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 and Purpose

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. The purpose of our particular project is to provide information on notable NBA players, the teams that they are on, and recent 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. It also contains attribute FinalsYears that indicate which year the team made it to the NBA Finals, which will link it to the Year class. The MVPs and FinalsMVPs attributes indicate the particular team's previous regular season and NBA Finals MVPs respectively. This establishes the relationship between team and player.

The Player class contains data for NBA players such as *name, position*, etc. It also contains an attribute called Current to identify the player's current team. This links the player to the Team class. There is also a section for awards, identified by attributes such as mvpYears, FinalsYears, allNBAYears, etc. which will contain a list of years that the player won the particular award. This creates the relationship between the Player class and Year class.

Finally, the Year class contains data for an NBA season (year). 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. This establishes the relationship between Year and Player.

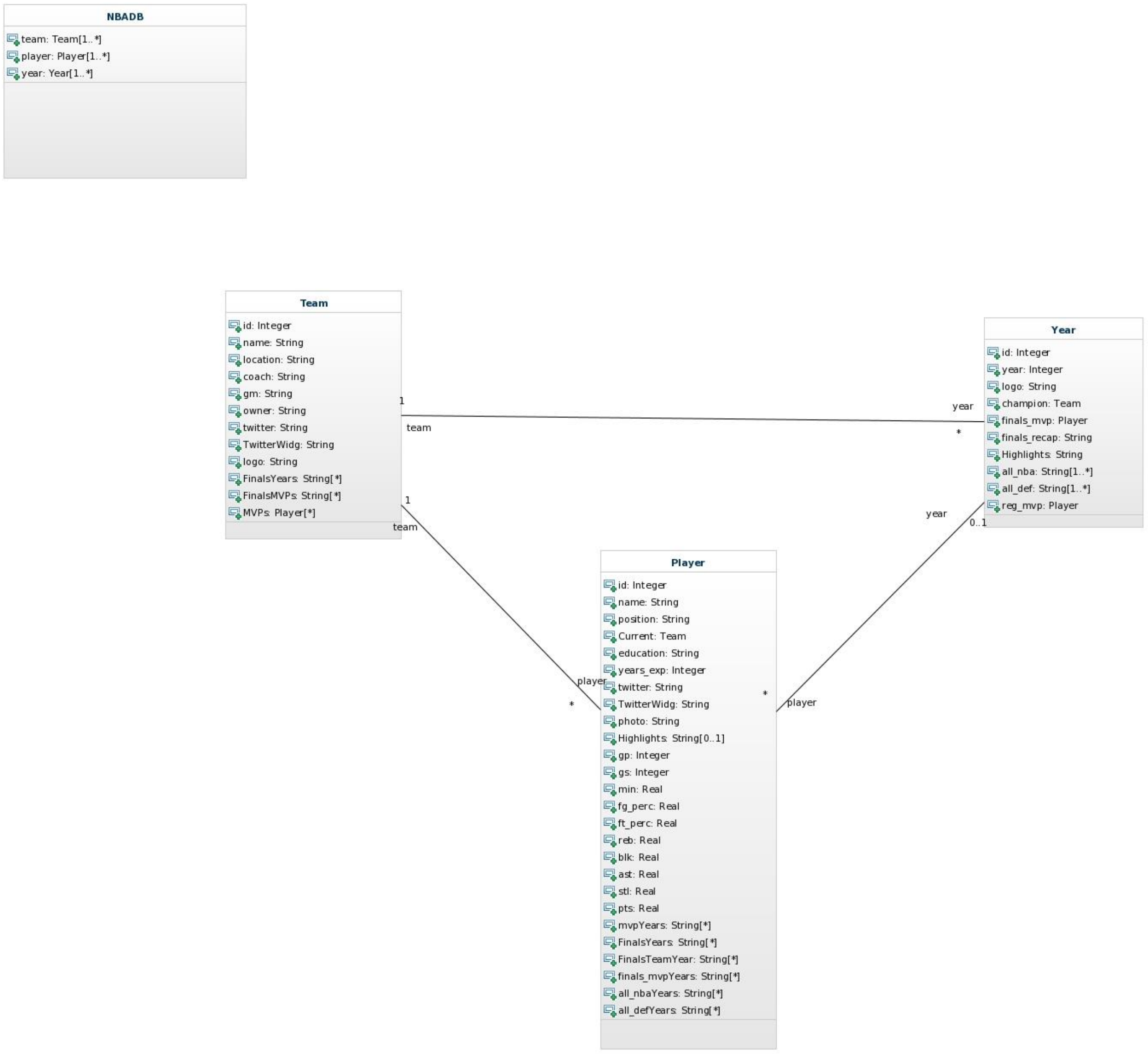


Figure 1: UML Diagram for NBA database

## 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 Django Models

The django models for the NBA database are designed to separate three things: players, teams, and 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, the recent years the team has been in the NBA Finals, recent MVPs, recent Finals MVPs, 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.

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, high school, or internationally), how many years he has played in the NBA, a link to his official twitter account, a recent photo of him, his nickname, YouTube highlights, a ForeignKey referencing his current team, and a list of statistics (awards, games played, points per game, assists per game, etc). The player's awards are listed by type of award and then a list of the years he received it.

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 (Foreign Key), the logo for the NBA Finals for the year, a text recap of how the NBA Finals went, YouTube highlights, and the MVP that is selected after the regular season has concluded.

## 2.2 RESTful API

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

There are three collections: Teams, Players, and Years. 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. Users do not have write access to our data.

For Teams you can list the entire collection (/api/teams), or you can list individual teams by passing an *id* parameter (/api/teams/{id}). Similarly, for Players you can list the entire collection (/api/players), or you can list individual players by passing an *id* parameter (/api/players/{id}). For NBA seasons, you can list the entire collection (/years) or access an individual season (/years/{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, and Year(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).

For more information on how to access our API, see the Apiary file on the public github repo.

### 2.2.1 API Development Tools

Apiary.io was used in developing the API blueprint based on the API Blueprint tutorial found on the website.

