Marauder's Map

By The Six Potters:
Amanda Steinwedel, Angela Hsu, David Moench, Isabella Bhardwaj, Kaelin Hooper, and Stephen Arnett

Introduction

Problem

Understanding the nature and relationship of every object and being throughout the Harry Potter landscape can be a monstrous task to tackle. There is a vast array of characters, spells, potions, and wands. *Marauder's Map* aims to simplify this data-overload problem by explicitly defining nearly every aspect of Harry Potter in a more easily navigable, dynamic manner. Every creature and object is clearly defined in a concise way that allows the user to easily navigate from model to model (defined in Django), portraying not only details about that model but also the relationships it holds with the rest of the Harry Potter world.

Use Cases

In the event that a user needs to find specific information on a character, object, or relationship, Marauder's Map can provide this path. For instance, if a user needs the name of Harry Potter's parents, they can navigate to the character page for Harry Potter. This page will contain a list of relationships, and more specifically, his family.

If at this point the user also needs to know more specific information about Harry Potter's parents, such as when they died, the user can then click on his parents, which are hyperlinks, to navigate to their character pages. These pages, like Harry Potter's character page, will then contain their birth and death dates, as well as other information.

Other, more rare, use cases may include using Marauder's site source for screen scraping large sets of information within the Harry Potter world.

Design

Restful API

JSON Response Structure (Documentation at http://docs.maraudersmap.apiary.io/)

General Description

Most of our database's model have been made publicly accessible through our API. Generally any accessible model can provide the user with a collection of all instances, or a single instance. This is done through HTTP GET requests to /api/collection, or /api/collection/{id} respectively. All responses are JSON formatted dictionaries that the caller can parse and use in an application.

Model Relationships

A model instance's relationships with other models are encoded into the JSON response. Consider the following example:

```
/api/artifacts/{1}
{
    "id": 1,
    "description": "The Pensieve is an object used to review memories. It has the
appearance of a shallow stone basin, into which are carved runes and strange
symbols.",
    "owners": [6],
    "kind": null,
    "name": "Pensieve",
    "shop": null
  }
```

This artifact instance's "owners" attribute holds a list of the primary key IDs of the characters that it belongs/belonged to. This information allows the API user to make a request to /api/characters/6 if he or she wishes to get more detailed information on that character.

Optional Attributes

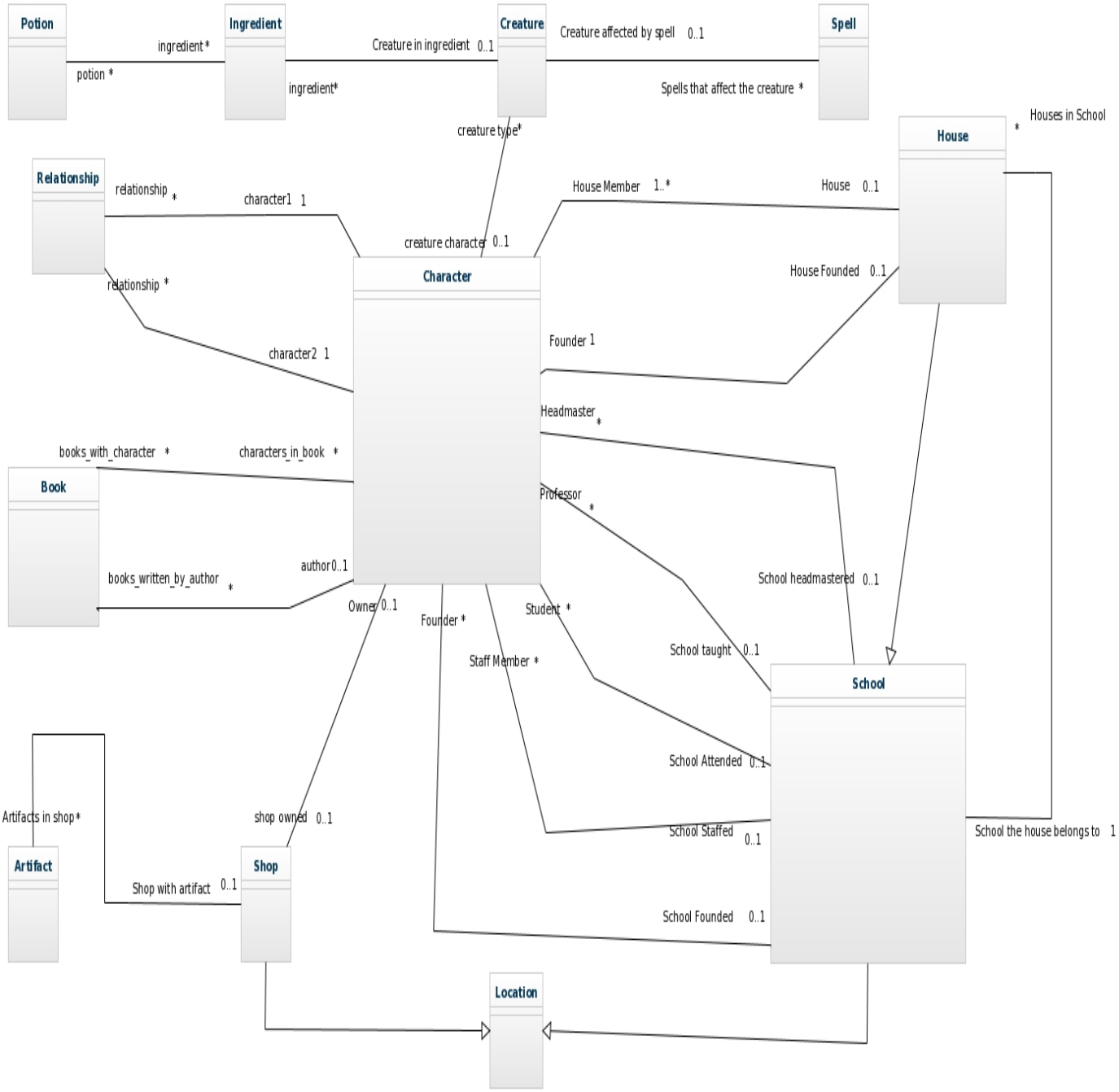
The keys of the JSON dictionary response will always be present for consistency, even for a model's optional attributes that may not be assigned a value. In this case the attribute's value will be null, so the user can easily check programmatically. For example, recall the above response. The pensieve can be found in no known shop, so its "shop" key is paired with the value null.

