ChatGPT

Best Practices für Discord.py Cogs im FUR-System-Kontext

Strukturierte Cog-Definition und Setup-Funktion

Ein gut strukturierter Discord.py-Cog kapselt zusammengehörige Befehle, Events und Tasks in einer Klasse. Jeder Cog sollte von `commands.Cog` erben und eine **Async-Setup-Funktion** bereitstellen, um den Cog zur Bot-Instanz hinzuzufügen [1]. Beispiel:

```
class ExampleCog(commands.Cog):
    def __init__(self, bot):
        self.bot = bot
    # ... Befehle und Events hier definieren ...

async def setup(bot):
    await bot.add_cog(ExampleCog(bot))
```

Dieses Muster ermöglicht es, den Cog als **Extension** per `bot.load_extension()` zu laden. Damit der Bot alle Cogs initialisiert, können im Hauptprogramm alle Cog-Module gelistet und via `load_extension` eingebunden werden. Wichtig ist, dass die Setup-Funktion asynchron ist (neu in discord.py v2.0) und den Cog per `await bot.add_cog(...)` hinzufügt [1]. So bleibt die Cog-Registrierung übersichtlich und modular.

Logging beim Laden von Cogs

Es ist empfehlenswert, beim Laden jedes Cogs einen Log-Eintrag zu schreiben, um im Betrieb nachvollziehen zu können, welche Module erfolgreich geladen wurden. In der Praxis wird dazu oft der Logger im Bot-Hauptprogramm genutzt, direkt nach `load_extension()`. Beispielsweise loggt man auf INFO- oder DEBUG-Level: *„Cog geladen: <Name>“*. Einige Projekte überschreiben sogar die `Bot.add_cog`-Methode, um beim Hinzufügen automatisch zu loggen [2]:

```
def add_cog(self, cog: commands.Cog, *, override: bool = False):
    super().add_cog(cog, override=override)
    logger.debug(f"Cog loaded: {cog.qualified_name}")
```

Im Kontext des FUR-Systems sehen wir bereits ähnliches Logging (z. B. `INFO:bot.bot_main: Cog geladen: ...`). Diese Logs helfen beim Debugging und bestätigen den erfolgreichen Start aller Module.

Reminder-Management und automatisierte Erinnerungen (Autopilot)

Automatisierte Reminder für Events oder Termine sollten idealerweise in Hintergrund-Tasks laufen, um Benutzer rechtzeitig zu benachrichtigen. Best Practices hierfür sind:

- **Tasks Loop verwenden:** Mit `discord.ext.tasks.loop` lässt sich ein wiederkehrender Task implementieren. Ein Beispiel (Pycord Guide) zeigt einen Loop, der regelmäßig eine teure Leaderboard-Berechnung durchführt [3]. Ähnlich kann ein Loop jede Minute prüfen, ob in Kürze ein Event ansteht, und dann Erinnerungen versenden. Wichtig: Vor dem Start des Loops einmal `await bot.wait_until_ready()` ausführen (z. B. in einer `before_loop`-Funktion) [4], damit Tasks erst nach Bot-Login starten. Im FUR-Reminder-Cog wird dies bereits umgesetzt (der Task wartet bis der Bot bereit ist, bevor er die DB abfragt und Nachrichten sendet).

- **Zeitscheiben und Toleranzen:** Definiere klare Intervalle (z. B. 24h, 1h, 30min, 5min vorher) und prüfe innerhalb des Loops, ob ein Event in dieses Zeitfenster fällt. Die FUR-Implementierung nutzt z.B. eine Toleranz von ±1 Minute, um genau zum richtigen Zeitpunkt zu senden. Bereits gesendete Erinnerungen sollten markiert werden (z. B. in einer Datenbank oder In-Memory-Set), damit sie nicht doppelt gesendet werden [5] [6].

