Devon Brown
Senior Project Paper
Dr. Jeffrey Jackson

WUBRG: Web Application for Magic: The Gathering Card Game

https://github.com/devonb946/WUBRG

https://wubrg-mtg.herokuapp.com

Background

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 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. 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 are considered. 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, share decks with others, and much more. WUBRG is built using the Python Django web framework and leverages existing MTG APIs to provide a fast, efficient, and up-to-date service for users. WUBRG stores MTG card data accessed through these APIs in a SQL server to prevent a

network dependency and to provide a stable, 24/7 access to a large database of cards. 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 connected to the deck, and statistics on the deck's structure become available to the user on a web page for the deck.

In WUBRG, users can build decks in an incredible number of ways: by comparing them to or altering existing decks, searching to find builds centered around a specific card, or walking through several prompts to help gauge a player's playstyle after which they are provided different suggestions for which direction to take. After a deck is built in WUBRG, it can be shared for others around the world to see. Then, the user can export their deck list to take to their local game store for purchase in person or be directly transferred to any of the most prominent card markets online. In addition to tools provided for deck building, WUBRG also provides access to different YouTube videos and deckbuilding articles written by prominent players, tools to use while playing such as a life total tracker, dice for randomly determined events, and other such utilities.

This project was developed as a team effort between two people. The developers are Nikolas Schmidt and I. We had divided all project tasks among us and each spent our time working on mostly separate features. However, we frequently helped each other and collaborated in certain aspects of the project that commanded a consensus, like design and decisions on which technologies to use.

Design

Before any focused programming was done, a feature-rich site like WUBRG commanded some design work. Nikolas and I knew we were going to be working on different sections, but it was still very important that we stayed on the same page when it came to implementation. The first step in the design endeavor was to draw up a viable sitemap for the website. This way, we would know how to structure the Django app accordingly without any confusion. As an additional measure, the URL paths were specified on each page of the site to set up a URL structure. Of course, this was subject to slightly change throughout the development process, but it stood as a good scaffolding to start.

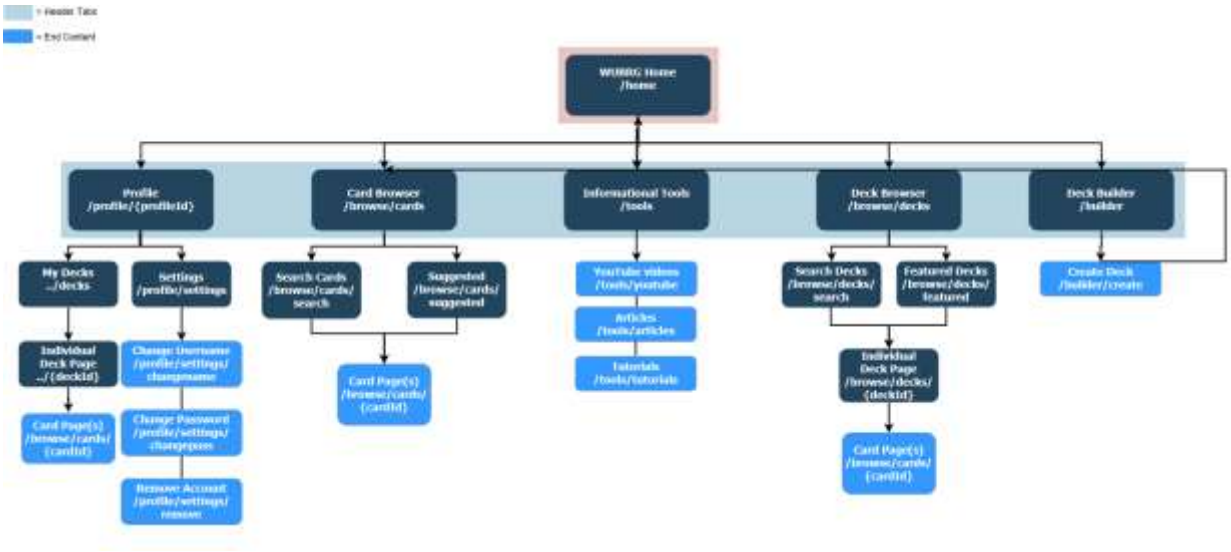

Figure 1: Sitemap for our website, detailing the main header tabs and their subsections

Development Process and Collaboration

Because of my recent background in a software engineering environment as an intern and the fact that this was a larger-scale team project, Nikolas and I followed an agile scrum software

development process. What this means is that the entire project was broken into three-week portions, called sprints. Each sprint contained a predefined set of tasks that was expected to be finished by the end of that sprint's deadline. For sheer convenience and consistency, we aligned our sprints with the due dates of the incremental senior project papers. Prior to each sprint, we sat down together to plan the tasks we set out to do in that timeframe. We made sure to clearly define each task based on the timeline given in the proposal paper, so that the development process stayed on track and continued in the direction we originally planned. A popular component of scrum agile development is a Kanban board, which is a graphical board that tracks issues as their status progresses. As an issue entered development, we would update the board to reflect this new development update. This way, Nikolas and I were able to constantly know what we were working on and how it affected all the other work.