404 Response

If an API user make a request for a resource that does not exist in the database, our backend will serve them a HTTP response with the 404 status code and the following body JSON content:

{"error": "Sorry. That item doesn't exist."}

UML Class Diagram of Models:



There are 13 Django models total: Character, Creature, Spell, Ingredient, Potion, Location, School, House, Shop, Artifact, Book, Story and Relationship. These models represent

a network of interacting phenomena in the Harry Potter series. Between these models there exist one-to-one, many-to-one and many-to-many relationships.

PRIMARY MODELS

Character Model

The Character model can represent any character, magical or non-magical, human or non-human. This model has fields for the character's wand and memorable quotes. We save who said the quote separately, as a quote may be said by the character or about the character, for formatting blockquotes in the views. We also save an image of the character. There is a 'magical' boolean to indicate whether the character has magical powers or not. The is_squib() function in this model returns whether or not the character is a “squib”, or a non-magical creature with magical parents. This function works because the Relationship model has a one-to-one relationship with two characters, which can be used to indicate the identity of a character’s parents. If a character has no interesting data and its only purpose of existing is to illustrate that another significant character is a squib, that character will have the “hidden” boolean field set to True. This will ensure that there will not be a page created for that character.

In this model there is a function called relationships() that returns a list containing objects representing all of the relationships that the character is a part of. This is necessary because of the way the Relationship model is structured. In the relationship model, there are two associations with the character model represented by fields called character1 and character2 that represent the two characters in the relationship. The relationship() function will return a list containing all of the relationships that the character is a part of, whether they are character1 or character2.

This model has a possible many-to-one relationship with the Creature model, which organizes different species in the Harry Potter world. If the character is a creature, such as Buckbeak the Hippogriff, the 'creature' field in the Buckbeak character model will be set to an instance of Creature named “Hippogriff”. It is many-to-one, because there may be multiple Hippogriff Characters, but a Character can only belong to one Creature species. If the character is a human being, the creature field will have no value.

The character model also has a possible one-to-many relationship with the shop model. A shop may have multiple owners, as is the case with Weasley’s Wizard Wheezes, which has two owners, Fred and George Weasley.

In the second iteration a ‘sex’ field was added to the character model so that the character can be described on the webpage as a witch if they are female and a wizard if they are male.

The character model also has five possible many-to-one associations with the school model, to reflect the fact that a character could have been a founder of a house, or a founder, headmaster, professor, student or staff member of a school. These associations use foreign keys and are connected to the school models by related_names. This replaces the functionality of the Academic model that was eliminated after the first iteration.

Creature Model

The Creature model represents any non-human species within the series. It has a field called “classification” that indicates which type of creature the instance represents. There is also a “rating” field, as given by the Department for the Regulation and Control of Magical Creatures, to indicate the level of danger that species of creature poses. We also save an image of the creature. This model also keeps track of all of the characters who are of a certain type of

creature. For example, the “House elf” creature would be connected to one or more characters representing different individual house elves.

