Alexander Flood

SNHU CS-499

5th October 2025

**Artifact Three Narrative**

For this enhancement, I decided to improve my Weight Tracker mobile application which I originally developed in CS-360: mobile architecture and programming. The application allows users to reliably record their weight entries and set target weight goals. Previously, I improved the application by introducing a RESTful API to improve reliability, extensibility, and scalability, allowing the app to separate its client and server logic for cleaner architecture and future updates.

For this milestone, I focused on the database layer of the project. The enhancement added an offline caching mechanism using a local SQLite database that mirrors the cloud-based MySQL database accessed through the RESTful API. This change was implemented through an updated DatabaseHelper class that was refactored out of the original project during the first enhancement, and it will help improve the robustness of the application, introducing fault tolerance and offline-first functionality.

**Justification**

I selected this artifact because it demonstrates my ability to design, implement, and integrate database-driven systems that balance efficiency, speed, and scalability. The original application I brought to this capstone initially only worked offline, relying on a local SQLite database for persistence. After the first enhancement, I restructured the application to connect to a RESTful API backed by a MySQL database, shifting the system to an online-only model that improved reliability and centralized the data storage, but required a constant connection.

For this milestone, I combined the strengths of both approaches by introducing an offline-first design that uses local caching to compliment the cloud-based API. The enhanced version now stores weight records and goals locally in SQLite, allowing users to view and manage their data even without an internet connection, while synchronizing with the cloud once it is restored. This approach not only improves the reliability of the application, but it also enhances the performance. Cache reads can reduce network latency, making the interface feel more responsive while reducing the load on network services (Parks, 2023). This results in a smoother and more usable application, even with intermittent network connectivity, providing a more consistent user experience.

**Outcomes**

This enhancement directly supports course outcome 4, which involves applying software engineering and database principles to create scalable, efficient, and maintainable solutions. By integrating a local caching system, I utilized established tools and patterns in mobile development to deliver a more responsive and fault-tolerant user experience. It also touches on outcome 5, by ensuring data is handled securely locally and in transit, and outcome 2, through clear documentation and concise comments that improve code readability and maintainability.

This enhancement shows that I can design and manage a multi-service architecture with robust data systems that preserve data integrity and handle failure scenarios gracefully. This work also compliments the token storage and encryption from my earlier enhancements.

**Reflection**

Enhancing this project taught me a great deal about data synchronization and the importance of designing for real-world connectivity constraints and proper debugging. Implementing the offline cache required thinking about how the app behaved around edge cases, like network failures or stale data. One of the key lessons I learned is that simplicity is often correlated with reliability. Rather than introducing a complicated background synchronization service, I implemented a straight-forward approach where data is fetched on an initial load and then refreshes it from the API if available. This not only made the implementation easier to test and maintain but demonstrated offline first architecture.

The main challenge I faced was ensuring that the data model between the API and SQLite schema remained consistent as both systems handled timestamps and IDs differently. After ensuring the structure between my MySQL schema and SQLite schema were accurate, I was able to maintain consistency across both storage layers. While this might seem simple at first, there are small differences between SQL dialects that can cause compatibility issues across systems. Each database vendor implements its own variant of SQL, known as a dialect, and while they share the same foundations, syntax, functions, and features can vary greatly (Kozubek-Krycuń, 2023). Understanding these differences and nuances are critical to making sure SQL code is portable, efficient, and behaves as I expect.

Overall, this enhancement strengthened my understanding of database design, caching, and user focused reliability. It also reflects my ability to enhance and improve existing codebases, turning my prior coursework into professional quality software to help me in my journey to employment as a software developer.

**References**

Kozubek-Krycuń, A. (2023, September 12). *What Is a SQL Dialect, and Which one Should You Learn?* LearnSQL.com. https://learnsql.com/blog/what-sql-dialect-to-learn/

Parks, Y. (2023, August 17). *The power of Caching: Boosting API performance and scalability*. dzone.com. https://dzone.com/articles/the-power-of-caching-boosting-api-performance-and