Syntax Institut

Modul 3 – Android App Entwicklung mit Kotlin

API Calls und Bilder

Gliederung

- retrofit
- User List App
- Coil
- Beispiel

Quelle: https://i.stack.imgur.com/tskAi.png

RESTful Service

Representational State Transfer

gängige Architektur für Webserver

Webservices in REST Architektur nennt man auch

RESTful Services

Serverresourcen werden über eine URI/URL

angesprochen

Kommunikation läuft über HTTP Protokoll

Quelle: https://www.opc-router.de/was-ist-rest/

URI, URL, HTTP

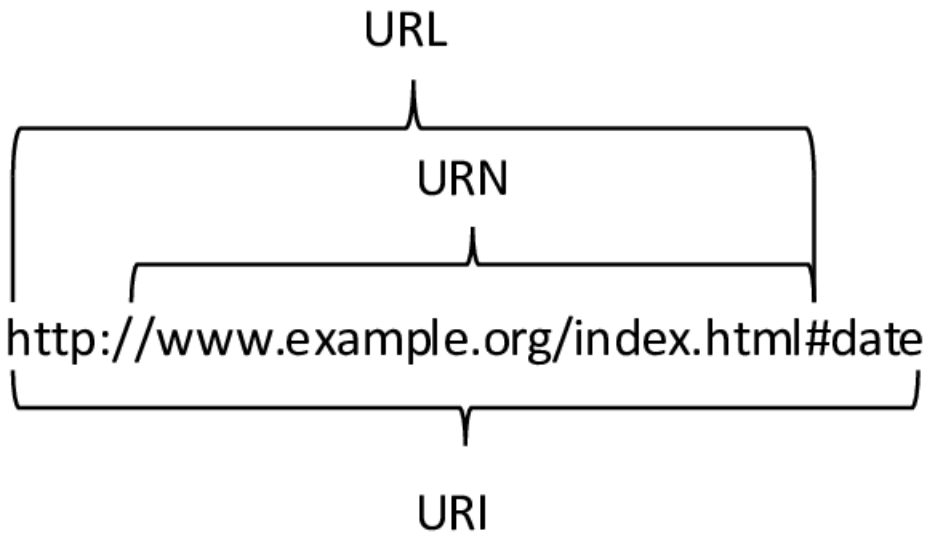
Uniform Resource Identifier

Uniform Resource Locator

Uniform Resource Name

Übliche HTTP Operationen:

- GET – um Daten vom Server abzufragen
- POST/PUT - um Daten am Server abzulegen oder zu verändern
- DELETE – um Daten am Server zu löschen



Quelle: https://www.researchgate.net/figure/The-illustration-of-the-URL-URN-and-URI-26_fig4_346585530

