ChatGPT

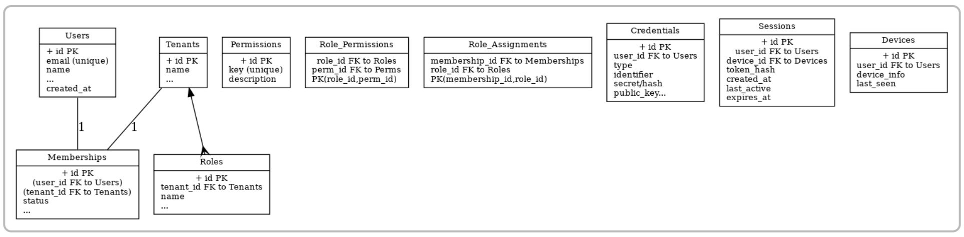
Identity and Tenant Service Schema (PostgreSQL)

Overview

This schema is designed to support a SaaS **Identity and Tenant** service with multi-tenant **RBAC** and multiple authentication methods. It separates core user profiles from authentication credentials, and uses linking tables to handle many-to-many relationships (users to tenants, roles to permissions, etc.). Key design features include:

- **Users & Tenants**: A global **users** table (for user profiles) and a **tenants** table (for organizations). A user can belong to multiple tenants via a **membership** junction table [1] [2]. Each membership represents one user's association with one tenant.
- **Roles & Permissions**: A robust RBAC model with a **roles** table scoped per tenant and a global **permissions** catalog. Roles have many-to-many relations with permissions (through **role_permissions**), and users are assigned roles per tenant (through **role_assignments**). This allows each tenant to define custom roles and permission sets [3].
- **Authentication Credentials**: A unified **credentials** table stores login identities and credentials of all types – passwords, OAuth/OIDC logins, SAML SSO identities, WebAuthn passkeys, TOTP secrets, etc. Each user can have multiple credentials (e.g. one password, several OAuth accounts, multiple WebAuthn keys, etc.). The schema can accommodate new auth methods by adding new credential types or provider entries (future-proof for emerging standards like passkeys).
- **Sessions & Devices**: **sessions** and **devices** tables provide user session management and device tracking on a per-user basis (not tenant-scoped). A user maintains one login session that can access multiple tenants they belong to, and multiple active sessions (e.g. from different devices) can be tracked concurrently. Device tracking improves security (e.g. for session revocation, MFA device recognition).

Below is an **ER diagram** of the main tables and relationships in the schema, illustrating how users, tenants, roles, and credentials interconnect:



Schema Overview – Key tables for users, tenants, roles, permissions, credentials, sessions, and devices (arrows denote foreign key relationships).

Users Table (`users`)

The **users** table holds global user identities and profile info. Each user has a unique identifier and login identity (e.g. email). We keep user data minimal and **normalize** credentials into a separate table for security (no plaintext passwords in this table). For flexibility, we include fields like username and display name, but *passwords are not stored here*. This design allows linking multiple login methods to one user account (e.g. Google OAuth, password, etc.) without duplicating user records [4]. Key points:

- **Email** is unique across all users (serves as primary contact/login for password-based auth).
- **Username** (optional) can be used for display or alternate login, also unique if used.
- Timestamps for account creation and last login are included for auditing and user management.

Table Definition – `users`:

```
CREATE TABLE users (
    id            BIGSERIAL PRIMARY KEY,
    email         VARCHAR(255) NOT NULL UNIQUE,
    username      VARCHAR(100) UNIQUE,
    display_name  VARCHAR(100),
    created_at    TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    last_login_at TIMESTAMPTZ
    -- (Additional profile fields like phone, avatar URL, etc., can be added
as needed)
);
```

Rationale: The `users` table is kept simple (id, email, name, etc.) [5]. Unique constraints on `email` (and `username` if used) prevent duplicate accounts. Storing `last_login_at` helps monitor activity. Sensitive auth data (password hashes, tokens) are stored in the `credentials` table rather than here, following security best practices.

