LAPORAN RESMI

MODUL IV

FRAGMENT DAN ROOM DATABASE

PEMROGRAMAN BERGERAK

NAMA	**:**	**SEVIN DIAS ANDIKA**
N.R.P	**:**	**210441100105**
DOSEN	**:**	**Ir. ACH. DAFID, S.T., M.T.**
ASISTEN	**:**	**DAVID NASRULLOH**
TGL PRAKTIKUM	**:**	**07 APRIL 2023**

Disetujui : .. APRIL 2022
Asisten

DAVID NASRULLOH
190441100060

LABORATORIUM BISNIS INTELIJEN SISTEM
PRODI SISTEM INFORMASI
JURUSAN TEKNIK INFORMATIKA
FAKULTAS TEKNIK
UNIVERSITAS TRUNOJOYO MADURA

BAB I
PENDAHULUAN

1.1 Latar Belakang

Fragman adalah bagian UI aplikasi Anda yang dapat digunakan kembali. Sebuah fragmen menentukan dan mengelola tata letaknya sendiri, memiliki siklus proses sendiri, serta dapat menangani peristiwa inputnya sendiri. Fragmen tidak dapat berjalan sendiri. Fragmen harus *dihosting* oleh aktivitas atau fragmen lain. Hierarki tampilan fragmen menjadi bagian dari, atau *dilampirkan ke*, hierarki tampilan host.

Android memperkenalkan fragmen di Android 3.0 (API level 11), terutama untuk mendukung desain UI yang lebih dinamis dan fleksibel pada layar besar, seperti tablet. Karena layar tablet jauh lebih besar daripada layar handset, maka lebih banyak ruang untuk mengombinasikan dan bertukar komponen UI. Fragmen memungkinkan desain seperti itu tanpa perlu mengelola perubahan kompleks pada hierarki tampilan.

Untuk membuat fragmen, kita harus membuat subclass Fragment (atau subclass-nya yang ada). Class Fragment memiliki kode yang mirip seperti Activity. Class ini memiliki metode callback yang serupa dengan aktivitas, seperti onCreate(), onStart(), onPause(), dan onStop(). Sebenarnya, jika kita mengonversi aplikasi Android saat ini untuk menggunakan fragmen, kita mungkin cukup memindahkan kode dari metode callback aktivitas ke masingmasing metode callback fragmen

1.2 Tujuan

- ➢ Mahasiswa dapat membuat aplikasi dengan menggunakan fragment.
- ➢ Mahasiswa dapat membuat Aplikasi dengan database.

BAB II
DASAR TEORI

2.1 Fragment

FRAGMENT

Fragment mewakili perilaku atau bagian dari antarmuka pengguna dalam FragmentActivity. Kita bisa mengombinasikan beberapa fragmen dalam satu aktivitas untuk membangun UI multipanel dan menggunakan kembali sebuah fragmen dalam beberapa aktivitas. Kita bisa menganggap fragmen sebagai bagian modular dari aktivitas, yang memiliki daur hidup sendiri, menerima kejadian masukan sendiri, dan yang bisa kita tambahkan atau hapus saat aktivitas berjalan (semacam "subaktivitas" yang bisa digunakan kembali dalam aktivitas berbeda).

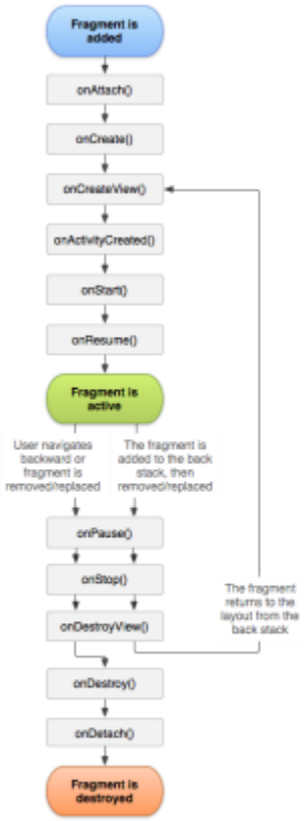
Fragmen harus selalu tersemat dalam aktivitas dan daur hidup fragmen secara langsung dipengaruhi oleh daur hidup aktivitas host-nya. Misalnya, saat aktivitas dihentikan sementara, semua fragmen di dalamnya juga dihentikan sementara, dan bila aktivitas dimusnahkan, semua fragmen juga demikian. Akan tetapi, saat aktivitas berjalan (dalam status daur hidup dilanjutkan), Kita bisa memanipulasi setiap fragmen secara terpisah, seperti menambah atau membuangnya. Saat melakukan transaksi fragmen, Kita juga bisa menambahkannya ke back-stack yang dikelola oleh aktivitas—setiap entri backstack merupakan catatan transaksi fragmen yang terjadi. Dengan back-stack pengguna dapat membalikkan transaksi fragmen (mengarah mundur), dengan menekan tombol Kembali.

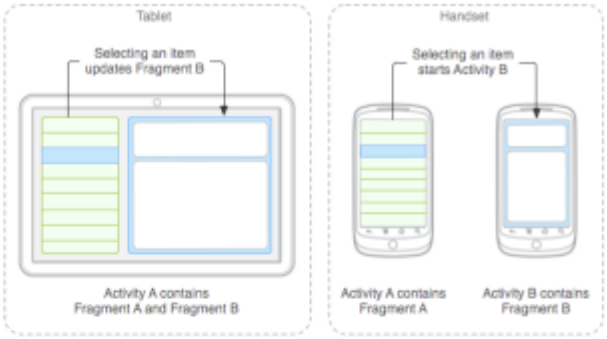
Bila kita menambahkan fragmen sebagai bagian dari layout aktivitas, fragmen tersebut berada di ViewGroup di dalam hierarki tampilan aktivitas dan fragmen menentukan layout tampilannya sendiri. Kita bisa menyisipkan fragmen ke dalam layout aktivitas dengan mendeklarasikan fragmen dalam file layout aktivitas, sebagai elemen , atau dari kode aplikasi kita dengan menambahkannya ke ViewGroup yang ada.

Kita akan membahas cara membuat aplikasi menggunakan fragmen, termasuk cara fragmen mempertahankan statusnya bila ditambahkan ke back-stack aktivitas, berbagi kejadian dengan aktivitas, dan fragmen lain dalam aktivitas, berkontribusi pada bilah aksi aktivitas, dan lainnya.

