Discord Bot – Cordic

Schülerrechenzentrum Dresden

Schuljahr 2021/22

Beteiligte:	Ahmad Yosef Florian Weinert Nick Edelhäuser
AG-Leiter:	Herr Stock
Firma:	Communardo Software GmbH
Firmenvertreter:	Herr Enrico Kutscher

Gliederung

1 Ziel der Arbeit

Wir haben von der Firma Communardo Software GmbH die Vorgabe bekommen einen Discord-Bot zu schreiben, welcher Teile der internen Projektarbeit unterstützt und dafür an das interne Tooling von Communardo angeschlossen ist. Dies ist in unserem Fall die Jira Software. Wir wollten also die Möglichkeit für Firmen bieten an einem zentralisierten Ort zu kommunizieren und ihre Projekte zu verwalten. Deshalb haben wir uns dazu entschieden die Projektverwaltungssoftware Jira von Atlassian mit der Online-Chatplatform Discord zu verbinden. Dafür nutzen wir die Discord API in Verbindung mit einem Python-Modul namens Nextcord um einen sogenannten „Bot“ zu schreiben. Diesen kann man sich wie einen normalen Nutzer von Discord vorstellen, nur dass dieser über Code angesteuert werden kann und bestimmte Aktionen in Discord automatisieren kann.

2 Nutzungshinweise

Damit ein Nutzer die Funktionen unseres Bots nutzen kann, muss er über einen Discord Account sowie über einen Chat-Raum genannt „Server“ verfügen. In diesen kann man den Bot dann über einen Link einladen und autorisieren.

Um auch die Jira-Funktionalität des Bots nutzen zu können, muss sich jedes Mitglied des Servers, welches die Jira-Features unseres Bots nutzen will, bei dem Bot registrieren. Dies kann über den bereitgestellten „Connect“ Befehl getan werden. Dabei müssen die Account Daten von Jira, sowie ein Jira API Token als Argumente übergeben werden. Da dies aber sensible Daten sind, ist der Befehl so konfiguriert, dass nur der Nutzer die Rückgabe des Befehls sehen kann, der ihn auch ausgeführt hat. Somit kann kein anderes Mitglied des Servers die Login Daten eines anderen Mitglieds über diesen Befehl herausfinden.

Die Befehle/ Funktionen des Bots können in vier Kategorien eingeteilt werden: Generelle Befehle, Jira-bezogene Befehle, Termin-Befehle und Debug-Befehle.

2.1 Voraussetzungen und Installation

1) **Vorraussetzungen:**

    1. Discord Account mit 2FA und eingeschalteten Entwicklermodus (Aktivierung in den Benutzereinstellungen im Unterpunkt „Erweitert“)
    2. Discord-Server auf dem der Bot genutzt werden soll
    3. Atlassian Account
    4. Jira Cloud Instanz (Jira Cloud Instanz für Developer ist kostenlos)
    5. Python 3.9 lokal installiert

2) **Bot über Discord erstellen**

1. Beim Discord Developer Portal (https://discord.com/developers/) mit dem Discord Account einloggen

2. Anschließend muss man eine neue Applikation erstellen und bei dem Unterpunkt „Bot“ muss man den Bot über „Add Bot“ hinzufügen

3. Nun muss man den Discord Bot auf ihren Server einladen. Dafür nehmen sie folgenden Link und tragen sie die Client ID des Bots in die URL ein. „Permission=8“ steht hierbei dafür, dass sie dem Bot auf dem Server Administrator Rechte geben

4. https://discord.com/api/oauth2/authorize?client_id=<id>&scope=bot%20applications.commands&permissions=8

3) Repository von Github clonen

1. “cd <Projektordner>”

2. “git clone https://github.com/ARez2/Cordic.git”

3. “cd DiscordManagementBot”

4) Venv anlegen und aktivieren

1. “python -m venv venv”

2. “venv\Scripts\activate“ (bzw auf *nix „source venv/bin/activate

5) Alle benötigten Module installieren

1. “pip install -r requirements.txt”

6) „GUILD_IDS“ (= Server ID, auf dem der Bot verwendet wird) Variable entsprechend bearbeiten

1. Aktiviere den Entwicklermodus auf deinem Discord-Account (Aktivierung in den Benutzereinstellungen im Unterpunkt „Erweitert“)

2. Klicke auf den Server, auf dem der Bot verwendet werden soll

3. Rechtsklicke den Namen des Servers und wähle den Punkt „ID kopieren“ aus

4. Weise der Variable „GUILD_IDS“ in „src\lib\discord_channel.py“ die ID des Servers zu

7) Discord API Token

1. Der Discord API Token wird über das Discord Developer Portal (https://discord.com/developers/) erstellt. Dafür müssen sie in die Applikation des

Discord-Bots namens „Cordic“ reingehen und im Unterpunkt „Bot“ den Token generieren

2. Erstelle im Hauptordner eine .env-Datei
3. Trage den Token nach folgendem Schema ein: „DISCORD_TOKEN=<Euer Token>“

8) Bot ausführen

1. „cd <Projektordner>\DiscordManagementBot“
2. Für Windows: “venv\Scripts\activate”, Für *Nix: „source venv\bin\activate“
3. python cordic.py (*nix: python3 cordic.py)

9) Allgemeiner Hinweis

1. Wählen sie bei der Benutzung des Bots zuerst den richtigen Command Client aus und zeigen sie sich über den Help Befehl des jeweiligen Command Client an, wie man die Befehle des Discord Bots richtig ausführt

10) Hinweise zu Jira

1. Um auf seine Jira Account ID zuzugreifen, loggt man sich zunächst auf Atlassian.com ein und wählt dann den Jira Work Management Punkt aus.
2. Daraufhin kann man oben rechts bei seinem Profilbild klicken und auf "Profil" gehen. Dann sieht man in der URL seine ID (letzter Teil der URL).

