ROYAL FLUSH
AMAR LAKHIAN
BRIAN TETZLAFF
JUAN REYES
LAWRENCE CHOI
TRAVIS CARPENTER

NBA DB

COMPREHENSIVE DATABASE WITH INFORMATION ON NBA TEAMS, PLAYERS, AND SEASONS

Contents

1. Introduction

NBA DB is a project that involved designing and building a database for the National Basketball Association. It will consist of information on NBA players, NBA teams, and NBA seasons. Figure 1 on the following page contains a UML class diagram that illustrates some of the relationships between the three main classes (corresponding to the three types of web pages). We plan to have pages for 10 players, 10 teams, and 10 seasons. Each player and team will have a set of statistics that go back 10 years/seasons to meet the requirements for this project.

1.1 Relationships and UML Class Diagram

The Team class will contain data for a particular NBA team such as *name*, *location*, etc. Note that it also contains an array called *seasons* of type TeamSeason. Each TeamSeason object will contain data such as win-loss records corresponding to a particular season for a particular team. To identify a certain season, there is an attribute for the *year* of type Season, which will establish a link between Team and Season classes.

The Player class contains data for NBA players such as *name, position*, etc. It also contains an array called *seasons* of type PlayerSeason. Each PlayerSeason object basically contains player stats for a particular season. It also has an attribute called *year* of type Season that identifies which season we are referring to; this will link Player to the Season class. Additionally, the PlayerSeason object has an attribute of type Team called *team* to identify which team a player was on for that season (after all, NBA players to get traded or sign with different teams as free agents), which will link to the Team class.

Figure 1: UML Diagram for NBA database

Finally, the Season class contains data for an NBA season such as the *year*, etc. In addition, it has an attribute called *champion* of type Team, which indicates the NBA champion for that particular season. There are also two attributes for the 1st Team All-NBA and 1st Team All-Defense,

called *all_nba* and *all_def* respectively. For those that aren't familiar with the NBA, the 1st Team All-NBA is a list of the 5 best players at each position for a particular year. Similarly, the 1st Team All-Defense is a list of the 5 best defensive players at each position for a particular year. So, *all_nba* and *all_def* will both be arrays of type Player that will hold the 1st Team All-NBA and 1st Team All-Defense respectively. Now we have established the link from Season to Team and from Season to Player.

1.2 Access

The User will have read-only access to all the data laid out in the previous section. Only the admins, however, will have write access. We do not want users to manipulate our database.

2. Design

2.1 RESTful API

The API HOST is http://nbaidb.pythonanywhere.com.

There are three collections: Teams, Players, and Seasons. These collections correspond to the three main types of web pages we will have. Currently, the API only defines GET functionality, so that users have read-only access to our data. We do not allow non-admins to have write access to our data.

For Teams you can list the entire collection (/teams), or you can list individual teams by passing an *id* parameter (/teams/{id}). Within each team, you can access the entire list of seasons for a team (/teams/{id}/seasons), or you can list a particular season for an individual team (/teams/{id}/seasons/{id}) by passing the correct season's *id* parameter.

Similarly, for Players you can list the entire collection (/players), or you can list individual players by passing an *id* parameter (/players/{id}). Within each player, you can access the entire list

of seasons for a player (/players/{id}/seasons), or you can list a particular season for an individual player (/players/{id}/seasons/{id}) by passing the correct season's *id* parameter.

For Seasons, you can list the entire collection (/seasons) or access an individual season (/seasons/{id}) by passing the correct *id* parameter.

The aforementioned *id* parameters correspond precisely to the *id* attribute that is in each object of type Team, Player, Season, TeamSeason, and PlayerSeason (see Figure 1 for further clarification).

If a GET request if valid, we respond with HTTP code 200 (for OK). If it's invalid, we respond with code 404 (Resource not found).

2.1.1 API Development Tools

The API development tool we used was apiary.io. For the most part, the API I developed was based on the API Blueprint tutorial. I also used http://sendgrid.com/blog/quickly-prototype-apis-apiary/ to help with defining GET access to subdirectories (i.e. /team/{id}/season/{id}).

2.2 Django Models

The django models for the NBA database are designed to separate three things: players, teams, and years. These are the three main models, followed by two subclasses that are player years and team years.

