Django Tango

Internet Baseball Database

Kendall Ahrendsen - Design, front end, Bootstrap, social media integration.
Clare Coleman -Apiary document, design, front-end, API testing, Bootstrap.
Deneb Garza - Template inheritance, front-end, UML, raw data parsing.
Jordan Graves - (Back-end) Models/API/Search Implementations and testing.
Tim Simmons - Basic site architecture using Django, site content, client page, SQL page, general help in all aspects.

Introduction

Everyone knows baseball is America's favorite pastime. What most people may not be aware of is the wealth of information and statistics available about the game. Every action on the field is recorded and used to formulate statistics and metrics by which fans and baseball executives may evaluate players and teams. The rich history of baseball is well documented, and many fans enjoy learning about the game's past to better their understanding of its current state.

The Internet Baseball Database (IBD) aims to help those who enjoy reviewing the sport's thoroughly documented history by providing quick and easy access to and navigation of comprehensive data on famous players, current teams, and the recent Major League Baseball seasons. It is a hub for information on historical performance of players, as well as current news, embedded social media, images, and video showcasing the events in a player's career.

Users of the IBD may simply be trying to remember who the Most Valuable Player was three years ago, how a particularly exciting playoff season played out, or perhaps to brush up on their knowledge of their favorite player and his humble beginnings. The IBD is a hub for information on historical performance of players, as well as current news, embedded social media, images, and videos showcasing the events in a player's career. The most common use case of the IBD is someone who is looking for information on a player, team, or season. The IBD displays everything they would ever have wanted to know in an attractive, convenient format.

Table of Contents

Models

What are Django Models?

Django models are Python classes that represent a table in a database. An instance of one of these Python classes represents an entry in the table. Each field of the model represents a data field or column of a table. These models serve as a convenient interface into Django's database by allowing the instances of these Python classes to create, retrieve, update or remove the database entries they represent.

Internet Baseball Database Models

There are eight models we use to organize our data:

- Player
- Player_Year
- Team
- Team_Year
- Year
- Player_Image
- Team_Image
- Year_Image

Their relationships are as follows:

- Player_Year to Player, many to one
- Player_Year to Year, many to one
- Player_Year to Team_Year, many to one

- Team_Year to Team, many to one
- Team_Year to Year, many to one
- Player_Image to Player, many to one
- Team_Image to Team, many to one
- Year_Image to Year, many to one

Player and Player_Year

The Player model has a composition relationship with the Player_Year model. Each Player model contains information about an individual player that remains persistent throughout a player's career, for example, a player's name, school and handedness. The Player_Year model, on the other hand, holds information about a player which is likely to change from year to year. For example, any yearly statistics, such as batting average or the team to which the player belonged for a specific year, would be attributes of this model.

To establish a link between the Player and Player_Year models, we include in the Player_Year model a ForeignKey to a Player model. This establishes a many to one relationship from Player_Years to Player. Django automatically maintains the backwards relation from Player to Player_Year. By using the option 'related_name' of the ForeignKey, we specify a meaningful name for the backwards relation which allows us to retrieve all Player_Years from a Player.

We want a similar relationship between the Player_Year and Team_Year model so that with an instance of a Team_Year, we can retrieve a set of Player_Year models that represent the players that were on the team represented by that Team_Year model. We establish the same connection with the Year model allowing for us to link to the players' specific statistics for a given

year.

With an instance of a Player model we can get all Player_Years that have that Player as a ForeignKey field.

Example:

```
class Player(models.Model):
        ...

class Player_Year(models.Model):
        player = models.ForeignKey(Player, related_name='player_years')

        def(__unicode__):
            return self.id
```

```
>>> player = Player(...)
>>> player_year1 = Player_Year(player=player)
>>> player_year2 = Player_Year(player=player)
>>> player.player_years.all()
>>> [<Player_Year: 1>, <Player_Year: 2>]
```

Team and Team_Year

The Team and Team_Year models are represented in the same way as the Player and Player_Year models, with the Team_Year model containing a ForeignKey field which links to a Team model. Since we will also want a link to individual Team_Years from a Year instance, we do the same for the Year model. The Team_Year to Year relationship allows us to filter all Team_Years by year using the Year model (so that we can display, for example, the team rankings for that year, an attribute held in the Team_Year model. The Team_Year to Team relationship allows us to organize statistics for an individual team by year.

```
class Team_Year(models.Model):

        team = models.ForeignKey(Team, related_name='team_years')
```

```
year = models.ForeignKey(Year, related_name='team_years')
```

Player, Team, Year Images

Players, Teams, and Years can have multiple images associated with them. Each image has a type, perhaps the logo or the stadium. This multiplicity relationship necessitates a separate model for images, so that they can be retrieved as a list, and accessed using their type.

