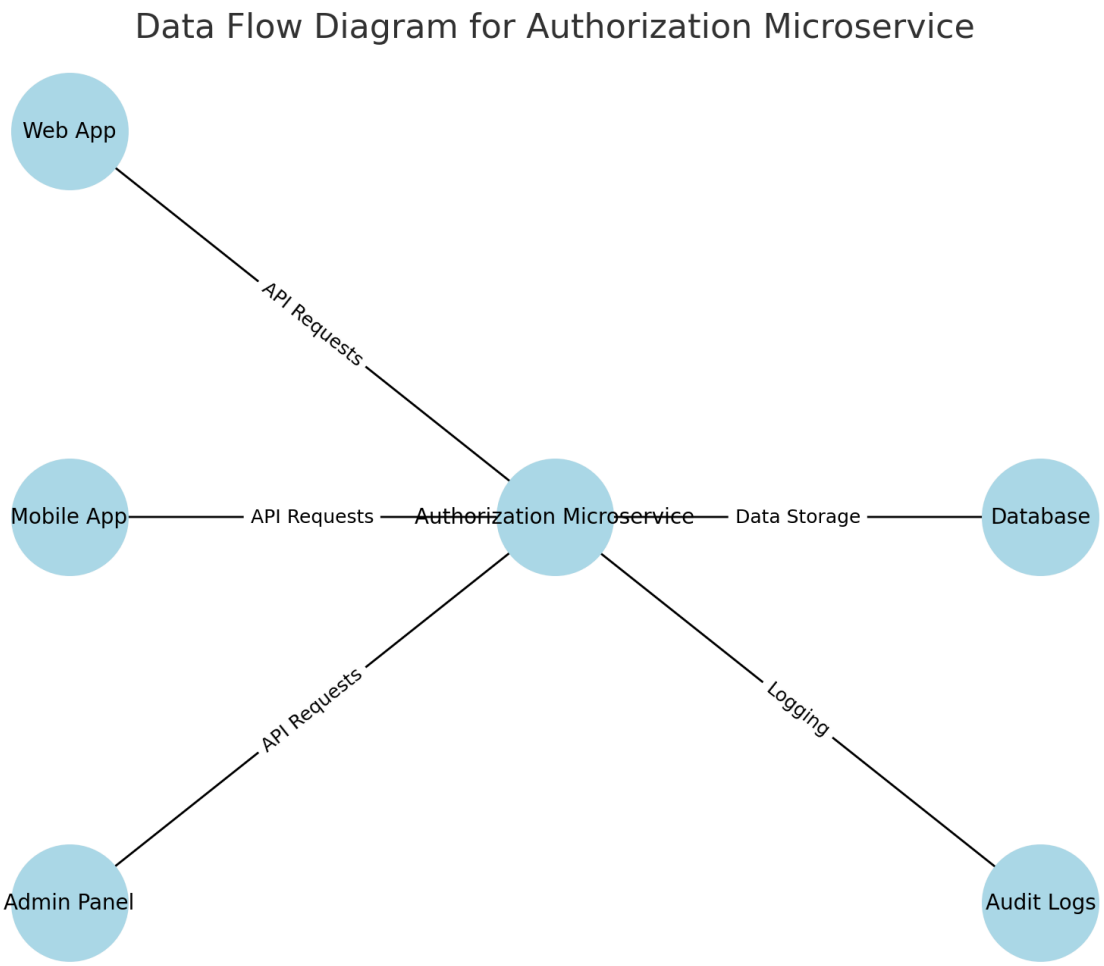
Section 2: System Design

Below is the data flow diagram for high-level architecture for the Authorization Microservice



Now I would like to explain the High-Level Architecture for Authorization Microservice-

2.1 Microservice Architecture

Handling Different Applications

- **Centralized Service**: A single authorization microservice acts as the central authority for managing roles, permissions, and access control.
- **Application Context**: Each application is assigned a unique identifier (e.g., **`AppID`**) to manage roles and permissions specific to that application.
- **API Integration**: Applications communicate with the service via RESTful APIs using JSON over HTTPS.

Multi-Tenancy Implementation

- **App-Specific Roles and Permissions**: Each application can define its roles and permissions within a segregated namespace. For example, roles for **`App1`** (e.g., Admin, Editor) are stored separately from those for **`App2`**.
- **Tenant Isolation**: Use a **`TenantID`** field in the database to ensure that roles, permissions, and data are isolated between applications.

