Twistory

Bryan Salas, Boomer Truong, Terrence Maas, Wu-Yang Tao, Young Seo, Nick DeChant

Introduction

What is the problem?

In a broad sense, practical information for visiting national parks is not well presented. If you have ever been to the National Park Service's website (nps.gov), you have noticed that the information on each individual park's page is not presented in the same way. Sure, the architecture of each web page is the same, but why would it ever not be: it was created by the same developer(s). The content of the pages, however, is another matter entirely.

Back to the park page: on the left side of the page there is a navigation menu, and on this navigation menu there is a button that will ostensibly help you "Plan Your Visit." Depending on which park's page you happen to be viewing, you might be greeted by a short page with links to more specific pages, a long page with videos and a short history of the park (alongside similar links to specific pages), or even a page with a lonely description unaccompanied by any links whatsoever. The madness does not stop there.

Being on the "Plan Your Visit" page will open up a submenu on the left-side navigation bar. Clicking on the "Things To Do" button directs you to another highly-non-standardized page. For Yellowstone National Park, the page is a (relatively) beautiful one, concise yet informative and with many helpful links. For Crater Lake National Park, the page has no links. Instead, it displays a body of text that could best be described as "knowledgeable stranger happens to cross your path and gives you an incredibly broad

run-down of what the park offers and how to get there, before walking off to wherever he was headed." This, as you can imagine, turns finding information about hikes and trails into a small nightmare. In some cases, one must actually search through pdf files of the park's various brochures to find the right answers.

I could go on, but I think you get the idea. In short, finding practical information, such as a trail guide or camping information, is not as simple as it could, and should, be.

What are the use cases?

Our website's principal use, as hinted at above, is helping people find practical information about national parks in a timely manner. While the information we display is already available online, the ease of access and organizational simplicity of our site are enormous improvements over the NPS's own website.

Time matters. Even seconds matter. Saving someone as little as a few seconds of time and effort, especially on a task that they will repeat several times, is historically proven to be almost revolutionary. Television remote controls, dial pads on phones, and online movie rental services are all ubiquitous. Furthermore, their predecessors (buttons on the televisions themselves, rotary dials, and video rental stores) all seem archaic by comparison, despite the fact that they accomplish precisely the same thing. The difference is time and effort. The few extra minutes that it will take someone to make a decision about their national park visit is worth just as much to them as the few extra minutes it takes to visit a video rental store.

Twistory API Design

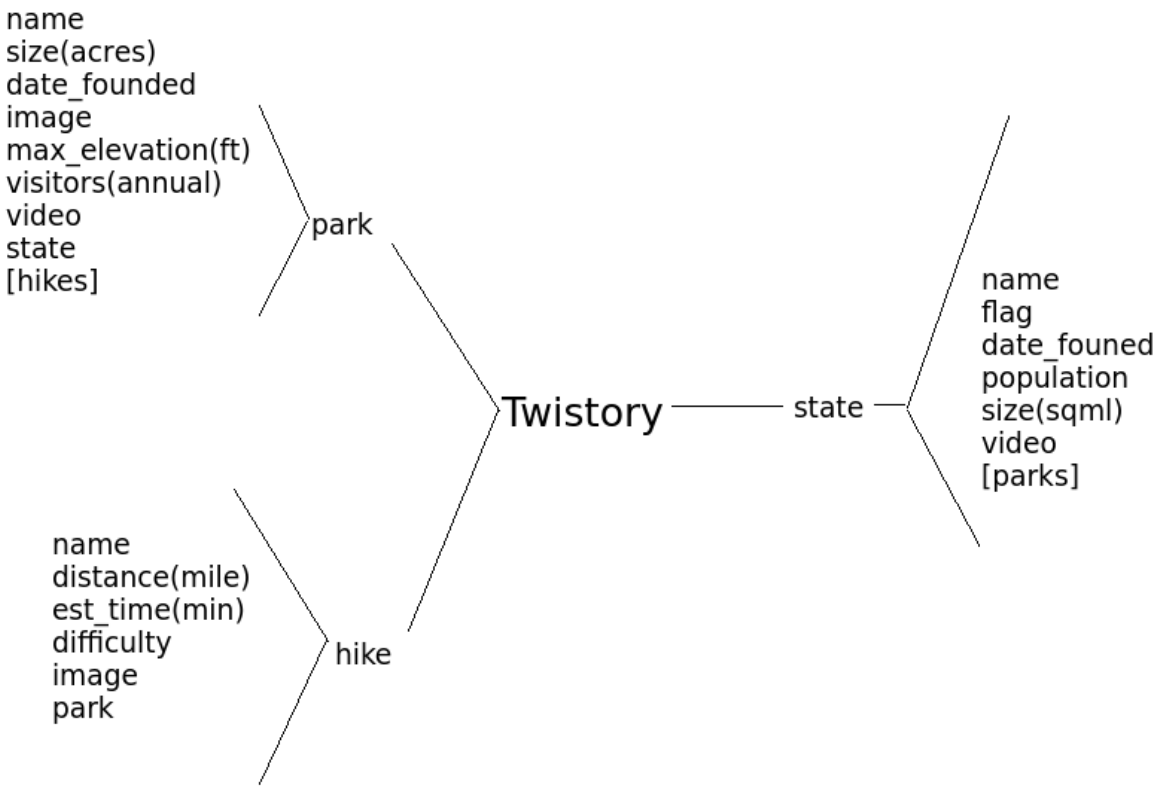
To adhere with the principles of REST, the general syntax for the resource URL should be undercase and plural, so if the API contained a resource for "dogs" the URI path would be "/dogs" and corgis resources would be "/dogs/corgis". If a resource identifier should span longer than one word, a single underscore will be the delimiter to group the words together, "/ugly_cats" or "/dumb_kittens". The main goal on deciding a style was to keep resource URLS consistent for our users and future developers working on this API. With all of the Twistory resource paths, when a HTTP Request is made, there should be a version code with the resource path following it. The version code will help identify changes in the API and increase if there is a change to the API that could potentially give the users a lot of headaches. So if the developers decide to deprecate a resource the users would have to upgrade accordingly.

