MyKedma Graph Feature

June 2017

OVERVIEW

As of now the My Kedma App allows for viewing any previous day's power consumption, generation, and the overall difference. This design proposal is for adding a data graph interface that users can use to visualize power usage, consumption, and savings across a variety of time scales. The main philosophy behind the decisions in this feature is to give the typical user a simple and easy to understand visualization, while also giving users the option to get more detail if they choose.

GOALS

1. Display visual data regarding a user's power usage using a visual time-dependent graph. The interface should be simple by default, but still give the user to view more information if wanted.
2. Create, edit and follow wire frame picture to best design a graph interface and interactions between views.
3. Include options for the user to opt-in to more detail across different unit scales, time scales, graph types.
4. Have a consistent color and font theme in relation with the rest of the application by following documentation for the graphing library.

SPECIFICATIONS

The graph will be constructed using the MPAndroidChart library, which will be added as a dependency to the project. This library allows for highly customizable graphs while also providing good documentation and demos. Reasons for selecting this library are laid out fully below.

The graph itself is a feature designed to be called upon clicking any of the gauges shown on the main dashboard. In order to demonstrate discoverability, the gauges should be adjusted to behave like a button, in that pressing down grays out the background and indicates an ability to click through. This clicking will open up to a new activity that details either total power usage, consumption, or generation depending on which gauge is pressed.

When constructing the graph interface the general idea is to default to a simple bar graph that displays the last week's data. The graph however can be changed between a bar graph and line graph, and the timescale should be able to change between 1-day, 7-day, 21-day, 3-months and 1-year.

In a similar fashion to Fitbit's graph page, individual daily entries can be seen and scrolled through. When selected these entries will pull up a daily graph for that day's power usage should the user want a more detailed look.

DESIGN MILESTONES

Why MPAndroidChart is the best for this project

In trying to find the best library for designing, building, and customizing visual graphs I came across a few notable libraries, each with their own strengths and weaknesses.

The first was GraphView , an open source library focused on programmatically building graphs on Android. At first the library seemed to be a great choice for the project. It provides a good amount of customization through a number of different settings like color, scale, size, view port size, etc. with documentation detailing how to get started. There are however two glaring issues that would not sit well with the project. First, the library reports to supporting a portrait orientation

viewport, however the documentation on this is lacking compared to the rest of the library and there a multiple unresolved issues related to crashing when building a portrait view graph. Since My Kedma maintains a style of portrait view throughout the rest of the application, designing a critical visual component in a wholly different orientation would confuse a typical user. Second, the library has beta support for adding data in real time, but it is not a fully featured function and bugs may be present. It would be best to have this functionality so later features like infinite scrolling would be easier to implement without loading lots of data at build time. Overall it is a good library but it needs more support and developers working on features and issues to be viable for this project.

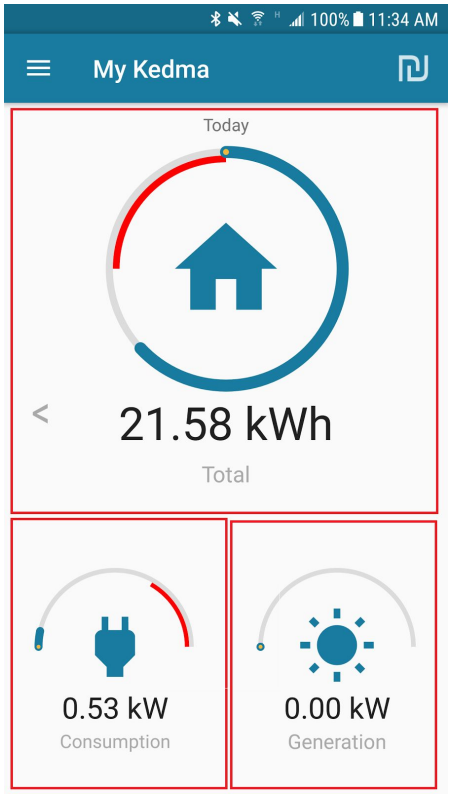
The next library discussed was Androidplot and is the oldest of the libraries discussed. This library showed promise with its wide use and a good getting started documentation. Implementing this would be pretty simple and key features such as dynamic graphs and portrait orientation support were present. However there are issues that again would confuse our typical user. The first big issue has to do with the look of the actual rendered graph. In the examples and demos given every graph shown had a more cartoony and “comic sans” like appearance which doesn’t really match the theme of the other aspects of the My Kedma app. Additionally, when rendering on my Galaxy S6, the Bar and Line graphs would suffer from jagged and cut edges near data points. Another issue discovered was the documentation focused heavily on the first steps to getting simple graphs up and running, while other features such as dynamic data addition and stylizing choices was pretty lacking. Essentially developing a simple graph could be done easily, but making a more complex graph with unique scales, colors, ranges etc. would take more time then it is worth.