### 2.2.2 API Implementation

The API was implemented using the Django REST Framework. First, the tool was installed using “pip3.4 install –user djangorestframework markdown django-filter” in the Bash console. Also, 'rest_framework' was added to INSTALLED_APPS in the settings.py file in the project directory.

In order to adapt the Django models to the API, the first requirement is a serializer, which converts complex data, such as Django models, into native Python data types, such as JSON. After importing serializer from rest_framework, we defined 3 serializer classes: TeamSerializer, PlayerSerializer, YearSerializer, which maps the fields from the Django models to the serializer object.

Then in api.py, we created instances of the various serializer objects and returned them as a Response (either status code 200 for valid queries or 404 for resources not found). Api.py contains get methods for a list of all the Teams, Players, or Years. It also contains methods for accessing individual Teams, Players, or Years by their primary keys.

Finally, we added the API URLs to urls.py in order to render the view for Teams, Players, and Years.

## 2.3 Search Engine

The search engine was built on three main functions that were added to views.py: normalize_query, get_query, and search. The normalize_query and get_query functions generate a django.db.model.Q object, which uses the query string and search fields passed to it to return the appropriate objects. Additionally, it returns both the AND and OR results of a particular query. The search function in views.py calls the aforementioned get_query and normalize_query functions, and sorts the returned objects into the appropriate Team, Player, or Year categories. These objects are placed in a dictionary and passed to the /search.html web page which then displays the results.

The above search engine was adapted from Julien Phalip's tutorial (http://julienphalip.com/post/2825034077/adding-search-to-a-django-site-in-a-snap).

## 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. The second test asserts that the fields that are allowed to be omitted/defaulted when creating the object default to the correct (expected) value.

For testing Team, the tests, again, make sure every entry for creating an object is stored as expected. The second test, just as with player, omits fields that are allowed to be defaulted and asserts that the expected values are stored for those fields.

For testing Year, it's the same as the test for creating a Team or Player for both the first and the second test.

The third test for all three types of models tries to clash the unique fields of models. If the objects were created successfully with the clashes the tests would fail, asserting that the unique fields work as expected.

## 3.2 Unit Tests for API

The API can return either a list of all the objects whether it's Teams, Players, or Years, or it can return a list of individual objects accessed by id #.

There are three types of lists: TeamList, PlayerList, and YearList. For each of these three lists, there are three tests each. The first test checks the response from a valid request (i.e. http://nbadb.pythonanywhere.com/api/teams/) and checks the response code, and validates that the length of the list (i.e. the number of teams in the list) is correct. The second test checks that an invalid request returns status_code 404. The third test checks a randomly chosen index of the data and makes sure that it matches up with the correct team, player, or year.

There are also three types of "details": TeamDetail, PlayerDetail, and YearDetail. This returns a particular team, player, or year of type dict, accessed by a particular index (i.e. http://nbadb.pythonanywhere.com/api/teams/1/). There are three tests for each of these three details. The first two tests will check the status code for a valid and invalid request, respectively. The third test will check that the actual data contained in the dict for a certain team, player, or year is correct. It checks either the entire dict, or it checks a randomly selected key-value pair within the dict.

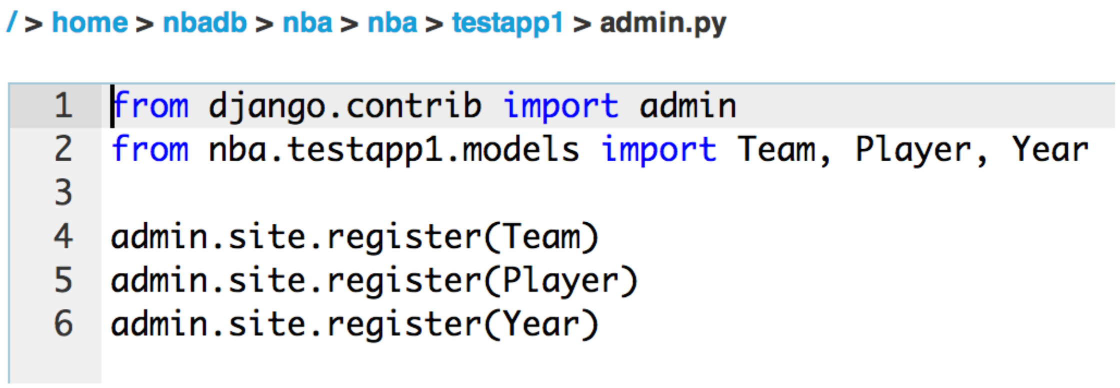
So in total, there are 6 classes for the API and 3 tests for each class, which gives a total of 18 unit tests for the API.

