Knights who say Nil!

Consisting of:

Ryan Brooks
Daniel Durbin
Mikkel Kim
Tim Sherlock
Jeremy Thompson
Thien Vo

Table of Contents

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.
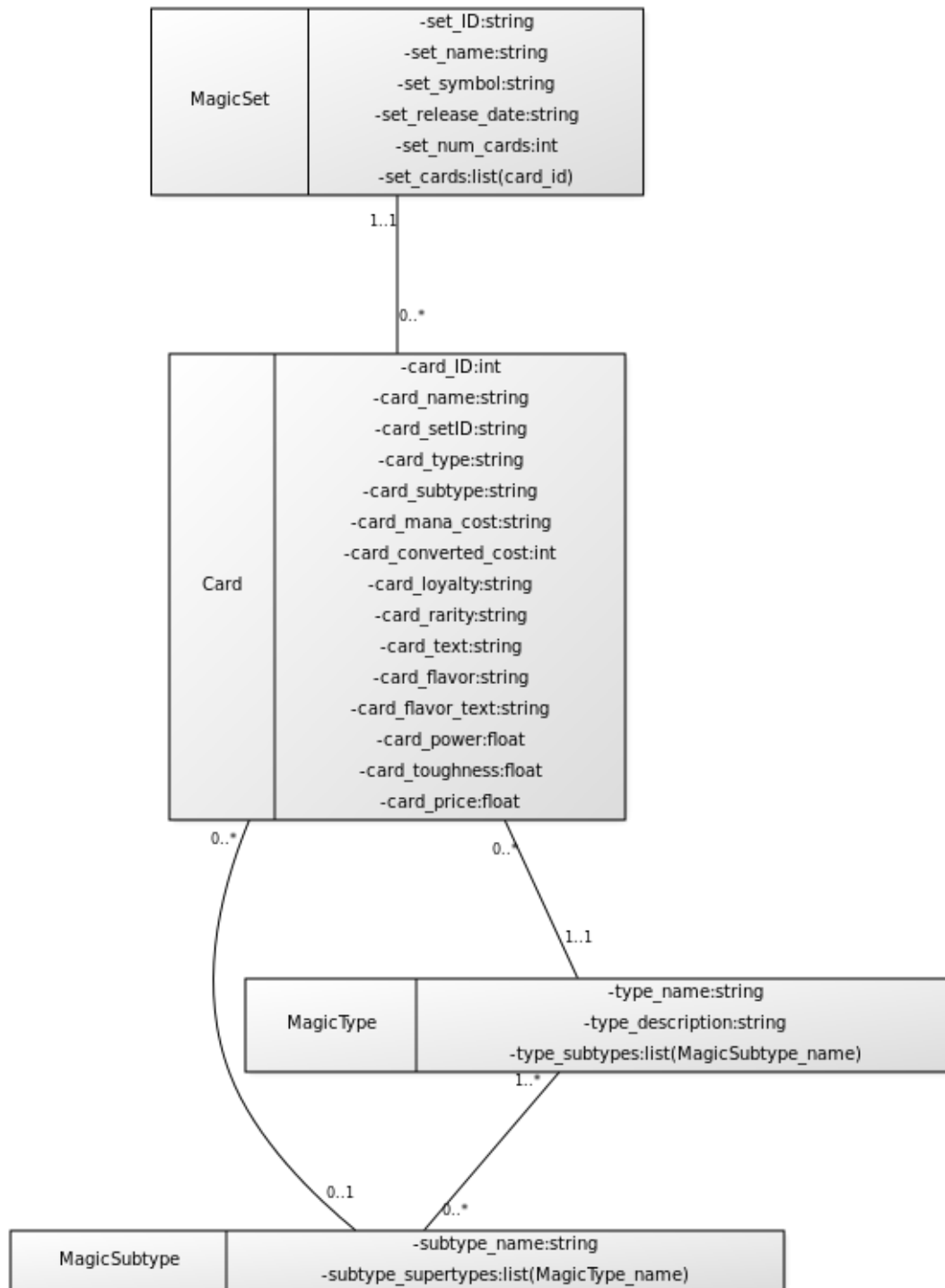
Before the layout and API is explained, a brief description of the game should be explained. *Magic* is divided up into sets, released generally once a year, that contain an average of 150-200 cards each. Each set contains cards that have various information including name, abilities, and cost to use. 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.