The next important part of the Twistory API is that the documentation is regularly kept up to date, detailing changes to a resource or documenting new resources, so users are able to take advantage of the API. REST explains that resources should be a noun, as the HTTP Requests describes the verb or action upon the noun: "GET /dogs/corgies". The Twistory Docs for any resource all follow a general outline by starting out with a description of the resource and its benefits. Next should describe a single instance or your resource and what the end point would look like,"Dogs [/dogs/id]", and then describe the endpoint to retrieve a collections of those resources. This format clearly shows how a user can easily request a single object or a collection of them and making the API more user friendly.

Lastly, if the resources or collections have parameters like an id, there should be a row that includes the parameter, a description of it, whether this parameter is optional or required, and an example of how the request would look.

Assuming a successful request was made, The Twistory Docs shows the formats response model with a header and a body. Currently The Twistory API will respond in JSON format as the content-type. The Header of the response will contain metadata about the resource, i.e. date last updated, language, character encoding scheme. The body will be the resources and the attributes that were decided upon. Finally, handling errors or bad requests is done by choosing a numerical value within the HTTP status codes which identifies what the error was, as opposed to a 404 page that would be extremely difficult to debug.

In order to access the api, the user needs to add /api/ after http://twistory.pythonanywhere.com/, follows by the 3 main resources we define, which are states, parks and hikes. For example, if wanting to access the api for states, then the url will be http://twistory.pythonanywhere.com/api/states/. The mind map below shows all the attributes of each resource:



There are several attributes that can contain multiple elements, namely those with square brackets around in the mind map, such as the attribute "parks" in states and "hikes" in parks. In our implementation, we decided to make each of those attributes holds a dictionary. The keys in these dictionaries are the respective foreign keys, for example, the attribute “parks” in the resource states has the park name as the key, and the link to its respective api page as the value.

Since we decide to make the api read-only, the user can only use the GET method.There are two type of get methods. They both return json files. The first type will return a list of all elements in the indicated resource. This type will only give a 200 response since even if there is no element in a list, an empty list is still a valid response. The key for this json is the name of the resource, such as the name of state or the name of park. And the value is just a string stating the path to its deatiled page, which can be returned by the second GET method. The second GET method returned all the info of a resource object as indicated by the user. For example, if the user wants to have more info on the state Ohio, the url will be http://twistory.pythonanywhere.com/api/states/Ohio/ . If the user gives an invalid name, then this method will return a 404 response with the error message saying this name does not exist in the file.

