Nipissing University

“Stormy” A Weather App

Android App Development

Dillon MacGregor

COSC 4207 WI 2019

Contents

[1.0 Introduction 4](#_Toc6065799)

[1.1 Technologies 4](#_Toc6065800)

[2.0 Preliminary Information 4](#_Toc6065801)

[2.1 What is an API? 5](#_Toc6065802)

[2.2 SDKs 5](#_Toc6065803)

[2.3 IDEs 5](#_Toc6065804)

[2.4 Networking 5](#_Toc6065805)

[3.0 App Development 6](#_Toc6065806)

[3.1 Setting up Android Studio 6](#_Toc6065807)

[3.2 Add APIs to the Application 6](#_Toc6065808)

[3.3 Making a Request to Dark Sky with OkHttp 7](#_Toc6065809)

[3.4 Handling Errors 9](#_Toc6065810)

[3.5 Parsing JSON Data 13](#_Toc6065811)

[3.6 Adding Icon Images 15](#_Toc6065812)

[4.0 User Interface Development 17](#_Toc6065813)

[4.1 Interface Constraint Layout 18](#_Toc6065814)

[4.2 Data Binding Library Connecting Data Model to View 19](#_Toc6065815)

[4.3 Updating the Icon Image View 21](#_Toc6065816)

[4.4 Refreshing the Interface 22](#_Toc6065817)

[4.5 Improving Data Loading Feedback for the User 25](#_Toc6065818)

[4.6 Adding App Icon 26](#_Toc6065819)

[5.0 Location Services 27](#_Toc6065820)

[5.1 Adding the Location API 27](#_Toc6065821)

[5.2 Adding Location Permissions 27](#_Toc6065822)

[5.3 Dynamically Retrieving the Users Location 27](#_Toc6065823)

[6.0 Adding Additional Data 31](#_Toc6065824)

[6.1 Adding Another Activity 31](#_Toc6065825)

[6.2 Passing Data to Another Activity 32](#_Toc6065826)

[6.3 Receiving Data in Another Activity 32](#_Toc6065827)

[6.4 The Second User Interface 35](#_Toc6065828)

[7.0 Conclusion 37](#_Toc6065829)

[7.1 Suggestions for Improvement 37](#_Toc6065830)

[7.2 Github Link 37](#_Toc6065831)

# 1.0 Introduction

The purpose of this document is to introduce Android App Development by walking through the entire process of creating an app using Android Studio 3.3 and multiple APIs. The app developed in this document is a weather app that uses the Dark Sky API and OkHttp API to dynamically determine current weather conditions for the user’s location. A number of the topics discussed in this tutorial was inspired by the tutorial provided by TeamTreeHouse available at: <https://teamtreehouse.com/library/build-a-weather-app-4>. However, a number of improvements were made.

## 1.1 Technologies

This app was created using the technologies listed below. However, versions may change resulting in slight changes to the overall process. Fortunately, the process should mainly remain the same. Please feel free to update this document if any of the listed technologies receive major changes and require code alterations. The entire project code will be provided in a ZIP file and explained throughout the document.

* Android Studio 3.3 using the following:
  + JRE: 1.8.0\_152-release-1248-b01 amd64
  + JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o
  + Running on Windows 10 10.0
* Dark Sky API (1000 free calls per day) <https://darksky.net/dev/docs>
* OkHttp API 3.12.1 <http://square.github.io/okhttp/>
* Google play location services API 16.0.0 <https://developer.android.com/training/location>
* Languages:
  + Java
  + XML

# 2.0 Preliminary Information

This application uses multiple technologies that should be clearly defined before proceeding. Key technologies and concepts that need to be discussed include, APIs, IDEs, SDKs, and basic networking information.

## 2.1 What is an API?

An API is an “Application Programming Interface” and are commonly used in virtually every commercial application. The interface of the API allows one to use an external or third party SDK or system. APIs allow developers to think abstractly by treating APIs as a sort of “black box” that perform a number of complex tasks. For example, Android Studio provides a number of APIs for button interaction and activity functionalities. Third party APIs such as Dark Sky and OkHttp perform additional complex tasks such as gathering weather data for us and sending HTTP requests and receiving responses.

