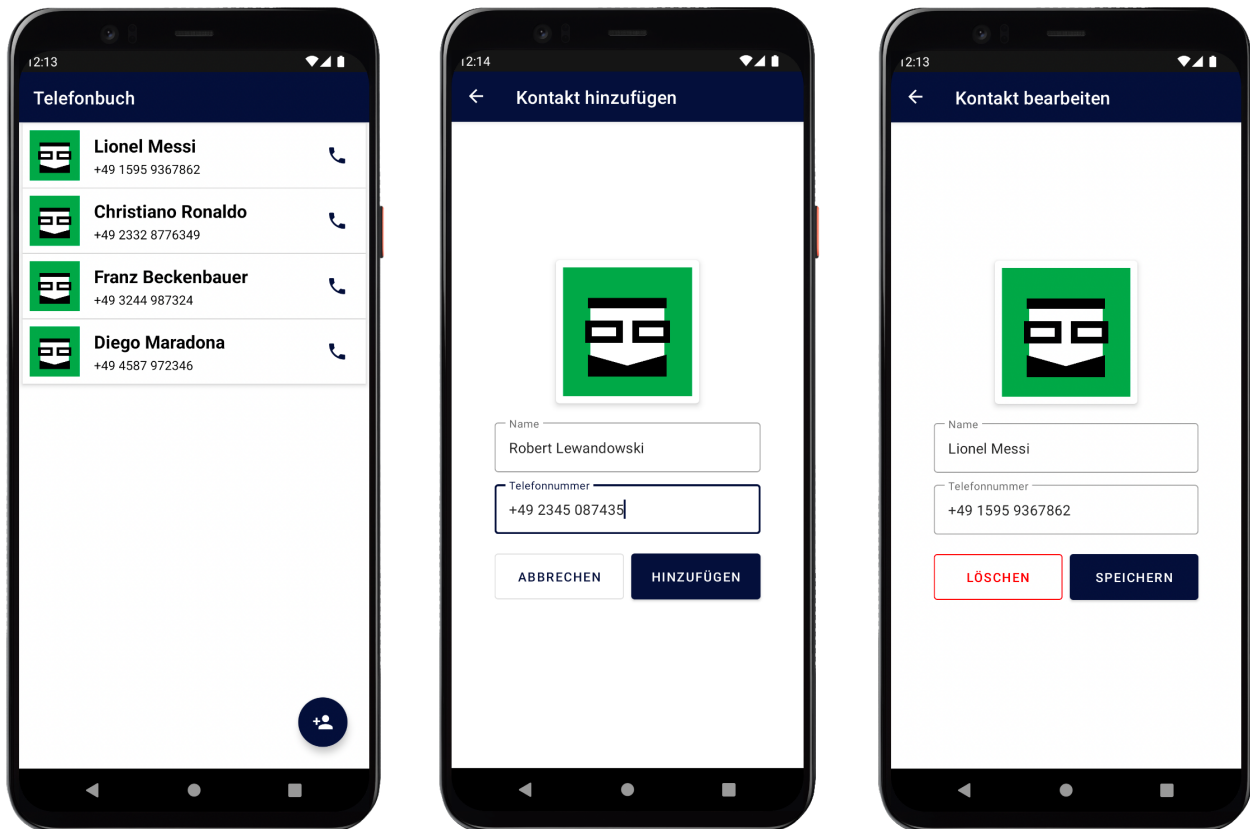


Hinweis: Zu bearbeiten ist Aufgabe 1. Aufgabe 2 ist eine Bonusaufgabe

## 1. TELEFONBUCH - Datenbank

In dieser Aufgabe erstellen wir eine Datenbank für die Telefonbuch-App. Zusätzlich programmieren wir die Funktion zum Hinzufügen und Bearbeiten von Kontakten.



