ChatGPT

Best Practices: Google OAuth2 mit Flask, Calendar-API, MongoDB, Discord-Bot und Railway

Google OAuth2 mit Flask in Railway

Verwenden Sie die Google-Auth-Library (`google-auth-oauthlib.flow`) in Flask und konfigurieren Sie die OAuth-Flow über Umgebungsvariablen für Client-ID, Secret und Redirect-URI. Sorgen Sie für HTTPS (Rails verwendet standardmäßig `*.up.railway.app`). Aktivieren Sie offline-Zugriff (`access_type="offline"`, `prompt="consent"`), um Refresh-Tokens zu erhalten. Beispiel:

```
flow = Flow.from_client_secrets_file(client_secrets, scopes=SCOPES,
redirect_uri=REDIRECT_URI)
authorization_url, state = flow.authorization_url(access_type="offline",
prompt="consent", include_granted_scopes="true")
session["oauth_state"] = state
```

[1] [2]

Speichern Sie das erhaltene Token nach erfolgreicher Callback-Abfrage nicht als Pickle, sondern als JSON:

```
flow.fetch_token(authorization_response=request.url)
creds = flow.credentials
TOKEN_PATH.write_text(creds.to_json())
```

[3]

Wichtig: Railway-Instanzen bieten nur ephemeren Speicher (~10 GB) [4]. Für dauerhafte Speicherung sollten Sie Volumes anbinden oder eine Datenbank nutzen. Standard-Flask-Sessions sind cookie-basiert (signiert mit `SECRET_KEY`) und funktionieren auch auf Railway; falls trotzdem Session-Daten verloren gehen (z.B. bei Neustart), kann man das OAuth-State-Token serverseitig (z.B. in einer DB oder In-Memory-Map) vorhalten [2].

Umgang mit `state` und Sessions

Der OAuth-Parameter `state` dient der CSRF-Prüfung und Verknüpfung mit dem Benutzer. Legen Sie `state` z.B. zufällig fest und speichern Sie ihn sowohl in der Session als auch (falls nötig) zusätzlich in einem kurzlebigen Store oder Ihrer Datenbank. Beim Callback vergleichen Sie `request.args.get("state")` mit dem gespeicherten Wert. Beispiel aus dem Code:

```
req_state = request.args.get("state")
stored_state = session.pop("oauth_state", None)
if req_state == stored_state or req_state in state_map:
    # State gültig
    state_map.pop(req_state, None)  # Aufräumen
else:
    return Response("Invalid OAuth state", status=400)
```

2

Falls Flask-Sessiondaten verloren gehen, hilft ein sekundärer Store (z.B. ein Dictionary oder Redis). Alternativ kann man `state` so gestalten, dass er z.B. einen verschlüsselten Nutzer- oder Client-Identifikator enthält.

Speicherung und Aktualisierung der OAuth-Token

Speichern Sie die OAuth-Credentials im JSON-Format (via `Credentials.to_json()`), nicht als Pickle. Beispielsweise:

```
creds = flow.credentials
TOKEN_PATH.write_text(creds.to_json())
```

3

Verwalten Sie das Refresh-Token sicher (z.B. verschlüsselt in der DB) und prüfen Sie beim Laden der Credentials, ob sie abgelaufen sind. Beispiel mit `google.oauth2.credentials.Credentials`:

```
creds = Credentials.from_authorized_user_file(TOKEN_PATH, SCOPES)
if creds and creds.expired and creds.refresh_token:
    creds.refresh(Request())
    TOKEN_PATH.write_text(creds.to_json())
```

5

In Cloud-Umgebungen ist es besser, Tokens in einem persistenten Store (Datenbank oder Redis) statt im Dateisystem abzulegen. Der Code zeigt exemplarisch, wie man das Sync-Token in MongoDB speichert:

```
await self.tokens.update_one({"_id": "google"}, {"$set": {"token": token}},
upsert=True)
```

6

Sie können für zusätzliche Sicherheit das Refresh-Token (und ggf. Access-Token) verschlüsseln (z.B. mit Fernet, wie in `oauth_utils.encrypt_refresh_token()` gezeigt 7) oder Credentials in einem Secrets-Store sichern.

Google Calendar API (Event-Sync)

Nutzen Sie die Google Calendar API über `googleapiclient.discovery.build`. Nach erfolgreicher Authentifizierung erstellen Sie einen Dienst-Client (`build("calendar", "v3", credentials=creds)`). Anschließend synchronisieren Sie Events etwa so:

- **Initialer Sync:** Abfrage aller künftigen Events über `events().list(calendarId, timeMin=now)`.
- **Inkrementeller Sync:** Speichern Sie `nextSyncToken` und verwenden Sie bei nachfolgenden Aufrufen `events().list(syncToken=lastSyncToken)`.
- **Fehler 410 behandeln:** Sollte das Sync-Token ungültig werden (HTTP 410), führen Sie einen Vollsync mit neuer `timeMin` durch.