The library of choice was MPAndroidChart as it delivers the best documentation, affords the most options with customizing style, and has a simple quickstart guide for getting up and running. In more detail, the documentation is concentrated on the GitHub wiki and has a large amount of examples detailing different graphs and style options. A notable difference between this project and others is the documentation discusses and demonstrates integrating the graph into a variety of views, such as combining the graphs in a ListView, or demonstrated using an EditText view to get user input to directly affect the graph. Beyond that the documentation is well defined and sections deliver an in depth overview on what functions are available and how to use them. This library also appears to be very popular with developers and the Github page shows active

support from the community. Issues are opened and closed regularly and functionality is well documented.

Wireframes and Interactions

DashboardActivity

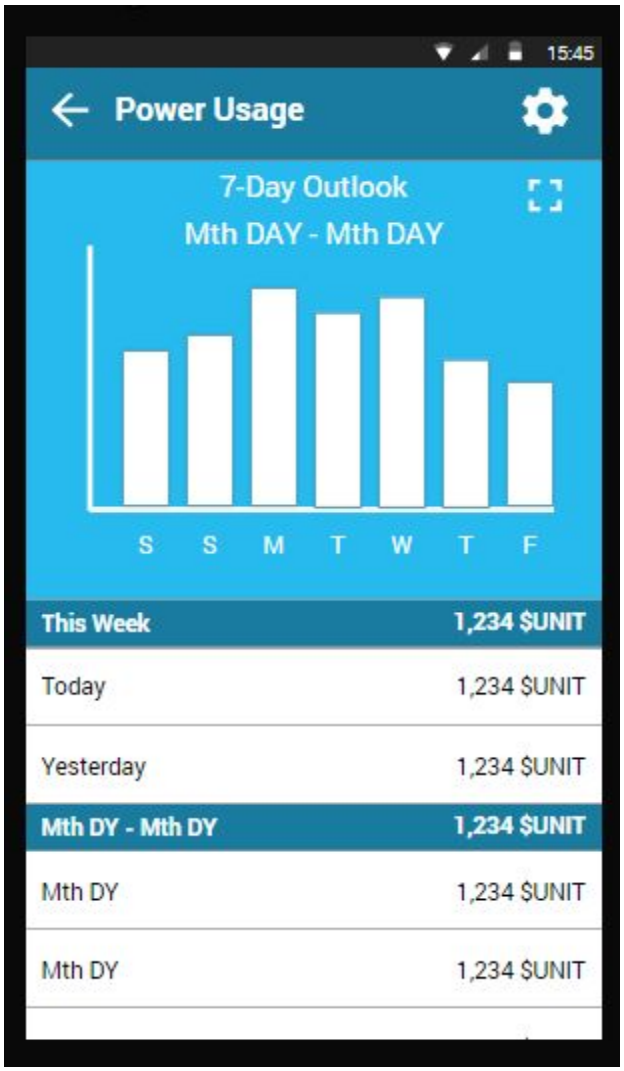


DashboardActivity has three gauges ,outlined in red, which can be clicked to bring up the graph.

The graphing feature is launched from the Dashboard by clicking on any of the three available gauges. Depending on which gauge is selected, the graph will use that data when displaying information to the user. So if the user were to launch the graph functionality by clicking on the “Total” gauge, then the graph would display the difference between the consumption and the generation, rather than just consumption or generation individually. Consistency between the dashboard and the graph is important and this serves to make the user feel comfortable moving between views.

In addition, the units that the user set in this screen, shekel or kWH, will be used in the graphing feature as well. This allows for less confusion to the typical user, who may be used to seeing a certain unit type, and expect it to remain consistent.

MainGraphActivity

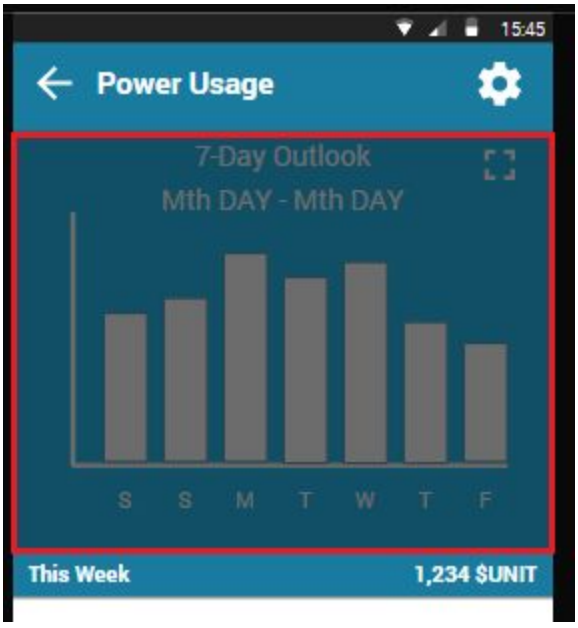


The default view of the MainGraphActivity with graph and daily entries views each taking up half of the screen.

Upon clicking a gauge on the main dashboard the user is presented with the above screen. This is the MainGraphActivity, the core activity of the graph feature. The layout can be split into two sections, one displaying a visual graph of values, while the other displays individual entries for

previous days. This double display gives the typical user a standard, easy to understand graph view and a simple list of items to scroll through.