- Öffne das Projekt "Telefonbuch"
- Schau dir die App Vorlage einmal an, enthalten ist bereits die Grundstruktur und der Aufbau mit der MVVM Struktur
- Bevor es losgehen kann, müssen wir die Room Libraries und den Annotationsprozessor einbinden:

In der `build.gradle(Module)` Datei unter `plugins`:

```
id 'kotlin-kapt'
```

In der `build.gradle(Module)` Datei unter `dependencies`:

```
// Room Dependencies
implementation "androidx.room:room-runtime:2.4.2"
kapt "androidx.room:room-compiler:2.4.2"
implementation "androidx.room:room-ktx:2.4.2"
```

- Wir starten und wollen eine Datenbank erstellen. Um eine Datenbank erstellen zu können, müssen wir definieren, wie eine gespeicherte Einheit aussieht
  - Erstelle in `data.datamodels` eine Data Klasse `Contact`. Markiere die Klasse mithilfe der `@Entity` Annotation.
  - Die Klasse beinhaltet eine Klassenvariable `id`. Diese ist so mit der `@PrimaryKey` Annotation versehen, dass der primäre Key automatisch generiert wird
  - Die Klasse beinhaltet zudem eine Variable `name`, in der der Name des Kontakts gespeichert ist und eine Variable `number`, in der die Telefonnummer des Kontakts gespeichert ist
- Um Aktionen in einer Datenbank ausführen zu können, brauchen wir ein `DatabaseAccessObject`, in dem Funktionen für Database Operationen definiert werden. Erstelle ein Interface `ContactDatabaseDao` im package `data.local` mit der Dao Annotation. Innerhalb der Klasse definieren wir folgende Funktionen:
  - Die Funktion `insert` bekommt einen `contact` übergeben und liefert nichts zurück. Sie ist mit einer `@Insert` Annotation versehen. Innerhalb der Annotation wird definiert, was bei einem Konflikt passiert (Ersetze die alte durch die neue Reihe)
  - Die Funktion `update` bekommt einen `contact` übergeben und liefert nichts zurück. Sie ist mit einer `@Update` Annotation versehen
  - Die Funktion `getAll()` bekommt nichts übergeben und liefert LiveData vom Typ "Liste vom Typ `Contact`" zurück. Sie ist mit einer `@Query` Annotation versehen, in der die SQL Anfrage definiert ist: Wähle alle Spalten von `Contact` aus
  - Die Funktion `getById` bekommt eine `id` als `Long` übergeben und liefert LiveData vom Typ `Contact` zurück. Sie ist mit einer `@Query` Annotation versehen, in der die SQL Anfrage definiert ist: Wähle alle Spalten von `Contact` aus, in denen die `id` der übergebenen `id` entspricht
  - Die Funktion `deleteById` bekommt eine `id` als `Long` übergeben und liefert nichts zurück. Sie ist mit einer `@Query` Annotation versehen, in der die SQL Anfrage definiert ist: Lösche die Spalte von `Contact`, in der die `id` der übergebenen `id` entspricht
  - Die Funktion `deleteAll()` bekommt nichts übergeben und liefert nichts zurück. Sie ist mit einer `@Query` Annotation versehen, in der die SQL Anfrage definiert ist: Lösche alle Spalten von `Contact`

Hinweis: Alle Funktionen, die etwas in der Database verändern, brauchen eine Weile und müssen als `suspend` Funktion definiert werden

- Jetzt, wo uns die Werkzeuge zum Bearbeiten der Database durch die DAO zur Verfügung stehen, können wir die Room Database aufsetzen. Erstelle dafür eine weitere Kotlin Datei `ContactDatabase` im gleichen package `data.local`
  - Erstelle innerhalb der Datei eine `abstract class ContactDatabase`, die von `RoomDatabase()` erbt. Sie beinhaltet eine `abstract` Variable `contactDatabaseDao` vom Typ `ContactDatabaseDao`. Sie ist zudem mit der `@Database` Annotation versehen, in der die `entities` als vom Typ `Contact` und die `version` als `1` definiert werden.
  - Erstelle außerdem eine private `lateinit` Variable `INSTANCE` vom Typ `ContactDatabase`, in der später eine Instanz der Datenbank gespeichert wird. Um die Instanz zu erstellen und zuzuweisen, brauchen wir eine Funktion:
  - Schreibe die Funktion `getDatabase`. Sie bekommt einen `Context` übergeben und liefert ein Objekt vom Typ `ContactDatabase` zurück. Am Ende der Funktion liefert sie die `INSTANCE` Variable zurück. Vor dem `return` statement passiert jedoch Folgendes in einem `synchronized` block\*: Sie überprüft, ob die `INSTANCE` Variable bereits initialisiert wurde. Falls nicht, soll `INSTANCE` initialisiert werden, und zwar mit `Room.databaseBuilder(...).build()`.

