F.C. Team Possible

Project: World Cup 2014 Website

Group Members:

Aseal Yousuf

Jeremiah Martinez

Eros Smith

Kim-yu Ng

Prithvi Shahi

Jungmo Ku

Table of Contents

Introduction

This report describes all the stages of building a responsive and attractive website using Django, Bootstrap, CSS, and javascript. This report contains the how-to of setting up the basic framework for the site's layout, developing the API, and working models. The final result will yield a fully functional dynamic website.

Problem

Around the world association football, or soccer, has millions of ardent fans and admirers. Despite the need, these millions (even billions) of soccer lovers don't have a single website where all the latest information is gathered and easily accessible. There is a lack of rich media content including video, twitter, stats, and events of the current world cup all collected in one place, especially in english speaking countries such as the United States. At the moment, the average soccer fan has to bounce around many different sites to get the latest information they want, such as match highlights, their favorite player's tweets, and player's statuses. Even though some sites attempt to address this need, there is a lack of design consideration; the currently existing websites lack natural and intuitive interfaces to make all available media content easy to access.

Use Cases

The site’s interface allows navigation from a particular point of interest such as a favorite player to a certain match, from one country to another, all the while enjoying the content they desire, because each web component is interlinked with another. In fact, any twitter feed, stat, or video connected to the object of interest is on the site, at most, four clicks away. This eliminates the need of scouring Youtube for the game or player highlights or needing to search twitter to see what players are tweeting. Because of its design innovation, this website will bring together all of the most desired media content of today’s World Cup Soccer fans and provide them access to any article of interest quickly at the same time saturating them with other interesting content, facilitating hours of soccer bliss.

Use case example #1: After watching a world cup match a brazilian soccer fan remembers an amazing play by a german defender and he/she wants to know more about that player. He doesn’t remember the player’s name but he remembers the player’s general appearance. With the given information the brazilian soccer fan can go to the players page from the homepage and type the country into the search field on the table. This will allow the fan to see a list of all the german players. By looking at the position field, the fan can look at each of the four defenders player page until they recognize the player’s image. Now they have access to the players full stats and biography.

Use case example #2: A german soccer fan wants to relive the glorious moments of his team’s world cup experience. The german soccer fan can go to the matches page from the homepage and pick all the matches that germany appears in and link to that match page. The fan can now watch the highlights from any of those matches directly from the match page.

Use case example #3: Joe knows his favorite player's first name, Tim Howard. Joe can use the search in the sidebar by typing Tim Howard into it. Joe will be redirected to a search result page. The most relevant results, those with all of the words in the search, will appear on top of the search list so that Joe can find the results he is looking for quickly.

Required technologies

Django

Django is an open source, Python based web-framework. For this website, Django delivers the dynamically driven content of the site by accessing a database filled with information that was fed into to it.

1. Setting up Django

Installing Django

The framework of the website utilizes version 1.6.5 of Django because it is the most stable iteration released thus far. Using pip3, Django can be installed in bash with the command:

```
sudo pip3.4 install django==1.6.5
```

(Other options can be found at https://docs.djangoproject.com/en/1.6/intro/install/#install-django).

Starting a New Project

A new project can be started using the command:

```
django-admin.py startproject project_name
```

After this command a new folder with *project_name* should be in the current working directory. Contained within that directory should be another folder with the name *project_name* and a file named *manage.py*. The *manage.py* file runs commands for Django and should not be edited. Inside the second project_name folder there should be four files: *__init__.py, settings.py,* *urls.py,* and *wsgi.py.* The *__init__.py* file is there just to let Python know that this a Python folder. The *settings.py* and *urls.py* files will be edited during the set up of the static site. The *wsgi.py* file will be used in the configuration of hosting the site on PythonAnywhere.[2]

At this point it will be possible to run a private server at *http://127.0.0.1:8000* using the command:

```
python3 manage.py runserver
```

Installing Additional Packages

Next, some additional packages should be installed to help in later production phases using the command:

```
pip3.4 install south django-registration stripe
```

Then, create a new app using the command:

```
python3 manage.py startapp app_name
```

