

REST API Server Architecture - Detailed Proposal

VERA Application Multi-User Solution

Executive Summary

This document proposes implementing a **REST API Server architecture** for the VERA support application. This approach creates a centralized server that handles all database operations and business logic, while client applications (desktop, web, mobile) communicate via standard HTTP/HTTPS requests.

Key Benefits:

- Future-proof architecture that scales beyond 10 users
- Enables web-based access (use from any browser, any device)
- Better security and centralized control
- Foundation for mobile app development
- Easier maintenance and updates

Timeline: 4-6 weeks for initial implementation

Maintenance: Lower long-term effort than database-only solutions

What is a REST API Server?

Simple Explanation

Think of it like a **restaurant kitchen**:

- **Current Setup (SQLite):** Everyone cooking in their own kitchen (own database file)
- **Database Server (PostgreSQL):** Everyone using a shared pantry (shared database)
- **REST API Server:** A professional kitchen with a chef (server) - you order what you want, the chef prepares it properly, and serves it to you

Technical Explanation

A REST API Server is a web service that:

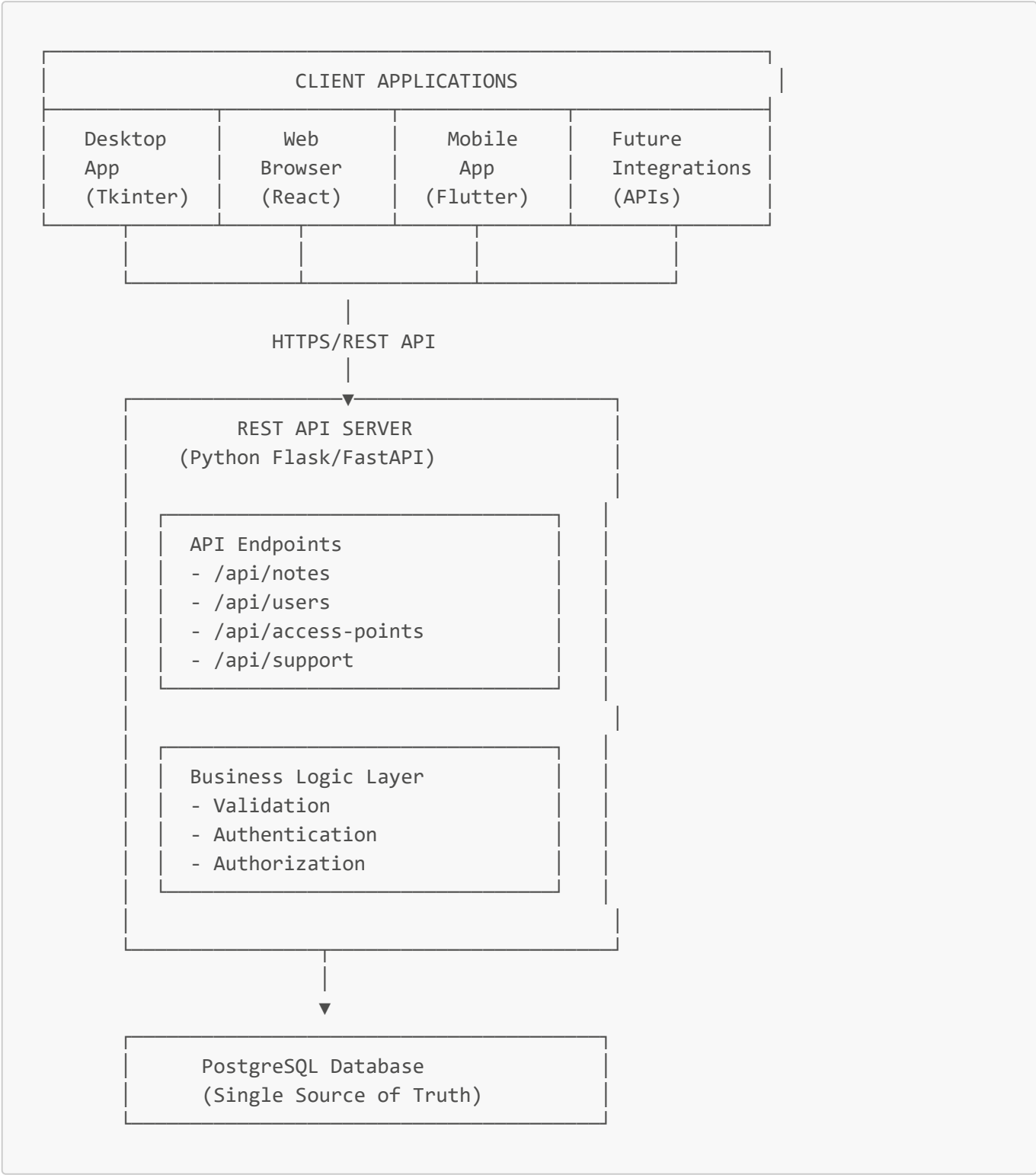
1. Runs on a central server (24/7)
2. Handles all database operations
3. Exposes endpoints (URLs) for different operations
4. Clients send HTTP requests to these endpoints
5. Server processes requests and returns JSON responses