3. Seinen Jira Api-Token kann man in den Kontoeinstellungen im Unterpunkt „Sicherheit“ erstellen

3 Programmentwicklung

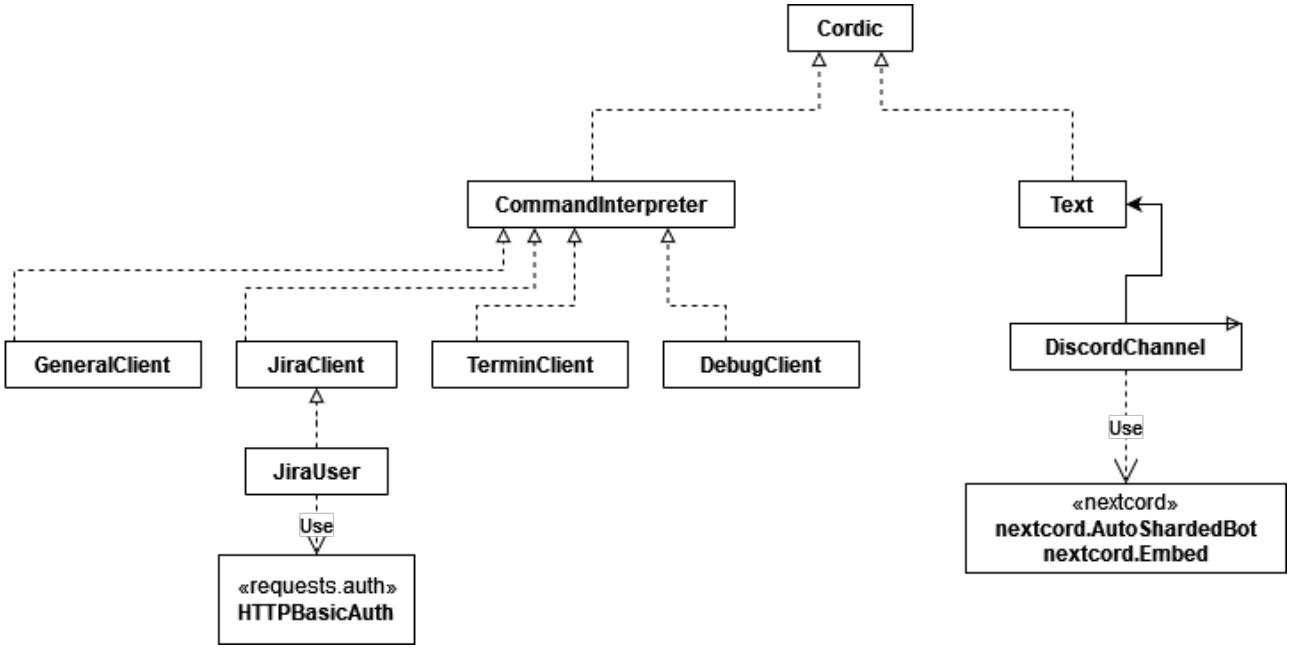
3.1 Genutzte Tools

Als Programmiersprache haben wir Python gewählt, da es auf Grund seiner Vielzahl an Modulen sehr universell nutzbar ist und auch für die Entwicklung eines Discord Bots einige Module anbietet. Zusätzlich machen einfach verständlichen Syntax und unsere Erfahrung mit Python, diese Programmiersprache perfekt geeignet für unsere Jahresarbeit. Die Schnittstelle zur Discord API wird durch das Modul nextcord realisiert. Für die Datenbanken wird SQLite3 und die darauf basierende Erweiterung SQLAlchemy verwendet. Um die Installationen und Einstellungen für das Projekt abzukapseln haben wird das Modul venv genutzt, welches es erlaubt Virtuelle Umgebungen zu erstellen. Zur Versionsverwaltung haben wird Git bzw. GitHub verwendet.

Im folgenden werden unter Umständen die Wörter Nextcord und Discord als Synonym verwendet (bsp. „Discord stellt … bereit“). Dies ist damit zu begründen, dass das Modul Nextcord die Funktionen welche durch Discord mithilfe eine REST-API angeboten werden, grundsätzlich nur in einfach handzuhabende Klassen verpackt. Dabei ist Nextcord selber aber auf die vorgegeben

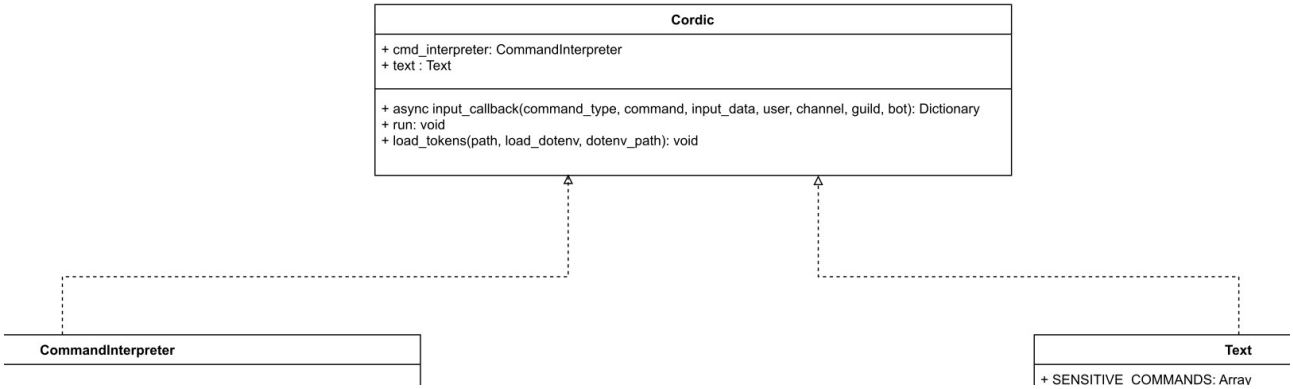
Funktionen von Discord beschränkt und fügt keine neue Funktionalität hinzu, welche von Discord nicht angeboten wird.

3.2 Aufbau des Programms



Das Programm wird von der Oberklasse Cordic aus initialisiert. Beim Aufruf des Python Programms wird ein Objekt der Klasse Cordic angelegt, welches dann in seiner „__init__“ Funktion Objekte der Klassen CommandInterpeter und Text anlegt. Dabei kann man unsere Programmstruktur generell in zwei Teile gliedern: Eingabe und Ausgabe sowie die Verarbeitung. Eingabe und Ausgabe werden von der Text Klasse übernommen und die Verarbeitung durch den Command Interpreter bzw. seine sogenannten „Command Clients“. Dies sind ebenso Objekte der Klassen GeneralClient, JiraClient, TerminClient oder DebugClient welche vom Command Interpreter instanziert werden. Diese Objekte stellen Funktionen für ihren jeweiligen Aufgabenbereich bereit. Die JiraClient Klasse zum Beispiel stellt Funktionen bereit, welche u.a. Daten von Jira abfragen können.

3.2.1 Cordic Klasse



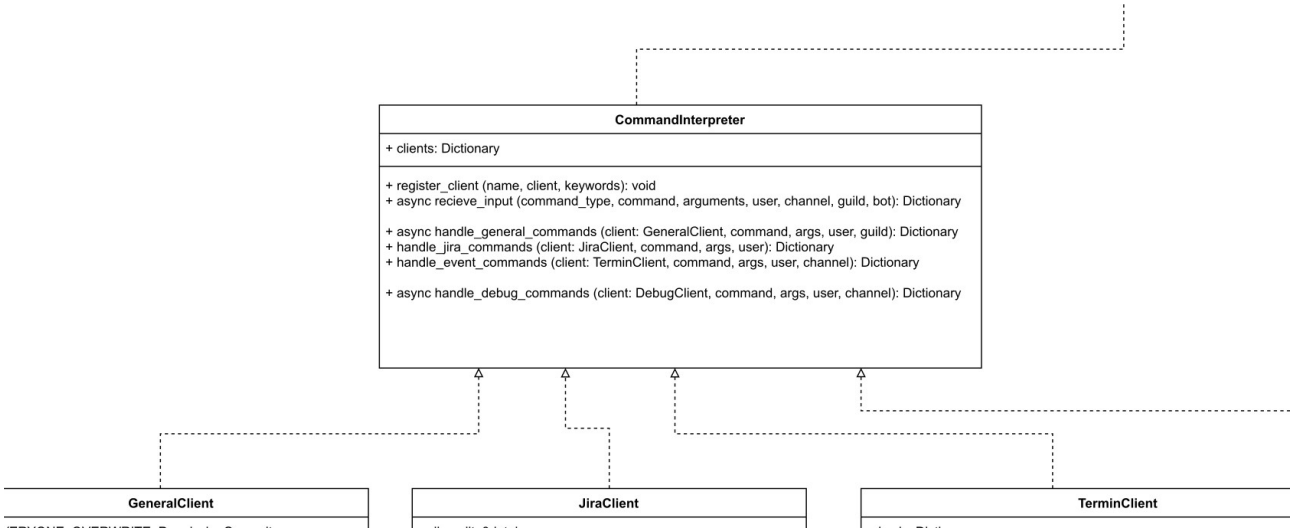
Die Cordic Klasse ist unsere Haupt-Klasse. Sie dient als Kommunikations-Brücke zwischen Eingabe bzw. Ausgabe und dem verarbeitenden Teil des Programms. Sie lädt beim Anlegen eines Objektes von Cordic die benötigten Tokens, welche dann an die Objekte der instanziierten Klassen (Text und CommandInterpreter) weitergegeben werden. Ein Beispiel dafür ist der Discord Token, welche die Discord API zur Authentifizierung eines jeden Bots braucht.

