OPEN SOURCE CODING (INTERMEDIATE)
MODULE MANUAL 2024

The Independent Institute of Education (Pty) Ltd is registered with the Department of Higher Education and Training as a private higher education institution under the Higher Education Act, 1997 (reg. no. 2007/HE07/002). Company registration number: 1987/004754/07.

DID YOU KNOW?

Student Portal

The full-service Student Portal provides you with access to your academic administrative information, including:

- an online calendar,
- timetable,
- academic results,
- module content,
- financial account, and so much more!

Module Guides or Module Manuals

When you log into the Student Portal, the 'Module Information' page displays the 'Module Purpose' and 'Textbook Information' including the online 'Module Guides or 'Module Manuals' and assignments for each module for which you are registered.

Supplementary Materials

For certain modules, electronic supplementary material is available to you via the 'Supplementary Module Material' link.

Module Discussion Forum

The 'Module Discussion Forum' may be used by your lecturer to discuss any topics with you related to any supplementary materials and activities such as ICE, etc.

To view, print and annotate these related PDF documents, download Adobe Reader at following link below:
www.adobe.com/products/reader.html

IIE Library Online Databases

The following Library Online Databases are available. These links will prompt you for a username and password. Use the same username and password as for student portal. Please contact your librarian if you are unable to access any of these. Here are links to some of the databases:

Library Website	**This library website gives access to various online resources and study support guides** [Link]
LibraryConnect (OPAC)	**The Online Public Access Catalogue. Here you will be able to search for books that are available in all the IIE campus libraries.** [Link]
EBSCOhost	**This database contains full text online articles.** [Link]
EBSCO eBook Collection	**This database contains full text online eBooks.** [Link]
SABINET	**This database will provide you with books available in other libraries across South Africa.** [Link]
DOAJ	**DOAJ is an online directory that indexes and provides access to high quality, open access, peer-reviewed journals.** [Link]
DOAB	**Directory of open access books.** [Link]
IIESPACE	**The IIE open access research repository** [Link]
Emerald	**Emerald Insight** [Link]
HeinOnline	**Law database** [Link]
JutaStat	**Law database** [Link]

Table of Contents

Using this Manual

This manual has been developed to meet the specific objectives of the module, and uses a number of different sources. It functions as a stand-alone resource for this module and no prescribed textbook or material is therefore required. There may, however, be occasions when additional readings are also recommended to supplement the information provided. Where these are specified, please ensure that you engage with the reading as indicated.

Various activities and revision questions are included in the learning units of this manual. These are designed to help you to engage with the subject matter as well as to help you prepare for your assessments.

Introduction

Welcome to Open Source Coding (Intermediate). This module follows on the Open Source Coding (Introduction) module that you have already completed. We will still be making use of Kotlin to build native Android apps, but here we will be introducing some more advanced concepts such as interacting with webservices and making use of libraries and software development kits.

Throughout this module, you will create several apps to master the intermediate skills needed to build an Android app. It is important to get hands-on experience in any programming module, so it is essential that you complete all the activities provided on Learn.

We hope you will enjoy the module and take the opportunity to use the knowledge and experience gained in both future modules, and in your career.

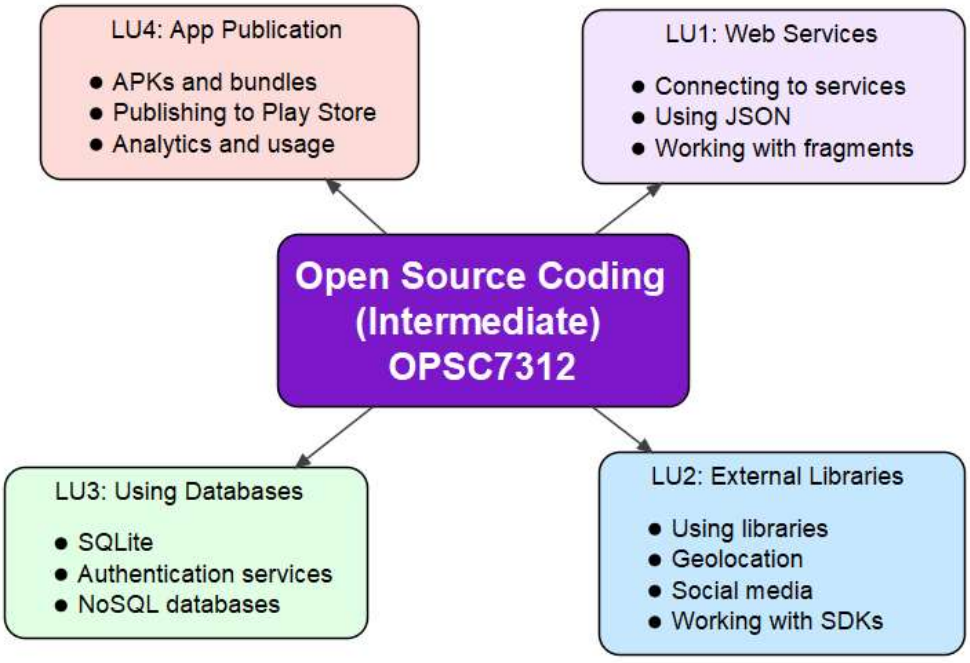


Figure 1. Module Structure

Learning Unit 1: Web Services	
Learning Objectives: • Explain how to code a Hypertext Transfer Protocol (HTTP) connection from scratch. • Explain how to connect to a RESTful Application Programming Interface (API). • Use a RESTful API in an Android app. • Explain the purpose of JSON files. • Use JSON objects to read data. • Use JSON objects to write data. • Use a library to work with JSON data. • Explain the purpose of using fragments in an Android app. • Apply fragments in an Android app.	**My notes**
Material used for this learning unit: • GitHub repository: LearningUnit1	
How to prepare for this learning unit: • Make sure you have Android Studio installed and fully updated	

1 Introduction

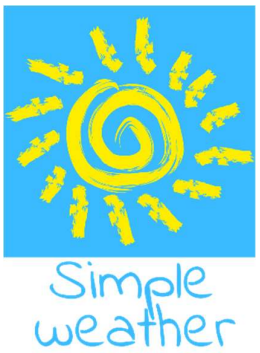
In Open Source Coding (Introduction) (OPSC7311), we have learned how to create basic Android apps with multiple activities. We have made use of the a NoSQL database to store data in the cloud. And we have made use of phone features such as taking a photograph.

In this module, we are going to build on what we have learned already to create more feature rich applications.

In this first learning unit, we are going to use existing web services to provide data to our application. Although creating and hosting such web services is beyond the scope of this module, you could consume data from your own web services in the same way.

2 Connecting to a Webservice

2.1 Module Theme: Building a Weather App



If you think back to OPSC7311, you will remember the Starsucks app example that was used throughout the module. This semester, our theme is a **weather app**.

Have you ever wondered where the weather apps get their data from? All the apps out there have data available for even the most remote locations on Earth. Each country has its own weather service, for example, the South African Weather Service that has their own website: https://www.weathersa.co.za/

But if you created an app, and you wanted world-wide weather data, how would you collect data from all the countries in the world? The answer is that you would not. Instead, find a company that does that already and make use of their services!

One such company is AccuWeather. You might be familiar with the AccuWeather apps that are available for both Android and iOS. But an important part of their business is their Application Programming Interfaces (APIs) that provide data to third parties. They even have services tailored to the needs of the retail and manufacturing industries. (AccuWeather, Inc., 2020)

We will make use of the AccuWeather RESTful APIs in this module.

2.2 What is a RESTful API?

If we want to request data from AccuWeather, we need to know how to communicate with their servers. It is quite easy to do since they expose RESTful APIs.

> "A **RESTful API** is an application program interface (API) that uses HTTP requests to GET, PUT, POST and DELETE data." (Rouse, 2020)

The first thing to note here is that the protocol that is used to exchange information is the HyperText Transfer Protocol (HTTP). This is the same protocol that is used when we access a website using a web browser. This means that a web browser is an HTTP client – it can make HTTP requests and handle the responses. (Rouse, 2019)

How are RESTful APIs accessed? The definition says by means of HTTP requests. So, the same kinds of requests that your browser makes every time you visit a website, will also get made by our app.

When a browser makes a request, the information about what is being requested is sent in the header of the request message. And when the server responds, the response again starts with a header that specifies, amongst other things, the type of content to follow. For example, in Figure 2, the browser requests a Hyper Text Markup Language (HMTL) file from the server. And in the header of the response, the server indicates that the data to follow has a text/html content type. (Seobility, n.d.)

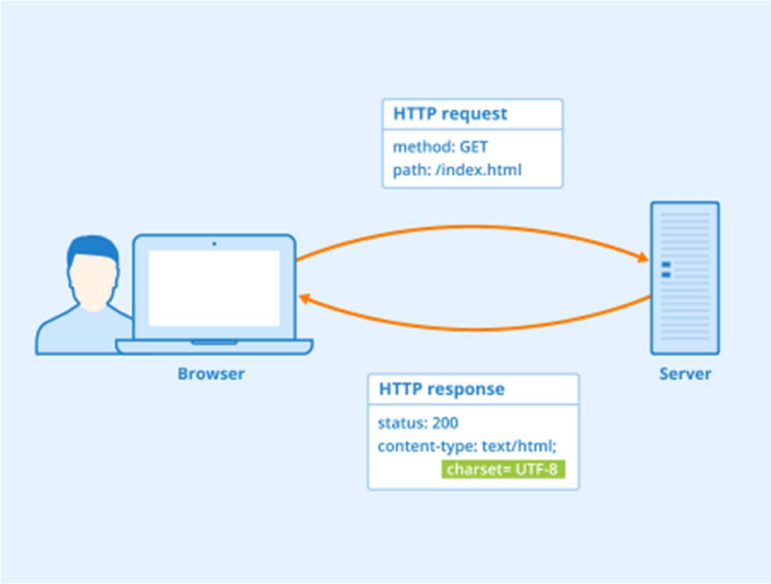


Figure 2. HTTP Headers (Seobility, n.d.)

Many kinds of things can be transported using this HTTP protocol. For example, you might have heard of Simple Object Access Protocol (SOAP) messages before, and those can also be transported using HTTP. SOAP messages are based on Extensible Markup Language (XML), and it is a format for sending data between servers. SOAP is still used in some enterprise software systems where very complex transactions are processed. (Monus, 2020)

RESTful APIs were created later in response to SOAP, to fix some of the issues such as very strict rules that had to be complied with. RESTful APIs can use other messaging formats such as JavaScript Object Notation (JSON). (Monus, 2020)

RESTful APIs are based on the architectural styled called REpresentational State Transfer (REST). (Rouse, 2020) There are a couple of guiding principles that RESTful services must follow, such as the interface being a client-server model and the services being stateless. (RESTfulAPI.net, n.d.)

For the purposes of what we are going to do, we are only reading data from an API. This means that we will make GET requests. We do not have any data to contribute, and there is no reason for us to have access to deleting data from the AccuWeather servers.

Using the developer tools built into the Chrome web browser, we can inspect the headers of the request and response messages when we call the AccuWeather service from our browser. Remember, a browser is an HTTP client, so it can interact with these services if we just browse to the Universal Resource Locator (URL) of the service. Do not worry about the details of the URL yet – we will get to that later.

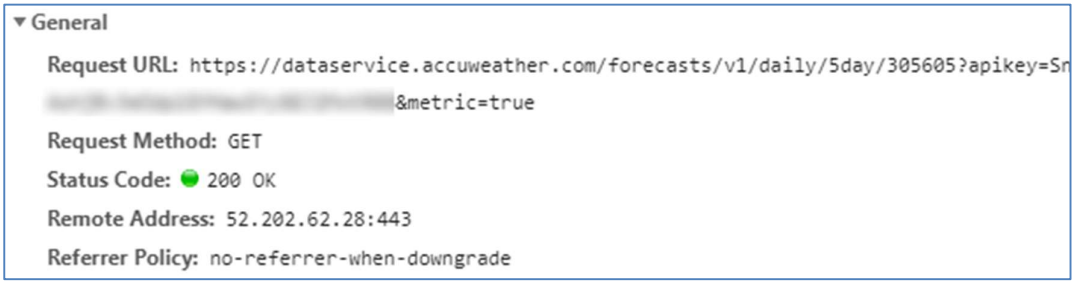


Figure 3. Request Header for an AccuWeather Call

Figure 3 shows the details of the request that was made. And we see that the browser made a GET request on our behalf just as we would expect.

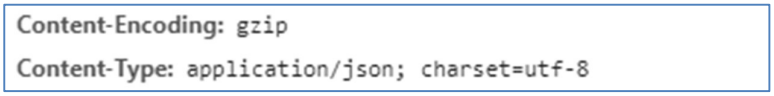


Figure 4. Values from the Response Header

If we inspect the response header, we see a huge number of different fields that are set. For our purposes, only the Content-Encoding and Content-Type fields are of interest right now, as shown in Figure 4. We can see that the content type here is application/json, so we are expecting to see JSON data. More on that later.

The content encoding is set to gzip, which means that the data is compressed. The size of the data that is sent from the server to your device is reduced by running it through an algorithm first. Using GZIP is one of the best practices recommended by AccuWeather. (AccuWeather, Inc., n.d.)

You might have noticed that the protocol used in the URL is https and not http. What is the difference? Hypertext Transfer Protocol Secure (HTTPS) is, as the name suggests, a secure way to send data. Transport Layer Security (TLS) is applied to the connection, which means that the data that is sent from the client to the server is encrypted. (Google, 2020)

If you browse to a site protected by this type of security, the browser will display a lock icon. Figure 5 shows what this looks like in Chrome.

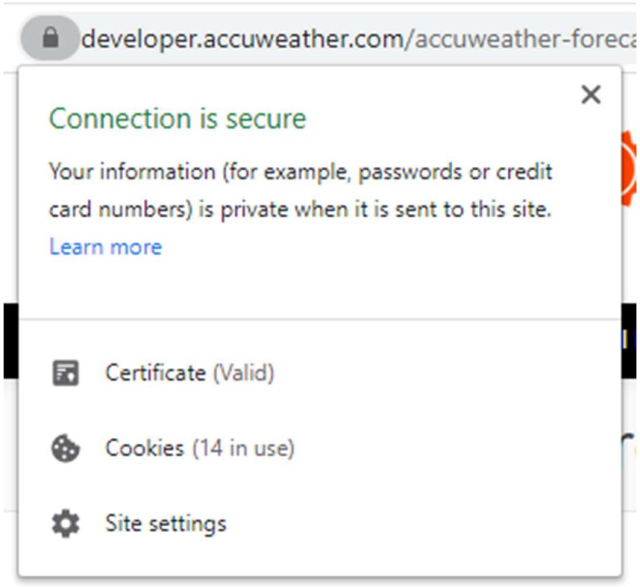


Figure 5. Secure Site Indicator in Chrome

2.3 Introducing the Source Code

There is a GitHub repository that contains all the source code for the examples that are described in this module manual. The repository can be found here:

https://github.com/iie-opsc/opsc7312kotlin

There is a folder for each learning unit, so the folder that is applicable here is LearningUnit1.

Note: All the example code and instructions in this module manual are written in Java. But you are free to use Kotlin if you want to.

You need to create the file apikey.properties to compile and run this app. See steps 1 to 5 on page 19.

2.4 Registering with AccuWeather

AccuWeather is a commercial company that provides their weather data to third parties to make a profit from it. That means that they do not allow everybody to just make unlimited calls to their servers for free. If you look carefully at the URL in Figure 3, you will see that there is an API key that is included in the URL. And that is how AccuWeather knows that the caller is authorised to make the calls.

You need a virtual device with Android Q (API level 29) or later installed to run the example source code.

Luckily, there is a free version that we can sign up for. Figure 6 shows what is included in the free version.

Note that there is a limitation of only 50 calls per day. So, if you make a lot of calls on the same day, you will eventually stop getting data back.

2.4.1 Signing Up for an AccuWeather Account

Browse to https://developer.accuweather.com/packages to see the details of all the packages. Also note at the bottom of the page that there are branding requirements when using this data. This means that each activity where you display weather data should also include the AccuWeather logo.

Click the **Get started now!** link under the **Free** version to sign up.

Once you have created the account, look for the email with the link to complete the sign-up process. Remember to set a strong password!

Features	Free	Standard
Locations	✓	✓
Current Conditions	✓	✓
24 Hours Historical Current Conditions	✓	✓
Daily Forecast	5 Days	5 Days
Hourly Forecast	12 Hours	12 Hours
Indices	5 Days	5 Days
Alarms		5 Days
Translations		✓
Tropical		
Alerts		
Imagery		
	Free 50 calls/day Limit 1 key/developer Free: Get started now!	**$25**/mo $0.12 CPM over 225,000 calls per month

Figure 6. Comparison of the Free and Standard accounts of AccuWeather as on 10 February 2023 (AccuWeather, Inc., 2023a)

Watch the videos in the recommended digital engagement section of this learning unit on page 40. The videos show step-by-step how to build this weather app.

2.4.2 Adding an App to AccuWeather

With a trial account, you only get one API key. So, the next step is to register an app to your account.

To add an app to your AccuWeather account:

1. Click on your email address at the top of the page to access the menu shown in Figure 7.

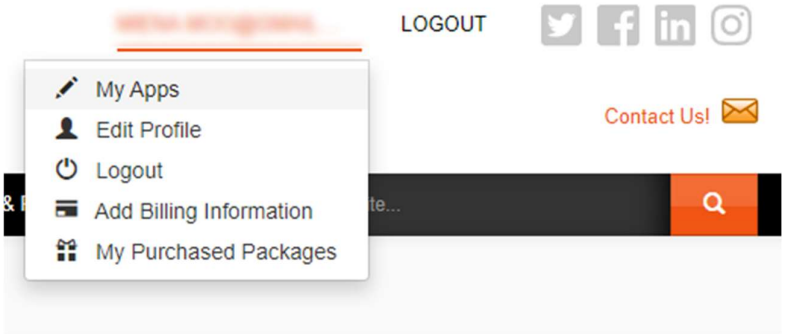


Figure 7. Accessing My Apps

2. Click **My Apps.**
3. Click **Add a new App**.
4. Specify the following values on the **Add App** page:

    - App Name: **Basic Weather**
    - Product: **Core Weather Limited Trial**
    - Where will the app be used? **Mobile App.**
    - What will you be creating with this API? **Weather App.**
    - What programming language is your app written in? **Other.**
    - Is this for Business to Business or Business to Consumer use? **Business to consumer.**
    - Is this Worldwide or Country specific use? **Worldwide.**

5. Click **Create App**.

Once the app has been created, you can click on it to access your very own API key. See Figure 8.

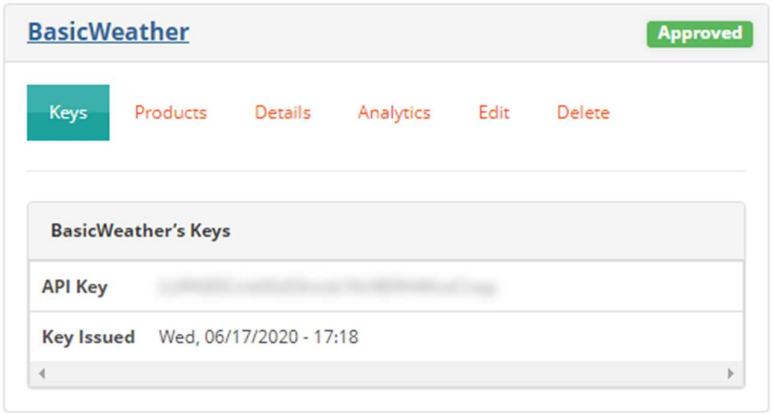


Figure 8. Viewing the API key

Note: No API key is included in the source code in the Git repository – **you need to insert your own API key to run the apps**. Otherwise, the 50 calls per day limit would be reached quite quickly if all the students try to run the app on the same day!

2.5 Requesting Data from AccuWeather

Now that we have signed up for an API key, we are ready to make our first request for data from AccuWeather. We have already seen that if we have the correct URL, we can access the services from a browser. Let us see how to find the URL to retrieve a five-day forecast.

2.5.1 Requesting Data using a Web Browser

On the AccuWeather APIs website, there is a good API reference that explains how to call the services. Start by looking at the API Flow Diagram page (AccuWeather, Inc., 2020c), that contains a diagram explaining what the process is to get information. There are basically two steps in the process:

1. Use the Locations API to find the locationKey for the place that we want to look up.
2. Then use that locationKey to request other data such as a forecast.

There are different ways to find a locationKey. But maybe the easiest one is to search by city name. Figure 9 shows the block in the flow diagram that searching using a city name. What is particularly useful to note here, is now the API key is included in the URL.

Figure 9. Extract from the API Flow Diagram showing location search by City Name (AccuWeather, Inc., 2020c)

Say we want to get the location details for Durban. Here is the URL. Copy it and add your own API key to the end of it:

https://dataservice.accuweather.com/locations/v1/search?q=Durban&apikey=

Figure 10 shows what the response looks like.

Figure 10. Response from the location API

Note that we use https instead of http. If you access the http address, your browser will automatically be redirected to the more secure https URL. But if we did that from our app, the calls from our app would fail. So, let us just use https to start with.

In the next section, we will look at how the JSON syntax works. For now, it is enough to see that it is human readable, and that it is the right city since it indicates that it is in South Africa. Remember, this database is worldwide. So, you might get cities with the same names in other countries too. And we can spot the key value that we were looking for (outlined in green).

The locationKey for Durban is 305605.

Now we can use that piece of information to request the five-day forecast for Durban. Figure 11 shows the information for the call used to get the 5-day forecast for a location.

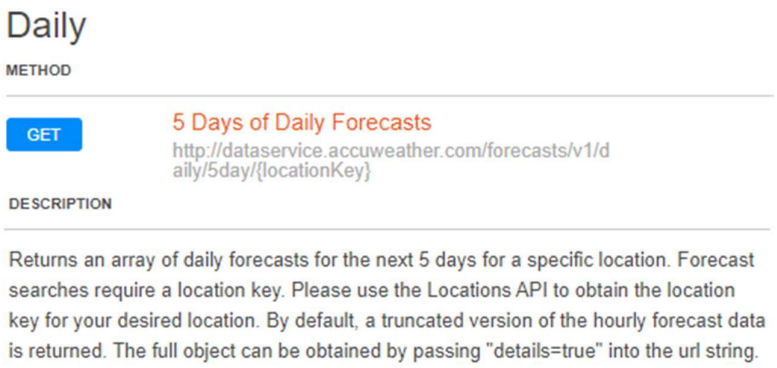


Figure 11. Forecast API (AccuWeather, Inc., 2020b)

Here is the URL then for getting the five-day forecast for Durban. Remember to add your API key at the end of the URL again.

http://dataservice.accuweather.com/forecasts/v1/daily/5day/305605?apikey=

Figure 12 shows an example of what the response looks like.

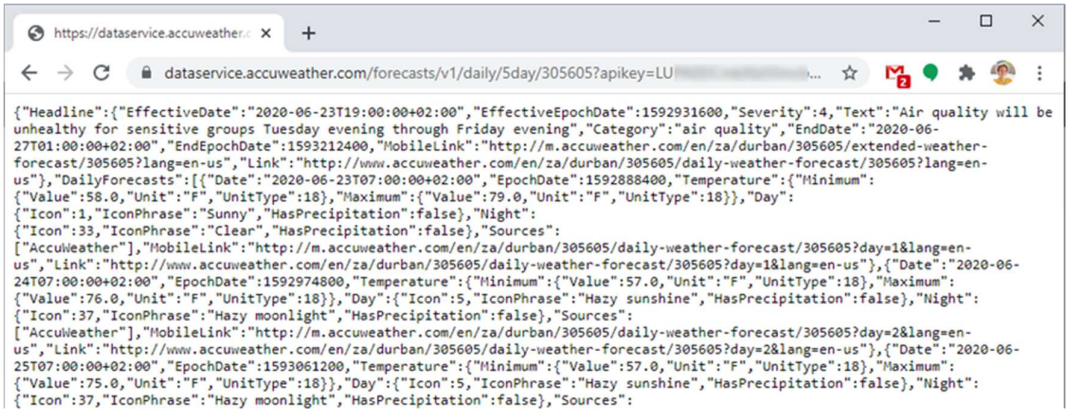


Figure 12. 5-Day Forecast Example

If you see this kind of data, then you know you have the URL ready to start building the app.

2.5.2 Using the Source Code

It is recommended that you follow along with the module manual by building your own app step-by-step as described here. However, the sample source code is available from the GitHub repository, so if you get stuck, have a look at that.

After you have cloned the repository, open in Android Studio the project in the folder LearningUnit1\BasicWeatherApp. Using the command line git tools, you can check out a specific tag, for example: `git checkout LU1-1.6.3`

Another way to get the source code for a specific tag, is to download the zip version of it from the GitHub web interface. Click on Tags, and then download the zip file for the tag.

There are three steps to working with the weather data in our app, as shown in Figure 13.

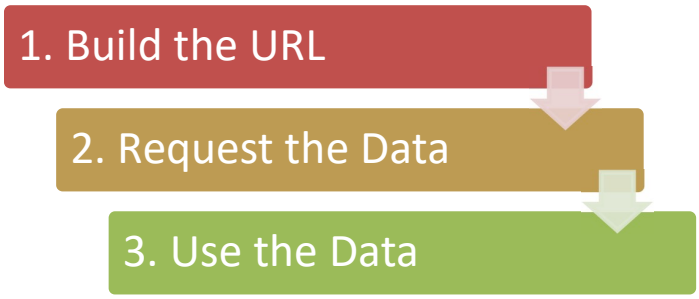


Figure 13. Working with Weather Data in Our App

2.5.3 Building the URL

To get started, create a new Android Studio project, starting with an Empty Activity. Remember to choose your language (Kotlin to follow along here). And use a minimum SDK version of at least API 23: Android 6.0 (Marshmallow).

> **Tip:** If you ever want to **publish** your app on the Google Play Store, the **package** name that you choose when creating the app needs to be unique. The default `com.example` suggested by Android Studio will NOT be allowed by the Play Store. So, you must choose something that is going to be unique.
>
> The Java (and Kotlin) naming convention for packages is to use your organisation's website since that is already guaranteed to be unique. You could use for example:
>
> `com.vegaschool.st21987654.weatherapp`
>
> or
>
> `za.co.varsitycollege.st21987654.weatherapp`
>
> if ST21987654 is your student number. Note how the top-level domain is first in the package name. It starts with com or za – the opposite of the website address.

Let us create a utility class that handles the network requests – call it `NetworkUtil` and put it in the same package as the `MainActivity`.

We want to build up the URL to get the five-day Durban weather forecast. So, let us define some constants in our `NetworkUtil` file that we can use to do that.

```
private val WEATHERBASE_URL =
  "https://dataservice.accuweather.com/forecasts/v1/daily/5day/305605"
private val PARAM_METRIC = "metric"
private val METRIC_VALUE = "true"
private val PARAM_API_KEY = "apikey"
private val LOGGING_TAG = "URLWECREATED"
```

We have the base URL, and some parameters that we want to add. By default, the weather data is returned in imperial units of measurement. Have a close look at Figure 12 – you will notice that the temperatures are in degrees Fahrenheit. So, we want to pass metric=true to get our temperatures in degrees Celsius instead. And we need to pass the API key.

At the end of section 2.5, it was mentioned that there is no API key in the GitHub repository source code. Beyond the logistics of a lot of students trying to use the same key on the same day, it is general good practice to safeguard sensitive information like this. We will store the API key in a build config, that will not be checked into the repository. The process followed here is from (CodePath Android Cliffnotes, n.d.)

To store the API key in a build config:

1. Switch to **Project Files** view.
2. **IMPORTANT!** If you are committing the code to a Git repository, open up the .gitgnore file now and add a line containing *apikey.properties*

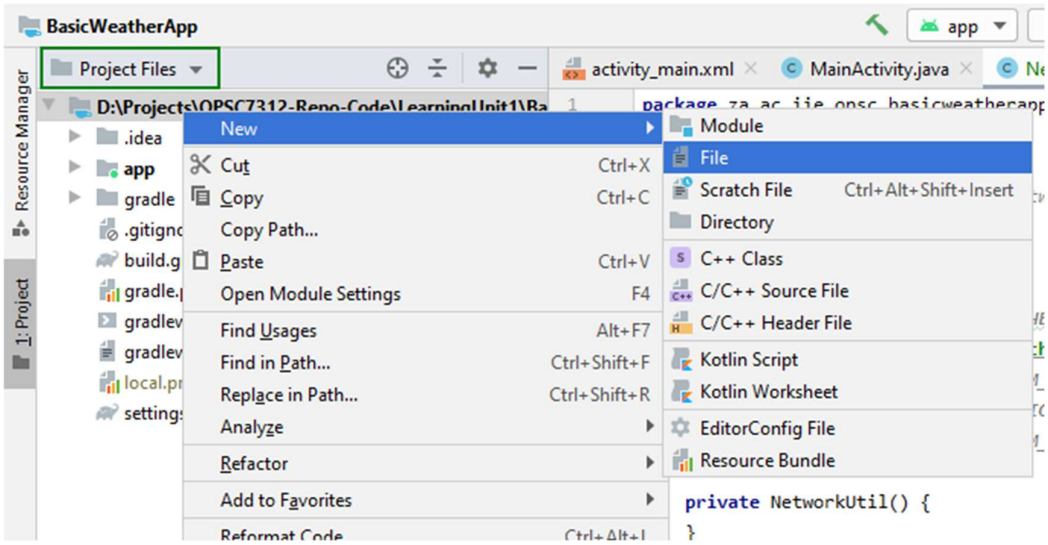


Figure 14. Creating a New File in the Project Root

3. Right-click the root folder of the project and click **New** then **File**.
4. Call the new file *apikey.properties*.

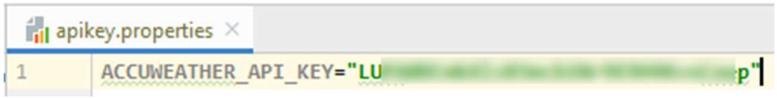


Figure 15. Add the New Property

5. Specify a single property `ACCUWEATHER_API_KEY` that stores your API key. Note that it should be enclosed in double quotes.
6. Open the app/build.gradle file, and add the following lines just below plugins:

```
def apikeyPropertiesFile = rootProject.file("apikey.properties")
def apikeyProperties = new Properties()
apikeyProperties.load(new FileInputStream(apikeyPropertiesFile))
```

7. Inside the default config brackets, add these lines:

```
buildConfigField("String", "ACCUWEATHER_API_KEY",
        apikeyProperties['ACCUWEATHER_API_KEY'])
```

8. The app/build.gradle file should now look like shown in Figure 16. The new parts are highlighted in green.

```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
}

def apikeyPropertiesFile : File = rootProject.file("apikey.properties")
def apikeyProperties = new Properties()
apikeyProperties.load(new FileInputStream(apikeyPropertiesFile))

android {
    namespace 'za.ac.iie.opsc.basicweatherapp'
    compileSdk 33

    defaultConfig {
        applicationId "za.ac.iie.opsc.basicweatherapp"
        minSdk 26
        targetSdk 33
        versionCode 1
        versionName "1.0"

        buildConfigField("String", "ACCUWEATHER_API_KEY",
                apikeyProperties['ACCUWEATHER_API_KEY'])

        testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
    }
```

Figure 16. Completed app/build.gradle File

9. Switch back to **Android** view.
10. On the main menu, click **Build** then choose **Make Project**.

Now we have all the information available that we need to build up the URL that we are going to call. There is a very useful class in the android.net package called Uri, that we can use here. The documentation for this class is available from https://developer.android.com/reference/android/net/Uri

The advantage of using the Uri class to build up the full URL, is that we do not need to know about the implementation details of what the URL must look like. The Uri class will add the ? and & into the URL as necessary.

Add this function to the NetworkUtil file:

```
fun buildURLForWeather(): URL? {
    val buildUri: Uri = Uri.parse(WEATHERBASE_URL).buildUpon()
        .appendQueryParameter(
            PARAM_API_KEY,
            BuildConfig.ACCUWEATHER_API_KEY
        ) // passing in api key
        .appendQueryParameter(
            PARAM_METRIC,
            METRIC_VALUE
        ) // passing in metric as measurement unit
        .build()
    var url: URL? = null
    try {
        url = URL(buildUri.toString())
    } catch (e: MalformedURLException) {
        e.printStackTrace()
    }
    Log.i(LOGGING_TAG, "buildURLForWeather: $url")
    return url
}
```

2.5.4 Requesting Data from the AccuWeather Service

Add a TextView with an ID of `tv_weather` to the main activity of your app, as a temporary place where we can just display the text that we get from the web service.

The call we want to make is going to require access to the Internet. And that is a permission that we must request. So, add the below line to AndroidManifest.xml:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Now we can add the code to the MainActivity class to call the web service, and display the raw data:

```
class MainActivity : AppCompatActivity() {

    lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // Thanks to https://stackoverflow.com/questions/46177133/
        //      http-request-in-android-with-kotlin
        thread {
```

```
            val weather = try {
                buildURLForWeather()?.readText()
            } catch (e: Exception) {
                return@thread
            }
            runOnUiThread { binding.tvWeather.text = weather }
        }
    }
}
```

There are very specific rules about what should happen on which thread in an Android app. All User Interface (UI) updates must happen on the UI thread because the Android UI toolkit is not thread safe. But other tasks, like web service calls that can take long, should not block the UI thread. (Android Open Source Project, 2020b)

By using thread, calling the methods on the right thread becomes easy. Everything in the brackets after thread is called on a separate thread. But the update needs to happen on the UI thread, so we use `runOnUiThread` to do that.

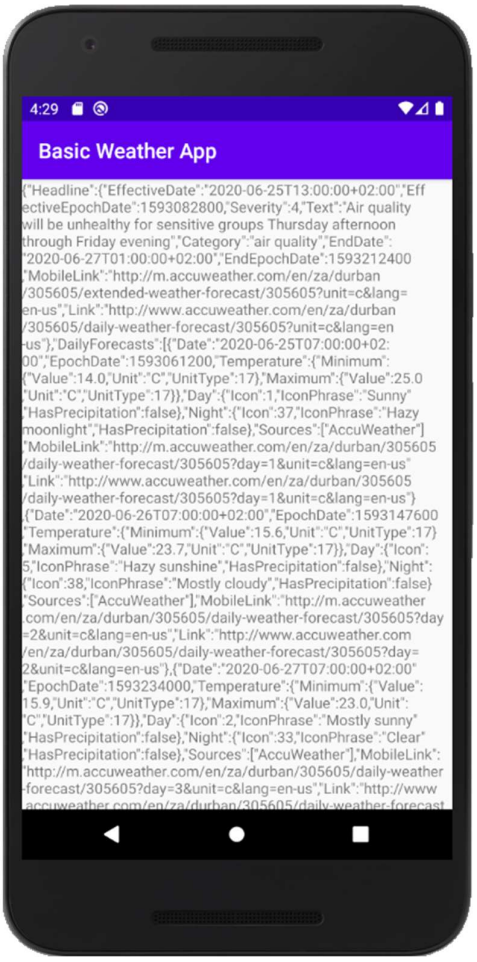


Figure 17. The Most Boring Weather App… Ever…

We are now successfully requesting the data, and just displaying it in the app. But this is a very long way from being user friendly.

Before we jump into how to parse the data and display it in a more user-friendly way, let us have a look at JSON.

3 JSON Files and JSON Objects

3.1 Why JSON?

We have already seen that JSON is not the only option for transferring information between a server and a client. XML is another option that is also human readable. The main benefit of JSON, applicable to all systems that use it, is that it is a much more compact format. If the same data is represented in JSON and XML, the XML version will use as much as double the number of characters. (Freeman, 2019)

JSON does also have some advantages that are more situational. For example, if the data is parsed by a JavaScript web UI, it takes only a single line of code to do so. (Freeman, 2019) And since JavaScript is such a widely used language on the web, this has contributed quite a bit to the widespread use of JSON today.

3.2 Quick Recap of JSON Syntax

In JSON, we can represent objects with properties, as well as arrays. Let us look at an example of a person with their name and age. This example was created in the online editor JSON Editor Online: https://jsoneditoronline.org/

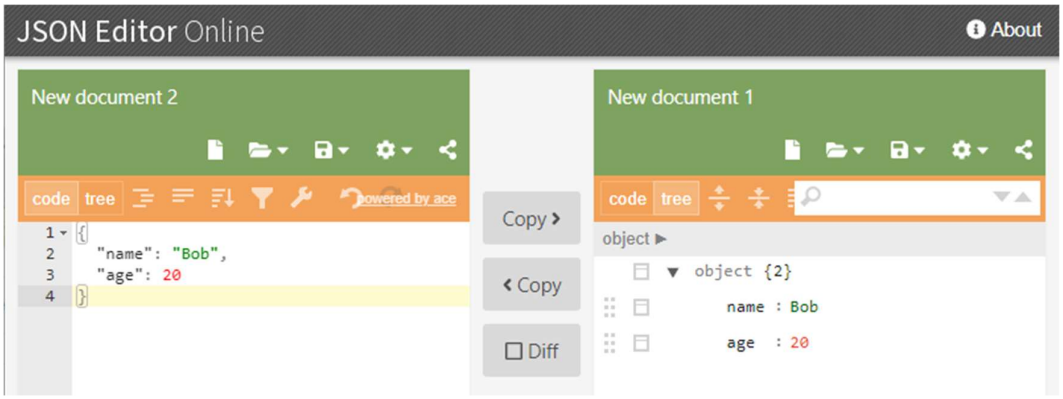


Figure 18. JSON Object

The curly brackets { } indicate the start and end of the object. And the properties are represented as key-value pairs. This is a very simple object, but objects could have many properties.

What if we wanted to represent another object inside of that, maybe containing the information about the degree that Bob is studying? Well, objects can be nested as shown in Figure 19.

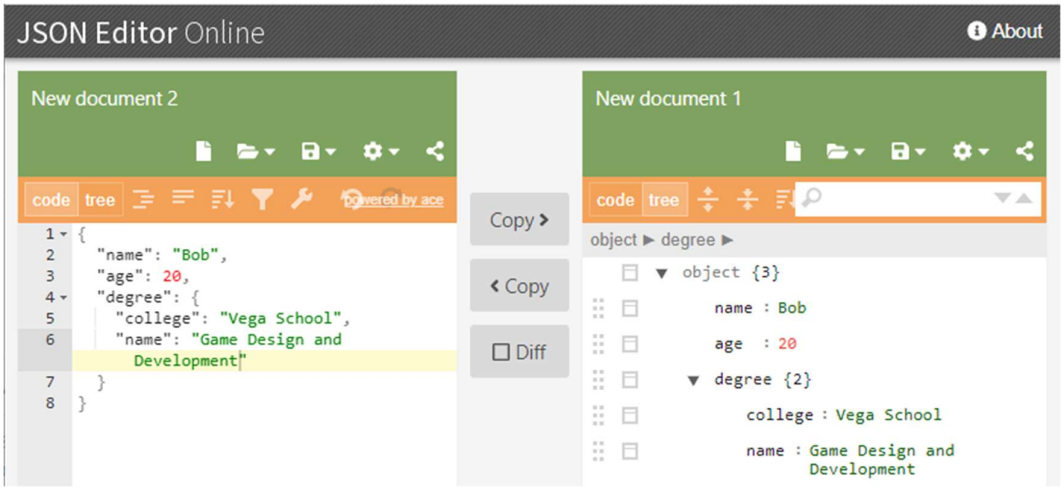


Figure 19. Nested Degree Object

One last thing that we want to be able to represent would be an array of objects. This is done using square brackets [ ], as shown in Figure 20.

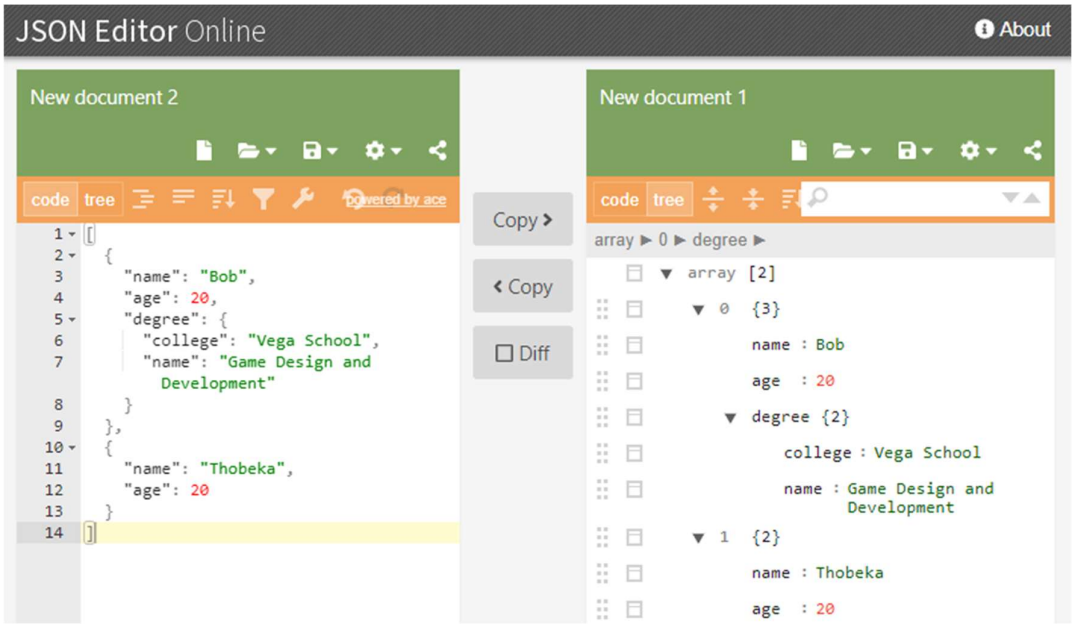


Figure 20. Array of Objects

Looking at this example, we see some of the flexibility of JSON. For the second person, no degree information is available. So, it is simply left out of the data representation.

A very useful feature of this online editor is that you can use it to format JSON data in an easy-to-read way. If we copy the data from the AccuWeather service into this editor, and then click the format button (indicated by the arrow in Figure 21), it will add indentation and newlines. And if you click the **Copy >** button, if will also appear in the tree structure on the right. Go ahead and do this now with your latest data (from a browser), so you can explore the structure of what the service provides.

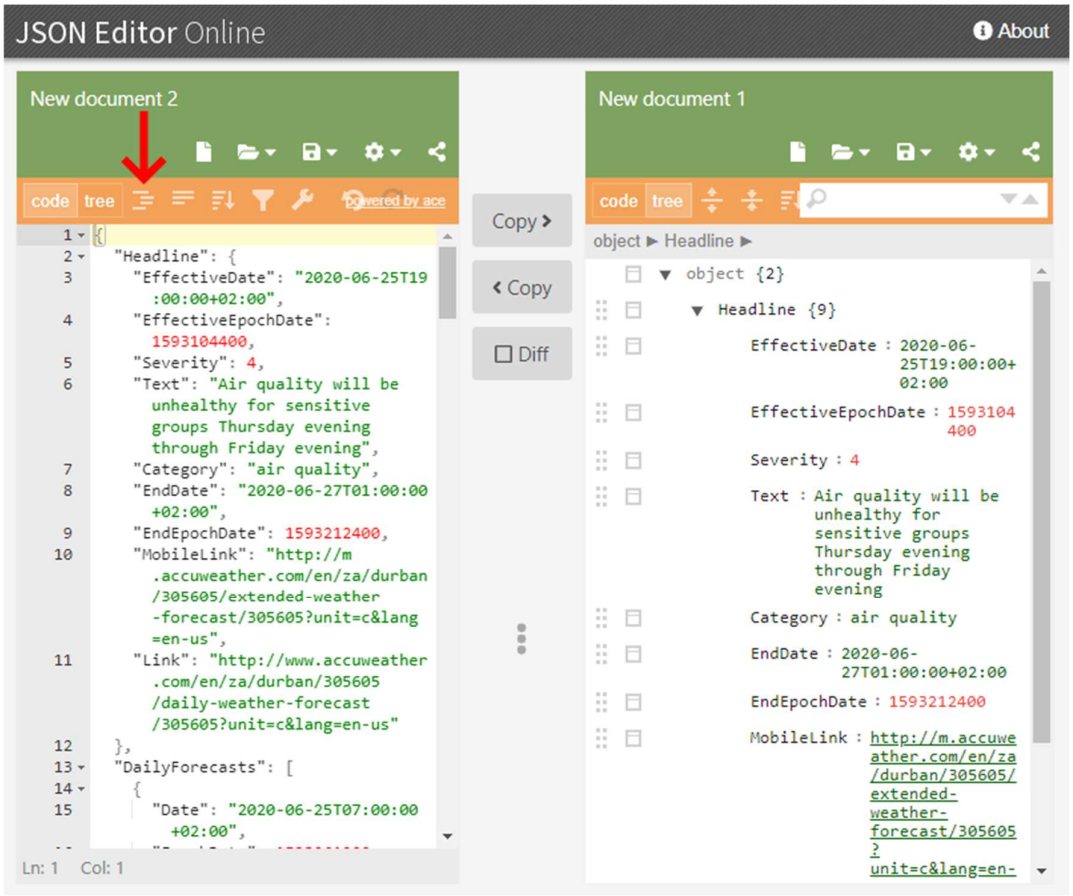


Figure 21. Formatting AccuWeather Data

Read more about JSON in (Tagliaferri, 2016).

3.3 Reading JSON Objects

Before we start parsing the data, let us quickly deal with the branding requirements specified by AccuWeather. At the time of writing of this Module Manual, the official logo did not exist. So, the logo included in the code might not be accurate.

Nevertheless, add an image view to your app that displays the AccuWeather logo, with ID `iv_accuweather`. The logo should be available on this page:

https://developer.accuweather.com/packages

And then add the following code to the `onCreate` method of the activity:

```
// add an event handler to open the Accu Weather
// website on click
binding.ivAccuweather.setOnClickListener {
    val intent = Intent(
        Intent.ACTION_VIEW,
        Uri.parse("http://www.accuweather.com/")
    )
    startActivity(intent)
}
```

Read more about opening a URL in a browser in (RIP Tutorial, n.d.)

As with all things in programming, there are different options for reading JSON data in a Java Android app. The first way of doing this, is by using JSONObject and JSONArray.

When creating a JSONObject, there is a constructor that takes a string as parameter. That is the input string that will be parsed to extract all the data inside it. So, if we have a String variable called `weatherJSON`, we can do:

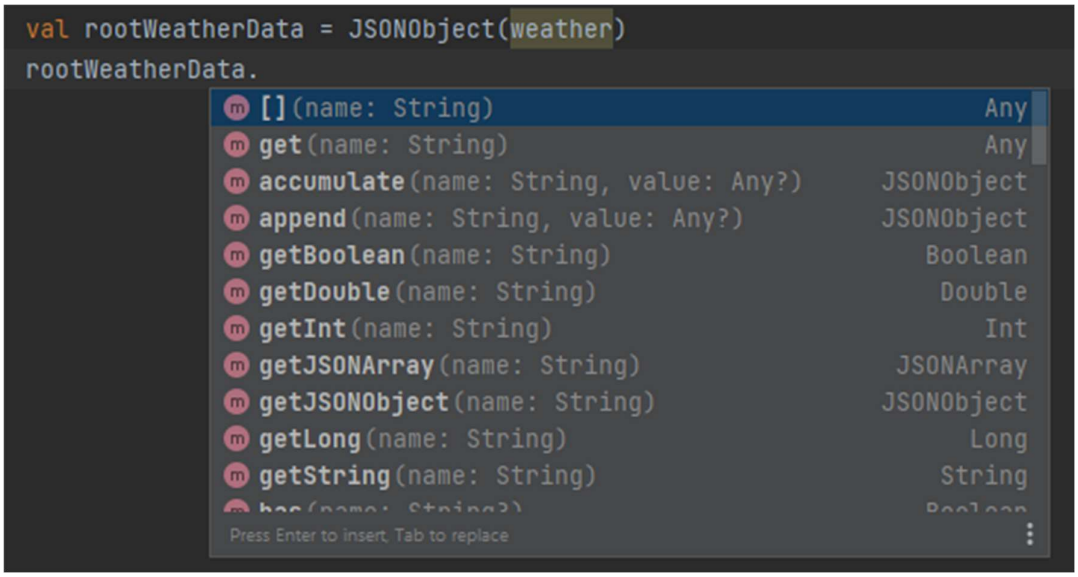


Figure 22. Autocomplete Showing JSONObject Methods

Autocomplete in Android Studio reveals that there are a few different methods that can be called on a `JSONObject` to get data, depending on the data type of the data that we are looking for. For example, to get a single string value, we could call `getString`. For a nested object, there is `getJSONObject`. And if you are reading an array, use `getJSONArray`.

Let us look at the forecast data that we get (see Figure 23).

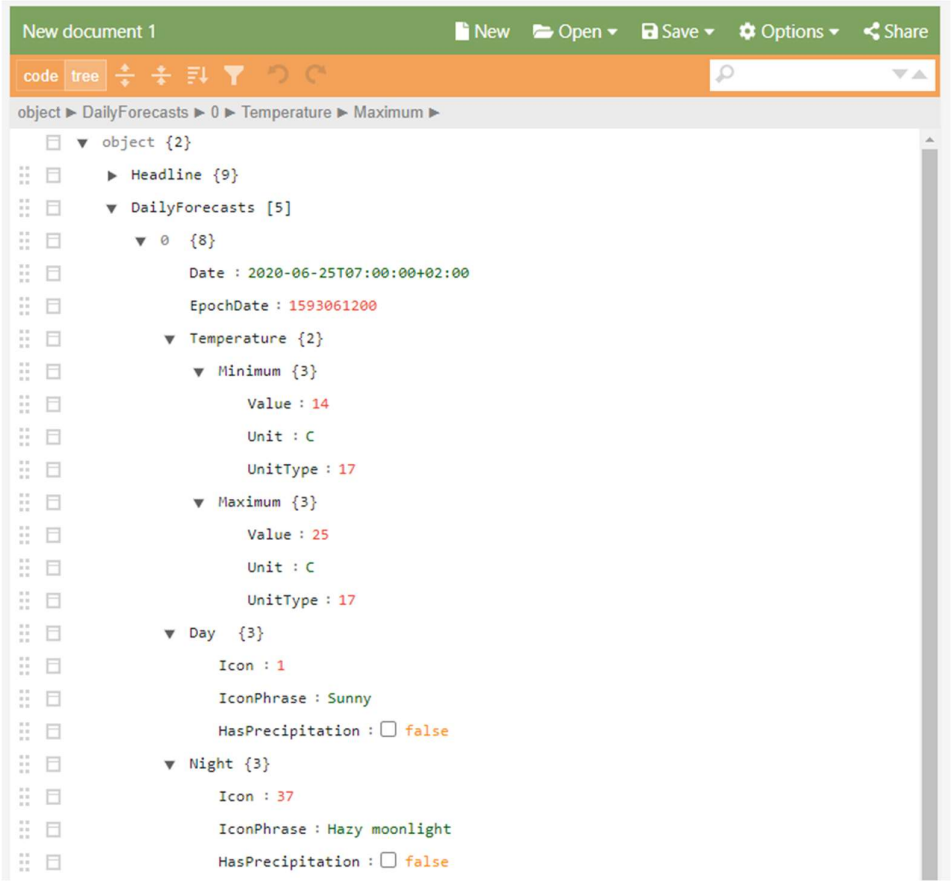


Figure 23. Parsed Daily Forecast Example

For every day, we have the date, minimum temperature, and maximum temperature. There are also indications of precipitation and which icon would be relevant to display. But let us leave that out for now.

Create a new class called `Forecast` to store the data and add fields for the date and temperatures.

```
class Forecast {
    var date: String = ""
    var minimumTemperature: String = ""
    var maximumTemperature: String = ""
}
```

Let us add a function to our MainActivity to read the data from the JSON string:

```
fun consumeJson(weatherJSON: String?) {
    if (fiveDaylList != null) {
        fiveDaylList.clear()
    }

    if (weatherJSON != null) {
        try {
            // get the root JSON object
            val rootWeatherData = JSONObject(weatherJSON)
            // find the daily forecasts array
            val fiveDayForecast =
                    rootWeatherData.getJSONArray("DailyForecasts")

            // get data from each entry in the array
            for (i in 0 until fiveDayForecast.length()) {
                val forecastObject = Forecast()
                val dailyWeather = fiveDayForecast.getJSONObject(i)

                // get date
                val date = dailyWeather.getString("Date")
                Log.i(LOGGING_TAG, "consumeJson: Date$date")
                forecastObject.date = date

                // get minimum temperature
                val temperatureObject =
                        dailyWeather.getJSONObject("Temperature")
                val minTempObject = temperatureObject.getJSONObject("Minimum")
                val minTemp = minTempObject.getString("Value")
                Log.i(LOGGING_TAG, "consumeJson: minTemp$minTemp")
                forecastObject.minimumTemperature = minTemp

                // get maximum temperature
                val maxTempObject = temperatureObject.getJSONObject("Maximum")
                val maxTemp = maxTempObject.getString("Value")
                Log.i(LOGGING_TAG, "consumeJson: maxTemp$maxTemp")
                forecastObject.maximumTemperature = maxTemp
                fiveDaylList.add(forecastObject)
                binding.tvWeather.append(
                    "Date: $date Min: $minTemp Max: $maxTemp\n"
                )
            }
        } catch (e: JSONException) {
            e.printStackTrace()
        }
    }
}
```

This method makes use of two fields that need to be added to the class too:

```
var fiveDaylList =  mutableListOf<Forecast>()
val LOGGING_TAG = "weatherDATA"
```

And then we can call the `consumeJson` method from the thread:

```
runOnUiThread { consumeJson(weather) }
```

Figure 24 shows what the program now looks like. Only the relevant information has been extracted and is now displayed in a somewhat useable format.

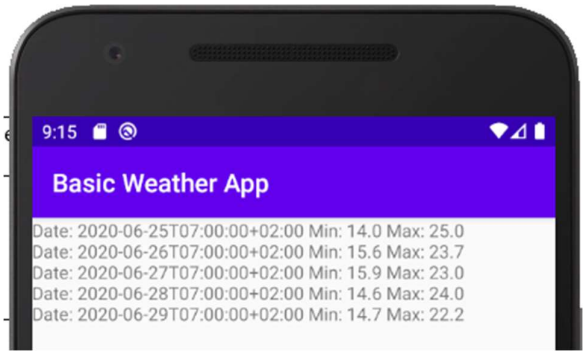


Figure 24. Slightly Better Text Display

This method of reading JSON does work, but it requires a lot of steps that must be taken for every single piece of information. We do end up with an ArrayList of objects that contain the data we need, but the process is a little error prone.

As we said at the beginning of this section, there is always another way. So, let us look at Gson next.

3.4 Using the Gson Library

Gson is a library that can "be used to convert a JSON string to an equivalent Java object." (Gson, 2020) It has functionality that is a lot like Newtonsoft's Json.NET library for C#, if you are familiar with that.

We need to add a dependency on the external module Gson. Open the Module: app build.gradle file – the same one that we added the buildConfigField to before. Add the following dependency:

```
implementation "com.google.code.gson:gson:2.8.6"
```

For us to use Gson, we need to have Kotlin objects that we can use when reading the data. Luckily, there are tools that can make the process easier. Using your favourite browser, go to https://www.json2kt.com/

This is a website that can generate the code for us that we need, from sample data. So, copy the AccuWeather data again, and paste it into the **Copy & Paste Json Here** text area (see Figure 25). Then click **Download Kotlin Files**.

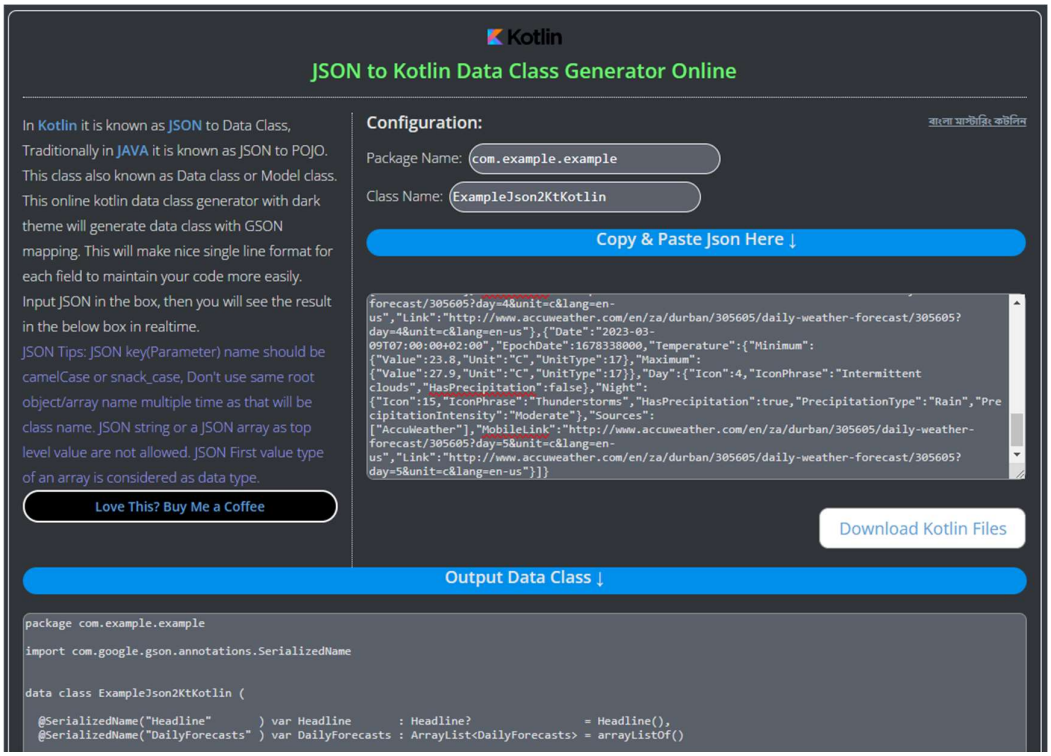


Figure 25. Converting JSON to Kotlin code with https://www.json2kt.com/

You will see that several classes are going to get generated and downloaded as a ZIP file. Create a new package called model and copy the downloaded classes into that folder. Make sure that you fix any missing imports (although there shouldn't be any if the dependency was added), and that you change the package name for each of the files to match your package.

Also, do double check the Minimum and Maximum classes. The value field should be float, not int. The tool that we used may misidentify that, depending on the exact sample data that we give it.

Now we can make use of Gson to convert the string into plain old Java objects (POJOs). This greatly simplifies the `consumeJson` method:

```
fun consumeJson(weatherJSON: String?) {
    if (weatherJSON != null) {
        val gson = Gson()
        val weatherData =
                gson.fromJson<ExampleJson2KtKotlin>(weatherJSON,
                        ExampleJson2KtKotlin::class.java)
        for(forecast in weatherData.DailyForecasts) {
            binding.tvWeather.append("Date: " +
                    forecast.Date?.substring(0, 10) +
                    " Min: " +
                    forecast.Temperature?.Minimum?.Value +
```

```
                        " Max: " +
                        forecast.Temperature?.Maximum?.Value +
                        "\n")
            }
        }
}
```

We no longer need the list of Forecast objects, and once you delete that you can also delete the Forecast class that we do not need anymore.

The only thing that is different in the display of the app is that only the date portion of the date is displayed now, with none of the time information.

But the rest all look the same. (See Figure 26.) So, our replacement of JSONObjects with using Gson was successful. And our code is much more concise now.

The app is still not user friendly though. So, let us improve the user interface next.

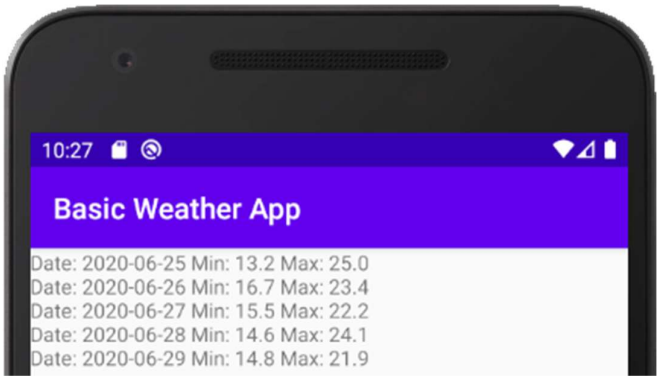


Figure 26. New Screenshot after Gson

4 Fragments

4.1 What is a Fragment?

> "You can think of a **fragment** as a modular section of an activity, which has its own lifecycle, receives its own input events, and which you can add or remove while the activity is running (sort of like a 'sub activity' that you can reuse in different activities)." (Android Open Source Project, 2020c)

Just like we use classes and methods to modularise our apps, we can make use of fragments to modularise our Android activities.

In this section, we are going to split the main activity into two fragments: one for the five-day forecast, and another for the AccuWeather logo. If we added more activities that display different data, we could then re-use the AccuWeather logo fragment on all of those.

4.2 Creating a Blank Fragment

Let us start by creating a fragment for the AccuWeather logo.

To create a new fragment:

1. Right click on the app in the Project view, click **New** then **Fragment** and finally **Fragment (blank)**.
2. Specify a name for the fragment (see Figure 27): **AccuWeatherLogoFragment**
3. Click **Finish**.

If your project is in a Git repository, you will be prompted to add the files to the repository.

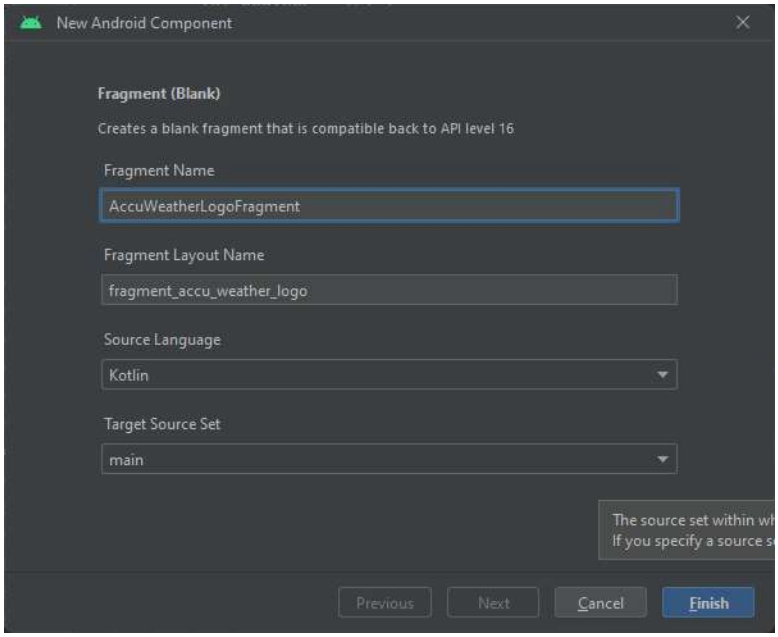


Figure 27. Creating a Fragment

When the fragment is created, two files get opened automatically in Android Studio: a layout file and a Kotlin class. If you look at the code, you will recognise some similarities with activities. For example, there is an `onCreate` method.

Copy the image view containing the AccuWeather logo from the main activity. You will notice that the image view will now have an ID of `iv_accuweather2`, to ensure that it is unique.

Now we need to add the code for the `OnClickListener` that we had on the image view before. This works slightly differently for the fragment. Let us have a look at the generated code in the `onCreateView` method:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    // Inflate the layout for this fragment
    return inflater.inflate(R.layout.fragment_accu_weather_logo,
                            container, false)
}
```

This is the method that is responsible for creating the UI for the fragment. So, if we want to access components, we should do that here. The view that is created by `inflater.inflate` is the view that will contain the components in our layout.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    // Inflate the layout for this fragment
    val view = inflater.inflate(R.layout.fragment_accu_weather_logo,
                                container, false)
    val imageView =
                view.findViewById<ImageView>(R.id.iv_accuweather2)
    // add an event handler to open the AccuWeather website
    imageView.setOnClickListener {
        val intent = Intent(
            Intent.ACTION_VIEW,
            Uri.parse("http://www.accuweather.com/")
        )
        startActivity(intent)
    }
    return view
}
```

Note that the `findViewById` method is now called on the `view` object, not on the fragment itself. And remember to change the id to `iv_accuweather2` if you are copying the code!

For this very simple fragment, we do not need to worry about anything else. So, let us see how to make use of it.

4.3 Using the AccuWeather Logo Fragment

In the layout of the main activity, delete the AccuWeather logo and the code to add the on click listener.

Open the main activity's design. In design view, drag the fragment container view from the palette onto the layout, and choose our new fragment called `AccuWeatherLogoFragment`.

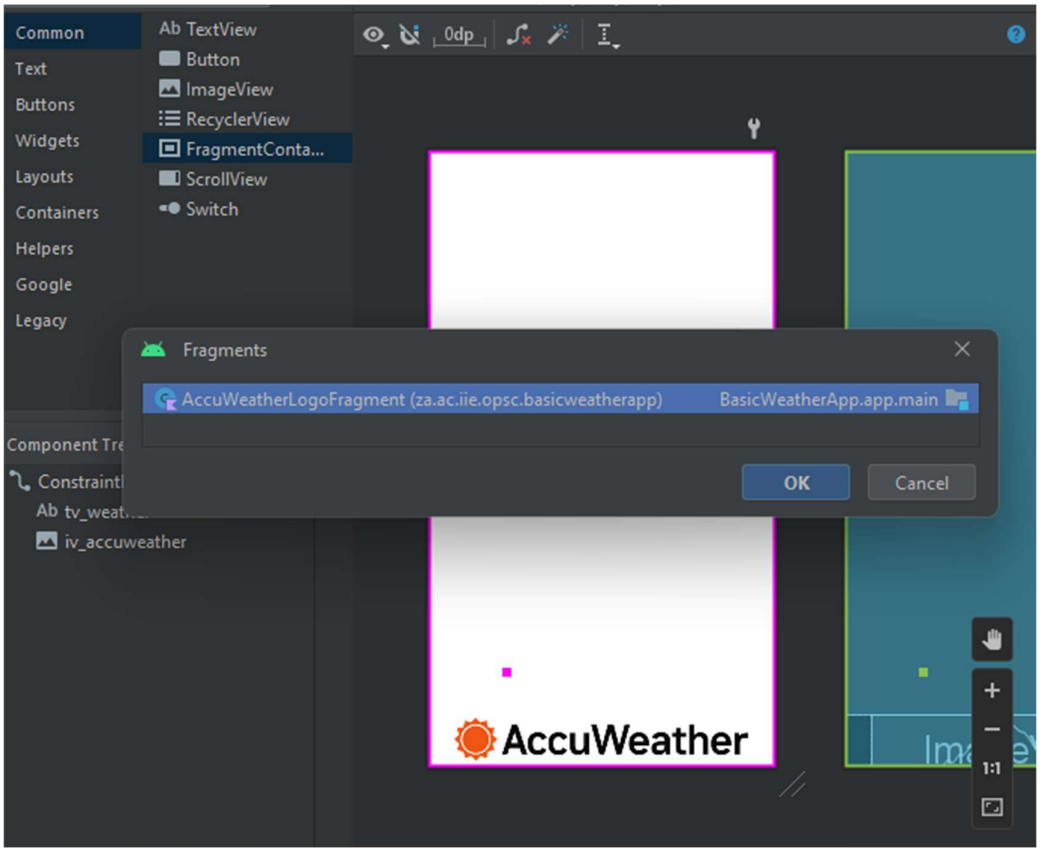


Figure 28. Adding the Fragment

Position the fragment in the same place where the logo was before. Double check that all the constraints still make sense, including those of the text view.

This fragment is very simple and does not require any data from its parent activity. So, we do not need to do anything more to get the image view to work as we want it to.

If we run the app again, it should still look the same as it did the before. But now the AccuWeather logo is a reusable component.

4.4 Using a List Adapter

We have not made any improvements to the user interface of the app yet. So, how can we improve the display? Let us make a fragment for the five-day weather, that displays a list of items.

Create a new fragment, but this time choose **Fragment (List)**.

In the New Android Component dialog, specify DailyForecasts as the object kind, and DailyForecastsRecyclerViewAdapter as the adapter class name (see Figure 29).

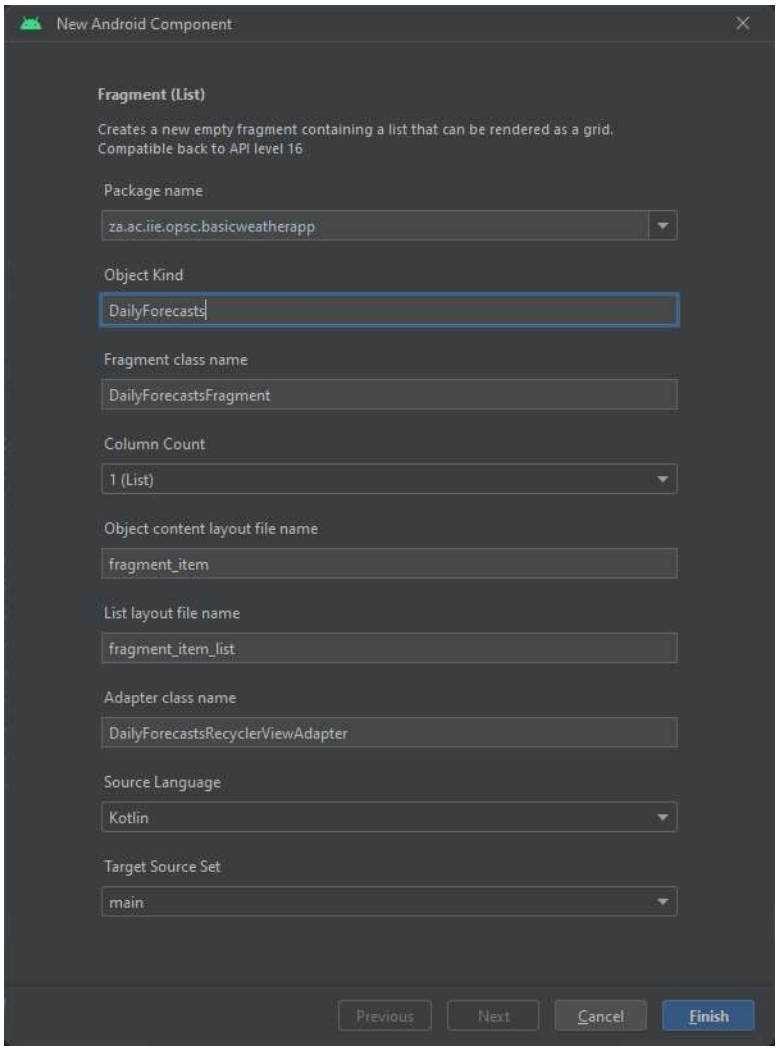


Figure 29. Creating the List Fragment

Several things get created for us during this process:

- `fragment_item_list.xml` – layout for the whole list of items that we are going to display.
- `fragment_item.xml` – layout for a single item in the list.
- `DailyForecastsFragment` – the main class of the fragment.
- `DailyForecastsRecyclerViewAdapter` – an adapter that is used when displaying a list item.
- `dummy\DummyContent` – a dummy model that we will be replacing completely.

If we look at the data that we are parsing with Gson, the entries that we are looking for are of class `DailyForecasts`. So, each item in the list will be representing a single `DailyForecasts` object.

4.4.1 Creating the layout for an item

Open the layout `fragment_item.xml`. This is where we are going to put the components to display the data. We will have available to display the date, the minimum temperature, and the maximum temperature.

Draw a layout that looks like the one in Figure 30.

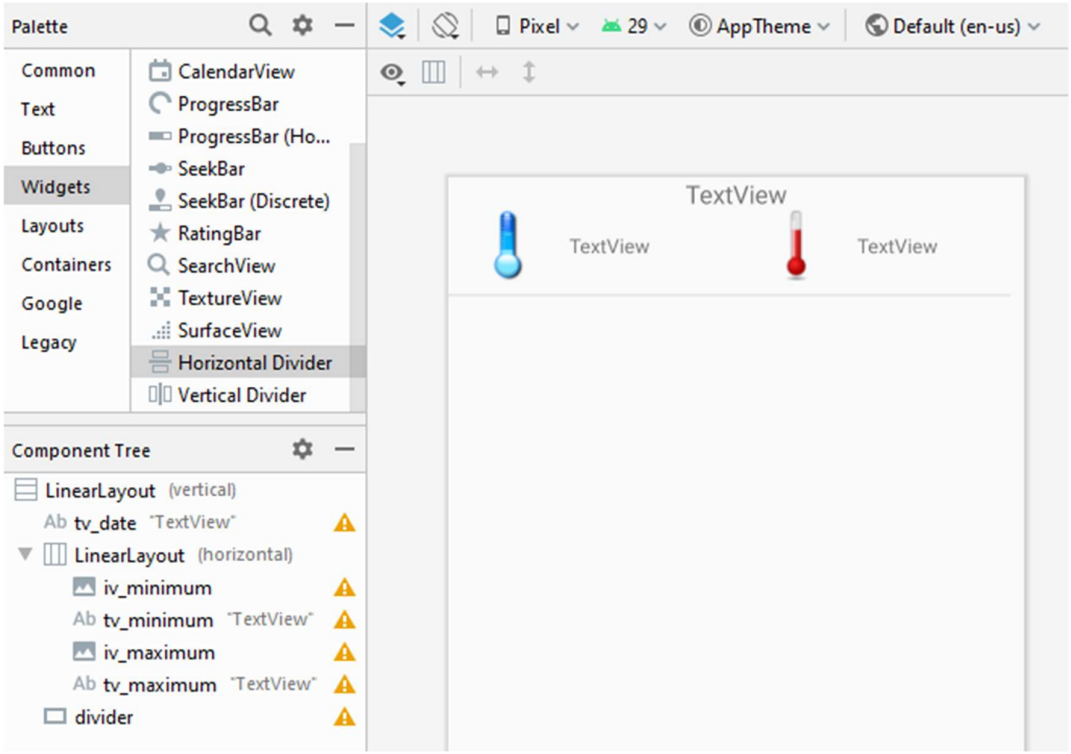


Figure 30. List Item Layout

4.4.2 Updating the View Adapter

Now that we have deleted the components that were in the list item layout, the `DailyForecastsRecyclerViewAdapter` will not compile. But that is okay, since it is the next thing that needs to be changed anyway.

Change all references to `PlaceholderItem` to `DailyForecasts`. Once all of those have been changed, also delete the import for `PlaceholderItem`.

This generated class has an inner class that manages a single item. You will see that it has a couple of generated TextView fields that are now no longer sensible. Replace those with fields for our three fields.

```
inner class ViewHolder(binding: FragmentItemBinding) :
                        RecyclerView.ViewHolder(binding.root) {
    val date: TextView = binding.tvDate
    val minimum: TextView = binding.tvMinimum
    val maximum: TextView = binding.tvMaximum

    override fun toString(): String {
        return super.toString() + " '" + date.text + "'"
    }
}
```

The last thing that remains to be changed in this class, is the onBindViewHolder method, that sets the values onto the view:

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val item = values[position]
    holder.date.text = item.Date?.substring(0, 10)
    holder.minimum.text = item.Temperature?.Minimum?.Value.toString()
    holder.maximum.text = item.Temperature?.Maximum?.Value.toString()
}
```

4.4.3 Updating the Fragment

The one thing that remains to be changed is the `DailyForecastsFragment`. You will notice that the line where the adapter is set no longer compiles. That is the line that sets the collection of data objects onto the view. We want to get out data from the AccuWeather web service, so copy the code to read that from the main activity.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    val view = inflater.inflate(R.layout.fragment_item_list,
                                container, false)

    // Set the adapter
    if (view is RecyclerView) {
        with(view) {
            layoutManager = when {
                columnCount <= 1 -> LinearLayoutManager(context)
                else -> GridLayoutManager(context, columnCount)
            }
            thread {
                val weatherJSON = try {
                    buildURLForWeather()?.readText()
                } catch (e: Exception) {
                    return@thread
                }
                if (weatherJSON != null) {
```

```
                    val gson = Gson()
                    val weatherData =
                          gson.fromJson<ExampleJson2KtKotlin>(
                                weatherJSON,
                                ExampleJson2KtKotlin::class.java)
                    activity?.runOnUiThread {
                        adapter = DailyForecastsRecyclerViewAdapter(
                                    weatherData.DailyForecasts)
                    }
                }
            }
        }
    }
    return view
}
```

One last bit of clean-up remains. Delete the `DummyContent` class and `dummy` package.

Build the project now. Everything should compile at this point. If there are still any imports using `DummyContent`, remove those and then the build should succeed.

4.4.4 Updating the Main Activity

So far, so good. But the fragment is not used by the app yet. Open the main activity's layout and replace the text view with the `DailyForecastsFragment.`

In the code for the main activity, delete the `tvWeather` field and the line in the `onCreate` method that assigns it a value. Also delete the async task and the call to it. The main activity should now look like this:

```
class MainActivity : AppCompatActivity() {

    lateinit var binding: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)
    }
}
```

If we run the app, the output now looks like this:

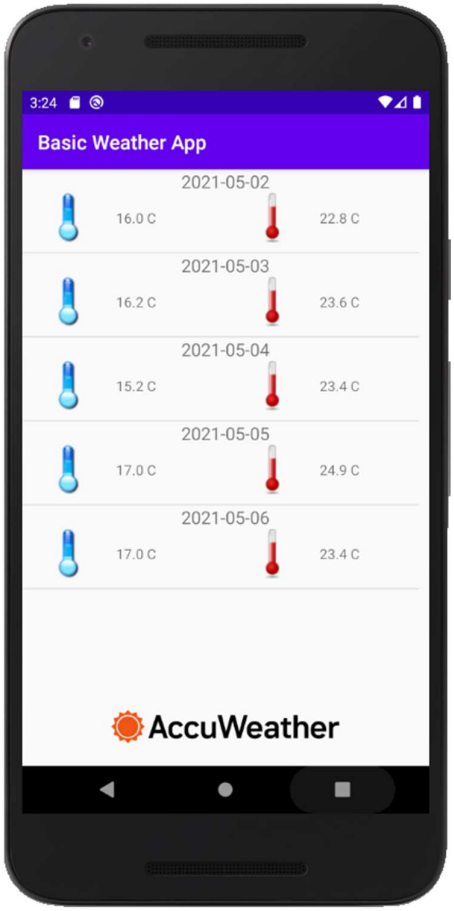


Figure 31. App with the List Display

Fragments can communicate with each other, and there can be call backs to the activity that the fragment is used in. For more information about fragments, read the first item on the recommended additional reading list below.

5 Recommended Additional Reading

Android Open Source Project, 2020c. *Fragments.* [Online] Available at: https://developer.android.com/guide/components/fragments [Accessed 31 July 2023].

RIP Tutorial, n.d. *Android: Open a URL in a browser.* [Online] Available at: https://riptutorial.com/android/example/549/open-a-url-in-a-browser [Accessed 31 July 2023].

Tagliaferri, L., 2016. *An Introduction to JSON.* [Online] Available at: https://www.digitalocean.com/community/tutorials/an-introduction-to-json [Accessed 31 July 2023].

6 Recommended Digital Engagement

Work through this code lab:

Get data from the internet
https://developer.android.com/codelabs/basic-android-kotlin-training-getting-data-internet#0

7 Activities

Complete the activities on Learn.

Learning Unit 2: External Libraries	
Learning Objectives: • Determine which external libraries to use. • Use an external library to solve a programming problem. • Connect to a geolocation service. • Display information from a geolocation service. • Connect to a social media service. • Use a social media service. • Determine which SDKs are available to connect to an app to. • Connect an app to an appropriate SDK.	**My notes**
Material used for this learning unit: • GitHub repository: LearningUnit2	
How to prepare for this learning unit: • Make sure that you have the GitHub source code available and that your Android Studio is up to date.	

1 Introduction

By the end of Learning Unit 1, we had built a working weather app. But it is not a very exciting one since it only displays the weather for Durban for the next five days. What if we live in Johannesburg, or Cape Town? And why can we not search for the weather of another city? In short, the app still needs some features that users have come to expect from weather apps.

In this learning unit, we will create a new app so we can start with a tabbed view. We will make use of the relevant template when we create the app. Once we have the tabbed user interface in place, we can copy the fragments from the previous learning unit's app. We will then be ready to make use of a different library to read the data from the AccuWeather web services.

Later in the learning unit, we will add geolocation services and sharing to social media.

2 Connecting to External Libraries

2.1 What is a Library?

> "A **software library** refers to a collection of files, programs, routines, scripts, or functions that can be referenced in the programming code." (Computer Hope, 2017)

A library is code that is written by somebody else, that you can make use of in your apps. Libraries can be free and open source, but there are also commercial libraries.

In the Android eco system, there are too many libraries to name. But a few interesting ones include CalendarView, Bubble Navigation and SmoothBottomBar. Read (Bialas, 2020) for a list of 30 awesome libraries.

We have made use of a library already, without much fanfare – Gson. Remember the dependency that we added to the app's build.gradle file? That is a dependency on an external library. Gradle is responsible for finding online and downloading this open-source library, so it can be included in the build.

2.2 Creating the Tabbed User Interface

Create a new project, this time using the **Tabbed Activity** project template. The wizard creates an activity that uses the CoordinatorLayout to switch between different tabs (see Figure 32).

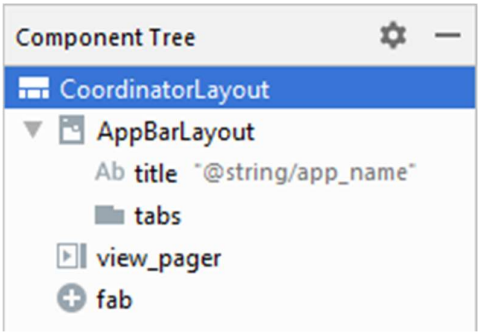


Figure 32. Components for the Tabbed UI

Change the background of the `CoordinatorLayout` to an image and make the `AppBarLayout` and `tabs` transparent. If you run the app, it should look like the screenshot shown in Figure 33.

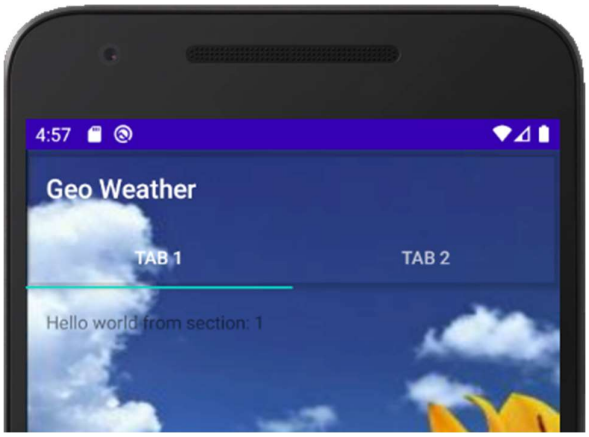


Figure 33. Tabs with Background

We can reuse some of the code that we created in Learning Unit 1, so copy across everything except the main activity. Remember to also copy any related resources, and to add the API key to the project, and to update the package names where applicable.

At this point, your code should look like the code for the tag **LU2-CopiedFiles** in the repository.

Remember to add a permission to access the Internet to the manifest file.

Include the `AccuWeatherLogoFragment` in the `AppBarLayout` between the title and the tabs.

This app is going to have three tabs: TODAY, 5-DAY FORECAST and CITY WEATHER. We already have an implementation for the five-day forecast that we can include. The generated class `SectionsPagerAdapter` (in the package ui.main) is where we need to go to manage the fragments that are displayed.

In the strings.xml file, make sure that you have the following three entries:

```
<string name="tab_text_1">TODAY</string>
<string name="tab_text_2">5-DAY FORECAST</string>
<string name="tab_text_3">CITY WEATHER</string>
```

The first two should already be there, just with default values. The third one you will need to add. Then we have everything ready to update the `SectionsPagerAdapter`:

1. Making use of the string resources, add all three tab titles to the `TAB_TITLES` array.
2. Update the implementation of getCount to return 3.
3. Add a switch statement to `getItem` to create the forecasts fragment.

For now, we only have an actual fragment for tab position 1 – the `DailyForecastsFragment`. So, use the generated `PlaceholderFragment` for both the other positions for now.

```
class SectionsPagerAdapter(private val context: Context, fm:
FragmentManager) :
    FragmentPagerAdapter(fm) {

    override fun getItem(position: Int): Fragment {
        // getItem is called to instantiate the fragment for the given page.
        when (position) {
            1 -> return DailyForecastsFragment()
        }
        // Return a PlaceholderFragment.
        return PlaceholderFragment.newInstance(position + 1)
    }

    override fun getPageTitle(position: Int): CharSequence? {
        return context.resources.getString(TAB_TITLES[position])
    }

    override fun getCount(): Int {
        // Show 3 total pages.
        return 3
    }
}
```

The superclass `FragmentPagerAdapter` keeps all the fragments in memory. This means that switching between them will be fast.

Running the app reveals that the text is not readable on the background image. So, make the background of the list semi-transparent and change the font to be more visible. The app now looks like Figure 34.

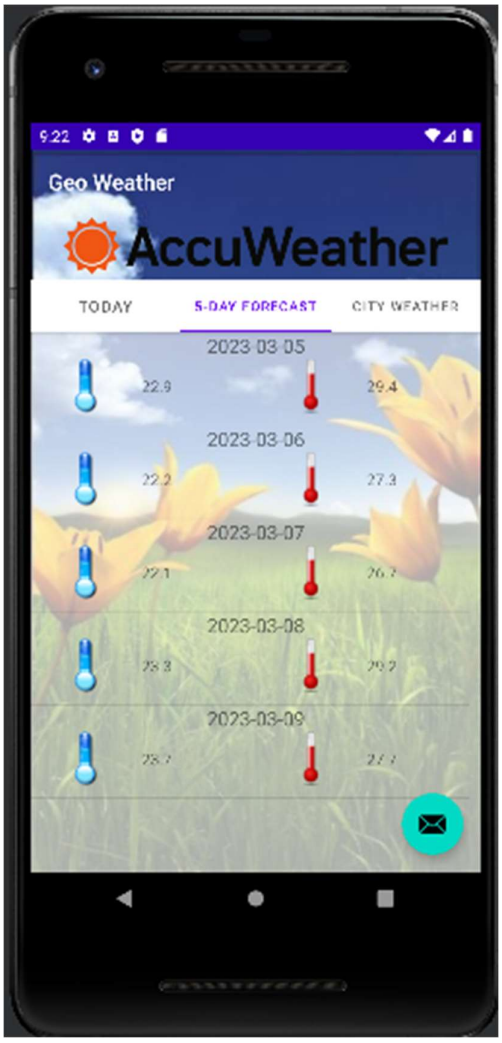


Figure 34. 5-Day Forecast Tab Included

Now we want to make use of libraries to improve the way in which we read the data.

2.3 Retrofit

We will be making use of a library when communicating with an HTTP API: Retrofit. Let us see how the creators of this library describes it.

> "**Retrofit** turns your HTTP API into a Java interface." (Square, Inc., n.d.)

We use Retrofit to implement an interface, that makes calling the AccuWeather APIs easier.

We are going to call multiple services eventually, for example to get the current weather.

So, in anticipation of that, rename the `ExampleJson2KtKotlin` class in the model package to `FiveDayForecast`. Right-click the file, choose **Refactor** and then **Rename**. That will automatically rename usages of the class too.

Include the following dependencies in the app module's build.gradle file:

```
    implementation "com.squareup.retrofit2:retrofit:2.9.0"
    implementation "com.squareup.retrofit2:converter-gson:2.9.0"
```

Remember to build the project to ensure that Gradle fetches these libraries.

2.3.1 Using Retrofit

Let us start using Retrofit. The documentation explaining how to use it, is available at https://square.github.io/retrofit/

If you read that page, you will see that we need to define an interface, that describes how to access the service that we are going to call. Create a package called retrofit, and inside that create an interface called `IAccuWeather`.

How do we know what to create in that interface? Well, we look at the API that we want to call as shown in Figure 35.

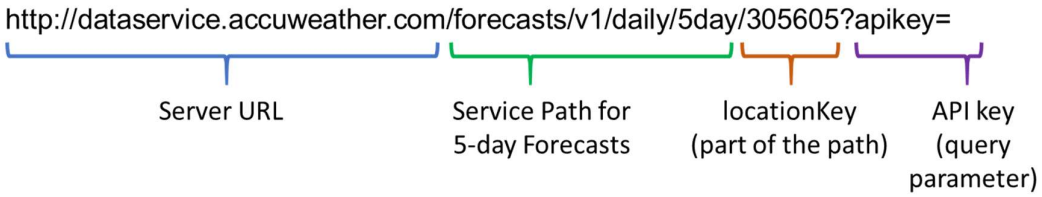


Figure 35. Parts of the AccuWeather URL

The example on the Retrofit website shows that the base URL is passed to the Retrofit class. And the interface then specifies everything that goes beyond that, using annotations.

Let us look at the annotations for the five-day forecast service as another example.

```
package za.ac.iie.opsc.geoweather.retrofit

import retrofit2.http.GET
import retrofit2.http.Path
import retrofit2.http.Query
import za.ac.iie.opsc.geoweather.model.FiveDayForecast
```

```
interface IAccuWeather {

    /**
     * Get the five-day forecast for a specific location key
     * @param locationKey The key for the location
     * @param apiKey The api key to use
     * @param metric Whether to get the data in metric units of
     *               measurement
     * @return The five-day forecast
     */
    @GET("forecasts/v1/daily/5day/{locationKey}")
    suspend fun getFiveDayForecast(
        @Path("locationKey") locationKey: String?,
        @Query("apikey") apiKey: String?,
        @Query("metric") metric: Boolean
    ): FiveDayForecast?

}
```

The @GET annotation specifies how to build up the URL for the specific service that we want to call. The `forecasts/v1/daily/5day/` part will always be the same, regardless of which location's weather we are getting. But the `{locationKey}` will change, so that is indicated using {}.

Where does the value for `{locationKey}` come from? From the method parameter with the corresponding annotation. Since it forms part of the path, the annotation is `@Path("locationKey")`.

The API key, as well as the parameter indicating whether we want the answer in metric units of measurement, are both query parameters. And that is why the annotation is then `@Query("apikey")`. The string that is provided here must be exactly what the service expects for the key of the query parameter.

The return type of the method specifies what Retrofit should expect from the service. Here we are using our newly renamed `FiveDayForecast` class.

So, the method header shown above has all the information that Retrofit needs at runtime to call the service.

Side Note: Annotations in Java and Kotlin can have different retention policies. These can be:

- `SOURCE`: processed when the program is compiled and then discarded.
- `CLASS`: also processed at compile time and stored in the class files but not available at runtime.
- `RUNTIME`: available for use at runtime. (java2s.com, n.d.)

Java annotations can be used in Kotlin. (Kotlin Foundation, 2023)

If we look at the definition of the annotations used by Retrofit (defined in Java), we see that these are defined as `RUNTIME`. For example, here is the definition of the `@Path` annotation:

```
@Documented
@Retention(RUNTIME)
@Target(PARAMETER)
public @interface Path {
  String value();
```

This means that Retrofit inspects the interface at runtime using reflection. Based on what is found on the methods, the correct code is then called in the background by Retrofit.

Now that we have specified everything that is needed to find the service, let us see how to call it. Create a new singleton class called `RetrofitClient`, that creates and configures the client instance and service.

```
package za.ac.iie.opsc.geoweather.retrofit

import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory

object RetrofitClient {
    var retrofit: Retrofit? = null
        get() {
            if (field == null) field = Retrofit.Builder()
                .baseUrl("https://dataservice.accuweather.com/")
                .addConverterFactory(GsonConverterFactory.create())
                .build()
            return field
        }

    var weatherService: IAccuWeather? = null
        get() {
            if (field == null) field =
                    retrofit?.create(IAccuWeather::class.java)
            return field
        }
}
```

Here we see that we specify the baseUrl, which we know already from the URLs that we called earlier. Then, we specify something that may sound familiar: the converter factory uses Gson. This tells Retrofit that we are expecting to receive the answer in JSON format, and that it should use Gson to parse it into the right kind of object.

2.3.2 Using RxJava

Now we are ready to make the call to the AccuWeather servers. To make the asynchronous service call, we create a class inheriting from `ViewModel` called `DailyForecastsViewModel`.

```
package za.ac.iie.opsc.geoweather

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch
import za.ac.iie.opsc.geoweather.model.DailyForecasts
import za.ac.iie.opsc.geoweather.retrofit.RetrofitClient

class DailyForecastsViewModel : ViewModel() {

    private var _fiveDayForecast =
                    MutableLiveData<List<DailyForecasts>>()
    var fiveDayForecast: LiveData<List<DailyForecasts>> =
                    _fiveDayForecast

    fun getFiveDayForecast(locationKey: String) {
        viewModelScope.launch {
            val weatherData = RetrofitClient.weatherService?.
                    getFiveDayForecast(locationKey,
                        BuildConfig.ACCUWEATHER_API_KEY, false)
            _fiveDayForecast.value = weatherData?.DailyForecasts
        }
    }
}
```

The `ViewModel` has a coroutine called `viewModelScope` that can be used to make the potentially long call to the webservice.

The call that we want to replace is in the `DailyForecastsFragment` class, so let us go there. The `onCreateView` method becomes much simpler now.

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    val view = inflater.inflate(R.layout.fragment_item_list,
                                container, false)

    // Set the adapter
    if (view is RecyclerView) {
        with(view) {
            layoutManager = when {
                columnCount <= 1 -> LinearLayoutManager(context)
                else -> GridLayoutManager(context, columnCount)
            }

            // call the webservice
            viewModel.getFiveDayForecast("305605")
```

```
            // observe the list in the model for changes
            val weatherObserver = Observer<List<DailyForecasts>> {
                newWeather ->
                adapter =
                   DailyForecastsRecyclerViewAdapter(newWeather)
            }
            viewModel.fiveDayForecast.observe(viewLifecycleOwner,
                                              weatherObserver)
        }
    }
    return view
}
```

We call the method `getFiveDayForecast` on the model to read the data, with the location key for Durban for now. And then we observe the list in the model to know when the values change.

Now the call is made using the libraries!

2.4 Getting Today's Weather

Now we want to get the current conditions for Durban, to display on the TODAY tab. Let us look at the AccuWeather API documentation.

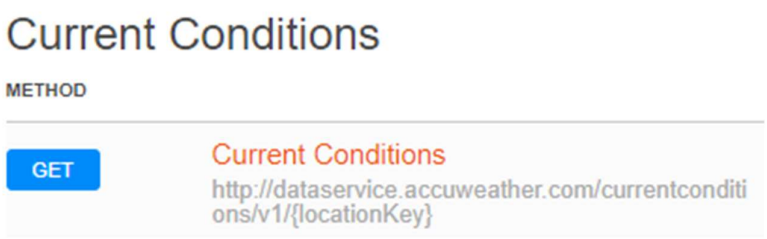


Figure 36. Current Conditions URL from (AccuWeather, Inc., 2020d)

Let us see what the data looks like when it is formatted (see Figure 37).

Here are some quick tips to do the implementation. You have all the information already about how to do these steps in general, but there are a few things you need to know about.

The data that we get back is an array containing one entry. When we make the call, we need to remember that.

Make a call to the URL, copy the JSON response and generate more entities using the code generator we used before: https://www.json2kt.com/

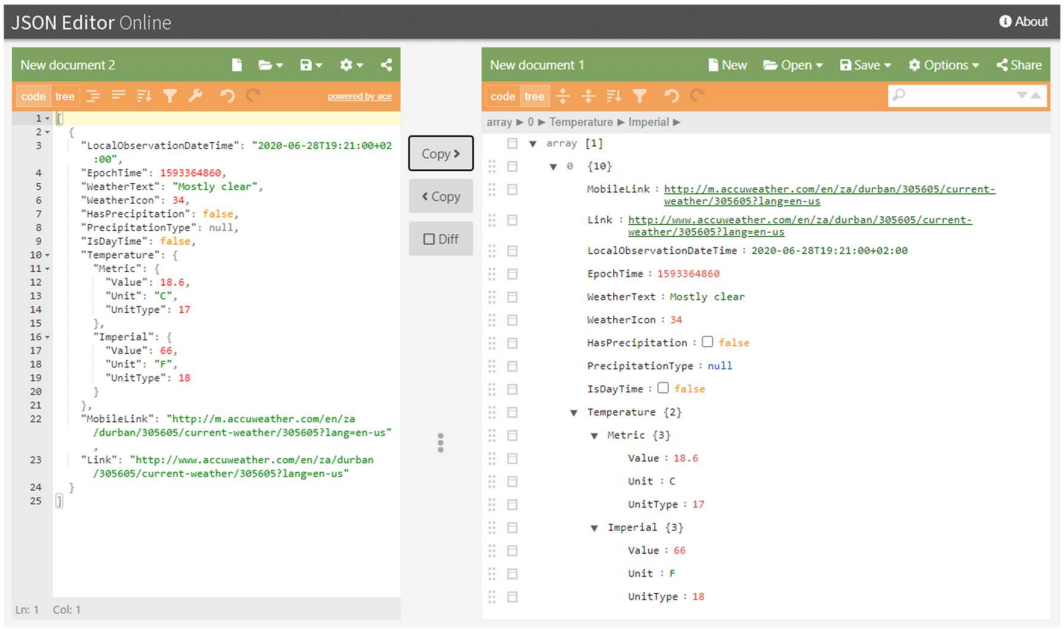


Figure 37. Current Conditions Example Data

In the model package, create a new package called currentweather. Create the classes in the new package. If you see a class called `ExampleJson2KtKotlin` (zero), make that `CurrentWeather` instead. And ignore the `Root` class for this one.

Then add another method to the `IAccuWeather` interface:

```
/**
 * Get the current conditions at a location.
 * @param locationKey The key for the location
 * @param apiKey The api key to use
 * @return The current conditions at the location
 */
@GET("currentconditions/v1/{locationKey}")
suspend fun getCurrentConditions(
    @Path("locationKey") locationKey: String?,
    @Query("apikey") apiKey: String?
): List<CurrentWeather?>?
```

Note that the method will return a `List` of `CurrentWeather` objects.

Add another fragment called `CurrentWeatherFragment` and display the weather text and current temperature.

Figure 38. Displaying the Current Weather

3 Using Geolocation Services in an App

Now this is all working very well, but we are still stuck with getting Durban's weather data. What if we wanted to get the weather for wherever the user is located? We could ask the user to configure that when the app first starts up. But that is not a great user experience.

So, we need to ask the phone where the user is at! And we can do that using the class `FusedLocationProviderClient`. (Droid By Me, 2018)

There are a few steps in the process for make this work, as shown in Figure 39.

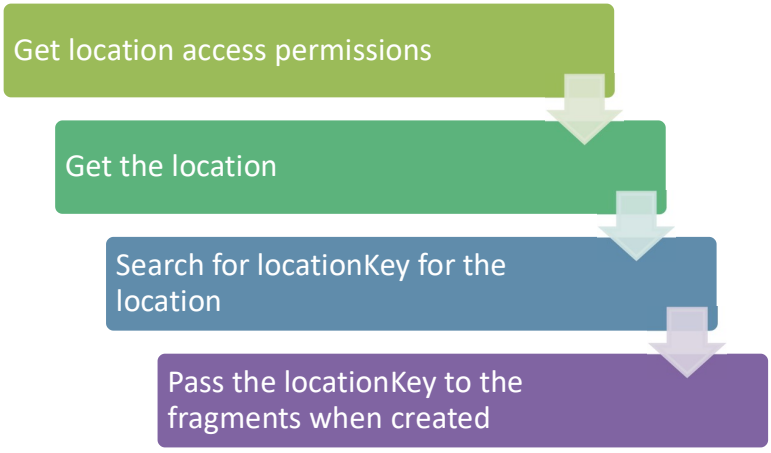


Figure 39. Steps to Working with the User's Location

3.1 Get Permissions

Add the following dependency to the app module's build.gradle file, and remember to build the project after that.

```
implementation 'com.google.android.gms:play-services-location:17.0.0'
```

Add these two permissions to the AndroidManifest.xml file:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION"/>
```

We need to get the location just once – it will be the same for all the tabs. So, let us add the code to the `MainActivity` to request the location.

Add the following field to `MainActivity`:

```
private var fusedLocationProviderClient:
                        FusedLocationProviderClient? = null
```

In the `onCreate` method, get the fused location provider and store that in the `fusedLocationProviderClient` field. And then call a method we will define in a second to request the location.

We will only set up the fragments after we have done everything that is required. So, that will get moved to another method, leaving us with an `onCreate` method that looks like the one below.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    fusedLocationProviderClient =
        LocationServices.getFusedLocationProviderClient(this)
    checkPermissionsAndRequestLocation();

    val fab: FloatingActionButton = binding.fab

    fab.setOnClickListener { view ->
        Snackbar.make(view, "Replace with your own action",
            Snackbar.LENGTH_LONG)
            .setAction("Action", null).show()
    }
}
```

Accessing the user's location is a permission that has security implications. So, this is one of the permissions where one must explicitly get the user to agree for Android 6 and above.

If we have either course or fine location permissions already, then just continue to ask for the location. Otherwise, request the permissions and wait for the call back before proceeding.

```
private fun checkPermissionsAndRequestLocation() {
    val hasFineLocationPermission =
        ActivityCompat.checkSelfPermission(
        this, "android.permission.ACCESS_FINE_LOCATION"
    )
    val hasCourseLocationPermission =
        ActivityCompat.checkSelfPermission(
        this, "android.permission.ACCESS_COARSE_LOCATION"
    )
    if (hasFineLocationPermission != PackageManager.PERMISSION_GRANTED
        && hasCourseLocationPermission !=
           PackageManager.PERMISSION_GRANTED
    ) {
        val permissions = arrayOf<String>(
            "android.permission.ACCESS_FINE_LOCATION",
            "android.permission.ACCESS_COARSE_LOCATION"
        )
        // Request permission - this is asynchronous
        ActivityCompat.requestPermissions(this, permissions, 0)
    } else {
        // We have permission, so now ask for the location
        getLocationAndCreateUI()
    }
}
```

The answer whether the user allowed the location access will be communicated back to us via the `onRequestPermissionsResult` method. Let us look at how we handle that.

```
override fun onRequestPermissionsResult(
    requestCode: Int,
    permissions: Array<String?>,
    grantResults: IntArray
) {
    super.onRequestPermissionsResult(
        requestCode, permissions,
        grantResults
    )
    // is this for our request?
    if (requestCode == 0) {
        if (grantResults.size > 0 &&
            (grantResults[0] ==
                    PackageManager.PERMISSION_GRANTED ||
                    grantResults[1] ==
                    PackageManager.PERMISSION_GRANTED)
        ) {
            getLocationAndCreateUI()
        } else {
```

```
            Toast.makeText(
                this@MainActivity,
                "Location permission denied",
                Toast.LENGTH_SHORT
            ).show()
        }
    }
}
```

3.2 Requesting the Location

To request the location, we need to create a request as well as a callback. So, there is yet another asynchronous process happening here. In our `getLocationAndCreateUI` method, we just make the call and then we wait for the callback to get called.

```
@SuppressLint("MissingPermission")
private fun getLocationAndCreateUI() {
    val locationRequest: LocationRequest = buildLocationRequest()
    val locationCallback: LocationCallback = buildLocationCallBack()
    fusedLocationProviderClient!!.requestLocationUpdates(
        locationRequest,
        locationCallback, Looper.myLooper()
    )
}
```

When creating the request, we can set a couple of different fields, which will change the behaviour of the fused location provider.

```
private fun buildLocationRequest(): LocationRequest {
    val locationRequest = LocationRequest()

locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY)
    locationRequest.setInterval(5000)
    locationRequest.setFastestInterval(3000)
    locationRequest.setSmallestDisplacement(10.0f)
    return locationRequest
}
```

The other parameter is the callback, which will be called with the location when the process is complete. Let us just create a basic implementation for now, so we can see what the logged result is.

```
private fun buildLocationCallBack(): LocationCallback {
    return object : LocationCallback() {
        override fun onLocationResult(locationResult: LocationResult)
{
            super.onLocationResult(locationResult)
            val location: Location = locationResult.lastLocation
            Log.i("LocationResult", "onLocationResult: $location")

            // TODO: Get the locationKey from AccuWeater
            val sectionsPagerAdapter = SectionsPagerAdapter(
                this@MainActivity,
```

```
                supportFragmentManager
            )
            val viewPager: ViewPager = binding.viewPager
            viewPager.adapter = sectionsPagerAdapter
            val tabs: TabLayout = binding.tabs
            tabs.setupWithViewPager(viewPager)
        }
    }
}
```

```
2020-06-28 21:39:26.212 18340-18340/za.ac.iie.opsc.geoweather
I/LocationResult: onLocationResult: Location[fused 37.421998,
-122.084000 hAcc=20 et=+12h21m55s628ms alt=5.0 vel=0.0 bear=90.0
vAcc=40 sAcc=??? bAcc=??? {Bundle[mParcelledData.dataSize=52]}]
```

We are successfully reading the location set by the emulator – Mountain View in California by default. You can change the location reported by the emulator in its settings.

3.3 Getting the Location Key from AccuWeather

Remember in Learning Unit 1, section 2.5.1 (page 15) when we requested the `locationKey` for Durban using a web browser? Well, now we need to make a similar request from our app but based on the longitude and latitude of the user this time. Let us look again at an extract from the API flow diagram to see what the call should look like.

Figure 40. Geo-Position Search (AccuWeather, Inc., 2020c)

If we wanted to get the location information for the position that was logged before, the URL would be as follows. Call it with your own API key, so you can generate the classes for the call.

http://dataservice.accuweather.com/locations/v1/cities/geoposition/search?q=37.421998,-122.084000&apikey=

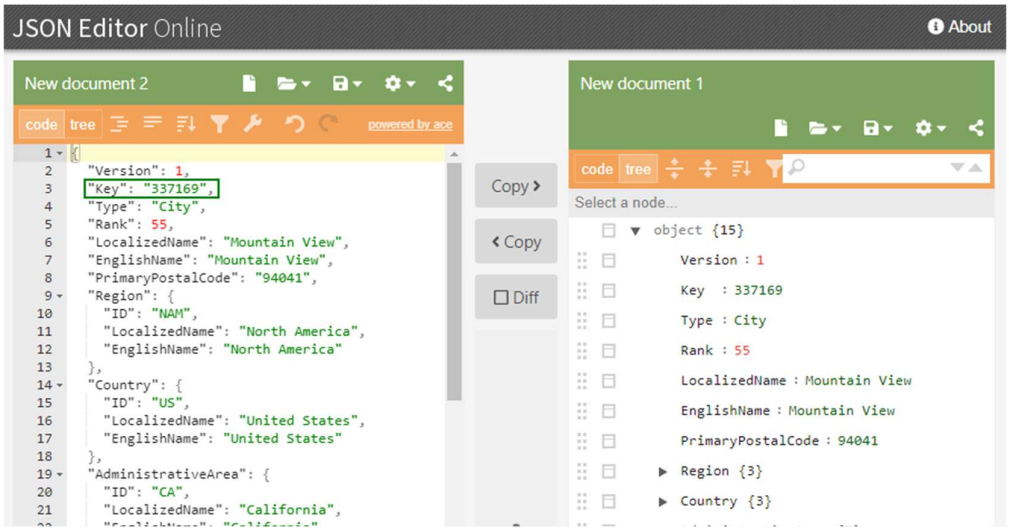


Figure 41. Location Data for Mountain View

Looking at the data in a parsed format, this looks good right. We were expecting Mountain View, and we got Mountain view. So, the call worked.

Using https://www.json2kt.com/ generate entities again, and this time create those in the package model.location. Call the root class `AccuWeatherLocation` this time.

Add another method to the IAccuWeather interface:

```
/**
 * Gets the location data based on the geoposition.
 * @param geoposition The geoposition as latitude,longitude
 * @param apiKey The api key to use
 * @return The location data for the geoposition
 */
@GET("locations/v1/cities/geoposition/search")
suspend fun getLocationByPosition(
    @Query("q") geoposition: String?,
    @Query("apikey") apiKey: String?
): AccuWeatherLocation?
```

Now we can change the location call back to call the AccuWeather Service.

```
private fun buildLocationCallBack(): LocationCallback {
    return object : LocationCallback() {
        override fun onLocationResult(locationResult: LocationResult)
{
            super.onLocationResult(locationResult)
            val location: Location = locationResult.lastLocation
            Log.i("LocationResult", "onLocationResult: $location")
            model.getLocation("${location.latitude}," +
                    "${location.longitude}")

            // observe the list in the model for changes
```

```
            val weatherObserver = Observer<AccuWeatherLocation> {
                location-> displayData(location)
            }
            model.location.observe(this@MainActivity,
                                   weatherObserver)
        }
    }
}
```

3.4 Passing the Location Key to Fragments

The last step that remains is to pass the location key and the location name to the fragments when they get created.

The `SectionsPagerAdapter` is responsible for creating the fragments, so let us pass it the information into its constructor.

```
/**
 * A [FragmentPagerAdapter] that returns a fragment corresponding to
 * one of the sections/tabs/pages.
 */
class SectionsPagerAdapter(private val context: Context,
                           fm: FragmentManager,
                           private val locationName: String,
                           private val locationKey: String) :
```

Then we can do the last part of the work in the `MainActivity` – now, finally, displaying the data.

```
fun displayData(location: AccuWeatherLocation) {
    val sectionsPagerAdapter = SectionsPagerAdapter(
        this@MainActivity,
        supportFragmentManager,
        location.LocalizedName.toString(),
        location.Key.toString()
    )
    val viewPager: ViewPager = binding.viewPager
    viewPager.adapter = sectionsPagerAdapter
    val tabs: TabLayout = binding.tabs
    tabs.setupWithViewPager(viewPager)
}
```

3.4.1 Today's Weather

The `SectionPagerAdapter` does not do anything with the two values yet though. There is a bit of a trick to this. We cannot just pass the two parameters to the constructors of the fragment since the fragment has to have a no-parameters constructor. But the code generated in the `CurrentWeatherFragment` (which was a blank fragment) has a clue.

The `newInstance` method in the companion object takes two parameters, which it adds to a bundle and passes to the fragment that it created. Let us make use of those parameters.

Rename `param1` everywhere to `locationName`, and param2 to `locationKey`.

If you renamed everything correctly, then the onCreate method should now look like this:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    arguments?.let {
        locationName = it.getString(ARG_PARAM1)
        locationKey = it.getString(ARG_PARAM2)
    }
}
```

Now you can use the `locationKey` field's value (add `.toString()`) to get the data instead of the hard coded 305605.

And add a new text view to display the name too.

Lastly, in `SectionPagerAdapter` call newInstance instead of the constructor, and pass the two values:

```
when (position) {
    0 -> return CurrentWeatherFragment.newInstance(locationName,
                                                   locationKey)
```

If we run the app again, we see the weather for Mountain View right now.

Figure 42. Today's Weather Using Geo Position

3.4.2 5-Day Forecast

Do a similar implementation for the `DailyForecastsFragment` too. If you need help with the implementation, have a look in the GitHub repository how it was done there.

3.5 Searching for a City's Weather

The last tab that we have not implemented yet, is the City Weather one. On this tab, we want to allow the user to enter the name of a city, and then display the current weather for that city.

Start by creating a new fragment and adding it to `SectionPagerAdapter`.

For the user interface, we want to use the Material SearchBar library. "This beautiful and easy to use library will help to add Lollipop Material Design SearchView in your project." (mancj, 2020)

To get started with that, include the following dependency in the app module's build.gradle file:

```
implementation 'com.github.mancj:MaterialSearchBar:0.8.5'
```

This dependency will not work yet though – we need to add a Maven repository to the project's settings.gradle file. The new line is highlighted in green below.

```
dependencyResolutionManagement {
    repositoriesMode.set(RepositoriesMode.FAIL_ON_PROJECT_REPOS)
    repositories {
        google()
        mavenCentral()
        maven{url 'https://jitpack.io'}
    }
}
```

Switch to the code view in the layout of the city weather fragment, and update the XML as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".CityWeatherFragment">

    <com.mancj.materialsearchbar.MaterialSearchBar
        android:id="@+id/sb_city_name"
        app:mt_searchBarColor="@android:color/transparent"
        app:mt_textColor="@android:color/white"
        style="@style/MaterialSearchBarLight"
        app:mt_hint="City Name"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

</FrameLayout>
```

Now change the layout to a vertical `LinearLayout`. Set the background of the search bar to transparent white, and the text colour to black. If you run the app now, it should look like shown in Figure 43.

Figure 43. App with the Search Bar

Add some `TextViews` to display the name of the city, the weather text, and the current temperature.

To populate the search field with values, we can call the Top Cities List API from AccuWeather to get the 150 top cities in the world. Try to that on your own – you know everything that is needed already. Then compare your code with what is in the GitHub repository. Here are some hints:

- Cities are locations.
- The `GmtOffset` field in `TimeZone` should be `float`, not `int`.

Making a call to a webservice is a (relatively) slow process. So, when we get the data for the top 150 cities, we are going to store the city names and their corresponding locationKey values in a hash map. Then, we do not need to make yet another call to a service to get the locationKey for the city that the user selects.

Create a new view model class to read the cities data from the webservice and contain the hash map to store the data. The model will also read the weather data for the city once we have it selected.

```
package za.ac.iie.opsc.geoweather

import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch
import za.ac.iie.opsc.geoweather.model.currentweather.CurrentWeather
import za.ac.iie.opsc.geoweather.model.location.AccuWeatherLocation
import za.ac.iie.opsc.geoweather.retrofit.RetrofitClient

class CityWeatherModel : ViewModel() {

    // cities list
    private var _citiesList = MutableLiveData<List<AccuWeatherLocation?>>()
    var citiesList: LiveData<List<AccuWeatherLocation?>> = _citiesList
    var citiesHashMap = HashMap<String, AccuWeatherLocation>()

    // current weather
    private var _currentWeather = MutableLiveData<CurrentWeather>()
    var currentWeather: LiveData<CurrentWeather> = _currentWeather

    fun getCityList() {
        viewModelScope.launch {
            val citiesListFromApi = RetrofitClient.weatherService?.
                getTop150Cities(BuildConfig.ACCUWEATHER_API_KEY)
            if (citiesListFromApi != null) {
                populateHashmap(citiesListFromApi)
                _citiesList.value = citiesListFromApi!!
            }
        }
    }

    private fun populateHashmap(citiesListFromApi:
                                List<AccuWeatherLocation?>?) {
        citiesHashMap.clear()
        if (citiesListFromApi != null) {
            for (city in citiesListFromApi) {
                citiesHashMap[city?.LocalizedName!!] = city
            }
        }
    }

    fun getCurrentWeather(locationKey: String) {
        viewModelScope.launch {
            val weatherData = RetrofitClient.weatherService?.
            getCurrentConditions(locationKey,
                BuildConfig.ACCUWEATHER_API_KEY)
            _currentWeather.value = weatherData?.get(0)
        }
    }
}
```

And then we can add the code in the fragment to get the data for the city that we search for.

```
class CityWeatherFragment : Fragment() {
    private var viewModel = CityWeatherModel()
    private lateinit var searchBar: MaterialSearchBar
    private lateinit var cityName: TextView
    private lateinit var weatherText: TextView
    private lateinit var temperature: TextView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
    }
```

```
    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        val view = inflater.inflate(R.layout.fragment_city_weather,
container, false)
        viewModel.getCityList()
        searchBar = view.findViewById<MaterialSearchBar>(R.id.sb_city_name)
        cityName = view.findViewById<TextView>(R.id.tvCityName)
        weatherText = view.findViewById<TextView>(R.id.tvWeatherText)
        temperature = view.findViewById<TextView>(R.id.tvCurrentTemperature)

        setupViewModel()
        setupSearchBar()

        return view
    }

    private fun setupViewModel() {
        // observe the list in the model for changes
        val citiesObserver = Observer<List<AccuWeatherLocation?>> {
            newCities ->
            searchBar.lastSuggestions =
                viewModel.citiesHashMap.keys.toList().sorted()
        }
        viewModel.citiesList.observe(viewLifecycleOwner, citiesObserver)

        // observe the list in the model for changes
        val weatherObserver = Observer<CurrentWeather> { weather ->
            run {
                weatherText.text = weather.WeatherText
                temperature.text = "${weather.Temperature?.Metric?.Value} " +
                        "${weather.Temperature?.Metric?.Unit}"
            }
        }
        viewModel.currentWeather.observe(viewLifecycleOwner, weatherObserver)
    }

    private fun setupSearchBar() {
        searchBar.isEnabled = true
        searchBar.addTextChangeListener(object : TextWatcher {
            override fun beforeTextChanged(
                s: CharSequence, start: Int,
                count: Int, after: Int
            ) {
            }

            override fun onTextChanged(
                s: CharSequence, start: Int,
                before: Int, count: Int
            ) {
                val suggest: MutableList<String> = ArrayList()
                for (city in viewModel.citiesHashMap.keys) {
                    if (city.lowercase(Locale.getDefault()).contains(
                            searchBar.getText().toLowerCase()
                        )
                    ) suggest.add(city)
                }
                Collections.sort(suggest)
                searchBar.setLastSuggestions(suggest)
            }

            override fun afterTextChanged(s: Editable) {}
        })

        searchBar.setOnSearchActionListener(
```

```
            object : OnSearchActionListener {
                override fun onSearchStateChanged(enabled: Boolean) {}
                override fun onSearchConfirmed(text: CharSequence) {
                    Log.d("Search:", text.toString() + "")
                    cityName.text = text.toString()
                    viewModel.getCurrentWeather(
                        viewModel.citiesHashMap[text.toString()]?.Key!!)
                }
                override fun onButtonClicked(buttonCode: Int) {}
            })
    }
}
```

4 Adding Social Media Services

Users enjoy sharing their experiences on social media. If the weather is particularly extreme, they might very well want to share that on social media. And it would be great if they could share it using our app, since that will create awareness of this amazing weather app that is out there!

You might have noticed the floating action button that was created by the project wizard together with the tabs. Well, now it will finally have a purpose: sharing a screenshot to social media. Change the image of the button to the sharing icon and change the button's ID to `fab_share`.

Create a new utility class called `ProcessImageUtil`. The class should have the below methods, to create a screenshot, save it to file, and then share it.

```
object ProcessImageUtil {
    /**
     * Convert a view to a Bitmap.
     * @param view The view to convert.
     * @return The converted Bitmap.
     */
    fun takeScreenshot(view: View): Bitmap {
        val screenView: View = view.rootView
        val bitmap: Bitmap =
            screenView.drawToBitmap(Bitmap.Config.ARGB_8888)
        return bitmap
    }

    /**
     * Write a screenshot bitmap to a folder.
     * @param context The activity that this is called from.
     * @param bitmap The bitmap to write.
     * @param fileName The name of the file to create.
     */
    fun storeScreenshot(
        context: Context, bitmap: Bitmap,
        fileName: String
    ) {
        // Get the application's folder - no permissions to write
        val directory: File? = context.getExternalFilesDir(null)
```

```
        if (directory != null && !directory.exists()) {
            val isCreated: Boolean = directory.mkdirs()
            Log.d("MakingDir", "Created: $isCreated")
        }
        val captureImage = File(directory, "$fileName.PNG")
        try {
            val writeImage = FileOutputStream(captureImage)
            bitmap.compress(Bitmap.CompressFormat.PNG, 85,
                            writeImage)
            writeImage.flush()
            writeImage.close()
        } catch (e: Exception) {
            e.printStackTrace()
        }
    }

    /**
     * Share the saved image using an intent.
     * @param context The context where the call is made from.
     * @param filename The name of the file to share.
     */
    fun pushToInstagram(context: Context, filename: String) {
        val directory: File? = context.getExternalFilesDir(null)
        val type = "image/*"
        val mimeTypeArray = arrayOf<String>(type)
        val mediaPath = "$filename.PNG"
        val share = Intent(Intent.ACTION_SEND)
        share.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
        share.type = type
        val media = File(directory, mediaPath)
        var uri: Uri? = Uri.fromFile(media)
        if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.N) {
            uri = FileProvider.getUriForFile(
                context,
                BuildConfig.APPLICATION_ID + ".provider", media
            )
        }
        share.clipData = ClipData.newRawUri("Sharing weather", uri)
        share.putExtra(Intent.EXTRA_STREAM, uri)
        share.putExtra(Intent.EXTRA_SUBJECT, "Sharing my weather")
        context.startActivity(Intent.createChooser(share,
                              "share to"))
    }
}
```

We also need to tell the operating system that we are going to provide files. To do this, add the following to the application in the AndroidManifest.xml file:

```
<provider
    android:name="androidx.core.content.FileProvider"
    android:authorities="${applicationId}.provider"
    android:grantUriPermissions="true"
    android:exported="false">
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/filepaths" />
</provider>
```


We still need to create a file called filepaths.xml. Android Studio provides a hint that will do that:

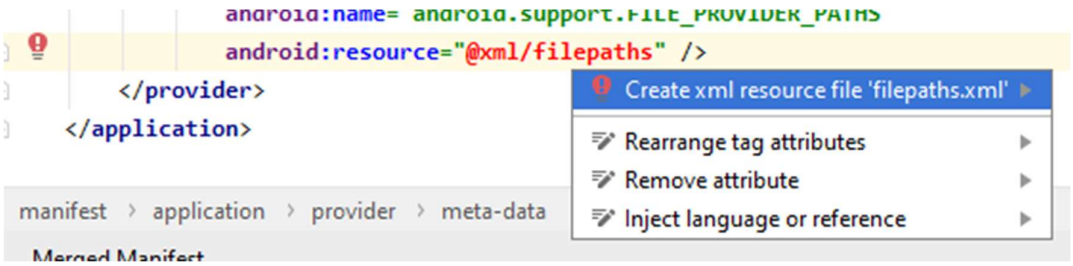


Figure 44. Creating the filepaths.xml File

Set the root element to be paths and click OK.

The content of the file should look as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<paths>

    <external-path
        name="external"
        path="." />
    <external-files-path
        name="external_files"
        path="." />
    <cache-path
        name="cache"
        path="." />
    <external-cache-path
        name="external_cache"
        path="." />
    <files-path
        name="files"
        path="." />

</paths>
```

In the main activity, we can then change the `OnClickListener` of the floating action button as follows:

```
fab.setOnClickListener { view ->
    val rootview: View = window.decorView.rootView
    val currentScreenshot = takeScreenshot(rootview)
    storeScreenshot(
        this@MainActivity,
        currentScreenshot, "Weather Today"
    )
    pushToInstagram(
        this@MainActivity,
        "/Weather Today"
    )
}
```

And that is all that is required. Now when they click the button, it will save an image and allow the user to choose where to share that to.

5 Working with SDKs

A **Software Development Kit** (**SDK**) provides "a set of tools, libraries, relevant documentation, code samples, processes, and or guides that allow developers to create software applications on a specific platform." (Sandoval, 2016)

Although the concept of an SDK might feel unfamiliar at this point, we have in fact been making use of an SDK all along – the Android SDK.

In Android Studio, the SDK Manager can be accessed from the Tools menu. There we can see that there is in fact an SDK for each version of the Android operating system. And on the second tab of the SDK manager, we see the list of tools (see Figure 45) that is installed.

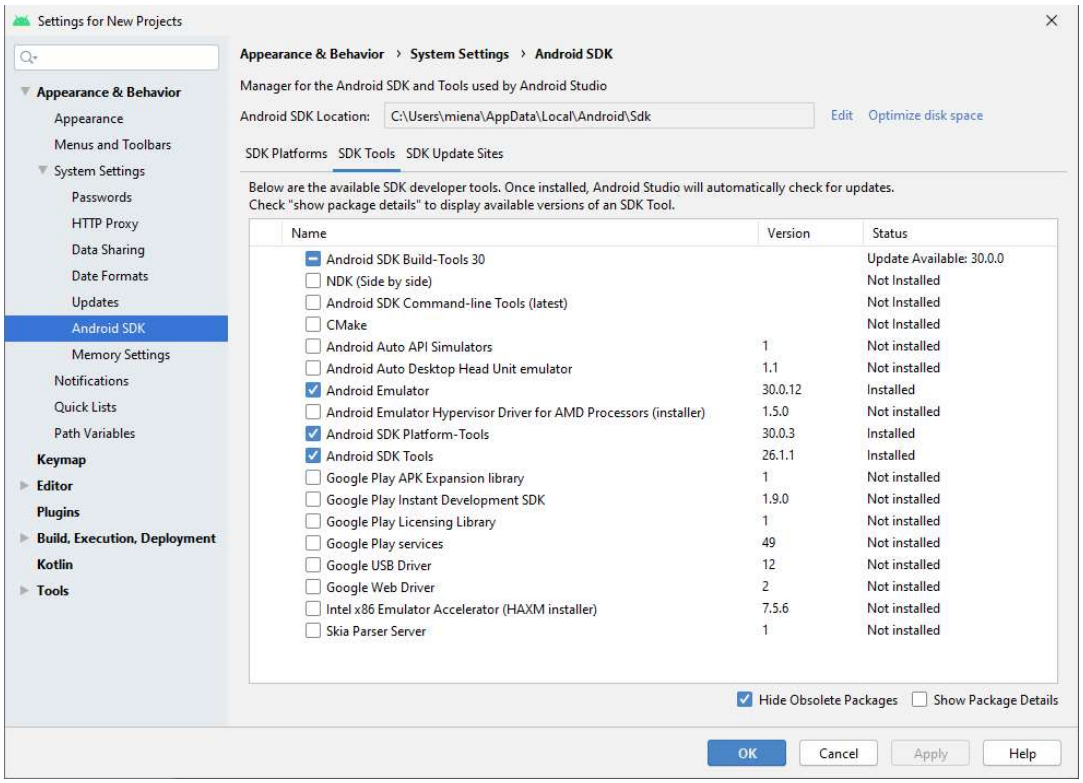


Figure 45. SDK Manager showing the included Tools

But the Android SDK is not the only one that you can use. For example, Facebook has an SDK for Android that includes logging into your app using Facebook logins and much more. (Facebook, 2020)

5.1 Mapbox

A very useful SDK to know about is the Mapbox Maps SDK. Mapbox has a lot of different products as well as tools, that makes it quite easy to display a map that is customised for a specific purpose. Sign up for a free account at https://www.mapbox.com/.

The Mapbox documentation is quite comprehensive, so we won't repeat all the same things here that are already well documented on their website. Start by reading this page:

https://docs.mapbox.com/android/maps/guides/install/

It explains step-by-step how to set up an app with a Mapbox map. Follow the instructions carefully – every single step is important. Your app should now look like the one shown in Figure 46.

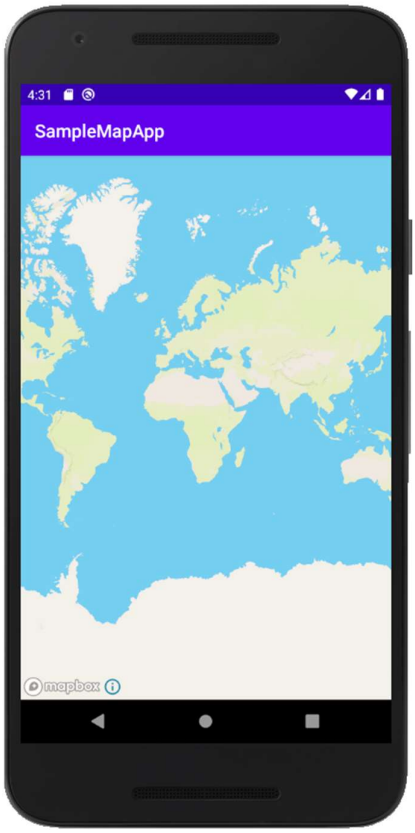


Figure 46. Basic map app

The most common problem that people encounter with Mapbox is the exception shown in Figure 47. This exception causes the app to crash on start-up.

```
Caused by: com.mapbox.mapboxsdk.exceptions.MapboxConfigurationException:
Using MapView requires calling Mapbox.getInstance(Context context, String accessToken) before inflating or creating the view.
Please see https://www.mapbox.com/help/create-api-access-token/ to learn how to create one.
More information in this guide https://www.mapbox.com/help/first-steps-android-sdk/#access-tokens.
    at com.mapbox.mapboxsdk.maps.MapView.initialize(MapView.java:132)
    at com.mapbox.mapboxsdk.maps.MapView.<init>(MapView.java:108)
        ... 28 more
```

Figure 47. Mapbox Exception

The exception gives us all the information that we need to solve the problem. This is the one case where you absolutely must call something before `setContentView`. The correct order is shown in Figure 48. First call `Mapbox.getInstance`, and then `setContentView`.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    Mapbox.getInstance( context: this, getString(R.string.mapbox_access_token));

    setContentView(R.layout.activity_main);
```

Figure 48. Correct Initialisation Code

Beyond the basic map app, there is a tool that is very useful for you to know about – Mapbox Studio. It is an online tool that allows you to create a custom map style. As part of the style, you can choose what is displayed, fonts, colours, and more.

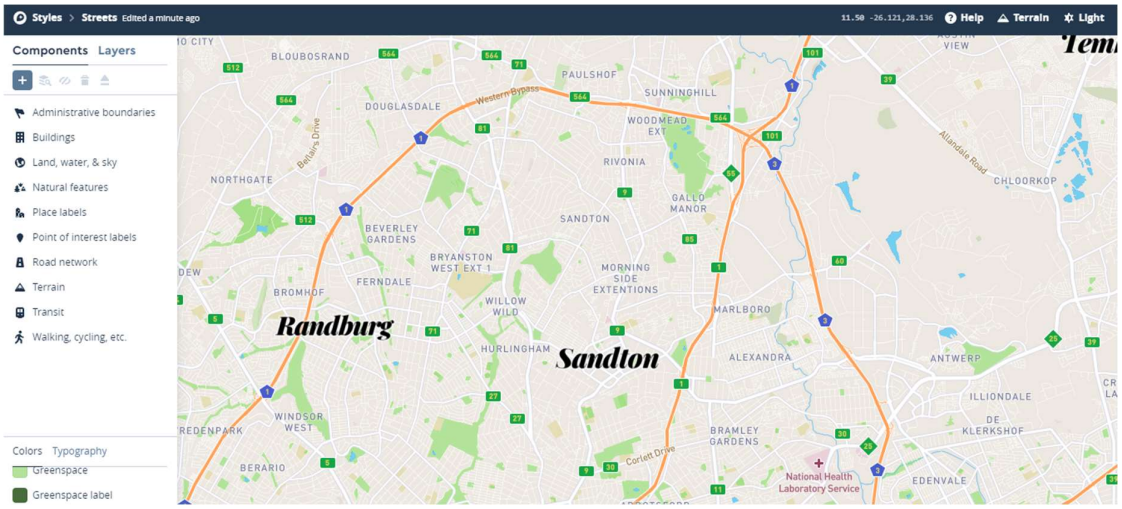


Figure 49. Mapbox Studio

Mapbox Studio can be accessed here:

https://studio.mapbox.com/

Read the documentation for Mapbox Studio here:

https://docs.mapbox.com/studio-manual/guides/

5.2 HERE maps

An alternative to Mapbox is HERE maps. Read more about the HERE SDK here: https://developer.here.com/products/here-sdk

6 Recommended Additional Reading

Droid By Me. 2018. *Get Current location using FusedLocationProviderClient in Android.* [Online] Available at: https://medium.com/@droidbyme/get-current-location-using-fusedlocationproviderclient-in-android-cb7ebf5ab88e [Accessed 31 July 2023].

Karnok, D. 2017. *ReactiveX / RxJava.* [Online] Available at: https://github.com/ReactiveX/RxJava/wiki [Accessed 31 July 2023].

Sandoval, K. 2016. *What is the Difference Between an API and an SDK?.* [Online] Available at: https://nordicapis.com/what-is-the-difference-between-an-api-and-an-sdk/ [Accessed 31 July 2023].

Square, Inc., n.d. *Retrofit: A type-safe HTTP client for Android and Java.* [Online] Available at: https://square.github.io/retrofit/ [Accessed 31 July 2023].

7 Activities

Complete the activities on Learn.

Learning Unit 3: Using Databases	
Learning Objectives: • Create an SQLite Database. • Use an SQLite Database to store data. • Use an SQLite database to read data. • Use an authentication service. • Use blob storage. • Use a NoSQL database to store data. • Use a NoSQL database to read data.	**My notes**
Material used for this learning unit: • GitHub repository: Learning Unit 3	
How to prepare for this learning unit: • Make sure that you have the GitHub source code available and that your Android Studio is up to date.	

1 Introduction

In this learning unit, we are going to create a photo memories app called Photo Memories. The first version of the app will store photos locally, with a description for each photo that is stored in an SQLite database. The second version of the app will add Firebase Authentication, storing the descriptions in the Firebase Realtime Database, and storing the photos using Firebase Storage.

2 Creating and Accessing an SQLite Database

2.1 What is SQLite?

Let us look at what the SQLite project says about itself:

> "**SQLite** is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day." (sqlite.org, n.d.)

SQLite is a Relational Database Management System (RDBMS) that implements Structured Query Language (SQL). All the SQL that you learned in Databases will finally be useful now.

Unlike many of the other RDBMSs out there though, SQLite does not require a server to be running. The file that stores the data is accessed directly using libraries. (sqlite.org, n.d.b)

2.2 Setting Up the User Interface

Let us start by creating a user interface. We are going to have two activities: a main activity that allows the user to choose whether to work with local or cloud images (the cloud button will remain disabled for now), and another that allows the user to manage local images.

Hint: Use a fragment to contain the buttons on the main activity. That will make it easier to swap between the login fragment and this fragment later.

Your user interface should look something like what is shown in Figure 50.

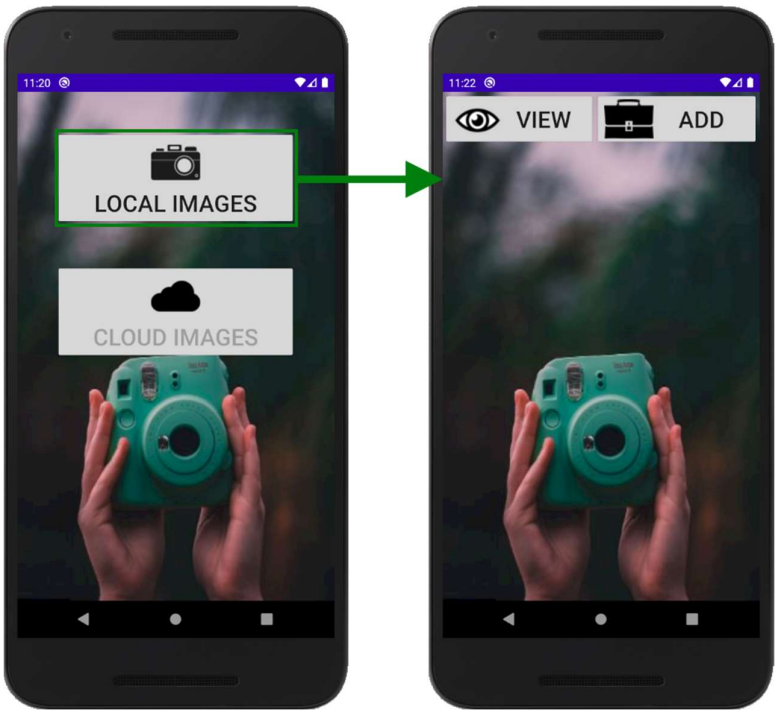


Figure 50. Initial User Interface for Photo Memories

Remember to add an app icon too.

To the local images activity, add a placeholder that we can later replace with a fragment when the user clicks either view or add.

Your code should now look like tag LU3-1.3 in the GitHub repository.

Question: How can you make the buttons look better?

2.3 Adding Photos

If you store images that are too big, the app might not be able to read those again.

First, we need to create a few more UI things. Create a new fragment called `LocalImagesStoreFragment`, which will allow the user to pick an image, and enter the description to store with the image.

To display the fragment, we need to first create an instance of the fragment in `LocalImagesActivity`. And then we can add an `OnClickListener` to the button in the `OnCreate` method, that uses the `FragmentManager` to switch to that fragment instance:

```
class LocalImagesActivity : AppCompatActivity() {

    private val storeFragment = LocalImagesStoreFragment()
    lateinit var binding: ActivityLocalImagesBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityLocalImagesBinding.inflate(layoutInflater)
        setContentView(binding.root)
        binding.btnAdd.setOnClickListener {
            val manager: FragmentManager = supportFragmentManager
            val transaction: FragmentTransaction =
                        manager.beginTransaction()
            transaction.replace(R.id.local_image_place_holder,
                                storeFragment)
            transaction.commitAllowingStateLoss()
        }
    }
}
```

Then add some controls to the fragment, that will allow the user to interact with it (see Figure 51).

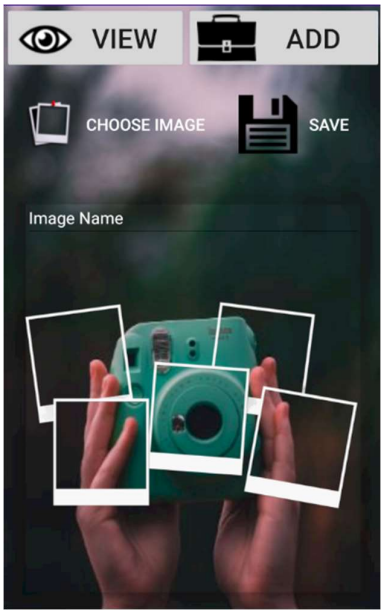


Figure 51. Store Images Design

In the `LocalImagesStoreFragment`, we are going to register for events that allow us to choose an image, and then add an `OnClickListener` to the choose button that launches the media request.

```
class LocalImagesStoreFragment : Fragment() {
    lateinit var binding: FragmentLocalImagesStoreBinding
    private var bitmap: Bitmap? = null
    private lateinit var pickMedia:
                ActivityResultLauncher<PickVisualMediaRequest>

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setupImageChooser()
    }

    override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflate the layout for this fragment
        binding = FragmentLocalImagesStoreBinding.inflate(inflater)
        binding.btnChooseImage.setOnClickListener {
            pickMedia.launch(
                PickVisualMediaRequest(
                    ActivityResultContracts.PickVisualMedia.ImageOnly
                )
            )
        }
        return binding.root
    }

    private fun setupImageChooser() {
        // From https://developer.android.com/training/data-
        // storage/shared/photopicker
        pickMedia = registerForActivityResult(
            ActivityResultContracts.PickVisualMedia()
        ) { uri ->
            // Callback is invoked after the user selects a media item or
```

```
            // closes the photo picker.
            if (uri != null) {
                val inputStream =
                     context?.contentResolver?.openInputStream(uri)
                bitmap = BitmapFactory.decodeStream(inputStream)
                binding.imgImagepane.setImageBitmap(bitmap)
            } else {
                Toast.makeText(
                    getContext(), "No image selected",
                    Toast.LENGTH_SHORT
                ).show();
            }
        }
    }
}
```

Now we are ready to store the image in the database. First, create a package called model with a class called `ImageModel`. This is going to store all the information that we need about an image, which for now is just the image and the name that the user entered.

```
data class ImageModel (public val imageName: String?,
                       public val imageBitmap: Bitmap?) {
}
```

Now we need to start interacting with the database. The SQLite libraries for Android includes a class called `SQLiteOpenHelper`, which makes it quite easy to connect to an SQLite database. (Android Open Source Project, 2020e)

We are going to create a class that extends from `SQLiteOpenHelper`. There are two abstract methods that we must implement. From the Java source code for `SQLiteOpenHelper`:

```
public abstract void onCreate(SQLiteDatabase db);
public abstract void onUpgrade(SQLiteDatabase db, int oldVersion,
                               int newVersion);
```

The `SQLiteOpenHelper` class not only helps with creating and accessing a database, it also makes it easy to have a different version of the database. The `onUpgrade` method will get called if the version of the database that is stored is not the same as the version that is required by the code. For now, we do not need to do anything in that method, since we do not have multiple versions of the database yet.

What we do need to implement is creating a database if it does not exist yet, using the `onCreate` method.

Create a package called database, and inside it a class called `DatabaseHandler`.


```
private const val DATABASE_NAME = "images.db"
private const val DATABASE_VERSION = 1
private const val createTableQuery =
    "create table imageStore(imageName TEXT, imageBitmap BLOB)"

class DatabaseHandler(private val context: Context?) :
    SQLiteOpenHelper(context, DATABASE_NAME, null,
        DATABASE_VERSION) {

    override fun onCreate(db: SQLiteDatabase?) {
        try {
            db?.execSQL(createTableQuery)
            Toast.makeText(
                context, "Table created",
                Toast.LENGTH_SHORT
            ).show()
        } catch (e: Exception) {
            Toast.makeText(
                context, e.message,
                Toast.LENGTH_SHORT
            ).show()
        }
    }

    override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int,
                           newVersion: Int) {
    }
}
```

In the constructor, we call the super class' constructor to initialise it with the correct name and database version.

In the `onCreate` method, we execute a SQL command to create the imageStore table, with imageName and imageBitmap fields. Note that the imageBitmap is a BLOB – a Binary Large OBject. This data type is used to store binary data, such as image data, in a database.

Declare the following field in `LocalImagesStoreFragment` and initialise it in the onCreate method:

```
private lateinit var imagedb: DatabaseHandler

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setupImageChooser()
    imagedb = DatabaseHandler(activity)
}
```

Now we are instantiating the `DatabaseHandler`, which is a good step in the right direction. But how do we write the data? Let us add another method to `DatabaseHandler` to store an image.

```
fun storeImageLocal(imageModel: ImageModel) {
    try {
        val imageDatabase = this.writableDatabase
        if (imageModel.imageBitmap != null) {
            val imageToStore: Bitmap = imageModel.imageBitmap
            val convertBitmapToByteArray = ByteArrayOutputStream()
            imageToStore.compress(
                Bitmap.CompressFormat.JPEG,
                100, convertBitmapToByteArray
            )
            val imageInBytes = convertBitmapToByteArray.toByteArray()
            val contentValues = ContentValues()
            contentValues.put("imageName", imageModel.imageName)
            contentValues.put("imageBitmap", imageInBytes)
            val checkIfQueryRuns = imageDatabase.insert(
                "imageStore",
                null, contentValues
            )
            if (checkIfQueryRuns != -1L) {
                Toast.makeText(
                    context, "Image saved",
                    Toast.LENGTH_SHORT
                ).show()
                imageDatabase.close()
            } else {
                Toast.makeText(
                    context, "Unable to save Image",
                    Toast.LENGTH_SHORT
                ).show()
            }
        }
    } catch (e: java.lang.Exception) {
        Log.i("SAVE TO DB ", "storeImageLocal: " + e.message)
    }
}
```

Here we create a JPG compressed version of the image, and then insert it into the database together with its description.

The `SQLiteDatabase.insert` method returns the row ID of the newly inserted row, or -1 if an error occurred.

Now we have everything in place to write the image to the database. Add an `OnClickListener` to the save button where you call the `saveImage` method.

```
binding.btnSave.setOnClickListener {
    if (binding.txtImageDescription.text.toString() != null &&
        bitmap != null) {
        val imageToStore = ImageModel(
            binding.txtImageDescription.text.toString(), bitmap
        )
        imagedb.storeImageLocal(imageToStore)
    }
}
```


2.4 Viewing Photos

Storing photos in a database might be exciting to us as developers. But the user will not care one bit about that if they cannot view their stored photos too. So, let us implement that next.

Create a new Fragment (List) `LocalImagesViewFragment` and switch to it when the view button is clicked.

Edit the item fragment layout to have a `TextView` and an `ImageView` to display an image.

Then update the recycler view adapter to display the correct information – we are going to use a list of `ImageModel` objects.
Hint: `ImageView` has a method called `setImageBitmap`.

Now we can add a method to the `DatabaseHandler` to read the data and create an `ArrayList` of `ImageModel` objects.

```
fun readDisplayImages(): ArrayList<ImageModel>? {
    return try {
        val imagDatabase = this.readableDatabase
        val dbImages: ArrayList<ImageModel> = ArrayList()
        val cursor: Cursor = imagDatabase.rawQuery(
            "select * from imageStore",
            null
        )
        if (cursor.getCount() !== 0) {
            while (cursor.moveToNext()) {
                val imageName: String = cursor.getString(0)
                val image: ByteArray = cursor.getBlob(1)
                val imageBitmap = BitmapFactory.decodeByteArray(
                    image,
                    0, image.size
                )
                dbImages.add(ImageModel(imageName, imageBitmap))
            }
            Toast.makeText(
                context, "Loading Images",
                Toast.LENGTH_SHORT
            ).show()
            dbImages
        } else {
            Toast.makeText(
                context, "No Images Found ",
                Toast.LENGTH_SHORT
            ).show()
            null
        }
    } catch (e: java.lang.Exception) {
        Log.i("SAVE TO DB ", "storeImageLocal: " + e.message)
        null
    }
}
```

In `LocalImagesViewFragment`, we need to read the data from the database. Create fields for the `recyclerView` and `imageDb`, then initialise those in the `onCreateView` method:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    val view = inflater.inflate(
        R.layout.fragment_local_images_view_list,
        container, false)

    // Set the adapter
    if (view is RecyclerView) {
        recyclerView = view
        with(view) {
            layoutManager = when {
                columnCount <= 1 -> LinearLayoutManager(context)
                else -> GridLayoutManager(context, columnCount)
            }
            var list = mutableListOf<ImageModel>()
            adapter = MyLocalImageModelRecyclerViewAdapter(list)
        }
        imageDb = DatabaseHandler(activity)
        getData()
    }
    return view
}
```

Lastly, we need the `getData` method, that calls the `DatabaseHandler` to get the data.

```
private fun getData() {
    try {
        val images = imageDb.readDisplayImages()
        if (images != null) {
            val photoViewAdapter =
                MyLocalImageModelRecyclerViewAdapter(images)
            recyclerView.setHasFixedSize(true)
            recyclerView.layoutManager = LinearLayoutManager(context)
            recyclerView.adapter = photoViewAdapter
        } else {
            Toast.makeText(
                context, "No Images found",
                Toast.LENGTH_SHORT
            ).show()
        }
    } catch (e: Exception) {
        Toast.makeText(context, e.message, Toast.LENGTH_SHORT).show()
    }
}
```

That is it – we can now view images from the database!

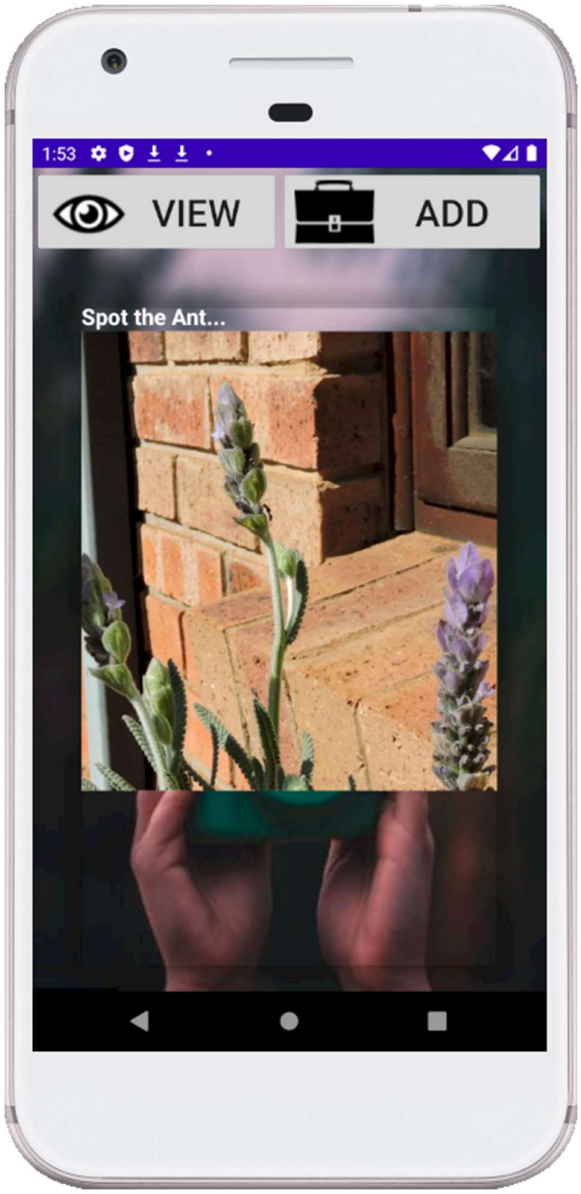


Figure 52: App Displaying Local Images

3 Access a NoSQL Database

In Open Source Coding (Introduction) (OPSC7311), we already made use of the Firebase Realtime Database as well as Cloud Firestore. Now we are going to take it to the next level and use some of the other Firebase services too.

You need to create an app in the Firebase console. The console can be accessed at https://console.firebase.google.com/

> If you are making use of the **example code** from the GitHub repository, you still need to create the project on your own Firebase account and connect the app to that project.

To create a project in the Firebase console:

1. Browse to https://console.firebase.google.com/
2. Click **Add project**.
3. Enter a name for the project: **PhotoMemories**.
4. Enable Google Analytics for this project on the following page.
5. Choose the account: **Default Account for Firebase**.
6. Click **Create Project**.

3.1 Connecting the App to Firebase

We are going to continue working on the app from section 1.

Important: The package name of the app that you are linking to the Firebase project is used as part of the whole process to identify the app. Remember how the Play Store doesn't allow that generic com.example name? Well, if you ever want to publish the app **now** is the best time to check that you have a unique package name. This can be viewed and changed in the build.gradle file. Read (Android Open Source Project, 2020f) for more details about that.

If you forget about this and need to fix issues with connecting Firebase later after changing the package name, exit Android Studio and delete the following file inside the project folder:

```
app\google-services.json
```

Then open the project in Android Studio again and connect to Firebase once more.

To connect the app to the newly created Firebase project:

1. In Android Studio, click **Tools** on the main menu and then click **Firebase**. The Firebase Assistant user interface will be displayed.

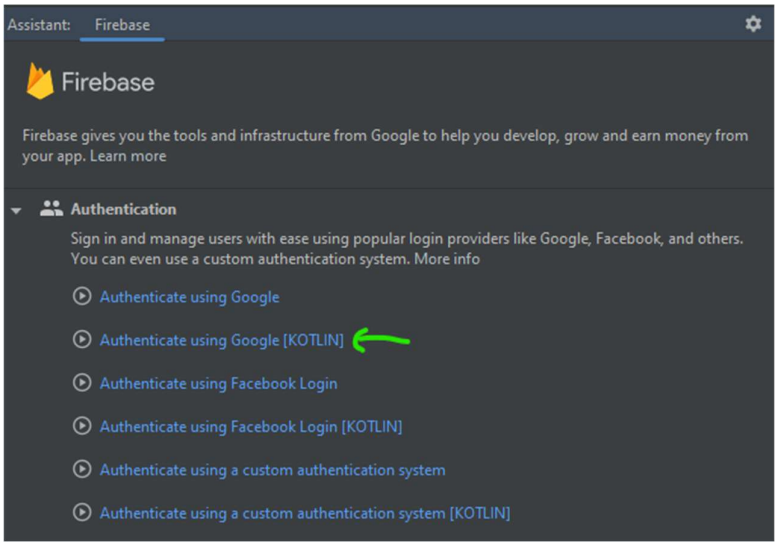


Figure 53. Firebase Assistant

2. Scroll up to **Authentication** and expand it.
3. Click **Authenticate using Google [KOTLIN]**.

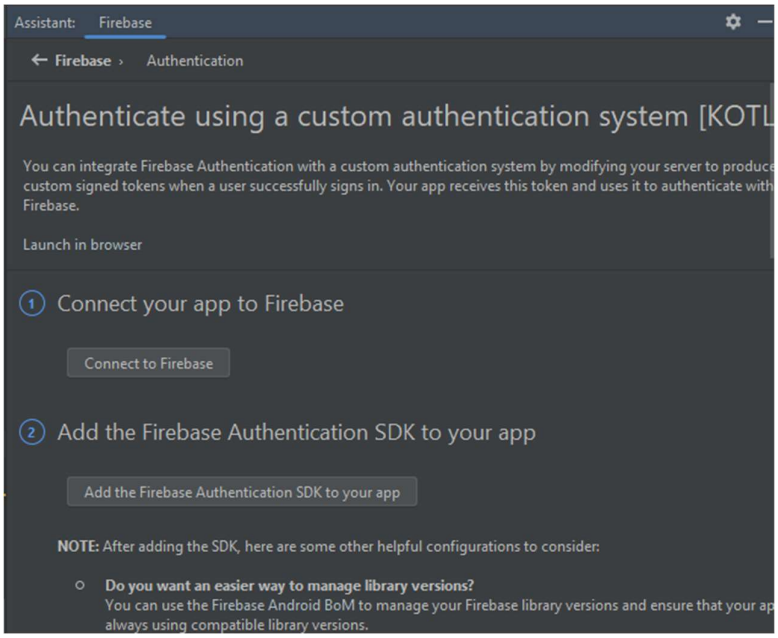


Figure 54. Connect to Firebase

4. Click **Connect to Firebase**.
5. A browser window will open where you can log in and choose which project to connect to. Follow the prompts and click **Connect** when that option becomes available.

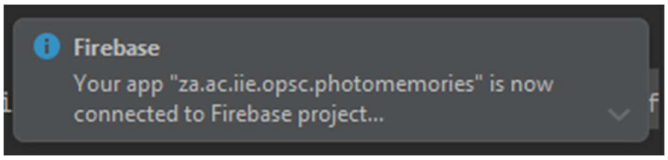


Figure 55. Firebase successfully connected

6. The assistant should now show that the app was connected to Firebase (as shown in Figure 55).
7. Click **Add Firebase Authentication SDK to your app**.
8. Follow the prompts to add the necessary libraries to your project and wait for the Gradle process to complete.

The details of which Firebase project to connect to is stored in the file app\google-services.json. It does not store your Google username and password, but rather the project name, project number and an API key. This is still information that you might not want to share with the whole world though, so think twice before checking this into a publicly available repository.

3.2 Allowing Users to Register

Now we need to enable email and password authentication in the Firebase project too.

To enable email and password authentication:

1. Browse to the **Firebase Console** again and select your **PhotoMemories** project.
2. Under the **Build** category, click **Authentication**.
3. Click **Get Started**.
4. Click the **Sign-in method** tab.

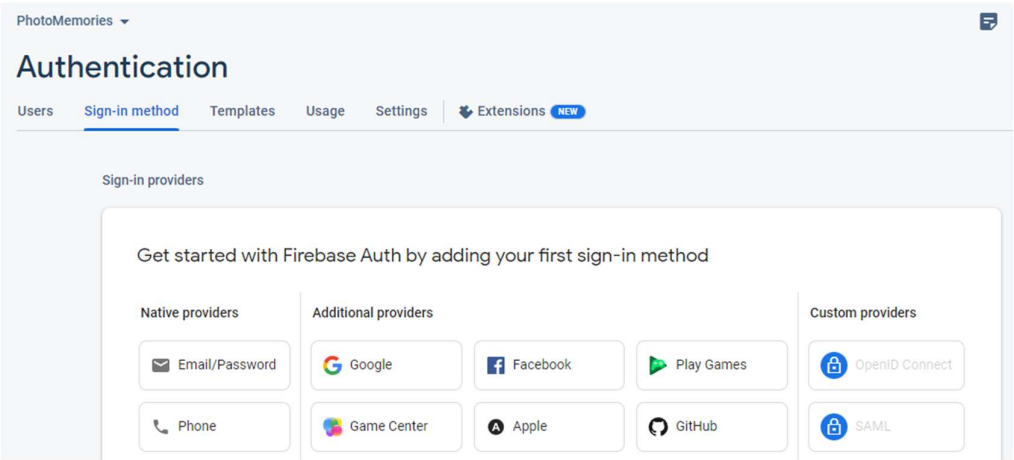


Figure 56. Enabling Email/Password Authentication

5. Click the **Email/Password** option and click the enable.

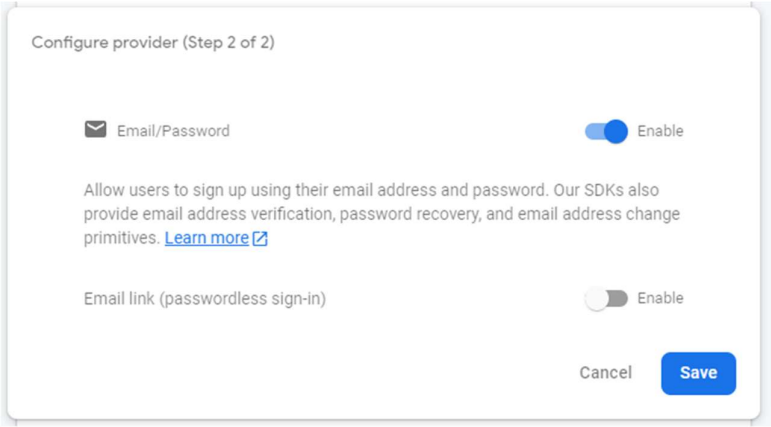


Figure 57. Enabling Authentication

6. Click the **Enable** toggle button (the top one) and then click **Save**.

If you think back to section 1, you might remember that we created a fragment for the main menu choices to make switching that out later easier. Well, the time has now arrived to do exactly that. Create a new **Blank Fragment** called **LoginFragment**. The fragment must contain email and password edit texts as well as sign in and register buttons. Add some graphics to it. Change the main activity to display the login fragment. The login screen should then look something like the one shown in Figure 58.

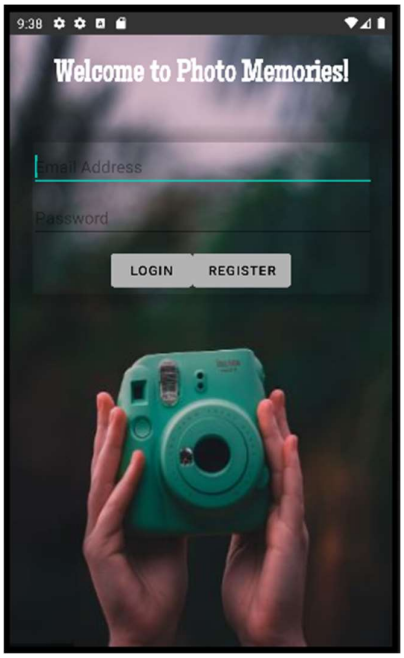


Figure 58. Login Screen

Now we can add the code for users to register. Add a field to the fragment for the authentication, and one for the binding:

```
private lateinit var auth: FirebaseAuth
lateinit var binding: FragmentLoginBinding
```

In the `onCreate` method, assign a value to that field:

```
auth = Firebase.auth
```

And change the `onCreateView` method to use binding:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    // Inflate the layout for this fragment
    binding = FragmentLoginBinding.inflate(inflater)
    return binding.root
}
```

Now we are ready to add an `OnClickListener` to the register button. We get the username and password from the two edit texts, and then pass that information to the Firebase `authentication` object.

```
binding.btnRegister.setOnClickListener {
    val email = binding.txtUName.text.toString()
    val password = binding.txtPword.text.toString()
    if (email.isNotEmpty() && password.isNotEmpty()) {
        auth.createUserWithEmailAndPassword(email, password)
            .addOnCompleteListener { if (it.isSuccessful) {
                Log.d("Login", "createUserWithEmail:success")
                val user = auth.currentUser
            } else {
                Log.w("Login", "createUserWithEmail:failure",
                      it.exception)
                Toast.makeText(context, "Unable to register",
                    Toast.LENGTH_SHORT).show()
            } }
    }
    else {
        Toast.makeText(context, "Username and password cannot " +
                               "be blank",
            Toast.LENGTH_SHORT).show()
    }
}
```

Notice that a listener is added that will get called **asynchronously** once the registration is complete. So, remember that any code that depends on the completed registration should happen in there, not in the next line of the `onClick` method.

Now we can run the app and allow a user to register. Then we can see the username and user UID in the Firebase Console as shown in Figure 59.

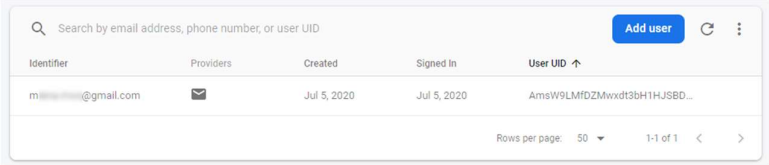


Figure 59. New User in the Firebase Console

Notice that we do not need to do anything to securely store the password, and we also cannot see the user's password. The Firebase Authentication service takes care of all of that for us.

3.3 Logging In

Now we are ready to let the users log in. Since all the work is getting done on a fragment, we need to find a way to communicate back to the parent activity.

When a login happens, there is a call back that is made to an `OnCompleteListener`. We can have our `MainActivity` implement this interface, then we can get the framework to call it when required.

To the fragment, add a property for the listener:

```
lateinit var loginListener: OnCompleteListener<AuthResult>
```

Then we can add a listener for the login button, that signs the user in and instructs the framework to call the `onCompleteListener`.

```
binding.btnLogin.setOnClickListener {
    val email = binding.txtUName.text.toString()
    val password = binding.txtPword.text.toString()
    if (email.isNotEmpty() && password.isNotEmpty()) {
        auth.signInWithEmailAndPassword(email, password)
            .addOnCompleteListener(loginListener)
    }
    else {
        Toast.makeText(context, "Username and password cannot" +
                "be blank",
            Toast.LENGTH_SHORT).show()
    }
}
```

Now the bulk of the remaining work is in the `MainActivity` class. We need to call on the `supportFragmentManager` so we can set the property for the listener.

```
class MainActivity : AppCompatActivity(),
                     OnCompleteListener<AuthResult> {
    override fun onCreate(savedInstanceState: Bundle?) {
        supportFragmentManager.addFragmentOnAttachListener {
                fragmentManager, fragment ->
            if (fragment is LoginFragment) {
                fragment.loginListener = this
            }
        }
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }

    override fun onComplete(task: Task<AuthResult>) {
        if (task.isSuccessful) {
            Toast.makeText(this, "You have signed in " +
                    task.result.user?.email,
                Toast.LENGTH_SHORT).show();
            loadFragment();
        } else {
            Toast.makeText(this,
                "Boo Boo Happened when logging in",
                Toast.LENGTH_SHORT).show();
        }
    }

    /**
     * Switch the fragment on successful login.
     */
    fun loadFragment() {
        val manager: FragmentManager = supportFragmentManager
        val transaction: FragmentTransaction =
                manager.beginTransaction()
        transaction.replace(R.id.layout_fragment,
                            MainChoiceFragment())
        transaction.commit()
    }
}
```

And we need to implement `OnCompleteListener<AuthResult>` and implement the `onComplete` method that handles the login response. Finally, we switch fragments if the user logged in successfully.

3.4 Storing Images in the Cloud

Now that the user has logged in, we are ready to finally enable that **Cloud Images** button and implement the functionality behind that.

We are going to store an entry for each photo in the Firebase Realtime Database. And the image itself is going to be stored in Firebase Storage.

First, we need to set up **Storage** in the Firebase project. Go to the **Firebase Console**, and under **Build** click **Storage**. Click **Get Started** and follow the prompts to set up the storage. Choose to store your data in Europe. Also, create a **Realtime Database**.

Create a new activity called `CloudImagesActivity`, that looks like the `LocalImagesActivity` that we created before. Also create the fragment for storing cloud images and call it `CloudImagesStoreFragment`.

We are going to be accessing the Internet, so add this permission to the AndroidManifest.xml file:

```
<uses-permission android:name="android.permission.INTERNET"/>
```

The choosing of the image still works the same, so go ahead and add the same code to the store fragment that we had for the local images. The only difference is that this time we need to keep the Uri too, so create a class field for `imageData`.

Add **Cloud Storage for Firebase** and **Firebase Realtime Database** to the project using the Firebase Assistant.

Now we need to write the image to the Firebase Storage, and that requires an instance of `StorageReference`. Add it to the fragment:

```
private lateinit var storage: FirebaseStorage
```

And initialise it in the `onCreate` method:

```
storage = Firebase.storage storage = Firebase.storage
```

We also need authentication information since we want to store the data for the currently logged in user. So, create these fields:

```
private lateinit var auth: FirebaseAuth
```

And initialise the `auth` in `onCreate` too:

```
auth = Firebase.auth
```

Lastly, we need to get the database:

```
private var myRef = Firebase.database.getReference("PhotoMemories")
```

When we create an image in Firebase Storage, we will get back the URI where it is stored. So, we need to include that in the `ImageModel` that we are going to store (new part highlighted). And we will need a no parameters constructor later when we read the data from the database.

```
data class ImageModel (public val imageName: String?,
                       public val imageBitmap: Bitmap?,
                       public val imageUri: String?) {

    public constructor() : this(null, null, null) {
    }
}
```

Fix the call to ImageModel in the LocateImagesStoreFragment as well as the database handler.

If you set up Storage in production mode, update the rules to allow access in the Firebase console to look as follows:

```
rules_version = '2';
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      allow read, write: if request.auth != null;
    }
  }
}
```

And for the Realtime Database:

```
{
  "rules": {
    ".read": "auth.uid !== null",
    ".write": "auth.uid !== null"
  }
}
```

Now we can write the image to storage and create an entry in the real-time database. Remember to call the `storeImage` method when the save button is clicked.

```
private fun getFileExtensions(uri: Uri): String? {
    val contentResolver: ContentResolver =
        requireActivity().contentResolver
    val mime: MimeTypeMap = MimeTypeMap.getSingleton()
    return mime.getExtensionFromMimeType(
        contentResolver.getType(uri))
}

private fun storeImage(name: String) {
    try {
        if (imageData != null) {
```

```
            // upload the image to storage
            val fileRef: StorageReference = storage.reference.child(
                System.currentTimeMillis().toString() + "." +
                        getFileExtensions(imageData!!)
            )
            var uploadTask = fileRef.putFile(imageData!!)
            var uriTask = uploadTask.continueWithTask { task ->
                if (!task.isSuccessful) {
                    task.exception?.let {
                        throw it
                    }
                }
                fileRef.downloadUrl
            }.addOnSuccessListener { taskSnapshot ->
                // write an entry to realtime database
                var currentUser = auth.getCurrentUser()
                val imageModel = ImageModel(
                    name.trim(),
                    null,
                    taskSnapshot.toString()
                )
                val uploadID = myRef.push().key
                currentUser?.uid?.let {
                    myRef.child(it).child((uploadID)!!)
                        .setValue(imageModel)
                }
                Toast.makeText(
                    context,
                    "Photo loaded to the cloud :-)",
                    Toast.LENGTH_SHORT
                ).show()

                // reset the user interface
                binding.imgImagepane.setImageResource(
                    R.drawable.photo)
                binding.txtImageDescription.setText("")
            }
        } else {
            Toast.makeText(
                context, "Please select an Image ",
                Toast.LENGTH_SHORT
            ).show()
        }
    } catch (e: Exception) {
        Toast.makeText(context, e.message, Toast.LENGTH_SHORT).show()
    }
}
```

Tip: If you still struggle with permissions despite adding the INTERNET permission, uninstall the app from the emulator and reinstall it.

Now we can run the app and save a photo. The entry will appear in the Realtime Database (see Figure 60), and Storage (see Figure 61).

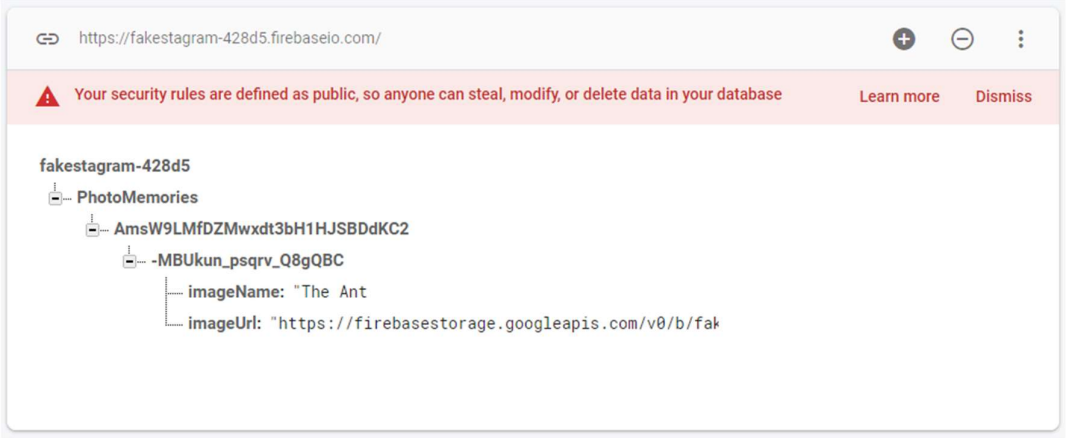


Figure 60. Entry in the Realtime Database

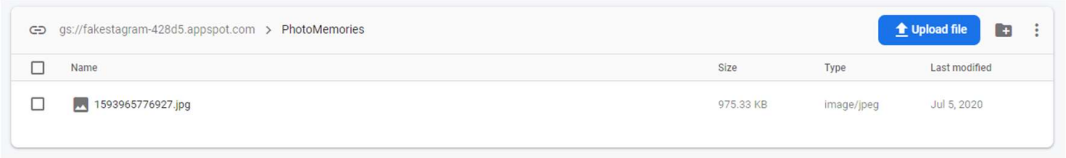


Figure 61. File Created in Storage

3.5 Viewing Images from the Cloud

The last part of the app that remains to be implemented is reading the images from the cloud and displaying them to the user. Of course, a real photo memories app would probably also have social networking aspects. But that is beyond this scope of this module.

We are again going to have a **List Fragment** that displays images, so go ahead and create that. Call the fragment `CloudImagesViewFragment` this time. **Hint:** You can reuse the layouts for the items and items list that was used for the local images.

We are going to make use of the library Picasso to load and display the images. So, add the following dependency to the app\build.gradle file:

```
implementation 'com.squareup.picasso:picasso:2.8'
```

Remember to do a Gradle sync so the library gets downloaded.

Now in the adapter for the cloud view, we can change the `onBindViewHolder` method to make use of Picasso to load the image from the URI, and display it.

```
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    val item = values[position]
    holder.nameView.text = item.imageName
    holder.imageView.setImageBitmap(item.imageBitmap)
    Picasso.get().load(item.imageUri).fit()
        .centerCrop().into(holder.imageView)
}
```

The last thing that remains is for us to read the data from the database and populate the list that the view displays. In the fragment, create the following fields:

```
private lateinit var auth: FirebaseAuth
private lateinit var storage: FirebaseStorage
private var myRef = Firebase.database.getReference("PhotoMemories")
private lateinit var recyclerView: RecyclerView
```

And then we can retrieve the data from the user's folder in the database in the onCreateView method:

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    val view = inflater.inflate(
        R.layout.fragment_local_images_view_list,
        container, false)

    // Set the adapter
    if (view is RecyclerView) {
        recyclerView = view
        with(view) {
            layoutManager = when {
                columnCount <= 1 -> LinearLayoutManager(context)
                else -> GridLayoutManager(context, columnCount)
            }
            var list = mutableListOf<ImageModel>()
            adapter = MyCloudImageModelRecyclerViewAdapter(list)
            readData()
        }
    }
    return view
}

private fun readData() {
    var cloudPicsList = arrayListOf<ImageModel>()
    val user: String? = auth.currentUser?.uid
    val userReference: DatabaseReference = myRef.child(user!!)
    userReference.addValueEventListener(object : ValueEventListener {
        override fun onDataChange(dataSnapshot: DataSnapshot) {
            for (cloudImages in dataSnapshot.children) {
                val imageModel = cloudImages.getValue(
                    ImageModel::class.java)
                if (imageModel != null) {
                    cloudPicsList.add(imageModel)
                }
            }
            var cloudPhotoViewAdapter =
                MyCloudImageModelRecyclerViewAdapter(cloudPicsList)
```

```
            recyclerView.setAdapter(cloudPhotoViewAdapter)
        }

        override fun onCancelled(databaseError: DatabaseError) {
            Toast.makeText(
                context, databaseError.message,
                Toast.LENGTH_SHORT
            ).show()
        }
    })

}
```

And that is that. Now we can view images from the cloud.

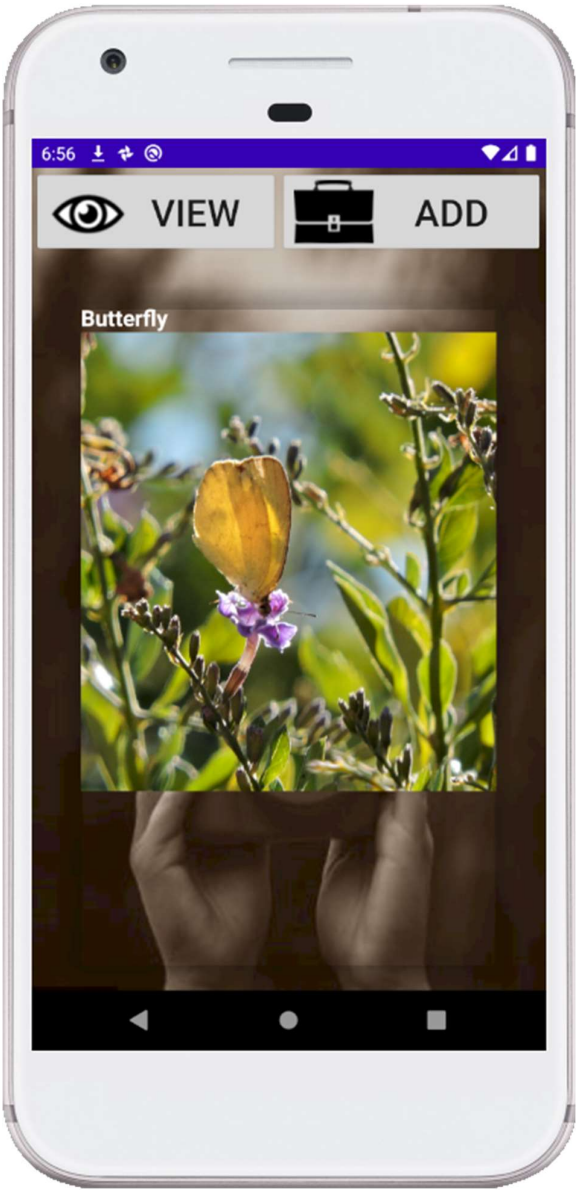


Figure 62. Viewing Cloud Images in the App

4 Recommended Additional Reading

sqlite.org. n.d. *Appropriate Uses For SQLite.* [Online] Available at: https://www.sqlite.org/whentouse.html [Accessed 31 July 2023].

5 Activities

Complete the activities on Learn.

Learning Unit 4: App Publication	
Learning Objectives: • Explain the purpose of generating an APK. • Create an APK from an existing project. • Explain the difference between an APK and a bundle. • Generate signed and unsigned bundles. • Describe how to deploy an app to the Play Store. • Prepare an app for publication to the Play Store. • Explain the purpose of analytics tracking. • Discuss how to track analytics on the Play Store.	**My notes**
Material used for this learning unit: • This module manual.	
How to prepare for this learning unit: • Make that your Android Studio is up to date.	

1 Introduction

We have built a few very interesting apps so far this semester. But all of that is not very useful if we cannot get the app published on the Google Play Store. How would people know about your amazing app if they could not search for it there?

In this learning unit, we will look at how to publish an app.

2 APKs and Bundles

Preparing an app for release is a multi-step process, as shown in Figure 63. Only once all the preparation is done, can you deploy (release) the app to the Play Store. (Android Open Source Project, 2019)

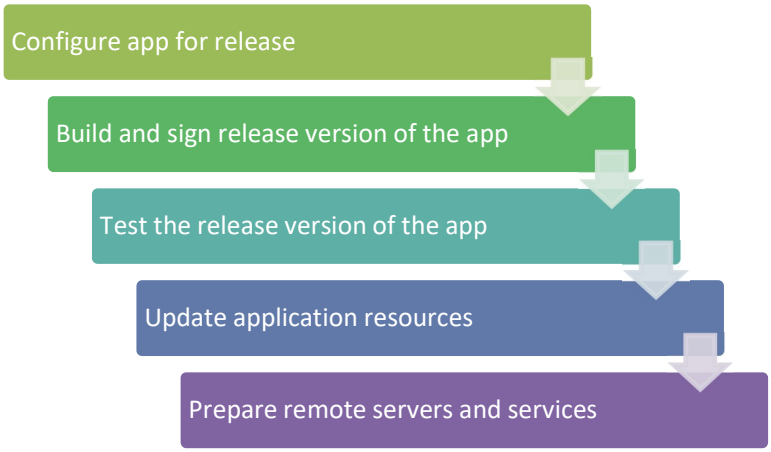


Figure 63. Preparing the App for Release
(Steps from (Android Open Source Project, 2019b))

When reading about building apps for release, you will come across two different concepts: an **APK** and a **bundle**. APK stands for Android App Package. It is a compressed file that contains all the compiled code and resources that is needed to run the app. (Margain, 2020) If you have built your apps to install on your own phone, this would be the format that you used.

More recently, the Google Play Store has introduced the concept of a bundle. “An Android App Bundle is a publishing format that includes all your app’s compiled codes and resources and defers APK generation and signing to Google Play.” (Android Open Source Project, 2020d)

Important: Since August 2021, the Google Play Store requires new apps to be published using a bundle. (Android Open Source Project, 2019b) The steps for creating an APK that are described below, are still useful if you want to load the app directly onto a phone. But for the Play Store, start using bundles now.

2.1 Android Studio Documentation

The Android Studio User Guide contains great information about the process to release an app. Read the following pages in order, for all the information that you need about generating an APK or a bundle.

Android Open Source Project, 2019b. *Publish your app.* [Online] Available at: https://developer.android.com/studio/publish [Accessed 31 July 2023].

Android Open Source Project, 2019. *Prepare for release.* [Online] Available at: https://developer.android.com/studio/publish/preparing [Accessed 31 July 2023].

Android Open Source Project, 2019c. *Version your app.* [Online] Available at: https://developer.android.com/studio/publish/ versioning [Accessed 31 July 2023].

Android Open Source Project, 2019d. *Sign your app.* [Online] Available at: https://developer.android.com/studio/publish/versioning [Accessed 31 July 2023].

2.2 Generating an APK

To generate an APK using Android Studio:

1. On the main menu click **Build**, and then click **Generate Signed Bundle/APK**.

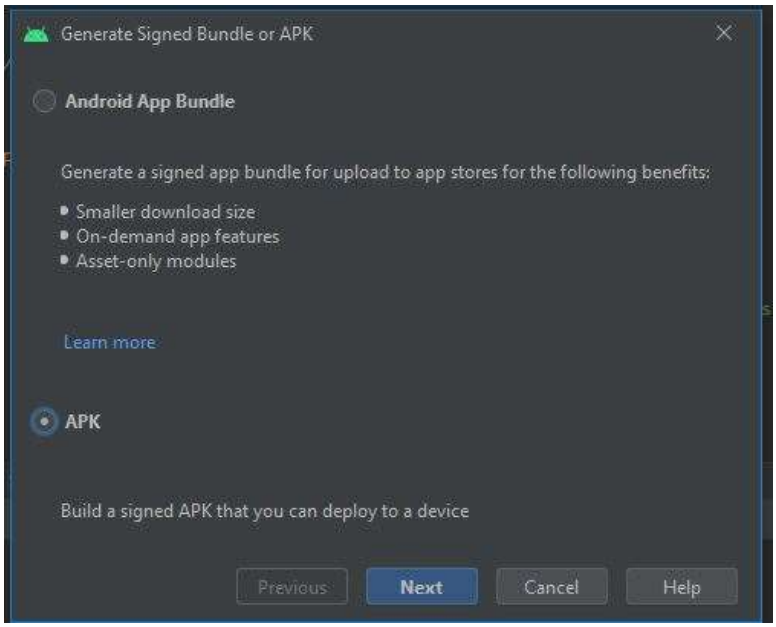


Figure 64. Generate Bundle or APK

2. Choose **APK** and click **Next**.

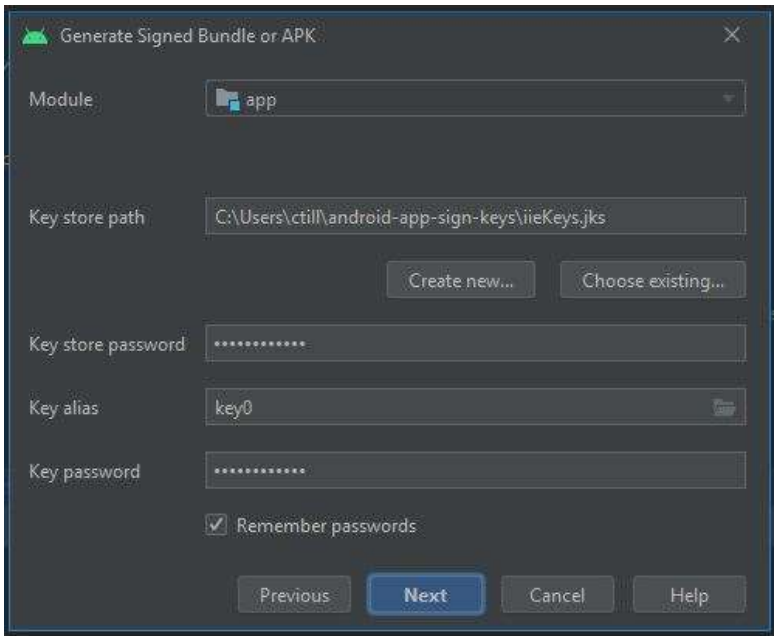


Figure 65. Choose Signing Key

3. Specify which **key store** to use, and the relevant credentials for the keystore. Click **Next**.

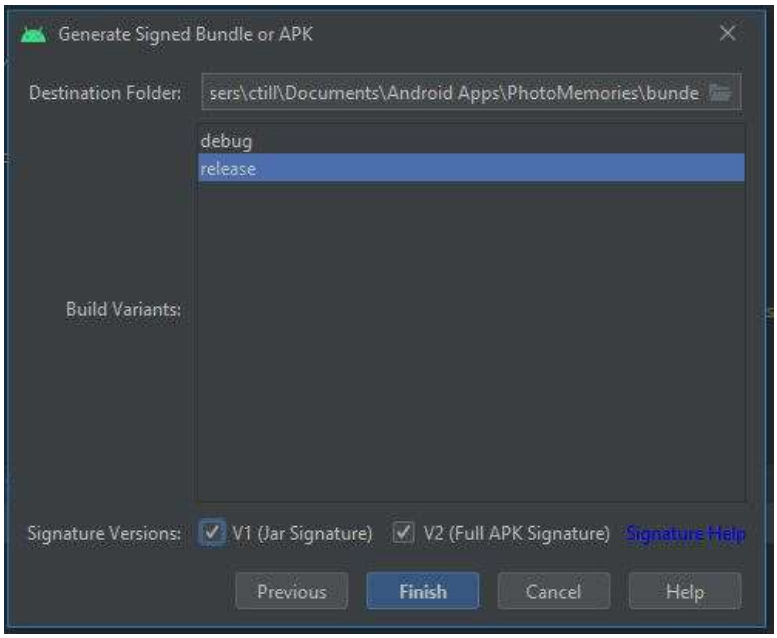


Figure 66. Signature Versions

4. Choose both **V1** and **V2** signature versions and click **Finish**.

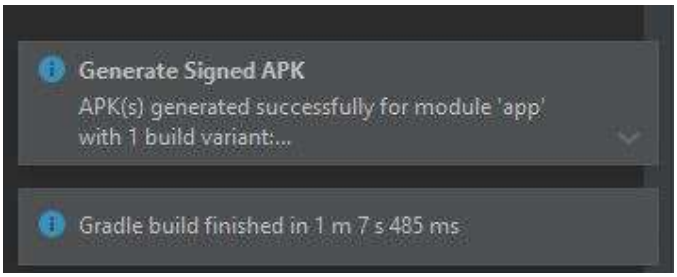


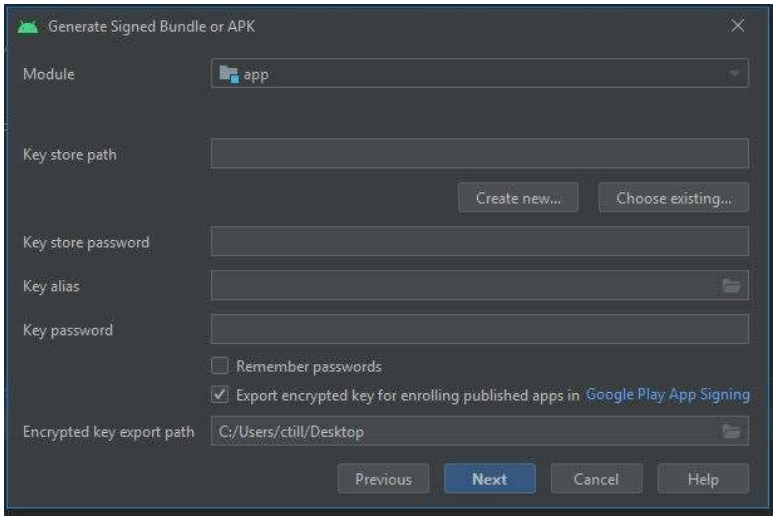
Figure 67. Gradle Build Complete

5. A Gradle build process will be started to generate the APK. A notification will let you know when it is done.

2.3 Generating a Bundle

To generate a bundle from Android Studio:

1. On the main menu click **Build**, and then click **Generate Signed Bundle / APK**.
2. Choose to generate an **Android App Bundle** and click **Next**.



3. On the next page, click the **Create new** button under the key store path to create a new keystore.

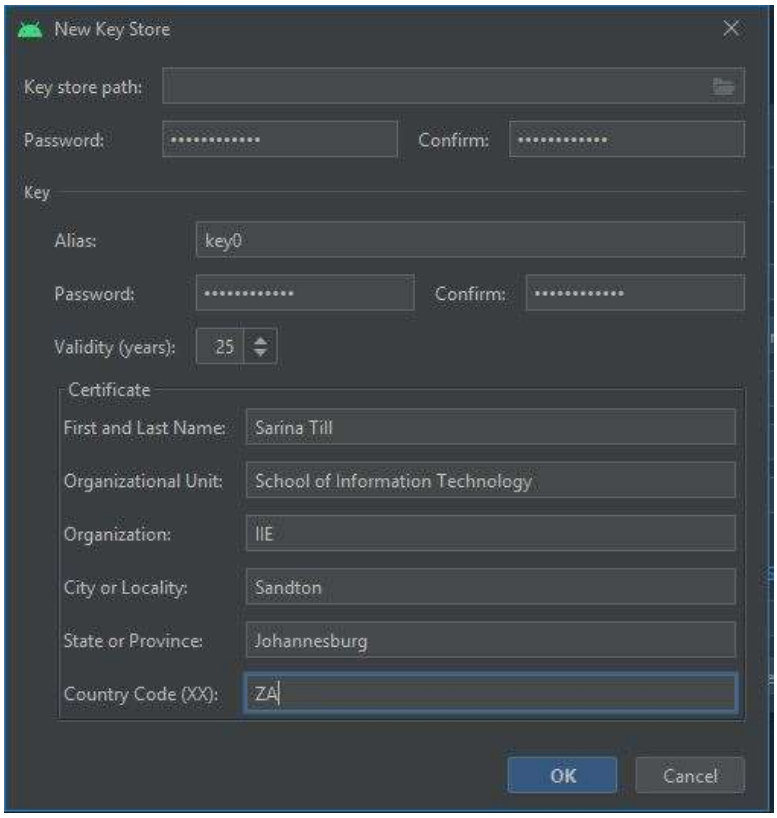


Figure 68. New Key Store Properties

4. Enter the required values and click **OK**.
5. Choose **where to save** the keystore.

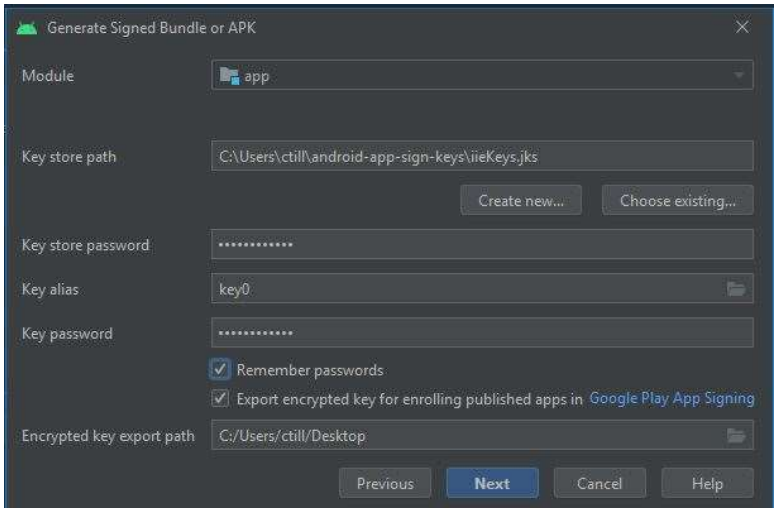


Figure 69. Key Store Values

6. The key store values are now populated. Click to select **Remember passwords**.
7. Click **Next**.

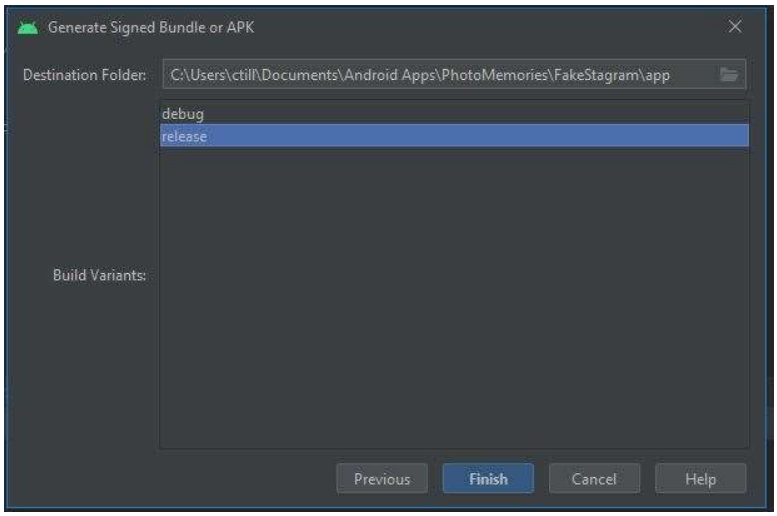


Figure 70. Select Build Variants

8. Choose the **release** variant and click **Finish**.
9. Choose the **location** where the bundle will get created.

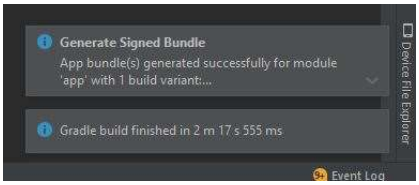


Figure 71. Build Complete Notification

10. A Gradle process will be started, and a message will notify you when the build is complete.

3 Deploying to the App Store

The Google Play Console can be accessed here: https://play.google.com/console/about/ https://developer.android.com/distribute/console[Accessed 31 July 2023].

You do need an account to access this. **Get in contact with your lecturer or navigator to get access to the IIE's account.** That way you don't need to pay for your own account.

To publish the Photo Memories app to the Play Store:

1. Log into the Play Console.
2. When you log into the Console, you first see the landing page with all the apps that have been published by the account.

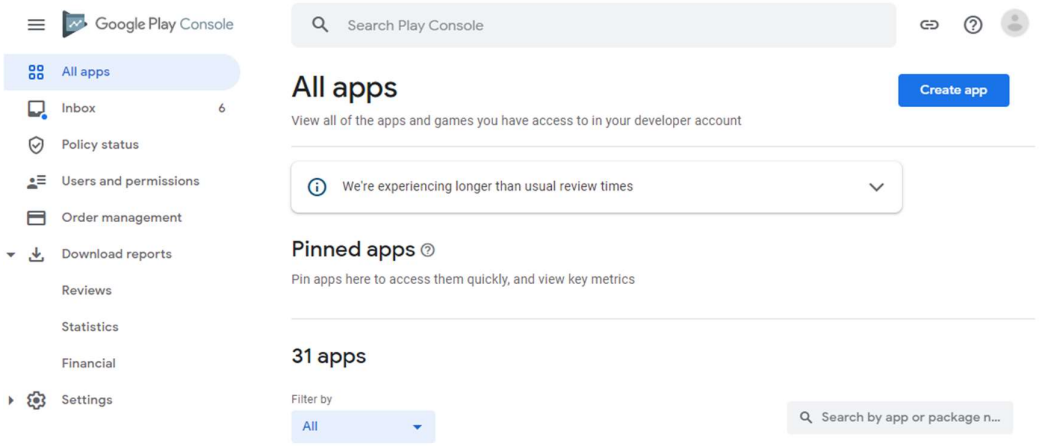


Figure 72. Google Play Console Landing Page

3. Click **Create app**.

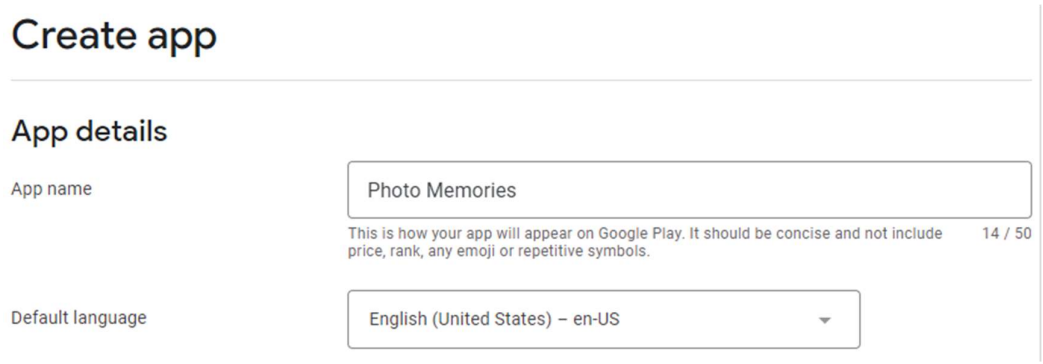


Figure 73. Default Language and App Title

4. Enter the **App Name** of the app.
5. Specify the **default language** of the app – English in our case.
6. Choose whether this is an **app** or a **game** – app in our case (see Figure 74).
7. Choose whether the app is **paid** or **free**. We are choosing Free for now.
8. Accept the two **declarations**. Make sure to read and understand these agreements before accepting!

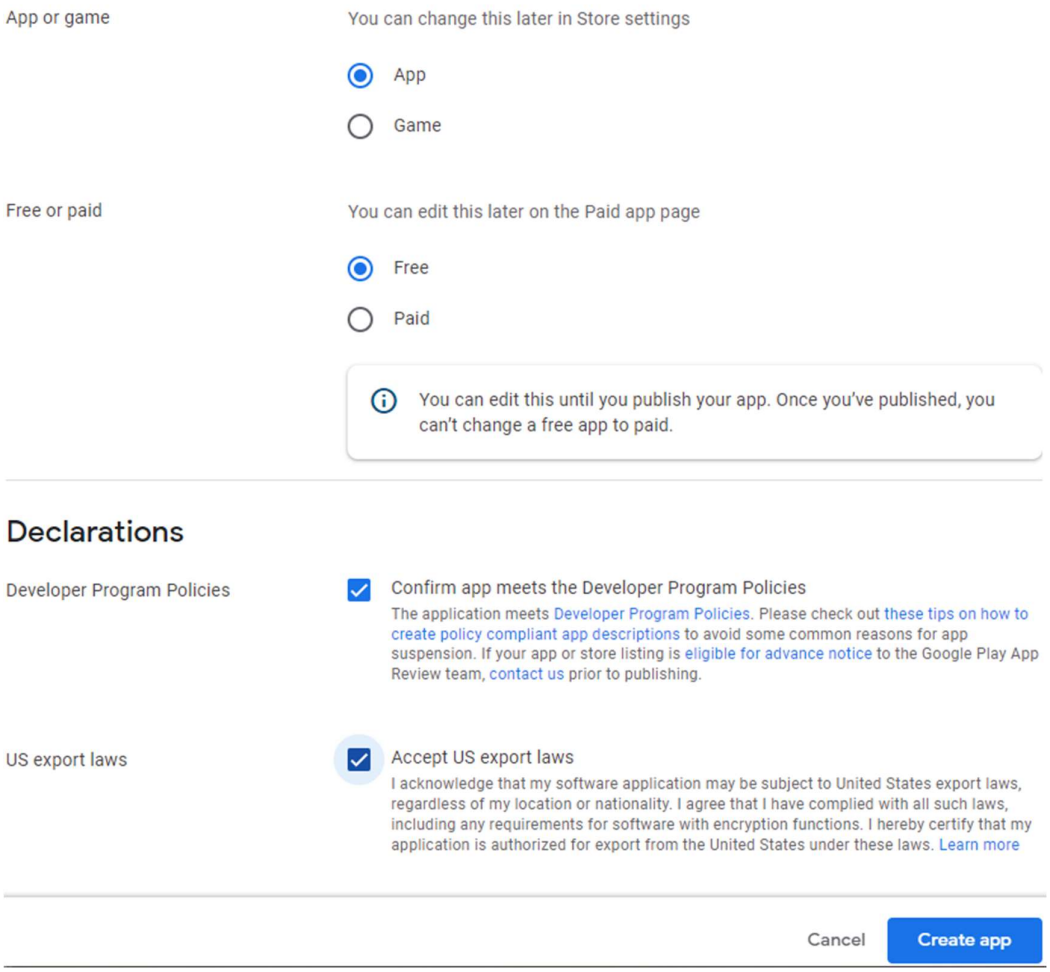


Figure 74. More App Settings

9. Click **Create app**.

Now that the app has been created, and the basic settings specified, the dashboard for the app (see Figure 75) shows which steps need to be taken next.

You can skip the **Start testing now** section, unless you are planning to release the app early to testers. For our purposes that leaves two aspects:

- **Set up your app**: specifying properties such as the content rating for the app, which will not change often during the lifecycle of an app.
- **Release your app**: a process that would need to be followed every time you release a new version of the app.

Click **View tasks** to show the detailed list of all the steps you need to take.

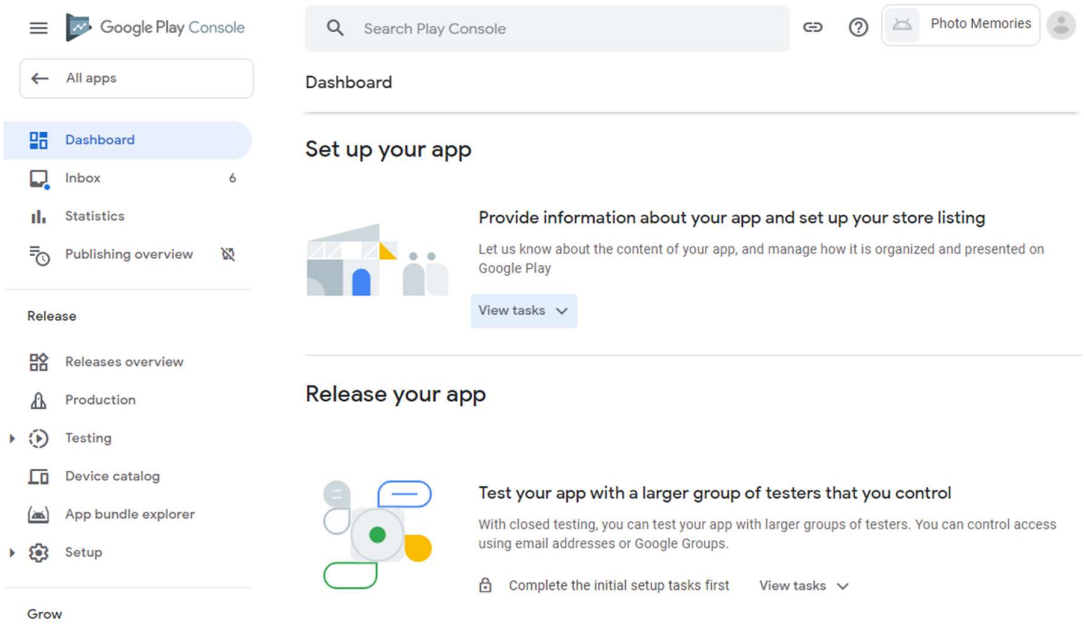


Figure 75. Dashboard Showing Steps

Go ahead and configure all the parts of the **Set up your app** section. The steps will take you through the process. As you complete it, they will be marked as complete.

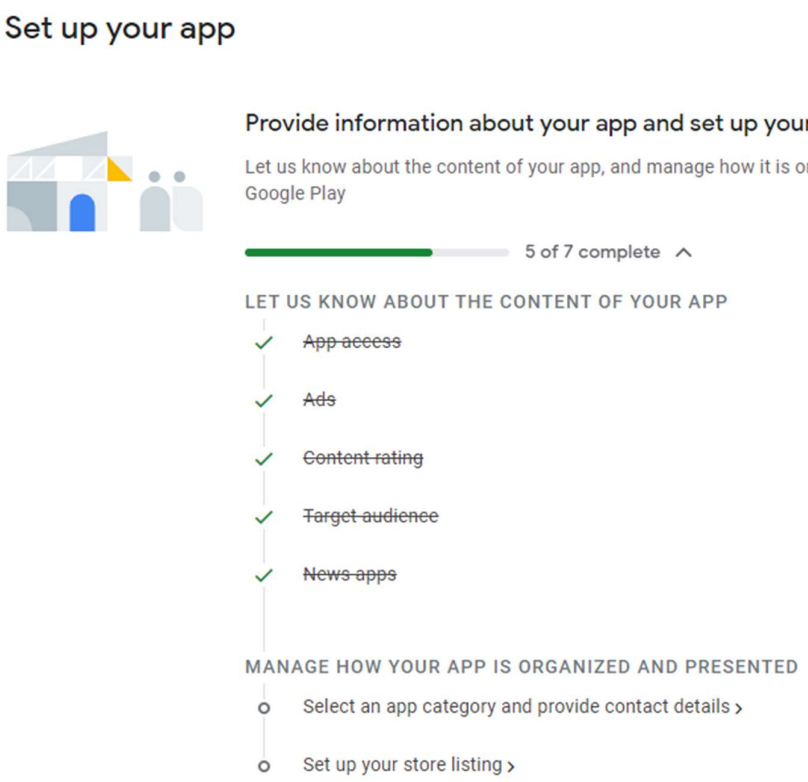


Figure 76. Dashboard Showing Steps Done

Note that you will need the following graphics:

- App icon – 512 x 512 px
- Feature graphic – 1024 x 500 px
- At least two phone or tablet screenshots – between 320 and 3840 px on each side

Once all of that is set up, we can now create a release. This can be accessed from the dashboard (see Figure 77) or from the Production page under the Release menu section (see Figure 78).

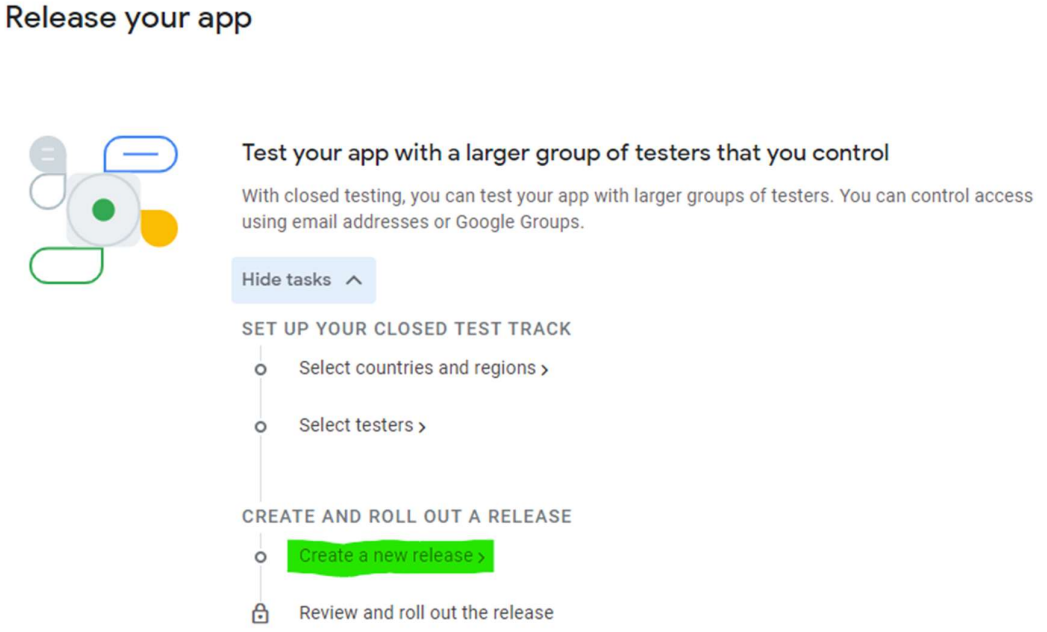


Figure 77. Create a new release – from Dashboard

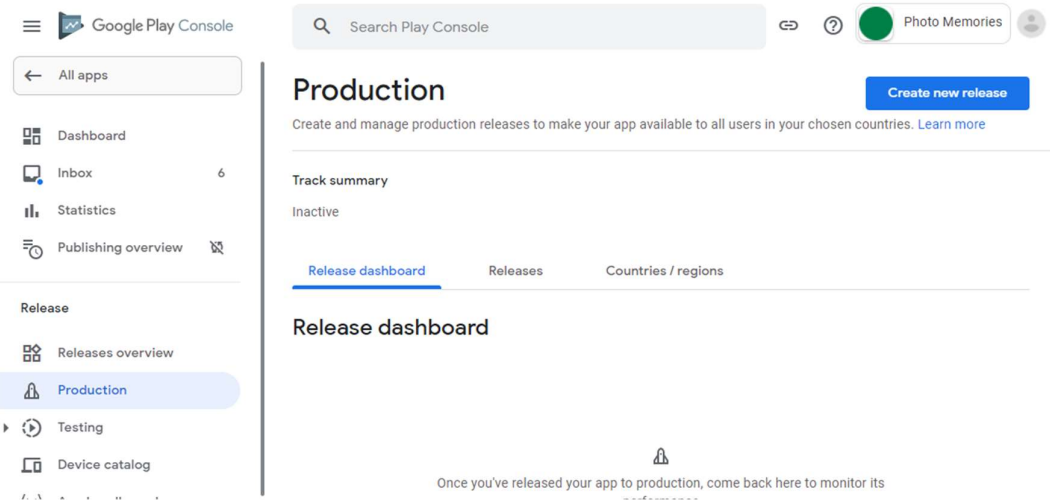


Figure 78. Create a new release – from Release > Production

Upload the bundle, give the release a name, and enter the release notes for this release. Then click **Review release** to finish the release process.

Apps are reviewed before they are published on the Play Store. This can take a few days, so publish early!

4 Track Analytics and Usage

Once your app is published on the Google Play Store, you can track several different metrics. For example, you can view how many new users installed your app. Or what your ratings were over time. You can even see which version of Android users have on their phones.

Read this page for more information about how to use the analytics on the Play Store Console:

Google, 2020. *View app statistics*. [Online] Available at: https://support.google.com/googleplay/android-developer/answer/139628?co=GENIE.Platform%3DDesktop&hl=en [Accessed 31 July 2023].

5 Recommended Digital Engagement

[YouTube] How To Upload Android App on Google Play Store https://www.youtube.com/watch?v=8v0r_6mYgF8 [Accessed 31 July 2023].

6 Recommended Additional Reading

Android Open Source Project, 2019b. *Publish your app.* [Online] Available at: https://developer.android.com/studio/publish [Accessed 31 July 2023].

Android Open Source Project, 2019. *Prepare for release.* [Online] Available at: https://developer.android.com/studio/publish/preparing [Accessed 31 July 2023].

Android Open Source Project, 2019c. *Version your app.* [Online] Available at: https://developer.android.com/studio/publish/ versioning [Accessed 31 July 2023].

Android Open Source Project, 2019d. *Sign your app*. [Online] Available at: https://developer.android.com/studio/publish/versioning [Accessed 21 June 2021].

Grinsted, T., 2020. *Introducing the new Google Play Console beta*. [Online] Available at: https://android-developers.googleblog.com/2020/06/introducing-new-google-play-console-beta.html [Accessed 31 July 2023].

Google, 2020. *View app statistics*. [Online] Available at: https://support.google.com/googleplay/android-developer/answer/139628?co=GENIE.Platform%3DDesktop&hl=en [Accessed 31 July 2023].

7 Activities

Complete the activities on Learn.

Bibliography

AccuWeather, Inc., 2020. *AccuWeather API Reference.* [Online] Available at: https://developer.accuweather.com/apis [Accessed 31 July 2023].

AccuWeather, Inc., 2020b. *Forecast API.* [Online] Available at: https://developer.accuweather.com/accuweather-forecast-api/apis [Accessed 31 July 2023].

AccuWeather, Inc., 2020b. *Packages.* [Online] Available at: https://developer.accuweather.com/packages [Accessed 31 July 2023].

AccuWeather, Inc., 2020c. *API Flow Diagram.* [Online] Available at: https://developer.accuweather.com/api-flow-diagram [Accessed 31 July 2023].

AccuWeather, Inc., 2020d. *Current Conditions API.* [Online] Available at: https://developer.accuweather.com/accuweather-current-conditions-api/apis [Accessed 31 July 2023].

AccuWeather, Inc., 2023a. *Packages.* [Online] Available at: https://developer.accuweather.com/packages [Accessed 131 July 2023].

AccuWeather, Inc., n.d.. *Best Practices.* [Online] Available at: http://apidev.accuweather.com/developers/best-practices [Accessed 31 July 2023].

Android Open Source Project, 2019b. *Publish your app.* [Online] Available at: https://developer.android.com/studio/publish [Accessed 31 July 2023].

Android Open Source Project, 2019. *Prepare for release.* [Online] Available at: https://developer.android.com/studio/publish/preparing [Accessed 31 July 2023].

Android Open Source Project, 2020. *AsyncTask.* [Online] Available at: https://developer.android.com/reference/android/os/AsyncTask [Accessed 31 July 2023].

Android Open Source Project, 2020b. *Processes and threads overview.* [Online] Available at: https://developer.android.com/guide/components/processes-and-threads [Accessed 31 July 2023].

Android Open Source Project, 2020c. *Fragments.* [Online] Available at: https://developer.android.com/guide/components/fragments [Accessed 31 July 2023].

Android Open Source Project, 2020d. *About Android App Bundles.* [Online] Available at: https://developer.android.com/guide/app-bundle [Accessed 31 July 2023].

Android Open Source Project, 2020e. *SQLiteOpenHelper.* [Online] Available at: https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper [Accessed 31 July 2023].

Android Open Source Project, 2020f. *Set the application ID.* [Online] Available at: https://developer.android.com/studio/build/application-id [Accessed 31 July 2023].

Bialas, M., 2020. *The 30 Best Android Libraries and Projects of 2019.* [Online] Available at: https://medium.com/better-programming/30-best-android-libraries-and-projects-of-2019-a1e35124f110 [Accessed 31 July 2023].

CodePath Android Cliffnotes, n.d.. *Storing Secret Keys in Android.* [Online] Available at: https://guides.codepath.com/android/Storing-Secret-Keys-in-Android [Accessed 31 July 2023].

Computer Hope, 2017. *Software library.* [Online] Available at: https://www.computerhope.com/jargon/s/softlibr.htm [Accessed31 July 2023].

Droid By Me, 2018. *Get Current location using FusedLocationProviderClient in Android.* [Online] Available at: https://medium.com/@droidbyme/get-current-location-using-fusedlocationproviderclient-in-android-cb7ebf5ab88e [Accessed 31 July 2023].

Facebook, 2020. *Facebook SDK for Android.* [Online] Available at: https://developers.facebook.com/docs/android/ [Accessed 31 July 2023].

Freeman, J., 2019. *What is JSON? A better format for data exchange.* [Online] Available at: https://www.infoworld.com/article/3222851/what-is-json-a-better-format-for-data-exchange.html [Accessed 31 July 2023].

Google, 2020. *Secure your site with HTTPS.* [Online] Available at: https://support.google.com/webmasters/answer/6073543?hl=en [Accessed 31 July 2023].

Gson, 2020. *Gson readme.* [Online] Available at: https://github.com/google/gson [Accessed 31 July 2023].

java2s.com, n.d.. *Specifying a Retention Policy : Annotations Create « Language « Java Tutorial.* [Online] Available at: http://www.java2s.com/Tutorial/Java/0020__Language/SpecifyingaRetentionPolicy.htm [Accessed31 July 2023].

Karnok, D., 2017. *ReactiveX / RxJava.* [Online] Available at: https://github.com/ReactiveX/RxJava/wiki [Accessed 31 July 2023].

Kotlin Foundation, 2023. *Annotations.* [Online] Available at: https://kotlinlang.org/docs/annotations.html [Accessed 31 July 2023].

mancj, 2020. *Material SearchBar Android.* [Online] Available at: https://github.com/mancj/MaterialSearchBar [Accessed 31 July 2023].

Margain, E., 2020. *Android App Bundles vs. APKs.* [Online] Available at: https://medium.com/better-programming/android-app-bundles-vs-apks-8b0306b38436 [Accessed 31 July 2023].

Monus, A., 2020. *SOAP vs REST vs JSON - a 2020 comparison.* [Online] Available at: https://raygun.com/blog/soap-vs-rest-vs-json/ [Accessed 31 July 2023].

RESTfulAPI.net, n.d.. *What is REST.* [Online] Available at: https://restfulapi.net/ [Accessed 31 July 2023].

RIP Tutorial, n.d.. *Android: Open a URL in a browser.* [Online] Available at: https://riptutorial.com/android/example/549/open-a-url-in-a-browser [Accessed 31 July 2023].

Rouse, M., 2019. *HTTP (Hypertext Transfer Protocol).* [Online] Available at: https://whatis.techtarget.com/definition/HTTP-Hypertext-Transfer-Protocol [Accessed 31 July 2023].

Rouse, M., 2020. *RESTful API (REST API).* [Online] Available at: https://searchapparchitecture.techtarget.com/definition/RESTful-API [Accessed 31 July 2023].

Ruzicka, V., 2017. *Avoid Utility Classes.* [Online] Available at: https://www.vojtechruzicka.com/avoid-utility-classes/ [Accessed 31 July 2023].

Sandoval, K., 2016. *What is the Difference Between an API and an SDK?.* [Online] Available at: https://nordicapis.com/what-is-the-difference-between-an-api-and-an-sdk/ [Accessed 31 July 2023].

Seobility, n.d.. *https://www.seobility.net/en/wiki/HTTP_headers.* [Online] Available at: https://www.seobility.net/en/wiki/HTTP_headers [Accessed 31 July 2023].

sqlite.org, n.d.b. *Appropriate Uses For SQLite.* [Online] Available at: https://www.sqlite.org/whentouse.html [Accessed 31 July 2023].

sqlite.org, n.d.. *SQLite Home Page.* [Online] Available at: https://www.sqlite.org/index.html [Accessed 31 July 2023].

Square, Inc., n.d.. *Retrofit: A type-safe HTTP client for Android and Java.* [Online] Available at: https://square.github.io/retrofit/ [Accessed 31 July 2023].

statista.com, 2020. *Number of available applications in the Google Play Store from December 2009 to March 2020.* [Online] Available at: https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store [Accessed 31 July 2023].

Tagliaferri, L., 2016. *An Introduction to JSON.* [Online] Available at: https://www.digitalocean.com/community/tutorials/an-introduction-to-json [Accessed 31 July 2023].

TutorialsPoint.com, 2020. *RxJava - Using CompositeDisposable.* [Online] Available at: https://www.tutorialspoint.com/rxjava/rxjava_compositedisposable.htm [Accessed 31 July 2023].

Intellectual Property

Plagiarism occurs in a variety of forms. Ultimately though, it refers to the use of the words, ideas or images of another person without acknowledging the source using the required conventions. The IIE publishes a Quick Reference Guide that provides more detailed guidance, but a brief description of plagiarism and referencing is included below for your reference. It is vital that you are familiar with this information and the Intellectual Integrity Policy before attempting any assignments.

Introduction to Referencing and Plagiarism

What is 'Plagiarism'?

'Plagiarism' is the act of taking someone's words or ideas and presenting them as your own.

What is 'Referencing'?

'Referencing' is the act of citing or giving credit to the authors of any work that you have referred to or consulted. A 'reference' then refers to a citation (a credit) or the actual information from a publication that is referred to.

Referencing is the acknowledgment of any work that is not your own, but is used by you in an academic document. It is simply a way of giving credit to and acknowledging the ideas and words of others.

When writing assignments, students are required to acknowledge the work, words or ideas of others through the technique of referencing. Referencing occurs in the text at the place where the work of others is being cited, and at the end of the document, in the bibliography.

The bibliography is a list of all the work (published and unpublished) that a writer has read in the course of preparing a piece of writing. This includes items that are not directly cited in the work.

A reference is required when you:

- Quote directly: when you use the exact words as they appear in the source;
- Copy directly: when you copy data, figures, tables, images, music, videos or frameworks;
- Summarise: when you write a short account of what is in the source;
- Paraphrase: when you state the work, words and ideas of someone else in your own words.

It is standard practice in the academic world to recognise and respect the ownership of ideas, known as intellectual property, through good referencing techniques. However, there are other reasons why referencing is useful.

Good Reasons for Referencing

It is good academic practice to reference because:

- It enhances the quality of your writing;
- It demonstrates the scope, depth and breadth of your research;
- It gives structure and strength to the aims of your article or paper;
- It endorses your arguments;
- It allows readers to access source documents relating to your work, quickly and easily.

Sources

The following would count as 'sources':

- Books,
- Chapters from books,
- Encyclopaedias,
- Articles,
- Journals,
- Magazines,
- Periodicals,
- Newspaper articles,
- Items from the Internet (images, videos, etc.),
- Pictures,
- Unpublished notes, articles, papers, books, manuscripts, dissertations, theses, etc.,
- Diagrams,
- Videos,
- Films,
- Music,
- Works of fiction (novels, short stories or poetry).

What You Need to Document from the Hard Copy Source You are Using

(Not every detail will be applicable in every case. However, the following lists provide a guide to what information is needed.)

You need to acknowledge:

- The words or work of the author(s),
- The author(s)'s or editor(s)'s full names,
- If your source is a group/ organisation/ body, you need all the details,
- Name of the journal, periodical, magazine, book, etc.,
- Edition,
- Publisher's name,
- Place of publication (i.e. the city of publication),
- Year of publication,
- Volume number,
- Issue number,
- Page numbers.

What You Need to Document if you are Citing Electronic Sources

- Author(s)'s/ editor(s)'s name,
- Title of the page,
- Title of the site,
- Copyright date, or the date that the page was last updated,
- Full Internet address of page(s),
- Date you accessed/ viewed the source,
- Any other relevant information pertaining to the web page or website.

Referencing Systems

There are a number of referencing systems in use and each has its own consistent rules. While these may differ from system-to-system, the referencing system followed needs to be used consistently, throughout the text. Different referencing systems cannot be mixed in the same piece of work!

A detailed guide to referencing, entitled Referencing and Plagiarism Guide is available from your library. Please refer to it if you require further assistance.

When is Referencing Not Necessary?

This is a difficult question to answer – usually when something is 'common knowledge'. However, it is not always clear what 'common knowledge' is.

Examples of 'common knowledge' are:

- Nelson Mandela was released from prison in 1990;
- The world's largest diamond was found in South Africa;
- South Africa is divided into nine (9) provinces;
- The lion is also known as 'The King of the Jungle'.
- $E = mc^2$
- The sky is blue.

Usually, all of the above examples would not be referenced. The equation $E = mc^2$ is Einstein's famous equation for calculations of total energy and has become so familiar that it is not referenced to Einstein.

Sometimes what we think is 'common knowledge', is not. For example, the above statement about the sky being blue is only partly true. The light from the sun looks white, but it is actually made up of all the colours of the rainbow. Sunlight reaches the Earth's atmosphere and is scattered in all directions by all the gases and particles in the air. The smallest particles are by coincidence the same length as the wavelength of blue light. Blue is scattered more than the other colours because it travels as shorter, smaller waves. It is not entirely accurate then to claim that the sky is blue. It is thus generally safer to always check your facts and try to find a reputable source for your claim.

Important Plagiarism Reminders

The IIE respects the intellectual property of other people and requires its students to be familiar with the necessary referencing conventions. Please ensure that you seek assistance in this regard before submitting work if you are uncertain.

If you fail to acknowledge the work or ideas of others or do so inadequately this will be handled in terms of the Intellectual Integrity Policy (available in the library) and/ or the Student Code of Conduct – depending on whether or not plagiarism and/ or cheating (passing off the work of other people as your own by copying the work of other students or copying off the Internet or from another source) is suspected.

Your campus offers individual and group training on referencing conventions – please speak to your librarian or ADC/ Campus Co-Navigator in this regard.

Reiteration of the Declaration you have signed:

1. I have been informed about the seriousness of acts of plagiarism.
2. I understand what plagiarism is.
3. I am aware that The Independent Institute of Education (IIE) has a policy regarding plagiarism and that it does not accept acts of plagiarism.
4. I am aware that the Intellectual Integrity Policy and the Student Code of Conduct prescribe the consequences of plagiarism.

5. I am aware that referencing guides are available in my student handbook or equivalent and in the library and that following them is a requirement for successful completion of my programme.
6. I am aware that should I require support or assistance in using referencing guides to avoid plagiarism I may speak to the lecturers, the librarian or the campus ADC/ Campus Co-Navigator.
7. I am aware of the consequences of plagiarism.

Please ask for assistance prior to submitting work if you are at all unsure.