The potions() method in the creature model returns the potions that use parts of the creature as ingredients. This is possible because the potion model has a relationship with the Ingredient model, which has a one-to-one relationship with the creature model. The Ingredient model is used as an intermediary between Creatures and Potions, which will be explained further in the “Ingredient Model” section below.

The neutralize (incantation) function in the Creature model returns whether or not a certain incantation, which is a field in the Spell model, will affect that given species of Creature. This is done through the one-to-one relationship between creatures and spells, which will be explained further in the “Spell Model” section, below.

Spell Model

The Spell model represents any kind of spell, including Transfiguration, Charm, Jinx, Hex, Curse, Defensive, or Healing spell. There is a boolean to indicate whether the spell is “unforgivable” or not. Spells are associated with the Character model, to store the creator of the spell. We also save its effect, its incantation, an alias, an image, and any notable uses. The Spell model has a possible one-to-one relationship with Creature model, because it is possible that a spell could affect a creature. An example of this would be the Patronus Charm affecting the dementors. If a certain spell has this relationship with a certain creature, the neutralize (incantation) function will return true when called on that creature with the spell’s incantation as an argument.

Potion Model

The Potion model has fields indicating its name, how difficult the potion is to make, its physical description, its effects, its recipe, an image, and any notable uses. This model has an indirect relationship with the Creature model through the Ingredient model, because some potions are made with one or more ingredients derived from creatures, like Amortentia (made with the egg of an Ashwinder creature). The Potion model has a possible one-to-many relationship with the Ingredient model, and the Ingredient model has a one-to-many relationship with the Creature model. This is explained more below, in the “Ingredient model” section.

There is a function called brew() in the potion model that takes as an argument a list of ingredients objects, and return true or false depending on whether or not the list contains all of the ingredients that are associated with the potion.

Location, School, House and Shop Models

The Location model provides the basic fields that would apply to a school, a shop, or any other location in the Harry Potter universe, such as the Little Hangleton Graveyard. The fields are: name, description, image, and type of location. The School and Shop models inherit from the Location model, and the House model inherits from the School model. The house, school and shop models are separate from the location model because characters can have specific relationships with schools, houses and shops that do not apply to locations in general, such as being the owner of a shop, the professor at a school, or member of a house. However, these fields all share common aspects with other types of locations. Locations in general can be associated with stories in a many-to-many relationship. Stories may have multiple settings, and different locations may have had multiple stories occur there. Houses also have attributes for house colors, mascot, ghost (which is a one-to-one relationship with Character models, who will

happen to be ghosts too), quote, and quote_by. Shops can belong to multiple locations, such as Ollivander's Wand Shop which has locations in Diagon Alley as well as in Hogsmeade.

Artifact Model

The Artifact model can represent any significant magical object. There are fields to represent the artifact's name, description, image, and type. This model also has a possible many-to-many relationship with the character model, because a character can own zero or more artifacts, and an artifact can be owned by zero or more characters throughout its lifetime. An example of this would be the Elder Wand, which has been owned by various wizards since its existence. Harry Potter also owns all three Deathly Hallow artifacts. There is also a many-to-one relationship with the Shop model, because a Shop can have zero or more artifacts. The Hand of Glory, for example, is sold at the shop Borgin and Burkes.

Book Model

The Book model has fields for any fictional book within the Harry Potter universe. The fields are: name, description, publisher, date published, and image. This model also has two different associations with the Character model. The first association is a many-to-one association, in that a book can have one author but a Character may have authored multiple books. The second association with the Character model is a many-to-many association, as a book may have multiple Characters as subjects in it, while a Character may be included in multiple books.

Story Model

The Story model represents significant events from the Harry Potter series. There are fields for the name, description, type of story, location, and date the story happened. Since many stories are historical, there is a century() method defined inside this model that returns the century the story happened based on the date. There is also a function called formatted_date in this model that returns a string representation of the exact date the story happened if the exact date is known, and a string representation of the century otherwise. This model has many-to-many relationships with the Location, Character, and Artifact models to reflect that there are many locations, characters and artifacts present in stories, and that a single location, character or artifact could be present in many stories. Besides model similarity, another reason we chose the House, School, and Shop models to inherit from the Location model is so that we can have any of those models as settings for Stories.

INTERMEDIARY MODELS

Ingredient Model

Potions and Creatures have a more complex relationship, since Potions do not necessarily use the entirety of a Creature. Rather than a Potion using a whole hippogriff, it might just use a hippogriff talon. Thus we need an intermediary model to describe this relationship, so we created the Ingredient model. Ingredient model instances hold the name of the specific ingredient (such as hippogriff talon) and a many-to-one association with Creatures (the hippogriff talon is only associated with the hippogriff Creature, but a hippogriff Creature may be composed of many different ingredients). Potions have a many-to-many collection of Ingredients, since Potions will have multiple Ingredients and Ingredients may be used by different Potions. We find the Potions associated with Creatures by a function called potions() inside the Creature class. This creates a list of Potions that are associated with the Ingredients that are associated with the

Creature. If a certain instance of a creature doesn't have any interesting facts associated with it and only exists because it is an ingredient in a potion, the "hidden" boolean in the creature model will be set to true so that there will not be a webpage created for the creature. Currently, there are no hidden creatures.