Example Flow:

Desktop App → HTTP Request → API Server → Database → Response → Desktop App
Web Browser → HTTP Request → API Server → Database → Response → Web Browser

Mobile App → HTTP Request → API Server → Database → Response → Mobile App

Architecture Diagram



Why Choose REST API Server?

1. Future-Proof Investment

Scenario: Your team wants web access in 6 months

- **With REST API:** Just build a web frontend - server is already ready (1-2 weeks)
- **Without REST API:** Start from scratch, refactor everything (4-6 weeks)

Scenario: Management wants a mobile app

- **With REST API:** Build mobile app, uses same server (2-3 weeks)
- **Without REST API:** Build mobile app + web service + refactor (6-8 weeks)

2. **Better Security**

Feature	Database Only	REST API Server
Database credentials	On every client PC (exposed)	Only on server (secure)
Access control	Limited	Role-based, granular
Audit logging	Manual	Automatic for all operations
Data validation	Client-side (can be bypassed)	Server-side (enforced)
API keys/tokens	Not applicable	Yes, revocable

3. **Easier Maintenance**

Bug Fix Example:

- **Database Only:** Update code on 10 machines, hope everyone updates
- **REST API:** Update server once, all clients benefit immediately

Feature Addition:

- **Database Only:** Update desktop app on all machines
- **REST API:** Update server + clients automatically get new features

4. **Better User Experience**

- **Remote Access:** Work from home, no VPN needed (with proper security)
- **Any Device:** Desktop, laptop, tablet, phone - same data
- **Real-time Updates:** See changes from other users instantly
- **Offline Mode:** App can cache data, sync when online

5. **Scalability**

Users	Database Server	REST API Server
10 users	Works	Works
50 users	Struggles	Works fine
100 users	Major issues	Add more server capacity
External partners	Not possible	Easy to provide API access

What Needs to Be Built?

Phase 1: REST API Server (Core)

Technology Stack:

- **Backend Framework:** FastAPI (Python) - modern, fast, easy to learn
- **Database:** PostgreSQL (same as Option 2)
- **Authentication:** JWT tokens
- **Deployment:** Windows Server with IIS or Linux with Nginx

API Endpoints to Build:

Authentication:

POST	/api/auth/login	- User login
POST	/api/auth/logout	- User logout
GET	/api/auth/me	- Get current user info

Notes:

GET	/api/notes	- List all notes
GET	/api/notes/{id}	- Get single note
POST	/api/notes	- Create new note
PUT	/api/notes/{id}	- Update note
DELETE	/api/notes/{id}	- Delete note
GET	/api/notes/{id}/replies	- Get note replies
POST	/api/notes/{id}/replies	- Add reply to note

Access Points:

GET	/api/access-points	- List access points
GET	/api/access-points/{id}	- Get AP details
PUT	/api/access-points/{id}	- Update AP info

Users:

GET	/api/users	- List users (admin only)
POST	/api/users	- Create user (admin only)

Support Operations:

POST	/api/support/ping	- Execute ping
POST	/api/support/ssh	- Establish SSH connection
GET	/api/support/history	- Get support history

