THE NIV DECKBUILDER

Matthew Spadafore

Overview

Magic the Gathering (MtG) is a popular trading card game where players construct decks of various types of cards and play against each other, and features various different formats of play. Niv is a browser-based MtG deck building and utility tool that uses the Scryfall API to enable various useful tools for the game, including searching, building a deck, and viewing stats about the deck. It features an account system in order to allow users to save their decks and view them at a later point.

The Scryfall API

Scryfall is a detailed MtG search website that allows for queries of all existing cards. Searching can be done via a simple text search, or advanced filters can be applied to add more specificity. The API for Scryfall allows for use of the search function via GET and POST requests, depending on the desired data. The Niv deck builder primarily uses two of the /cards API functions: search and collection. The search function takes in a properly formatted Scryfall query and executes it, returning back applicable cards. The collection function is used in a POST request, sending a dictionary of unique identifiers of cards, and then the card objects will be returned.

Classes

Niv features three main classes in its operation, as seen in Figure 1, representing the User operating the website, individual Cards, and the various Decks that the user constructs.

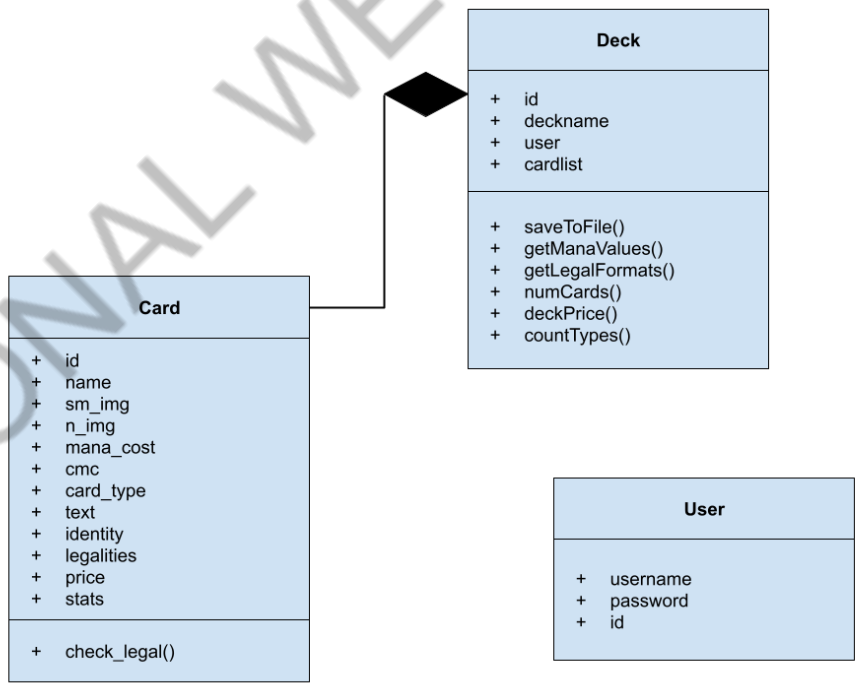


Figure 1. Niv Classes UML Diagram

User Class

The user class is a class used to track the current user of the website, and features a username and password, as well as an id. The user class is represented by the table 'users' in the niv.sqlite database, with the schema described in Figure 2. The id parameter is used to represent the primary key of the user. The username parameter is unique to ensure only one user with the same name can exist at once. This allows for correlation of decks with a user, as decks will be saved by using the user's username in the filename.

id	username	password
INTEGER, primary key	VARCHAR, unique	VARCHAR

Figure 2. User Class Database Schema

Card Class

To represent individual MtG cards in the deckbuilder, the card object is used, which stores both visible (on-card) and other information not accessible by simply viewing the card. The card class has 12 attributes, obtained from the information Scryfall returns. The id attribute represents a unique string of characters that is specific to each card, and is used in saving a deck as cards can be represented by this id. Name, as shown on point 1 in Figure 3, is the card name. Sm_img and n_img represent differently sized images of the card - normal images are used in the homepage to represent the random card, and the small image is used in the deckbuilder and search. The mana cost, seen as point 2 in the figure, represents the cost to play the card in game, and is seen as a number (for any color of mana) and then small icons (for specific colors of mana). Two other attributes relate to the mana cost: cmc and identity. CMC (converted mana cost) represents the total number of mana to play a card, and is the number in gray + number of colored mana icons. Identity represents all the colors of icon a card has, including in its cost and its text. Card type (point 3) specifies which in-game type(s) a card has. The primary types of card are Instant, Sorcery, Creature, Enchantment, Artifact, Land, Planeswalker and Battle, and a card may have one or more of these as well as various subtypes such as Dinosaur or Zombie. Text (point 4) represents the card's special abilities if it has any. Legalities is a dictionary that stores whether a card can be used in the many MtG formats (ways to play). Stats is a special attribute that has different values based on the card type. For creatures, as seen in point 5, they each have a power value and a toughness value (power/toughness), and for planeswalkers, they have a loyalty value, which represents a number of counters they are put in the game with. For other card types, this attribute will be empty. The method a Card object has is check_legal, which, given a format, will return True if the card is legal in that format, and False if not.