Filosofi Desain

Android memperkenalkan fragmen di Android 3.0 (API level 11), terutama untuk mendukung desain UI yang lebih dinamis dan fleksibel pada layar besar, seperti tablet. Karena layar tablet jauh lebih besar daripada layar handset, maka lebih banyak ruang untuk mengombinasikan dan bertukar komponen UI. Fragmen memungkinkan desain seperti itu tanpa perlu mengelola perubahan kompleks pada hierarki tampilan. Dengan membagi layout aktivitas menjadi beberapa fragmen, kita bisa mengubah penampilan aktivitas saat





Gambar 1. Contoh bagaimana dua modul UI yang ditentukan oleh fragmen bisa digabungkan ke dalam satu aktivitas untuk d esain tablet, namun dipisahkan untuk desain handset.

Misalnya—untuk melanjutkan contoh aplikasi berita —aplikasi bisa menyematkan dua fragmen dalam *Aktivitas A*, saat berjalan pada perangkat berukuran tablet. Akan tetapi, pada layar berukuran handset, ruang untuk kedua fragmen tidak sehingga *Aktivitas A* hanya menyertakan fragmen untuk daftar artikel, dan saat pengguna memilih artikel, *Aktivitas B* akan dimulai, termasuk fragmen kedua untuk membaca artikel. Sehingga, aplikasi mendukung tablet dan handset dengan menggunakan kembali fragmen dalam kombinasi berbeda, seperti diilustrasikan dalam gambar 1.

Gambar 2. Daur hidup fragmen (saat aktivitasnya berjalan).

waktu proses dan mempertahankan perubahan itu di back-stack yang dikelola oleh aktivitas. Mode-mode tersebut kini tersedia secara luas melalui library dukungan fragmen. Misalnya, aplikasi berita bisa menggunakan satu fragmen untuk menampilkan daftar artikel di sebelah kiri dan fragmen lainnya untuk menampilkan artikel di sebelah kanan—kedua fragmen ini muncul di satu aktivitas, berdampingan, dan masing-masing fragmen memiliki serangkaian metode callback

daur hidup dan menangani kejadian masukan penggunanya sendiri. Sehingga, sebagai ganti menggunakan satu aktivitas untuk memilih artikel dan aktivitas lainnya untuk membaca artikel, pengguna bisa memilih artikel dan membaca semuanya dalam aktivitas yang sama, sebagaimana diilustrasikan dalam layout tablet pada gambar 1.

Kita harus mendesain masing-masing fragmen sebagai komponen aktivitas modular dan bisa digunakan kembali. Yakni, karena setiap fragmen mendefinisikan layoutnya dan perilakunya dengan callback daur hidupnya sendiri, kita bisa memasukkan satu fragmen dalam banyak aktivitas, sehingga kita harus mendesainnya untuk digunakan kembali dan mencegah memanipulasi satu fragmen dari fragmen lain secara langsung. Ini terutama penting karena dengan fragmen modular kita bisa mengubah kombinasi fragmen untuk ukuran layar yang berbeda. Saat mendesain aplikasi untuk mendukung tablet maupun handset, kita bisa menggunakan kembali fragmen dalam konfigurasi layout yang berbeda untuk mengoptimalkan pengalaman pengguna berdasarkan ruang layar yang tersedia. Misalnya, pada handset, fragmen mungkin perlu dipisahkan untuk menyediakan UI panel tunggal bila lebih dari satu yang tidak cocok dalam aktivitas yang sama. cukup,

Membuat Fragmen

Untuk membuat fragmen, kita harus membuat subclass Fragment (atau subclass-nya yang ada). Class Fragment memiliki kode yang mirip seperti Activity. Class ini memiliki metode callback yang serupa dengan aktivitas, seperti onCreate(), onStart(), onPause(), dan onStop(). Sebenarnya, jika kita mengonversi aplikasi Android saat ini untuk menggunakan fragmen, kita mungkin cukup memindahkan kode dari metode callback aktivitas ke masingmasing metode callback fragmen. Biasanya, kita harus mengimplementasikan setidaknya metode daur hidup berikut ini:

onCreate() : Sistem akan memanggilnya saat membuat fragmen. Dalam implementasi, kita harus melakukan inisialisasi komponen penting dari fragmen yang ingin dipertahankan saat fragmen dihentikan sementara atau dihentikan, kemudian dilanjutkan.

onCreateView() : Sistem akan memanggilnya saat fragmen menggambar antarmuka penggunanya untuk yang pertama kali. Untuk menggambar UI fragmen, kita harus mengembalikan View dari metode ini yang menjadi root layout fragmen. Hasil yang dikembalikan bisa berupa null jika fragmen tidak menyediakan UI.

onPause() : Sistem akan memanggil metode ini sebagai indikasi pertama bahwa pengguna sedang meninggalkan fragmen kita (walau itu tidak selalu berarti fragmen sedang dimusnahkan). Di sinilah biasanya kita harus mengikat perubahan yang harus dipertahankan di luar sesi pengguna saat ini (karena pengguna mungkin tidak akan kembali).

Kebanyakan aplikasi harus mengimplementasikan setidaknya tiga metode ini untuk setiap fragmen, tetapi ada beberapa metode callback lain yang juga harus kita gunakan untuk menangani berbagai tahap daur hidup fragmen. Perhatikan bahwa kode yang mengimplementasikan aksi daur hidup dari komponen dependen harus ditempatkan di komponen itu sendiri, bukan dalam implementasi callback fragmen.

Menambahkan antarmuka pengguna

Fragmen biasanya digunakan sebagai bagian dari antarmuka pengguna aktivitas dan menyumbangkan layoutnya sendiri ke aktivitas. Untuk menyediakan layout fragmen, kita harus mengimplementasikan metode callback onCreateView(), yang dipanggil sistem Android bila tiba saatnya fragmen menggambar layoutnya. Implementasi kita atas metode ini harus mengembalikan View yang menjadi root layout fragmen. Untuk mengembalikan layout dari onCreateView(), Kita bisa memekarkannya dari resource layout yang ditentukan di XML. Untuk membantu melakukannya, onCreateView() menyediakan objek LayoutInflater. Misalnya, terdapat subclass Fragment yang memuat layout dari file example_fragment.xml:

```
class ExampleFragment : Fragment() {
 override fun onCreateView( inflater:
LayoutInflater, container: ViewGroup?,
savedInstanceState: Bundle?
 ): View {
```

```
// Inflate the layout for this fragment
return inflater.inflate(R.layout.example_fragment, container, false) }
}
```

Parameter container yang diteruskan ke onCreateView() adalah induk ViewGroup (dari layout aktivitas) tempat layout fragmen akan disisipkan. Parameter savedInstanceState adalah Bundle yang menyediakan data tentang instance fragmen sebelumnya, jika fragmen dilanjutkan.

