The key constraint is to minimize the amount of power consumption for data downloading requirements in the app**.** Android makes the following suggestions:

* **Prefetch Data** - Download all relevant data in a single burst, over a single connection, at full capacity - reduce the data transfer sessions. Front loading transfers to conserve battery life, improve the latency, lower bandwidth, and reduce download times. Prefetch for 6 seconds or 1-2 Mb. Prefetching can also increase battery drain by downloading data that isn't used, so a balance needs to be struck.
* **Reduce & Reuse Connections** - Reuse existing network connections rather than initiate new ones. Bundle requests into a single GET. Closing a connection too early can requires additional overhead for establishing a new connection.

What else is relevant to battery and power conservation in Android? What actually causes battery drain and what should we be worried about, in addition to network operations?

Research has identified that GUI & Image Manipulation and Database ops represent 60% of the energy gobblers in Android. Location Based Apps, with continuous location update of user and which use location information to display third party advertisements, are causes of concern. Monitoring the use of wakelocks which keep hardware components awake, and better managing sensor hardware, autosync and notification frequencies can improve power efficiency. The brightness of the backlight, maintaining network connections consume significant power, and GSM, CPU and graphics dominate energy drain. Video games eat power, and moving between cells/networks can eat up energy – since radio ramps up to new cell, and BroadcastIntents are fired. GPRS consumes more power than WiFi by a significant margin. Suggestions then are to conserve battery by better managing Wi-Fi, 2G and 3G connections, brightness level, CPU frequency and GPS (Carroll & Heiser, 2010; Datta, Bonnet & Nikaein, 2012; Linares-Vásquez et. al., 2014).

Exploring specifics at the code level, it is found that, in GUI & Image Manipulation, there are specific methods that need to be managed – for example the method “notifyDataSetChanged” of class “ArrayAdapter” is reportedly energy greedy. This method notifies observers of data changes and needs to refresh all views when data changes happen, which leads to energy drain (Linares-Vásquez et. al., 2014).

In the Database category - opening “SQLiteDatabase.openDatabase”, querying “SQLiteDatabase.query”, and deletion “ContextWrapper.deleteDatabase”, represent some of an identified 30 energy greedy APIs. XML files manipulation is apparently less energy greedy - only method “XML.newSerializer” of the File Manipulation category is greedy, and therefore persistent data may be better stored in XML files or using “SharedPreferences” rather than a database (Linares-Vásquez et. al., 2014). This has been implemented in the app and is one way it conserves power.

In the Data Structure Manipulation category, simple getter methods – such as “Bitmap.getPixel(int,int))” – are reported to be very energy greedy. Android API documentation suggests using getters and setters when accessing internal class fields, but this may be actually bad idea from the energy pov. In the Web category and the APIs related to internet surfing, seven methods are reported to have high energy consumption. In particular, the constructors of the WebView class are held to be especially energy-greedy. Other energy-greedy APIs are the Context.bindService method, and send requests and receive responses (Linares-Vásquez et. al., 2014).

**Energy Hungry Patterns** (Linares-Vásquez et. al., 2014):

* The pattern <Activity.setContentView(int); Activity.findViewById(int); View.setVisibility(int)> is reported to be a very energy-greedy sequence, consuming 0.20 Joules.
* Sequences of calls to the SQLiteDatabase.execSQL(String) method are expensive, especially when the statements are used to create/drop database elements. Database operations - <SQLiteOpenHelper.getWritableDatabase(); SQLiteOpenHelper. getReadableDatabase()> has an average consumption of 0.16 Joules.
* It is reportedly power expensive to use <ConnectivityManager. getNetworkInfo(int); NetworkInfo.isConnected()>. A possible solution is to replace this pattern with ConnectivityManager.getActiveNetworkInfo().
* Two widgets - ProgressBar and Toast - are energy-greedy. An instance of <ProgressBar.setProgress(int); ProgressBar.setProgress(int)> consumes 0.007 Joules. The pattern <Toast.makeText(Context,CharSequence,int); Toast.show()>, consumes 0.008 Joules.