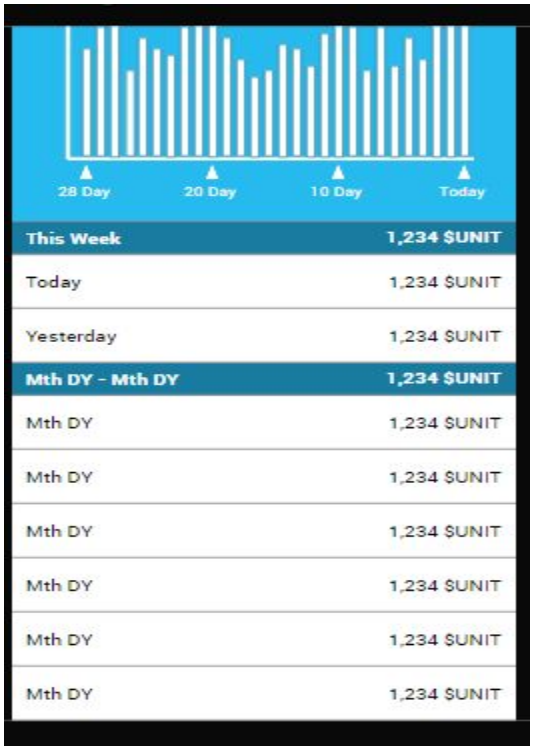
The graph view is a configurable view where the units, time scale, and graph type can be set should the user want to get a more detailed and specific visualization. These settings are going to be set in a separate activity launched by pressing the settings icon in the top right corner. Separating the settings into a separate activity (described below) allows for a number of advantages both in implementation and in keeping with following the “keep it simple” design philosophy. In terms of implementation, separating into another activity allows for easy addition of settings should additional options become viable and wanted in later cycles of development. But more importantly, keeping settings on the default page runs the risk of confusing the typical user with too many options and cluttering the page with unwanted drop down menus or selectors. This runs against the previously stated philosophy of making the default view simple and easy for the typical user while also giving options for more detail if wanted. A new settings activity can simplify all that complexity on the default page into a simple settings icon in the upper right corner.



The graph view, outlined in red, launches the graph in a full screen view.

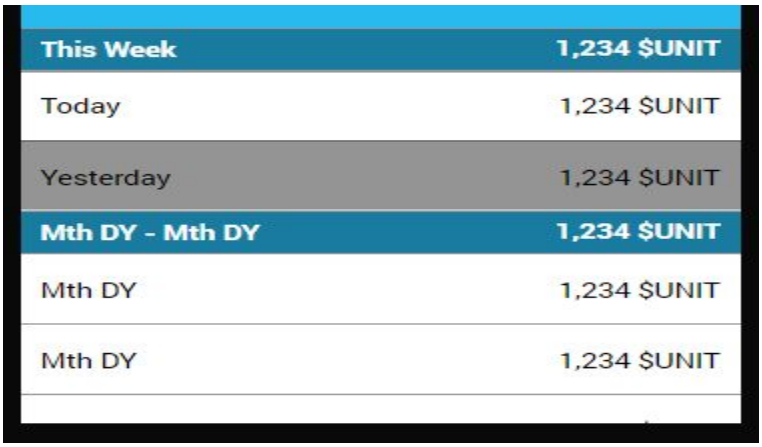
The graph view is a single view that acts as a small display for power usage. As iterated above, the settings can change aspects of the graph. It is also important to understand how the graph

can be expanded to get a full screen view with additional features and details described below. The entire graph view can be clicked to launch the full screen view. To indicate this ability the fullscreen icon will be placed in the upper right corner of the graph view. While executing the clicking action the entire view will gray out, indicating the ability to click on the graph.



The daily entry view when scrolled should take up the rest of the screen.

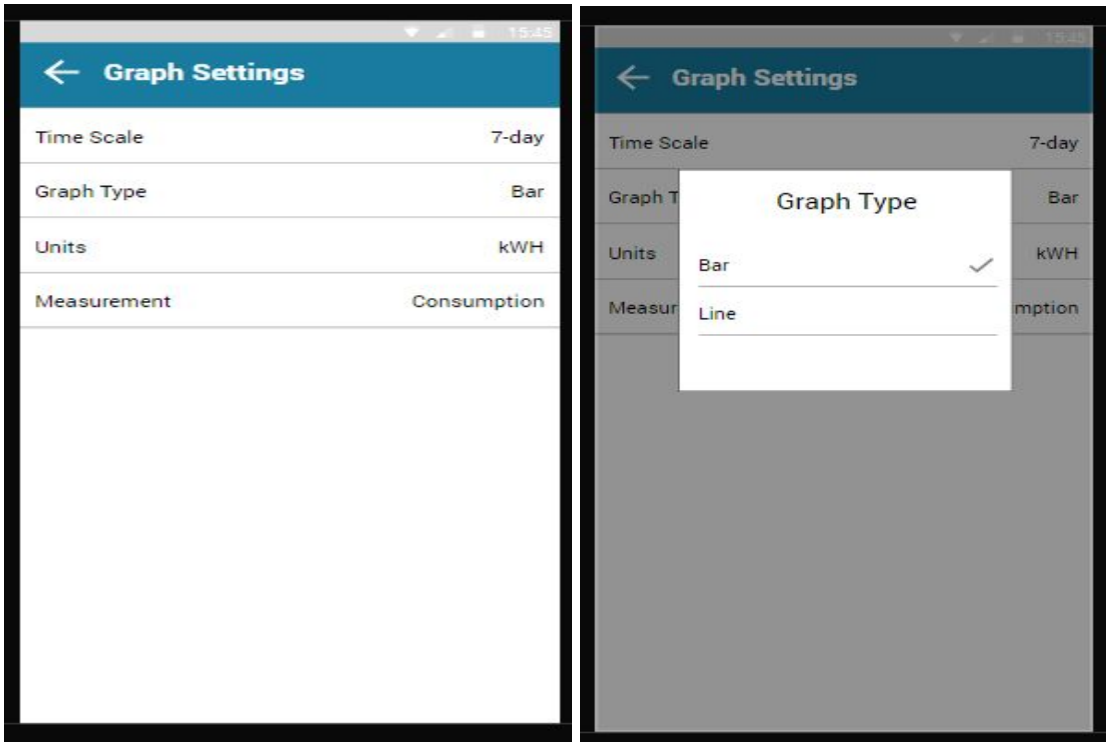
Below the graph view is a list of daily entries for previous days. Each of these daily entries contains the same information for each given day including the date and the power usage with appropriate unit. These provide a simple list that can be scrolled through and provide more information about usage between recent days than the dashboard. The units used in the entries will be the same as those used in the graph above.