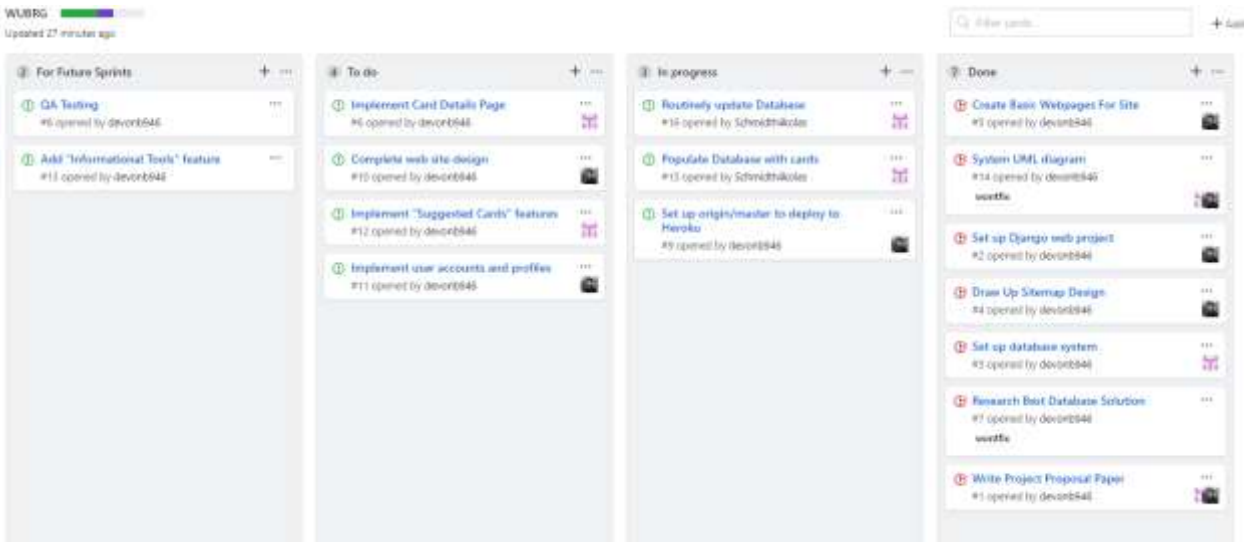

Figure 2: Our Kanban board on GitHub, which helped us track issues for each sprint

In addition to Git, we also frequently talked over Internet chat as well as occasionally meeting in-person if serious collaborative work needed to be done. This form of communication did not hinder our development process, and it allowed us to work on our own schedules. There would be common times where we had overlapping long periods of time to develop, and this is when most of the design, planning, and technology usage discussion took place.

There were several unknown variables in the process of developing the application at the start, so there had been some unexpected twists and turns taken to get to the desired result. For example, we knew that the final product was to be deployed on a production server, but price limitations and lack of knowledge on proper solutions delayed the final decision of any kind of deployment until after the first sprint. The resulting decision was a cloud hosting solution called Heroku, which was set up even before the project was finished. We figured that setting up a production environment prior to the final sprint would give us ample time to iron out any issues. As a plus, Heroku gave us the convenience to set up our project with our GitHub project, our primary resource for collaboration.

In the software development world, Git reigns as a golden standard for version control in software projects. It allows for continuous collaboration, branching, and convenient incremental stamps which you can revert to if the current project has a critical issue. GitHub is a Git-based repository hosting service that allows for hosting code bases for any kind of project. With a student account, the platform becomes even more robust, allowing for private repositories and a very large collection of services with free credits to start – Heroku was among this collection, which is another reason why we chose this service. With Git, Nikolas and I were able to simultaneously collaborate on the project by committing our individual changes and pushing them to the private repository GitHub hosted for us.

Technologies

In addition to these supporting technologies, the primary technology stack used to create WUBRG comprised of Python's Django web framework, HTML Django Templates with CSS for styling, JSON for data representation, and PostgreSQL for the database management system. Heroku is the cloud platform that runs our production application. We chose this stack because of our previous experience with Python and the ease of developing with it, as well as the solid compatibility of the supporting technologies with the Django framework. PostgreSQL is an old but popular database management system that is fairly easy to use and plays nicely with many different kinds of applications. The PostgreSQL database, provided by Heroku, was easy to configure in the Django settings; the information we planned to store in the database was also easy to convert into relational models.

