

## **Knights who say Nil!**

Consisting of:

Ryan Brooks

Daniel Durbin

Mikkel Kim

Tim Sherlock

Jeremy Thompson

Thien Vo

## **Table of Contents**

1. Introduction.....	3
2. Django Model.....	5
3. Database Design.....	7
4. Django Tests.....	8
5. RESTful API.....	9
6. API Breakdown.....	10

## **Introduction and statement of purpose**

This project is focused upon modeling a database for the trading card game *Magic: The Gathering*. This website is primarily targeted towards current players of the game who are familiar with the rules and semantics, but is approachable to any audience curious about the game. The main purpose is to be a resource for retrieving information about particular cards. Since this is a large collectible card game, keeping track of every card and its abilities becomes difficult, if not impossible. The website will be searchable, and for each card, display all the necessary statistics and data as well as a stock photograph of the card.

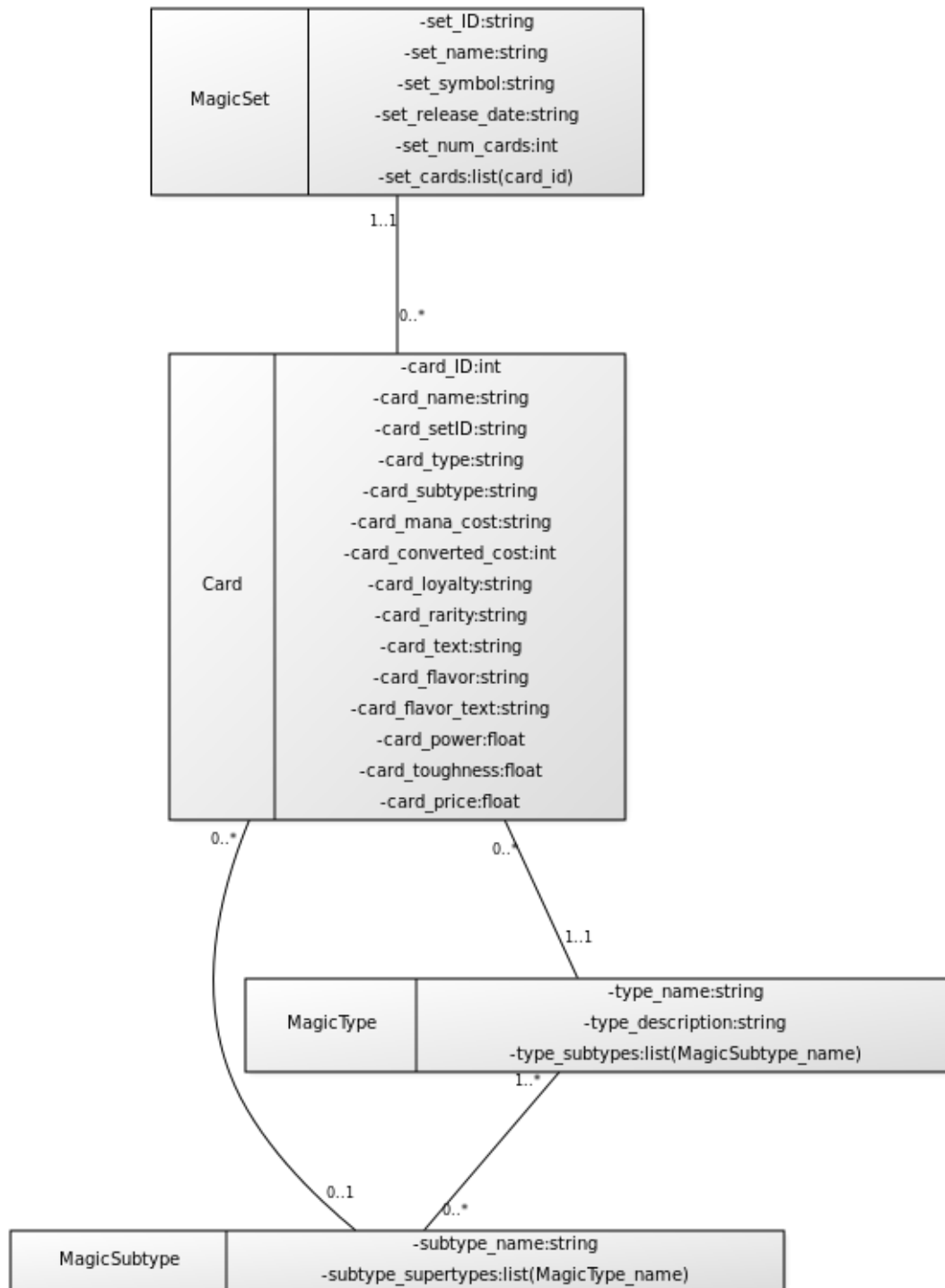
Before the layout and API is explained, a brief description of the game should be explained. *Magic* is divided up into sets, released generally once a year, that contain an average of 150-200 cards each. Each set contains cards that have various information including name, abilities, and cost to use. They also have different types with one type per card. These types range from creatures, instants, enchantments, etc. The actual purpose and use of each type is not important for use of the website, but the ability to sort by types is an important requirement to a useful *Magic* database.

