ER Diagram - Detailed Technical Explanation

Central Multi-Application Database Architecture

📐 Diagram Overview

The Entity-Relationship (ER) diagram represents the complete database structure showing:
- **12 core tables** (entities)
- **Relationships** between tables (one-to-many, many-to-many)
- **Primary Keys** (PK) - unique identifiers
- **Foreign Keys** (FK) - references to other tables
- **Unique Keys** (UK) - fields that must be unique

🎯 Core Design Principles

1. **Normalization**

Data is organized to minimize redundancy and maintain integrity:
- Each piece of information stored only once
- Changes propagate automatically through relationships
- No duplicate or conflicting data

2. **Referential Integrity**

Foreign keys ensure data consistency:
- Can't create a session for non-existent user
- Can't assign a role that doesn't exist
- Automatic cleanup when records deleted (CASCADE)

3. **Flexibility**

JSONB columns allow schema extension:
- Add new fields without ALTER TABLE

- Store semi-structured data
- Query JSON fields efficiently

🔍 Detailed Table Relationships

🟦 GROUP 1: Authentication & User Management

USERS (Central Hub)

```
users
├─ user_id [PK] - UUID unique identifier
├─ email [UK] - Must be unique, used for login
├─ username [UK] - Must be unique
├─ password_hash - Encrypted password (bcrypt/Argon2)
├─ first_name, last_name, phone_number - Profile info
├─ is_active - Can this user log in?
├─ is_verified - Has email been verified?
├─ metadata [JSONB] - Custom fields (department, employee_id, etc.)
└─ timestamps - created_at, updated_at, last_login_at
```

Why it's designed this way:
- **UUID primary key** - Globally unique, secure (can't guess sequential IDs)
- **Unique email/username** - Prevents duplicate accounts
- **metadata JSONB** - Add any field without schema changes
- **is_active flag** - Soft delete (deactivate instead of delete)

Relationships FROM users:
- `users ||--o{ user_sessions` - One user can have many active sessions (login from multiple devices/apps)
- `users ||--o{ user_roles` - One user can have many roles (Admin in App A, User in App B)
- `users ||--o{ user_app_preferences` - One user has preferences for each app
- `users ||--o{ audit_logs` - One user generates many audit log entries
- `users ||--o{ password_reset_tokens` - One user can request multiple password resets (over time)

Example Scenario:

```
John (user_id: 123)
├─ Active session in App A (mobile)
├─ Active session in App B (desktop)
├─ Has "Admin" role in App A
├─ Has "User" role in App B
├─ Preferences stored for both apps
└─ All his actions logged in audit_logs
```

USER_SESSIONS (Active Login Tracking)

```
user_sessions
├─ session_id [PK] - Unique session identifier
├─ user_id [FK → users] - Which user?
├─ app_id [FK → applications] - Which app?
├─ session_token [UK] - JWT or session identifier
├─ refresh_token - For token renewal
├─ ip_address - Security tracking
├─ user_agent - Device/browser info
├─ expires_at - When does session expire?
└─ last_activity_at - Last active timestamp
```

Why it’s designed this way:
- **Composite relationship** (user_id + app_id) - Track sessions per user per app
- **Unique session_token** - Prevents session hijacking
- **expires_at** - Automatic security (sessions don’t last forever)
- **ip_address & user_agent** - Security audit trail

Example Data:

session_id	user_id	app_id	session_token	expires_at
sess-1	john-123	app-a	eyJhbGc...	2025-11-06 10:00
sess-2	john-123	app-b	eyJhbGd...	2025-11-06 11:00
sess-3	john-456	app-a	eyJhbGe...	2025-11-06 09:30

Single Sign-On Flow:

```
1. User logs into App A
   → Create session: (user_id=john, app_id=app-a, token=xxx)

2. User opens App B (same browser)
   → Check if user_id=john has valid session for any app
   → Yes! Create new session: (user_id=john, app_id=app-b, token=yyy)
   → No password needed!

3. User logs out from App A
   → Delete session for app-a
   → Session for app-b remains active
```

PASSWORD_RESET_TOKENS (Password Recovery)

```
password_reset_tokens
├─ token_id [PK]
├─ user_id [FK → users] - Which user requested reset?
├─ token [UK] - One-time use token (emailed to user)
├─ expires_at - Usually 1 hour
├─ is_used - Prevent reuse
└─ created_at
```

Security Features:
- Token expires quickly (1 hour)
- Single-use only (is_used flag)
- Unique token prevents guessing