Metode inflate() mengambil tiga argumen:

1. ID sumber daya layout yang ingin dimekarkan.
2. ViewGroup akan menjadi induk dari layout yang dimekarkan. container perlu diteruskan agar sistem menerapkan parameter layout ke tampilan akar layout yang dimekarkan, yang ditetapkan dalam tampilan induk yang akan dituju.
3. Boolean yang menunjukkan apakah layout yang dimekarkan harus dilampirkan ke ViewGroup (parameter kedua) selama pemekaran. (Dalam hal ini, ini salah karena sistem sudah memasukkan layout yang dimekarkan ke dalam container—meneruskan true akan membuat tampilan grup berlebih dalam layout akhir.) Kita kini telah melihat cara membuat fragmen yang menyediakan layout. Berikutnya, kita perlu menambahkan fragmen ke aktivitas.

Menambahkan fragmen ke aktivitas

Biasanya, fragmen berkontribusi pada sebagian UI ke aktivitas host, yang disematkan sebagai bagian dari hierarki tampilan keseluruhan aktivitas. Ada dua cara untuk menambahkan fragmen ke layout aktivitas:

Deklarasikan fragmen dalam file layout aktivitas.

Dalam hal ini, Kita bisa menetapkan properti layout fragmen seakan-akan sebuah tampilan. Misalnya, berikut ini adalah file layout untuk aktivitas dengan dua fragmen:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
android:orientation="horizontal" android:layout_width="match_parent"
android:layout_height="match_parent">
 <fragment android:name="com.example.news.ArticleListFragment"
android:id="@+id/list" android:layout_weight="1"
android:layout_width="0dp"
 android:layout_height="match_parent" />
 <fragment android:name="com.example.news.ArticleReaderFragment"
android:id="@+id/viewer" android:layout_weight="2"
android:layout_width="0dp"
 android:layout_height="match_parent" />
</LinearLayout>
```

Atribut android:name dalam menetapkan class Fragment untuk dibuat instance-nya dalam layout.

Saat sistem membuat layout aktivitas, sistem membuat instance setiap fragmen sebagaimana yang ditetapkan dalam layout dan memanggil metode onCreateView() masing-masing, untuk mengambil setiap fragmen. Sistem akan menyisipkan View yang dikembalikan oleh fragmen secara langsung, menggantikan elemen . Atau, secara programatis tambahkan fragmen ke ViewGroup yang ada. Kapan saja saat aktivitas berjalan, Kita bisa menambahkan fragmen ke layout aktivitas. Kita cukup menetapkan ViewGroup di tempat memasukkan fragmen.

Untuk membuat transaksi fragmen dalam aktivitas (seperti menambah, membuang, atau mengganti fragmen), kita harus menggunakan API dari

FragmentTransaction. Kita bisa mengambil instance FragmentTransaction dari FragmentActivity seperti ini:

val fragmentManager = supportFragmentManager val fragmentTransaction

= fragmentManager.beginTransaction()

Selanjutnya kita bisa menambahkan fragmen menggunakan metode add(), dengan menetapkan fragmen yang akan ditambahkan dan tampilan tempat menyisipkannya. Sebagai contoh:

val fragment = ExampleFragment()

fragmentTransaction.add(R.id.fragment_container, fragment)

fragmentTransaction.commit()

Argumen pertama yang diteruskan ke add() adalah ViewGroup tempat fragmen harus dimasukkan, yang ditetapkan oleh ID resource, dan parameter kedua merupakan fragmen yang akan ditambahkan. Setelah membuat perubahan dengan FragmentTransaction, Kita harus memanggil commit() untuk menerapkan perubahan.

Mengelola Fragmen

Untuk mengelola fragmen dalam aktivitas, kita perlu menggunakan FragmentManager. Untuk mendapatkannya, panggil getSupportFragmentManager() dari aktivitas kita. Beberapa hal yang dapat Kita lakukan dengan FragmentManager antara lain:

Dapatkan fragmen yang ada di aktivitas dengan findFragmentById() (untuk fragmen yang menyediakan UI dalam layout aktivitas) atau findFragmentByTag() (untuk fragmen yang menyediakan atau tidak menyediakan UI). 2. Tarik fragmen dari back-stack, dengan popBackStack() (menyimulasikan perintah Kembali oleh pengguna). 3. Daftarkan listener untuk perubahan pada back-stack, dengan addOnBackStackChangedListener().

Melakukan Transaksi Fragmen

Fitur menarik terkait penggunaan fragmen di aktivitas adalah kemampuan menambah, membuang, mengganti, dan melakukan tindakan lain dengannya, sebagai respons atas interaksi pengguna. Setiap set perubahan yang kita lakukan untuk aktivitas disebut transaksi dan kita bisa melakukan transaksi menggunakan API di FragmentTransaction. Kita juga bisa menyimpan setiap transaksi ke back-stack yang dikelola aktivitas, sehingga pengguna bisa mengarah mundur melalui perubahan fragmen (mirip mengarah mundur melalui aktivitas).

Kita bisa memperoleh instance FragmentTransaction dari FragmentManager seperti ini:

val fragmentManager = supportFragmentManager val fragmentTransaction

= fragmentManager.beginTransaction()

Setiap transaksi merupakan serangkaian perubahan yang ingin dilakukan pada waktu yang sama. Kita bisa menyiapkan semua perubahan yang ingin dilakukan untuk transaksi mana saja menggunakan metode seperti add(), remove(), dan replace(). Kemudian, untuk menerapkan transaksi pada aktivitas, kita harus memanggil commit(). Akan tetapi, sebelum memanggil commit(), kita mungkin perlu memanggil addToBackStack(), untuk menambahkan transaksi ke back-stack transaksi fragmen. Back-stack ini dikelola oleh aktivitas dan memungkinkan pengguna kembali ke status fragmen sebelumnya, dengan menekan tombol Kembali. Misalnya, dengan cara ini kita bisa mengganti satu fragmen dengan yang fragmen lain, dan mempertahankan status sebelumnya di back-stack:

```
val newFragment = ExampleFragment() val transaction =
supportFragmentManager.beginTransaction()
transaction.replace(R.id.fragment_container, newFragment)
transaction.addToBackStack(null) transaction.commit()
```

findFragmentByTag(). Sebagai contoh:

```
val fragment =
supportFragmentManager.findFragmentById(R.id.example_fragment) as
ExampleFragment
```

Membuat callback kejadian pada aktivitas

Dalam beberapa kasus, kita mungkin perlu fragmen untuk membagikan kejadian atau data dengan aktivitas dan/atau fragmen lain yang di-host oleh aktivitas. Untuk membagikan data, buat ViewModel bersama, seperti diuraikan dalam Membagikan data antar bagian fragmen di panduan ViewModel. Jika harus menyebarkan kejadian yang tidak dapat ditangani dengan ViewModel, Kita dapat mendefinisikan antarmuka callback di dalam fragment dan mengharuskan kejadian host menerapkannya. Saat aktivitas menerima callback melalui antarmuka, aktivitas akan bisa berbagi informasi itu dengan fragmen lain dalam layout jika perlu. Misalnya, jika sebuah aplikasi berita memiliki dua fragmen dalam aktivitas—satu untuk menampilkan daftar artikel (fragmen A) dan satu lagi untuk menampilkan artikel (fragmen B)—maka fragmen A harus memberi tahu aktivitas bila item daftar dipilih sehingga aktivitas bisa memberi tahu fragmen B untuk menampilkan artikel. Dalam hal ini, antarmuka OnArticleSelectedListener dideklarasikan di dalam fragmen A:

```
public class FragmentA : ListFragment() {     ...
   // Container Activity must implement this interface     interface
OnArticleSelectedListener {        fun onArticleSelected(articleUri:
Uri)
   }
   ...
}
```

Selanjutnya aktivitas yang menjadi host fragmen akan mengimplementasikan antarmuka OnArticleSelectedListener dan menggantikan onArticleSelected() untuk memberi tahu fragmen B mengenai kejadian dari fragmen A. Untuk memastikan bahwa aktivitas host mengimplementasikan antarmuka ini, metode callback fragmen A onAttach() (yang dipanggil sistem saat menambahkan fragmen ke aktivitas) membuat instance OnArticleSelectedListener dengan membuat Activity yang diteruskan ke onAttach():

```
public class FragmentA : ListFragment() {

    var listener: OnArticleSelectedListener? = null
    ...
    override fun onAttach(context: Context) {
        super.onAttach(context)
        listener = context as? OnArticleSelectedListener
        if (listener == null) {
            throw ClassCastException("$context must implement
                                      OnArticleSelectedListener")
        }

    }
    ...
}
```

Jika aktivitas belum mengimplementasikan antarmuka, maka fragmen akan melontarkan ClassCastException. Jika berhasil, anggota mListener yang menyimpan referensi ke implementasi aktivitas OnArticleSelectedListener, sehingga fragmen A bisa berbagi kejadian dengan aktivitas, dengan memanggil metode yang didefinisikan oleh antarmuka OnArticleSelectedListener. Misalnya, jika fragmen A adalah ekstensi dari ListFragment, maka setiap kali pengguna mengklik item daftar, sistem akan memanggil onListItemClick() di fragmen, yang selanjutnya memanggil onArticleSelected() untuk berbagi kejadian dengan aktivitas:

```
public class FragmentA : ListFragment() {
   var listener: OnArticleSelectedListener? = null    ...
   override fun onListItemClick(l: ListView, v: View, position: Int,
                                                id: Long) {
      // Append the clicked item's row ID with the content provider Uri        val noteUri: Uri
=
                  ContentUris.withAppendedId(ArticleColumns.CONTENT_URI, id)
      // Send the event and Uri to the host activity
listener?.onArticleSelected(noteUri)    }    ...
}
```

Parameter id yang diteruskan ke onListItemClick() merupakan ID baris dari item yang diklik, yang digunakan aktivitas (atau fragmen lain) untuk mengambil artikel dari ContentProvider aplikasi.

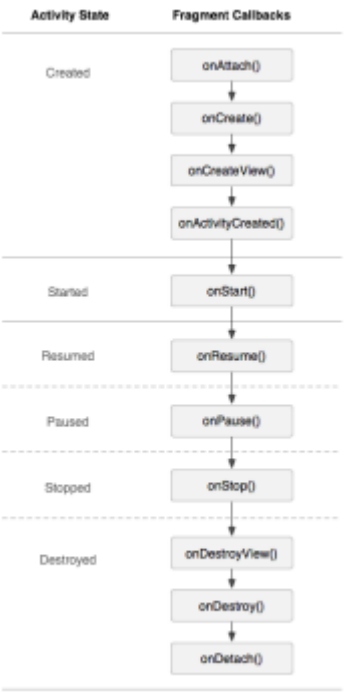
Menambahkan item ke Bilah Aplikasi

Fragmen kita bisa menyumbangkan item menu ke Menu Opsi aktivitas (dan, konsekuensinya, bilah aplikasi) dengan mengimplementasikan onCreateOptionsMenu(). Agar metode ini bisa menerima panggilan, Kita harus memanggil setHasOptionsMenu() selama onCreate(), untuk menunjukkan bahwa

fragmen ingin menambahkan item ke Menu Opsi. Jika tidak, fragmen tidak menerima panggilan ke onCreateOptionsMenu(). Setiap item yang selanjutnya Kita tambahkan ke Menu Opsi dari fragmen akan ditambahkan ke item menu yang ada. Fragmen juga menerima callback ke onOptionsItemSelected() bila item menu dipilih. Kita juga bisa mendaftarkan tampilan dalam layout fragmen untuk menyediakan menu konteks dengan memanggil registerForContextMenu().

Bila pengguna membuka menu konteks, fragmen akan menerima panggilan ke onCreateContextMenu(). Bila pengguna memilih item, fragmen akan menerima panggilan ke onContextItemSelected().

Menangani Daur Hidup Fragmen



Gambar 3. Efek daur hidup aktivitas pada daur hidup fragmen.

Mengelola daur hidup fragmen mirip sekali dengan mengelola daur hidup aktivitas. Seperti aktivitas, fragmen bisa berada dalam tiga status:

Dilanjutkan

Fragmen terlihat dalam aktivitas yang berjalan.

Dihentikan sementara

Aktivitas lain berada di latar depan dan memiliki fokus, namun aktivitas tempat fragmen berada masih terlihat (aktivitas latar depan sebagian terlihat atau tidak menutupi seluruh layar). Dihentikan Fragment tidak terlihat. Aktivitas host telah dihentikan atau fragmen telah dihapus dari aktivitas namun ditambahkan ke back-stack. Fragmen yang dihentikan masih hidup (semua status dan informasi anggota masih disimpan oleh sistem). Akan tetapi, fragmen tidak terlihat lagi oleh pengguna dan akan dimatikan jika aktivitas dimatikan.

Seperti halnya aktivitas, kita dapat mempertahankan status UI fragment di seluruh perubahan konfigurasi dan habisnya proses menggunakan kombinasi onSaveInstanceState(Bundle), ViewModel, serta penyimpanan lokal persisten. Perbedaan paling signifikan dalam daur hidup antara aktivitas dan fragmen ada pada cara penyimpanannya dalam back-stack masing-masing. Aktivitas ditempatkan ke dalam backstack aktivitas yang dikelola oleh sistem saat dihentikan, secara default (sehingga pengguna bisa mengarah kembali ke aktivitas dengan tombol Kembali). Namun, fragmen ditempatkan ke dalam back-stack yang dikelola oleh aktivitas host hanya jika kita secara eksplisit meminta instance tersebut disimpan dengan memanggil addToBackStack() selama transaksi yang menghapus segmen tersebut. Jika tidak, pengelolaan daur hidup fragmen mirip sekali dengan mengelola daur hidup aktivitas; berlaku praktik yang sama.

Mengoordinasi dengan daur hidup aktivitas

Daur hidup aktivitas tempat fragmen berada akan memengaruhi secara langsung siklus hidup fragmen sedemikian rupa sehingga setiap callback daur hidup aktivitas menghasilkan callback yang sama untuk masing-masing fragmen. Misalnya, bila aktivitas menerima dalam aktivitas akan menerima onPause(). Namun fragmen memiliki beberapa callback daur hidup ekstra, yang menangani interaksi unik dengan aktivitas untuk melakukan tindakan seperti membangun onPause(), maka masing-masing fragmen dan memusnahkan UI fragmen. Metode callback tambahan ini adalah:

onAttach() Dipanggil bila fragmen telah dikaitkan dengan aktivitas (Activity diteruskan di sini). onCreateView() Dipanggil untuk membuat hierarki tampilan yang dikaitkan dengan fragmen. onActivityCreated() Dipanggil bila metode

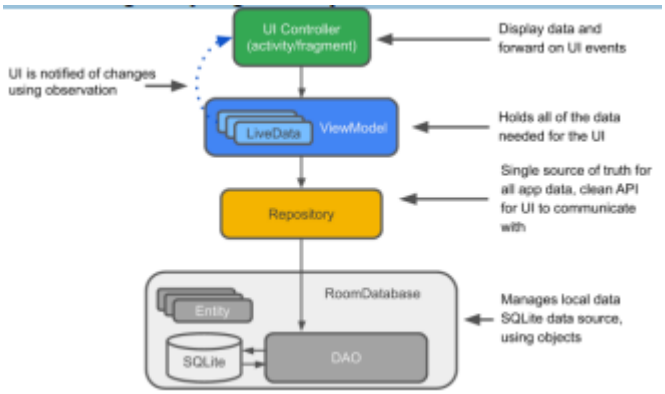
onCreate() aktivitas telah dikembalikan. onDestroyView()Dipanggil bila hierarki tampilan yang terkait dengan fragmen dihapus. onDetach()Dipanggil bila fragmen diputuskan dari aktivitas.

Alur daur hidup fragmen, karena dipengaruhi oleh aktivitas host-nya, diilustrasikan oleh gambar 3. Dalam gambar tersebut, kita bisa melihat bagaimana setiap status aktivitas yang berurutan menentukan metode callback mana yang mungkin diterima fragmen. Misalnya, saat aktivitas menerima callback onCreate(), fragmen dalam aktivitas akan menerima tidak lebih dari callback onActivityCreated(). Setelah status aktivitas diteruskan kembali, Kita bisa bebas menambah dan membuang fragmen untuk aktivitas tersebut. Sehingga, hanya saat aktivitas berada dalam status dilanjutkan, daur hidup fragmen bisa berubah secara independen. Akan tetapi, saat aktivitas meninggalkan status dilanjutkan, fragmen akan kembali didorong melalui daur hidupnya oleh aktivitas.

2.2 Menu dan Dialog

https://codelabs.developers.google.com/codelabs/android-room-with-a-view-kotlin/#0 Komponen arsitektur membantu kita menyusun aplikasi dengan cara yang kuat, dapat diuji, dan dapat dipelihara dengan kode yang lebih sederhana. Saat ini kita akan berfokus pada subset komponen, yaitu LiveData, ViewModel, dan Room.

Berikut diagram yang menunjukkan bentuk dasar arsitektur:



Entity: Kelas beranotasi yang menjelaskan tabel database saat bekerja dengan Room. SQLite database:: Di penyimpanan perangkat. Room persistence library membuat dan mengelola database ini untuk kita.

DAO: Data access object. Pemetaan query SQL ke fungsi. Ketika kita menggunakan DAO, kita memanggil metode, dan Room mengurus sisanya.

Apa itu DAO? Di DAO (objek akses data), kita menentukan kueri SQL dan mengaitkannya dengan panggilan metode. Kompiler memeriksa SQL dan menghasilkan kueri dari anotasi kenyamanan untuk kueri umum, seperti @Insert. Room menggunakan DAO untuk membuat API bersih untuk kode kita.

DAO harus berupa antarmuka atau kelas abstrak. Secara default, semua kueri harus dijalankan pada thread terpisah. Room memiliki dukungan coroutines, memungkinkan pertanyaan kita dijelaskan dengan pengubah penangguhan dan kemudian dipanggil dari coroutine atau dari fungsi suspensi lain.

Room database : Menyederhanakan pekerjaan basis data dan berfungsi sebagai titik akses ke basis data SQLite yang mendasarinya (menyembunyikan SQLiteOpenHelper). Database Room menggunakan DAO untuk mengeluarkan pertanyaan ke database SQLite.

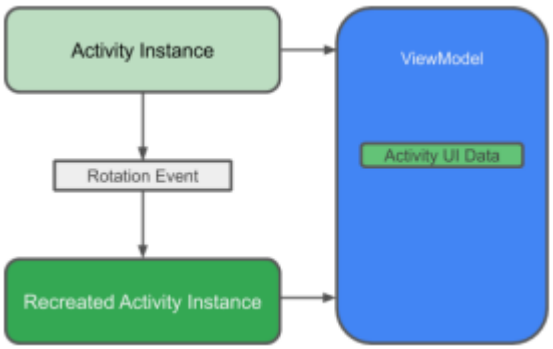
Apa itu database Room? Room adalah lapisan basis data di atas basis data SQLite. Room menangani tugas-tugas biasa yang kita gunakan untuk menangani dengan SQLiteOpenHelper. Room menggunakan DAO untuk mengeluarkan pertanyaan ke basis datanya. Secara default, untuk menghindari kinerja UI yang buruk, Room tidak memungkinkan kita untuk mengeluarkan pertanyaan pada thread utama. Ketika kueri Room mengembalikan LiveData, kueri secara otomatis dijalankan secara tidak sinkron pada background thread. Room menyediakan pemeriksaan waktu kompilasi terhadap pernyataan SQLite.

Repository: Kelas yang kita buat yang terutama digunakan untuk mengelola beberapa sumber data

Apa itu Repository? Kelas repository mengabstraksi akses ke banyak sumber data. Repository bukan bagian dari library Komponen Arsitektur, tetapi merupakan praktik terbaik yang disarankan untuk pemisahan kode dan arsitektur. Kelas Repository menyediakan API bersih untuk akses data ke seluruh aplikasi.

Mengapa menggunakan Repositori? Repositori mengelola kueri dan memungkinkan kita untuk menggunakan beberapa backend. Dalam contoh paling umum, Repositori mengimplementasikan logika untuk memutuskan apakah akan mengambil data dari jaringan atau menggunakan hasil yang di-cache dalam database lokal.

ViewModel: Bertindak sebagai pusat komunikasi antara Repositori (data) dan UI. UI tidak perlu lagi khawatir tentang asal-usul data. Contoh ViewModel adalah Activity/Fragment. Apa itu ViewModel? Peran ViewModel adalah untuk menyediakan data ke UI dan survive dari perubahan konfigurasi. ViewModel bertindak sebagai pusat komunikasi antara Repositori dan UI. Kita juga dapat menggunakan ViewModel untuk berbagi data antar fragmen. ViewModel adalah bagian dari lifecycle library.



Mengapa menggunakan ViewModel? ViewModel menyimpan data UI aplikasi kita dengan cara yang sadar siklus yang selamat dari perubahan konfigurasi. Memisahkan data UI aplikasi kita dari kelas Activity dan Fragmen, memungkinkan kita mengikuti prinsip tanggung jawab tunggal dengan lebih baik: Activity dan fragmen bertanggung jawab untuk menggambar data ke layar, sementara ViewModel dapat menangani memegang dan memproses semua data yang diperlukan untuk UI . Di ViewModel, gunakan LiveData untuk data yang dapat

diubah yang akan digunakan atau ditampilkan oleh UI. Menggunakan LiveData memiliki beberapa manfaat:

- Kita dapat menempatkan pengamat pada data (bukan polling untuk perubahan) dan hanya memperbarui UI ketika data benar-benar berubah.
- Repositori dan UI sepenuhnya dipisahkan oleh ViewModel.
- Tidak ada panggilan database dari ViewModel (ini semua ditangani di Repositori), membuat kode lebih dapat diuji. viewModelScope. Di Kotlin, semua coroutine dijalankan di dalam CoroutineScope. Lingkup mengontrol masa pakai coroutine melalui pekerjaannya. Ketika kita membatalkan pekerjaan lingkup, itu membatalkan semua coroutine dimulai dalam lingkup itu. Pustaka siklus hidup AndroidX-viewmodel-ktx menambahkan viewModelScope sebagai fungsi ekstensi dari kelas ViewModel, memungkinkan kita untuk bekerja dengan scope.

LiveData: Kelas pemegang data yang dapat diamati. Selalu memegang / menyimpan versi data terbaru, dan memberi tahu pengamatnya ketika data telah berubah. LiveData sadar akan siklus hidup. Komponen UI hanya mengamati data yang relevan dan jangan berhenti atau melanjutkan pengamatan. LiveData secara otomatis mengelola semua ini karena menyadari perubahan status siklus hidup yang relevan saat mengamati.

Saat data berubah, kita biasanya ingin mengambil tindakan, seperti menampilkan data yang diperbarui di UI. Ini berarti kita harus mengamati data sehingga ketika itu berubah, kita dapat bereaksi. Tergantung pada bagaimana data disimpan, ini bisa rumit. Mengamati perubahan pada data di berbagai komponen aplikasi kita dapat membuat jalur ketergantungan yang eksplisit dan kaku antar komponen. Ini membuat pengujian dan debugging menjadi sulit, antara lain. LiveData, lifecycle library class untuk observasi data, memecahkan masalah ini. Gunakan nilai kembalian tipe LiveData dalam deskripsi metode kita, dan Room menghasilkan semua kode yang diperlukan untuk memperbarui LiveData ketika database diperbarui.

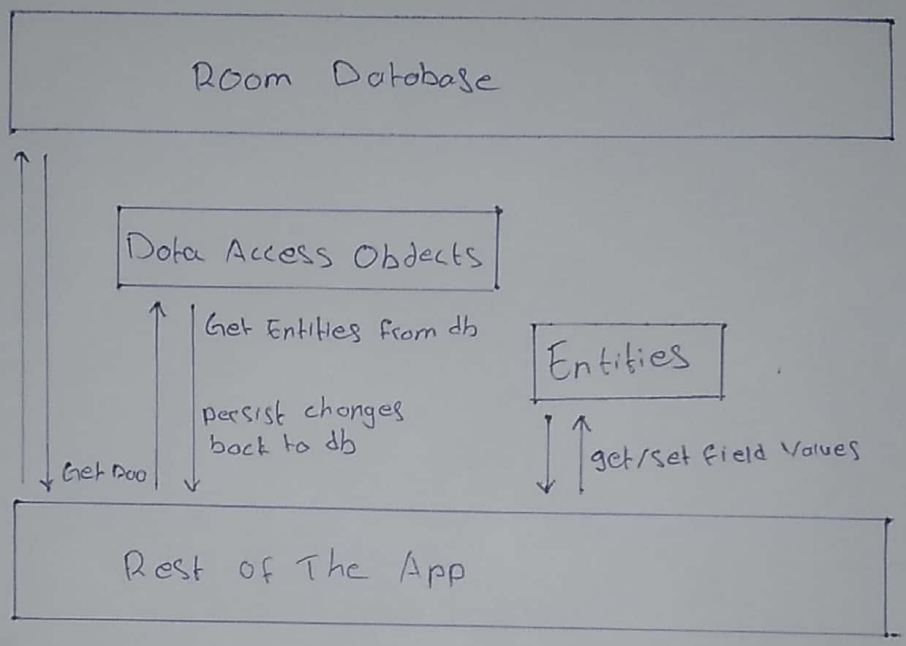
BAB III
TUGAS PENDAHULUAN

3.1 Soal

1. Gambarlah diagram Arsitektur room pada android!
2. Jelaskan keterkaitan room database pada diagram arsitektur yang kalian gambar!

3.2 Jawaban.

1.



2. Class database menyediakan aplikasi Anda dengan instance DAO yang terkait dengan database tersebut. Selanjutnya, aplikasi dapat menggunakan DAO untuk mengambil data dari database sebagai instance dari objek entitiy data terkait. Aplikasi juga dapat menggunakan entity data yang ditentukan untuk memperbarui baris dari tabel yang sesuai atau membuat baris baru untuk menyisipkan.

BAB IV
IMPLEMENTASI

4.1 Tugas Praktikum

1. Buat aplikasi yang menerapkan Intent dan Room Database

4.2 Source Code

1. Activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".MainActivity">
   <androidx.recyclerview.widget.RecyclerView
      android:id="@+id/recycler_view"
      android:layout_width="match_parent"
      android:layout_height="match_parent"/>
 <com.google.android.material.floatingactionbutton.FloatingActionButton
      android:id="@+id/fab"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_alignParentEnd="true"
      android:layout_alignParentBottom="true"
      android:layout_margin="16dp"
      android:src="@drawable/baseline_add_24"/>
</RelativeLayout>
```

