DISTRIBUTED DOCUMENT SERVICE

Design Document

Abstract

This system provides a secure, scalable, multi-tenant document management and search platform that enables efficient document indexing, fast full-text search, and high availability using Spring Boot, Elasticsearch, PostgreSQL, Redis, and asynchronous processing.

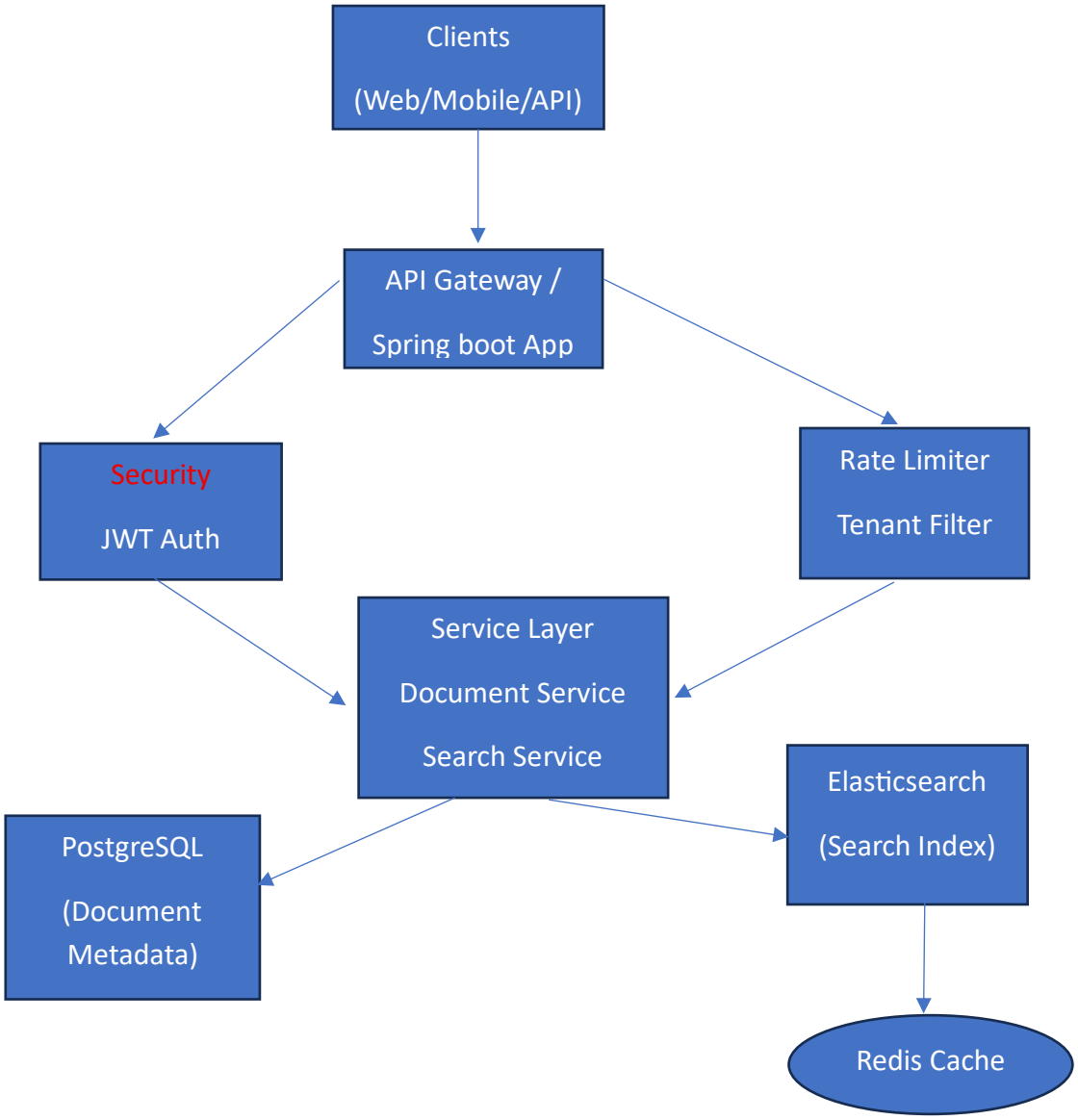
Aakanksha Saxena

Aakankshasaxena55@gmail.com

Contents

Distributed Document Search Service

1. High-Level System Architecture



Components:

- **Spring Boot API** – Handles HTTP requests, security, multi-tenancy, and rate-limiting.
- **JWT Authentication** – Secures endpoints and carries tenant & role info.

- **DocumentService** – CRUD operations on documents.
- **SearchService** – Indexes and queries Elasticsearch.
- **PostgreSQL** – Stores canonical document metadata.
- **Elasticsearch** – Full-text search engine for high-performance querying.
- **Redis Cache** – Caches search results and frequently accessed data.
- **Tenant & Rate Limiter Filters** – Enforces multi-tenancy and throttling.
- **Message Queue (Optional)** – Async indexing of documents into Elasticsearch.

2. Data Flow – Indexing & Search

2.1 Indexing a Document (POST /documents)

1. Client sends POST /documents with JSON document and JWT token.
2. API verifies JWT, extracts tenantId & roles.
3. DocumentService:
    - Stores document metadata in PostgreSQL.
    - Publishes an async message to a queue for Elasticsearch indexing (optional).
4. SearchService consumes message:
    - Indexes document in Elasticsearch.
    - Optionally updates Redis cache.

Flow Diagram:

2.2 Searching Documents (GET /search?q=&tenant=)

1. Client sends search query with JWT token.
2. API validates token and tenant.
3. SearchService checks Redis cache:
    - If hit → return cached results.

- If miss → query Elasticsearch → cache results → return response.

Flow Diagram:

3. Database & Storage Strategy

Layer	Choice & Reasoning
Relational DB	PostgreSQL – reliable ACID storage for document metadata
Search Engine	Elasticsearch – full-text, faceted search, relevance scoring
Cache	Redis – improves performance for frequently accessed queries

Trade-offs:

- PostgreSQL for consistency and transactional operations.
- Elasticsearch for fast, flexible search but eventual consistency with DB.
- Redis caching reduces search load but requires cache invalidation on updates.

4. API Design (Key Endpoints)

Endpoint	Method	Request/Response Example	Notes
/auth/token	POST	{ "username": "admin1", "password": "pass" } → { "accessToken": "..." }	Returns JWT
/documents	POST	{ "title": "Doc1", "content": "..." }	Indexes doc, multi-tenant
/documents/{id}	GET	/documents/1 → document JSON	Tenant isolation enforced
/documents/{id}	DELETE	/documents/1	Admin-only
/search?q={query}&tenant={tenant}	GET	Returns list of SearchDocument	Tenant isolation enforced

Security:

- @PreAuthorize("hasRole('USER')") for read
- @PreAuthorize("hasRole('ADMIN')") for delete

5. Consistency Model & Trade-offs

- **PostgreSQL:** Strong consistency (ACID).
- **Elasticsearch:** Eventual consistency – documents may not appear immediately after creation.
- **Redis Cache:** Stale reads possible until cache invalidation occurs.

Trade-off: Faster reads vs. potential temporary inconsistency.

6. Caching Strategy

- **Redis:** Search queries and results cached with TTL (10 min).
- **Invalidate Cache:** On document creation or update using @CacheEvict annotation.
- **Multi-layer caching:** Optional in-memory cache for very frequent reads.

7. Asynchronous Operations (Message Queue)

- Optional MQ (Kafka/RabbitMQ) used for:
    - Indexing documents in Elasticsearch asynchronously.
    - Reduces latency of POST /documents.
- Guarantees eventual consistency.

8. Multi-Tenancy & Data Isolation

- **Tenant context** stored in JWT token: tenantId.
- **TenantFilter** sets tenant context per request.
- **Database & Elasticsearch:** TenantId used to filter documents.
- **Access Enforcement:**

```
if (!tenantId.equals(jwtTenant) && !isAdmin) {
   return 403;
}
```

- Ensures **logical isolation** across tenants.

Production Readiness Analysis

1. Scalability (100× Growth in Documents & Traffic)

Horizontal Scalability

- **Stateless Services**: Spring Boot services remain stateless; scale via Kubernetes HPA.
- **API Layer**: Use load balancers (NGINX / ALB) with round-robin or least-connection strategy.
- **Search Engine**:
    - Elasticsearch/OpenSearch with:
        - Shard-based horizontal scaling
        - Index rollover and ILM (hot–warm–cold architecture)
- **Database**:
    - PostgreSQL with read replicas
    - Partition tables by tenant_id or time (range/hash partitioning)

Data Volume Growth

- Index documents asynchronously
- Archive cold documents to cheaper storage (S3-compatible object storage)
- Apply TTL policies for temporary or expired documents

2. Resilience & Fault Tolerance

Circuit Breakers & Retries

- **Resilience4j**:
    - Circuit breakers for DB, Redis, Elasticsearch
    - Retry with exponential backoff
    - Bulkhead isolation per tenant (optional)

Failover Mechanisms

- Database:
    - Primary–replica with automated failover (Patroni / RDS)
- Redis:
    - Redis Sentinel or Redis Cluster
- Search:
    - Elasticsearch replica shards across availability zones

Graceful Degradation

- If search engine fails:
    - Fallback to cached results
    - Partial results or delayed indexing
- If Redis fails:
    - Bypass cache, read from DB/search engine

3. Security

Authentication & Authorization

- **JWT-based authentication**
    - Roles: USER, ADMIN
    - Tenant ID embedded in JWT claims
- **Method-level security** (@PreAuthorize)
- **Tenant enforcement** at:
    - Filters
    - Repository queries
    - Index naming (documents_tenant1)

Data Security

- **Encryption in Transit**:
    - TLS everywhere (API, DB, Redis, Search)
- **Encryption at Rest**:
    - Database disk encryption
    - Encrypted Redis persistence
    - Encrypted Elasticsearch indices

API Security

- Rate limiting per tenant
- Input validation & schema validation
- Protection against:
    - SQL injection
    - Elasticsearch query abuse
- Secret management via Vault / Kubernetes Secrets

4. Observability

Metrics

- **Micrometer + Prometheus**
    - Request latency (p95/p99)
    - Error rates
    - Cache hit/miss ratio
    - Indexing lag
    - Per-tenant usage metrics

Logging

- Structured JSON logs
- Correlation IDs propagated across services
- Centralized logging (ELK / OpenSearch)

Distributed Tracing

- OpenTelemetry + Jaeger/Tempo
- Trace:
    - API → Service → Cache → DB/Search
- Tenant ID added as trace attribute

5. Performance Optimization

Database

- Index on:
    - tenant_id
    - document_id
    - Frequently filtered fields
- Use projections / DTOs instead of entities
- Batch inserts for indexing pipelines

Search Engine

- Pre-analyzed fields (edge n-grams for autocomplete)
- Avoid wildcard queries
- Filter-first, score-later query design
- Use search_after instead of deep pagination

Cache Optimization

- Cache hot queries (search results)
- Cache document metadata
- Avoid caching large blobs

6. Operations & Deployment

Deployment Strategy

- Containerized using Docker
- Kubernetes deployment with:
    - Rolling updates
    - Readiness & liveness probes
- Canary or blue-green deployments for risk-free releases

Zero-Downtime Updates

- Backward-compatible APIs
- Schema migrations using Flyway/Liquibase
- Dual-write or versioned indices for search reindexing

Backup & Recovery

- PostgreSQL:
    - Daily full backups + WAL archiving
- Elasticsearch:
    - Snapshot repositories (S3/GCS)
- Redis:
    - RDB/AOF backups
- Disaster recovery drills tested quarterly

7. SLA & High Availability (99.95%)

Availability Target

- 99.95% ≈ **~22 minutes downtime/month**

How It's Achieved

- Multi-AZ deployment
- Redundant instances for all critical components
- Auto-scaling based on traffic
- Health checks and self-healing pods
- Graceful degradation instead of total failure

SLA Safeguards

- Rate limiting to prevent abuse
- Tenant-level quotas
- Priority queues for critical tenants
- Alerting on:
    - Error spikes
    - Latency breaches
    - Indexing backlog

Summary

By combining horizontal scalability, strong tenant isolation, asynchronous processing, layered caching, resilient infrastructure, and deep observability, the system can safely scale 100× while maintaining security, performance, and a 99.95% availability SLA, making it fully production-ready.

Enterprise Experience Showcase

1. Similar Distributed System Built (Scale & Impact)

I worked on a multi-tenant document processing and search platform built using Spring Boot microservices, PostgreSQL, Redis, and Elasticsearch. The system supported tenant-isolated data ingestion, indexing, and full-text search with role-based access control. It handled asynchronous document indexing through message queues and exposed REST APIs for search and retrieval.

At peak, the platform served tens of thousands of documents per tenant and sustained several thousand requests per minute. By introducing caching at the query and metadata levels and decoupling indexing from API requests, we significantly improved responsiveness while maintaining strict tenant isolation and security.

2. Performance Optimization with Significant Improvement

In one system, search response times degraded under load due to repeated Elasticsearch queries for identical search patterns. I introduced a layered caching strategy using Redis, caching normalized search queries with tenant-specific keys and short TTLs. Additionally, I optimized Elasticsearch queries by replacing wildcard searches with analyzed fields and filter-first queries.

These changes reduced average search latency by over 60% and cut Elasticsearch query volume nearly in half. The system was able to handle traffic spikes without scaling the search cluster immediately, resulting in both performance gains and infrastructure cost savings.

3. Critical Production Incident Resolved

I handled a production incident where API response times suddenly spiked, eventually causing request timeouts across multiple services. Investigation showed a cascading failure triggered by a slow database query, which exhausted connection pools and propagated delays through dependent services.

I mitigated the issue by temporarily disabling a non-critical feature, increasing connection pool limits, and introducing circuit breakers to prevent further cascading failures. Post-incident, I added query optimizations, request timeouts, and resilience patterns (retries with backoff and bulkheads), which prevented similar outages in the future.

4. Architectural Decision Balancing Competing Concerns

In a multi-tenant system, we needed to balance strict data isolation with operational simplicity and cost efficiency. Instead of deploying separate databases per tenant, I designed a shared database with tenant-aware schemas, row-level filtering, and indexed tenant_id columns, combined with application-level enforcement and JWT-based tenant validation.

This approach maintained strong logical isolation and security while allowing easier scaling, simpler deployments, and lower operational overhead. It proved flexible enough to onboard new tenants quickly without compromising performance or maintainability.

AI tool usage

AI-assisted tools were used to accelerate development and improve code quality across the system lifecycle. These tools supported rapid API design, boilerplate generation, query optimization suggestions, and validation of security and architectural best practices. AI was also leveraged for log analysis, error diagnosis, and generating test scenarios, enabling faster troubleshooting and reduced development cycles.

All AI-generated outputs were reviewed and refined by engineers to ensure correctness, security, and alignment with production standards. AI tools complemented—not replaced—engineering judgment, helping improve productivity while maintaining accountability and code ownership.