This should create a new folder named *app_name* with files *__init__.py, admin.py, models.py, tests.py*, and *views.py.* Add the string *'signups'* to the the tuple labeled *INSTALLED_APPS* in the *settings.py* in the first folder named *project_name.*[2]

2. Setting Up the Django Database for the Models

Setup MySQL Environment and Installation Guide

The operating system used is Ubuntu 14.04 and the command to install the MySQL connector from Oracle is:

```
sudo pip3.4 install --user https://dev.mysql.com/get/Downloads/Connector-Python/mysql-connector-python-1.1.6.tar.gz
```

Following this, MySQL 5.5 server has to be installed. The MySQL website details the instruction specific to the three different major operating systems. Next, the *settings.py* file has to be updated to use the oracle Django backend. This changes the default Django database engine from sqlite to MySQL so that the model can interact with a server backed by a MySQL database.[7]

Create Database Scheme from MySQL through terminal

In order to access MySQL as the root user, the command:

```
mysql -u root -p
```

has to be run through command prompt terminal. If there are permission problems with running the previous commands with password, the root user privilege of the MySQL has to be flushed. [7]

Grant user access to the database

A new user has to be created for the specific database and the user has to be granted access. This is done by the following commands which are entered in the MySQL interactive command prompt.[7]

```
CREATE USER ‘user’@’localhost’ IDENTIFIED BY ‘mypass’;
CREATE DATABASE any_name;
GRANT ALL ON any_name.* TO ‘user’@’localhost’;
```

Configure the database settings in Settings File

With the newly created user, the Django *settings.py* file has to be modified to recognize the changes made to MySQL i.e. set the location of which database is to be refered to and the data will be stored on that newly created database. This allows the Django model to interact with MySQL when the user is trying to perform data insertion, deletion or modification. The following is an example of the *settings.py* format:

```
DATABASES = {
    'default': {
        'ENGINE': 'mysql.connector.django',
        'NAME': 'schema_name'),
        'USER': 'username',
        'PASSWORD': 'user_password',
        'HOST': 'localhost', #On pythonanywhere, change to “mysql.server”
        'PORT': '3360',
        }
}
```

Create the designed model from *model.py*

Following these steps, the hypothetical model becomes realized and all the tables are created from model design.

```
python3  manage.py syncdb
```

Load required data with Python scripts

Instead of inserting player, county, and match information by hand, an automated script was created and fed data encoded as a dictionary.

Obtain the data in the JSON format first and then process the data i.e. take all necessary bits of information and reformat it in a dictionary format. This reformatted data is also stored in a .json format with a .txt file extension.

This is an example command of how to feed in Python script in order to insert country information into the tables.

python3 manage.py shell < country_insert.py

The above step is repeated with this command.

data: the "country_insert.py", "player_insert_script.py", "match_insert_script.py"

All the data and information inserted can be inspected through the administrative panel.

localhost:8000/admin

3. Using TastyPie to Build API

How to install TastyPie

Use this command to install tastypie:

pip3.4 install --user django-tastypie

Install using the *pip3.4* command on PythonAnywhere in order to link to the correct version. Add *tastypie* to the list of installed apps in *settings.py*. Create a new file named *api.py.* The file with one resource added looks like:

```
from tastypie.authorization import Authorization
from tastypie import fields
from tastypie.resources import ModelResource
from wc_app.models import *
from wc_app.prettyPrint import *
from tastypie.constants import *
from django.conf.urls import *

class CountryResource(ModelResource):
    class Meta:
        queryset = Country.objects.all()
        resource_name = 'countries'
        authorization = Authorization()
        serializer = PrettyJSONSerializer()
        filtering = {
            "country_name" : ALL,
            "country_code" : ALL
        }
        detail_uri_name = 'country_name'

    def prepend_urls(self):
        return [
            url(r"^(?P<resource_name>%s)/(?P<country_name>[\w\d_.-]+)/$" % self._meta.resource_name,
self.wrap_view('dispatch_detail'), name="api_dispatch_detail"),
        ]

    def determine_format(self, request):
        return "application/json"
```

In order to hook up the resource, add new urls to the *urls.py* file*.* The file with one resource added to the url patterns like:

```
from django.conf.urls import patterns, include, url
from django.conf import settings
from wc_app import views
from django.contrib import admin
from wc_app.api import *

urlpatterns = patterns('',
...
 #RESTful API
url(r'^api/', include(CountryResource().urls)),
...)
```

After restarting the django server, the API is ready to serve up default content. [3]

Using TastiePie to Show All Elements in a Resource

By default, the entire resource is available after adding it correctly. The resource can be retrieved through adding *?offset={offset}&limit={limit}* onto the url. For example the url

../api/countries/?offset=0&limit=32

would return all of the elements in the country resource.

