ChatGPT

Google Calendar Synchronization

- Use secure OAuth 2.0 with minimal scopes and avoid storing user credentials insecurely. For example, use PKCE and state parameters and keep the Google API token encrypted or in a protected store [1] [2]. It's best to use a dedicated service account or calendar (not a personal account) so you can hand over the project easily [3].
- Schedule syncs via a background task and limit the query window. For instance, run a Discord `tasks.loop` (e.g. every few minutes) and only fetch upcoming events in a narrow time window to reduce load [3] [4]. In our code, the Google sync loop runs periodically inside an app context [5], and the Calendar cog also uses hourly loops to DM upcoming events [4].
- Follow Google's best practices: use incremental updates (where possible), check for rate-limit headers, and refresh expired tokens automatically. Store the OAuth token in a credentials file or secure parameter store (never log or hardcode secrets) [1] [6]. In the FUR codebase, the config defines `GOOGLE_CALENDAR_SCOPES`, client ID/secret, and a credentials file path [6].

Reminder Opt-Out Cog

- **Provide an opt-out command.** Allow users to disable reminders with a slash command (e.g. `/reminder_stop`), and store their ID in a database. In the example cog, the command simply does an upsert on a `reminder_optout` collection [7].
- **Acknowledge quietly.** Reply with an *ephemeral* confirmation so only the user sees it [8]. This avoids cluttering public channels.
- **Respect Discord policy.** Always honor user opt-out requests. Discord's developer policy explicitly says you must allow users to opt out of notifications [9]. In practice, check this opt-out flag before sending any DM (as `ReminderCog` does by calling `is_opted_out()`) [10].

DM Broadcast Cog

- **Restrict usage.** Only allow server admins (or a specific role) to invoke bulk DMs [11]. In the code, the command checks the user's roles against `Config.ADMIN_ROLE_IDS` before proceeding [12] [11]. This prevents abuse and respects privacy.
- **Rate-limit and lock.** Prevent overlapping or too-frequent broadcasts. For example, track the last-use timestamp and reject new broadcasts within a minute [13]. Use an `asyncio.Lock` to ensure only one broadcast runs at a time [14]. This avoids race conditions and saturating the API.
- **Throttle message sending.** Pause between each DM to avoid rate limits (e.g. 1–2 seconds delay) [15] [16]. The sample code uses a 1.5s delay per member [15]. Also catch `discord.Forbidden` exceptions if a member has DMs blocked, and log failures [17].
- **Report results.** After sending, reply with a summary (counts of successes/failures), ideally in an embed or message [18]. This gives the admin feedback. In our code, an embed shows "Sent" vs "Failed" counts [18].
- **Follow Discord policy.** Do *not* send unsolicited or irrelevant DMs. Discord requires explicit permission to message users [19]. Only send broadcast messages that members expect (e.g. important announcements), and honor opt-outs (if any). The code checks for permissions and blocks by design [11] [19].

Base Commands Cog

- **Organize global commands.** Put utility commands (ping, info, etc.) in their own cog. In the example, `BaseCommands` defines `/ping` and `/fur_info` [20] [21]. This keeps the code modular.
- **Use ephemeral replies for simple checks.** For something like `/ping`, reply ephemerally with "pong" to avoid channel spam [20]. In contrast, more informative commands (like server info) can post publicly.
- **Personalize responses.** Look up user preferences (e.g. language) in your DB and use them. The code reads the user's `lang` field from Mongo and passes it to the i18n function `t()` [22].

Leaderboard Cog

- **Precompute and cache.** Instead of querying the database on every `/top` command, use a background task to refresh the top scores periodically. For example, the cog starts a `@tasks.loop` that updates the top 10 for each category every 10 minutes [23]. The results are stored in an in-memory cache (`leaderboard_cache`) [24].
- **Optimize DB queries.** When updating, query with a sort and limit to only fetch what you need. The code uses `find().sort("score", -1).limit(10)` [25], which is efficient.
- **Use the cache at command time.** The `/top` command first checks the cache and only refreshes if needed [26] [27]. This minimizes database load and latency in responding to users.
- **Graceful error handling.** If a category is unknown or empty, reply with a helpful message instead of crashing [28]. The code sends an ephemeral error if there are no rows for the given category [29].