## 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.

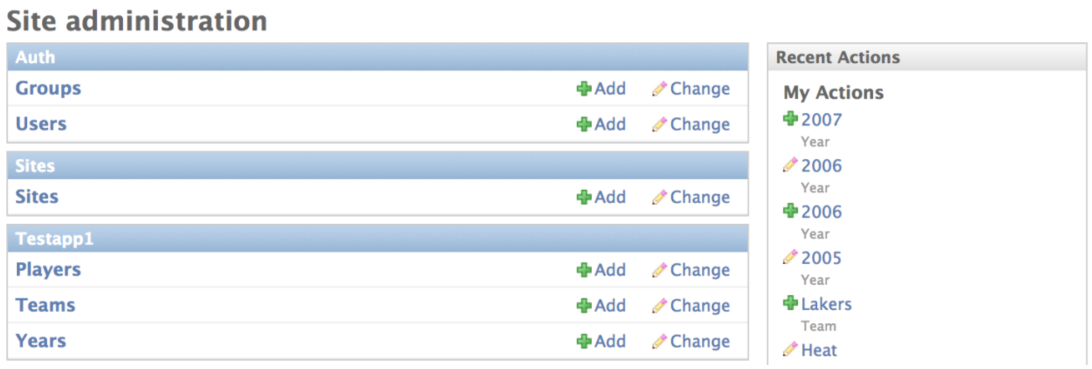


/ > home > nbadb > nba > nba > testapp1 > admin.py

```
from django.contrib import admin
from nba.testapp1.models import Team, Player, Year

admin.site.register(Team)
admin.site.register(Player)
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.



Site administration

| Auth | | |
|---|---|---|
| Groups | Add | Change |
| Users | Add | Change |

| Sites | | |
|---|---|---|
| Sites | Add | Change |

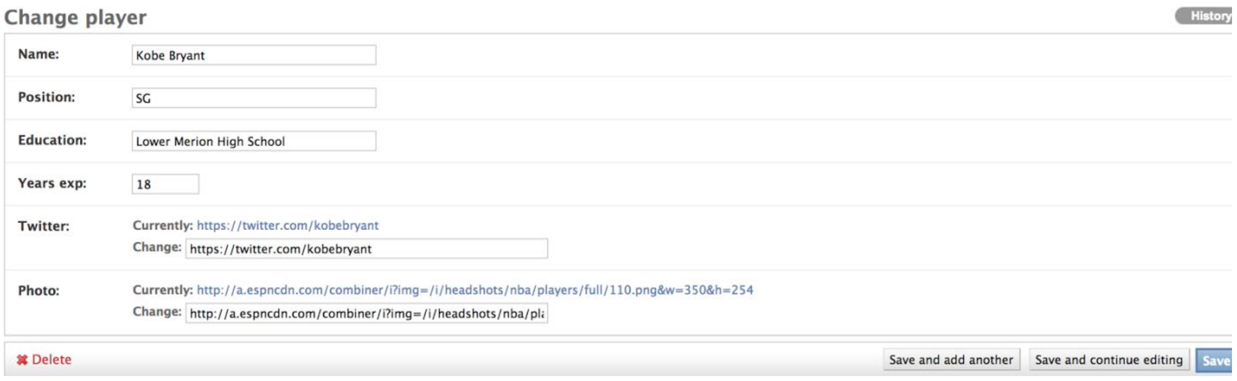
| Testapp1 | | |
|---|---|---|
| Players | Add | Change |
| Teams | Add | Change |
| Years | Add | Change |

Recent Actions

My Actions

2007
Year
2006
Year
2006
Year
2005
Year
Lakers
Team
Heat

Figure 3



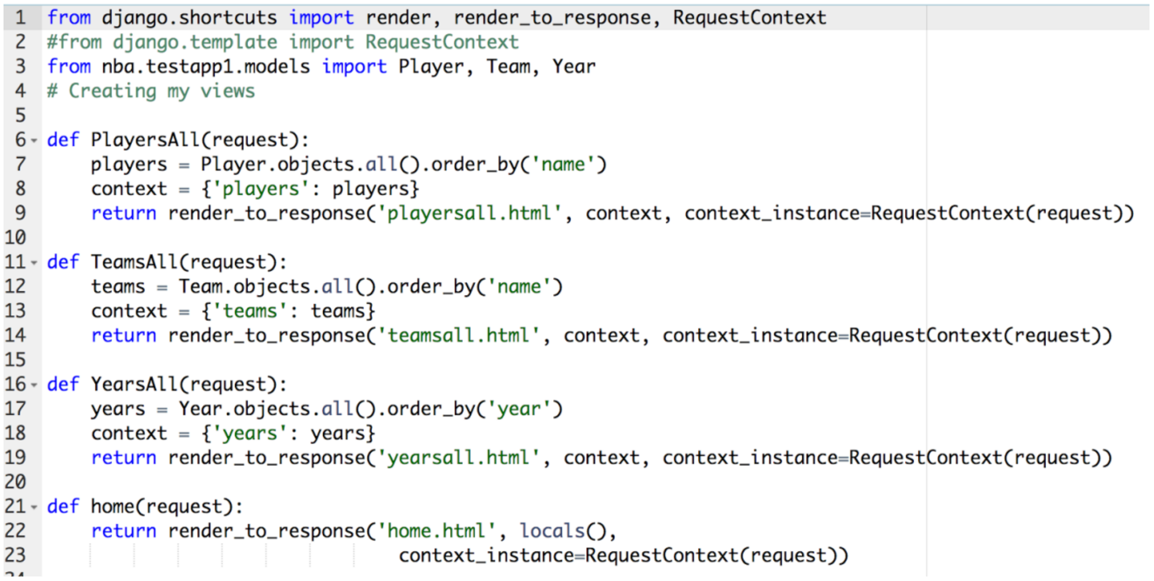
Change player

| | |
|---|---|
| Name: | Kobe Bryant |
| Position: | SG |
| Education: | Lower Merion High School |
| Years exp: | 18 |
