Battery meter documentation

Contents

1 Project Assumptions

1.1 Goal

The main goal of our application is to measure battery consumption by a selected application on an Android device. The user can choose any installed application and monitor battery usage. The application allows the user to start and stop the battery consumption measurement. Additionally, there is an option to choose the preferred measurement unit, [mAh] or

2 Technologies Used

2.1 Android Studio

Due to prior experience with "Android Studio" and its compatibility with our chosen technologies, we decided to use this environment to write the application. The application is entirely written in Java.

2.2 Application Interface

To create an intuitive and simple interface, we used the online application "Canva" to visualize the initial version, which we then implemented in the "Android Studio" environment. The prototype can be seen here.

2.3 Web Application

To create the website, we chose the NetBeans development environment. We opted for it because none of us had prior experience in web development, and we found NetBeans online as a free and easily accessible tool. Creating the site required connecting a series of files in HTML, JavaScript, and CSS. The main structures of individual parts of the web application were created using HTML files, which were then styled using CSS files. JavaScript files were used, among other things, to connect the site to the Firebase database and to add interactive and dynamic functionalities to the site.

2.4 Databases

To correctly save measurement data concerning battery condition, we were looking for an offline database with simple implementation and compatibility with the "Android Studio" environment. One of the first ideas and research results was using the "Realm" database, but we abandoned this idea because

despite its high performance and flexibility in data modeling, it requires precise management of database object lifecycle and a specific approach to multithreading, and we were aiming for simplicity in data storage.
Ultimately, we chose the SQLite database with the Room Persistence Library as the optimal solution for our needs. The Room library offers simple integration with "Android Studio," enabling easy and fast implementation and testing of the application. By using it, we can utilize a convenient API that facilitates CRUD (Create, Read, Update, Delete) operations and provides automatic database management from the code level.

3 Implemented Classes

3.0.1 BatteryEntry

This is an entity, a class representing a table in the database, which is supposed to store data about the battery condition. The entity represents a data model (stores information about individual measurements).

3.0.2 DAO - DataAccess

This is an interface that defines what operations (adding, reading) we can perform on the entity data.

3.0.3 Room Database

The Room database enables the creation of local SQLite databases that allow storing them directly on the device. Thanks to it, even without an internet connection, the application can still use data, perform operations on them, store changes, and display them to the user. We considered using it in our project because of many advantages but finally, we did not use this technology. Below there are some of advantages:
- easy error detection, as they are detected already at compile time, eliminating potential problems related to database queries during application operation
- by using separation of logic and database operations, it is possible to quickly and easily perform operations on the database without writing complex queries
- migration capability in case of data structure changes, ensuring smooth application updates

3.0.4 BatteryChecker

BatteryChecker is a background component of the application that monitors changes in the battery level of the device. It is a class that inherits from AppCompatActivity, meaning it can also be used as an activity in an Android

application. Inside this class, a BroadcastReceiver is defined, which listens for system intents informing about changes in the battery state. Main functions:
- Registration of BroadcastReceiver: In the onCreate method of the BatteryChecker class, the BroadcastReceiver is registered to listen for system intents regarding battery level.
- Receiving battery state updates: The BroadcastReceiver receives updates on the battery state, such as the current charge level and battery scale, and then calculates the charge percentage.
- Saving data to the database: After calculating the charge percentage, this data along with the current timestamp is saved to the local database using the Room Database.

4 Research on Battery Consumption Measurement Algorithm

4.1 Saving Data Locally

Saving data locally is the process of storing information on the device without the need to use external servers or the cloud. Methods of saving data locally include using databases (e.g., SQLite), text files, JSON, XML, or system preferences. To save data locally, the application must obtain appropriate permissions to write on the device. Proper data management can involve, for example, an automatic data cleaning algorithm or allowing the user to manually delete old data.

4.2 Using Sensors

Using sensors to monitor battery consumption allows collecting data based on the actual conditions of device usage. This can be done with various types of sensors, such as light, motion, temperature, or location sensors. In the Android system, there are dedicated APIs for handling different types of sensors. By using these programming interfaces, it is possible to obtain data from sensors that can be used to write an energy consumption monitoring algorithm for the application. Using sensors mainly helps us develop energy-saving strategies.

4.3 Rooting the Phone

Rooting the phone allows access to system logs that can contain detailed information about battery consumption by various applications and system processes. It also enables access to advanced system functions, such as controlling the CPU frequency, managing background services, and other settings that can impact energy consumption. However, using this method can violate user privacy and is generally inadvisable for security and ethical reasons.

4.4 Available System Statistics

They provide detailed information about energy consumption by individual applications, processes, and system resources. System statistics can also provide information about the device's battery runtime and the remaining time until the battery is depleted.

4.5 Analysis of System Processes

This involves studying and monitoring the activity and resources used by various processes running in the operating system. Analyzing system processes allows monitoring resource consumption, such as CPU, RAM, or network usage. We can identify which processes consume the most resources and how this translates into battery consumption.

