Flappy Birds
Yu Lu
Daniel Garza
Jesús Galván
José Coello
Juan Pablo Mata
Samuel Acuña

Phase III Report

Introduction

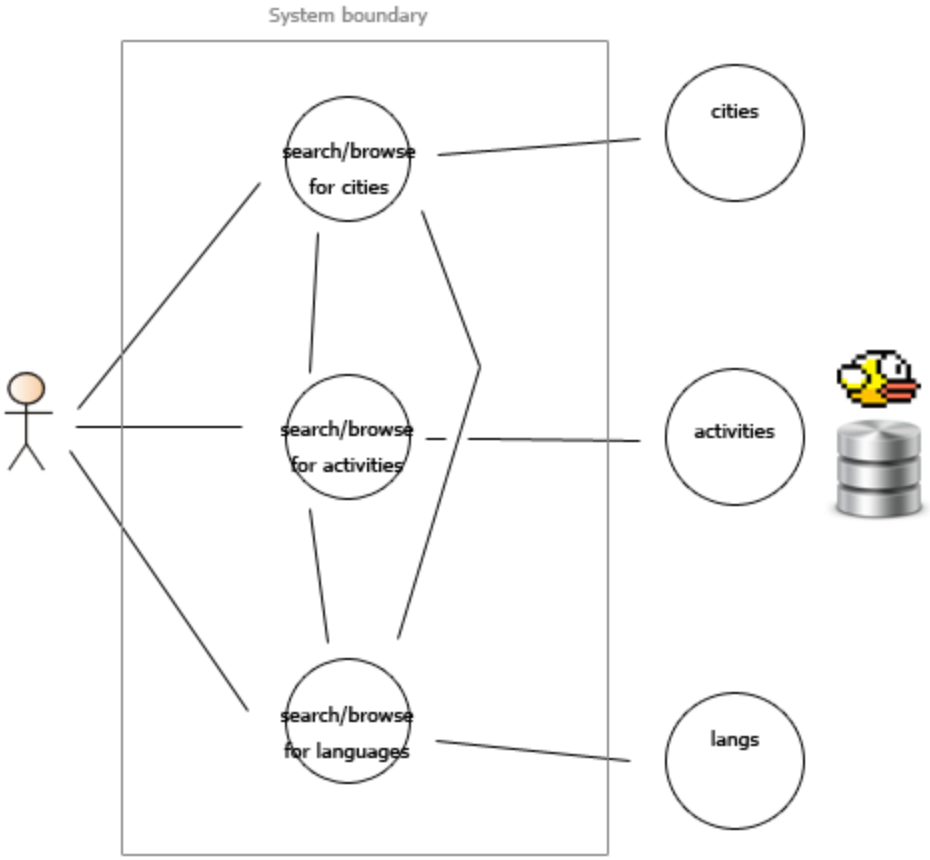
After discussing several ideas, we settled on developing a webapp that stores, relates, and displays data on tourism destinations based on corresponding cities, activities, and languages. We decided to name the webapp “Flappy World” due to our initial interest for this project in indie video games and our resulting group name “Flappy Birds”.

Problem

There is a large variety of tourism destinations where their locations, landscapes, history, and attractions differ. It is often hard to decide where to go for a visit. Flappy World aims to make it a fun process by exploring different cities by their locations, attractions and even the local languages, so that users have sufficient information to find the perfect destination for vacation.

Use Cases

The users navigate to the website seeking a touristic destination. They have the options to browse from a list of cities, activities, and/or languages or simply search within any of those categories. Once they have selected an item, more specific information is presented, as well as items with the same or related characteristics.



Design

RESTful API

Our RESTful API can be accessed both internally and externally, namely, from our own webapp and server or from external ones. This is a public API that can be used to extract information from our database (read-only). OAuth is not required. Any client can request languages (a list of them all or particular ones by id or by name). Similar requests can be done with cities. For activities, a list of activities by type can be requested as well as, like for languages or cities, particular ones by id or name. Below is the documentation for the requests and responses.

