CROUPIER

Architecture & Design Review

Multi-Tenant Organization Management Service

This document provides a comprehensive architectural analysis of Croupier, a production-ready multi-tenant API built with FastAPI and MongoDB. It examines design decisions, scalability considerations, multi-tenancy patterns, and trade-offs in technology stack selection.

Author: Vibhor Srivastava

Institution: SRM IST NCR Campus

Assignment: The Wedding Company - Backend Intern

Date: December 13, 2025

Executive Summary

Croupier is a multi-tenant organization management service designed to demonstrate enterprise-grade backend architecture. The system employs a **collection-per-tenant** multi-tenancy pattern, balancing data isolation with operational efficiency. Built on FastAPI and MongoDB, the architecture prioritizes horizontal scalability through stateless JWT authentication and layered design patterns.

The core architectural principles include: (1) **Complete data isolation** via dedicated MongoDB collections per organization, (2) **Stateless authentication** enabling horizontal scaling without session management overhead, (3) **Layered architecture** with clear separation between routing, business logic, and data access layers, and (4) **Security-first design** with bcrypt password hashing and JWT bearer tokens.

This architecture serves 100-10,000 tenants efficiently, with documented evolution paths for larger scale. The modular design enables incremental improvements without requiring complete system rewrites.

1. Architecture Overview

1.1 System Design Principles

Croupier implements a **layered architecture pattern** with four distinct tiers, each responsible for specific concerns. This separation enables independent scaling, testing, and maintenance of each layer while maintaining clean interfaces between components.

Layer	Component	Responsibility
Presentation	FastAPI Routers	HTTP handling, request validation, response serialization
Business Logic	Services	Transaction orchestration, business rules enforcement
Data Access	Repositories	Database abstraction, CRUD operations
Persistence	MongoDB	Master database + per-tenant dynamic collections

1.2 Request Lifecycle

Request Flow: Client → Router (validate) → Service (business logic) → Repository (data access) → MongoDB
Response Flow: MongoDB → Repository (model mapping) → Service (transform) → Router (serialize) → Client

This unidirectional flow ensures clear responsibility boundaries and prevents circular dependencies between layers.

> ***Key Design Choice:*** *Stateless architecture with JWT-based authentication enables horizontal scaling without session management overhead. Multiple API instances can serve requests without requiring shared session storage (Redis/Memcached), simplifying deployment and reducing operational complexity.*

2. Key Design Decisions

2.1 Multi-Tenancy Strategy

Decision: Separate MongoDB collection per organization (`org_<name>`)

Rationale:

• **Data Isolation:** Complete separation prevents data leakage between tenants

• **Scalability:** Collections can be sharded independently based on tenant size

• **Performance:** Queries don't need tenant filters (WHERE org_id = ?), reducing overhead

• **Compliance:** Simplified GDPR/data residency compliance (delete collection = delete all tenant data)

> ***Trade-off:*** *Higher collection count vs. simpler queries and better isolation. This pattern scales to ~10,000 tenants before architectural changes are required.*

2.2 Authentication Architecture

Decision: JWT tokens with embedded `admin_id` + `organization_id`

Rationale:

• **Stateless:** No server-side session storage required → horizontal scaling

• **Performance:** Single token validation vs. database session lookup

• **Multi-tenant:** Organization context encoded in token → automatic tenant routing

> ***Security Measures:*** *bcrypt with 12 rounds for password hashing, JWT expiration enforcement, bearer token validation on all protected endpoints.*

3. Implementation Examples

The following code snippets illustrate key architectural patterns. These examples demonstrate **design decisions** rather than usage instructions.

3.1 Dynamic Collection Management

```
class DatabaseManager: def get_org_collection(self, organization_name: str): # Get or create organization-specific collection collection_name = f"org_{organization_name}" return self.master_db[collection_name] def drop_org_collection(self, organization_name: str): # Delete organization collection and all its data collection_name = f"org_{organization_name}" self.master_db.drop_collection(collection_name)
```

This pattern enables per-tenant data isolation while maintaining a single database connection. Collections are created on-demand when first accessed, eliminating pre-provisioning overhead.