| Twitter: | Currently: https://twitter.com/kobebryant<br>Change: https://twitter.com/kobebryant |
| Photo: | Currently: http://a.espncdn.com/combiner/i?img=/i/headshots/nba/players/full/110.png&w=350&h=254<br>Change: http://a.espncdn.com/combiner/i?img=/i/headshots/nba/pl |

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.

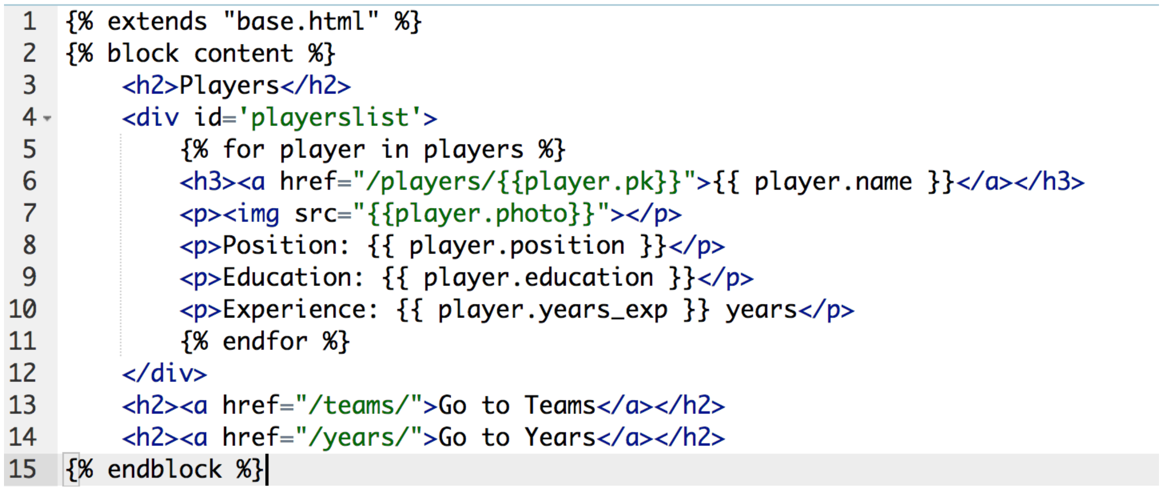
Once the necessary objects are added to the website, it is time to make the first html page of the website. The "home.html" was already made from the tutorial. In order to enable other html files to work and present them in the web, views.py and urls.py files have to be modified to handle each files. 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

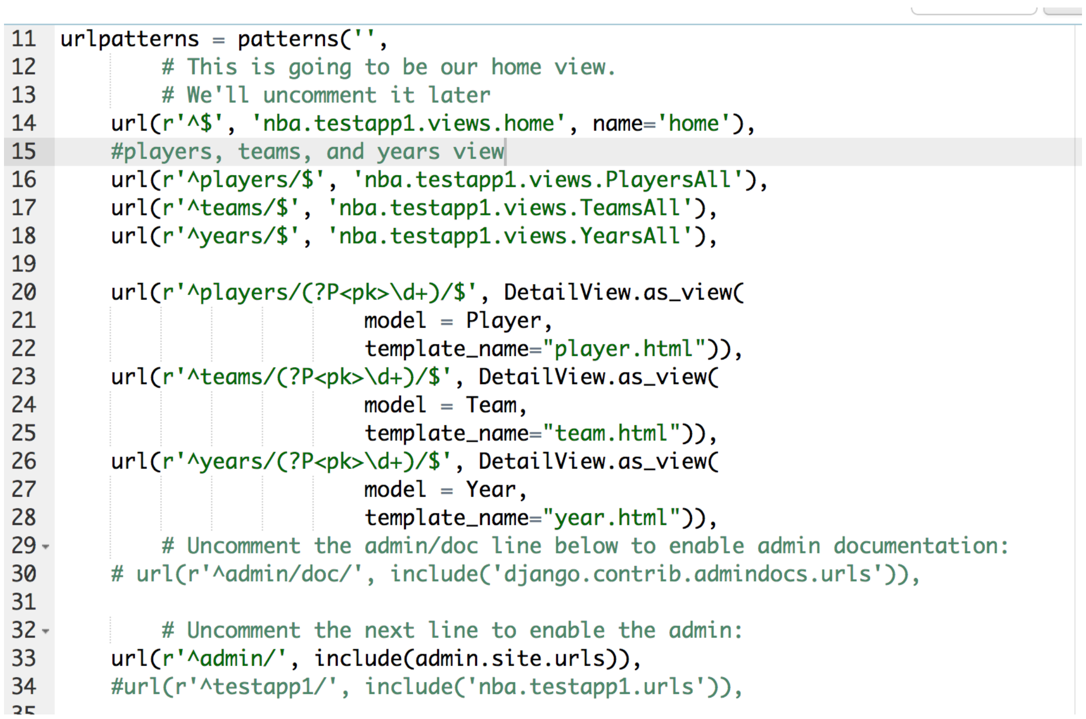
Apart from the request for “home.html,” all the requests in view.py returns a HttpResponse object, which makes it possible to display the appropriate html page such as “playersall.html,” which is displayed in Figure 6.



