Nama : M Rezky Raya Kilwouw

NIM : 222313190

Kelas : 3SI1

PRAKTIKUM 14

PEMROGRAMAN PLATFORM KHUSUS

A. Mendapatkan Data dari Internet
   1. Pengantar Coroutine di Android Studio
      a. Ringkasan aplikasi

         Aplikasi Race Tracker menyimulasikan dua pemain yang berlari dalam sebuah perlombaan. UI aplikasi terdiri dari dua tombol, Start/Pause (Mulai/Jeda) dan Reset, serta dua status progres untuk menampilkan progres pembalap. Pemain 1 dan 2 bersiap untuk memulai perlombaan dengan kecepatan yang berbeda. Saat perlombaan dimulai, Pemain 2 melaju dua kali lebih cepat dari Pemain 1.

      b. Menerapkan progress perlombaan
         - Buka class RaceParticipant yang merupakan bagian dari kode awal.
         - Di dalam class RaceParticipant, tentukan fungsi suspend baru yang bernama run().

```
class RaceParticipant(
    ...
) {
    var currentProgress by mutableStateOf(initialProgress)
        private set

    suspend fun run() {

    }
    ...
}
```

         - Untuk menyimulasikan progres perlombaan, tambahkan loop while yang berjalan hingga currentProgress mencapai nilai maxProgress yang disetel ke 100.

```
class RaceParticipant(
    ...
    val maxProgress: Int = 100,
    ...
) {
    var currentProgress by mutableStateOf(initialProgress)
        private set

    suspend fun run() {
        while (currentProgress < maxProgress) {

        }
    }
    ...
}
```

- Nilai currentProgress disetel ke initialProgress, yaitu 0. Untuk menyimulasikan progres peserta, tambahkan nilai currentProgress dengan nilai properti progressIncrement di dalam loop while. Perhatikan bahwa nilai default progressIncrement adalah 1.

```
class RaceParticipant(
    ...
    val maxProgress: Int = 100,
    ...
    private val progressIncrement: Int = 1,
    private val initialProgress: Int = 0
) {
    ...
    var currentProgress by mutableStateOf(initialProgress)
        private set

    suspend fun run() {
        while (currentProgress < maxProgress) {
            currentProgress += progressIncrement
        }
    }
}
```

- Untuk menyimulasikan berbagai interval progres dalam perlombaan, gunakan fungsi penangguhan delay(). Teruskan nilai properti progressDelayMillis sebagai argumen.

```
suspend fun run() {
    while (currentProgress < maxProgress) {
        delay(progressDelayMillis)
        currentProgress += progressIncrement
    }
}
```

c. Memulai perlombaan
   - Buka file RaceTrackerApp.kt yang berada dalam paket com.example.racetracker.ui.
   - Buka composable RaceTrackerApp() dan tambahkan panggilan ke composable LaunchedEffect() di baris setelah definisi raceInProgress.

```
@Composable
fun RaceTrackerApp() {
    ...
    var raceInProgress by remember { mutableStateOf(false) }

    LaunchedEffect {

    }
    RaceTrackerScreen(...)
}
```

   - Untuk memastikan bahwa jika instance playerOne atau playerTwo diganti dengan instance lain, maka LaunchedEffect() harus membatalkan dan meluncurkan kembali coroutine dasar, kemudian menambahkan objek playerOne dan playerTwo sebagai key ke LaunchedEffect. Serupa dengan cara

merekomposisi composable Text() saat nilai teksnya berubah, jika salah satu argumen utama dari LaunchedEffect() berubah, coroutine dasar akan dibatalkan dan diluncurkan kembali.

```
LaunchedEffect(playerOne, playerTwo) {
}
```

- Tambahkan panggilan ke fungsi playerOne.run() dan playerTwo.run().

```
@Composable
fun RaceTrackerApp() {
    ...
    var raceInProgress by remember { mutableStateOf(false) }

    LaunchedEffect(playerOne, playerTwo) {
        playerOne.run()
        playerTwo.run()
    }
    RaceTrackerScreen(...)
}
```

