Architectural Plan: Backend System for World of Warcraft Guild Management

Section 1: Authentication Strategy with Blizzard Battle.net OAuth2

This section details the authentication mechanism utilizing Blizzard Entertainment's Battle.net OAuth2 protocol, specifically focusing on the Authorization Code Flow required for accessing protected user data within the World of Warcraft (WoW) ecosystem.

1.1. Application Registration and Configuration

The foundational step involves registering the application within the Battle.net Developer Portal.[1] This process is mandatory to obtain the necessary credentials for interacting with the Blizzard API.

- **Client Credentials:** Upon successful registration, the portal provides a unique client_id and a corresponding client_secret.[2] These credentials identify the application to Blizzard's authorization server and are essential for both initiating the OAuth flow and exchanging authorization codes for access tokens. Secure management of the client_secret is paramount.
- **Redirect URIs:** The application must specify one or more redirect_uri(s) during registration.[2] These URIs represent the endpoints within the application where Blizzard's authorization server will redirect the user after they have authenticated and granted (or denied) permissions. For security, Blizzard mandates that these URIs must use HTTPS (SSL-enabled).[3] This ensures that the sensitive authorization code exchanged during the redirect is transmitted securely. The Developer Portal does not validate these URIs upon entry, placing the responsibility on the developer to ensure they are correct and secure.[3]
- **Scopes:** Scopes define the specific permissions the application requests from the user. To access WoW character data and user identity information, the application must request the appropriate scopes. The wow.profile scope grants access to the user's World of Warcraft characters and related profile data (collections, etc.).[3] The openid scope is used in conjunction with OpenID Connect (OIDC) to retrieve basic user identity information, such as their unique Battle.net account ID and BattleTag, via the /oauth/userinfo endpoint.[3] Requesting only necessary scopes adheres to the principle of least privilege.
- **Client Naming and Security:** Client names chosen during registration should be descriptive and are globally unique across all developer accounts.[2] These names

might be visible to users during the authorization process. Furthermore, Blizzard requires developers to enable Two-Factor Authentication (2FA) on their Battle.net accounts used for the Developer Portal.[2] This requirement underscores the sensitivity of API access and implies an expectation that developers will implement robust security measures within their own applications when handling user data obtained via the API.

1.2. Implementing the Authorization Code Flow

Accessing user-specific data, such as character lists or account collections, mandates the use of the OAuth 2.0 Authorization Code Flow.[3] This flow ensures explicit user consent before granting the application access to their private information. The Client Credentials flow, while simpler, is designed for server-to-server authentication where the application acts on its own behalf and is unsuitable for accessing user-delegated resources.[2]

The Authorization Code Flow proceeds as follows:

1. **Authorization Request:** The application initiates the flow by redirecting the user's browser to Blizzard's authorize_uri. Region-specific URIs exist (e.g., https://oauth.battle.net/authorize for US/EU/APAC).[3] This redirect includes several query parameters:
    - client_id: The application's registered client ID.[6]
    - scope: A space-separated list of requested scopes (e.g., wow.profile openid).[6]
    - state: A randomly generated, opaque value used to prevent Cross-Site Request Forgery (CSRF) attacks. The application should store this value temporarily and verify it upon receiving the callback.[6]
    - redirect_uri: One of the pre-registered HTTPS redirect URIs where the user will be sent back after authorization.[6]
    - response_type: Must be set to code to indicate the Authorization Code Flow.[6]
2. **User Authentication and Consent:** The user is presented with a Battle.net login page (if not already logged in) and an authorization screen detailing the permissions requested by the application. The user can choose to grant or deny these permissions. It's important to note that users may grant fewer scopes than initially requested.[6] The application must be prepared to handle scenarios where expected permissions (like wow.profile) are not granted, potentially by checking the scopes returned in the token response or by gracefully handling errors during subsequent API calls.
3. **Authorization Response:** If the user grants permission, Blizzard redirects the browser back to the application's specified redirect_uri. This redirect includes the authorization code (a short-lived, single-use credential) and the original state

parameter in the query string.[6]

4. **Token Exchange:** The application's backend server receives the request at the redirect_uri. It must first validate that the received state parameter matches the one generated in step 1 to mitigate CSRF risks. Then, the backend makes a secure, server-to-server POST request to Blizzard's token_uri (e.g., https://oauth.battle.net/token for US/EU/APAC).[3] This request includes:
    - HTTP Basic Authentication header containing the client_id and client_secret (-u <client_id>:<client_secret>).[6]
    - Request body parameters:
        - grant_type: Must be authorization_code.[6]
        - code: The authorization code received in the previous step.[6]
        - redirect_uri: The same redirect URI used in the initial authorization request.[6]

Blizzard strongly recommends using established and stable OAuth 2.0 libraries for implementing this flow rather than building it from scratch, due to the security complexities involved.[3]

1.3. Token Management (Acquisition, Storage, Validation)

Successfully exchanging the authorization code yields access tokens necessary for making authenticated API calls.

- **Token Response:** The response from the /oauth/token endpoint is a JSON object containing [6]:
    - access_token: The token used to authenticate API requests on behalf of the user.
    - token_type: Typically Bearer, indicating how the token should be used in API requests.[6]
    - expires_in: The lifetime of the access token in seconds. Battle.net access tokens have a duration of 24 hours (86400 seconds).[3]
    - scope: A string listing the scopes actually granted by the user.[6]
    - (Potentially) refresh_token: While not explicitly confirmed in the provided materials for Battle.net, standard OAuth2 flows often include refresh tokens for obtaining new access tokens without user re-interaction. Verification during implementation is needed.
- **Token Usage:** Access tokens are provided in the Authorization HTTP header of API requests, prefixed by Bearer (e.g., Authorization: Bearer <access_token>).[2] Sending tokens via query string parameters is deprecated and will cease to be supported.[6]
- **Secure Storage:** Access tokens (and refresh tokens, if used) are sensitive

credentials and must be stored securely on the server-side. They should **never** be stored in client-side code or insecure locations. Recommended practice involves encrypting the tokens before storing them in the database (e.g., in the users table, potentially using pgcrypto or Supabase Vault). The relatively long 24-hour lifespan of Battle.net access tokens [3] increases the potential impact if a token is compromised, further emphasizing the critical need for robust encryption at rest and secure transport (HTTPS) for all communication involving tokens.

- **Token Validation and Expiration:** Applications must handle token expiration and invalidation. An access token becomes invalid after 24 hours [3], or if the user changes their Battle.net password or explicitly revokes the application's authorization via their Battle.net account settings.[3] When an API call fails with an authentication error (e.g., 401 Unauthorized), the application should attempt to use a refresh token (if available and valid) to obtain a new access token. If refreshing is not possible or fails, the user must be prompted to re-authenticate via the full OAuth Authorization Code Flow. While Blizzard provides a /oauth/check_token endpoint for explicit token validation [7], relying on the token's stored expiration time and handling API errors is often a more practical approach in the application flow.

1.4. Linking Battle.net Identity to Application Users in Postgres

To associate a Battle.net user with an internal application user account, the application needs a stable, unique identifier from Battle.net.

- **Retrieving User Identity:** After obtaining an access token via the Authorization Code Flow, the application calls the /oauth/userinfo endpoint.[6] This endpoint requires the user's access token in the Authorization header [6] and returns a JSON object containing the user's unique Battle.net account id (a numerical identifier) and their battletag (e.g., PlayerName#1234).[6]
- **Database Association:** The application's users table in the Postgres database must store this Battle.net identifier to link the accounts. A suggested schema includes an internal id (UUID primary key), battlenet_id (e.g., bigint, marked unique), battletag (text), encrypted tokens, and token expiration timestamp. The battlenet_id serves as the immutable link between the application's user record and the external Battle.net identity. While the battletag is useful for display purposes, it can be changed by the user on Battle.net and therefore should not be used as the primary key or unique identifier for linking accounts; it should be updated upon login if it has changed.
- **Sign-Up/Sign-In Logic:** The integrated login process works as follows:
  1. User clicks "Login with Battle.net" in the application.

2. Application executes the OAuth Authorization Code flow (1.2).
3. Upon successful token exchange, the application calls /oauth/userinfo using the obtained access token to retrieve the user's battlenet_id and battletag.[6]
4. The application queries its users table for a record matching the received battlenet_id.
5. **Existing User:** If a matching record is found, the application updates the stored (encrypted) tokens and their expiration time, then establishes the user's session within the application.
6. **New User:** If no matching record exists, the application creates a new row in the users table, storing the battlenet_id, battletag, encrypted tokens, and expiration time. It then establishes the user's session.