Password Reset Flow:

```
1. User clicks "Forgot Password"
   → Generate random token
   → INSERT into password_reset_tokens
   → Email token link to user

2. User clicks email link
```

```
  → Check token exists and not expired
  → Check is_used = FALSE
  → Allow password change
  → Set is_used = TRUE

3. Token expires or is used
  → Can't be reused
  → User must request new token
```

🟩 GROUP 2: Access Control System

ROLES (Job Functions)

```
roles
├─ role_id [PK]
├─ role_name [UK] - admin, user, moderator, etc.
├─ description - Human-readable explanation
├─ is_system_role - Built-in vs custom roles
├─ permissions_config [JSONB] - Flexible config
└─ timestamps
```

Example Roles:

role_id	role_name	description
r1	super_admin	Full system access
r2	admin	App administrator
r3	user	Standard user
r4	guest	View-only access
r5	content_manager	Can manage content only

Why separate roles from permissions?

- **Flexibility** - Change permissions without touching users
- **Scalability** - Add new roles easily
- **Maintenance** - Update role once, affects all users with that role

---

PERMISSIONS (Granular Actions)

```
permissions
├─ permission_id [PK]
├─ permission_name [UK] - "users.create", "documents.delete"
├─ resource_type - What (users, documents, reports)
├─ action - How (create, read, update, delete)
└─ description
```

Naming Convention: `{resource}.{action}`

Example Permissions:

permission_name	resource_type	action	description
users.create	user	create	Can create new users
users.read	user	read	Can view user profiles
users.update	user	update	Can edit user info
users.delete	user	delete	Can delete users
documents.create	document	create	Can upload documents
documents.read	document	read	Can view documents
reports.generate	report	generate	Can create reports

Why granular permissions?
- **Security** - Give minimum necessary access
- **Flexibility** - Mix and match as needed
- **Audit** - Know exactly what users can do

USER_ROLES (Assign Roles to Users)

```
user_roles
├─ user_role_id [PK]
├─ user_id [FK → users] - Which user?
├─ role_id [FK → roles] - Which role?
├─ assigned_at - When assigned?
├─ expires_at - Optional expiration (temporary admin)
└─ assigned_by [FK → users] - Who assigned it?
```

Many-to-Many Relationship: Users ↔ Roles
- One user can have multiple roles
- One role can be assigned to multiple users

Example:

```
John (user_id: 123)
├─ Role: Admin (in App A) - expires: never
└─ Role: User (in App B) - expires: never

Mary (user_id: 456)
├─ Role: Admin (in App A) - expires: 2025-12-31 (temporary)
└─ Role: Content Manager (in App B) - expires: never
```

Temporary Access Example:

```
-- Make John temporary admin for 30 daysINSERT INTO user_roles (user_id, role_id, expires_at)
VALUES (
  'john-123',
  'admin-role-id',
  NOW() + INTERVAL '30 days');
-- After 30 days, role automatically invalid-- Application checks: expires_at > NOW()
```

ROLE_PERMISSIONS (Assign Permissions to Roles)

```
role_permissions
├─ role_permission_id [PK]
├─ role_id [FK → roles] - Which role?
├─ permission_id [FK → permissions] - Which permission?
├─ constraints [JSONB] - Additional rules (time-based, IP-based)
└─ created_at
```

Many-to-Many Relationship: Roles ↔ Permissions

Example Configuration:

```
Admin Role
├─ users.create
├─ users.read
├─ users.update
├─ users.delete
├─ documents.create
├─ documents.read
├─ documents.update
└─ documents.delete

User Role
├─ documents.read
└─ documents.create (with constraints)

Guest Role
└─ documents.read (view only)
```

Advanced: Constraints Example

```
{
  "time_restriction": {
    "allowed_hours": "09:00-17:00",
    "timezone": "UTC"
  },
  "ip_whitelist": ["192.168.1.0/24"],
  "max_actions_per_day": 100
}
```

🟨 GROUP 3: Application Management

APPLICATIONS (App Registry)

```
applications
├─ app_id [PK]
├─ app_name [UK] - "Sales Dashboard", "Inventory System"
├─ app_key [UK] - Public identifier (like API key)
├─ app_secret - Private key for authentication
├─ description
├─ config [JSONB] - App-specific settings
├─ is_active - Can this app connect?
└─ timestamps
```

Example Apps:

app_name	app_key	description
Sales Dashboard	sales-app-2024	Customer sales tracking
Inventory Manager	inv-app-2024	Stock management system
HR Portal	hr-app-2024	Employee management

config JSONB Examples:

```
{
  "api_rate_limit": 1000,
  "allowed_origins": [
    "https://app-a.com",
    "https://app-b.com"
  ],
  "features_enabled": [
    "sso",
    "audit",
    "api_access"
  ],
  "ui_theme": "dark",
  "max_file_upload_size": 10485760
}
```