Year

The Year model serves as a filter for standings across teams for a particular year and allows us to easily show year-to-year rankings of Teams with links to each Team's Team_Year. This is accomplished by placing the Year model as a Foreign Key inside of a Team_Year which holds that team's ranking for that year.

The Year model also contains Foreign Keys to Player_Year models, which represents the award winners for various awards for that year.

Implementation

Frameworks

Django is the main technology used in the Internet Baseball Database. It is used to route URLs, generate dynamic pages, and interact with the database.

Twitter Bootstrap is used in the Django Templates to make them attractive. There are also many static elements in the pages from Bootstrap's library of CSS and HTML objects that are incorporated in the IBDB. Bootstrap makes use of JQuery, HTML and CSS. Javascript and JQuery were also used in the making of some dynamic web pages.

Tools

We used Apiary for API Documentation, it does a great job of providing a clear reference for the RESTful API.

The 'psql' PostgreSQL client was used in the making of the SQL page.

The Django Admin interface was used extensively to view and masage data in the database.

Python regular expressions, itertools, unittest, json, and Django QuerySets were used in the making of the API and search functions and their testing.

The excellent requests package was invaluable in testing, and in the making of the API for easy HTTP calls.

The website is served on Heroku using WSGI.

Datastore

The IBDB uses PostgreSQL as it's database, with the Django ORM used primarily for access.

Processes

The Django Tango group used HipChat for day-to-day project communications. It proved very effective in keeping everyone in sync.

The workflow as described here was used to coordinate code on GitHub. Developers cut new branches for each addition, and made pull requests to be reviewed by others before merging into the main codebase. The changes were then pushed to Heroku.

The raw data was put into JSON structures in idb/data, and subsequently used by idb/populate.py to populate the database should a change ever be made. This was very effective, as whenever a change to the models necessitated a new database, the repopulation was painless.

API

The Internet Baseball Database has an JSON API that allows users to programmatically retrieve the information stored in the database. This allows users to develop applications using the IBD easily, avoiding the need to scrape the IBD website for data. It also allows the IBD developers to maintain an idea of how their data is being used by giving out API keys to potential developers, as well as make a small profit when applications that make a certain number of queries are charged accordingly. The API can be accessed at:

http://frozen-plateau-5382.herokuapp.com/api/resource

/players

The players endpoint of the API contains resources related to Major League Baseball players.

GET	/players
GET	/players/id
POST	/players
PUT	/players/id
DELETE	/players/id

Player Information

id	The IBD assigned identifier for players in the IBD
name	Player names, first and last
social	Twitter handle, if available
number	Jersey number
bats	The handedness of the player when hitting
throws	The handedness of the player when throwing
height	Height, in inches
weight	Weight, in pounds
school	The last school the player attended, could be a high school, college, or international location

Example Response

[GET] /players/1

```
Response 200 (application/json)

        {
        "model": "idb.player",
```

```
"fields": {
       "name": "Bryce Harper",
       "school": "College of Southern Nevada",
       "social": "@Bharper3407",
       "weight": 230,
       "number": 34,
       "position": "OF",
       "height": 74,
       "throws": "R",
       "bats": "L"
},
"pk": 1
}
```


/players/id/years

The player years endpoint of the API contains resources related to Major League Baseball seasons for individual players.

GET	/players/id/years
GET	/players/id/years/id
POST	/players/id/years
PUT	/players/id/years/id
DELETE	/players/id/years/id

Player Year Information

id	The IBD assigned identifier for a player year
year	The year of the season, e.g. 2013
team_year	Three-letter abbreviation for the team that the player played for, e.g. TEX
type	The type of player; choices include [hitter, pitcher]
games	The number of games in which the player appeared
player_id	The id of the player to whom the year belongs

Hitter Attributes

There are two types of players: hitters and pitchers. They do different things on the field, and thus they have different attributes associated with their performances.