This ensures each application user corresponds to exactly one Battle.net account, using the battlenet_id as the reliable, unique foreign identifier.

Section 2: Detailed Supabase Postgres Database Schema

This section outlines the proposed Postgres database schema designed for deployment on Supabase, incorporating the specific entities, relationships, and business rules required for the WoW guild management application.

2.1. Table Definitions

The following table definitions provide the core structure for storing application data. UUIDs are proposed for primary keys to facilitate potential future distribution or decoupling, while standard Postgres timestamp features are leveraged.

Table: Core Database Schema

Table Name	Column Name	Data Type	Constraints	Notes
users	id	uuid	PK, DEFAULT gen_random_uuid()	Internal application user ID
	battlenet_id	bigint	UNIQUE, NOT NULL, INDEXED	Unique Battle.net account ID from /oauth/userinfo
	battletag	text	NOT NULL	User's BattleTag (e.g.,

				Player#1234)
	encrypted_bnet_access_token	bytea	NULL	Encrypted Battle.net access token
	encrypted_bnet_refresh_token	bytea	NULL	Encrypted Battle.net refresh token (if applicable)
	bnet_token_expires_at	timestamp with time zone	NULL	Expiration time of the access token
	created_at	timestamp with time zone	NOT NULL, DEFAULT now()	Supabase default timestamp
	updated_at	timestamp with time zone	NOT NULL, DEFAULT now()	Supabase default timestamp
characters	id	uuid	PK, DEFAULT gen_random_uuid()	Internal character ID
	user_id	uuid	FK REFERENCES users(id) ON DELETE SET NULL, INDEXED	Link to the owning user (nullable if user unknown)
	blizzard_id	integer	NOT NULL	Character ID from Blizzard API (unique within region)
	name	text	NOT NULL	Character name
	realm_slug	text	NOT NULL	Realm slug (e.g., area-52)

	region	text	NOT NULL	Region (e.g., us, eu)
	level	integer	NULL	Character level
	playable_class_id	integer	NULL	ID of the character's class
	playable_race_id	integer	NULL	ID of the character's race
	gender	text	NULL	Character gender
	faction	text	NULL	Character faction (e.g., ALLIANCE, HORDE)
	active_spec_id	integer	NULL	ID of the currently active specialization
	active_spec_name	text	NULL	Name of the active specialization
	role	text	NULL	Calculated role (Tank, Healer, DPS)
	mount_collection_hash	text	NULL	Hash of mount collection (for potential future use, if feasible)
	toy_collection_hash	text	NULL	Hash of toy collection (for potential future use, if feasible)

	last_modified_timestamp	bigint	NULL	Timestamp from API indicating last character update (if available)
	last_synced_at	timestamp with time zone	NULL	Timestamp of last successful sync by this application
	created_at	timestamp with time zone	NOT NULL, DEFAULT now()	
	updated_at	timestamp with time zone	NOT NULL, DEFAULT now()	
			UNIQUE (name, realm_slug, region)	Business rule: Character uniqueness
			INDEX (blizzard_id, region)	Potential index if lookups by Blizzard ID are common
guilds	id	uuid	PK, DEFAULT gen_random_uuid()	Internal guild ID
	blizzard_id	integer	NOT NULL	Guild ID from Blizzard API (unique within region)
	name	text	NOT NULL	Guild name
	realm_slug	text	NOT NULL	Realm slug
	region	text	NOT NULL	Region
	faction	text	NULL	Guild faction

	last_synced_at	timestamp with time zone	NULL	Timestamp of last successful sync
	created_at	timestamp with time zone	NOT NULL, DEFAULT now()	
	updated_at	timestamp with time zone	NOT NULL, DEFAULT now()	
			UNIQUE (name, realm_slug, region)	Business rule: Guild uniqueness
			INDEX (blizzard_id, region)	Potential index
guild_members	id	uuid	PK, DEFAULT gen_random_uuid()	
	character_id	uuid	FK REFERENCES characters(id) ON DELETE CASCADE, UNIQUE, NOT NULL, INDEXED	Links character to a guild membership; UNIQUE ensures 1 guild max
	guild_id	uuid	FK REFERENCES guilds(id) ON DELETE CASCADE, NOT NULL, INDEXED	Links to the guild
	rank	integer	NOT NULL, CHECK (rank >= 0)	Guild rank (0 = GM)
	is_main	boolean	NOT NULL, DEFAULT false	Indicates if this is the user's main character in this guild

	created_at	timestamp with time zone	NOT NULL, DEFAULT now()	
	updated_at	timestamp with time zone	NOT NULL, DEFAULT now()	
			INDEX (guild_id, user_id)	Index needed for main/alt logic (requires joining through characters)
ranks	id	uuid	PK, DEFAULT gen_random_uuid()	
	guild_id	uuid	FK REFERENCES guilds(id) ON DELETE CASCADE, NOT NULL, INDEXED	Guild these rank permissions apply to
	rank_level	integer	NOT NULL, CHECK (rank_level >= 0)	The numeric rank level (0-N)
	rank_name	text	NULL	Optional name for the rank (from API or custom)
	permissions_id	uuid	FK REFERENCES permissions(id) ON DELETE RESTRICT, NOT NULL	Links to the set of permissions for this rank
	created_at	timestamp with time zone	NOT NULL, DEFAULT now()	
	updated_at	timestamp with time zone	NOT NULL, DEFAULT now()	

			UNIQUE (guild_id, rank_level)	Ensures one permission set per rank level per guild
permissions	id	uuid	PK, DEFAULT gen_random_uuid()	
	can_manage_roster	boolean	NOT NULL, DEFAULT false	Permission to invite/kick/promote/demote members
	can_edit_guild_info	boolean	NOT NULL, DEFAULT false	Permission to edit guild settings/profile
	can_manage_ranks	boolean	NOT NULL, DEFAULT false	Permission to change permissions for ranks
	can_delete_guild	boolean	NOT NULL, DEFAULT false	Permission to delete the guild record from the application
	created_at	timestamp with time zone	NOT NULL, DEFAULT now()	
	updated_at	timestamp with time zone	NOT NULL, DEFAULT now()	

(Note: Additional permissions can be added to the permissions table as needed.)

2.2. Data Types, Constraints, and Relationships

The schema utilizes standard Postgres data types. uuid is chosen for primary keys for global uniqueness, potentially beneficial if parts of the system are ever distributed. bigint or integer are used for IDs originating from Blizzard, matching their likely representation. text is generally preferred over varchar(n) unless a strict length limit is essential. timestamp with time zone is crucial for accurately recording event times

across different regions.

Constraints enforce data integrity:

- **Primary Keys (PK):** Uniquely identify rows within each table.
- **Foreign Keys (FK):** Establish and enforce relationships between tables (e.g., guild_members.character_id links to characters.id). ON DELETE clauses define behavior when referenced rows are deleted (e.g., CASCADE deletes dependent rows, SET NULL removes the link).
- **UNIQUE Constraints:** Ensure uniqueness across specified columns (e.g., users.battlenet_id, characters.(name, realm_slug, region)).
- **NOT NULL Constraints:** Ensure mandatory fields contain values.
- **CHECK Constraints:** Enforce specific value conditions (e.g., guild_members.rank >= 0).

Relationships defined:

- users 1 <-> N characters (A user can have multiple characters)
- characters 1 <-> 0..1 guild_members (A character can be in at most one guild via the membership record)
- guilds 1 <-> N guild_members (A guild has multiple members)
- guilds 1 <-> N ranks (A guild defines multiple rank levels)
- ranks N <-> 1 permissions (Multiple ranks can share the same permission set)

Appropriate indexes are defined on foreign keys and frequently queried columns (battlenet_id, composite keys like (name, realm_slug, region)) to optimize query performance.

2.3. Implementing Business Rules and Data Integrity

Specific business rules are enforced through a combination of schema constraints and application logic:

- **Character Uniqueness:** Guaranteed by the characters.id primary key and the UNIQUE(name, realm_slug, region) constraint.
- **Character Role:** Stored in characters.role. This field is derived data, calculated based on playable_class_id and active_spec_id. The calculation logic (detailed in Section 4.3) should be applied before inserting or updating character records.
- **Character-Guild Link:** The guild_members table acts as the link. The UNIQUE constraint on guild_members.character_id ensures a character can only appear once in this table, thus belonging to at most one guild at any time.
- **Character-User Link:** The nullable characters.user_id foreign key allows a

character to be associated with zero or one user.

