

Modul 3 – Android App Entwicklung mit Kotlin

Repository Pattern



Gliederung

- Repository Pattern
- Vorteile
- Caching
- Anwendung



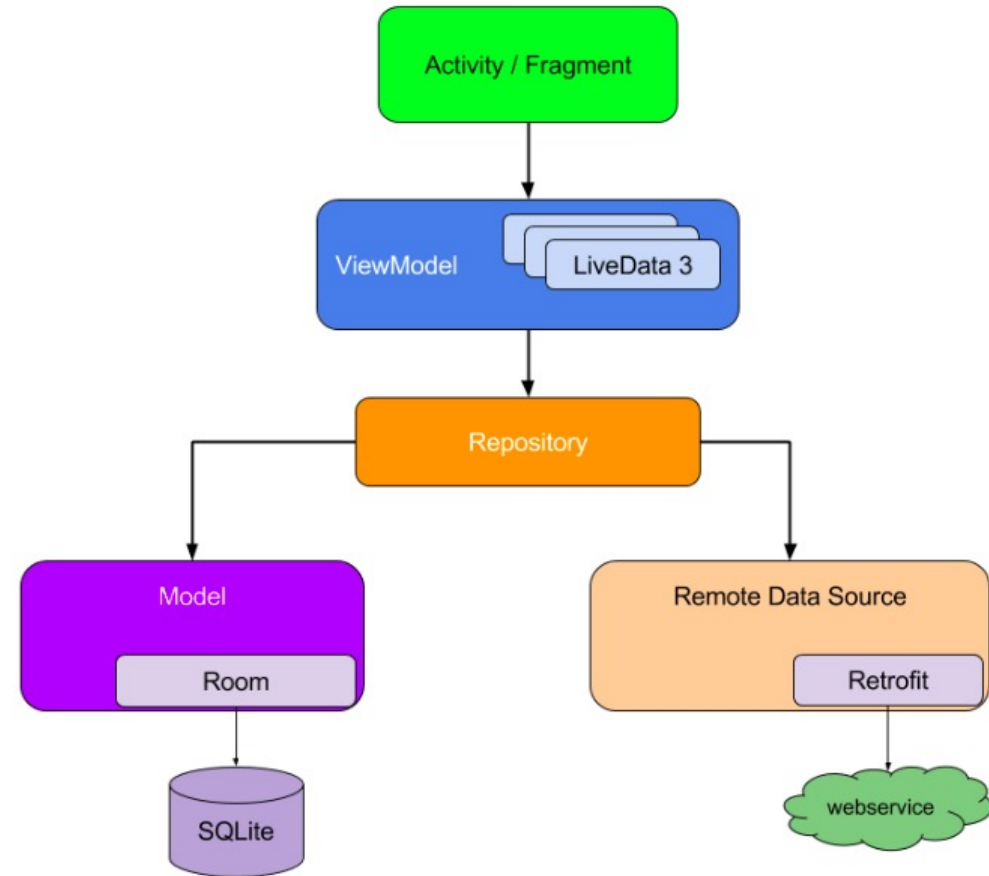
Quelle: <https://www.welcher-beruf-passt.org/images/berufe/teaser/buchhalter-buchhalterin.jpg>

Repository Pattern

Designpattern welches den Data Layer vom Rest der App isoliert

Data Layer enthält API Calls, Room Datenbanken, Exception Handling
(alles was Daten liest oder manipuliert)