2. MainActivity.kt

```
package com.example.praktikummodul34
import android.annotation.SuppressLint
```

```
import android.content.DialogInterface
import android.content.Intent
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import androidx.appcompat.app.AlertDialog
import androidx.recyclerview.widget.DividerItemDecoration
import androidx.recyclerview.widget.LinearLayoutManager
import androidx.recyclerview.widget.RecyclerView
import androidx.recyclerview.widget.RecyclerView.VERTICAL
import com.example.praktikummodul34.adapter.UserAdapter
import com.example.praktikummodul34.data.AppDatabase
import com.example.praktikummodul34.data.entity.User
import
com.google.android.material.floatingactionbutton.FloatingActionButton
class MainActivity : AppCompatActivity() {
    private lateinit var recyclerView: RecyclerView
    private lateinit var fab:FloatingActionButton
    private var list = mutableListOf<User>()
    private lateinit var adapter: UserAdapter
    private lateinit var database: AppDatabase
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        recyclerView = findViewById(R.id.recycler_view)
        fab = findViewById(R.id.fab)
        database = AppDatabase.getInstance(applicationContext)
        adapter = UserAdapter(list)
        adapter.setDialog(object : UserAdapter.Dialog{
            override fun onClick(position: Int) {
                // membuat dialog view
                val dialog = AlertDialog.Builder(this@MainActivity)
                dialog.setTitle(list[position].fullName)
```

```
            dialog.setItems(R.array.items_option,
DialogInterface.OnClickListener{ dialog, which ->
                if (which==0){
                    // coding ubah
                    val intent = Intent(this@MainActivity,
EditorActivity::class.java)
                    intent.putExtra("id", list[position].uid)
                    startActivity(intent)
                } else if (which==1){
                    // coding hapus
                    database.userDao().delete(list[position])
                    getData()
                } else {
                    // coding batal
                    dialog.dismiss()
                }
            })
            // untuk menampilkan dialog
            val dialogView = dialog.create()
            dialogView.show()
        }
    })
    recyclerView.adapter = adapter
    recyclerView.layoutManager =
LinearLayoutManager(applicationContext, VERTICAL, false)
recyclerView.addItemDecoration(DividerItemDecoration(applicationCont
ext, VERTICAL))
    fab.setOnClickListener {
        startActivity(Intent(this, EditorActivity::class.java))
    }
}
override fun onResume() {
```

```
        super.onResume()
        getData()
    }
    @SuppressLint("NotifyDataSetChanged")
    fun getData(){
        list.clear()
        list.addAll(database.userDao().getAll())
        adapter.notifyDataSetChanged()
    }
}
```