Why register apps?
- **Security** - Only registered apps can connect
- **Tracking** - Know which app user is using
- **Configuration** - Different settings per app
- **Analytics** - Usage statistics per app

USER_APP_PREFERENCES (Per-App User Settings)

```
user_app_preferences
├── preference_id [PK]
├── user_id [FK → users] - Which user?
├── app_id [FK → applications] - Which app?
├── preferences [JSONB] - UI settings, theme, etc.
├── app_specific_data [JSONB] - App can store anything here
├── last_accessed_at
└── timestamps
```

Why separate preferences per app?
- Different apps need different settings
- User may prefer dark mode in App A, light mode in App B
- Apps can store custom data without new tables

Example Data:

```
// John's preferences for Sales Dashboard
{
  "preferences": {
    "theme": "dark",
    "language": "en",
    "notifications": true,
    "dashboard_layout": "compact"
  },
  "app_specific_data": {
    "favorite_reports": [
      "sales-q4",
      "revenue-trend"
    ],
    "saved_filters": {
      "region": "North America",
      "date_range": "last_30_days"
    },
    "pinned_items": [1, 5, 12]
  }
}
```

APP_RESOURCES (App-Specific Content)

```
app_resources
 ├─ resource_id [PK]
 ├─ app_id [FK → applications] - Which app owns this?
 ├─ resource_type - document, report, file, video, etc.
 ├─ resource_name - Human-readable name
 ├─ resource_data [JSONB] - Flexible storage
 ├─ access_rules [JSONB] - Who can access?
 └─ timestamps
```

Purpose: Store app-specific data without creating new tables

Examples:

Document Resource:

```
{
  "resource_type": "document",
  "resource_name": "Q4 Sales Report",
  "resource_data": {
    "url": "https://s3.../report.pdf",
    "size_bytes": 1048576,
    "page_count": 15,
    "author": "john@company.com",
    "created_date": "2025-11-01"
  },
  "access_rules": {
    "roles": [
      "admin",
      "sales_manager"
    ],
    "specific_users": [
      "john-123",
      "mary-456"
    ],
    "expiration": "2026-01-01"
  }
}
```

Video Resource:

```
{
  "resource_type": "video",
  "resource_name": "Training: New Features",
  "resource_data": {
    "url": "https://vimeo.com/...",
    "duration_seconds": 300,
    "resolution": "1080p",
    "subtitles": [
      "en",
      "es",
      "fr"
    ]
  },
  "access_rules": {
    "roles": [
      "user",
      "admin"
    ],
    "viewed_by": [
      "john-123",
      "mary-456"
    ]
  }
}
```

🟥 GROUP 4: Audit & Security

AUDIT_LOGS (Complete Activity Trail)

```
audit_logs
 ├─ log_id [PK]
 ├─ user_id [FK → users] - Who did it?
 ├─ app_id [FK → applications] - In which app?
 ├─ action - login, create, update, delete, view
 ├─ resource_type - What was affected?
 ├─ resource_id - Specific item ID
 ├─ old_values [JSONB] - Before change
 ├─ new_values [JSONB] - After change
 ├─ ip_address - Security tracking
 └─ created_at - When?
```

Why complete audit trail?

- **Compliance** - GDPR, HIPAA, SOX requirements
- **Security** - Detect suspicious activity
- **Debugging** - “What happened to this record?”
- **Analytics** - User behavior patterns

Example Audit Entries:

User Login:

```
{
  "action": "login",
  "user_id": "john-123",
  "app_id": "sales-app",
  "ip_address": "192.168.1.100",
  "created_at": "2025-11-05 09:00:00"
}
```

Record Update:

```
{
  "action": "update",
  "user_id": "john-123",
  "app_id": "inventory-app",
  "resource_type": "product",
  "resource_id": "prod-456",
  "old_values": {
    "price": 99.99,
    "stock": 10
    },
  "new_values": {
    "price": 89.99,
    "stock": 10
  },
  "ip_address": "192.168.1.100",
  "created_at": "2025-11-05 09:15:00"
}
```

Failed Access Attempt:

```
{
  "action": "access_denied",
  "user_id": "mary-456",
  "app_id": "admin-panel",
  "resource_type": "admin_settings",
  "new_values": {
    "reason": "insufficient_permissions"
  },
  "ip_address": "192.168.1.101",
  "created_at": "2025-11-05 09:30:00"
}
```