The unit tests for the API are made for the following purposes:

- def test_api_state_1 is to see if the 404 response can be properly returned will the user enters an invalid name for state.
- def test_api_state_2 checks that the list of states can be returned correctly. Note that it is an empty list since the Client class used to perform can not get access to the database we have.
- def test_api_state_3 makes sure that even if the final ‘/’ is omitted in the url, it can still be directed to the right page and give response 301.

Similar tests are written for parks and hikes. Note that Client class can not connected with our database, so tests on obtaining info on individual resource object can not be done. But by observing the website, the 2nd GET method does perform successfully.

Use of API of the Other Group

The page "Hungry?" exercises the API of the website from the group "Witty CS Pun". Their website contains information about famous food, which includes recipes, chefs and regions of food. We decide that the focus of our use of their API will be on presenting all the food on their website and the instruction on how to make each food. All the info we used can be found on the recipe site of their website. Their API allows us to obtain a json file that contains all the info of their recipe site. However, their json file is an array of dictionaries instead of just a single dictionary. Also, most of the significant info is contained in a separate dictionary within each dictionary in the array, hence it is important to distinguish the right dictionary to use in different parts of our code. From that json file, we obtain the name and image of each food and display them as gallery on our page. The cooking instruction is also extracted from the json. In order to see it, the users need to click on the name of the food they want to see, then a dialog model which shows the cooking instruction will pop up. To close the modal, the user just needs to click on anywhere outside the modal.

The purpose of this page is not only to provide dynamic content from another team’s website, but also to integrate more useful information for our users. Visiting a park and hiking on a trail can be tiring, so there is potential use for our users to gain value from the recipes on our website.

More technically we injected HTML class code for each individual recipe. The HTML utilized Bootstrap and its jQuery modules to build the modals. First, the json

information was extracted and then inserted into sections of the HTML to property render the right image. Next we took the cooking instructions from the other team's json and inserted it into the body section of the modal class for that recipe.

Django Models

We designed our model to represent the three categories - State, Park and Hike. State refers to the states in the United States. Park refers to a national park that is in these states. Hike refers to a hike of a trail that is in the national parks. Each category represents a class which is defined by the attributes that are within that class.

State

The State class has the attributes name, date_founded, flag, population, size, and video:

- The name attribute is the string of the name of the State and it is the primary key for the state.
- The date_founded attribute is the string of the date the state was founded in the format “00/00/0000”. It is the date of when the state was admitted into the Union.
- The flag attribute holds the url of the corresponding flag of that state.
- The population attribute is the population of the state.
- The size attribute is the area of the state in square mile.
- The video attribute holds the url of the corresponding video of that state.

Park

The Park class has the attributes state, name, size, max_elevation, date_founded, park_image, num_visitors, and video:

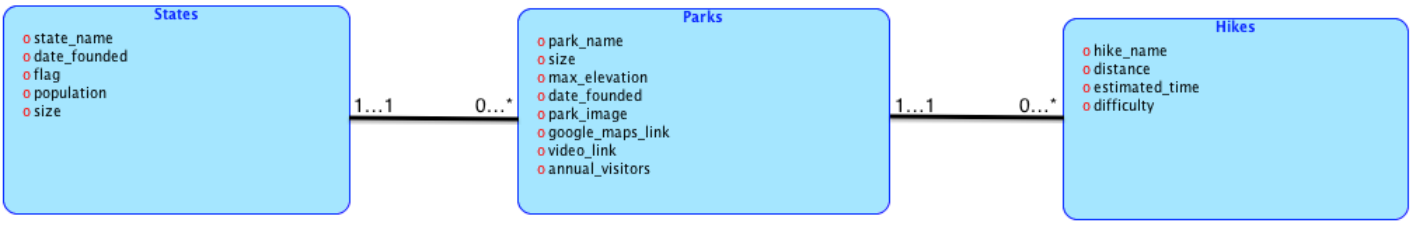
- The state attribute is the state that the park belongs in. It is a foreign key that references the state object that has the same name.
- The name attribute is the string of the name of the park and it is the primary key for the park.
- The size attribute is the size of the park in acres.
- The max_elevation attribute is the tallest peak of the park measured in feet.
- The date_founded attribute is the string of the date the park was founded in the format “00/00/0000”.
- The park_image attribute holds the url of the picture of the corresponding park. The num_visitors attribute is the number of visitors that come to the park annually. Most of our num_visitors data is from the year 2011.
- The video attribute holds the url of the video of the corresponding park.

