Nikolas Schmidt
Dr. Jackson
Senior Project
Fall 2018

WUBRG – Introduction

Magic: The Gathering (MTG) is a competitive trading card game (TCG) in which players take on the role of Planeswalkers – spellcasters with immense and diverse magical skill sets – to summon incredible creatures and cast spells in order to best their opponents. Released to immediate success in 1993, the game has since expanded to become one of the foremost TCGs, boasting over 12,000 cards, being printed in eleven different languages, and played and loved by over 12 million people across the globe.[1] With such a staggering number of cards and playstyles, figuring out the ideal deck for a player to build can be a burden - even more so when an individual's' budget, in-game format restrictions, and other desires and limitations are taken into account. In an effort to reduce the burden of creating decks and streamline the process from conception to purchase, we have decided to create WUBRG (Read "WOO-burgh").

WUBRG is an MTG companion in which players can search for cards, build decks, playtest a build, share decks with others, and much more. WUBRG is built using the Python Django framework and leverages existing MTG APIs in order to provide a fast, efficient, and up-to-date service for users. WUBRG stores MTG card data accessed through these APIs in a SQL database server to prevent a network dependency. By using the WUBRG web application, a user can create their own account to retain various information, such as decks and favorite cards. When a user saves a deck, cards from the master database are associated with the deck. In addition to providing users with the tools to create a deck, WUBRG also allows users to browse for decks other users have created and copy them to edit as desired. WUBRG is integrated with the online MTG site TCGPlayer to allow users to quickly and easily purchase decks or cards they

[1] https://magic.wizards.com/en/content/history

have made. WUBRG also provides access to the latest official MTG blog posts and a collection of videos by popular MTG YouTube channels to provide its users with new content as soon as it becomes available.

Beginnings and Database Structure

In the first stage of production on WUBRG, time was spent researching best practices for working with Django, the database, existing APIs, and other utilities that would prove to be of use within the system. As a player, my familiarity with Scryfall as a tool for finding information on cards led me to discover its API[2], and ultimately the decision to use it as the MTG API of choice. Using Django as the platform of choice came about from my personal experiences with Python. Django, being one of the most prominent and well-documented web platforms in both the Python 2 and Python 3 series, was another leap made from personal tastes and preferences. Python 3 was selected over Python 2 in order to ensure the site avoids the impending end-of-life of the Python 2.[3] Similarly, Git was selected as a version control system due to prior use and its strong presence and reputation in the software industry. Beyond these preliminary decisions, time was also spent looking over numerous other tools, including the implementation of test-driven development in Django and other Python packages that could be used with Django's built-in test server.

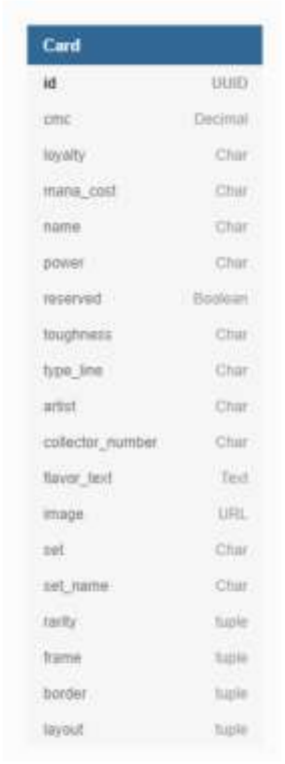
The decisions made regarding the initial setup for the database were made with less familiarity. Django as a web framework is database-agnostic, meaning that any instructions that interact with the database are constant, and all operations can be translated from one database to

[2] https://scryfall.com/docs/api
[3] https://pythonclock.org/

another simply by changing a small number of configurations within Django. With many choices, the initial approach to the system was to use MongoDB as the database engine. Mongo is different from most database engines, in that it stores information in an object-oriented style rather than a traditional relational database design. While looking over Mongo, however, it was discovered that using Mongo's Atlas Cloud service to access the database at the free level (our desired price limit as students) was limited to 512MB of storage. This led to the search of whether or not 512 MB would be suitable for operations.

To determine whether or not 512MB would be sufficient, we began working with the data from Scryfall to determine how much storage space would be necessary. The data provided by Scryfall's API, in addition to being available through requests to the API, is also available through a regularly-updated JSON file. While making requests to Scryfall would be an option, the complexities of handling remote requests, the speed of executing such requests, request limits imposed by Scryfall, and other issues encouraged the use of their JSON. Downloading the JSON from Scryfall's site reveals that zipped, the contents are 55.5 MB by themselves, and unzipped 555 MB, well over the MongoDB limit for unpaid use. As such, other database options were researched, and sqlite was then decided upon. Sqlite is a database system that exists on a device's file system as opposed to its own separate server and is supported by Django natively – whereas support for certain other database engines, including Mongo, are primarily maintained by third parties. The reliability of a database backend that was natively supported by Django, as well as the ease of access in working with a local file as opposed to a remote database were what ultimately led to the decision to use sqlite.

Card Data Models and Early Development



Card	
id	UUID
cmc	Decimal
loyalty	Char
mana_cost	Char
name	Char
power	Char
reserved	Boolean
toughness	Char
type_line	Char
artist	Char
collector_number	Char
flavor_text	Text
image	URL
set	Char
set_name	Char
rarity	tuple
frame	tuple
border	tuple
layout	tuple

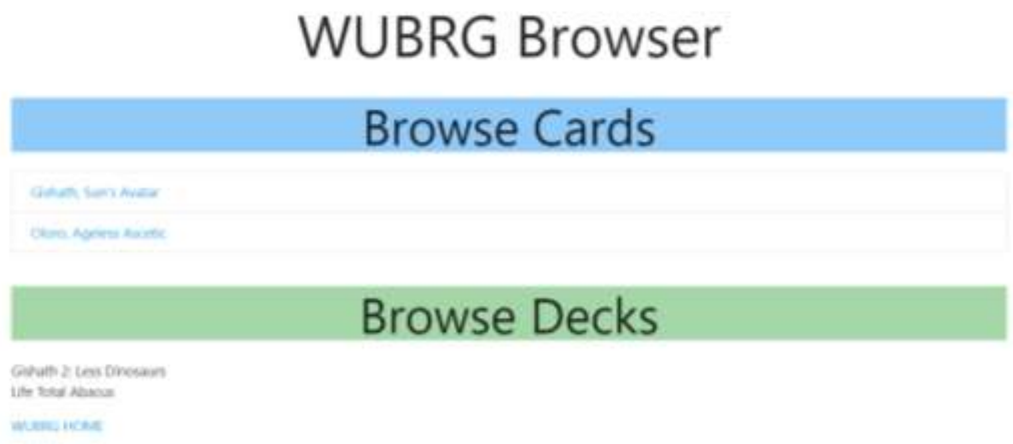
The above shows a simplistic version of the original model used for Card objects in the database.