Relationship Model

This is another intermediary model to associate Characters with other Characters. This is a many-to-many association with the model Relationship. Relationship holds two foreign keys to two different Character instances, and descriptors to explain the relationship. This is used to associate, for example, James Potter and Harry Potter. We need to save both characters and give James Potter the descriptor "dad", and Harry Potter the descriptor "son." This model enables the is_squib() function in the Character model to work. The is_squib() function determines whether or not a given character is a "squib", meaning they are non-magical but have at least one magical parent, by checking a non-magical character's relationships to find out whether they have a magical parent listed as their mother or father.

Django Templates

```
├── hogwarts                ← Django project
    ├── hogwarts            ← contains settings common to all apps under this project
    ├── marauder            ← Our web app
    │   ├── __init__.py
    │   ├── admin.py
    │   ├── models.py
    │   ├── static/
    │   ├── templates/
    │   ├── tests.py
    │   ├── urls.py
    │   └── views.py
```

Our URL configuration file ‘urls.py’ is located in our app directory ‘marauder’. For each of the pages we need, we map a regular expression to a view function. In this second iteration, we left index.html and base.html mapped to generic TemplateViews, but we created our own ModelViews specific to each model (more on this in section ‘Django Views’). For example,

```
from django.conf.urls import patterns, url
from django.views.generic import TemplateView
from marauder import views
from django.conf import settings
from django.conf.urls.static import static

urlpatterns = patterns('',
    (r'^$', TemplateView.as_view(template_name='index.html')),
    (r'^base.html$', TemplateView.as_view(template_name='base.html')),

    #Creatures
    url(r'^creatures/$', views.CreatureListView.as_view(), name='creatures'),
    url(r'^creatures/(?P<pk>\d+)/$', views.CreatureDetailView.as_view(),
name='creature'),

    #Characters
    url(r'^characters/$', views.CharacterListView.as_view(), name='characters'),
    url(r'^characters/(?P<pk>\d+)/$', views.CharacterDetailView.as_view(),
name='character'),

    #Potions
    url(r'^potions/$', views.PotionListView.as_view(), name='potions'),
    url(r'^potions/(?P<pk>\d+)/$', views.PotionDetailView.as_view(), name='potion'),
    ...
)
```

The Django default TEMPLATE_LOADER looks for a ‘templates’ subdirectory in every app inside INSTALLED_APPS in ‘hogwarts/settings.py’. So we added a ‘templates’ subdirectory to ‘marauder’, and put all our html files there. When the URLconf file searches for files matching a regex expression, it starts with the ‘templates’ directory as root path.

Inside the root path, there is a subdirectory for every model. We also have an ‘index.html’ as our splash page (shown when a user goes to our root url). The other file at root directory level is ‘base.html,’ which includes common html elements and styling that every html file in

‘templates’ except for ‘index.html’ will inherit using Django’s template tags. Our templates directory looks like this:

```
templates/
├── base.html
├── index.html
├── artifacts/
├── characters/
├── creatures/
├── locations/
├── potions/
├── shops/
└── spells/
```

In addition, every model subdirectory has its own ‘index.html’ and ‘base.html’. This ‘base.html’ is for the detail pages of each model.

Besides loading all necessary .js and .css file dependencies, our root ‘base.html’ includes two main elements that will be common to all our pages: the jumbotron and the navbar. Our navbar makes use of Twitter Bootstrap’s `active` class and Django’s template block tags to indicate which model a viewer is currently looking at (e.g., Characters, Stories, Spells).

```
<li class='{%block char-active %}{% endblock char-active %}'>
<a href="{% url 'characters' %}">Characters</a>
</li>

<li class='{%block story-active %}{% endblock story-active %}'>
<a href="{% url 'stories' %}" >Stories</a>
</li>
...
```

On each model’s index.html and base.html page, we place the ‘active’ keyword within the corresponding template tags like so: `{% block loc-active %}active{% endblock loc-active %}` to take advantage of Bootstrap’s built-in active class styling.