Figure 3. MtG Card Example

In deciding which aspects of the deckbuilder to represent with a database, representing cards was one of the first proposed ideas. However, this was ultimately decided against for a few reasons. First off, a few attributes of the card object may change. The text parameter, especially for older cards, is sometimes updated when a card receives a newer printing. More importantly, however, is the price and legalities attributes. Scryfall updates the price of a card daily, and cards could have drastic price increases for a variety of reasons. The web page does utilize price in an analytics section, so it was necessary to have the latest available updated price. The legalities section, also used in the analytics section, changes often, such as if a card receives a newer printing, is banned by a rules committee, or a set number of years pass (for the "Standard" format). In addition, storage of decks (discussed in the next section) is implemented by storing the id of each card, which can be POSTed to Scryfall as a single dictionary to receive the data necessary to build the Card objects. So, instead of making many queries to a database to get every card, it can be done with one (or two, for particularly large decks) POST requests. Storing each card only as its id (and quantity, in a separate file) also lowers the amount of storage per card compared to storing the full object in a database.

Deck Class

The Deck Class represents the deck a player has constructed for use in games of MtG. The id attribute of the deck class is used in the database as the primary key, and is not used otherwise. The deckname represents the name of the deck, which is not unique, as multiple users could have the same deck name - though deckname-user combinations will be unique (a user cannot have two decks with the same name). The user attribute represents which user the deck belongs to. The cardlist dictionary is how the deck stores the cards that it is composed of, by using a dictionary of {card object:quantity} to represent the deck construction. The Deck class also contains many methods, usually for determining statistics about the deck. The saveToFile() method creates files that the deck will be saved to in the /decks directory. The way that decks are saved in these files is by splitting them into two files: *user_deckname*.deck, and *user_deckname*.qty. The .deck file stores a list of unique card ids, and can be POSTed to Scryfall to return data about each card, and the .qty file stores the amount of each card in a deck. Note that Scryfall has a limit in the number of ids POSTed to it in a request, so the deck size limit is 143. The getManaValues() and countTypes() functions are very similar: each returns a dictionary with the number of cards in each value (either card type or converted mana cost). The numCards function is an auxiliary function used in these two (and other) functions, which counts the number of cards in the deck. getLegalFormats() goes through the cards and their legalities, and returns a set of formats the card is legal in. However, because there are a wide variety of formats, each with its own set of special rules, this function does not represent every single rule in format deck construction, only the legality of each card and the deck size. Finally, the getPrice() method goes through the price of each card and its quantity to determine the total price of the deck.

Storage of decks was a topic that required much thinking. Similar to Cards, Decks were thought of as something that could be represented by a database, as the important aspect of a deck is its card list. However, instead of representing a full deck in a database, it was decided to represent the deck as a more limited database, shown in Figure 4, and then save the deck into the .deck and .qty figures described earlier. When thinking about storing the full database, there were two options proposed. The first option was to have a decks table with an entry for each deck. This would have been simple to query and analyze, but was deemed not feasible, as storing the

list of cards would have been impractical for an entry in a single row. The second thought was to create a new table for each deck, and then have this table contain all of the cards each as an entry. This was also deemed impractical, as it would necessitate creation of many tables and navigation of using so many tables. By storing the data needed to access the deck files, the table is able to be very lightweight and allow for fast loading and saving of decks.

id	deckname	user
INTEGER, primary key	VARCHAR	VARCHAR

Figure 4. Deck Database Schema

Other Functions

In addition to the functions in the classes, there are various other functions used in the operation of the deckbuilder, which are contained in the niv_funcs.py file. The central function used in this file is the toCard() function. This function takes in the dictionary of data returned by Scryfall, and converts it into a Card object by obtaining the relevant information from the proper place. The random_card() function utilizes Scryfall's random card function, and returns a single random Card object. buildSearch() is a function that takes in a dictionary of responses from the Search tab form (described in the Web Page Layout section) and builds a query encoded in the way Scryfall uses to search. The scrySearch() function takes in an encoded query, writes a GET request to Scryfall using the query, and returns a list of Card objects based on what the query returned. The loadDeck() function takes in a username and a deckname, and finds the .deck and .qty files that correspond to the user/deck pairing, and loads it into a Deck object. This uses the Scryfall collections function, which returns a list of cards given a POST request of unique identifiers. Depending on the size of deck, this may take two requests, as each POST is limited to 75 cards at most. The getCardsList() function is used in this, as it is what is used to perform the POST request, returning the list of cards obtained. Finally, the fuzzy_search() function, given a card name, performs a fuzzy search on Scryfall: essentially, if the given string is enough to determine exactly one card, it will return that card. This function is used in the deck building page of the web site.

Web Page Layout

The first step in using the web site is the home page. The home page is a simple screen with information on how to use the deckbuilder, seen on Figure 5. On the left a random MtG card is depicted, which will change each time the user visits the page. On the right is a brief user guide, letting the user know the flow of the program. A navbar appears at the top, and is common to all pages. Each of the tabs of the navbar on the left side will direct the user towards that part of the site - though trying to create a deck will require being logged in, redirecting the user there, and trying to view analytics requires the user to build a deck first, redirecting them towards that. The Change Deck button will clear the current deck that a user is using (if any) and allow them to select another. The logout button will clear the user, requiring another login if the user would like to interact with decks. Figure 6 shows the initial planned layout of the homepage - in comparison to the final, they appear very close to identical, although the plan had a second text area under the left side image. In implementation, the overall description and how to use them

were rolled up into a single container on the left. The image was also chosen to be a random MtG card.