Immutable Logs:
- Never UPDATE or DELETE audit logs
- Only INSERT new entries
- Preserves complete history

RESOURCE_ACCESS_LOGS (Resource-Specific Tracking)

```
resource_access_logs
├── access_log_id [PK]
├── resource_id [FK → app_resources] - Which resource?
├── user_id [FK → users] - Who accessed?
├── action - view, download, edit, delete
├── is_allowed - TRUE/FALSE (access granted/denied)
├── denial_reason - Why denied (if is_allowed=FALSE)
└── accessed_at - When?
```

Difference from audit_logs:
- More specific to resource access
- Tracks denied attempts
- Useful for security analysis

Example: Document Access Tracking

```
Document: "Confidential Report 2025"
├── John viewed - ALLOWED (09:00)
├── Mary viewed - ALLOWED (09:15)
├── Guest123 tried to download - DENIED (09:30)
│   └── Reason: "insufficient_permissions"
└── John downloaded - ALLOWED (09:45)
```

Security Use Case:

```
-- Find suspicious access patterns
SELECT user_id, COUNT(*) as denied_attempts FROM resource_access_logs
WHERE is_allowed = FALSE  AND accessed_at > NOW() - INTERVAL '1 hour'G
ROUP BY user_id HAVING COUNT(*) > 5;
-- Alert: User attempting too many unauthorized accesses!
```

🔄 Complete Data Flow Examples

Example 1: New User Registration & First Login

```
STEP 1: User signs up in App A
┌─────────────────────────────────────────┐
│ INSERT INTO users                       │
│ ├─ email: john@example.com              │
│ ├─ password_hash: $2b$10$...            │
│ ├─ is_active: TRUE                      │
│ └─ is_verified: FALSE                   │
└─────────────────────────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────────┐
│ INSERT INTO user_roles                  │
│ ├─ user_id: john-123                    │
│ └─ role_id: user-role-id                │
└─────────────────────────────────────────┘
                    │
                    ▼
┌─────────────────────────────────────────┐
│ INSERT INTO audit_logs                  │
│ ├─ action: user_registered              │
│ └─ user_id: john-123                    │
└─────────────────────────────────────────┘


STEP 2: User logs into App A
```

```
┌─────────────────────────────────────┐
│ SELECT * FROM users                 │
│ WHERE email = 'john@example.com'    │
│ AND is_active = TRUE                │
└─────────────────────────────────────┘
              │ (Password verified)
              ▼
┌─────────────────────────────────────┐
│ INSERT INTO user_sessions           │
│  ├─ user_id: john-123               │
│  ├─ app_id: app-a-id                │
│  ├─ session_token: jwt-token        │
│  └─ expires_at: NOW() + 1 hour      │
└─────────────────────────────────────┘
              │
              ▼
┌─────────────────────────────────────┐
│ INSERT INTO audit_logs              │
│  ├─ action: login                   │
│  ├─ user_id: john-123               │
│  └─ app_id: app-a-id                │
└─────────────────────────────────────┘

STEP 3: User opens App B (same browser)
┌─────────────────────────────────────┐
│ SELECT * FROM user_sessions         │
│ WHERE user_id = 'john-123'          │
│ AND expires_at > NOW()              │
└─────────────────────────────────────┘
                │ (Active session found!)
                ▼
┌─────────────────────────────────────┐
│ INSERT INTO user_sessions           │
│  ├─ user_id: john-123               │
│  ├─ app_id: app-b-id                │
│  ├─ session_token: new-jwt-token    │
│ └─ expires_at: NOW() + 1 hour       │
└──────────────────────────────────────┘
                    │
                    ▼
     ✅ Logged into App B automatically!
```

Example 2: Permission Check Flow

```
QUESTION: Can John create users in App A?

STEP 1: Get John's roles
┌──────────────────────────────────────┐
│ SELECT role_id FROM user_roles      │
│ WHERE user_id = 'john-123'         │
│ AND (expires_at IS NULL OR         │
│     expires_at > NOW())           │
└──────────────────────────────────────┘
                    │ Result: role_id = 'admin'
                    ▼

STEP 2: Get permissions for Admin role
┌──────────────────────────────────────┐
│ SELECT permission_id               │
│ FROM role_permissions              │
│ WHERE role_id = 'admin'            │
└──────────────────────────────────────┘
                    │ Result: multiple permission IDs
                    ▼

STEP 3: Check if 'users.create' included
┌──────────────────────────────────────┐
│ SELECT * FROM permissions          │
│ WHERE permission_id IN (...)       │
│ AND permission_name = 'users.create'│
└──────────────────────────────────────┘
```

```
                    |
                    ▼
        ✅ YES! John can create users

    Or use helper function:
    SELECT user_has_permission('john-123', 'users.create')
    → Returns TRUE
```

