

Modul 3 – Android App Entwicklung mit Kotlin

Room II



Gliederung

- nochmal Room
- Database, DAO und Entities
- Gästeliste wird verbessert

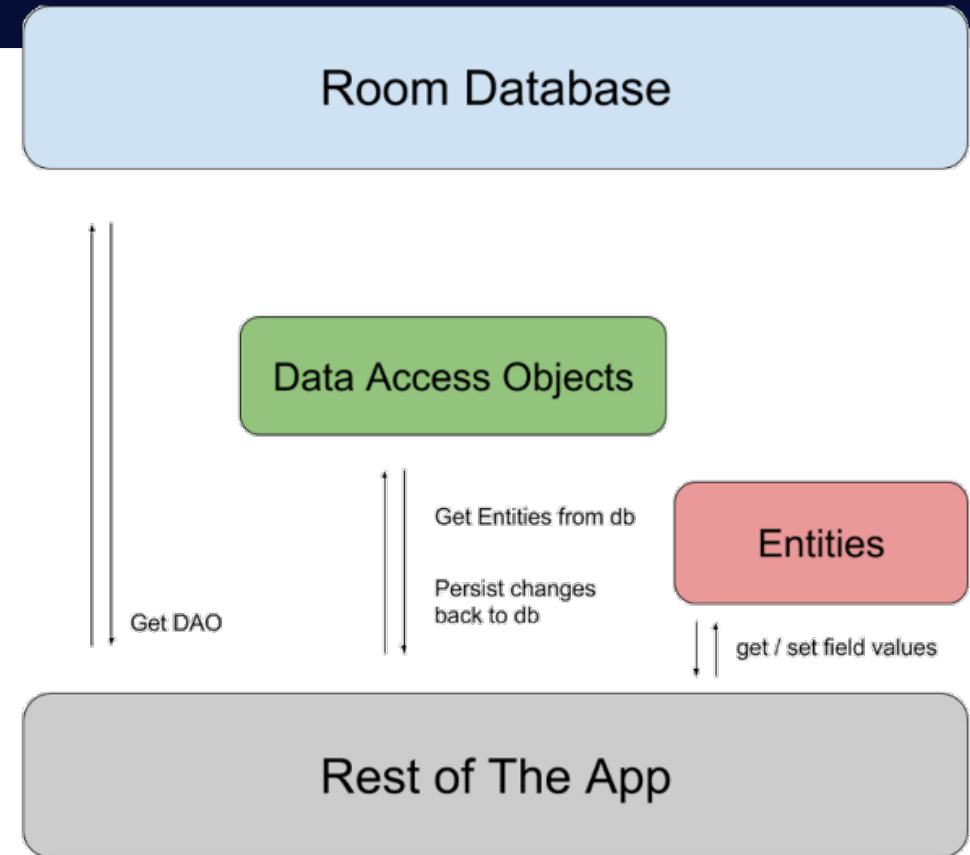


Quelle: <https://www.kangarooselfstorage.co.uk/wp-content/uploads/2018/12/Business-Storage-Archive-Storage-1-1200x900.jpg>

Room

Dient als Abstraktionsschicht und vereinfacht die Kommunikation mit **SQLite**
(ähnlich wie retrofit für API Calls)