JSON

GET

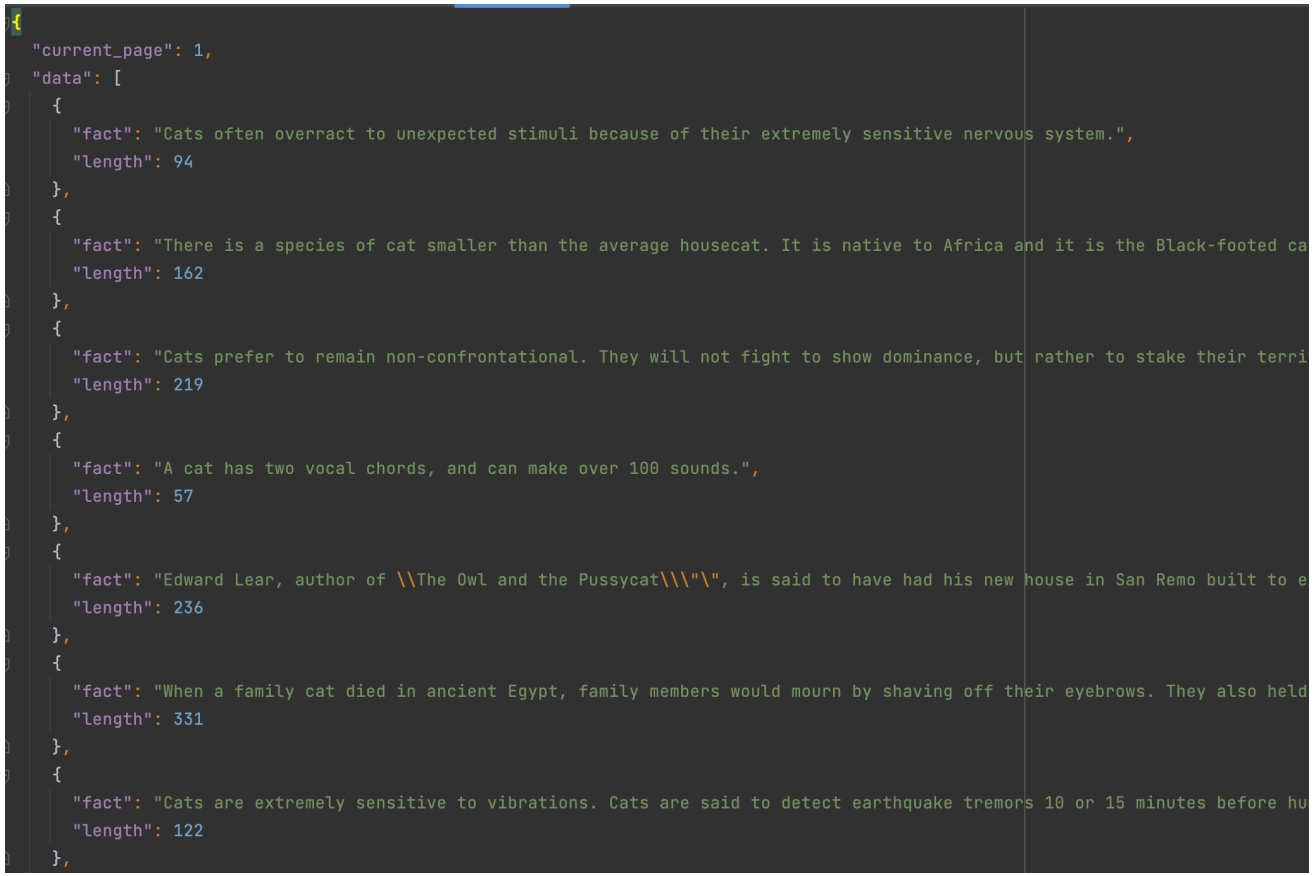
https://catfact.ninja/facts

Stellt eine Liste an Katzenfakten als JSON bereit

JSON

Java**S**cript **O**bject **N**otation

Gängiges Format um Objekte für Datenaustausch aufzubereiten

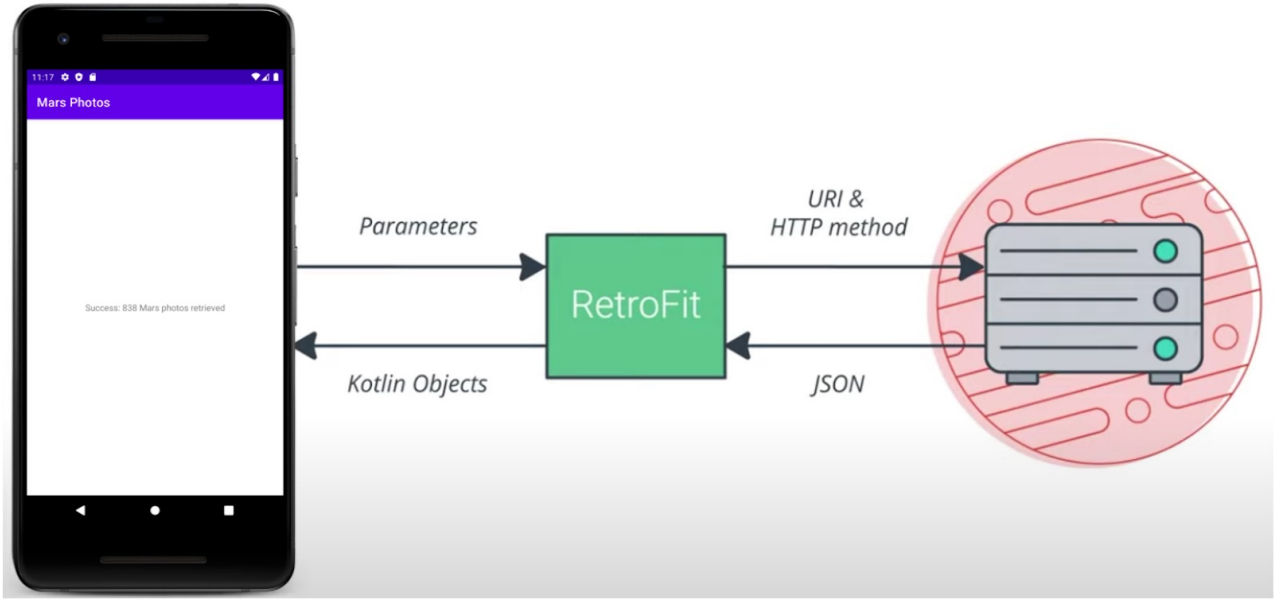


retrofit

https://square.github.io/retrofit/

Ist ein HTTP Client für Android

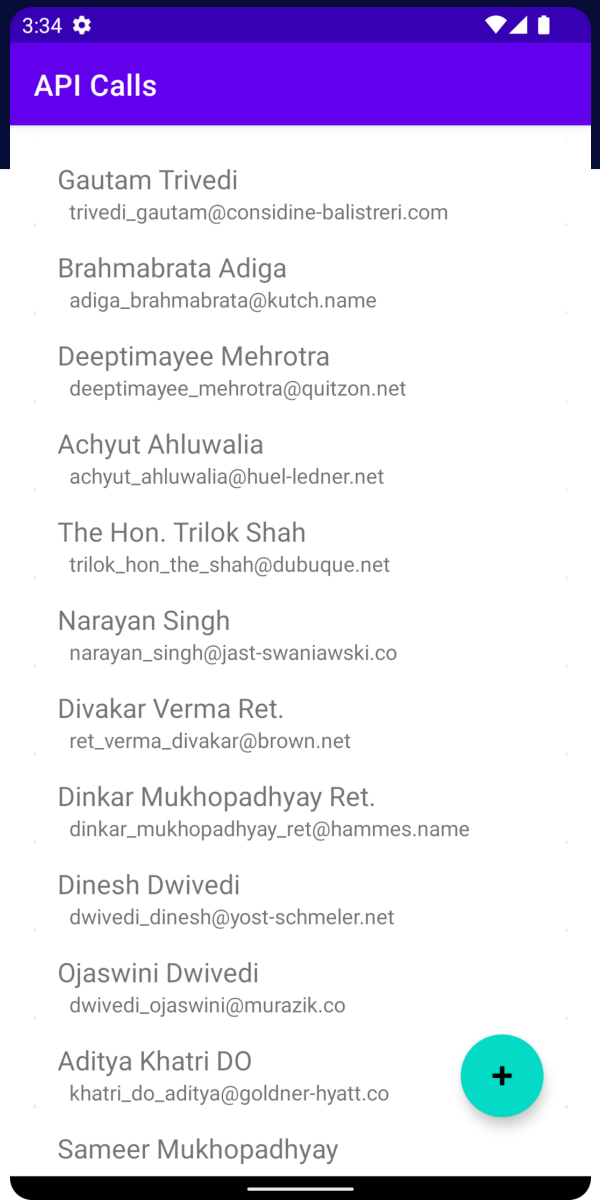
Ein **RetrofitService** übernimmt die
Kommunikation mit dem Server
und übersetzt die Antwort in Kotlin Objekte



Quelle: https://developer.android.com/codelabs/basic-android-kotlin-training-getting-data-internet/img/a8f10b735ad998ac.png

Beispiel: User List

- ein **Repository** kümmert sich um die Organisation der Daten
- ein **Retrofitservice** kommuniziert mit dem Server
- mittels **GET** wird eine Liste an Usern geladen
- mittels **POST** wird ein neuer User angelegt



Coil

https://coil-kt.github.io/coil/

Coroutine Image Loader

- download
- speichern
- dekodieren
- Hintergrund Thread
- angepasst an Netzwerk und CPU Leistung

Coil

https://coil-kt.github.io/coil/

Coroutine Image Loader

- download
- speichern
- dekodieren
- Hintergrund Thread
- angepasst an Netzwerk und CPU Leistung

```
imageView.load("https://www.example.com/image.jpg") {
    crossfade(true)
    placeholder(R.drawable.image)
    transformations(CircleCropTransformation())
}
```

