**GROUP 27: John Geyer, Omar Paladines, Juan Padilla, Max Shestov**

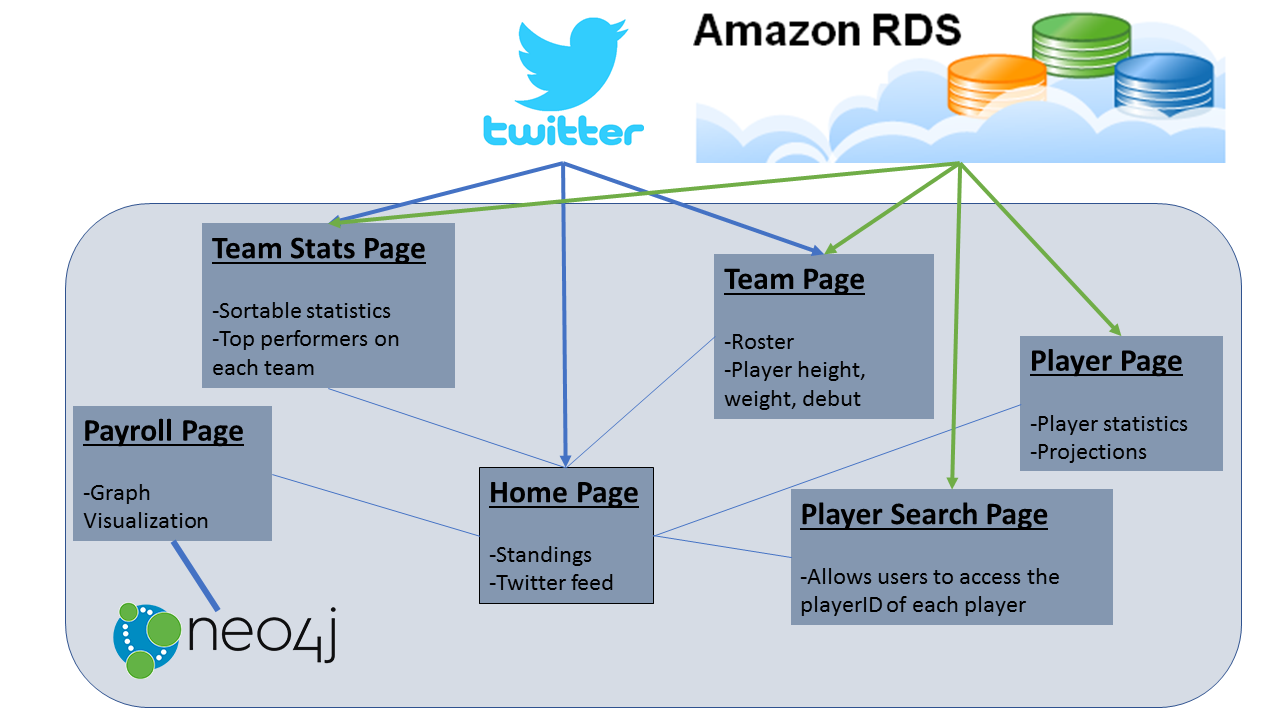
**CIS 550 Final Project: Baseball Data**

1. **Introduction and Project Goals**

Our project will provide a way of accessing, querying, and visualizing baseball statistics. Our goal is to provide users with an easy way to search for statistics about any player they choose, to see team statistics for all the active teams, and to have sortable leaderboards for each team so that they can see who the best performers are. In addition we also wanted to include player salary information in a visual way. Currently the only online resources list salary information and payroll data in tables, but our project uses Neo4J to provide a visual representation of this data. Lastly, we include current standings and a live twitter feed so that users can stay up-to-date on the current baseball happenings.

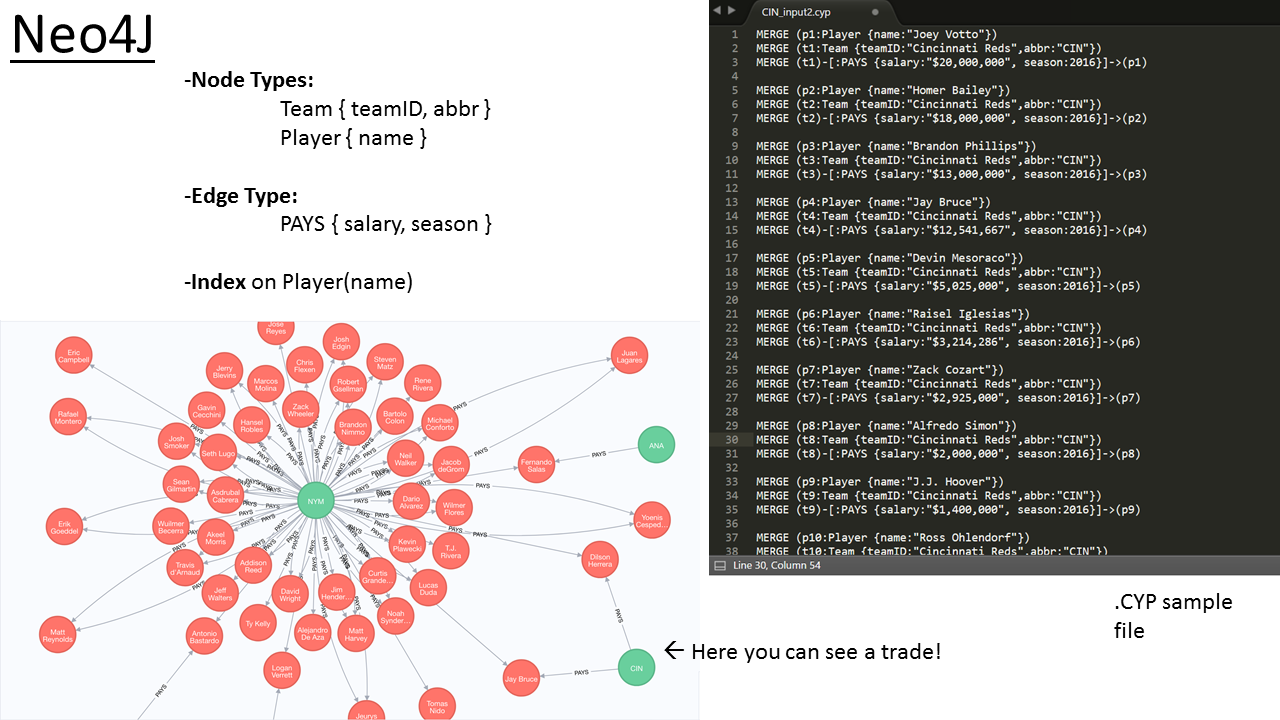
**2. Basic Architecture**

See the below diagram:



