Team: Discord Dragons

Members: Dana, Kevin, Max, Swamik

Project: 2A - Preliminary Architecture

Product Proposal: [LINK]

Primary Author: Dana, Kevin, Max, Swamik

Slip Days: 0

Preliminary Architecture

Function Delivery

Discord is a multi-use free messaging and VoIP application that we are using as a platform for a 'bot' that helps in facilitating table-top roleplaying games (TTRPGs) such as Dungeons & Dragons (D&D). For release 1.0, we have three main functionalities we want to implement. First and foremost is dice rolling, which includes the capability to roll custom-sided dice, fudge or predetermine rolls if you have the game master role, and manually input dice rolls to cater to players that would prefer to roll physical dice for their games. Our second focus is an enhanced combat turn tracker, which lists the initiative order and prompts our users to end their turn and update the turn tracker. Finally, we would also like to implement a way to look up rules and character features for D&D. To access these features, users will need to first create a Discord server. A Discord server is a hub with text channels and voice channels allocated to specific tasks. Once a server is created, the administrator can then invite our 'bot', allowing the members of the server to be able to access its functionality.

To engage with a Discord 'bot', the user needs to begin a message in a text channel with a certain prefix, followed by a valid 'bot' command. Most commonly, 'bots' use the following prefix-command structure: ![command]. Because the exclamation point prefix is familiar to our users, we have decided that for the sake of usability, our bot will use the same structure and allow our users to change the prefix by using a 'setprefix' command. This allows our users to use a more familiar or practical prefix within their server if they prefer. Depending on the command they use, our 'bot' will run different functions. For example, the command "!r 1d20" would indicate to our 'bot' that the user would like to use the dice rolling functionality using a 20-sided dice. Our 'bot' would then return the output of this function in the form of a Discord embed to

the text channel, allowing all the members of the text channel to see the outcome. This process will be the same for all our functions.

Interfaces

The only external interface we have is Discord. In order to use our 'bot', our users need to have a means of accessing Discord, either from a computer, tablet, or smartphone. They must then have a Discord account, and be part of a Discord server. The administrator must have incited our 'bot' to the server for it to be accessible. On our side, we will potentially need a web scraper to interact between our 'bot' and the online D&D Wikipedia page. This is because, while we will be able to use the D&D 5E SRD API to look up the most commonly used features, some things are still behind a paywall. In order to combat this, we had to look for an alternative means of getting that information in a presentable way for our users. After much deliberation, we settled on using the D&D Wikipedia page as a resource. Since we have never written a web scraper before, this will be a challenge and could pose some major problems down the road.

Runtime Components

Our Discord bot should be inviteable to any Discord servers and should be able to respond in real time to commands. Its responses will almost always take the form of Discord embeds. For commands, it should be able to parse anything after the recorded command symbol and retain the username as well as the text of the command. It should respond in the channel it was called for most purposes, but may have need of direct messages to give specific users information on their rolls.

The roller should be able to accept a variety of dice styles (eg. 4-sided dice). It should return an embed showing the result of each die, as well as a sum of the rolls. It should accept two commands: r and f. The former rolls the dice as specified, the latter would accept extra parameters to tell it exactly which rolls to return. In later versions, we also want to include functionality for further commands being embedded in the text parameters. For example, it should be able to handle "!r 2d20 h1" which would mean that it rolls two 20-sided dice and keeps the highest value.

During our elicitation panel, a compendium lookup was a very popular choice. Our lookup will be based on the 5e SRD API, which includes all the basic rules available for free.

The bot should be able to return an embed with the information presented therein. If the specific search is not available in the SRD, we can instead use a web scraping search to return an embed with the wiki page for D&D 5e, since the wiki includes most of the rules.This web search functionality may be pushed to version 2.0, since we think the API should provide most of what we need.

Emoji buttons came about because our panel was pretty consistent in saying that they did not want too many commands to work through the initiative feature, which crowd out messages in the chat box. Since Discord does not have functionality for embedded buttons, we thought we could get our bot to respond to reacting with an emoji to the initiative list. We think this is a solid workaround, although it is a "nice to have" feature and we could potentially accomplish the same thing with a command or two. We have proof that this is feasible, as it is a common feature in many major discord bots.