Beispielhafte Logik (aus `CalendarService`):

```
params = {"calendarId": cal_id, "singleEvents": True, "maxResults": 2500}
if last_sync_token:
    params["syncToken"] = last_sync_token
else:
    params["timeMin"] = datetime.utcnow().isoformat() + "Z"

try:
    data = service.events().list(**params).execute()
except HttpError as exc:
    if exc.resp.status == 410:
        # Sync-Token abgelaufen – Vollsync
        params.pop("syncToken", None)
        params["timeMin"] = datetime.utcnow().isoformat() + "Z"
        data = service.events().list(**params).execute()
    else:
        raise
events = data.get("items", [])
# Speichern oder Aktualisieren der Events in der DB ...
next_token = data.get("nextSyncToken")
```

8 9

Für Schreibzugriff (neue oder geänderte Events) verwenden Sie `service.events().insert(...)` oder `update(...)`. Denken Sie daran, Berechtigungen in den OAuth-Scopes entsprechend zu setzen (z.B. `auth/calendar.events`).

Speicherung von Events in MongoDB

Speichern Sie die synchronisierten Events in einer MongoDB (z.B. Atlas oder Railway Plugin). Empfehlenswert ist ein Ereignis-Dokument mit Feldern wie `google_id`, `title`, `start`, `end`, `description`, `updated`, `event_time` usw. (das Beispiel nutzt UTC `event_time` für Sortierung). Beispiel aus `CalendarService._build_doc`:

```
return {
    "google_id": event.get("id"),
    "title": event.get("summary", ""),
    "description": event.get("description"),
    "location": event.get("location"),
    "updated": event.get("updated"),
    "start": start_dt, "end": end_dt,
    "event_time": event_time,
    "source": "google",
    "status": event.get("status"),
}
```

10

Aktualisieren Sie Einträge per `update_one({"google_id": id}, {"$set": doc}, upsert=True)`. So bleiben nur einmalige Datensätze pro Event bestehen. Achten Sie auf Indexe (z.B. auf `event_time` für Abfragen).

Für die Verbindung: Verwenden Sie die offizielle MongoDB-URL in Umgebungsvariablen (`MONGODB_URI`). Im Beispiel wird `AsyncIOMotorClient` (async) oder PyMongo (sync) genutzt.

Discord-Bot Integration (Aufgaben und Cog)

Verknüpfen Sie den Google-Login über einen Discord-Slash-Befehl: Der Bot erzeugt eine OAuth-URL mit PKCE und leitet den Nutzer (z.B. per Button) dorthin. Beispiel:

```
verifier = generate_code_verifier()
challenge = generate_code_challenge(verifier)
state = secrets.token_urlsafe(16)
get_collection("oauth_states").update_one(
    {"discord_id": str(user.id)},
    {"$set": {"verifier": verifier, "state": state}},
    upsert=True,
)
url = build_authorization_url(client_id, redirect_uri, scopes, state,
challenge)
```

11

Damit kann der Bot nach Callback und Code-Austausch das zugehörige Discord-Konto anhand des gespeicherten `state` oder `verifier` in der Datenbank zuordnen.

Verwenden Sie `discord.ext.tasks.loop`, um regelmäßige Sync- und Erinnerungsaufgaben auszuführen. Beispiel aus dem Cog (`daily_loop`) – stündlich wird geprüft, ob ein täglicher Reminder fällig ist, und dann Events per DM verschickt:

```
@tasks.loop(hours=1)
async def daily_loop(self):
    await self.bot.wait_until_ready()
    now = datetime.now(timezone.utc)
    if should_send_daily(now):
        events = await self.service.get_events_today()
        await self._send_reminders(events, title)
```

[12]

Ebenso kann eine eigene Task (`google_sync_loop`) im Hintergrund das `sync_to_mongodb()`-Modul starten. Falls Sie Flask-Funktionen innerhalb des Bots aufrufen, denken Sie daran, `with app.app_context():` zu verwenden (siehe `google_sync_task` im Beispiel).

Railway-Deployment: Filesystem und Volumes

Railway-Services haben nur **ephemeren Speicher** (Standard ~10 GB) [4]. Beim Neustart der App oder Skalierung gehen lokale Dateien verloren. Für Persistenz **verwenden Sie Volumes** oder externe DBs/Services. Beispiele:
- MongoDB als verwalteter Dienst (Atlas oder Railway-Plugin).
- Volumes für Daten, die zu groß/zu oft geändert sind (z.B. wenn Sie doch lokal Dateien ablegen wollen).

Der Code speichert Tokens in `/data/google_token.json`, was im Docker-Container liegt – aber beachten Sie, dass `/data` in Railway ebenfalls flüchtig ist. Besser: Mongo/Redis oder ein angehängtes Volume nutzen.

