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

Phase I Report

Introduction

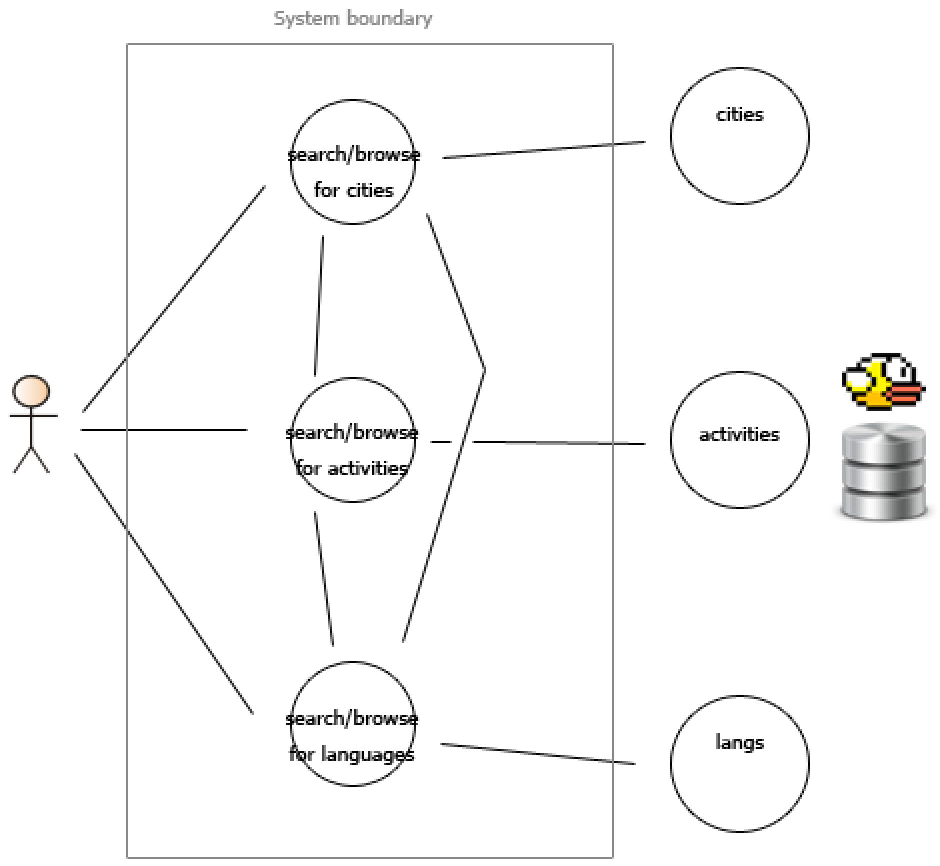
After discussing several ideas, we settled on developing a webapp that stores, relates, and displays data on tourism destinations based on corresponding cities, activities, and languages. We decided to name the webapp “Flappy World” due to our initial interest for this project in indie video games and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

}]
```

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", ... }
```

`/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", ... }
```

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", ...

}, ...]
```

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", ... }
```

`/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", ... }
```

