ChatGPT

Technisches Konzept: FUR-Kalender v2

Architekturprinzipien (Clean Architecture & Async Patterns)

Der Kalenderbot folgt einem **Clean-Architecture-Ansatz**: Kernlogik (Use Cases) wird strikt von UI- und Infrastrukturschichten getrennt, um Testbarkeit und Erweiterbarkeit zu maximieren [1] [2]. Wir verwenden Cogs als **Controllers** (Discord-UI-Schicht), Services für Geschäftslogik (Applikationsschicht) und Models/Utilities für Datenlogik (Entity/Infrastructure-Schicht). Jede Schicht kennt nur die darunter liegende, um Abhängigkeiten zu minimieren. Asynchrone Programmierung (async/await) ist obligatorisch: Discord-API-Aufrufe und Google-API-Aufrufe werden nicht-blockierend ausgeführt, z.B. mit `discord.ext.tasks` für geplante Hintergrundaufgaben [3]. So kann der Bot gleichzeitig auf Slash-Kommandos reagieren und im Hintergrund synchronisieren, ohne den Event-Loop zu blockieren.

Modulstruktur (Dateien & Layer)

- **Cogs (UI-Schicht)**: Jeder Hauptbefehl (`/calendar today/week/link/timezone`) wird in einem eigenen Cog implementiert. Cogs importieren keine Geschäftslogik direkt, sondern delegieren an Service-Klassen. So bleiben sie schlank.
- **Services (Applikationsschicht)**: Enthalten die Kerngeschäftslogik: Google-Calendar-Synchronisation, Reminder-Logik, DM-Versand. Z.B. eine `CalendarSyncService` mit Methoden wie `sync_user_events(user_id)` und `get_upcoming_events(user_id, timeframe)`. Diese Services können von mehreren Cogs wiederverwendet werden.
- **Models/Repos (Domain-Schicht)**: Definition von Datenmodellen (z.B. `User`, `Event`) und Repository-Klassen für DB-Zugriff. Durch Pydantic/Dataclasses strukturieren wir Event-Daten (Titel, Startzeit, Beschreibung, Kalender-ID). Eine Abstraktionsebene (z.B. `CalendarRepository`) isoliert MongoDB- oder SQLite-Operationen.
- **Utilities & Helpers**: Hilfsklassen für OAuth2/PKCE-Flow, Datum/Uhrzeit-Konvertierungen, Logging, Fehlerbehandlung. Diese stehen allen Schichten zur Verfügung, aber Services und Cogs importieren sie sparsam, um Abhängigkeiten gering zu halten.

Datenfluss (Sync und DM-Trigger)

1. **OAuth-Flow & Token-Speicherung**: Beim Befehl `/calendar link` erhält der User einen OAuth2-PKCE-Link (Google-Client-ID, Code Challenge, Scope `calendar` usw.). Nach erfolgreicher Authentifizierung speichert der Bot das `access_token` und `refresh_token` sicher in der DB.
2. **Initiale Volle Synchronisation**: Nach dem ersten Login eines Users führt `CalendarSyncService` eine vollständige Abfrage (`events().list`) an Google Calendar durch. Die Antwort liefert ein `nextSyncToken` [4], das persistent abgespeichert wird.
3. **Inkrementeller Sync**: Später führt ein Hintergrundtask (z.B. täglich 2 Uhr UTC via `@tasks.loop`) einen inkrementellen Sync durch. Dabei wird der vorherige `syncToken` als Parameter verwendet. Die Google-API liefert nur geänderte und gelöschte Events seit dem letzten Sync, und einen neuen `nextSyncToken` [5] [6]. Geänderte Events aktualisieren wir in der Datenbank, gelöschte entfernen wir. So sparen wir Bandbreite und API-Aufrufe.
4. **Reminder-Generierung**: Ein weiterer `@tasks.loop` prüft passend voreingestellte Zeiten (z.B. morgens für Tagesübersicht, Montags für Wochenübersicht). Er erstellt für jeden Benutzer eine

DM-Zusammenfassung der anstehenden Events (inkl. Uhrzeit, Titel, Beschreibung). Alternativ können Nutzer sofort `/calendar today` oder `/calendar week` aufrufen, was direkt aus DB oder per Echtzeit-Query bedient wird.

5. **Discord-Nachrichten**: Der Bot sendet die Event-Übersichten als Embed-Personalisierung per `await user.send(embed=...)`. Fehlerhafte DMs (z.B. `discord.Forbidden` bei blockierten DMs) werden abgefangen und geloggt, ohne den gesamten Task abzubrechen. Zwischen zahlreichen DMs setzen wir ggf. kurze Delays, um Discord-Rate-Limits zu schonen.