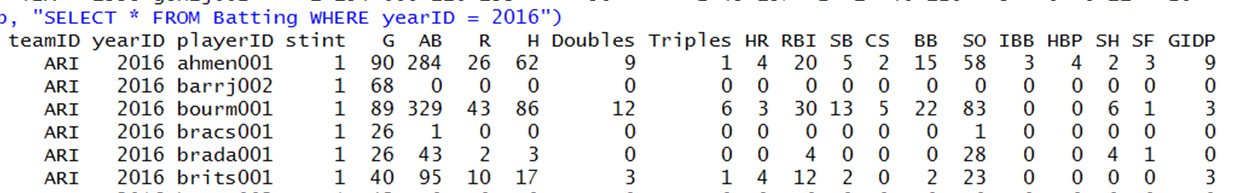
The Neo4J instance was hosted locally since hosting it on AWS cost money. Each of the thin blue lines represents links between pages. The Team Stats, Home Page, and Team Page all have twitter feeds. Moreover, each of the Team Stats, Team Page, Player Search Page, and Player Page access/query the MySQL instance hosted on AWS in some way or another. The Payroll page is the only page to interact with the Neo4J instance.

Below you can see the Neo4J instance information, and a small subset of the total graph that was stored. Moreover, we provided a sample .CYP file so that you may see how we uploaded our data into the Neo4J database:



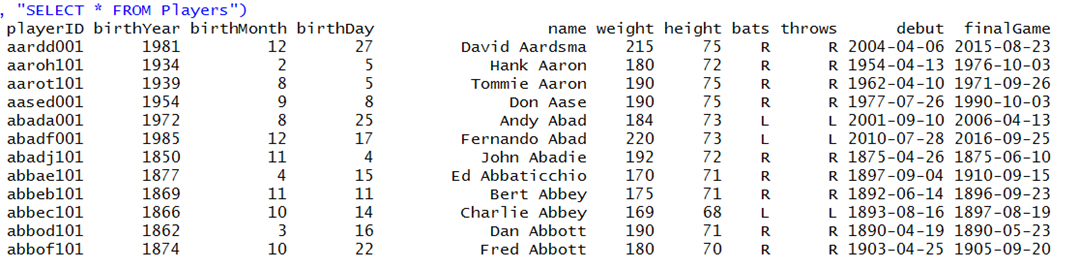
As for the relational database, we used a MySQL instance hosted on AWS. The tables in this database were:

1. Batting (with an index on playerID)
   1. Each tuple is a series of stats for a given player in a given year
   2. Contains stats from 1871 to 2016
   3. Columns:



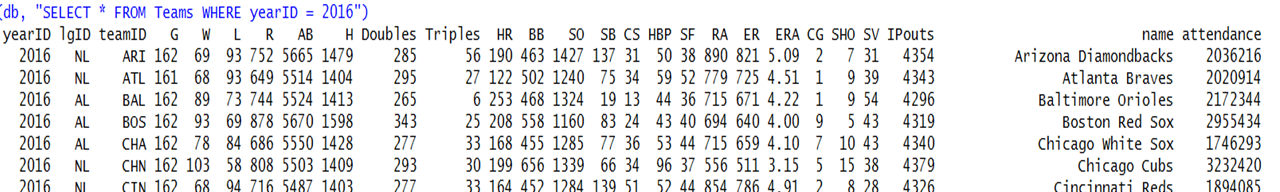
2) Players (with an index on name)

1. Contains player information for players born as early as 1820:
2. Columns:



3) Teams

1. Contains information about all teams (some of this data was unused but we wanted to have it just in case we extended our application)
2. Columns:



4) Twitter

1. Contains twitter account information for all the teams
2. Columns: twitter(team\_name, twitter\_feed)

5) active\_players

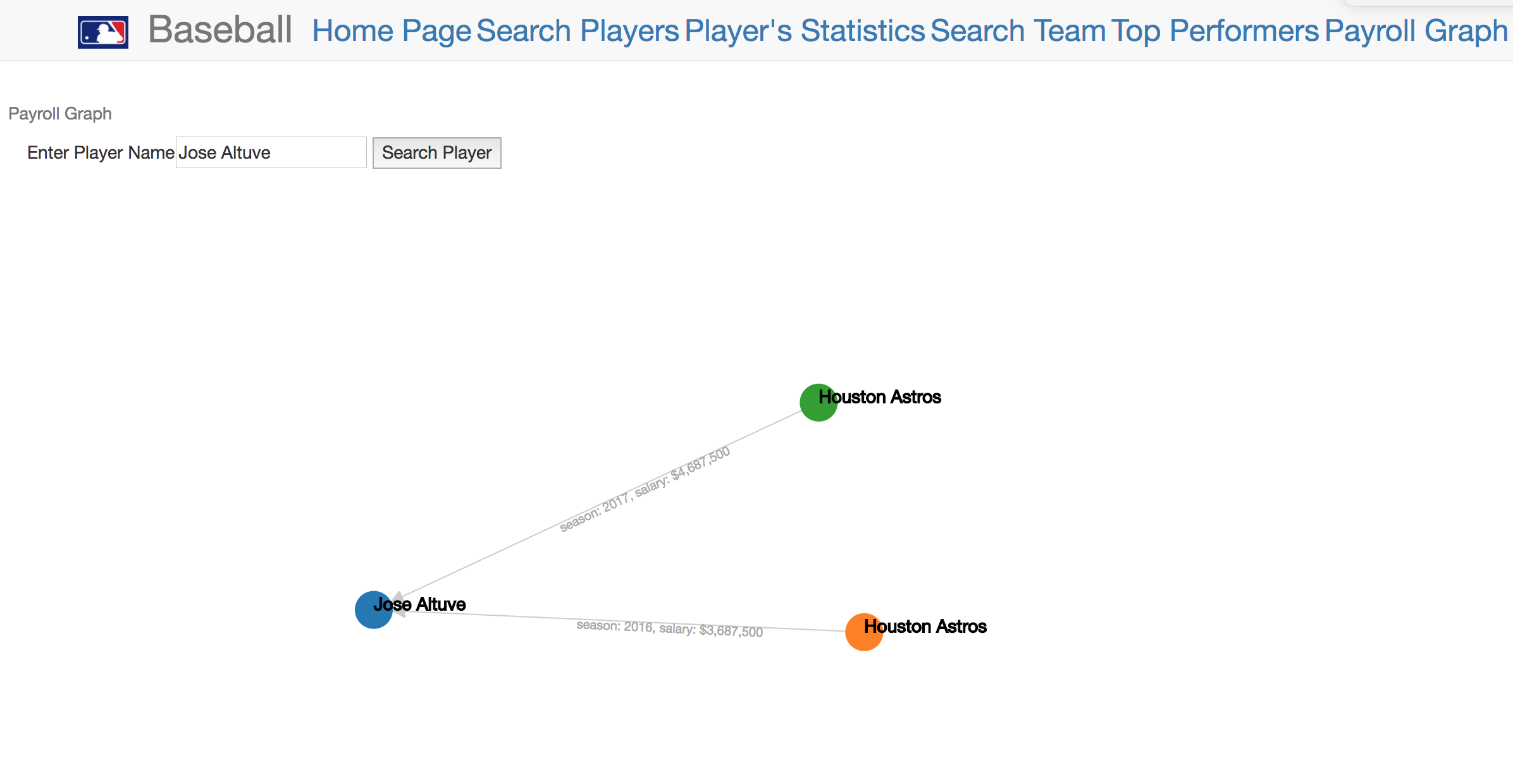
1. A view (not a table) that was useful for making projections
2. Columns: active\_players(playerID)

6) Other tables for future extensions:

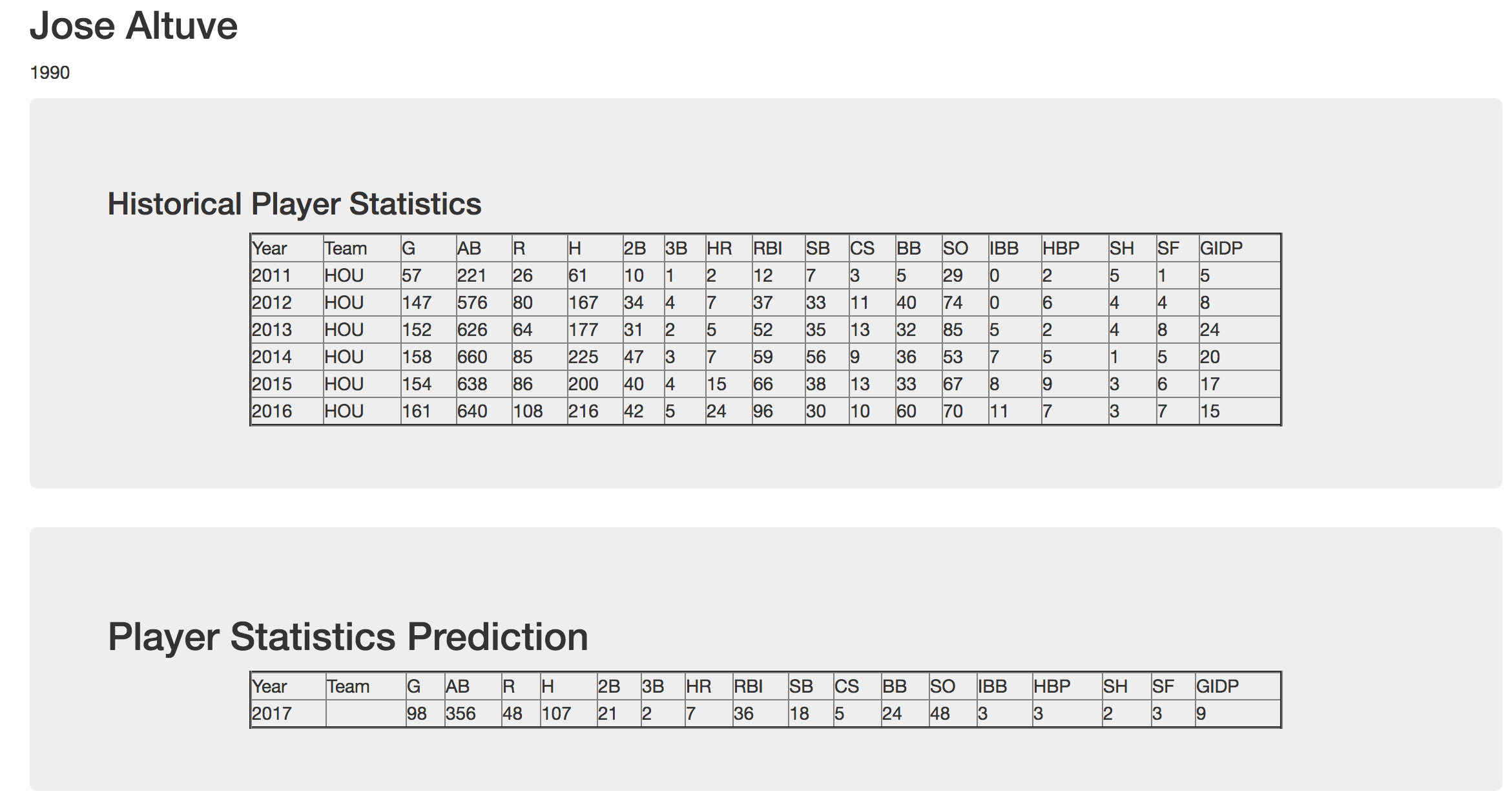
1. Pitching
2. Events
3. Games
4. Users (log in information)
5. Payroll (NOTE: this is NOT related to our Neo4J payroll data)