Using TastiePie to Show Specific Elements in a Resource

The filtering dictionary contained within the resource *api.py* allows the API to return specific elements based on a key value contained in the listed fields. In the case of the country resource a specific country element can be returned by appending the search to the url, such as

../api/countries/USA

will return the USA element. Also, to increase the flexibility of using the created API, for instance, one could obtain country information of a specific country through attributes of filter that allowed on the correspond resource class

../api/countries/?country_code=argentina.

../api/countries/?country_code=arg

4. Adding 404 error

An error page titled “404 error” is presented when an illegal url is accessed through our web host. In order to achieve this, edits were made to *urls.py.*

```
#Error 404
url(r'./$', views.handler404, name='handler404')
```

In addition these lines were added to *views.py*:

```
#Error 404 page
def handler404(request):
   return render(request, '404.html')
```

Most views will call handler404 function if the requested data does not exist. Anything other than the defined paths will return the customize 404 page to user.

5. Watson search

Installing watson

1. Add ‘watson’ to your INSTALLED_APPS in the *settings.py* file.

2. Run the command *python3 manage.py syncdb*.

3. Run the command *manage.py installwatson*.

3. Register models inside *models.py* with the following code:

```
import watson
watson.register(YourModel)
watson.register(YourOtherModel)
```

Implementing search page

Using a form in html of a page with a search field which is contained within *sidebar.html*, a request is passed with a “q” property which contains the search string inside of *views.py*. The query obtained via GET parameter is split using white space for the “or” search and put in a list of strings, while the string is kept whole inside another variable for the “and” search. Using watson, a query is made on all of the strings with the command *results=watson.search(query, ranking=True)*. The “and” results and “or” results are combined into a list. For every search the results can be iterated over and the context and url of each result can be extracted. For all the context in results the search words are replaced with bold text. The results are then passed back to the html for display. Because the and results are appended first the results are naturally ordered in the correct priority of importance. [11]

Bootstrap

Bootstrap is a open source front-end framework that provides responsive web-design with little to no effort while offering customizable elements and templates.

1. Setting up Bootstrap

In order to make a sleek, intuitive, and powerful mobile first front-end framework for faster and easier web development, the up-to-date Twitter bootstrap css v3.2.0 was included. Go to http://getbootstrap.com/ and download the code available on the site and include it in the project folder. Then set a path in the base HTML file so that it enables the use of bootstrap.

jQuery library for bootstrap dropdown

```
<script src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.0/jquery.min.js"></script>
```

Twitter bootstrap css in minified form

```
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
```

Twitter bootstrap default javascript

```
<script src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
<script src="/static/js/bootstrap.min.js"></script>
```

2. Getting Templates

Templates are designs used in designing web pages. To find a template search on Google and choose a template. There are many templates available as open source on the web for bootstrap.

3. Customizing templates using CSS

Include a *custom.css* file to override current css attributes. When using a web browser like Firefox there is an inspect element option which allows the CSS to be viewed and edited within the browser. By selecting the appropriate code, editing it, and placing it in the *custom.css* file the default format can be overridden.

Apiary

Apiary provides a convenient way to mock up a RESTful API. Using the in-site API Blueprint Editor, a mock up can be generated, documented, and shared with team members.

1. Setting up Apiary

Go to www.*apiary.io* and sign in with the project's git hub account.

2. Creating a mock up

Use the Apiary site's embedded editor to create and modify the API mock up.

3. Resulting API

GET /api/countries/?offset={offset}&limit={limit}

List first {limit} Matches starting from {offset}. Queries are bounded by the number of matches which is 64.

GET /api/countries/{id}

Displays all the details of a specific Country object.

GET /api/players/?offset={offset}&limit={limit}

List first {limit} Players starting from {offset}. Queries are bounded by the number of Players which is 736.