Sicherheit (OAuth2 mit PKCE & Tokenhandling)

Für OAuth2 verwenden wir den **Authorization Code Flow mit PKCE**. Dabei generiert der Bot clientseitig einen geheimen `code_verifier` und einen darauf basierenden `code_challenge`. Der Auth-Request an Google enthält nur den `code_challenge` [7], sodass der Client-Secret bei diesem Schritt nicht offengelegt wird. Erst beim Token-Austausch übergeben wir `code_verifier` zusammen mit dem erhaltenen Auth-Code an Googles Token-Endpoint [7]. Dies verhindert das Abgreifen des Tokens durch Angreifer, da diese ohne den `code_verifier` nichts mit dem Auth-Code anfangen können. Die auf diese Weise erhaltenen Access- und Refresh-Tokens speichern wir verschlüsselt oder zumindest als Umgebungs- bzw. DB-Secret. Bei jedem API-Aufruf setzen wir Zugriffstoken ein, und bei Ablauf automatisch den Refresh-Token ein, um neue Access-Tokens zu holen.

Erweiterbarkeit für weitere Kalendersysteme

Die Architektur ist Provider-unabhängig ausgelegt. Wir definieren eine **abstrakte Kalender-Schnittstelle** (z.B. `CalendarProvider`) mit Methoden wie `list_events(sync_token)` oder `create_event()`. Für Google realisieren wir sie mit `GoogleCalendarProvider`, später könnten wir etwa `OutlookCalendarProvider` (Microsoft Graph API) oder `ICloudCalendarProvider` (z.B. über die `pyicloud`-Bibliothek) hinzufügen. Die Services arbeiten nur mit dieser Abstraktion, sodass die Unterstützung weiterer Systeme modular erweitert werden kann. Auch das Event-Modell bleibt generisch (Titel, Beschreibung, Start/End, Zeitzone), um Multi-Provider-Daten zu vereinen. Für einfache Fälle könnte man auch auf das iCalendar-Format (ICS) setzen.

Risiken & Bottlenecks

- **API-Rate-Limits**: Google begrenzt Anfragen auf z.B. 5 pro Sekunde pro Nutzer und 1M/Tag pro Projekt. Überschreitungen liefern 403/429-Fehler [8]. Wir müssen mit **Exponential Backoff** arbeiten (Clients handhaben das meist automatisch). Für Discord gilt ähnliches – der Bot kann nur eine begrenzte Anzahl von Nachrichten pro Sekunde absetzen. Deshalb gruppieren wir DM-Versand oder verteilen ihn über Sekunden hinweg.
- **API-Quota & Synchronisierung**: Die Verwendung von `syncToken` minimiert abrufende Daten. Bei großen Kalendern und Updates („Spiky Traffic“) folgen wir Googles Empfehlung, Traffic zu verteilen und Push-Notifications zu erwägen [9]. Falls ein `syncToken` abläuft (HTTP 410), muss ein kompletter Neusync gestartet werden.
- **Fehler- und Ausfallbehandlung**: Wir implementieren try/except-Logik bei Netzwerkfehlern oder Discord-Ausfällen, loggen Ausnahmen und wiederholen kritische Schritte (z.B. beim Lockdown von DMs). Ein Fehler in der Synchronisation soll nicht den gesamten Bot lahmlegen.
- **Deployment-Einschränkungen**: Auf Railway ist der Dateispeicher flüchtig. Daher speichern wir persistent auf MongoDB (besser für Docker/Cloud) statt lokal mit SQLite, es sei denn, Railway stellt ein Speicher-Volume zur Verfügung. Die DB-Anbindung (z.B. MongoDB Atlas) muss mit Umweltvariablen konfiguriert werden.

Empfohlene Bibliotheken und Technologien

- **Discord-Interaktion**: discord.py bzw. Pycord. Cogs und `discord.ext.tasks` zur Strukturierung und Terminplanung [3]. Für Buttons/Interaktionen: `discord.ui.View`, `discord.ui.Button`. Embed-Erstellung via `discord.Embed`.
- **Google API**: `google-api-python-client` zusammen mit `google-auth-oauthlib` für OAuth2/PKCE. Das Beispiel hier zeigt den Zugriff auf den Calendar-Service [10]:

```
from google.oauth2.credentials import Credentials
from googleapiclient.discovery import build

creds = Credentials.from_authorized_user_file("token.json", SCOPES)
service = build("calendar", "v3", credentials=creds)
now = datetime.datetime.utcnow().isoformat() + 'Z'
events = service.events().list(
    calendarId="primary", timeMin=now,
    maxResults=10, singleEvents=True, orderBy="startTime"
).execute().get('items', [])
```

(Quellenbeispiel: Google Calendar API Quickstart [10])