Des weiteren verfügt die Cordic-Klasse über die „input_callback“ Funktion.

```
async def input_callback(self, command_type, command, input_data, user, channel, guild, bot):
    return await self.cmd_interpreter.receive_input(command_type, command, input_data, user, channel, guild, bot)
```

Diese Funktion wird beim Initialisieren des Textklassen-Objektes an dieses übergeben. Die Text-Klasse ruft dann diese Funktion mit den Daten aus Discord (Eingabe-Text, Autor der Nachricht etc.) auf. Die „input_callback“ Funktion leitet diese Eingaben dann an die „receive_input“ Funktion des Command Interpreters weiter, welche die Eingabe mithilfe der Command Clients verarbeitet und ein Dictionary zurückgibt, welches dann in der „input_callback“ Funktion an die Text-Klasse zurückgegeben wird. Somit läuft die gesamte Kommunikation von Eingabe/ Ausgabe und Verarbeitung über die Cordic-Klasse. Diese Struktur wurde mit Absicht gewählt, um eine hierarchische Aufgliederung des Programms zu ermöglichen und das Programm somit leichter verständlich zu machen.

3.2.2 Command Interpreter



Der Command Interpreter ist zuständig für die Verarbeitung der Eingabedaten. Wie schon im Abschnitt „Cordic Klasse“ erwähnt, besitzt er ein Dictionary an Command Clients. Diese werden bei der Initialisierung der Klasse in dieses Dictionary eingetragen.

```python
class CommandInterpreter():
    def __init__(self) -> None:
        self.clients = {}
        self.register_client("general", GeneralClient(), ["general", "g"])
        self.register_client("jira", JiraClient(), ["jira", "j"])
        self.register_client("event", TerminClient(), ["event", "e"])
        self.register_client("debug", DebugClient(), ["dbg", "debug"])


    def register_client(self, name, client, keywords):
        if name in self.clients:
            return
        self.clients[name] = {
            "client": client,
            "keywords": keywords,
        }
```

Die „register_client“ Funktion sorgt dafür, dass man später beim Aufruf mit dem Input auch Abkürzungen für den Befehlstyp eingeben kann.

Von der Cordic-Klasse aus wird die „receive_input“ Funktion aufgerufen. Diese hat verschiedene Eingabedaten als Argumente wie z.B. Befehlstyp, Befehl, Befehlsargumente, den Kanal wo der Befehl ausgeführt wurde etc.

```python
    async def receive_input(self, command_type, command, arguments, user, channel, guild, bot):
        if arguments is not None:
            arguments = arguments.split(" ")
        target_client = None
        for c in self.clients:
            if command_type in self.clients[c]["keywords"]:
                command_type = c
                target_client = self.clients[c]["client"]
                break
```

Dabei wird am Anfang der „receive_input“ Funktion geprüft, welche Instanz im Dictionary der Command Clients dem „command_type“ Input entspricht. Dabei werden auch mithilfe einer standartisierten Fehlermeldung mögliche erste Fehler abgefangen.

```
        embedstructure = {
            "title": "Error",
            "description": "Could not find command: Maybe you are missing `t`/`t
            "fields": None,
            "color": COLOR_RED
        }
        output = embedstructure

        if target_client is None:
            return output
```

Nun wird einfach für den Command Client, welcher der Eingabe „command_client“ entspricht eine Funktion aufgerufen, welche die Argumente verarbeitet, eventuell formatiert und dem richtigen Command Client übergibt. Der Rückgabewert dieser Funktion ist immer ein Dictionary und wird von der „receive_input“ Funktion schließlich an Cordic zurückgeben.

```
        if command_type == "general":
            output = await self.handle_general_commands(target_client, command, arguments, user, guild)
        elif command_type == "debug":
            target_client.init_bot(bot)
            output = await self.handle_debug_commands(target_client, command, arguments, user, guild)
        elif command_type == "jira":
            output = self.handle_jira_commands(target_client, command, arguments, user)
        elif command_type == "event":
            output = self.handle_event_commands(target_client, command, arguments, user, channel)

        return output
```

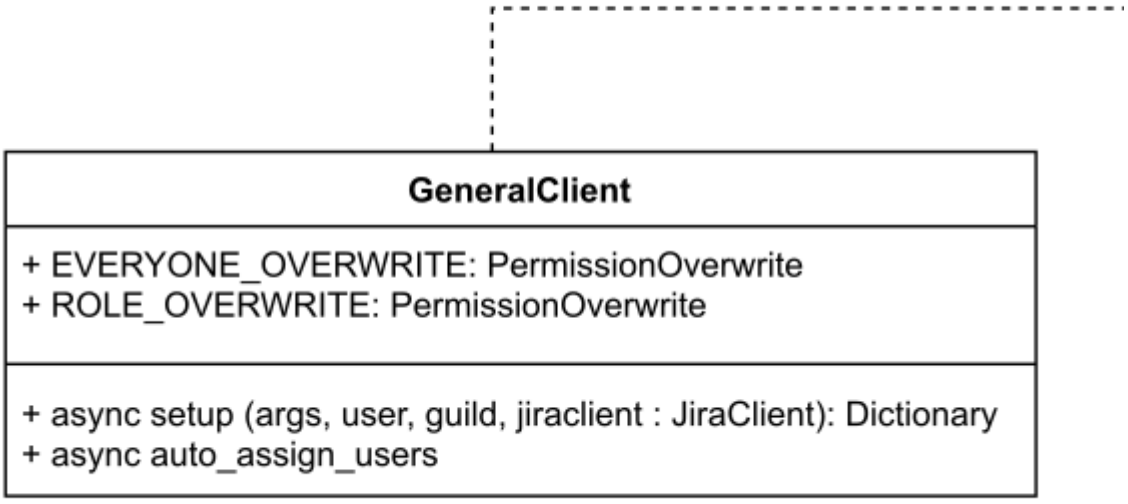
3.2.3 Command Clients

Im Folgenden wird am Beispiel des „JiraClients“ der generelle Aufbau der Funktionen („handle_jira_commands“, „handle_event_commands“ etc.) erklärt. Jede dieser Funktionen bekommt als Argument einen Client von der „receive_input“ Funktion. Dies ist ein Objekt des Command Clients, welcher von der Eingabe/ dem Nutzer angesteuert werden soll.

```
73      def handle_jira_commands(self, client: JiraClient, command, args, user):
74          try:
75              connect_kw = ["connect_user", "connect"]
76              initdomain_kw = ["setup_domain", "domain"]
77              userissues_kw = ["issues_by_user", "issues", "userissues"]
78              issuedetails_kw = ["issue_details", "details"]
79              projects_kw = ["projects"]
80              userprojects_kw = ["projects_by_user", "userprojects"]
81              if command in connect_kw:
82                  if args is None:
83                      return {
84                          "title": "ERROR",
85                          "description": "Missing Arguments!",
86                          "color": COLOR_RED
87                      }
88                  else:
89                      return client.connect_user({
90                          "dc_user": user,
91                          "email": args[0],
92                          "jiratoken": args[1],
93                          "account_id": args[2],
94                      })
```

Dafür können am Anfang der Funktion Listen mit ein paar Schlagwörtern definiert werden, damit der Befehl bei der Eingabe auch abgekürzt werden kann. Als nächstes wird geprüft ob der eingegebene Befehl („command“) in einer der Listen vorkommt. Wenn ja werden eventuelle Fehler aufgrund von fehlenden Eingabeargumenten mit einer Fehlermeldung abgefangen. Wenn alle geforderten Argumente vorhanden sind, wird die Funktion des Command Clients aufgerufen, welche von der Eingabe durch „command“ angesprochen wurde (siehe Zeile 89). Der Funktion werden die Argumente übergeben und ihr Rückgabewert wird and die „receive_input“ Funktion zurückgegeben.