Languages

Languages Collection [GET]

`/api/languages` : List all languages

Response

```
200 (OK)
Content-Type: application/json
[{
        "id": 1, "name": "Chinese"
},{
        "id": 2, "name": "Spanish"
}]
```

Response

```
400 (Bad Request)
Content-Type: application/json
{"error": "Data does not exist. Bad Request."}
```

Particular Language [GET]

`/api/languages/id/{id}` : Retrieves a language by id

Parameters

`id` - Numeric id of the language to perform action with. Required.

Request

```
/api/languages/id/2
```

Response

```
200 (OK)
Content-Type: application/json
X-My-Header: The Value
{ "id": 2, "name": "Spanish", "description": "About Spanish", ... }
```

Response

```
400 (Bad Request)
Content-Type: application/json
{"id": 351, "error": "Language with id 351 does not exist in the database."}
```

`/api/languages/name/{name}` : Retrieves a language by name

Parameters

`name` - String name of the language to perform action with. Required.

Request

```
/api/languages/name/Spanish
```

Response

`200` (OK)

```
Content-Type: application/json
X-My-Header: The Value
{ "id": 2, "name": "Spanish", "description": "About Spanish", ... }
```

Response

`400` (Bad Request)

```
Content-Type: application/json
{ "name": Tibetan, "error": "Language with name Tibetan does not exist in the database."}
```

Cities

Cities Collection [GET]

`/api/cities` : List all cities

Response

```
200 (OK)
Content-Type: application/json
[{
    "id": 1, "name": "Li Juang", "description": "some text", ...
}, {
    "id": 2, "name": "Chichen Itza", "description": "some text", ...
}, ...]
```

Response

```
400 (Bad Request)
Content-Type: application/json
{"error": "Data does not exist. Bad Request."}
```

Particular City [GET]

`/api/cities/id/{id}` : Retrieves a city by id

Parameters

`id` - Numeric id of the city to perform action with. Required.

Request

```
/api/cities/id/2
```

Response

```
200 (OK)
Content-Type: application/json
X-My-Header: The Value
{ "id": 2, "name": "Chichen Itza", "description": "some text", ... }
```

Response

```
400 (Bad Request)
Content-Type: application/json
{"id": 351, "error": "City with id 351 does not exist in the database."}
```

`/api/cities/name/{name}` : Retrieves a language by name

Parameters

`name` - String name of the city to perform action with. Required.

Request

```
/api/cities/name/Chichen Itza
```

Response

```
200 (OK)
Content-Type: application/json
X-My-Header: The Value
{ "id": 2, "name": "Chichen Itza", "description": "some text", ... }
```

Response

```
400 (Bad Request)
Content-Type: application/json
{ "name": Austin, "error": "City with name Austin does not exist in the
database."}
```

Activities

Activities Collection [GET]

`/api/activities/type/{type}` : List all activities of a given type

Parameters

`type` - String type of the activity to perform action with. Required.

Request

```
/api/activities/type/Scenery
```

Response

```
200 (OK)
Content-Type: application/json
[{
```

```
        "id": 1, "name": "Some activity", "description": "some text", "type":
        "Scenery", ...
    }, {
        "id": 2, "name": "Other activity", "description": "some text", "type":
        "Scenery", ...
    }, ... ]
```

Particular Activity [GET]

`/api/activities/id/{id}` : Retrieves an activity by id

Parameters

`id` - Numeric id of the activity to perform action with. Required.