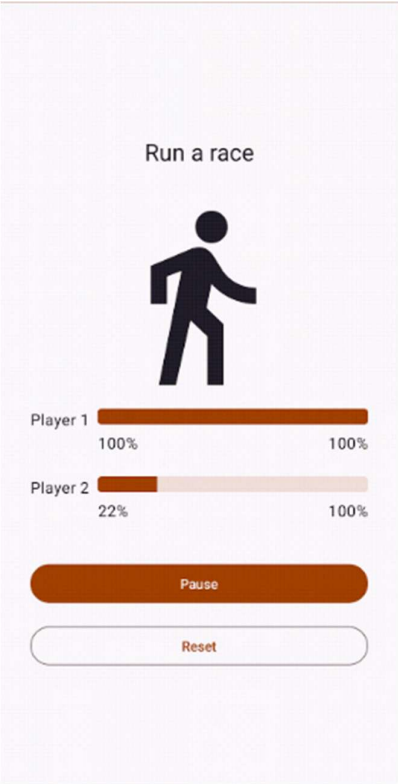
- Gabungkan blok LaunchedEffect() dengan kondisi if. Nilai awal untuk status ini adalah false. Nilai untuk status raceInProgress diperbarui menjadi true saat pengguna mengklik tombol Start dan LaunchedEffect() dieksekusi.

```
if (raceInProgress) {
    LaunchedEffect(playerOne, playerTwo) {
        playerOne.run()
        playerTwo.run()
    }
}
```

- Perbarui tanda raceInProgress ke false untuk menyelesaikan perlombaan. Nilai ini akan disetel ke false saat pengguna juga mengklik Pause. Jika nilai ini disetel ke false, LaunchedEffect() akan memastikan bahwa semua coroutine yang diluncurkan dibatalkan.

```
LaunchedEffect(playerOne, playerTwo) {
    playerOne.run()
    playerTwo.run()
    raceInProgress = false
}
```

- Jalankan aplikasi, lalu klik Start. Akan terlihat pemain satu menyelesaikan lomba sebelum pemain dua mulai berlari, seperti yang ditunjukkan dalam video berikut:



d. Meluncurkan tugas serentak

- Agar kedua peserta dapat berlari secara bersamaan, Disini harus meluncurkan dua coroutine terpisah dan memindahkan setiap panggilan ke fungsi run() di dalam coroutine tersebut. Gabungkan panggilan ke playerOne.run() dengan builder launch.

```
LaunchedEffect(playerOne, playerTwo) {
    launch { playerOne.run() }
    playerTwo.run()
    raceInProgress = false
}
```

- Demikian pula, gabungkan panggilan ke fungsi playerTwo.run() dengan builder launch. Dengan perubahan ini, aplikasi akan meluncurkan dua coroutine yang dieksekusi serentak. Sekarang kedua pemain dapat dijalankan secara bersamaan.

```
LaunchedEffect(playerOne, playerTwo) {
    launch { playerOne.run() }
    launch { playerTwo.run() }
    raceInProgress = false
}
```

- Jalankan aplikasi, lalu klik Start. Saat menunggu lomba dimulai, teks tombol tiba-tiba langsung berubah kembali menjadi Start.

Saat kedua pemain menyelesaikan lomba lari, aplikasi Race Tracker harus mereset teks tombol Pause kembali ke Start. Namun, sekarang aplikasi akan langsung memperbarui raceInProgress setelah meluncurkan coroutine tanpa menunggu pemain menyelesaikan perlombaan:

```
LaunchedEffect(playerOne, playerTwo) {
    launch {playerOne.run() }
    launch {playerTwo.run() }
    raceInProgress = false // This will update the state
immediately, without waiting for players to finish run()
execution.
}
```

Flag raceInProgress segera diperbarui karena:

- Fungsi builder launch meluncurkan coroutine untuk mengeksekusi playerOne.run() dan segera kembali untuk mengeksekusi baris berikutnya dalam blok kode.
- Alur eksekusi yang sama terjadi dengan fungsi builder launch kedua yang menjalankan fungsi playerTwo.run().
- Segera setelah builder launch kedua ditampilkan, tanda raceInProgress akan diperbarui. Pembaruan flag ini akan langsung mengubah teks tombol menjadi Start dan perlombaan tidak dimulai.