Using APIs is essential for any effective developer. They allow developers to save time by performing the heavy lifting of applications and complex tasks such as gathering and processing vast amounts of weather data.

## 2.2 SDKs

An SDK is a “Software Development Kit”. Software development kits contain all the pieces required for a software system. Additionally, an API is part of an SDK, that is, an API is the interface to an SDK where all the work is done. This allows one to simply satisfy the interface requirements of an API and acquire useful results without needing the underlying components of the SDK.

Some examples of SDKs include JDK (Java Development Kit) and Android SDK which this application uses extensively.

## 2.3 IDEs

An IDE is an “Integrated Development Environment”. IDEs are software applications, such as Android Studio, Eclipse, and Visual Studio, that provide multiple tools for software development. The tools included generally contain, source code editors, build automation tools, and a debugger. Additional tools such as version control and code suggestion functionality (each contained in Android Studio) are also sometimes included or can be added.

## 2.4 Networking

In order for this app to have powerful features it must be able to communicate with the web. For this feature the OkHttp API is used to do most of the work of sending and receiving customized HTTP messages. This is essential in order to work with the Dark Sky API which follows a RESTful programming pattern or “Representational State Transfer” programming pattern. RESTful programming APIs are generally web APIs which act as a black box and provide all the required information to a user through what is called an “identifier”. The “identifiers” in this case and most cases are URLs such as “[https://api.darksky.net/forecast/[apikey]/[latitude],[longitude](https://api.darksky.net/forecast/%5bapikey%5d/%5blatitude%5d,%5blongitude)]”. Since the Dark Sky API is web API that follows the RESTful programming pattern a basic idea of HTTP methods will be beneficial.

The OkHttp API is utilized in this application in order to send HTTP messages mainly GET, and POST requests to the Dark Sky identifier URL. The OkHttp API will also receive the responses from the Dark Sky API allowing the app to display the weather data received.

# 3.0 App Development

Section three walks through the entire app development process for the weather app. This includes setting up the Android Studio IDE, adding the external APIs to the application, describing main classes and features, interface development, and introducing features of Android Studio along the way.

## 3.1 Setting up Android Studio

Note that version changes to Android Studio may alter this procedure. First download and install Android Studio including the Android SDK at <https://developer.android.com/studio/>. After installing Android Studio open the application and create a new project. Give the application a name and click next. Select API version 19 KitKat as that was used in this walkthrough then click next. When prompted for an activity selection just create an empty activity and click next keep the activity name as the default MainActivity.

## 3.2 Add APIs to the Application

Since the Dark Sky API is simply a web API no dependency needs to be added to the project for this API, instead go to <https://darksky.net/dev/register> and create a free account to obtain a unique private API key.

In order to use the web API the OkHttp dependency needs to be added to the project. In order to do this open the “build.gradle (Module:app)” file under the “Gradle Scripts” directory in the IDE. See figure 1 below for reference. Go to <http://square.github.io/okhttp/> and scroll down to the download section where a gradle example should appear. Simply copy and paste the line of code (at the time of making this document it was: “implementation 'com.squareup.okhttp3:okhttp:3.12.1'”) and add it into the “dependencies” section of the “build.gradle” file mentioned above and shown in figure 1. Now the OkHttp should be available to the application.

## 3.3 Making a Request to Dark Sky with OkHttp

Figure : Dependency File for Android Studio

In order to optimally make a web request to the Dark Sky API with OkHttp while running the app, the request should be made on a separate asynchronous thread to prevent the app from freezing by waiting for a response on the main thread. Recall that a thread is a simple process of execution. By having multiple threads, the app is concurrent and can support multiple tasks executing at once and allow users to continually interact with the app even when new data is loading. Additionally, if the main thread is backed up by waiting for network responses then no user interface (UI) updates can occur since in Android Studio requires all UI updates to be done on the main thread.

