In a **ThingsBoard-style dashboard system**, roles and permissions are crucial for managing access to dashboards, widgets, and data. Here's a breakdown of how you can implement this in your **React + TypeScript dashboard app** (and potentially integrate with the ThingsBoard backend):

**👥 Dashboard Roles**

| **Role** | **Description** |
| --- | --- |
| **Admin** | Full access to all dashboards, widgets, settings, and user management |
| **Editor** | Can create, edit, and configure dashboards and widgets |
| **Viewer** | Can view dashboards but cannot edit or configure anything |
| **Guest** | Limited, read-only access to specific dashboards (e.g., public or shared) |

**🔐 Permissions Model**

Permissions can be broken into categories like:

| **Permission** | **Role Level** | **Example** |
| --- | --- | --- |
| view\_dashboard | All roles | Can open and view dashboards |
| edit\_dashboard | Admin, Editor | Can rearrange or rename dashboards |
| configure\_widgets | Admin, Editor | Can change widget settings |
| add/remove\_widgets | Admin, Editor | Can insert or delete widgets |
| manage\_users | Admin only | Add/remove users or roles |
| share\_dashboard\_public | Admin only | Make dashboard accessible without login |

**🧩 How to Implement It**

**1. Role Definition**

In your app config or backend, define roles:

ts

CopyEdit

export type Role = 'admin' | 'editor' | 'viewer' | 'guest';

export interface UserContext {

id: string;

name: string;

role: Role;

}

**2. Permission Check Utility**

ts

CopyEdit

const rolePermissions: Record<Role, string[]> = {

admin: ['view', 'edit', 'configure', 'manage', 'share'],

editor: ['view', 'edit', 'configure'],

viewer: ['view'],

guest: ['view']

};

export const hasPermission = (role: Role, action: string): boolean => {

return rolePermissions[role].includes(action);

};

**3. Frontend Example Usage**

tsx

CopyEdit

import { hasPermission } from '@/utils/permissions';

const DashboardToolbar = ({ role }: { role: Role }) => {

return (

<div className="toolbar">

{hasPermission(role, 'edit') && <button>Edit Dashboard</button>}

{hasPermission(role, 'configure') && <button>Configure Widget</button>}

</div>

);

};

**4. ThingsBoard Integration (optional)**

If integrating with the backend:

* Assign roles in ThingsBoard UI (System Admin > Users)
* Use JWT or OAuth to pass the role to the frontend
* Backend APIs like /api/dashboard/<id> may enforce ACLs

Would you like me to generate:

* A roles.ts utility with permission checks
* A UI mock with conditional buttons based on role
* Integration of this logic into the existing dashboard?

Let me know how deep you'd like to go!

**You said:**

Dashboard roles and permission, assignment, Shared and private explain

**ChatGPT said:**

Here’s a full explanation of **Dashboard Roles, Permissions, Assignment, and Sharing** — tailored for a ThingsBoard-style React + TypeScript dashboard:

**👥 Roles & Permissions**

**✅ Common Roles**

| **Role** | **Capabilities** |
| --- | --- |
| **Admin** | Full access: manage dashboards, users, assign roles, share public/private |
| **Editor** | Create/edit dashboards and widgets, configure settings |
| **Viewer** | Read-only access to assigned dashboards |
| **Guest** | View-only, typically public dashboards |

**🛡️ Permissions per Role**

| **Permission** | **Admin** | **Editor** | **Viewer** | **Guest** |
| --- | --- | --- | --- | --- |
| View dashboard | ✅ | ✅ | ✅ | ✅ |
| Edit dashboard | ✅ | ✅ | ❌ | ❌ |
| Configure widgets | ✅ | ✅ | ❌ | ❌ |
| Share dashboard | ✅ | ❌ | ❌ | ❌ |
| Assign roles | ✅ | ❌ | ❌ | ❌ |

**🔁 Assignment**

Dashboard access is **assigned per user or user group**:

* At login, the user's role is checked via token/session.
* Each dashboard stores an access list:

ts

CopyEdit

interface DashboardAccess {

dashboardId: string;

accessLevel: 'read' | 'write' | 'admin';

userIds: string[];

groupIds?: string[];

}

You can store this in:

* A backend database
* JSON-based access config
* ThingsBoard backend via API

**🔗 Shared vs. Private Dashboards**

| **Type** | **Description** |
| --- | --- |
| **Private** | Only visible to assigned users or roles. Used for internal dashboards. |
| **Shared** | Accessible via special link or public role (e.g. embedded in kiosk/TV dashboard) |