3.2.3.1 General Client



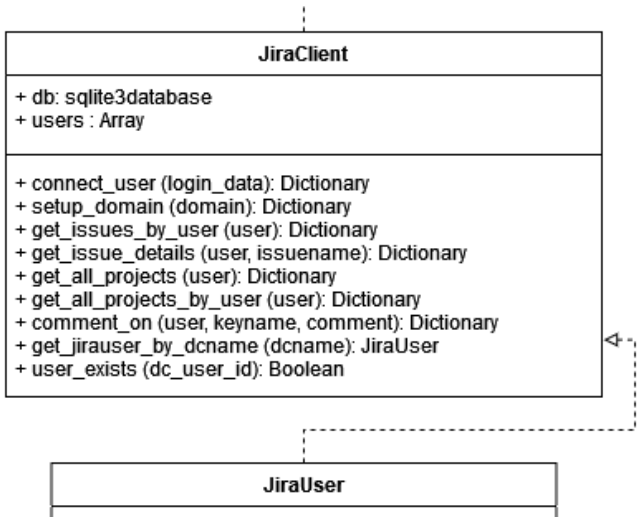
Der General-Client stellt bisher ausschließlich die Setup-Funktion bereit, welche allerdings sehr wichtig für den Bot ist. Sie nutzt Florians Jira-Client um beim Eingeben des Setup Befehls ein

neues „Projekt“ auf dem Discord-Server der Firma aufzusetzen. Dazu wird eine Kategorie mit einem Text Kanal innerhalb von Discord erstellt. Eine Kategorie ist eine Möglichkeit in Discord Text- und Sprachkanäle zu gruppieren. Sie erlaubt außerdem das Aufsetzen von Zugriffsberechtigungen, die für alle Unter-Kanäle gelten, genannt „Rolle“. Eine Rolle kann einem Benutzer zugewiesen werden, um ihm die Rechte/ Autorität dieser Rolle zu gewähren. Im Setup Befehl wird deshalb eine Rolle für das anzulegende Projekt erstellt, welchem durch die „auto_assign_users“ Funktion automatisch Mitglieder des Servers zugewiesen werden.

```
60      async def auto_assign_users(self, domain_and_project_name, rolename, role, guild, jiraclient):
61          auto_assigned_someone = False
62          async for user in guild.fetch_members(limit=None):
63              if user.bot: continue
64              project_embed = jiraclient.get_projects_by_user(user)
65              project_list = project_embed["fields"]["projects"]
66              if len(project_list) <= 0:
67                  continue
68              for project in project_list.split(","):
69                  if project.lower() == domain_and_project_name.lower() and nextcord.utils.get(user.roles, name=rolename) is None:
70                      auto_assigned_someone = True
71                      await user.add_roles(role)
72          return auto_assigned_someone
```

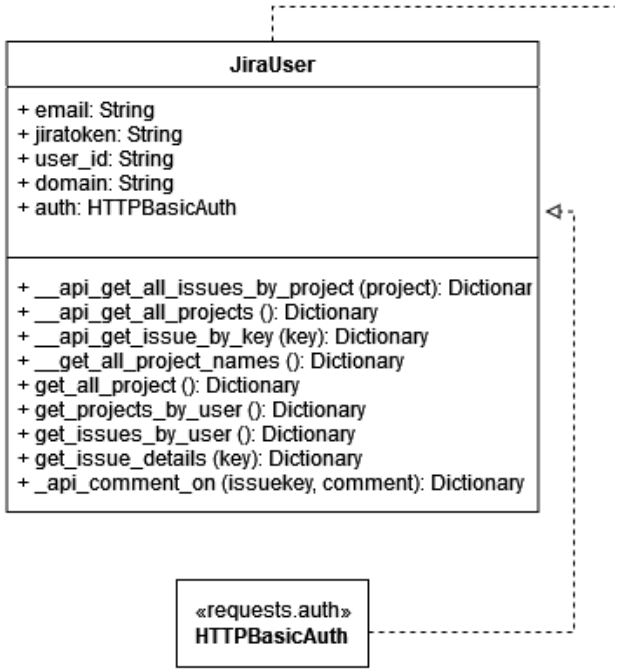
Dafür wird die Funktion „get_projects_by_user“ des Jira-Clients verwendet (siehe Zeile 64 im oberen Bild). So kann direkt aus Jira ausgelesen werden, welcher Nutzer an welchem Projekt teilnimmt (vorausgesetzt er ist in die Datenbank des Bots eingetragen). Wenn der Bot einen Nutzer findet, der an einem Jira-Projekt mitarbeitet, welches den selben Namen wie der Inhalt des ersten Arguments (welches den Projektnamen beinhaltet), wird dieser Nutzer der Rolle zugewiesen und kann somit die Kategorie für das Projekt sehen. Alle Mitglieder des Servers (Admins ausgeschlossen) können die Kategorie (ohne die Rolle) nicht sehen. So wird die Oberfläche für den Nutzer sauber und übersichtlich gehalten.

3.2.3.2 Jira Client



Der Jira Client wird wie bereits erwähnt beim Command Interpreter registriert. Über den Command Interpreter werden dann im JiraClient die Funktionen aufgerufen. Die connect_user Funktion übergibt dabei die Argumente an die Funktion connect_user der jiradatabase Klasse. Über diesen

Weg werden dann sozusagen die Daten in der Datenbank abgespeichert, welche den Discord Account des Nutzers, sowie den Jira Account miteinander verknüpft. Darunter fallen die einzigartige Discord-ID, die einzigartige Jira-ID und des Weiteren die E-Mail-Adresse vom Jira Account und ein API-Token, welche beide für die Authentifizierung bei der Rest-API von Jira/Atlassian notwendig sind. Die Funktion „setup_domain“ registriert den Namen der Domain der Jira Cloud Instanz über den gleichen Weg in der Datenbank.



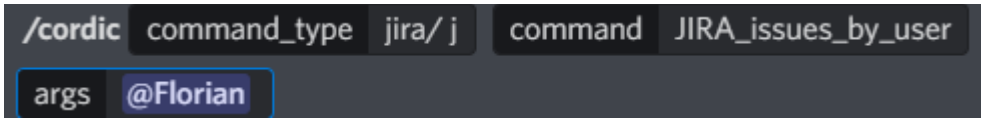
Die anderen Funktionen im JiraClient erschaffen als erstes ein Objekt der JiraUser Klasse. Dabei werden über die Discord ID aus der Datenbank durch die Klasse jiradatabase die für den Nutzer entsprechenden Daten (e-mail, token, domain) an das Objekt übergeben.

```
91      def get_jirauser_by_dcuserID(self, dcuserID):
92          if self.db.get_domain() != None:
93              user = JiraUser(self.db.get_user_email(dcuserID), self.db.get_user_jiratoken(dcuserID), dcuserID, self.db.get_domain())
94              return user
```

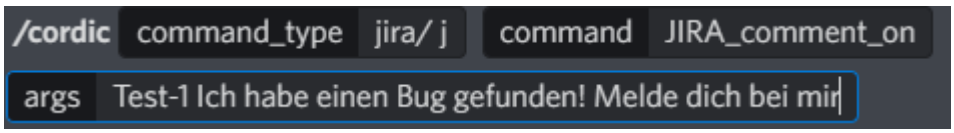
Mit der E-Mail und den API-Token wird die HTTP-Anfrage für die Rest-API zu der Jira Cloud Instanz authentifiziert. Somit bildet ein Objekt der Klasse JiraUser einen einzelnen DiscordNutzer, mit seinen Zugangsdaten und den Funktionen dar. Es gibt also keinen administrativen Token, welcher für alle Nutzer gleichermaßen gilt. So bekommt also jeder Nutzer auch nur die Informationen von der API, bei denen er berechtigt ist, sie zu sehen. Die Funktionen mit den Zusatz „__api“ greifen hierbei auf die API-Schnittstelle zu. Sie übergeben den anderen Funktionen entweder die Ergebnisse in Form eines JSON oder eine Fehlermeldung in Form eines String. Die Fehlermeldung und die Ergebnisse werden dabei in den restlichen Funktionen ausgewertet und die Ergebnisse in ein Dictionary umgewandelt. Die Struktur des Dictionary entspricht dabei der typischen Struktur, welche wir für die Umwandlung in ein Embed in der Text-Klasse benötigen. Sie besteht dabei aus dem Titel, der Beschreibung, den einzelnen Feldern und der Farbe des Embeds.

```
353             fields = {
354                 "project": str(issue["fields"]["project"]["key"]),
355                 "projecttype": str(issue["fields"]["project"]["projectTypeKey"]),
356                 "issuetype": str(issue["fields"]["issuetype"]["name"]),
357                 "status": str(issue["fields"]["status"]["name"]),
358                 "priority": str(issue["fields"]["priority"]["name"]),
359                 "created": str(issue["fields"]["created"]),
360                 "lastUpdate": str(issue["fields"]["updated"]),
361                 "duedate": str(issue["fields"]["duedate"]),
362                 "assignee": assignee,
363                 "descriptions": descriptions,
364                 "link": f"https://{self.domain}.atlassian.net/browse/{key}"
365             }
366             embedstructure = {
367                 "title": key + ": " + str(issue["fields"]["summary"]),
368                 "description": None,
369                 "fields": fields,
370                 "color": COLOR_BLUE
371             }
```