- **Character Existence Constraint:** The rule "A character should not exist without a user or a guild linked" presents challenges for pure constraint enforcement. A character might legitimately exist in WoW without a guild, and the application might discover a character via a guild roster sync *before* the associated user logs in (user_id is NULL, guild_id is known via guild_members). If the character then leaves the guild in WoW, the guild_members record would be removed by the next sync, potentially leaving the character temporarily "orphaned" in the database until the user logs in. Deleting such characters immediately based on the strict rule could lead to data loss. A more pragmatic approach involves:
  1. Allowing characters with user_id IS NULL and no corresponding guild_members entry to exist. These represent characters known only through past guild associations.
  2. Optionally, implementing a periodic cleanup job that flags or removes character records that have user_id IS NULL and haven't been associated with a guild_members entry for an extended period (e.g., 30-60 days), assuming they are unlikely to be relevant anymore.
- **Main/Alt Determination:** The logic (detailed in Section 4.3) identifies the main character per user per guild, primarily based on rank. The result is stored in guild_members.is_main. A database trigger or check constraint could potentially enforce the rule that only one is_main = true entry exists per user_id per guild_id, although handling this in application logic when updating ranks or character associations might be simpler.
- **Rank 0 = Guild Master:** This is a convention enforced by the application logic when interpreting the rank field in guild_members and the rank_level in ranks.

Section 3: Blizzard API Endpoint Integration

Effective integration requires identifying the correct Blizzard API endpoints, understanding their authentication requirements, required parameters, and expected response structures.

3.1. Required API Endpoints and Namespaces

The application will need to interact with several endpoints across different Blizzard API categories (OAuth, Profile, Game Data). Each API call requires specifying the correct namespace, which dictates the context (e.g., game patch, data volatility) of the requested data.[8] Namespaces are provided either via the Battlenet-Namespace HTTP header or the ?namespace= query parameter.[9]

Key endpoints include:

- **User Account Info:**
  - Endpoint: /oauth/userinfo [6]
  - API Type: OAuth
  - Auth: Authorization Code Token [3]
  - Scope: openid [3]
  - Namespace: Not applicable (OAuth specific endpoint)
  - Purpose: Retrieve battlenet_id and battletag.
- **User Character List:**
  - Endpoint: /profile/user/wow [4]
  - API Type: Profile
  - Auth: Authorization Code Token [3]
  - Scope: wow.profile [3]
  - Namespace: profile-{region} [4]
  - Purpose: Get an index of characters associated with the logged-in user's account. Critically, this endpoint returns only basic identifiers (ID, name, realm link), not full details.[5] This necessitates subsequent calls for each character during onboarding.
- **Character Profile Summary:**
  - Endpoint: /profile/wow/character/{realmSlug}/{characterName} [4]
  - API Type: Profile
  - Auth: Client Credentials Token *or* Authorization Code Token [2] (Client Credentials preferred for background jobs if possible).
  - Namespace: profile-{region} [4]
  - Purpose: Retrieve detailed character information (level, class, race, gender, faction, active spec, etc.). This call is required for each character identified via /profile/user/wow during onboarding.[5]
- **Character Specializations:**
  - Endpoint: /profile/wow/character/{realmSlug}/{characterName}/specializations [4]
  - API Type: Profile
  - Auth: Client Credentials Token *or* Authorization Code Token [4]
  - Namespace: profile-{region} [4]
  - Purpose: Get details on all specializations for a character, if needed beyond the active one provided in the summary.
- **Character Equipment:**
  - Endpoint: /profile/wow/character/{realmSlug}/{characterName}/equipment [4]
  - API Type: Profile

    - Auth: Client Credentials Token *or* Authorization Code Token [4]
    - Namespace: profile-{region} [4]
    - Purpose: Retrieve equipped items, potentially used for item level calculations (relevant for main/alt heuristics if rank is insufficient).
- **Account Collections (Mounts & Toys):**
    - Endpoints: /profile/user/wow/collections/mounts, /profile/user/wow/collections/toys [4]
    - API Type: Profile
    - Auth: Authorization Code Token [3]
    - Scope: wow.profile [3]
    - Namespace: profile-{region} [4]
    - Purpose: Retrieve lists of mounts and toys owned by the user's account. This data is needed for the collection hash calculation specified in the main/alt determination rule, although its feasibility for unknown users is questionable (see Section 4.3).
- **Guild Information:**
    - Endpoint: /data/wow/guild/{realmSlug}/{nameSlug} [12]
    - API Type: Game Data (path suggests Game Data, but content might relate to Profile)
    - Auth: Client Credentials Token [2]
    - Namespace: Likely profile-{region} for guild-specific data, but potentially static-{region} or dynamic-{region} depending on the specific data fields.[9] Requires verification. Documentation examples [11] suggest profile-{region} might be used even for /data paths when dealing with specific entities like guilds.
    - Purpose: Retrieve basic guild information (name, faction, etc.).
- **Guild Roster:**
    - Endpoint: /data/wow/guild/{realmSlug}/{nameSlug}/roster [12]
    - API Type: Game Data (path suggests Game Data, but content is profile-related)
    - Auth: Client Credentials Token.[2]
    - Namespace: profile-{region}.[4]
    - Purpose: Retrieve the list of guild members, including their character name, realm, level, class, race, and rank. Essential for populating guild_members and discovering characters.
- **Static Game Data (Classes, Races, Realms, etc.):**
    - Endpoints: e.g., /data/wow/playable-class/index, /data/wow/playable-race/index, /data/wow/realm/index [10]
    - API Type: Game Data

    - Auth: Client Credentials Token [2]
    - Namespace: static-{region} [9]
    - Purpose: Fetch reference data (IDs, names, etc.) needed for mapping IDs stored in character/guild data to human-readable names or other properties. This data changes infrequently (typically with game patches) [9] and can be cached locally.

The distinction between Game Data (static/dynamic namespaces) and Profile (profile namespace) APIs is crucial.[8] Static data relates to core game systems, dynamic data changes with world events, and profile data pertains to specific accounts or characters. Profile data updates upon character logout or at regular intervals for guilds.[9] The need to fetch character details individually after getting the index from /profile/user/wow [5] significantly increases the API call volume required for onboarding compared to older API versions, making efficient processing and rate limiting critical. Ambiguities regarding namespaces and authentication for guild endpoints [4] necessitate careful testing during development; the assumption based on recent examples [11] is that profile-{region} and Client Credentials are used for roster data.

3.2. Authentication Requirements per Endpoint

Different endpoints necessitate different authentication methods:

- **Authorization Code Flow Token Required:** Accessing data specific to the logged-in user requires a token obtained via the Authorization Code Flow, typically with the wow.profile scope. This includes:
    - /oauth/userinfo [3]
    - /profile/user/wow (Character List) [3]
    - /profile/user/wow/protected-character/{realm-id}-{character-id} (Protected Character Summary) [3]
    - /profile/user/wow/collections/* (Mounts, Pets, Toys) [3]
- **Client Credentials Flow Token Sufficient:** Accessing general game data or profile data that can be identified without a specific user context (e.g., by realm/name) can often use a token obtained via the Client Credentials flow. This includes:
    - All /data/wow/* endpoints (Static/Dynamic Game Data) [2]
    - /profile/wow/character/{realmSlug}/{characterName} (Public Character Profile) [4]
    - /profile/wow/character/{realmSlug}/{characterName}/specializations [4]
    - /profile/wow/character/{realmSlug}/{characterName}/equipment [4]
    - /data/wow/guild/{realmSlug}/{nameSlug} (Guild Info) [12]

- /data/wow/guild/{realmSlug}/{nameSlug}/roster (Guild Roster) [11]

The Client Credentials flow involves making a POST request to /oauth/token with grant_type=client_credentials and the application's client_id and client_secret (via Basic Auth).[2] This yields an access token representing the application itself, not a specific user. This is highly advantageous for background synchronization jobs (cron jobs), as the application can maintain its own token without depending on potentially expired user tokens to fetch character profiles or guild rosters.[2] However, any operation requiring access to account-level data, like fetching collections for the main/alt hash calculation [3], will *always* require a valid Authorization Code token obtained from the user. This implies that certain data updates (like collection hashes) can only reliably occur when the user is actively interacting with the application or if a robust refresh token mechanism is implemented and maintained.

3.3. Request Parameters and Expected Response Payloads

Successful API interaction requires correctly formatting requests and parsing responses.