Tenants Table (`tenants`)

The **tenants** table represents organizations or teams in this multi-tenant system. Each tenant has a unique identifier and a name. Additional fields can capture tenant-specific settings or SSO configuration (if each tenant has its own SAML/OIDC provider, etc.), but for now we include basic attributes:

- **Name**: A human-readable name for the tenant (e.g. company or team name). This could be unique or non-unique; if needed, a unique code or subdomain field can be added for tenant identification in URLs or SSO callbacks.
- **created_at**: Timestamp of tenant creation for record-keeping.

Table Definition – `tenants`:

```
CREATE TABLE tenants (
    id          BIGSERIAL PRIMARY KEY,
    name        VARCHAR(150) NOT NULL,
    created_at  TIMESTAMPTZ NOT NULL DEFAULT NOW()
    -- (Additional fields like unique slug, domain, plan_level, etc., can be
added here)
);
```

Rationale: The `tenants` table holds minimal info (id and name). We do not enforce name uniqueness in this schema (different organizations might coincidentally share names), but an optional unique code or subdomain can be introduced if each tenant needs a distinct identifier for routing or SSO. The `created_at` helps with audit and lifecycle management. If a tenant is deleted, all related memberships, roles, etc., will cascade-delete (see foreign keys below), ensuring no orphaned data.

User Memberships (`memberships`)

Memberships link users to tenants, enabling the many-to-many relationship (a user can join multiple tenants, and each tenant has many users) [1]. This table is essentially a join table with possible extra attributes. We enforce one record per user-tenant pair. Key features:

- **user_id, tenant_id** form a composite unique pair, preventing duplicate memberships.
- Foreign keys ensure referential integrity (`user_id` → `users.id`, `tenant_id` → `tenants.id`). Deleting a user or tenant cascades to remove related memberships.
- We include a **status** field to indicate if the membership is active, pending invite, etc., which can help manage invitations or soft deletes of users from tenants.
- Timestamps (`joined_at`) could log when the user joined the tenant.

Table Definition – `memberships`:

```
CREATE TABLE memberships (
    id         BIGSERIAL PRIMARY KEY,
    user_id    BIGINT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    tenant_id  BIGINT NOT NULL REFERENCES tenants(id) ON DELETE CASCADE,
    role_id    BIGINT NULL,  -- (optional default role assignment; see Role
Assignments below)
    status     VARCHAR(50) DEFAULT 'active',
    joined_at  TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    UNIQUE (user_id, tenant_id)
);
```

Rationale: The `memberships` table implements the user-to-tenant **many-to-many** via a normalized structure [2]. The composite unique constraint ensures a user cannot be added to the same tenant twice. We set `ON DELETE CASCADE` so that removing a user or tenant automatically removes the linking membership records. A `status` field is included for extensibility (e.g., mark memberships as "invited", "active", "suspended"). We also show an optional `role_id` here as a convenience for a default or primary role – in practice, detailed role assignments are handled in the `role_assignments` table (below).

Roles and Permissions

Each tenant can define its own **roles** (collections of permissions) to implement RBAC within that tenant. The schema uses a normalized approach with separate **roles** and **permissions** tables, plus mapping tables, which is a standard RBAC pattern [3]:

- **roles**: Defines a role name and scope (which tenant it belongs to). For example, tenants might have roles like "Admin", "Editor", "Viewer", etc. Roles are typically unique per tenant (but different tenants can use the same role names without conflict).
- **permissions**: A global list of fine-grained permissions (actions or privileges in the application). For example, permissions might include `CREATE_PROJECT`, `EDIT_USER`, `VIEW_REPORTS`, etc. This list is defined by the application and can be extended as new features are added.

- **role_permissions**: A join table linking roles to the permissions they include. This allows each role to carry a custom set of permissions. (E.g. an "Editor" role might have `CREATE_CONTENT` and `EDIT_CONTENT` permissions but not administrative permissions.)
- **role_assignments**: A join table linking user memberships to roles (i.e., which roles a user has in a given tenant). This decouples user-tenants from roles, allowing flexible assignment (a user can have multiple roles in one tenant, or no roles if just a basic member). It is effectively the user-to-role mapping within the context of a tenant.