**3. Key Features**

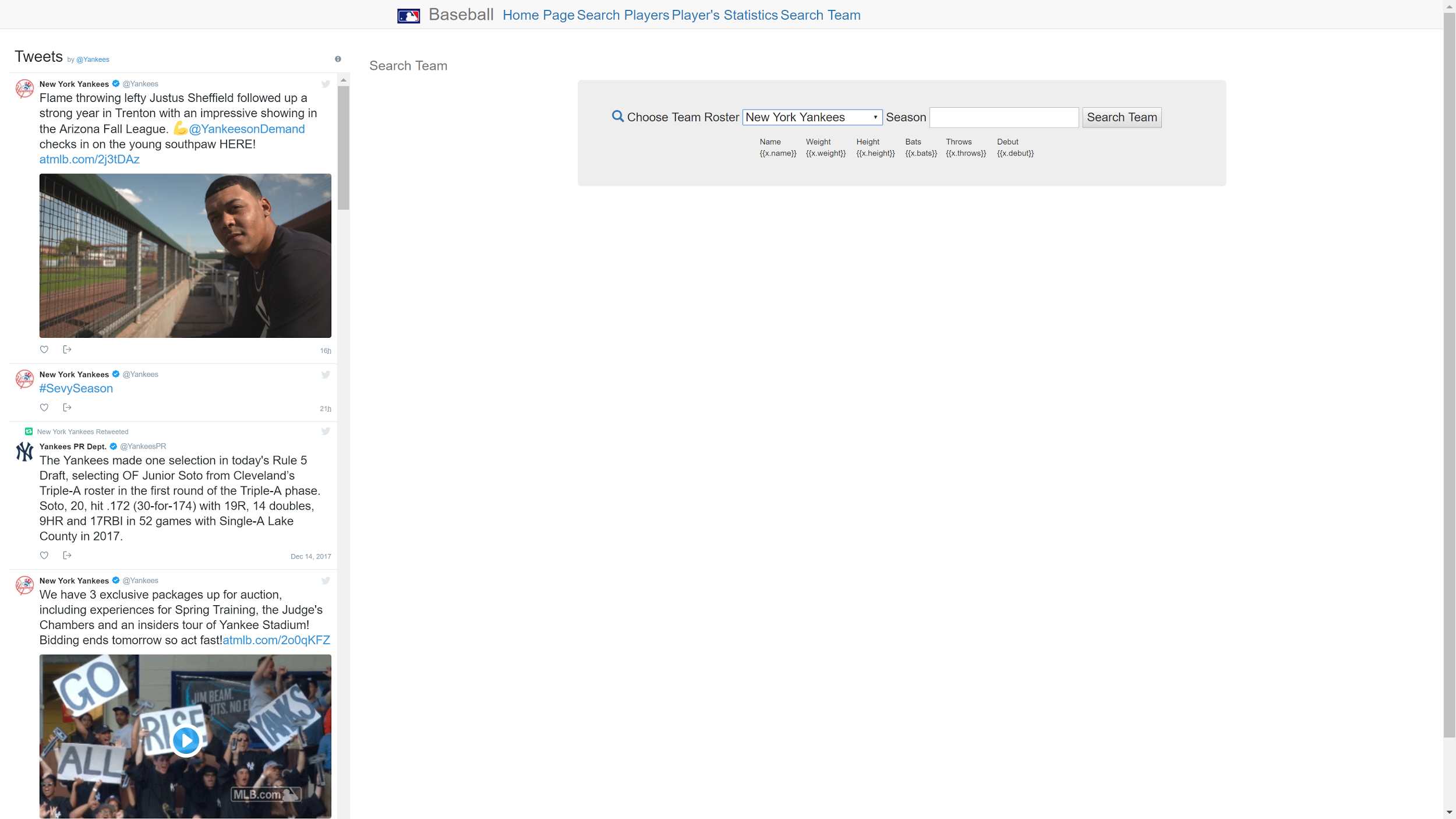
Payroll Graph: Our application provides a feature where users may search for any player (active in 2016 or 2017) and see a graphical representation of which teams paid him each year and how much. This graph representation is based on Neo4J: each player and each team is a node and there are edges of type PAYS(year, amount) from teams to nodes that give the specifics of how much the player was paid in each year. For some players who have remained on the same team for multiple years, the graph is rather uninteresting (one team node with multiple arrows to the player node); on the other hand, for players who have been traded multiple times, the graph becomes more complex since multiple teams may pay that player multiple salaries in multiple years.



