalty_obj = penalties.Penalty()
    penalty_obj.get_penalty_by_id(json_data['penalty_id'])
```

```
# Update penalty object's values #
penalty_obj.person = json_data['person_name']
penalty_obj.given_date = json_data['penalty_given_date']
penalty_obj.type = json_data['penalty_type']

# Update db #
result = penalty_obj.update_db()

if result:
    description = "Updated Element With id=" + json_data['penalty_id'] + " in Penalties"
    log_info = log.Log(description, session['alias'], datetime.datetime.now())
    log_status = log_info.add_to_db()

return jsonify({'result': result})
```

API method for update operation.

```
"""SELECT id FROM penalty_type WHERE penalty_type_name=%s"""
"""UPDATE penalty
    SET penalty_given_date=%s, penalty_given_person=%s, penalty_type=%s
    WHERE penalty_id=%s"""
```

SQL Queries used to update stored information on database.

Delete Operation As it was in person table, API invokes `delete_from_db()` method to delete given ids.

```
def api_delete_penalty():
    # Prevent unauthorized access #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    status = False
    # Get request #
    penalty_id_json = request.get_json()
    # Delete every requested id #
    for penalty_id in penalty_id_json:
        penalty_obj = penalties.Penalty()
        penalty_obj.get_penalty_by_id(penalty_id)
        # print(penalty_id)
        status = penalty_obj.delete_from_db()

    if status:
        description = "Deleted Penalty For " + penalty_obj.person + " from Penalties"
        log_info = log.Log(description, session['alias'], datetime.datetime.now())
        log_status = log_info.add_to_db()

    return jsonify({'result': status})
```

API method for delete operation.

```
"""DELETE FROM penalty WHERE penalty_id = %s"""
```

SQL Query used to delete stored information from database.

Popularity Records

Popularity table one of the weakest relations in the database. It has three foreign keys to other tables for team, player and match and also an integer value for supporters.

```
class Popularity(object):
    def __init__(self, team=None, match=None, player=None, supporters=None, popularity_id=None):
        self.id = popularity_id
```

```

        self.team = team
        self.match = match
        self.player = player
        self.supporters = supporters

    def get_popularity_by_id(self, get_id=None):
        pass

    def add_to_db(self):
        pass

    def delete_from_db(self):
        pass

    def update_db(self):
        pass

```

Class hierarchy for Popularity class.

Get Operation Again JOIN operation has been used for getting all data in same manner as people table. But this time it as more joins.

```

def api_get_popularity(data_id):
    # Create empty popularity and fill it from db #
    popularity_obj = popularity.Popularity()
    popularity_obj.get_popularity_by_id(data_id)

    # Create a dict for jsonify #
    data = {
        'id': popularity_obj.id,
        'team': popularity_obj.team,
        'match': popularity_obj.match,
        'player': popularity_obj.player,
        'supporters': popularity_obj.supporters
    }

    return jsonify(data)

```

API method for get operation

```

"""SELECT * FROM popularity
    JOIN team AS team1 ON popularity.team_name = team1.team_id
    JOIN matches ON popularity.most_popular_match = matches.match_id
    JOIN team AS team2 ON matches.match_team_1 = team2.team_id
    JOIN team AS team3 ON matches.match_team_2 = team3.team_id
    JOIN person ON popularity.most_popular_player = person.person_id"""

```

SQL query used for get operation.

In order to display multiple teams there has been multiple joins on teams used.

Add Operation Add operation takes foreign key values directly from the user in order to optimize queries.

```

def api_add_popularity():
    # Prevent unauthorized access from API #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    # Get json request from AJAX Handler #
    json_post_data = request.get_json()
    # print(json_post_data)
    # Create an popularity object #

```

```
popularity_info = popularity.Popularity(json_post_data['team'], json_post_data['match'],
                                       json_post_data['player'], json_post_data['supporters'])

# Add it to db and send result #
result = popularity_info.add_to_db()

if result:
    description = "Added Popularity Info for " + json_post_data['team'] + " to Popularity"
    log_info = log.Log(description, session['alias'], datetime.datetime.now())
    log_status = log_info.add_to_db()

return jsonify({'result': result})
```

API method for add operation.

```
"""INSERT INTO popularity(team_name, most_popular_match, most_popular_player, supporters)
VALUES (%s, %s, %s, %s)"""
```

SQL Queries used to store information to database.

Update Operation Again it is similar to add operation when updating record.

```
def api_update_popularity():
    # Get request from AJAX #
    json_data = request.get_json()
    # Get person from db #
    popularity_obj = popularity.Popularity()
    popularity_obj.get_popularity_by_id(json_data['popularity_id'])

    # Update person object's values #
    popularity_obj.team = json_data['team']
    popularity_obj.match = json_data['match']
    popularity_obj.player = json_data['player']
    popularity_obj.supporters = json_data['supporters']

    # Update db #
    result = popularity_obj.update_db()

    if result:
        description = "Updated Element With id=" + json_data['popularity_id'] + " in Popularity"
        log_info = log.Log(description, session['alias'], datetime.datetime.now())
        log_status = log_info.add_to_db()

    return jsonify({'result': result})
```

API method for update operation.

```
"""UPDATE popularity
SET team_name=%s, most_popular_match=%s, most_popular_player=%s, supporters=%s
WHERE popularity_id=%s"""
```

SQL Query used to update stored information on database.

Delete Operation As it was in person table, API invokes `delete_from_db()` method to delete given ids.

```
def api_delete_popularity():
    # Prevent unauthorized access #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    status = False
    # Get request #
```

```

popularity_id_json = request.get_json()
# Delete every requested id #
for popularity_id in popularity_id_json:
    popularity_obj = popularity.Popularity()
    popularity_obj.get_popularity_by_id(popularity_id)
    # print (penalty_id)
    status = popularity_obj.delete_from_db()

    if status:
        description = "Deleted Popularity Info For " + popularity_obj.team + " from Penal
        log_info = log.Log(description, session['alias'], datetime.datetime.now())
        log_status = log_info.add_to_db()

return jsonify({'result': status})

```

API method for delete operation.

```

"""DELETE FROM popularity WHERE popularity_id = %s"""

```

SQL Query used to delete stored information from database.

City Records

City table does not contain any foreign key. It uses Google Maps Geocode API in order to store location information.

```

class City(object):
    def __init__(self, city_name=None, city_population=None, city_coordinates=None, city_id=None):
        self.id = city_id
        self.name = city_name
        self.coordinates = city_coordinates
        self.population = city_population

    def get_city_by_id(self, get_id=None):
        pass

    def add_to_db(self):
        pass

    def delete_from_db(self):
        pass

    def update_db(self):
        pass

```

Class hierarchy for City class.

Get Operation Get operation is simple for city table. There is no joins since it does not have any foreign key.