Cakupan Coroutine

Fungsi penangguhan coroutineScope membuat CoroutineScope dan memanggil blok penangguhan yang ditentukan dengan cakupan saat ini. Cakupan mewarisi coroutineContext dari cakupan LaunchedEffect().

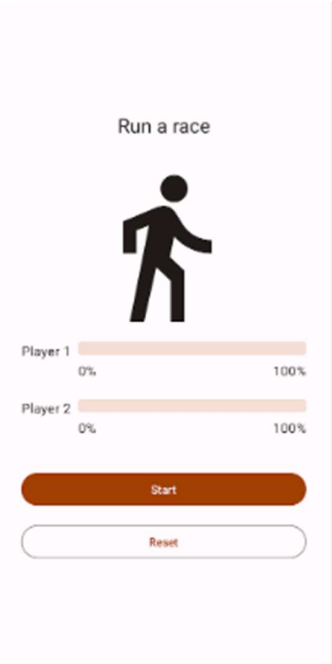
Cakupan akan segera ditampilkan setelah blok yang diberikan dan semua coroutine turunannya selesai. Untuk aplikasi RaceTracker, cakupan akan ditampilkan setelah kedua objek peserta selesai menjalankan fungsi run().

- Untuk memastikan fungsi run() dari playerOne dan playerTwo menyelesaikan eksekusi sebelum memperbarui tanda raceInProgress, gabungkan kedua builder peluncuran dengan blok coroutineScope.

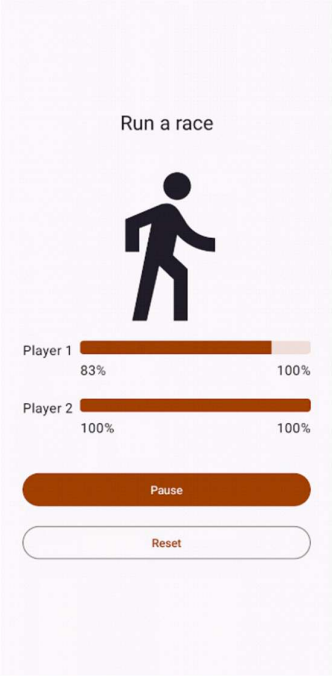
```
LaunchedEffect(playerOne, playerTwo) {
    coroutineScope {
        launch { playerOne.run() }
        launch { playerTwo.run() }
    }
```

```
    raceInProgress = false
}
```

- Jalankan aplikasi di emulator/perangkat Android. Akan terlihat layar berikut:



- Klik tombol Start. Pemain 2 berlari lebih cepat daripada Pemain 1. Setelah perlombaan selesai, yaitu saat kedua pemain mencapai progres 100%, label untuk tombol Pause akan berubah menjadi Start. Dapat mengklik tombol Reset untuk mereset perlombaan dan menjalankan kembali simulasi. Lomba ditampilkan dalam video berikut.



Untuk mengetahui cara membatalkan coroutine saat pengguna mengklik tombol Reset, ikuti langkah-langkah berikut:

- Gabungkan isi metode run() dalam blok try-catch

```
suspend fun run() {
    try {
        while (currentProgress < maxProgress) {
            delay(progressDelayMillis)
            currentProgress += progressIncrement
        }
    } catch (e: CancellationException) {
        Log.e("RaceParticipant", "$name: ${e.message}")
        throw e // Always re-throw CancellationException.
    }
}
```

- Jalankan aplikasi lalu klik tombol Start.
- Setelah beberapa penambahan progres, klik tombol Reset.
- Pastikan terlihat pesan yang dicetak di Logcat berikut:

e. Menulis pengujian unit untuk coroutine

Untuk menerapkan skenario pengujian, ikuti langkah-langkah berikut:

- Buka file RaceParticipantTest.kt yang berada di bagian set sumber pengujian.
- Untuk menentukan pengujian, setelah definisi raceParticipant, buat fungsi raceParticipant_RaceStarted_ProgressUpdated() dan anotasikan dengan anotasi @Test. Gunakan sintaksis ekspresi untuk menampilkan blok runTest() sebagai hasil pengujian karena blok pengujian harus ditempatkan di builder runTest.

```
class RaceParticipantTest {
    private val raceParticipant = RaceParticipant(
        ...
    )

    @Test
    fun raceParticipant_RaceStarted_ProgressUpdated() = runTest
{
    }
}
```

- Tambahkan variabel expectedProgress hanya baca dan setel ke 1.

```
@Test
fun raceParticipant_RaceStarted_ProgressUpdated() = runTest {
    val expectedProgress = 1
}
```

- Untuk menyimulasikan dimulainya perlombaan, gunakan builder launch untuk meluncurkan coroutine baru dan memanggil fungsi raceParticipant.run().

```
@Test
fun raceParticipant_RaceStarted_ProgressUpdated() = runTest {
    val expectedProgress = 1
    launch { raceParticipant.run() }
}
```

- Gunakan fungsi bantuan advanceTimeBy() untuk memajukan waktu dengan nilai raceParticipant.progressDelayMillis. Fungsi advanceTimeBy() membantu mengurangi waktu eksekusi uji.

```
@Test
fun raceParticipant_RaceStarted_ProgressUpdated() = runTest {
    val expectedProgress = 1
    launch { raceParticipant.run() }
    advanceTimeBy(raceParticipant.progressDelayMillis)
}
```

- Kita harus memanggil fungsi runCurrent(). karena advanceTimeBy() tidak menjalankan tugas yang dijadwalkan pada durasi tertentu. Fungsi ini akan menjalankan semua tugas yang tertunda pada saat ini.

```
@Test
fun raceParticipant_RaceStarted_ProgressUpdated() = runTest {
    val expectedProgress = 1
    launch { raceParticipant.run() }
    advanceTimeBy(raceParticipant.progressDelayMillis)
    runCurrent()
}
```

- Untuk memastikan progres diperbarui, tambahkan panggilan ke fungsi assertEquals() untuk memeriksa apakah nilai properti raceParticipant.currentProgress cocok dengan nilai variabel expectedProgress.

```
@Test
fun raceParticipant_RaceStarted_ProgressUpdated() = runTest {
    val expectedProgress = 1
    launch { raceParticipant.run() }
    advanceTimeBy(raceParticipant.progressDelayMillis)
    runCurrent()
    assertEquals(expectedProgress,
raceParticipant.currentProgress)
}
```

- Jalankan pengujian untuk mengonfirmasi bahwa pengujian berhasil.

Untuk memeriksa apakah progres perlombaan diperbarui dengan benar setelah perlombaan selesai, perlu menegaskan bahwa ketika perlombaan telah selesai, progres saat ini akan disetel ke 100.

Ikuti langkah-langkah berikut untuk menerapkan pengujian:

- Setelah fungsi pengujian raceParticipant_RaceStarted_ProgressUpdated(), buat fungsi raceParticipant_RaceFinished_ProgressUpdated() dan anotasikan dengan anotasi @Test. Fungsi ini akan menampilkan hasil pengujian dari blok runTest{}.

```
class RaceParticipantTest {
    ...

    @Test
    fun raceParticipant_RaceStarted_ProgressUpdated() = runTest
{
        ...
```

```
    }

    @Test
    fun raceParticipant_RaceFinished_ProgressUpdated() =
runTest {
    }
}
```

- Gunakan builder launch untuk meluncurkan coroutine baru dan menambahkan panggilan ke fungsi raceParticipant.run() di dalamnya.

```
@Test
fun raceParticipant_RaceFinished_ProgressUpdated() = runTest {
    launch { raceParticipant.run() }
}
```

- Untuk menyimulasikan hasil akhir perlombaan, gunakan fungsi advanceTimeBy() untuk memajukan waktu dispatcher dengan raceParticipant.maxProgress * raceParticipant.progressDelayMillis:

```
@Test
fun raceParticipant_RaceFinished_ProgressUpdated() = runTest {
    launch { raceParticipant.run() }
    advanceTimeBy(raceParticipant.maxProgress *
raceParticipant.progressDelayMillis)
}
```