Roles Table (`roles`)

```
CREATE TABLE roles (
    id          BIGSERIAL PRIMARY KEY,
    tenant_id   BIGINT NOT NULL REFERENCES tenants(id) ON DELETE CASCADE,
    name        VARCHAR(100) NOT NULL,
    description TEXT,
    is_default BOOLEAN NOT NULL DEFAULT FALSE,
    UNIQUE (tenant_id, name)
);
```

Rationale: Each `roles` record belongs to a tenant (`tenant_id` as a foreign key). The **unique constraint on (tenant_id, name)** ensures no duplicate role names within the same tenant (e.g. each tenant can have an "Admin" role, but one tenant cannot have two roles both named "Admin"). If a tenant is deleted, all its roles are removed (`ON DELETE CASCADE`). We include a boolean `is_default` as an example extension – it could mark a default role automatically given to new users (this is optional). A `description` helps document what the role means. This design supports **custom role definitions per tenant**, as required.

Permissions Table (`permissions`)

```
CREATE TABLE permissions (
    id           BIGSERIAL PRIMARY KEY,
    key          VARCHAR(100) NOT NULL UNIQUE,
    description TEXT
    -- (Could include module or category info if needed)
);
```

Rationale: The `permissions` table lists all possible permission **keys** that can be granted. We use a unique `key` (e.g. "EDIT_ARTICLE", "DELETE_USER") to identify each permission. This table is not tenant-specific – permissions are defined globally (since the set of actions in the application is global). The many-to-many assignment of these permissions to roles happens via the `role_permissions` table. By keeping permissions normalized, we avoid having sparse boolean columns for each permission on roles [6] (which would be inflexible). Instead, we can add new permissions as rows in this table and update role mappings accordingly.

Role-Permissions Mapping (`role_permissions`)

```
CREATE TABLE role_permissions (
    role_id       BIGINT NOT NULL REFERENCES roles(id) ON DELETE CASCADE,
    permission_id BIGINT NOT NULL REFERENCES permissions(id) ON DELETE
CASCADE,
    PRIMARY KEY (role_id, permission_id)
);
```

Rationale: This join table assigns multiple permissions to each role. The composite primary key `(role_id, permission_id)` ensures no duplicate assignment (each permission can appear at most once per role). For example, if a role "Admin" should have full access, we would insert multiple rows into `role_permissions` linking "Admin"'s role_id to the id of each permission that Admin includes. If a role is deleted, its mappings are cascade-deleted; similarly, if a permission is removed (or deprecated), it will be removed from all roles. This design cleanly supports roles sharing permissions and permissions being reused in multiple roles [3].

Role Assignments (`role_assignments`)

```
CREATE TABLE role_assignments (
    membership_id BIGINT NOT NULL REFERENCES memberships(id) ON DELETE
CASCADE,
    role_id       BIGINT NOT NULL REFERENCES roles(id) ON DELETE CASCADE,
    PRIMARY KEY (membership_id, role_id)
);
```

Rationale: This table assigns **roles to users within a tenant**. Rather than a direct `user_role` table, we tie roles to the `membership_id` (which already couples a specific user with a specific tenant). This ensures that the role being assigned belongs to the same tenant as the membership – a user can only gain roles in tenants they are a member of. The composite PK `(membership_id, role_id)` prevents duplicate role assignments. For example, if user U is a member of tenant T, and T has a role "Editor", adding an entry here gives U the "Editor" role in T. A user can have multiple roles in one tenant by having multiple entries (or none, if perhaps a membership without any special role defaults to a baseline access). We would enforce (via application logic or a trigger) that the `role_id`'s tenant matches the `membership`'s tenant. If a membership is deleted (user leaves tenant) or a role is deleted, the corresponding assignments are automatically removed (`ON DELETE CASCADE`).