Request

```
/api/activities/id/1
```

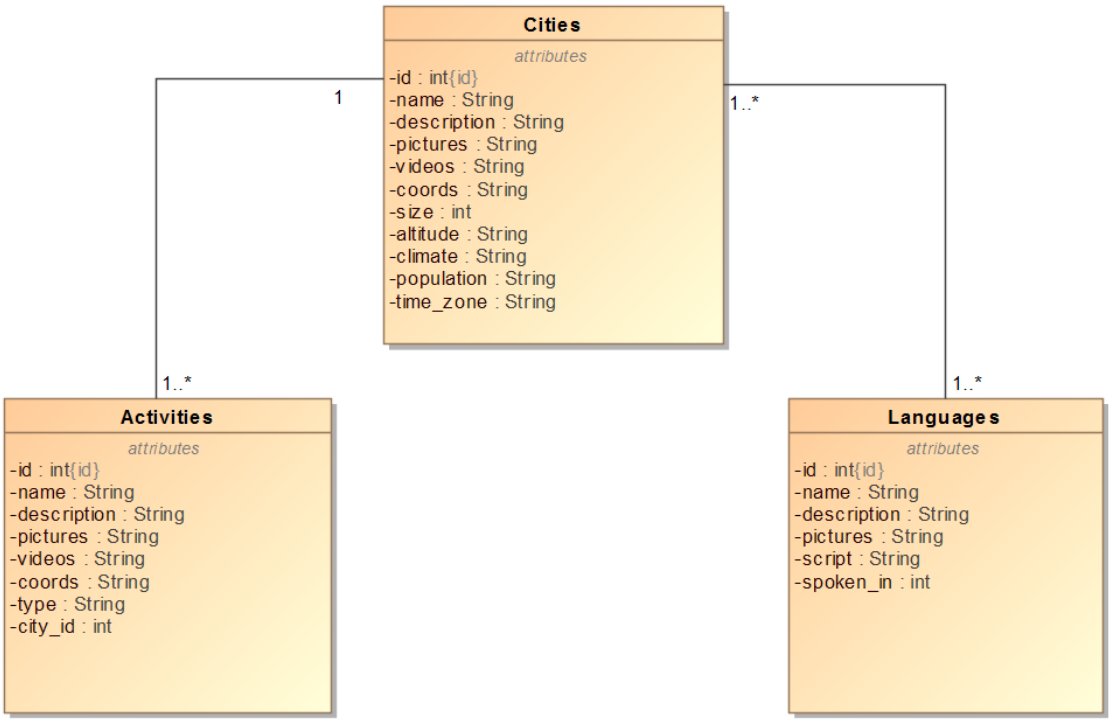
Response

```
200 (OK)
Content-Type: application/json
X-My-Header: The Value
{ "id": 1, "name": "Some activity", "description": "some text", "type":
"Scenery", ... }
```

Django Models

Django Models represent how data will look and function in a DataBase. Each model can be described as a table that contains attributes that are essentially the names of the columns that will be in a table. If you are familiar with SQL it is very similar to doing a “CREATE TABLE”. Models in Django are a bit more abstract in the sense that they act more as an interface that does the SQL part in the background hidden from the developer. Instead of “CREATE TABLE” Django has a class module you can import that is called models and you create a “class <table name>” that contains the attributes you want. Django also handles creating the database itself if you wish.

Our Django Models were defined in our Django project as is stated on the Django documentation. We have one app called content_display that contains models corresponding to Cities, Activities, and Languages. In our Cities model we only have information that is necessary for the Cities pages on our website, this includes name, description, specific pictures and videos. The Activities and Languages models also have information that is specific to either a particular activity or a particular language. The UML diagram below shows the relationship between our tables. It also displays the attributes we chose for each table.



This UML diagram was the basis for our model implementation. In the table, Activities, city_id is a Foreign key that will relate to a specific city in Cities.. The relationships can be read as one city can have one or many activities and one or many activities can belong to one cities. This is due to the fact that they are particular activities that are specific to the city. The other relationship can be read as one or many cities can have one or many languages and one or many languages can belong to one or many cities. The UML to Django Model translation was very simple. Below is an example of what a model would look like for Cities.

```
class Cities(models.Model):
    '''
    Cities model has information for the Cities pages, this includes name, description,
    specific pictures, videos, map, coordinates, and other city information.
    '''
    name = models.CharField(max_length=30)
    country = models.CharField(max_length = 30)
    description = models.TextField()
    pictures = models.TextField()
    videos = models.TextField()
    coords = models.CharField(max_length=30)
    size = models.CharField(max_length=30)
    altitude = models.CharField(max_length= 30)
    climate = models.CharField(max_length= 30)
    population = models.CharField(max_length=20)
    time_zone = models.CharField(max_length= 20)


    def __str__(self):
        return '%s %s %s %s %s %s %s %s %s %s' % (self.name, self.description, self.pictures,
            self.videos, self.coords, self.size, self.altitude, self.climate, self.population,
            self.time_zone)
```

As you can see it follows almost the same format as the UML diagram above. For each attribute we defined how we wanted it to be stored in the database. We mostly stored charField and TextFields which are strings. The only big difference from this models and the UML is that in the Django model we did not make an attribute called id even though we specified it in UML. This is because Django automatically gives you an attribute called id if you do not explicitly call it. The bottom __str__ method is used by django for information.