Clicking on an entry an entry launches a full screen graph for that day which is described below

Each daily entry can be clicked upon to launch FullscreenGraphActivity with a graph displaying that day's power usage. This is indicated by the graying out of the entry upon clicking, although no explicit fullscreen symbol should be given so as to not clutter the list.

GraphSettingsActivity



Left, the list of preferences the user can set, with the current active settings displayed. Right, upon clicking a preference, a selection box should open up.

Should the user want to adjust the settings of the graph, the settings icon will launch this activity where preferences of the user can be set. As said before, the list allows for easy addition of settings in development while also providing a well understood interface convention that the typical user is used to. The preferences are outlined and justified here.

In keeping with consistency of style throughout the app, the settings in this activity would have to have icons similar to what is already found on the app level settings. These have not been mocked up and serve as a low priority design choice.

The “Time Scale” preference is one of the more important options as it allows the user to select a predetermined window of time. All of these time scales will include the data of the current day.

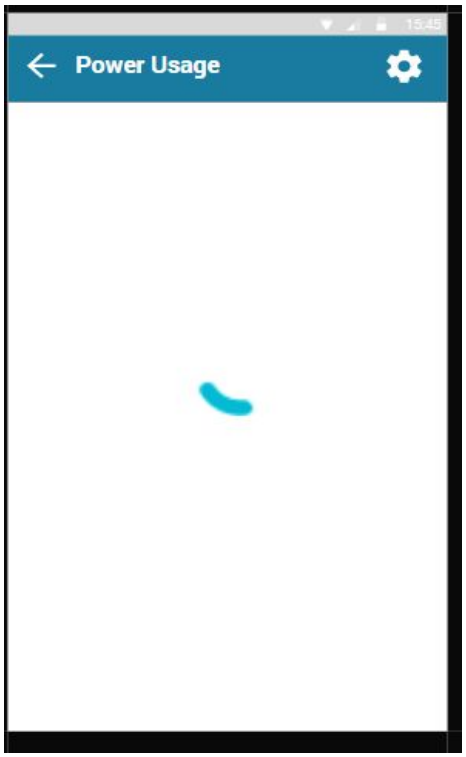
1. The first option is the simple “1-week” timeframe which displays the last 7 days worth of data. This is a simple and reasonable time scale that will serve as the default time scale option. Each datapoint is for a single day.
2. The second option is a “1-Month” timeframe that spans a slightly larger window of time. This view would give data from the last 30 days as individual data points. Since billing is done on a monthly basis, this timeframe serves as a good outlook for the typical user.
3. The third option is a “3-month” which in actuality simply spans a period of the past 13 weeks. The reason for this disparity in name versus the actual measurement is to give the user a simple and easy to understand scale. If it were to say “13-week scale” then the user might be confused as to why such an odd number is being used. They might believe “13 weeks is a weird amount of time, how is it significant?” Also using the Google time conversion, 13 weeks is equal to 2.99178 months. In this case, individual data points span a week each.
4. The fourth option is a “1-year” time span which is a rather simple yet long term time scale. Each data point spans a single month, allowing the user to view the past 12 months in a single graph.

The “Graph Type” preference allows for the user to determine which type of graph to use in the display. Initially, the user will only be able to select and view the bar graph display, with the line graph being integrated later.

The “Units” preference allows for the user to select a unit that can be used for displaying the actual values on the graph. These include “Shekel” and “kWH”, and adding additional currencies can be integrated at a later time.