```html
{% 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”).



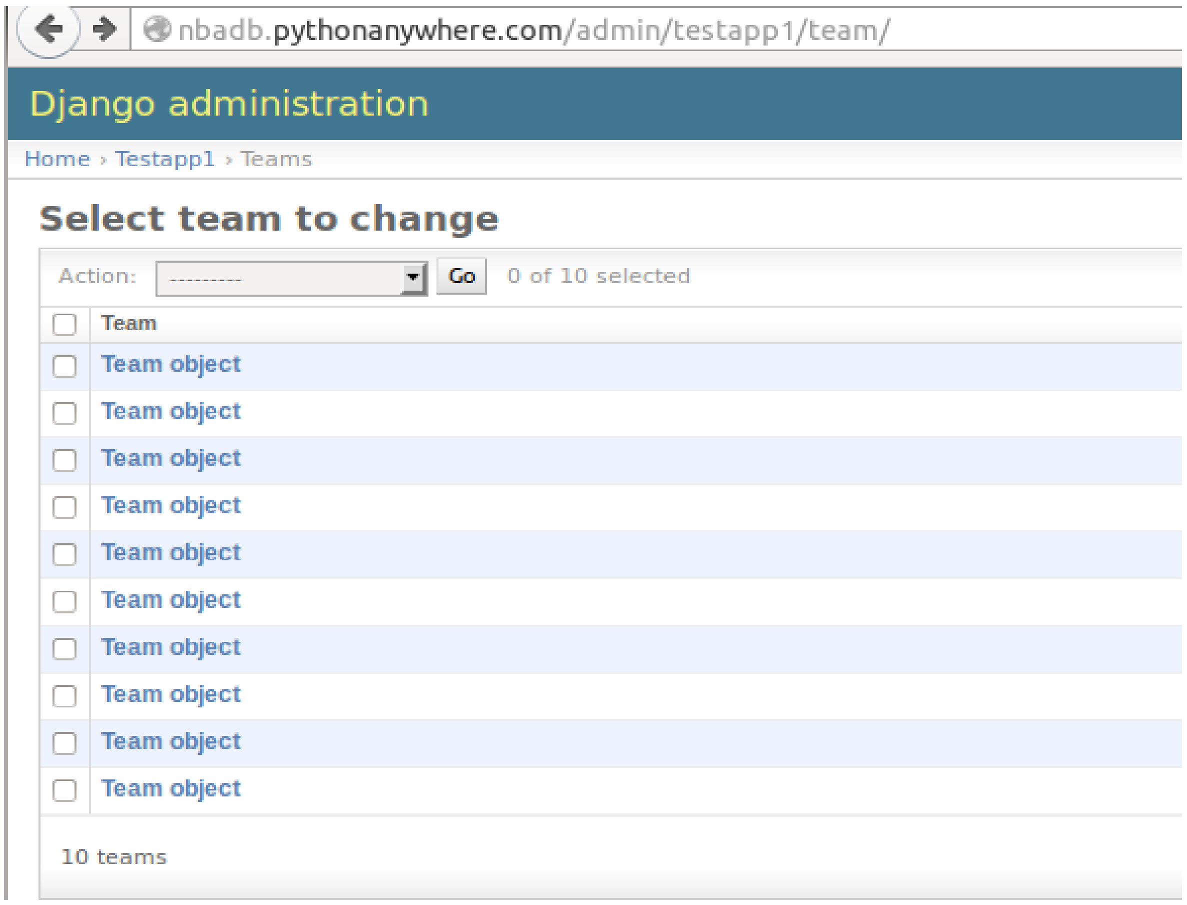
```
11 urlpatterns = patterns('',
12         # This is going to be our home view.
13         # We'll uncomment it later
14     url(r'^$', 'nba.testapp1.views.home', name='home'),
15     #players, teams, and years view
16     url(r'^players/$', 'nba.testapp1.views.PlayersAll'),
17     url(r'^teams/$', 'nba.testapp1.views.TeamsAll'),
18     url(r'^years/$', '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 Dynamic HTML

To make the HTML pages dynamic, it is important to make sure all the classes and attributes in the models.py file are up-to-date in the admin page (http://nbadb.pythonanywhere.com/admin/ in our group’s case).


nbadb.pythonanywhere.com/admin/testapp1/team/

Django administration

Home › Testapp1 › Teams

Select team to change

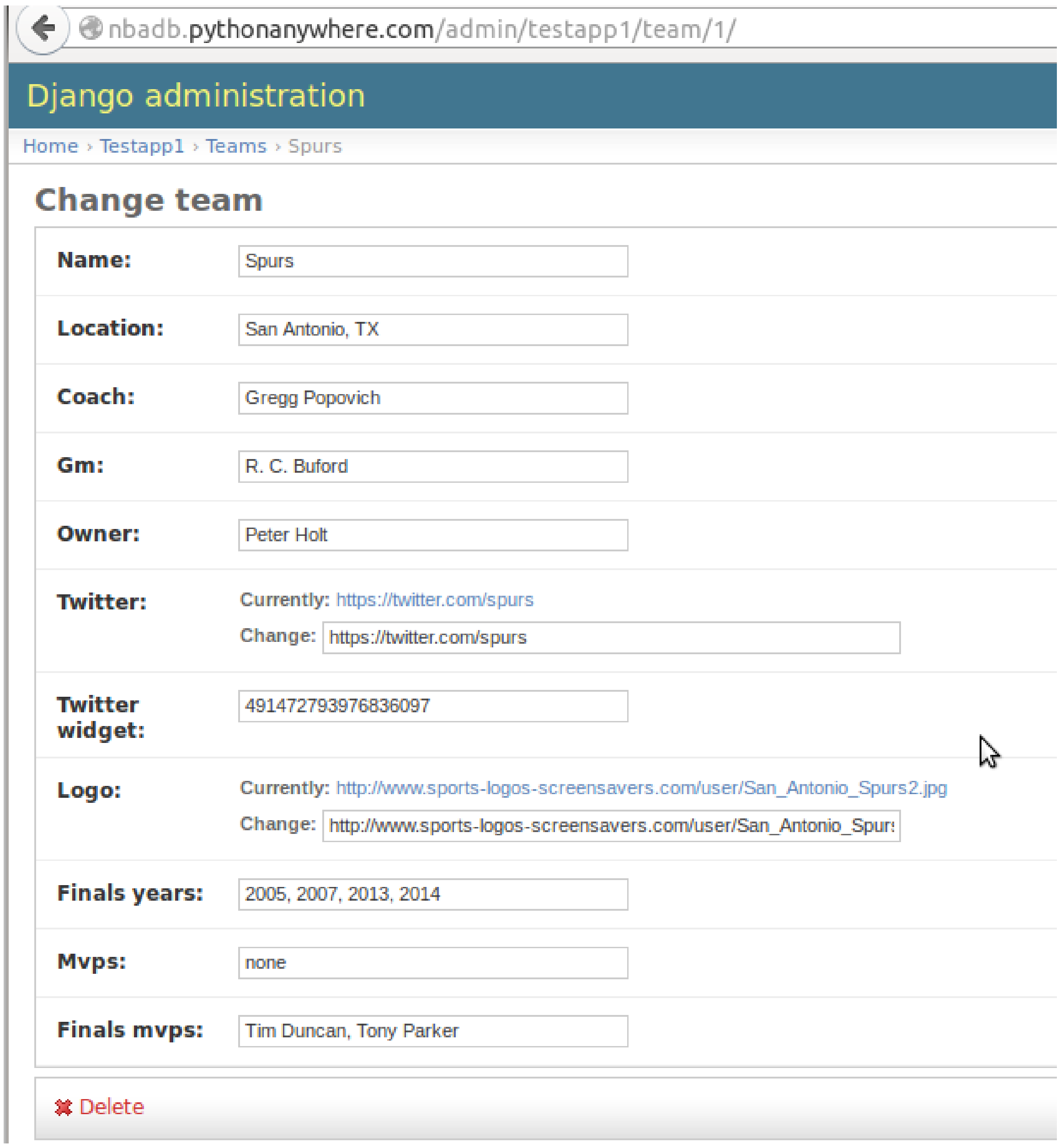
Action: --------- Go 0 of 10 selected

| Team |
|---|
| Team object |
| Team object |
| Team object |
| Team object |
| Team object |
| Team object |
| Team object |
| Team object |
| Team object |
| Team object |

10 teams

Figure 8

The figure above shows an example of 10 class objects (10 “Teams” in our case) added to the admin page.



nbadb.pythonanywhere.com/admin/testapp1/team/1/