The Activities class model specifies the foreign key with city_id. This creates the many to one relationship between activities and cities. The rest of the attributes for activities are also strings as you can see in the picture below.

```
class Activities(models.Model):
    '''
    The Activities has information that is specific
    to a particular activity. This includes name, description,
    pictures, videos, city, map coordinates, and type of activity.
    '''
    name = models.CharField(max_length=30)
    description = models.TextField()
    pictures = models.TextField()
    videos = models.TextField()
    coords = models.CharField(max_length=60)
    type_activity = models.CharField(max_length=60)
    city = models.ForeignKey('Cities')

    def __str__(self):
        return '%s %s %s %s %s %s' % (self.name, self.description, self.pictures,
                self.videos, self.coords, self.type_activity)
```

The final class model is Languages. This one is the one that has the many to many relationship with cities. The way to specify this is by putting the ManyToManyField in Languages pointing to cities as in the picture below.

```
class Languages(models.Model):
    '''
    The Languages has information that is specific
    to either a particular language. This includes name, description,
    and languages.
    '''
    name = models.CharField(max_length=30)
    description = models.TextField()
    pictures = models.TextField()
    script = models.TextField()
    spoken_in = models.ManyToManyField('Cities')

    def __str__(self):
        return '%s %s %s %s' % (self.name, self.description,
                self.pictures, self.script)
```

Once we made all the models inside content_display we started creating the MySQL database. When we sync this model django automatically creates another table in

between cities and languages that has attributes language_id and cities_id and keeps track of the relationships through that table. Once our we made the connection to the database we ran the syncdb command in django and it created the tables inside the database for us.

We then created JSON files of all our data research and compiled it into one 2 files. We put our data with the same attributes as our tables in so it would be easier to handle once we tried inserting in the database. We then made a script to format all the data into Insert query statements. Once we had all the query statements we created a script that ran all the queries and inserted all the data automatically into the database. Below is a snippet of what the queries looked like.

```
insert into content_display_languages (name,description,pictures,script) values ('German','German is a West Germanic language. It derives most of its vocabulary
insert into content_display_languages (name,description,pictures,script) values ('English','English is a West Germanic language that was first spoken in early
insert into content_display_languages (name,description,pictures,script) values ('Italian',' Italian is a Romance language spoken mainly in Europe: Italy, Swit
insert into content_display_languages (name,description,pictures,script) values ('Spanish','Spanish is a Romance language that originated in Castile, a region
insert into content_display_languages (name,description,pictures,script) values ('Egyptian Arabic','Egyptian Arabic is the language spoken by most contemporary
insert into content_display_cities (name,country,description,pictures,videos,coords,size,altitude,climate,population,time_zone) values ('Sydney','Australia','Sy
insert into content_display_cities (name,country,description,pictures,videos,coords,size,altitude,climate,population,time_zone) values ('New Delhi','India','Ne
insert into content_display_cities (name,country,description,pictures,videos,coords,size,altitude,climate,population,time_zone) values ('Saint Petersburg','Rus
insert into content_display_cities (name,country,description,pictures,videos,coords,size,altitude,climate,population,time_zone) values ('Bangkok','Thailand','Ba
insert into content_display_cities (name,country,description,pictures,videos,coords,size,altitude,climate,population,time_zone) values ('Cape Town','South Afri
insert into content_display_cities (name,country,description,pictures,videos,coords,size,altitude,climate,population,time_zone) values ('Rio de Janeiro','Brazi
insert into content_display_cities (name,country,description,pictures,videos,coords,size,altitude,climate,population,time_zone) values ('Nanjing','China','To be
insert into content_display_cities (name,country,description,pictures,videos,coords,size,altitude,climate,population,time_zone) values ('Lijiang','China','Liji
insert into content_display_cities (name,country,description,pictures,videos,coords,size,altitude,climate,population,time_zone) values ('Vatican City','Italy',
insert into content_display_cities (name,country,description,pictures,videos,coords,size,altitude,climate,population,time_zone) values ('Grand Canyon National
insert into content_display_cities (name,country,description,pictures,videos,coords,size,altitude,climate,population,time_zone) values ('Cancun','Mexico','Canc
insert into content_display_cities (name,country,description,pictures,videos,coords,size,altitude,climate,population,time_zone) values ('Giza','Egypt','Giza is
insert into content_display_activities (name,description,pictures,videos,coords,type_activity) values ('Sydney Opera House','The Sydney Opera House is a multi-
insert into content_display_activities (name,description,pictures,videos,coords,type_activity) values ('Shelly Beach','Shelly Beach (also known as Shelley Beac
insert into content_display_activities (name,description,pictures,videos,coords,type_activity) values ('Taj Mahal','The Taj Mahal from Persian and Arabic, "cro
```