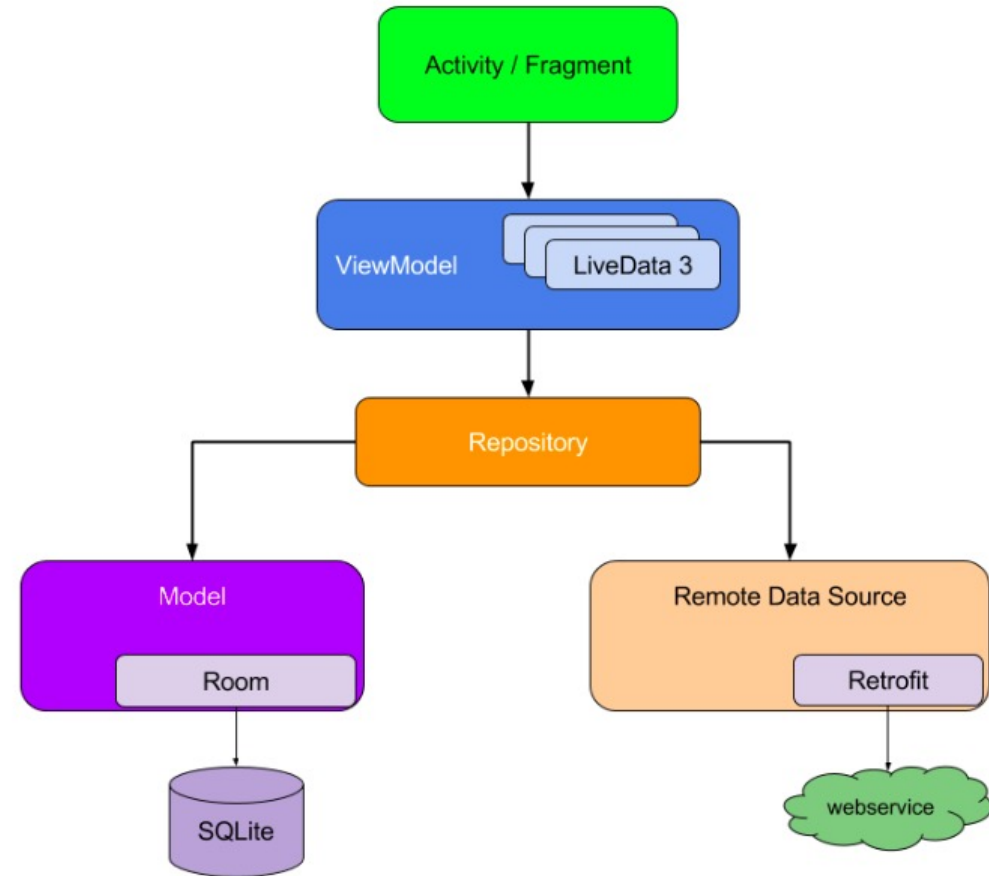
Die App kann somit nur über das Repository auf Daten zugreifen



Quelle: <https://developer.android.com/codelabs/basic-android-kotlin-training-repository-pattern/img/69021c8142d29198.png>

Vorteile

- Code ist modular
falls sich **Datenquellen ändern** müssen
Änderungen nur im **Repository**
vorgenommen werden
- Repository ist **Single Source of Truth**
falls sich die Daten von **verschiedene**
Quellen unterscheiden löst das
Repository den **Konflikt** und versucht
der App **aktuelle** und **genaue Daten** zu
präsentieren



Quelle: <https://developer.android.com/codelabs/basic-android-kotlin-training-repository-pattern/img/69021c8142d29198.png>