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
```

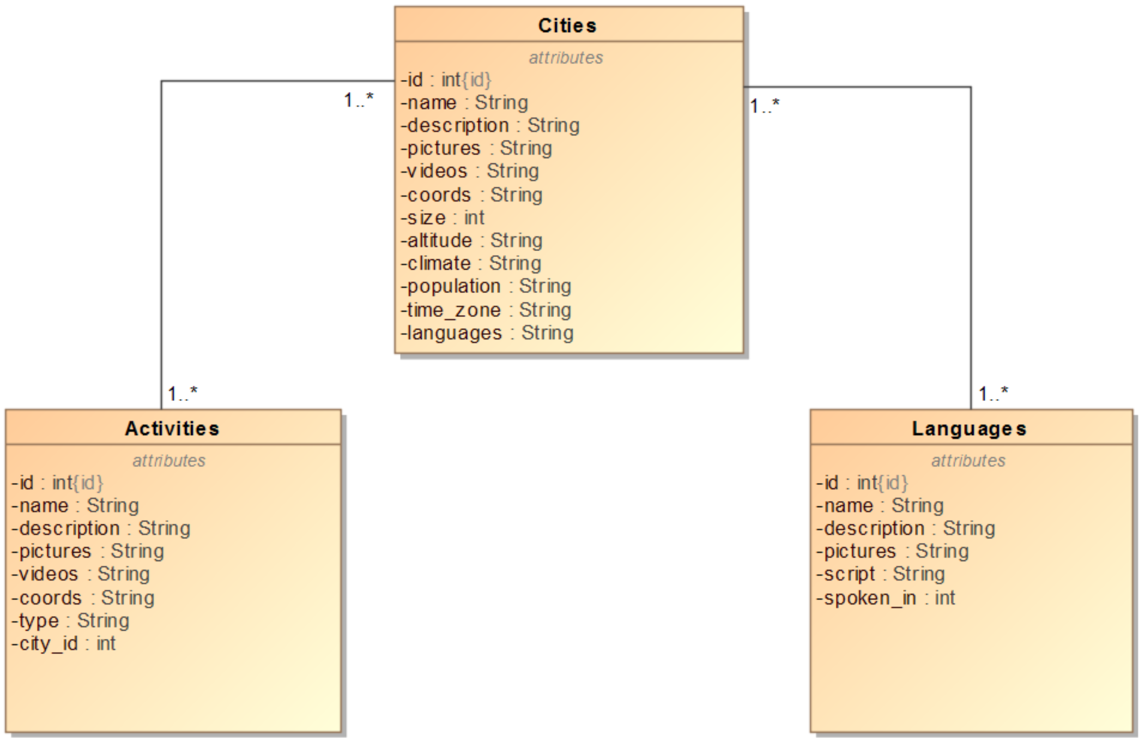
Response

```
200 (OK)

Content-Type: application/json

X-My-Header: The Value

{ "id": 1, "name": "Some activity", "description": "some text", "type":

"Scenery", ... }
```

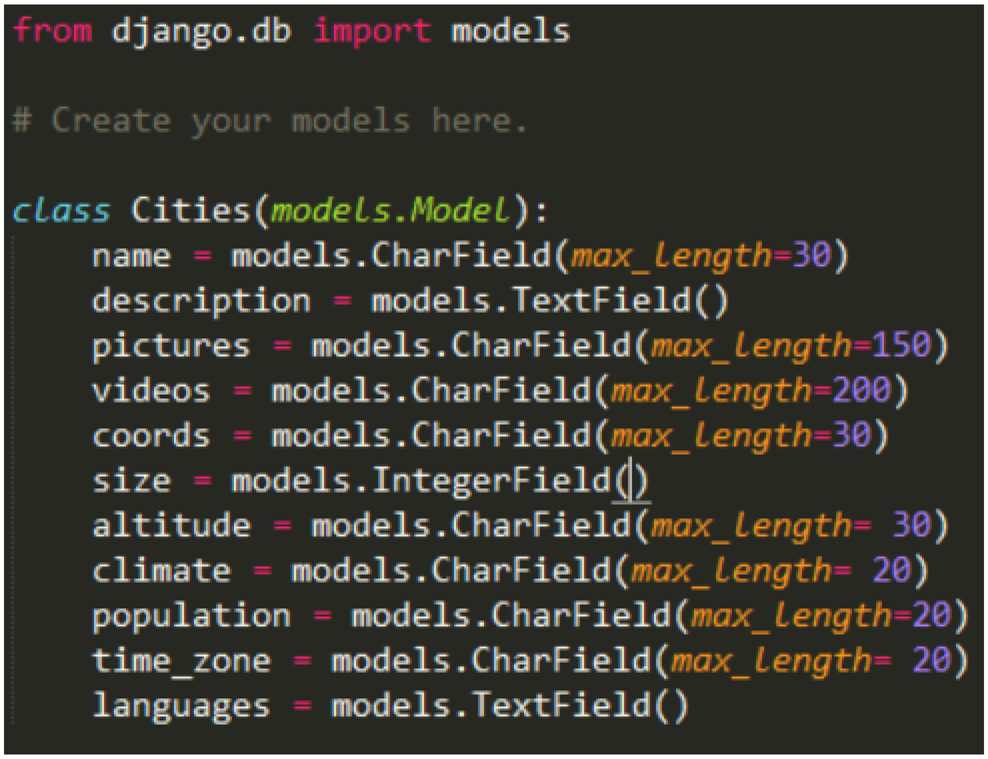
Django Models

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 three apps one for Cities, Activities, and Languages and inside those apps we have its corresponding model because django app models can be imported. By splitting the into three apps instead of just one big app we can distribute the functionality. 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 and in the Languages table the same is true for the attribute spoken_in. The relationships can be read as one or many cities can have many one or many activities and one or many activities can belong to one or many cities. The other relationship is 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.