Players then build a deck consisting of any number of cards, generally 60, and take turns drawing cards each turn and playing them. There is a lot of statistical work involved with building a deck. The probability of drawing particular cards can be calculated and strategies are developed with this based upon certain cards working together. This is further reason for a comprehensive database to assist players who want to build a better deck. New ideas can be developed by simply browsing around. Do you want to know if

there are any more creatures with a particular sub-type, e.g. goblin? You can display all cards with that subtype to see if the card will work with your setup. Of course, new sets come out and thus a dynamically updatable database is a must, and we will allow for insertions to update this database. The current goal for this first phase is to simply display static pages displaying the sets, and a sampling of a card from each set displaying 12 of the 21 types.

## Django Model

Our model has classes for Sets, Cards, Types, and Subtypes. This can be read by travelling down the line from the class of interest and reading the number at the end. For example, starting at class Card and checking its relation to MagicSet, we see that a Card can have one and only one Set it belongs to, but a Set can belong to 0 or many Cards. More about the database setup can be found under “[Database Design](#)”, page 6.



Inside of models.py we have modeled the above graphic encapsulated by classes for each of the 4 main tables. A snippet of class MagicSet is shown below. Each column in the table is modeled with a corresponding Django function call. This will eventually make calls to the SQL database we have set up. For now, in the tests, it will merely model a database we hard-coded in.

```
26 class MagicSet(models.Model):
27     set_ID = models.CharField(max_length = 8, primary_key = True)
28     set_name = models.CharField(max_length = 255)
29     set_symbol = models.CharField(max_length = 255)
30     set_release_date = models.CharField('date released', max_length = 255) # mm/yyyy
```