Caching

Zwischenspeichern von Daten

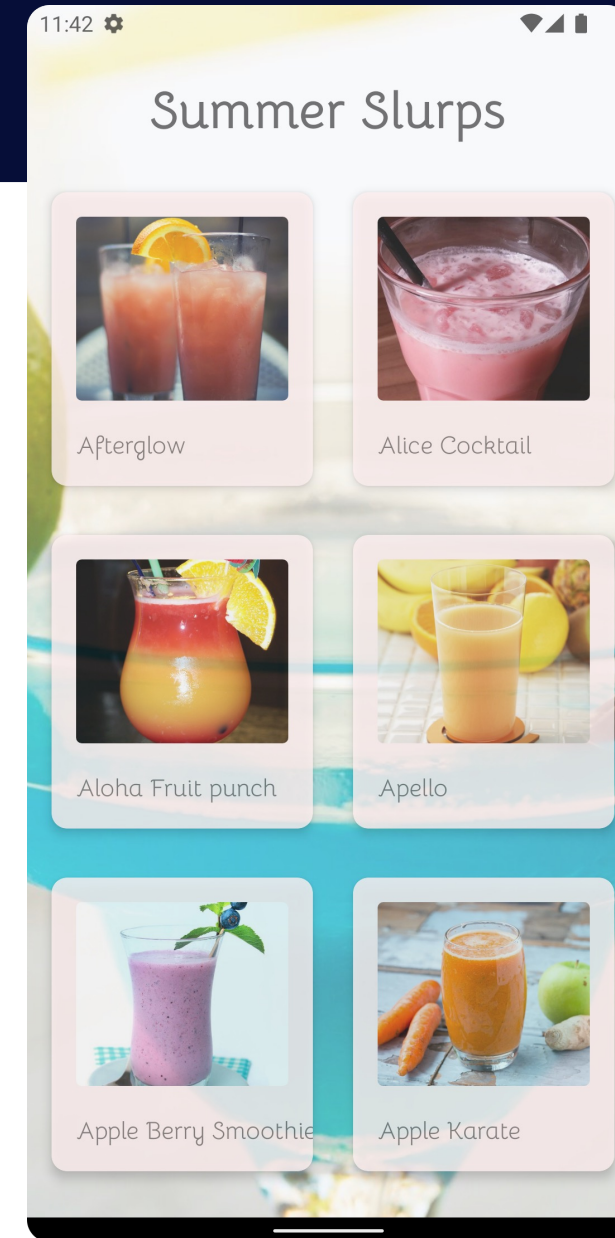
falls die Internetverbindung unterbrochen wird kann die App trotzdem auf zwischengespeicherte Daten zurückgreifen und diese werden aktualisiert sobald die API wieder erreicht wird



Quelle: <https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTaxbTb3eWqd3I8jAThktEnwDv2a3J4IkCQ1w&usqp=CAU>

Beispiel: Summer Slurps

- API-Service
um Drinkdaten vom **Server** zu holen
- Datenbank
um Daten zu **speichern**
- Repository
versucht beim Start der App **neue Daten vom Server** zu holen und in Datenbank zu speichern
Gibt Daten von **Datenbank** **direkt an App** weiter