Hitters

pa	Plate appearances: the number of times the hitter took a full turn in the batters box
avg	Batting average: the percentage of plate appearances that the hitter earned a hit
obp	On-base percentage: the batting average, in addition to the percentage o base on balls (walks), that the hitter earned
slg	Slugging percentage: similar to batting average, except that hits for more bases count for extra hits. For example: ● Singles count as one hit ● Doubles count as two hits ● Triples count as three hits ● Home Runs count as four hits
hr	The number of home runs the player hit
rbi	The number of runs batted in that the hitter had

Pitchers

w	Wins: the number of pitching wins a pitcher earned
l	Losses: the number of pitching losses a pitcher suffered
g	Games: the number of games the pitcher appeared in, equal to the number of games started plus games appeared in relief
gs	Games started: the number of games the pitcher started
era	Earned run average: the average number of earned runs scored against a pitcher in nine innings (one game)
s	Saves: credited to a pitcher who finishes a game for the winning team under certain prescribed circumstances.
ip	Innings pitched: the number of innings the player pitched

whip	Walks + Hits Per Inning Pitched

Example Requests/Responses

[GET] players/id/years/2012

Response 200 (application/json)


```
{
"model": "idb.player_year",
"fields": {
        "gs": null,
        "pa": 597,
        "rbi": 59,
        "year": 2012,
        "team_year": 32,
        "avg": 0.27,
        "w": null,
        "whip": null,
        "player": 1,
        "hr": 22,
        "l": null,
        "ip": null,
        "slg": 0.477,
        "kind": "hitter",
        "games": 139,
        "s": null,
        "era": null,
        "obp": 0.34
},
"pk": 1
}
```

/teams

The teams endpoint contains information on the teams that participate in Major League Baseball.

GET	/teams
GET	/teams/id
POST	/teams
PUT	/teams/id
DELETE	/teams/id

Team Information

id	The IBD assigned identifier for a team
name	The team name
abbr	The three-letter abbreviation for the team
city	The city in which the team plays
state	The state in which the team plays
park	The name of the stadium in which the team plays
div	The division of the league in which the team plays, e.g. NL East
social	The Twitter handle of the team
mgr	The team manager

Example Requests/Responses

[GET] teams/id/

```
Response 200 (application/json)
       {
       "model": "idb.team",
       "fields": {
              "name": "Los Angeles Angels of Anaheim",
              "state": "CA",
              "mgr": "Mike Scioscia",
```

```
        "abbr": "LAA",
        "city": "Los Angeles",
        "park": "Angel Stadium of Anaheim",
        "social": "@Angels",
        "div": "AL West"
    },
    "pk": 1
    }
```


/teams/id/years

The team years endpoint of the API contains resources related to Major League Baseball seasons for individual teams.

GET	/teams/id/years
GET	/teams/id/years/id
POST	/teams/id/years
PUT	/teams/id/years/id
DELETE	/teams/id/years/id

Team Year Information

id	The IBD assigned identifier for a team year
year	The year of the season, e.g. 2013
wins	The number of games the team won in the regular season
losses	The number of games the team lost in the regular season
standing	The position in which the team finished in the division
playoffs	The playoff finish of the team for a season, empty if they did not make the playoffs
attend	Attendance: the number of people who attended home games during the regular season
payroll	The payroll for players for the year

Example Requests/Responses

[GET] teams/id/years/2013

Response 200 (application/json)


```
{
"pk": 1,
"model": "idb.team_year",
"fields": {
        "standing": 3,
        "playoffs": "",
        "attend": 3019505,
        "wins": 78,
        "losses": 84,
        "payroll": 116032500,
        "year": 2013,
        "team": 1
}
```


/years

The years endpoint of the IBD API contains information about Major League Baseball seasons, and the individual seasons of teams and players who participated in them.

