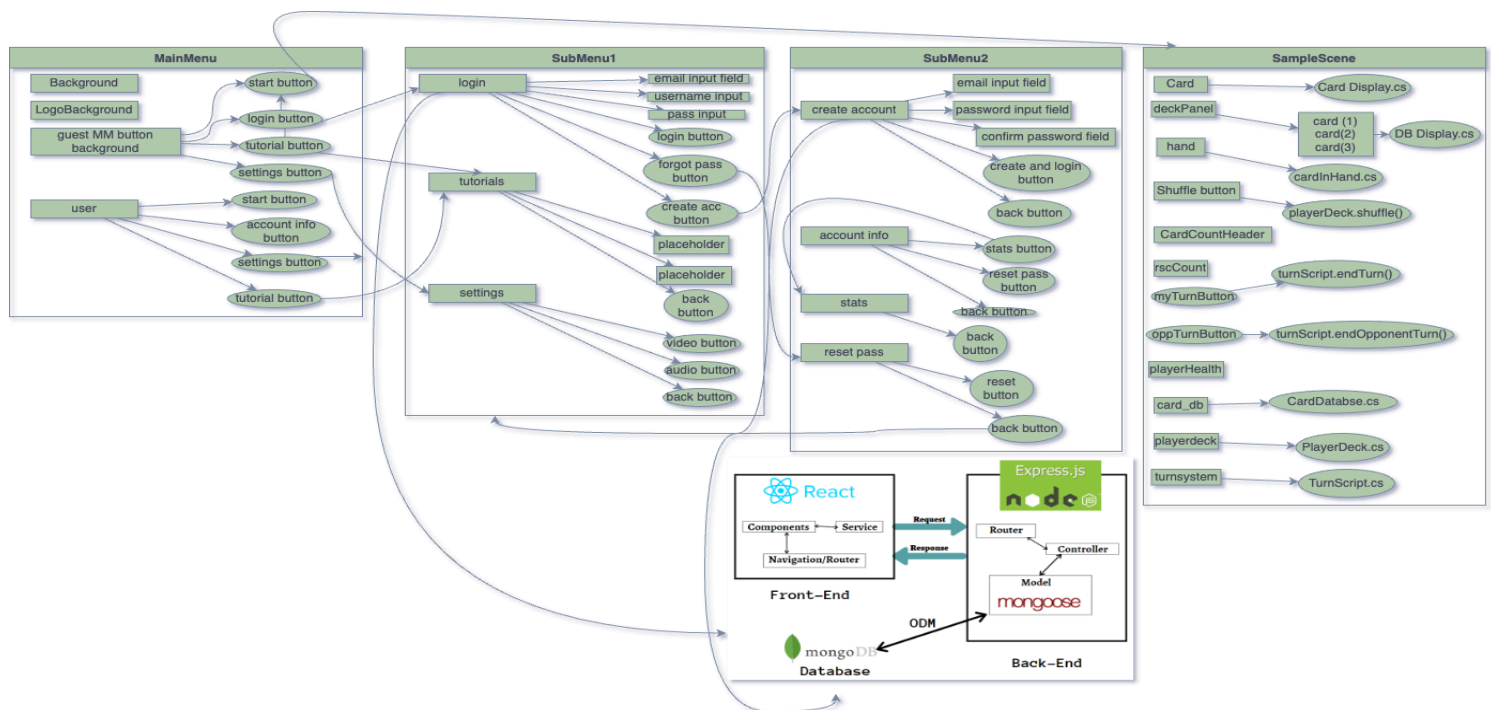


TEAM 19 - So for this milestone, our prototype has gained a lot of functionality, for example, we set up an admin system for our website's information, users can create accounts and login to them in order to track their stats, they can reset their passwords using a secure encryption key system, and we continued to improve the actual card game system.