GET /api/players/{id}

Displays all the details of a specific player object.

GET /api/matches/?offset={offset}&limit={limit}

List first {limit} Matches starting from {offset}. Queries are bounded by the number of Matches which is 64.

GET /api/matches/{id}

Displays all the details of a specific match object.

PythonAnywhere

PythonAnywhere is a web hosting service as well as an online IDE based on the Python programming language. This acts as a server to host the site. PythonAnywhere has the resources and the environment to setup Django application as well as CSS bootstrap.

1. Setting up PythonAnywhere

Create a new PythonAnywhere account. To clone the github repository on PythonAnywhere go to the bash console. Then at the root directory use the git clone command and drag and drop the clone link on the console to clone.

2. Hosting the site

To allow the site for public access, go to dashboard and under the web tab, make a new web application. Then, there will be a configuration popup, choose manual configuration and choose the corresponding Python version. In this project, the project is setup to environment with Python 3.4.1 support. Next, change the settings on *wsgi.py* as they came with default flask settings which is not related to the project.[8]

```
import os
import sys
if path not in sys.path:
        sys.path.append(path) os.environ['DJANGO_SETTINGS_MODULE'] = 'world_cup.settings'
import django.core.handlers.wsgi application = django.core.handlers.wsgi.WSGIHandler()
```

The above is an example of how to link to the project repository that has just been cloned in PythonAnywhere so that the hosting site can recognize where the project is located at. One should change *world_cup.settings* to his or her Django application path respectively. The reason to use manual configuration instead of opting for Django configuration on PythonAnywhere is because the default Python, bundled with the Django settings, might require more modification to the project repository as compared to manual configuration of Python that does not serve with the Django module. The reason to do it this way is to publish the Django application from PythonAnywhere as it was like running the project on localhost without having the need to touch the source files.[8]

Setting up Python Anywhere on MySQL[9]

1. Create the database with ‘*username$db_schema_name*’

2. Clone the github directory and setup wsgi to the path of cloned folder

3. Change settings.py db_name to ‘*username$db_schema_name*’

4. Install django, mysql-connector, tastypie module through pip as described before in the documentation. In our case, for example:

```
pip3.4 install --user django-tastypie
```

5. Run syncdb to generate tables from model

6. Run 3 scripts to automate the process of inserting data

7. Reload the web application and everything should be good to go

Testing

The tests were written for Django models, TastyPie API, and Watson search. For the Django models, the tests make sure that calls to a Django class are correct, since one HTML page is used for each category of pages with only the subject's information (country, player, or match) changing. The TastyPie tests made sure that a call with website/api/<page> will return the correct information that is presented from that page. Like the TastyPie tests, the Watson tests make sure that a search result will return appropriate information, in our case the search characters are a subset of the results displayed.

Building tests for the models using Django unit tests

The required imports:

Add these imports to the *test.py* file:

```
import json
from django.test import TestCase
from name_of_app.models import *
```

import json #is needed to read the json files that is fed for testing. from django.test import TestCase #is needed so that the unit tests are able to run properly lastly from name_of_app.models import * #are needed to import the models for testing.

Make a class for the Django model tests

An example of a class created in *test.py* :

```
class ModelTestCase(TestCase):
        def test_country_model(self):
                #Dictionary Key: Country Name
                #Dictionary Value: [Country_code, country_rank]
                country_test_dict = {"Brazil": ["BRA", 5]}
                Country.objects.create(country_name="Brazil",
                country_code=country_test_dict1["Brazil"][0],
                rank = country_test_dict1["Brazil"][1]facacd
                Country_Brazil = Country.objects.get(country_name="Brazil")
                self.assertEqual(Country_Brazil.country_name, "Brazil")
                self.assertEqual(Country_Brazil.country_code, "BRA")
                self.assertEqual(Country_Brazil.rank, 5)
```

The test class above creates a new country object using the model by the following code:

```
Country.objects.create(country_name="Brazil",
country_code=country_test_dict1["Brazil"][0],rank = country_test_dict1["Brazil"][1]).
```

The parameters taken in by the country model is the country name, country code and rank. The Country.objects.get(country_name="Brazil") code is used to retrieve the country object. Asserts are used to test whether the data inputted to the database are correctly returning the data that is expected. All nine tests included in the *test_model.py* almost follow the exact pattern of testing. Instead of taking input data from a hard coded dictionary, three of the tests take in data from a file.