Newsletter Opt-Out Cog

- **Mirrors reminder opt-out.** Provide a `/newsletter_stop` command that inserts the user into a `newsletter_optout` collection [30].
- **Ephemeral confirmation.** As with reminders, send a private success/failure message back [31].
- **Abide by opt-out policies.** Allowing users to unsubscribe from news or vote reminders is mandatory on platforms like Discord [9]. Always check this opt-out list when sending newsletters or announcements.

Newsletter Autopilot Cog

- **Scheduled loops.** Use Discord tasks (e.g. `@tasks.loop(hours=1)`) to check if it's time to send the newsletter (weekly) or daily overview. The example cog runs two separate hourly loops – one for the weekly digest and one for the daily digest [32] [33]. Each loop checks the current time and only sends on the target hour.
- **Enable/disable flag.** Include a config flag (e.g. `ENABLE_NEWSLETTER_AUTOPILOT`) so you can easily turn off automatic sends (useful in dev or maintenance) [34].
- **Skip bots and opt-outs.** When looping through `guild.members`, skip any bots and any users who have opted out (checked via the DB) [35]. The code uses `get_collection("newsletter_optout")` to filter these out.
- **Catch and count failures.** For each DM attempt, catch `discord.Forbidden` to count blocked users, and catch general exceptions so that one failure doesn't stop the loop [36]. Maintain counters (sent/block/error) for reporting.

- **Manual trigger.** Provide an admin-only slash command (e.g. `/newsletter_now`) that immediately sends the newsletter and reports stats [37]. This is useful for testing or emergency announcements.

Newsletter Cog (Announcements)

- **Permission check.** Only allow designated roles or admins to use the announce command [38]. The example `user_is_admin()` function checks for certain role IDs or the built-in administrator permission.
- **Input validation.** Ensure the message isn't empty or too long before sending [39]. If invalid, respond with an ephemeral usage hint.
- **Robust error handling.** Wrap the send in a try/except and catch `discord.Forbidden` if the bot lacks send permissions in the channel [40]. Reply with a user-friendly error message in that case. Always log the action or any error [41] [42].
- **Ephemeral confirmations.** Let the announcer know privately that the announcement succeeded or failed [43], while sending the actual announcement publicly in the target channel.

Hourly Reminders Cog

- **Background loop with readiness.** Use a `@tasks.loop(minutes=60)` for hourly posts [44]. Include a `before_loop` that waits for the bot to be ready [45] to prevent race conditions.
- **Environment gating.** Skip sending reminders in non-production environments to avoid spam in testing servers. The code checks `is_production()` at runtime [46].
- **Check channel existence.** Fetch the target channel by ID and log a warning if it's missing [47]. This prevents exceptions if the channel was deleted or misconfigured.
- **Slash trigger for admins.** Provide a `/reminder_now` command restricted to server administrators so they can manually send the reminder anytime [48]. Respond ephemerally to the invoker with a success or failure notice.
- **Localized content.** Use an i18n function to include dynamic content (like the current time) in the message [49]. In our code, the reminder message includes UTC time and is localized through `t("reminder_hourly", ...)`.

Reminder Cog (Event & Personal Reminders)

- **Timed event reminders.** In a background loop (every few minutes), query upcoming events in a specific future window (e.g. 60 minutes from now) [50]. For each event, iterate over participants in the `event_participants` collection.
- **Avoid duplicates and honor opt-outs.** Before sending, check if this (event,user) pair is already in `reminders_sent`; if so, skip it [10]. Also skip any user who has opted out of reminders [51].
- **Fetch users safely.** Use `bot.get_user()` or `fetch_user()` to get the `discord.User` object. Handle `None` if the user is not found [52]. If DMs are disabled for that user, catch `discord.Forbidden` and log it [53].
- **Record sent reminders.** After sending a DM, insert a record into `reminders_sent` with timestamp, so you don't DM them again for the same event [54].
- **Slash-based personal reminders.** Provide commands like `/remind <minutes>` to let users set ad-hoc reminders. Validate the input range (e.g. 1–1440 minutes) [55]. Store these in a `user_reminders` collection with timestamps [56].
- **List and cancel.** Implement `/remind_list` to fetch and format a user's upcoming reminders, and `/remind_cancel` to clear them [57] [58]. Use Discord's timestamp formatting (`<t:...:R>`) for readability [59], and always reply ephemerally to keep personal data private.