MagicSet	-set_ID:string
	-set_name:string
	-set_symbol:string
	-set_release_date:string
	-set_num_cards:int
	-set_cards:list(card_id)

1..1

0..*

Card	-card_ID:int
	-card_name:string
	-card_setID:string
	-card_type:string
	-card_subtype:string
	-card_mana_cost:string
	-card_converted_cost:int
	-card_loyalty:string
	-card_rarity:string
	-card_text:string
	-card_flavor:string
	-card_flavor_text:string
	-card_power:float
	-card_toughness:float
	-card_price:float

0..*   0..*

1..1

MagicType	-type_name:string
	-type_description:string
	-type_subtypes:list(MagicSubtype_name)

1..*

0..1   0..*

| MagicSubtype | -subtype_name:string |
| | -subtype_supertypes:list(MagicType_name) |

5

Inside of models.py we have modeled the above graphic encapsulated by classes for each of the 4 main tables. MagicSet is shown below. Each column in the table is modeled with a corresponding Django function call. This is used to setup the database.

```python
51 ▾ class MagicSet(models.Model):
52       """
53       MagicSet is a class that models an  actual magic set.
54       Magic sets are released every year containing 150-200 magic cards.
55       Fields inlcuded contain info that describe the set and its attributes
56       """
57       set_ID = models.CharField(max_length =8)
58       set_name = models.CharField(max_length = 255)
59       set_release_date = models.CharField('date released', max_length = 255) # mm/yyyy
60       set_num_cards = models.IntegerField()
61
```

In this case, we have 4 columns for MagicSet. set_ID used to be our primary key, but we have changed this so Django maintains these keys for us. Still, set_ID is a unique string for each set which is the initials. The column named set_symbol in the above UML has been deprecated. The symbols to be displayed on the website are generated in views.py in the set_template method. We found this to be easier than trying to store links to the images.

The set_name is the full name for the set, release_date is in a yyyy/mm/dd format. Finally, we have a column for the number of cards.

```python
 3 ▾ class MagicCard(models.Model):
 4       """
 5       MagicCard is a class that models an actual magic card.
 6       Magic cards are used to make a deck and  play the Magic the gathering.
 7
 8       fields inlcuded contain info that describe the card and its attributes
 9       """
10       card_ID = models.IntegerField()
11       card_name = models.CharField(max_length = 255)
12       card_setID = models.ForeignKey('MagicSet', related_name = 'magic_cards_set')  #'Model' in single quotes if not yet been defined
13       card_type = models.ForeignKey('MagicType', related_name = 'magic_cards_set')
14       card_subtype = models.ForeignKey('MagicSubtype', related_name = 'magic_cards_set', null = True)
15       card_mana_cost = models.CharField(max_length = 255)
16       card_converted_cost = models.IntegerField()
17       card_loyalty = models.SmallIntegerField(null = True)
18       card_rarity = models.CharField(max_length = 255)
19       card_text = models.TextField()
20       card_flavor_text = models.TextField()
21       card_power = models.FloatField()
22       card_toughness = models.FloatField()
23       card_price = models.FloatField()
```

The MagicCard class models a card. As in set, card_ID is no longer a primary key, but it is a unique number for each card. The card_name is just the name of the card. The

next three columns are foreign keys to the other 3 tables so cards can be selected on them. All cards <u>must</u> have a set and a type so they cannot be null, but subtype is not always present so it is defaulted to null. The rest of the fields are fairly self-explanatory. Line 17, card_loyalty has the same issue as subtype where it is not always used.

```python
73  class MagicType(models.Model):
74      """
75      MagicType is a class that models an magic type.
76      a type in magic is a clasification of card.
77      examples of types a card can be include: "Creature", "Land" , "Interrupt"
78      magic types can but not necessarily have subtypes.
79      all MagicCards MUST have a type.
80
81      fields inlcuded contain info that describe the type and its attributes
82      """
83      type_name = models.CharField(max_length = 255)
84      type_description = models.TextField()
85      type_subtypes = models.ManyToManyField('MagicSubtype', related_name = 'magic_subtype_set', null = True)
```