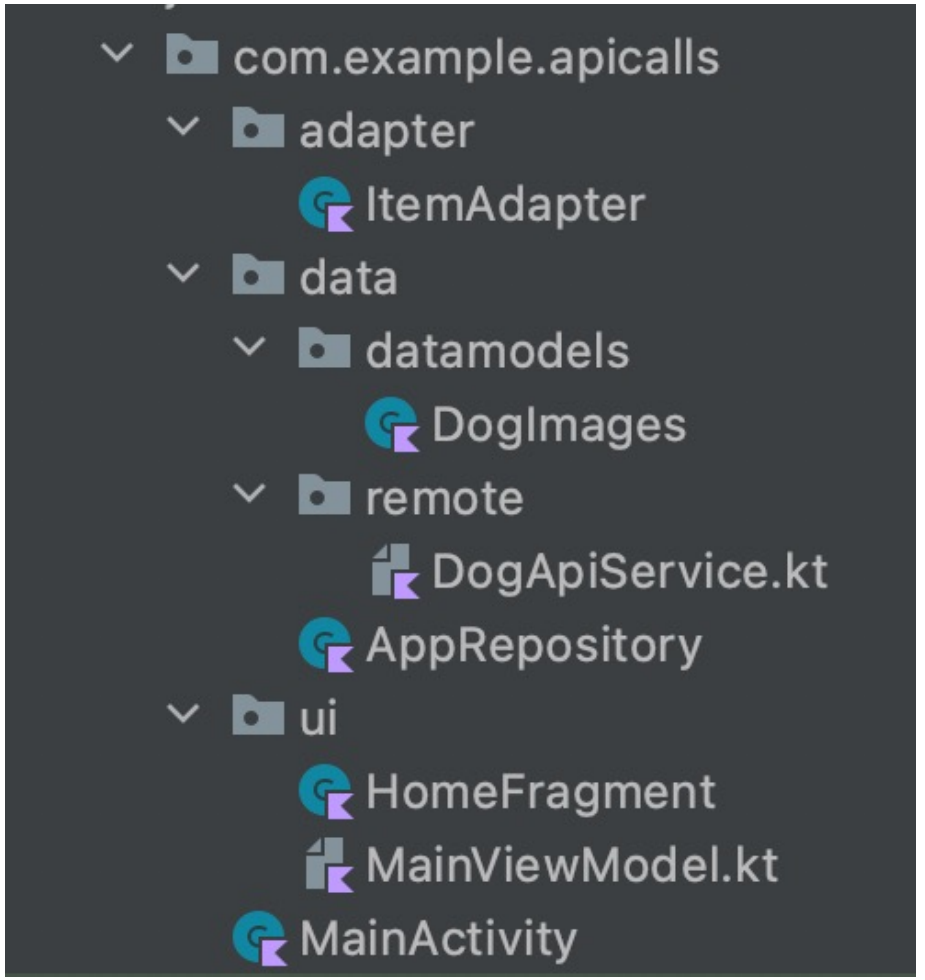
```
// Coil
implementation "io.coil-kt:coil:1.1.1"
```

Beispiel: Hounds of Love

- ein **Repository** kümmert sich um die Organisation der Daten
- ein **Retrofitservice** kommuniziert mit dem Server
- mittels **GET** wird eine Liste an Urls geladen
- mittels **Glide** werden die entsprechenden Bilder geladen

Struktur

gewohnte **MVVM** Architektur



AndroidManifest.xml

Folgende Zeile erlaubt der App sich mit dem **Internet** zu verbinden

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

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

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="API Calls"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.APICalls">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

build.gradle (app Module)

Im GradleScript werden neben den Dependencies für Navigation, LiveData und ViewModel

auch jene von **retrofit**, **moshi** und **Coil**

eingefügt.

```
dependencies {

    implementation 'androidx.core:core-ktx:1.7.0'
    implementation 'androidx.appcompat:appcompat:1.4.1'
    implementation 'com.google.android.material:material:1.5.0'
    implementation 'androidx.constraintlayout:constraintlayout:2.1.3'
    implementation 'androidx.legacy:legacy-support-v4:1.0.0'
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
    implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.4.1'
    implementation 'androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1'
    implementation 'androidx.fragment:fragment-ktx:1.4.1'


    // Navigation
    implementation "androidx.navigation:navigation-fragment-ktx:2.4.2"
    implementation "androidx.navigation:navigation-ui-ktx:2.4.2"

    //Retrofit
    implementation "com.squareup.retrofit2:retrofit:2.9.0"
    implementation "com.squareup.retrofit2:converter-moshi:2.9.0"
    implementation "com.squareup.moshi:moshi-kotlin:1.13.0"

    // Coil
    implementation "io.coil-kt:coil:1.1.1"
}
```