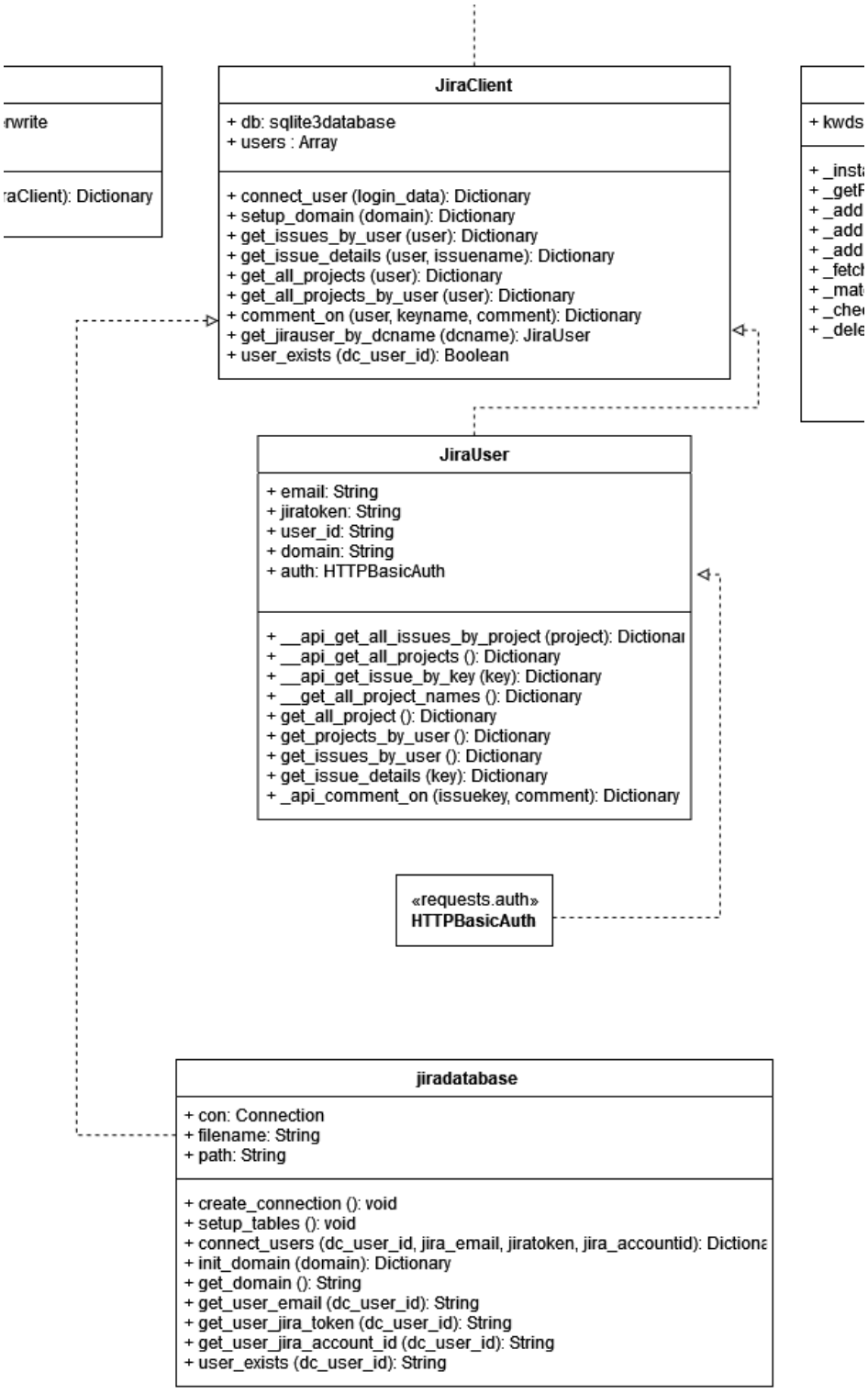
Die ersten drei API Funktionen nutzen dabei HTTP-Get und die Kommentar Funktion http-Post. Die „__get_all_project_names“ Funktion ist dabei eine Hilfsfunktion, welche die anderen Funktionen für die Auswertung der Daten benötigen und die eninzelnen Keynamen eines jeden Projektes zurückgibt (Die Keynamen für das Projekt sind einzigartig für jede Cloud Instanz und werden bei der Erstellung eines Projektes festgelegt). Die Ergebnisse der restlichen Funktionen kann man auf Discord bei der Benutzung des Discord-Bots sehen. Der „Jira_help“ Befehl gibt eine Liste aller Befehle mit deren Funktionsweise und Argumenten zurück. Mit dem „Jira_connect_user“ Befehl kann man den Discord Nutzer mit einen Jira Account verbinden und mit dem „Jira_setup_domain“ wird die Domain festgelegt (Die JiraID findet man in der URL bei seinem Profil und den API-Token kann man bei Jira in den Kontoeinstellungen erstellen). Bei der Funktion „Jira_issues_by_user“ kann man die Aufgaben von sich selber oder von jemanden anderen einsehen, wenn man diese Person in den Argumenten anpingt.



Mit „Jira_issue_details” kann man die Details einer Aufgabe einsehen und mit „Jira_projects“ alle Projekte, welche aktuell auf der Cloud Instanz laufen. Der „Jira_projects_by_user“ Befehl gibt alle Projekte zurück bei denen man beteiligt ist oder alle Projekte bei den jemand anderes beteiligt ist, wenn man diesen in den Argumenten anpingt. Mit „Jira_comment_on“ kann man unter einer Aufgabe einen Kommentar dalassen.