- **Persistente Speicherung:** Verwende eine Datenbank oder Dateispeicher, um Events und bereits verschickte Reminder zu verwalten. So überleben geplante Erinnerungen auch einen Bot-Neustart. In Best-Practice-Beispielen werden Events z.B. aus einer DB abgefragt und versendete Reminder in einer Collection/DB festgehalten [7] [8]. Das FUR-System nutzt hier MongoDB (`events`, `event_participants`, `reminders_sent`) bzw. SQLite – beide Ansätze sind gängig, wichtig ist konsistente Zeitbasis (UTC speichern) und Prüfungen auf Formatfehler [9] [10].

- **Fehler- und Ausnahmebehandlung:** Beim Versand von Erinnerungen per DM oder Channel sollte der Code Exceptions gezielt abfangen. Insbesondere `discord.Forbidden` (wenn z.B. der Bot keine DM senden darf) sollte behandelt werden, um nicht jedes Mal den Task abzubrechen [11]. Im FUR-Code wird das vorbildlich gelöst: Fehlende Berechtigungen oder nicht auffindbare Nutzer werden per `log.warning` protokolliert, ohne den ganzen Loop zu stoppen [11]. Unerwartete Fehler im Loop selbst fängt man ab (try/except um DB-Zugriff und Sende-Logik) und loggt sie mit Stacktrace [12] [13], anstatt den Task abstürzen zu lassen.

Zusätzlich existiert oft ein **manueller Trigger** für Admins, um den Reminder-Check sofort auszuführen (z.B. ein `/reminder_autopilot_now` Slash-Command, wie in FUR implementiert) [14]. Dieser sollte durch einen Admin-Check geschützt werden (siehe nächster Abschnitt).

Opt-Out Verwaltung für Benachrichtigungen

Benutzern die Kontrolle zu geben, ob sie Erinnerungen oder Newsletter erhalten wollen, ist essenziell. **Opt-Out-Management** bedeutet: wir halten pro User fest, ob er bestimmte automatisierte DMs empfangen möchte oder nicht. Best Practices:

- **Persistente Einstellung speichern:** Eine einfache Möglichkeit ist eine Datenbank-Tabelle oder -Collection (`user_settings`) mit Feldern wie `allow_dm` oder spezifischen Opt-Out-Flags. Der

Reminder- oder Newsletter-Cog sollte diese vor dem Senden abfragen. So wird vermieden, dass ein Nutzer gegen seinen Willen massenhaft DMs bekommt.

- **Checks vor dem Senden:** Bevor der Bot eine DM verschickt, sollte er prüfen, ob der Empfänger zugestimmt hat. Im Newsletter-Cog des FUR-Systems wird z.B. zunächst eine Liste aller Mitglieder und ihrer `allow_dm`-Einstellungen geladen, bevor in einer Schleife DMs verschickt werden [15] [16]. Nutzer mit Opt-Out werden einfach übersprungen.

- **Opt-In statt Opt-Out:** Beachte, dass das ungefragte Versenden von Massen-DMs ohne Einwilligung der Nutzer gegen die Discord-Richtlinien verstößt [17]. Es ist daher **besser**, ein Opt-In-System zu haben (Nutzer abonnieren aktiv den Newsletter), oder zumindest ein Opt-Out leicht zugänglich zu machen (Befehl wie `!newsletter optout`). Das FUR-System hält sich daran, indem es nur Mitgliedern Nachrichten schickt, die nicht auf Opt-Out stehen. Eine kurze Notiz im Code oder der Doku sollte erklären, wie Nutzer sich an- oder abmelden können.

- **Feedback und Transparenz:** Implementiere Befehle wie `!mysettings` oder spezifisch `!newsletter subscribe/unsubscribe`, die dem Nutzer Rückmeldung geben („Du hast den Newsletter abonniert/abbestellt."). Dies ist zwar kein Muss, aber verbessert die Usability und stellt sicher, dass Nutzer die Kontrolle wahrnehmen können.