MainViewModel

eine Liste an ImageUrls wird vom **Repository** bereitgestellt

Über **loading** wird der Status des API-Calls als **ApiStatus** preisgegeben um den Ladespinner bzw. Error ein-/auszublenden

weiters gibt es eine Funktion um neue **Daten** zu **laden**

beide Funktionen werden innerhalb **Coroutines** abgerufen um das UI nicht zu blockieren

```
const val TAG = "MainViewModel"

enum class ApiStatus { LOADING, ERROR, DONE }

class MainViewModel : ViewModel() {

    private val repository = AppRepository(DogApi)

    private val _loading = MutableLiveData<ApiStatus>()
    val loading: LiveData<ApiStatus>
        get() = _loading

    val images = repository.imageList

    init {
        loadData()
    }

    fun loadData() {
        viewModelScope.launch { this: CoroutineScope
            _loading.value = ApiStatus.LOADING
            try {
                repository.getImages()
                _loading.value = ApiStatus.DONE
            } catch (e: Exception) {
                Log.e(TAG, msg: "Error loading Data from API: $e")
                _loading.value = ApiStatus.ERROR
            }
        }
    }
```

AppRepository

imageList stellt eine Liste an urls als **LiveData** zur Verfügung

getImages()

lädt über den **retrofitService** eine aktuelle Liste an urls in die **LiveData**

```
class AppRepository(private val api: DogApi) {

    private val _imageList = MutableLiveData<List<String>>()
    val imageList: LiveData<List<String>>
        get() = _imageList

    suspend fun getImages() {
        delay( timeMillis: 2000)
        _imageList.value = api.retrofitService.getImages().message
    }
}
```

DogApiService

da die dogAPI einen Zugangsschlüssel (API_TOKEN) verlangt muss kein extra **client** angelegt werden

um Antworten direkt zu übersetzen wird ein **moshi** angelegt

danach wird ein **retrofit** gebaut welcher moshi und BASE_URL verwendet

```
const val BASE_URL = "https://dog.ceo/api/breed/hound/"


private val moshi = Moshi.Builder()
    .add(KotlinJsonAdapterFactory())
    .build()

private val retrofit = Retrofit.Builder()
    .addConverterFactory(MoshiConverterFactory.create(moshi))
    .baseUrl(BASE_URL)
    .build()

interface DogApiService {

    @GET( value: "images")
    suspend fun getImages(): DogImages
}

object DogApi {
    val retrofitService: DogApiService by lazy { retrofit.create(DogApiService::class.java) }
}
```

DogApiService

das Interface **DogApiService** bestimmt wie mit dem Server kommuniziert wird

- ein **GET-Request** am Endpunkt images welcher **DogImages** zurückliefert

UserApi dient als Zugangspunkt für den Rest der App und stellt das Interface als **retrofitservice** zur Verfügung

```
const val BASE_URL = "https://dog.ceo/api/breed/hound/"


private val moshi = Moshi.Builder()
    .add(KotlinJsonAdapterFactory())
    .build()

private val retrofit = Retrofit.Builder()
    .addConverterFactory(MoshiConverterFactory.create(moshi))
    .baseUrl(BASE_URL)
    .build()

interface DogApiService {

    @GET( value: "images")
    suspend fun getImages(): DogImages
}

object DogApi {
    val retrofitService: DogApiService by lazy { retrofit.create(DogApiService::class.java) }
}
```

DogImages

Weil in der **JSON** vom Server die String-Liste der Hundebilder in einer **message** Variable kommt

Muss unsere **DogImages** DataClass genauso gebaut werden damit der **moshi** Konverter weiß wie und was er umzuwandeln hat


```
{
  "message": [
    "https:\/\/images.dog.ceo\/breeds\/dachshund\/Dachshund_rabbit.jpg",
    "https:\/\/images.dog.ceo\/breeds\/dachshund\/Daschund-2.jpg",
    "https:\/\/images.dog.ceo\/breeds\/dachshund\/Daschund_Wirehair.jpg",
    "https:\/\/images.dog.ceo\/breeds\/dachshund\/Dash_Dachshund_With_Hat.jpg",
    "https:\/\/images.dog.ceo\/breeds\/dachshund\/Miniature_Daschund.jpg",
    "https:\/\/images.dog.ceo\/breeds\/dachshund\/Standard_Wire-hair_Dachshund.jpg"
```


```
data class DogImages(
    val message: List<String>
)
```