Note: In simpler cases where each user has at most one role per tenant, one could instead store a `role_id` on the `memberships` table itself. However, our approach with a separate `role_assignments` table is more flexible and future-proof: it supports multi-role users and easy extension to group-based roles or temporary roles. It also allows auditing and managing role grants separately. (If needed, you could also have a direct `user_permissions` table for one-off permission grants to users [7], but typically we rely on roles for cleaner RBAC.)

Authentication Identities / Credentials (`credentials`)

The **credentials** table (also called *identities*) stores all authentication data (credentials and identity provider links) for users. This design supports **multiple authentication methods** per user: traditional passwords, OAuth/OIDC logins, SAML SSO, WebAuthn/passkeys, TOTP-based 2FA, etc. By normalizing credentials here, the system can easily accommodate new auth mechanisms in the future without altering the core user schema. Key aspects:

- **user_id**: Foreign key linking the credential to a user in the `users` table (with cascade delete – removing a user will remove all their credential records for security).
- **type**: The authentication type or method (e.g. `'password'`, `'oauth'`, `'saml'`, `'webauthn'`, `'totp'`, `'passkey'`, etc.). This can be an `ENUM` or a text/varchar constrained to known types. It indicates how to interpret the credential data.
- **provider** (nullable): For federated logins, this field specifies the provider (e.g. `'google'`, `'github'`, `'azureAD'`, or a SAML IdP name). For non-OAuth credentials (like password, TOTP, WebAuthn), this can be null or an internal identifier.
- **identifier**: A unique identifier for the credential on the provider side. For example, for OAuth/OIDC it might store the external user ID or username given by the provider (such as Google's subject ID); for SAML it could be the NameID; for WebAuthn it can store the **credential ID** of the passkey device (which is a unique binary string, stored here as a text/base64). For a password credential, this might be unused (or could redundantly store the email/username). This field, combined with `provider` when applicable, helps look up the correct user when an external IdP login occurs [4]. We ensure uniqueness of each external `(provider, identifier)` pair so the same external account cannot be linked to two users.
- **secret / hash**: This field (e.g. `secret_hash`) stores credential secrets when applicable. For a password, it holds the password hash (never the plaintext password) plus salt if needed (could be in a separate column or included in the hash). For TOTP, it stores the base32 secret for generating codes. For OAuth/OIDC, this might store a refresh token or nothing (as tokens are typically short-lived and not stored long-term, except refresh tokens if offline access is needed). For SAML, likely not used (no secret, just assertion on login). This field should be encrypted or hashed as appropriate for security.
- **public_key & related fields**: For WebAuthn credentials (passkeys), we need to store the **public key** associated with the credential, and some metadata: e.g. the sign counter (to prevent replay attacks) and potentially the attestation data. In our schema, we include a `public_key` text column. We also include `credential_id` (which we are using as the `identifier` field above) and can have fields like `sign_count` (integer) and **attestation info**. For example, we could store `attestation_type` (as an ENUM of `'none'/'direct'/'indirect' if needed) and the device's **AAGUID** (a 128-bit ID for the authenticator model) [8] [9]. These fields allow full support of WebAuthn passkey authentication data if needed.
- **totp_active / mfa**: We can include a boolean or status to indicate if a TOTP is active for the user (or simply check if a type='totp' record exists for them). Similarly, backup codes for 2FA could be stored in a separate table or as part of this record's data.
- **metadata**: A JSON or text field (optional) can store miscellaneous data for certain providers (for example, OAuth access token metadata, device information for WebAuthn like the user's device name, or any other attributes not covered by fixed columns). This provides future extensibility – if a new auth method requires storing additional fields, we can use this JSON column or add new columns.
- **timestamps**: `created_at` for when the credential was added, `last_used_at` for last authentication using this credential. This is useful for security monitoring (e.g. to detect stale or unused login methods).