MagicType class models a type. All cards have a type, and only one type. This is modeled by a type_name, which is unique, a description, which can be fairly long, and any associated subtypes. A type can have none to many subtypes. This is modeled by the ManyToManyField in django and produces an entirely separate table in the database.

```python
93  class MagicSubtype(models.Model):
94      """
95      MagicSubtypes is a class that models an actual magic subtype
96      A subtype in magic is a specialized customization of a given type.
97
98      subtypes can belong to multiple type
99      not all cards have a subtype
100     """
101     subtype_name = models.CharField(max_length = 255)
```

MagicSubtype is the simplest class to model. All it has is a name. The complexity comes from the many to many relationship with MagicTypes.

```python
32      def __str__(self):
33          return self.set_name;
```
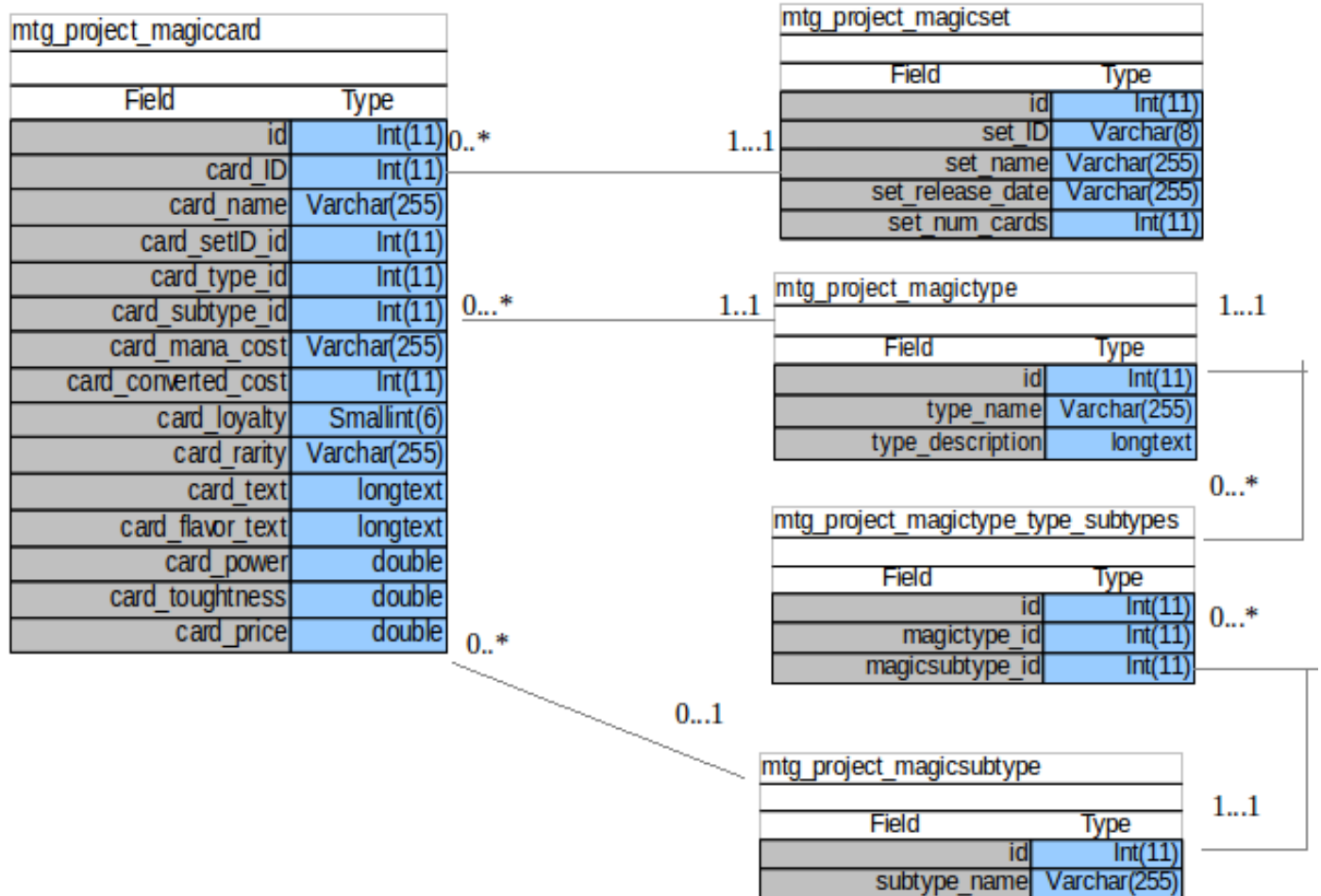Methods within each class are also defined. They can be as simple as returning a specific data field of interest. An example