```python
from django.db import models

# Create your models here.

class Cities(models.Model):
    name = models.CharField(max_length=30)
    description = models.TextField()
    pictures = models.CharField(max_length=150)
    videos = models.CharField(max_length=200)
    coords = models.CharField(max_length=30)
    size = models.IntegerField()
    altitude = models.CharField(max_length= 30)
    climate = models.CharField(max_length= 20)
    population = models.CharField(max_length=20)
    time_zone = models.CharField(max_length= 20)
    languages = models.TextField()
```

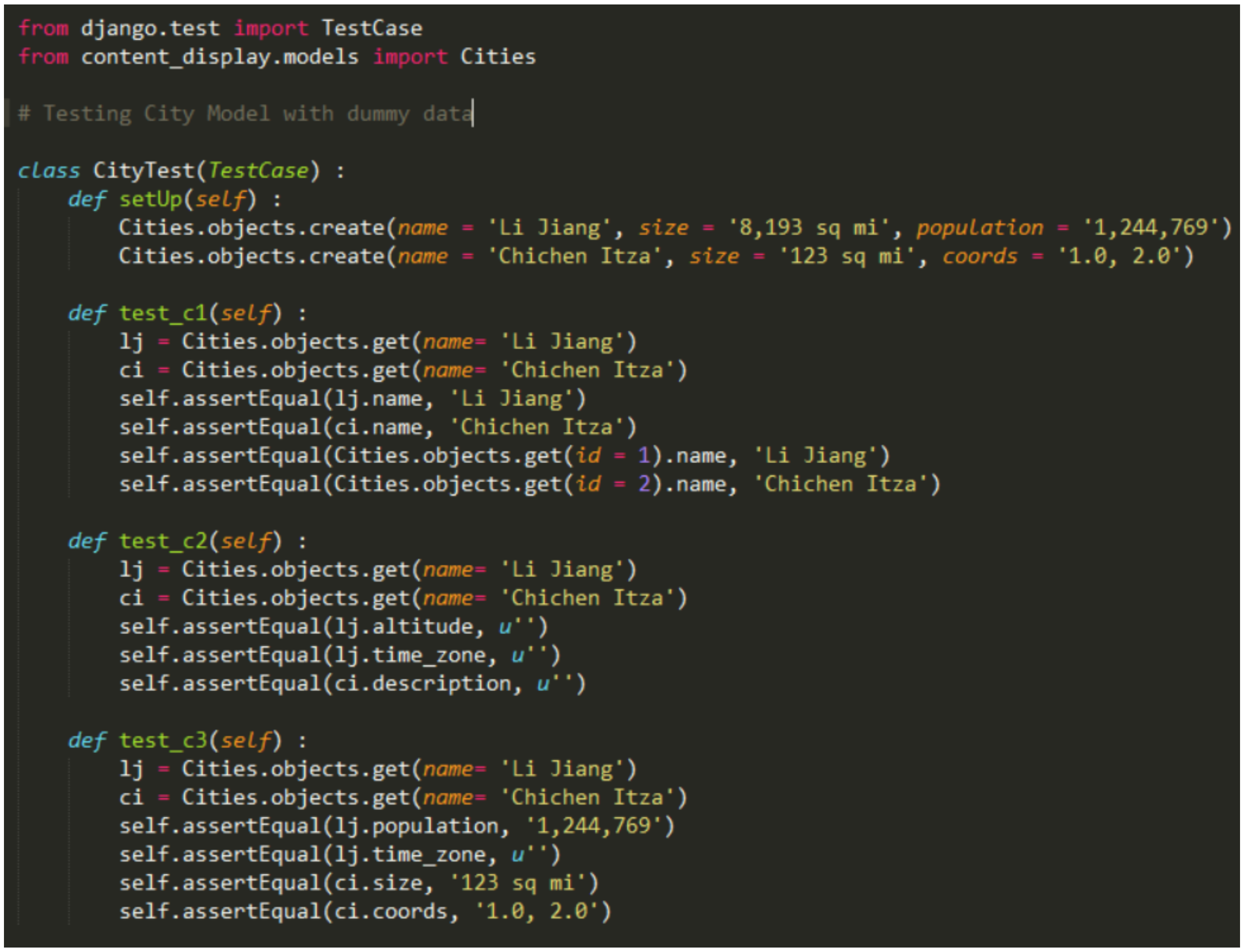
As you can see it follows almost the same format as the UML diagram above. For each attribute we defined how we wanted it to be stored in the database. We mostly stored charFields which are strings. The only big difference from this models and the UML is that in the Django model we did not make an attribute 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.

Once we made all the models inside the specific apps we started creating tests that would check if our nonexistent database would actually work. Even though we have made the models we have not yet created the database. This leads us to the test creation phase.

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.



