Jonathan Wiseman, Ian Bowler, Matthew Grohotolski
CS 341
May 8, 2020

Looking for Jays (LFJ)

https://github.com/jonwiseman/LFJ

1. MOTIVATION

1.1. Background

1.1.1. Discord

Discord is a popular messaging and communication application catering primarily to the gaming community. Its use is widespread, with about 963,000,000 messages exchanged daily between users [1]. Discord operates primarily on a server-channel model: servers bring together large groups of people, with channels serving specific interests, topics, or purposes (such as voice or video chat).

In addition to human users, Discord also allows programmable "bots" to operate in servers, serving a range of functions. A bot's function can be as simple as posting images or replying to users when summoned, or as complex as performing managerial and organizational roles for the whole server.

1.1.2. Elizabethtown's E-Sports Team

This wide applicability immediately suggests an application for Elizabethtown's E-Sports team, which primarily uses Discord for communication, event scheduling, and scrimmage signups. The team's primary focus is on League of Legends (LoL), an online, team-based multiplayer game with a robust ranking system. In addition to LoL, the team has a desire to expand to additional games, including Rocket League (RL) and Counter Strike: Global Offensive (CSGO). Each of these games operates on separate clients: LoL has its own client, owned and operated by Riot Games [7]; RL and CSGO are both hosted on the Steam games client [8].

It is out of this context that we propose *Looking for Jays (LFJ)*, a Discord bot primarily focused on helping Elizabethtown's E-Sports team plan and organize their scrimmage events.

1.2. Goals

There are five primary goals for LFJ, motivated by both the team's personal needs and additional features that we believe would be useful:

1. Handling event signup
2. Balancing scrimmage teams
3. Posting reminders for registered events
4. Allowing members to self-assign to groups
5. Relevant statistics collection and querying

1.2.1. Handling Event Signup

The team needs to be able to create and manage upcoming events such as meetings, practices, and scrimmages. This ensures clear communication between members, and provides transparency to the whole group. LFJ will allow users to register for events by a simple voting mechanic, implemented by reacting to a message posted by the bot to the whole server; in this case, the reaction is a form of explicit feedback indicating attendance.

The primary measure of success for this goal is the ability of the bot to record responses and properly remind users of their commitments. This itself is contained in goal number 3, wherein the bot is required to remind users of upcoming events for which they have registered. If all users are consistently reminded of their events, then this goal will be considered met.

1.2.2. Balancing Scrimmage Teams

After a game event has been scheduled, it will be necessary to assign users to teams. For each of the three games with proposed coverage, this includes splitting the users into two evenly balanced teams. LFJ assigns users into teams with a simple first-come, first-serve balancing mechanic with the option for random sorting.

There is no easy way to assess how evenly balanced the assigned teams are. With sufficient data collection, it may be possible to assign a degree of fairness based on the outcome: a longer match with a more even score implies more evenly balanced teams. However, with no real-world application this was a task beyond the first implementation of LFJ.

1.2.3. Posting Reminders for Registered Events

After signups and team assignments are complete, users should be reminded of their commitments. This can be implemented as a simple message to each user that has registered for an event, with messages being sent out at customized intervals. LFJ currently operates by sending a reminder 24 hours before the event is scheduled to begin.

This goal's measures of success are consistency and accuracy. LFJ works as intended if every user that has registered for an event is sent a reminder exactly when they are supposed to be sent.

1.2.4. Allowing Members to Self-Assign to Groups

Each member of the team has specific games that they are interested in playing, and others that they do not want to play. Event sign ups should be limited to players to whom the events are relevant; this helps with team balancing and also reduces the amount of information that will need to be stored about each team member. LFJ will allow users to self-assign to groups that determine which events they receive notifications about and can participate in.

As a measure of success for this goal, team members should only be able to sign up for events related to the games to which they belong; a user not authorized to sign up for an event should get an error message.

1.2.5. Relevant Statistics Collection and Querying

LFJ will keep track of important statistics for games, players, and groups for access control and to improve team balancing. LFJ contains commands that allow users to submit .csv files containing performance statistics. The primary motivation of this feature is to move beyond team balancing by explicitly labeled skill level, possibly by creating a statistical or machine learning model to predict if a team is balanced or not. While this step is beyond the implementation of LFJ at the current moment, the scaffolding is in place for this to be implemented.

Devising a measure of success for this goal is difficult, due primarily to the short-term nature of this project. However, the goal is considered met if users are able to upload their statistics and have them stored for future use. This will give the team and any others who want to work on or expand the bot the structure they need to do so.