Example 3: Adding New App-Specific Feature

```
REQUIREMENT: Add "Department" field for users in HR App only

TRADITIONAL WAY (Slow):
❌ ALTER TABLE users ADD COLUMN department VARCHAR(100);
❌ Requires downtime
❌ Affects all apps
❌ Can't have different departments per app

OUR WAY (Instant):
✅ UPDATE users
   SET metadata = metadata || '{"hr_department": "Engineering"}'
   WHERE user_id = 'john-123';

✅ No downtime
✅ Only HR app uses this field
✅ Other apps unaffected
✅ Can add more fields anytime

ACCESS IN APPLICATION:
SELECT
  user_id,
  email,
  metadata->>'hr_department' as department
```

```
FROM users
WHERE metadata->>'hr_department' IS NOT NULL;
```

📊 Relationship Cardinality Explained

One-to-Many (||–o{)

Meaning: One record in Table A relates to many records in Table B

Example: `users ||--o{ user_sessions`
- One user can have many sessions (mobile, desktop, different apps)
- Each session belongs to exactly one user

Database Enforcement:

```
-- user_sessions table has foreign key to users
user_id UUID REFERENCES users(user_id)
-- This prevents:
❌ Creating session for non-existent user
❌ Orphaned sessions (user deleted but session remains)
-- CASCADE: When user deleted, all their sessions deleted too
```

Many-to-Many (resolved with junction table)

Example: Users ↔ Roles

Without junction table:
❌ Can’t store (user can’t have multiple roles)

With junction table (user_roles):
✅ users ||–o{ user_roles }o–|| roles
- One user → many user_roles entries
- One role → many user_roles entries
- Creates many-to-many relationship

Visual:

```
USERS       USER_ROLES       ROLES

┌───────┐   ┌─────────────┐   ┌────────┐
```

```
│John│──────────→│john,    │←──────────│Admin│
└──────┘    │    │admin    │      └──────┘
    │    └───────────┘
    │    ┌───────────┐      ┌─────┐
    └────→│john,    │←─────────│User│
       │user    │    └──────┘
       └───────────┘
```

🎨 Visual Representation of Key Flows

Single Sign-On Visualization

```
USER: John
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━

[App A]                [Central DB]              [App B]
   │                        │                        │
   │  1. Login                 │                        │
   ├───────────────────────────────────→│                        │
   │     (email, password)        │                        │
   │                        │                        │
   │  2. Verify credentials       │                        │
   │     ✅ Valid                │                        │
   │                        │                        │
   │  3. Create session           │                        │
   │←──────────────────────────────────┤                        │
   │     (session_token_A)        │                        │
   │                        │                        │
   │  4. User opens App B        │  5. Check existing session  │
   │                        │←───────────────────────────────────│
   │                        │     (user_id from token)      │
   │                        │                        │
   │                        │  6. Session exists!         │
   │                        │     Create new session       │
```

```
    │                    ├────────────────────────────────│
    │                    │   (session_token_B)      │
    │                    │                    │
    │                    │ 7. Logged in automatically!│
    │                    │   ✅ No password needed    │
```

💡 Key Takeaways

1. **Centralized = Consistent**

- One place for all user data
- No synchronization issues
- Single source of truth

2. **Flexible = Future-proof**

- JSONB columns for extensibility
- Add features without downtime
- Adapt to changing requirements

3. **Secure = Auditable**

- Every action logged
- Granular permissions
- Complete access control

4. **Scalable = Efficient**

- Proper indexes for performance
- Normalized design prevents redundancy
- Can handle millions of users

🔧 Database Maintenance Queries

Clean expired sessions (run daily)

```
SELECT clean_expired_sessions();
```

Find users with specific permission

```
SELECT u.email, u.username
FROM users u
JOIN user_roles ur ON u.user_id = ur.user_id
JOIN role_permissions rp ON ur.role_id = rp.role_id
JOIN permissions p ON rp.permission_id = p.permission_id
WHERE p.permission_name = 'users.delete'  AND (ur.expires_at IS NULL OR ur.expires_at > NOW());
```

Audit: Who accessed sensitive resource?

```
SELECT
  u.email,
  ral.action,
  ral.accessed_at,
  ral.is_allowed
FROM resource_access_logs ral
JOIN users u ON ral.user_id = u.user_id
WHERE ral.resource_id = 'sensitive-doc-123'ORDER BY ral.accessed_at DESC;
```