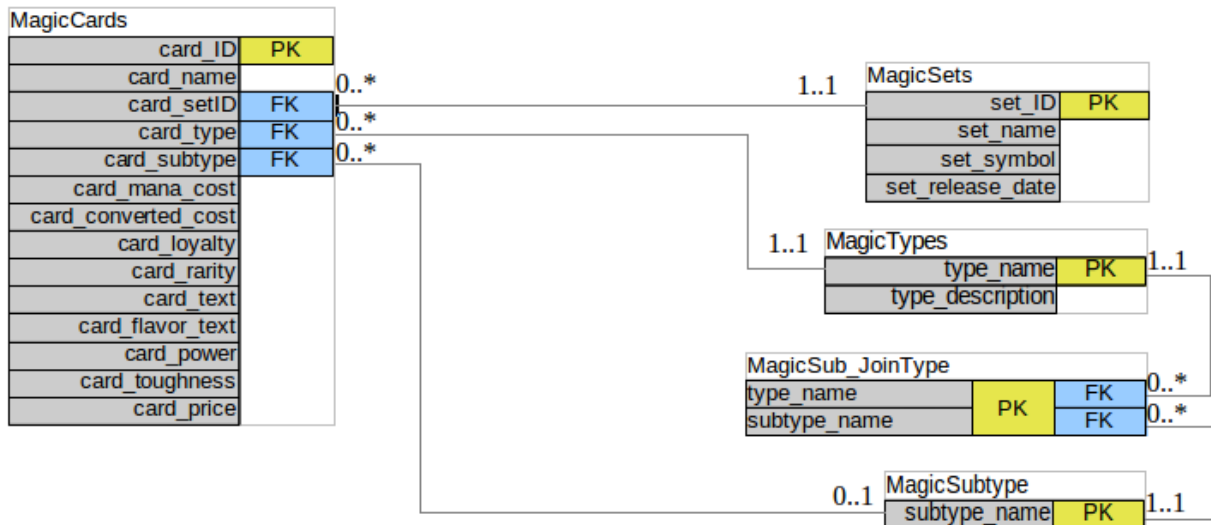
Methods within each class are also defined. They can be as simple as returning a specific data field of interest. An example is this method named `__str__` which returns the name of the set. This

```
32 def __str__(self):
33     return self.set_name;
```

over-writes the current string function so calling print will return the card's name.

The data fields inside each class model how they are placed inside the database. Thus, in the above picture on line 28, notice that `set_name` expects a CharField with maximum length of 255. Others, such as `card_power`, will expect a floating point number.

## Database Design



While the database is not a goal until the next phase, it is worth discussing as it helps the Django models and the API make more sense. The Django test framework creates a “fake” database for testing based upon this model. The main Class is of course the Magic Card itself, called MagicCards.

Since all cards belong to a set, and only one set, but many cards belong to the same set, we have a 1 to many relationship here. Multiple cards, in fact over 200 at times, can belong to one set, but it is a unique set each card belongs to. The card also must belong to a set. No empty set is allowed. This is the same with type.

Many cards will have the type “Creature”, but may or may not have a subtype named after the main type. This will show up on the card as “Creature - Minotaur”. The Minotaur will be a subtype of Creature. A card can only have one subtype, if it has one at all, but multiple subtypes can belong to one type. This necessitated the MagicSub\_JoinType table, created due to the many-many relationship between Type and Subtype.

The relationships should be straightforward, and allow for SQL queries to be easily implemented. We will also be allowing for insertions.

## Unit Tests

A good model must be tested against various inputs, and the user of our API demands for consistency and accuracy. As stated before, we have hard-coded a database in our tests.py file to attempt this. Each instance of a class is defined as a dictionary with a key as the column name, and a value. Under the function db\_setup(self) we have various models. We will use the example of a MagicSet class using the set “Born of the Gods”.

```
34         set_dict = {"set_ID" : 'BNG',
35                     "set_name" : 'Born of the Gods',
36                     "set_symbol" : '',
37                     "set_release_date" : '02/2014'}
38         MagicSet.objects.create(**set_dict)
```

The first step is to create the dictionary as shown above, and then create a django object. This class is imported from the models.py file. The dictionary is stored as if it was a database reply. We would expect an SQL query such as “select \* from MagicSet where set\_ID = ‘BNG’” to return data similar to this.

The next step is to setup the actual test itself.

```
139 def test_MagicSet_1(self) :
140     self.db_setup()
141     set_object = MagicSet.objects.get(set_ID = 'BNG')
142
143     self.assertEqual(set_object.set_ID, 'BNG')
144     self.assertEqual(set_object.set_symbol, '')
```

We define a function called test\_MagicSet\_X where X is a number that is unique to each test. We first call self.db\_setup() to setup the database for testing. We then attempt to get a set\_object by querying the database with the set\_ID = ‘BNG’. Earlier we created a set\_dict that had this ID. On line 143, we expect the set\_object’s set\_ID data member to have returned the set\_ID of ‘BNG’. It did, and this was successful test. Line 144 is for a symbol that we will be using as a link to the image file on our web server. Right now, it is blank so the test passes.