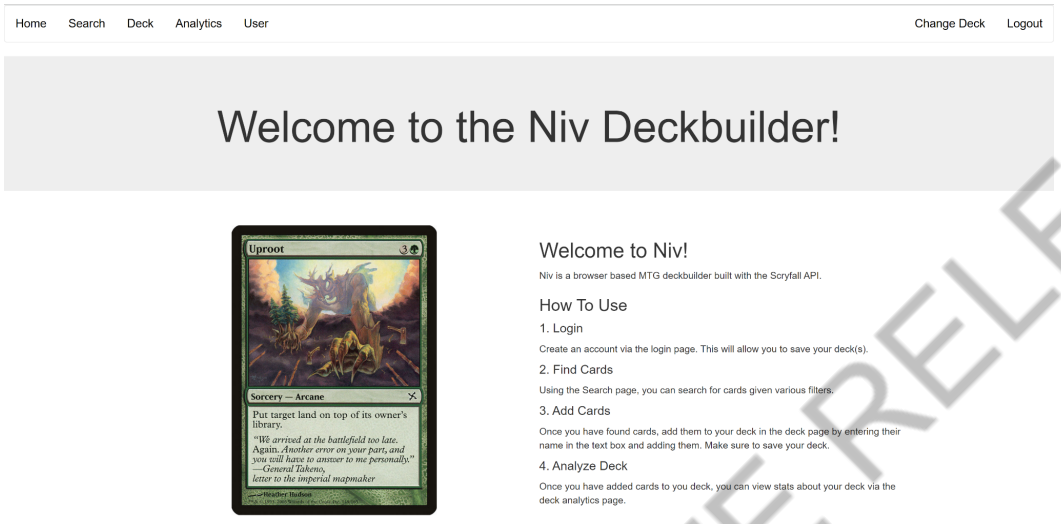


Figure 5. Niv Homepage

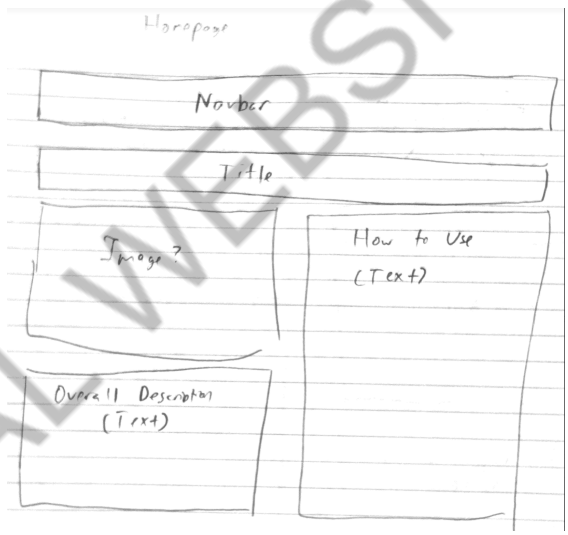


Figure 6. Initial Homepage Sketch

The next stage in the use of the web page is the search page (Figure 7). Initially, only the left side of the page will be populated. Here appears a list of filters that the user can apply, which correspond to different search parameters for cards. While Scryfall has a huge amount of ways to filter cards, the aspects chosen for this search feature consisted of the most popular and useful filters: searching by text, color identity, mana value, and card type. The user can put in values for any amount of these filters, and then a Scryfall search will be executed with these parameters. Behind the scenes, the query will first be built, and then a request will be sent to Scryfall, and the list of cards returned.

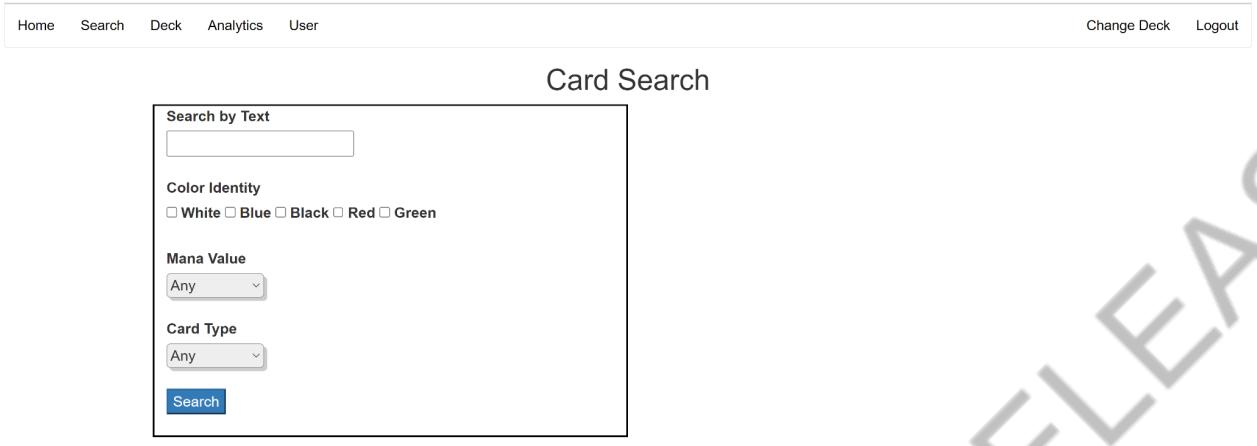


Figure 7. Scryfall Search Page

Once the search is executed, and the list of cards is obtained, the right side of the page will be populated with the list of cards (see Figure 8). For each card, a box will appear, with the card name, type, text, power/toughness or loyalty (if applicable), and a small image of the card. This will be done for every card returned from the query, though there is a limit of 175 cards. The idea with this page is that a user can search for cards of a certain type when they are building a deck, and then input them into the deck builder page when they find a card they wish to use.