1.3. Existing Solutions

There are two primary existing solutions for the problems facing the Elizabethtown College E-Sports team:

1. The current solution: manual curation within the Discord app
2. Existing Discord bots

1.3.1. Manual Curation within Discord

This is the current approach utilized by the E-Sports team. Currently, a session is planned with a simple message that contacts everyone on the Discord server. As explicit feedback, users react to the message with emojis. The team members are expected to hold to their commitment and show up to the event. This works well with informal events, as it is simple and easy to understand. However, it falls short of many of the desired goals stated by the E-Sports team: there is no event registration, no reminders about upcoming events, and no selectivity in who receives notifications. Perhaps most importantly, there is no team balancing for scrimmages; instead, this is handled on a case-by-case basis within the team.

This approach is clearly functional, but it does not take advantage of the automation and record-keeping potential of Discord's bots. Here, LFJ would represent an improvement by reducing human error and automating many functions such as registration, reminders, and record-keeping.

1.3.2. Existing Discord Bots

There are a number of existing Discord bots that cover a few of the stated requirements for this project. Below is a discussion of each of the goals above, and potential Discord bots that could satisfy them:

1. Handling event signup: Pollmaster [4], VoteBot [3], Simple Poll [10]
2. Balancing scrimmage teams: VoxLight [9]
3. Reminders for Events: DisCal [2]
4. Self-Assignment to Groups: none

5. Statistics collection and querying: none

Handling event signup is perhaps the goal that has the most outside the coverage provided by existing Discord bots. However, these bots fail to integrate with other steps in the pipeline that LFJ covers. For example, the calendar application DisCal requires integration with Google Calendar, requiring a manual input step between event registration and notification sending [2]. Finally, there are no bots that currently handle user self-assignment or statistics collection. The implementation of these backend services is unique to LFJ, and represents a significant improvement on existing Discord bots.

2. APPROACH

2.1. Architecture

LFJ follows the Model-View-Controller architecture pattern. In this case, the model is the bot's back end: the databases and scripts that control the bot's behavior and contain the data that allow it to function. The view is Discord, the platform that displays the bot's results (such as messages). Finally, the controller is the Discord API that allows users to send and receive messages from the bot.

Looking for Jays is implemented as a Discord bot, coded in Python via the discord.py wrapper. Discord.py is a Python API wrapper for Discord's developer API; the documentation for discord.py is available in [6], and the documentation for Discord's official API is available in [5]. LFJ's backend is implemented as a relational database in MySQL. LFJ is conceptualized in three stages: user, bot front end, and bot back end. Figure 1 displays the interaction flow between users, the bot, and the backend:

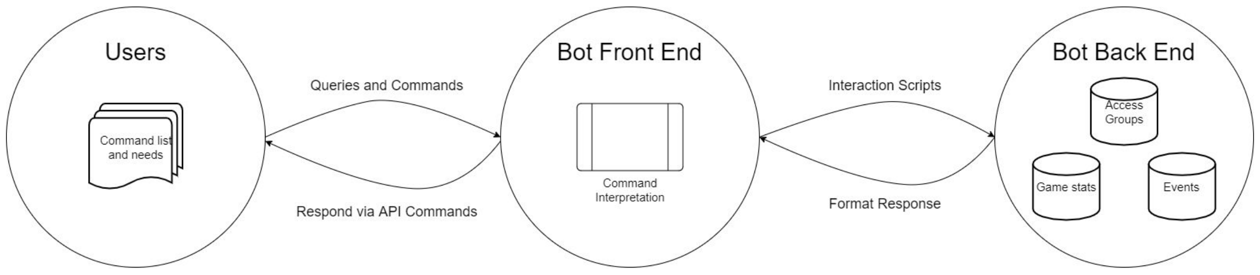


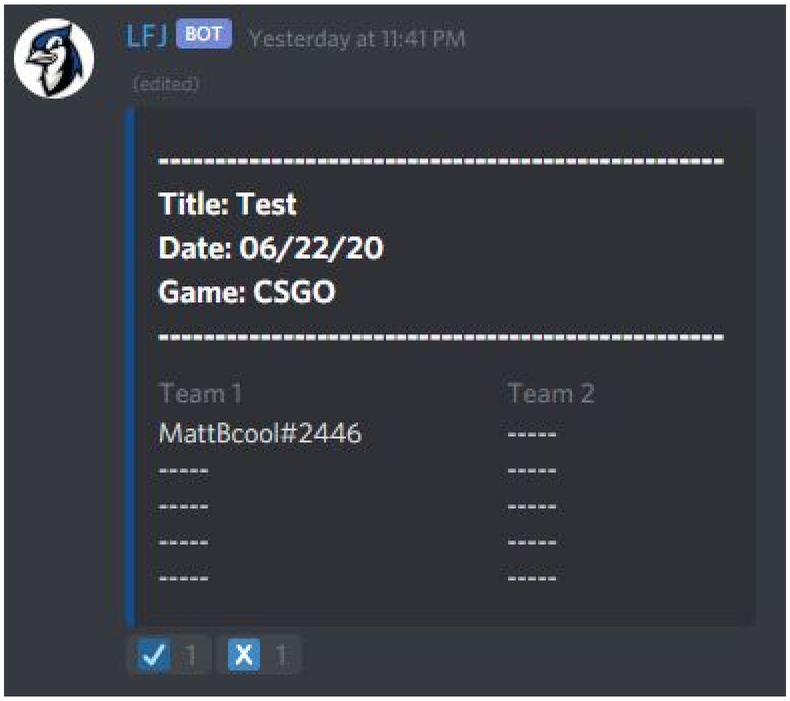
Figure 1: LFJ Overview

2.1.1. *Users*

Users constitute the members of the Discord server (the members of the E-Sports team, in this case). Users have clearly defined needs, listed in the Goals subsection above. Users are allowed to interact with the bot's front end in two ways: via simple text commands and via comment reactions.

Text commands are structured in a way that can be parsed by the bot. As of now, all commands must start with $COMMAND. However, the user is able to specify which prefix they want commands to start with by changing the configuration file. The bot will only be able to respond to pre-defined text commands; for a list of all commands recognized by the bot, consult Section 5.4.

Reactions are explicit feedback to a comment posted by the bot. The only comments that are ever posted by the bot are event creation messages. Figure 2 shows an example of an event creation message:


LFJ BOT Yesterday at 11:41 PM
(edited)

Title: Test
Date: 06/22/20
Game: CSGO

Team 1	Team 2
MattBcool#2446	-----
-----	-----
-----	-----
-----	-----
-----	-----

✓ 1 ☒ 1

Figure 2: Event Creation Message

Many of the voting bots discussed in the Existing Discord Bots subsection above are operated in this fashion. We followed a similar implementation, with reactions corresponding to specific voting choices. Voting is binary: either the user registers (the '☑' in Figure 2) or deletes their registration (the '☒' in Figure 2).

2.1.2. Bot Front End

The bot's front end receives interaction from users and formats them for use in the bot's scripts. The bot's front end receives and transmits messages via the official Discord API, as monitored by the bot's back end scripts. Once the required information has been accessed and formatted by the bot's back end, it will be sent to users via the Discord API. Commands are processed and arguments are passed using the discord.py commands extension.

The commands extension allows the bot's functionality to be split into several modules, called cogs. Each cog contains the commands and helper functions associated with the different classes of interaction. For a discussion of these cogs and related interaction classes consult Section 5.5 below.

2.1.3. Bot Back End

The bot's back end consists of the bot's running scripts and database. The back end is interacted with primarily through MySQL queries, managed by the MySQL Python connector. Finally, the requested information will be formatted and sent via the bot's front end to relevant users. For a discussion of the MySQL implementation of the database, consult Section 2.5 below.

2.2. Hardware Requirements

In order for the bot to run continuously and receive user commands, it will need to be hosted on an active server. This server will require constant internet access and power in order to provide reliable service. The MySQL backend also needs a machine for hosting.

LFJ was initially hosted on an external server with only one running instance of the bot. However, as the project progressed it became increasingly clear that multiple instances of the bot can be run in parallel, provided they have unique names and identifiers in Discord. This allows the bot to be used by multiple groups at the same time, with each group able to change and customize their LFJ implementation.

2.3. Software Requirements

There are four software requirements for building and running LFJ:

1. An installation of Python 3.7
2. An installation of MySQL to run the backend
3. A Discord account
4. The LFJ bot token

The MySQL installation is required to run the bot's backend, while the LFJ bot token is required to run the bot. Please note that the primary LFJ bot token is private; for users to download, build, and run the bot they should provide their own token.

In addition to the software requirements listed above, the running directory of an LFJ installation should have a configuration file called configuration.conf. The configuration file contains important database and Discord login information that allows the bot to function properly; its structure is as follows:

```
[Discord]
token =
events_channel =
prefix =
[Database]
username =
password =
host =
database =
```

In order to run the tests that validate LFJ's functionality, the user must also provide the following fields:

```
[Testing]
display_name =
id =
email =
admin =
```

2.4. Software Design

Figure 3 gives an overview of the system's use cases:

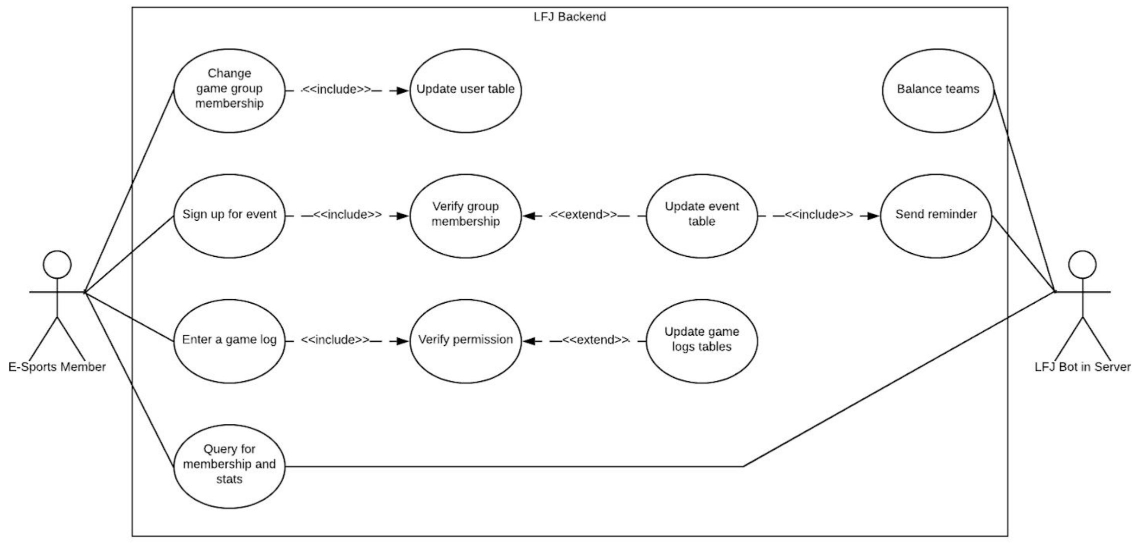


Figure 3: LFJ Use Case Diagram

There are two actors in the system: the E-Sports team member is the primary actor, initiating contact with the LFJ bot via text commands within the Discord server; the LFJ Bot is the secondary actor, responding to users in the Discord server. An E-Sports team member is able to:

1. Create, delete, and modify new users
2. Create, delete, query, and update information about games
3. Create, delete, and query information about events
4. Register and deregister memberships for games
5. Upload a game log
6. Register and deregister for events

When a team member updates his game membership status (i.e. enters that he plays CSGO), then the user table in the database will be updated. When a user attempts to sign up for an event, the system first checks to make sure that he is a member of the relevant group. If he does, then the event table will be updated; the LFJ bot can then send reminders to the users in the server. When a user attempts to enter a game log, the bot will check to see if he has permission; if he does, then the game logs in the database will be updated. Finally, the user can query the bot for membership and stats, which will be displayed by the bot in the server.

The LFJ bot in the server can interact with users via messages. These messages can display the teams that have been created, remind users of their registered events, or be query responses.

2.5. Database Design

Figure 4 displays the bot's backend database design:

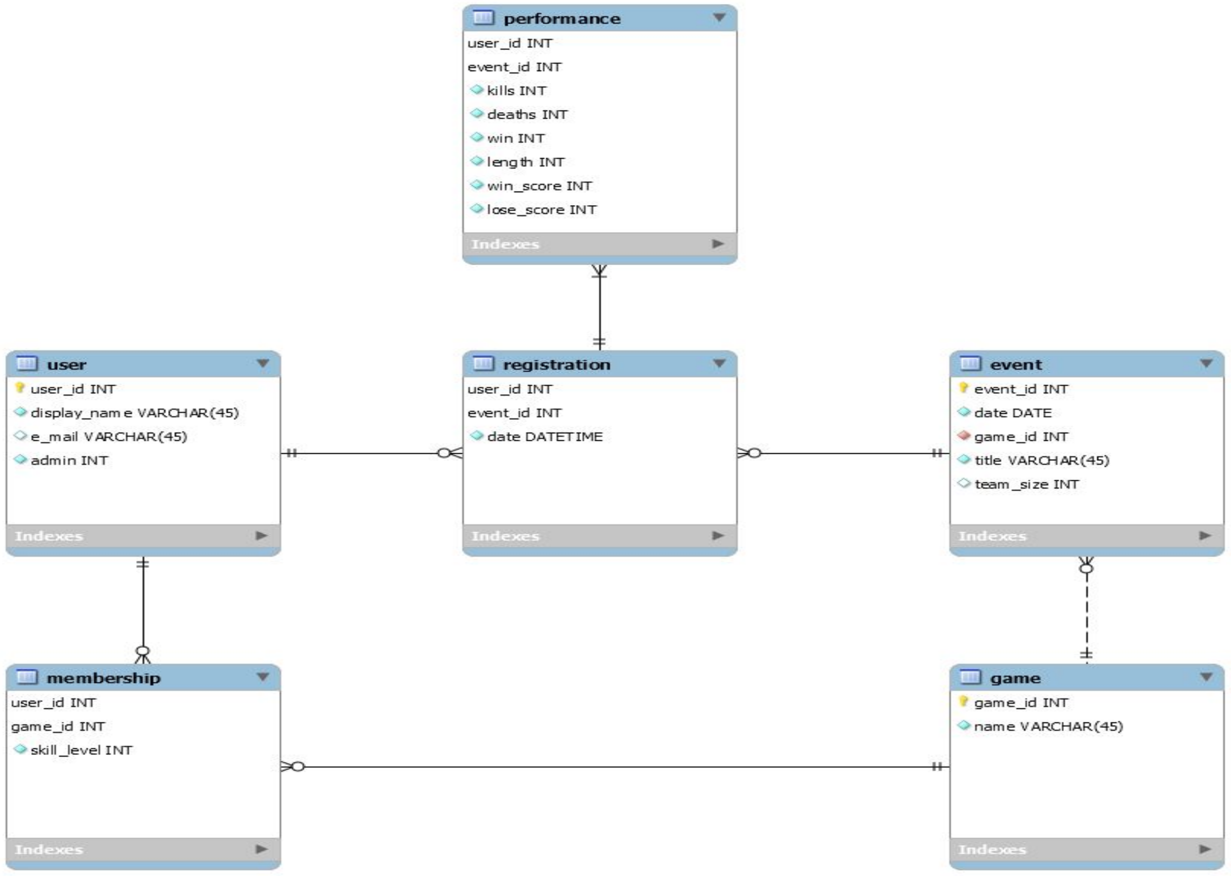


Figure 4: Bot Backend Database

The business rules for the database are as follows:

- A user registers for zero or more events; an event consists of zero or more users.
  - This n:n relationship necessitates a bridge entity: registration
- A user plays zero or more games; a game is played by zero or more users
  - This n:n relationship necessitates a bridge entity: membership
- An event is about exactly one game; each game may be played in zero or more events

- Each registration produces a performance; each performance matches exactly one registration

Below is an entity summary for the database:

- User:
    - user_id: the user's unique integer identifier
    - display_name: the user's Discord display name
    - e-mail: an (optional) way to contact the user
    - admin: 0 for no, 1 for yes; admins can change the database
- Event:
    - event_id: the event's unique integer identifier
    - game_id: the ID of the game being played in this event
    - date: the date of the event
    - title: name of the event
    - team_size: size of each team
- Game:
    - game_id: the game's unique integer identifier
    - name: the game's title (ex. League of Legends)
- Membership:
    - user_id: the ID of the user that plays the game
    - game_id: the ID of the game that the user plays
    - skill_level: a measure of the user's skill at the specified game (used for matchmaking)
- Registration:
    - user_id: the ID of the user making the registration
    - event_id: the ID of the event being registered for
    - date: date of the event being registered for
- Performance:
    - user_id: the ID of the user who competed
    - event_id: the ID of the event that was competed in
    - kills: measure of positive performance
    - deaths: measure of negative performance
    - win: did the player win this event?
    - length: length of the match
    - win_score: score of the winning team
    - lose_score: score of the losing team

3. CHALLENGES AND RISKS

3.1. Challenges

There were four primary challenges involved in the creation of LFJ:

1. Acquiring domain knowledge of Discord and specific games
2. Learning Discord's API and associated Python wrapper
3. Data collection and storage
4. Organization and design improvements

3.1.1. Acquiring Domain Knowledge

Acquiring domain knowledge was not particularly challenging, but was time consuming. There are a few things that we needed to know about Discord in order to implement LFJ, such as: how bots are able to see reactions to messages, what the API limits for bots are, how quickly interactions with the bot can be processed, how to obtain unique user and server IDs, how to represent messages as integer IDs, and how to add and delete bots from servers.

Most of these features were learned through experimentation with the discord.py library and through Discord's Developer Mode.

3.1.2. Learning Discord's API and discord.py

Perhaps the biggest part of this project was learning Discord's API structure and associated Python wrapper. The documentation for discord.py was a good starting point, and there were a number of publicly available articles and examples to view.

In particular, we spent a lot of time learning how to use the command extension to discord.py. This made it possible for the bot to automatically parse and implement commands without additional work on our part. Section 3.1.4 below details our experience in redesign and avoiding extra work via the commands extension.

3.1.3. Data Collection and Storage

We had originally viewed this part of the project as potentially being the most difficult. However, as work on implementation went on, we found that data collection, storage, and use were largely outside of the current scope of LFJ. We have given users the ability to collect and store statistics, but these statistics are not used for anything particularly useful right now. However, they provide a scaffolding that other developers can utilize.