is this method named __str__ which returns the name of the main field in that object, the name.

MagicSet also has a way of returning the set_ID by calling the method set_id. MagicCard has a method called get_card_attr. This is a method that will return a list of all the attributes of a card. This is similar to performing select * from mtg_project_magiccard.

Database Design

mtg_project_magiccard

Field	Type
id	Int(11)
card_ID	Int(11)
card_name	Varchar(255)
card_setID_id	Int(11)
card_type_id	Int(11)
card_subtype_id	Int(11)
card_mana_cost	Varchar(255)
card_converted_cost	Int(11)
card_loyalty	Smallint(6)
card_rarity	Varchar(255)
card_text	longtext
card_flavor_text	longtext
card_power	double
card_toughness	double
card_price	double

mtg_project_magicset

Field	Type
id	Int(11)
set_ID	Varchar(8)
set_name	Varchar(255)
set_release_date	Varchar(255)
set_num_cards	Int(11)

mtg_project_magictype

Field	Type
id	Int(11)
type_name	Varchar(255)
type_description	longtext

mtg_project_magictype_type_subtypes

Field	Type
id	Int(11)
magictype_id	Int(11)
magicsubtype_id	Int(11)

mtg_project_magicsubtype

Field	Type
id	Int(11)
subtype_name	Varchar(255)

Our database has changed since our last iteration. We are now using django to maintain the database. This is accomplished through the syncdb command through the python shell. Previously, we were using our own ID's, but have found that allowing for django to implement keys was easier, and more intuitive. The id fields in all tables are from django itself, and fixed many errors we were having.

The setup of the database is similar to last project's, but because the web pages are dynamic, and we have to implement our API fully, we found it necessary to make these changes. Django handles the SQL queries through the model calls.

9

It is still entirely possible to do your own SQL queries, however. Select statements can be used to get the necessary data. A "select * from mtg_project_magiccard" will give all of the columns from the card table. If you want to filter by card, you can add a where clause based on card_ID so that the resulting statement is "select * from mtg_project_magiccard where card_ID=373572;". This will return information for one magic card, in this case named "Yoked Ox". You can also do the same query with the card name, by replacing card_ID with "card_Name = "Yoked Ox".

Unit Tests

A good model must be tested against various inputs, and the user of our API demands 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 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'.

Similar tests are made for Card, Subtype, and Type. All tests start with self.db_setup() and obtain a set_object based upon what we are looking for. We then check the particular field on the object returned. These can be found in the django model. There are 3 tests per class.

More tests have now been added to test all of our API. The same layout for the previous class tests are used with self.db_setup() being called to set up a test database. We decided to not use our actual database due to fear of contaminating the data, or worse, deleting it. The actual API calls will use the real database.

The first set of tests, starting on line 220, test the magic_sets_all function. These functions can all be found in the api.py file located in our mtg_project directory.   We expect all of the sets to be returned. After the database is setup, we make an HttpRequest object, and set its .method to 'GET' to simulate a get request to the server. We then call the magic_sets_all method and pass the HttpRequest object, called r. The function takes in a request object, and sets an indent = to 4.

