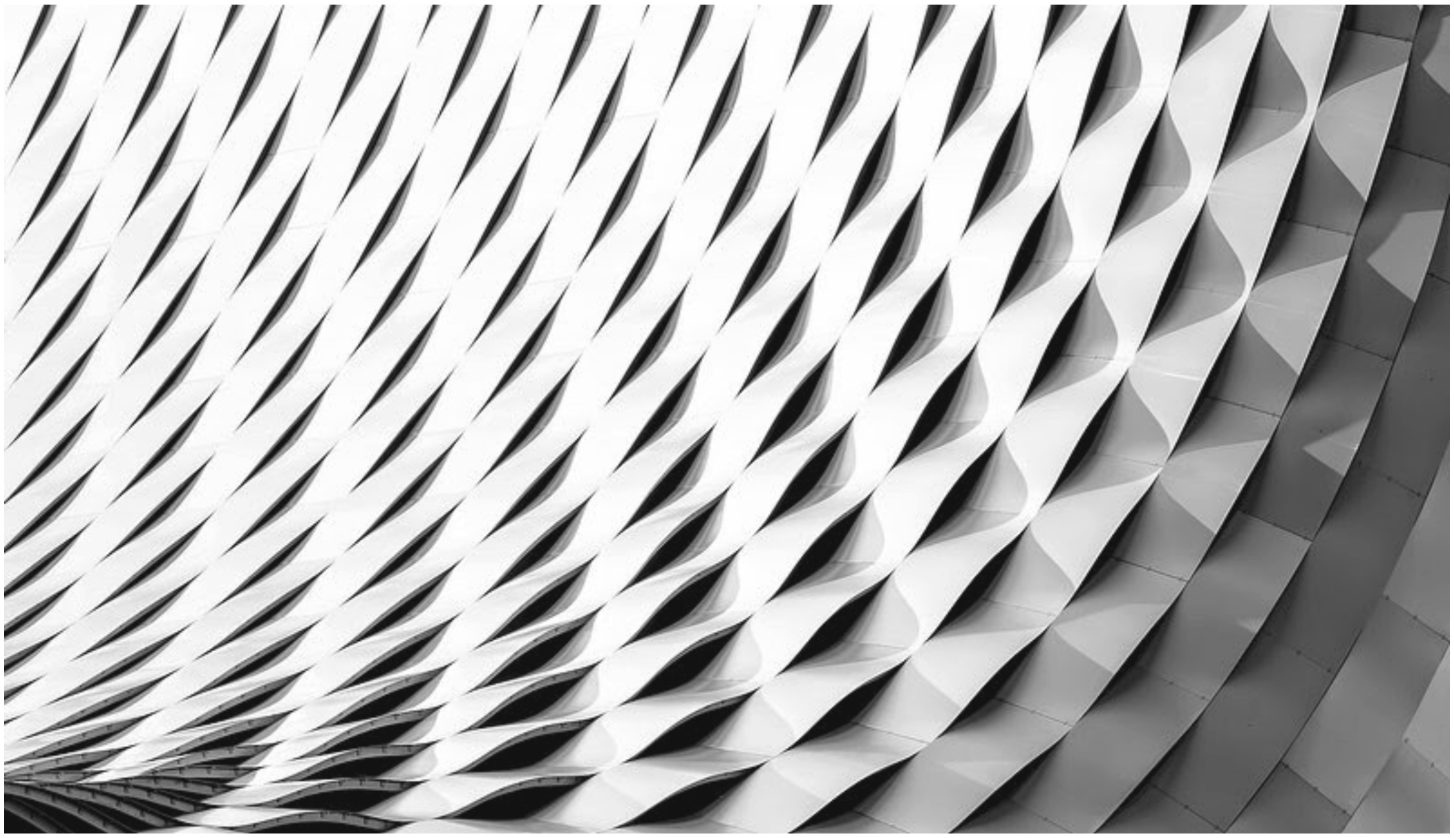


QUOTE-FLOW

Secure RFQ & Vendor Bidding Platform (Backend)

Abinash Mohanty

Full-Stack Developer

Tech Stack

Java 21 | Spring Boot | Spring Security | JWT | Angular 19 | H2 Database

February 2026

Project Statement

Problem Overview

In many service-based and procurement-driven businesses, customers often need to request quotations from multiple vendors before selecting the best offer. Traditionally, this process is handled through emails, spreadsheets, or informal communication channels, which leads to several problems such as lack of transparency, difficulty in tracking responses, security concerns, and inefficient decision-making.