Leaderboard-Interaktionen

Leaderboards (Ranglisten) erfordern oft, kontinuierlich Daten (z.B. Punktestände, XP, Beiträge) zu sammeln und anzuzeigen. Hier einige Best Practices:

- **Datenmodell und Aktualisierung:** Erfasse die relevanten Metriken (z.B. Nachrichtenanzahl, Punkte) persistent. Ein eventbasiertes Sammeln (z.B. im `on_message` Event die XP erhöhen) ist gängig. Der Speichermedium kann eine Datenbank oder ein JSON File sein – wichtig ist, dass für jedes Guild-Mitglied Werte vorliegen [18] [19]. In einem **on_message-Listener** den Score zu erhöhen und anschließend `await bot.process_commands(message)` aufzurufen, ist ein verbreitetes Muster, um sowohl Befehle als auch Zähler zu verarbeiten [20] [21].

- **Hintergrund-Task für teure Berechnungen:** Falls die Berechnung der Top-Liste sehr aufwendig ist (z.B. viele Nutzer, komplexe Sortierung), empfiehlt sich ein Task, der die Rangliste periodisch aktualisiert. Das Pycord-Guide-Beispiel nutzt einen 10-Minuten-Loop, um ein Leaderboard-Embed vorzuberechnen [3]. Der aktuelle Stand wird dann zwischengespeichert (`self.leaderboard_embed`) und bei Abruf direkt verwendet, statt jedes Mal die Datenbank zu durchforsten [22] [3]. Dieses **Caching** verbessert die Performance deutlich.

- **Anzeige mit Embeds oder Paginierung:** Für die Präsentation der Top-N Ergebnisse sind Discord **Embeds** ideal, da sie formatierten Text in Feldern darstellen können. Ein Beispielsortierung und Embed-Bau für eine Top-10-Liste aus einer JSON-Datei zeigt, wie man die Daten nach Score sortiert und in ein Embed-Feld pro Nutzer einfügt [23]. Ggf. kann eine Rangliste viele Einträge umfassen – in dem Fall ist eine Seiteneinteilung sinnvoll (Pagination mit Buttons oder durch mehrere Commands wie `!leaderboard page2`). Für den Anfang reicht aber oft die Top 5 oder Top 10 in einem einzelnen Embed.

- **Interaktion und Aktualisierung:** Biete Befehle an, um das Leaderboard abzurufen (`/leaderboard` Slash-Command oder `!leaderboard`). Wenn Daten live aktualisiert werden sollen, kann man auch Reaktionen/Buttons nutzen, um das Embed upzudaten. Bei

Befehlsausführung sollte der Bot ggf. auf neue Daten aktualisieren (sofern kein Hintergrund-Task alle X Minuten läuft).

- **Reset und Zeiträume:** Überlege, ob Leaderboards nach einem Zeitraum zurückgesetzt werden (monatliche Rankings, etc.). Das FUR-System hat z.B. einen `monthly_champion_job.py`, was auf eine monatliche Auswertung hindeutet. Solche Jobs können via Scheduled Task (z.B. mit APScheduler oder einem Cron über Celery) ausgeführt werden, oder man nutzt Discord.py-Tasks mit langen Intervallen/Schlafphasen.

Newsletter-System und DM-Broadcasting

Ein wöchentlicher Newsletter per Discord-DM erfordert sorgfältige Planung, um **Spam zu vermeiden** und die Auslieferung technisch zu meistern. Best Practices hierfür:

- **Geplante Versendezeit mittels tasks.loop:** Ähnlich wie bei Remindern kann ein `@tasks.loop` verwendet werden, um z.B. stündlich zu prüfen, ob der Versandzeitpunkt erreicht ist. Im FUR-Newsletter-Cog wird der Loop jede Stunde geweckt und sendet nur sonntags um 12:00 UTC die Nachrichten [24] [25]. Dieses Muster – häufig prüfen, aber nur bei Erfüllung einer Bedingung agieren – stellt sicher, dass Timing eingehalten wird ohne auf komplexe Scheduler zurückgreifen zu müssen.