Here is the link to our Design Video:

https://drive.google.com/file/d/1nP_a0nMSmpJPXWQY1vJqZ7yrdaKDs_zJ/view?usp=sharing

Unity Architecture Design:



For our Unity architecture for the game, we have two primary scenes named MainMenu and SampleScene. Each scene has objects that point to the elements contained in that particular object, which further point the user towards the action they request when they click on the game object. SubMenu1 and SubMenu2 are sub-branches of the MainMenu. The login menu and create account menu game objects retrieve and store data to our useracc table in the Database.

In our website stack, we are integrating MongoDB, React, Express, and Node for our stack. While React specifically uses a High Order Component (HOC), it does not cover the entire stack architecture. The stack is aligned with MVC architecture, where MongoDB serves as the Model Layer, handling data with the NoSQL structure. React serves as View Layer, which employs HOC for the front end. Express and Node acting as the Controller, managing logics, routing, and API endpoints.

We chose this architecture because it offers a well-organized and scalable system structure. The MVC architecture pattern also provides a solid foundation for our website and ensures separations between functions, enhancing code maintainability and scalability as our website evolves.

Database Design:


















Website Database Design:

Cards	Rules
* * _id	* * _id
* * Name	* context
* type	* title
* hp	* * order
* defense	
* attack	
* ability	

Users
* * _id
* * username
* password

Left Star: Required Right Star: Unique

Game Main Menu User Database:

useracc	
uid  	int
useremail  	varchar(30) NN
hash 	varchar(100) NN
salt 	varchar(50) NN
datecreated  	date NN
gamesplayed  	int NN
gameswon  	int NN
damagedealt  	int NN
reset_token_hash  	varchar(64)
reset_token_expires_at  	datetime

The website data is modeled this way to represent unique collections with their specific attribute in the MongoDB NoSQL database. Each collection functions independently, without direct connections between them. The actual game user data is modeled in one SQL table that stores the user's login information and stats and doesn't interact with the website data. Currently, we use two different databases because we wanted to restrict what the website has to load vs what the game has to load. For example, only site admins need to load the website's database since they are the intended "Users" of it, and most of our actual users only want to play the game, so they don't need to load it every time. Not to mention that the game user database will only have to be loaded by players wishing to log in as well.

Prior to the current website database, we considered combining all entities in a single collection, where the schema would be arrays of objects for Cards, Rules, and Users. However, we opted out of this model to avoid potential overhead when accessing the data and to increase the efficiency of the data operations. In the game user database, we considered making a new table for the useracc's stats to be separate from their login information. Still, it seemed like the benefit would be negligible (queries to log in would technically be faster, but the Unity script needs to query the stats table immediately after logging in anyways, so being able just to do one query to login and get all the account's information in one go is more efficient).

The website data's NoSQL approach allows for flexibility in schema design, as each collection can evolve independently without predefined relationships. This structure adheres to MongoDB's document-oriented nature, facilitating scalability and adaptability. We might end up connecting the

website's database to the game user account database in the future if we decide that admins should be able to manage player accounts from the site. Still, they can only control the website's information, which isn't necessary.

Game UI navigation flow design changes:

Game Board:

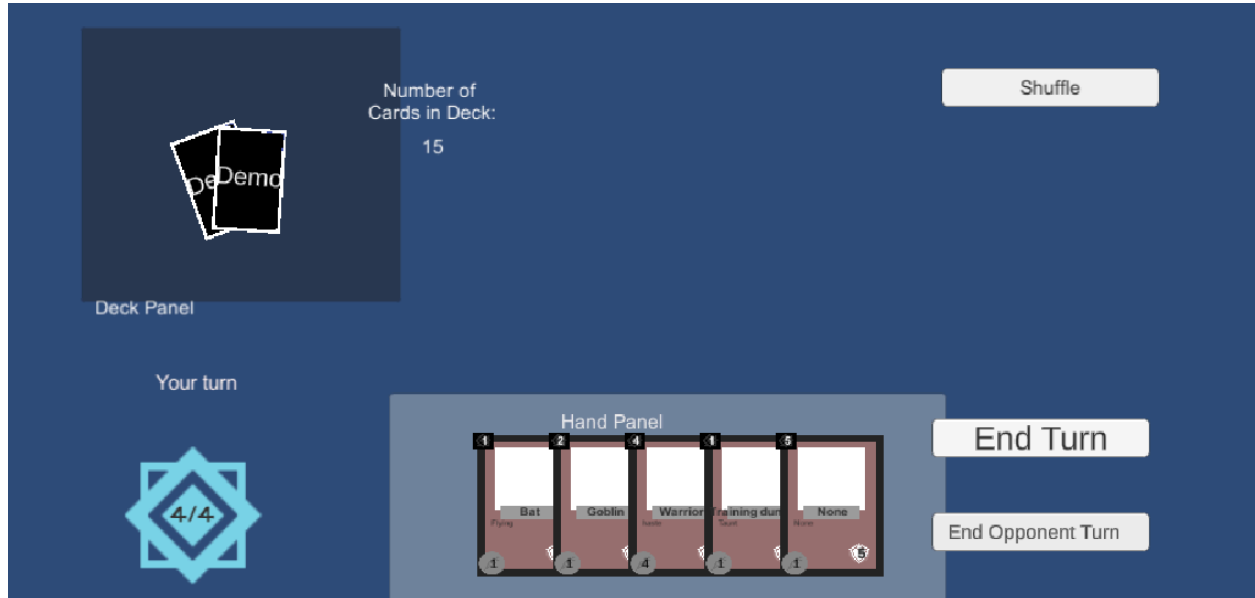


Figure 1.0 Alternative and Unused Design

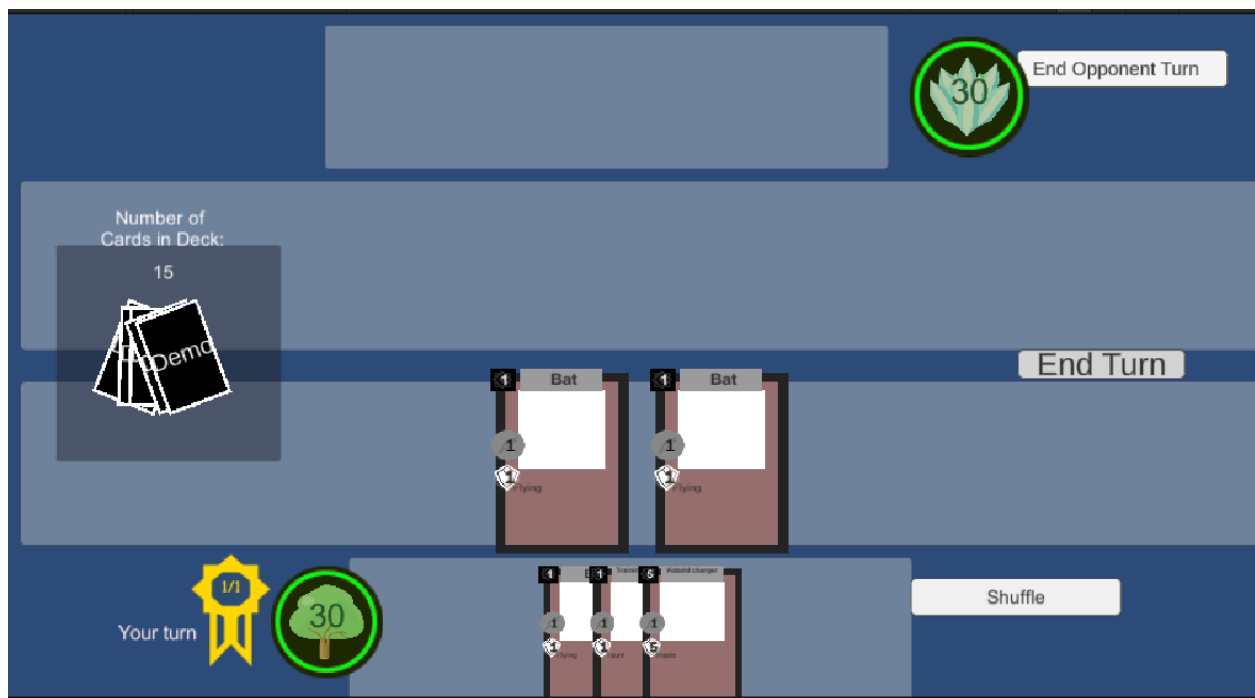
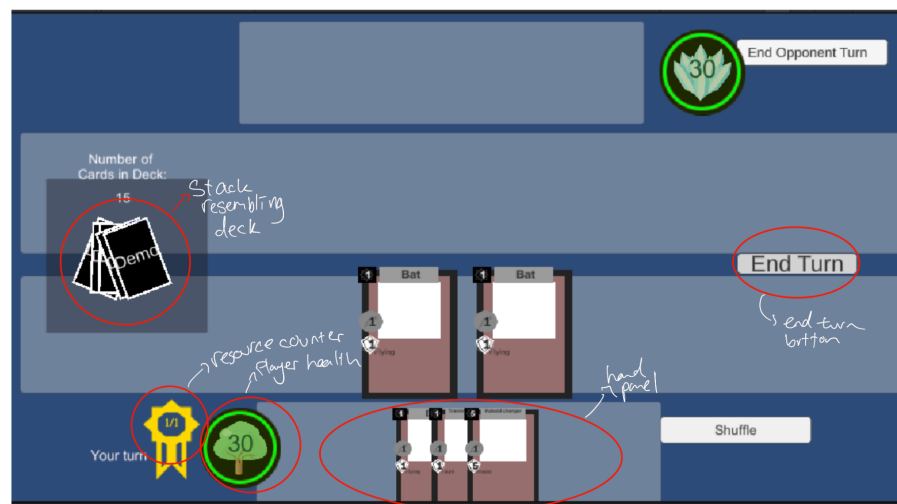