There is a clear need for a **structured, secure, and role-based system** that can manage the entire quotation lifecycle—from request creation to contract finalization—while ensuring that only authorized users can perform specific actions.

Core Problem to Solve

Design and implement a backend system that enables:

- Customers create **Requests for Quotation (RFQs).**
- Vendors to view open RFQs and submit **competitive quotes.**
- Customers to review quotes and **finalize contracts.**
- Secure access control so that users can only perform actions allowed by their role.

Challenges in the Real World

While designing such a system, several real-world challenges must be addressed:

- **Authentication & Security:**
  Users must be securely authenticated, and the system should be stateless to scale efficiently.
- **Role-Based Authorization:**
  Customers and vendors have fundamentally different permissions, and these must be enforced strictly at the backend level.
- **Data Integrity:**
  Vendors should not be able to submit multiple quotes for the same RFQ, and RFQs should not accept quotes once closed.
- **Traceability & Debugging:**
  During development, it should be easy to inspect data, test APIs, and debug issues without relying on complex infrastructure.

Solution Approach (High-Level)

To address these problems, the backend was designed as a **RESTful Spring Boot application** with:

- JWT-based authentication for stateless security.
- Role-based access control enforced using Spring Security.
- A clean layered architecture (Controller → Service → Repository).
- An in-memory database (H2) for fast development and debugging.
- Clear separation between entities and API responses using DTOs.

This approach ensures the system is **secure, maintainable, scalable, and close to real-world enterprise applications**.

Core Features

User Registration & Login (JWT-based Authentication)

What it does:

The system allows users to register and log in using an email and password. Upon successful login, the backend issues a **JWT (JSON Web Token)**, which the frontend uses for all subsequent authenticated requests.

Why it is needed:

- Ensures only authenticated users can access protected APIs.
- Enables **stateless authentication**, meaning the backend does not store session data.
- Makes the application scalable and suitable for distributed systems.

Design decision:

- JWT contains:
    - userId as the subject
    - role as a claim
- Token is validated on every request using a security filter.

Role-Based Access Control (RBAC):

- Customer: Create RFQs, view submitted quotes, create contracts.
- Vendor: View open RFQs, submit quotes.

Without strict role enforcement:

- Vendors could create RFQs.
- Customers could submit quotes.
- Security and data integrity would break.

How it's enforced

- **Backend:** @PreAuthorize annotations + role checks via Spring Security.
- **Frontend:** Route guards prevent unauthorized navigation.

This ensures **defense-in-depth** (security at multiple layers).

RFQ (Request for Quotation) Management

What it does:

Customers can create RFQs by providing:

- Title.
- Description.

Each RFQ:

- Is linked to the customer who created it.
- Starts in the OPEN state.
- Can later transition to CONTRACT_CREATED.

Why it matters

RFQs are the **core business object** of the system. Everything else (quotes, contracts) revolves around them.

Key rules enforced

- Only **customers** can create RFQs.
- RFQs are tied to the authenticated user (not passed via request body).
- RFQs cannot be modified once a contract is created.

Quote Submission by Vendors

What it does:

Vendors can submit quotes against **open RFQs**, including:

- Quoted amount.
- Delivery duration.

Business rules enforced:

- Only vendors can submit quotes.
- One vendor can submit **only one quote per RFQ.**
- Quotes cannot be submitted if the RFQ is closed.

Why this is important because this ensures:

- Fair competition
- Clean data
- Predictable system behavior

These validations are implemented in the **service layer**, not the controller, which aligns with clean architecture principles.

Contract Creation

What it does:

After reviewing vendor quotes, a customer can select **one quote** and create a contract.

What happens internally:

- A contract entity is created.
- RFQ status changes from OPEN → CONTRACT_CREATED.
- Further quotes are blocked.

Why this design:

This models a real-world business flow:

- One RFQ → many quotes → one final contract

It also prevents edge cases like:

- Multiple contracts for the same RFQ
- Quotes being added after finalization

Layered Backend Architecture

Each feature follows a strict flow:

Controller → Service → Repository

Why this matters

- Controllers handle HTTP concerns only.
- Services contain business logic.
- Repositories handle database access.

This separation:

- Makes the code testable.
- Makes debugging easier.
- Matches enterprise-grade backend standards.

Architecture Flow

This section outlines a structural blueprint of the RFQ manager. These diagrams clarify the separation of concerns between the security, logic, and persistence layers, ensuring a standardized understanding of data flow and system boundaries.
Note: *All diagrams are made using mermaid open source extension.*

Controller Layer

Responsibility:

- Handles HTTP requests and responses.
- Performs request validation.
- Delegates business logic to the service layer.