**Shared Dashboard Example**

ts

CopyEdit

{

dashboardId: 'abc-123',

shared: true,

sharedUrl: 'https://yourdomain.com/dash/abc-123',

accessToken: 'xyz987' // optional

}

**🧠 Implementation in React**

**1. Role-based rendering**

tsx

CopyEdit

{role === 'admin' && <button onClick={editDashboard}>Edit</button>}

**2. Permission utility**

ts

CopyEdit

const permissions = {

admin: ['edit', 'view', 'share'],

editor: ['edit', 'view'],

viewer: ['view'],

guest: ['view'],

};

export const can = (role: string, action: string) => permissions[role]?.includes(action);

**3. Shared Dashboard View**

* Use ?token= in URL for public access.
* Read-only UI, hide edit/config buttons.

Would you like me to:

* Generate a roles.ts file with permission logic?
* Add a dashboard access management UI (with share button + user assignment)?
* Implement backend mock for user-role assignment?

**What is a Backend Dashboard System?**

A **dashboard backend** is the central logic and data layer responsible for storing, processing, and delivering dashboards, widgets, and user configurations to the frontend.

It is built to:

* Handle **authentication & authorization**
* Serve **dashboard structures & widget data**
* Manage **user roles, sharing, and permissions**
* Interface with **data sources** (telemetry, databases, APIs)
* Provide **real-time streaming** or polling for data updates

**🧱 Core Components of a Dashboard Backend**

**1. Authentication & User Management**

* Sign-in / Sign-up
* Token-based auth (JWT or OAuth)
* Role-based access control (RBAC)

📁 Tables: users, roles, user\_roles

**2. Dashboard Service**

Handles all CRUD operations for dashboards:

* Create, Read, Update, Delete dashboards
* Export / Import dashboard layout as JSON
* Assign dashboard ownership and visibility

📁 Tables: dashboards, dashboard\_permissions, dashboard\_widgets

**3. Widget Service**

Handles widget types, configurations, and templates:

* Register widget bundles (e.g., Charts, Gauges)
* Store widget configurations (layout, settings)
* Deliver widget-specific data and metadata

📁 Tables: widget\_types, widget\_bundles, widget\_templates

**4. Permission System**

Implements **RBAC (Role-Based Access Control)** and **ACL (Access Control List)**:

* Who can view/edit/share dashboards?
* Which users or roles have what level of access?

📁 Tables: permissions, dashboard\_permissions, shared\_links

**5. Data Source & Stream Layer**

* Fetch data for widgets (from DB, APIs, telemetry)
* Handle **real-time data** using WebSockets or MQTT
* Include caching, transformation, aggregation if needed

📁 Tech: Redis, Kafka, PostgreSQL, MQTT, WebSocket server

**🔄 Backend Flow Example**

**"User opens a dashboard":**

1. 🧑 User sends request to /api/dashboard/:id
2. 🔐 Backend validates JWT and checks permission
3. 🧱 Backend queries dashboard layout and widgets
4. 📊 Widget configs and data sources are resolved
5. 📦 Response returns dashboard + widget layout + config
6. 🔄 Real-time data starts streaming (if enabled)

**🧩 Design Principles**

| **Principle** | **Description** |
| --- | --- |
| **Modular Widgets** | Widgets are separate, pluggable units with metadata and config |
| **User-Centric** | Each dashboard is tied to a user or group, with permissions |
| **Real-Time Ready** | Use WebSocket, Kafka, or MQTT for pushing data |
| **Multi-Tenant** | Each user/org can have isolated data and dashboards |
| **Exportable** | Dashboards and widgets can be saved/restored via JSON |

**📁 Typical Database Schema (Simplified)**

**users**

| **id** | **name** | **email** | **role** |
| --- | --- | --- | --- |

**dashboards**

| id | name | owner\_id | shared | shared\_url |

**dashboard\_permissions**

| id | dashboard\_id | user\_id | access\_level |

**widget\_types**

| id | name | bundle | schema |

**widget\_instances**

| id | dashboard\_id | type\_id | config | layout |

**🛠️ Technologies Commonly Used**

| **Layer** | **Tech Stack** |
| --- | --- |
| Backend Framework | **Spring Boot**, **Node.js**, **NestJS**, **Django** |
| Database | **PostgreSQL**, **MongoDB**, **MySQL** |
| Auth | **JWT**, **OAuth 2.0**, **Keycloak** |
| Realtime | **WebSocket**, **MQTT**, **Kafka** |
| APIs | **REST**, **GraphQL**, **gRPC** |
| Caching | **Redis** |
| Deployment | **Docker**, **Kubernetes**, **Cloud-native** |