```

def api_get_city(city_id):
    # Create empty city and fill it from db #
    city_obj = cities.City()
    city_obj.get_city_by_id(city_id)

    # Create a dict for jsonify #
    data = {
        'id': city_obj.id,
        'city_name': city_obj.name,
        'city_coordinates': city_obj.name,
        'city_population': city_obj.name
    }

```

```
return jsonify(data)
```

API method for get operation

```
"""SELECT * FROM city WHERE city_id = %s"""
```

SQL Queries used for get operation

Add Operation Add operation get city nme and population as input, then sends city name to Maps API and gets geolocation to store.

```
def api_add_city():
    # Prevent unauthorized access #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    # Get request #
    json_post_data = request.get_json()
    # print(json_post_data)

    city_info = cities.City(json_post_data['city_name'],
                             json_post_data['city_population'])

    # Add it to db #
    result = city_info.add_to_db()

    if result:
        description = "Added " + json_post_data['city_name'] + " to Cities"
        log_info = log.Log(description, session['alias'], datetime.datetime.now())
        log_status = log_info.add_to_db()

    return jsonify({'result': result})
```

API method for add operation

```
"""INSERT INTO city (city_name, city_population, city_coordinates)
VALUES (%s, %s, %s)"""
```

SQL Queries used for add operation

Update Operation

Again update operation also does same thing as ha been done in add operation.

```
def api_update_city():
    # Get request from AJAX #
    json_data = request.get_json()
    # Get city from db #
    city_obj = cities.City()
    city_obj.get_city_by_id(json_data['city_id'])

    # Update city object's values #
    city_obj.name = json_data['city_name']
    city_obj.population = json_data['city_population']

    # Update db #
    result = city_obj.update_db()

    if result:
        description = "Updated Element With id=" + json_data['city_id'] + " in Cities"
        log_info = log.Log(description, session['alias'], datetime.datetime.now())
        log_status = log_info.add_to_db()
```

```
return jsonify({'result': result})
```

API method for update operation

```
"""UPDATE city
    SET city_name=%s, city_population=%s, city_coordinates=%s
    WHERE city_id=%s"""
```

SQL Queries used for update operation

Delete Operation Delete operation directly deletes data from database.

```
def api_delete_city():
    # Prevent unauthorized access #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    status = False

    # Get request #
    city_id_json = request.get_json()

    for city_id in city_id_json:
        city_obj = cities.City()
        city_obj.get_city_by_id(city_id)
        status = city_obj.delete_from_db()

        if status:
            description = "Deleted " + city_obj.name + " from Cities"
            log_info = log.Log(description, session['alias'], datetime.datetime.now())
            log_status = log_info.add_to_db()

    return jsonify({'result': status})
```

API method for delete operation

```
"""DELETE FROM city WHERE city_id = %s"""
```

SQL Queries used for delete operation

Parts Implemented by Umut Can Ozyar

Sponsorships

All the sponsorships data interaction with the database happens with queries send to the server from the objects created by the sponsorship class. This table has three foreign keys, sponsorship_league, sponsorship_team and sponsorship_person, which refers to leagues, teams and people table respectively. Id is in type serial, therefore it's generated automatically, with each new entry. The rest of the fields requires new user input.

```
class Sponsorship(object):
    def __init__(self, sponsorship_name=None, sponsorship_start_date=None, sponsorship_league=None,
                 sponsorship_team=None, sponsorship_person=None, sponsorship_id=None):
        self.id = sponsorship_id
        self.name = sponsorship_name
        self.start_date = sponsorship_start_date
        self.league = sponsorship_league
        self.team = sponsorship_team
        self.person = sponsorship_person
```

Get Sponsorship This operation is the most essential one as it's used for several key actions. Either by sending an id to get a specific tuple or to get the whole table `get_sponsorship_by_id` method is called by the API. Then the SELECT queries found below, will be called with the only difference of WHERE `sponsorship_id = %s` which indicates that a unique id is specified. Three different OUTER JOIN operations are made to get the league, team and person names by joining these tables over their ids.

```
@app.route('/api/sponsorship/<int:data_id>', methods=['GET'])
```

```
def api_get_sponsorship(data_id):
    # Create empty sponsorship and fill it from db #
    sponsorship_obj = sponsorships.Sponsorship()
    sponsorship_obj.get_sponsorship_by_id(data_id)

    # Create a dict for jsonify #
    data = {
        'id': sponsorship_obj.id,
        'name': sponsorship_obj.name,
        'start_date': sponsorship_obj.start_date.strftime('%d/%m/%Y'),
        'league': sponsorship_obj.league,
        'team': sponsorship_obj.team,
        'person': sponsorship_obj.person
    }

    return jsonify(data)
```

```
def get_sponsorship_by_id(self, get_id=None):
```

```
    connection = db_connect()
    cursor = connection.cursor()
```

```
    if get_id is not None:
```

```
        statement = """SELECT sponsorship.sponsorship_id, sponsorship.sponsorship_name, sponsorship.sponsorship_league, sponsorship.sponsorship_team, sponsorship.person.person_name FROM sponsorship
        LEFT OUTER JOIN league ON league.league_id = sponsorship.sponsorship_league
        LEFT OUTER JOIN team ON team.team_id = sponsorship.sponsorship_team
        LEFT OUTER JOIN person ON person.person_id = sponsorship.sponsorship_person
        WHERE sponsorship_id = %s"""
```

```
    try:
```

```
        cursor.execute(statement, (get_id,))
        connection.commit()
```

```
    except connection.Error:
        connection.rollback()
```

```
    data = cursor.fetchone()
```

```
    if data is not None:
```

```
        self.id = data[0]
        self.name = data[1]
        self.start_date = data[2]
        self.league = data[3]
        self.team = data[4]
        self.person = data[5]
        cursor.close()
        connection.close()
        return self
```

```
    else:
```

```
        cursor.close()
        connection.close()
        return None
```

```
    else:
```

```
        statement = """SELECT sponsorship.sponsorship_id, sponsorship.sponsorship_name,
        sponsorship.sponsorship_start_date, sponsorship.sponsorship_league,
        sponsorship.sponsorship_team, sponsorship.sponsorship_person,
        league.league_id, league.league_name,
```



```

        team.team_id, team.team_name,
        person.person_id, person.person_name FROM sponsorship
        LEFT OUTER JOIN league ON league.league_id = sponsorship.sponsorship_league_id
        LEFT OUTER JOIN team ON team.team_id = sponsorship.sponsorship_team_id
        LEFT OUTER JOIN person ON person.person_id = sponsorship.sponsorship_person_id

    try:
        cursor.execute(statement, (get_id,))
        connection.commit()
    except connection.Error:
        connection.rollback()

    sponsorship_array = []
    data = cursor.fetchall()
    for sponsorship in data:
        sponsorship_array.append(
            {
                'id': sponsorship[0],
                'name': sponsorship[1],
                'start_date': sponsorship[2].strftime('%d/%m/%Y'),
                'league': sponsorship[7],
                'team': sponsorship[9],
                'person': sponsorship[11]
            }
        )

    cursor.close()
    connection.close()
    return sponsorship_array

```

Add Sponsorship After the forms on the modal for adding sponsorship are submitted, first the authorization process is made for the user by the API. If the authorization is successful, the API gets the json request from the AJAX handler. This data is then used to create a sponsorship object by calling the sponsorship constructor. Then `add_to_db` function is called on this object to perform the insertion query for sponsorship that can be found below. Note that the INSERT query is called by using foreign keys to league, team and person tables ids. Thus their ids should be fetched by using provided names.