The Team model is the first of the three main class models. The information that is contained in the Team model is not meant to be constantly changing. It contains the team name, location of the team, its coach, its general manager, its owner, a twitter link for the official organization, and a link to its logo. In the real world, these all change from time to time, but for our purposes there is no need to be changing these often. Each team has multiple corresponding team years, denoting a one-to-many relationship.

The second main class model is the Player class. This model is meant to associate with an individual Player in the NBA. It is designed to contain static information that will probably not change during the duration of this project. The model contains things like the player's name, position, where he went to school (college or high school), how many years he has played in the NBA, a link to his official twitter account, and a recent photo of him. Note the team that the individual currently plays for is not included in the Player model because players often change teams and as a result that information is included in the subclass. Each team has multiple years that it has been in the NBA, so the information for those years are held in Team Years and are a one-to-many relationship.

The Year model is the third main model class. It contains overall information for a specific NBA season that will never change since the seasons have already happened. The model mostly focuses on the end of the season, and not what happens in the middle. Hence the information included is the year the model pertains to, the list of players on the All-NBA 1st Team that is selected after the regular season, the players selected to the All-Defensive 1st Team, the NBA Finals MVP, the team that won the NBA Finals, the logo for the NBA Finals for the year, and a text recap of how the NBA Finals went. Note the All-NBA team, All-Defensive team, and Finals MVP should all link to Player models, and the champion field link to a Team model. In the current edition, however, there were problems with the ManyToMany field for the All-NBA and All-Defensive teams, so those are string fields for the time being.

Each Player has multiple subclasses called PlayerYear's that contain a player's statistics and information about a specific year. Along with the usual stats and information such as the team he played for, points per game, assists, steals, blocks, points per game, etc. this field also contains Boolean fields that denote if the player received any MVP or All-NBA awards that season.

Each Team also has multiple subclasses called TeamYear's that contain how a specific team performed in a specific year. For now, this model only contains which Team and Year this "season," if you will, pertains to, as well as their wins and losses and a text recap of how their playoffs went (if they went, otherwise it should be say something along the lines of "Did not make postseason").

3. Tests

3.1 Unit Tests of Django Models

The django unit tests for models.py are tests for creating objects using each of the model classes.

For testing player, it makes sure every entry for creating an object is stored as entered. Its second test asserts that years of experience for a player is defaulted to zero if not given. The last test is making sure the str() method works as expected for a Player.

For testing Team, the tests, again, make sure every entry for creating an object is stored as entered. It also tests the str() method just as the Player Test did.

For testing year, it's the same as the test for creating a Team or Player but one player object and one team object were created to test entering foreign keys into the creation of a Year.

Player Year test includes every main class object, so all 3 were created and then used to create a player year (aka. season). Then asserted that all stats and info entered upon creation are correct.

Team Year test, again, includes all three main class objects, so all three were created and then used in creation of a team year (season). The assertions test that the foreign keys were entered and stored correctly and all the input for the object is returned as entered.

4. Other

4.1 Static HTML

To make to static HTML pages that are served by Django on Pythonanywhere, one of the easiest ways to start is to follow the Django tutorial provided in Pythonanywhere.com (the exact link is “https://www.pythonanywhere.com/wiki/DjangoTutorial”). The Django tutorial is straightforward up to “Defining your urls” section. The tutorial instructs to fill in the first line of urls.py as “from django.conf.urls.defaults import patterns, include, url” which would result in an error for users with Django 1.6 or later versions because “django.conf.urls.defaults” has been removed in Django 1.6. The correct way to import would be “from django.conf.urls import patterns, include, url” for Django 1.6 users.

After finishing the tutorial, the user should have all the necessary ingredients for building basic html pages in his pythonanywhere.com directory. The user should now be able to access into “http://user-id.pythonanywhere.com/admin/” (“http://nbadb.pythonanywhere.com/admin/” in our case) and administer the Django-powered website. The first thing the administrator would want to do is adding class objects, which are defined in the models.py file, into the website. To do this, models.py file should be in the app directory along with views.py. Also, an admin.py file with information about the models.py class objects should be in the same directory as well.