**✅ Summary**

A backend dashboard system provides:

* A **secure and flexible foundation** for multi-user dashboards
* **Dynamic, configurable widgets** and layout management
* **Real-time and historical data** integration
* **Granular access control** for roles, users, and sharing

Would you like:

* A sample Java Spring Boot backend implementation for this?
* ERD (Entity Relationship Diagram) or PlantUML schema?
* GraphQL version of the dashboard API?

table to store dashboard is private or public with permissions and role table

**ChatGPT said:**

To store **dashboard visibility** (private/public) and associate **permissions and roles**, you can create a schema that supports:

* Public or private dashboards
* Role-based access to public/shared dashboards
* Granular access types (VIEW / EDIT)

**✅ Tables Overview**

| **Table** | **Purpose** |
| --- | --- |
| dashboards | Stores dashboard metadata |
| roles | Stores system roles |
| permissions | Stores permissions like dashboard.view |
| dashboard\_roles | Maps dashboards to roles with access types |

**📐 SQL Schema**

**📋 dashboards table**

sql

CopyEdit

CREATE TABLE dashboards (

id UUID PRIMARY KEY,

name VARCHAR(255) NOT NULL,

owner\_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,

is\_public BOOLEAN DEFAULT FALSE,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

* is\_public = TRUE means the dashboard is visible to everyone (read-only or viewable without login, depending on implementation).
* is\_public = FALSE means private or role-restricted.

**👥 roles table**

sql

CopyEdit

CREATE TABLE roles (

id UUID PRIMARY KEY,

name VARCHAR(100) UNIQUE NOT NULL,

description TEXT

);

**🔐 permissions table**

sql

CopyEdit

CREATE TABLE permissions (

id UUID PRIMARY KEY,

name VARCHAR(100) UNIQUE NOT NULL, -- e.g. 'dashboard.view', 'dashboard.edit'

description TEXT

);

**🧩 dashboard\_roles table (Dashboard ↔ Role mapping with access)**

sql

CopyEdit

CREATE TABLE dashboard\_roles (

id UUID PRIMARY KEY,

dashboard\_id UUID REFERENCES dashboards(id) ON DELETE CASCADE,

role\_id UUID REFERENCES roles(id) ON DELETE CASCADE,

access\_type VARCHAR(10) CHECK (access\_type IN ('view', 'edit')),

UNIQUE(dashboard\_id, role\_id)

);

✅ This enables **shared dashboards** by role with either view or edit access.

**🧠 Access Resolution Logic**

To determine if a user can access a dashboard:

1. If dashboard.is\_public = TRUE ➝ **grant VIEW**
2. Else if dashboard.owner\_id = current\_user.id ➝ **grant VIEW + EDIT**
3. Else:
   * Get user’s roles
   * Check dashboard\_roles for matching role\_id and access\_type

**🧾 Example Data**

sql

CopyEdit

-- Dashboard owned by Alice

INSERT INTO dashboards (id, name, owner\_id, is\_public)

VALUES ('dash-uuid', 'My Dashboard', 'alice-uuid', FALSE);

-- Viewer Role

INSERT INTO roles (id, name)

VALUES ('viewer-role', 'Viewer');

-- Give VIEW access to 'Viewer' role

INSERT INTO dashboard\_roles (dashboard\_id, role\_id, access\_type)

VALUES ('dash-uuid', 'viewer-role', 'view');

**🛠️ Optional Enhancements**

* Add user\_roles and role\_permissions if you want granular, extendable RBAC
* Support user\_id in dashboard\_roles to allow user-specific access
* Add expiration dates for access control (e.g., temporary dashboard sharing)

Would you like:

* A full ERD (PlantUML)?
* JPA entities or SQLAlchemy models?
* Integration with your current Spring Boot backend?

Let me know how deep you want to go!