```

@app.route('/api/sponsorship/add', methods=['POST'])
def api_add_sponsorship():
    # Prevent unauthorized access from API #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    # Get json request from AJAX Handler #
    json_post_data = request.get_json()
    # print(json_post_data)
    # Create a sponsor object #
    sponsorship_info = sponsorships.Sponsorship(json_post_data['sponsorship_name'],
                                                json_post_data['sponsorship_start_date'],
                                                json_post_data['sponsorship_league'],
                                                json_post_data['sponsorship_team'],
                                                json_post_data['sponsorship_person'])

    # Add it to db and send result #
    result = sponsorship_info.add_to_db()

    if result:
        description = "Added " + json_post_data['sponsorship_name'] + " to Sponsorships"
        log_info = log.Log(description, session['alias'], datetime.datetime.now())
        log_status = log_info.add_to_db()

    return jsonify({'result': result})

```

```
def add_to_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    new_league = None
    new_team = None
    new_person = None

    select_league = """SELECT league_id FROM league WHERE league_name = %s"""
    select_team = """SELECT team_id FROM team WHERE team_name = %s"""
    select_person = """SELECT person_id FROM person WHERE person_name = %s"""

    statement = """INSERT INTO sponsorship (sponsorship_name, sponsorship_start_date,
        sponsorship_league, sponsorship_team, sponsorship_person )
        VALUES (%s, %s, %s, %s, %s)"""

    try:
        cursor.execute(select_league, (self.league,))
        connection.commit()
        new_league = cursor.fetchone()

        cursor.execute(select_team, (self.team,))
        connection.commit()
        new_team = cursor.fetchone()

        cursor.execute(select_person, (self.person,))
        connection.commit()
        new_person = cursor.fetchone()

        cursor.execute(statement, (self.name, self.start_date, new_league, new_team, new_person))
        connection.commit()
        status = True
    except connection.Error:
        connection.rollback()
        status = False

    cursor.close()
    connection.close()
    return status
```

Delete Sponsorship Delete operation is a single DELETE query. `delete_from_db` function is called after the id of the selected rows' data is fetched and corresponding objects are found.

```
@app.route('/api/sponsorship/delete', methods=['POST'])
def api_delete_sponsorship():
    # Prevent unauthorized access #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    status = False
    # Get request #
    sponsorship_id_json = request.get_json()
    # print(sponsorship_id_json)
    # Delete every requested id #
    for sponsorship_id in sponsorship_id_json:
        sponsorship_obj = sponsorships.Sponsorship()
        sponsorship_obj.get_sponsorship_by_id(sponsorship_id)
        status = sponsorship_obj.delete_from_db()

    if status:
        description = "Deleted " + sponsorship_obj.name + " from Sponsorships"
        log_info = log.Log(description, session['alias'], datetime.datetime.now())
        log_status = log_info.add_to_db()
```

```
return jsonify({'result': status})
```

```
def delete_from_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    statement = """DELETE FROM sponsorship WHERE sponsorship_id = %s"""

    try:
        cursor.execute(statement, (self.id,))
        connection.commit()
        status = True
    except connection.Error:
        connection.rollback()
        status = False

    cursor.close()
    connection.close()
    return status
```

Update Sponsorship Update operation works similar to the add operation except the fact that there is existing data. The AJAX handler provides the data to the API which assigns them to corresponding data members. Finally the UPDATE query is executed to apply the changes to the database

```
@app.route('/api/sponsorship/update', methods=['POST'])
def api_update_sponsorship():
    # Get request from AJAX #
    json_data = request.get_json()
    # Get sponsorship from db #
    sponsorship_obj = sponsorships.Sponsorship()
    sponsorship_obj.get_sponsorship_by_id(json_data['sponsorship_id'])

    # Update sponsorship object's values #
    sponsorship_obj.name = json_data['sponsorship_name']
    sponsorship_obj.start_date = json_data['sponsorship_start_date']
    sponsorship_obj.league = json_data['sponsorship_league']
    sponsorship_obj.team = json_data['sponsorship_team']
    sponsorship_obj.person = json_data['sponsorship_person']

    # Update db #
    result = sponsorship_obj.update_db()

    if result:
        description = "Updated element with id=" + json_data['sponsorship_id'] + " in Sponsorship."
        log_info = log.Log(description, session['alias'], datetime.datetime.now())
        log_status = log_info.add_to_db()

    return jsonify({'result': result})
```

```
def update_db(self):
    connection = db_connect()
    cursor = connection.cursor()
    status = False

    new_league = None
    new_team = None
    new_person = None

    select_league = """SELECT league_id FROM league WHERE league_name = %s"""
    select_team = """SELECT team_id FROM team WHERE team_name = %s"""
    select_person = """SELECT person_id FROM person WHERE person_name = %s"""
```

```
statement = """UPDATE sponsorship
                SET sponsorship_name=%s, sponsorship_start_date=%s, sponsorship_league=%s,
                sponsorship_team=%s, sponsorship_person=%s
                WHERE sponsorship_id=%s"""

try:
    cursor.execute(select_league, (self.league,))
    connection.commit()
    new_league = cursor.fetchone()

    cursor.execute(select_team, (self.team,))
    connection.commit()
    new_team = cursor.fetchone()

    cursor.execute(select_person, (self.person,))
    connection.commit()
    new_person = cursor.fetchone()

    cursor.execute(statement, (self.name, self.start_date, new_league, new_team, new_person, ))
    connection.commit()
    status = True
except connection.Error:
    connection.rollback()
    status = False

cursor.close()
connection.close()
return status
```

Team Statistics

Team statistics class functions are mostly given as prototypes except for their queries and class data members as they are constructed in a relatively simple manner. The API functions are also omitted for the sake of simplicity since the only meaningful difference is the table names.

Team_stat table also has an id as its primary key. The rest of the data members are all in integer type and required to be provided by the user.

```
class Team_stat(object):
    def __init__(self, team_stat_name=None, team_stat_run=None, team_stat_hit=None,
                 team_stat_save=None, team_stat_win=None, team_stat_draw=None,
                 team_stat_loss=None, team_stat_id=None):
        self.id = team_stat_id
        self.name = team_stat_name
        self.run = team_stat_run
        self.hit = team_stat_hit
        self.save = team_stat_save
        self.win = team_stat_win
        self.draw = team_stat_draw
        self.loss = team_stat_loss

    def get_team_stat_by_id():

    def add_to_db():

    def delete_from_db():

    def update_db():
```

Please refer to *Sponsorships* for examples of the omitted parts of the team statistics class and API functions.

Get Team Statistics Fetching a team's statistics is done when def `get_team_stat_by_id` function is called by the API which executes the following query. It's a select query that gets the only tuple from the database for the provided id since all ids are unique.

