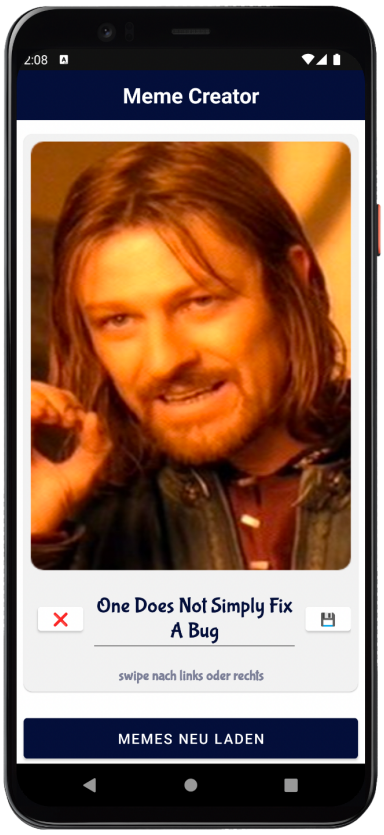
Kotlin 5.4 - Repository Pattern

# Aufgaben

Hinweis: *Letzte ÜbungsApp :)* Zu bearbeiten ist Aufgabe 1. Aufgabe 2 ist eine Bonusaufgabe.

## 1. MEME CREATOR - Repository Pattern

In dieser Aufgabe programmieren wir die bekannte App “Meme Creator” um. Ziel ist es, eine Datenbank einzubauen und die App im Repository Pattern zu strukturieren.




- Öffne das Projekt “Meme Creator”

  Die App funktioniert, jedoch werden Änderungen nicht gespeichert, wenn die App geschlossen wird. Wir wollten eine Datenbank einbauen, in der Änderungen, wie das Ändern des Untertitels oder das Entfernen ganzer Memes, gespeichert werden.

- Passe die Klasse `Meme` an, indem du ihr eine weitere Variable `id` vom Typ `long`, und die für die Datenbank nötigen Annotationen gibst

  Es soll der Aufbau der App in die Struktur des Repository Patterns gebracht werden. Erstelle dafür ein neues `package` `data.local`, in welchem jetzt die Datenbank eingerichtet wird

- Erstelle innerhalb des Ordners ein Interface `MemeDatabaseDao` mit der `@Dao` Annotation. Das DAO der Datenbank soll fünf Funktionen (Operationen auf der Datenbank) enthalten. Jede Funktion hat eine entsprechende Annotation und, wenn nötig, SQL Anweisungen:
    - Eine Funktion `insertAll`, die eine Liste aus Memes übergeben bekommt und nichts zurückliefert. Sie soll alle Memes in die Datenbank einfügen. Bei einem Konflikt soll das Ersetzen des alten Eintrags die Strategie sein
    - Eine Funktion `update`, die ein Meme übergeben bekommt und nichts zurückliefert. Sie soll das übergebene Meme in der Datenbank aktualisieren
    - Eine Funktion `getAll`, die nichts übergeben bekommt und mit LiveData verpackt eine Liste aus Memes zurückliefert. Sie soll dabei alle Memes aus der Datenbank auswählen
    - Eine Funktion `deleteAll`, die nichts übergeben bekommt und nichts zurückliefert. Sie soll alle Memes aus der Datenbank löschen
    - Eine Funktion `deleteById`, die eine ID vom Typ `long` übergeben bekommt und nichts zurückliefert. Sie soll das Meme aus der Datenbank löschen, bei dem die ID des Memes mit der übergebenen ID übereinstimmt
- Als Nächstes richten wir die Room Datenbank ein. Erstelle dafür eine Datei `MemeDatabase` im Ordner `local`
    - Erstelle die abstrakte Klasse `MemeDatabase`, welche von `RoomDatabase` abgeleitet ist und mit der `@Database(...)` Annotation versehen ist. In ihr enthalten ist die abstrakte Variable `memeDatabaseDao`
    - Erstelle eine `private lateinit` Variable `INSTANCE` vom Typ `MemeDatabase`
    - Schreibe die Funktion `getDatabase`. Sie bekommt einen Kontext übergeben und liefert die `INSTANCE` zurück. Falls noch keine Instanz der Datenbank existiert, erstellt sie eine neue Instanz
- In der Klasse `AppRepository` verwalten wir jetzt sowohl die Daten aus dem API-Call, als auch die Daten aus der Datenbank
    - Füge dem Konstruktor also noch eine private Variable `database` hinzu, in der eine Instanz der `MemeDatabase` übergeben wird
    - Die Variable `memes` wird Live mit den Einträgen der Datenbank gefüllt. Konkret wird die LiveData Rückgabe der Funktion `getAll` aus dem DAO der `database` in der Variablen gespeichert. Da die Variable nicht innerhalb des Repository verändert wird, sondern nur durch die darin gespeicherte LiveData, kannst du auf die Verschachtelung verzichten.