4o

**You said:**

yes

**ChatGPT said:**

Awesome! Here's a full **ERD (Entity Relationship Diagram)** in **PlantUML**, along with a detailed breakdown of each entity — specifically designed for a **dashboard system** with:

* ✅ Private & public dashboards
* ✅ Role-based access control (RBAC)
* ✅ Permission support
* ✅ Dashboard sharing via role access levels

**🌐 PlantUML ERD: Dashboards, Roles, Permissions, Access**

plantuml

CopyEdit

@startuml DashboardAccessERD

entity "users" {

\* id : UUID <<PK>>

--

username : varchar

email : varchar

}

entity "roles" {

\* id : UUID <<PK>>

--

name : varchar

description : text

}

entity "permissions" {

\* id : UUID <<PK>>

--

name : varchar

description : text

}

entity "user\_roles" {

\* user\_id : UUID <<FK>>

\* role\_id : UUID <<FK>>

}

entity "role\_permissions" {

\* role\_id : UUID <<FK>>

\* permission\_id : UUID <<FK>>

}

entity "dashboards" {

\* id : UUID <<PK>>

--

name : varchar

owner\_id : UUID <<FK>>

is\_public : boolean

created\_at : timestamp

}

entity "dashboard\_roles" {

\* id : UUID <<PK>>

--

dashboard\_id : UUID <<FK>>

role\_id : UUID <<FK>>

access\_type : varchar (view/edit)

}

users ||--o{ user\_roles

roles ||--o{ user\_roles

roles ||--o{ role\_permissions

permissions ||--o{ role\_permissions

users ||--o{ dashboards : "owns"

dashboards ||--o{ dashboard\_roles

roles ||--o{ dashboard\_roles

@enduml

**📦 Tables Recap**

**dashboards**

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| id | UUID | Primary key |
| name | String | Dashboard name |
| owner\_id | UUID | Foreign key to users(id) |
| is\_public | Boolean | Public or private visibility |
| created\_at | Timestamp | Creation date |

**dashboard\_roles**

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| dashboard\_id | UUID | Dashboard foreign key |
| role\_id | UUID | Role foreign key |
| access\_type | Enum | 'view' or 'edit' |

**Permissions Example**

| **name** | **Description** |
| --- | --- |
| dashboard.view | View dashboards |
| dashboard.edit | Modify dashboards |
| dashboard.share | Share dashboards |

**🔍 How it Works**

1. User logs in → system loads their roles (user\_roles)
2. To load dashboards:
   * If is\_public = true ➝ allow view
   * If user is owner\_id ➝ allow edit
   * If user’s roles match dashboard\_roles.role\_id, respect access\_type

Would you like:

* ✅ Full **Spring Boot JPA entity classes**?
* ✅ A working **repository/service layer** example?
* ✅ A React UI design for managing shared dashboards?

Happy to generate any or all for you!

**🧠 What is ThingsBoard?**

**ThingsBoard** is an open-source IoT platform for:

* Device management
* Data collection
* Visualization (via dashboards & widgets)
* Alarming and rule engine automation

The **dashboard system** is one of its key modules that allows users to build real-time, interactive UI to visualize telemetry from devices and assets.

**🏗️ Backend Stack Overview**

| **Layer** | **Tech** |
| --- | --- |
| Language | Java |
| Framework | **Spring Boot** |
| Data Access | **JPA / Hibernate** |
| Messaging | Akka / Kafka (optional) |
| Database | PostgreSQL, Cassandra |
| Real-time | WebSockets, MQTT |
| Auth | JWT, OAuth2 |
| APIs | REST + WebSocket |

**📁 Key Backend Modules for Dashboards**

**1. Dashboard Entity**

Path:

swift

CopyEdit

dao/src/main/java/org/thingsboard/server/common/data/Dashboard.java

This is the main model for a dashboard. It includes:

java

CopyEdit

public class Dashboard extends SearchTextBasedWithAdditionalInfo<DashboardId> {

private TenantId tenantId;

private CustomerId customerId;

private String title;

private JsonNode configuration;

}

✅ configuration holds layout, widgets, settings, and datasources as a **JSON blob**

**2. Dashboard Service Layer**

Path:

swift

CopyEdit

service/src/main/java/org/thingsboard/server/dao/dashboard/DashboardService.java

Provides CRUD operations for dashboards:

* createDashboard()
* findDashboardById()
* deleteDashboard()
* assignDashboardToCustomer() (sharing)

**3. Dashboard Repository (DAO)**

Path:

swift

CopyEdit

dao/src/main/java/org/thingsboard/server/dao/dashboard/DashboardDao.java

This uses **JPA** to fetch dashboards by ID, tenant, or customer:

java

CopyEdit

DashboardEntity findByTenantIdAndTitle(UUID tenantId, String title);