- **Datenbank**: Für asynchrone Zwecke Motor (AsyncIO-Driver für MongoDB) oder ein ODM wie Beanie. Motor erlaubt etwa `await collection.insert_one(doc)` [11]. Bei SQLite: `aiosqlite` für Async, wenn ein Dateispeicher doch sinnvoll ist. Für ~100 Nutzer ist beides performant genug.
- **OAuth & Sicherheit**: Für PKCE kann z.B. Authlib genutzt, oder eigenständig `secrets.token_urlsafe()` plus `hashlib.sha256` für Challenge generieren. TLS ist bei Google verpflichtend (HTTPS). Secrets (Client-ID) und Tokens nie öffentlich einbinden, sondern über Umgebungsvariablen/einen Secret-Manager.
- **Datum/Uhrzeit**: Python `zoneinfo` (oder `pytz`) zur Zeitzonenkonvertierung. Benutzerzeit individuell speichern (z.B. `"Europe/Berlin"`).
- **Fehlerüberwachung**: Logging und Monitoring (Sentry o.ä.) zur Laufzeitüberwachung. So erkennt man schnell, falls DM-RateLimits oder API-Quotas Probleme machen.

Beispielcode – Discord-Interaktionen und Event-Darstellung

Im Cog können Slash-Kommandos definieren werden, z.B. `/calendar today`, das die Events des Tages als Embed schickt:

```
@commands.slash_command(description="Zeige deine Events für heute")
async def calendar_today(self, ctx):
    user = ctx.author
    tz = ZoneInfo(user_timezone)  # z.B. "Europe/Berlin"
    start = datetime.now(tz).replace(hour=0, minute=0, second=0)
    end   = start + timedelta(days=1)
    events = await calendar_service.get_events(user.id, start, end)
    embed = discord.Embed(title=f"Deine Events am {start.date()}",
color=0x3498db)
```

```
    for e in events:
        time_str = e.start.astimezone(tz).strftime("%H:%M")
        embed.add_field(name=time_str, value=e.summary, inline=False)
    await user.send(embed=embed)  # DM an den Benutzer
```

Für **Buttons** (z.B. zum Verknüpfen des Google-Kontos) nutzen wir `discord.ui.View`:

```
view = discord.ui.View()
auth_url = build_oauth_url(...)  # PKCE-URL für Google OAuth2
view.add_item(discord.ui.Button(label="Google-Kalender verknüpfen",
url=auth_url))
await ctx.respond("Bitte verknüpfe deinen Google-Kalender:", view=view)
```

Integration in bestehenden Bot (Rollen & Berechtigungen)

Das Kalender-Module wird als neuer Cog geladen. Berechtigungen steuern wir z.B. über `@commands.has_role("Kalenderrolle")` oder Discord-Permissions (wenn nur bestimmte Mitglieder Zugriff haben sollen). Bei Slash-Kommandos können `default_member_permissions` oder `dmp_only=True` gesetzt werden, damit nur befugte Nutzer (z.B. mit einer bestimmten Serverrolle) Befehle nutzen können. Die Logik für Erinnerungen läuft ohnehin Benutzer-spezifisch (DMs), sodass Server-Rollen vor allem für Verwaltungskommandos nötig sind (z.B. `/calendar settimezone`, `/calendar setfrequency`).

Durch diese modulare Architektur bleibt das FUR-System übersichtlich: Der Kalender-Cog kümmert sich nur um Discord-UI und delegiert an klar getrennte Services. Kerndaten (Tokens, Events, Einstellungen) speichern wir in der DB. So entsteht ein sauberes, skalierbares und wartbares Event-Synchronisationssystem mit Discord als Benutzeroberfläche [1] [4].

Quellen: Google Calendar API (SyncToken) [4], Discord.py Tasks/Dokumentation [3], Clean Architecture-Prinzipien [1] [2], OAuth2-PKCE (Auth0) [7], Motor Async MongoDB docs [11].

---

[1] [2] Why Clean Architecture makes debugging easier | Product Blog • Sentry
https://blog.sentry.io/why-clean-architecture-makes-debugging-easier/

[3] discord.ext.tasks – asyncio.Task helpers
https://discordpy.readthedocs.io/en/stable/ext/tasks/

[4] [5] [6] Synchronize resources efficiently | Google Calendar | Google for Developers
https://developers.google.com/workspace/calendar/api/guides/sync

[7] Authorization Code Flow with Proof Key for Code Exchange (PKCE)
https://auth0.com/docs/get-started/authentication-and-authorization-flow/authorization-code-flow-with-pkce

[8] [9] Manage quotas | Google Calendar | Google for Developers
https://developers.google.com/workspace/calendar/api/guides/quota

[10] Python quickstart | Google Calendar | Google for Developers
https://developers.google.com/workspace/calendar/api/quickstart/python

[11] Tutorial: Using Motor With asyncio - Motor 3.7.1 documentation
https://motor.readthedocs.io/en/stable/tutorial-asyncio.html