```
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.

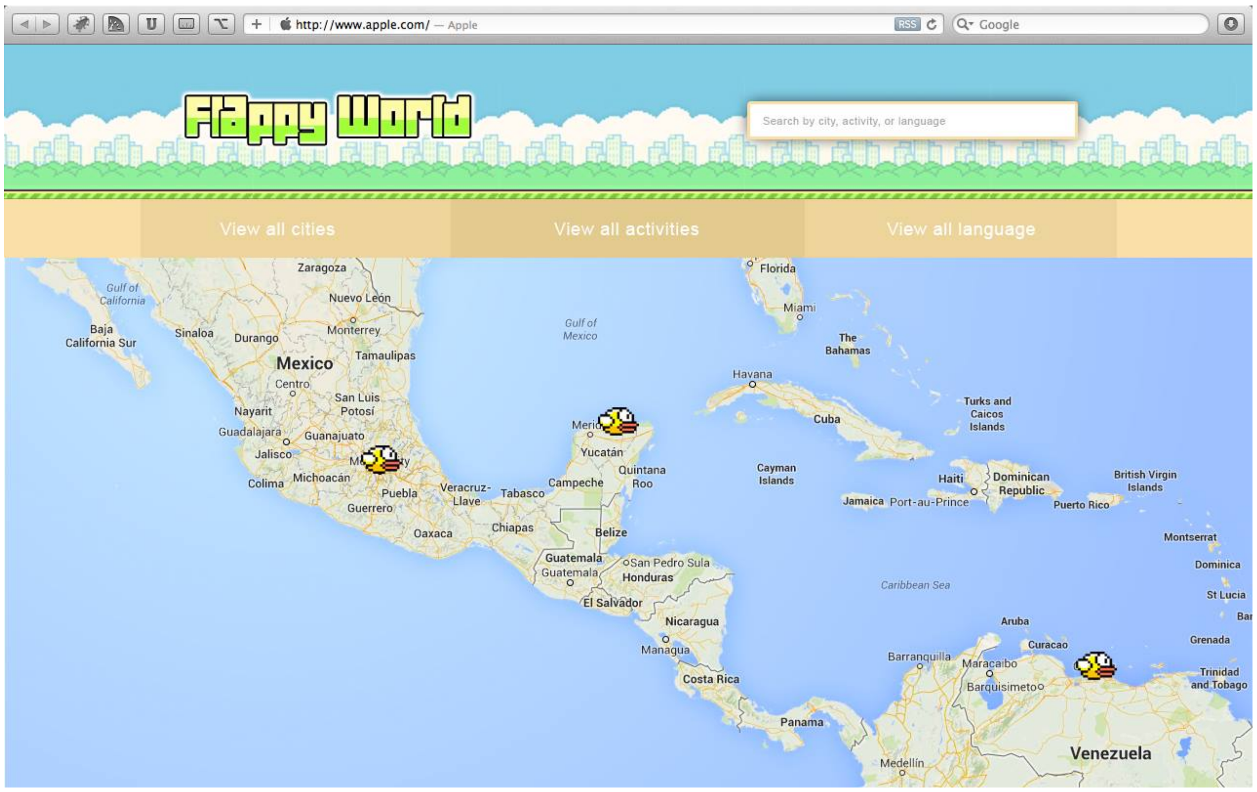
Front-End Design

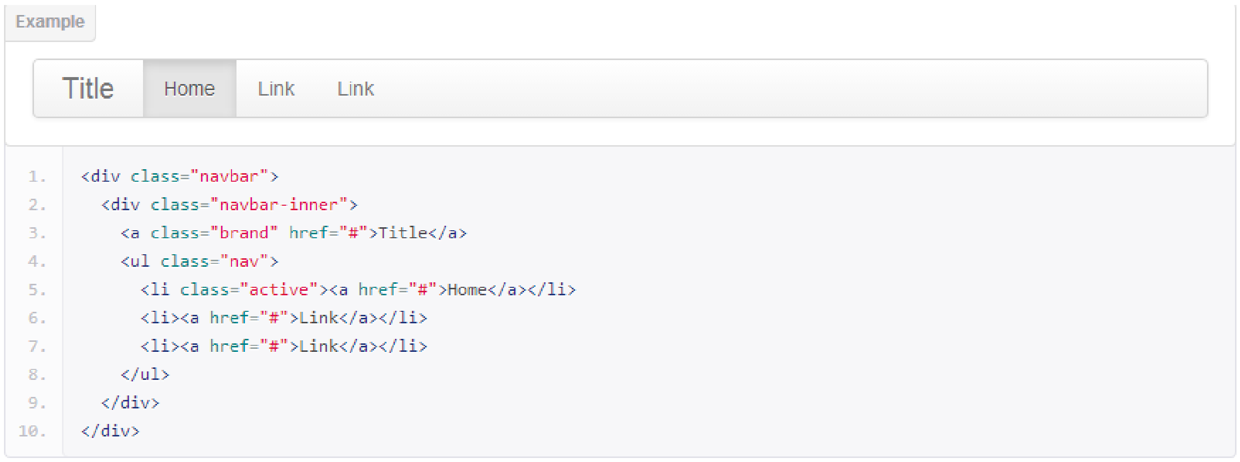
Why Bootstrap?

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.

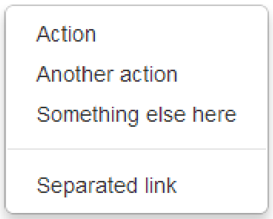


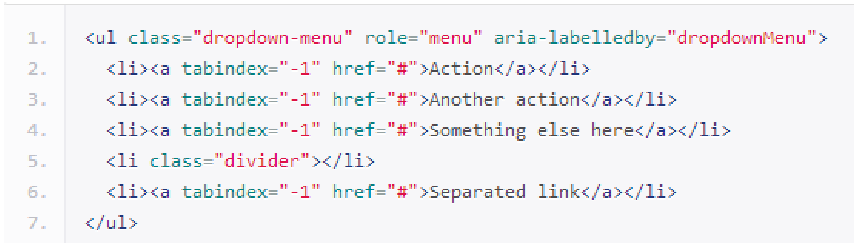


```
<div class="navbar">
  <div class="navbar-inner">
    <a class="brand" href="#">Title</a>
    <ul class="nav">
      <li class="active"><a href="#">Home</a></li>
      <li><a href="#">Link</a></li>
      <li><a href="#">Link</a></li>
    </ul>
  </div>
</div>
```