- Tambahkan panggilan ke fungsi runCurrent() untuk menjalankan tugas yang tertunda.

```
@Test
fun raceParticipant_RaceFinished_ProgressUpdated() = runTest {
    launch { raceParticipant.run() }
    advanceTimeBy(raceParticipant.maxProgress *
raceParticipant.progressDelayMillis)
    runCurrent()
}
```

- Untuk memastikan progres diperbarui dengan benar, tambahkan panggilan ke fungsi assertEquals() untuk memastikan nilai properti raceParticipant.currentProgress sama dengan 100.

```
@Test
fun raceParticipant_RaceFinished_ProgressUpdated() = runTest {
    launch { raceParticipant.run() }
    advanceTimeBy(raceParticipant.maxProgress *
raceParticipant.progressDelayMillis)
    runCurrent()
    assertEquals(100, raceParticipant.currentProgress)
}
```

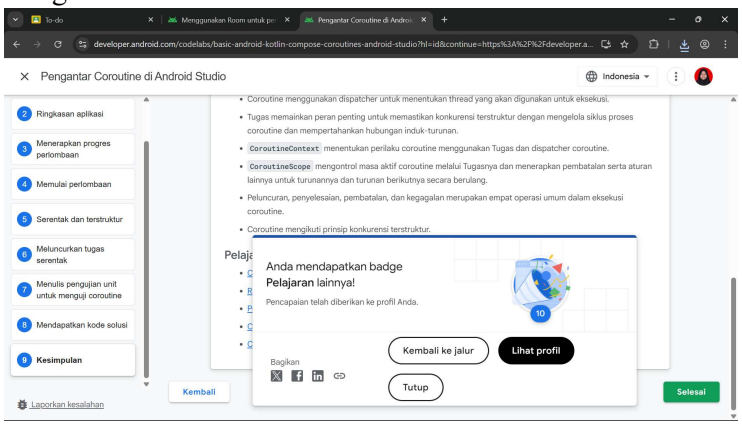
- Jalankan pengujian untuk mengonfirmasi bahwa pengujian berhasil.

f. Kesimpulan

- Coroutine memungkinkan menulis kode berdurasi panjang dan berjalan serentak tanpa mempelajari gaya pemrograman baru. Eksekusi coroutine memiliki desain yang berurutan.

- Kata kunci suspend digunakan untuk menandai fungsi, atau jenis fungsi, guna menunjukkan ketersediaannya untuk mengeksekusi, menjeda, dan melanjutkan kumpulan petunjuk kode.
- Fungsi suspend hanya dapat dipanggil dari fungsi penangguhan lainnya.
- Dapat memulai coroutine baru menggunakan fungsi builder launch atau async.
- Konteks coroutine, builder coroutine, Tugas, cakupan coroutine, dan Dispatcher adalah komponen utama untuk menerapkan coroutine.
- Coroutine menggunakan dispatcher untuk menentukan thread yang akan digunakan untuk eksekusi.
- Tugas memainkan peran penting untuk memastikan konkurensi terstruktur dengan mengelola siklus proses coroutine dan mempertahankan hubungan induk-turunan.
- CoroutineContext menentukan perilaku coroutine menggunakan Tugas dan dispatcher coroutine.
- CoroutineScope mengontrol masa aktif coroutine melalui Tugasnya dan menerapkan pembatalan serta aturan lainnya untuk turunannya dan turunan berikutnya secara berulang.
- Peluncuran, penyelesaian, pembatalan, dan kegagalan merupakan empat operasi umum dalam eksekusi coroutine.
- Coroutine mengikuti prinsip konkurensi terstruktur.

g. Badge



2. Mendapatkan Data dari Internet
   a. Jelajahi aplikasi awal Mars Photos

   ui\MarsPhotosApp.kt:
   - File ini berisi composable, MarsPhotosApp, yang menampilkan konten di layar, seperti panel aplikasi atas dan composable HomeScreen. Teks placeholder di langkah sebelumnya ditampilkan di composable ini.
   - Dalam codelab berikutnya, composable ini menampilkan data yang diterima dari server backend foto Mars.

   screens\MarsViewModel.kt:
   - File ini adalah model tampilan yang sesuai untuk MarsPhotosApp.
   - Class ini berisi properti MutableState bernama marsUiState. Memperbarui nilai properti ini akan memperbarui teks placeholder yang ditampilkan di layar.