ItemAdapter

In unserem **RecyclerViewAdapter** wird zuerst

die StringVariable in ein URI Format gebracht

und anschließend verwendet um mittels **Coil** das

Bild zu laden

dieses bekommt noch Runde Ecken

und sollte was schieflaufen wird stattdessen ein

broken_image aus den Resources geladen

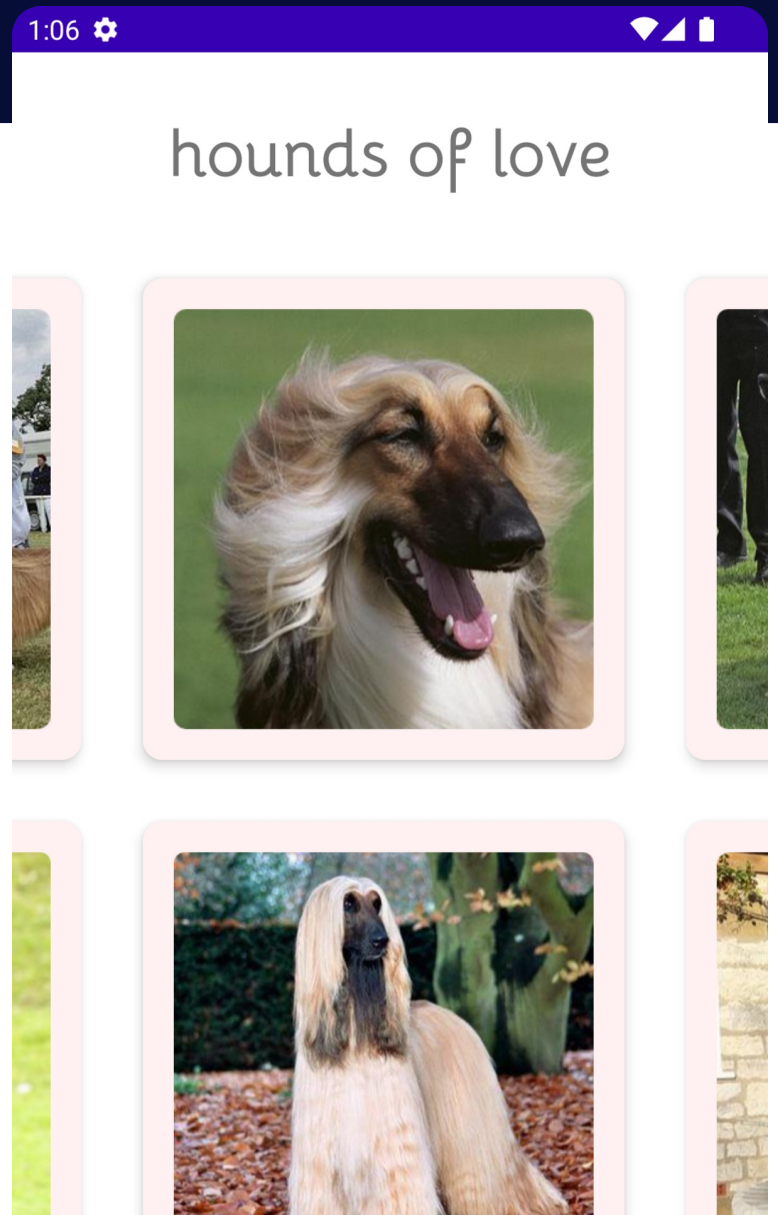
```kotlin
override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
    val item = dataset[position]

    val imgUri = item.toUri().buildUpon().scheme( scheme: "https").build()

    holder.imgView.load(imgUri) {  this: ImageRequest.Builder
        error(R.drawable.ic_round_broken_image_24)
        transformations(RoundedCornersTransformation( radius: 10f))
    }
}
```

Fertig

Unsere App zeigt Hundebilder von der API



Activities Lifecycle

Wiederholung - Was haben wir heute gelernt?

1	retrofit
2	UserList App
3	Coil

Quelle: https://www.viridea.it/consigli/il-galgo-espanol/

Syntax Institut

Viel Spaß!