Django is built using the Model-Template-View architecture, a slight deviation from the traditional Model-View-Controller architecture.[4] In Django, a data model is distinct from the page associated with it, which allows for changes to be made to either aspect without directly interacting with one another. This is useful in development as it allows for multiple developers to work on different aspects of the project at the same time without interrupting one another. In Django, when a request to access a url is made, the url is processed in pieces, with each piece being passed to the appropriate handlers, the views. A view is a method built to assemble content to be used with a generated page. Pages in Django are built using templates, which are essentially HTML files featuring special, non-HTML markup that allows them to incorporate information provided by a view. Views traditionally access data stored in a database, which

[4] https://docs.djangoproject.com/en/2.1/faq/general/

requires the data to be modeled in some consistent manner, allowing for the reuse of templates across any given model. As such, the next stages of development included working with the Scryfall data in order to structure the design of the data models.

To begin, a few cards were removed from the master JSON and examined. Relevant fields were determined as a group and the information for these cards was manually entered into the database. While selecting these cards, a number of edge cases were discovered, such as double-faced cards, cards with different frame layouts, and other similar mechanics that, while intuitive and fun in paper, are difficult to represent in a digital model. As such, these edge case cards were simply ignored for the time being. A model was then created that serves the needs of standard-templated cards containing the relevant fields, and sample pages were created for the selected cards. This required some back-and-forth changes related to deprecation of certain Django functions, specifically the use of a `path` function in place of a `url` function in Django 2.x for routing requests. The `url` function was prominent in earlier versions of Django and remains functional for backwards-compatibility.



WUBRG Browser

Browse Cards

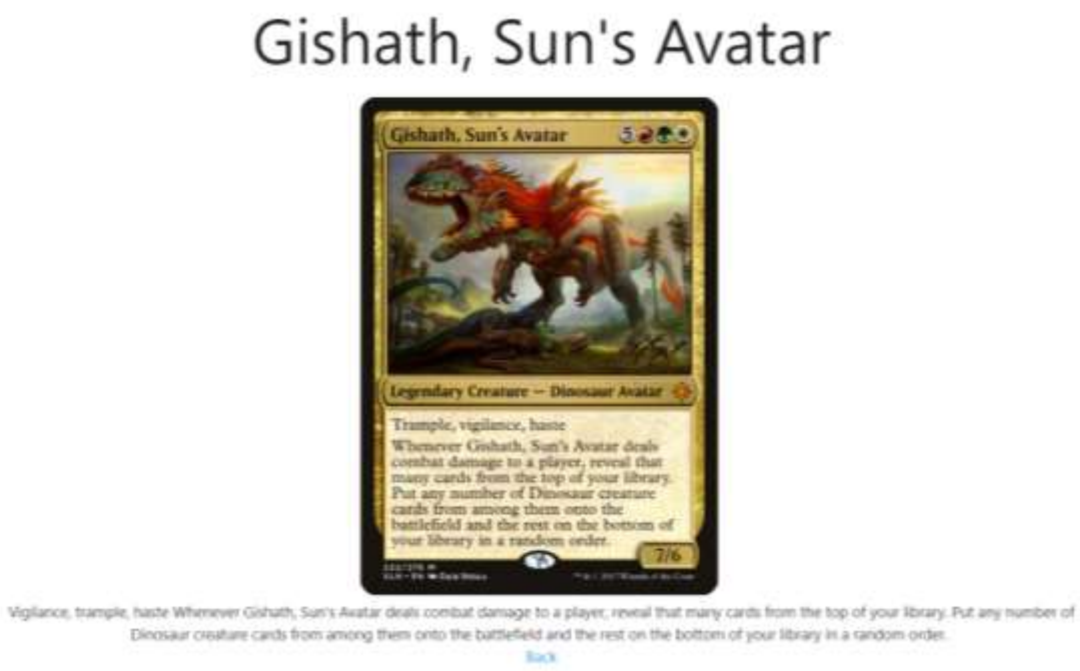
Gishath, Sun's Avatar

Okoro, Ageless Ascetic

Browse Decks

Gishath 2: Less Dinosaurs
Life Total Abacus

WUBRG HOME



Gishath, Sun's Avatar

Gishath, Sun's Avatar
Legendary Creature — Dinosaur Avatar
Trample, vigilance, haste
Whenever Gishath, Sun's Avatar deals combat damage to a player, reveal that many cards from the top of your library. Put any number of Dinosaur creature cards from among them onto the battlefield and the rest on the bottom of your library in a random order.
7/6

Vigilance, trample, haste Whenever Gishath, Sun's Avatar deals combat damage to a player, reveal that many cards from the top of your library. Put any number of Dinosaur creature cards from among them onto the battlefield and the rest on the bottom of your library in a random order.

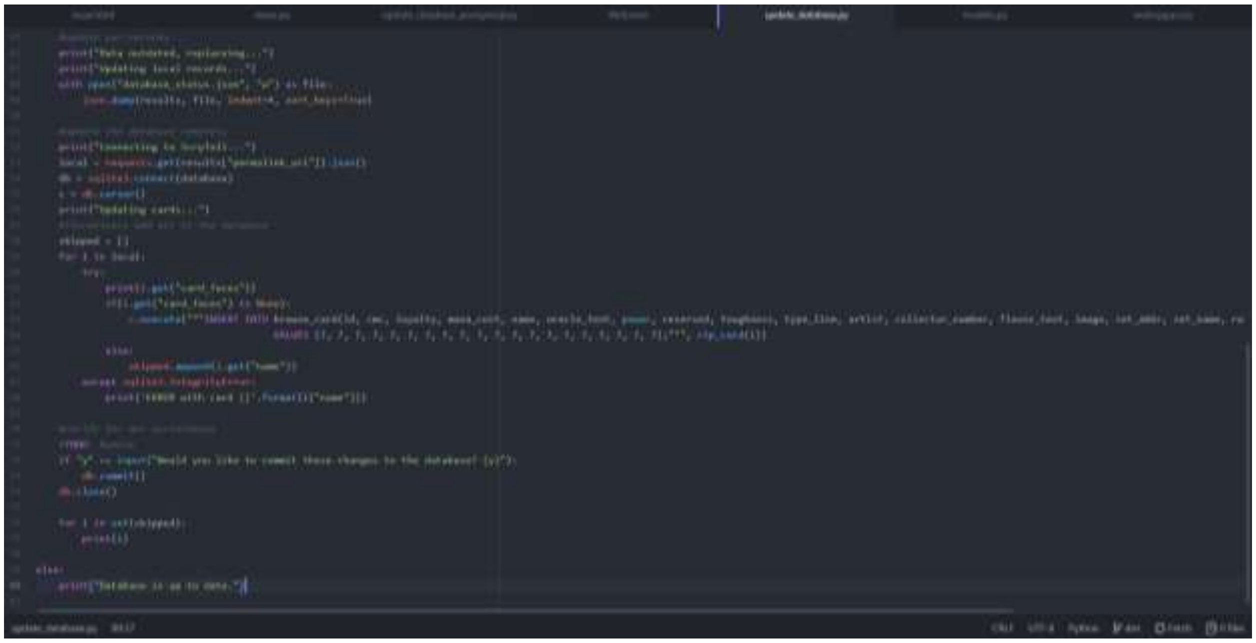
Back

The above show the initial development designs for the Browse menu and individual card pages.