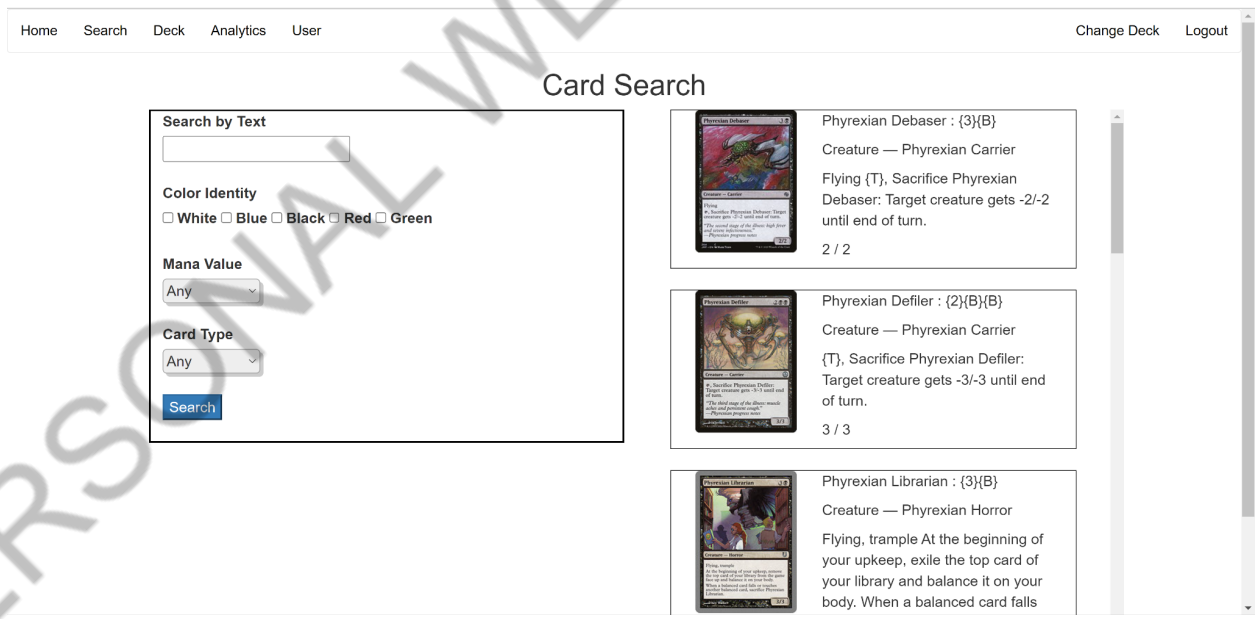


Figure 8. Search Executed

The initial draft (Figure 9) of the search page and the final implementation appear the same. The only real difference between the two is that in the draft, it did not specify how a large amount of cards would be displayed. In the final result, the card info area has a size limit, and if enough cards appear that would overflow, a scroll bar is present to scroll through the list of cards.

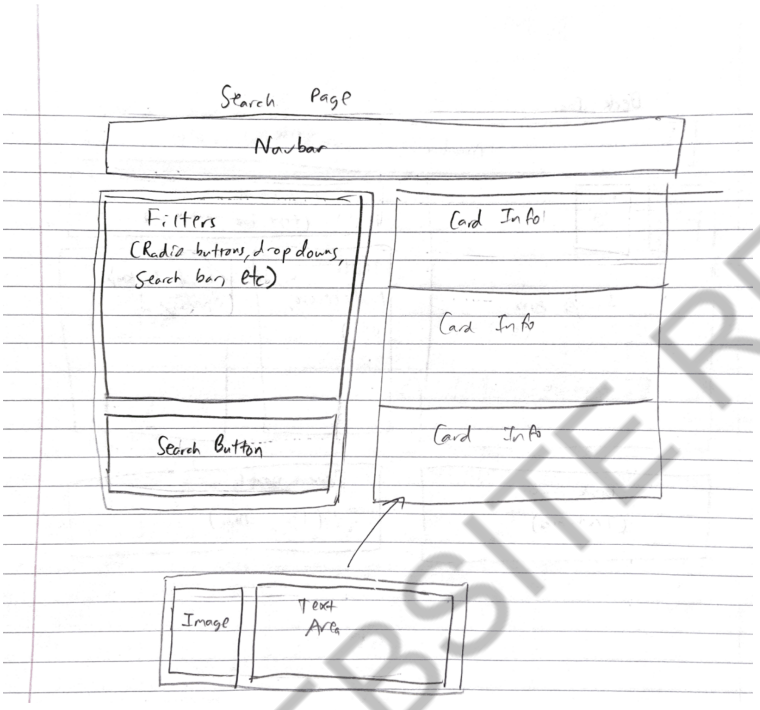


Figure 9. Search Page Draft

While the next page in the navbar is the Deck page, analyzing the login page first is important to the flow of the program. In order to construct a deck, the user must have an account. If the user clicks on the deck page before having an account, they will be redirected to the user page. Here is presented a simple login page (Figure 10), prompting for a username and password, and then a login button. The user enters a username and a password, and if they have an account, they will be logged in, if not, a new account will be created. Account existence is checked by querying the users database and seeing if the username appears. If the username matches, and the password does not, an incorrect password message will be displayed.