Communication Flow

1. Applications (web, mobile, admin panel) send authorization requests to the service.
2. The microservice validates tokens, checks roles/permissions, and returns the result.
3. Responses include authorization decisions or error messages (e.g., Unauthorized).

API Endpoints

- **`POST /auth/login`**: Authenticate users and issue JWT tokens.
- **`POST /auth/logout`**: Revoke tokens.
- **`GET /roles`**: Retrieve roles for a specific application.
- **`POST /roles`**: Add new roles.
- **`POST /roles/{id}/permissions`**: Assign permissions to a role.
- **`GET /permissions`**: Retrieve permissions for a resource.
- **`GET /audit/logs`**: Retrieve audit logs (admin-only).

2.2 Security & Authentication

Authentication Approach

- **OAuth 2.0 with JWT**:
    - Users authenticate via OAuth 2.0 and receive a JWT token.
    - JWT tokens contain user ID, roles, permissions, and expiration time.
- **Validation**:
    - Tokens are verified using a signing algorithm.
    - Tokens include claims for multi-tenancy (e.g., **`AppID`**, **`TenantID`**).

Token Management

- **Expiration**: Tokens expire within a predefined time (e.g., 15 minutes).
- **Refresh Tokens**:
    - Refresh tokens are stored securely in the database and expire after a longer period.
    - Applications use refresh tokens to request new JWT tokens.

Secure Transmission

- Enforce **HTTPS** for all communications to prevent eavesdropping.
- Use **HSTS** to ensure secure connections.

Preventing Vulnerabilities

- **SQL Injection**: Use parameterized queries and ORM libraries (e.g., Eloquent in Laravel).
- **Cross-Site Scripting (XSS)**: Sanitize all user inputs and validate API payloads.
- **Cross-Site Request Forgery (CSRF)**: Use anti-CSRF tokens for all state-changing requests.

2.3 Data Model & Database Design

Entities and Relationships

- **Users**: Stores user information (ID, username, password, email).
- **Roles**: Defines roles (Admin, User, etc.).
- **Permissions**: Specifies actions (read, write, delete) for resources.
- **Resources**: Represents protected entities (API endpoints, files).
- **Audit Logs**: Records access attempts, actions, results, and timestamps.

Schema Design

```
CREATE TABLE users (
   id INT AUTO_INCREMENT PRIMARY KEY,
   username VARCHAR(255) UNIQUE NOT NULL,
   password VARCHAR(255) NOT NULL,
   email VARCHAR(255),
   tenant_id INT NOT NULL,
   created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE roles (
   id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(50) UNIQUE NOT NULL,
   tenant_id INT NOT NULL,
   description VARCHAR(255)
);

CREATE TABLE permissions (
   id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(50) UNIQUE NOT NULL,
   description VARCHAR(255)
);
```

```
CREATE TABLE role_permissions (
    role_id INT,
    permission_id INT,
    PRIMARY KEY (role_id, permission_id),
    FOREIGN KEY (role_id) REFERENCES roles(id),
    FOREIGN KEY (permission_id) REFERENCES permissions(id)
);

CREATE TABLE audit_logs (
    id INT AUTO_INCREMENT PRIMARY KEY,
    user_id INT,
    resource VARCHAR(255),
    action VARCHAR(50),
    result VARCHAR(50),
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

2.4 Scalability & High Availability

Scaling the Service

- **Stateless Architecture**: Use JWT tokens to ensure API servers are stateless.
- **Load Balancer**: Distribute requests across multiple instances.
- **Horizontal Scaling**: Add more instances during peak load.

Caching Strategy

- Use **Redis** to cache frequently accessed data like roles and permissions.
- Cache tokens to speed up validation.

Fault Tolerance

- Deploy the service in multiple regions to handle regional failures.
- Use **Kubernetes** for automatic scaling and failover.

2.5 Rate Limiting & Security

Rate Limiting

- Use tools like **Laravel Rate Limiting** or API Gateway features to:
    - Limit requests per user/IP.
    - Apply stricter limits for sensitive endpoints (e.g., login).

IP Blocking

- Identify and block IPs making malicious requests using **firewall rules**.

Sensitive Information Protection

- Mask sensitive data in logs.
- Exclude user credentials and tokens from responses.

2.6 Auditing & Logging

Detailed Logging

- Log every access control decision:
    - User ID, role, resource accessed, action performed, result, timestamp.
- Store logs in a secure database or logging service (e.g., ELK stack).

Compliance with Standards

- Retain logs for a defined period to meet compliance (e.g., GDPR).
- Anonymize user data in logs when required.