3.2 Repository Pattern

```
class OrganizationRepository: def create(self, organization_data: Dict[str, Any]): # Create new organization with metadata organization_data['created_at'] = datetime.utcnow() result = self.collection.insert_one(organization_data) return self._serialize_document(organization_data) def find_by_name(self, organization_name: str): # Find organization by name doc = self.collection.find_one( {"organization_name": organization_name} ) return self._serialize_document(doc) if doc else None
```

Repositories abstract database operations, enabling testing with mock databases and future migration to alternative storage systems without modifying business logic.

3.3 Service Layer Orchestration

```
class OrganizationService: def create_organization(self, org_data): # Complete
organization creation workflow # 1. Validate uniqueness if
self.org_repo.exists(org_data.organization_name): raise HTTPException(400,
"Organization exists") # 2. Create organization metadata created_org =
self.org_repo.create({...}) # 3. Create admin user with hashed password
self.admin_repo.create({...}) # 4. Create dynamic collection with indexes
org_collection = self.db_manager.get_org_collection(...)
org_collection.create_index("created_at") return created_org
```

Services coordinate multiple repository operations to implement complete business workflows. This orchestration layer ensures consistent state across multiple database operations.

3.4 JWT Token Generation

```
class JWTHandler: @staticmethod def create_access_token(data: Dict[str, Any]): #
Create JWT token with expiration to_encode = data.copy() expire = datetime.utcnow()
+ timedelta(minutes=60) to_encode.update({ "exp": expire, "iat": datetime.utcnow(),
"admin_id": str(data["admin_id"]), "organization_id": data["organization_id"] })
return jwt.encode(to_encode, SECRET_KEY, algorithm="HS256")
```

Embedding tenant context (organization_id) in JWT tokens enables automatic request routing without database lookups. This stateless approach eliminates session storage requirements.

4. Scalability Analysis

Question: Is this architecture scalable? What are the limitations?

Answer: Yes, to a significant extent, but with important caveats and documented evolution paths.

Scalability Strength	Explanation
Horizontal Scaling	Stateless JWT allows adding multiple API servers without session synchronization
Data Isolation	Each organization has its own collection enabling targeted optimization
Independent Growth	Organizations can grow independently without affecting others
FastAPI Performance	Async capabilities handle high concurrency efficiently

Scalability Challenge	Impact & Mitigation
Collection Explosion	MongoDB has practical limits at 100,000+ collections; use collection pooling for small tenants
Namespace Limits	MongoDB namespaces have size constraints; monitor collection count with alerts
Schema Migrations	Changes must be applied to N collections; implement versioned migration scripts
Resource Overhead	Each collection has index metadata; archive inactive organizations to cold storage

Scalability Threshold: *This architecture efficiently serves 100-10,000 tenants. Beyond 10,000 tenants, transitioning to a hybrid tiered strategy (described in Section 6) becomes necessary to manage operational complexity.*

5. Technology Stack Trade-offs

Every technology choice involves trade-offs between competing priorities. This section analyzes the advantages and limitations of Croupier's stack decisions.

5.1 FastAPI Framework

Advantages	Limitations
High performance (comparable to Node.js/Go)	Smaller ecosystem vs. Django/Flask
Automatic OpenAPI documentation	Async paradigm adds complexity for simple CRUD
Native type safety with Pydantic	Less mature ORM integrations
Modern async/await for I/O-bound ops	Steeper learning curve for sync developers

5.2 MongoDB Database

Advantages	Limitations
Schema flexibility for evolving models	No ACID transactions across collections
Horizontal scaling through sharding	Collection count limits (~100k collections)
Fast write operations (document-based)	Application-layer referential integrity
Natural JSON representation for REST	Higher memory footprint vs. SQL

5.3 Collection-per-Tenant Pattern

Advantages	Limitations
Complete data isolation between tenants	Collection explosion beyond 10k tenants
No tenant_id filters needed in queries	Schema migrations require N operations
Simple tenant deletion (drop collection)	Cross-tenant analytics complexity
Per-tenant indexing strategies	Metadata overhead per collection

6. Alternative Multi-Tenancy Patterns

MongoDB supports three primary multi-tenancy strategies, each optimized for different scale and isolation requirements. This section compares these approaches.