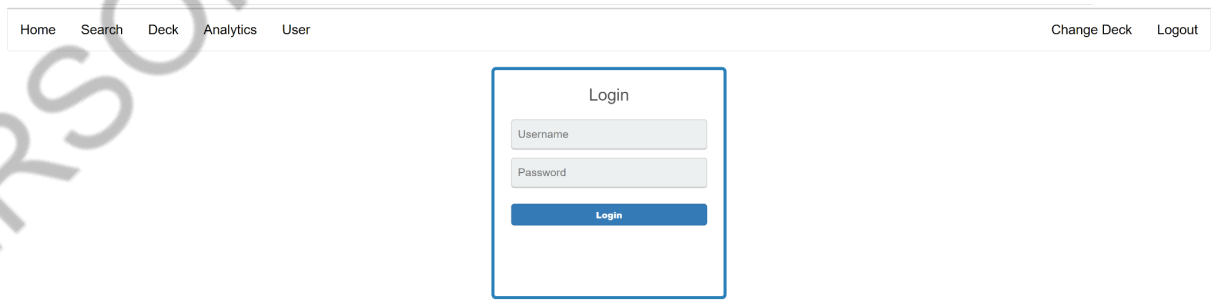


Figure 10. Login Page

In planning for implementation of accounts, an entire user page was planned, which would contain information such as the user's password and username, with the option to change, other settings, and all of the user's decks (see Figure 11 for a diagram). When implementing this into the web site, however, it was deemed unnecessary. It was difficult to think of settings that could actually be useful to save per user, and the saved decks appear at a different place (when loading in a deck). Thus, it was decided that the only aspect needed for users was the simple login page, as well as the logout button previously described.

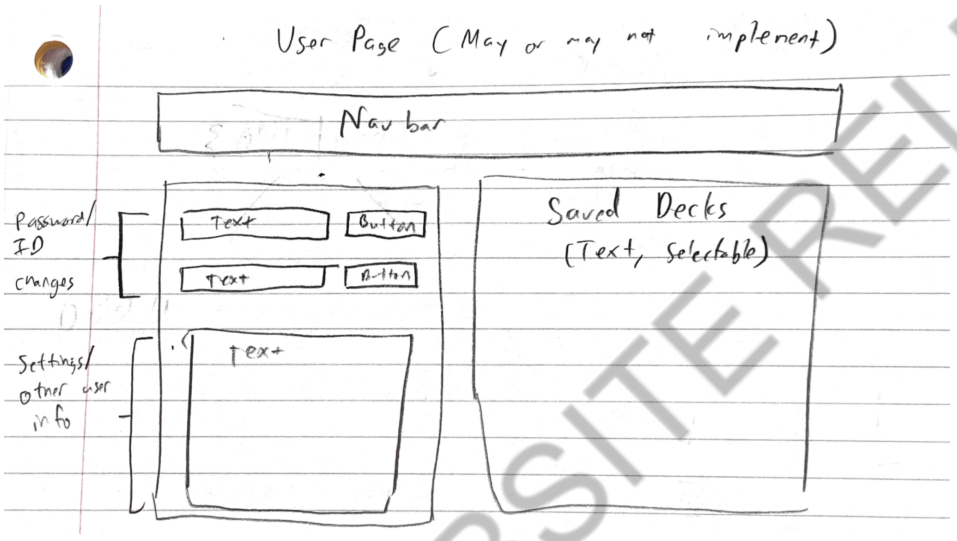


Figure 11. Planned User Page

Once the user logs in and enters the deck page, they will be presented with a page consisting of the navbar and a deck selection box, shown in Figure 12. Here they have two options to proceed, either creating a new deck or selecting an existing deck. If they create a new deck, they enter a name for it in the box and click the "Create Deck" button. If they select existing, a drop down menu appears with all that user's decks (obtained through a query of the decks table). Note that if they try to create a new deck with a name that already exists, it will load the existing deck. This aspect of the deck building page was not initially planned, but was adapted from the user page of the draft's selectable decks.