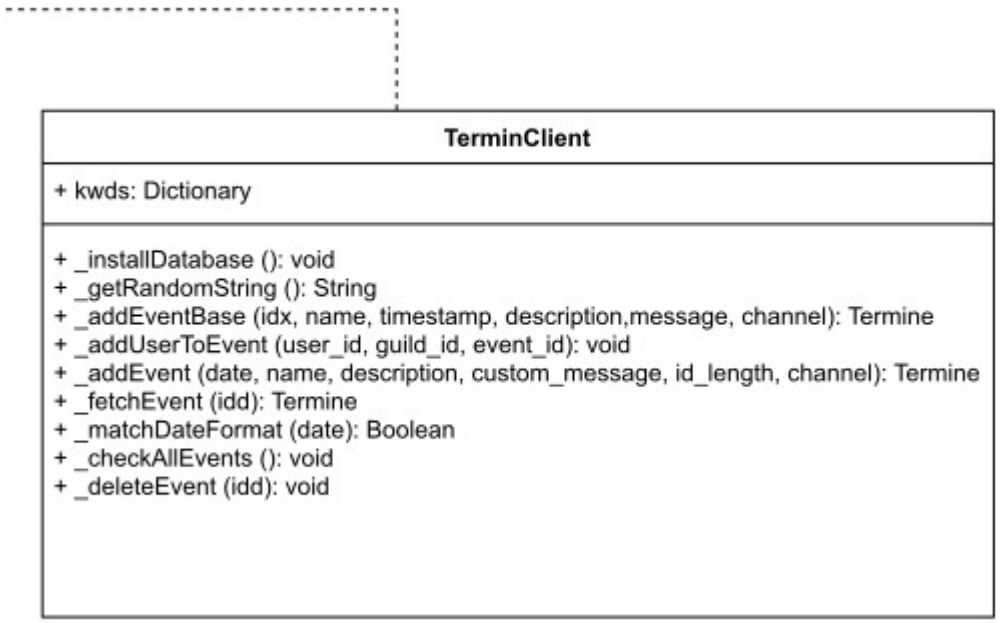
Jiradatabase



Die Klasse jiradatabase nutzt für den Zugriff auf die Datenbank SQlite3. Durch die Funktion „_create_connection“ wird ein Cursor (Verbindung zum Datenbankfile) erzeugt, welcher auf die Datenbank „database.db“ zeigt. Durch die „setup_tables“ Methode wird überprüft, ob alle Tabellen, welche für den JiraClient notwendig sind, vorhanden sind und erstellt sie gegeben falls. In der Tabelle users, werden die Zugangsdaten (wie bereits im JiraClient erwähnt) abgespeichert und in der Tabelle domain, die domain der Jira Cloud Instanz. Die Funktion „connect_user“ trägt dabei die Userdaten in die Tabelle users ein und die Funktion init_domain schreibt die domain in die Tabelle

domain. Falls sich ein User zweimal verbindet, werden seine alten Daten überschrieben. Dasselbe gilt für die Domain. Die get-Funktionen sind dabei, wie beim JiraClient erwähnt, wichtig, um den JiraUser über die DiscordID mitsamt seinen Zugangsdaten registrieren zu können.

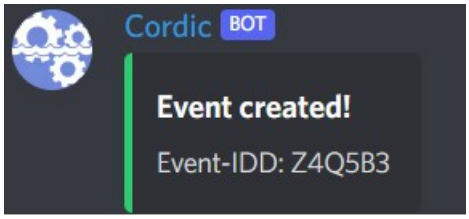
3.2.3.3 Termin Client



Dieser Client ermöglicht Terminverwaltung.

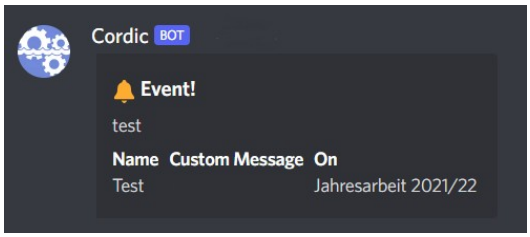
Wird ein Event per Command erstellt, wird zuerst addEvent() aufgerufen und dann wird der Nutzer, der das Event erstellt hat per addUserToEvent als Teilnehmer registriert.

Nach Registrierung eines Termins antwortet der Bot mit einer Nachricht, die die ID des Events zeigt



Diese ID ist einzigartig und wird von allen weiteren Funktionen zur Identifizierung des Events genutzt.

Mit Hilfe des EVENT_join-Commands kann man sich als Teilnehmer einschreiben. Wenn das Event startet, bekommen alle, die beigetreten sind, eine Direktnachricht, die ihnen sagt, dass ein Event gestartet ist.



Das System, dass nach laufenden Terminen sucht, ist sehr simpel konstruiert:

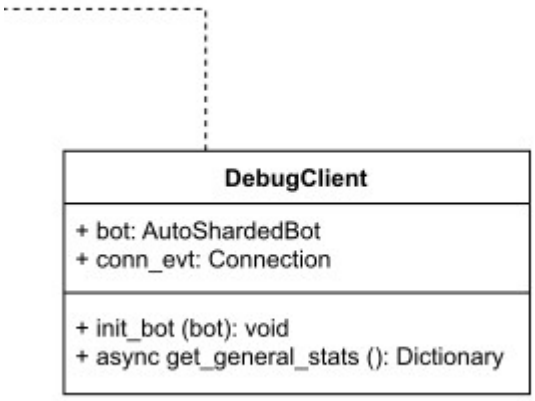
Eine Schleife durchläuft einmal pro Minute diese Schritte:

1. Rufe alle Events aus der Datenbank ab (TODO nur laufende Events abrufen)
2. Iteriere durch diese Events
    - Ist der Zeitstempel des Events bereits abgelaufen?
        - Ja: Schreibe alle Teilnehmer an, sende eine Nachricht in den Kanal und lösche das Event
        - Nein: Zähle weiter

Ein abgelaufenes Event wird aus der Datenbank entfernt. Tritt ein Fehler auf (z.B. der Kanal wurde gelöscht), wird das Event direkt entfernt und es wird weitergezählt.

3.2.3.4 Debug Client

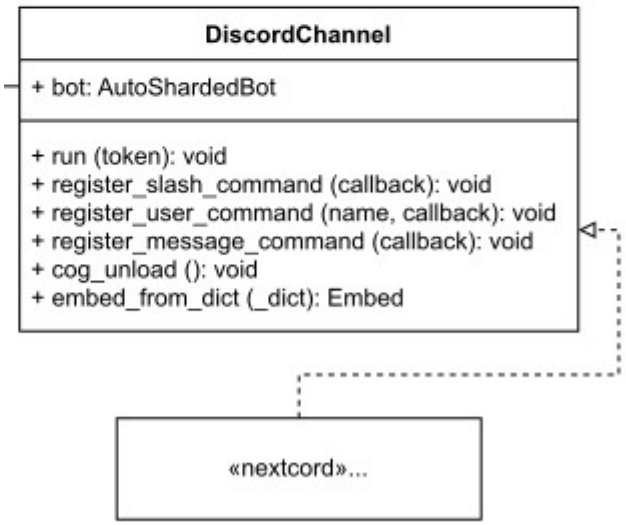
Der Debug-Client ist ein Tool, das nur während der Entwicklung implementiert ist, aber entfernt wird, sobald der Bot fertig ist. Mit dem Client lassen sich Daten vom Bot (Wie z.B. Laufzeit, Fehlerlogs, etc) abrufen.



Der Client wird initialisiert mit einem Bot und einer Datenbankverbindung zur Event-Tabelle.

Die asynchrone Funktion get_general_stats gibt Statistiken zurück, wie Laufzeit, Ping, Fehlerlogs und offene Events in der Datenbank.

3.2.4 DiscordChannel Klasse



Die DiscordChannel-Klasse führt die Initialisierung der Schnittstelle zwischen der Discord API (= nextcord) und unserem Programm durch. Dafür legt sie ein Objekt von „nextcord.AutoShardedBot“ an, welches die Grundlage für die gesamte Kommunikation mit Discord bildet.

```
6   class DiscordChannel(commands.Cog):
7       bot = None
8
9       def __init__(self) -> None:
10          if DiscordChannel.bot is None:
11              intents = Intents.default()
12              intents.members = True
13              DiscordChannel.bot = commands.AutoShardedBot(command_prefix="j!", intents=intents)
14              @self.bot.event
15              async def on_ready():
16                  print(f'{self.bot.user} has logged in.')
17
18              try:
19                  self.bot.add_cog(self)
20              except Exception as e:
21                  print(f"An error has occured when trying to add {self} as Cog: {e}")
22
23      def run(self, token):
24          self.bot.run(token)
```

Außerdem werden die Berechtigungen für den Bot (in Discord) gesetzt wie z.B. der Zugriff auf Daten von Mitgliedern aus dem Server (siehe Zeile 11-13).

Des weiteren besitzt die DiscordChannel-Klasse eine run Funktion welche nach der Initialisierung aller Programmteile von der Cordic-Klasse aufgerufen wird. Diese Funktion nimmt den Token für die Discord API (vorher in der „load_tokens“ Funktion in Cordic aus Datei geladen) und sorgt dafür, dass der Bot auf Discord hochfährt bzw. ansprechbar ist.