GET	/years
GET	/years/id
PUT	/years/id
DELETE	/years/id

Year Information

id	The IBD assigned identifier for the year
champion	Winner of the World Series for the year
AL_MVP	The American League MVP for the year
NL_MVP	The National League MVP for the year
NL_CY	The National League Cy Young Award winner for the year

AL_CY	The American League Cy Young Award winner for the year

Example Requests/Responses

[GET] year/id/2013

Response 200 (application/json)


```
{
"pk": 2004,
"model": "idb.year",
"fields": {
"AL_CY": "Johan Santana",
"champion": "Boston Red Sox",
"NL_MVP": "Barry Bonds",
"NL_CY": "Roger Clemens",
"AL_MVP": "Vladimir Guerrero",
"year": "2004"
}
```

Unit Tests

Unit tests provide a modular means of ensuring that individual components of the IBD API are functioning properly. Each test mimics an HTTP request and verifies that the responses are correct, which will depend upon the request method. There are four cases to consider: GET, POST, PUT, and DELETE. In each case, the test will examine the data returned, which may be treated as JSON objects if there is any, and the response code.

GET

The IBD API makes use of GET in two general circumstances: retrieving a list of entities (players, teams, years), and retrieving a single one of those entities. The two cases are tested similarly; both should have a response code of 200, and then a sampling of the data of a player, team, or year is tested against known values. If the response includes a list of JSON objects, the length of the list can be tested as well as objects with in it.

POST

The IBD API uses POST in the creation of new players, teams, and years. The response code should be 201, indicating a successful creation, and the data returned will be a JSON object that may have a sampling of its data tested against known values.

PUT

The IBD API uses PUT in the modification of a single player, team, or year. The response code should be 200, indicating a successful query, and the returned data should be a single JSON object whose data may be tested against known values.

DELETE

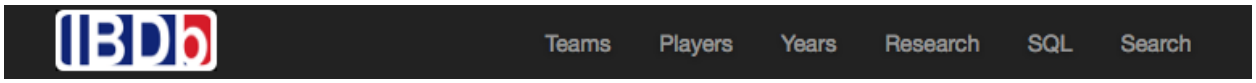
The IBD API uses DELETE in the removal of a single player, team or year. The response code should be 204, indicating no data. In this instance, attempting to load returned data as JSON will throw an error.

SEARCH

The IDB API includes a search feature for simple string queries. Tests for the search engine are written using http calls and verify expected results for a substantial number of specific queries.

Design and Navigation

The IBD website can be easily navigated using the toolbar at the top of all pages.



The menu items for Teams, Players, and Years also allows for a quick view of all of the available resources.

Teams	Players	Years
Boston Red Sox	 Albert Pujols	2004
Detroit Tigers	 Andrew McCutchen	2005
	 Bryce Harper	

Individual Team, Player, and Year pages link to each other when available as well.

Miguel Cabrera

#24

Position: 1B
Bats: R
Throws: R
Height: 76
Weight: 240
School: Maracay, Venezuela

Year	Team	G	PA	BA	OBP	SLG	HR	RBI
2004	MIA	160	685	0.294	0.366	0.512	33	112
2005	MIA	158	685	0.323	0.385	0.561	33	116
2006	MIA	158	676	0.339	0.43	0.568	26	114
2007	MIA	157	680	0.32	0.401	0.565	34	119
2008	DET	160	684	0.292	0.349	0.537	37	127

Design Choices

We decided that pages that had immediately available information would be served as cleanly as possible, to emphasize the data as the focal point. These pages, like the individual resource pages (players/id, etc) were to be as simple as possible. On pages that had little readily available data, like the splash page, and the search page, we decided to use a background image.