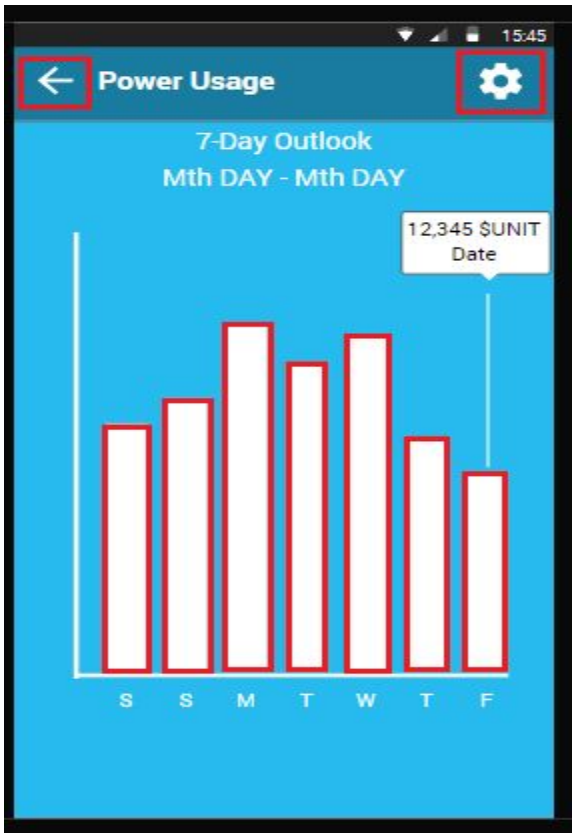
LoadingSpinner

When the user is asking for a large amount of data, loading times can drastically increase as millions of seconds worth of data will be loaded. To ensure the user that the application is loading, a loading spinner will be implemented. This does not have to exist as an entirely separate activity and can be defined using the pre-built ProgressBar utility in android.



An animated spinner will be shown above the graph activity while loading data. This is not a separate activity, but rather a view placed above the graph.

FullscreenGraphActivity



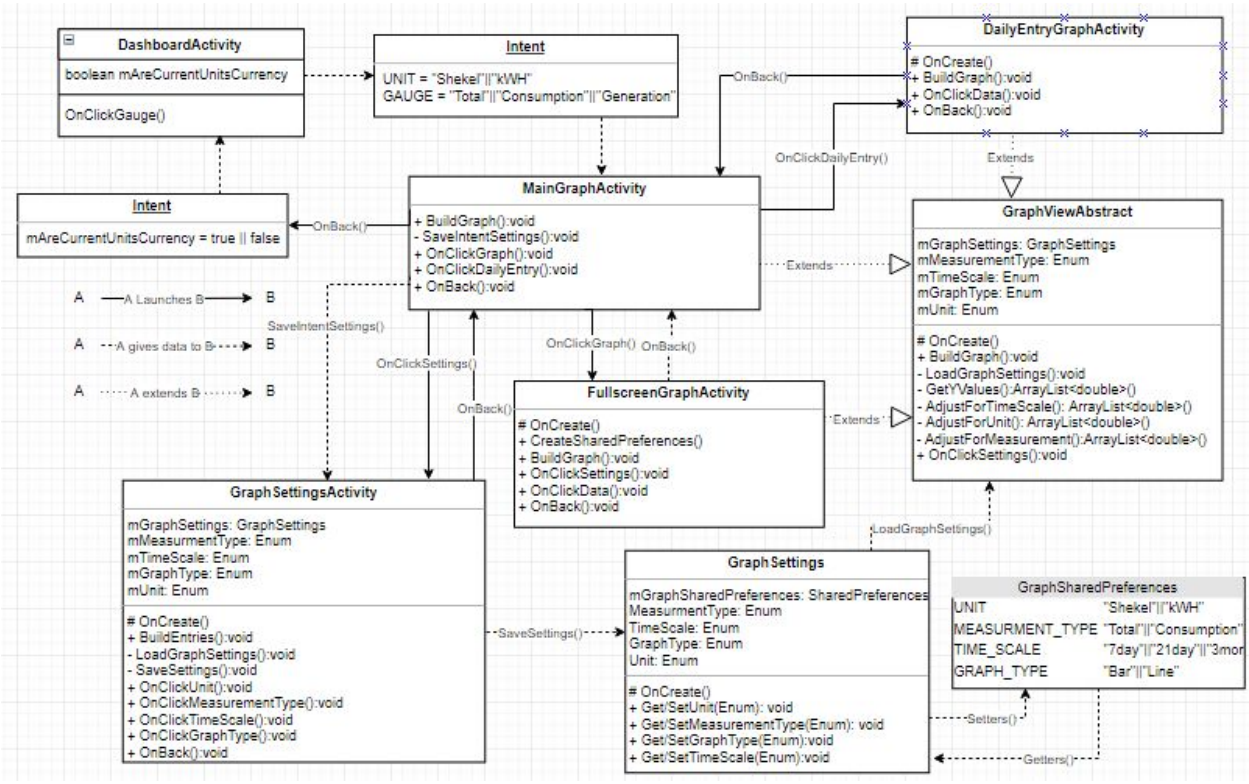
In the fullscreen activity, individual data points are selectable for additional detail, and more room is available.

The fullscreen activity is launched upon clicking on the graph from the MainGraphActivity and offers additional detail and features beyond more room to display information. The entries in the graph view are now individually selectable and can display the date and data is a key feature of the graph. The data should pop out as a small bubble indicating which piece of data was selected. This gives the user the option of getting a more exact reading of the power usage for that time period.