For the detail pages of each model, the ‘base.html’ might include custom blocks specific to the model. For instance, for the Characters model, we have the following blocks:

```
{% block info_group %}
{% block quote %}
{% block quote_footer %}
```

```
{% block stories %}
```

Also, for our second iteration, we no longer need to do 1.html, 2.html, or 3.html…; we can do all our detail pages dynamically through our base.html. To render each individual instance, we can write it like so: `{{ model.attribute }}`. If some model instances have an optional attribute and others don't, we can enclose the variable attribute in if tags:

```
{% if model.attribute %}.... {% endif %}.
```

If we want to render a one-to-many relationship, we use for loops like so:

```
{% for character in artifact.owners.all %}
    <a href="{% url 'character' character.id %}">{{ character.name }}</a>
     <br />
{% endfor %}
```

For our CSS styling, we created a static/ directory to sit beside the templates/ directory in 'marauder'. Back in the 'Hogwarts' project folder, we modified settings.py to include the variable STATIC_URL = '/static/'. (Incidentally, our BASE_DIR is set to the directory from which the application is deployed, so Django will read '/static/' as '/hogwarts/marauder/static/'.) In our html templates, we link any static files like so:

```
<link rel="stylesheet" href="{{ STATIC_URL }}css/base.css">
```

Django Views

As noted above, each model has its own ListView (for its index.html) and its own DetailView (for its detail pages that match a model instance). For our ListView, we imported SingleTableView from the module django_tables2 to provide us with table-sorting functionality, nice formatting, and pagination to only show 10 instances per page. Our ListViews inherit from this SingleTableVIew. We set up the tables to be used in the ListViews in tables.py, where we choose a limited number of fields to show and format those fields appropriately to contain links, or to better format dates. The only model that doesn't use tables in this way for their ListView is

the Houses. Since there are only 4 instances of Houses, and most fields are unique (thus not useful for sorting), we simply show the images of the houses in their ListView.

For instances of models, we have DetailView generic views. There is one special case for the details view of locations. Since the models for Houses, Schools, and Shops all inherit from Locations, we needed a way of showing more specific information for Schools and Shops and Houses than what is provided by the base Location model. So in views.py for the Locations DetailView, during the get(self, request, **kwargs) call, we check to see if the instance model is a school or shop (set with location.kind). If it is, we redirect to ShopDetailView or SchoolDetailView accordingly. These show a different template for more specific details. Houses are not included here, because we simply filter them out of the Locations ListView queryset with Location.objects.exclude(kind='House'). In the next iteration, House will not inherit from School (and then neither will it inherit from Location). It originally inherited from Location so that either could be referenced in the Academic model, but since we removed that intermediary model in this iteration, that inheritance is no longer required.

One final way we handled views is how we dealt with hidden data. We have some model instances that have incomplete information, simply to populate relationships with other model instances. An example of this is how we have a character with the name "James Potter", but no other information about him. This is so we can set up a Relationship instance specifiying that "James Potter" is the father of "Harry Potter," that way we can show the familial relationships of "Harry Potter" in his details view. However, we don't want to be able to show a details view of James Potter since it would be empty. Thus we have a boolean field for characters and creatures, called hidden, to check when setting up links or views. In the ListView of Creatures and Characters, we set our queryset to filter out hidden instance models so that they will not appear in the index tables. For the DetailView of Creatures and Characters, we check to see if

the object is hidden, and if it is hidden we raise Http404 to redirect the user to a 404 page. Thus even if the user manually typed out a URL with a hidden object's ID, we still would not show an empty view.

Search

The large amount of interconnected data available through our website necessitates search functionality. We implemented this functionality using the Django Haystack module with a Whoosh backend. Haystack allows us to define what models and attributes are searchable, generates index files, and provides APIs to perform fast searches on those indexes. We deliberately decided to leave some attributes (character and location descriptions for example) out of search results in order to limit the number of search results to a reasonable number.

Search can be conducted from the dedicated search page at the /search URL, from the homepage, and from right side of the upper menu on every other page. When searching from the home page or menu bar, the user will be taken to the dedicated search page to view the results.

Haystack does not provide the ability to serve both 'AND' and 'OR' results together from multi-word search queries. In order to implement this feature we extended and customized the haystack.SearchView class to add this functionality. Manually breaking the search query string into its components, querying the indexes, and passing the results to the front end template.

Using Twistory's API

Overview

Twistory's API is used on a page titled The Muggle Experience at /experience.html. On this page we use django templates to dynamically load data about hikes from Twistory's api. The

page consists of descriptions of three hikes that consist of static made-up explanations of how the hikes could relate to Harry Potter, with words embedded within the explanation that describe factual information about the hike from Twistory's api. This factual information is dynamically loaded using the python requests library. The images used on Twistory's website for that hike are also dynamically loaded in. A function called get_json() uses requests to visit Twistory's api and parse the json file from the webpage into a dictionary:

```
def get_json(url):
	try:
		resp = requests.get(url)
		return resp.json()
	except Exception:
		return None
```

Twistory's api has an attribute for each hike called "difficulty", which is either strenuous, moderate or easy. On our page, we dynamically load one random hike from each of the three difficulty levels. This way, every time we load the page, there could be a different hike in each position.