Testing

Unit Tests

Our tests were made using the TestCase we imported from django.test. With this library we were able to test our Django Models without having the database setup and have

at least some of them pass. Without it we would have to create the database and load all the data then test it. WIth TestCase we loaded dummy data based on our models and just checked to see if the tables would get created and the data added. To do this we did Cities.objects.create(<data>) and this would load a fake database with dummy data and we would check to see if what we put in was what we got out. We also tested by not filling in all the attribute fields and this would return “ u’’ “ instead of None. After getting used to the syntax it is very simple to test with TestCase and to run you just run the command “python manage.py test” this automatically runs all of the tests and gives you immediate feedback if it passed or failed or even if you had an error. Below is an example of the tests for the Cities model.

```python
from django.test import TestCase
from content_display.models import Cities

# Testing City Model with dummy data

class CityTest(TestCase) :
    def setUp(self) :
        Cities.objects.create(name = 'Li Jiang', size = '8,193 sq mi', population = '1,244,769')
        Cities.objects.create(name = 'Chichen Itza', size = '123 sq mi', coords = '1.0, 2.0')

    def test_c1(self) :
        lj = Cities.objects.get(name= 'Li Jiang')
        ci = Cities.objects.get(name= 'Chichen Itza')
        self.assertEqual(lj.name, 'Li Jiang')
        self.assertEqual(ci.name, 'Chichen Itza')
        self.assertEqual(Cities.objects.get(id = 1).name, 'Li Jiang')
        self.assertEqual(Cities.objects.get(id = 2).name, 'Chichen Itza')

    def test_c2(self) :
        lj = Cities.objects.get(name= 'Li Jiang')
        ci = Cities.objects.get(name= 'Chichen Itza')
        self.assertEqual(lj.altitude, u'')
        self.assertEqual(lj.time_zone, u'')
        self.assertEqual(ci.description, u'')

    def test_c3(self) :
        lj = Cities.objects.get(name= 'Li Jiang')
        ci = Cities.objects.get(name= 'Chichen Itza')
        self.assertEqual(lj.population, '1,244,769')
        self.assertEqual(lj.time_zone, u'')
        self.assertEqual(ci.size, '123 sq mi')
        self.assertEqual(ci.coords, '1.0, 2.0')
```

In this example you can see that you can just call the model by name and use methods defined in TestCase to create the data and get the data. So Cities.objects.create(<data>) creates an entry to a table called Cities and Cities.objects.get(name=<cityname>) will get the data from the table that has that particular name. We are assuming that all names are unique in our data sets, but we have the id field that can always be certain to be unique. By doing these tests we can make sure that our models work and can be later implemented when creating the actual database.