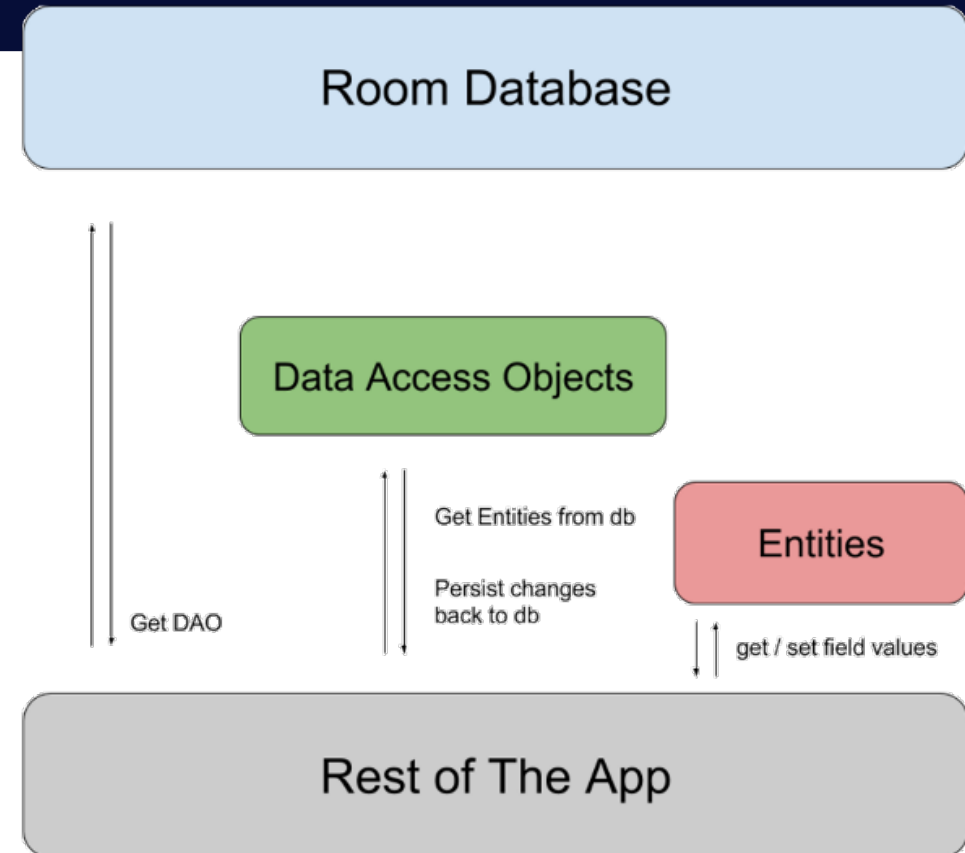
- Überprüft SQL Abfragen
- erspart mittels **Annotations** viel Boilerplate Code
- Optimierte Datenbankmigrationspfade



Quelle: https://developer.android.com/images/training/data-storage/room_architecture.png

Room

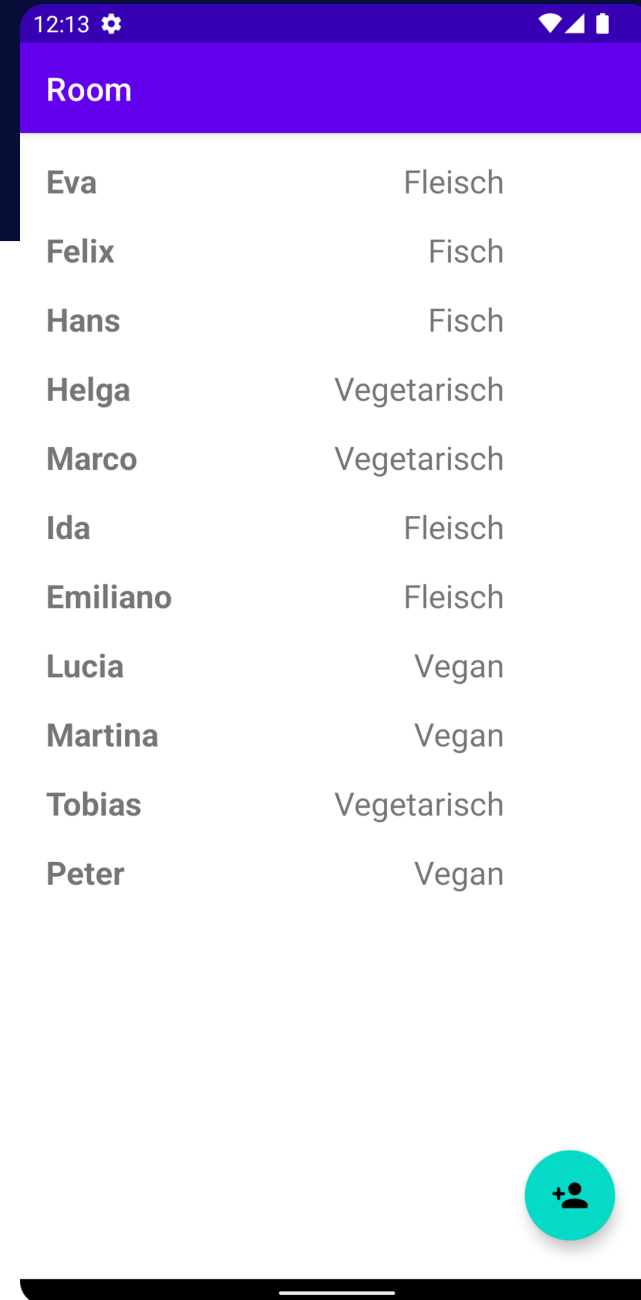
- Datenbank-Klasse
 - enthält die Datenbank und dient als Hauptzugriffspunkt
- Entities (Daten Klassen)
 - stellen Tabellen der Datenbank dar
- DatabaseAccessObject (DAO)
 - stellt Methoden für Datenbankoperationen bereit