Django Application

A big feature that I worked on during the project's development was the Django application. After designing the sitemap shown on figure 1, I set out to create a Django project that was divided into modular components that suited our needs according to the design. I planned the sitemap so that it had several main sections. For instance, the diagram shows a section for card browsing, a section for a user's profile, and a few others. Because the design was made out this way, it made sense to create several modules inside of the Django application to reflect the major portions of the website. Within these portions include pages and actions that pertain to that specific feature.

Inside of this application, many settings had to be configured, including the database, the application modules, and the URL for serving static files like CSS. In addition to this, I began to set up all the URLs in each component. Each URL defines a page or an action on the website, as dictated by the sitemap design. For example, the URL of the site that ends with “tools/youtube” is responsible for retrieving the information on YouTube that pertains to the “informational tools” feature.

```
urlpatterns = [
    path('', views.index, name='index'),

    # builder paths
    path('create/', views.create, name='create'),
    path('add/card/<uuid:card_id>', views.add_card, name='add_card'),
    path('remove/card/<uuid:card_id>', views.remove_card, name='remove_card'),
    path('add/deck/<uuid:deck_id>', views.copy_deck, name='add_deck'),
    path('follow/deck/<uuid:deck_id>', views.follow_deck, name='follow_deck'),
    path('remove/deck/<uuid:deck_id>', views.remove_deck, name='remove_deck'),
    path('unfollow/deck/<uuid:deck_id>', views.unfollow_deck, name='unfollow_deck'),
    path('validate/deck/<uuid:deck_id>', views.validate_deck, name='validate_deck'),
    path('update/art/<uuid:card_id>', views.update_art_card, name='update_art'),
    path('add/card/massentry', views.mass_entry, name='mass_entry'),
]
```

Figure 3: Django code to specify the URLs for the deck building module of the application

The HTML pages were the next step in this application, so that the URLs that were created to show a page could do so. These pages were very basic to start with, but they served as a starting point for each feature of the application. Django makes use of a unique type of HTML,

called templates. These templates are normal HTML mixed in with Django variables to support a more dynamic kind of page.

```
<p>Unfinished deck: {{deck.is_draft}}</p>
<p>Creator: {{deck.creator}}</p>
<p>Format: {{deck.format}}</p>
<p>Colors: {{deck.colors}}</p>
<p>Date Created: {{deck.date_created}}</p>
<p>Number of cards in deck: {{deck.card_count}}</p>
```

Figure 4: When deck information is passed into the Django template, it can be rendered in the HTML.

User Accounts and Authentication

After the initial project with basic pages had been created, a feature that I worked to implement was the handling of user accounts tied to the website. One of the Django applications created was specifically meant to handle everything that dealt with user accounts, registration, logging in, and viewing information tied to an account. For example, if a user were to build a deck using the "builder" application, it would tie that specific deck to that user. After building the deck, the user can view it on their profile.

Thanks to Django's account-related, premade forms and views, I found the "Register" and "Log in" pages on the website very easy to implement. Although I could have created my own forms and custom user model, Django's premade code for this was everything that I needed.

Even though I had to create template pages to log in and to register, the forms to submit this information could be passed in through the view context. While inside a template, a form can be set to render as paragraph text, a table, or as a list. As a bonus, the web application comes with a form of web security known as Cross Site Request Forgery (CSRF) protection already activated in the middleware. CSRF is a type of attack that uses a user's credentials in the form of links on other websites to perform unintended actions. By including a small line of code in the form template, a token is used to prevent CSRF attacks.

```
<form method="POST">
  {% csrf_token %}
  {{ form.as_p }}

  <input type="submit" value="Login">
</form>
```

Figure 5: Template code for the form to log in, using CSRF tokens and Django's pre-built form. The "form.as_p" statement tells the app to render the form as paragraph text.

After a form for the user is submitted, the view handles the processing. Because I am using the pre-built forms and user model, it made sense to also use the pre-built views supplied by Django. Based on the settings for the application, the account information is stored in the database. To handle password encryption for storage, Django employs the PBKDF2 (Password-Based Key Derivation Function 2) algorithm along with a SHA256 hash. The PBKDF2 uses the secret key defined in the settings to encrypt the password string. According to the security documentation on Django's site, this method offers plenty of security for any normal application.

Since the default database in the settings is the PostgreSQL cloud database, the information is stored there. If you were to run a query on this table, all user information tied to an account would be shown, apart from the password, which would look like a very long string of random characters.

Deck Builder

A main purpose for giving users accounts and access to all of Magic: The Gathering's cards is to provide them with the necessities needed to build decks. Deck building is a very important, yet time-consuming aspect of the hobby; there are countless matchups and card combinations that make up a deck. Competitive players are very picky when it comes to card selection, so a good deck-building feature must excel at editing cards and displaying information. With our application, we hoped to deliver a deck-building experience that was informative and easy to perform so that users may create any deck to their liking.