Django administration

Home › Testapp1 › Teams › Spurs

Change team

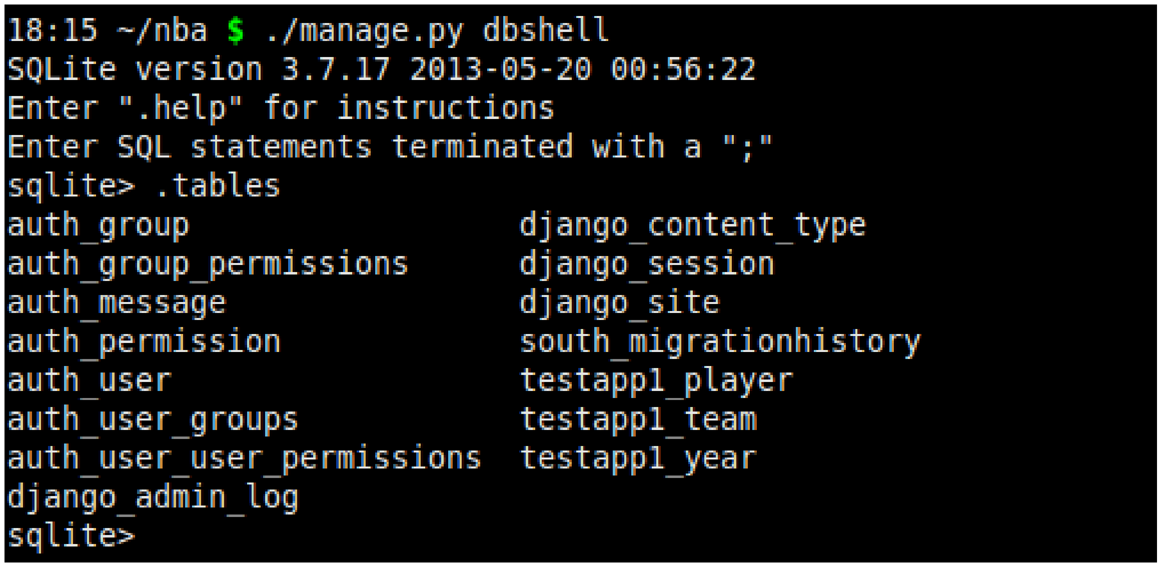
| | |
|---|---|
| Name: | Spurs |
| Location: | San Antonio, TX |
| Coach: | Gregg Popovich |
| Gm: | R. C. Buford |
| Owner: | Peter Holt |
| Twitter: | Currently: https://twitter.com/spurs<br>Change: https://twitter.com/spurs |
| Twitter widget: | 491472793976836097 |
| Logo: | Currently: http://www.sports-logos-screensavers.com/user/San_Antonio_Spurs2.jpg<br>Change: http://www.sports-logos-screensavers.com/user/San_Antonio_Spur: |
| Finals years: | 2005, 2007, 2013, 2014 |
| Mvps: | none |
| Finals mvps: | Tim Duncan, Tony Parker |

Figure 9

All the objects should have their attributes filled out as shown in Figure 9. This particular figure shows how one of the "Team" objects was filled out.

After updating the admin page, the objects in the models.py it is essential to match all the data to the SQLite database in Pythonanywhere. This can be done using migrations in "south" that Pythonanywhere provides. It is important to add "south" to the "Installed Apps" in settings.py file before actually using "south." The details and syntax for using "south" are explained in

http://south.readthedocs.org/en/latest/tutorial/part1.html.

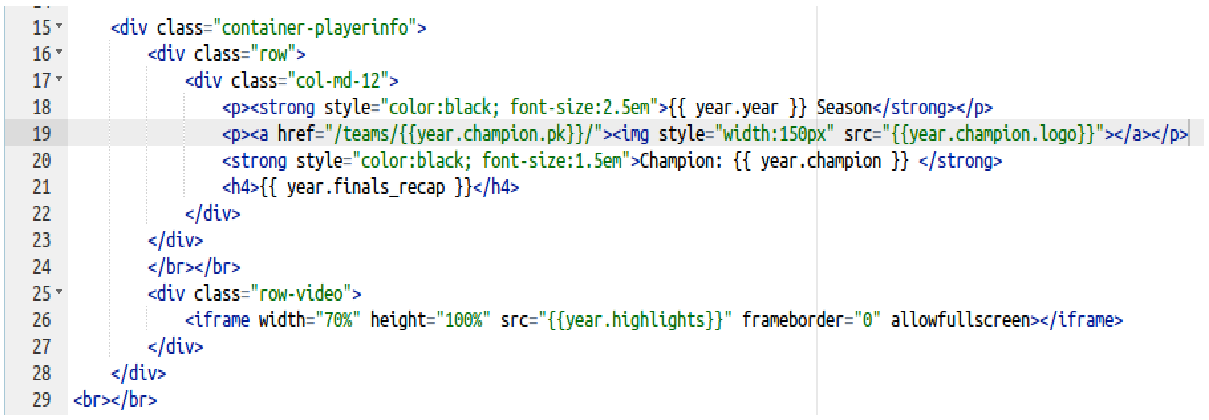