- **Request Components:**
  - **Path Parameters:** Many endpoints include variables in the path, such as {region}, {realmSlug}, {characterName}, {nameSlug} (for guilds).[4] These must be substituted with appropriate values.
  - **Query Parameters:** Common query parameters include namespace (mandatory for WoW Game Data/Profile APIs [4]) and locale (optional, for localized strings [4]).
  - **Headers:** The Authorization: Bearer <token> header is required for authenticated endpoints.[6] The Battlenet-Namespace: {namespace} header is an alternative way to specify the namespace.[9]
- **Response Structure:** APIs return JSON documents.[8] Key endpoints provide data needed for the database schema:
  - /profile/user/wow: Returns a structure containing a wow_accounts array, each element having a characters array. Each character object provides basic info like id, name, level, playable_class (with id and key.href), playable_race (with id and key.href), gender, faction, and realm (with id, slug, key.href).[5]
  - /profile/wow/character/{realmSlug}/{characterName}: Returns a rich object with id, name, gender, faction, race (id/name), character_class (id/name), active_spec (id/name), realm (id/slug), level, experience, achievement_points, last_login_timestamp, etc..[4]
  - /data/wow/guild/{realmSlug}/{nameSlug}/roster: Returns a members array. Each member object contains character (with name, id, realm slug, level,

playable_class id, playable_race id) and rank (integer).[12]
    - /profile/user/wow/collections/mounts: Returns a mounts array, each element containing mount (with id, name, key.href) and potentially is_useable, etc..[4] Toys endpoint likely similar.
- **Linked Data:** API responses often use links (_links object or key.href properties) to reference related resources.[8] For instance, a character profile might provide a class ID and a link to the full class details endpoint. The application must parse these structures and decide whether the directly embedded information (like ID and name) is sufficient or if follow-up requests using the provided HREFs are necessary. To manage API usage efficiently, making these additional calls should be minimized, perhaps by pre-fetching and caching static reference data (like class or race details).

Section 4: User Onboarding and Initial Data Population Workflow

This section details the process for fetching and storing a user's WoW data upon their initial registration and login, including the calculation of derived fields.

4.1. Process Flow for New User Onboarding

The onboarding process is triggered immediately after a user successfully links their Battle.net account to the application for the first time (as described in Section 1.4) or when manually initiated by an administrator. Given the potentially large number of API calls involved, especially due to the structure of the /profile/user/wow response [5], this process should be executed asynchronously to avoid blocking the user interface and provide a better user experience. A message queue is recommended for managing the required tasks.

The asynchronous workflow proceeds as follows:

1. **Initiation:** Upon successful user authentication and linking, the backend obtains a valid Authorization Code access_token for the user.
2. **Enqueue Task:** A message is placed onto a task queue (e.g., pgmq) indicating the start of the onboarding process for the specific user_id.
3. **Worker Dequeue:** An asynchronous worker (e.g., a Supabase Edge Function or dedicated service) picks up the onboarding task.
4. **Fetch Character Index:** The worker uses the user's access_token to call /profile/user/wow (Namespace: profile-{region}) to retrieve the index of the user's characters.[5]
5. **Fetch Collection Data:** The worker makes calls to /profile/user/wow/collections/mounts and /profile/user/wow/collections/toys

(Namespace: profile-{region}, requires user access_token) *once* for the user's account.[4] The responses are used to calculate the mount_collection_hash and toy_collection_hash. This step is performed early as the hashes apply to all characters of the user.

6. **Enqueue Character Detail Tasks:** For each character listed in the index from step 4, the worker enqueues a separate task onto the queue (e.g., "fetch_character_details" task type) containing the character's name, realmSlug, region, the user_id, and the pre-calculated collection hashes.
7. **Worker(s) Process Character Details:** One or more workers dequeue the "fetch_character_details" tasks. For each task:
    - Call /profile/wow/character/{realmSlug}/{characterName} (Namespace: profile-{region}, potentially using a Client Credentials token if sufficient, otherwise the user's token) to get detailed character info.[4]
    - Calculate the character's role based on class and active specialization (see 4.3).
    - Perform an INSERT or UPDATE (upsert) operation on the characters table in the database, storing all fetched and calculated data, linking it to the user_id, and storing the collection hashes.
8. **Determine Main/Alt (Post-Character Sync):** After all character detail tasks for the user are likely processed (or via a separate follow-up task), trigger the logic to determine the main/alt status for the user's characters across their guild memberships (see 4.3). Update the is_main flag in the guild_members table accordingly.

This asynchronous, queue-based approach allows the initial user interaction to complete quickly while the potentially lengthy data fetching occurs in the background. It also facilitates error handling and retries for individual character fetches.

4.2. Fetching and Storing Character Data via Blizzard API

Data retrieved from the API endpoints identified in Section 3 needs to be mapped carefully to the database schema defined in Section 2.

- **Field Mapping:** Direct mapping occurs for fields like id (Blizzard ID), name, realm.slug, region, level, playable_class.id, playable_race.id, gender.type, faction.type, active_spec.id, active_spec.name.
- **Data Transformation:** Some fields might require minor transformation (e.g., converting faction/gender types from uppercase strings like ALLIANCE to a consistent format if desired). Timestamps like last_login_timestamp should be stored appropriately.
- **Handling Missing Data:** The API might occasionally return incomplete data or

omit optional fields. The database schema allows NULL for many character attributes (level, class_id, etc.) to accommodate this. Default values should be used where appropriate.

- **Error Handling:** API calls can fail due to various reasons (rate limits, temporary server issues, invalid character data). The asynchronous workers processing character details must implement robust error handling:
  - Catch specific HTTP error codes (e.g., 404 Not Found, 403 Forbidden, 429 Too Many Requests, 5xx Server Errors).
  - Implement retry logic with exponential backoff, especially for transient errors like 429 or 5xx.
  - If retries fail, the task should be moved to a dead-letter queue or logged for investigation, rather than being discarded, to prevent data loss.

4.3. Implementing Derived Data Logic (Roles, Main/Alt Determination)

Business rules require calculating certain fields based on fetched data.

- **Role Calculation:** The characters.role field (Tank, Healer, DPS) is determined by the combination of the character's class and active specialization.
  - **Mapping:** Define a clear mapping. Examples:
    - Warrior (ID: 1) / Protection (Spec ID: 73) -> 'Tank'
    - Paladin (ID: 2) / Holy (Spec ID: 65) -> 'Healer'
    - Priest (ID: 5) / Shadow (Spec ID: 258) -> 'DPS'
    - (Define for all relevant class/spec combinations)
  - **Implementation:** This logic can be implemented either:
    - In the worker process (e.g., TypeScript/Python) before writing to the database. This allows for easier unit testing.
    - As a PL/pgSQL function in the database (e.g., calculate_role(class_id int, spec_id int) RETURNS text), potentially called via a trigger or before insert/update. This keeps the logic close to the data.
  - **Dependencies:** Requires fetching static data for class and specialization IDs and names if the API only provides IDs in some contexts. This static data should be cached.
- **Main/Alt Determination:** The goal is to identify a user's primary ("main") character within each guild they are a member of.
  - **Logic for Known Users:** This logic applies only after a user has logged in and their characters are linked via characters.user_id.
    1. Fetch all characters associated with the user_id.
    2. For each guild the user has characters in (query guild_members joining characters on character_id where user_id matches):

        - Find the character within that guild with the highest rank (lowest numerical rank value in guild_members, where 0 is Guild Master).
        - If there's a tie in rank, additional heuristics could be applied (e.g., highest level, highest item level - requires fetching equipment data [4]), though the primary rule specified is rank. If no further heuristics are defined, one of the tied characters can be arbitrarily chosen or the concept of a single "main" might be relaxed in tie situations.
        - Set guild_members.is_main = true for the identified main character in that specific guild.
        - Ensure all other characters of the same user in the same guild have guild_members.is_main = false.
    - **Logic for Unknown Users (Feasibility Issue):** The original requirement suggested using collection hashes (mount_collection_hash, toy_collection_hash) to group characters belonging to the same *unknown* user (discovered via guild sync but not yet logged into the app). However, fetching account collections requires an Authorization Code token for the specific user account.[3] It is impossible to obtain collection data for a character identified only by name/realm without the owning user logging in. Therefore, **the collection hash comparison method for identifying mains among characters of unknown users is technically infeasible with the current Blizzard API capabilities.**
        - **Revised Approach:** Main/alt determination should be restricted to characters whose owning user is known (i.e., characters.user_id is not NULL). For characters discovered via guild sync where the user is unknown, the is_main flag should remain false or undetermined until the user logs in.
    - **Implementation:** This logic involves querying across users, characters, and guild_members. It can be implemented:
        - In the application layer or worker process after character data is updated.
        - As a PL/pgSQL function (e.g., determine_mains_for_user(target_user_id uuid)) called after onboarding or significant character/guild updates.