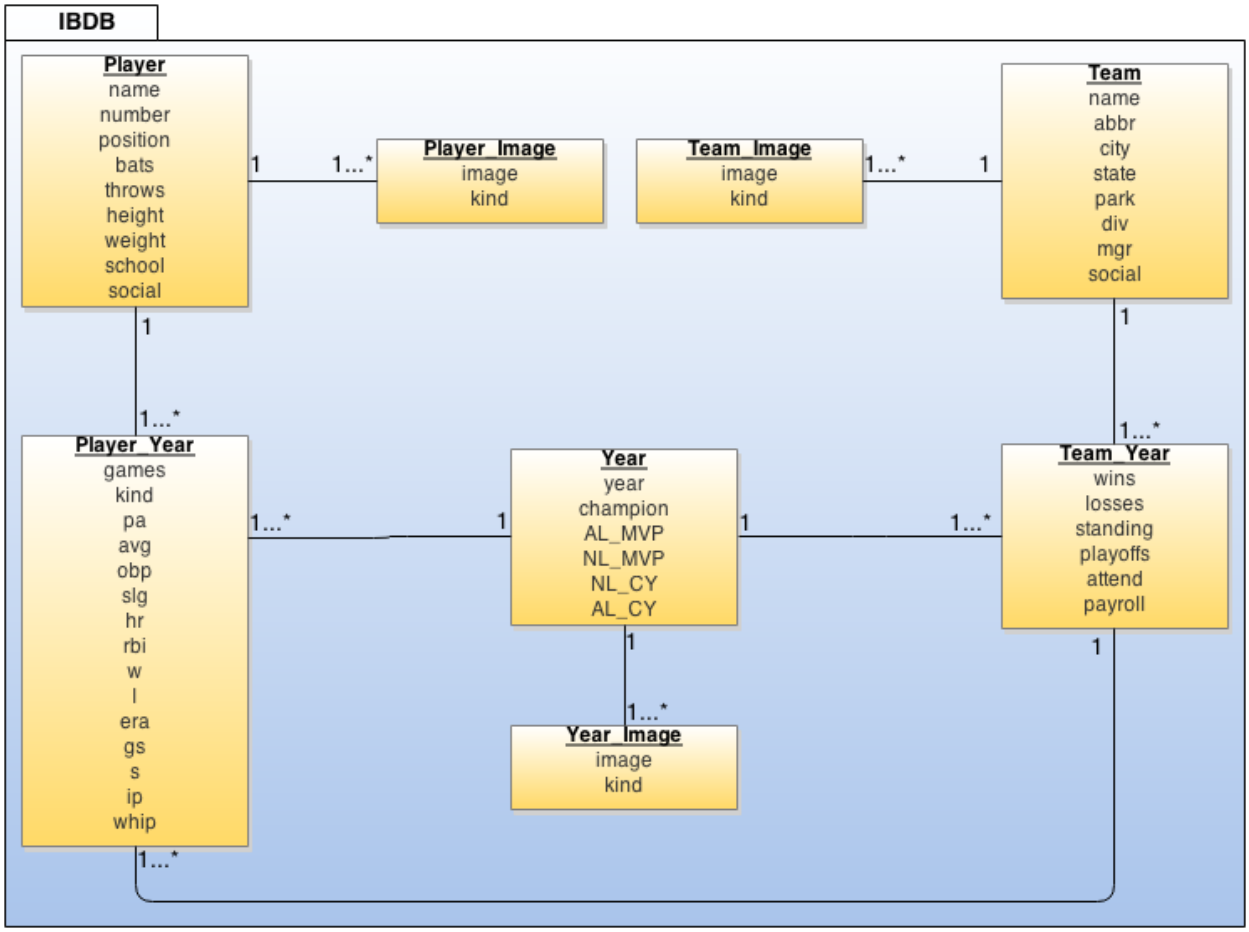
We wanted the sub-splash pages (/players, /teams, /years) to again be focused on the data, but with a little more flair, so we used our available images to help users make choices on where to click.

Template Inheritance

In order to maximize code re-usability and maintenance efficiency, we used Django template language's inheritance capabilities. Template inheritance allows for the sharing of common code from a parent template to child templates who inherit from it. If multiple templates share the exact same code, it is generally a good idea to extract the code into a parent template which can be inherited from.

All of our templates share common code which includes the header, navigation bar and other boilerplate code such as scripts and styles. We created a “base.html” parent template file to include all common code. This allowed for all our other templates to share one copy of this common code, which made maintenance a lot easier. When wanting to change something on the navigation bar or main style, we could simply change the code for base.html and have the changes be reflected in all child templates.

UML Diagram



This diagram shows the Django Models and the relationships between them in the Internet Baseball Database.