In version 2.0, we hope to include a database of homebrew materials, so that users can add custom content. This would come with its own search and creation tools, but it would be done directly through the bot, by a loop of questions like: "What is the name", "What class is this NPC", etc.
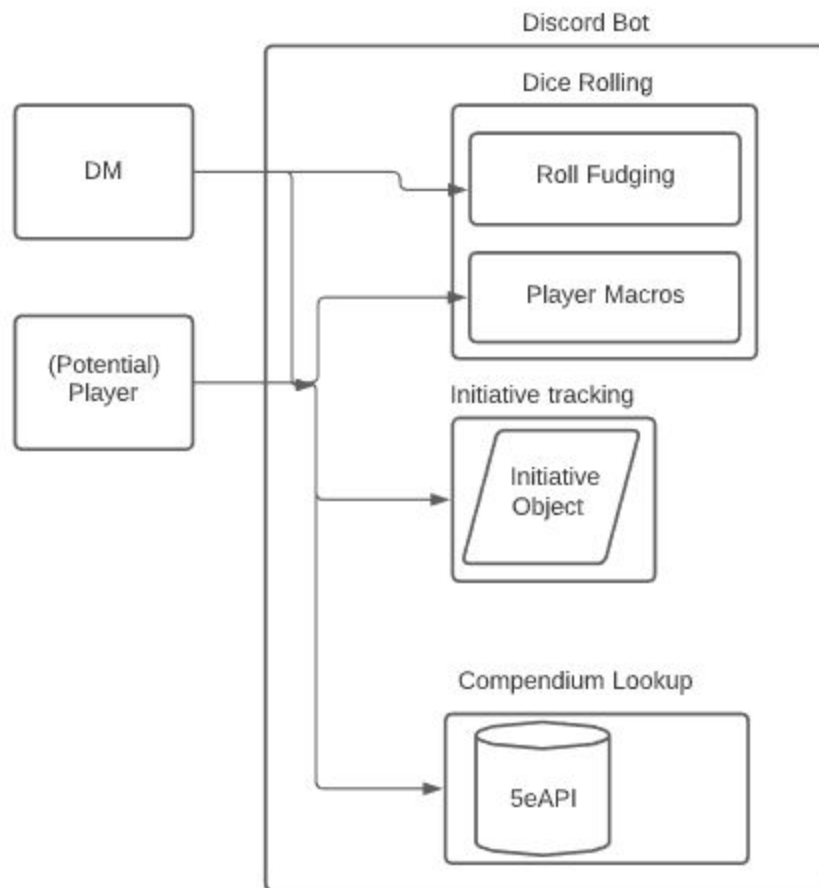
Major Persistent Data Objects

In the scope of this project, we have 3 major modules containing persistent data objects. Those are the dice rolling module, the initiative tracking module, and the compendium lookup module. We will discuss the object requirements for these three modules in order. For the dice rolling module, we will either represent a 'dice roll' as a Python object with attributes pertaining to the roll, including quantity, size, and modifier. Alternatively, this feature set can be implemented without class structure through a simple function. Any saved player macros will be stored simply in a persistent .csv file, such that players can access this data during play time through a simple command. The initiative tracking will be represented as an initiative table object in Python. This will similarly be stored in a .csv file for the same ease of access and low overhead described earlier. We can then parse the information and represent it as an embed in Discord. Finally, compendium lookup will likely be our challenging module. For this we are dependent on the DnD 5e API, meaning we rely on a third party host to maintain and update the

compendium. On our end, lookup is extremely simple, as the API is well documented and consistent in data access methodology. Should this prove insufficient, we will look into using web scraping tools such as Beautifulsoup, a Python library designed for easy web scraping.

