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/books/{2}
{
   "id": 2,
   "name": "Magical Me",
   "description": "Magical Me is the autobiography of Gilderoy Lockhart.",
   "author": 21
}
```

This book instance's "author" attribute holds the primary key ID of the author character. This information allows the API user to make a request to /api/characters/{21} 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 set. In this case that attribute key's value will be null, so the user can check. For example, let's another possible response from the Books collection below. This second book has no known author, so its "author" key is paired with the value null.

```
/api/books/{5}
{
   "id": 5,
   "name": "Hunting Werewolves",
   "description": "Lupus' favorite bedtime story.",
   "author": null
}
```

Django Models

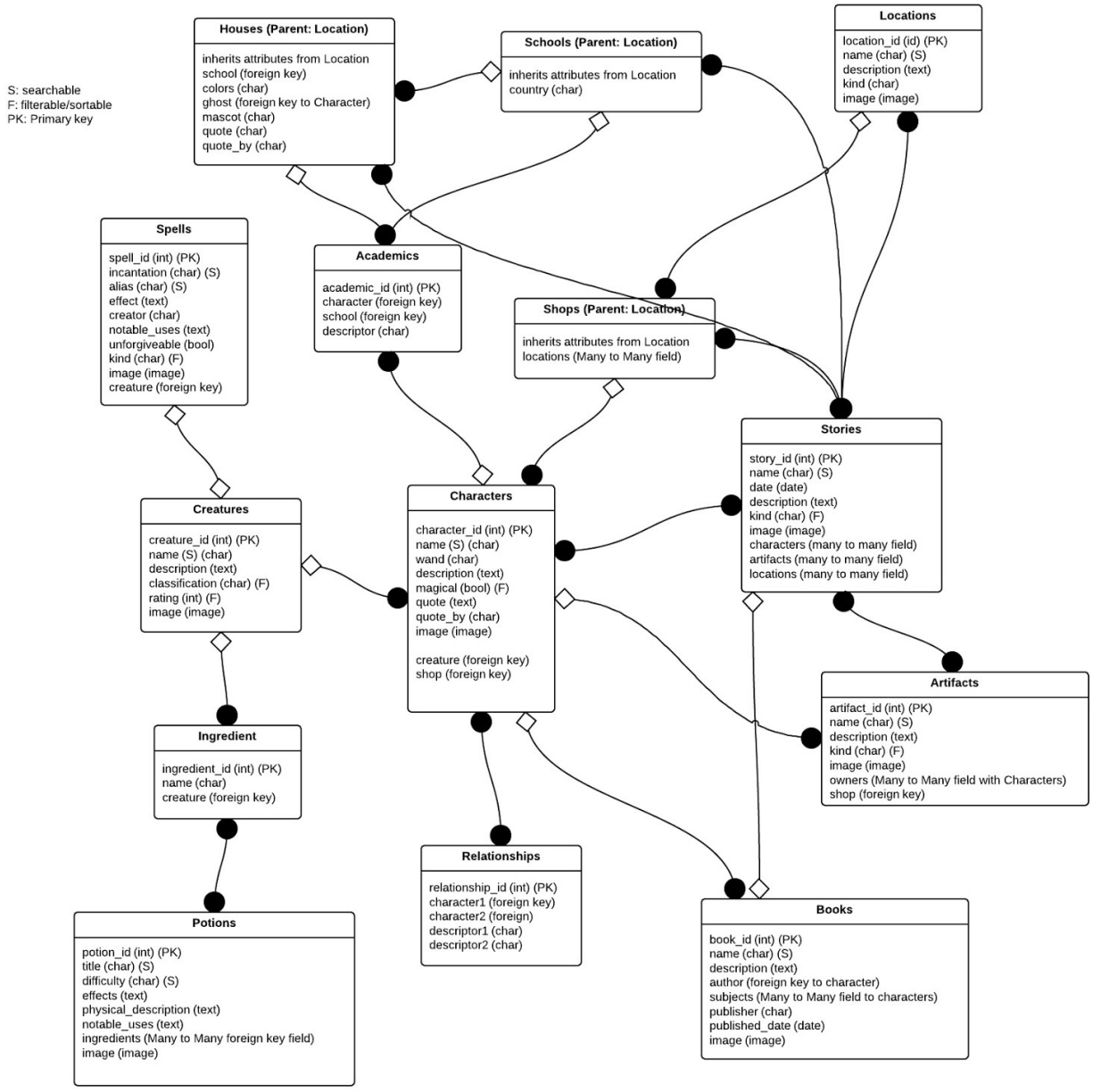

Figure 1: Entity Relationship Model for our Django Models, Phase 1

There are 14 Django models total: Character, Creature, Spell, Ingredient, Potion, Location, School, House, Shop, Artifact, Book, Story, Academic, 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.

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.

The Character model is also associated with Houses and Schools through an intermediary Academic model, which will be described further in the “Academic Model” section below.

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.

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 House, School and Shop models all inherit from the Location model. These 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. 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.

Academic Model

Another intermediary model exists to connect Characters to Schools and Houses. A Character may be a student or a founder of a House. A Character may also be a student, founder, headmaster, staff, or professor of a School. The Academic model exists to connect Characters to a School model (House inherits from School) with a descriptor describing the association. Academic has a many-to-one relationship with School types (Houses or Schools will have many Academic instances, while an Academic instance will only have one House or School). Likewise, Academic instances have a many-to-one relationship with Characters (Characters can have a house and a school through Academic instances, but an Academic instance will only have one character). In the next iteration of the project, we are going to remove this intermediary model and simply use related_name parameters to define these associations. This means that Schools will have associations with headmasters, founder, students, and professors all using Character foreign keys and distinguishing between the differences with appropriate related_names.

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.” We may use

related_names in the second iteration, like with the Academic model, to remove this model, but that is currently undecided since there are so many potential relationships (aunt, paternal ancestor, wife, etc). We will probably keep this intermediary model. 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 Views and 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 generic view function.

For example,

```
from django.conf.urls import patterns, url
from django.views.generic import TemplateView
import marauder.views as mar_views

urlpatterns = patterns('django.views.generic.simple',
    (r'^$', TemplateView.as_view(template_name='index.html')),
    (r'^base.html$', TemplateView.as_view(template_name='base.html')),
    #Characters
    (r'^characters/$', TemplateView.as_view(template_name='characters/index.html')),
    (r'^characters/1$', TemplateView.as_view(template_name='characters/1.html')),
    (r'^characters/2$', TemplateView.as_view(template_name='characters/2.html')),
```

```
    (r'^characters/3$', TemplateView.as_view(template_name='characters/3.html')),
    ...
)
```

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’ (to be completed in Phase II) and ‘base.html’. This ‘base.html’ is for the detail pages of each model. The detail pages are named unexcitingly by their ID in the database:

Filename	Detail page for
artifacts/	
1.html	pensieve
2.html	elder wand
3.html	invisibility cloak
characters/	

1.html Albus Dumbledore
2.html Ariana Dumbledore
3.html Harry Potter

potions/
1.html Amortentia
2.html Veritaserum
3.html Regeneration Potion

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. We include the following blocks so that the pages that inherit from base.html can insert their specific content:

```
{% block title %}
{% block scripts %}
{% block content %}
```

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 %}
```

For our CSS styling, images, and later JS scripts, 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">
```

Testing

Unit Testing

For the most part, our unit tests 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. 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(). Two other common test functions assert that: a) the string value of that object's name is what we expect, and b) the image that is associated with that object is what we expect.