

# Organization Management Service – Detailed Documentation

## 1. System Overview

The Organization Management Service is designed as a lightweight multi-tenant backend using **FastAPI** and **MongoDB**. The system allows administrators to:

- Create organizations
- Update organization details
- Delete organizations
- Authenticate using JWT
- Automatically create dedicated MongoDB collections per organization

The goal of the design is simplicity, clear separation of responsibilities, and an intuitive workflow for managing multiple organizations inside a single application.

---

## 2. Architecture Explanation

The system is divided into three major layers:

### 2.1 FastAPI Backend

The backend is modular, containing:

#### Auth Module / Router

- Handles admin login
- Verifies credentials using hashed passwords
- Issues JWT tokens after successful authentication
- Protects organization CRUD routes

#### Organization Module / Router

- Creates organizations
- Updates organization names
- Deletes organizations
- Fetches metadata
- Communicates with MongoDB to create or rename dynamic collections

#### Utils Module

- Password hashing utilities
- JWT encoding and decoding functions
- Small helper utilities used across modules

This separation makes the application easier to maintain and scale by keeping authentication, business logic, and utilities independent.

## 2.2 MongoDB Master Database

A single **master database** is used for global metadata:

### Collections:

- **admins** – Stores admin login credentials (hashed passwords)
- **organizations** – Contains metadata for each created organization (organization\_name, email, hashed password, timestamps, etc.)

The master DB acts as the control plane for the entire system.

## 2.3 Tenant Collections (Dynamic Collections)

For each organization, the system automatically creates a dedicated collection with the naming pattern:

text

org\_<organization\_name>

This provides clean data separation between tenants and avoids accidental data collision.

### Key Functions:

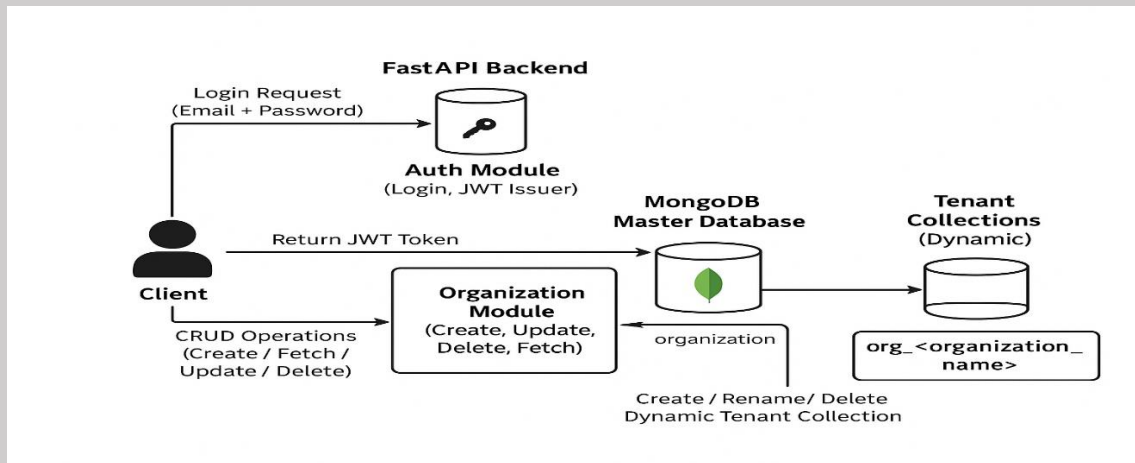
- Create new collection when a new org is registered
- Rename collection when organization name changes
- Drop collection when an organization is deleted

This ensures that each organization's data is physically isolated inside MongoDB.