What controllers do NOT do:

- They do not contain business rules.
- They do not talk directly to the database.

Example responsibilities:

- Accept RFQ creation requests.
- Expose endpoints for listing RFQs.
- Protect endpoints using role-based annotations (@PreAuthorize).

Service Layer

Responsibility:

- Contains all **business logic.**
- Enforces rules like:
    - Who can create RFQs?
    - Who can submit quotes?
    - When can contracts be created?
- Coordinates between multiple repositories if needed.

Why this layer is critical:

Business rules change more frequently than APIs or database schemas. Keeping logic in services allows:

- Easier updates.
- Cleaner controllers.
- Better testability.

Example logic handled here

- Extracting the logged-in user from the security context
- Role-based branching (Customer vs Vendor)
- Preventing duplicate quotes
- Updating RFQ status during contract creation

Repository Layer

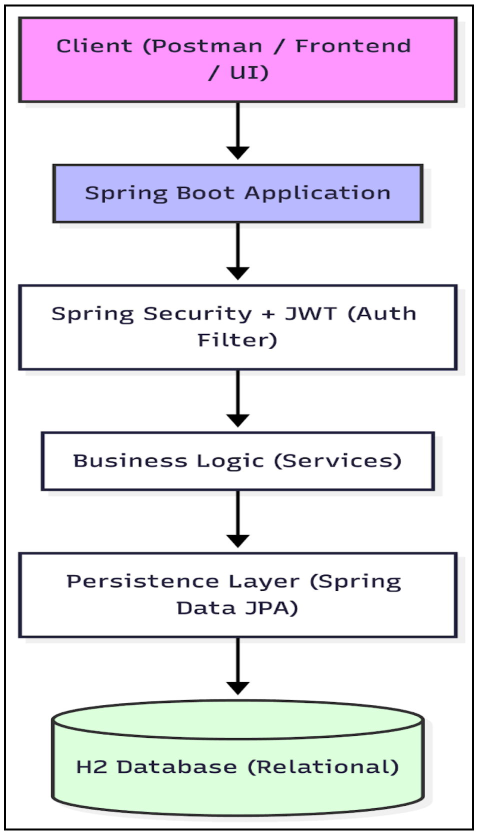
Responsibility:

- Handles all database interactions.
- Uses Spring Data JPA to abstract SQL queries.

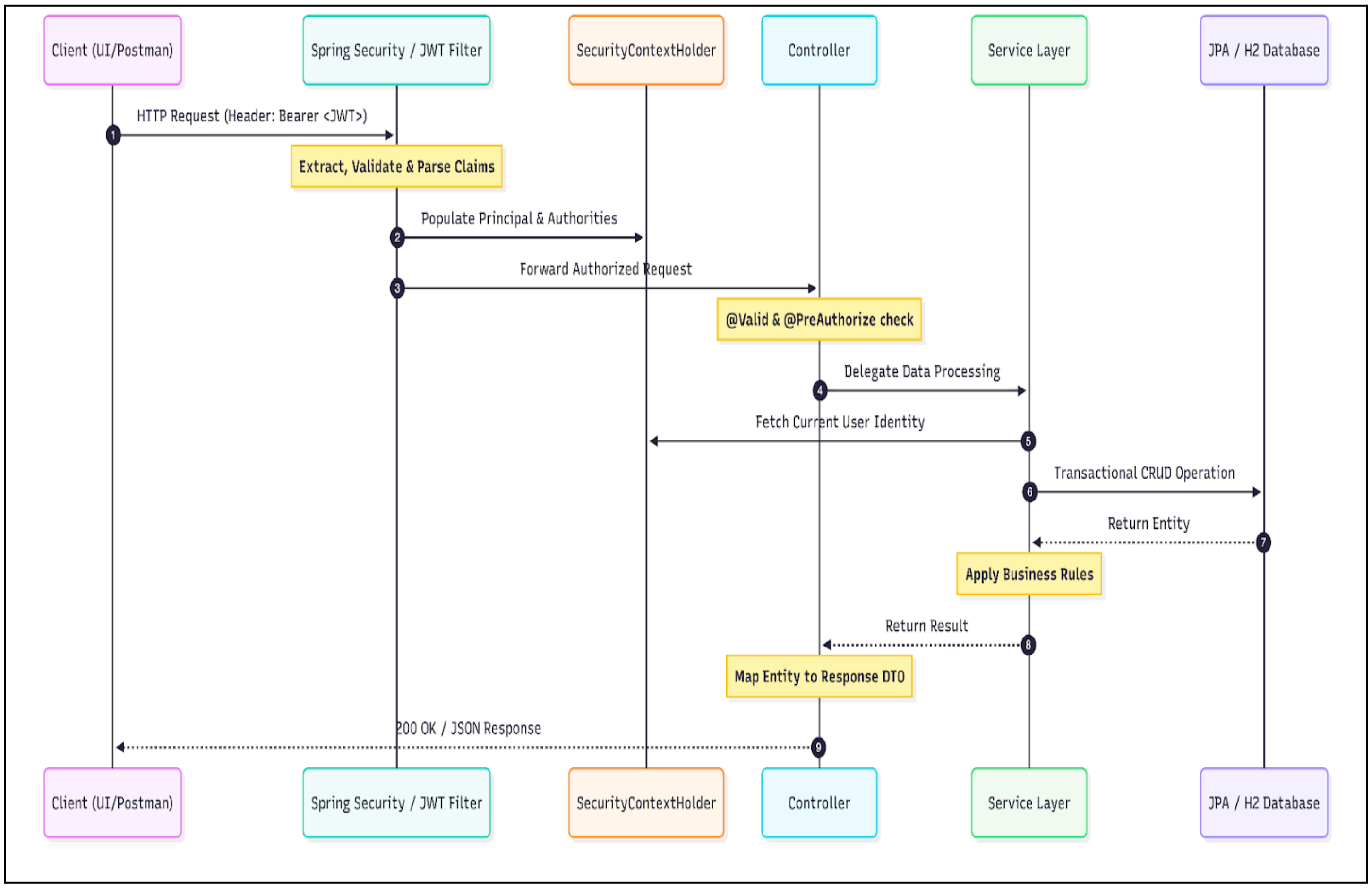
Key design choices:

- Repository methods are **intention-revealing**, such as:
    - findByCustomer
    - findByStatus
- No business logic inside repositories.

This ensures that persistence logic remains **simple and predictable**.


Client (Postman / Frontend / UI)
Spring Boot Application
Spring Security + JWT (Auth Filter)
Business Logic (Services)
Persistence Layer (Spring Data JPA)
H2 Database (Relational)

High Level Design


Client (UI/Postman)
Spring Security / JWT Filter
SecurityContextHolder
Controller
Service Layer
JPA / H2 Database
1 HTTP Request (Header: Bearer <JWT>)
Extract, Validate & Parse Claims
2 Populate Principal & Authorities
3 Forward Authorized Request
@Valid & @PreAuthorize check
4 Delegate Data Processing
5 Fetch Current User Identity
6 Transactional CRUD Operation
7 Return Entity
Apply Business Rules
8 Return Result
Map Entity to Response DTO
9 200 OK / JSON Response

Detailed Request Flow (END-TO-END)

<u>Security Design</u>

Security was treated as a **core design concern**, not an afterthought. The system implements **stateless, role-based security** using JWT and Spring Security, closely aligning with modern backend architectures.

Why JWT (JSON Web Token)?

Problem with traditional session-based auth:

- The server must store session data.
- Harder to scale horizontally.
- Tight coupling between client and server.

Why JWT was chosen

- **Stateless authentication**: no session stored on the server.
- Each request carries its own authentication proof.
- Works seamlessly with REST APIs and frontend frameworks.

What the JWT contains

- **Subject (sub)** → userId
- **Custom claim** → user role (CUSTOMER / VENDOR)
- **Expiration time** → prevents infinite validity

This allows the backend to:

- Identify *who* the user is
- Decide *what* the user is allowed to do
- Do so **without database lookups on every request**

Authentication Flow (Step-by-step)

1. The user logs in with an email and password.
2. Spring Security authenticates credentials.
3. Backend generates a JWT containing:
    - userId
    - role
4. Token is returned to the frontend.
5. Frontend stores a token in localStorage.
6. Token is sent with every secured API request.

Backend Security Enforcement

JWT Authentication Filter:

A custom filter runs **before** controller execution. Its responsibilities:

- Extract JWT from Authorization header
- Validate token signature and expiration
- Extract userId and role
- Populate SecurityContext

Role-Based Authorization (@PreAuthorize)

Endpoints are protected using role-based annotations such as:

- Customers can create RFQs
- Vendors can submit quotes
- Customers can create contracts

This ensures:

- Unauthorized requests are blocked **before business logic executes**
- Role checks are explicit and easy to audit

This design prevents accidental exposure of sensitive operations.