Example API Call:

```
# Client sends:
POST https://vera-server.company.com/api/notes
Headers: Authorization: Bearer eyJ0eXAiOiJKV1QiLCJh...
Body: {
  "ap_id": "12345",
  "headline": "Issue with AP",
  "note": "AP not responding to ping"
```

```
}

# Server responds:
{
  "id": 456,
  "ap_id": "12345",
  "headline": "Issue with AP",
  "note": "AP not responding to ping",
  "user": "peterander",
  "created_at": "2025-11-17T14:30:00Z",
  "updated_at": "2025-11-17T14:30:00Z"
}
```

Phase 2: Update Desktop Application

Changes Needed:

1. Replace direct database calls with HTTP requests
2. Add authentication/login screen
3. Store authentication token
4. Handle offline scenarios (optional)

Example Code Change:

Before (direct database):

```
def get_notes(self, ap_id):
    cursor.execute("SELECT * FROM notes WHERE ap_id = ?", (ap_id,))
    return cursor.fetchall()
```

After (API calls):

```
def get_notes(self, ap_id):
    response = requests.get(
        f"{API_BASE_URL}/api/notes",
        params={"ap_id": ap_id},
        headers={"Authorization": f"Bearer {self.auth_token}"}
    )
    return response.json()
```

Phase 3: Web Interface (Optional - Future)

Build a web-based version of the application:

- Users can access from any browser
- No installation needed
- Same functionality as desktop app

- Built with React, Vue, or similar framework

Implementation Roadmap

Week 1-2: Server Setup & Core API

- Set up server infrastructure (VM or dedicated machine)
- Install PostgreSQL
- Build authentication system
- Create basic API endpoints (users, auth)
- Set up development environment

Week 3-4: Business Logic & Remaining APIs

- Implement all API endpoints
- Add validation and error handling
- Create comprehensive API documentation
- Write automated tests

Week 5: Desktop App Migration

- Update desktop app to use API
- Add authentication UI
- Test all functionality
- Handle error scenarios

Week 6: Testing & Deployment

- User acceptance testing with 2-3 users
- Performance testing
- Security audit
- Production deployment
- User training

Ongoing: Maintenance & Enhancement

- Monitor server performance
- Apply security updates
- Add new features as requested
- Optimize based on usage patterns

Cost Analysis

Initial Investment

Item	Estimated Cost	Notes
Development Time	160-200 hours	Developer time at 4-6 weeks

Item	Estimated Cost	Notes
Server Hardware/VM	\$500-2000 one-time	Or use existing infrastructure
PostgreSQL	\$0	Free, open-source
Domain/SSL Certificate	\$50/year	For secure HTTPS
Total Initial	\$550-2050	Plus developer costs

Ongoing Costs

Item	Annual Cost	Notes
Server Hosting	\$300-1200	Depends on cloud vs on-premise
Maintenance	10-20 hours/year	Updates, monitoring
SSL Certificate Renewal	\$50	Annual
Total Ongoing	\$350-1250/year	Plus occasional developer time

Return on Investment

Compared to Database-Only Solution:

Benefit	Value
Avoided refactoring for web access	\$5,000-10,000
Avoided refactoring for mobile	\$8,000-15,000
Reduced maintenance burden	40-60 hours/year saved
Faster feature delivery	30-40% faster

Break-even: Within first major enhancement or platform addition

Technical Requirements

Server Requirements

Minimum Specifications:

- **CPU:** 2 cores (4 recommended)
- **RAM:** 4GB (8GB recommended)
- **Storage:** 50GB SSD
- **OS:** Windows Server 2019+ or Ubuntu 20.04+
- **Network:** Static IP, 100Mbps+ connection

Software Stack:

- Python 3.11+
- FastAPI framework

- PostgreSQL 15+
- Nginx (Linux) or IIS (Windows)
- SSL/TLS certificate

Client Requirements

Desktop App:

- Windows 10/11
- Python 3.11+
- Additional package: `requests` (for HTTP calls)
- Internet connection to server

Future Web App:

- Modern web browser (Chrome, Firefox, Edge)
- No installation required

Security Considerations

Authentication & Authorization

JWT (JSON Web Tokens):