The tests that are checking foreign keys must make a foreign object first and then pass it in as a parameter for the model that is using the foreign key otherwise the test will fail since the model is expecting a foreign key object. For example:

```
def test_player_model1(self):
        #Dictionary Key: Player full name
        #Dictionary Value: [sur_name, country,Clubname,Position,Birthdate]
        player_test_dict1 = {"Andrea Barzagli": ["Barzagli", "Italy", "Juventus FC", "Defender",
        "1981-05-08"]}

        Country.objects.create(country_name = player_test_dict1["Andrea
        Barzagli"][1])
        c1 = Country.objects.create(country_name = player_test_dict1["Andrea
        Barzagli"][1])

        Player.objects.create(country=c1, sur_name= player_test_dict1["Andrea
        Barzagli"][0],full_name = "Andrea Barzagli" ,clubname =
        player_test_dict1["Andrea Barzagli"][2], position = player_test_dict1["Andrea Barzagli"]
        [3], birth_date =player_test_dict1["Andrea Barzagli"][4])

        player_get = Player.objects.get(full_name = "Andrea Barzagli")
        self.assertEqual(player_get.country.__str__(),
        player_test_dict1["Andrea Barzagli"][1])
        self.assertEqual(player_get.sur_name, player_test_dict1["Andrea Barzagli"][0])
        self.assertEqual(player_get.full_name, "Andrea Barzagli")
        self.assertEqual(player_get.clubname, player_test_dict1["Andrea Barzagli"][2])
        self.assertEqual(player_get.position, player_test_dict1["Andrea Barzagli"][3])
        self.assertEqual(player_get.birth_date.__str__(), player_test_dict1["Andrea
        Barzagli"][4])
```

The country object is being created first then passed in as a parameter for the player model. One thing to note is the usage of the .__str__() in the context of the country foreign key. This is used so that the name of the country is returned as a string instead of the country object being returned. The .__str__() function in the context get.birth_date is used because the format in which the date is saved is different then the string of the birth date.

Building Unit Tests for TastyPie API

Currently, API tests are written inside *tests.py* under the *wc_app* folder. In order to test API, below is an example of testing countries API with local server. After loading the correct data into the database, the response will contain a list or dictionary containing relevant data to the specific url. We test that data against the expected results. The test will run along with the commands below:

```
# will run all the py file starts with test
# require to import unittest from django
python manage.py test

# run server so that tests of API can access information from db
python manage.py runserver
```

Repeat the above for matches and players APIs. Same things go to specific country, specific player and specific match designed API.

```
class APItests(unittest.TestCase) :
    url = "http://127.0.0.1:8000/"

    #Countries
    def test_get_all_countries(self) :
        request = Request(self.url+"api/countries/")
        response = urlopen(request)
        response_body = response.read().decode("utf-8")
        self.assertEqual(response.getcode(), 200)
        response_data = loads(response_body)
        response_objects = response_data["objects"]
        expected_response = [{Answer Dictionary}]

        for obj in response_objects:
```

```
        for key in obj:
            if type(obj[key]) == list:
                obj[key] = sorted(obj[key])

    for obj in expected_response:
        for key in obj:
            if type(obj[key]) == list:
                obj[key] = sorted(obj[key])

    for obj in response_objects:
        self.assertTrue(obj in expected_response)
```

Building Unit Tests for Watson Search

The watson tests rely on the watson module so watson must be added to the imports. Also in order to test the watson search, watson needs to be able to build itself during runtime. This is done by importing *call_command* from the library *django.core.management.*

The first test creates three country objects with one matching text field and creates two different queries with which to search them with. Two of the objects contain, as a substring, the contents of the first query while only one of the objects contain an exact match of the second query. Due to this setup, it is known that the first query should return two results and the second query should return one result. These predicted results are hardcoded as the expected results for each search. When the test runs a watson search on the actual objects, the expected results are tested against the runtime results.

