SHAJARA TECHNICAL DECISION LOG

Technical Decision Log

Document 3-5 key technical decisions you made, including:

The problem you were solving

My problem statement is I am making a personal journal that allows users to record their day to day experiences , get their mood for the day, create mood categories and create drafts to allow them to continue with a journal entry on a later date.

The app should then be able to provide sentiments to each user based on their previous entries and provide other journal analytics.

Options you considered

Whether to do a monolith or a microservices application

Whether to use my own auth or use a third party (clerk)

What DB to use

Your chosen approach and rationale

1.Why i chose monolith instead of microservice

1. Simplicity and Speed of Development

- A monolith has a single codebase, making it easier to set up, develop, and deploy.
- Teams can focus on building features without the complexity of managing multiple services, APIs, and inter-service communication.

2. Lower Operational Overhead

- Debugging and monitoring a monolith are straightforward compared to tracing issues across distributed microservices.
- Deployment is simpler because you only need to deploy one application rather than coordinating deployments of multiple services.

3. Cost Efficiency

- Monoliths typically require fewer resources since you avoid duplicating infrastructure for each service.
- There's no need for additional infrastructure like API gateways, load balancers for every service, or database-per-service setups.

4. Easier for Early Prototyping

- Monoliths are ideal for prototyping and validating ideas, as you can iterate quickly without the complexity of managing multiple services.
- If the product succeeds and grows, you can gradually modularize the monolith or transition parts into microservices.

When to Consider Microservices Instead?

- **Scalability Needs**: If you predict rapid growth requiring independent scaling of components.
- **Complex Domains**: When your app encompasses distinct domains that would benefit from independent teams or services.
- **Distributed Teams**: If different teams manage specific features or services.

2. Why i chose clerk over implementing my own auth

1. Ease of Integration

- Clerk provides prebuilt components and middleware tailored for Next.js, making it quick and easy to set up authentication without reinventing the wheel.

2. Security

- Clerk adheres to industry-standard security practices, such as encryption, token-based authentication, and secure session management. This ensures your app is protected against vulnerabilities.

3. Feature-Rich

- It supports multiple authentication methods, including email, social logins (Google, Facebook, GitHub, etc.), and even Web3 authentication. This flexibility enhances user experience.

4. Scalability

- Clerk is designed to handle authentication for apps of all sizes, so you don't have to worry about scaling your custom solution as your user base grows.

5. Time and Cost Efficiency

- Building and maintaining a custom authentication system can be time-consuming and expensive. Clerk saves you from managing infrastructure, updates, and compliance.

6. Customization

- While Clerk provides out-of-the-box solutions, it also allows customization to match your app's branding and specific requirements.

3. Why choose postgresql over other dbs

1. Feature-Rich Capabilities

- **ACID Compliance**: PostgreSQL fully supports Atomicity, Consistency, Isolation, and Durability, ensuring reliable transactions.
- **Advanced Data Types**: It offers support for JSON/JSONB, arrays, and even custom data types, making it suitable for modern applications.
- **Full-Text Search**: Native full-text search capabilities are ideal for search-heavy applications.

2. Scalability

- PostgreSQL can handle large datasets and supports horizontal scaling with techniques like replication and sharding.

3. Open Source and Community-Driven

- It’s free to use and supported by a vibrant community, which means frequent updates, extensive documentation, and no vendor lock-in.

4. Extensibility

- With extensions like PostGIS (for geospatial data) and TimescaleDB (for time-series data), PostgreSQL can be adapted to specialized use cases.
- You can also define custom functions using SQL or procedural languages like PL/pgSQL, Python, or JavaScript.

5. Performance

- PostgreSQL is known for its excellent query optimization and indexing features.
- Features like parallel query execution, materialized views, and powerful caching ensure high performance.

6. Compatibility

- Its support for standards like SQL and its integrations with various tools and programming languages make it versatile for developers.

7. Robust Security Features

- PostgreSQL includes features like roles, row-level security, and encrypted connections, ensuring data protection.

8. Reliable for Complex Applications

- It’s trusted for enterprise-grade applications, making it suitable for use cases ranging from financial systems to large-scale analytics.

Trade-offs and consequences of your decision(s).

1.Tradeoffs of monolith

1. Scalability Challenges

- **Limitation**: Monoliths are harder to scale horizontally compared to microservices.
- **Example**: If one part of the app requires heavy computation (e.g., image processing), the entire app's performance might suffer.
- **Mitigation**: Scaling monoliths usually requires adding more powerful hardware or breaking the app into modules.

2. Team Collaboration Bottlenecks

- **Limitation**: As the codebase grows, multiple developers working on the same monolith can face conflicts and dependencies.
- **Example**: Developers may struggle with overlapping changes to core components.
- **Mitigation**: Implement strict code management practices like feature flags and CI/CD pipelines.

3. Limited Fault Isolation

- **Limitation**: A bug in one part of the monolith can crash the entire application.
- **Example**: A memory leak in a single service can bring down the whole app.
- **Mitigation**: Employ comprehensive error handling and monitoring to catch issues before they propagate.

4. Deployment Complexity

- **Limitation**: Deploying updates to a monolith often means redeploying the entire application.
- **Example**: A small bug fix or feature addition requires testing and deploying the entire app, increasing the risk of introducing regressions.
- **Mitigation**: Use techniques like containerization (e.g., Docker) to streamline deployment.

5. Reduced Flexibility for Independent Scaling

- **Limitation**: Different parts of the application may have varying scaling requirements but are forced to scale together.
- **Example**: A heavily used search functionality and a rarely used reporting system are both scaled unnecessarily.

- **Mitigation**: Modularize the monolith or transition critical components to microservices over time.

---

6. Long-Term Maintenance Costs

- **Limitation**: As the app grows, the monolith becomes harder to maintain, refactor, and innovate upon due to increased complexity.
- **Example**: Technical debt accumulates faster, making future updates more challenging.
- **Mitigation**: Invest in modular design principles to isolate functionality within the monolith.

2.Tradeoffs of using clerk

Using Clerk in a Next.js app provides significant benefits, but as with any third-party service, it comes with tradeoffs. Here's a balanced look:

Advantages of Using Clerk

1. **Ease of Integration**
   Clerk offers prebuilt authentication components designed for Next.js, significantly reducing setup time compared to building your own authentication system.

2. **Security**
   Clerk is compliant with industry standards and handles complex security features like token management, encryption, and OAuth, allowing you to focus on your app instead of worrying about security vulnerabilities.

3. **Feature Richness**
   Clerk supports multiple authentication methods (social logins, multi-factor authentication, etc.) out of the box, enabling better user experiences.

4. **Scalability**
   Clerk's infrastructure scales to handle thousands or millions of users, avoiding

the headaches of scaling a custom solution.

5. **Maintenance-Free**
   With Clerk, you don't need to manage or maintain the authentication system, updates, or compliance; these are handled by the service provider.

6. **Customizable**
   Clerk offers tools to adapt its components to match your app's branding and user flow.

Tradeoffs of Using Clerk

1. Vendor Lock-In

- **Issue**: Relying on Clerk means you're dependent on a third-party provider. If Clerk changes pricing, policies, or becomes unavailable, you may need to re-implement your authentication system.
- **Mitigation**: Keep an eye on alternatives or maintain an abstraction layer to ease future transitions.

2. Cost

- **Issue**: Clerk's pricing may not scale well for smaller apps with tight budgets, especially as your user base grows.
- **Mitigation**: Compare pricing to the cost of implementing and maintaining a custom solution.

3. Limited Customization

- **Issue**: While Clerk offers some customization, it may not accommodate highly specific or unique authentication workflows compared to a custom-built solution.
- **Mitigation**: Explore Clerk's SDKs and APIs to extend functionality where possible.

4. Privacy and Control

- **Issue**: Using Clerk means handing over sensitive user data to a third-party provider. Some projects might have strict privacy or regulatory requirements that discourage external services.
- **Mitigation**: Verify Clerk's compliance with GDPR, HIPAA, or other relevant standards if necessary.

5. Dependency on Their Uptime

- **Issue**: If Clerk experiences downtime or service interruptions, your app's authentication may be affected.
- **Mitigation**: Ensure Clerk provides robust SLA guarantees and build contingency plans for critical features.

3. Tradeoffs of using postgresql

Tradeoffs of Using PostgreSQL

1. Complexity

- **Tradeoff**: PostgreSQL's rich feature set can be overwhelming for smaller teams or simpler applications that don't need advanced capabilities.
- **Mitigation**: Use a managed PostgreSQL service (e.g., Amazon RDS or Supabase) to reduce operational complexity.

2. Performance at Scale

- **Tradeoff**: As the app grows, high write or read throughput can strain PostgreSQL if not properly optimized.
- **Mitigation**: Implement indexing, caching (e.g., Redis), and read replicas to handle high query volumes.

3. Learning Curve

- **Tradeoff**: PostgreSQL has a steeper learning curve compared to simpler databases like SQLite or Firebase.
- **Mitigation**: Invest in developer training and leverage the extensive documentation and community resources.

4. Operational Overhead

- **Tradeoff**: Self-hosting PostgreSQL requires setup, maintenance, backups, and monitoring, which can be time-consuming for small teams.
- **Mitigation**: Opt for a managed PostgreSQL service to offload maintenance tasks.

5. Cost of Scaling

- **Tradeoff**: As your app scales, running and maintaining a performant PostgreSQL instance with features like replication or clustering can increase costs.
- **Mitigation**: Use auto-scaling infrastructure and optimize database usage with query tuning and caching.