Table Definition – `credentials`:

```
CREATE TYPE auth_type AS ENUM
('password','oauth','oidc','saml','webauthn','totp','passkey');
```

```
CREATE TABLE credentials (
    id              BIGSERIAL PRIMARY KEY,
    user_id         BIGINT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    type            auth_type NOT NULL,
    provider        VARCHAR(50),    -- e.g. 'google', 'github', 'okta'; NULL
for password, totp, etc.
    identifier      TEXT,           -- external user ID or credential ID
(unique per provider or credential)
    secret_hash     TEXT,           -- password hash OR TOTP secret, etc.
(sensitive data hashed/encoded)
    public_key      TEXT,
-- for WebAuthn: the credential public key (PEM/Base64)
    sign_count      INTEGER,        -- for WebAuthn: signature counter to
prevent replay
    attestation_aaguid CHAR(36),    -- for WebAuthn: device AAGUID (UUID
format) if needed
    created_at      TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    last_used_at    TIMESTAMPTZ,
    UNIQUE (provider, identifier)
);
```

Rationale: This single table handles all credential types in a normalized way. We use an `auth_type` ENUM for clarity and data integrity on `type`. The combination of `provider` and `identifier` is constrained to be unique to avoid linking the same external identity to multiple users. For example, no two users can have the same Google account ID in their credentials [4]. (For local credentials like passwords or TOTP, `provider` might be null; the unique constraint would then effectively apply just on `identifier` for those where it's set. Alternatively, a partial unique index could be used to enforce uniqueness of credential identifiers per type/provider as needed.)

The `secret_hash` stores sensitive secrets **hashed or encrypted**. Passwords should be stored as a salted hash (e.g. Bcrypt/Argon2) – we never store plaintext. TOTP secrets should be stored in a secure format as well (possibly encrypted since they need to be re-used for code generation). We include fields to fully support WebAuthn **passkeys**: `public_key`, `sign_count`, `attestation_aaguid` etc., following recommended storage guidelines [8]. For example, when a user registers a WebAuthn credential, we create a `credentials` row with `type='webauthn'`, store the device's public key and a unique `credential_id` (here in `identifier`), possibly an attestation GUID and initial `sign_count`. On login, we update `last_used_at` and the `sign_count`.

By adjusting the `auth_type` ENUM or adding new provider strings, this table can accommodate future authentication methods without altering the database structure – making it **future-proof** for emerging standards. (If a completely new set of fields is needed, one can either extend this table's columns or use the `metadata` JSON to store them as a stopgap.) This design cleanly separates authentication factors from the core user profile, allowing multiple login options to map to the same user account.

Sessions Table (`sessions`)

The **sessions** table tracks active login sessions or tokens issued to users. This is useful for supporting logout (session invalidation), tracking concurrent logins, and device-specific session management. Each session is not tied to a particular tenant – once a user is authenticated, their session can be used to access any tenant they belong to (authorization to specific tenant resources is checked via roles, not separate sessions). Key fields:

- **session identifier**: We use a primary key `id` (which can be a random UUID or bigserial) and/or a `token` field. In many implementations, sessions are identified by a secure random token (e.g. a JWT or a random string in a browser cookie). We might store a hash of that token for security (so the actual token cannot be stolen from DB). In this schema, we include `id` as a surrogate PK and could also include a `token_hash` if needed.
- **user_id**: Links to the `users` table (foreign key). This is the user who owns the session. This allows querying all active sessions for a user (for a "log out all sessions" feature, for example).
- **device_id**: Links to the `devices` table (if we have identified the device; can be NULL if not applicable). This ties the session to a known device record.
- **created_at, last_active_at, expires_at**: Timestamps for when the session was created, last seen, and when it will expire. `expires_at` helps implement session timeouts or persistent login expiration. `last_active_at` can be updated periodically to track activity (for idle session timeout or display to the user).
- **ip_address**: The IP address from which the session was initiated (IPv4/IPv6, we can use PostgreSQL `INET` type for this). Useful for security auditing (e.g. alerting user of new login from unfamiliar IP).
- **user_agent**: The User-Agent string or device info of the client. This provides context on the device/browser used for the session. Storing this can help show the user a list of "logged in devices" and detect anomalies.