SQL Schema

This is the SQL Schema for the Internet Baseball Database.

```
BEGIN;
CREATE TABLE "idb_player" (
        "id" serial NOT NULL PRIMARY KEY,
        "name" varchar(30) NOT NULL,
        "number" integer NOT NULL,
        "position" varchar(3) NOT NULL,
        "bats" varchar(1) NOT NULL,
        "throws" varchar(1) NOT NULL,
        "height" integer NOT NULL,
        "weight" integer NOT NULL,
        "school" varchar(50) NOT NULL,
        "social" varchar(30) NOT NULL
)
;
CREATE TABLE "idb_team" (
        "id" serial NOT NULL PRIMARY KEY,
        "name" varchar(30) NOT NULL,
        "abbr" varchar(3) NOT NULL,
        "city" varchar(30) NOT NULL,
        "state" varchar(2) NOT NULL,
        "park" varchar(30) NOT NULL,
        "div" varchar(10) NOT NULL,
        "mgr" varchar(30) NOT NULL,
        "social" varchar(30) NOT NULL
)
;
CREATE TABLE "idb_year" (
        "id" serial NOT NULL PRIMARY KEY,
        "year" varchar(4) NOT NULL,
        "champion" varchar(30) NOT NULL,
        "AL_MVP" varchar(30) NOT NULL,
        "NL_MVP" varchar(30) NOT NULL,
        "NL_CY" varchar(30) NOT NULL,
        "AL_CY" varchar(30) NOT NULL
)
;
CREATE TABLE "idb_team_year" (
        "id" serial NOT NULL PRIMARY KEY,
```

```
	"team_id" integer NOT NULL REFERENCES "idb_team" ("id") DEFERRABLE INITIALLY
DEFERRED,
	"year_id" integer NOT NULL REFERENCES "idb_year" ("id") DEFERRABLE INITIALLY
DEFERRED,
	"wins" integer NOT NULL,
	"losses" integer NOT NULL,
	"standing" integer NOT NULL,
	"playoffs" varchar(20) NOT NULL,
	"attend" integer NOT NULL,
	"payroll" integer NOT NULL
)
;
CREATE TABLE "idb_player_year" (
	"id" serial NOT NULL PRIMARY KEY,
	"player_id" integer NOT NULL REFERENCES "idb_player" ("id") DEFERRABLE
INITIALLY DEFERRED,
	"team_year_id" integer NOT NULL REFERENCES "idb_team_year" ("id") DEFERRABLE
INITIALLY DEFERRED,
	"year_id" integer NOT NULL REFERENCES "idb_year" ("id") DEFERRABLE INITIALLY
DEFERRED,
	"games" integer NOT NULL,
	"kind" varchar(10) NOT NULL,
	"pa" integer,
	"avg" double precision,
	"obp" double precision,
	"slg" double precision,
	"hr" integer,
	"rbi" integer,
	"w" integer,
	"l" integer,
	"era" double precision,
	"gs" integer,
	"s" integer,
	"ip" integer,
	"whip" double precision
)
;
CREATE TABLE "idb_player_image" (
	"id" serial NOT NULL PRIMARY KEY,
	"player_id" integer NOT NULL REFERENCES "idb_player" ("id") DEFERRABLE
INITIALLY DEFERRED,
	"image" varchar(200) NOT NULL,
	"kind" varchar(10)
```

```
)
;
CREATE TABLE "idb_team_image" (
        "id" serial NOT NULL PRIMARY KEY,
        "team_id" integer NOT NULL REFERENCES "idb_team" ("id") DEFERRABLE INITIALLY
DEFERRED,
        "image" varchar(200) NOT NULL,
        "kind" varchar(10)
)
;
CREATE TABLE "idb_year_image" (
        "id" serial NOT NULL PRIMARY KEY,
        "year_id" integer NOT NULL REFERENCES "idb_year" ("id") DEFERRABLE INITIALLY
DEFERRED,
        "image" varchar(200) NOT NULL,
        "kind" varchar(10)
)
;
CREATE INDEX "idb_team_year_team_id" ON "idb_team_year" ("team_id");
CREATE INDEX "idb_team_year_year_id" ON "idb_team_year" ("year_id");
CREATE INDEX "idb_player_year_player_id" ON "idb_player_year" ("player_id");
CREATE INDEX "idb_player_year_team_year_id" ON "idb_player_year" ("team_year_id");
CREATE INDEX "idb_player_year_year_id" ON "idb_player_year" ("year_id");
CREATE INDEX "idb_player_image_player_id" ON "idb_player_image" ("player_id");
CREATE INDEX "idb_team_image_team_id" ON "idb_team_image" ("team_id");
CREATE INDEX "idb_year_image_year_id" ON "idb_year_image" ("year_id");

COMMIT;
```