\*Hinweis:

Ein `synchronized` block verhindert, dass der Code mehrmals gleichzeitig ausgeführt wird, z.B. wenn er aus verschiedenen Threads heraus aufgerufen wird und dadurch unbeabsichtigt mehrere Datenbanken erstellt werden. Er ist folgendermaßen aufgebaut:

```
synchronized(ContactDatabase::class.java) {  
    ...  
}
```

- In der Klasse `Repository` können wir die Daten aus der Datenbank jetzt mithilfe der DAO Funktionen verwalten und bearbeiten. Hier werden die Daten ebenfalls für den Rest der App verfügbar gemacht.
  - Füge eine private Variable `database` vom Typ `ContactDatabase` in den Konstruktor ein. Hier erhalten wir die Datenbank, auf die wir im weiteren Verlauf zugreifen
  - Erstelle eine `LiveData` Variable `contactList` vom Typ "Liste vom Typ `Contact`". Sie wird durch die `getAll()` Funktion des `contactDatabaseDao` der `database` befüllt
  - Schreibe drei `suspend` Funktionen, `insert` zum Einfügen, `update` zum Aktualisieren und `delete` zum Löschen von Kontakten. Jede der Funktionen bekommt einen Kontakt übergeben und ruft damit die entsprechende Funktion des `contactDatabaseDao` der `database` auf. Um mögliche Fehler aufzufangen,

geschieht dies jeweils in einem `try` block. Im `catch` Teil wird die Fehlermeldung mit `Log` ausgegeben.

- In unserem `Main ViewModel` können wir jetzt die Zustände und die Funktionalität der App verwalten.
  - Erstelle mithilfe der Funktion `getDatabase` eine Datenbank Instanz und speichere sie in einer Variablen ab. Als Kontext für die Funktion `getDatabase` übergibst du hier einfach den Parameter `application`
  - Erstelle eine neue `Repository` Instanz (Objekt) mit der eben erstellten Datenbank und speichere sie in einer Variablen ab
  - Hole die Kontaktliste aus der `Repository` Instanz und speichere sie in einer eigenen Variable ab
  - Erstelle eine verschachtelte `LiveData` Variable `completedAction` vom Typ `Boolean`. Diese Variable speichert den Status einer Aktion, also ob der Kontakt bereits erfolgreich hinzugefügt/aktualisiert/gelöscht wurde. Der Sinn dahinter ist, dass die `Add / Edit` Fragmente automatisch wieder zurück zum `Main` Fragment navigieren sollen, sobald die Aktion abgeschlossen wurde.
  - Schreibe drei Funktionen `insertContact`, `updateContact` und `deleteContact`. Jede dieser Funktionen bekommt ein `Contact` Objekt übergeben und liefert nichts zurück. Jede Funktion startet jeweils eine `Coroutine`, die so lange läuft, wie das `ViewModel` existiert. Innerhalb der `Coroutine` ruft die Funktion die richtige `Repository`-Funktion zum Erfüllen ihrer Aufgabe auf und setzt anschließend die Aktion auf "erledigt" (über `completedAction`)
  - Wir brauchen noch eine Funktion `unsetComplete()`, die nichts bekommt und nichts zurückgibt und nur dafür verantwortlich ist, die Aktion wieder auf "nicht erledigt" zu setzen. Schreibe diese Funktion
- Die Datenbank und die Funktion der App sind definiert, jetzt kümmern wir uns im letzten Schritt um die richtige Darstellung der Informationen. Dafür bearbeiten wir die drei Fragmente für die drei Screens der App
  - Öffne das `MainFragment`, welches alle Kontakte in einer Liste darstellt
    - In der Funktion `onViewCreated` beobachten wir die Kontaktliste und bei jeder Änderung wird der `RecyclerView rvContacts` ein neuer `ContactAdapter` zugewiesen, mit der Kontaktliste als Parameter.
    - Damit der Adapter die Liste verwenden kann, muss in der Klasse `ContactAdapter` der Typ des `dataset` von `Any` zu `Contact` geändert werden. In der Funktion `onBindViewHolder` muss in den Views aus dem `holder` noch der Name und die Telefonnummer gesetzt werden. Außerdem ist die `contactId` momentan immer `0`. Hier muss ebenfalls die `id` aus dem aktuellen Kontakt zugewiesen werden.


Im `ContactAdapter` ist bereits ein Click Listener gegeben, der bei einem Klick auf die `CardView` eine Navigation zum Edit Fragment auslöst. Dabei wird die ID des aktuellen Kontakts übergeben, die wir jetzt im Edit Fragment benutzen können.