Table Definition – `sessions`:

```
CREATE TABLE sessions (
    id             BIGSERIAL PRIMARY KEY,
    user_id        BIGINT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    device_id      BIGINT REFERENCES devices(id) ON DELETE CASCADE,
    token_hash     TEXT,        -- hash of session token or refresh token (if
using token-based auth)
    ip_address     INET,
    user_agent     TEXT,
    created_at     TIMESTAMPTZ NOT NULL DEFAULT NOW(),
    last_active_at TIMESTAMPTZ,
    expires_at     TIMESTAMPTZ
);
```

Rationale: The `sessions` table is not scoped by tenant – a single login session allows the user to act in any of their tenants, with authorization checks done per request. We relate sessions to users (one user to many sessions). We also link to `devices` (one device can have multiple sessions, e.g. if the user logged in multiple times on the same device or different browsers on that device). We set `ON DELETE CASCADE` on `user_id` and `device_id` so that if a user is deleted or a device record is purged, related sessions are removed for cleanliness. (Cascading on device deletion also helps if we implement a "logout from this device" which might remove the device and its sessions.)

Including `ip_address` and `user_agent` allows us to display or analyze session provenance (e.g. "Logged in from Chrome on Windows, IP 203.0.113.X"). The `token_hash` column is for storing an opaque session token in a secure manner – for instance, if using JWTs, we might not store them at all (if stateless) or store only a refresh token's hash for validation on logout. This design is flexible: it can support both stateful sessions (by storing a session record for each login) or stateless JWT with refresh tokens. If not needed, the table can simply track active refresh tokens or be used for manual session invalidation lists.

Note: *Audit Logging:* The question scope excludes audit logging, but session records are a form of security audit. For full audit trails of user actions, a separate audit log table would record events (with user_id, action, timestamp, etc.) – that is outside this schema but can be added without affecting the core design.

Devices Table (`devices`)

The **devices** table records trusted or known devices used by users to authenticate. This works in tandem with sessions to provide device-specific tracking – for example, a user can review a list of devices where their account is logged in and revoke ones they don't recognize. It's also useful for implementing **multi-factor authentication** device enrollment (e.g. storing a WebAuthn authenticator or remembering a device that was used for login). Fields include:

- **user_id**: The user who owns/uses the device. (This could also be many-to-many in theory, but typically a device is personal to one user account in context of identity management).
- **device_info**: A descriptive name or fingerprint for the device. This might include the device model, OS, and browser (or we can store user_agent here for simplicity). Alternatively, we might have separate fields for device type, OS, browser, etc., but a single text or JSON field can capture a device fingerprint. For example: "Chrome on Windows 10" or an internal fingerprint string.
- **last_seen_at**: Timestamp of last activity from this device (updated whenever the user uses this device to authenticate or perform an action).
- **registered_at**: When the device was first seen or registered. If this device is tied to a WebAuthn credential, it might coincide with credential creation. If it's just from sessions, it could be first login time.
- **is_trusted**: A boolean that could mark whether the user has labeled this device as "trusted" (to potentially bypass some MFA prompts). Not necessary for core function but commonly useful in an identity system.