Additionally, the design, implementation, and maintenance of the database back end was a lengthy process that involved multiple revisions. Although its basic structure never changed, fields and data types were added, deleted, and changed throughout development.

3.1.4. Organization and Design Improvements

The overall project organization and continuous design improvements were the most labor-intensive portions of the project. As mentioned above, we initially developed LFJ without using the commands extension. We were essentially reinventing the wheel: we had to develop command recognition, argument parsing, syntax checking, and message creation from scratch. This had time and readability costs that we discovered in late March.

We eventually found out that many of these features were already implemented in the commands extension of discord.py. We spent a good deal of time converting our first LFJ version into its current module structure. The current structure makes use of "cogs" to separate command functionality into clearly distinguished modules. Each cog deals with commands relating to a specific need, such as commands that deal with the user table, commands related to creating and deleting events, or commands that deal with registrations. Consult Section 5.6 for details about each module.

In addition to the module design, we decided to separate our code into two sections to take advantage of Python's built in testing system. What was originally a directory named "Scripts" was replaced with a "backend" directory split into library and testing code. This provided some needed clarity and allowed us to put all materials relating to testing into their own folder. However, it also introduced some difficulty with Python's import and path system. After a lot of experimentation, we settled on a structure in which you run the bot from the root directory, from which all imports are addressed.

3.2. Risks

In addition to the general design challenges listed above, there were specific problems and risks associated with each goal that we outlined when we started the project. The subsections below deal with the risks that faced each goal, while Section 3.3 deals with how we mitigated or failed to mitigate those risks.

3.2.1. Handling Event Signup

There were two related risks associated with the bot's ability to handle event signups: user privacy and group voting. Both of these issues were related in that the voting actions of users may have influence on the voting actions of other users. For example, some users may be less likely to sign up for an event if nobody else has signed up yet; alternatively, popular events may draw lots of votes without real user intent to attend.

3.2.2. Balancing Scrimmage Teams

Balancing scrimmage teams suffered primarily from a lack of method and data. Users will have to self-ascribe their skill levels in order to achieve a simple balancing method. There is also very little way to measure the success of this goal, since it will depend on long-term satisfaction (explicit feedback) or analysis of game statistics (implicit feedback such as game length and final score).

3.2.3. Posting Reminders for Scheduled Events

There were two major risks for this goal: relying on users to check their Discord notifications and users potentially changing their votes after the polling period ends. It is entirely possible that event notifications will be completely ignored by users, leading to wasted effort on our part. Finally, it is possible that users will withdraw their attendance to an event or want to join after the polling period is finished. Sending notifications to users who have withdrawn will be unnecessary, and users who want to sign up later may not receive any notifications about the event.

3.2.4. Allowing Members to Self-Assign to Groups

The risks relating to this goal were primarily related to user access control. The largest risk is with users not being able to sign up for events due to not having permission. An additional risk is users changing their status after signing up for an event and subsequently not getting relevant notifications. On one hand, it would be nice for users to be able to change their group status with a simple command; on the other, they may do this too often and break the backend database. Navigating these access control risks will have to be met with robust controls on which commands users are allowed to execute.

3.2.5. Relevant Statistics Collection and Querying

In addition to the challenge of relevant data collection, this goal also carried the long-term risk of usefulness. While we want to gather statistics to improve our matchmaking capabilities long-term, it may be possible that the statistics we choose to gather are irrelevant. Implementing this feature is obviously time consuming and design-intensive, so we must invest time up-front in background research and proper design.

3.3. Solutions

For each of the risks above, the subsections below detail how we mitigated or failed to mitigate each risk.

3.3.1. Handling Event Signup

The risks associated with this goal were primarily related to privacy and ensuring independent voting. LFJ did not end up addressing either of these risks, due in part to time and platform constraints. Users did end up reacting to a message in order to register for events, but there is no way to hide these counts since all messages ended up being public. There is also no easy way to time registrations, since this would require deleting the message after a certain time. This makes it impossible to send out reminders later on.

However, we have managed to build a system that is robust enough to ensure organization in event signup. There are fixed limits on team size and checks to ensure membership prior to registration.

3.3.2. Balancing Scrimmage Teams

Team balancing was perhaps the most difficult aspect of LFJ to get right. It required a lot of real-world testing and data, which we did not have. As such, our solution was a simple first-come, first-serve balancing with structures in place for future improvement and the option for random sorting of teams. Membership commands allow the input of user skill levels, and the data collection commands allow the data that can be used for more robust balancing to be collected and stored for future use.