The top bar also changes to have a back button to bring the user back to the previous view. The preferences set will remain constant and the graph in the default view reflects the preferences set. Pressing the back button will accomplish the same thing as clicking the arrow in the top left corner.

IMPLEMENTATION MILESTONES

Flowchart



MainGraphActivity with a PowerGraphView that extends the GraphViewAbstract with no customization options - 15.5HR

This first milestone will serve as the first step in creating a full and responsive graph activity by starting with a simple and immutable activity. In this state, the activity is launched by clicking any of the gauges on the dashboard. The graph is locked to a 7-day timespan, bar graph and no settings activity can be launched. However the graph can be changed depending on which gauge is pressed, and the units that the user has selected, as this information is found on the dashboard. Settings are grabbed from SharedPreferences and put into member variables to minimize SharedPreference calls.

These member variables will be Strings which indicate which setting has been selected by the user. The possible values that each flag can take on will be set in the class . They will follow a format following a format of “GRAPH_SETTINGS_A_B” where “A” is the graph setting, and B is the value it is taking. For example, if I want to check if the user has set the time scale to 7 days, than mTimeScale will take on the value of “GRAPH_SETTINGS_TIME_SCALE_7DAY”. Whereas setting the unit to shekels sets mUnit to “GRAPH_SETTINGS_UNIT_SHEKEL”. Below are the steps needed for the first milestone.

1. Create a new activity with its own xml layout such that the content of the activity is split between the GraphView and the DailyEntriesView. **2HR**
2. Create the GraphViewAbstract containing the variables and methods described in the diagram above. **2HR**
3. Create a generic OnGaugeClick() in DashboardActivity which is called by each of the three gauges upon being clicked and creates an intent including the current unit the user has set and the gauge the press is coming from. **2HR**
4. Implement CreateSharedPreference() in the GraphViewAbstract which at this point, stores the UNIT and MEASUREMENT_TYPE that was passed through the intent. **1.5HR**
5. Create the LoadGraphSettings() method which will get the UNIT and MEASUREMENT_TYPE values from SharedPreferences and set them to the proper members variables (mUnit, mMeasurementType). **1.5HR**
6. In BuildGraph() construct an AggregateBatch that gets the data from the past 7 days of usage. **2HR**
7. Implement a GetYValues() which returns a list of 7 values. This method will be fully implemented to respond to mTimeScale. **1HR**
8. Implement AdjustForUnit() to check the value of UNIT convert the numerical value of values in the double[] using the EnergyUtil if necessary. **1HR**
9. Within BuildGraph(), construct a simple 7 value bar graph using 7-day results, the setting member variables and the Getting Started page. This should loop through the data values and format the size of each. **4HR**

Launchable GraphSettingsActivity that saves to SharedPreferences - 9HR

Saving to SharedPreferences is the best way to have settings be consistent through the use of the app as it allows for common data across activities. The GraphSettingsActivtiy will contain methods for handling click events and other operations needed for the settings activity itself. There will be a GraphSettings class where units can be set and gotten easily from SharedPreferences. If any of the graphing activities were to need the user settings, they would be coming from the GraphSettingsActivity page, which uses GraphSettings to get these settings from SharedPreferences This milestone is related only to setting the SharedPreferences in a new activity and involves no steps in drawing the graph.

The SharedPreferences should only be accessed when building and leaving the GraphSettingsActivity. Accessing the SharedPreferences is more costly than accessing predefined local variables, and if the user is changing multiple settings then it is worthwhile to limit actual writes to the SharedPreferences only when entering and leaving.

1. Create a new activity with its own xml layout that follows as a PreferenceScreen demonstrated in the wireframes above for the settings activity. **2HR**
2. Add in a settings icon in the top right corner of the top bar to launch this new activity when clicked. **1.5HR**
3. Create ListPreference entries for UNIT, MEASUREMENT_TYPE, TIME_SCALE, and GRAPH_TYPE. Make sure that the key matches those set before in the previous milestone as “SHARED_PREF_KEY_XYZ” **0.5HR**
4. Following the above link, make sure to set proper values, default values, titles etc. by using the localized strings for all of the entries. **1HR**
5. Create a LoadSettings() method which takes values from SharedPreferences and saves them to variables for the GraphSettingsActivity to access. This should be called in the activity’s OnCreate() **1.5HR**
6. Create a SaveSettings() method which takes values set by the user and saves them back to SharedPreferences. **1.5HR**

Graph Responds to measurement type changes - 4HR

While the user is able to change which measurement type is used by selecting the respective gauge, they should also be able to adjust this from the graph settings. This milestone will result in

a response of the graph from changing the “Measurement Type” setting. The changes to be made in this will be done in the GetYValues() method of the GraphViewAbstract.