- Metode getMarsPhotos() memperbarui respons placeholder. Selanjutnya di codelab, metode ini untuk menampilkan data yang diambil dari server. Tujuan codelab ini adalah memperbarui MutableState dalam ViewModel menggunakan data nyata yang didapatkan dari internet.

screens\HomeScreen.kt:

- File ini berisi composable HomeScreen dan ResultScreen. ResultScreen memiliki tata letak Box sederhana yang menampilkan nilai marsUiState dalam composable Text.

MainActivity.kt:

- Satu-satunya tugas untuk aktivitas ini adalah memuat ViewModel dan menampilkan composable MarsPhotosApp.

b. Layanan web dan Retrofit

Menambahkan dependensi Retrofit

- Buka file gradle level modul build.gradle.kts (Module :app).
- Di bagian dependencies, tambahkan baris berikut untuk library Retrofit:

```
// Retrofit
implementation("com.squareup.retrofit2:retrofit:2.9.0")
// Retrofit with Scalar Converter
implementation("com.squareup.retrofit2:converter-scalars:2.9.0")
```

Kedua library bekerja bersama. Dependensi pertama adalah untuk library Retrofit2, dan dependensi kedua adalah untuk pengonversi skalar Retrofit. Retrofit2 adalah versi library Retrofit yang diperbarui. Pengonversi skalar ini memungkinkan Retrofit menampilkan hasil JSON sebagai String. JSON adalah format untuk menyimpan dan memindahkan data antara klien dan server.

- Klik Sync Now untuk membangun ulang project dengan dependensi baru.

c. Menghubungkan ke internet

Tambahkan lapisan data ke project Mars Photos yang digunakan ViewModel untuk berkomunikasi dengan layanan web. Disini menerapkan API layanan Retrofit, dengan langkah-langkah berikut:

- Buat sumber data, class MarsApiService.
- Buat objek Retrofit dengan URL dasar dan factory pengonversi untuk mengonversi string.
- Buat antarmuka yang menjelaskan cara Retrofit berkomunikasi dengan server web.
- Buat layanan Retrofit dan ekspos instance-nya ke layanan API ke seluruh aplikasi.

Terapkan langkah-langkah di atas:

- Klik kanan pada paket com.example.marsphotos di panel project Android dan pilih New > Package.
- Di pop-up, tambahkan network ke akhir nama paket yang disarankan.
- Buat file Kotlin baru di bawah paket baru. Beri nama MarsApiService.
- Buka network/MarsApiService.kt.
- Tambahkan konstanta berikut ke URL dasar untuk layanan web.

```
private const val BASE_URL =
   "https://android-kotlin-fun-mars-server.appspot.com"
```

- Tambahkan builder Retrofit tepat di bawah konstanta tersebut untuk membangun dan membuat objek Retrofit.

```
import retrofit2.Retrofit

private val retrofit = Retrofit.Builder()
```

- Panggil addConverterFactory() pada builder dengan instance ScalarsConverterFactory.

```
import retrofit2.converter.scalars.ScalarsConverterFactory

private val retrofit = Retrofit.Builder()
   .addConverterFactory(ScalarsConverterFactory.create())
```

- Tambahkan URL dasar untuk layanan web menggunakan metode baseUrl().
- Panggil build() untuk membuat objek Retrofit.

```
private val retrofit = Retrofit.Builder()
   .addConverterFactory(ScalarsConverterFactory.create())
   .baseUrl(BASE_URL)
   .build()
```

- Pada panggilan ke builder Retrofit, tentukan antarmuka yang disebut MarsApiService yang menentukan cara Retrofit berkomunikasi dengan server web menggunakan permintaan HTTP.

```
interface MarsApiService {
}
```