3. Activity_editor.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="14dp"
    tools:context=".EditorActivity">
    <com.google.android.material.textfield.TextInputLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:boxBackgroundMode="filled"
        android:layout_marginBottom="14dp">
        <EditText
            android:id="@+id/full_name"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:inputType="textPersonName"
```

```
            android:hint="Nama Lengkap"/>
        </com.google.android.material.textfield.TextInputLayout>
        <com.google.android.material.textfield.TextInputLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            app:boxBackgroundMode="filled"
            android:layout_marginBottom="14dp">
            <EditText
                android:id="@+id/email"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:inputType="textEmailAddress"
                android:hint="Alamat Email"/>
        </com.google.android.material.textfield.TextInputLayout>
        <com.google.android.material.textfield.TextInputLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            app:boxBackgroundMode="filled"
            android:layout_marginBottom="14dp">
            <EditText
                android:id="@+id/phone"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:inputType="phone"
                android:hint="Nomor Telephon"/>
        </com.google.android.material.textfield.TextInputLayout>
        <Button
            android:id="@+id/btn_save"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="Simpan"/>
    </LinearLayout>
```

4. EditorActivity.kt

```
package com.example.praktikummodul34

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import android.widget.Button
import android.widget.EditText
import android.widget.Toast
import androidx.room.RoomDatabase
import com.example.praktikummodul34.data.AppDatabase
import com.example.praktikummodul34.data.entity.User
class EditorActivity : AppCompatActivity() {
    private lateinit var fullName: EditText
    private lateinit var email: EditText
    private lateinit var phone: EditText
    private lateinit var btnSave: Button
    private lateinit var database: AppDatabase
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_editor)
        fullName = findViewById(R.id.full_name)
        email = findViewById(R.id.email)
        phone = findViewById(R.id.phone)
        btnSave = findViewById(R.id.btn_save)
        database = AppDatabase.getInstance(applicationContext)
        val intent = intent.extras
        if (intent!=null){
            val id = intent.getInt("id", 0)
            val user = database.userDao().get(id)
            fullName.setText(user.fullName)
            email.setText(user.email)
```

```
            phone.setText(user.phone)
        }
        btnSave.setOnClickListener {
            if (fullName.text.isNotEmpty() && email.text.isNotEmpty() &&
phone.text.isNotEmpty()) {
                if (intent!==null){
                    // coding edit data
                    database.userDao().update(
                        User(
                            intent.getInt("id", 0),
                            fullName.text.toString(),
                            email.text.toString(),
                            phone.text.toString()
                        )
                    )
                }else{
                    // coding tambah data
                    database.userDao().insertAll(
                        User(
                            null,
                            fullName.text.toString(),
                            email.text.toString(),
                            phone.text.toString()
                        )
                    )
                }
                finish()
            } else {
                Toast.makeText(
                    applicationContext,
                    "Silahkan Isi Semua Data Dengan valid",
                    Toast.LENGTH_SHORT
```

```
            ).show()
        }
    }
}
```

5. Rowuser.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    android:padding="14dp">
    <TextView
        android:id="@+id/full_name"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textStyle="bold" />
    <TextView
        android:id="@+id/email"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView
        android:id="@+id/phone"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

6. UserAdapter.kt

```
package com.example.praktikummodul34.adapter
import android.view.LayoutInflater
```