In the event that something is wrong with Twistory's api and we get an exception trying to load from there, we print out "Voldemort Intercepted This Message" in place of the header and factual information, and a picture of voldemort is given in place of a picture of the hike.

Implementation:

In views.py, we have a function called get_hikes() that returns 3 arrays called difficult, moderate and easy, each one containing one string consisting of the name of one of the hikes at that level of difficulty. It does that by using get_json() to visit Twistory's api at api/hikes, which contains a json file containing names of all of the hikes as keys:

```
try:
	hikes = get_json(os.path.join(twistory, 'api/hikes'))
```

In the event of get_json() throwing an exception due to their api being down, None is returned. requests loads that json dictionary into a local variable called ‘hike’, and iterates through each key in the dictionary until finding one hike of each difficulty, and then puts each one in its own array:

```
            difficult = []
            moderate = []
            easy = []
      for key in hikes:
            try:
                hike = get_json(os.path.join(twistory, 'api/hikes', key))
                if hike['difficulty'].lower() == 'strenuous':
                    difficult.append(key)
                elif hike['difficulty'].lower() == 'moderate':
                    moderate.append(key)
                elif hike['difficulty'].lower() == 'easy':
                    easy.append(key)
                if(just_one and len(difficult) >= 1 and len(moderate) >= 1 and len(easy)
>= 1):
                    break
            except MaxRetryError:
                pass
```

If a MaxRetryError is encountered in the loop trying to access a single hike, the exception is caught so that the loop can just move onto trying to access the next hike.

This function is called from get_one_hike_for_each_difficulty():

```
      def get_one_hike_for_each_difficulty():
            hikes = get_hikes(True)
            if(hikes is None):
                        return (None, None, None)
            difficult, moderate, easy = hikes
```

If get_hikes() returns None due to an exception caused by the api not being available, it returns a tuple of three Nones, which will eventually lead to voldemort being printed out. In the rare case that every single hike of a certain difficulty level returns a MaxRetryError, the array for that difficulty would be empty, which would also lead to voldemort being printed out in the position for that difficulty level.

get_one_hike_for_each_difficulty() returns the json dictionary from Twistory's api associated with the hike represented by the key in each array. It does this by going to the api page of the hike in each one-element array by going to api/hikes/{array_element}:

```
try:
    difficult = get_json(os.path.join(twistory, 'api/hikes', difficult.pop()))
except Exception:
    difficult = None
try:
    moderate = get_json(os.path.join(twistory, 'api/hikes', moderate.pop()))
except Exception:
    moderate = None
try:
    easy = get_json(os.path.join(twistory, 'api/hikes', easy.pop()))
except Exception:
    easy = None
return difficult, moderate, easy
```

An exception could occur using the pop() function if any of the easy, medium or difficult arrays are empty because there was an error on every page of the api for hikes of that difficulty level. In this rare case, that array would be assigned to None, so that the voldemort message would be printed out only in place of hike level.

This means that difficult, moderate and easy now each either contain a dictionary with one hike's attributes as keys and the values of the attributes as values, or are None. The elements returned by get_one_hike_for_each_difficulty() are randomly chosen, since get_hikes() was iterating through a dictionary, which doesn't have a defined order.

To retrieve the parks that each of the three hikes was associated with, we get the value associated with the key 'park' in the dictionaries returned by get_one_hike_for_each_difficulty(). This returns the string name of the attribute 'park' associated with that hike. Then that value is passed to the get_park() method in views.py, which goes to api/parks/{park name} and returns the correct dictionary:

```
def get_park(name):
    try:
        web_name = name.replace(' ', '%20')
        park = get_json(os.path.join(twistory, 'api/parks', web_name))
```

```
            park['name'] = name
            park['max_elevation'] = park['max_elevation(ft)']
            park['visitors'] = park['visitors(annual)']
            return park
        except Exception:
             return None
```

get_one_hike_for_each_dictionary() and get_park() are called in views.py in the function otherapi(request). For experiment.html, the view that we pass to the url function in urls.py is views.otherapi, which is defined in views.py by otherapi(request). In otherapi, ‘context’ is a dictionary containing as keys the strings difficult_hike, moderate_hike, and easy_hike. Each of these keys has as its value one of the dictionaries returned by get_one_hike_for_each_difficulty(). It also has keys called difficult_park, moderate_park, and easy_park, which each contain as values the dictionaries for the parks returned by get_park() called once for each of the three hikes.

```
difficult, moderate, easy = get_one_hike_for_each_difficulty()
context = {
        'difficult_hike': error_hike if not difficult else difficult,
        'moderate_hike': error_hike if not moderate else moderate,
        'easy_hike': error_hike if not easy else easy,
        'difficult_park': error_park if not difficult_park else difficult_park,
        'moderate_park': error_park if not moderate_park else moderate_park,
        'easy_park': error_park if not easy_park else easy_park,
    }
    return render(request, 'experience.html', Context(context))
```