Section 5: Recurring Data Synchronization Plan

To keep the application data current, regular synchronization with the Blizzard API is necessary. This involves fetching updated information for users, characters, and guilds already present in the database.

5.1. Identifying Data to Synchronize

The synchronization process needs to refresh various data points:

- **User Data:** Primarily the battletag. This is best updated opportunistically when a user logs into the application, by calling /oauth/userinfo [6] as part of the login flow. Scheduled updates for this are less critical.
- **Character Data:** For characters stored in the characters table (whether linked to a known user or discovered via guilds), key attributes need updating: level, active_spec_id, active_spec_name, potentially item level (derived from equipment), last_modified_timestamp. The primary endpoint for this is /profile/wow/character/{realmSlug}/{characterName}.[4]
- **Guild Data:** Basic guild information like name or faction might change, although less frequently. The /data/wow/guild/{realmSlug}/{nameSlug} endpoint can be used.[12]
- **Guild Rosters:** This is a critical piece of dynamic data. Rosters change as members join, leave, or change rank. The /data/wow/guild/{realmSlug}/{nameSlug}/roster endpoint provides the current member list and their ranks.[12] Synchronization should focus on guilds relevant to the application's users (e.g., guilds containing characters linked to registered users).

5.2. Cron Job Strategy (Scheduling, Frequency)

A scheduled job system is required to trigger the synchronization tasks periodically. Utilizing pg_cron available on Supabase [14] is a viable, integrated approach.

- **Scheduler:** pg_cron allows defining jobs using standard cron syntax directly via SQL.[16]
- **Job Granularity:** Instead of one monolithic sync job, multiple jobs with different frequencies based on data volatility and importance are recommended:
    - **Active Character Sync:** Characters belonging to recently active users or designated as "main" could be updated more frequently (e.g., every 12-24 hours). Schedule: 0 */12 * * * (twice daily) or 0 0 * * * (daily at midnight).
    - **Guild Roster Sync:** Guilds associated with active users could be synced daily or every few days. Schedule: 0 2 * * * (daily at 2 AM). The frequency depends heavily on the number of guilds and the impact on API rate limits.
    - **Less Active Character Sync:** Characters belonging to users who haven't logged in recently could be updated less often (e.g., weekly). Schedule: 0 4 * * 0 (weekly on Sunday at 4 AM).
    - **Static Data Refresh:** Fetching updates for static data like classes, races, etc. (Namespace static-{region}) only needs to happen occasionally, perhaps weekly or even monthly, or triggered manually after major game patches.

Schedule: 0 5 1 * * (monthly on the 1st at 5 AM).
    - **Data Pruning (Optional):** A job to implement the logic discussed in 2.3 for cleaning up old, unassociated character records. Schedule: 0 6 * * 0 (weekly on Sunday at 6 AM).
- **Job Action:** pg_cron jobs should ideally trigger lightweight functions.[15] These functions would not perform the API calls directly but would instead enqueue tasks into the message queue (pgmq) for asynchronous processing by workers (see Section 6 and 7). For example, the "Active Character Sync" job might call a PL/pgSQL function enqueue_active_character_updates(), which queries the database for relevant character IDs and sends corresponding "update_character" messages to pgmq.

5.3. Update Logic and Handling Data Discrepancies

The workers processing synchronization tasks from the queue need efficient logic to update the database.

- **Fetch and Compare:** For each character or guild task, the worker fetches the latest data from the relevant Blizzard API endpoint.
- **Change Detection:** To minimize unnecessary database writes and potential trigger overhead, compare the fetched data with the currently stored data.
    - Use the last_modified_timestamp field from the character API response [4], if available and reliable, comparing it against the stored characters.last_modified_timestamp. Only proceed with a full update if the fetched timestamp is newer.
    - If no reliable timestamp is available, perform a field-by-field comparison of key attributes (level, spec, rank, etc.).
- **Database Updates:** If changes are detected, update the corresponding record(s) in the Postgres database (characters, guilds, guild_members). Update the last_synced_at timestamp for the record.
- **Roster Changes:** When processing a guild roster sync task:
    - Compare the fetched member list with the current members stored in guild_members for that guild_id.
    - **New Members:** If a character from the API roster is not in guild_members, insert a new record. Also, perform an upsert into the characters table to ensure the character record exists (initially with user_id = NULL if the user is unknown).
    - **Departed Members:** If a character in guild_members is *not* in the API roster, delete the corresponding guild_members record. Do *not* delete the characters record itself (per discussion in 2.3).

    - **Rank Changes:** If a character's rank in the API roster differs from the rank in guild_members, update the guild_members record. This might also trigger the main/alt determination logic if the user is known.
- **Error Handling:** Synchronization workers must handle API errors (rate limits, temporary outages) using retries and backoff, similar to the onboarding process. Failed sync tasks should be logged or moved to a dead-letter queue.

Efficient change detection is crucial for managing API rate limits during synchronization. Blindly re-fetching and updating every known character and guild would be highly inefficient and likely exceed hourly quotas [User Query]. Utilizing API-provided timestamps or implementing smart prioritization (e.g., updating more active characters/guilds more frequently) is essential.

Section 6: Rate Limiting Architecture for Blizzard API

The Blizzard API imposes strict rate limits: 100 requests per second and 36,000 requests per hour [User Query]. Exceeding the per-second limit results in 429 Too Many Requests errors, while exceeding the hourly quota can lead to throttling [User Query]. A robust architecture is required to operate within these constraints, especially given the potentially high volume of calls during onboarding and synchronization.

6.1. Strategies for Throttling Requests

Client-side rate limiting must be implemented before any call is made to the Blizzard API. This logic prevents the application from overwhelming the API and incurring errors or throttling.

- **Algorithm:** A common approach is the Token Bucket algorithm. A bucket holds tokens, added at a fixed rate (e.g., 100 tokens per second). Each API request consumes one token. If the bucket is empty, the request must wait. This smooths out bursts and enforces the average rate. Alternatively, a Leaky Bucket algorithm queues requests and processes them at a fixed rate.
- **Implementation Location:** The rate-limiting logic should reside within the asynchronous workers that consume tasks from the message queue (pgmq) and make the actual API calls. Implementing this directly within PL/pgSQL using pg_net would be complex and is not recommended (see Section 7.2).
- **Centralized vs. Distributed:** If multiple worker instances run concurrently (e.g., multiple Edge Functions), simply having each worker limit itself to, say, 10 requests/sec isn't sufficient to guarantee the global limit of 100/sec is respected. A mechanism for coordinating the global rate is needed. Options include:

    - **Centralized Counter:** Using an external fast data store like Redis to maintain a shared counter or token bucket state.
    - **Database Coordination:** Using a Postgres table with atomic operations or advisory locks to manage the shared rate limit state. This can introduce contention and complexity.
    - **Static Partitioning:** Assigning a fraction of the rate limit to each worker (less flexible). A centralized counter (like Redis, if available in the environment) is often the most robust solution for accurately managing shared rate limits across distributed workers.

6.2. Implementing Parallel API Calls Safely

While parallelism can speed up bulk operations like onboarding, it must be carefully controlled to avoid hitting the per-second rate limit.

- **Concurrency Limiting:** The number of simultaneous outgoing requests to the Blizzard API across all workers must be strictly limited. This can be achieved using techniques like:
    - **Worker Pool Size:** Limiting the number of active worker instances processing API call tasks.
    - **Semaphores:** Using a semaphore (potentially managed via the centralized coordination mechanism) to limit concurrent access to the API calling logic. The maximum concurrency should be set conservatively, well below the 100 req/sec limit, to accommodate variations in request latency and prevent bursts from exceeding the limit.
- **Request Distribution:** Workers should aim to spread their requests evenly over time rather than firing them in rapid bursts. Combining concurrency limiting with the per-worker throttling (6.1) helps achieve this.

6.3. Leveraging Message Queues for Rate Control

Using a message queue like pgmq [18] is central to the rate limiting strategy.

- **Buffering:** The queue acts as a buffer between request generation (onboarding, cron jobs) and request execution (API calls). Producers can add tasks to the queue rapidly without worrying about immediate rate limits.
- **Decoupling:** It decouples the task generation logic from the API interaction logic.
- **Controlled Consumption:** Consumers (workers) dequeue messages at a rate dictated by the throttling (6.1) and concurrency (6.2) controls. If workers are configured correctly, the overall rate of API calls originating from the queue consumers will stay within Blizzard's limits.
- **Resilience:** If workers are temporarily stopped or slowed down (e.g., due to

hitting rate limits), tasks remain safely in the queue (pgmq provides persistence [20]) to be processed later.