- Tambahkan fungsi yang bernama getPhotos() ke antarmuka MarsApiService untuk mendapatkan string respons dari layanan web.

```
interface MarsApiService {
    fun getPhotos()
}
```

- Gunakan anotasi @GET untuk memberi tahu Retrofit bahwa ini adalah permintaan GET dan tentukan endpoint untuk metode layanan web tersebut. Dalam kasus ini, endpoint-nya adalah photos. Seperti yang disebutkan dalam tugas sebelumnya, disini menggunakan endpoint /photos dalam codelab ini.

```
import retrofit2.http.GET


interface MarsApiService {
    @GET("photos")
    fun getPhotos()
}
```

- Tambahkan jenis nilai yang ditampilkan fungsi ke String.

```
interface MarsApiService {
    @GET("photos")
    fun getPhotos(): String
}
```

Deklarasi objek

- Di luar deklarasi antarmuka MarsApiService, tentukan objek publik yang disebut MarsApi untuk menginisialisasi layanan Retrofit. Objek ini adalah objek singleton publik yang dapat diakses oleh aplikasi lainnya.
- Di dalam deklarasi objek MarsApi, tambahkan properti objek retrofit yang diinisialisasi dengan lambat bernama retrofitService dari jenis MarsApiService.
- Inisialisasi variabel retrofitService menggunakan metode retrofit.create() dengan antarmuka MarsApiService.

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

Memanggil layanan web di MarsViewModel

- Di file MarsApiService.kt, jadikan getPhotos() sebagai fungsi penangguhan untuk menjadikannya asinkron dan tidak memblokir thread panggilan. Disini memanggil fungsi ini dari dalam viewModelScope.

```
@GET("photos")
suspend fun getPhotos(): String
```

- Buka file ui/screens/MarsViewModel.kt. Scroll ke bawah, ke metode getMarsPhotos(). Hapus baris yang menetapkan respons status ke "Set the Mars API Response here!" sehingga metode getMarsPhotos() kosong.
- Di dalam getMarsPhotos(), luncurkan coroutine menggunakan viewModelScope.launch.

```
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch


private fun getMarsPhotos() {
    viewModelScope.launch {}
}
```

- Di dalam viewModelScope, gunakan objek singleton MarsApi untuk memanggil metode getPhotos() dari antarmuka retrofitService. Simpan respons yang ditampilkan dalam val yang bernama listResult.

```
import com.example.marsphotos.network.MarsApi

viewModelScope.launch {
    val listResult = MarsApi.retrofitService.getPhotos()
}
```

- Tetapkan hasil yang baru saja diterima dari server backend ke marsUiState. marsUiState adalah objek status yang dapat diubah yang mewakili status permintaan web terbaru.

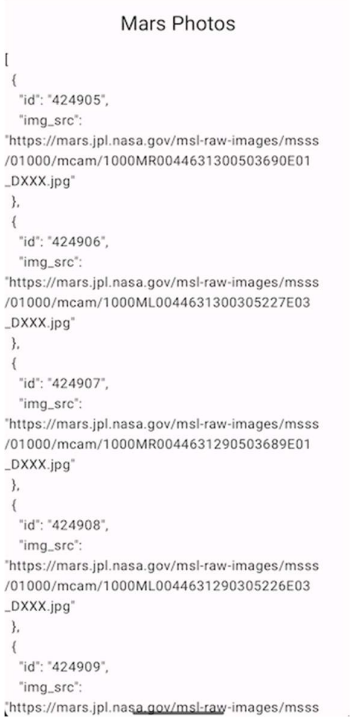
```
val listResult = MarsApi.retrofitService.getPhotos()
marsUiState = listResult
```

- Jalankan aplikasi. Perhatikan bahwa aplikasi segera ditutup, dan mungkin menampilkan atau tidak menampilkan pop-up error. Ini adalah error aplikasi.
- Klik tab Logcat di Android Studio dan catat error dalam log, yang diawali dengan baris seperti ini: "------- beginning of crash"

d. Menambahkan izin internet dan Penanganan Pengecualian
   - Buka manifests/AndroidManifest.xml. Tambahkan baris ini sebelum tag <application>:

```
<uses-permission android:name="android.permission.INTERNET" />
```

   - Kompilasi dan jalankan aplikasi lagi.