Quelle: https://developer.android.com/images/training/data-storage/room_architecture.png

Gästeliste

- Daten Klasse - Guest
 - mit Variablen für ID, Namen und bevorzugtem Essen
- Datenbankklasse
 - mit Singletonpattern damit es auch wirklich nur eine Datenbank in der App gibt
- DatabaseAccessObject (DAO)
 - mit Lese- und Schreibfunktionen um mit der Datenbank zu arbeiten

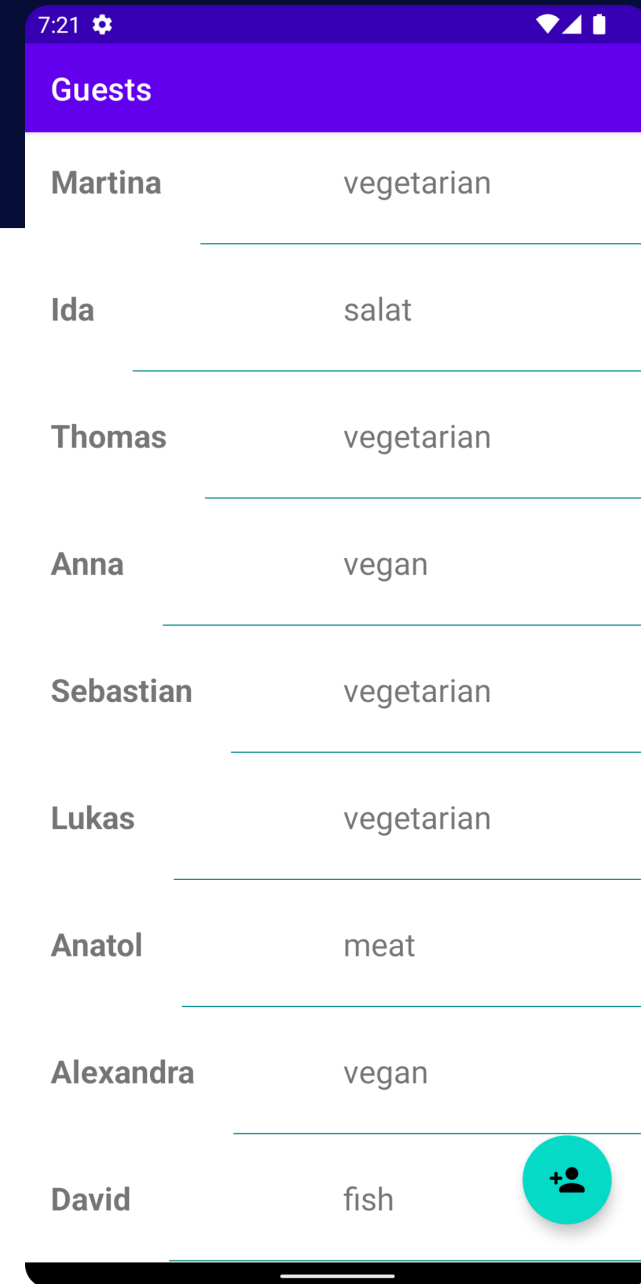


The screenshot shows an Android application interface. At the top, there is a status bar with the time 12:13, a settings gear icon, and signal/battery icons. Below the status bar is a purple header bar with the word "Room" in white. The main content area is a table with two columns: names and food preferences. The names are in bold, and the food preferences are in regular weight. At the bottom right, there is a red circular button with a white plus sign and a person icon. The bottom of the screen shows a black home indicator bar.