/ > home > nbadb > nba > nba > testapp1 > admin.py

```
1  from django.contrib import admin
2  from nba.testapp1.models import Team, Player, Year
3
4  admin.site.register(Team)
5  admin.site.register(Player)
6  admin.site.register(Year)
```

Figure 2

By adding an admin.py file to the app directory as shown in Figure 2, the administrator can now add class objects to the website as shown below in Figure 3.

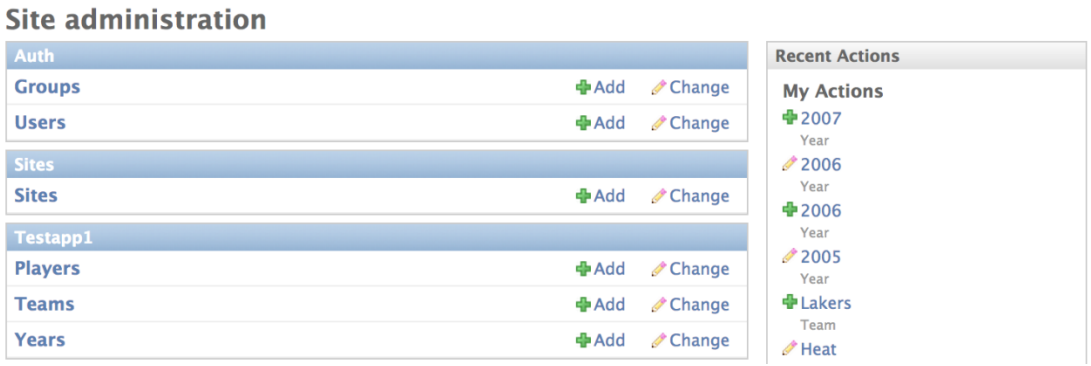

Figure 3

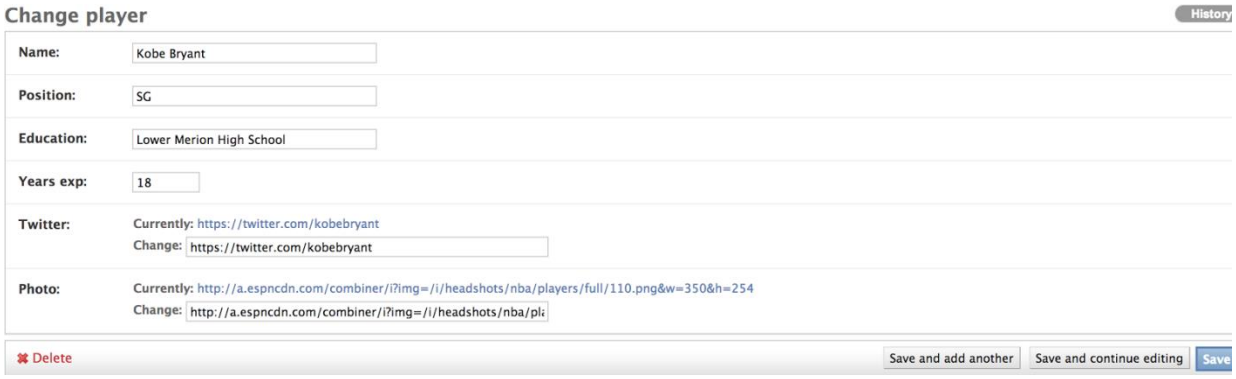

Figure 4

The classes that our model.py file define are “Players,” “Teams,” and “Year,” as shown in Figure 3. By clicking the “Add” and “Change” buttons, the administrator can manipulate information about the class objects (shown in Figure 4) in the website.

Once the necessary objects are added to the website, it is time to make the first html page of the website. The “home.html” was already made from the tutorial. In order to enable other html files to work and present them in the web, views.py and urls.py files have to be modified to handle each files. For the views.py file, “render_to_response” function was mainly used to handle the class objects (Documentations and more information about “render_to_response” function can be found

in “https://docs.djangoproject.com/en/dev/topics/http/shortcuts/”). Our group’s views.py file so far looks as follows:

```python
from django.shortcuts import render, render_to_response, RequestContext
#from django.template import RequestContext
from nba.testapp1.models import Player, Team, Year
# Creating my views

def PlayersAll(request):
    players = Player.objects.all().order_by('name')
    context = {'players': players}
    return render_to_response('playersall.html', context, context_instance=RequestContext(request))

def TeamsAll(request):
    teams = Team.objects.all().order_by('name')
    context = {'teams': teams}
    return render_to_response('teamsall.html', context, context_instance=RequestContext(request))

def YearsAll(request):
    years = Year.objects.all().order_by('year')
    context = {'years': years}
    return render_to_response('yearsall.html', context, context_instance=RequestContext(request))

def home(request):
    return render_to_response('home.html', locals(),
                              context_instance=RequestContext(request))
```