**4. Dashboard Controller (REST API)**

Path:

swift

CopyEdit

application/src/main/java/org/thingsboard/server/controller/DashboardController.java

Provides the REST API endpoints like:

* GET /api/dashboard/{dashboardId}
* POST /api/dashboard
* DELETE /api/dashboard/{dashboardId}
* POST /api/customer/{customerId}/dashboard/{dashboardId} → sharing

It also supports **role-based access control** for:

* Admins
* Customers (users under a tenant)

**5. Widget Metadata**

Widgets are managed separately via:

* WidgetType (widget config and metadata)
* WidgetBundle (logical grouping of widget types)

These live in:

bash

CopyEdit

common/data/WidgetType.java

common/data/WidgetBundle.java

dao/widget/

**6. Widget Configuration**

Each widget instance is stored inside the Dashboard.configuration JSON:

json

CopyEdit

{

"widgets": {

"widget1": {

"type": "timeseries",

"title": "Temperature",

"config": {

"datasources": [...],

"settings": {...}

}

}

},

"layouts": {

"main": {

"widgets": {

"widget1": { "row": 0, "col": 0, "sizeX": 3, "sizeY": 2 }

}

}

}

}

Widgets are rendered based on this dynamic schema.

**7. Real-time Data Handling**

ThingsBoard supports **WebSocket** and **MQTT**:

* Device telemetry is sent in real time
* Backend pushes updates via WebSocket to the dashboard
* Widget subscribes to a datasource and listens for updates

**🔐 Permissions & Access**

Dashboards are tied to:

* **Tenant** → Organization owner
* **Customer** → Assigned users
* **Public** → Shared via link

Sharing is handled via customer assignment and visibility flags.

**📊 Summary**

| **Component** | **Responsibility** |
| --- | --- |
| Dashboard.java | Represents dashboard metadata and layout |
| DashboardService | Business logic for dashboard CRUD & sharing |
| DashboardController | REST API layer |
| WidgetType / WidgetBundle | Reusable widget components |
| configuration (JSON) | Layout, widget state, data bindings |
| WebSocket/MQTT | Real-time data to frontend widgets |

Would you like:

* A simplified React + Spring Boot version of this?
* JSON schema for configuration?
* Live editable dashboard API example?

I can help scaffold it for you!

Awesome! Here's a full **ERD (Entity Relationship Diagram)** in **PlantUML**, along with a detailed breakdown of each entity — specifically designed for a **dashboard system** with:

* ✅ Private & public dashboards
* ✅ Role-based access control (RBAC)
* ✅ Permission support
* ✅ Dashboard sharing via role access levels

**🌐 PlantUML ERD: Dashboards, Roles, Permissions, Access**

plantuml

CopyEdit

@startuml DashboardAccessERD

entity "users" {

\* id : UUID <<PK>>

--

username : varchar

email : varchar

}

entity "roles" {

\* id : UUID <<PK>>

--

name : varchar

description : text

}

entity "permissions" {

\* id : UUID <<PK>>

--

name : varchar

description : text

}

entity "user\_roles" {

\* user\_id : UUID <<FK>>

\* role\_id : UUID <<FK>>

}

entity "role\_permissions" {

\* role\_id : UUID <<FK>>

\* permission\_id : UUID <<FK>>

}

entity "dashboards" {

\* id : UUID <<PK>>

--

name : varchar

owner\_id : UUID <<FK>>

is\_public : boolean

created\_at : timestamp

}

entity "dashboard\_roles" {

\* id : UUID <<PK>>

--

dashboard\_id : UUID <<FK>>

role\_id : UUID <<FK>>

access\_type : varchar (view/edit)

}

users ||--o{ user\_roles

roles ||--o{ user\_roles

roles ||--o{ role\_permissions

permissions ||--o{ role\_permissions

users ||--o{ dashboards : "owns"

dashboards ||--o{ dashboard\_roles

roles ||--o{ dashboard\_roles

@enduml

**📦 Tables Recap**

**dashboards**

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| id | UUID | Primary key |
| name | String | Dashboard name |
| owner\_id | UUID | Foreign key to users(id) |
| is\_public | Boolean | Public or private visibility |
| created\_at | Timestamp | Creation date |

**dashboard\_roles**

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| dashboard\_id | UUID | Dashboard foreign key |
| role\_id | UUID | Role foreign key |
| access\_type | Enum | 'view' or 'edit' |

**Permissions Example**

| **name** | **Description** |
| --- | --- |
| dashboard.view | View dashboards |
| dashboard.edit | Modify dashboards |
| dashboard.share | Share dashboards |

**🔍 How it Works**

1. User logs in → system loads their roles (user\_roles)
2. To load dashboards:
   * If is\_public = true ➝ allow view
   * If user is owner\_id ➝ allow edit
   * If user’s roles match dashboard\_roles.role\_id, respect access\_type

Would you like:

* ✅ Full **Spring Boot JPA entity classes**?
* ✅ A working **repository/service layer** example?
* ✅ A React UI design for managing shared dashboards?