Room	
Eva	Fleisch
Felix	Fisch
Hans	Fisch
Helga	Vegetarisch
Marco	Vegetarisch
Ida	Fleisch
Emiliano	Fleisch
Lucia	Vegan
Martina	Vegan
Tobias	Vegetarisch
Peter	Vegan

Verbesserungen

- Layout
- Add Guest
eigenes Fragment um Gäste hinzuzufügen
- Edit Guest
eigenes Fragment um Datenbankeinträge zu bearbeiten oder zu löschen



Add / Edit Guest

7:23 ⚙️

← Add

name

Georg

just desert ▼

CANCEL SAVE

7:23 ⚙️

← Edit

name

Anna

vegan ▼

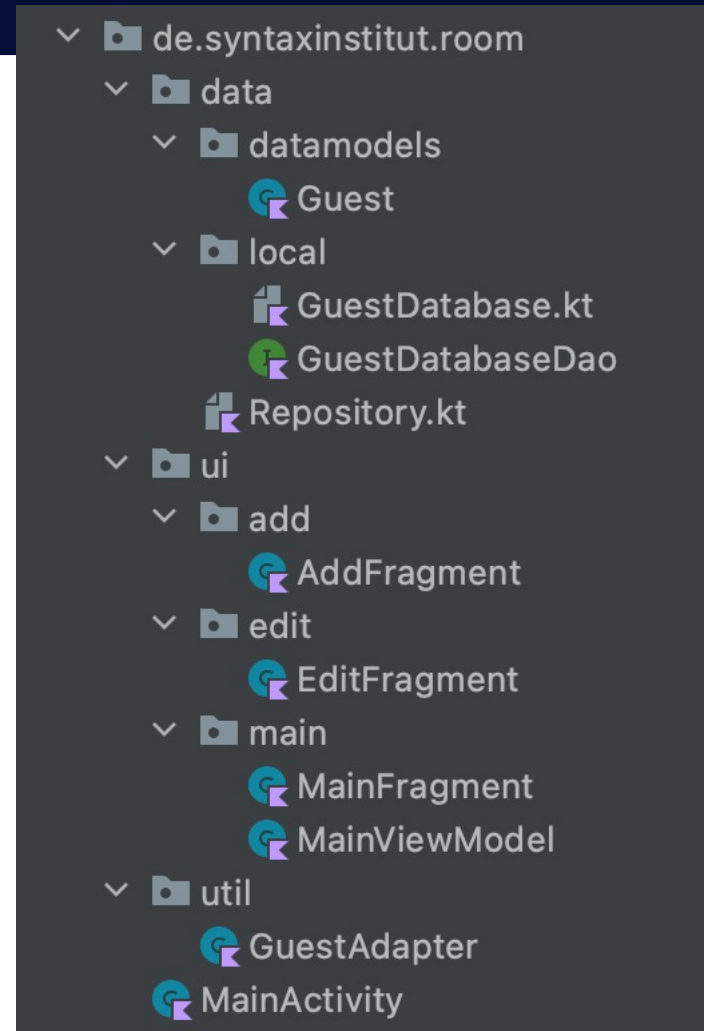
DELETE SAVE

Struktur

MVVM Struktur

Die zusätzlichen Screens bekommen ein eigenes
Package

Da sich hinter den Screens nicht sehr viel Logik
versteckt werden sie sich alle das **MainViewModel**
teilen



GuestDatabaseDao.kt

`update(guest: Guest)`

wird verwendet um Datenbankeinträge zu
bearbeiten

`deleteById(id: Long)`

wird verwendet um einen bestimmten
Eintrag zu löschen

```
@Dao
interface GuestDatabaseDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(guest: Guest)

    @Update
    suspend fun update(guest: Guest)

    @Query(value: "SELECT * FROM Guest")
    fun getAll(): LiveData<List<Guest>>

    @Query(value: "SELECT * from Guest WHERE id = :key")
    fun getById(key: String): LiveData<Guest>

    @Query(value: "DELETE from Guest WHERE id = :id")
    suspend fun deleteById(id: Long)

    @Query(value: "DELETE from Guest")
    suspend fun deleteAll()
}
```

Repository.kt

guestList

Speichert das Ergebnis der `getAll()` Anfrage des DAO

insert(guest: Guest)

Versucht mittels der `insert` Funktion des DAO einen neuen Gast in die Datenbank einzufügen

update(guest: Guest)

Verwendet die `update` Funktion des DAO

delete(guest: Guest)