This architecture effectively isolates the complexity of rate limiting and parallel execution management within the queue consumers (workers). pgmq appears well-suited for this role within a Supabase environment.[20]

6.4. Handling 429 Too Many Requests Errors

Despite careful throttling, 429 errors may still occur due to network latency variations, brief bursts exceeding the limit, or temporary reductions in the allowed rate by Blizzard.

- **Retry Mechanism:** When a worker receives a 429 response, it must not immediately fail the task. Instead, it should implement a retry strategy with exponential backoff: wait for a short period (e.g., 1 second), retry; if it fails again, wait longer (e.g., 2 seconds), retry; then 4 seconds, and so on, up to a maximum number of retries.
- **Dead-Lettering:** If a task consistently fails with 429 (or other persistent errors) after exhausting retries, it should not be simply dropped. It should be moved to a dead-letter queue or mechanism. Within pgmq, this could involve:
  - Using pgmq.archive() to move the message to the archive table for later inspection.[20]
  - Having the worker explicitly insert the failed task details into a separate failed_api_calls table.
- **Monitoring:** Frequent 429 errors are a signal that the rate limiting parameters (throttle rate, concurrency limit) need adjustment. Logging these events and monitoring their frequency is crucial for tuning the system.

Section 7: Feasibility Study: Postgres-Driven Backend on Supabase

A core requirement is to investigate the extent to which backend functionalities (scheduling, API calls, queuing, business logic) can be handled directly within Supabase Postgres using available extensions, potentially minimizing or eliminating the need for a separate application server layer.

7.1. Evaluating pg_cron for Scheduled Tasks

pg_cron is a Postgres extension provided by Supabase for scheduling recurring jobs.[14]

- **Capabilities:** It allows scheduling SQL commands or function calls using standard cron syntax.[16] Job execution details are logged to cron.job_run_details

for monitoring.[15] Management is done via SQL functions like cron.schedule() and cron.unschedule().[15]

- **Advantages:** Integrated within the database, managed via familiar SQL, simple syntax, provides basic monitoring.
- **Limitations:** Supabase documentation suggests a limit of 32 concurrent jobs.[15] Complex or long-running jobs executed directly within the cron.schedule() command can potentially strain database resources (CPU, memory, connections).[15] Debugging requires querying specific cron tables or Postgres logs.[15] Automatic retry logic for failed jobs might need custom implementation within the called function. Requires a recent Postgres version for latest fixes.[15]
- **Assessment:** pg_cron is well-suited for its intended purpose: *scheduling*. It is ideal for triggering the synchronization processes at regular intervals (as described in Section 5.2). However, executing the entire data fetching and processing logic (potentially involving hundreds or thousands of API calls) directly within the cron command is inadvisable due to potential performance impacts and timeouts.[15] The recommended use is to have pg_cron schedule lightweight PL/pgSQL functions that simply enqueue the necessary tasks into a message queue (pgmq).

7.2. Evaluating pg_net for Direct Blizzard API Calls

pg_net is a Supabase extension enabling asynchronous HTTP requests directly from Postgres.[14]

- **Capabilities:** Provides functions like net.http_get() and net.http_post() callable from SQL/PL/pgSQL.[25] Its asynchronous nature means it doesn't block the calling transaction, making it suitable for triggers.[25] Requests are queued internally and executed by a background worker after the calling transaction commits.[25] Secrets can be managed using Supabase Vault.[27]
- **Advantages:** Allows network calls from database logic, potentially simplifying architecture by avoiding external components for simple calls. Asynchronous behavior is beneficial in certain contexts.
- **Limitations:**
  - **Beta Status:** The extension is explicitly marked as beta, implying potential instability or API changes.[25]
  - **Durability:** Uses unlogged tables for its internal request queue and response storage.[25] Data in unlogged tables is lost upon database crash or unclean shutdown, meaning pending requests or recent responses could disappear.
  - **Error Handling/Delivery Guarantees:** Offers basic error reporting in the response table but lacks built-in robust retry mechanisms. Delivery is

described as "at-most-once" [28], meaning requests might be lost if errors occur during processing or if the background worker fails.
    - **Retention:** Response data is kept for only 6 hours by default.[25]
    - **Performance/Rate Limits:** Has an internal processing limit (intended for ~200 req/sec) [25] and managing external rate limits (like Blizzard's 100 req/sec) would require complex custom logic within PL/pgSQL.
    - **Limited Functionality:** Primarily supports GET/POST/DELETE with JSON bodies for POST; lacks support for other methods (PATCH/PUT) or data formats.[25]
- **Assessment:** Given the critical nature of data synchronization with the Blizzard API, the strict rate limits, and the need for reliable error handling and retries, **pg_net is deemed unsuitable and too risky for implementing the core Blizzard API interaction loop.** The beta status, reliance on unlogged tables, and lack of robust delivery guarantees pose significant risks of data loss or inconsistency. While potentially useful for non-critical, low-volume, fire-and-forget tasks like sending a notification webhook [27], it should not be used for fetching character or guild data.

7.3. Evaluating Postgres Message Queues (pgmq, LISTEN/NOTIFY) for API Call Management

A message queue is essential for buffering API calls and managing rate limits. Two Postgres-based options are considered:

- **pgmq:** A Supabase-supported extension providing durable message queues.[18]
    - **Capabilities:** Creates persistent queues (pgmq.create), allows sending messages (pgmq.send), reading messages without deleting (pgmq.read), popping messages (read and delete pgmq.pop), archiving, and deleting.[19] Offers "exactly once" delivery semantics within a configurable visibility timeout.[20] Supports transactional sends, RLS for security, and dashboard monitoring.[20]
    - **Advantages:** Integrated within Postgres, leverages database transactions and durability [20], provides necessary queuing primitives (send, pop, read, archive), avoids external dependencies [20], seems robust and battle-tested.[21]
    - **Limitations:** Requires consumers (workers) to poll the queue for new messages.[21] "Exactly once" delivery relies on consumers being idempotent and correctly managing message lifecycle (delete/archive upon success). Potential performance impact under extreme load needs consideration.
    - **Assessment:** pgmq is the **strongly recommended** Postgres-native solution for managing the queue of Blizzard API call tasks. Its persistence, delivery

guarantees, and integration with Postgres transactions make it ideal for buffering requests generated by onboarding and cron jobs, ensuring reliable processing by rate-limited workers.

- **LISTEN/NOTIFY:** Postgres's built-in asynchronous notification mechanism.[30]
  - **Capabilities:** Allows sessions to LISTEN on named channels and others to send NOTIFY messages with optional payloads.[30]
  - **Advantages:** Built-in, simple pub/sub.
  - **Limitations: Not a persistent queue.** Notifications are lost if no client is actively listening when the NOTIFY occurs.[30] Payload size is limited. Can have issues with connection poolers commonly used in cloud environments like Supabase.[28] Less robust for guaranteed task delivery compared to dedicated queues.[28] Notifications are delivered only upon transaction commit.[30]
  - **Assessment:** LISTEN/NOTIFY is **unsuitable** for reliably queuing critical API call tasks due to its lack of persistence and potential message loss.

7.4. Evaluating PL/pgSQL Functions for Business Logic

PL/pgSQL is Postgres's procedural language, allowing complex logic to be embedded within database functions and triggers.[18]

- **Capabilities:** Can execute SQL, perform conditional logic (IF/ELSE), loops (FOR), error handling (EXCEPTION blocks), and be called from SQL, triggers, or pg_cron jobs.[23]
- **Advantages:** Logic resides close to the data, leveraging database performance and transactionality. Can simplify deployment if it reduces the need for a separate application layer. Supabase fully supports its use.[18]
- **Limitations:** Writing and maintaining very complex logic (e.g., multi-step workflows with external dependencies) can become difficult compared to general-purpose languages like TypeScript or Python. Debugging PL/pgSQL can be less intuitive (though tools like plpgsql_check exist [18]). Poorly written functions or heavy synchronous processing within triggers can cause performance bottlenecks. Library support for external tasks (like complex data validation or transformations) is limited compared to application languages.
- **Assessment:** PL/pgSQL is **well-suited for implementing data-centric business rules and transformations** that are tightly coupled to the database state. Examples include:
  - calculate_role(class_id, spec_id) function.
  - Data validation constraints or simple triggers.
  - Functions to encapsulate the logic for determining main/alt characters based on rank (determine_mains_for_user(user_id)).