1. Within GetYValues() the UNIT check should already be taken care of from the last milestone and the values of shekel/kWH should already be accounted for.
2. To minimize calculations and conversions, the MEASUREMENT_TYPE check should be done before unit conversion. This means the next steps should be done before the steps done in the previous milestone within GetYValues().
3. Implement the AdjustForMeasurement() to check the value of mMeasurementType using a switch statement and based on its value, create an ArrayList<double> of either the “Consumption”, “Generation”, or the difference between the two. **1.5HR**
4. Adjust the previously implemented code to do unit conversions on this newly created ArrayList. **1HR**
5. Return the fully formatted ArrayList<double> and ensure that BuildGraph() gets the proper values to graph. **1.5HR**

Graph Responds to Time Scale changes - 10HR

The GetYValues() method should now be fully implemented, but the graph is only receiving the past 7 days worth of data. This milestone is to have the graph respond to requests to get a different time frame from the user. The timeframes available are: “1-Week”, “1-Month”, “3-Month”, “1-Year” and are outlined above in the design section. TImeScale adjustment must be done before GetYValues() is called so that the proper aggregates can be combined in a list that GetYValues() uses. Thus the implementation should be done in BuildGraph() and call for a proper AggregateBatch.

1. Previously the AggregateBatch made will call for 7 DailyAgreggates, now it must adjust to get a variable amount based on mTimeScale.
2. Implement the AdjustForTimeScale() method to use a switch statement based on set enums, create different AggregateBatches for each of the above described timeframes. **2HR**
3. Format the size of the bars to take up an equal size and leave no awkward gaps between datapoints. Take time to ensure spacing by testing heavily in this step for each of the time frames. **3HR**

4. Funnel the single network call into an ArrayList that can be used by GetYValues() to adjust for the remaining settings() **2HR**
5. BuildGraph() should already properly loop through and add values to the graph, regardless of the size of the list.

Scrolling list of Daily Entries - 9 HR

At this point the graph should be working correctly and be adjustable in Units, Measurement type, and time scale. With the graph in this functioning state, the daily entries list should be implemented before adjusting the graph again. Instead of using the typical ListView, it is best to use a RecyclerView as it will integrate better with infinite scrolling and ensuring memory use is kept at a minimum. An example use of AdapterView can be found here. This implementation should also use an ArrayAdapter to add entries, and data will have to be handled by an Adapter, such that new DailyAggregates will have to be added through the Adapter. Use this Github repo.

1. Create the RecyclerView on the MainGraphActivity to contain the daily entries. It should initially take up half the screen, but when scrolled, should overtake the rest of the screen as described in the wireframes above. **4HR**
2. It is possible to recycle methods such as GetYValues() to get the proper data for creating a daily entry by simply calling these methods on a separate AggregateBatch for the daily entries. Adapt these methods and format batch properly to have an ArrayList of each individual day's data. **2HR**
3. Create a Custom Adapter which extends the Recycle.View. Adapter as shown in the example project. This Adapter should pull new data from an AggregateBatch of 14 days worth of data. **2.5HR**

Have Daily Entries launch a 24-Hour graph on clicking - 4.5 HR

By creating a GraphViewAbstract, this milestone should be rather simply to implement, as there are only distinction that have to be made between this fullscreen graph and the abstract class. One difference lies in AdjustForTimeScale(), which can be overridden in this activity. We only want a 24 hour time frame for the DailyEntry that was selected, so changing the time frame should not be possible. Another distinction should be made in launching the settings from this

view. The Time Scale option should be grayed out and selection should not be possible when coming from this view.

1. Override the AdjustForTimeScale() method by restricting to only 24 HourlyAggregates. **2HR**
2. Adjust the OnClickSettings() to pass along a boolean value, isFromDailyEntryGraph, that in this case is true. In the OnStart(), handle this value from the intent and if true, gray out the Time Scale option. **1.5HR**

Insert Loading Spinner while loading the graph- 3.5HR

Loading large amounts of data can occur when using this feature, so a loading spinner is needed to assure the user that nothing has gone wrong. This can be placed as a view that is set as visible while data is being received, and can be set to invisible once the graph has finished building. It is not a full progress bar, and does not indicate percentage towards completion. This will apply to all graphs and thus should be present in the top level abstract class.

1. Add the ProgressBar View to the activity xml and set its initial visibility value to visible. This view should take up the whole screen and no other elements should be placed along with it. **1HR**
2. In BuildGraph(), start the view as visible, and at the end, set it to invisible. Test this by clearing the cache and running the graph on large time frames. **2HR**

Have the MainGraphActivity respond to clicking to launch FullscreenGraphActivity - 4HR

With the MainGraphActivity in its current state, the only important feature remianing is the ability to expand the graph into a larger view. As shown in the design above, the graph should be clickable and react to a click by launching the FullscreenGraphActivity, which is another extension of the GraphViewAbstract.

1. Create and layout an activity containing only a GraphView object that encompases the entire screen. **2HR**
2. Using the an OnClickListener, set the whole GraphView of the MainGraphActivity to launch the FullscreenGraphActivity. **1HR**

3. The GraphView Abstract should already be equipped to create and draw the graph. Test and ensure that the graph is correct and no errors arise when using in a full screen setting. **1HR**

Add the ability to click on data points for additional detail on the FullscreenGraphActivity - HR

Have Graph Respond to Graph Type - HR

The GraphViewAbstract at this point will just build a Bar graph with no ability to adjust to line graph based on the settings. This milestone will adjust the top level BuildGraph() method to also call an AdjustForGraphType() method that creates proper data points for Bar vs. Line graphs. However, this is a low priority setting and may not be fully implemented as the addition of a new graph type will not influence user experience as much as previously mentioned settings.