Legt Umgebungskonfiguration vollständig per ENV-Variablen fest (`GOOGLE_CLIENT_ID`, `GOOGLE_SECRET`, `GOOGLE_REDIRECT_URI`, `MONGODB_URI`, etc.) und vermeiden Sie harte Pfade. Railway setzt `/data` als Wurzel des Containersystems, weitere Pfade können in `settings.json` oder `Procfile` definiert sein.

Alternatives zu Datei-Token: MongoDB oder Redis

Statt `token.json` im Dateisystem speichern viele Produktions-Apps Tokens in Datenbanken oder Caches. Beispiele:
- **MongoDB**: Speichern Sie JSON-Credentials direkt in einer Collection. Beispiel: `{_id: user_id, credentials: {...}}`. Beim Start laden und ggf. aktualisieren.
- **Redis/Key-Value-Store**: Temporäre Speicherung für schnellen Zugriff, wenn mehrere Bot-Instanzen parallel laufen.
- **Datenbank** (SQL/NoSQL): Tokens verschlüsselt in einer Tabelle/Collection (Schlüssel: Nutzer-ID).

Die Vorteile: Keine Abhängigkeit vom lokalen Dateisystem, automatische Persistenz über Deployments, leichtere Skalierung (mehrere Dynos greifen auf denselben Store).

Redirect-URI-Strategien und Railway-spezifische Fallen

Der Redirect-URI muss **exakt** mit der Google-Cloud-Konfiguration übereinstimmen (einschließlich Protokoll und Host). Railway verwendet standardmäßig `https://{project}.up.railway.app`. Bei

Verwendung eines **eigene Domainnames** (via CNAME) müssen Sie sicherstellen, dass die App-Logik diesen Domänennamen verwendet, und beide URIs in der Google-Konsole eintragen. Falls die App ungefragt das Railway-„*.up.railway.app“-Domain verwendet, führt das zu einem Fehler „invalid_request“ [13].

Tipps: - Definieren Sie die Redirect-URI per ENV, z.B. `GOOGLE_REDIRECT_URI=https://myapp.example.com/oauth2callback`.
- Fügen Sie in Google Cloud sowohl die Railway-URI als auch ggf. die Custom-URI als autorisierte Redirect-URIs hinzu.
- Achten Sie auf das http(s)-Präfix: Google verlangt https (außer im Debug). In Entwicklung können Sie `os.environ['OAUTHLIB_INSECURE_TRANSPORT'] = '1'` setzen, damit `http://localhost:...` zulässig ist (wie im Beispielcode [14]).
- Überprüfen Sie, dass nach Deployments die Domain gleich bleibt oder passen Sie die Google-Settings an.

Zusammenfassung

Für eine robuste OAuth2-Integration in einer Discord-Bot-Architektur auf Railway gilt: Verwenden Sie Googles offizielle Bibliotheken in Flask, speichern Sie Tokens sicher (JSON in DB statt Pickle-Dateien), synchronisieren Sie Kalender-Events über die Calendar-API mit Sync-Tokens, und legen Sie Daten in einer persistenten Datenbank wie MongoDB ab. Nutzen Sie Discords Task-Loop-System für periodische Syncs und Reminders. Berücksichtigen Sie Railway-spezifische Limitierungen (ephemeres Filesystem, Domain-Zwischenschichten) und sichern Sie Sitzungs- und State-Daten extern ab, um Ausfälle zu vermeiden [4] [3] [2].

Quellen: Beispiele aus dem Code (OAuth-Flow und Sync-Logik [3] [2] [6]) sowie Railway-Dokumentation zu Speichergrenzen [4] und Community-Berichten zu Redirect-URI-Fallen [13].

---

[1] [2] [3] [5] google_oauth_web.py
https://github.com/Rabbit-Fur/try/blob/1b47301dde9d367a96a7b6b9e2acb558335852c1/web/routes/google_oauth_web.py

[4] Managing Services | Railway Docs
https://docs.railway.com/guides/services

[6] [8] [9] [10] calendar_service.py
https://github.com/Rabbit-Fur/try/blob/1b47301dde9d367a96a7b6b9e2acb558335852c1/services/calendar_service.py

[7] oauth_utils.py
https://github.com/Rabbit-Fur/try/blob/1b47301dde9d367a96a7b6b9e2acb558335852c1/utils/oauth_utils.py

[11] [12] calendar_cog.py
https://github.com/Rabbit-Fur/try/blob/1b47301dde9d367a96a7b6b9e2acb558335852c1/bot/cogs/calendar_cog.py

[13] Loading...
https://station.railway.com/questions/bug-report-google-drive-api-o-auth-redir-10e36397

[14] oauth_server.py
https://github.com/JaHe03/Canvas-Discord-Bot/blob/ef9499e027f43733595152ff46a6f39e1c8c1b6b/web/oauth_server.py