The first step to create this feature was to edit the user account model to have a field to hold decks, since I planned to bind decks to accounts. Thanks to Django's model fields support, I was able to add a "many-to-many" field, called "decks," to a new custom user model that inherited the base model I had created previously. "Many-to-many" suits the "decks" field well because it means that a user can have many decks, and a deck can belong to many users. This enables the social functionality of users copying or following other users' decks.

With the deck builder, there are five main operations: create a deck, add a card to a deck, remove a card from a deck, add/copy another user's deck, and remove a deck from a user's account. When a user creates a deck, they must fill out a form, the "DeckCreationForm," that is

generated by custom Python code and links to the actual “Deck” model. Many of the fields in the model are hidden from the user and cannot be typed in – for example, the “date created” and “deck owner” are automatically assigned when that specific user clicks the submit button on the creation form. However, there are some required fields that the user needs to fill out, such as the name, playing format, and description. After the user clicks submit, the deck will be bound to their account until they decide to remove it.

```
class DeckCreationForm(ModelForm):
    class Meta:
        FORMATS = (
            ('Standard', 'Standard'),
            ('Modern', 'Modern'),
            ('Commander/EDH', 'Commander/EDH'),
            ('Legacy', 'Legacy'),
            ('Vintage', 'Vintage'),
            ('Brawl', 'Brawl'),
        )

        model = Deck
        fields = ('name', 'format', 'description')
        widgets = {
            'format': Select(choices=FORMATS),
            'description': Textarea(attrs={'cols': 50, 'rows': 5}),
        }
```

Figure 6: Python code that defines the deck creation form that is generated. Popular playing formats are predefined and are selectable from a drop-down list.

Within the deck model, there are several fields describing the deck and the cards within it. The most important field is the "many-to-many" field that holds a deck's cards. Since a card has a unique identifier, there was an issue with adding multiple copies of the same card to a deck: if the card was already present in the deck, it could not be added again. To circumvent this, I created an intermediary model for a card that belongs to a deck, called a "DeckCard." This new model allowed me to add in an extra metadata field that gave the count of copies of any given card in a deck.

After a deck is created, the user is directed towards the "browse" section of the site where they can add cards to their deck. On an individual card's page, a user can click the "Add to deck" button next to a drop-down list of their decks to specify which deck the card will be added to. The user will also find that the deck is now attached to their list of decks on their profile page. The art thumbnail of the deck is determined by the first card of the deck to give a visual representation of the deck when browsing.

Clicking on a deck will take a user to a page detailing that deck's information, including the description, colors of its cards, the number of cards in the deck, and a list of its cards. If the user viewing the deck is the one that owns it, then they will be permitted to delete cards in the deck from the list or remove the deck from their account. If a user is not the owner, but is logged in as another user, then they will have the option to add the deck to their account. Two options become available when adding the deck to another user's account: they can either create a copy of the deck with the option to edit it, therefore diverging from the original deck, or "following" the deck by adding the original deck without the option to edit it but receiving further updates from the original owner.