Table Definition – `devices`:

```
CREATE TABLE devices (
    id           BIGSERIAL PRIMARY KEY,
    user_id      BIGINT NOT NULL REFERENCES users(id) ON DELETE CASCADE,
    device_info  TEXT,
    last_seen_at TIMESTAMPTZ,
    registered_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

Rationale: Devices are linked to users (with cascade delete if user is removed, to purge associated devices). We keep the schema simple with a generic `device_info` field – this could be expanded or structured (for example, storing a device fingerprint hash). We record timestamps to track usage. Whenever a session from a device is created or used, we would update `last_seen_at`. The `devices` table is updated independently of sessions so that even after a session expires or is deleted, we might retain a record that a user had a device registered (until the user or system revokes it). For

WebAuthn credentials specifically, each credential could correspond to a "device" entry with info on that authenticator (though often the `credentials` table itself holds enough info for WebAuthn). We could choose to integrate those (for example, store the `credential_id` or nickname of a security key in `device_info`). The separation of devices and credentials is flexible – not every device is a credential (a device might be a browser that uses a password), and not every credential is a physical device (TOTP isn't a device per se).

In summary, the **sessions** and **devices** tables together enable per-user session management without tenant scoping, as required. A user logs in once (creating a session tied to their user account), and can then select or access multiple tenants they belong to, with authorization checks ensuring they only act within roles granted [1]. Device tracking enhances security by allowing recognition of known devices and invalidation of sessions on lost/stolen devices.

Relationships and Constraints Summary

To ensure data integrity, we establish the following **key relationships and constraints** among the tables (many already noted above):

- **Foreign Keys & Cascades:**
- `memberships.user_id → users.id` (CASCADE DELETE on user removal)
- `memberships.tenant_id → tenants.id` (CASCADE DELETE on tenant removal)
- `roles.tenant_id → tenants.id` (CASCADE on tenant removal, removing that tenant's roles)
- `role_permissions.role_id → roles.id` (CASCADE on role removal)
- `role_permissions.permission_id → permissions.id` (CASCADE on permission removal)
- `role_assignments.membership_id → memberships.id` (CASCADE on membership deletion, e.g. user left tenant)
- `role_assignments.role_id → roles.id` (CASCADE on role deletion)
- `credentials.user_id → users.id` (CASCADE on user deletion, removing all auth credentials)
- `sessions.user_id → users.id` (CASCADE on user deletion, end all sessions)
- `sessions.device_id → devices.id` (CASCADE on device deletion, end sessions from that device)
- `devices.user_id → users.id` (CASCADE on user deletion, remove associated devices)

These cascades align with expected behavior (e.g. deleting a tenant cleanly removes all subordinate data like memberships, roles, etc. [2]). In production, one might use soft-deletes or disable cascades for certain cases to avoid accidental data loss, but here we assume complete removal propagates appropriately.

- **Unique Constraints:**
- `users.email` (unique per user) – ensures a single identity per email address.
- `tenants.name` is *not* globally unique here, but you may add a unique constraint or separate unique field if each tenant needs a unique identifier (e.g. a code or domain).
- `memberships.user_id, memberships.tenant_id` (composite unique) – a user cannot have two membership entries in the same tenant.
- `roles.tenant_id, roles.name` (composite unique) – no duplicate role names in one tenant.
- `permissions.key` (unique) – each permission defined once globally.
- `role_permissions.role_id, role_permissions.permission_id` (PK composite unique) – no duplicate permission in a role.

- `role_assignments.membership_id, role_assignments.role_id` (PK composite unique) – no duplicate role assignment for the same user in a tenant.
- `credentials.provider, credentials.identifier` (composite unique) – an external identity (provider + id) maps to only one user. E.g. one Google account → one user. Also, each WebAuthn `credential_id` (stored in `identifier`) is unique. *(If needed, additional partial uniques could be enforced, such as ensuring a user has only one* `'password'` *credential, etc. This can be handled by application logic or by a unique index on* `(user_id, type)` *where type is constrained to certain values.)*

- **Indexes:** By default, primary keys and unique constraints create indexes (e.g. on user email, permission key). We should also ensure indexes on foreign key fields that are frequently queried: e.g. an index on `memberships.user_id` (to quickly find all tenants for a user), on `memberships.tenant_id` (all users in a tenant), on `role_assignments.membership_id` (roles for a given user+tenant), on `credentials.user_id`, etc. This ensures lookups and join performance are optimized. In a production setup, additional indexes might be added based on query patterns (for instance, an index on `sessions.user_id` if we often query sessions per user).

Extensibility and Future Considerations