Verwendet die `delete` Funktion des DAO

```
const val TAG = "Repository"

class Repository(private val database: GuestDatabase) {

    val guestList: LiveData<List<Guest>> = database.guestDatabaseDao.getAll()

    suspend fun insert(guest: Guest) {
        try {
            database.guestDatabaseDao.insert(guest)
        } catch (e: Exception) {
            Log.d(TAG, msg: "Failed to insert into Database: $e")
        }
    }

    suspend fun update(guest: Guest) {
        try {
            database.guestDatabaseDao.update(guest)
        } catch (e: Exception) {
            Log.d(TAG, msg: "Failed to update Database: $e")
        }
    }

    suspend fun delete(guest: Guest) {
        try {
            database.guestDatabaseDao.deleteById(guest.id)
        } catch (e: Exception) {
            Log.d(TAG, msg: "Failed to delete from Database: $e")
        }
    }
}
```

MainViewModel.kt

Hier werden die `insert`, `update` und `deleteGuest` Funktionen des Repositories in Coroutines aufgerufen

`complete` wird von den Fragmenten beobachtet und lässt sie wissen wann eine Operation fertig ist

`unsetComplete()` erlaubt Fragmenten den Wert von `complete` auf `false` zu setzen

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

    private val database = getDatabase(application)
    private val repository = Repository(database)

    val guestList = repository.guestList

    private val _complete = MutableLiveData<Boolean>()
    val complete: LiveData<Boolean>
        get() = _complete

    fun insertGuest(guest: Guest) {
        viewModelScope.launch { this: CoroutineScope
            repository.insert(guest)
            _complete.value = true
        }
    }

    fun updateGuest(guest: Guest) {
        viewModelScope.launch { this: CoroutineScope
            repository.update(guest)
            _complete.value = true
        }
    }

    fun deleteGuest(guest: Guest) {
        viewModelScope.launch { this: CoroutineScope
            Log.d( tag: "ViewModel", msg: "Calling repository delete with ${guest.id}")
            repository.delete(guest)
            _complete.value = true
        }
    }

    fun unsetComplete() {
        _complete.value = false
    }
}
```

AddFragment.kt

die LiveData Variable **complete** wird beobachtet und sobald sie **true** ist wird die Navigation gestartet und anschließend wird complete wieder auf **false** gesetzt

getValuesandSave() holt die Werte aus der Texteingabe und dem Spinner, erstellt einen neuen Gast und gibt ihn an **insertGuest()** vom viewModel weiter

```
@SuppressWarnings("ClickableViewAccessibility")
override fun onCreateView(view: View, savedInstanceState: Bundle?) {
    super.onCreateView(view, savedInstanceState)

    viewModel.complete.observe(
        viewLifecycleOwner,
        Observer { it: Boolean! } {
            if (it) {
                findNavController().navigate(AddFragmentDirections.actionAddFragmentToMainFragment())
                viewModel.unsetComplete()
            }
        }
    )

    binding.foodSpinner.setOnTouchListener { view, motionEvent -> val imm: InputMethodManager =
        requireContext().getSystemService(Context.INPUT_METHOD_SERVICE) as InputMethodManager
        imm.hideSoftInputFromWindow(view.windowToken, flags: 0) ^setOnTouchListener
    }

    binding.saveButton.setOnClickListener { it: View! }
        getValuesAndSave()
    }

    binding.cancelButton.setOnClickListener { it: View! }
        findNavController().navigate(AddFragmentDirections.actionAddFragmentToMainFragment())
    }
}

private fun getValuesAndSave() {
    val name = binding.nameEdit.text.toString()
    val food = binding.foodSpinner.selectedItem.toString()
    val newGuest = Guest(name = name, food = food)
    viewModel.insertGuest(newGuest)
}
```

EditFragment.kt

In der Navigation wird eine `id` übergeben mit deren Hilfe der geklickte Guest in der `guestList` gefunden wird

Falls der Guest gefunden wurde wird das `EditText` und der `Spinner` auf die passenden Werte gesetzt

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

    arguments?.let { it: Bundle?
        guestId = it.getLong( key: "guestId")
    }
}

override fun onCreateView(
    inflater: LayoutInflater,
    container: ViewGroup?,
    savedInstanceState: Bundle?,
): View? { ... }

override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    val guest = viewModel.guestList.value?.find { it.id == guestId }

    if (guest != null) {
        binding.editNameEdit.setText(guest.name)
        val index = resources.getStringArray(R.array.food_array).indexOf(guest.food)
        binding.editFoodSpinner.setSelection(index)
    }

    viewModel.complete.observe(
        viewLifecycleOwner,
        Observer { it: Boolean?
            if (it) {
                findNavController().navigate(EditFragmentDirections.actionEditFragmentToMainFragment())
                viewModel.unsetComplete()
            }
        }
    )

    binding.editSaveButton.setOnClickListener { it: View?
        if (guest != null) {
            getValuesAndUpdate(guest)
        }
    }

    binding.editDeleteButton.setOnClickListener { it: View?
        if (guest != null) {
            viewModel.deleteGuest(guest)
        }
    }
}
```