- Lightweight functions called by pg_cron to enqueue tasks into pgmq. However, orchestrating complex, multi-step processes involving multiple external API calls, state management across calls, and sophisticated error handling/retry logic (like the full onboarding workflow) is likely better implemented in external workers (e.g., Edge Functions) for improved testability, maintainability, and access to mature HTTP client and utility libraries.

7.5. Synthesis: Viability and Recommended Architecture (Postgres-only vs. Hybrid)

The investigation reveals that while Supabase Postgres offers powerful extensions, aiming for a backend driven *entirely* by these extensions for all functionalities presents significant risks and challenges, primarily due to the limitations of pg_net for critical external API interactions.

Conclusion: A 100% Postgres-driven backend using pg_cron + pg_net for scheduling, queuing, and executing all Blizzard API calls is **not recommended** due to the reliability concerns associated with pg_net.

Recommended Architecture: Hybrid Approach

This approach leverages the strengths of Supabase Postgres and its extensions while mitigating risks by offloading specific tasks to a more suitable environment:

1. **Database (Supabase Postgres):** Core data storage, relationships, constraints, RLS for data visibility, simple data-centric business logic implemented via PL/pgSQL functions (role calculation, main/alt logic, validation).
2. **Scheduling (pg_cron):** Used to trigger background task initiation at scheduled intervals (e.g., daily character sync). Executes lightweight PL/pgSQL functions.
3. **Task Queuing (pgmq):** Used as a durable, transactional buffer for all asynchronous tasks, particularly Blizzard API calls generated by onboarding or cron triggers.
4. **API Interaction & Complex Logic (External Workers):** Implemented as Supabase Edge Functions or a small, dedicated external service (e.g., containerized application, serverless functions).
    - These workers consume tasks from the pgmq queue.
    - They contain robust HTTP client logic (using standard libraries like node-fetch or Python's requests) to interact with the Blizzard API.
    - They implement the necessary rate limiting (throttling, concurrency control, potentially using a shared Redis cache), error handling, and retry logic.

    - They process complex orchestration logic (like the multi-step onboarding flow).
    - They update the Supabase Postgres database with results.
5. **Authentication (Minimal Backend Layer / Edge Functions):** Handles the server-side aspects of the OAuth2 flow (redirect handling, token exchange), secure token storage (encrypting/decrypting tokens stored in Postgres), and token refresh logic.

This hybrid model provides a balanced solution:

- It utilizes Postgres and its extensions (pg_cron, pgmq) effectively for what they do best: scheduling and reliable, transactional queuing close to the data.
- It avoids the risks associated with using the beta pg_net extension for critical, high-volume, rate-limited external API calls.
- It places complex orchestration, external communication, and sophisticated error/rate-limit handling logic in a standard application environment (Edge Functions/service) where mature libraries, better testing frameworks, and more familiar debugging tools are available.
- This results in a more robust, maintainable, and scalable backend system.

Table: Comparative Analysis: Backend Task Handling

Task	**Pure Postgres (Extensions) Approach**	**Hybrid Model Approach**	**Recommendation**
Scheduling	pg_cron executes jobs directly.	pg_cron triggers functions that enqueue tasks into pgmq.	**Hybrid**
	Pros: Integrated. *Cons:* Can strain DB, limited concurrency. [15]	*Pros:* Decouples scheduling from execution, lightweight trigger. *Cons:* Slightly more complex.	
API Calls (Blizzard)	pg_net called from PL/pgSQL functions/cron jobs.	External Workers (Edge Functions/Service)	**Hybrid**

		using standard HTTP libraries consume pgmq tasks.	
	Pros: No external service. *Cons:* **High Risk** (beta, unlogged, poor error handling, rate limit complexity). [25]	*Pros:* Robust, reliable, uses mature libraries, easier rate limit/error handling. *Cons:* Requires external compute.	
Queuing	pg_net internal queue (risky) or pgmq.	pgmq used as the central task queue. [20]	**Hybrid**
	Pros: Integrated (pgmq). *Cons:* pg_net queue unreliable. [25]	*Pros:* Reliable, transactional, integrated. *Cons:* Requires polling consumers. [21]	
Business Logic	Complex PL/pgSQL functions for all logic.	Simple/data-centric logic in PL/pgSQL; complex orchestration in External Workers.	**Hybrid**
	Pros: Close to data. *Cons:* Hard to test/debug complex logic.	*Pros:* Balance of proximity and maintainability/testability. *Cons:* Splits logic location.	

Section 8: Role-Based Access Control (RBAC) System Design

An RBAC system is required to control user actions within the application, particularly concerning guild management, based on their rank within a guild.

8.1. Defining Permissions and Roles (Ranks)

- **Permissions:** Specific actions within the application should be defined as granular permissions. These are represented as boolean flags in the permissions table. Initial examples include:
  - can_manage_roster: Invite, kick, promote, or demote guild members.

    - can_edit_guild_info: Modify guild profile settings within the application.
    - can_manage_ranks: Assign permission sets to different guild ranks.
    - can_delete_guild: Remove the guild's data from the application (a potentially destructive action). (More permissions can be added as application features evolve.)
- **Roles (Ranks):** In this context, roles directly correspond to the numerical World of Warcraft guild ranks, where 0 represents the Guild Master [User Query]. The ranks table links a specific guild_id and rank_level (integer) to a specific set of permissions defined in the permissions table via permissions_id.

8.2. Storing and Managing Rank Permissions

- **Storage:** The ranks table stores the mapping between a guild, a rank level, and its associated permissions set (permissions_id). Each guild will have multiple entries in this table, one for each rank level defined (or relevant) within that guild.
- **Initial Setup:** Upon adding a guild to the system (e.g., when the Guild Master first logs in or during roster sync), default permissions should be established. Typically:
    - Create a permissions record representing full administrative rights (all flags true).
    - Create an entry in the ranks table for guild_id, rank_level = 0, linking to the full admin permissions_id.
    - Create permissions records representing default, restricted permissions for other ranks.
    - Create entries in the ranks table for other relevant rank_levels (e.g., 1 through 9), linking them to the restricted permissions_id.
- **Management Interface:** The application must provide an interface (accessible only to users with the can_manage_ranks permission, initially just the Guild Master) allowing authorized users to:
    - View the permissions assigned to each rank level within their guild.
    - Modify the permissions_id associated with specific rank_levels (below their own rank), effectively changing the capabilities of members at that rank. This might involve selecting from pre-defined permission templates or potentially creating custom permission sets (if the permissions table allows for multiple combinations).
- **Application Admins:** A separate mechanism, independent of guild ranks, is needed for global application administrators. This could be a boolean flag is_admin on the users table or a separate admin role system. Admin checks should bypass the standard guild rank permission checks.

8.3. Implementing Permission Enforcement Logic

Permission checks must occur before any protected action is executed. The preferred location for these checks is within the backend API endpoint handlers that correspond to the protected actions.

The enforcement flow for a user attempting an action on a specific guild:

1. **Identify Context:** Determine the acting user_id (from the session) and the target guild_id (from the request).
2. **Determine User's Rank:** Find the user's character(s) within the target guild. Query guild_members joining characters where characters.user_id matches the acting user and guild_members.guild_id matches the target guild. Identify the relevant character (e.g., the one marked is_main = true for that guild, or the one with the highest rank if no main is designated). Retrieve the rank (numeric level) for that character from the guild_members record.
3. **Lookup Permissions:** Query the ranks table for the entry matching the target guild_id and the user's determined rank_level. Retrieve the associated permissions_id.
4. **Fetch Permission Flags:** Query the permissions table using the retrieved permissions_id to get the specific boolean permission flags (e.g., can_manage_roster).
5. **Check Required Permission:** Check if the specific flag corresponding to the attempted action is true.
6. **Authorize or Deny:** If the flag is true (or if the user is a global admin), allow the action to proceed. Otherwise, deny the action, typically returning an HTTP 403 Forbidden status code.

While Postgres Row Level Security (RLS) is powerful for controlling data visibility (e.g., ensuring users only see their own characters), implementing complex, action-based RBAC tied to guild ranks via RLS can be cumbersome. It would require embedding the multi-table lookup logic (User -> Character -> GuildMember -> Rank -> Permissions) within RLS policy functions, which could impact performance and maintainability. Therefore, performing these checks procedurally within the API logic or dedicated PL/pgSQL permission-checking functions called by the API layer is generally more straightforward for this type of RBAC.