- **Inhalt vorbereiten:** Bevor DMs verschickt werden, sollte die Newsletter-Nachricht (z.B. Wochenübersicht der Events) als String oder Embed fertig gebaut sein. Im FUR-Beispiel werden zunächst alle anstehenden Events der nächsten Woche aus der DB geholt. Dann wird eine Nachricht mit Überschrift, Begrüßung und Auflistung der Events erstellt [26] [27]. Durch das Sammeln aller Daten vorab muss die Datenbank nur einmal kontaktiert werden, was effizienter ist.

- **Massenversand mit Rate Limiting beachten:** Discord setzt Limits für Nachrichten. Ein Bot kann pro Sekunde nur eine bestimmte Anzahl DMs senden, bevor er ins Ratenlimit läuft. Daher sollte ein Massenversand an alle Guild-Mitglieder in **Batches** oder mit kleinen Pausen erfolgen. Man kann z.B. nach X versendeten DMs ein `await asyncio.sleep(1)` einfügen. Alternativ sendet man z.B. 20 DMs pro Minute. In der FUR-Implementierung ist nicht explizit Sleep zu sehen – möglicherweise reicht die verarbeitungsbedingte Verzögerung pro Nachricht aus, oder die Mitgliederzahl ist überschaubar. Bei sehr großen Servern (>1000 Nutzer) sollte man aber aktiv drosseln.

- **Fehlerbehandlung und Zähler:** Es ist wichtig, am Ende einen Überblick zu haben, wie viele DMs erfolgreich gesendet wurden und wo es Probleme gab. Führe Buch über Erfolge und Fehlschläge: Initialisiere Zähler für `sent_count`, `blocked_count` (DM deaktiviert) und `error_count`. Jede erfolgreich gesendete DM erhöht `sent_count`. Fängt man `discord.Forbidden` ab, erhöht `blocked_count` (Benutzer hat DMs aus oder Bot blockiert) und fährt mit dem nächsten fort. Andere Exceptions erhöhen `error_count`. Nach dem Versand loop kann man dann ins Log schreiben, wie viele verschickt wurden und wie viele nicht [28]. So hat man Transparenz über den Newsletter-Versand.

- **Opt-Out respektieren:** Wie schon erwähnt, unbedingt die Nutzer ausschließen, die keinen Newsletter wollen. Im FUR-System wird vermutlich `allow_dm` in `user_settings` berücksichtigt – der Code lädt alle DM-Einstellungen vorab (`dm_prefs`) [15] und prüft dann pro

Nutzer, ob gesendet werden soll. Ein sauberes Opt-Out-System stellt sicher, dass der Bot nicht als Spam wahrgenommen wird.

- **Bestätigung und Abmeldung:** Erwäge, nach Versand an jeden Nutzer ggf. eine **einfache Abmeldemöglichkeit** mitzusenden (z.B. „Antworte mit `STOP`, um keine weiteren Newsletter zu erhalten.“). Alternativ in jeder Newsletter-Nachricht am Ende darauf hinweisen, wie man sich abmelden kann (z.B. via Befehl). Das erhöht die Zufriedenheit und Compliance.

Hintergrund-Tasks, Fehlerbehandlung und Berechtigungs-Checks

Über alle genannten Cogs hinweg gelten einige allgemeine Best Practices:

- **tasks.loop und cog_unload:** Wenn ein Cog periodische Tasks startet (wie `tasks.loop`), sollte im Cog auch `cog_unload` definiert werden, um diese Loops beim Entladen zu stoppen [29] [30]. FUR-Cogs setzen dies konsequent um (Loop in `__init__` starten, in `cog_unload` via `loop.cancel()` stoppen). So werden Ressourcen frei und bei einem Reload des Cogs doppelte Tasks vermieden.