Die Klasse stellt außerdem einige Funktionen bereit, die helfen neue Befehle aus den abgeleiteten Klassen zu erstellen:

```
26      def register_slash_command(self, callback):
27          # bot.slash_command erstellt einen Dekorator um unsere Funktion (unsere Funktion = callback)
28          # Dieser Dekorator erweitert unsere Funktion um Nextcord Slash Cmd Logik
29          decorator = self.bot.slash_command(guild_ids=[908740156306112512])
30          # Wir müssen den Dekorator auch mit unserer Funktion (=callback) aufrufen, damit intern in
31          # Nextcord eine Instanz von 'ApplicationCommand' erstellt wird, welche sich als Attribut
32          # unser Callback merkt. Wenn dieser ApplicationCommand von Nextcord aufgerufen wird,
33          # wird unsere Funktion mit den Args (welche ApplicationCommand mitgegeben wurden) aufgerufen
34          application_command = decorator(callback)
35
36      def register_user_command(self, name: str, callback):
37          self.bot.user_command(guild_ids=[908740156306112512])(callback)
38
39      def register_message_command(self, callback, **kwds):
40          self.bot.message_command(**kwds)(callback)
```

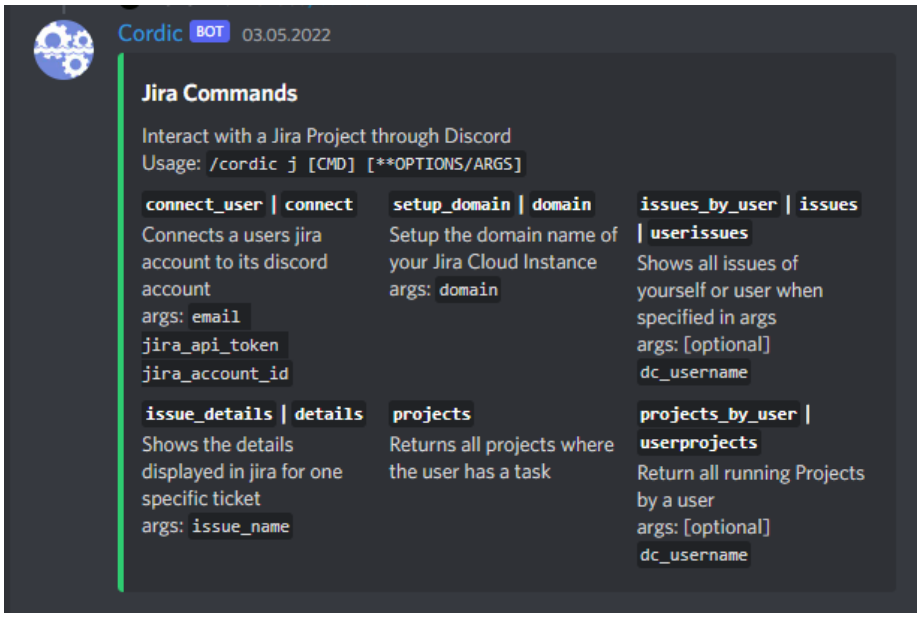
Dafür wird die eingebaute Funktionalität von nextcord genutzt, welche es erlaubt sog. „Slash Commands“ zu erstellen. Slash Commands ist eine von Discord angebotene Methode zur Kommunikation zwischen Nutzer und Bot. Eine Slash Command besteht aus einem Schrägstrich (welcher in einen Textkanal eingegeben wird) und zusätzlichen Argumenten. Der Vorteil von Slash Commands ist eine Autovervollständigung bei der Eingabe eines Befehls. Dies macht die Interaktion mit unserem Bot sehr viel übersichtlicher.

Die DiscordChannel stellt allerdings auch eine Funktion bereit um die Rückgabe des CommandInterpreters (Dictionary) in eine für Discord formatierte Ausgabeform zu bringen:

```
45      def embed_from_dict(self, _dict):
46          e = Embed()
47          e.title = _dict.get("title")
48          e.description = _dict.get("description")
49          e.colour = _dict.get("color") or 0x36393F
50          if _dict.get("fields") is not None:
51              for key in _dict.get("fields", []):
52                  if isinstance(_dict["fields"][key], list):
53                      delimiter = "\n"
54                      part = delimiter.join(_dict["fields"][key])
55                      e.add_field(name=key, value=part)
56                  else:
57                      e.add_field(name=key, value=_dict["fields"][key])
58          return e
59
```

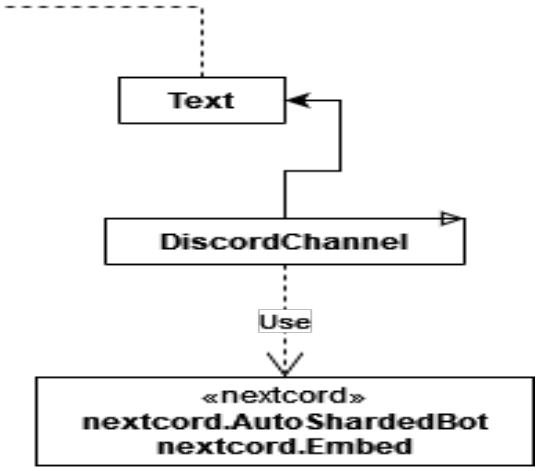
Dazu werden die von Discord bereitgestellten Embeds genutzt. Diese erlauben es die Ausgabe in besonders hervorgehobenen Blöcken in einem Chat Kanal zu senden. Außerdem kann man so eine

Farbe festlegen, Felder, Links und Überschriften einfügen. Ein Beispiel für ein Embed (hier der Hilfe-Befehl von Jira):

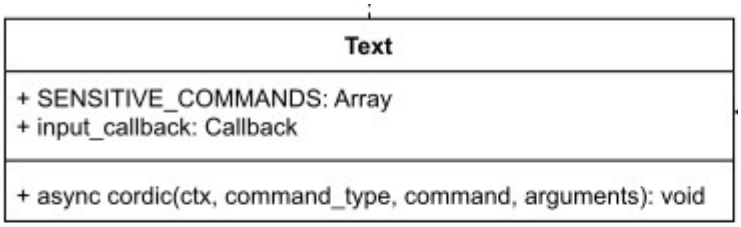


3.2.5 Text-Klasse

Die Text-Klasse ist von der DiscordChannel Klasse abgeleitet (Ausschnit; gesamte Abbildung siehe Aufbau des Programms):



Hier sind die Variablen und die Funktion der Text-Klasse noch einmal genauer dargestellt:



Die Text Klasse definiert mithilfe der oben erwähnten Slash Commands unseren Haupt-Befehl, „cordic“ genannt. Dies erlaubt es dem Nutzer alle Funktionen unseres Discord-Bots mithilfe des Slash Commands „/cordic […]“ anzusteuern. Die Argumente für diesen Slash Command sind: der Typ des Befehls (aka. Welcher Command Client angesteuert werden soll), der Command (/ Name) und zusätzliche Argumente für den Befehl.

```
29      async def cordic(self,
30          ctx,
31          command_type : str = SlashOption(name="command_type",
32                                           description="Wether command is a jira or event command",
33                                           required=True,
34                                           choices={
35                                               "Cordic general commands": "general",
36                                               "jira/ j": "jira",
37                                               "event/ e": "event",
38                                               "debug /dbg":"debug"
39                                           }
40                                           ),
41          command : str = SlashOption(name="command",
42                                      description="Command to run",
43                                      required=True,
44                                      choices={ ···
62                                      ),
63          arguments : str = SlashOption(name="args",
64                                        description="Look into the command types help list for a list of arguments",
65                                        required=False)):
```

Diese Daten werden alle in einem von nextcord bereitgestellten Objekt names „SlashOption“ verpackt. Dies hilft nextcord bei der Verarbeitung unseres Befehls. Wenn man den Wert „required“ bei einem SlashOption Objekt auf „False“ setzt ist dieses Argument nicht mehr verpflichtend auszufüllen bei der Eingabe des Befehls in Discord.