```
18:15 ~/nba $ ./manage.py dbshell
SQLite version 3.7.17 2013-05-20 00:56:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
auth_group                  django_content_type
auth_group_permissions      django_session
auth_message                django_site
auth_permission             south_migrationhistory
auth_user                   testapp1_player
auth_user_groups            testapp1_team
auth_user_user_permissions  testapp1_year
django_admin_log
sqlite>
```

Figure 10

Figure 10 shows that three tables (testapp1_player, testapp1_team, and testapp1_year) were added successfully to the Pythonanywhere SQLite database.

Now that all the tables in Pythonanywhere SQLite database are up-to-date, it is possible to make an html page that can be generated dynamically by retrieving data from the database.



```
15      <div class="container-playerinfo">
16          <div class="row">
17              <div class="col-md-12">
18                  <p><strong style="color:black; font-size:2.5em">{{ year.year }} Season</strong></p>
19                  <p><a href="/teams/{{year.champion.pk}}/"><img style="width:150px" src="{{year.champion.logo}}"></a></p>
20                  <strong style="color:black; font-size:1.5em">Champion: {{ year.champion }} </strong>
21                  <h4>{{ year.finals_recap }}</h4>
22              </div>
23          </div>
24          </br></br>
25          <div class="row-video">
26              <iframe width="70%" height="100%" src="{{year.highlights}}" frameborder="0" allowfullscreen></iframe>
27          </div>
28      </div>
29  <br></br>
```

Figure 11

Figure 11 is part of the "year.html" file. Line 19 shows how the page retrieves the "champion" for a particular year. "Champion" is a "Team" foreign key defined inside the "Year" class. Therefore, {{year.champion.pk}} on line 19 results in the id (or primary key) of a particular team. {{year.champion.logo}} would yield the logo of a particular team. Also, on line 21 and line 26, {{year.finals_recap}} and {{year_highlights}} would result in a text of a particular year's recap and a link to a video of a particular year's highlight respectively. Team.html and player.html pulls data using the same method. Thus, a single html page has the capability of displaying ten different (11 in case of players) pages. Clearly, these html pages are generated dynamically. If the pages were static, each object should have its own html page (year2005.html, year2006.html, etc…).

## 4.3 Twitter Bootstrap

Twitter Bootstrap is a powerful framework that can be used to create elegant, responsive websites that are also supported on mobile devices in a simple way. There are many templates available for download at getbootstrap.com. HTML, Javascript, CSS, and other files are included in the templates that can be easily changed to suit whatever style is desired, using the many neat features that are available in bootstrap. Features that are included are navigation bars, available in both static and fixed positions, jumbotrons, buttons, tables, carousels, drop-down menus, and sidebars, among others. This project included a fixed navigation bar with links to static HTML pages, a cover box that acts as an introduction to the site, and marketing content that link to teams, players, and years pages. A background photo can also be included to give the site a personal touch.

The HTML files are used to build the features included, using the many classes available. Unique to Twitter Bootstrap is the 12 grid system that it uses. If something is going to span the whole width of the page, then it will use all 12 grids to do it. Since this project uses three marketing columns, the lines

```
<div class="row">
    <div class="col-lg-4">
```

will be used. The 'row' class will reserve the 12 grids that it will use and each of the three marketing columns will use the class 'col-lg-4' because they will be split in three equal columns that are four grids wide.

The CSS files are used for setting purposes, such as what size, color, font the lettering will be are all set in the CSS files. Each feature used can be set in these files. For example, the lines

```
body{
     background: url('http://us.cdn281.fansshare.com/photos/kevindurant/kevin-durant-pictures-
    nba-wallpaper-hd-127610132.jpg') no-repeat center center fixed;
    -webkit-background-size: cover;
    -moz-background-size: cover;
    -o-background-size: cover;
    background-size: cover;
    margin-top: 5%;
}
```

sets what the background photo will be, as well as what its dimension will be.

To link Twitter Bootstrap to pythonanywhere, the files are stored in the static directory:

/ > home > nbadb > nba > static

When they are stored here, the lines

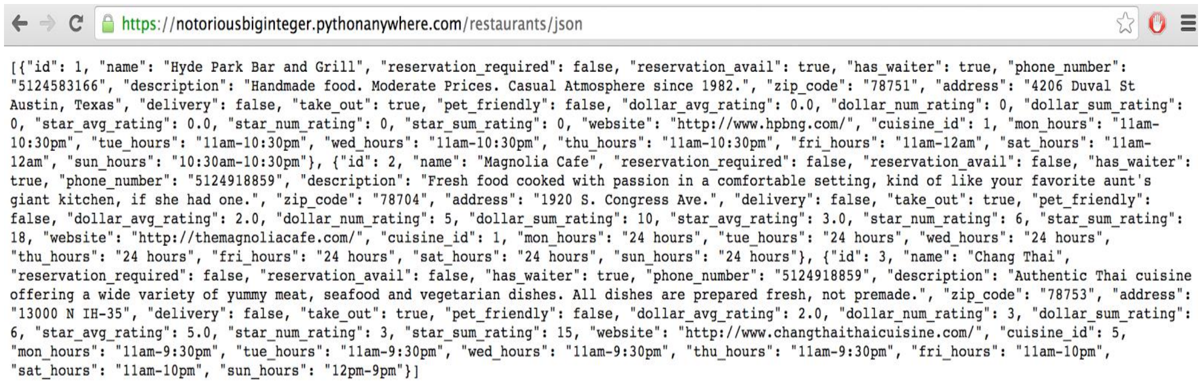
```
<!-- Bootstrap core CSS -->
    <link href="{{STATIC_URL}}/css/bootstrap.min.css" rel="stylesheet">
<!-- Custom styles for this template -->
    <link href="{{STATIC_URL}}/css/cover.css" rel="stylesheet">
```

are included in the index.html file so that it will have access to them.

## 4.4 Using Another Group's API

In order to implement another team's API (in our group's case, group "Syscall of the Wild"), we first had to check how their API was constructed. The API for "Syscall of the Wild" is available in the link: http://docs.kravester.apiary.io/. At a glance, it is easy to spot that the API's are in json format. Therefore, using jQuery's "getjson" function is one of the simplest ways to take these json formatted results to our webpage and implement (Detailed documentaion for "getjson" function can be found in http://api.jquery.com/jquery.getjson/).

One difficulty that our team faced was that pythonanywhere.com seems to block HTTP access on certain sites for free accounts. For example, our website was able to access a random json formatted url (http://graph.facebook.com/zombies), but was not able to access group "Syscall of the Wild's" json formatted url (https://notoriousbiginteger.pythonanywhere.com/restaurants/json is one example).

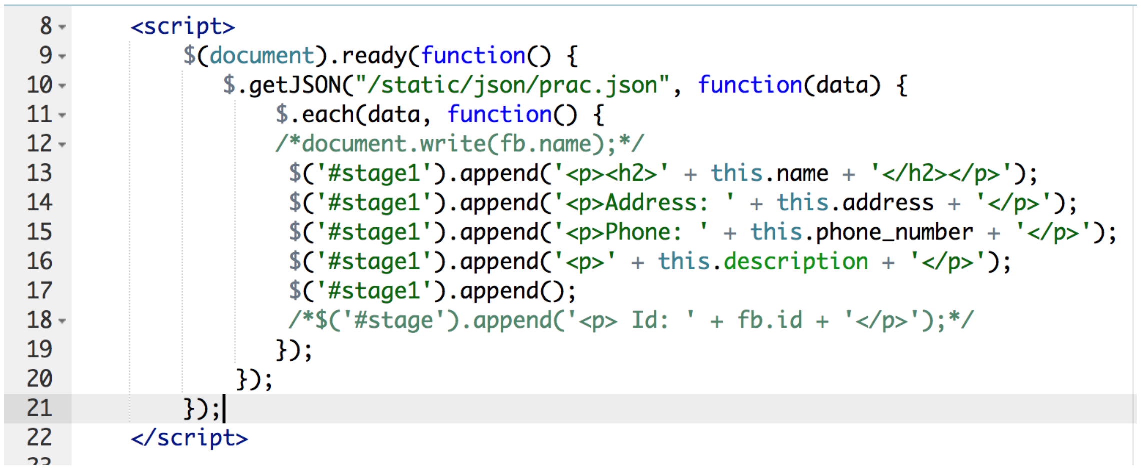

https://notoriousbiginteger.pythonanywhere.com/restaurants/json