Looking forward, the need to keep the database up to date became another concern. New MTG sets are released every quarter, featuring usually around 3000 new cards. Supplemental sets with smaller numbers of cards as well as reprints are also released on a similar schedule.

Cron jobs were discussed as a means by which to run a script to analyze the Scryfall JSON and update the database as the data is altered. This, however, would require a new platform to host the application. Up until this point, Django's built-in test server (running off of localhost on the developer's machine) had served the site's needs well, but this was not sustainable. Of the various hosting options, Google App Engine was our first target to investigate. Unfortunately, the requirement of a credit card to access the service, regardless of the project's level of the service, did not meet our criteria. Heroku was then decided on due to the lack of restrictions on its free service, as well as ease of integration with Git and GitHub. This new functionality encouraged us to create another branch to develop code on, so that the master branch could function live on Heroku. The dev (as in "development") branch became our primary working environment as we moved forward.

With a production and a development environment established, we began working on extracting the relevant fields from the Scryfall API and incorporating the standard-templated cards into the database. A Python script that could be run independently of the Django application was written to enter the data from Scryfall into our database. The script would record the version of the Scryfall API that was last used with the database to a local JSON for metadata about the updates in order to determine whether an update to our database was necessary. If an update was, it would update the local metadata JSON with the information about the newest Scryfall data, then begin working to update the database. This script, highlighted below, was never actually completed, however, due to conditions of the production environment leading to the abandonment of sqlite.



```
    [illegible]
    print("Data outdated, replacing...")
    print("Updating local records...")
    with open("database_status.json", "w") as file:
        json.dump(results, file, indent=4, sort_keys=True)

    [illegible]
    print("Connecting to Scryfall...")
    local = requests.get(results["permalink_uri"]).json()
    db = sqlite3.connect(database)
    c = db.cursor()
    print("Updating cards...")
    [illegible]
    skipped = []
    for i in local:
        try:
            print(i.get("card_faces"))
            if(i.get("card_faces") is None):
                c.execute("""INSERT INTO browse_card(id, cmc, loyalty, mana_cost, name, oracle_text, power, reserved, toughness, type_line, artist, collector_number, flavor_text, image, set_abbr, set_name, ra
                                  VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)""", zip_card(i))
            else:
                skipped.append(i.get("name"))
        except sqlite3.IntegrityError:
            print("ERROR with card {}".format(i["name"]))

    [illegible]
    [illegible]
    if "y" == input("Would you like to commit these changes to the database? (y)"):
        db.commit()
    db.close()

    for i in set(skipped):
        print(i)

else:
    print("Database is up to date.")
```

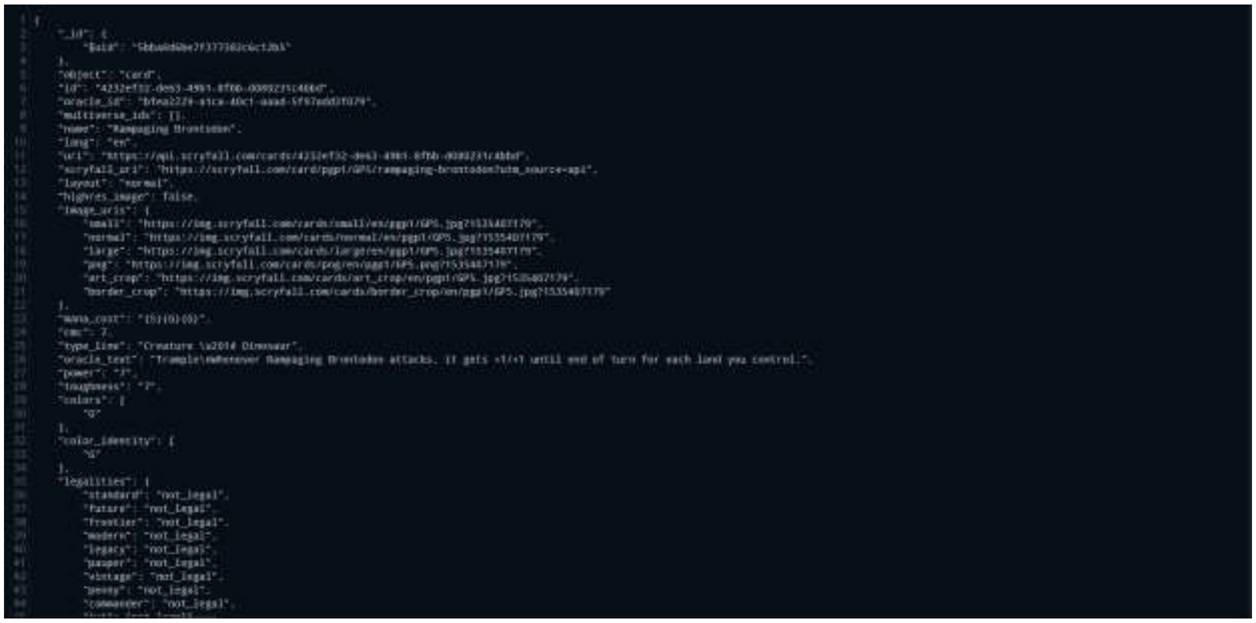
The above shows the original script, requiring confirmation, written to add data to our database.

Issues Arising in Development

Working with Heroku, it became clear that using sqlite as our database would not continue to be practical. Conditions of the Heroku environment, such as occasional restarts of the dyno hosting the app, would often lead to data loss, dropped tables, and other insecurities that became impediments to the successful operation of WUBRG. This forced us to find another database backend that could better support our application. This restructuring of the database engine also provided an opportunity to find a more effective solution to handle the numerous edge-case cards, which we took advantage of by looking into a database that would more effectively handle Scryfall data in its native JSON format. This would alleviate the issues involved with edge-case cards, such as double-faced cards, which have two `face` objects stored in a single `card` object. In addition, this also would make it easier to access certain attributes of cards which were stored in lists in the Scryfall data structure, such as a card's various reprintings or a list of a card's colors, a fundamental aspect of MTG.