Figure 1.1 Chosen Design

We have two designs for the game board: one alternative and unused, and our preferred choice. We have a turn-based game where players take an action, end their turn, then pass their turn to the opponent. With each turn, players are able to play cards from their hand. During the design phase we were going to design an environment where there would always be a hand panel where cards would be drawn to. The main elements of the game board should always have ***the hand panel, end turn button, deck panel, drop zone, resource, and health.***

Having these five elements available to players the whole time is essential. The main changes were having a clearer ***drop zone*** available for players and moving the ***end turn*** button. This change creates a clearer user experience. It is important to have a clearer place to drop cards as in Figure 1.0 it is unclear where players should place their cards. Moving the end turn button to the middle right instead of leaving it next to the deck panel reduces accidental end turns and makes it easier to access.



Some of the UI elements here are only present for functionality purposes for now and will be removed later on. Specifically, shuffle and end opponent turn will be turned to a proper aspect of the game. The UI elements highlighted in red are the most important ones.

Game Main Menu UI Navigation Flow change:

So in our last demo, we had a Quit button on the main menu for both guest users and logged in users. Initially, we thought it could be good to have some sort of quit game button to restart the game if users wanted or needed to, but once it was running properly in a web browser, we realized that it didn't make much sense. Since players can just refresh the page to restart the window, and because the button actually just stopped the game window dead and froze the game which needed to be refreshed anyways, we got rid of it, and replaced it with a logout button. However, the logout button is only in the logged in user menu, while the button is completely gone from the guest menu. This makes much more sense now, as it declutters the guest main menu, and adds functionality to the logged in user main menu (there was no way to log out before and return to the guest menu once the user was logged in, but now you can).