Struktur

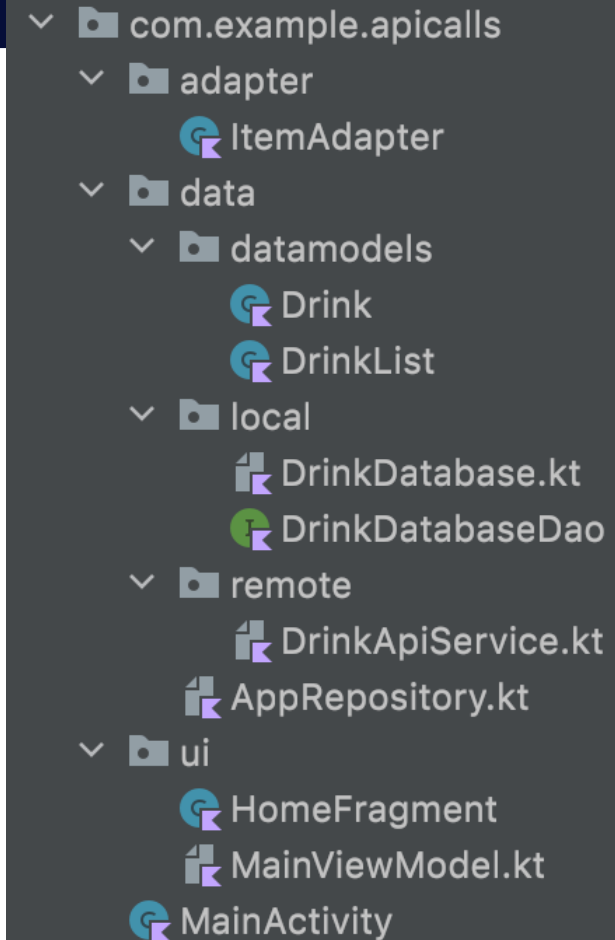
MVVM Pattern

Repository Pattern

im `data/remote` Package befindet sich der
Code für den **APIService**

in `data/local` der Datenbankcode

das **Repository** wird mit beiden arbeiten



```
graph TD; com[com.example.apicalls] --> adapter[adapter]; com --> data[data]; com --> ui[ui]; adapter --> ItemAdapter[ItemAdapter]; data --> datamodels[datamodels]; data --> local[local]; data --> remote[remote]; ui --> HomeFragment[HomeFragment]; ui --> MainViewModel[MainViewModel.kt]; ui --> MainActivity[MainActivity]; datamodels --> Drink[Drink]; datamodels --> DrinkList[DrinkList]; local --> DrinkDatabase[DrinkDatabase.kt]; local --> DrinkDatabaseDao[DrinkDatabaseDao]; remote --> DrinkApiService[DrinkApiService.kt]; remote --> AppRepository[AppRepository.kt];
```

com.example.apicalls

- adapter
 - ItemAdapter
- data
 - datamodels
 - Drink
 - DrinkList
 - local
 - DrinkDatabase.kt
 - DrinkDatabaseDao
 - remote
 - DrinkApiService.kt
 - AppRepository.kt
- ui
 - HomeFragment
 - MainViewModel.kt
 - MainActivity

JSON Response

da uns der Server die Getränkeliste **innerhalb** einer **drinks variable** übergibt müssen wir 2 Datenklassen anlegen

Drinklist

- Getränkeliste namens drink

Drink

- id
- name
- picture (Link zum Bild)

```
{
  "drinks": [
    {
      "strDrink": "Afterglow",
      "strDrinkThumb": "https://www.thecocktaildb.com/images/media/drink/vuquyv1468876052.jpg",
      "idDrink": "12560"
    },
    {
      "strDrink": "Alice Cocktail",
      "strDrinkThumb": "https://www.thecocktaildb.com/images/media/drink/gvqtpv1468876144.jpg",
      "idDrink": "12562"
    },
    {
      "strDrink": "Aloha Fruit punch",
      "strDrinkThumb": "https://www.thecocktaildb.com/images/media/drink/wsyyrt1468876267.jpg",
      "idDrink": "12862"
    },
    {
      "strDrink": "Apello",
      "strDrinkThumb": "https://www.thecocktaildb.com/images/media/drink/uptxtv1468876415.jpg",
      "idDrink": "15106"
    },
    {
      "strDrink": "Apple Berry Smoothie",
      "strDrinkThumb": "https://www.thecocktaildb.com/images/media/drink/xwqvur1468876473.jpg",
      "idDrink": "12710"
    },
  ],
}
```


Datamodels

Da `strDrink` oder `strDrinkThumb` unpraktische Bezeichnungen für unsere Variablen sind können wir unsere eigene Bezeichnung verwenden und mittels `@Json(name = „blub“)` für unseren `moshiConverter` anmerken welche Variable im JSON gemeint ist

```
data class DrinkList(  
    val drinks: List<Drink>  
)
```

```
@Entity  
data class Drink(  
  
    @PrimaryKey  
    @Json(name = "idDrink")  
    val id: Long,  
  
    @Json(name = "strDrink")  
    val name: String,  
  
    @Json(name = "strDrinkThumb")  
    val picture: String  
)
```

DrinkApiService.kt

`getDrinkList()`

Liefert eine **Drinklist** Variable zurück welche eine Liste an Getränken beinhaltet (**drinks**)

```
const val BASE_URL = "https://www.thecocktaildb.com/api/json/v1/"

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

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

interface DrinkApiService {

    @GET("1/filter.php?a=Non_Alcoholic")
    suspend fun getDrinkList(): DrinkList
}

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

DrinkDatabase.kt

Datenbank namens `drink_database` wird erstellt und
eine **Tabelle** aus der `Drink` Klasse erstellt

```
@Database(entities = [Drink::class], version = 1)
abstract class DrinkDatabase : RoomDatabase() {

    abstract val drinkDatabaseDao: DrinkDatabaseDao
}

private lateinit var INSTANCE: DrinkDatabase

// if there's no Database a new one is built
fun getDatabase(context: Context): DrinkDatabase {
    synchronized(DrinkDatabase::class.java) {
        if (!::INSTANCE.isInitialized) {
            INSTANCE = Room.databaseBuilder(
                context.applicationContext,
                DrinkDatabase::class.java,
                name: "drink_database"
            )
                .build()
        }
    }
    return INSTANCE
}
```

DrinkDatabaseDao.kt

insertAll()

speichert eine Liste an Drinks in die Datenbank falls ein Eintrag mit derselben id schon existiert wird dieser überschrieben

getAll()

stellt alle Element der Drink Tabelle als LiveData zur Verfügung