- **Error Handler pro Cog oder global:** Discord.py erlaubt es, Fehler von Commands abzufangen, entweder global über Ereignisse (`on_command_error` oder neuerdings `on_application_command_error` für Slash-Commands) oder individuell pro Cog/Befehl. Eine **globale Fehlerbehandlung** kann häufige Fälle wie fehlende Berechtigungen, ungültige Eingaben, Cooldowns etc. abfragen und dem Nutzer eine freundliche Meldung senden, statt den Traceback im Chat zu posten [31]. Im Hackster-Bot Beispiel oben werden z.B. `MissingRequiredArgument` oder `MissingPermissions` abgefangen und dem Nutzer mitgeteilt [32]. Implementiere ähnliches für dein Bot-System: etwa ein Cog `error_handle` oder direkt in jedem Cog eine Methode `@commands.Cog.listener("on_command_error")` die Fehler für Befehle dieses Cogs behandelt. Dadurch kann man gezielt reagieren, z.B. bei einem `/leaderboard`-Befehl einen Datenbankfehler elegant abfangen und „⚠ Ranking momentan nicht verfügbar“ ausgeben, statt nichts zu tun oder einen Fehler zu zeigen.

- **Checks und Berechtigungen:** Nutze die Decorators aus discord.py, um Befehle nur bestimmten Nutzern zugänglich zu machen. Beispielsweise kann ein Opt-Out-Setzen für alle ggf. auf Admins beschränkt sein. Im Code kann man `@commands.has_guild_permissions(administrator=True)` verwenden, um nur Admins Zugriff zu geben, oder innerhalb des Commands prüfen wie im FUR-Remindercog: `if not interaction.user.guild_permissions.administrator: ...` [33]. Diese Checks stellen sicher, dass normale User keine administrativen Funktionen ausführen. Ebenso können **Umgebungs-Checks** sinnvoll sein – z.B. ein `@commands.check` Decorator, der sicherstellt, dass ein Befehl nur in der Produktionsumgebung läuft (falls man denselben Code auch in einer Testumgebung einsetzt). Das FUR-System trennt ja echten Bot und Stub anhand von `ENABLE_DISCORD_BOT`. So könnte man z.B. gewisse Cogs im Stub-Modus gar nicht laden oder innerhalb der Funktionen abfragen `if not USE_DISCORD_BOT: return` um keine echten Aktionen durchzuführen.

- **Umgebungsabhängiges Verhalten:** Für Entwicklungszwecke ist es hilfreich, wenn der Bot erkennt, ob er in **Development** oder **Production** läuft (etwa via ENV-Var). Entsprechend kann Logging-Level angepasst werden (Debug vs. Info), und vor allem kann man verhindern, dass in Dev versehentlich echte DMs rausgehen oder produktive Aktionen passieren. Best Practice ist hier, wichtige Aktionen an die ENV zu knüpfen. Beispiel: Im Dev-Modus könnte der Newsletter-

Cog keinen echten Versand machen, sondern die Nachricht nur in der Konsole loggen oder an einen Test-Channel schicken. Solche Bedingungen (z.B. `if Config.FLASK_ENV != 'production': return`) vermeiden peinliche Situationen beim Testen.

Zusammengefasst: **verwende die Checks, Fehlerbehandlungs- und Logging-Möglichkeiten**, die discord.py bietet, um deinen Bot robust und sicher zu machen. Ein gut strukturiertes Logging (Info für Normalbetrieb, Debug für Diagnosen, Warn/Error für Probleme) in allen Cogs erleichtert später die Wartung erheblich.