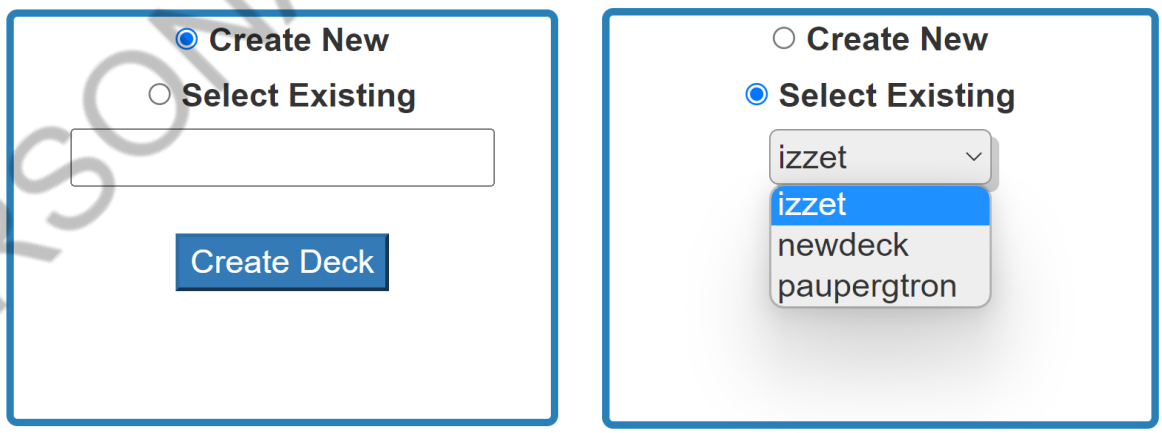


Figure 12. Deck Selection Options

When the user has selected a deck, they will enter the Deck Builder page. On the left, all of the cards in the deck will be displayed, and on the right will appear a section for searching cards (see Figure 13). There is also a button to save the deck - after the user has modified the deck as they please, they must select this to save the deck to the .deck and .qty files to be loaded in the future. As for the add/remove cards box, the user enters the name of a card into the text box (as close to the exact name as they can) and clicks “Find Card”.

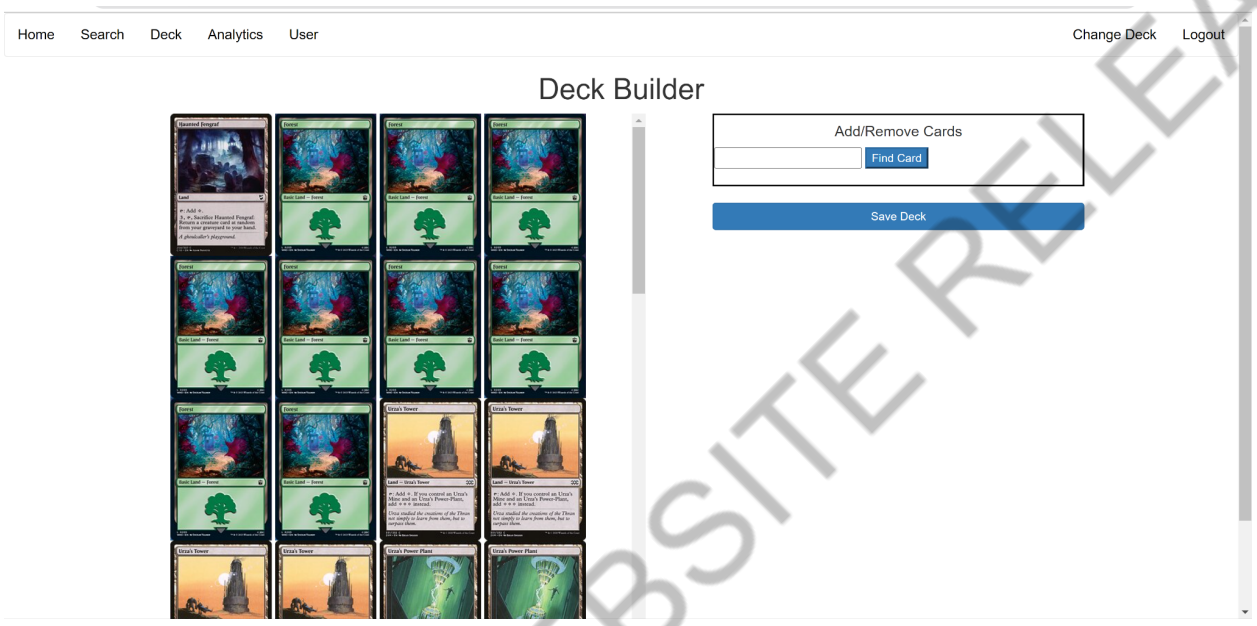


Figure 13. Deck Builder Page

Once the user has searched for a card, the right side box will populate with the searched card. Here there is the basic information about the card, as well as a button to add or remove, and then a quantity (1-4). The user can click these options to add or remove cards. Note that for cards other than basic lands, trying to add more than 4 will not work, and for all cards, you cannot remove cards that are not present.

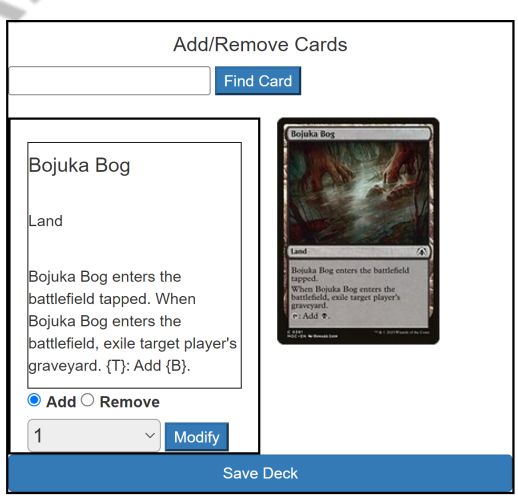


Figure 14. Add/Remove Search

The draft for the deck page (Figure 15) was mostly the same as the final implementation, albeit without the lower two elements. The Deck Info was moved to the Analytics page, and the Text Decklist was deemed unneeded on this page, so it was moved into Analytics as well.