This schema is designed to be **extensible** for future needs:

- **Additional Auth Factors:** New multi-factor auth methods (e.g. SMS OTP, email magic links, recovery codes) can be integrated by adding new `auth_type` values and corresponding fields or by using the `credentials` table's flexible structure. For example, adding an `sms` type with a phone number identifier, or storing one-time password seeds. The current design already supports TOTP and WebAuthn which are common 2FA methods. The `credentials` table can also store **passkeys** (which are essentially WebAuthn credentials synced via platform authenticators) by marking them as `type='webauthn'`/`passkey` and storing the same public key info – passkeys are just WebAuthn credentials with potential cloud backup, so our model covers them.
- **SSO Integration:** For enterprise SSO (SAML or OIDC), the `credentials` table can store federated identities. Additionally, one could introduce a table for **identity providers** (tenant-specific IdP configurations), which might look like: `id, tenant_id, type (SAML/OIDC), metadata (XML metadata or OIDC client info)`. The user's SSO login would create a credential linked to that tenant's provider (e.g. `provider='tenant42-okta', identifier='<Okta UID>'`). Our schema can accommodate this by either encoding the provider name to include tenant context or by adding a `tenant_id` column to `credentials` for credentials that are tenant-scoped. However, since a user might use the same SSO across tenants or different ones, it may be cleaner to keep credentials global and handle tenant-specific logic in application code or via a mapping table. The important point is that no changes to core tables are needed to support SSO – we can store the necessary IDs and trust relationships as additional data.
- **Permission Policies and Hierarchies:** The RBAC model here can be expanded to support more complex authorization schemes. For instance, if in the future you implement **attribute-based access control (ABAC)** or need per-tenant policy rules, you could add a table for policy rules referencing roles or permissions. If role hierarchies or inheritance are needed, a self-referential table linking roles as parent/child could be introduced. Because roles and permissions are normalized, such changes are feasible without redesigning the entire schema.

- **Groups or Teams within Tenants:** Currently, every user is directly linked to a tenant and gets roles. If tenants later require sub-grouping of users (e.g. teams or projects within a tenant each with their own roles), one could add a `groups` table (with tenant_id) and a membership of users to groups, then assign roles to groups. This would complement the user-role assignment. Our schema doesn't include this out-of-box, but it wouldn't conflict with it – it would be an additive feature (for example, adding a `group_roles` table similar to how `role_assignments` works but for groups).
- **Audit and Logs:** Though not included by requirement, an audit log table could record changes and important actions (who did what, when). This would typically reference `user_id` (and possibly `tenant_id`) and store an event description or type. It can be related to the above schema via foreign keys to `users` and `tenants` if needed. Adding such a table does not require modifying existing tables – it just consumes their keys.

Overall, this schema provides a **normalized, flexible foundation** for identity and access management in a multi-tenant SaaS. It cleanly separates concerns: user info, authentication methods, and authorization (roles/permissions) are all modular. This makes it easier to maintain and extend. For example, revoking a user's access to a tenant is as simple as deleting their membership (which automatically removes role assignments) – no duplication of user entries per tenant. Likewise, adding a new login method for a user is just inserting a new credentials row. The structure avoids data anomalies (each relationship is properly constrained) and is optimized for typical queries (with appropriate indexing and clear join paths). By following established patterns for RBAC [3] and account linking, the design is **production-grade** and can evolve with future security requirements.

---

[1] Implement Multi-Tenancy Role-Based Access Control (RBAC) in MongoDB
https://www.permit.io/blog/implement-multi-tenancy-rbac-in-mongodb

[2] [6] postgresql - What would a "properly designed" schema for roles and permissions look like? - Database Administrators Stack Exchange
https://dba.stackexchange.com/questions/185504/what-would-a-properly-designed-schema-for-roles-and-permissions-look-like

[3] Should I create a separate table for each permission when implementing access control? : r/SQL
https://www.reddit.com/r/SQL/comments/oauu5b/should_i_create_a_separate_table_for_each/

[4] Database structure for multiple authentication sources of users in a web app - Stack Overflow
https://stackoverflow.com/questions/48249206/database-structure-for-multiple-authentication-sources-of-users-in-a-web-app

[5] [8] [9] Passkey & WebAuthn Database: Guide for Columns & Fields
https://www.corbado.com/blog/passkey-webauthn-database-guide

[7] mysql - Best table structure for users with different roles - Stack Overflow
https://stackoverflow.com/questions/13660154/best-table-structure-for-users-with-different-roles