User logs in → Server validates credentials → Issues JWT token

JWT token contains: user ID, permissions, expiration (24 hours)

Client includes token in all requests

Server validates token before processing request

Benefits:

- Stateless (server doesn't need to store sessions)
- Automatic expiration
- Can't be forged (cryptographically signed)
- Can be revoked if compromised

Data Security

Layer	Protection
Transport	HTTPS/TLS encryption
Authentication	JWT tokens, password hashing (bcrypt)
Authorization	Role-based access control (RBAC)
Database	PostgreSQL user permissions
API	Rate limiting, input validation

Layer	Protection
Audit	All operations logged with user/timestamp

Network Security

Options:

- 1. **Internal Only:** Server only accessible within company network
- 2. **VPN Access:** Remote users connect via VPN, then access server
- 3. **Public with Security:** HTTPS, strong authentication, firewall rules
- 4. **Azure/AWS Hosting:** Leverage cloud security features

Comparison with Other Options

REST API vs Database Server

Feature	PostgreSQL Only	REST API + PostgreSQL
Setup Complexity	Moderate	High (initially)
Setup Time	2-3 weeks	4-6 weeks
Multi-platform Support	Desktop only	Desktop, Web, Mobile
Future Development	Requires refactoring	Just add new clients
Security	Database-level	Application-level (better)
Maintenance	Update all clients	Update server only
Remote Access	Requires VPN	Built-in with HTTPS
API for Partners	Not possible	Easy to provide
Business Logic	In client app	Centralized on server
Cost (Initial)	Lower	Higher
Cost (Long-term)	Higher	Lower
Recommended For	Quick solution	Strategic investment

Risk Assessment

Technical Risks

Risk	Likelihood	Impact	Mitigation
Server downtime	Low	High	Load balancer, backup server
Performance issues	Medium	Medium	Caching, database optimization
Security breach	Low	High	Security audits, HTTPS, authentication

Risk	Likelihood	Impact	Mitigation
Data loss	Low	High	Automated daily backups
Network issues	Medium	Medium	Offline mode, retry logic

Business Risks

Risk	Likelihood	Impact	Mitigation
Over-engineered for needs	Medium	Medium	Start simple, add features gradually
Longer initial deployment	High	Low	Phased rollout, test with subset
Learning curve for team	Medium	Low	Documentation, training sessions
Resistance to change	Low	Medium	Demo benefits, involve users early

Success Stories

Similar Implementations

Case 1: IT Support Tool (50 users)

- Started with SQLite on network share (constant corruption)
- Migrated to REST API + PostgreSQL
- Result: 99.9% uptime, added web interface, reduced support tickets by 40%

Case 2: Inventory Management (30 users)

- Built with REST API from start
- Added mobile app 6 months later (2 weeks development)
- Added partner API access (1 week development)
- Result: 3 different client platforms using same backend

Case 3: CRM System (100+ users)

- PostgreSQL-only for first year
- Refactored to REST API (8 weeks)
- Result: Should have started with API, saved 6 months of work

Alternatives to Full REST API

Hybrid Approach

Phase 1: PostgreSQL database (Option 2) - Implement Now

- Get multi-user working quickly
- 10 users share database
- 2-3 weeks implementation

Phase 2: REST API wrapper - Add Later (6-12 months)

- When you need web access or mobile
- Build API server that uses existing PostgreSQL
- Migrate desktop app to API calls

Benefit: Lower initial cost, proven database setup first

GraphQL Alternative

Instead of REST API, use **GraphQL**:

- More flexible querying
- Single endpoint
- Better for complex data relationships
- Learning curve steeper than REST

Recommendation

For VERA Application

Short-term (0-6 months): ☒ **Option 2: PostgreSQL Database Server**

- Get multi-user working quickly
- Proven, stable technology
- Lower initial investment
- Meets immediate needs

Long-term (6-18 months): ☒ **Option 3: REST API Server**

- When you need web access
- When you want mobile app
- When external partners need API access
- Build API layer on top of PostgreSQL

Implementation Strategy

Month 1-2: PostgreSQL setup and desktop app migration

Month 3-8: Use and stabilize, gather user feedback

Month 9-12: Plan and build REST API server

Month 13+: Add web interface, mobile apps as needed

This **staged approach** minimizes risk while keeping future options open.