Internationalisierung (i18n) und dynamische Command-Beschreibungen

Für ein mehrsprachiges Projekt wie FUR-System (es gibt Anzeichen von `fur_lang.i18n`) sind Übersetzungen und dynamische Texte ein wichtiger Aspekt:

- **Inhaltliche Übersetzungen:** Bot-Nachrichten (z.B. Reminder-Texte, Newsletter-Inhalt) sollten abhängig von der bevorzugten Sprache des Nutzers erstellt werden. Im FUR-Reminder-Autopilot wird z.B. pro User die Sprache aus der DB geladen (`lang = await self.get_user_language(user_id)`) und dann ein lokalisiertes Template via `t(...)` eingefügt [34] [35]. Diese Funktion `t(key, lang=lang)` greift vermutlich auf eine JSON oder PO-File Übersetzung zurück. Best Practice ist hier, alle nutzerseitigen Strings in eine Übersetzungsdatei auszulagern und mit Schlüsseln zu referenzieren. So kann man leicht neue Sprachen ergänzen oder Formulierungen ändern, ohne im Code wühlen zu müssen.

- **Command-Namen und -Beschreibungen lokalisieren:** Discord hat inzwischen native Unterstützung für **Slash-Command-Lokalisierung**. In discord.py v2 kann man dafür einen `Translator` setzen und `locale_str` benutzen. Ein StackOverflow-Beispiel zeigt, wie ein benutzerdefinierter Translator die Namen/Beschreibungen je nach Discord-Clientsprache ausliefert [36] [37]. Im Code würde man z.B.:

```
bot.tree.set_translator(MyTranslator())

@bot.tree.command(
    name=locale_str("addcolorrole"),
    description=locale_str("Creates or updates a color role")
)
async def addcolorrole(interaction, color: str):
    ...
```

```
class MyTranslator(app_commands.Translator):
    async def translate(self, string: app_commands.locale_str, locale,
context):
        translations = {
           "Creates or updates a color role": { "de": "Erstellt oder
aktualisiert eine Rollenfarbe" },
           # ... weitere Übersetzungen ...
        }
```

```
        return translations.get(string.message, {}).get(locale.value,
string.message)
```

Auf diese Weise würde Discord z.B. einem deutschen Nutzer die Beschreibung auf Deutsch anzeigen (wenn gepflegt). **Dynamische Befehlsbeschreibungen** lassen sich so ebenfalls realisieren – etwa könnte man die Beschreibung zur Laufzeit zusammenbauen, aber in der Regel sind sie statisch oder nur von der Sprache abhängig. Wenn wirklich kontextabhängig (z.B. ein Command, der je nach Server etwas anderes tut), kann man den Description-String auch zur Laufzeit setzen bevor man den Command registriert.

- **i18n-Bibliotheken:** Alternativ zum manuellen Ansatz kann man Bibliotheken wie `discord.py-i18n` [38] oder `discord-ext-i18n` nutzen, die einige Boilerplate abnehmen. Diese nutzen meist die oben genannte Mechanik intern. Wichtig ist, die Übersetzungen synchron mit der Codebasis zu halten.

- **Platzhalter und Formatierung:** Achte darauf, bei übersetzten Strings Platzhalter zu verwenden statt String-Konkatenation, damit Satzbau korrekt bleibt. Zum Beispiel: `t("reminder_event_5min", title=event_title, lang=lang)` sollte in der Sprachdatei einen Platzhalter `{title}` haben, den die Funktion einsetzt. So verhindert man, dass manuell zusammengesetzte Strings in verschiedenen Sprachen grammatikalisch falsch sind.

- **Fallbacks:** Definiere eine Default-Sprache (im Config z.B. `DEFAULT_LANGUAGE = 'de'` oder `'en'`), falls ein Nutzer keine Sprache gesetzt hat. Das FUR-System scheint Deutsch als Default zu nutzen, aber unterstützt mehrere (`SUPPORTED_LANGUAGES` in Config). Stelle sicher, dass für jede dieser Sprachen die wichtigsten Texte übersetzt sind – zumindest die Benutzeroberfläche (Befehle, Bestätigungen, Fehlermeldungen).