Unit Tests for Models Phase 2

The database was created using the original models we had designed for cities, activities and languages. Originally we had made each table be a separate app. We encountered a problem when implementing the database with three different apps, only a single app could be read and we would get an exception error for the other two apps. We unsuccessfully tried debugging the apps, including switching the order in which we read the apps but the error persisted. We then considered a different way to connect the database. We included the models in a file called content_display. This resolved the issue of the different apps not getting read. Then we realized the data tables were missing foreign keys, which would not allow us to relate information from multiple tables. As a result from missing foreign keys, we redesigned our models for cities, activities, and languages. Since django creates primary keys when not specified, we focused on including attributes to both activities and language models so we can reference which activity can be done and what language is spoken in a particular city. We had to edit our unit tests to match our new models. Editing the unit tests included removing test cases for attributes that were no longer existent and adding test cases for the new attributes that were added to our tables. We then made sure to include all the models for the tables in one file called content_display. When testing, we encountered a problem with theapps that were not deleted that were also copied into the content_display file. The models and tests we had for those apps were interfering with those in content_display so we solved the issue by deleting the apps that were outside the content_display file.

```
class CityTest(TestCase) :
    def setUp(self) :
        Cities.objects.create(name = 'Li Jiang', country = 'China', description = 'awesome', p
        Cities.objects.create( name = 'Chichen Itza', country = 'Mexico', description = 'aweso
        Cities.objects.create( name = 'Cape Town', country = 'Africa', description = 'awesome'
        Cities.objects.create()

    def test_c1(self) :
        ci = Cities.objects.get(name= 'Chichen Itza')
        self.assertEqual(ci.name, 'Chichen Itza')
        self.assertEqual(ci.country, 'Mexico')
        self.assertEqual(ci.description, 'awesome')
        self.assertEqual(ci.pictures, 'test.jpg')
        self.assertEqual(ci.videos, 'test.jpg')
        self.assertEqual(ci.coords, '0.0, 0.0')
        self.assertEqual(ci.size, '8,193 sq mi')
        self.assertEqual(ci.population,'1,244,769')
        self.assertEqual(ci.time_zone, 'central')
```

The picture above displays a section of our unit test file in which we created several classes to test cities, activities, and languages. Once the models were established, we then created test cities, languages, and activities in which we tested the attributes of each.