‘error_hike’ and ‘error_park’ are assigned to an element in the context dictionary if get_one_for_each_difficulty() returned a none value for that level of difficulty as a result of get_json() raising an exception due Twistory’s api being down or having an error on a certain page:

```
error_hike = {'name': '[Voldemort has intercepted this message]', 'image':
settings.STATIC_URL+'images/happy-voldemort.gif', 'error': True}
error_park = {'name': '[Voldemort has intercepted this message]', 'visitors': '2
Death Eater', 'state': '[Voldemort has intercepted this message]', ‘error’:True}
```

Note that an extra key ‘error’ that is not an attribute of a hike or a park was added to error_hike and error_park, and its value is set to True.

In experience.html, the header for a hike uses the template {{other.difficult_hike.name}} (or moderate_hike or easy_hike) so that the title is the name of a random hike. Within the description paragraph, attributes of the hike such as the state its in would be printed as text using the template {{other.difficult_hike.state}}. In order to include the name of the park a hike is in, the template {{other.difficult_park.name}} is used. Each hike has an image associated with it, which is included with the template {{other.difficult_hike.image}}. This is possible because the url of an image is an attribute of hike in Twistory’s api.

If Twistory’s api was down, all of these django templates would return the values within error_hike and error_park, which would cause the page to display voldemort rather than information about hikes. If Twistory is not down, some of the templates act as hyperlinks directly to Twistory’s webpages for certain parks or hikes. However, if Twistory’s api is down we avoid linking to their website by not including the link if the ‘error’ value in that hike or that park’s dictionary is set to True:

```
{% if not easy_hike.error %} <a
href='http://twistory.pythonanywhere.com/hikes/{{easy_hike.name}}'>
{% endif %}
{{easy_hike.name}}
{% if not easy_hike.error %}
</a>
{% endif %}
```

and

```
{% if not easy_park.error %} <a
href='http://twistory.pythonanywhere.com/parks/{{easy_park.name}}'>
{% endif %}
{{easy_park.name}}
{% if not easy_park.error %}
</a>
{% endif %}
```

Unit Tests

Model Unit Tests

For the most part, the unit tests that test our models typically follow a basic framework when testing each of the models. After declaring the test Class for the model, the first member function setUp() is called, which creates a model object, sets its data members, and saves that instance. For example, for the House model:

```
class HouseTest(TestCase):
       def setUp(self):
              house = lm.House()
              ghost = lm.Character()
              ghost.name = 'Fat Friar'
              ghost.save()

              founder = lm.Character()
              founder.name = 'Helga Hufflepuff'
              founder.save()

              house.name = "Hufflepuff"
              house.description = "Nobody wants to be a Hufflepuff."
              house.quote = 'You might belong in Hufflepuff, Where they are just and
loyal, Those patient Hufflepuffs are true, And unafraid of toil.'
              house.quote_by = 'Sorting Hat'
              house.colors = 'Yellow and Black'
              house.mascot = 'Badger'
              house.ghost = ghost
              house.founder = founder

              school2 = lm.School()
              school2.name = "Hogwarts"
              school2.description = "The best school ever"
              school2.save()
              house.school = school2
              house.save()
```

The test functions can then access these instances by calling lm.model_name.object.first(). With this new variable as a local variable, we have access to all

the saved data members of that object and proceed to test that they were set properly by using assertEqual(). We tested model instance creation like so:

```
	def test_create_house(self):
		houses = lm.House.objects.all()
		self.assertEqual(len(houses), 1)
		house_created = houses.first()
		self.assertEqual(house_created, house_created)
		self.assertEqual("Hufflepuff", house_created.name)
		self.assertEqual("Nobody wants to be a Hufflepuff.",
house_created.description)
		self.assertEqual("images/empty.jpg", house_created.image)

		self.assertEqual(lm.Character.objects.first(), house_created.ghost)
		self.assertEqual('You might belong in Hufflepuff, Where they are just and
loyal, Those patient Hufflepuffs are true, And unafraid of toil.', house_created.quote)
		self.assertEqual('Sorting Hat', house_created.quote_by)
		self.assertEqual('Yellow and Black', house_created.colors)
		self.assertEqual('Badger', house_created.mascot)
		self.assertEqual('Helga Hufflepuff', house_created.founder.name)
```

We also tested that the string value of that object's name is what we expect:

```
	def test_house_string(self):
		house = lm.House.objects.first()
		self.assertEqual(str(house), "Hufflepuff")
```