    - Die Funktion `getMemes` funktioniert jetzt etwas anders. Schreibe sie folgendermaßen um: Die Funktion wechselt den Kontext (`Dispatchers.IO`) und holt sich mit einem API-Call die neue Liste an Memes. Anschließend wird die gesamte Liste über die DAO Funktion in die Datenbank eingefügt.
    - Zusätzlich brauchen wir ein paar neue Funktionen, die die Daten aus der Datenbank verwalten:
        - Eine Funktion `deleteMeme`. Sie bekommt ein Meme übergeben und versucht in einem `try catch` Block das Meme aus der Datenbank zu löschen. Fall dies fehlschlägt, soll die Fehlermeldung geloggt werden
        - Eine Funktion `deleteAllMemes`. Sie versucht in einem `try catch` Block das Meme aus der Datenbank zu löschen. Fall dies fehlschlägt, soll die Fehlermeldung geloggt werden
        - Eine Funktion `updateMeme`. Sie bekommt ein Meme übergeben und versucht in einem `try catch` Block das Meme aus der Datenbank zu löschen. Fall dies fehlschlägt, soll die Fehlermeldung geloggt werden
- Die Daten werden jetzt über das Repository verwaltet und bereitgestellt. Wechsle in das ViewModel, wo wir die Zustände der App jetzt entsprechen setzen können
    - Um Zugriff auf die Application zu haben muss der Konstruktur des ViewModels angepasst werden

```
class MemesViewModel(application: Application): AndroidViewModel(application)
```

    - Da die Repository Instanz nun auch eine Datenbank Instanz benötigt, müssen wir zuerst eine Datenbank Instanz erstellen und dem Repository Konstruktor übergeben.
    - Die Funktion `loadData` wird leicht umgeschrieben, zuerst werden alle Einträge aus der Datenbank gelöscht und anschließend wird in einem `try catch` Block versucht, den API-Call über das Repository zu starten
    - Schreibe zusätzlich zwei weitere Funktionen des ViewModel auf die wir später zugreifen können
        - Die Funktion `deleteMeme`, sie bekommt ein Meme übergeben und löscht es in einer Coroutine über das Repository aus der Datenbank
        - Die Funktion `updateMeme`, sie bekommt ein Meme übergeben und aktualisiert den Datenbankeintrag in einer Coroutine über das Repository

- Weiter geht es im `MemesFragment`. Hier bauen wir ein paar Features ein, die schon durch die zusätzlichen XML Elemente angedeutet sind. Durch die neue Art der Datenverwaltung können wir Änderungen vornehmen, die auch nach einem Neustart der App bestehen bleiben. Folgende Änderungen sollen eingebaut werden:

    - Die Memes aus der API sollen nicht wie bisher bei jedem Start der APP abgerufen werden, sondern nur noch auf Knopfdruck (`btnRefresh`). Entferne daher die Zeile an Code, die dafür sorgt, dass bei jedem Start des Fragments erneut die Daten geladen werden.
    - Bei einem Klick auf den Speichern-Button oder den Löschen-Button soll die Änderung auch in der Datenbank vorgenommen werden. Da diese Buttons aber in den Listeneinträgen sitzen, müssen wir dies aus dem Adapter heraus starten. Dafür müssen wir zwei Pfeilfunktionen (Lambdas) schreiben, die wir dem Adapter im Konstruktor übergeben und die dieser dann verwenden kann (siehe Vorlesungsfolien zu Kotlin Basics)
        - In einer Variable `deleteMeme` wird eine Pfeilfunktion gespeichert, die ein `Meme` übergeben bekommt und nichts zurückgeliefert (`Unit`). Konkret soll die Pfeilfunktion mit dem ihr übergebenen `meme` die Funktion `deleteMeme` aus dem ViewModel aufrufen.
        - In einer Variable `updateMeme` wird eine Pfeilfunktion gespeichert, die ein `Meme` übergeben bekommt und nichts zurückgeliefert (`Unit`). Konkret soll die Pfeilfunktion mit dem ihr übergebenen `meme` die Funktion `updateMeme` aus dem ViewModel aufrufen.
- Im Adapter müssen wir den Konstruktor ebenfalls um zwei private Variablen `deleteMeme` und `updateMeme` erweitern. Beide erwarten eine Pfeilfunktion, bei der ein `Meme` übergeben wird und nicht zurückgeliefert wird (`Unit`)
    - Erweitere den Konstruktor entsprechend
    - Bei einem Klick auf den Speicher-Button soll die entsprechende Pfeilfunktion ausgeführt werden, um die Änderungen auch in der Datenbank zu übernehmen

    - Setze ebenfalls einen Click Listener auf den Löschen-Button, in dem ebenfalls die entsprechende Pfeilfunktion zum Ändern der Daten in der Datenbank ausgeführt wird
- Zurück im `MemesFragment` muss der Aufruf des Adapterkonstruktors noch angepasst werden. Damit der LayoutManager der Recyclerview nach Löschen eines Memes wieder auf die aktuelle Position zurückspringt und nicht zum Anfang der Liste, müssen wir der Recyclerview zudem noch die richtige Position mitteilen
    - speichere die aktuelle Position der RecyclerView in einer Variablen. Dafür musst du über `binding` den Layout Manager der Recyclerview holen und diesen anschließend als `LinearLayoutManager` casten. Dann kannst du eine Funktion aufrufen, die dir die Position des ersten sichtbaren Items zurückliefert
    - Weise der Recyclerview danach den Adapter mit aktualisiertem Konstruktor zu
    - Zum Schluss kannst du der RecyclerView mit einem Aufruf der richtigen Funktion mitteilen, zur vorher gespeicherten Position zu scrollen
- Die App sollte nun funktionieren! Die Daten aus der API und der Datenbank werden richtig verwaltet und Änderungen bleiben bestehen. Führe die App aus und teste, ob alles funktioniert!

Viel Erfolg! 💾

## 2. LADEANIMATION - Bonus

In dieser Aufgabe soll für ein benutzerfreundlicheres Erlebnis eine Ladeanimation gestartet werden, jedes Mal, wenn die Memes erneut aus der API geladen werden.


- Gehe in die ViewModel Datei der App
- Erstelle hier eine `enum` Klasse mit den Zuständen `LOADING`, `ERROR` und `DONE`
- Erstelle innerhalb des ViewModels eine neue Live Date Variable `loading`, in der der Status des API-Calls, in Form von Zuständen der Enum Klasse, gespeichert wird
- Erweitere die Funktion `loadData`, sodass die Variable `loading` immer korrekt aktualisiert wird. Bevor
- Im `MemesFragment` wird `loading` beobachtet und die `visibility` der `progressBar` und des `errorImage` entsprechend gesetzt

Viel Erfolg! 🔄