**CS-499 Milestone Four  
John Kirven**

**About this artifact:**This artifact is an Android application built with Java whose primary purpose is to manage the inventory/stock of a warehouse. It supports simple (local) account creation and uses a local database to hold item information and user accounts. The app uses a model-view-viewmodel (MVVM) architectural pattern to separate the UI logic from the database data. The database is handled via Room, a library for implementing SQLite in Android and currently houses two tables, Account and InventoryItem. The app itself consists of three Activities; login/account creation, adding new items, and browsing the list of all items.

**About the enhancement:**In the previous enhancement, the data structure of InventoryItem was altered to include more fields and sorting by those new fields was implemented. In this enhancement, the locally stored Room based SQLite database was replaced by a MySQL database controlled by a RESTful API constructed in Node.js/Express. To communicate with this new API, Entity classes (Account & InventoryItem) had their original annotations removed, the InventoryDatabase and DAO classes were removed entirely, and Retrofit was introduced to communicate with the API.

**The process:**The first thing I did was create a new MySQL database to replace the local SQLite database, keeping the original database structure with a slight change to the Account table. Since we’re moving from local to remote having passwords stored in plain text was an immediate security issue so the password field was replaced by two new fields, salt and hash.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

With the new database set up and with the existing items imported into it, it was time to work on the API. The API was built using Node.js and Express. It is a relatively simple API that uses a JSON Web Token (JWT) to prevent unauthorized access to the database. CRUD functions were added for both the Account table and the InventoryItem table and then endpoints for each CRUD function were added (using authentication when needed). This mostly mimics the previous database operations but when creating an account, the supplied password is salted and hashed and those values are stored in place of a raw password. Lastly an endpoint for logging in was added, which generates a JWT when the login is successful.

A screen shot of a computer

AI-generated content may be incorrect.

Prior to integrating the API into the Android app, I tested each endpoint with valid and invalid data using Postman.

With the endpoints confirmed to be fully functional I moved on to the Android app. Because I used a model–view–viewmodel (MVVM) architecture I knew my views (activities) were going to be largely unchanged and the viewmodels, models, and repository would be the main area of refactoring. My DAOs and Database classes could be and were removed entirely. The Database class was replaced with a new class, ApiClient, that constructs a new Retrofit client for interacting with the new API. My DAO classes were replaced by service interfaces (AccountService and InventoryService) which define the GET/PUT/POST/DELETE calls that can be made to the API and their required parameters. I then removed all the Room/Entity annotations from the two model classes (Account and InventoryItem) and added new SerializedName annotations to ensure each field points to a field within the database.

The InventoryRepo class now fetches two Retrofit clients, one for accounts and one for items, which also meant that there now only needs to be one instance of InventoryRepo needs to exist throughout the entire program, so I switched to a singleton design.

Authentication also needed some adjustments since many requests now required a valid JWT, I added a simple TokenManager class to store the JWT that is supplied when a valid login request is returned.

The Room/SQLite approach was a synchronous process, all requests happened on the main thread, but Retrofit is asynchronous, so each call to the API needed a callback for success and/or failure. I primarily accomplished this by overriding the Callback interface after calling Retrofits enqueue method in addition to Runnable parameters for success and failure that the activities use to adjust the UI when a callback succeeds or fails. Some slight modifications had to be made to the Activity classes to work with the new callbacks, but they were largely unchanged. Once implemented, quite a few debugging statements needed to be added to work out a few kinks in query execution and proper callbacks but eventually all endpoints were tested inside the app, and everything is working as it was with the previous local database.

A screenshot of a computer

AI-generated content may be incorrect.

**Outcomes:**

Design and evaluate computing solutions that solve a given problem using algorithmic principles and computer science practices and standards appropriate to its solution, while managing the trade-offs involved in design choices (data structures and algorithms)

Demonstrate an ability to use well-founded and innovative techniques, skills, and tools in computing practices for the purpose of implementing computer solutions that deliver value and accomplish industry-specific goals (software engineering/design/database)

Develop a security mindset that anticipates adversarial exploits in software architecture and designs to expose potential vulnerabilities, mitigate design flaws, and ensure privacy and enhanced security of data and resources