```
[{"id": 1, "name": "Hyde Park Bar and Grill", "reservation_required": false, "reservation_avail": true, "has_waiter": true, "phone_number":
"5124583166", "description": "Handmade food. Moderate Prices. Casual Atmosphere since 1982.", "zip_code": "78751", "address": "4206 Duval St
Austin, Texas", "delivery": false, "take_out": true, "pet_friendly": false, "dollar_avg_rating": 0.0, "dollar_num_rating": 0, "dollar_sum_rating":
0, "star_avg_rating": 0.0, "star_num_rating": 0, "star_sum_rating": 0, "website": "http://www.hpbng.com/", "cuisine_id": 1, "mon_hours": "11am-
10:30pm", "tue_hours": "11am-10:30pm", "wed_hours": "11am-10:30pm", "thu_hours": "11am-10:30pm", "fri_hours": "11am-12am", "sat_hours": "11am-
12am", "sun_hours": "10:30am-10:30pm"}, {"id": 2, "name": "Magnolia Cafe", "reservation_required": false, "reservation_avail": false, "has_waiter":
true, "phone_number": "5124918859", "description": "Fresh food cooked with passion in a comfortable setting, kind of like your favorite aunt's
giant kitchen, if she had one.", "zip_code": "78704", "address": "1920 S. Congress Ave.", "delivery": false, "take_out": true, "pet_friendly":
false, "dollar_avg_rating": 2.0, "dollar_num_rating": 5, "dollar_sum_rating": 10, "star_avg_rating": 3.0, "star_num_rating": 6, "star_sum_rating":
18, "website": "http://themagnoliacafe.com/", "cuisine_id": 1, "mon_hours": "24 hours", "tue_hours": "24 hours", "wed_hours": "24 hours",
"thu_hours": "24 hours", "fri_hours": "24 hours", "sat_hours": "24 hours", "sun_hours": "24 hours"}, {"id": 3, "name": "Chang Thai",
"reservation_required": false, "reservation_avail": false, "has_waiter": true, "phone_number": "5124918859", "description": "Authentic Thai cuisine
offering a wide variety of yummy meat, seafood and vegetarian dishes. All dishes are prepared fresh, not premade.", "zip_code": "78753", "address":
"13000 N IH-35", "delivery": false, "take_out": true, "pet_friendly": false, "dollar_avg_rating": 2.0, "dollar_num_rating": 3, "dollar_sum_rating":
6, "star_avg_rating": 5.0, "star_num_rating": 3, "star_sum_rating": 15, "website": "http://www.changthaithaicuisine.com/", "cuisine_id": 5,
"mon_hours": "11am-9:30pm", "tue_hours": "11am-9:30pm", "wed_hours": "11am-9:30pm", "thu_hours": "11am-9:30pm", "fri_hours": "11am-10pm",
"sat_hours": "11am-10pm", "sun_hours": "12pm-9pm"}]
```

Figure 12

The image above shows the format of the https://notoriousbiginteger.pythonanywhere.com/restaurants/json that pythonanywhere.com denies our access into. Since the url is in proper json format, we should have been able to obtain the data with no problem.

As a short term solution, we stored the json formatted url into our pythonanywhere database.



```
    <script>
        $(document).ready(function() {
           $.getJSON("/static/json/prac.json", function(data) {
               $.each(data, function() {
               /*document.write(fb.name);*/
                $('#stage1').append('<p><h2>' + this.name + '</h2></p>');
                $('#stage1').append('<p>Address: ' + this.address + '</p>');
                $('#stage1').append('<p>Phone: ' + this.phone_number + '</p>');
                $('#stage1').append('<p>' + this.description + '</p>');
                $('#stage1').append();
                /*$('#stage').append('<p> Id: ' + fb.id + '</p>');*/
               });
           });
        });
    </script>
```

Figure 13

The image above shows how “getjson” function was called for our website. For detailed explanation about the jQuery syntax, see the link: http://learn.jquery.com/about-jquery/how-jquery-works/.

Our group decided to implement the restaurants’ API (https://notoriousbiginteger.pythonanywhere.com/restaurants/json) and “Chang Thai”’s (the restaurant with the most positive reviews) dishes API (https://notoriousbiginteger.pythonanywhere.com/restaurants/3/dishes/json). We used their API to call the name, address, phone number, and description of the restaurants. Also, we implemented their API to display the name, average ratings, and description of the dishes.

## 4.5 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.