Hike

The Hike class has the attributes name, distance, est_time, hike_image, difficulty, and park:

- The name attribute is the name of the trail of the hike and it is the primary key for the hike.
- The distance attribute is the distance of the trail in miles.
- The est_time attribute is the number of minutes it would take an average person to finish the hike.
- The hike_image attribute holds the url of the picture of the corresponding trail.
- The difficulty attribute gives the level of difficulty of the actual hike. There are three levels of difficulty - easy, moderate, and strenuous.
- The park attribute is the park that the hike is in, and it is the foreign key that references the park object that has the same name.

These models were then expanded to a relational model which can be viewed below. It was important for us to map out exactly what each model did and mapped to in order to better visualize our design and optimize it:



Unit Tests of Django Models

For this project we are using the built in Django SQLite database. With this in mind we wanted to write effective unit tests so that our website will be as successfully functional as possible. We chose to build and tear down a temporary database with new values to avoid risking harming our database with our production values stored in it. The following is a technical explanation for our unit tests:

- def test_state1 :

This test creates a temporary State object in the database and checks to make sure the type of the state is actually State as well as the name and date_founded attributes of the state are correct.

- def test_state2 :

This test creates a temporary State object in the database and checks to make sure the population attribute is greater than 10 and that the size attribute is exactly 10 as well as making sure that the type of the state is actually type State. This is an important test to make sure the integer fields are saved in the database properly.

- def test_state3 :

This test creates a temporary State object in the database and checks to make sure that the population equals a particular negative value and that the size is equal to 0. These tests are important because the values may not always be positive and we want to be able to store all values properly in the database. Additionally it checks to make sure the type of the object is actually State. We also check the CharField attributes in the State model to make sure those values are being stored and returned properly.

- def test_park1 :

This test creates both a State and a Park object with defined attributes. This test checks to make sure the type of the park object is actually Park and that the state object stored in the Park object as the Foreign Key is of type State. This test is important because we must make sure our Foreign Key values in the database are being stored properly.

- def test_park2 :

This test creates both a State and a Park object with defined attributes. This test checks to make sure the integer values of both the Park and State objects are correctly stored by comparing them to pre-determined values. This test is important because we must ensure that our integer values are being stored in the database properly.

- def test_park3 :

This test creates both a State and Park object with defined attributes. This test checks to make sure the type of the State object stored as the state attribute in the Park object is actually of type State. It also checks the video attributes of both the State and Park objects. This test is important because we must have the correct video URLs stored in our database to render our videos correctly.

- def test_hike1 :

This test creates State, Park, and Hike objects properly nested with each individual's Foreign Keys. This test checks the type of the Hike object to make sure it is actually of type Hike. It also checks the CharField values of the Hike object to make sure they match pre-determined values. This test is important because we want the strings stored in our Hike rows to be correct.

- def test_hike2 :

This test creates State, Park, and Hike objects that are properly nested according to their Foreign Keys. This test checks to make sure that the park attribute of the Hike

model is actually an object of type Park. We also check to make sure the State attribute of the Park object is actually of type State. This test is important because we must ensure that we can access the Park and State objects associated with each Hike object properly.

- def test_hike3 :

This test creates State, Park, and Hike objects that are properly nested according to their Foreign Keys. This test checks to make sure that the distance attribute of the Hike object is less than the estimate time to complete attribute of the Hike object. This was done with pre-determined values. Additionally this test makes sure the difficulty attribute saves the correct string even though the string contained integers. Finally we check to make sure that the names of all the objects do not actually equal each other.

Database

For our website it was important to store our data in a database, so that we could dynamically populate and make interesting webpages. In this case we decided to use the built-in Django SQLlite database. This was convenient because it allowed us to enter our data in a familiar object-oriented-Pythonic fashion. Our website templates could then be set up with Django to create a potentially infinite number of pages, depending on what type of data is in the database. All of the data in the database came from the National Parks Service website (nps.gov). Here is a technical explanation of how we set up and entered our data in the database:

- We first created our database using manage.py and then syncdb.

- We then set up our database by creating objects from our data models.

  example:

  myState = State(name = “Arizona”, date_founded = “date”, etc…)

- Next, we would create the parks associated with each state, as they require a State object to be stored in each park’s state attribute.

  example:

  myPark = Park(name = “Grand Canyon”, state = myState, etc…)

- Then, we continued following our UML model by creating a Hike object after the Park object, as each Hike stores a Park object in it.

  example:

  myHike = Hike(name = “Red Mountain Trail”, park = myPark, ect…)

- We then called .save() on these variables to save them in our database.