Reaction Signup Cog

- **Use raw events.** Listen to `on_raw_reaction_add` and `on_raw_reaction_remove` so the bot picks up reactions even if the message isn't cached [60].
- **Filter by emoji.** Only respond to the designated signup emoji (e.g.). Ignore other reactions immediately to save work [60].
- **Extract event ID from the message.** Use a regex pattern (e.g. `\[ID:...`) to parse the event's unique ID from the message content [61]. If no match is found, ignore it.
- **Upsert participation.** When a user reacts, upsert a document in `event_participants` (with `$setOnInsert`) to record their sign-up [62]. This ensures no duplicate entries if they react twice. Log success (like "User X joined event Y") [63].
- **Remove on reaction removal.** On `on_raw_reaction_remove`, delete the corresponding document [64]. If a document was deleted (check `deleted_count`), log that the user left the event. This keeps the signup list accurate.

Calendar Cog

- **Group related commands.** Use an `app_commands.Group` (e.g. `/calendar`) to namespace calendar operations (today, week, link, timezone) [65]. This organizes slash commands neatly.
- **OAuth link with PKCE.** For Google authorization, generate a code verifier/challenge and a random state, store them per user, and build the OAuth2 URL. The code does this with `generate_code_verifier()` and `build_authorization_url(...)`, then sends the user a clickable button [2] [66].
- **Store user config.** Allow the user to set preferences like timezone via `/calendar timezone <name>`. Save their selection in the DB (e.g. in the `users` collection) for future use [67]. This is used when formatting event times.
- **Task-based summaries.** Use hourly loops to send daily or weekly schedules to everyone. For example, the cog runs a loop that checks if it's 8:00 UTC to send "events today" or Sunday 12:00 UTC for weekly [4] [68]. It fetches events via a `CalendarService` and sends each member a DM with an embed of events.
- **DM handling.** As always, skip bots and catch `discord.Forbidden` if a member blocks DMs [69]. Use a helper that converts event datetimes to each user's timezone (stored earlier) so that users see local times [70].

Sources: Relevant code and guidelines from the FUR bot's repository and community best practices have been cited throughout (see citations). These include example implementations of each feature and Discord's developer policies [9] [19].

---

[1] [3] Automating Event Management: A Discord to Google Calendar Bot - DEV Community
https://dev.to/c6z3h/automating-event-management-a-discord-to-google-calendar-bot-2i82

[2] [4] [65] [66] [67] [68] [69] [70] calendar_cog.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/bot/cogs/calendar_cog.py

[5] google_sync_task.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/utils/google_sync_task.py

[6] config.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/config.py

[7] [8] reminder_optout.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/bot/cogs/reminder_optout.py

[9] [19] Discord Developer Policy – Developers
https://support-dev.discord.com/hc/en-us/articles/8563934450327-Discord-Developer-Policy

[10] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] reminder_cog.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/bot/cogs/reminder_cog.py

[11] [12] [13] [14] [15] [16] [17] [18] dm_broadcast_cog.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/bot/cogs/dm_broadcast_cog.py

[20] [21] [22] base_commands.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/bot/cogs/base_commands.py

[23] [24] [25] [26] [27] [28] [29] leaderboard.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/bot/cogs/leaderboard.py

[30] [31] newsletter_optout.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/bot/cogs/newsletter_optout.py

[32] [33] [34] [35] [36] [37] newsletter_autopilot.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/bot/cogs/newsletter_autopilot.py

[38] [39] [40] [41] [42] [43] newsletter.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/bot/cogs/newsletter.py

[44] [45] [46] [47] [48] [49] reminders.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/bot/cogs/reminders.py

[60] [61] [62] [63] [64] reaction_signup.py
https://github.com/Rabbit-Fur/try/blob/1294a11125fe2979652cc30334cb30c11f47c7fd/bot/cogs/reaction_signup.py