```
statement = """SELECT team_stat.team_stat_id, team_stat.team_stat_name, team_stat.team_stat_run,
                    team_stat.team_stat_hit, team_stat.team_stat_save, team_stat.team_stat_win,
                    team_stat.team_stat_draw, team_stat.team_stat_loss
FROM team_stat
WHERE team_stat_id = %s"""
```

Add Team Statistics Adding team statistics can be cumbersome since the matches data is not structured taking total wins, draws and losses into consideration. Therefore a few SELECT queries should be executed prior to the main insertion query. Extra queries for draws and losses as well as the foreign key related queries are omitted for the sake of simplicity. The different tables are counted to get home and away wins. These values are then summed before inserted into the table. Draws and losses are calculated in the same manner. Each team's total matches are also counted with a simple SELECT count query.

```
count_win1 = """SELECT COUNT(*) FROM matches
                LEFT OUTER JOIN team ON team.team_id = matches.match_team_1
                WHERE (matches.match_team1_score > matches.match_team2_score AND team.team_name = %s)
                GROUP BY team.team_name"""

count_win2 = """SELECT COUNT(*) FROM matches
                LEFT OUTER JOIN team ON team.team_id = matches.match_team_2
                WHERE (matches.match_team1_score < matches.match_team2_score AND team.team_name = %s)
                GROUP BY team.team_name"""

total_wins = count_win1 + count_win2

statement = """INSERT INTO team_stat (team_stat_name, team_stat_run,
                    team_stat_hit, team_stat_save, team_stat_win,
                    team_stat_draw, team_stat_loss )
                VALUES (%s, %s, %s, %s, %s, %s, %s)"""

count_matches = """SELECT count(match_id) FROM matches"""
```

Delete Team Statistics Deletion is a simple operation which is executed after getting the ids of the selected rows from the table.

```
statement = """DELETE FROM team_stat WHERE team_stat_id = %s"""
```

Update Team Statistics Update also has several extra queries like the add function which calculates total wins, draws and losses. Runs, hits and saves fields can also be updated when provided with new data.

```
statement = """UPDATE team_stat
                SET team_stat_name=%s, team_stat_run=%s, team_stat_hit=%s, team_stat_save=%s,
                    team_stat_win=%s, team_stat_draw=%s, team_stat_loss=%s
                WHERE team_stat_id=%s"""
```

Stadiums

Stadiums class functions are mostly given as prototypes except for their queries and class data members as they are constructed in a relatively simple manner with the sponsorship class. The API functions are also omitted for the sake of simplicity since the only meaningful difference is the table names.

Stadium table also has an id as its primary key. The rest of the data members are all in integer type and required to be provided by the user. Location is a foreign key to the city_id column of the city table. Another foreign key is stadium_team which points to the team table.

```
class Stadium(object):
    def __init__(self, stadium_name=None, stadium_team=None, stadium_location=None,
                  stadium_capacity=None, stadium_id=None):
        self.id = stadium_id
        self.name = stadium_name
        self.team = stadium_team
        self.location = stadium_location
        self.capacity = stadium_capacity

    def get_team_stat_by_id():

    def add_to_db():

    def delete_from_db():

    def update_db():
```

Please refer to *Sponsorships* for examples of the omitted parts of the stadium class and API functions.

Get Stadium Fetching a team's stadium is done when def get_stadium_by_id function is called by the API which executes the following query. It's a select query that gets the only tuple from the database for the provided id since all ids are unique.

```
statement = """SELECT stadium.stadium_id, stadium.stadium_name, stadium.stadium_team,
                    stadium.stadium_location, stadium.stadium_capacity,
                    team.team_id, team.team_name,
                    city.city_id, city.city_name
                FROM stadium
                LEFT OUTER JOIN team ON team.team_id = stadium.stadium_team
                LEFT OUTER JOIN city ON city.city_id = stadium.stadium_location
                WHERE stadium_id = %s"""
```

Add Stadium There are only two foreign keys for stadium table which are teams and locations. Id is automatically generated for each new entry therefore the rest of the fields like name and capacity should be provided by the user. After the API gets the data from the AJAX handler add_to_db function of the stadium is called which executes the following query to add the new stadium to the database.

```
statement = """INSERT INTO stadium (stadium_name, stadium_team,
                                    stadium_location, stadium_capacity )
                VALUES (%s, %s, %s, %s)"""
```

Delete Stadium Deletion is just a single query which is executed after getting the ids of the selected rows from the table.

```
statement = """DELETE FROM stadium WHERE stadium_id = %s"""
```

Update Stadium After selecting the correct team_id for the chosen team and city_id for the chosen city name, all the inputs are passed to the UPDATE query which applies the changes to the database.

```
statement = """UPDATE stadium
                SET stadium_name=%s, stadium_team=%s, stadium_location=%s, stadium_capacity=%s
                WHERE stadium_id=%s"""
```

Parts Implemented by Mert Şeker

Database Operations for Each Entity

For each database operations of entities, appropriate SQL queries are written and they are executed within the functions in the .py class files.

Team Team tuples have three columns; id, name and coach. Coach is a foreign key to the person table.

Get Team By Id

In order to get teams and use them in functions, the primary key(team_id) is used. A dictionary is created with the chosen team's data and it is returned. You can see how this operation is done in the code below:

```
def get_team_by_id(self, get_id=None):
    connection = db_connect()
    cursor = connection.cursor()

    if get_id is not None:
        query = """SELECT t.team_id, t.team_name, t.team_couch, person.person_name
                    FROM team AS t
                    LEFT OUTER JOIN person ON person.person_id = t.team_couch
                    WHERE team_id = %s"""

        try:
            cursor.execute(query, (get_id,))
            connection.commit()

            data = cursor.fetchone()
            if data is not None:
                self.id = data[0]
                self.name = data[1]
                self.couch = data[2]
                cursor.close()
                connection.close()
                return self

            else:
                cursor.close()
                connection.close()
                return None

        except connection.Error as error:
            print(error)
            connection.rollback()

    else:
        query = """SELECT team.team_id, team.team_name, team.team_couch, person.person_id, person.
                    LEFT OUTER JOIN person ON person.person_id = team.team_couch"""

        try:
            cursor.execute(query, (get_id,))
            connection.commit()

            array = []
            data = cursor.fetchall()
            for team in data:
                array.append(
                    {
                        'id': team[0],
                        'name': team[1],
                        'couch': team[4]
```

```
        }
    )
    cursor.close()
    connection.close()

    return array

except connection.Error as error:
    print(error)
    connection.rollback()
```

Add Team To Database

In order to add team tuples to the database, INSERT INTO queries are used and executed. The foreign keys are selected from the referenced tables by id. You can see it in the code below:

```
def add_to_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    select_person = """SELECT person_id FROM person WHERE person_name = %s"""

    # query to add given team tuple to database
    query = """INSERT INTO team (team_name, team_couch)
              VALUES (%s, %s)"""

    try:
        cursor.execute(select_person, (self.couch,))
        connection.commit()
        new_person = cursor.fetchone()

        cursor.execute(query, (self.name, new_person))
        connection.commit()
        status = True

    except connection.Error as error:
        print(error)
        connection.rollback()
        status = False

    cursor.close()
    connection.close()
    return status
```

Delete Team From Database

The team to be deleted is selected by id and deleted by using DELETE FROM query. You can see it in the code below:

```
def delete_from_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    query = """DELETE FROM team WHERE team_id = %s"""

    try:
        cursor.execute(query, (self.id,))
        connection.commit()
```



```

        status = True

    except connection.Error as error:
        print(error)
        connection.rollback()
        status = False

    cursor.close()
    connection.close()
    return status

```

Update Team

The team to be updated is selected by id and updated by the UPDATE query. Just like in add operation, the foreign keys are selected from the referenced table by id. You can see it in the code below:

```

def update_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    select_person = """SELECT person_id FROM person WHERE person_name = %s"""

    query = """UPDATE team
                SET team_name=%s, team_couch=%s
                WHERE team_id=%s"""

    try:
        cursor.execute(select_person, (self.couch,))
        connection.commit()
        person_id = cursor.fetchone()

        cursor.execute(query, (self.name, person_id, self.id))
        connection.commit()
        status = True
    except connection.Error as error:
        print(error)
        connection.rollback()
        status = False

    cursor.close()
    connection.close()
    return status

```

Player Player tuples have four columns; id, name, team and number of goals. Team is a foreign key to the teams table.

Get Player By Id

In order to get players and use them in functions, the primary key(player_id) is used. A dictionary is created with the chosen player's data and it is returned. You can see how this operation is done in the code below:

```

def get_player_by_id(self, get_id=None):
    connection = db_connect()
    cursor = connection.cursor()

    if get_id is not None:
        query = """SELECT *
                    FROM player
                    JOIN team ON team.team_id = player.player_team

```

```
WHERE player_id = %s"""

    try:
        cursor.execute(query, (get_id,))
        connection.commit()
        data = cursor.fetchone()
        if data is not None:
            self.id = data[0]
            self.name = data[1]
            self.goals = data[3]
            self.team = data[5]

            cursor.close()
            connection.close()
            return self

        else:
            cursor.close()
            connection.close()
            return None

    except connection.Error as error:
        print(error)
        connection.rollback()

else:
    query = """SELECT * FROM player
              JOIN team ON team.team_id = player.player_team"""

    try:
        cursor.execute(query)
        connection.commit()
    except connection.Error as error:
        print(error)
        connection.rollback()

    array = []
    data = cursor.fetchall()

    for player in data:
        array.append(
            {
                'id': player[0],
                'name': player[1],
                'goals': player[3],
                'team': player[5]
            }
        )
    print(array)

    cursor.close()
    connection.close()

    return array
```

Add Player To Database

In order to add player tuples to the database, INSERT INTO queries are used and executed. The foreign keys are selected from the referenced tables by id. You can see it in the code below:

```
def add_to_db(self):
    connection = db_connect()
    cursor = connection.cursor()
```

```

# query to get referenced team by its id
query_team = """SELECT team_id FROM team
                WHERE team_name = %s"""

# query to add given player tuple to database
query = """INSERT INTO player (player_name, player_team, player_goals)
          VALUES (%s, %s, %s)"""

try:
    cursor.execute(query_team, (self.team,))
    connection.commit()
    team_id = cursor.fetchone()

    cursor.execute(query, (self.name, team_id, self.goals,))
    connection.commit()
    status = True

except connection.Error as error:
    print(error)
    connection.rollback()
    status = False

cursor.close()
connection.close()

return status

```

Delete Player From Database

The player to be deleted is selected by id and deleted by using DELETE FROM query. You can see it in the code below:

```

def delete_from_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    query = """DELETE FROM player WHERE player_id = %s"""

    try:
        cursor.execute(query, (self.id, ))
        connection.commit()
        status = True

    except connection.Error as error:
        print(error)
        connection.rollback()
        status = False

    cursor.close()
    connection.close()

    return status

```

Update Player

The player to be updated is selected by id and updated by the UPDATE query. Just like in add operation, the foreign keys are selected from the referenced table by id. You can see it in the code below:

```

def update_db(self):
    connection = db_connect()

```

```
cursor = connection.cursor()

query_team = """SELECT team_id FROM team WHERE team_name=%s"""
query = """UPDATE player
          SET player_name=%s, player_team=%s, player_goals=%s
          WHERE player_id=%s"""

try:
    cursor.execute(query_team, (self.team, ))
    connection.commit()
    team_id = cursor.fetchone()

    cursor.execute(query, (self.name, team_id, self.goals, self.id,))
    connection.commit()
    status = True
except connection.Error as error:
    print(error)
    connection.rollback()
    status = False
finally:
    cursor.close()
    connection.close()
    return status
```

Tournament Tournament tuples have seven columns; id,name,number of matches,start date,end date,country and prize. Country is a foreign key to the countries table.

Get Tournament By Id

In order to get tournaments and use them in functions, the primary key(tournament_id) is used. A dictionary is created with the chosen tournament's data and it is returned. You can see how this operation is done in the code below:

```
def get_tournament_by_id(self, get_id=None):
    connection = db_connect()
    cursor = connection.cursor()

    if get_id is not None:
        query = """SELECT * FROM tournament
                  JOIN country ON country.country_id = tournament.tournament_country
                  WHERE tournament_id = %s"""

        try:
            cursor.execute(query, (get_id,))
            connection.commit()
            data = cursor.fetchone()
            if data is not None:
                self.id = data[0]
                self.name = data[1]
                self.matches = data[2]
                self.start_date = data[3]
                self.end_date = data[4]
                self.country = data[8]
                self.prize = data[6]

                cursor.close()
                connection.close()
                return self

            else:
                cursor.close()
```

```

        connection.close()
        return None

    except connection.Error as error:
        print(error)
        connection.rollback()

    else:
        query = """SELECT * FROM tournament
                    JOIN country ON country.country_id = tournament.tournament_country"""

        try:
            cursor.execute(query)
            connection.commit()
        except connection.Error as error:
            print(error)
            connection.rollback()

        array = []
        data = cursor.fetchall()
        for tournament in data:
            array.append(
                {
                    'id': tournament[0],
                    'name': tournament[1],
                    'matches': tournament[2],
                    'start_date': tournament[3].strftime('%d/%m/%Y'),
                    'end_date': tournament[4].strftime('%d/%m/%Y'),
                    'country': tournament[8],
                    'prize': tournament[6]
                }
            )
        cursor.close()
        connection.close()

    return array

```

Add Tournament To Database

In order to add tournament tuples to the database, INSERT INTO queries are used and executed. The foreign keys are selected from the referenced tables by id. You can see it in the code below:

```

def add_to_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    # query to get referenced country by its id
    query_country = """SELECT country_id FROM country
                        WHERE country_name = %s"""

    # query to add given tournament tuple to database
    query = """INSERT INTO tournament (tournament_name, tournament_matches, tournament_start_date,
                                        tournament_country, tournament_prize)
              VALUES (%s, %s, %s, %s, %s, %s)"""

    try:
        cursor.execute(query_country, (self.country,))
        connection.commit()
        country_id = cursor.fetchone()

        cursor.execute(query, (self.name, self.matches, self.start_date, self.end_date, country_id, self.prize))
        connection.commit()

```

```
        status = True

    except connection.Error as error:
        print(error)
        connection.rollback()
        status = False

    cursor.close()
    connection.close()

    return status
```

Delete Tournament From Database

The tournament to be deleted is selected by id and deleted by using DELETE FROM query. You can see it in the code below:

```
def delete_from_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    query = """DELETE FROM tournament WHERE tournament_id = %s"""

    try:
        cursor.execute(query, (self.id, ))
        connection.commit()
        status = True

    except connection.Error as error:
        print(error)
        connection.rollback()
        status = False

    cursor.close()
    connection.close()

    return status
```

Update Tournament

The tournament to be updated is selected by id and updated by the UPDATE query. Just like in add operation, the foreign keys are selected from the referenced table by id. You can see it in the code below:

```
def update_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    query_country = """SELECT country_id FROM country WHERE country_name=%s"""
    query = """UPDATE tournament
                SET tournament_name=%s, tournament_matches=%s, tournament_start_date=%s, tournament_end_date=%s,
                WHERE tournament_id=%s"""

    try:
        cursor.execute(query_country, (self.country, ))
        connection.commit()
        country_id = cursor.fetchone()

        cursor.execute(query, (self.name, self.matches, self.start_date, self.end_date, country_id, self.id))
        connection.commit()
        status = True

    except connection.Error as error:
```

```

    print(error)
    connection.rollback()
    status = False
finally:
    cursor.close()
    connection.close()
return status

```

Parts Implemented by Furkan Akgün

Change Log

We created log class to be able to track user activities and also debug the site when a problem occurs. Log class simply consists of three major columns excluding id; first is the description and generated right after an operation is performed, second is the user logged in when given operation performed, this way we are able to track any user activities, and finally third one is the date of operation.

```

class Log (object):
    def __init__(self, log_description=None, log_author=None, log_time=None, log_id=None):
        self.id = log_id
        self.description = log_description
        self.author = log_author
        self.time = log_time

```

Fig. 1.84: Log Properties

Inside log class, we have three functions; retrieve a log by passing an id or retrieve all without passing an id, adding log data to database and deleting log data from the database. Now we will cover these three functions respectively.

Adding a Log to the Database To be able to show logs in home screen or manager main screen we needed to add them to the database. To add log data to the database, we simply created an object and then set its properties. After an instance of object have all properties set, we simply call `add_to_db()` function. This function basically use insert query, the variables in query are the properties of this log instance.

In log class, one of the properties was user or author and it is a foreign key to the user table. But we were setting author or user by its name, so to get user's id with that name we needed to run a query first to find user id.

If instance of log class have all properties set to appropriate values then the function will add log to the database.

Getting Log(s) To show all logs or some logs in the front view, we needed a function to return all log data or just a single one with given id. `get_log_by_id(get_id)` function simply takes an id parameter; if the id is none (or no parameter entered), the query will be executed with no specific id parameter and all logs will be returned from query and all will be stored in an array. Thne the function will just simply return that array.

On the other hand if an id value is entered as a parameter, then the query will be executed with “WHERE id=get_id” and only the log with specific id will be returned.

```
def add_to_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    # query to get referenced user by its id
    query_user = """SELECT user_id FROM users
                    WHERE user_name = %s"""

    # query to add given log tuple to database
    query = """INSERT INTO log (log_description, log_author, log_time)
              VALUES (%s, %s, %s)"""

    try:
        cursor.execute(query_user, (self.author,))
        connection.commit()
        user_id = cursor.fetchone()

        cursor.execute(query, (self.description, user_id, self.time,))
        connection.commit()
        status = True

    except connection.Error as error:
        print(error)
        connection.rollback()
        status = False

    cursor.close()
    connection.close()

    return status
```

Fig. 1.85: Function to Add Logs to Database


```
def get_log_by_id(self, get_id=None):
    connection = db_connect()
    cursor = connection.cursor()

    if get_id is not None:
        query = """SELECT * FROM log
                    JOIN users ON log.log_author = users.user_id
                    WHERE log_id = %s"""
        try:
            cursor.execute(query, (get_id,))
            connection.commit()

            data = cursor.fetchone()
            if data is not None:
                self.id = data[0]
                self.description = data[1]
                self.time = data[3]
                self.author = data[5]
                cursor.close()
                connection.close()
                return self

            else:
                cursor.close()
                connection.close()
                return None

        except connection.Error as error:
            print(error)
            connection.rollback()
```

Fig. 1.86: Function to Retrieve All Logs

```
else:
    query = """SELECT * FROM log
               JOIN users ON log.log_author = users.user_id
               ORDER BY log_id DESC"""
    try:
        cursor.execute(query)
        connection.commit()

        array = []
        data = cursor.fetchall()
        for log in data:
            array.append(
                {
                    'id': log[0],
                    'description': log[1],
                    'time': log[3].strftime('%d/%m/%Y'),
                    'author': log[5]
                }
            )

        cursor.close()
        connection.close()

        return array

    except connection.Error as error:
        print(error)
        connection.rollback()
```

Fig. 1.87: Function to Retrieve A Log

```
def delete_from_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    query = """DELETE FROM log WHERE log_id = %s"""

    try:
        cursor.execute(query, (self.id,))
        connection.commit()
        status = True

    except connection.Error as error:
        print(error)
        connection.rollback()
        status = False

    cursor.close()
    connection.close()
    return status
```

Fig. 1.88: Function to Delete a Log From the Database

Deleting a Log Deleting a log is not implemented in front view, but is ready in class as a function. Simply we get referenced instance of log and then call `delete_from_db()` function.

Creating Logs After an Operation Logs are instantly created when user performs an operation in the database. It is generic in all parts of operations, a description is created right after the operation and a log instance is created with this description, user and date. After that `add_to_db()` function on that log instance is called and log is added to the database.