Research into new databases that could support this structure inevitably led us back to MongoDB. Mongo, being an object-oriented database, can quickly and effectively store JSON data. Despite our concerns with pricing, we moved forward with Mongo again, as we believed that our model would strip the amount of data needed from Scryfall to a level below the 512MB limit. Before we did this, however,a simple script was written to store all of Scryfall's data in the free tier Mongo database, simply to identify what would happen if we went over their data cap. It was interesting to note that, because the script completed all the transactions at once, we could actually reach 551MB, 39 MB more than advertised limit for that free tier. Unfortunately, any further additions to the database, such as adding users or cards, would not be allowed due to our database being over capacity.

Fortunately, further research revealed that our Heroku hosting through Devon's GitHub Student account provided us with a $200 credit that could be charged against in order to bypass our prior storage limits, and provided us with numerous database services, including Mongo. With this new knowledge and source of funding, a script was written to incorporate Scryfall's data into a new, higher-tier Mongo database, and WUBRG was altered to connect to Mongo instead of sqlite. A new branch, dubbed "mongo" was created to develop these new changes. Django, being a database-agnostic platform, allows for easy transitions from one backend to another, and soon enough the cards, users, and other information were all accessed via a Mongo database.



```
{
    "_id": {
        "$oid": "5bba0d6be7f377302c6ct2b5"
    },
    "object": "card",
    "id": "4232ef32-dee3-4961-8fbb-d0892231c4bbd",
    "oracle_id": "bfea2229-e1ca-40c1-aaad-5f97add2f079",
    "multiverse_ids": [],
    "name": "Rampaging Brontodon",
    "lang": "en",
    "uri": "https://api.scryfall.com/cards/4232ef32-dee3-4961-8fbb-d0892231c4bbd",
    "scryfall_uri": "https://scryfall.com/card/pgp1/GP5/rampaging-brontodon?utm_source=api",
    "layout": "normal",
    "highres_image": false,
    "image_uris": {
        "small": "https://img.scryfall.com/cards/small/en/pgp1/GP5.jpg?1535407179",
        "normal": "https://img.scryfall.com/cards/normal/en/pgp1/GP5.jpg?1535407179",
        "large": "https://img.scryfall.com/cards/large/en/pgp1/GP5.jpg?1535407179",
        "png": "https://img.scryfall.com/cards/png/en/pgp1/GP5.png?1535407179",
        "art_crop": "https://img.scryfall.com/cards/art_crop/en/pgp1/GP5.jpg?1535407179",
        "border_crop": "https://img.scryfall.com/cards/border_crop/en/pgp1/GP5.jpg?1535407179"
    },
    "mana_cost": "{5}{G}{G}",
    "cmc": 7,
    "type_line": "Creature \u2014 Dinosaur",
    "oracle_text": "Trample\nWhenever Rampaging Brontodon attacks, it gets +1/+1 until end of turn for each land you control.",
    "power": "7",
    "toughness": "7",
    "colors": [
        "G"
    ],
    "color_identity": [
        "G"
    ],
    "legalities": {
        "standard": "not_legal",
        "future": "not_legal",
        "frontier": "not_legal",
        "modern": "not_legal",
        "legacy": "not_legal",
        "pauper": "not_legal",
        "vintage": "not_legal",
        "penny": "not_legal",
        "commander": "not_legal",
```

An example of a Scryfall card stored in a Mongo database.

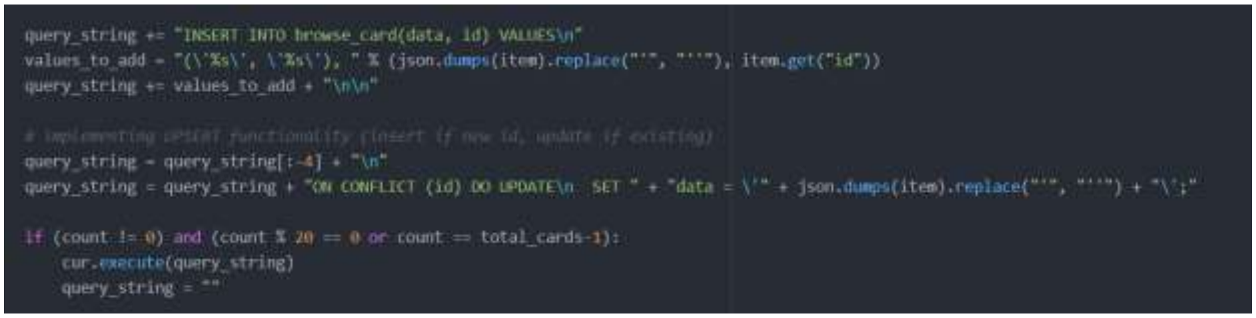
The next step required us to revise our models in order to access the data in a new format, as our prior data models that interacted with a relational database like sqlite would not be able to interact with an object-oriented Mongo database. To do this, Django needed a new database engine to communicate with Mongo. The most prominent Django engine for Mongo exists as a Python package known as "djongo", a clever portmanteau of "Django" and "Mongo"; albeit one that is easily misidentified as a typo. Working to get djongo running, unfortunately, proved to be a fruitless matter in Django 2.1, which WUBRG was running on. Due to djongo being developed by third parties, independent of both Django and Mongo, the latest version of Django supported by djongo was 1.5. Rolling back the version of our main web engine was immediately rejected, due to the many unknowable issues that might result from such a decision. As such, Mongo was abandoned for a second time, and alternatives provided by out $200 credit that could still support our JSON structure were investigated yet again.

PostgreSQL became an increasingly prominent database option to use for our purposes. Being a relational model, PostgreSQL is strongly supported by the Django Model-View-

Template architecture. Additionally, PostgreSQL also includes a data type to store JSON strings within the database. These JSON strings are able to be interacted with in the database as well as in Django as though they were raw JSON data themselves. All of this, combined with our $200 credit being applicable towards PostgreSQL hosting, led us to replicate our existing scripts to function with a PostgreSQL database; and rework our Django backend to connect to PostgreSQL. This was all done on the mongo branch, for convenience sake, so as to not create another new branch.

Working with PostgreSQL seemed to alleviate all of our issues with database engine, model structure, engine support, and limited funds. We loaded a small number of cards into the database to make the necessary changes and the application worked in the same manner as it had before transitioning. These changes were merged with both production and dev branches, and mongo was deleted. All current aspects of all versions of WUBRG seemed to be functional and replicated across all branches. Having everything working, we began running our script to initially populate the database.