Perhaps most importantly, LFJ gives a sense of organization to events and teams. There are clear, readable reports of who is attending each event. There is also a visualization of how many people can be on each team, as shown in Figure 2. Thus, while a robust team balancing system was not implemented, the structures are in place for future improvement. Additionally, the system does work in the short-term, even though it might not provide an ideal solution.

3.3.3. Posting Reminders for Scheduled Events

The risks associated with reminders were largely mitigated by the structure adopted. The bot creates events upon use request and posts them to a dedicated events channel, where users can react to the message to register. The registration system is binary: either register for the event (if there is space) or remove their registration.

The problems of changing their vote or getting users to look at the notifications were mitigated by this system. For one, the events channel provides a centralized location where users can see all active events; additionally, users have commands to retrieve information about specific events and their times. Reminders are sent out 24 hours before an event, giving users time to prepare and review their commitments. FInally, vote changes are handled in a simple way: simply deregister for the event and open up a space for other users.

3.3.4. Allowing Members to Self-Assign to Groups

Our solution to most access control problems was the introduction of admin status for specific users. Some users in the server are entered as admins, either upon database creation or with the bot itself. Only these users can add, delete, and change important parts of the database (such as user information, game information, event information, and registrations). This prevents unauthorized users from breaking the backend or deleting other users' events.

The introduction of a specific AdminPermissionError accomplished most of this checking. In addition, all unit tests make sure that AdminPermissionErrors are thrown in appropriate spots.

3.3.5. Relevant Statistics Collection and Querying

These were perhaps the most difficult risks to mitigate, since they involve long-term planning and execution. Seeing the use of these statistics long-term will require dedicated collection by team members. However, we were able to implement some automation that makes statistical collection easier and more streamlined. Users can submit .csv files of performances and retrieve them later. As mentioned above, this provides the scaffolding for others to enhance LFJ's functionality.

4. DEVELOPMENT TIMELINE

4.1. Development Model

For the completion of the LFJ project, we chose a scrum model. The scrum roles were as follows:

- Product owner: Elizabethtown E-Sports Team
- Scrum master: Ryan Ehmann* (since he is a member of the E-Sports team)
- Development team: Ian Bowler, Jonathan Wiseman, Matthew Grohotolski

*even though Ryan left the project, his ideas and knowledge about the needs of the E-Sports team informed much of the project's purpose and direction

The division of work was as follows:

- Jonathan: database design, initial command structure, project redesign, SQL commands interacting with the database, testing structure and some unit tests, CI setup
- Matthew: database changes, event creation, registrations, implementation of registrations as reactions, setup guide, project organization, helper command implementation, event formatting, response formatting, simple team balancing, SQL commands interacting with database
- Ian: server setup, project redesign, data collection and storage, code cleanup, error checking, custom errors, testing, CI setup, response formatting, SQL commands interacting with database

4.2. Timeline

Our initial timeline is available in previous versions of the LFJ project document. However, we suffered some initial delays in work, had to do an intensive redesign halfway through the project, and suffered some interruptions (Spring break and COVID-19 were pretty big disruptions).

Nonetheless, we usually met up weekly or bi-weekly to discuss our progress and take tasks for the week. We used a Kanban organizer to track our progress and assign tasks for the upcoming periods. While we didn't get to do a traditional Sprint cycle, our meetings were spread over roughly every two weeks and involved completion of specific tasks. Below is a rough outline of our development timeline:

Sprint 1: February 10 - February 24

This sprint focused primarily on planning and organization for the upcoming project. Noteable tasks:

1. First database design
2. Use case diagram
3. First project report

Sprint 2: February 24 - March 9

This sprint saw some initial implementation and the creation of our original project layout. Noteable tasks:

1. Working database creation via script
2. Implementation of user commands
3. Small documentation in user commands

Sprint 3: March 9 - March 23

This sprint saw some disruption from COVID-19 but work continued on our initial project implementation. Noteable tasks:

1. Implementation of event commands
2. Tweaks to user commands: ability to modify existing users
3. Separation of helper commands

4. Added ability to modify database via init_db.py

Sprint 4: March 23 - April 6

This sprint oversaw massive changes in the project's structure and the implementation of testing. Noteable tasks:

1. Project redesign overhaul
2. Error checking via custom errors
3. Testing using Python's testing system
4. Creation of CI system

Sprint 5: April 6 - April 20

This sprint saw the streamlined implementation of important commands, such as event creation and registration. Noteable tasks:

1. More work on CI integration
2. Data collection commands
3. Event creation via event channel
4. Registration via reactions
5. More testing

Sprint 6: April 20 - April 29

This final sprint saw some important work in finalizing the project. Noteable tasks:

1. Sending reminders
2. Updating project document
3. Presentation creation

5. IMPLEMENTATION

5.1. Testing

A suite of unit tests was developed with coverage of most of the system's functions. Many of the functions that deal directly with the LFJ database backend are covered in the tests. Test scripts were organized according to function; there are three test scripts:

1. test_event_queries.py
2. test_game_queries.py
3. test_user_queries.py

Running the suits can be accomplished with the following commands:

```
python -m unittest backend.tests.test_user_queries
python -m unittest backend.tests.test_game_queries
python -m unittest backend.tests.test_event_queries
```

5.2. Build Automation

We chose to use Buddy as our automated build system. It was integrated with our Github repository. There are two active pipelines: one for the master branch and one for the testing branch. All code should be run through the testing branch and the build verified before pushing to the master branch.