```
@app.route('/api/person/add', methods=['POST'])
def api_add_person():
    # Prevent unauthorized access from API #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    # Get json request from AJAX Handler #
    json_post_data = request.get_json()
    # print(json_post_data)
    # Create an person object #
    person_info = people.Person(json_post_data['person_name'], json_post_data['person_birth_date'],
                                json_post_data['person_birth_place'], json_post_data['person_type'])

    # Add it to db and send result #
    result = person_info.add_to_db()

    if result:
        description = "Added " + json_post_data['person_name'] + " to Persons"
        log_info = log.Log(description, session['alias'], datetime.datetime.now())
        log_status = log_info.add_to_db()

    return jsonify({'result': result})
```

Fig. 1.89: Generating a Log After an Operation

As you can see right before function is completed a description is created given the operation. Simply “Added”, “Updated” or “Deleted” expressions are used for all operations. Here user is passed to the object constructor as `session['alias']`.

Displaying Last 5 Changes in Home Page After we have a function to get all logs, it was too easy to select only last five of logs sorted by date. In query of selecting all logs we did already sorted logs in descended by date column. So it is now reduced only to chose first five rows returned from `SELECT` statement.

Only five log data are stored in array, and then array is sent to the home page. In home page we can now simply display them with a for loop.

Displaying All Changes in Manager Main Page Just like displaying last five logs, but now there is no need to use a constraint. We simply retrieve all data and store them in an array. Then send the array to manager main page as data.

Then simply display each of them by a for loop.

```

@app.route('/', methods=['GET', 'POST'])
def home():
    array = []
    log_obj = log.Log()
    log_array = log_obj.get_log_by_id()
    i = 0
    for log_data in log_array:
        if i is 5:
            break

        array.append(log_data)
        i += 1

    return render_template('home.html', log_data=array)

```

Fig. 1.90: Choosing Last Five Changes

```

<div class="col-md-6">
  <div class="panel panel-default">
    <div class="panel-heading">
      <h2>Last Changes</h2>
    </div>
    <div class="panel-body">
      <table class="table borderless">
        {% if log_data %}
          {% for log in log_data %}
            <tr style="...">
              <td style="...">{{log['description']}} by <b>{{log['author']}}</b> at {{log['time']}}</td>
            </tr>
          {% endfor %}
        {% endif %}
      </table>
    </div>
  </div>
</div>

```

Fig. 1.91: Displaying Last Five Changes

```

@app.route('/manage')
def manage():
    if not session.get('logged_in'):
        flash("Unauthorized Access. Please identify yourself")
        return redirect(url_for('home'))

    log_obj = log.Log()
    log_array = log_obj.get_log_by_id()

    return render_template("manager/main.html", log_data=log_array)

```

Fig. 1.92: Choose All Logs

```

{% if log_data %}
    {% for log in log_data %}
        <tr>
            <td>{{log['description']}} </td>
            <td>{{log['author']}}</td>
            <td>{{log['time']}}</td>
        </tr>
    {% endfor %}
{% endif %}

```

Fig. 1.93: Displaying All Logs

Generic Function Bodies All classes have same function bodies. They differ with only the queries they have. So to reduce explanation for each of them, I will show generic function bodies.

First is add_to_db() Function,

All classes share these bodies, only difference is queries. Another thing is just like in the above example some class properties are set with name values but we instead use id values for them. So first we must call another queries to get their ids. Then simply execute operational query.

update_db() Function,

delete_from_db() Function,

get_(classname)_by_id() Function,

Functions up to now were only class operations. Each class have four functions above. Next functions are for add, delete, and update operations done in website. These operation are again same for other classes except some extra operations for getting referenced objects.

Add Operation,

As can be seen above, add operation creates an instance of class with json data provided by forms. After an instance is created that objects is added to the database. After a log will created for this given operation and the operation ends.

Delete Operation,

In delete operation we get all selected item ids in an array, then in a for loop we delete all selected items.

Update Operation,

Just like in the add operation we get json data from forms and instead creating a new entry, we set properties of this instance to what we get from the forms and then update the item.

Country

Country object has four properties; id, name, capital and population. Capital is a foreign key to the cities table.

```
def add_to_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    ### ANY QUERIES TO GET NECESSARY IDs FIRST

    ###

    ### INSERT INTO QUERY

    try:
        ### GET IDs FIRST
        cursor.execute(query, (parameters))
        connection.commit()
        id = cursor.fetchone()
        ###

        ### EXECUTE INSERT QUERY
        cursor.execute(query, (parameters))
        connection.commit()
        ###
        status = True

    except connection.Error as error:
        print(error)
        connection.rollback()
        status = False

    cursor.close()
    connection.close()
    return status
```

Fig. 1.94: Generic Add Function

```
def update_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    ### ANY QUERIES TO GET NECESSARY IDs FIRST

    ###

    ### UPDATE INTO QUERY

    try:
        ### GET IDs FIRST
        cursor.execute(query, (parameters))
        connection.commit()
        id = cursor.fetchone()
        ###

        ### EXECUTE UPDATE QUERY
        cursor.execute(query, (parameters))
        connection.commit()
        ###
        status = True

    except connection.Error as error:
        print(error)
        connection.rollback()
        status = False

    cursor.close()
    connection.close()
    return status
```

Fig. 1.95: Generic Update Function

```
def delete_from_db(self):
    connection = db_connect()
    cursor = connection.cursor()

    ### DELETE FROM QUERY WITH ID PARAMETER

    try:
        cursor.execute(query, (self.id,))
        connection.commit()
        status = True

    except connection.Error as error:
        print(error)
        connection.rollback()
        status = False

    cursor.close()
    connection.close()
    return status
```

Fig. 1.96: Generic Delete Function

INSERT INTO QUERY We have already provided bodies of all the functions. Those bodies were all same for all classes. What makes each class different are their unique queries for operations. These queries are executed in those functions and we complete what we try to accomplish.

In above queries, first is used to get id of the referenced capital, and then all properties of class are used as parameters to add this instance to the database.

DELETE FROM QUERY

Country with given id is deleted from the database.

SELECT QUERY

In case we pass no parameter to `get_country_by_id()` function, the query with no “WHERE” clause will be used. Above query is used when we pass an id parameter.

UPDATE QUERY

Just like in the add operation queries excluding update query gets referenced item ids and then use them as parameter in the update query.

Matches

Match object has nine properties; id, home team, score of home team, away team, score of away team, stadium, referee, league and match date. Team, stadium, referee and league are all foreign keys.