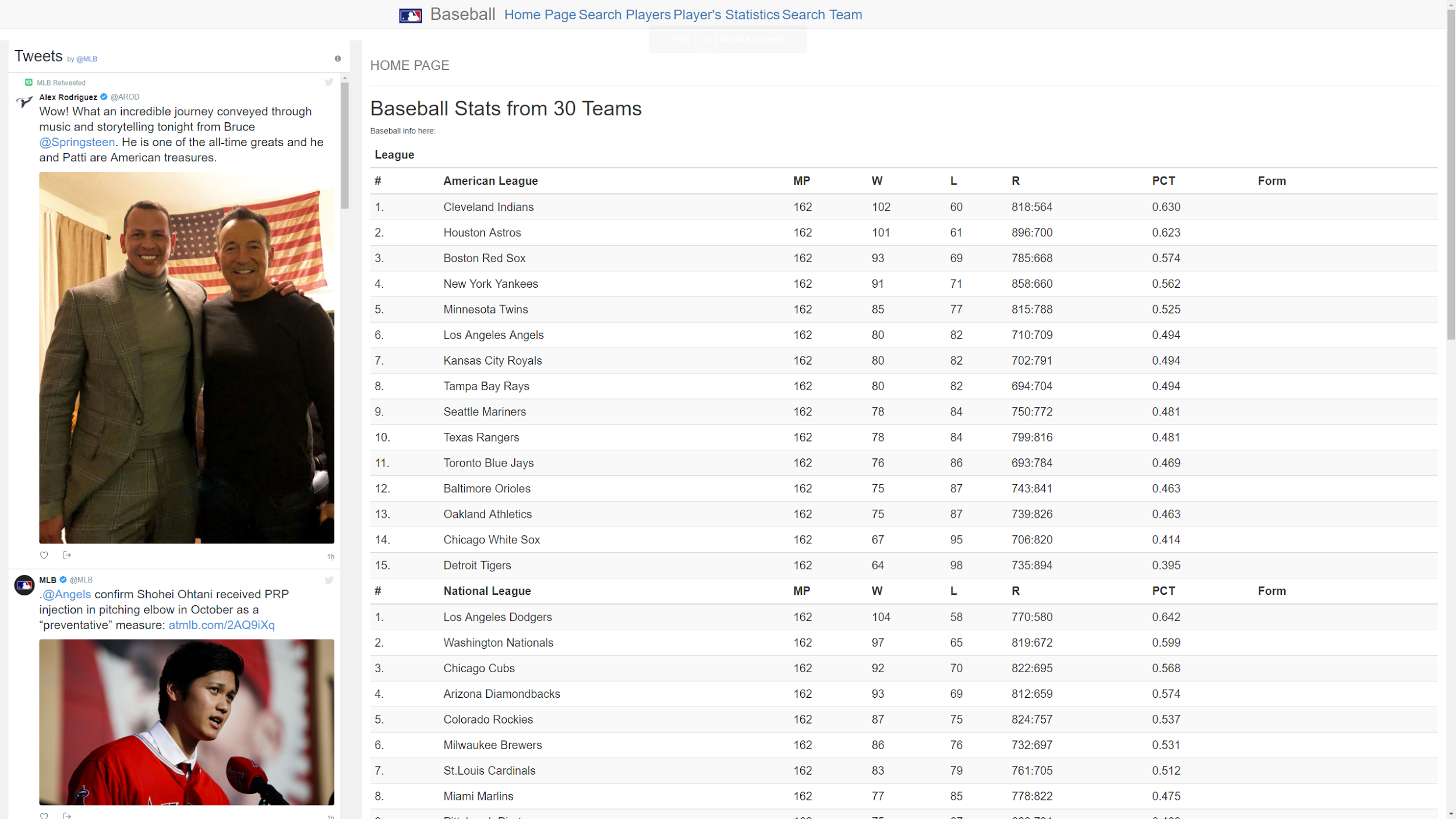
Player Statistics: Users have the ability to search for any player (dating from 1871 to 2016) and see their season statistics on the player’s “player page”. For players who played in 2016 (the most recent year of data), the player page offers a projected stat-line for the 2017 season. Based on a naive projection formula we found online, it simply takes a weighted average of the players previous seasons’ statistics and the all-time average of the statistics (which essentially acts as a ‘regression to the mean’ component).



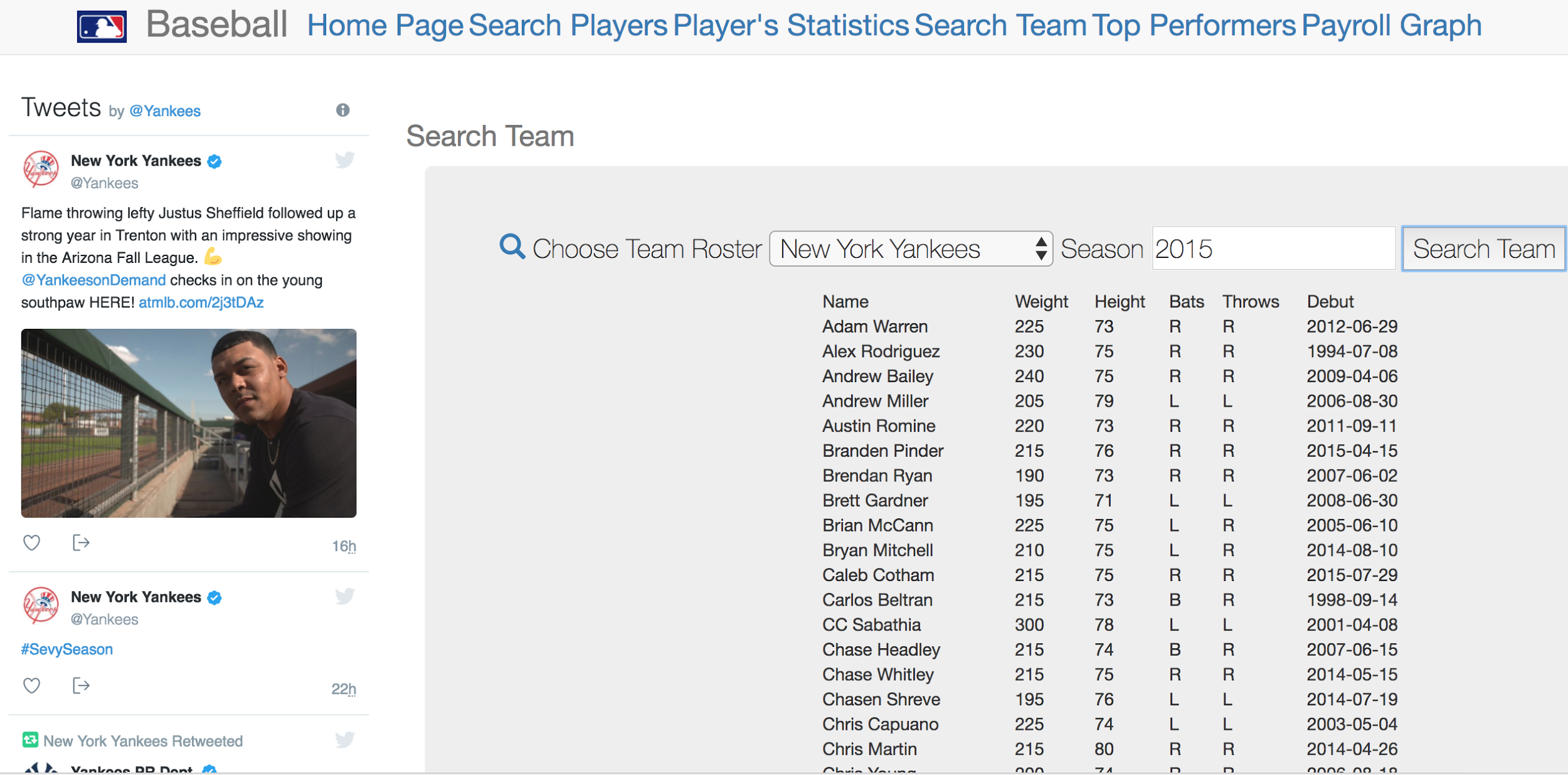
Twitter Feed: The Homepage of our application incorporates a live twitter feed for the MLB twitter page as well as the twitter pages of MLB teams that correspond to the selection made by the user such as the Red Sox providing users up-to-date info about baseball happenings.