```
@Dao
interface DrinkDatabaseDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertAll(drinks: List<Drink>)

    @Query("SELECT * from Drink")
    fun getAll(): LiveData<List<Drink>>

    @Query("DELETE from Drink")
    fun deleteAll()

}
```

AppRepository.kt

drinkList

stellt alle Elemente der DrinkDatabase als
LiveData zur Verfügung

getDrinks()

wechselt innerhalb der Coroutine des
Dispatcher und versucht eine neue DrinkList
zu laden um diese direkt anschließend in die
Datenbank zu speichern

```
class AppRepository(private val api: DrinkApi, private val database: DrinkDatabase) {  
  
    val drinkList: LiveData<List<Drink>> = database.drinkDatabaseDao.getAll()  
  
    suspend fun getDrinks() {  
        withContext(Dispatchers.IO) {  
            val newDrinkList = api.retrofitService.getDrinkList().drinks  
            database.drinkDatabaseDao.insertAll(newDrinkList)  
        }  
    }  
}
```

MainViewModel.kt

repository

bekommt die DrinkApi und die Datenbank
übergeben

loading

wird vom Fragment beobachtet und bestimmt ob
ein Ladebalken oder ein Fehlersymbol angezeigt wird

drinks

stellt die LiveData von der Datenbank zur Verfügung

```
class MainViewModel(application: Application) : AndroidViewModel(application) {

    private val database = getDatabase(application)
    private val repository = AppRepository(DrinkApi, database)

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

    val drinks = repository.drinkList

    init {
        loadData()
    }

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

MainViewModel.kt

loadData()

startet eine **Coroutine** welche einen **Ladebalken** einblendet (LOADING) und in welcher versucht wird **getDrinks()** vom **Repository** durchzuführen funktioniert diese wird der Ladebalken ausgeblendet (DONE)

falsch etwas schief läuft wird überprüft ob man auf **Daten** der **Datenbank** zurückgreifen kann sollten **keine Daten** vorhanden sein wird ein **Fehlersymbol** angezeigt (ERROR) ansonsten wird der Ladebalken ausgeblendet (DONE)

```
class MainViewModel(application: Application) : AndroidViewModel(application) {

    private val database = getDatabase(application)
    private val repository = AppRepository(DrinkApi, database)

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

    val drinks = repository.drinkList

    init {
        loadData()
    }

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


HomeFragment.kt

imageList

ist die **RecyclerView** und bekommt zu Beginn eine leere Liste in den Adapter

loading

wird **beobachtet** und je nach Zustand erscheinen und verschwinden **Ladebalken** und **Fehlersymbol**

drinks

sobald eine Liste von **drinks** zur Verfügung steht wird sie über **submitList** in den **RecyclerViewAdapter** geladen

```
override fun onCreateView(view: View, savedInstanceState: Bundle?) {  
    super.onCreateView(view, savedInstanceState)  
  
    val imageList = binding.imageList  
  
    val imageListAdapter = ItemAdapter(emptyList())  
  
    imageList.adapter = imageListAdapter  
  
    viewModel.loading.observe(  
        viewLifecycleOwner  
    ) { it: ApiStatus!  
        when (it) {  
            ApiStatus.LOADING -> binding.progressBar.visibility = View.VISIBLE  
            ApiStatus.ERROR -> {  
                binding.progressBar.visibility = View.GONE  
                binding.errorImage.visibility = View.VISIBLE  
            }  
            else -> {  
                binding.progressBar.visibility = View.GONE  
                binding.errorImage.visibility = View.GONE  
            }  
        }  
    }  
  