Go to <https://github.com/square/okhttp/wiki/Recipes> and view the asynchronous get method. In order to test the Dark Sky API and ensure that the private API key is working properly create a test request by adding the following code (figure 2) to the MainActivity class. The keyboard shortcut ALT+ENTER should automatically include the imports you need ensure you use the okhttp3 libraries for the networking code. Next under the manifests folder, open AndroidManifest.xml and add “<uses-permission android:name="android.permission.INTERNET"/>

” above the opening application tag like in figure 3 below.



Figure : MainActvity Class For Initial Request

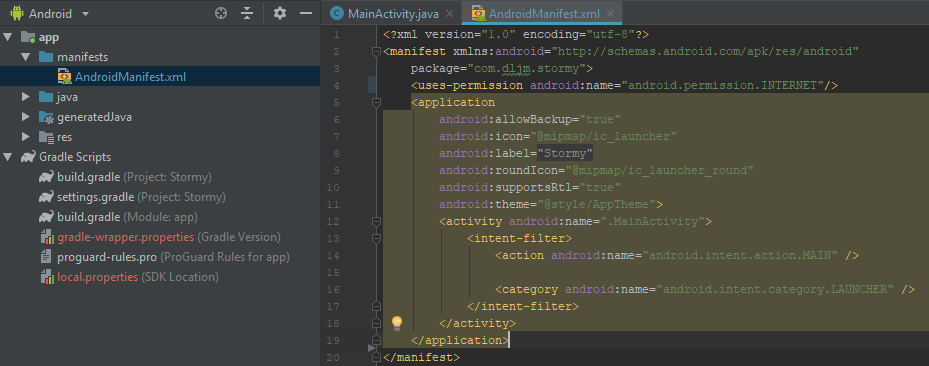


Figure : Android Manifest file for App Permissions

Once the above code is implemented build the app by clicking the build button in the top right toolbar or top left toolbar or by pressing CTRL+F9. If the build is successful run the app with SHIFT+F10 or by clicking the play button near the top right of the IDE. This should open a prompt window asking for a deployment target, select one of the available virtual devices or add a new one then click ok. An android phone emulator should appear and show the app. Check the logcat in Android Studio in the button left to view output from the app. If everything is functioning correctly a line formatted in JSON (JavaScript Object Notation) should appear containing forecast information (see figure 4). Otherwise, an error may have occurred and perhaps the URL request is incorrect. Ensure the request is formatted properly and using your own unique private API key.

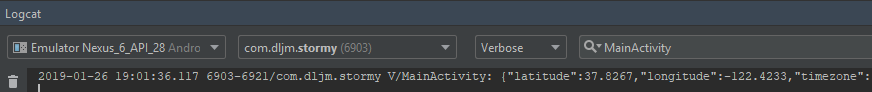


Figure : Dark Sky API Response in Logcat

## 3.4 Handling Errors

In order to handle errors and notify the user of what is going on the app uses a class called “AlertDialogFragment”. To create this class, open the java folder directory in Android Studio and right click the subfolder containing the “MainActivity” class. Then select new 🡪 Java Class 🡪 Add the details indicted by figure 5 below 🡪 Click ok.

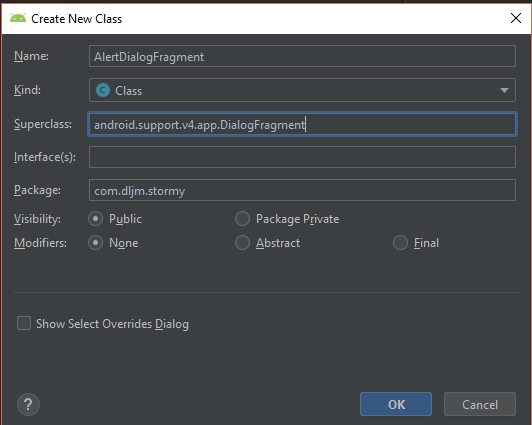


Figure : AlertDialogFragment Class Creation

The “AlertDialogFragment” class is used to create a popup in the app to display an error message created by the developer. The class is shown below in figure 6, notice that the error title will always remain the same but the message will change depending on what is passed to the “args” variable. Also, notice that the messages are stored in “app🡪res🡪values🡪strings.xml” for easier localization in the future and text management. One can easily add strings to this file by clicking on the string in the java class and pressing ALT+ENTER 🡪 Extract String Resource.