Current Standings: The Homepage also includes the current MLB standings which dynamically update (unfortunately, since it is the offseason, there is no way to show off this functionality…) providing users a way to see current team standings/progress. This information comes from Scoreboard.com

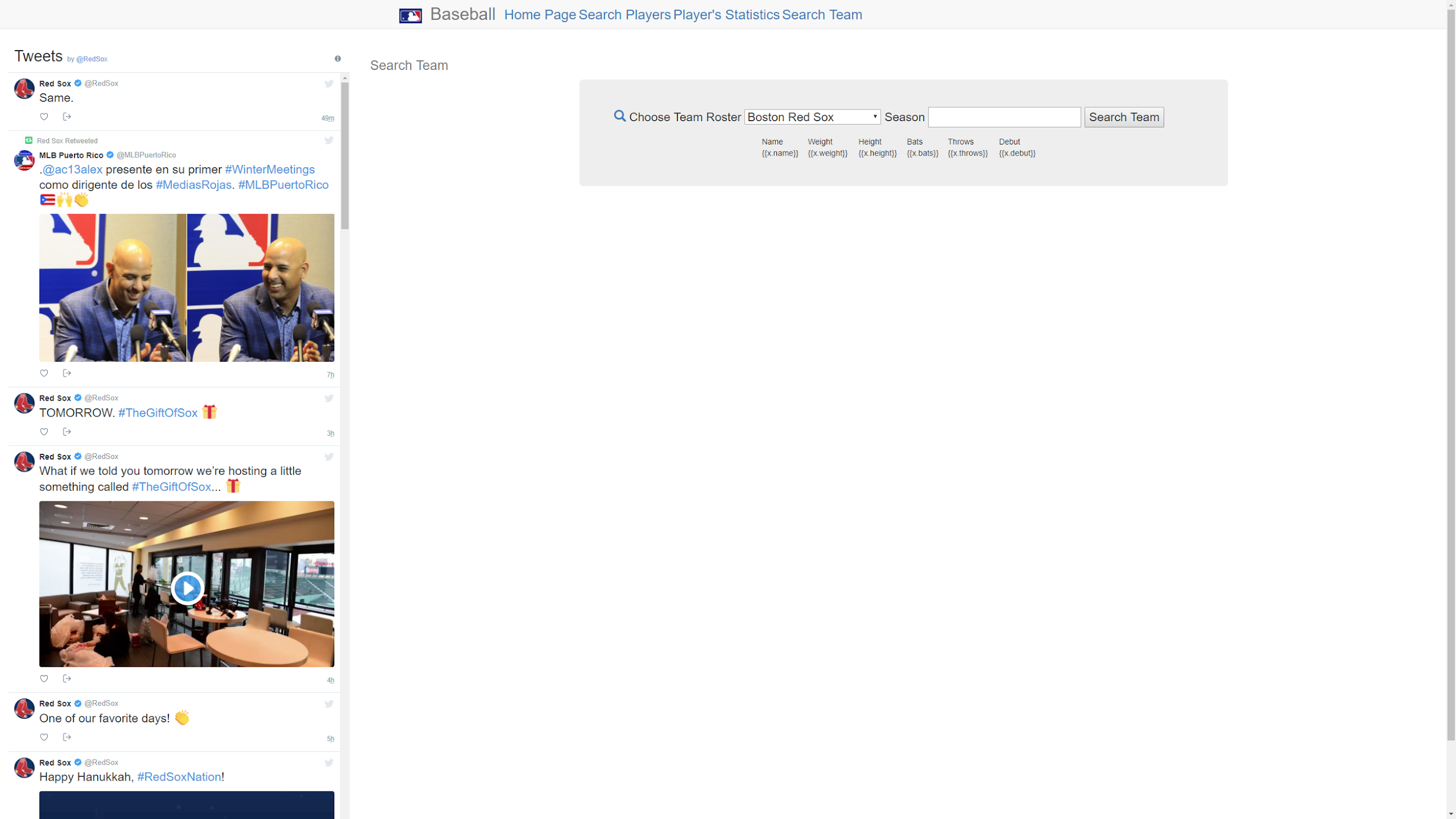


Team Statistics: Each of the 30 teams has a page (actually, it’s implemented as a single dynamically generated page…) where users can see the top players based on any statistic they choose. For example, users may see a list of the 2008 Cincinnati Reds roster sorted by Home Runs.