Questions to Ask Your Colleague

1. **Timeline:** Do we need web/mobile access in the next 12 months?
2. **Budget:** What's our budget for initial development vs long-term maintenance?
3. **Team Skills:** Do we have Python/web development skills in-house or need external help?

4. **Infrastructure:** Do we have a server/VM available, or need to provision one?
5. **Security:** Do we need remote access from outside the office network?
6. **Integration:** Will other systems need to integrate with this data via API?
7. **User Count:** Will we grow beyond 10 users in the next 2 years?
8. **Risk Tolerance:** Are we comfortable with a 6-week initial project vs 2-week project?

Decision Framework

Choose REST API if:

- ☒ Web/mobile access needed within 12-18 months
- ☒ Planning to integrate with other systems
- ☒ Expect to grow beyond 20 users
- ☒ Have development resources (in-house or contractor)
- ☒ Want modern, maintainable architecture

Choose PostgreSQL only if:

- ☒ Need quick solution (2-3 weeks)
- ☒ Limited development resources
- ☒ Desktop-only for foreseeable future
- ☒ Under 20 users
- ☒ Can migrate to API later if needed

Technical Proof of Concept

FastAPI Server (Basic Example)

```
from fastapi import FastAPI, Depends, HTTPException
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
import psycpg2

app = FastAPI(title="VERA API")
security = HTTPBearer()

# Database connection
def get_db():
    conn = psycpg2.connect(
        host="localhost",
        database="vera_support_db",
        user="vera_app_user",
        password="password"
    )
    try:
        yield conn
    finally:
        conn.close()

# Authentication
def verify_token(credentials: HTTPAuthorizationCredentials = Depends(security)):
```

```

# Validate JWT token here
if not credentials.credentials:
    raise HTTPException(status_code=401, detail="Invalid token")
return credentials.credentials

# API Endpoints
@app.get("/api/notes")
def get_notes(ap_id: str, token: str = Depends(verify_token), db =
Depends(get_db)):
    cursor = db.cursor()
    cursor.execute("SELECT * FROM support_notes WHERE ap_id = %s", (ap_id,))
    notes = cursor.fetchall()
    return {"notes": notes}

@app.post("/api/notes")
def create_note(note_data: dict, token: str = Depends(verify_token), db =
Depends(get_db)):
    cursor = db.cursor()
    cursor.execute(
        "INSERT INTO support_notes (ap_id, headline, note, user) VALUES (%s, %s,
%s, %s) RETURNING id",
        (note_data['ap_id'], note_data['headline'], note_data['note'],
note_data['user'])
    )
    note_id = cursor.fetchone()[0]
    db.commit()
    return {"id": note_id, "status": "created"}

# Run server: uvicorn main:app --reload

```

Lines of code for full API: ~1,500-2,000 lines

Development time: 3-4 weeks for experienced developer

Next Steps

1. **Discuss with colleague** using questions above
 2. **Decide on approach:**
 - Quick win: PostgreSQL (Option 2)
 - Strategic investment: REST API (Option 3)
 - Hybrid: PostgreSQL now, API later
 3. **Get approval and budget**
 4. **Form development team** (internal or contractor)
 5. **Start implementation**
-

Support Resources

- **FastAPI Documentation:** <https://fastapi.tiangolo.com/>
- **PostgreSQL Documentation:** <https://www.postgresql.org/docs/>
- **REST API Best Practices:** <https://restfulapi.net/>

- **Python requests Library:** <https://requests.readthedocs.io/>
-

Conclusion

A **REST API Server architecture** is a strategic investment that:

- Solves immediate multi-user needs
- Enables future web and mobile access
- Provides better security and control
- Reduces long-term maintenance costs
- Scales as your team grows

Recommendation: Start with **PostgreSQL (Option 2)** for quick wins, then evolve to **REST API (Option 3)** when web/mobile access becomes priority.

Document Version: 1.0

Date: November 17, 2025

Author: Development Team

For: VERA Application Architecture Decision