In the function, it checks to make sure the request is a 'Get', and if it is not, it throws a 404 error. We then make a dictionary with the preface key "meta" and then use list comprehension to build a list of keys by calling .objects.all() on MagicSet. This will create a __dict__ than we can grab the keys from. The end result is a list of dictionaries that is then returned. A sample follows:

```
    {
  "meta": {},
  "objects": [
    {
       "set_ID": "BNG",
       "set_name": "Born of the Gods",
       "set_num_cards": 165,
       "set_release_date": "2014-02-07"
    },
    {
       "set_ID": "DGM",
       "set_name": "Dragon"s Maze",
       "set_num_cards": 142,
```

        "set_release_date": "2013-05-03"

   },

      Back in the test, we created a python dictionary with the jsonLoadsFromResponse function and then check to make sure they key "objects" is in there. The next test checks to make sure the correct columns are returned. A set should have an id, name, and release date. The third test makes sure the meta tag is in the dictionary.

      The next set of tests test requesting an individual set by giving a set id. This is similar to the the all test, but it calls magic_sets_all, and gives a set ID in addition to the Httprequest, which is a 2 to 3 digit string. The first test just makes sure we get the correct fields. The second test asserts the set_ID is in fact "UNH" which we gave to magic_set_id. The third test checks the release date and if we got the correct card IDs. Since we made three cards in db_setup, we received three card IDs.

      We then test Types, which has a magic_types_all that will return a list of dictionary items similar to set, and a way to request an individual type with magic_type_name. As before, for the all we do not check to make sure all the types are correctly returned, but simply the correct fields. The magic_type_name tests, starting on line 332 will test to make sure we receive the correct types from the database.

      The next two sets of tests are for subtypes. To get all subtypes you can call magic_subtypes_all, and to request by name you can do magic_subtype_name. These subtypes are attached to a type, so we have a supertype associated with the type. In this case, we have an Enchant Creature as a type, and Beast as a subtype. The test will return the supertype.

      The last two sets of tests test the MagicCard class itself. We have a way to retrieve all cards with magic_card_all, and a card by card_ID with magic_card_id. More information can be found in the next section.

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 to get the approriate return value. A screenshot of sets is below...

List all Sets

```
GET        /sets.json                                                      ⬍
```

Example

Show code sample python ▼

Request   python

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

Response

```
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_num_cards": 165,
8               "set_release_date": "2014-02-07"
9           },
10          {
11              "set_ID": "DGM",
12              "set_name": "Dragon''s Maze",
13              "set_num_cards": 142,
14              "set_release_date": "2013-05-03"
15          },
16          {
17              "set_ID": "FUT",
18              "set_name": "Future Sight",
19              "set_num_cards": 180,
20              "set_release_date": "2007-05-04"
21          }
22      ]
23  }
```

If you want a list of all Sets, you would go to ni42.pythonanywhere.com/sets.json and expect a return value modeled similarly to the returns shown above. This is a list of all the sets with their ID, name, number of cards, and release date. Each of the four classes has a way to get all of the associated values, or to do a request for one particular example. For sets, after you query all the sets and decide on one you are interested in, you can goto

ni42.pythonanywhere.com/sets/{ID}.json to retreive a list of all cards associated with the one particular set. This link will take you to the listing for the set, GateCrash with ID GTC. This will be in a JSON formatted file with keys setID, set_cards, set_name, set_num_cards, and set_release date. You can retrieve the values for any of these keys.

The format for the other three classes is very similar. There will be one for cards, subtypes, and types.

For cards you can do ni42.pythonanywhere.com/cards.json to retreive all cards, then append an ID to get a .json file for the particular card. For the card "Yoked Ox" you can goto ni42.pythonanywhere.com/cards/373572.json where 373572 is the card_ID for Yoked Ox. We can see that all the information about the card is presented in json.

For types, you can goto ni42.pythonanywhere.com/types.json and if you want more information on Enchante Creature, you can goto ni42.pythonanywhere.com/types/Enchant Creature.json.

For subtypes, you can goto ni42.pythonanywhere.com/subtypes.json to get a list of all subtypes. Once you select a subtype, we'll do "Equipment", we can do ni42.pythonanywhere.com/subtypes/Equipment.json and see that an equipment subtype has supertypes or artifact and legendary artifact.

API Breakdown

1. **Card:**

   1.1. **List All:** /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**: /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:** /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:** /sets/{id}. This will return a JSON object containing one set equaling the {id} sent in.

3. **Type:**

    3.1. **List All:** /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:** /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:** /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:** /subtypes/{id}. Returns a subtype based on {id} where {id} is the name of the subtype.