List of features contributed by every member of the team

Henry:

1. Migrated the existing static HTML web draft to a more dynamic and user-friendly ReactJS framework. The original HTML version was heavily hardcoded, making it challenging to implement changes efficiently. By migrating to ReactJS, We can improve maintainability, ease of modification, and reduce overall development overhead.
2. Implemented APIs using Node and Express, allowing the frontend to make HTTP calls and dynamically update the website content using the appropriate API route. This feature enhances the overall architecture by establishing a clear separation between the frontend and backend, promoting scalability and maintainability. This feature not only facilitates current functionalities but also sets the stage for future expansion and feature development.

Leo:

3. Created a password reset system, which takes the users email or asks for his email if user is not logged in and sends a reset token. The script (resetPass.php) takes the email input from unity, checks the database for that username and saves a unique reset token. The token is emailed to the user and they can enter it in the game to change their password.
4. Added audio and video settings menu which allows the user to turn on/off game music as well as change the volume and brightness. These settings should save to the database so that they stay the same each time the user logs back in.

Brenner:

5. Created the entire user login and create an account system inside the game's main menu scene. I created a database table to hold a user's login information and their stats, then wrote 2 different php files that connect to and query that database when the "ReadInput.cs" script (that I also wrote) connects to them when a user attempts to login to their account OR create a new account. I also wrote roughly 30 automated tests for the readInput script that connect to the php files using all sorts of username and password combinations, and check that the correct error codes are produced by said different combinations.
6. Set up the Account information and Stats menus. I had to update the useracc database table to include the proper stats when I first set this up. This is because I ended up integrating the login and create account systems to send all the user's information to a new script called "DBManager.cs" whenever a user successfully logs in to or creates a new account. The Account Information page and the Stats page inside the account information page accesses all of the new stats I added. Writing automated tests was sort of difficult for this, since the script relies on the other script to populate its values, and most of what happens is set up in the Unity scene, not a lot of code. But I have tests in the login and create tests that confirm what the readInput.cs is storing at this script's values, so I'm confident that it is working as intended.

Adrian:

7. Created the health system script and game object. In terms of the game object it's composed of several layers. The most important layers are the healthText layer and circleBar layer. The healthText layer shows how much health a player has while the circleBar layer is supposed to act as a regular health bar would. The script, 'playerHealth.cs', mainly controls the circleBar's fill amount. The fill amount is dependent on the player's max health and their current health. The script will also ensure that players with negative health will be set to 0 health (the least possible health) and players that overheal would be set to 30 health (the maximum possible health). Testing wise, there are three play mode test scripts that tests the health bars.
8. Created the drag and drop system for the cards. To do this I had to create a drag and drop script ('dragScript.cs') and update several prefabs. The drag and drop system works by enabling event systems, which detects dragging and dropping. Dragging is done by having the card follow the cursor through transforming its position. The challenge for this feature was identifying how to bounce cards back to the hand and creating designated play zones. Play zones or drop zones are designated areas where certain objects can be dropped, to create this I had to learn about Unity's Box Collider and Rigidbody. Cards could only be played on spots where it collided with a designated play zone. If cards are dropped into non-play zones or cards do not collide with the play zone, it gets sent back to the hand. This is done through manipulating collider shapes and the card's parent. Through manual testing we are able to confirm that everything runs fine. Automated testing is still a work in progress as more exploration is required as I need to learn more about input systems in Unity's testing framework.

Aditya:

9. Created the automated testscript file named numberTest.cs which contains five test methods testing the game's features which we have implemented so far. The CardDatabase_PopulationTest checks if the deck of the cards is being populated with cards or not. The DeckSize test checks if the deck has twenty cards. The Initial_Draw_test checks if the player's hand has five cards to play the game. The ShuffleDeck_ShouldChangeOrder test checks if the deck of cards is being shuffled when it is pressed. The turn_test checks if the turn system of the game functions as expected with the correct values.
10. Created the turnscript.cs game script which adds functionality of the turn system in the game and keeps track of the mana the player has. The script has several variables initialized and has start, update, end turn, end opponent turn methods which update those variables accordingly to show the users which player's turn it is in the game.