Nextcord wird jetzt automatisch unsere Funktion mit den definierten Argumenten (= den Eingabedaten) unserer Funktion aufrufen, sobald der Nutzer in Discord den Befehl abschickt.

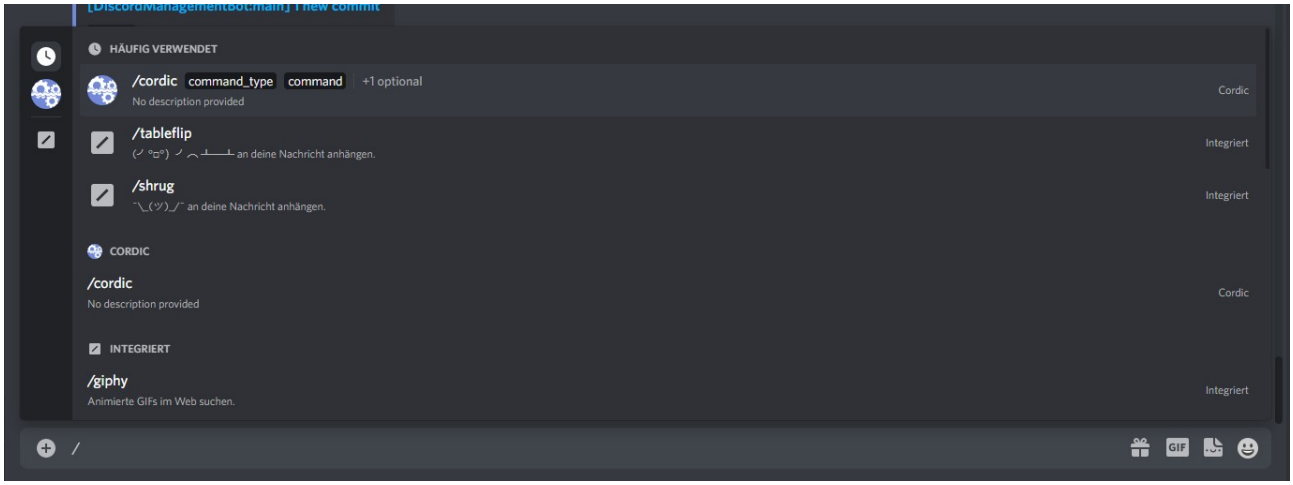
Innerhalb der „cordic“ Funktion wird diese Eingabe dann durch die Funktion „input_callback“ der Text-Klasse (welche ein Callback der Funktion „input_callback“ aus der Cordic-Klasse ist) an den Command Interpreter weitergeleitet, welcher dann diese mithilfe der Command Clients verarbeitet. Die Rückgabe wird in der Variable „output_data“ gespeichert. Als nächstes wird die Rückgabe (welche immer ein Dictionary ist) durch die „embed_from_dict“ Funktion der DiscordChannel-Klasse in ein Embed-Objekt umgewandelt, welches dann durch nextcord an Discord geschickt wird.

```
67      output_data = await self.input_callback(command_type, command, arguments, ctx.user, ctx.channel, ctx.guild,self.bot)
68
69      # Convert to json object for easier access
70      if isinstance(output_data,str):
71          output_data = json.loads(output_data)
72
73      await ctx.send(embed=self.embed_from_dict(output_data), ephemeral=command in self.SENSITIVE_COMMANDS)
```

Der „cordic“ befehl wird dann mithilfe der „register_slash_command“ Funktion aus der DiscordChannel-Klasse für Discord registriert, sodass Discord weiß, dass es diesen Befehl gibt und es ihn in der Liste der Möglichen Befehle anzeigen kann.

```
19          # Fügt automatisch noch argumente wie z.b. guild_id=[...] hinzu
20          self.register_slash_command(self.cordic)
```

Nach der erfolgreichen Registrierung in der Discord API sieht das dann so aus:



Man kann erkennen, dass Discord beim Eingeben eines Schrägstriches nun unseren Befehl als mögliche Auswahl anzeigt. Zusätzlich kann man sehen, dass Discord direkt die Argumente für unseren Befehl anzeigt, da diese ja in die SlashOption Objekte geschrieben wurden und von nextcord an Discord geschickt wurden.

4 Aufgabenverteilung

4.1 Nick

Nick hat am Anfang der Jahresarbeit an der Spracheingabe gearbeitet. Nachdem diese verworfen wurde (siehe Diskussion) hat er die Projektstruktur/ die Gliederung des Projektes in die verschiedenen Klassen umgesetzt. Dabei hat er Platzhalterklassen für die Command Clients angelegt, damit Florian und Ahmad direkt Wissen an welcher Stelle im Programm sie arbeiten können. Außerdem hat er den Command Interpreter, die DiscordChannel- und die Textklasse realisiert.

4.2 Florian

Florian hat sich hauptsächlich mit den Klassen Jira Client und Jira User beschäftigt, sowie der dazugehörigen Datenbank für alle Vorgänge in den beiden Klassen. Dazu hat er die Strukturen für die Embeds (zuerst Dictionary, dann Weiterverarbeitung in ein Embed) realisiert damit, die Ausgaben besser visualisiert werden können. Dafür hat er auch beim Command Interpreter und in der DiscordChannel Klasse mitgewirkt. Beim Command Interpreter hat er die Verwendung von Jira Commands übernommen und die Regeln darin definiert.

4.3 Ahmad

Ahmad hat das dritte Kernmodul des Bots erstellt: Die Terminverwaltung. Damit lassen sich schnell Meetings planen, ansehen und löschen. Außerdem kann man sich zu Terminen eintragen, um eine Direktnachricht zu erhalten, wenn das Event losgeht. Im CommandInterpreter hat er die Terminverwaltungsbefehle eingebaut.

5 Diskussion

5.1 Problem Spracheingabe

Nick hat sich zu Projektbeginn intensiv mit einem Spracheingabe-Modul und dessen Möglichkeiten auseinandergesetzt. Dazu wurde ein experimenteller Fork des offiziellen nextcord-Repositories verwendet, welcher allerdings nicht sehr stabil war, von Änderungen des Autors abhängig war und noch nicht in den main/ master Branch des nextcord Repositories integriert war. Der größte Nachteil dieses Forks war, das er etwas veraltet war und in der Zwischenzeit das nextcord Repository nativ Slash Commands unterstützt hat. Als wir also den experiementellen Fork verwendet hatten, mussten wir zusätzlich ein Modul nutzen, welches Slash Commands bereitgestellt war, aber nicht offiziell von nextcord kam. Dieses Modul war zusätzlich noch eingeschränkt in der Funktionalität und unterstützte keine- Autovervollständigung. Alles Dinge, die die Implementation von Slash Commands im offiziellen Repository bereitgestellt hat.

Aus diesem Grund standen wir vor der Entscheidung zwischen Autovervollständigung und Spracheingabe und haben uns nach umfangreichen Analysen und durch Rücksprache mit Communardo Software GmbH entschieden, dass die Integration einer Spracheingabe generell möglich ist jedoch nicht im zeitlichen Rahmen des Projektes liegt, sowie nicht in dem Maße zum Ziel des Projektes beigetragen hätte wie Autovervollständigung. Deswegen wird in der finalen Version auf dieses ohnehin als optional definierte Feature verzichtet. Die Ergebnisse stehen jedoch auch weiterhin in einem separaten Branch zur Verfügung und können generell für zukünftige Aktivitäten genutzt werden bzw. als Grundlage dienen. Dies wird auch von der aktuellen Projektstruktur gefördert, welche es sehr einfach erlaubt, neue Input bzw. Output Methoden zu definieren (als abgeleitete Klasse von DiscordChannel zum Beispiel).

5.2 Mögliche Erweiterungen

- Erinnerungsfunktion für das Erledigen von Issues/Aufgaben
- Jira’s sqlitedatabase.py auf SQLAlchemy vereinheitlichen
- Performance-Optimierung bei Datenbankabfragen des Termin-Clients