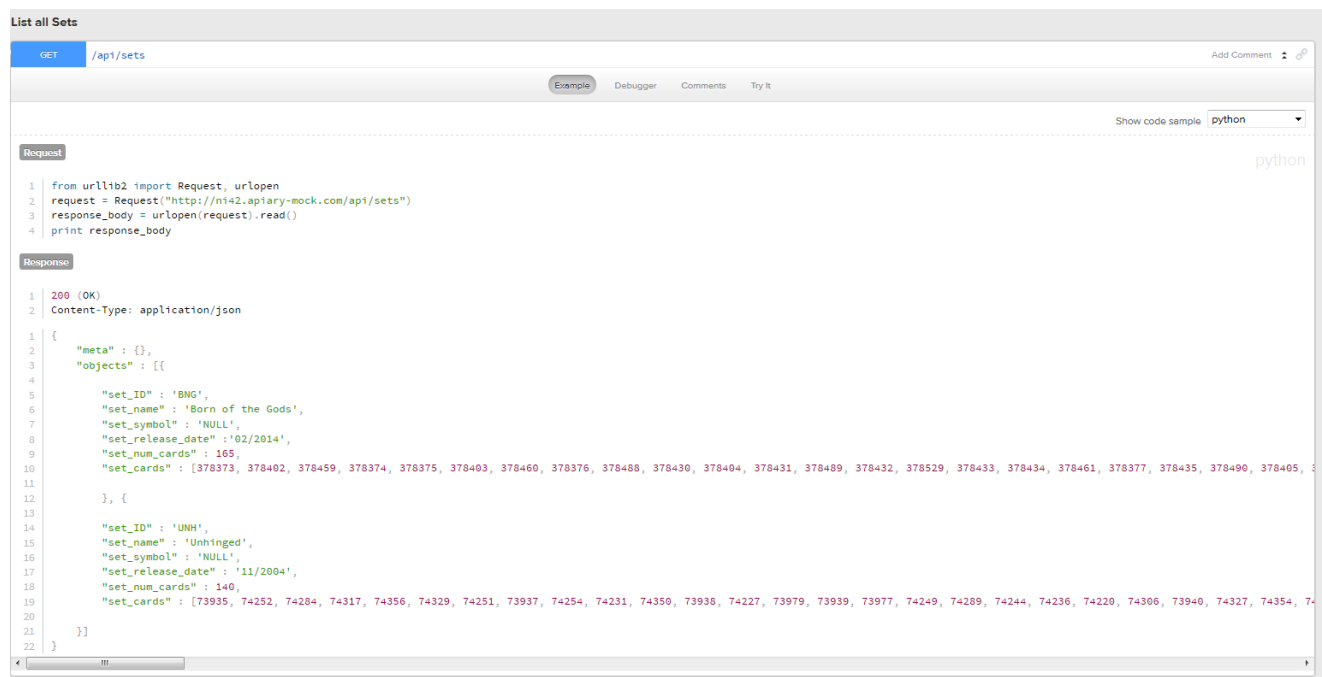


## RESTful API

We will be allowing access to the database for developers. The calls that we will support are defined in our API which can be found at <http://docs.ni42.apiary.io/> . As of this writing, we allow for multiple get methods, as defined at the documentation page.

This is subdivided into our 4 main classes; Cards, Sets, Types, and Subtypes. Selecting the “get” button under each category will reveal example(s) of an expected result to be returned from the request. One can select their language of choice from the drop-down menu and receive code for that request.

After each “get” button is a relative url address. This can be appended to our main project page [ni42.pythonanywhere.com](http://ni42.pythonanywhere.com) to get the appropriate return value.