Unfortunately, this proved to be another difficulty. JSON strings written to the database included quotation marks and apostrophes, which are Python string delimiters. Because our database scripts were running in Python, it was difficult to effectively create these SQL statements in Python. This issue was exacerbated by cards which included these characters in fields, such as the apostrophe character in the name of the card "Gishath, Sun's Avatar". Additionally, the speed with which transactions could be made to the database were much slower compared to the file system writing operations of sqlite, or the bulk JSON data entry in Mongo. Resolving the issues with quotations took some time, as both Python and PostgreSQL use different escape characters and quotation characters, so using one language to execute expressions in another required some careful use of each language's capabilities. One of the most helpful features of Python was the language's `string.replace()` function. This function takes two arguments: a search string and a replacement string. When called on a string, this function replaces all instances of a search string with the replacement. This allowed us to translate the strings delimiting fields in the JSON to a state in which they did not interfere with our Python code. Resolving the issues related to optimizing the database operations took a quite a bit longer.



```python
query_string += "INSERT INTO browse_card(data, id) VALUES\n"
values_to_add = "(\'%s\', \'%s\'), " % (json.dumps(item).replace("'", "''"), item.get("id"))
query_string += values_to_add + "\n\n"

# implementing UPSERT functionality (insert if new id, update if existing)
query_string = query_string[:-4] + "\n"
query_string = query_string + "ON CONFLICT (id) DO UPDATE\n  SET " + "data = \'" + json.dumps(item).replace("'", "''") + "\';"

if (count != 0) and (count % 20 == 0 or count == total_cards-1):
    cur.execute(query_string)
    query_string = ""
```

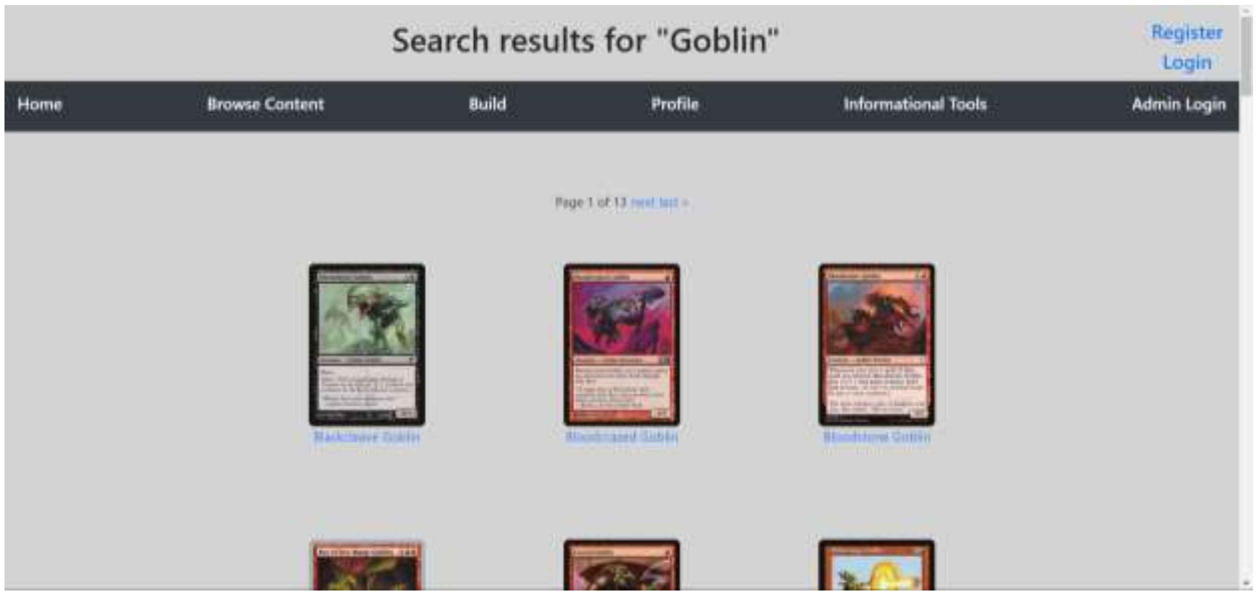
A portion of the database script, highlighting the use of Python's `string.replace()`.

The initial production version of the database update script worked by creating a new table in the database, inserting all cards from the most recent update to Scryfall into this new table, dropping the original table, and renaming the new table to the name of the original table. This worked well but was slow because every insert operation was executed independently, requiring a large amount of time to generate the string, execute the SQL, and transfer the data for each card in the database. Work to reduce this time-expensive operation began by reducing the number of times SQL was executed. To achieve this, trial and error led us to execute transactions for twenty cards at a time. This provided the optimal tradeoff between inserting multiple cards at once, and effectively updating our database in a reasonable timeframe. Unfortunately, later updates to WUBRG led to issues pertaining to dropping the database, as relationships between cards and decks, something that was implemented shortly after this script was written, were destroyed during the deletion of the first table. This required an “upsert” operation, which either updates or inserts a record in a database depending on certain criteria. Upserting led to some slowing down, as we couldn’t explicitly insert 20 records in one transaction, and instead had to create 20 different statements then execute them all independently in a row. This emulated the fast structure of the prior version, while improving upon the issues encountered with our new deck data models.

Caching and Searching

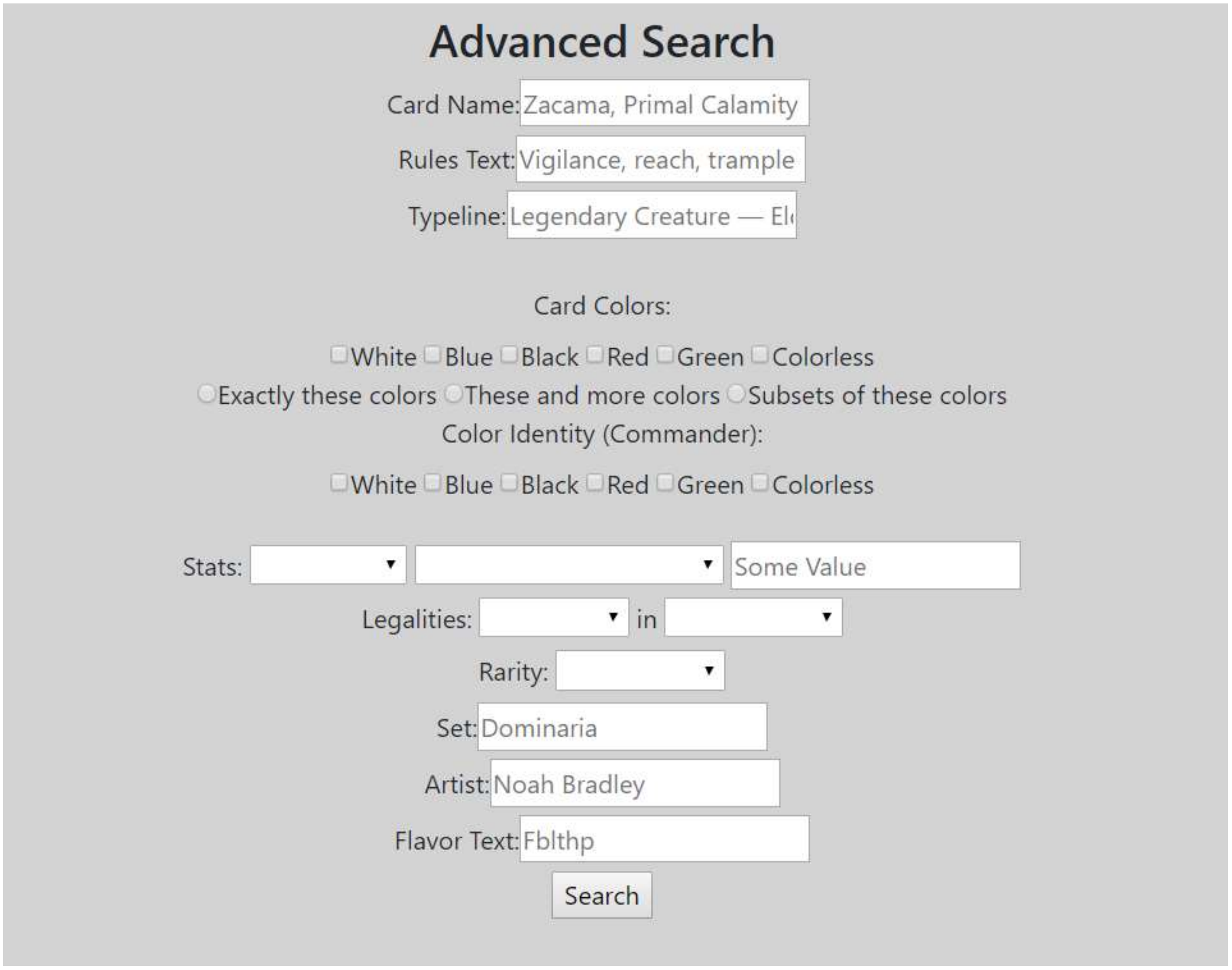
At this point, with all cards now in the finalized database, the only way to find individual cards through WUBRG was to search through roughly 450 pages each displaying 42 cards (excluding the last page) worth of cards sorted alphabetically, using only the art from their first printing. These limitations, filters, and renderings were time consuming and inefficient, which led to two new features – the first being caching of the page, and the second being the ability to search for individual cards using their attributes. To implement caching, Django provides a number of built-in options. Caching can be enabled in Django to store cached data in memory, in the database, or in the file system based on the needs and use of the site. For WUBRG, we decided to implement database caching, as our PostgreSQL database had much more space than we intended to use for development, and we believed that there was significant benefit in caching pages for all users to interact with. Implementing this required specifying a table to store cached versions of rendered pages in, as well as a timeframe for how long a page would be stored. With caching implemented, Django would check to see if a specific page had been saved in the cache before attempting to render the page. If a cache hit was found, Django would load the rendered page as stored in the database. Otherwise, the appropriate page would be rendered dynamically and cached for future use. We initially set our cache to store pages for one day, and this greatly improved load times across the site. However, due to the nature of loading from the cache, development could not persist with Django loading pre-rendered pages, largely due to changes being actively made to the pages. Discussion was held over whether or not to set our cache time as an environment variable, but it was decided that simply disabling caching (by setting the cache time to 0 seconds) for development purposes would be a simpler solution until full production.

Focus then shifted towards allowing a user to search for individual cards. Our primary page to view cards at present consisted of all first-printings of cards sorted in alphabetical order. To enhance this, a search function was created. The initial search feature revolved around searching for cards by name. Searching for cards this way would return a list of cards whose names contained a case-insensitive version of the search string. This was passed as a URL query with a string parameter that was perpetuated through the duration of the search. This allows a user to return to their search should they wish to view the details of a specific card and return to the search. Page numbers were also preserved for both searches and the list of all cards, to allow a user to easily return to their location in a search. This simple search feature was also replicated for decks.



A page showing the initial version of a simple search for cards whose names contain "Goblin".

Extending the primitive name-based search, an advanced search feature was also incorporated into WUBRG. Cards have a number of fields that a user may wish to search by that include typeline, functional rules text, mana cost, colors, and more. Because a user may want to filter on more criteria than just one of these, we decided to create a new search feature that allows a user to filter on as many of the advanced search fields as desired at once. Django allows for simple filtering of a result list, which is how the advanced search feature was implemented. The form allows the user to filter the results by a number of different card attributes. These are all passed to the view by a url querystring parameter. The view checks to see what values, if any, are present in the querystring and filters as applicable, starting with a set of all cards. These results are displayed to the user in the same manner as a simple search, leveraging Django's use of templates.



Advanced Search

Card Name: Zacama, Primal Calamity

Rules Text: Vigilance, reach, trample

Typeline: Legendary Creature — El

Card Colors:

☐ White ☐ Blue ☐ Black ☐ Red ☐ Green ☐ Colorless

○ Exactly these colors ○ These and more colors ○ Subsets of these colors

Color Identity (Commander):

☐ White ☐ Blue ☐ Black ☐ Red ☐ Green ☐ Colorless

Stats: ▾ ▾ Some Value

Legalities: ▾ in ▾

Rarity: ▾

Set: Dominaria

Artist: Noah Bradley

Flavor Text: Fblthp

Search

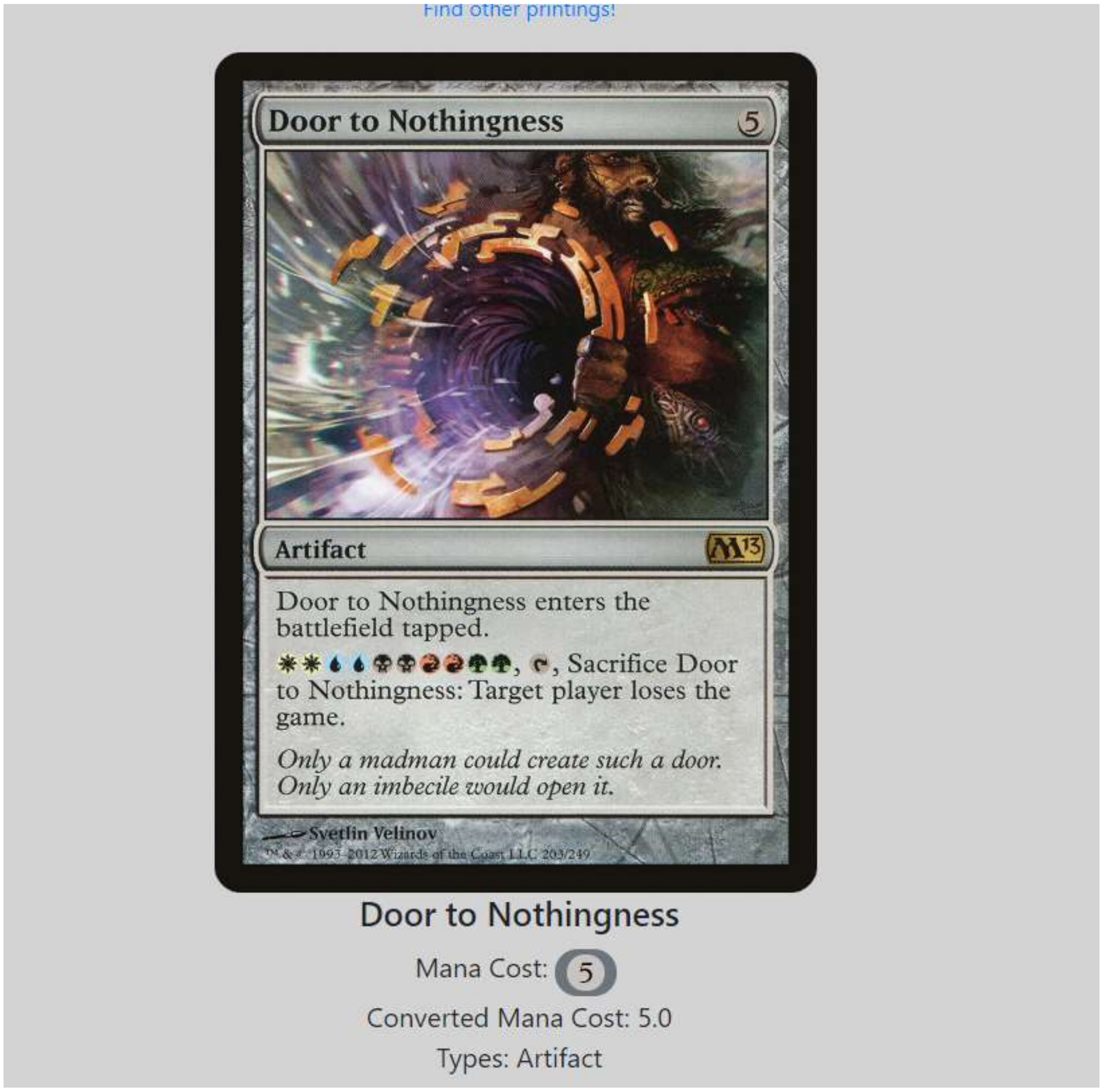
The Advanced Search form. Some fields have placeholder values to indicate to users what attribute of a card the field represents.

Feature Completion and Styling

With the ability to find cards now complete, and the functionality associated with decks being completed by Devon, the majority of the major functionality of the site was completed. Of the remaining tasks, providing access to MTG blog posts and YouTube content became the next focus. Devon and I split this task, leaving the responsibility of handling YouTube content in my hands. To accomplish this, I selected a small number of YouTube channels known for their MTG content. I created a list in which each element was another list containing a plaintext representation of the channel name and the actual url extension for that channel. The template simply iterates through the list describing the channel with the plaintext and embedding a video player to the channel using the url extension. These were organized in a Bootstrap accordion in which only one video player can be visible at a time.

One of the other tasks we had discussed over the course of development was a large restyling of the site. I volunteered to take on this task due to some in-depth prior experience with CSS and Bootstrap in particular. One of my first tasks was cleaning up the Card Details page. Up until this point, this page displayed only the card image and the JSON of the card. This was useful for development, though never our intended final product. I began by identifying some useful attributes to display in the page, such as the oracle text, power, toughness, typeline, and more. These were then extracted from the card JSON and formatted in the Django template.

One of the more interesting aspects of this task was incorporating a card's costs into these formatted fields. Cards in MTG often have costs associated with them, whether it be to play the card or activate one of its abilities. These costs are paid using a resource called *mana* which occurs in a variety of forms, most prominently colors. Mana is represented on a card through a pictogram. In the oracle text, card mana cost, and other fields, these are often formatted in brackets (i.e. "{W}" representing a cost of one white mana). While this is useful in identifying the costs, these fail to accurately portray the symbology of the game, which is usually easier to read than a card’s ability that costs {W}{W}{U}{U}{B}{B}{R}{R}{G}{G}.


Find other printings!

Door to Nothingness 5

Artifact

Door to Nothingness enters the battlefield tapped.

, , Sacrifice Door to Nothingness: Target player loses the game.

Only a madman could create such a door. Only an imbecile would open it.

Svetlin Velinov

™ & © 1993-2012 Wizards of the Coast LLC 203/249

Door to Nothingness

Mana Cost: 5

Converted Mana Cost: 5.0

Types: Artifact

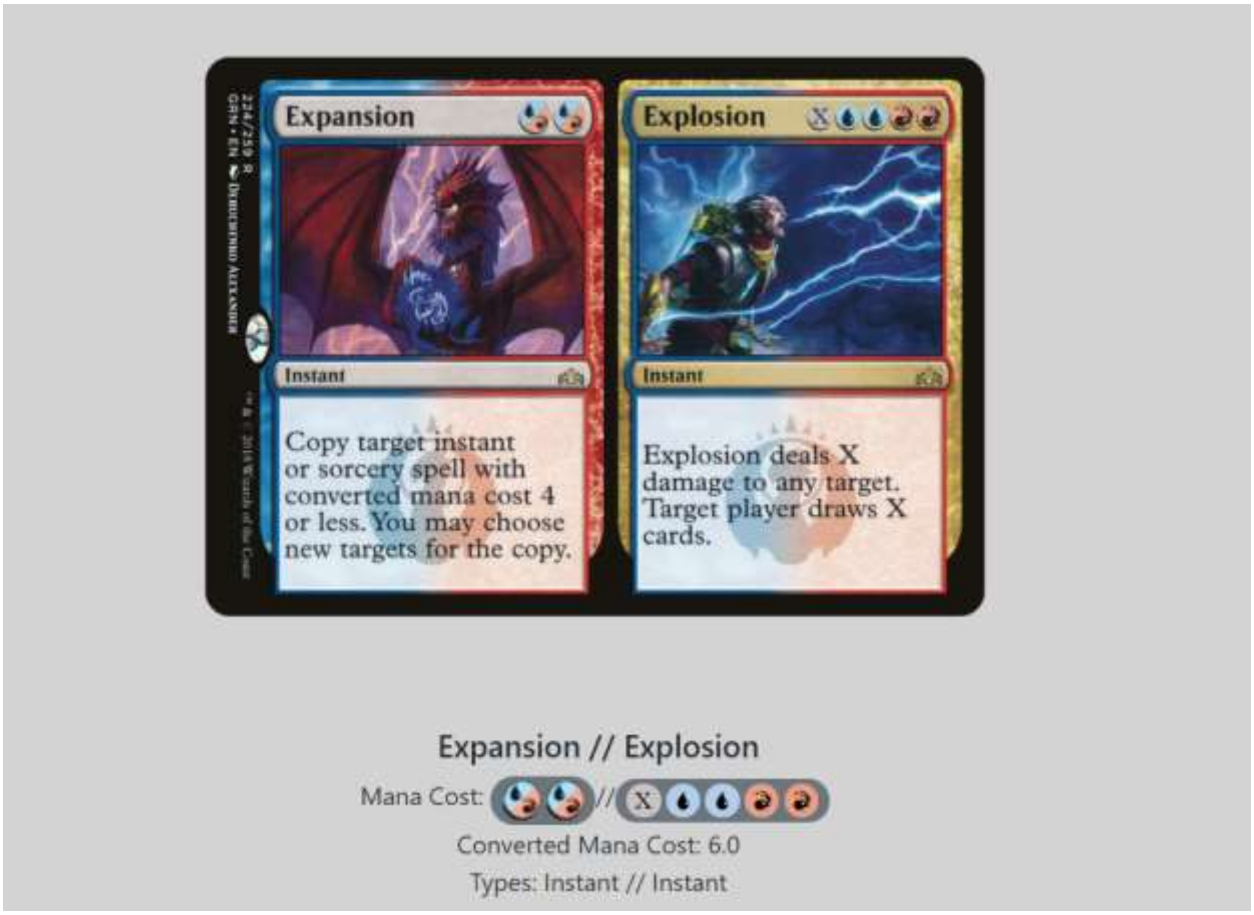
Door to Nothingness highlights how the above cost appears more intuitive to a player using the game’s symbology than the corresponding text does.