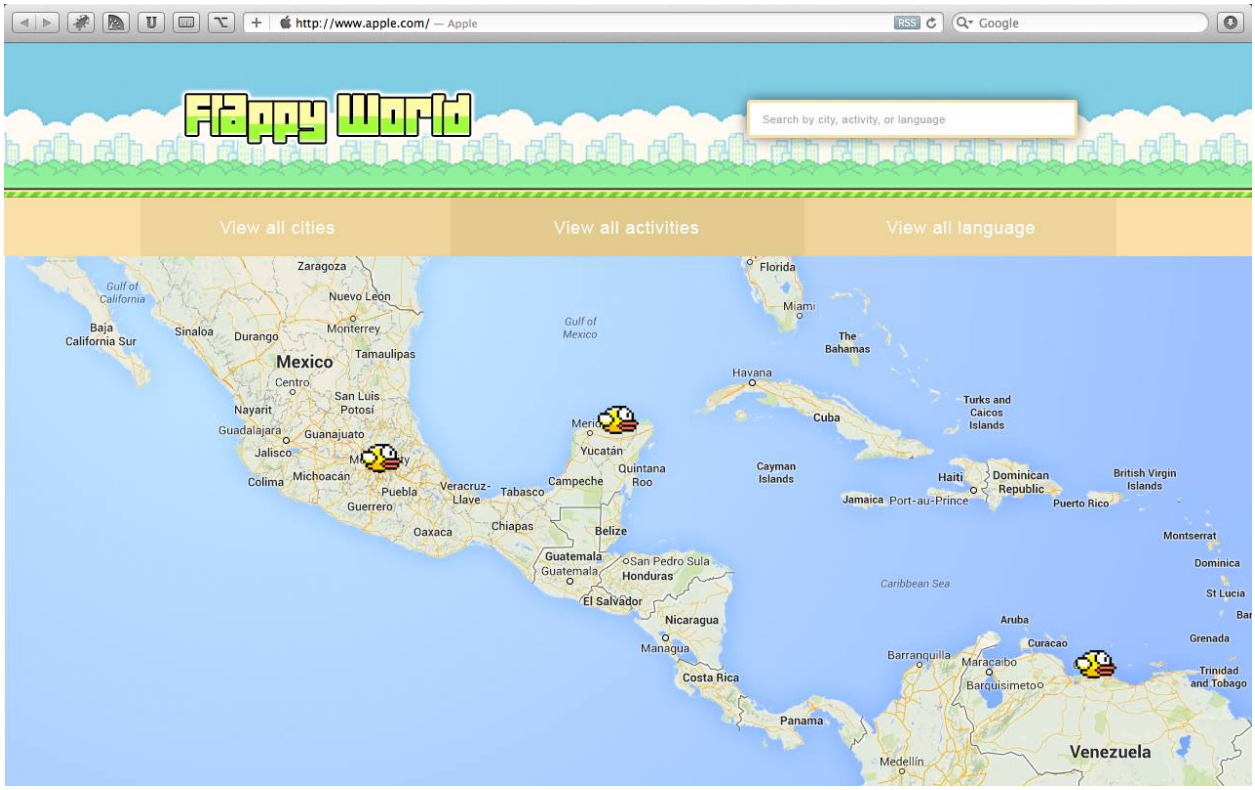
Front-End Design

Why Bootstrap?

While creating a web application that utilized many different technologies like previously stated, it was important to be able to display all the gathered data in a simple and appealing design for the user. Twitter Bootstrap is great tool for developing front-end design as it provides a wide variety of tools that contain from design templates for many interface components. It is also works on a grid system in which it divides the screen equally into space that can be resized and moved around to fit different sizes of screens automatically. Bootstrap speeds up the development of the front-end as many of the members had no prior experience with either HTML or CSS were able to pick it up quickly.

Design

For our design and usage of bootstrap we began by first creating a mockup of what we wanted the application to look like .One consistent feature that is displayed at the top of application is the navbar. The navbar is a simple template available through bootstrap that is very useful to simplified the navigation through the application.



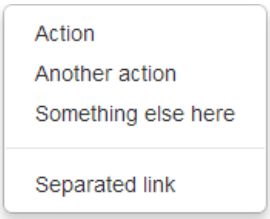
Example

Title Home Link Link

```
<div class="navbar">
  <div class="navbar-inner">
    <a class="brand" href="#">Title</a>
    <ul class="nav">
      <li class="active"><a href="#">Home</a></li>
      <li><a href="#">Link</a></li>
      <li><a href="#">Link</a></li>
    </ul>
  </div>
</div>
```

This a simple example of HTML code combined with the Bootstrap classes to create a navbar.

Within the navbar we replaced the regular buttons with drop down menus as the appearance of a sleek simplified designed appealed to us. For the drop down menus the bootstrap provided class drop down was use. This allowed to display all the information hidden and keep the focus on the tourist destinations on our map.

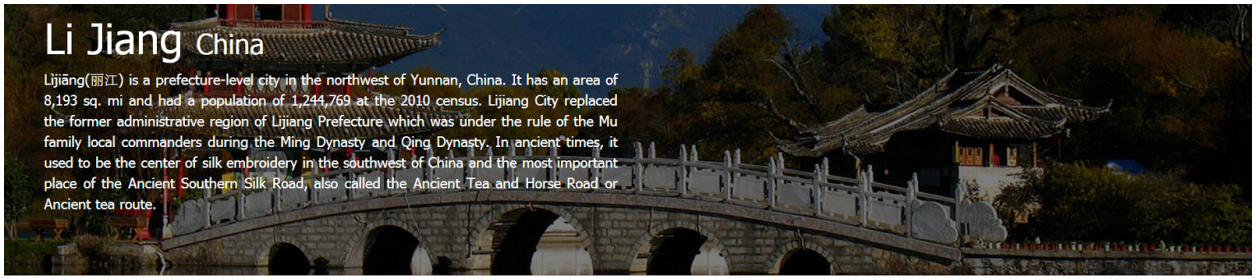



```
<ul class="dropdown-menu" role="menu" aria-labelledby="dropdownMenu">
  <li><a tabindex="-1" href="#">Action</a></li>
  <li><a tabindex="-1" href="#">Another action</a></li>
  <li><a tabindex="-1" href="#">Something else here</a></li>
  <li class="divider"></li>
  <li><a tabindex="-1" href="#">Separated link</a></li>
</ul>
```

One of the great features in bootstrap that help the application be responsive is the grid system that is available. The grid system works on a 12 point system, so the screen is divided to a maximum of 12 columns. With bootstrap you can manipulate the

columns and combine them with others to make larger columns as such.