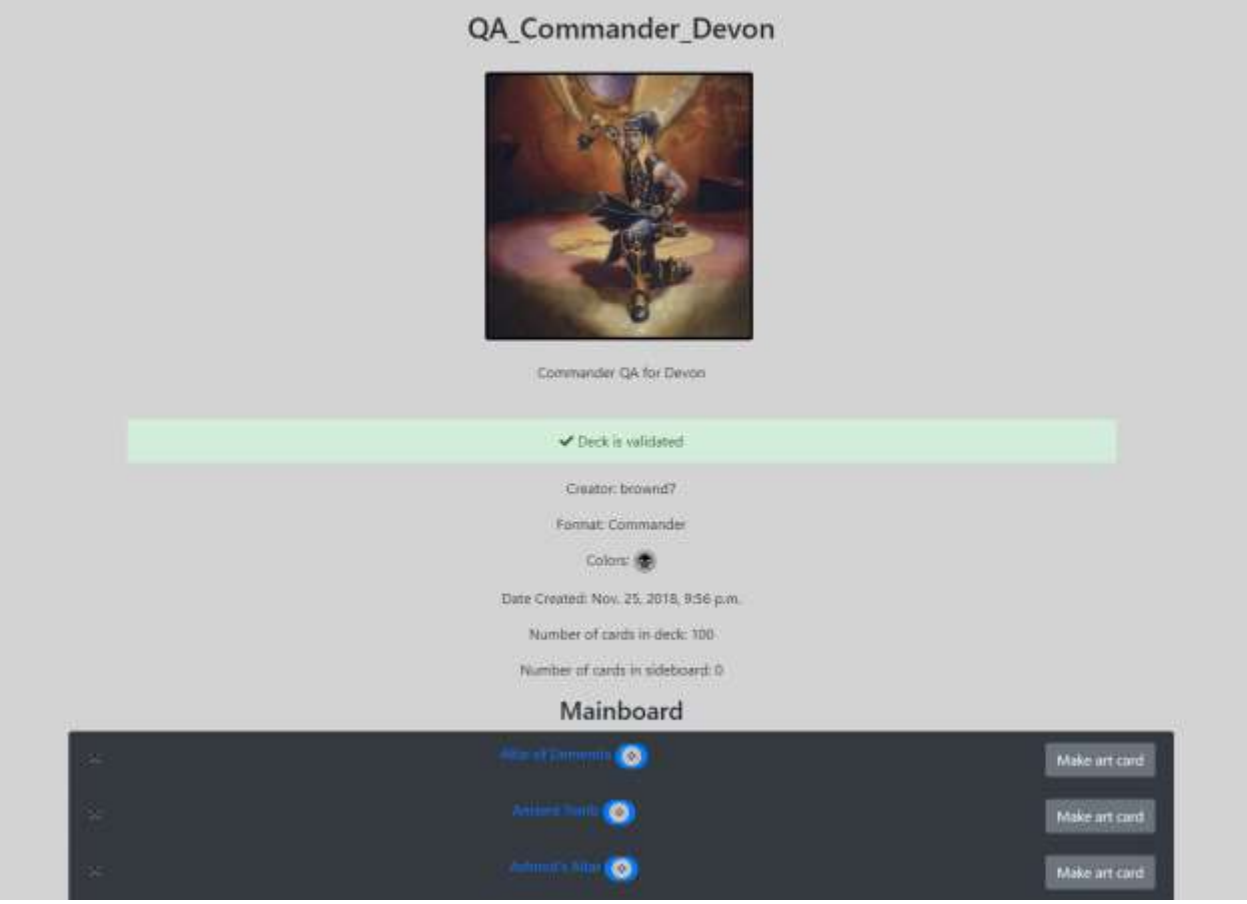

Figure 7: Example of a page showing a deck's details, with the bottom showing the top of the list of cards in that deck

After the simple operations of deck building were developed, the need arose for a better way to add multiple cards at once to a deck. After all, decks usually require a minimum of 60 cards to be considered playable. Inspired by other deck-building sites, I built a text-based mass-entry tool for adding several cards at once. To use this, a user will enter in a card per line in a text box and then click a button to add all the cards. To help the user out with correct card names, a search box is provided at the top with autocomplete functionality to insert the card in the

correct format. Additionally, the user can also specify a special card for their deck called a "commander," which is used in some formats of the game. Before the actual card name, a quantity can be added to add multiple copies of the same card in the deck.

The code that parses the string in the text box and adds the cards goes through the input line-by-line and determines which options to enable. To look for how many copies of a card it needs to add, the code uses a regex pattern that looks for a number from 1 to 999, followed by the letter "x" at the very beginning of the line. If this pattern is not found, it only adds one card. At the end of the line, it looks for the unique string "__commander__" that tells the code that the card should be added as a commander card. Aside from these two checks, the rest of the line is queried by name in the database to check if a card with that name exists. If it exists, it is added to the deck; if it does not exist, that line is ignored, and no card is added. To avoid an overflow of messages if given a severely bad mass entry string (an example would be a plethora of random characters for at least a hundred lines), the user is not told about this ignored line.

```
# pattern is any number followed by x
pattern = re.compile('[0-9]+x$')
for line in cards_text_lines:
    quantity_candidate = line.split(' ')[0].strip()
    # check if the word we pulled back is actually the quantity
    # and not the first word of the card name
    if pattern.match(quantity_candidate):  # can't input more than 999 of a card
        quantity = int(quantity_candidate[:-1])
        card_name = ' '.join(line.split(' ')[1:]).strip()
    else:
        quantity = 1
        card_name = line.strip()
```

Figure 8: Python code snipped for grabbing the quantity and card name from a line in the text box

Deck Validators

In addition to giving users the opportunity to build decks on WUBRG, the full completion of a deck is marked when a deck is deemed valid. The deck model contains an attribute that determines whether the deck has been validated as a legal, playable deck for the format that it is in. Different formats have different specifications for legality. For instance, the minimum number of cards for a "Modern" deck is 60, while the minimum for the "Commander" deck is 100. Individual cards have a legality standing with each format as well; a card that is legal in "Modern" might not be legal in "Standard." To check every format requirement, I created code that runs through a deck and evaluates its validation status. This code mimics the rules for each format based on the official *Magic: The Gathering* rules.