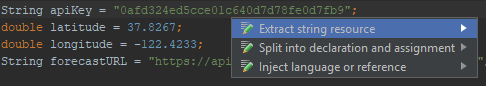




Figure : AlertDialogFragment Class

Now that the class for displaying alerts was created, objects from the class need to be created and used in “MainActivity” to fully implement the alerts in the app. To do this the method “alertUserAboutError(String message)” will be created and called when the network connection is not functioning properly or when the response from the Dark Sky API is not successful. Determining whether or not the network is available is more difficult than checking if the response is successful. In order to check the network connection, the app will need an additional permission in it’s manifest as shown in figure 3, this time add the permission: “<uses-permission android:name="android.permission.ACCESS\_NETWORK\_STATE"/>”. Once the above has been performed view the updated “MainActivity” code segments below to implement error notifications for the user. After adding the new permission and changes below try running the app and turning on airplane mode within the emulator. Then restart the app on the phone to view the error notification stating the network is unavailable (remember to add your own messages to the strings.xml file manually or by the process above figure 6).



Figure : Helper Functions in MainActivity Implementing Error Notifications



Figure : Error Checking Code in MainActivity.onCreate

## 3.5 Parsing JSON Data

Many Android Apps use the MVC (Model View Controller) design pattern and this app is no exception. The view for this app will be the UI that the user interacts with, the model will be a set of Java classes used to store data retrieved by the Dark Sky API, and the Controller consists of the activities that present views and respond to user input within the app. JSON stands for JavaScript Object Notation and is an important data format used in many applications and RESTful APIs. JSONs consist entirely of data and are represented as structure objects with key value pairs, making them easy to process with an object oriented language such as Java. They are also small in size in comparison to XML data formats making them faster for sending over the internet. Overall, learning how to work with JSON has many benefits as it is used heavily in the software industry.

To see the data received from the Dark Sky API in JSON format simply go to <https://api.darksky.net/forecast/yourAPIkey/37.8267,-122.4233> in your browser. This will give you an example of the JSON data format, to make the webpage more readable install a JSON viewer extension from the chrome store. To handle the JSON data in this application create a new class that will represent a POJO (plain old java object) called “CurrentWeather” to hold the JSON data received from Dark Sky like the one shown in figure 9 below (simply double click on the image to view entire class code). Note that the object will not contain all the data received from Dark Sky at this stage but can be added in the future for further app functionality.

In order to parse the JSON object Andriod’s JSONObject class (<https://developer.android.com/reference/org/json/JSONObject>) is used. A function “getCurrentDetails” is created in “MainActivity” which returns a “CurrentWeather” object containing the location, timezone, humidity, icon, time, weather summary, temperature, and precipitation chance. The response body from the Dark Sky API contains the JSON data needed to be parsed so the body is stored as a string and passed to the “getCurrentDetails” function. Within the function a JSONObject is created from the response body string received from the Dark Sky API. The changes in “MainActivity” are shown in figure 10 below, refer to the link above for more information on the JSONObject class in Android Studio.



Figure : CurrentWeather Class POJO for Model Component



Figure : MainActivity Class JSON Data Handling

## 3.6 Adding Icon Images

You may have noticed in figure 9 that there are provided icon images for each icon string returned by the Dark Sky API. The icon images are contained in a file named “ProjectResources.zip” where this document is located. If you have not retrieved to final project from GitHub and are following along and need to import the icon images perform the following steps: open the “ProjectResources.zip” file, copy all the files within the zip file to the “res” folder in the Android Studio project by right clicking “res” under the “app” folder and selecting “Show in Explorer” (see figure 11) from the option list that appears, select the replace all files in destination folder if prompted. After following the above steps, all the icon images should be imported into your project resource file.