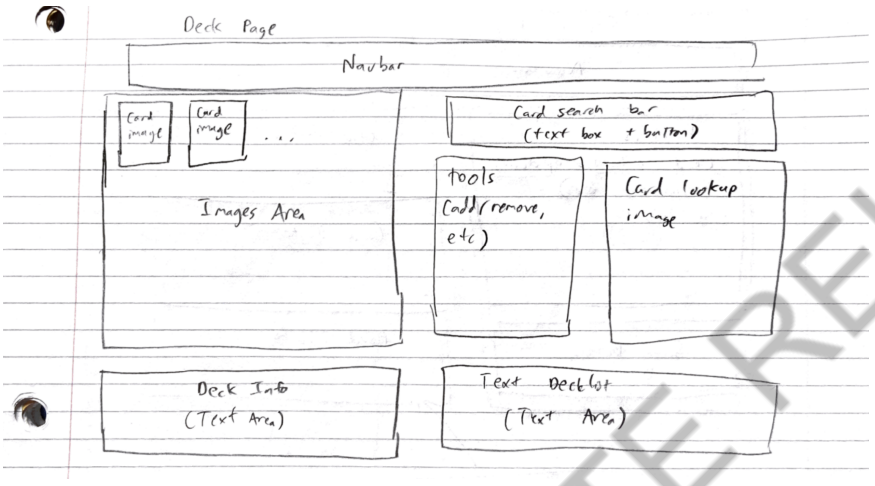
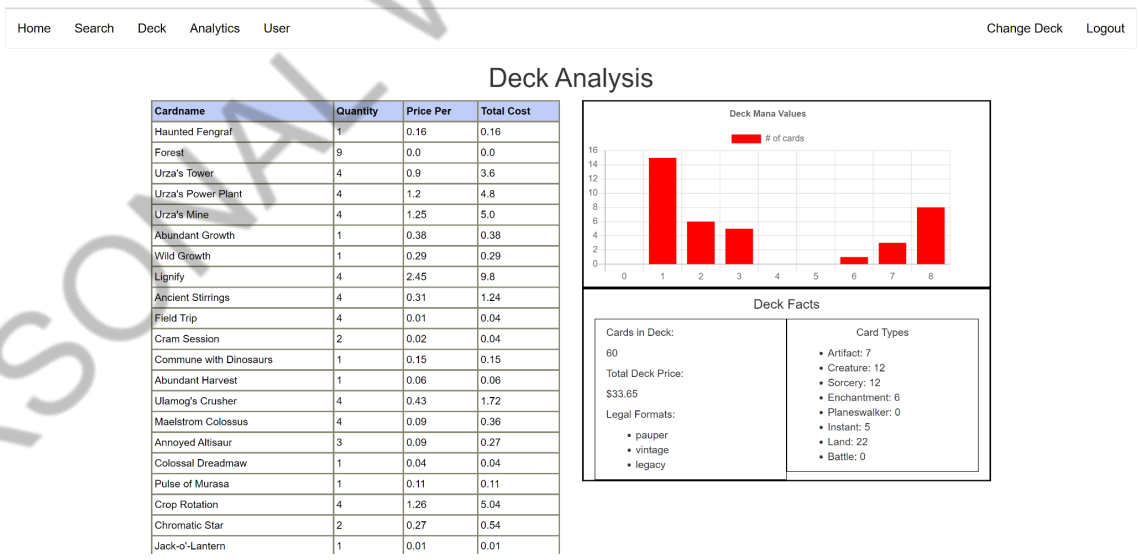


Figure 15. Draft for Deck Builder Page

Once the user has a deck loaded in, they can access the deck analytics page, shown in Figure 16. This page contains information that would be useful for a MtG player to know about their deck. On the left, a table displays the cards in the deck, as well as their quantity and prices for them. On the right, a chart displays the mana curve for the deck. Mana curve is a graph of the number of cards of each converted mana cost (excluding Lands) in the deck. Below the chart is a box of information, including number of cards in the deck, total price, legal formats, and number of each card type in the deck.



Cardname	Quantity	Price Per	Total Cost
Haunted Fengraf	1	0.16	0.16
Forest	9	0.0	0.0
Urza's Tower	4	0.9	3.6
Urza's Power Plant	4	1.2	4.8
Urza's Mine	4	1.25	5.0
Abundant Growth	1	0.38	0.38
Wild Growth	1	0.29	0.29
Lignify	4	2.45	9.8
Ancient Stirrings	4	0.31	1.24
Field Trip	4	0.01	0.04
Cram Session	2	0.02	0.04
Commune with Dinosaurs	1	0.15	0.15
Abundant Harvest	1	0.06	0.06
Ulamog's Crusher	4	0.43	1.72
Maelstrom Colossus	4	0.09	0.36
Annoyed Altisaur	3	0.09	0.27
Colossal Dreadmaw	1	0.04	0.04
Pulse of Murasa	1	0.11	0.11
Crop Rotation	4	1.26	5.04
Chromatic Star	2	0.27	0.54
Jack-o'-Lantern	1	0.01	0.01

Figure 16. Deck Analytics

The draft for the analytics page, shown in Figure 17, is quite similar in overall content, though the areas were moved around for the final implementation. The decklist was combined with the price info, and the deck logistics were moved to the right side. A major difference, however, was that the suggestions tab was removed. While this was not an easy decision, this was done as ultimately, the types of suggestions that could be provided would be able to be gathered by using the infographics on the page. Suggestions that were thought of were taking out outlier expensive cards, changing mana values to alter the mana curve, and adding/removing types of cards. However, due to all this data being present on the page, suggestions were removed. In addition, some decks would be constructed in a manner that naturally causes strange values for these elements - for example, the deck pictured in the analytics page above has a skewed mana curve, but that is by design due to playing a certain type of land cards.