Some formats have overlap with their rules, so the code has buckets of logic it can fall into. For example, if the format happens to be either "Commander" or "Brawl," then it checks the deck for a special card called a "commander." The first check of the validator makes sure that the card count for the deck is above the minimum. Then, additional validations include checking for a "commander" card and checking the legality for each individual card. The former will determine the commander card based on two specifications: whether the card object is marked as a commander, and whether a specific field marking on every other card matches that commander's same field. This field is known as the color identity of a card, and it could be any combination and subset of the five main colors that define the game's structure and playing styles. Checking the legality is much simpler than this step, and it is required for each format. Each card in the JSON data has a list of legalities for each format. The code simply looks for that format name in the list and passes if it sees that the value is "legal."

```python
def check_commander_identity(deck_cards):
    commanders = [card for card in deck_cards if card.is_commander]
    if commanders:
        color_identity = set()
        for commander in commanders:
            color_identity = color_identity.union(commander.card.data['color_identity'])
        color_identity = list(color_identity)
        for deck_card in deck_cards:
            is_within_identity = all(color in color_identity for color in deck_card.card.data['color_identity'])
            if not is_within_identity:
                return False
        return True
    else:
        return False
```

Figure 9: Python code to check for commanders and color identities. It is possible for a deck to have multiple commanders in some edge cases where cards state this exception.

After pressing the validation button, the user will be taken to either a success or failure page. Returning to the deck, they will see that the red information box will now be green with a message explaining that the deck is now valid. Going to the “Browse All Decks” page on the website, the user will now be able to see their deck in the list. This browsing page is restricted to decks that have been validated to ensure that invalid, and possibly empty decks don’t heavily dilute the decks that are completed and more preferably browsed. Regardless, a user can still find invalid decks through the “Search Decks” page by specifying in a checkbox that they would like to include invalid decks in the search.

Informational Tools

One of the last major features added during WUBRG's development process was the Informational Tools section of the site. Originally, the feature page had been planned to feed in a lot of external information from the wide span of *Magic: The Gathering* sites on the Internet, but this idea fell short when there was no acceptable way to provide information from several different sites consistently without explicitly copying information that was already on another website. To add to this, many sites do not have APIs that allow for open communication and useful data retrieval. To circumvent these shortcomings, Nikolas and I added two major sections on the page that we hoped would provide users enough information to dig into the information surrounding *Magic: The Gathering* to learn the basics of the game, stay up to date on the latest news, and to explore prominent figures in the community.

The first content column on the page is a plugin of the official *Magic: The Gathering* RSS feed; this feed is run by Wizards of the Coast, the publisher of the card game. Initially, I sought to use an external widget to inject the feed into the site. However, all the free widgets that I found either heavily restricted customizability or they added additional, unwanted content. To take matters into my own hands, I developed a small RSS feed parser on WUBRG's "Informational Tools" module.

RSS feeds are created in an XML file format, which is a way of holding data in a syntactic structure. Because RSS feeds all follow the same conventions, it is easy to interpret the contents of one. After examining the XML file for the feed, I determined which info I wanted on the site and which to exclude. This way, I had full control of the display of data from an external source. Every RSS feed uses the same element to describe entries, which is called the "items" element; each entry in a feed is like a new post on a blog or a new journal article. On the

“Informational Tools” page, a scrolling content box shows each entry, and a user can click the arrow buttons on each side of the box to navigate through newer or older entries. The code on our Django application is linked to this RSS feed, so it will always update to contain the latest entries.

Thankfully, Python has an excellent library called BeautifulSoup for navigating through XML files; these files have a tree-like structure in which a root node contains child nodes, grandchild nodes, and so on. The code I have written will initially grab each node (element) called “item” and look through its contents. The meaningful data inside of “item” that is displayed on WUBRG includes the title of the entry, a link to the full entry on Wizards of the Coast’s website, a preview of the contents of the entry, the creator of the entry, and the date it was published. It was easy enough grabbing this information and storing it on Django, but some information needed to be manipulated to be displayed correctly on our website. For example, the links provided within the entry were relative, so they could not take a user to the original site unless the absolute path was specified. For this, the inner HTML of the description node needed to be edited with BeautifulSoup to include the complete link back to the entry. This involved traversing the nodes inside of the HTML until the element for the link was found, manipulating the inner string, and reassembling the full HTML for storage. After this is done, the contents are packaged up and displayed on the Django template in a clean format.