Prototypes

One of the main concerns that we had about this project is the feasibility of making this Discord bot. Because other bots have some of the features that we want to achieve, we knew that the functionality that we wanted to build was possible; however, we did not know anything about the difficulty of making the actual bot. To get a better grasp of our project, we decided to prototype a bot with simple functionality. For the prototype, we wanted to test two essential parts to our bot: response to user and dice rolling. Looking at an online tutorial, we were able to make a discord bot that responded to specific user inputs. For example, when the user gives the command "!hello," the discord bot tags the user and then replies with our specific message. This was an important task because we wanted to include initiative tracking in our final version. Tagging people and replying to their inputs is essential. After constructing this function, we had an idea of how user inputs to our bot would work.

The "!hello" command showed us how we can respond to users, but we wanted to also test how we can call functions while responding to the users. For the bot, we just made a simple die roller that responds to the command "!roll." For the prototype, the bot will just return a roll from a 6-sided die. After the construction of our bot, we were able to invite it to our server and interact with it using commands.

```python
# when a user joins the server, the bot will greet them
@client.event
async def on_member_join(member):
    await member.create_dm()
    await member.dm_channel.send(
        f'Hi {member.name}, we were waiting for you!'
    )


@client.event
async def on_message(message):
    # we do not want the bot to reply to itself
    if message.author == client.user:
        return

    # checkes to see if a messege starts with !hello and acts
    if message.content.startswith('!hello'):
        msg = 'Hello {0.author.mention}'.format(message)
        await message.channel.send(msg)

    if message.content.startswith('!roll'):
        msg = diceRoll(6)
        await message.channel.send(msg)
```

Because the rolling mechanism that we want to implement is not as simple as rolling a 6-sided die, we decided to prototype a more difficult die roller. For this prototype, we implemented a simple feature set including multi rolling (simulation of rolling multiple dice at the same time) as well as some major mechanics such as 'rolling with advantage' (Rolling two dice and taking the better or worse of the two depending on status). We will later implement persistent player macro storage and game master dice fudging. From the prototyping, we have a better understanding of the time needed to code our different modules and how difficult it would be to implement certain functions.

```python
def roll(die):
    return random.randint(1, die)
def multiroll(q, die, mod = 0):
    totalroll = []
    sum = 0
    for i in range(q):
        d = roll(die)
        totalroll.append(d)
        sum+=d
    print(totalroll)
    return sum + mod
```

Key Issues

For this project, we will utilize a top-down design approach. We are confident in the overarching structure of our architecture, as we have all used similar tools and recognize the necessary subcomponents to make the architecture cohesive. For this reason, we will design the abstract architecture and then move down in scale to specific implementations and module design. Additionally, none of us have significant experience working with and designing bots for Discord, so we wanted to understand our general development direction before moving on to specific designs. We then conducted research and concluded that this project was feasible before we concerned ourselves with low-level structuring.

Our major area of unfamiliarity is interface with the Discord web server. Our entire team has significant experience developing projects in Python, but very little knowledge pertaining to web interface and external data structures. We have done significant research since the project started, and have concluded that there is enough documentation and proofs of concept for us to successfully implement these features, but we anticipate that these two aspects will still be the major sink for development resources. As such, our final architecture consists of the Discord interface working with our Python code for dice rolling and initiative tracking, which can pull information from the 5e API.

Outstanding Issues

In terms of complexity, web scraping will be one of our biggest issues, for two reasons. For one, none of us have ever made any sort of web scraper, so we'll have to teach ourselves as we go. A bigger problem is which sources to rely on. There are wikis, subreddits, and fan pages hosting the information, but we have to be sure the information our bot returns is reliable. We know of one wiki, D&DWiki, that is generally very reliable. However, an over-reliance on this one webpage might lead to serious issues if we optimize our bot for this one page and it gets deleted, reformatted, or simply loses popularity and stops being updated.

Another issue is using emojis as buttons. We have seen it implemented in other bots, and it is not always the most reliable feature. Preliminary research points to the cause being infrequent checking of reactions. We're going to have to define functionality to check for responses via emoji, but not do it so often that it clogs up the processes, or so little that the bot

does not respond in a timely manner. The best solution we can think of is to define a list of active posts that still need to be checked. Posts outside of that list will probably not respond to emoji reactions.