Happy to generate any or all for you!

4o

**You said:**

table mapping defining dashboard is shared/private and public

**ChatGPT said:**

Great! Let's define a clean and efficient **SQL table structure** that clearly captures:

* Whether a **dashboard is private**, **shared** (to specific users or roles), or **public** (accessible to everyone).
* The **type of access**: VIEW or EDIT.
* The **entity it's shared with**: a **user** or a **role**.

**✅ Core Table: dashboards**

sql

CopyEdit

CREATE TABLE dashboards (

id UUID PRIMARY KEY,

name VARCHAR(255) NOT NULL,

owner\_id UUID NOT NULL REFERENCES users(id) ON DELETE CASCADE,

visibility VARCHAR(10) NOT NULL CHECK (visibility IN ('private', 'shared', 'public')),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**🔍 visibility values:**

| **Value** | **Meaning** |
| --- | --- |
| private | Only the owner\_id can access |
| shared | Shared with specific users or roles via access mappings |
| public | Everyone (authenticated or not) can access (view only) |

**🧩 Table: dashboard\_access**

This table maps a **dashboard** to a **user** or **role**, with specific access\_type.

sql

CopyEdit

CREATE TABLE dashboard\_access (

id UUID PRIMARY KEY,

dashboard\_id UUID NOT NULL REFERENCES dashboards(id) ON DELETE CASCADE,

user\_id UUID REFERENCES users(id) ON DELETE CASCADE,

role\_id UUID REFERENCES roles(id) ON DELETE CASCADE,

access\_type VARCHAR(10) NOT NULL CHECK (access\_type IN ('view', 'edit')),

CONSTRAINT one\_target CHECK (

(user\_id IS NOT NULL AND role\_id IS NULL) OR

(user\_id IS NULL AND role\_id IS NOT NULL)

),

UNIQUE (dashboard\_id, user\_id, role\_id)

);

**✅ What this does:**

* Ensures a dashboard can be shared **either** with a user **or** a role, not both at once per entry.
* Defines the **level of access**: view or edit.

**🧠 Example Usage**

**🛠️ Insert a private dashboard:**

sql

CopyEdit

INSERT INTO dashboards (id, name, owner\_id, visibility)

VALUES (gen\_random\_uuid(), 'Admin Dashboard', 'user-uuid', 'private');

**🌐 Insert a public dashboard:**

sql

CopyEdit

INSERT INTO dashboards (id, name, owner\_id, visibility)

VALUES (gen\_random\_uuid(), 'Public Metrics', 'user-uuid', 'public');

**👥 Share a dashboard with a role (shared):**

sql

CopyEdit

INSERT INTO dashboards (id, name, owner\_id, visibility)

VALUES (gen\_random\_uuid(), 'Team Dashboard', 'user-uuid', 'shared');

INSERT INTO dashboard\_access (id, dashboard\_id, role\_id, access\_type)

VALUES (gen\_random\_uuid(), 'dashboard-uuid', 'viewer-role-uuid', 'view');

**🔐 How to Resolve Access (in service/backend)**

1. If visibility = 'public' ➝ allow view
2. If owner\_id = current\_user\_id ➝ allow edit (and view)
3. If visibility = 'shared':
   * Check dashboard\_access for:
     + user-specific access (matching user\_id)
     + role-based access (matching role\_id IN user.roles)
   * Respect access\_type

**🧭 Summary**

| **Table** | **Role** |
| --- | --- |
| dashboards | Tracks visibility: private, shared, or public |
| dashboard\_access | Manages access rights to users or roles |

Would you like a version of this in:

* ✅ PostgreSQL with constraints and indexes?
* ✅ Spring Boot JPA entity class version?
* ✅ SQLAlchemy or Prisma schema?

Let me know your stack and I’ll generate the full setup for you!