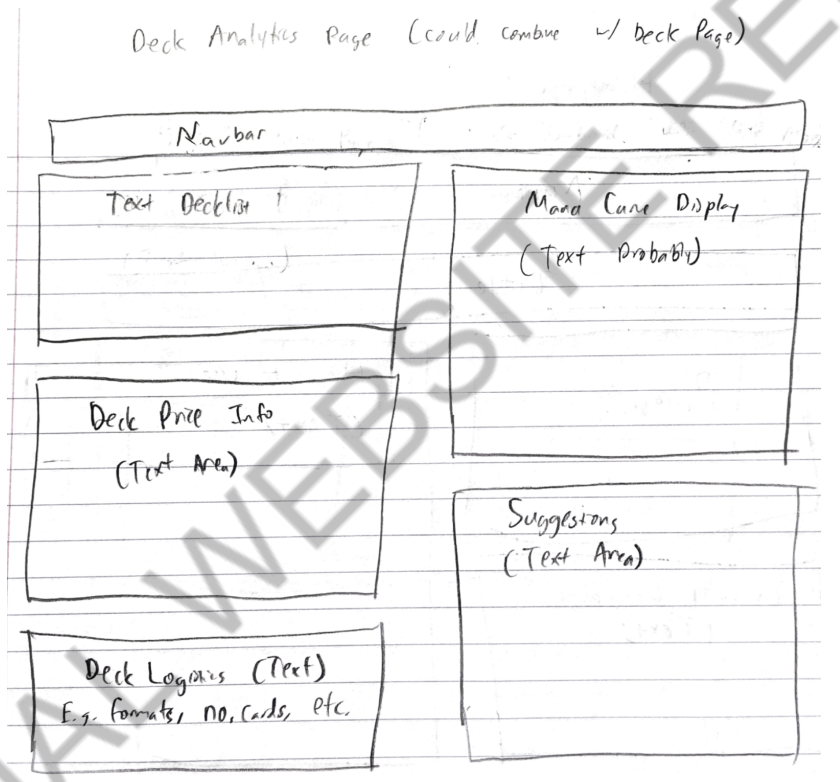


Figure 17. Analytics Page Draft

Paradigms Exemplified

The type of paradigm most exemplified in this web page is Object Oriented Programming (OOP). The primary aspects of the program that are used are instances of classes, usually a Deck and various amounts of Cards, as well as a User. The flask server code always has an instance of a Deck and a User, keeping track of the active instance of each - though they may be set to None or 'default' if they have not been accessed. As the user accesses the various points of the program, the Deck object may be modified, specifically the cardlist attribute. This attribute is composed of many Card objects - showcasing a Composition relationship. Card objects can also be seen separately from a Deck, such as when using the Search page. In the Analytics page, many methods of the Deck object are called, one of which calls a method from each Card object.

The program's use of Deck and User are also reminiscent of the Singleton design structure, as there is always only one instance of each, and two users/decks cannot be used at a time.

Due to the use of SQL queries, as well as HTML, there are also elements of declarative paradigms. The SQL Select commands, used to obtain a user or deck, simply specify what is wished to be obtained. So, in giving the command, the actual low-level operations performed are not known, just what the output should be. The use of HTML, though not technically a programming language, is also considered to be declarative, as it also specifies what is to be output (the web page layout) and not necessarily how the browser interprets it.

Challenges

Towards the beginning of development, it was proposed that decks could be stored in a database. When it came to actual programming, though, this became quite the challenge, as previously described when discussing .deck and .qty files. After some more research into the Scryfall API, the collections POST endpoint was found, which made it feasible to store decks by storing unique ids of cards, and then sending a post request. Once this was figured out, it was determined that a separate file for quantities could be made so that a deck's quantity per card could be known while still having the .deck file as a POST-able JSON coded file. From this, it was just discovering the proper ways to parse the files so that they could be written to and read from to save a Deck.

Another challenge that appeared early into testing was due to Magic's many different types of cards. While it was realized early that the Card object's init could be written to handle all types of cards, problems were discovered when it tried to adapt a dual-faced card (cards that contain two different forms of a card, one on each side). These did not follow the same format in Scryfall's return data as a regular single faced card. Thus, a way to test whether a card was dual faced had to be written, which was ultimately performed via try/except blocks. Similarly, in populating the stats attribute of the card, regular expressions were used to either add power and toughness, loyalty, or nothing, depending on the type of card.

Though not as specific as the previous types of challenges, it was quite difficult navigating through form requests and request methods in order to decide what to display. For example, on just the deck endpoint, it needed to be determined what type of request was made, if the user had a deck already loaded, if the user was searching, if the user was adding/removing, or if the user was saving, all on the same endpoint.