To incorporate the game’s symbology into WUBRG, some research led us to two GitHub repositories by Andrew Gioia, *Keyrune* and *Mana*. [5] These repositories provide access to the

[5] https://github.com/andrewgioia

symbology used in MTG and are available for use with almost any web-enabled service. Following the documentation, both *Keyrune* and *Mana* were incorporated into WUBRG through their respective CDNs over jsDelivr. Both of these provide their imagery in a format that can be accessed through classes applied to `<i>` tags. With these new assets incorporated into the site, after recoloring some of them through CSS, work began on displaying fields with this symbology instead of the text replacements.

The two fields that needed to be formatted for these attributes were mana cost and oracle text. To format the mana cost, my first intuition was to simply perform string replacement in Python on any strings that contained an existing symbol in brackets. This, unfortunately, encountered some issues with edge-case cards such as multi-faced cards like "Expansion // Explosion" (below) which not only has two faces, stored as two lists of values, but also has hybrid mana costs on Expansion formatted as "{U/R}". Hybrid mana costs are not currently supported by *Mana*. In order to address these issues, I extracted all symbols in mana costs contained between braces and stored them in a list. Then, when rendering, the template iterates through the list, applying *Mana* symbology to any applicable characters, and using external imagery to display any other symbols that are not supported by *Mana*.


Expansion
Instant
Copy target instant or sorcery spell with converted mana cost 4 or less. You may choose new targets for the copy.
Explosion
Instant
Explosion deals X damage to any target. Target player draws X cards.

Expansion // Explosion

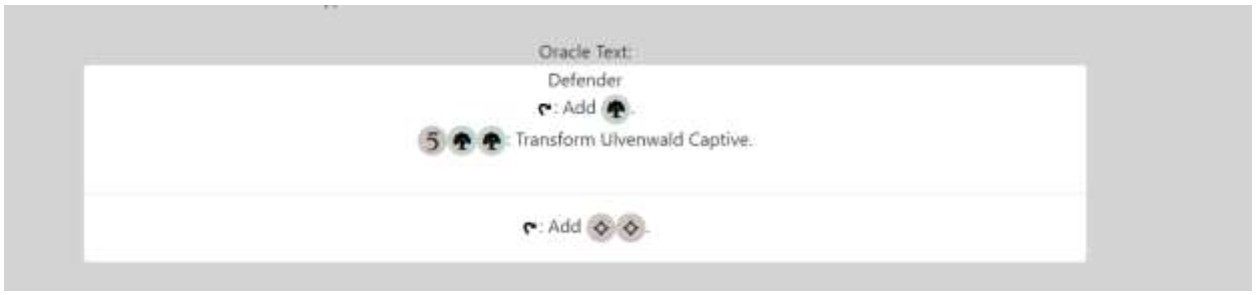
Mana Cost: //

Converted Mana Cost: 6.0

Types: Instant // Instant

Expansion // Explosion highlights the rendering of multiface cards and hybrid mana costs.

Formatting the oracle text for cards that featured a cost in their textbox followed after this. This process, fortunately, proved to be much easier, as all oracle texts were simply text values, which could be altered more easily with Python's string replacement functions. For cards that featured cost values on different faces, these were simply joined with a `<hr />` tag after being replaced with the appropriate symbology or imagery where applicable. Beyond these minor issues, the remainder of styling was simply a process of repeatedly altering CSS and ensuring the changes behaved as desired. These changes were among the final alterations to WUBRG.


Oracle Text:
Defender
: Add .
5 : Transform Ulvenwald Captive.

: Add .

The oracle text for "Ulvenwald Captive // Ulvenwald Abomination", a card featuring symbology on both faces.

Conclusion and Looking Forward

Working on WUBRG was an incredibly satisfying venture. MTG, being an interest of mine, encouraged me to work towards developing a tool that I, as a player, would want to use. Being able to see cards that I have used to success in real games appear in my own project was encouraging and kept me working throughout the project. Working with Devon was also a very insightful aspect of the venture. The majority of my other undergraduate coursework was done independently or required minimal work with others. Having a partner as immediately focused on my work required me to be able to communicate more aspects of the project more exactly to ensure that we both had a complete understanding of the project. It also provided me a guide when working with technologies with which I had limited use prior to beginning WUBRG, such as Git and GitHub.

Given more time, there are a number of additional features I would like to include in WUBRG. Our initial proposal incorporated a companion app, which is something I would still be interested in developing. Features such as game tools, including random number generators, dice, counters, and other resources would also be interesting to incorporate. The game is subject to a steep learning curve, so developing more learning resources is also something I feel would be of use. In addition to this, there is also a casual lexicon that only experienced players become familiar with, such as “storm” referring to a deck archetype that wins suddenly in one turn, or “bear” referring to any creature with power, toughness, and converted mana cost each equal to two. Providing a shortcut to this type of information would only help to further the type of information accessible through WUBRG.