Stylize the graphs to match app style - 7.5 HR

Once the functionality of the graph has been reached there are going to be changes to the style. This can range from axis value scales, colors, scale of bars or lines etc. Changes should be made in the GraphViewAbstract. Below are some of the steps that will definitely need configurations:

1. Format color of graphs to follow the design set in the rest of the application. This includes bar/line colors, background color, font use(although it should be fine), size/scale, location of labels on axes. **3HR**
   (https://github.com/PhilJay/MPAndroidChart/wiki/Setting-Colors)
   (https://github.com/PhilJay/MPAndroidChart/wiki/General-Chart-Settings-&-Styling)
2. Ensure the scales on the X and Y axes scale properly to changing units and graphing types when going to fullscreen. **1HR**
3. When selecting a data point in the full screen view, change the color to be a slightly gray to stylize as selected. **0.5HR**

4. Have proper formatting for markers to follow the time scales shown in the wireframes. Ex. The 24 hour graph should have carrots denoting 0:00, 6:00, noon, 18:00, 23:00. **1HR**
5. **Total Time: 7.5HR**

Low Priority Features

1. Hold down on graph to save to gallery.
    a. This is a simple feature that can be easily implemented as follows:
        i. Add in a “WRITE_EXTERNAL_STORAGE" permission to the manifest. (Are we okay with this?)
        ii. Set an OnLongPressListerner on the fullscreen graph view that calls MPAndroidChart’s saveToPath(String title, String pathOnSD) method **0.5HR**
        iii. Give the picture a title of “DATE_MONTH_YEAR-SCALE_TYPE.jpg” and save it to a “/KedmaSolar/” path on the SD card. **1HR**
        iv. **Total Time: 1.5HR**
2. Animations when drawing the graph.
    a. Go through the graphing library and find a good animation for the bar and line charts. Bar will most likely be a “build bars up from the x-axis” while line might be “build the chart from right to left” animation. Just do some searching and see which is best. **1.5HR**
    b. If any of the predefined ones are good then just attach mChart.animate()
    c. **Total Time: 1.5HR**
3. Add in option to view Consumption and Generation separately as opposed to only showing the difference between the two. To do this, the following additions have to be made:
    a. The settings view needs another segment in the ListView to change the measurement.
    b. Another settings flag needs to be added to SharedPreferences.
    c. GetSettings() needs to be updated to return this new flag as well.
    d. The last thing to change is getting the correct data from the aggregates based on this setting. In the GetData(List settings) method implemented before, there must

be an additional check for the measurements flag. Before the difference between consumption and generation was the only value we cared about. Simply make a switch statement that alternates between difference, consumption and generation. **1HR**

e. **Total Time: 1HR**

4. Add option to see an “average” and “overuse” use line on the graph.
    a. Using the LimitLine bult in functionality, average and overusage lines can be constructed.
5. Daily/Weekly/Monthly Averages on DefaultGraph view - On the default screen it should be possible to add in an average count on the top.
    a. The averages would have to follow these conventions:
        i. 1-day should have an hourly averages for today’s power usage
        ii. 7-day should have a daily average for 7 days.
        iii. 21-day should have a weekly average.
        iv. 3-Month should have a weekly average
        v. 1-year should have a monthly average
    b. To implement there must be considerations involving where to insert code.
        i. It is best to create a new method to be called after GetData() called FindAverage(List Settings, List Data) and returns the average as a well formated double. This can be added to the graph’s title as “Daily Average: W,XYZ $UNIT”. To do this the following should be done:
            1. (If Low priority feature #3 has been implemented) Check to see whether we are getting an average for the consumption, generation, or the difference with a switch statement and settings flag check. **1HR**
            2. Loop through the List of data based on the scale set. Doing it this way will allow for an easier time making it work on the full screen view with infinite scrollability. **2HR**
                a. 1-day #loops = hours of day recorded. (If today this number is < 24)
                b. 7-day #loops = 7 days; dayAverage = 7-day total / 7
                c. 21-day #loops = 7 days; weekAverage = 21-day total / 4

d. 3-month #loops = 12 weeks; weeklyAverage = 12-week total / 12
e. 1-year #loops = 12 month; monthlyAverage = 12-month total / 12

3. Once calculated there is just some floating point formatting that needs to be done before returning. Confirm a good precision based on time scale (Only 1 or 2 places after decimal point) **0.5HR**

ii. With a returned double it is best to insert this number just under the date as part of the graph title. So when building the title just add a new line with the proper string “Daily/Weekly/Monthly average: XYZ”. **1HR**

6. Daily/Weekly/Monthly Averages on Fullscreen view (With infinite scrolling) -This feature is nontrivial to add if there is no infinite scrolling done, but with infinite scrolling implemented to problem becomes more complex as the average has to now adapt to values that are currently visible on screen.
7. Infinite scrolling of data on the FullscreenGraphActivity - When displaying data on the FullscreenGraphActivtiy, the user can scroll along the X-axis to get more data from previous points in time. Essentially this would extend the time-scale functionality to not be as restricted by allowing the user to view previous timeframes that have already passed.