4.6 General Research Based on Points

During the research on possibilities for monitoring battery condition in mobile devices with Android, we focused on various available APIs and tools. We analyzed the BatteryManager API, BatteryStats API, and Battery Historian. Each of these tools offers specific functions and capabilities relevant to monitoring and analyzing energy usage by mobile devices.

The BatteryStats API provides access to detailed data regarding energy consumption by various applications and hardware, which is useful in identifying components that most drain the battery. However, we did not select this tool because, although it offers extensive analysis capabilities, the API data often requires additional permissions, which can be difficult to obtain, especially on devices with security restrictions. Some of these permissions require rooting the device, which is generally not recommended for privacy and security reasons.

Battery Historian is a tool for analyzing and visualizing battery usage data, allowing for a deeper understanding of energy consumption patterns by applications. However, we did not choose this library due to the complexity of the information it provides. It requires extracting data from log files, processing and analyzing them before they can be useful in chart form. This further complicates the process of implementing functionality in the mobile application and requires additional tools for data analysis. Battery Historian mainly analyzes system logs, which are typically generated and available for export only after performing specific operations on the device. These logs are not created or updated in real-time in a way that would allow them to be directly and immediately transformed into chart data, so we rejected this option.

Ultimately, we chose the BatteryManager API, which is an integral part of the Android platform, providing information about battery charge level, power status, and other energy-related parameters. This decision was guided by several key factors:

- The BatteryManager API offers real-time access to battery data, allowing for easy data collection throughout the monitoring period. After the session ends, collected data can be immediately processed and charted.
- As part of the native Android API, BatteryManager integrates seamlessly with the rest of the system and our application, facilitating implementation and ensuring stable operation.
- BatteryManager does not violate Google Play security policies and does not require excessive permissions, which is crucial for user privacy.

Using the BatteryManager API, we developed two measurement methods: one utilizing the coulomb counter to read the current nano-amp-hour value in the battery. Then, by observing changes in the battery, i.e., the current draw from the battery, we can calculate the current draw by the entire phone. To isolate individual applications, we use a baseline measurement, where we measure the power draw by the phone, system applications, and our measurement application. We then subtract the baseline measurement from the observed value to obtain the power draw by each individual application. This measurement method incurs a measurement error of 20% in extreme moments but averages 10%. The values are compared with Battery Historian, which also provides estimated values. This measurement error is easily correctable and predictable, as it always overestimates. Unfortunately, this measurement method cannot be applied to every phone because not every phone is equipped with a coulomb counter, and the method used to check if the coulomb counter exists does not always return the same value in its absence.

The second method involves checking the current power draw by the phone in nano-amperes and performing this measurement multiple times per second to obtain nano-ampere-seconds, which are then scaled to mAh. This measurement method also has a baseline measurement to determine the power draw by the application and the system and its peripherals, which is also subtracted from the measured value when measuring a particular application. This measurement method is subject to a larger error of around 30% in extreme cases, with an average of 20%. Increasing the measurement frequency does not significantly improve this result; the standard value is 10 measurements per second. By increasing this value to, for example, 20 measurements per second, with 100 measurements checked, the average error decreases from 20% to 17-18%, with the possibility of an error occurring in which there is a momentary increase in power draw that completely discredits the measurement.

The measurement is performed in a foreground service, and after reading the instructions, the user knows that only our application and the measured application must be running during the measurement. The application must also be running on the "1 plane" for the measurement to be valid.

5 Web Application

5.1 Connecting the application to the Firebase database

To connect the mobile and web applications for synchronization purposes, we utilized the firebaseConfig generated by Firebase. This configuration enabled user registration and login on both mobile devices and the website.

5.2 Registration

For user registration, we used a specific function provided by Firebase called createUserWithEmailAndPassword. Users are required to provide an email address, password, and confirm the password for account authentication. Another Firebase function, getAuth, is used to ensure successful account creation. Additional requirements such as a minimum password length of 8 characters and the inclusion of at least 1 special character were implemented to enhance user security.

5.3 Login

User login functionality utilized another Firebase function called signInWithEmailAndPassword. Users must provide their email address and the correct password associated with their account. These credentials are verified using the getAuth function mentioned earlier. Upon successful login, users receive a confirmation message and are redirected to the next page.

5.4 Password Reset

In case a user forgets their password, Firebase provides a function called sendPasswordResetEmail to facilitate password reset. By clicking on the appropriate button, a password reset email is sent to the provided email address containing a link to reset the password.

5.5 Measurement History

To access measurement history from the mobile application on the computer, we utilized the Firestore library provided by Firebase. The following Firestore functions were utilized:

- getFirestore - initializes a Firestore instance and allows interaction with the database.
- collection - refers to a specific collection within the Firestore database.
- query - used for creating queries in Firestore, allowing for conditions and data filtering.
- orderBy - employed in Firestore queries to sort results based on specified conditions.
- getDocs - used to retrieve data from Firestore based on a specified query. It returns a collection of documents that meet the specified conditions.
- signOut - logs the currently logged-in user out of the application, ending the session.

6 Bibliography

References

[1] Canva:https://www.canva.com/design/DAF_2MkRM2E/tbpQaVONVMAGE_PMmRPASg/editutm_content=DAF_2MkRM2E&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton