

MODUL 5

Essential Libraries: Retrofit, Moshi, and Glide

THEME DESCRIPTION

This module covers the steps needed to present app users with dynamic content fetched from remote servers. Students will be introduced to the different libraries required to retrieve and handle this dynamic data.

WEEKLY LEARNING OUTCOME (SUB-LEARNING OUTCOME)

Students will be able to fetch data from a network endpoint using Retrofit, parse JSON payloads into Kotlin data objects using Moshi, and load images into ImageView using Glide.

TOOLS/SOFTWARE USED

- Android Studio

CONCEPTS

API

APIs (Application Programming Interface) are mechanisms that enable two software components to communicate with each other using a set of definitions and protocols. We will be using the **Cat API** for this tutorial.

Retrofit

Retrofit is a REST Client for Java and Android allowing to retrieve and upload JSON (or other structured data) via a REST based web service. We will be using this library to **fetch data** from our **API**.

Moshi

Moshi is a modern JSON library for Android, Java and Kotlin. It makes it easy to parse JSON objects into Java and Kotlin objects. We will be using this library to **parse** our **JSON** fetched data to **kotlin objects**.

Glide

Glide is a fast and efficient image loading library for Android focused on smooth scrolling. Glide offers an easy to use API, a performant and extensible resource

decoding pipeline and automatic resource pooling. We will be using this library to **load** our fetched **images** into our application.

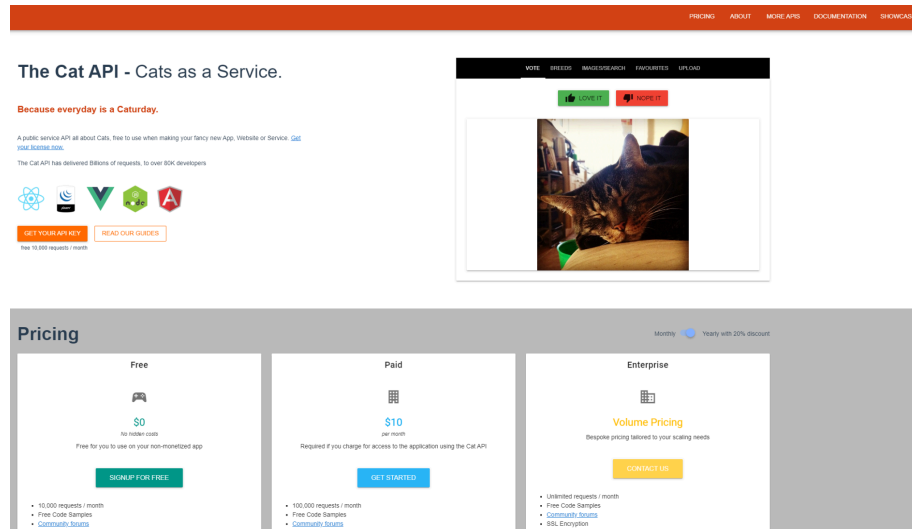
PRACTICAL STEPS

Part 1 - Reading Data from an API

1. Open Android Studio and click **New Project**.
2. Choose the **Empty Views Activity** to start with.
3. Name your project "**LAB_WEEK_05**".
4. Set the minimum SDK to "**API 24: Android 7.0 (Nougat)**".
5. Click **Finish**, and let your android application build itself.
6. In this part, we will be focusing on how we can **read data from an API** in Android using **Retrofit**. In order to use **Retrofit**, we need to add the necessary libraries to our application. Add the code below to the **dependencies** section in **build.gradle.kts** (**Module :app**) and **sync** your gradle files.

```
implementation("com.squareup.retrofit2:retrofit:2.9.0")
implementation("com.squareup.retrofit2:converter-scalars:2.9.0")
```

7. Next, because we're fetching data from an **API (RESTful)**, for this tutorial we're gonna use the **Cat API** provided by the following URL "**https://thecatapi.com/**".



The Cat API - Cats as a Service.

Because everyday is a Saturday.

A public service API all about Cats, free to use when making your fancy new App, Website or Service. [Get your API key here.](#)

The Cat API has delivered billions of requests, to over 80K developers.

[GET YOUR API KEY](#) [READ OUR GUIDES](#)

Free 10,000 requests / month

Pricing

Monthly ☒ Yearly with 20% discount

Free	Paid	Enterprise
<p>\$0</p> <p>no hidden costs</p> <p>Free for you to use on your non-commercial app</p> <p>SIGN UP FOR FREE</p> <ul style="list-style-type: none"> 10,000 requests / month Free Code Samples Community Support 99% Uptime 	<p>\$10</p> <p>per month</p> <p>Required if you charge for access to the application using the Cat API</p> <p>GET STARTED</p> <ul style="list-style-type: none"> 100,000 requests / month Free Code Samples Community Support 99% Uptime 	<p>Volume Pricing</p> <p>Bespoke pricing tailored to your scaling needs</p> <p>CONTACT US</p> <ul style="list-style-type: none"> Unlimited requests / month Free Code Samples Community Support SSL Encryption Custom Plans / Issues / Quotes

8. As you may know, most **API** need an **API KEY** for us to be able to create a data request. But for this **Cat API**, you don't need one for any data requests that are less than 10 data. To keep things simple, in this tutorial we will only be requesting **1 data**, therefore an **API KEY is not needed**.
9. Our initial preparations are done. Now we need to set the **permission** for our app to use the **internet**. Update your **AndroidManifest.xml** to the code below.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

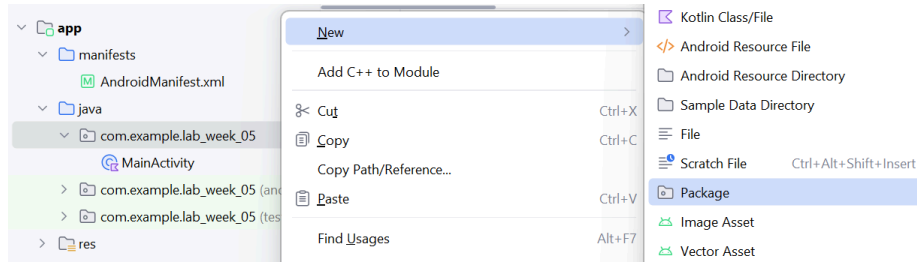
    <uses-permission
        android:name="android.permission.INTERNET" />

    <application ...>
        ...
    </application>
</manifest>
```

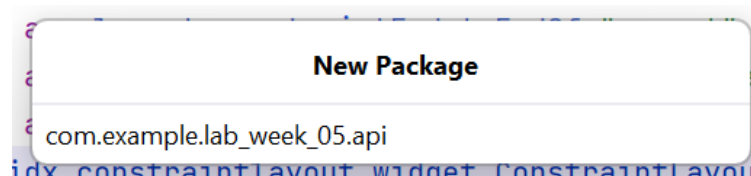
10. Our initial setup is done, now let's update the layout. Add an **ID** called "**api_response**" to the **TextView** to display the response of the API. Update your **activity_main.xml** to the code below.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/api_response"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

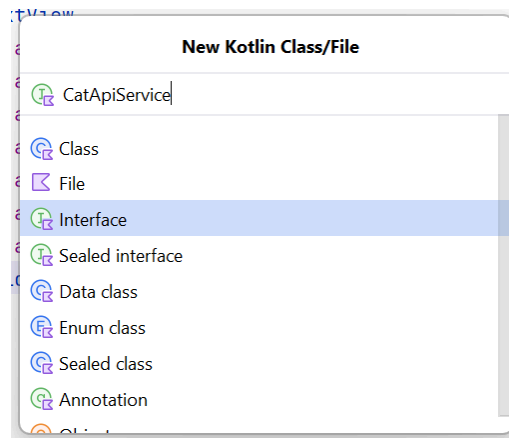
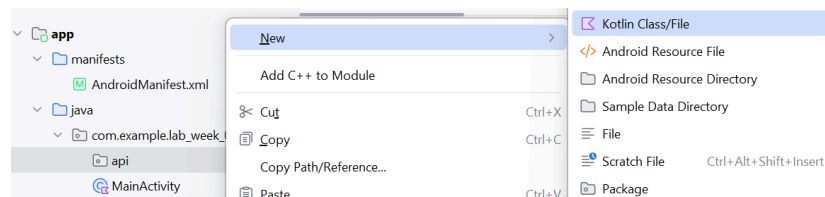
11. Our layout's done, now let's add and initialize **Retrofit** into our application. First, we need to define a **Service Interface** to set our **Endpoints** for the **API Calls**. Let's define this service in a different **Package**. **Right click** your **Package** (com.example.lab_week_05) > **New** > **Package**.



12. Name the new package “**com.example.lab_week_05.api**”.



13. Make a new **Kotlin File** in the new package. **Right click the API folder > New > Kotlin Class/File**. Name the **Kotlin File “CatApiService.kt”** and set the kind to **Interface**.



14. Update the **CatApiService.kt** to the code below.

```
import retrofit2.Call
import retrofit2.http.GET
import retrofit2.http.Query
```

```
interface CatApiService {
    @GET("images/search")
    fun searchImages(
        @Query("limit") limit: Int,
        @Query("size") format: String
    ) : Call<String>
}
```

15. From the code above, here are some important notes:

- `@GET("images/search")` defines an **endpoint** to “images/search” with the **GET** method.
- `@Query("limit") limit: Int` defines a **parameter** to send with our query when the **searchImages** function is called.

16. Next, let’s call our service to get the necessary data. Normally, we would make use of the **MVVM (Model - View - ViewModel)** concept to put in our **Retrofit Code** as it is more efficient and a lot more scalable. But unfortunately you will learn the **MVVM** architecture in the later **Week 13** module. As for now, we will be **instantiating** the **Retrofit Class** in our **Activity** (which is not scalable at all). You are welcome to use this in your upcoming **mid term** test, but don’t ever use it in a **production app** as it is **NOT** a good practice.

17. First, instantiate the **Retrofit** in your **MainActivity.kt**. Add the code below above your **onCreate** callback.

```
private val retrofit by lazy{
    Retrofit.Builder()
        .baseUrl("https://api.thecatapi.com/v1/")
        .addConverterFactory(ScalarsConverterFactory.create())
        .build()
}
```

18. You may notice the `.addConverterFactory(ScalarsConverterFactory.create())` in the code above. The **Converter Factory** basically acts as a **parser** for our response (In this case, converts the response to a **string**).

19. Next, instantiate the **CatApiService** in your **MainActivity.kt**. Add the code below after the instantiation of your **Retrofit Class**.

```
private val retrofit by lazy{
    ...
}

private val catApiService by lazy{
    retrofit.create(CatApiService::class.java)
```

```
}
```

20. Next, Get the reference of our **TextView** and put it in a variable. Add the code below after the instantiation of your **CatApiService**.

```
private val retrofit by lazy{
    ...
}

private val catApiService by lazy{
    ...
}

private val apiResponseView: TextView by lazy{
    findViewById(R.id.api_response)
}
```

21. Our instantiations are done, now let's create the function to be called on the creation of the activity. Update your **MainActivity.kt** to the code below.

```
private val retrofit by lazy{
    ...
}

private val catApiService by lazy{
    ...
}

private val apiResponseView: TextView by lazy{
    ...
}

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    getCatImageResponse()
}

private fun getCatImageResponse() {
    val call = catApiService.searchImages(1, "full")
    call.enqueue(object: Callback<String> {
        override fun onFailure(call: Call<String>, t: Throwable) {
            Log.e(MAIN_ACTIVITY, "Failed to get response", t)
        }
    })
}
```

```
        override fun onResponse(call: Call<String>, response:
Response<String>) {
            if(response.isSuccessful){
                apiResponseView.text = response.body()
            }
            else{
                Log.e(MAIN_ACTIVITY, "Failed to get response\n" +
                    response.errorBody()?.string().orEmpty()
                )
            }
        }
    })
}

companion object{
    const val MAIN_ACTIVITY = "MAIN_ACTIVITY"
}
```

22. From the code above, here are some important notes:

- `call.enqueue()` calls the specified query **asynchronously**.
- `override fun onFailure(call: Call<String>, t: Throwable)` is a callback called when the response **fails**.
- `override fun onResponse(call: Call<String>, response: Response<String>)` is a callback called when the response **succeeds**.

Commit to GITHUB at this point.
Commit Message: "Commit No. 1 - Retrofit Setup and Fetch API JSON Response"

23. Run your application, and you should get the **JSON Response** as a **String**.

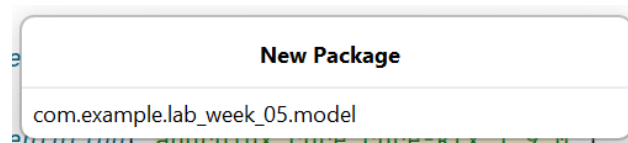


Part 2 - Extracting the Image URL from the API Response

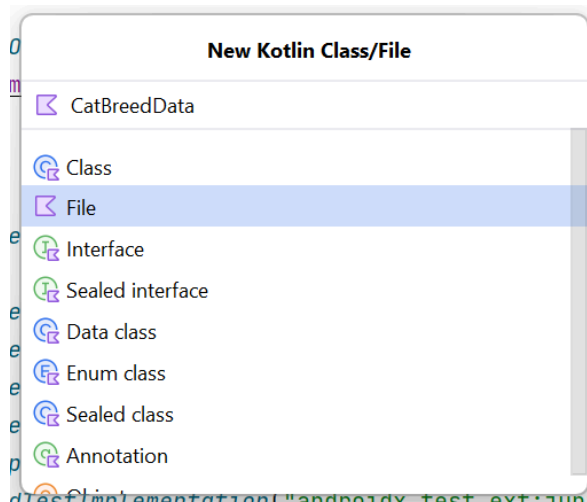
1. Continue your **“LAB_WEEK_05”** project.
2. In this part, we will be focusing on how you can **Parse the JSON Response that you get from an API Call** in Android using the **Moshi** library. In order to use **Moshi**, we need to add the necessary library to our application. Add the code below to the **dependencies** section in **build.gradle.kts (Module :app)** and **sync** your gradle files.

```
implementation("com.squareup.retrofit2:converter-moshi:2.9.0")
```

3. Now in order for our **Moshi** to know what type of data it should convert the **JSON** into, we need to make our own **Model**. First create a new package called **“com.example.lab_week_05.model”**.



- Next, create a new **Kotlin File** inside the new **Package**. Name it **“CatBreedData”**.

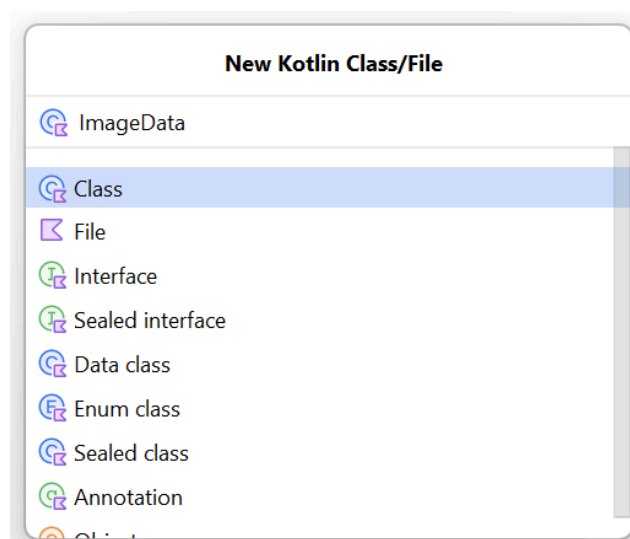


- Update your **CatBreedData.kt** to the code below.

```
package com.example.lab_week_05.model

data class CatBreedData(
    val name: String,
    val temperament: String
)
```

- Next, create another **Kotlin File** called **“ImageData”**.



- Update your **ImageData.kt** to the code below.

```
package com.example.lab_week_05.model
import com.squareup.moshi.Json
```

```
data class ImageData(
    @field:Json(name = "url") val imageUrl: String,
    val breeds: List<CatBreedData>
)
```

8. You may notice the peculiar `@field:Json(name = "url") val imageUrl: String` in the code. **Moshi** automatically maps JSON data to the available fields if they happen to have the **exact same name**. The `@field:Json(name = "url")` is basically used to **rename** the field name to a **meaningful** field name.
9. We've created our **Models**. Now let's update the **CatApiService** and change the **return type** to our new list of **Model**.

```
interface CatApiService {
    @GET("images/search")
    fun searchImages(
        @Query("limit") limit: Int,
        @Query("size") format: String
    ) : Call<List<ImageData>>
}
```

10. Update your **MainActivity.kt** also to change the **Converter** from **Scalar** to **Moshi**.

```
private val retrofit by lazy{
    Retrofit.Builder()
        .baseUrl("https://api.thecatapi.com/v1/")
        .addConverterFactory(MoshiConverterFactory.create())
        .build()
}
```

11. Lastly, let's update the **getCatImageResponse** function to change how the response is being read now that we have defined a different response type.

```
private fun getCatImageResponse() {
    val call = catApiService.searchImages(1, "full")
    call.enqueue(object: Callback<List<ImageData>> {
        override fun onFailure(call: Call<List<ImageData>>, t: Throwable) {
            Log.e(MAIN_ACTIVITY, "Failed to get response", t)
        }
    })
}
```

```

    }

    override fun onResponse(call: Call<List<ImageData>>,
response: Response<List<ImageData>>) {
        if(response.isSuccessful){
            val image = response.body()
            val firstImage = image?.firstOrNull()?.imageUrl ?: "No URL"
            apiResponseView.text = getString(R.string.image_placeholder,
firstImage)
        }
        else{
            Log.e(MAIN_ACTIVITY, "Failed to get response\n" +
                response.errorBody()?.string().orEmpty()
            )
        }
    }
}
})
}

```

12. Don't forget to also change your **Strings.xml**.

```

<resources>
    <string name="app_name">LAB_WEEK_05</string>
    <string name="image_placeholder">Image URL: %s</string>
</resources>

```

Commit to GITHUB at this point.
Commit Message: "Commit No. 2 - Parse JSON with Moshi and Display Image URL"

13. Now run your application. This time, the **actual URL** should appear on your screen, rather than the raw JSON string.

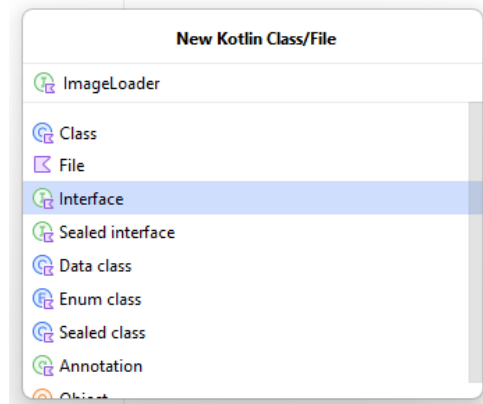


Part 3 - Loading the Image from the Obtained URL

1. Continue your “**LAB_WEEK_05**” project.
2. In this part, we will be focusing on how you can **fetch the image URL from the web** and **load it** in Android using **Glide** library. In order to use **Glide**, we need to add the necessary library to our application. Add the code below to the **dependencies** section in **build.gradle.kts (Module :app)** and **sync** your gradle files.

```
implementation("com.github.bumptech.glide:glide:4.14.2")
```

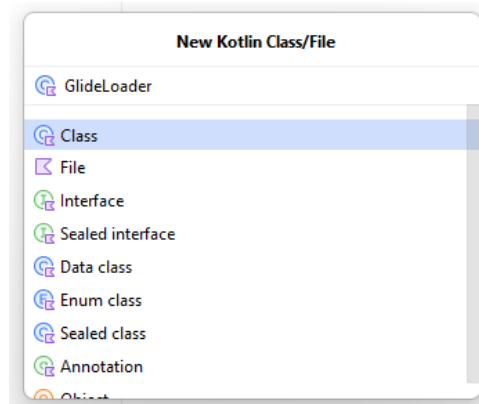
3. First, let's define an interface to load the images. Right click your **com.example.lab_week_05 > New > Kotlin Class/File**, set the type to **Interface** and set the name to **ImageLoader**.



4. Update the **ImageLoader.kt** to the code below.

```
interface ImageLoader {
    fun loadImage(imageUrl: String, imageView: ImageView)
}
```

5. Create a new **Kotlin File** again, set the type to **Class**, and name it **GlideLoader**.



6. Update the **GlideLoader.kt** to the code below.

```
class GlideLoader(private val context: Context) : ImageLoader {
    override fun loadImage(imageUrl: String, imageView: ImageView) {
        Glide.with(context)
            .load(imageUrl)
            .centerCrop()
            .into(imageView)
    }
}
```

7. Next, update your **activity_main.xml** to the code below.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/api_response"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintBottom_toTopOf="@+id/image_result"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

    <ImageView
        android:id="@+id/image_result"
        android:layout_width="match_parent"
        android:layout_height="400dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

8. Lastly, let's update the **MainActivity.kt**. Let's define the new **ImageView** reference in our kotlin file. Update the corresponding code in your **MainActivity.kt**.

```
private val catApiService by lazy{
    retrofit.create(CatApiService::class.java)
}

private val apiResponseView: TextView by lazy{
    findViewById(R.id.api_response)
}

private val imageResultView: ImageView by lazy {
    findViewById(R.id.image_result)
}

private val imageLoader: ImageLoader by lazy {
```

```
GlideLoader(this)
}
```

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

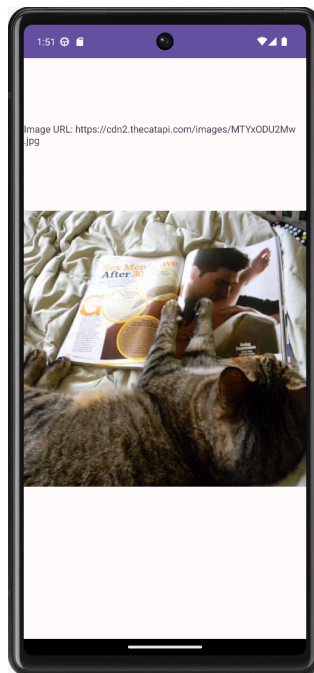
9. Now let's update how we load the image to the **ImageView**. Update your **getCatImageResponse** function in your **MainActivity.kt** to the code below.

```
private fun getCatImageResponse() {
    val call = catApiService.searchImages(1, "full")
    call.enqueue(object: Callback<List<ImageData>> {
        override fun onFailure(call: Call<List<ImageData>>, t: Throwable) {
            Log.e(MAIN_ACTIVITY, "Failed to get response", t)
        }

        override fun onResponse(call: Call<List<ImageData>>, response:
Response<List<ImageData>>) {
            if(response.isSuccessful){
                val image = response.body()
                val firstImage = image?.firstOrNull()?.imageUrl.orEmpty()
                if (firstImage.isNotBlank()) {
                    imageLoader.loadImage(firstImage, imageView)
                } else {
                    Log.d(MAIN_ACTIVITY, "Missing image URL")
                }
                apiResponseView.text = getString(R.string.image_placeholder,
firstImage)
            }
            else{
                Log.e(MAIN_ACTIVITY, "Failed to get response\n" +
                    response.errorBody()?.string().orEmpty()
                )
            }
        }
    })
}
```

Commit to GITHUB at this point.
Commit Message: "Commit No. 3 - Load and Display Cat Image using Glide"

10. Run your application, and the **image** should appear below the **URL**.



ASSIGNMENT

Continue your **LAB_WEEK_05** project. Remember that you defined the “**breeds**” variable in **ImageData.kt**. Currently it’s only showing the **URL** of the loaded image. Change the **URL Text** to the **breed** of the current **Cat Image**.

Note: Due to the incompleteness of the **Cat API**, not all cats have the **Breed** data, so instead you can display “**Unknown**” for the ones that are **not found**.