```

def get_country_by_id(self, get_id=None):
    connection = db_connect()
    cursor = connection.cursor()
    if get_id is not None:
        ### SELECT QUERY WITH ID
        try:
            cursor.execute(query, (get_id,))
            connection.commit()
            data = cursor.fetchone()
            if data is not None:
                self.id = data[0]
                ### SET ALL OTHER ATTRIBUTES
                cursor.close()
                connection.close()
                return self
            else:
                cursor.close()
                connection.close()
                return None
        except connection.Error as error:
            print(error)
            connection.rollback()
    else:
        ### SELECT QUERY
        try:
            cursor.execute(query)
            connection.commit()
            array = []
            data = cursor.fetchall()
            for x in data:
                array.append(
                    {
                        'id': x[0],
                        ### OTHER ATTRIBUTES
                    }
                )
            cursor.close()
            connection.close()
            return array
        except connection.Error as error:
            print(error)
            connection.rollback()

```

Fig. 1.97: Generic Retrieve

```
@app.route('/api/country/add', methods=['POST'])
def api_add_country():
    # Prevent unauthorized access from API #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    # Get json request from AJAX Handler #
    json_post_data = request.get_json()
    country_info = country.Country(json_post_data['country_name'],
                                   json_post_data['country_population'], json_post_data['capital'])
    # Add it to db and send result #
    result = country_info.add_to_db()

    if result:
        description = "Added " + json_post_data['country_name'] + " to Countries"
        log_info = log.Log(description, session['alias'], datetime.datetime.now())
        log_status = log_info.add_to_db()

    return jsonify({'result': result})
```

Fig. 1.98: Generic Add Operation

```
@app.route('/api/country/delete', methods=['POST'])
def api_delete_country():
    # Prevent unauthorized access #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    # Get request #
    country_json = request.get_json()
    # Deletes every object in the request from database
    status = False
    for country_id in country_json:
        country_obj = country.Country()
        country_obj.get_country_by_id(country_id)
        status = country_obj.delete_from_db()

        if status:
            description = "Deleted " + country_obj.name + " from Countries"
            log_info = log.Log(description, session['alias'], datetime.datetime.now())
            log_status = log_info.add_to_db()

    return jsonify({'result': status})
```

Fig. 1.99: Generic Delete Operation

```

@app.route('/api/country/update', methods=['POST'])
def api_update_country():
    # Prevent unauthorized access #
    if not session.get('logged_in'):
        return jsonify({"result": "Unauthorized Access. Please identify yourself"})

    # Get request #
    json_data = request.get_json()

    country_obj = country.Country()
    country_obj.get_country_by_id(json_data['country_id'])

    # Update country values #
    country_obj.name = json_data['country_name']
    country_obj.population = json_data['country_population']
    country_obj.capital = json_data['capital']

    # Update #
    result = country_obj.update_db()

    if result:
        description = "Updated Element With id=" + json_data['country_id'] + " in Countries"
        log_info = log.Log(description, session['alias'], datetime.datetime.now())
        log_status = log_info.add_to_db()

    return jsonify({'result': result})

```

Fig. 1.100: Generic Update Operation

```

query_capital = """SELECT city_id FROM city
                    WHERE city_name = %s"""

query = """INSERT INTO country (country_name, country_population, capital)
          VALUES (%s, %s, %s)"""

```

Fig. 1.101: Country Insert Into Query

```

query = """DELETE FROM country WHERE country_id = %s"""

```

Fig. 1.102: Country Delete Query

```

query = """SELECT * FROM country
          JOIN city ON country.capital=city.city_id
          WHERE country_id = %s"""

```

Fig. 1.103: Country Select Query

```

query_capital = """ SELECT city_id FROM city
                    WHERE city_name = %s"""

query = """UPDATE country
          SET country_name=%s, country_population=%s, capital=%s
          WHERE country_id=%s"""

```

Fig. 1.104: Country Update Query

INSERT INTO QUERY We have already provided bodies of all the functions. Those bodies were all same for all classes. What makes each class different are their unique queries for operations. These queries are executed in those functions and we complete what we try to accomplish.

```
query_team = """SELECT team_id FROM team
                WHERE team_name = %s"""

query_league = """SELECT league_id FROM league
                WHERE league_name = %s"""

query_stadium = """SELECT stadium_id FROM stadium
                WHERE stadium_name = %s"""

query_referee = """SELECT person_id FROM person
                WHERE person_name = %s"""

query = """INSERT INTO matches (match_team_1, match_team_2, match_league,
                                match_stadium, match_referee, match_date,
                                match_team1_score, match_team2_score)
                VALUES (%s, %s, %s, %s, %s, %s, %s, %s)"""
```

Fig. 1.105: Match Insert Into Query

In above queries, queries except the last one are used to get ids of the referenced items, and then all properties of class are used as parameters to add this instance to the database.

DELETE FROM QUERY

```
query = """DELETE FROM matches WHERE match_id = %s"""
```

Fig. 1.106: Match Delete Query

Match with given id is deleted from the database.

SELECT QUERY

```
query = """SELECT * FROM matches
            JOIN team ON matches.match_team_1 = team.team_id
            JOIN team AS t ON matches.match_team_2 = t.team_id
            JOIN league ON matches.match_league = league.league_id
            JOIN stadium ON matches.match_stadium = stadium.stadium_id
            JOIN person ON matches.match_referee = person.person_id
            WHERE match_id = %s"""
```

Fig. 1.107: Match Select Query

In case we pass no parameter to `get_match_by_id()` function, the query with no “WHERE” clause will be used. Above query is used when we pass an id parameter.

UPDATE QUERY

```

query_team = """SELECT team_id FROM team
                WHERE team_name = %s"""

query_league = """SELECT league_id FROM league
                WHERE league_name = %s"""

query_stadium = """SELECT stadium_id FROM stadium
                WHERE stadium_name = %s"""

query_referee = """SELECT person_id FROM person
                WHERE person_name = %s"""

query = """UPDATE matches
            SET match_team_1=%s, match_team_2=%s, match_league=%s,
              match_stadium=%s, match_referee=%s, match_date=%s,
              match_team1_score=%s, match_team2_score=%s
            WHERE match_id=%s"""

```

Fig. 1.108: Match Update Query

Just like in the add operation queries excluding update query gets referenced item ids and then use them as parameter in the update query.

League

League object has four properties; id, name, country and start date. Country is a foreign key to the country table.

INSERT INTO QUERY We have already provided bodies of all the functions. Those bodies were all same for all classes. What makes each class different are their unique queries for operations. These queries are executed in those functions and we complete what we try to accomplish.

```

query_country = """SELECT country_id FROM country
                WHERE country_name = %s"""

query = """INSERT INTO league (league_name, league_country, league_start_date)
            VALUES (%s, %s, %s)"""

```

Fig. 1.109: League Insert Into Query

In above queries, first is used to get id of the referenced country, and then all properties of class are used as parameters to add this instance to the database.

DELETE FROM QUERY

League with given id is deleted from the database.

```
query = """DELETE FROM league WHERE league_id = %s"""
```

Fig. 1.110: League Delete Query

SELECT QUERY

```
query = """SELECT * FROM league
          JOIN country ON country.country_id = league.league_country
          WHERE league_id = %s"""
```

Fig. 1.111: League Select Query

In case we pass no parameter to `get_league_by_id()` function, the query with no “WHERE” clause will be used. Above query is used when we pass an id parameter.

UPDATE QUERY

```
query_country = """SELECT country_id FROM country WHERE country_name=%s"""

query = """UPDATE league
          SET league_name=%s, league_country=%s, league_start_date=%s
          WHERE league_id=%s"""
```

Fig. 1.112: League Update Query

Just like in the add operation queries excluding update query gets referenced item ids and then use them as parameter in the update query.