The second test differs in that a mix of different text fields and non-text fields are added to multiple player objects. The test checks that non-text field are indeed unsearchable at the same confirming the text fields of multiple non-matching are searchable.

The third test explores the edge cases involving the dash within a text field. The dash makes exact match searches fail but show that the dash does not prevent substring searches from returning true.

Running the tests

Since the database is backed by MySQL, in order to run the unit tests, one has to setup the model mentioned above. The unit test is located at the path: *world_cup/wc_app/test_Model.py* The following is the command to run the test:

```
python3 manage.py test
```

Design Choices

Final Web Design

The website has a navigation bar that links to the homepage, the “about us” page and dropdowns for countries, players and matches. The homepage has three links that link to a page containing the countries, a page containing all of the players, and to a page that contains all of the matches.

The countries page currently contains a view of 8, 16, or 32 countries at a time, which can be sorted alphabetically, by rank, and by region. On each country page there is an embedded Google Map with the country location, a grid of players for that nation, average statistics for the country, and the countries biography in terms of their World Cup history, and a video of interviews with players. Inside an individual country’s page, all the players are linked to their details.

The players page contains a view of 23 (for a whole team) - 736 players at a time, which can be sorted alphabetically, shirt number, country, and position. On each player page there are two tabs, one containing their player details, biography, and photo and the other having their country flag and club-team name.

The matches page contains a list of all 64 matches, showing both team's emblems, the match score, match winner, and the location of where the match took place, with the matches sorted by match number. On each match page the score is shown along with both countries' flags, an embedded highlight video of the match, an embedded Google map showing the location of the match, and attacking and defending match statistics for each team. There are links to each subsection at the top-right corner of the page.

To integrate the Flappy API, a table was constructed from the city json returned from api/cities/ command the which contained the location, description, and ID of each city. Then relationships were derived from the jsons resulting from the api/languages/ and api/activities/ commands effectively joining the three tables together. Links were inserted into each of the html text of the cities which open a pop-up window that explores the derived relationships, offering extra information relating to each city. This allows a tourist to have greater insight into potential traveling spots.

Django Model Design

The Django models represent the three main subjects of the site: the country model, the player model, and the match model. The country model consists of the country name, a country code and the rank of that country. The player model consists of a foreign key that comes from the country model that links the player to that specific country. The remaining fields contain the player surname, full name, the club he/she plays for, the position he plays and his birthdate. The matches model uses two foreign keys from the country model. This is done to link the two countries playing against each other. The other fields consist of the match number, the winner of the match, the scores of the match, the location where it was held and when it was held.

The Django model generates default primary keys by assigning them to the created id field as integers. The following model view shows the foreign keys on the match table as country_A_id and country_B_id respectively. It makes sense to design the model to have the country as the foreign key in the match table because every single match will have a home team and an away team. By doing this the object can be used to access all the attributes from the country class. For example, Brazil is playing against Germany and a user wants to know about the ranking of Brazil in FIFA ranking, he or she can get the rank easily as the rank variable can be accessed through the match model. Same thing goes for player table; every player is associated with a country.

Updated:

Player and Matches have been changed since the beginning of second iteration. Below are extra fields added to improve the comprehensiveness of data to be represent on our web site:

<u>In Player Class of wc_app.model:</u>

```
    #add
  international_caps = models.IntegerField(default=0)
  goals = models.IntegerField(default=0)
  height = models.IntegerField(default=0)
  first_international_appearance = models.CharField(max_length=500)
  biography = models.CharField(max_length=5000)
```

<u>In Match Class of wc_app.model:</u>

```
  map_location = models.CharField(max_length=500)
  highlight_url = models.CharField(max_length=500)
```