Pattern	Scale	Isolation	Complexity	Best For
Shared Collection (tenant_id filter)	<1,000 tenants	Low	Low	Similar-sized tenants, frequent cross-tenant analytics
Collection per Tenant (Current)	100-10,000 tenants	High	Medium	Moderate isolation needs, operational efficiency balance
Database per Tenant	<100 high-value tenants	Maximum	High	Enterprise clients, compliance requirements (HIPAA/SOC2)

6.1 Shared Collection Pattern

All tenant data stored in a single collection with `tenant_id` discriminator. Every query must filter by tenant ID to ensure data isolation.

Pros: Simple management, no collection limits, easy cross-tenant analytics, single schema migration
Cons: Weak isolation, noisy neighbor effects, all queries require tenant filter, index bloat

Use Case: <1,000 similar-sized tenants with frequent cross-tenant reporting needs.

6.2 Database per Tenant Pattern

Complete physical isolation with dedicated database per tenant. Maximum security and customization at the cost of operational complexity.

Pros: Maximum isolation, independent backups, database-level security, per-tenant tuning
Cons: High operational complexity, connection pool challenges, MongoDB database limits (~10k)

Use Case: Enterprise clients with HIPAA/SOC2 requirements, <100 high-value tenants, geographic data residency needs.

6.3 Recommended: Hybrid Tiered Strategy

For production systems serving diverse tenant profiles, implement a **tiered hybrid approach**:

Tier	Pattern	Criteria	Examples
Enterprise	Database per Tenant	>$10k/month, compliance	Fortune 500, Healthcare
Professional	Collection per Tenant	Mid-size, standard isolation	SMBs, Startups
Starter	Shared Collection	<100 users, cost-sensitive	Freemium, Trials

Benefits: *Cost optimization (free/starter users share infrastructure), revenue optimization (enterprise clients get dedicated resources), operational efficiency (manage 1000s of small tenants), independent scaling per tier.*

This approach is used by production SaaS platforms like Slack, Atlassian, and Salesforce.

7. System Assessment & Future Evolution

7.1 Architecture Evaluation

Croupier's **collection-per-tenant architecture** provides an optimal balance for organizations scaling from hundreds to thousands of tenants. The implementation demonstrates production-ready patterns with clear growth paths.

• **Layered Design:** Clean separation enables independent testing and maintenance

• **Type Safety:** Pydantic validation throughout reduces runtime errors

• **Security-First:** Industry-standard bcrypt + JWT patterns

• **Scalability:** Stateless design enables horizontal scaling

• **Modularity:** Clear interfaces between layers simplify future enhancements

7.2 Known Limitations

• Collection count scales to ~10,000 tenants before architectural changes needed

• No token revocation mechanism (requires Redis integration for production)

• Cross-tenant analytics requires aggregation across collections

• Synchronous data migration during organization rename (should be async for large datasets)

7.3 Evolution Path

Phase	Scale	Actions Required
Phase 1	0-1k tenants	Current architecture sufficient
Phase 2	1k-10k tenants	Add Redis caching, implement MongoDB sharding
Phase 3	10k+ tenants	Migrate to hybrid tiered strategy (shared/collection/database)
Phase 4	100k+ tenants	Microservices decomposition, event-driven architecture

Final Assessment: *This architecture is* ***production-ready for initial deployment*** *and provides clear scaling paths. The modular design allows incremental improvements without requiring complete rewrites. With the documented enhancements, this system can grow from a startup MVP to an enterprise-scale platform serving hundreds of thousands of organizations.*

For practical setup instructions, API usage examples, and deployment guides, please refer to the project README.md file.