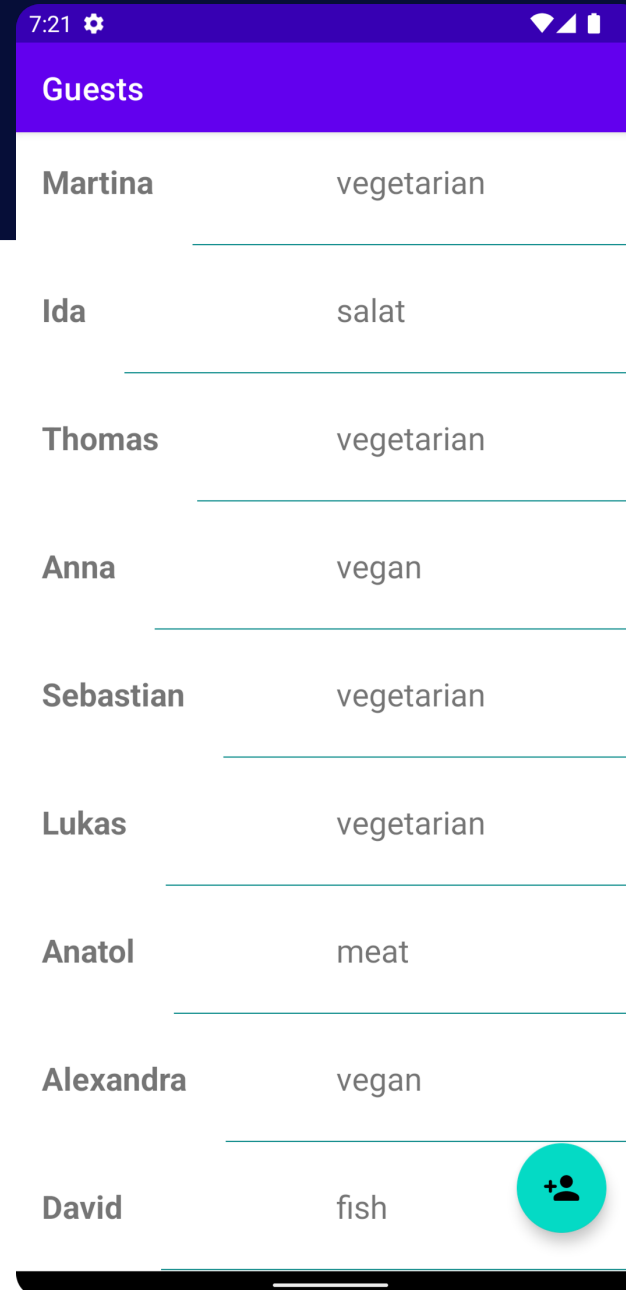
EditFragment.kt

ein Klick auf den SaveButton führt
`getValuesAndUpdate()` aus wo die neuen Werte in
den Guest gespeichert werden um diese
Änderungen dann über `updateGuest()` des
viewModels auch in die Datenbank zu schreiben

```
private fun getValuesAndUpdate(guest: Guest) {  
    guest.name = binding.editNameEdit.text.toString()  
    guest.food = binding.editFoodSpinner.selectedItem.toString()  
    viewModel.updateGuest(guest)  
}
```

Fertig

Gut gemacht!

A smartphone mockup displaying a guest list application. The status bar at the top shows the time 7:21, a settings gear icon, and signal/battery icons. The app has a purple header titled 'Guests'. Below it is a list of guests with their names and dietary preferences. A teal circular button with a plus and person icon is at the bottom right.

Guests	
Martina	vegetarian
Ida	salat
Thomas	vegetarian
Anna	vegan
Sebastian	vegetarian
Lukas	vegetarian
Anatol	meat
Alexandra	vegan
David	fish

Wiederholung

Wiederholung - Was haben wir heute gelernt?

1

nochmal Room

2

Database, DAO, Entities

3

Gästeliste verbessert

Viel Spaß!



Quelle: <https://cheapandcheerfulcooking.com/vegan-schnitzel-mit-kartoffelsalat/>