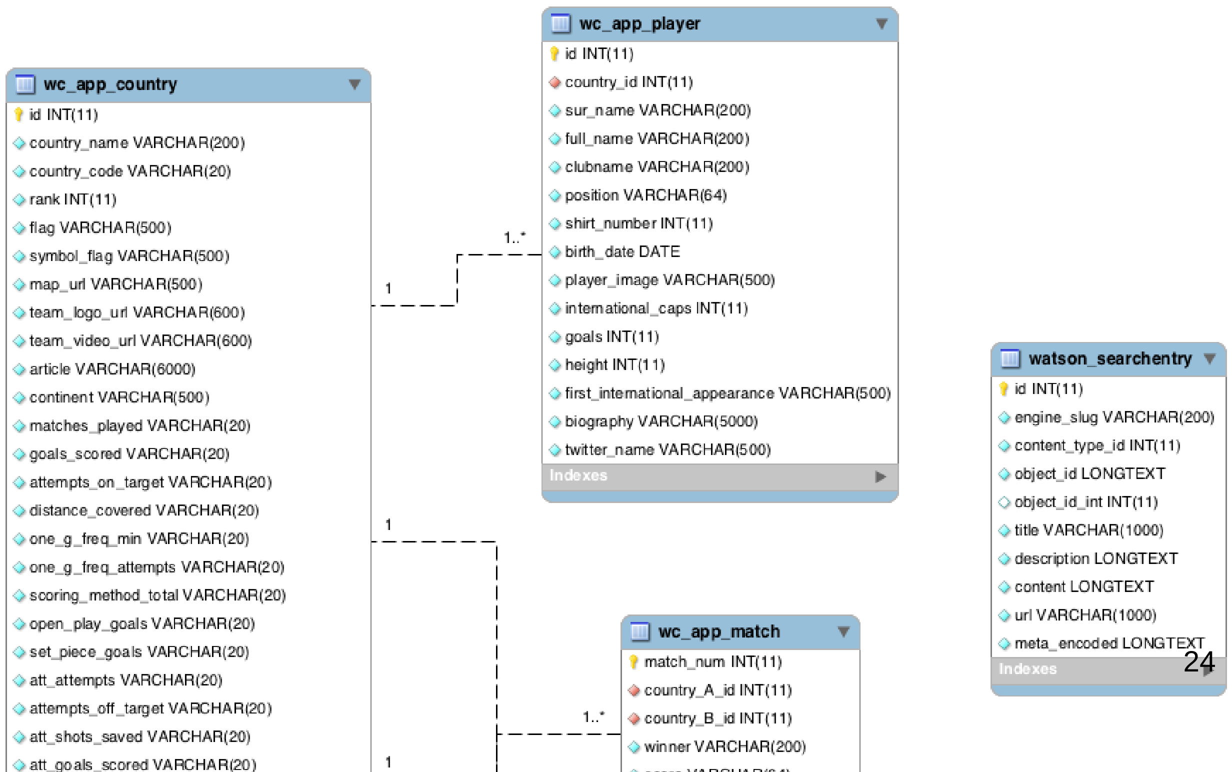
How to Insert Text Dynamically

In order to insert dynamically generated text, inline python scripts can be inserted into the html.

For example, to insert a player name code would look like:

```
<li class="p-name">{{full_name}}</li><li class="p-team">
```

The python code is contained *{{here}}* and pulls data from the database.



24

How to Insert Images Dynamically

The images are loaded dynamically by inserting the link text as per the method explained above and loaded in via the link.

How to Insert Videos Dynamically

The videos are loaded dynamically by inserting the link text as per the method explained above and loaded in via the link.

How to Insert Google Maps Dynamically

The Google Maps are loaded dynamically by inserting the link text as per the method explained above and loaded in via the link.

Conclusion

While the implementation for this project is specifically about designing a website that provides information for the current FIFA World Cup, the principles and design could be directly applied to almost any current event of public interest, and more generally to any website that could benefit from scalable and dynamically generated content.

References

For apiary RESTful API.

http://apiary.io/,

Tutorial steps for Django.

https://docs.djangoproject.com/en/1.6/intro/tutorial01/

Documentation for tastypie.

http://django-tastypie.readthedocs.org/en/latest/

For match highlight videos

http://www.espnfc.us/video/highlights/114/index

For biographies, pictures, player data.

http://www.fifa.com/worldcup/index.html

For twitter bootstrap

http://getbootstrap.com/

Documentation for mysql.

http://dev.mysql.com/doc/

For PythonAnywhere questions.

https://www.pythonanywhere.com/wiki/

For setting up PythonAnywhere on MySQL.

https://www.pythonanywhere.com/wiki/UsingMySQL

Where we found match data.

http://worldcup.sfg.io/

Watson github

https://github.com/etianen/django-watson