Search

Django offers a search package, Haystack, for retrieving database models from simple word search queries. The general approach taken by Haystack is to first build a text document for each model entity which contains the values of the model's fields which will be searched. Upon a search request, the implementation returns all models which correspond to the text documents the string or strings are found in. While an efficient implementation, this approach also adds a significant amount of transparency and requires installing third party indexing and search libraries such as Elasticsearch or Whoosh. To minimize the transparency of our search feature, we elected to implement our own simple search engine which interacts in an SQL-like way with our models rather than with text documents generated from the models. To do this, we make use of the filter method passing to it, as an argument, a query on all fields we would like to search on. A simple example of the implementation looks like this:

Our Model:

```
class Player(models.Model):
   name = models.CharField(max_length=30)
   school = models.CharField(max_length=50)
   social = models.CharField(max_length=30)
```

And our query:

```
Player.objects.all().filter( Q(name__icontains=q) |
                             Q(school__icontains=q) |
                             Q(social__icontains=q) )
```

Where `q` is the query string.

Translated to SQL this call is simply:

```
SELECT * FROM PLAYERS
WHERE   name    LIKE   %q%
    OR    school  LIKE   %q%
    OR    social    LIKE   %q%
```

The result of the filter method call is a QuerySet, or list of all matching models.This implementation allows us to easily search for matching string patterns in the models we choose on the fields we choose.

Now let us consider the case where the query string is multiple words, “cardinals miguel cabrera”, for example. Using the entire string as the query will return no matches simply because no field will contain an exact match for “cardinals miguel cabrera”. Therefore we manipulate the search string by slicing it up and querying for all permutations and ‘sub-permutations’. By querying for all possible strings built from 1, 2 or 3 letter combinations of the words in the original query string, we can effectively implement an AND / OR search and get the team, St. Louis Cardinals, and the player, Miguel Cabrera, in our search results.

Our list of results is built by querying for each permutation or ‘subpermutation’ which ultimately can results in a list containing duplicates. Consider the same search string as above, “cardinals miguel cabrera.” From our subpermutations we can obtain “miguel”, “cabrera” and “miguel cabrera.” Each of these strings will return the instance of the player model with the name “Miguel Cabrera,” possibly adding 3 instances to our list. To avoid this, we simply check, before adding a result to our list, if a matching instance is already in the results list.

While this implementation is effective for retrieving multiple results without duplicates, it is quite slow and unsophisticated. For a typical search, up to 3 strings, this method is practical due to it’s simplicity and easy implementation. However, as the number of search strings increases the order of the algorithm increases exponentially. Obviously, a large part of this increase is due to the number of permutations which must each be processed as an individual query. Another

factor is duplication filtering. For each match that is found, we must loop through our existing results list checking for duplicates before adding the result. Because the number of results returned will generally increase with the number of words in the initial search, this causes a substantial increase in the total time for the search.

Ultimately, for the scope of this project, we chose our implementation not based on it's performance but on it's ease of implementation.

Client-side API Page

The page at /client uses Javascript to call the IBDB API many times. It uses the data to calculate a new metric for measuring the efficiency of Major League Baseball teams in terms of wins, payroll and playoff appearances.

Financial Playoff Efficiency Rating (FPER) is equal to the Payroll, divided by wins squared, and one more divided by playoff appearances plus one. The result is a single number that is used to evaluate how well a team spends its money with the ultimate goal of winning games, making the playoffs, and not overspending. A lower number is better.

The API calls made are a GET on /teams/id for each team to get team name and other basic information, and then a GET on /teams/id/years to get data on wins, payroll, and playoff appearances for each year in the database. The page then calculates FPER over the period of the last ten years for each team.

Interesting SQL Queries

The “Interesting SQL Queries” page at /sql shows a collection of interesting SQL queries that were made of the IBDB database. Among them were queries that showed:

- Players who have had historically great seasons
- World Series champions
- End-of-year Award winners
- Average MLB payroll by year and by team
- The average number of home runs per season, which reveals an interesting trend

The queries and results were gathered using the psql client for Linux.