5.3. Code Review

In addition to reviews on branch merge requests, we have elected to use Codefactor for automated code review. Our current Codefactor score is an A, and the report is available in our Github repository.

5.4. Implemented Commands

There are twenty-two commands available in LFJ right now, organized into 6 categories:

User Queries
These are queries that allow interaction with the user table:

1. add_user
2. delete_user
3. query_user
4. set_admin_status

Game Queries
These are queries that allow interaction with the game table:

5. add_game
6. delete_game
7. query_game
8. edit_id
9. edit_name
10. list_games

Event Queries
These are queries that allow interaction with the event and registration tables:

11. create_event
12. delete_event
13. get_events
14. query_event
15. sort_teams

Membership Queries
These are queries that allow interaction with the membership table:

16. create_membership

17. delete_membership
18. set_skill

Performance Queries
These are commands that allow the input of game statistics.

19. perf_template
20. perf_update

Miscellaneous Commands
21. help
22. exit

You can specify which prefix is used to address the bot by changing the configuration file. Documentation of each command and its syntax is available at the Github repository.

5.5. Goals Met

Each of our major goals was met, with varying degrees of coverage and success. Goals 1, 4, and 5 were met in full. The commands that implement these goals are clearly defined, usable, and documented. While data collection is general and largely unutilized, it is present and workable. Event signup is robust and interactive, and users can self-assign to groups easily.

Goals 2 and 3 are met with large caveats. While teams are balanced in terms of numbers, they are not balanced in terms of skill level or past performance. Reminders are also general and are only posted 24 hours before an event begins. Additionally, creating an event does not notify specific users that there is an event they might be interested in. However, as mentioned previously, the scaffolding is there for these features to be implemented or expanded upon.

5.6. Implementation Details

The system is split into two primary components:

1. Database files
2. Backend implementation code

There are two major database files: dump.sql and lfj.sql. The dump.sql file is used in the continuous integration system to initialize a containerized MariaDB server. The lfj.sql file can be used by the user to initialize the database on his/her machine.

The backend implementation code is split into two directories: library code and testing code. Library code includes the following:

1. event_actions.py: allow users to register for events via reactions
2. event_queris.py: create, delete, and query events
3. game_queries.py: create, delete, and query games
4. helper_commands.py: functions and errors common to multiple modules

5. init_db.py: a helpful script that allows users to modify the backend and initialize with users
6. performance_queries.py: implements statistics gathering
7. user_queries.py: add, delete, and modify users

6. REFERENCES

1. Discord Inc. Company. (February 9, 2020). Retrieved February 9, 2020 from https://discordapp.com/company.

2. DreamExposure. DisCal-Discord-Bot. (August 6, 2019). Retrieved February 9,2020 from https://github.com/DreamExposure/DisCal-Discord-Bot

3. ForYaSee. voteBot. (October 17,2019). Retrieved February 9, 2020 from https://github.com/Votebot/voteBot

4. Matnad. Pollmaster. (September 1, 2019). Retrieved February 9, 2020 from https://github.com/matnad/pollmaster

5. Muddyfish. Discord API. (February 8, 2020). Retrieved February 9, 2020 from https://github.com/discordapp/discord-api-docs

6. Rapptz. Discord.py. (February 2, 2020). Retrieved February 9, 2020 from https://github.com/Rapptz/discord.py

7. Riot Games. Our Story. (2017). Retrieved February 9, 2020 from https://www.riotgames.com/en/who-we-are/values.

8. Valve Corporation. About. (2020). Retrieved February 9, 2020 from https://store.steampowered.com/about/.

9. Vox Light. Discord-Balance-Bot. (May 7, 2018). Retrieved February 9, 2020 from https://github.com/VoxLight/Discord-Balance-Bot

10. Wilhelmklopp. Simple-poll. (May 19, 2016). Retrieved February 9, 2020 from https://github.com/wilhelmklopp/simple-poll

6. FEEDBACK

(2/9/2020) All feedback has been addressed.

Link to original proposal: EtownLFG