In our application we use to divide sections of the screen and as the screen is resized or opened up in different size screen it adjust.

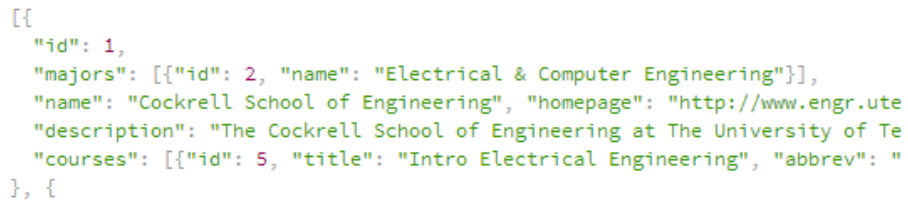


This section is divided into two columns for the page with the following code Using the class **col-(size)-(amount of columns)** it is possible select what screen size we aiming for, and columns how many columns there are per row in the grid. Tools like this help with organizing the feel of the application.

Using Git-a-Grep API

At first when the thought occurred to us about having to another group API it was kind of daunting considering how different it could be from what we had designed. However till our pleasant surprise it was very similar to our own design. Since their output when their

API was called was identical to ours. Here is an example of call to their API. As seen it returns a JSON with pretty similar to our own formatted JSON.



So to call the API we had to add another url to url.py to reach it. Then in our views.py is where most of the work was done. First we need to **import JSON** and **urllib.** There is a urllib2 and urllib3 that are used, but for some strange reason they are not compatible with django, but urllib was all that was necessary.The entirety of of the process to pass all the API information is just 4 lines.

```
template = loader.get_template('ut_api.html')
response = urllib.request.urlopen('https://gitagrep.pythonanywhere.com/rest/colleges/')
html = response.read().decode("utf-8")
content = json.loads(html)
```

We load our template page then request the api from Git-A-Grep with functions from urllib and we read the response while decoding using utf-8 because it will give you an error otherwise if not decoded since urllib doesn't do it automatically like its future iterations. Finally just like in the Netflix project, we load the JSON formatted string through json.loads() which turns it into a python list of dictionaries and we place it in the context and send it to the template.

To use it on our website we decided that since our index page consist of a map with tourist locations and Git-A-Grep info was about The University of Texas at Austin we placed an icon to serve as a direct link to

the info. Of course instead of the FlappyBird icon we gave it a nice Longhorn.

Displaying the information on the page itself was very simple as we had a lot of previous experience with it. Using Bootstrap again making it fluid was very simple.

Seach

Quering

For our search, we used the django urls to pass in a search query to our views where we build an sql query to feed the database to search for word matches. We opted for this method since we can prevent the user from typing single and double quotes on the url's regex patterns to prevent sql injection. Thus, we know we can use the query that does make it to the views. We also know how to get direct access to the database and look for content that does not exactly match the entries. In order to do this, we used the LIKE sql operator. For example, we can build a query as follows:

```
5
6   def search(query=""):
7       if(query!=""):
8           sql = "SELECT * FROM content_display_cities WHERE description LIKE '%"+query+"%'"
9
```

The '%' would help us wrap the query around any other content that does not exactly match the query. So, if our query was "Cape Town" our database could match the query to the entries containing "the city of Cape Town is beautiful," or "cape town." or just "Cape Town". We search not only in the description field, but also in the name field, and in the case of activities we search in the type_activity field.

And/Or

The previous search procedure is the procedure we use for the “and” search. For the “or” search, we pretty much split que search query by words and did a similar procedure to get results that contained any of those words. We ended up with queries like the following: ie/ if i search for “city sydney” this is what the sql query would look like:

```
SELECT id,name,description
FROM content_display_cities
WHERE  description LIKE '%city%'
    OR description LIKE '%sydney%';
```

We are also preventing fetching duplicates by adding the “distinct” keyword on the sql query after “select.”