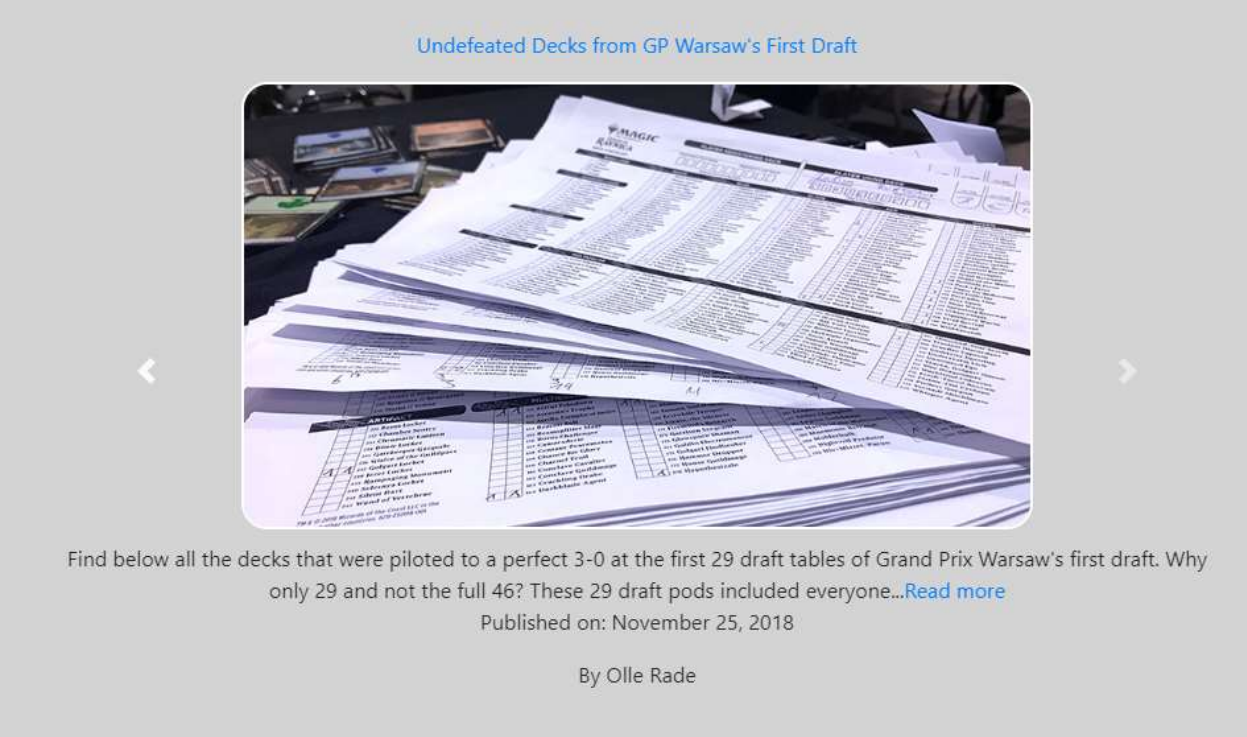

Figure 10: Example of a rendered RSS feed entry on the page

```
description = ''.join([str(d) for d in soup_item.description.contents]) if soup_item.description else None

# open up the inner HTML and edit the hyperlink for "Read more" before we parse it
# since WotC likes to use relative links
html_soup = BeautifulSoup(description, 'html.parser')
anchor = html_soup.find('a', {'class': ['cta', 'learn-more']})
if anchor:
    anchor['href'] = 'https://magic.wizards.com' + anchor['href']
    anchor['target'] = '_blank'

# use our newly cooked soup for our custom made description
item.description = ''.join([str(d) for d in html_soup.contents]) if html_soup else None
```

Figure 11: Python code to extract the description from the RSS XML, feed it into an HTML parser, edit the attributes, and then stitch up the data for storage

In addition to the RSS feed that gave some valuable information about the latest events and entries about the game, we wanted to give users another media type that would help them learn the aspects of the game and how to build the best decks. To this end, Nikolas led the development effort to integrate *Magic: The Gathering*-related YouTube videos into the second column. Although we only offer six user channels on the page currently, we hope to add more as we discover more content creators that provide informational videos. These channels are hard coded on Django and they are sent into the template to render a playlist of the channel's videos, starting with the most recent.

Deployment

Around halfway through the development of the application, WUBRG was deployed to a production environment in Heroku, a service that can run applications for you on a dedicated cloud machine. The setup to deploy on Heroku involved a number of tasks: configuring the application settings to be production-friendly, creating commands to handle running the application in the new environment, and setting up our GitHub master branch to automatically deploy after a push. After doing so, I was able to verify the result on Heroku's site where the application's build and run status is shown. If your app does not have a custom domain name, Heroku supplies a pre-set one for you based on what you called your built application. Clicking the link takes you to the cloud-hosted application.

Thanks to the credits received for registering as a student on GitHub, I was able to use these towards a Heroku package that included a 24/7, private cloud machine called a dyno.

Dynos, as defined by Heroku, are environments that contain everything needed to run and host an application. This method of hosting, called containerization, has also been popularized by other services such as Docker. The end result is a lightweight, simplified, isolated environment that contains an application with minimal hassle. The plan that we chose to host our app on is the "Hobby Dyno," which offers 512 MB RAM, metrics, and 24/7 uptime for small scale projects.

Even though an initial deployment was successful, two new challenges arose. The first challenge was creating an environment-specific settings file for Django. Within the settings.py file in the application, there existed settings set only for a development environment. For example, a debug setting was set to "True," which definitely should not be the case for a production environment. In addition, the file also contained sensitive information such as database connection credentials and a secret key used for security and hashing. Of course, this could have been quickly remedied by changing the settings file for only the master branch, but this would have restricted the file from being used in version control.