Section 9: Conclusion and Recommendations

This report outlines a detailed architectural plan for the backend system of a World of Warcraft guild management web application, focusing on leveraging Supabase

Postgres while ensuring robustness and scalability.

9.1. Summary of Architectural Decisions

The proposed architecture incorporates the following key decisions:

- **Authentication:** Utilizes Blizzard's Battle.net OAuth2 Authorization Code Flow for secure user login and access delegation, linking accounts via the unique battlenet_id.
- **Database:** Employs Supabase Postgres with a detailed schema designed to model users, characters, guilds, memberships, and rank-based permissions accurately, enforcing business rules through constraints and application logic.
- **Data Synchronization:** Implements asynchronous onboarding and recurring synchronization processes using a combination of pg_cron for scheduling, pgmq for reliable task queuing, and external workers for API interaction.
- **API Interaction:** Manages Blizzard API rate limits through controlled consumption from the pgmq queue by workers implementing throttling, concurrency limits, and robust error handling (retries, backoff).
- **Access Control:** Implements a Role-Based Access Control system tied to WoW guild ranks, managed via database tables and enforced primarily through procedural checks in the backend logic.

9.2. Final Recommendation on the Postgres-Centric Approach

While the goal of maximizing the use of Postgres capabilities via Supabase extensions is valid, a purely Postgres-driven backend (relying on pg_net for critical external API calls) is **not recommended** due to significant reliability and maintainability risks associated with pg_net's current limitations (beta status, durability, error handling).

The **recommended approach is a hybrid architecture**:

- **Leverage Supabase Postgres, pg_cron, and pgmq** for data storage, scheduling, reliable task queuing, and data-centric business logic (PL/pgSQL).
- **Utilize External Workers (e.g., Supabase Edge Functions or a dedicated service)** for handling the complexities of interacting with the external Blizzard API (rate limiting, retries, complex orchestration) and executing intricate business logic.

This hybrid model balances the benefits of an integrated database environment with the robustness and flexibility of standard application development practices for external communication and complex workflows, leading to a more resilient and maintainable system.

9.3. High-Level Next Steps

The following steps are recommended to proceed with development:

1. **Prototype Authentication:** Implement the Battle.net OAuth2 Authorization Code Flow and user linking (/oauth/userinfo).
2. **Implement Core Schema:** Create the primary database tables (users, characters, guilds, guild_members) on Supabase based on Section 2.
3. **Develop Worker & Queue:** Implement the pgmq setup and create a basic worker (e.g., Edge Function) capable of consuming a task, making a single type of Blizzard API call (e.g., /profile/wow/character/...) using a standard HTTP library, implementing basic rate limiting/retry logic, and updating the database.
4. **Implement Onboarding:** Build the asynchronous onboarding process using the queue and worker infrastructure.
5. **Verify API Details:** Conduct practical tests with the Blizzard API to resolve ambiguities regarding specific endpoint authentication requirements and namespaces (particularly for guild data).
6. **Develop Synchronization:** Implement the pg_cron jobs to enqueue sync tasks and enhance workers to handle updates and roster changes.
7. **Implement RBAC:** Build the permission management and enforcement logic.

Sources des citations

1. Battle.net Developer Portal, consulté le avril 28, 2025, https://develop.battle.net/
2. Getting Started | Documentation - Battle.net Developer Portal, consulté le avril 28, 2025, https://develop.battle.net/documentation/guides/getting-started
3. Using OAuth | Documentation - Battle.net Developer Portal, consulté le avril 28, 2025, https://develop.battle.net/documentation/guides/using-oauth
4. World of Warcraft Profile APIs - Battle.net Developer Portal, consulté le avril 28, 2025, https://develop.battle.net/documentation/world-of-warcraft/profile-apis
5. World of Warcraft API Update - Visions of N'zoth - Blizzard Forums, consulté le avril 28, 2025, https://us.forums.blizzard.com/en/blizzard/t/world-of-warcraft-api-update-visions-of-nzoth/3461
6. Authorization Code Flow | Documentation - Battle.net Developer Portal, consulté le avril 28, 2025, https://develop.battle.net/documentation/guides/using-oauth/authorization-code-flow
7. OAuth APIs | Documentation - Battle.net Developer Portal, consulté le avril 28, 2025, https://develop.battle.net/documentation/battle-net/oauth-apis
8. Game Data APIs | Documentation - Battle.net Developer Portal, consulté le avril 28, 2025, https://develop.battle.net/documentation/guides/game-data-apis
9. Namespaces | Documentation - Battle.net Developer Portal, consulté le avril 28, 2025,

https://develop.battle.net/documentation/world-of-warcraft/guides/namespaces
10. World of Warcraft Game Data APIs - Battle.net Developer Portal, consulté le avril 28, 2025, https://develop.battle.net/documentation/world-of-warcraft/game-data-apis
11. [Era] Guild and Guild Roster API no longer works - Page 2 - WoW Classic Bug Report, consulté le avril 28, 2025, https://us.forums.blizzard.com/en/wow/t/era-guild-and-guild-roster-api-no-longer-works/1969144?page=2
12. vmroycroft/wowql: A GraphQL wrapper for the World of Warcraft Game Data APIs and Profile APIs. - GitHub, consulté le avril 28, 2025, https://github.com/vmroycroft/wowql
13. World of Warcraft | Documentation | Postman API Network, consulté le avril 28, 2025, https://www.postman.com/api-evangelist/blizzard/documentation/bn1uyqk/world-of-warcraft
14. Postgres Extensions | Supabase Features, consulté le avril 28, 2025, https://supabase.com/features/postgres-extensions
15. Troubleshooting | pg_cron debugging guide - Supabase Docs, consulté le avril 28, 2025, https://supabase.com/docs/guides/troubleshooting/pgcron-debugging-guide-n1KTaz
16. pg_cron: Job Scheduling - Supabase Docs - Vercel, consulté le avril 28, 2025, https://docs-pgth9qjfy-supabase.vercel.app/docs/guides/database/extensions/pgcron
17. Schedule Recurring Jobs in Postgres - Supabase Cron, consulté le avril 28, 2025, https://supabase.com/modules/cron
18. PostGIS: Geo queries | Supabase Docs, consulté le avril 28, 2025, https://supabase.com/docs/guides/database/extensions/postgis
19. Durable Message Queues with Guaranteed Delivery - Supabase, consulté le avril 28, 2025, https://supabase.com/modules/queues
20. Supabase Queues, consulté le avril 28, 2025, https://supabase.com/blog/supabase-queues
21. PSA: Message Queues with Supabase - Reddit, consulté le avril 28, 2025, https://www.reddit.com/r/Supabase/comments/1fjo22p/psa_message_queues_with_supabase/
22. Cron | Supabase Docs, consulté le avril 28, 2025, https://supabase.com/docs/guides/cron
23. Using a Supabase function with the built-in cron (pg_cron) - Reddit, consulté le avril 28, 2025, https://www.reddit.com/r/Supabase/comments/192e7yv/using_a_supabase_function_with_the_builtin_cron/
24. Postgres Extensions Overview | Supabase Docs, consulté le avril 28, 2025, https://supabase.com/docs/guides/database/extensions
25. pg_net: Async Networking | Supabase Docs, consulté le avril 28, 2025, https://supabase.com/docs/guides/database/extensions/pg_net

26. supabase/pg_net: A PostgreSQL extension that enables asynchronous (non-blocking) HTTP/HTTPS requests with SQL - GitHub, consulté le avril 28, 2025, https://github.com/supabase/pg_net
27. Secure API Calls from DB Functions with Supabase pg_net and Vault - Tomas Pozo, consulté le avril 28, 2025, https://tomaspozo.com/articles/secure-api-calls-supabase-pg-net-vault
28. All the ways to react to changes in Supabase - Sequin Blog, consulté le avril 28, 2025, https://blog.sequinstream.com/all-the-ways-to-react-to-changes-in-supabase/
29. Has anyone ever needed to expand pg_net beyond 200 requests per second? : r/Supabase, consulté le avril 28, 2025, https://www.reddit.com/r/Supabase/comments/1hc4398/has_anyone_ever_needed_to_expand_pg_net_beyond/
30. Documentation: 17: NOTIFY - PostgreSQL, consulté le avril 28, 2025, https://www.postgresql.org/docs/current/sql-notify.html
31. Trusted Language Extensions for Postgres - Supabase, consulté le avril 28, 2025, https://supabase.com/blog/pg-tle