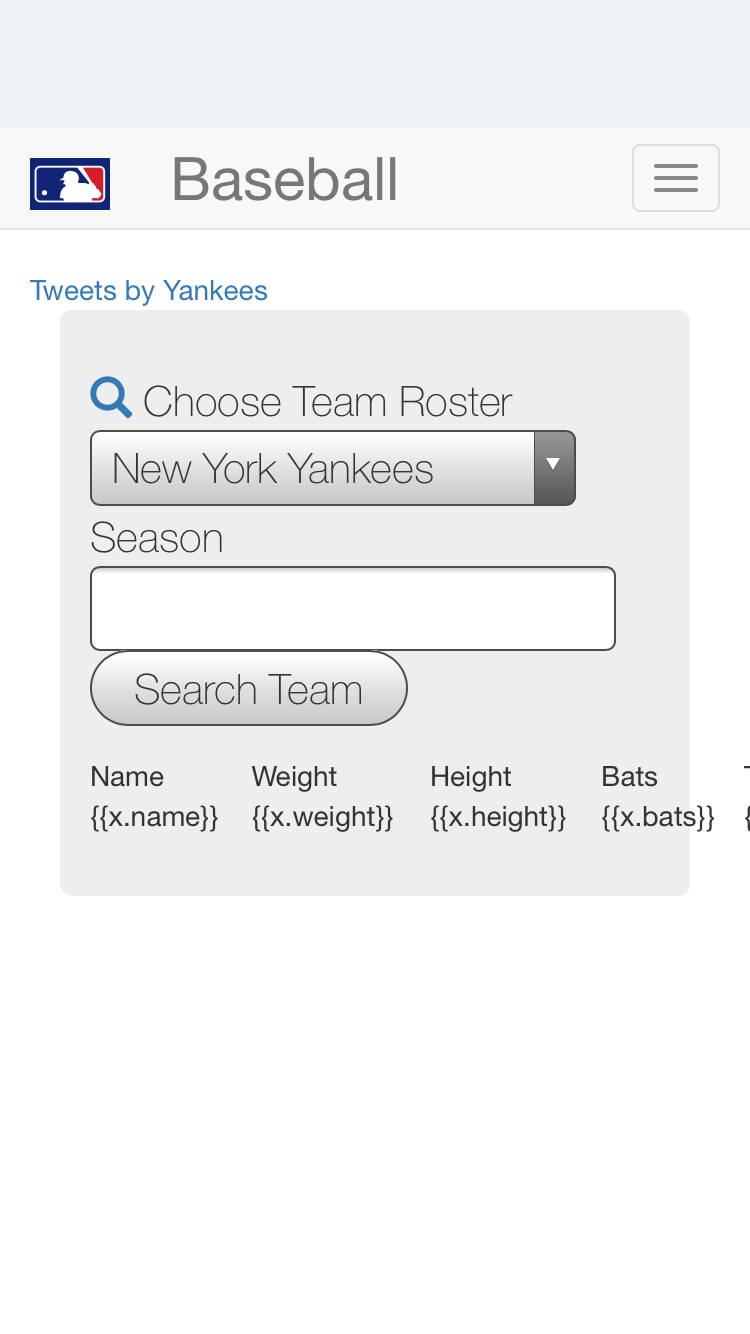


Team Rosters: Our web-application also provides the ability for users to see the rosters of each team, and for each player on the roster, their height, weight, handedness (lefty or righty) and the date of their major-league debut.

Team Twitter: Each team page has a twitter feed corresponding from that team’s official twitter page.

****

Mobile support: Our application scales well for different devices and supports mobile use with mobile menu option.



**4. Technical Challenges**

Loading and Cleaning Data: Obtaining the payroll data proved to be extremely tedious. The data was stored in a series of google-sheets online, and so we had to write a script which downloaded them all and converted them to .CSV files. Unfortunately, there was not a uniform naming convention for these files, so we had to manually rename each file in order to process it. Once the files were organized, we had to write a program that output a .CYP file so that it could be imported into Neo4J. Getting the formatting correct and ensuring that we didn’t create duplicate players/teams took a bunch of trial and error.

Importing into Neo4J also gave us problems: we found the application to be extremely slow; creating indices on the database helped a little bit, but unfortunately we were only able to input about two years of payroll data (we had planned to put in almost a decade’s worth…) before the Neo4J interface become unreasonably slow.

Moreover, as stated above, hosting a Neo4J instance on AWS cost money, so we made the choice to host it locally instead (we realize this decision leads to scalability issues, but we hope the instructors recognize that if this were a serious application, we’d pay the fee to host our instance on Neo4J). This led to some issues during the development phase since the database was not easily accessible by all our group members. Luckily, though, we divided up the work in a way that this didn’t become too much of an obstacle.

The only other challenging part was ensuring that all of our tables had the same playerID field. We obtained our data from Retrosheets and the Lahman database, and each uses their own identifiers for players. Luckily we were able to do some fancy JOINing to standardize these identifiers so that they all used Retrosheet ids.

Front End Issues: Most of our technical issues from the front-end came from inexperience with Javascript and HTML. Linking the databases to the front-end was not terribly problematic, nor were making queries and interacting with the databases once they were linked to the front-end . On the other hand, formatting turned out to be challenging, as did incorporating the twitter feeds and dynamically updating standings. This required a lot of time reading documentation and looking at online examples of how to integrate these components into our application. We also had some minor issues with AngularJS as transferring the data to the front-end also came with some formatting problems, but we were able to get around this by reading the appropriate documentation and adjusting our code accordingly. Specifically, we found problematic the Angular constraint that we can only perform a single query to populate the HTML with data from the back-end. As a result, we had to split functionality across different pages. For the most part, everything else went fairly smoothly.

**5. Performance Tweaks**

Since player-searches are conducted by searching for a player’s name to get his ID, then searching for his ID to get his statistics, we put an index on the “name” field in the Players table and on the “playerID” field in the Batting table to speed up queries. As a result, despite having decades of players and hundreds of thousands of tuples in our Batting table, querying the tuples for a given player is nearly instantaneous.