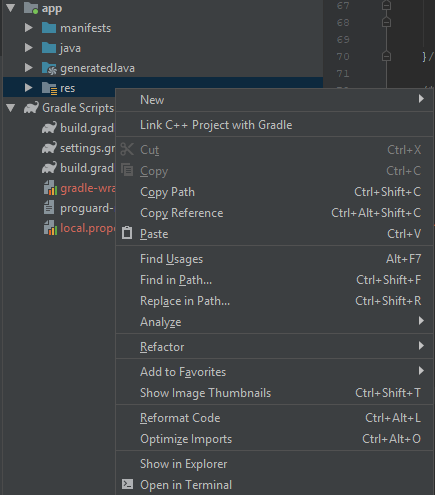


Figure : Opening the "res" Folder in File Explorer

# 4.0 User Interface Development

Most of the user interface work is done in “activity\_main.xml” under the “app🡪res🡪layout” folders. The interface design is shown in figure 12 below. The interface is rather simple and only requires one activity. The required images are already imported into the project if you added the icon images in section 3.6.



Figure : Interface Mockup

## 4.1 Interface Constraint Layout

Feel free to adjust interface components such as text sizes, colours, or positions throughout the interface development process. Android Studio provides an interactive editor for creating user interfaces. References to figure 13 below are made a few times while describing how to create the interface. On the left side of “activity\_main.xml” (design view) the attributes for a selected component are displayed. There is an option (View all attributes) at the bottom of the “Attributes” tab in order view all the attributes for further component configuration, however, most configuration can be done in the standard view shown below.

To add a component, click and drag the type of component you would like to add from the right “Palette” tab. For this particular app “TextView” and “ImageView” components are used. Everything in the figure below besides the cloud and degree symbol are “TextViews” with either a white text colour or half white text colour (white with 50% opacity). Each component will eventually be made dynamic and display the data stored in the “CurrentWeather” object previously created.

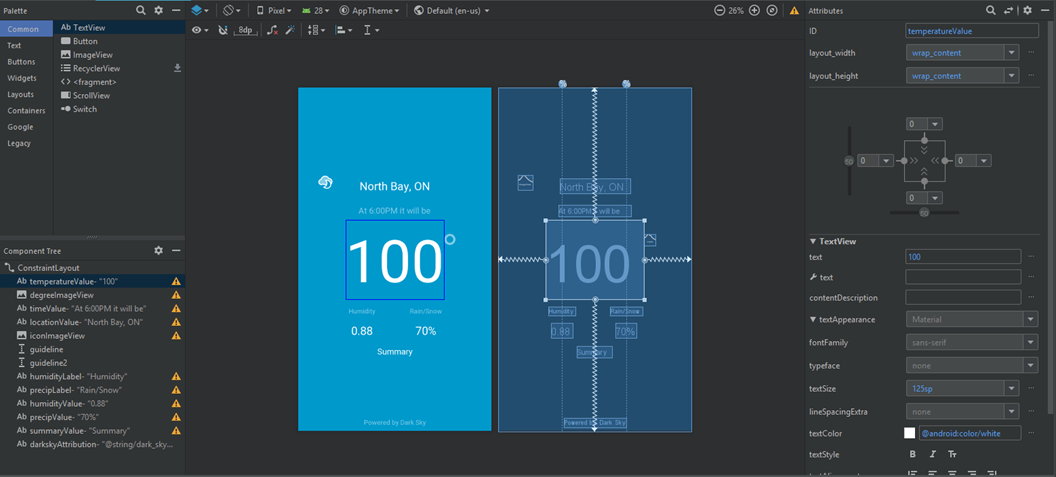
 An important aspect for constraint layouts is of course constraints. Since this concept is hard to fully explain, view Android Studio’s documents for constraint layouts: <https://developer.android.com/training/constraint-layout/>. Try to build the interface shown below setting appropriate constraints, text sizes, placements, and spacing you like.

Figure : activity\_main.xml Design View

## 4.2 Data Binding Library Connecting Data Model to View

First we need to add the Dark Sky attribution link in our app as required by their terms of usage. Figure 13 shows the attribution link in the bottom of the interface. Simply add a new text element to the interface and set the ID to “darkSkyAttribution” and the text to “Powered by Dark Sky” (you can also use the wrench icon that appears to automatically create a new string resource where you can set the redirect link). To make the text attribute a link to the website add the URL to the text by making it a string resource. This can be done in the “strings.xml” file under the “values” folder by adding the resource: “<string name="dark\_sky\_message"><a href="https://darksky.net/poweredby">Powered by Dark Sky</a></string>” and setting the text to “@string/dark\_sky\_message” (if you used to auto-generate method the new string resource will have been auto created already). Next, add the lines of code in figure 15 that define a new text view called “darkSky” that supports the “LinkMovementMethod” method and that is all that is needed for the attribution link.

In order to bind the view component to the data model Android’s data binding library is used. This is not the only way to bind view components to data components but is convenient to use. In order to set up the data binding library a few changes need to be made to files within the project. To easily understand what is going on, read the Android documentation on data binding: <https://developer.android.com/topic/libraries/data-binding/#java> and <https://developer.android.com/topic/libraries/data-binding/start>.

Make the changes to the “build.gradle” app file as shown in the second link above. Next, make the appropriate changes to the “activity\_main.xml” file described by <https://developer.android.com/topic/libraries/data-binding/expressions> and illustrated in figure 14 below. Note the new “layout”, “data”, and “variable” tags. The “variable” tag declares the property or class object used within the view. In figure 14 below the class is “CurrentWeather” and the instance of the class is named “weather”. This enables one to get data values stored within the weather object as shown by the line: “android:text="@{String.valueOf(Math.round(weather.temperature)), default=`20`}"”. Replace all of the hard-coded text values throughout your xml document with data values stored in the weather object by using “@{weather.nameOfDataVariable}” (remember the value within the parenthesis must be a string so use String.valueOf() where necessary).

Now that the “activity\_main.xml” file is set up properly the “MainActivity” class needs to implement the data binding library. All of the code shown in figure 15 is explained in the links above and with comments so less detail will be given here. Simply add the code in figure 15 to the “MainActivity” class in order to finish connecting the data model to the view allowing text fields to be dynamically updated.



Figure : activity\_main.xml Content



Figure : MainActivity Code Additions

## 4.3 Updating the Icon Image View

All of the view data is updating dynamically except for the image icon that displays the current weather conditions. To change the image dynamically depending on the weather the “findViewById” method can be used. This is the same method used for the Dark Sky attribution link in the UI.

To implement the dynamic image icon a few lines of code are needed in “MainActivity”. A new private variable called “iconImageView” and a tricky function implementation is needed. The function “runOnUiThread” is needed to update the icon image since all UI updates need to be done by the main thread. However, as you may remember, the response processing is done by a background thread to support asynchronous functionality. The new code is shown in figure 16 below.



Figure : Updating the Icon Image

## 4.4 Refreshing the Interface

The interface mock-up in figure 12 shows a refresh image icon that acts as a button to refresh the user interface including the weather details displayed in it. In order to support the refresh functionality some code refactoring is necessary. In the “onCreate” method in “MainActivity” select all of the code except for “super.onCreate(savedInstance)”, right click 🡪 Refactor 🡪 Export 🡪 Method and name the new method “getForcast”. The “getForcast” method will take two doubles for latitude and longitude, for now they will be hardcoded until location services are added. Next, add the function “refreshOnClick(View view)” that simply calls “getForcast(latitude, longitude)”. All of the new code and refactoring are shown in figure 18 below.

To make the refresh icon image clickable modifications to the “activity\_main.xml” file are required. Add the following line of code to the ImageView tag for the refresh icon: “android:onClick="refreshOnClick"” (figure 17 below). By adding this code to the xml file the “refreshOnClick(View view)” method created in MainActivity will be called whenever a user taps on the refresh icon in the UI. Note that in order for this button functionality to work the method signature must be exactly as shown in figure 18 (double click on the image to view all the code), that is, it must be a public void function that takes a single “View” parameter.



Figure : activity\_main.xml Refresh Icon



Figure : Refactored Code to Enable Refresh Functionality

## 4.5 Improving Data Loading Feedback for the User

The user interface is almost complete. In order to improve the interface even more it would be nice to provide feedback to the user when forecast data is being retrieved from the Dark Sky API. There are two important times feedback should be provided when retrieving weather data. The first is one the user first launches the app. For the first instance it would be nice to display a progress circle indicating that app data is loading, and slightly blur out the background since there is no data to display. The second instance occurs when the user taps the refresh button. For this instance, we don’t want to hide the previous loaded weather data and only want to display a small progress circle around the refresh image button. This section will walk through each of the progress circle additions.

In “activity\_main.xml” add the loading panel and progress circle by adding the code shown below in figure 19. Switch back over to the design view to adjust the positions of the progress circles in case they are not aligned properly for your UI.



Figure : Loading Panel and Progress Circle Addition

Next, the loading panel and progress circles need to be set to visible when the app is requesting weather data and removed when the data is received. This is a straightforward process but since the progress circles are UI components only the main thread can alter them. Figure 20 shows the necessary code changes to implement the progress circles and loading panel.



Figure : MainActivity Code Additions for Loading User Feedback

## 4.6 Adding App Icon

The app icon is already provided under the “res” folder. In order to use this icon simply modify the “AndroidManifest.xml” file like shown below in figure 21.



Figure : App Icon Addition

# 5.0 Location Services

The app is almost complete. However, it still uses a hardcoded location value which makes it a pretty useless weather app. To solve this issue, we will use Google Play’s location service API. To add the location service functionality the app “build.gradle”, “MainActivity.java”, and “AndroidManifest.xml” files must each be modified.

## 5.1 Adding the Location API

Adding Google location API is quite easy. Simply open the application “build.gradle” file and add the following dependency: “implementation 'com.google.android.gms:play-services-location:16.0.0'”. Be sure to check the current version of the API at <https://developers.google.com/android/guides/setup> as the version may be different at the time of reading this.

## 5.2 Adding Location Permissions

In order to use location services permission must be given by the user. Similar to section 3.3 and figure 3 a new permission needs to be added to the “AndroidManifest.xml” file. Add the permission: “<uses-permission android:name="android.permission.ACCESS\_COARSE\_LOCATION"/>” to the file similar to how the internet permission was added in section 3.3.

## 5.3 Dynamically Retrieving the Users Location

Now for the actual implementation of the location service API. A number of functions need to be added to “MainActivity” which are: onStart, checkPermissions, startLocationPermissionRequest, requestPermissions, onRequestPermissionsResult, and getLastLocation. Since location data is a serious privacy concern we must explicitly get permission from the user, simply adding the permission in the app manifest is not enough. Most of the functions are straightforward and explained in the comments. For additional information on the functions shown below please read the android location services documentation at <https://developer.android.com/training/location>.

First the private member variables and initialization process needs to be done. Figure 22 below shows the code for this process and also the updated “refreshOnClick” and “onStart” function.



Figure : Setting Up Location Services in MainActivity

The “onStart” function is called after “onCreate” and more information about activity life cycles can be seen here <https://developer.android.com/reference/android/app/Activity>. Once the “onStart” function or “refreshOnClick” function is called the location request is set in motion. The following figures show each of the required functions enabling location services.