```
import android.view.TextureView
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
import com.example.praktikummodul34.data.entity.User
import com.example.praktikummodul34.R;
class UserAdapter(var list: List<User>) :
RecyclerView.Adapter<UserAdapter.ViewHolder>() {
   private lateinit var dialog: Dialog
   fun setDialog(dialog: Dialog){
      this.dialog = dialog
   }
   interface Dialog{
      fun onClick(position: Int)
   }
   inner class ViewHolder(view: View) :
RecyclerView.ViewHolder(view){
      var fullname: TextView
      var email: TextView
      var phone: TextView
      init {
         fullname = view.findViewById(R.id.full_name)
         email = view.findViewById(R.id.email)
         phone = view.findViewById(R.id.phone)
         view.setOnClickListener{
            dialog.onClick(layoutPosition)
         }
      }
   }
   override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
```

```
        val view = LayoutInflater.from(parent.context).inflate(R.layout.row_user, parent, false)
        return ViewHolder(view)
    }
    override fun getItemCount(): Int {
        return list.size
    }
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.fullname.text = list[position].fullName
        holder.email.text = list[position].email
        holder.phone.text = list[position].phone
    }
}
```

7. AppDatabase.kt

```
package com.example.praktikummodul34.data
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import com.example.praktikummodul34.data.dao.UserDao
import com.example.praktikummodul34.data.entity.User
@Database(entities = [User::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
    companion object{
        private var instant: AppDatabase? = null
        fun getInstance(context: Context): AppDatabase{
            if(instant==null){
                instant = Room.databaseBuilder(context, AppDatabase::class.java, "app-database")
```

```
                .fallbackToDestructiveMigration()
                .allowMainThreadQueries()
                .build()
            }
            return instant!!
        }
    }
}
```

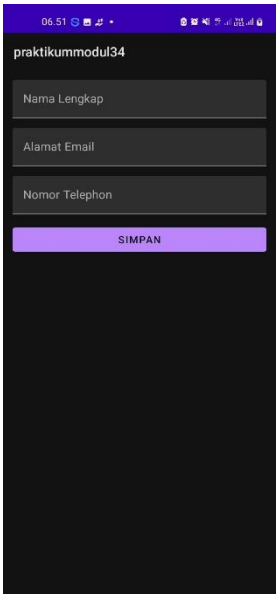
8. UserDao.kt

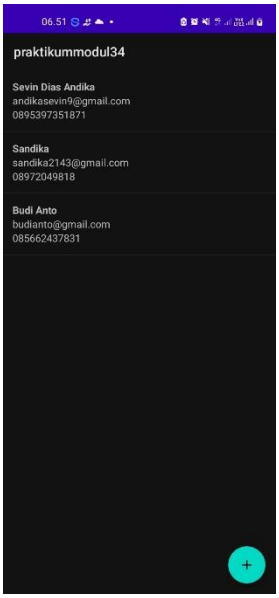
```
package com.example.praktikummodul34.data.dao
import androidx.room.*
import com.example.praktikummodul34.data.entity.User
@Dao
interface UserDao {
    //untukmengambilsemuadata//
    @Query("SELECT * FROM user")
    fun getAll(): List<User>
    //mengambildataberdasarkanuserID
    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    fun loadAllByIds(userIds: IntArray): List<User>
    @Insert
    fun insertAll(vararg users: User)
    @Delete
    fun delete(user: User)
    @Query("SELECT * FROM user WHERE uid = :uid")
    fun get(uid: Int) : User
    @Update
    fun update(user: User)
}
```

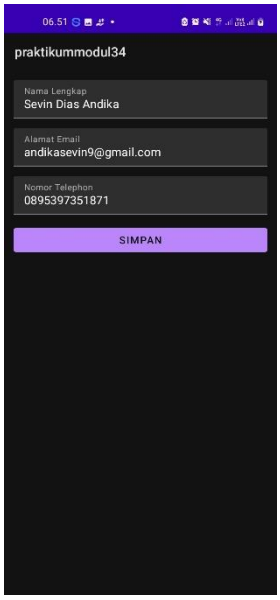
9. String.xml

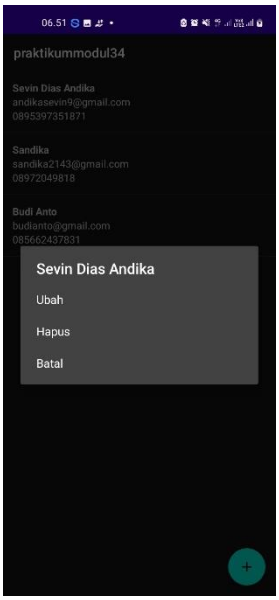
```
<resources>
    <string name="app_name">praktikummodul34</string>
    <string-array name="items_option">
        <item>Ubah</item>
        <item>Hapus</item>
        <item>Batal</item>
    </string-array>
</resources>
```

4.3 Hasil









BAB V
PENUTUP

5.1 Analisa

Dari hasil praktikum, praktikan menganalisa bahwa fragman adalah bagian UI aplikasi Anda yang dapat digunakan kembali. Sebuah fragmen menentukan dan mengelola tata letaknya sendiri, memiliki siklus proses sendiri, serta dapat menangani peristiwa inputnya sendiri. Fragmen tidak dapat berjalan sendiri. Fragmen harus *dihosting* oleh aktivitas atau fragmen lain. Hierarki tampilan fragmen menjadi bagian dari, atau *dilampirkan ke*, hierarki tampilan host.

Untuk membuat fragmen, kita harus membuat subclass Fragment (atau subclass-nya yang ada). Class Fragment memiliki kode yang mirip seperti Activity. Class ini memiliki metode callback yang serupa dengan aktivitas, seperti onCreate(), onStart(), onPause(), dan onStop(). Sebenarnya, jika kita mengonversi aplikasi Android saat ini untuk menggunakan fragmen, kita mungkin cukup memindahkan kode dari metode callback aktivitas ke masingmasing metode callback fragmen

5.2 Kesimpulan

1. Fragman adalah bagian UI aplikasi Anda yang dapat digunakan kembali.
2. onCreate() : Sistem akan memanggilnya saat membuat fragmen. Dalam implementasi, kita harus melakukan inisialisasi komponen penting dari fragmen yang ingin dipertahankan saat fragmen dihentikan sementara atau dihentikan, kemudian dilanjutkan.
3. onCreateView() : Sistem akan memanggilnya saat fragmen menggambar antarmuka penggunanya untuk yang pertama kali.
4. onPause() : Sistem akan memanggil metode ini sebagai indikasi pertama bahwa pengguna sedang meninggalkan fragmen kita (walau itu tidak selalu berarti fragmen sedang dimusnahkan).
5. Class Fragment memiliki kode yang mirip seperti Activity. Class ini memiliki metode callback yang serupa dengan aktivitas, seperti onCreate(), onStart(), onPause(), dan onStop().