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

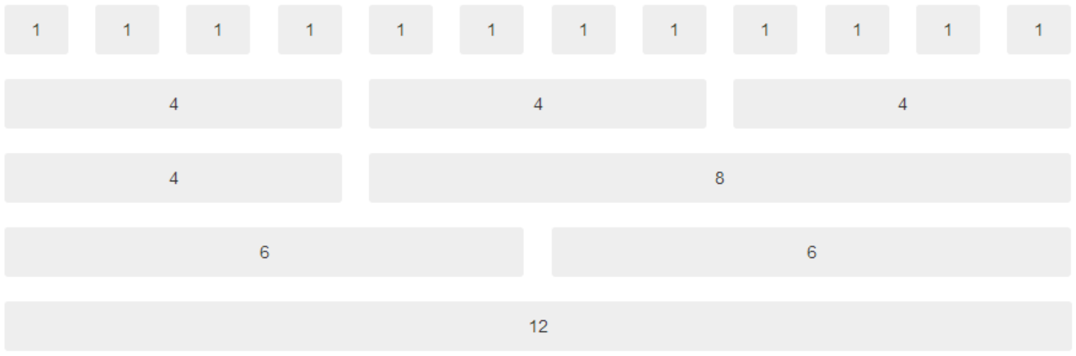
Within the navbar we replaced the regular buttons with drop down menus as the appearance of a sleek simplified designed appealed to us. For the drop down menus the bootstrap provided class drop down was use. This allowed to display all the information hidden and keep the focus on the tourist destinations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



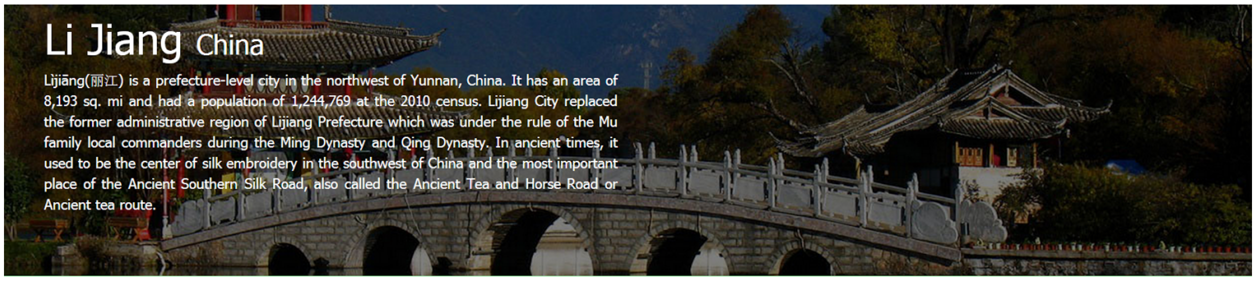


```html
<ul class="dropdown-menu" role="menu" aria-labelledby="dropdownMenu">
  <li><a tabindex="-1" href="#">Action</a></li>
  <li><a tabindex="-1" href="#">Another action</a></li>
  <li><a tabindex="-1" href="#">Something else here</a></li>
  <li class="divider"></li>
  <li><a tabindex="-1" href="#">Separated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



Li Jiang China

Lijiāng(丽江) is a prefecture-level city in the northwest of Yunnan, China. It has an area of 8,193 sq. mi and had a population of 1,244,769 at the 2010 census. Lijiang City replaced the former administrative region of Lijiang Prefecture which was under the rule of the Mu family local commanders during the Ming Dynasty and Qing Dynasty. In ancient times, it used to be the center of silk embroidery in the southwest of China and the most important place of the Ancient Southern Silk Road, also called the Ancient Tea and Horse Road or Ancient tea route.

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.