**Recommended energy-saving recipes (ESRs)** are:

1. Carefully design the storage strategy – The app uses “SharedPreferences”.
2. Limit the use of the Model-View-Controller (MVC) pattern, since refreshing views is expensive. Avoid unnecessary refreshing, i.e., refresh operations made on inactive or invisible views.
3. Limit the use of energy-greedy widgets for cyclic activities.
4. Carefully design and review apps that make use of several Views. The navigation of View components is energy expensive since it is necessary to browse the layout files in Android.
5. Carefully analyze the trade-off between design principles and battery saving. Design principles such as information hiding is quite expensive in Android. Thus, disregarding this design principle and giving direct access to private fields may save battery life (Linares-Vásquez et. al., 2014).

The app is power intensive because it has to perform 5 API calls - network operations with associated Views. In accordance with Android recommendations, the app uses prefetching and implements SharedPreferences storage, thereby eliminating an additional network call. The app reuses a View in DisplayBookbyIdActivity by listing the results of the GetMoreInfo() function within the current View, rather than opening a new View as initially implemented. The app could have further limited the number of Views – for example it could have eliminated BookListActivity by incorporating list display and search results within the MainActivity itself. A balance has to be struck between energy conservation and demonstrating the implementation of a range of features. Therefore, the app implements WebView, Toast and ProgressBar solutions despite their power drawbacks.

The battery levels can be monitored in MainActivity using BatteryManager, BroadcastReceiver and a sticky Intent. This returns the percentage of battery remaining, and can be checked after running different activities in the app. Implementing power conservation measures does not adversely affect the architecture or functionality of the app, on the contrary, it makes for a more elegant and stream lined solution. Anticipation user actions and prefetching and caching data makes for a faster, more responsive and intuitive user experience. Software code redesign and refactoring can go a long way in managing power consumption, and alternative strategies to achieve the same result can and should be considered at every stage of development.

**Annexure: Other recommendations**

Location based framework - employ four principles: substitution, suppression, piggybacking and adaptation. Substitution uses different positioning systems instead of GPS, suppression uses accelerometer-less power sensors to position the user, piggybacking is used for caching and distributing the location information to all the LBAs. Adaptation to manage position information when the battery is low (Datta, Bonnet & Nikaein, 2012).

Use an efficient data format and parser; Use "stream" parsers instead of tree parsers; Consider binary formats that can easily mix binary and text data into a single request; Fewer round-trips to server. Use GZIP for text data whenever possible - Framework GZIP libs go directly to native code, and are perfect for streams. Use coarse network location, it's much cheaper (Thiagarajan et. al., 2012).

Optimizing Mobile Web Pages: JavaScript is one of the most energy consuming components in a web page. Optimizations: Shrinking JavaScript on a mobile page to contain only functions used by the page greatly reduces energy cost. Using HTML links instead of JavaScript greatly reduces the rendering energy for the page. Large CSS files with unused CSS rules consume more than minimum required energy. Optimizations: CSS should be web page specific and contain only the rules required by the elements in the web page. Using simple HTML table element to position elements on the page instead of CSS saves energy. JPEG is the most energy efficient format for all image sizes - converting images to JPEG saves energy (Thiagarajan et. al., 2012).

**References**

Carroll, A., and Heiser, G., (2010), “An Analysis of Power Consumption in a Smartphone”, Proceedings of the 2010 USENIX conference on USENIX, Boston, MA

Datta, S.K., Bonnet, C., Nikaein, N., (2012), “Android Power Management: Current and Future Trends”, First IEEE Workshop on Enabling Technologies for Smartphone and Internet of Things (ETSIoT). IEEE, 2012.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D., (2014), “Mining Energy-Greedy API Usage Patterns in Android Apps: An Empirical Study”, Proceedings of the 11th Working Conference on Mining Software Repositories, Hyderabad, India.

Thiagarajan, N., Aggarwal, G., Nicoara, A., Boneh, D., and Singh, J. (2012), “Who Killed My Battery: Analyzing Mobile Browser Energy Consumption”, Proceedings of the 21st International World Wide Web conference