Challenges Faced During Development

While building the system, multiple non-trivial issues were encountered across security, data flow, and frontend–backend integration. Each challenge revealed gaps in understanding and led to design improvements.

This section documents what went wrong, why it happened, and what was learned.

Challenge 1: Repeated 500 Internal Server Error

What happened:

Several APIs initially failed with generic 500 errors, especially during:

- RFQ creation
- Quote submission
- Contract creation

Root cause:

- Missing repository injections in service classes
- Repository methods referenced in services were not declared
- Entity relationships were assumed but not fully wired

Example issue:

Service logic attempted to call methods like:

- findByCustomer
- findByStatus

Without those methods being defined in the repository.

What this taught:

- Spring fails at runtime, not compile time, for repository method mismatches
- Service-layer code must be verified against repository contracts

Challenge 2: Role Mismatch Between JWT and Security Context

What happened:

Even when using a vendor token, the backend responded with:

> "Only customers can create contracts"

Root cause:

- Role stored in JWT did not match Spring Security's expected format
- Spring expects roles as `ROLE_<NAME>`
- Token initially stored roles inconsistently

Fix applied:

- Normalized role format everywhere
- Ensured JWT claim → SecurityContext authority mapping was correct

Lesson learned:

Authentication may succeed, but authorization can silently fail if role formats are inconsistent.

Challenge 3: Interceptor Breaking Login Requests

What happened:

- Login requests started failing
- Backend rejected requests even before authentication

Root cause:

- HTTP Interceptor attached JWT headers to *every* request
- Login endpoints do not expect or require tokens
- Invalid or expired tokens caused request rejection

Fix applied:

- Interceptor logic updated to exclude auth endpoints
- Token added only when present and required

Lesson learned:

Global interceptors must be written defensively. Not every request should be authenticated.

Challenge 4: Frontend Routing Not Working as Expected

What happened:

- Clicking “Register” did nothing
- Routes worked only when entered manually in the browser

Root cause:

- Missing router outlet in root component
- Incorrect assumptions from older Angular versions
- Standalone component routing behaves differently

Fix applied:

- Verified route configuration
- Ensured router outlet was present
- Used proper Angular 19 routing patterns

Lesson learned:

Frontend routing is declarative, not implicit. Without router outlets, navigation silently fails.

Challenge 5: Token Appeared in Response but Not in Local Storage

What happened:

- Login API returned a token (200 OK)
- Token did not appear in browser storage

Root cause:

- Token save logic was not executed correctly
- Subscription handling was incomplete
- Misplaced responsibility between services

Fix applied:

- Centralized token handling in a dedicated service
- Explicitly stored token on successful login

Lesson learned:

Successful HTTP responses do not guarantee state persistence. Front-end logic must explicitly manage the application state.

What I'd Improve Next (Production Readiness & Scalability)

Pagination & Sorting

Current state:

- RFQs and quotes are fetched as full lists
- Suitable only for small datasets

Improvement:

- Add pagination using PageRequest
- Support sorting by:
  - Creation date
  - Status
  - Amount (for quotes)

Why it matters:

- Prevents performance issues with large datasets
- Reduces payload size
- Improves frontend UX

Search & Filtering

Current state:

- RFQs are fetched based only on role

Improvement:

- Filter RFQs by:
  - Status (OPEN, CLOSED)
  - Date range
  - Customer/vendor

Why it matters:

- Real users don't scroll through long lists
- Enables meaningful dashboards

Improved Authentication & Token Handling

Current state:

- Single JWT with fixed expiry
- No refresh mechanism

Improvement:

- Add refresh tokens
- Short-lived access tokens
- Logout token invalidation (blacklisting)

Why it matters:

- Better security
- Reduced risk of token leakage
- Industry-standard auth flow

Production Database (PostgreSQL)

Current state:

- H2 in-memory database
- Data lost on restart

Improvement:

- Migrate to PostgreSQL
- Use Flyway or Liquibase for migrations

Why it matters:

- Data persistence
- Schema versioning
- Real production compatibility

Dockerization & Deployment

Current state:

- Local development only

Improvement:

- Dockerize backend and frontend
- Use Docker Compose
- Deploy to cloud (AWS / GCP / Render)

Why it matters;

- Environment consistency
- Easier onboarding
- CI/CD readiness

Automated Testing

Current state:

- Manual testing via Postman

Improvement:

- Unit tests for services
- Integration tests for controllers
- Security tests for role access

Why it matters:

- Regression prevention
- Confidence in changes
- Production readiness