REST API Unit Tests

There are also separate tests to test the interaction of the models with the API. In order to mock an API call, we use Django's Test Client class. Each test class is able to call this simple method:

```
	from django.test.client import Client

	def fetch_url(url):
	    client = Client()
	    response = client.get(url)
	    return json.loads(response.content.decode("utf-8"))
```

We have a separate test class for each model that we made accessible via the API. Each test class inherits from Django's TransactionTestCase class, instead of the usual TestCase class. This is because when we set up each class, we create model object instances

in order to call them from the API. Using the TransactionTestCase allows us to reset the database before each test so that we know exactly which objects are present in the database during any given time, since the tests can run in any order. For example, for the Character model, we start the test class like so:

```
from django.test import TransactionTestCase

class TestCharacterAPI(TransactionTestCase):

    def setUp(self):
        c = Character.objects.create(
            id=1,
            name='Albus Dumbledore',
            wand='15 inch elder wood with thestral hair',
            description=('Albus Dumbledore was the only wizard '
                         'Voldemort was wary of.'),
            magical=True,
            quote="Why doesn't anyone ever give me warm socks?",
            quote_by='Albus Dumbledore',
            sex='M'
        )
        d = Character.objects.create(
            id=24,
            name='Neville Longbottom',
            wand='a very good wand',
            description=('Neville became a surprisingly attractive teenager.'),
            magical=True,
            quote="Die, basilisk!",
            quote_by='neville on the battlefield',
            sex='M',
        )
```

After setting up, each test class tests the ‘detail’ API call and then the ‘index’ API call. For the Character model, the ‘detail’ API test looks like this:

```
    def testDetail(self):
        c = Character.objects.all()[0]
        url = reverse('character_api', kwargs={'id': c.id})
        response = fetch_url(url)
        expected = {
            "id": c.id,
            "name": c.name,
            "wand": c.wand if c.wand else None,
            "description": c.description,
            "magical": c.magical,
            "quote": c.quote if c.quote else None,
            "quote_by": c.quote_by if c.quote_by else None
```

```
            }
            self.assertEqual(response, expected)
```

We use Django's 'named url' functionality to get the URL through the function reverse(). Then we call our main helper function fetch_url() and compare the HTML response with the object we obtained directly from our model backend using Django's built-in model.objects.all() API call.

The API 'index' test works much the same way:

```
  def testIndex(self):
        all_characters = Character.objects.all()
        self.maxDiff = None
        url = reverse('characters_api')
        response = fetch_url(url)
        expected = [{'quote_by': 'Albus Dumbledore', 'id': 1, 'wand': '15 inch
elder wood with thestral hair', 'quote': "Why doesn't anyone ever give me warm socks?",
'description': 'Albus Dumbledore was the only wizard Voldemort was wary of.', 'name':
'Albus Dumbledore', 'magical': True}, {'quote_by': 'neville on the battlefield', 'id':
24, 'wand': 'a very good wand', 'magical': True, 'quote': 'Die, basilisk!',
'description': 'Neville became a surprisingly attractive teenager.', 'name': 'Neville
Longbottom'}]
        self.assertEqual(response, expected)
```

Search Unit Tests

In order to isolate search functionality from database state we defined a temporary database and search index for the lifetime of our search unit tests. We used django test fixtures to generate our temporary database from a JSON SQL dump of our production database. This provides our tests with real but static data for deterministic test results. The code example below shows the technique by which we defined our temporary database using fixtures, built a temporary index in setUp(), and destroyed that index in tearDown(). The function testNameSearchability() shows the general technique by which we compare actual search results with expected model instances. All of our search test classes follow the template of TestCharacterSearch.

```
TEST_INDEX = {
```

```
    'default': {
        'ENGINE': 'haystack.backends.whoosh_backend.WhooshEngine',
        'PATH': os.path.join(os.path.dirname(__file__), 'whoosh_test_index'),
    },
}

@override_settings(HAYSTACK_CONNECTIONS=TEST_INDEX)
class TestCharacterSearch(TestCase):

    # Snapshot of our real data
    fixtures = ['test_data.json']

    def setUp(self):
        super(TestCharacterSearch, self).setUp()
        haystack.connections.reload('default')
        call_command('rebuild_index', verbosity=0, interactive=False)

    def tearDown(self):
        call_command('clear_index', interactive=False, verbosity=0)

    def testNameSearchability(self):
        sq = generateSearchQuery('rowena', 'OR')
        sqs = SearchQuerySet().filter(sq)
        actual_results = sqsToModelList(sqs)
        self.assertEqual(len(actual_results), 1)

        rowena = Character.objects.get(pk=49)
        self.assertEqual(rowena, actual_results[0])
```