With respect to our NoSQL database, I mentioned briefly above that importing data took an unreasonably long amount of time--to the point where we couldn’t even import a half-season’s worth of payroll data. We are still unsure whether this is due to the Neo4J GUI being slow or if it was our data. In any case, we decided to put an index on the “name” field of our Payroll data so that when importing the data, MERGE statements could be performed more quickly. Indeed, we found that this sped up the import process fairly substantially: in the end we were able to import two full seasons of payroll data before the imports became unreasonably slow. Unfortunately, we didn’t know what other optimizations to make so that we could import the entirety of our data (which stretched back to 2009) without the GUI slowing down…

**6. Description of Complementary Data and Extraction**

We used R for all of our data cleaning, extraction, and database writing. The RMySQL library provided the functionality needed to interact with the AWS MySQL instance. Moreover, R has built in dataframe types that are very conducive to modifying tables and writing to relational database tables. Since R has a built in read.csv function, our goal was to get all of our data into .CSV files to make importing into the database as easy as possible. Here’s how we did that for all of our various data sources:

Our payroll information came from Cot’s Contracts. This was certainly the most high-maintenance data extraction/cleaning we did. The payroll data was stored in a series of Google Sheets files, one file per team per year. Hence we had to write a script that downloaded all of these files, and converted them to .CSV files. Unfortunately, the names of the files didn’t follow any special format, and so we had to manually rename all of them. Once this was done, we wrote a script in R to parse the files, import them as dataframes into R, used R to drop the unnecessary columns, then wrote them out to .CYP files in Cypher. This took a lot of trial and error to get the syntax correct and moreover, there were several players with ‘nicknames’ which we had to manually fix (looking at you Jose “Jumbo” Diaz…). From here, it was relatively straightforward to import these .CYP files into the Neo4J database.

Our baseball statistics data came from two sources: the Lahman database and Retrosheet. We accessed the Lahman database by importing it using R. From here we had access to the entire database in the form of R dataframes, from which it was relatively straightforward, albeit tedious, to modify the dataframes by selecting only the columns/tables that were relevant to our application. We had to do a bit of renaming of the columns (there were problems on the front-end when the column names began with a number, so we had to change “2B” to “Doubles” for example).

The Retrosheet data had to be downloaded in “event” files (a .EVN or .EVA extension). We had to download software that extracted the information in these types of files and exported them to .CSV files. We had to write a BASH file that did this for all of the event files (there were a couple dozen of them), and once this was done, we could upload the .CSV files to R and clean the data appropriately.

One major issue we had to overcome was that Retrosheet and Lahman used different identifiers for players. In order to standardize our tables, we (arbitrarily) chose to use the Retrosheet playerIDs. Hence for all the Lahman data, we had to do some (somewhat complicated) JOINing of dataframes to match up players and change their playerID’s from Lahman IDs to Retrosheet IDs.

The payroll data was kept pretty separate from the other data, however the Retrosheet and the Lahman data were used together pretty extensively in our application. You’ll see in our code that we several joins between these tables in some of our functions (which is why we had to standardize the playerID’s to all be Retrosheet IDs).

**7. Future Extensions**

The first obvious extension we’d like to make is to include Pitching data as well. We actually have the pitching statistics going back to 1871 stored in the Pitching table in our SQL database for this reason. We’d also like to implement a more sophisticated prediction system, perhaps one that takes into account opponent talents and park-factors. In addition, incorporating some sabermetric statistics would also be a cool extension: with the data we currently have in our database, we could extend our web application to include stats like Field Independent Pitching (FIP) and Wins Above Replacement (WAR) into our player statistics page.

As another possible extension, we’d like to somehow combine the Payroll data and the player statistics. In particular, we would like to answer questions such as “How much did the average home run cost?” by comparing the number of homeruns hit by each player to their salary. Clearly we could do this for other statistics beyond home runs too.

Lastly we have a few extraneous tables in our relational DB, namely Pitching, Events, and Games which are there for use in possible extensions of our application. The extensions involving the Pitching table have already been mentions. The Games table contains the scores, team statistics, and winner/loser for each game over the past few decades. We’d like to extend our application to have additional functionality so that it makes use of this data: for example, by having a “this date in history” where users can see the games played on this date in years past. The Events table contains play-by-play information for every game over the past few decades as well. We were thinking it might be neat to use this to to create a feature that gives batter-pitcher matchup statistics. For example, it could be used to see how well player A performed when batting against pitcher B (and vice versa).

**Group Members’ Contributions**

John’s Contributions: Responsible for finding, extracting, and cleaning the data for the relational database. Also set up the MySQL database on AWS, populated it, and put indexes on certain columns to speed up common queries. Given the raw data for the payroll information, I re-formatted all the files so that they had a standardized naming scheme, uploaded them into R and cleaned the data, and then wrote the cypher .CYP files that were used to import the data into the Neo4J instance. Basically, I handled most of the backend database set-up, data extraction, data cleaning, and database population.

Max’s Contributions: Wrote script to download payroll data from the web and put it into .CSV files. Also designed the homepage, including incorporating a dynamically updating team standings chart which displays team winning percentages found on the web. Incorporated twitter feeds both on the homepage and on each individual team page using javascript to update twitter widget for corresponding teams when populating views. Designed the homepage setup and the navigation bar across all the views to be aesthetically pleasing and scalable across different platforms including mobile support.

Omar’s Contributions: Developed full-stack functionality for **Searching Players**: It shows all of the players, along with their date of birth and player ID, whose prefix matches the query string. In addition, developed full-stack functionality for **Player Statistics**, which shows all the batting statistics given its query id, and filtering for a range of given years. In addition, I incorporated predictions for the statistics for batting for players that are currently active. Finally, I developed full-stack functionality for the **Payroll Graph**, which shows the salaries and the teams for a given player during 2016-2017. I used open source Javascript library D3.js for graph visualization in a force layout graph style.

Juan’s Contributions: Created query/front-end functionality for **Team Rosters,** which shows all the players along with their height, weight, L/R for batting, and debut date for a given team and a given season, i.e. “show me the team roster for the 2003 New York Yankees”. Also, created **Top Player** functionality/front-end that queries the players with the highest stats for a given category, team, and season, i.e. “give me the list of the players with the most home runs for the 2012 Philadelphia Phillies” (returns a list of the players sorted by their home run numbers in descending order).