---

### 3. Data Flow Explanation

Below are explanations for the diagrams



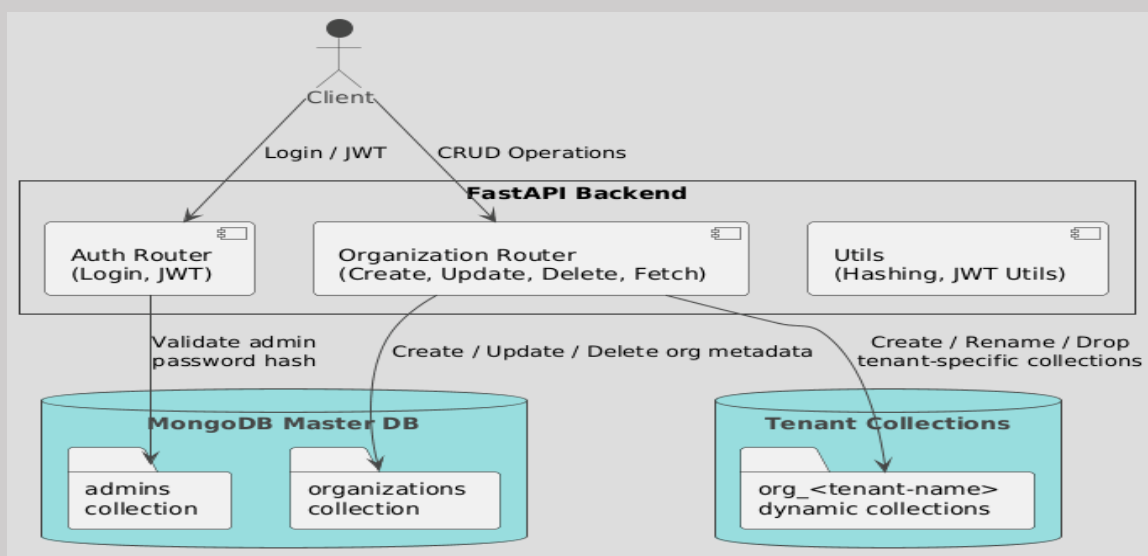
#### 3.1 Diagram 1: High-Level Workflow

This diagram shows the entire end-to-end process:

1. **Client sends login request** → Auth module
2. **Auth module validates credentials**
3. **JWT token returned to the client**
4. Client makes **CRUD operations** → Organization module
5. Organization module reads/writes to the master DB
6. MongoDB creates/renames/deletes tenant collections dynamically

This illustrates the complete lifecycle from authentication to organization-specific operations.

#### 3.2 Diagram 2: Backend Module Architecture



This diagram focuses on how the backend is internally structured:

- Client → FastAPI Backend
- FastAPI routes:
  - Auth Router
  - Organization Router
  - Utils Module
- MongoDB Master DB:
  - admins collection
  - organizations collection
- MongoDB Tenant Collections:
  - org\_<tenant-name>

It highlights how the backend modules map to MongoDB layers and ensures good logical separation.

4. API Endpoints Overview

Operation	Method	Endpoint	Description
Create Organization	POST	/org/create	Creates org metadata + dynamic collection
Admin Login	POST	/admin/login	Generates JWT token for admin
Check Auth	GET	/auth/check	Validates JWT
Update Organization	PUT	/org/update	Renames org and its collection
Delete Organization	DELETE	/org/delete	Deletes org metadata + tenant collection

## 5. Testing Workflow

1. **Create an Organization**
2. **Login using Email + Password**
3. **Use JWT token for authenticated routes**
4. **Fetch the organization**
5. **Update organization name**
6. **Delete organization**

This ensures that each feature works and verifies the full lifecycle of tenant management.

---

## 6. Design Choices & Reasoning

### ✓ **Simplicity**

Dynamic collections are easy to manage for small or medium SaaS platforms.

### ✓ **Clear isolation**

Each organization gets its own collection → less risk of data mixing.

### ✓ **Easy onboarding**

Creating a new tenant is as simple as creating a collection.

### ✓ **Low development overhead**

No advanced sharding, schema-level tenant filtering, or heavy infra.

---

## 7. Scalability Considerations & Trade-offs

Although this architecture is simple and effective, it comes with trade-offs.

### **Pros**

- Easy to develop and maintain
- Clean data separation
- Fast onboarding of new organizations
- Simple backup/restore per tenant

## Cons / Limitations

- 1. Too many collections can slow MongoDB** – Large-scale systems with hundreds or thousands of tenants may face performance issues.
  - 2. Noisy-neighbor problem** – All tenants share the same underlying cluster; heavy usage from one tenant affects others.
  - 3. Difficult to scale horizontally** – Dynamic collection creation can complicate sharding and replication.
- 

## 8. Better Alternative Designs (For Larger Scale)

### 1. Shared Collection (with tenant\_id field)

- Best for thousands of tenants
- Easier to index and shard
- Harder to isolate data, but very scalable

### 2. Separate Database per Tenant

- Strongest isolation (security + performance)
- Easy per-tenant backup/restore
- Higher resource and management overhead

### 3. Hybrid Approach

- Small tenants → shared collection
  - Large tenants → dedicated DB or collection
  - This gives both scalability and isolation
- 

## 9. Recommended Best Practices

- Use HTTPS everywhere
  - Strict input validation for organization names
  - Logging + monitoring
  - Strong JWT secret rotation policy
  - Dockerize services for portability
  - Add automated CI/CD pipeline
  - Prefer async DB operations (Motor) for high-concurrency workloads
-

## Final Notes

This design is excellent for an internship assignment or early-stage SaaS system. It is clean, understandable, and showcases important backend engineering concepts:

- Multi-tenancy
- JWT authentication
- Dynamic collection management
- Modular API design
- MongoDB schema modeling

With slight improvements, this can evolve into a highly scalable production architecture.