Search Capability

When adding a search function, our first priority was to have a nice looking search bar that would be easy for users to use. We ended up choosing the default search bar that Twitter Bootstrap provides since it is basic and looks clean on our navigation bar. To make the searching easier to implement we used Haystack (a module for Django). Haystack can not run alone, but needs a backend to host indexes - we used Whoosh for this backend. Haystack has a few files we had to change/add to begin running search queries:

- urls.py
    - We had to update urls.py so that when a user sent a query using our seach bar, Django would know which view function to run. We used the GET method to send the query to our custom views.py function. Originally we had used the built in SearchView that Haystack provides but this class would not perform the special types of searches that we wanted available (see section on views.py).
- search_indexes.py
    - This file acts as a blueprint for Haystack to tell Whoosh which data from our models to actually store and index. This file has a very similar set up to models.py except the naming scheme of the fields are a little different. We set up our name indexes with EdgeNgrams to allow for partial searches (example: if a user seaches for ‘tex’, the search returns ‘Texas’ as a result).
- <model_name>_text.txt

    - Each model has to have its own .txt file (example: state_text.txt). This file contains variables in the same format as Django template variables. ({{ <variable_name> }}). The variables in this file tell Whoosh which attributes the user is actually going to be able to search for when running a query.

- views.py
    - We updated views.py with our own custom view function using some of Haystack's built in features, such as the SeachQuerySet. We wrote this function to be able to interpret multi-word search queries as either a <word> AND <word> search or a <word> OR <word> search (note that search queries can have more than two words in them). The back end architecture that makes this work involves running a query on each word and then finding the intersection (AND) or the union (OR) of the set of results. We display both sets of results on the search.html page.
- search.html
    - This is the template for the page of search results. The Search function in views.py sends this template a dictionary of two sets called and_results and or_results. These sets contain SearchResult objects (results of the from the search query). This template has logic in it that iterates through each set and displays some content about the object (example: for state objects we display the name, flag, date founded, population and the size).

Design Choices

-Goals

We intended our website to fulfill the following criteria, some of which seemed contradictory. Specifically, we wanted our website to:

1) feel lightweight yet be richly textured

2) be modern and directed at a young demographic

3) display the allure of nature.

-Implementation

Our site can appropriately be grouped into the following: the homepage, the list pages, the single-item pages, and the about page. This list does not include the page where we implemented another group's API. Additionally, all our pages are served by the same header, which contains the navigation bar and search field, and footer, which gives our citations.

Our homepage consists of a background image, a welcome message, and an interactive sample interface. Since we want our website to emphasize natural beauty, the background of the homepage plays a very important role in setting the mood of our site. To accomplish this, our image has a high resolution, warm colors, and a road leading away from the perspective of the viewer, which serves to introduce the idea of actively going out into nature rather than merely showcasing a scenic landscape. Both the welcome message and sample interface are partially transparent, which serves to both give the page a lightweight feel and further emphasize the background. Our welcome message is

simple, enthusiastic, and encouraging, which we feel resonates with modern viewers while again suggesting the idea of actively going out. The sample interface is positioned and colored to mesh well with the sky and clouds behind it and also to let the background landscape remain the focus of the page. The style of the interface itself is like the welcome message: casual, simple, modern, and lightweight.

Our list pages are where we display our full collections of states, parks, and hikes. Unlike the homepage and single-item pages, we want each individual item of the list to receive the viewer's attention. Therefore, the background is white and each item is represented with a picture instead of only text. The hover effect makes the currently-selected button slightly transparent, keeping with our theme of feeling lightweight.

Our single-item pages are where we display the information for individual objects in our database. To continue emphasizing the allure of nature, each individual page has a background image that is randomly selected from a collection of historic national park photos (in sepia, like old photographs) and other, higher-resolution photos. This background image is randomly selected each time the page is visited. The image is also partially transparent in order to not overwhelm the viewer with a the combination of imagery and information. The information for the individual object is spread across multiple panels which, when displayed against the faded background, give the page the simple and lightweight feel that is also present on our other pages.

The about page is separated into three sections. The first is a simple message stating the goal of our site and the names of our team members. The second gives a more detailed overview of how our responsibilities were divided. The third gives a summary of

the tools we used to create our site. Each section is accompanied by an image that serves as a monument to classic depictions of national parks and nature in general.