    viewModel.drinks.observe(  
        viewLifecycleOwner  
    ) { it: List<Drink>!  
        imageListAdapter.submitList(it)  
    }  
}
```

ItemAdapter.kt

dataset

wird im **Konstruktor** auf **var** gesetzt damit wir diese verändern können

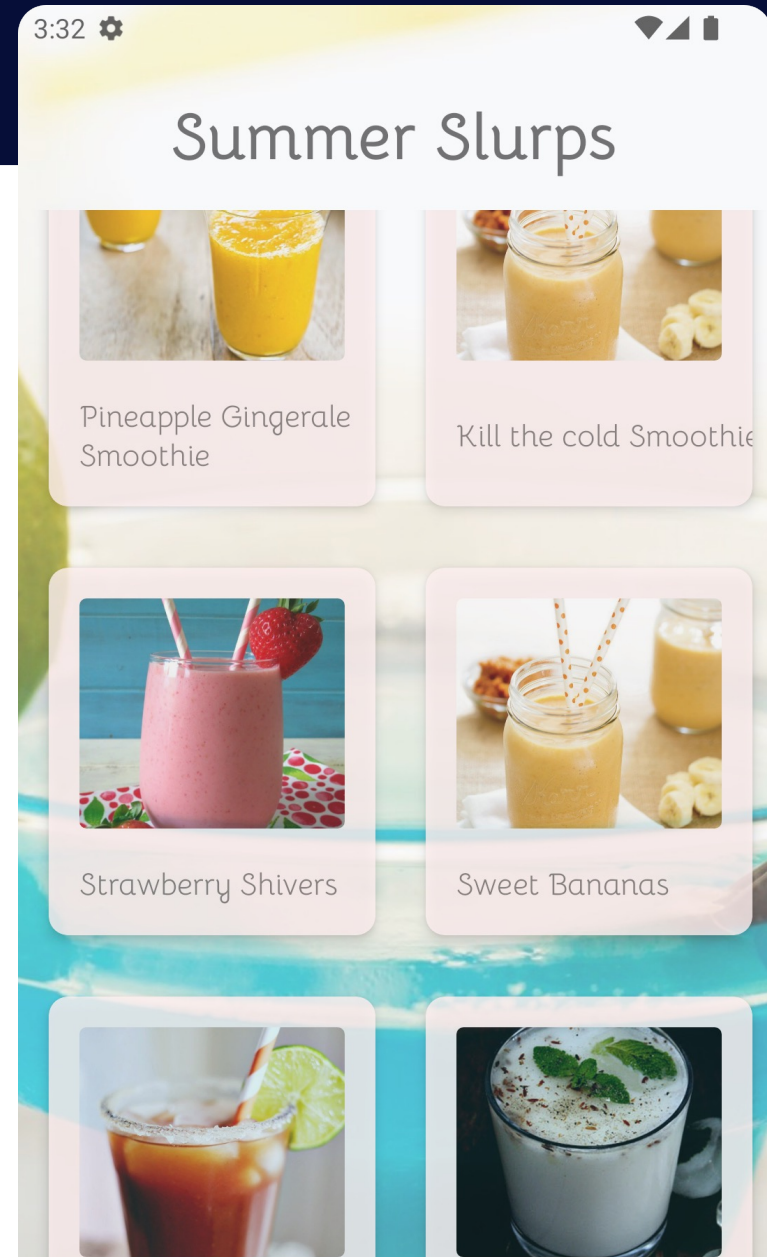
submitList

lädt die **neue Liste** in dataset und mittels **notifyDataSetChanged()** wird die RecyclerView neu erstellt um die **neuen Elemente** auch **anzuzeigen**

```
class ItemAdapter(  
    private var dataset: List<Drink>  
) : RecyclerView.Adapter<ItemAdapter.ItemViewHolder>() {  
  
    @SuppressWarnings("NotifyDataSetChanged")  
    fun submitList(list: List<Drink>) {  
        dataset = list  
        notifyDataSetChanged()  
    }  
}
```

Fertig

Gut gemacht!



Wiederholung

Wiederholung - Was haben wir heute gelernt?

1

Repository Pattern

2

Caching

3

Anwendung



Viel Spaß!

Quelle: <https://www.bonappetit.com/drinks/slideshow/summer-drinks-recipes>