- **Dynamische Inhalte in Befehlslisten:** Wenn die Frage nach „dynamischen Befehlsbeschreibungen“ auf Dinge wie Prefix oder kontextabhängige Infos abzielt: Bei Textcommands könnte man den Helfer (help command) anpassen, um z.B. den aktuell gesetzten Prefix anzuzeigen. Bei Slash-Commands ist das seltener nötig, da sie fest im UI stehen. Ein Trick für dyn. Descriptions könnte sein: Beschreibung kurz halten, Details in die Ephemeral-Response des Commands packen, wo man dynamisch alles anpassen kann.

Zusammengefasst sollte das FUR-System seine bestehende i18n-Integration (das `fur_lang.i18n` Modul) weiter nutzen und ggf. für Slash-Commands die neuen Mechanismen anwenden, um Nutzern konsistent **mehrsprachige Interaktionen** zu bieten.

Fazit: Durch Vergleich mit bewährten Open-Source-Implementierungen lässt sich das FUR-System weiter optimieren. Die Cogs sollten klar getrennte Verantwortlichkeiten haben, beim Laden geloggt werden und durch Tasks effizient arbeiten. Fehlerbehandlung und Logging tragen zur Stabilität bei, und Nutzer-Einstellungen (Opt-Out/I18n) sorgen für positive User Experience. Mit diesen Best Practices – inspiriert von offiziellen Beispielen und Community-Projekten – kann die aktuelle Implementierung ausgerichtet und verbessert werden, ohne grundlegende Funktionalität zu ändern, aber in Wartbarkeit und Compliance zu gewinnen.

Quellen: Best-Practice-Codeausschnitte und Erläuterungen stammen aus der Discord.py-Dokumentation und Community-Beispielen [1] [2] [17] [3] [36] [39], die als Referenz für die oben genannten Empfehlungen dienten.

---

[1] python - How to use extensions/cogs in discord.py - Stack Overflow
https://stackoverflow.com/questions/78165811/how-to-use-extensions-cogs-in-discord-py

[2] [31] [32] bot.py
https://github.com/hackthebox/Hackster/blob/3cbaf12b38be8590b687247b0d3b5e2e9bfbb54c/src/bot.py

[3] [4] [22] Tasks | Pycord Guide
https://guide.pycord.dev/extensions/tasks

[5] [6] [9] [10] [13] [15] [16] [24] [25] [26] [27] [28] [30] projektstruktur.txt
https://github.com/Rabbit-Fur/System-by-FUR/blob/ac05d09cb52d1156761d558700f2b2ff27e534ee/projektstruktur.txt

[7] [8] [11] [12] [14] [29] [33] [34] [35] reminder_autopilot.py
https://github.com/Rabbit-Fur/try/blob/9eb8fc0f305f2a6242392062c6eb9d57439929bd/bot/cogs/reminder_autopilot.py

[17] I'm wondering how this is possible? Discord bot - Stack Overflow
https://stackoverflow.com/questions/69810414/im-wondering-how-this-is-possible-discord-bot

[18] [19] [20] [21] Add Custom Leaderboards to Your Discord Bots | by Daniel Kogan | Dev Genius
https://blog.devgenius.io/discord-leaderboards-with-firebase-a9d17d5228fd?gi=16819e26f3f3

[23] python - How do I make a leaderboard command? Discord.py - Stack Overflow
https://stackoverflow.com/questions/70490972/how-do-i-make-a-leaderboard-command-discord-py

[36] [37] [39] python - How to add localization to describe a command and its arguments in discord.py - Stack Overflow
https://stackoverflow.com/questions/79466962/how-to-add-localization-to-describe-a-command-and-its-arguments-in-discord-py

[38] thegamecracks/discord.py-i18n-demo: A guide to ... - GitHub
https://github.com/thegamecracks/discord.py-i18n-demo