- Öffne das `EditFragment`, welches die Bearbeitung eines Kontaktes darstellt
  - Wir arbeiten in der Funktion `onViewCreated`
  - Hier wird die ID des aktuellen Kontakts bereits aus den Arguments geholt. Hole den dazugehörigen Kontakt aus der Kontaktliste des ViewModel und speichere ihn in einer Variable  
Hinweis: nutze dafür die `find{}` Funktion
  - Prüfe, ob es einen Kontakt mit der ID gab (also ob die Variable nicht `null` ist) und wenn ja, setze mit `binding` den Text der beiden `editTextInput` auf den Namen bzw die Telefonnummer aus dem Kontakt.
  - Beobachte die `completedAction` Variable (also den Status der Aktion). Falls sie `true` wird, finde den Nav Controller und navigiere zum Main Fragment. Rufe in dem Fall außerdem die Funktion `unsetComplete` aus dem ViewModel auf
  - Setze einen Click Listener auf den `editBtnSave` Button, in dem zuerst geprüft wird, ob der aktuelle Kontakt `null` ist. Wenn nicht, wird der Name und die Nummer des Kontakts mit den eingegebenen Werten aktualisiert. Zum Speichern wird die `updateContact` Funktion des ViewModel aufgerufen.
  - Setze einen Click Listener auf den `editBtnDelete` Button, in dem zuerst geprüft wird, ob der aktuelle Kontakt `null` ist. Wenn nicht, wird die `deleteContact` Funktion des ViewModel aufgerufen.
- Öffne das `AddFragment`, welches das Hinzufügen neuer Kontakte darstellt
  - Wir arbeiten in der Funktion `onViewCreated`
  - Beobachte die `completedAction` Variable (also den Status der Aktion). Falls sie `true` wird, finde den Nav Controller und navigiere zum Main Fragment. Rufe in dem Fall außerdem die Funktion `unsetComplete` aus dem ViewModel auf
  - Setze einen Click Listener auf den `addBtnAdd` Button, in dem ein neuer Kontakt mit dem eingegebenen Namen und der eingegebenen Telefonnummer eingefügt wird. Nutze dafür die ViewModel Funktion `insertContact`.

Hinweis:

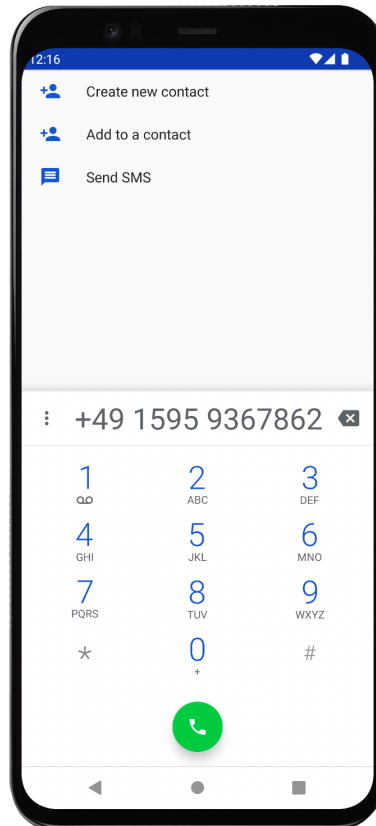
Übergebe beim Erstellen eines neuen Kontaktes keinen Wert für die ID, sondern nur für den Namen und die Nummer, da die ID automatisch generiert werden soll. Dies erreichst du, indem du die Parameter benennst, z.B.: `val person = Person(name = "max", age = 33)`

- Die App mitsamt Datenbank sollte jetzt funktionieren! Führe die App aus und teste alle Funktionen. Dank der Datenbank werden alle Kontakte gespeichert, das heißt, selbst wenn du die App schließt, das Handy ausschaltest o.ä. bleiben die Kontakte beim erneuten Ausführen in der Liste!

Viel Erfolg! 

## 2. TELEFONBUCH - Bonus

Baue einen Dial Intent in die App ein, sodass bei einem Klick auf das Telefonhörersymbol die folgende Anrufansicht mit der Telefonnummer aufgerufen wird.



### BONUS:

Erweitere die App und baue weitere Features ein, wie z.B. die Möglichkeit, beim Hinzufügen eines Kontaktes ein Bild zuweisen zu können.

Viel Erfolg! 