The screenshot shows a REST client interface with a tab titled "List all Sets". The URL bar shows "GET /api/sets". Below the URL bar, there are buttons for "Example", "Debugger", "Comments", and "Try it". A "Show code sample" dropdown menu is set to "python". The "Request" section shows a Python code snippet using urllib2 and urlopen to fetch data from "http://ni42.apiary-mock.com/api/sets". The "Response" section shows a 200 OK status with a Content-Type of application/json. The response body is a JSON object with a "meta" field and an "objects" array containing two set objects. The first set is "Born of the Gods" with 165 cards, and the second is "Unhinged" with 140 cards. Each set object includes fields for set\_ID, set\_name, set\_symbol, set\_release\_date, set\_num\_cards, and set\_cards (a list of card IDs).

```
1 from urllib2 import Request, urlopen
2 request = Request("http://ni42.apiary-mock.com/api/sets")
3 response_body = urlopen(request).read()
4 print response_body

1 200 (OK)
2 Content-Type: application/json

1 {
2   "meta" : {},
3   "objects" : [[
4
5     "set_ID" : 'BNG',
6     "set_name" : 'Born of the Gods',
7     "set_symbol" : 'NULL',
8     "set_release_date" : '02/2014',
9     "set_num_cards" : 165,
10    "set_cards" : [378373, 378402, 378459, 378374, 378375, 378403, 378460, 378376, 378488, 378430, 378404, 378431, 378489, 378432, 378529, 378433, 378434, 378461, 378377, 378435, 378490, 378405,
11
12    ], {
13
14    "set_ID" : 'UNH',
15    "set_name" : 'Unhinged',
16    "set_symbol" : 'NULL',
17    "set_release_date" : '11/2004',
18    "set_num_cards" : 140,
19    "set_cards" : [73935, 74252, 74284, 74317, 74356, 74329, 74251, 73937, 74254, 74231, 74350, 73938, 74227, 73979, 73939, 73977, 74249, 74289, 74244, 74236, 74220, 74306, 73940, 74327, 74354, 7
20
21    ]
22  }
23 ]
24 }
```

If you want a list of all Sets, you would go to [ni42.pythonanywhere.com/api/sets](http://ni42.pythonanywhere.com/api/sets) and expect a return value modeled similarly to the returns shown above.

## **API Breakdown**

### **1. Card:**

- 1.1. List All:** /api/cards. This will return all cards in the database. Be warned, this will be a very large set (Over 2000 cards), so be prepared for a lot of data. It is better to use 1.2. which will return one card based on id. The cards will be returned as a JSON dictionary separated by commas. There will be 14 keys per card set. Two keys, card\_text and card\_flavor\_text can both be long text fields.
- 1.2. Retrieve a Card:** /api/card/{id}. This allows for retrieval of a particular card based on card\_id. It returns a JSON object similar to 1.1. Card\_ID is a unique identifier to all cards.

### **2. Set:**

- 2.1. List All:** /api/sets. This will return all 12 sets, in a JSON object. This will have an ID, name, Symbol, Release date, Number of cards, and also a list of all cards belonging to each set. Since one card can only belong to one set, these will all be unique ids.
- 2.2. Retrieve a Set:** /api/sets/{id}. This will return a JSON object containing one set equaling the {id} sent in.

### 3. Type:

**3.1. List All:** /api/types. This returns all types in the database, currently 21. As before, this will be a JSON object. Each type contains a name, description, and a list of all subtypes associated with that type. Multiple subtypes are possible for a type, and they are not unique to one type.

**3.2. Retrieve a Type:** /api/types/{id}. Returns a JSON object containing one type matching the {id} where ID will be the type\_name.

### 4. Subtype:

**4.1. List All:** /api/subtypes. Returns all subtypes. Two fields are returned, the name, and the corresponding supertype. This is a bit more complicated than the others due to subtypes sometimes sharing their supertype. See database design for further information on this many-many relationship.

**4.2. Retrieve a Subtype:** /api/subtypes/{id}. Returns a subtype based on {id} where {id} is the name of the subtype.