Figure : checkPermissions function in MainActivity



Figure : startLocationPermissionRequest and requestPermissions function in MainActivity



Figure : onRequestPermissionsResult callback function



Figure : getLastLocation function

After “getLastLocation” is called when location permissions have been granted the longitude and latitude are updated. This process works reasonably effectively but the android emulator in android studio has issues with location services so most of the time the location returned in null. In order to fix this location updates can be implemented or the app works well on any android phone instead of an emulator. The last thing that needs to be done is to update the location label based on the longitude and latitude. The “Geocoder” API provided by Android Studio is used for this and shown in figure 27 below.

After making these changes the app should function effectively on any android phone. However, there are endless amounts of improvements that can be made to this app. The next section will conclude this tutorial and suggest future improvements.



Figure : Get the City Name

# 6.0 Adding Additional Data

So far all the necessary components have been provided for you to add any of the additional data provided to you by DarkSky. This section will introduce you to adding additional pages to your app in order to display increased weather details.

## 6.1 Adding Another Activity

There are a few approaches to adding additional activities to an app. The simplest is use “Intents”. A more difficult but more atheistically pleasing is to use fragments. Figure 28 shows how to use intents to add an additional activity to your app. The code is located in MainActivity.



Figure : Intent use to add activities

## 6.2 Passing Data to Another Activity

You may have noticed in figure 28 that a few objects were added to the intent object. This is necessary to properly pass data across activities. There are also two new variables introduced “hourlyWeather” and “dailyWeather”. As the name implies each of these objects contain hourly weather or daily weather data respectively. They are initialized similar to “currentWeather” see figure 29 below. The functions are called in the same location where “getCurrentDetails” is called. Also, note the “serial” names since the objects must be serialized by Java in order to pass them across activities. Each POJO class now needs to implement the Serializable class in Java. To do this simply change the class declarations to: “public class CurrentWeather implements Serializable{ ... }”.

## 6.3 Receiving Data in Another Activity

Most of the work in the new activity “ForecastActivity” is done in the “onCreate” method. Figure 30 shows how the serialized data objects are received in the new activity and also contains all of the interface binding and initialization. Note the additional binding variable initialization this is very relevant for the “activity\_forecast.xml” file i.e. the user interface.



Figure : DailyWeather and HourlyWeather Objects



Figure : onCreate in ForecastActivity

## 6.4 The Second User Interface

The second user interface is similar to the first and uses all of the techniques explained in the first interface section so not as much detail will be given here. Figure 31 shows how the second interface design. You are encouraged to design your own interface as you now have the necessary tools to do so. The “activity\_forecast.xml” file or the code for the interface is shown in figure 32. The main difference is that multiple variables are now being bound using the binding library and used at once within a single interface. This is a required trait for most real world apps.

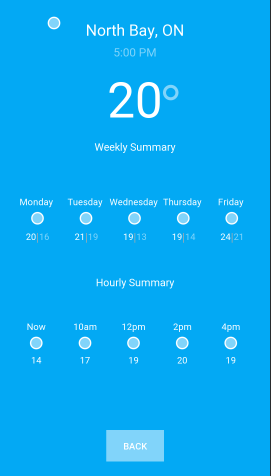


Figure : Second Interface Mockup



Figure : activity\_forecast.xml Double click to view all

# 7.0 Conclusion

This app served as a great introduction to android app development. APIs, asynchronous processing, network communications, JSON data processing, the MVC design pattern, the android data binding library, location services, and intents for multiple activities were all introduced during this introduction to android app development.

## 7.1 Suggestions for Improvement

Currently the app is very basic. There are numerous areas open for improvement. You are encouraged to improve this app any way you choose. A list of possible improvements is shown below:

* Show additional weather data there is still much more provided by DarkSky
* Instead of using a button to switch screens allow the user to swipe across activities
* Refactor project to use fragments instead of intents
* Add another view for additional weather data
* Improve the UI with new graphics (perhaps use a graph to display temperature changes from hour to hour)
* Display additional hours in the hourly forecast
* Allow the user to choose a location (harder)

## 7.2 Github Link

The completed app and all of the resources can be found in the following Github repository: <https://github.com/dljm/WeatherApp.git>.