# Technical Analysis – Note Taking API

## 1. Approach Overview

The application is built as a layered REST API using Express.js and TypeScript. I've kept the architecture pretty straightforward with clear separation of concerns:

- Controllers handle HTTP request/response logic
- Services contain business logic (versioning, access control, caching)
- Models (Sequelize ORM) represent domain entities and relationships
- Middleware manages cross-cutting concerns (auth, caching, uploads)
- Infrastructure layer (MySQL, Redis) is abstracted via Singleton services

This structure makes the code easier to maintain and test, and it's not overly complex for what we're trying to achieve.

## 2. Key Design Decisions & Reasoning

### Database & ORM (MySQL + Sequelize)

I went with Sequelize because it gives us good developer productivity without completely abstracting away the database. The relational model fits naturally here—users, notes, versions, and shares all have clear relationships.

I enabled soft deletes (paranoid: true) to preserve history, and added full-text indexes on title and content for search functionality without needing something like Elasticsearch right away.

Why MySQL? It gives us strong consistency guarantees, which matters for things like note updates and version tracking. The transactional support is reliable and performance is predictable.

### Version Control Strategy

The way versioning works:

- notes table stores the current state
- note_versions table stores historical snapshots
- Every create, update, or revert operation creates a new version record

This keeps the main notes table lean while still giving us a complete audit trail and the ability to revert to any previous version.

## Concurrency Handling (Optimistic Locking)

Each note has a version field. When a client wants to update a note, they need to provide the current version number. If it doesn't match what's in the database, we return a 409 Conflict.

I chose optimistic locking over pessimistic locking because it scales better for read-heavy workloads and avoids database-level locks. The trade-off is that clients need to handle conflicts, but that's generally acceptable for this type of application.

## Authentication & Session Management

Pretty standard JWT setup:

- Access tokens for stateless authentication
- Refresh tokens stored in the database with expiry timestamps
- Token rotation handled via a dedicated service

This balances security with user experience users don't get logged out constantly, but we still have server-side control over sessions.

## Caching Strategy (Redis)

Redis caches:

- Single note retrieval
- Paginated note lists
- Search results
- Version history

Cache gets invalidated on create, update, delete, and revert operations.

The caching layer significantly reduces database load for read operations, which I expect to be way more frequent than writes. The invalidation strategy is straightforward enough that it shouldn't cause consistency issues.

## Note Sharing & Permissions

Many-to-many relationship via note_shares table with explicit permissions (READ, EDIT). All the ownership and access checks are centralized in a service, so the logic stays consistent across all endpoints.

## Media Attachments

Files are stored as BLOBs with a 5MB limit per file. They're tightly coupled with notes via foreign keys.

I know storing BLOBs in the database isn't always ideal, but for the scope of this assessment it keeps things simple and transactional. More on this in the trade-offs section.

# 3. Use of Design Patterns

## Singleton Pattern

Used for:

- Database connection
- Redis client
- Environment configuration
- Cache service

This ensures we're not creating multiple connection instances and makes lifecycle management simpler.

# 4. Trade-offs & Future Improvements

## Scalability

What works well:

- Optimistic locking and Redis caching handle concurrent reads effectively
- Service-based architecture makes it easy to scale horizontally

What could become a bottleneck:

- Full-text search on MySQL might struggle at very large scale
- BLOB storage in the database isn't great for high media volume

Future path: Relatively easy to migrate to Elasticsearch for search and object storage for media without major refactoring.

## Media Storage (Current vs. Production-Ready)

Current approach: Files are stored as BLOBs in the database. Uploads happen in-memory and get persisted with the note data.

Trade-offs:

- Simple deployment and strong transactional consistency
- Database size and I/O overhead increases with media volume
- Scaling the database becomes more expensive

How I'd improve this for production:

Use object storage like S3 with presigned URLs:

- Clients upload directly to S3 using presigned URLs
- Store only metadata (URL, key, size, mime type) in the database
- This offloads binary transfers from our application server
- Reduces database storage costs significantly
- Opens up CDN integration for faster media delivery
- Presigned URLs improve both performance and security by avoiding proxy uploads

For this assessment, I kept it simple with database storage, but the architecture makes it straightforward to swap in S3 later.

## Version Storage (Current vs. Optimized)

Current approach: Every version stores the complete title and content as a snapshot in note_versions.

Trade-offs:

- Simple retrieval and reliable reverts
- No complex reconstruction logic needed
- Storage grows linearly with version count
- Inefficient for large notes with small changes

How I'd optimize this for production:

Implement delta-based versioning store only what changed:

- Use text diffs (line-based or character-based)
- Track modified content ranges using offsets

Benefits:

- Drastically reduces storage footprint for frequently edited notes
- Scales better long-term for heavy versioning workloads

Trade-offs of delta approach:

- More complex read and revert operations
- Content needs to be reconstructed by applying diffs sequentially
- Conflict resolution becomes trickier

For this implementation, I prioritized correctness and maintainability with full snapshots. The migration path to delta-based versioning is clear if the use case demands it.

### Performance

What's optimized:

- Indexed queries on common lookup patterns
- Redis caching for read-heavy operations
- Current-state storage keeps the main table fast

What increases over time:

- Version history storage
- Cache invalidation complexity as features grow

### Maintainability

What I'm happy with:

- Clear separation of concerns makes testing easier
- TypeScript catches a lot of issues at compile time
- Centralized services keep business logic consistent

Honest trade-off:

- Slightly higher initial complexity compared to a single-file Express app
- Worth it for anything beyond a prototype

## 5. Summary

This solution focuses on:

- Correctness – Versioning and concurrency safety work as expected
- Performance – Caching and indexed queries handle read-heavy workloads well
- Maintainability – Clean architecture makes future changes manageable
- Security – JWT, refresh tokens, and password hashing follow best practices

The current design meets all the required and bonus criteria. While there's room for optimization (S3 for media, delta-based versioning), the architecture provides a solid foundation that can scale with the product.

The design choices reflect a balance between delivering working software now and keeping the door open for future improvements without major rewrites.