Figure 5

Apart from the request for “home.html,” all the requests in view.py returns a HttpResponse object, which makes it possible to display the appropriate html page such as “playersall.html,” which is displayed in Figure 6.

```
{% extends "base.html" %}
{% block content %}
    <h2>Players</h2>
    <div id='playerslist'>
        {% for player in players %}
        <h3><a href="/players/{{player.pk}}">{{ player.name }}</a></h3>
        <p><img src="{{player.photo}}"></p>
        <p>Position: {{ player.position }}</p>
        <p>Education: {{ player.education }}</p>
        <p>Experience: {{ player.years_exp }} years</p>
        {% endfor %}
    </div>
    <h2><a href="/teams/">Go to Teams</a></h2>
    <h2><a href="/years/">Go to Years</a></h2>
{% endblock %}
```

Figure 6

The basic layout is the same for all the other html files (teamsall.html and yearsall.html): They all display the class objects' information that the administrator added. At the bottom of the html block are the links to other html pages. The files also have links that present a more detailed view (using DetailView) of a certain object (for example, line 6 of "playersall.html").

```
11 urlpatterns = patterns('',
12         # This is going to be our home view.
13         # We'll uncomment it later
14     url(r'^$', 'nba.testapp1.views.home', name='home'),
15     #players, teams, and years view
16     url(r'^players/$', 'nba.testapp1.views.PlayersAll'),
17     url(r'^teams/$', 'nba.testapp1.views.TeamsAll'),
18     url(r'^years/$', 'nba.testapp1.views.YearsAll'),
19
20     url(r'^players/(?P<pk>\d+)/$', DetailView.as_view(
21                         model = Player,
22                         template_name="player.html")),
23     url(r'^teams/(?P<pk>\d+)/$', DetailView.as_view(
24                         model = Team,
25                         template_name="team.html")),
26     url(r'^years/(?P<pk>\d+)/$', DetailView.as_view(
27                         model = Year,
28                         template_name="year.html")),
29         # Uncomment the admin/doc line below to enable admin documentation:
30     # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),
31
32         # Uncomment the next line to enable the admin:
33     url(r'^admin/', include(admin.site.urls)),
34     #url(r'^testapp1/', include('nba.testapp1.urls')),
```

Figure 7

The urls.py file (from line 20 to 28), from Figure 7, shows how DetailView was implemented (a full documentation and explanation can be found in "https://docs.djangoproject.com/en/dev/ref/class-based-views/generic-display/").

4.2 Twitter Bootstrap

4.3 Data Collection

For each team, we gathered data on their name, city of origin, years since their original founding, prior names for the teams and their locations, years that they went to the playoffs since

the 1984 playoffs (the first year that the playoffs were fixed with sixteen teams going), years since the 1984 playoffs that the team won the playoffs if any, and a link to current roster data for the team. This was done so that every team would have multiple links to other types of pages once completed. First, each of the thirty teams would have a link to all of the player pages for players that are currently signed to the team. The team page would also have a link to the playoff pages for every year they went to playoffs. The data for the teams was gathered from many different sites on the webs, such as the homepage for the various NBA teams on the NBA website, wikipedia pages for the team, and the RealGM website for teams playoff history. The first two were chosen primarily for the fact that they had the majority of data needed for each team. The third site was chosen because it could be copied easily to a text editor, saved, and using a quickly made program, edited to be formatted into a easily usable format. The player data contains the player's name, position, previous schooling or NCAA data if available, years of experience, stats per game, teams the player has been on and for how long they were on that team, awards the players have won, and the years that the players went to playoffs. Using this data, the player pages would have multiple links to both playoff pages as well as other team pages. The playoff data was gathered from the land of basketball website, and contains the playoff bracket for that year and the finals MVP for that year. This was done so that each playoff page would have links to the teams that went to the playoffs for that year and a link to at least one player page.