To fix this, any information that depends on its environment was removed from the file and made into an environment variable. Heroku's app management interface allows for the creation of environment variables that live in the dyno. After the production variables were specified, it then became a task to create the same variables on our local machines. On both Windows and MacOS, this is not too difficult of a task; both operating systems offer ways to add environment variables. The local machine variables represent settings used for development. Finally, after configuring our local machines, the Python code was modified to read environment variables instead of hard-coded values.

```
# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = os.environ.get('WUBRG_SECRET_KEY')

# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = os.environ.get('WUBRG_DEBUG')
```

Figure 12: Python code in settings.py to retrieve environment variables

The second challenge after deploying the application was selecting a database to replace SQLite, our original storage solution. Because Heroku uses an ephemeral file system, and because SQLite is a file-based database store, any sudden change or deployment in our application would erase the entire database. Luckily, Heroku offers many add-ons to a dyno. We had a broad selection of solutions to pick from and settled on using a PostgreSQL database provided by Heroku to host our database.

Although this feature came with a cost, the existing credits on my account were enough to cover both the database and the dyno for more than enough time. The database plan we selected for our project allowed us to have ten million records, which was more than sufficient to store all the cards and support a potentially substantial number of users and decks. Because Nikolas took a majority of the responsibility for handling the database, this issue was handled primarily by him. Regardless, this was a production-related issue that was luckily solved in time because of our eagerness to deploy our application before it was fully developed. It has surely been a lesson learned to deploy sooner rather than later to avoid serious, system-wide issues right at the deadline.

Bug Fixing

Because our application requires a larger technology stack then we are used to working with, it was only natural that bugs would appear in our system. From the database side to the front-end side, various bugs appeared in many forms. For example, the script to update the database broke when I added dependencies to the card model on the deck model. Our method of dropping the card table and starting anew with a fresh, updated table did not work anymore since the table was depended on. Therefore, it had to be rewritten to use one table; the script would then update values if they already existed or insert new ones if they did not.

On the front-end side, we encountered various bugs with user registration and accounts after the creation of the new custom user model. The new model required its own form and an update to the settings to point to the new model as the application's default user model. In addition to these bugs that I fixed, Nikolas dealt with many other bugs as well. The takeaway from this development endeavor is that whenever new things are added, it is always essential to integration-test the system to verify that the additional content did not break the old content.

Future Development Plans

Even though WUBRG's original design has been developed to completion, Nikolas and I have discussed ideas about additional functionality for the application post-release. The other deck-building *Magic* sites that exist on the Internet such as TappedOut and deckstats.net have given us inspiration to drive towards a more feature-rich application to provide the best deck-building experience possible in a Django project. As we move forward, the development

endeavors we have considered include social functionality for users, comprehensive deck analytics, and more ways to assemble and disassemble decks quickly.

For the social aspect, we would like to implement a way to comment on decks and cards, connect users through friend requests, and view other users' pages. This way, users will not only benefit from building decks on our site, but they will also get to hear opinions from the community about various content. Allowing friends on user accounts will also allow for close social interaction, which is a big feature on other deck-building sites.

The second feature, the deck analytics, would look more closely at the contents of each card in a deck, and form some pleasant visuals for a user. For instance, our current site presents the current colors in a deck. However, it does not go into detail about the percentages of each color, nor does it compare the amount of non-land cards to land cards, which is understood by any seasoned deck-builder as a very important aspect of balancing a deck.

Currently, the only way to remove a card from a deck is to click the "X" button on the deck details page. Going forward, I would like to implement a way to "mass remove" cards, just as there is a way to "mass add" cards currently. For a web application that serves thousands of *Magic* cards, I also believe that there should be more user-friendly ways to interact with the cards to build decks. There will always be room for improvement in implementing this, so taking a step to provide quicker ways to assemble and disassemble decks is on the future agenda.

Although WUBRG is currently supported through Heroku using free student credits, this will not always be the case. Soon, the application will have to either be served on Heroku or a similar site though payments, or it will have to be personally served. To ensure maximum efficiency and stability, the former is usually preferable. To that, there would have to be methods

to have the site earn this revenue. One possible method that is already partway implemented is the TCGPlayer button integration on all card and deck pages. This button will allow users to purchase cards or decks on TCGPlayer, a popular online marketplace for trading cards. This website allows external applications to apply for a partner code; if this code was added to our current system, it would give us a small percentage of the amount paid for the deck or card on their site.

In conclusion, WUBRG is a web application that serves as an efficient companion for interacting with *Magic: The Gathering* trading cards. Using a reliable and frequently-updated SQL database to hold a very large collection of these cards, users on the site can find the cards they need to build their dream decks. There is ample room for growth on the site to offer more features to users and to support the site's maintenance. As the site grows, we hope to provide a bigger and better service to those who wish to explore the world of *Magic: The Gathering*.