   - Ketuk tombol Kembali di perangkat atau emulator untuk menutup aplikasi.

Pengecualian

- Di getMarsPhotos(), di dalam blok launch, tambahkan blok try di sekitar panggilan MarsApi untuk menangani pengecualian.
- Tambahkan blok catch setelah blok try.

```
import java.io.IOException
```

```
viewModelScope.launch {
   try {
       val listResult = MarsApi.retrofitService.getPhotos()
       marsUiState = listResult
   } catch (e: IOException) {

   }
}
```

- Jalankan aplikasi sekali lagi. Perhatikan bahwa aplikasi tidak mengalami error kali ini.

Menambahkan UI Status

- Buka file ui/MarsViewModel.kt. Setelah pernyataan impor, tambahkan antarmuka tertutup MarsUiState. Penambahan ini membuat nilai objek MarsUiState dapat dilengkapi.

```
sealed interface MarsUiState {
    data class Success(val photos: String) : MarsUiState
    object Error : MarsUiState
    object Loading : MarsUiState
}
```

- Di dalam class MarsViewModel, perbarui definisi marsUiState. Ubah jenis ke MarsUiState dan MarsUiState.Loading sebagai nilai defaultnya. Buat penyetel menjadi pribadi untuk melindungi penulisan ke marsUiState.

```
var marsUiState: MarsUiState by
mutableStateOf(MarsUiState.Loading)
  private set
```

- Scroll ke bawah, ke metode getMarsPhotos(). Perbarui nilai marsUiState ke MarsUiState.Success dan teruskan listResult.

```
val listResult = MarsApi.retrofitService.getPhotos()
marsUiState = MarsUiState.Success(listResult)
```

- Di dalam blok catch, tangani respons kegagalan. Setel MarsUiState ke Error.

```
catch (e: IOException) {
   marsUiState = MarsUiState.Error
}
```

- Cabut penetapan marsUiState dari blok try-catch.

```
private fun getMarsPhotos() {
   viewModelScope.launch {
       marsUiState = try {
           val listResult = MarsApi.retrofitService.getPhotos()
           MarsUiState.Success(listResult)
       } catch (e: IOException) {
           MarsUiState.Error
       }
   }
}
```

- Dalam file screens/HomeScreen.kt, tambahkan ekspresi when di marsUiState. Jika marsUiState adalah MarsUiState.Success, panggil ResultScreen dan teruskan marsUiState.photos. Abaikan error untuk saat ini.

```
import androidx.compose.foundation.layout.fillMaxWidth

fun HomeScreen(
   marsUiState: MarsUiState,
   modifier: Modifier = Modifier
) {
    when (marsUiState) {
        is MarsUiState.Success -> ResultScreen(
            marsUiState.photos, modifier =
modifier.fillMaxWidth()
        )
    }
}
```

- Di dalam blok when, tambahkan pemeriksaan untuk MarsUiState.Loading dan MarsUiState.Error. Minta aplikasi menampilkan composable LoadingScreen, ResultScreen, dan ErrorScreen yang diterapkan

```
import androidx.compose.foundation.layout.fillMaxSize

fun HomeScreen(
   marsUiState: MarsUiState,
   modifier: Modifier = Modifier
) {
    when (marsUiState) {
        is MarsUiState.Loading -> LoadingScreen(modifier =
modifier.fillMaxSize())
        is MarsUiState.Success -> ResultScreen(
            marsUiState.photos, modifier =
modifier.fillMaxWidth()
        )

        is MarsUiState.Error -> ErrorScreen( modifier =
modifier.fillMaxSize())
    }
}
```

- Buka res/drawable/loading_animation.xml. Drawable ini adalah animasi yang memutar drawable gambar, loading_img.xml, mengitari titik tengah.
- Pada file screens/HomeScreen.kt, di bawah composable HomeScreen, tambahkan fungsi composable LoadingScreen berikut untuk menampilkan animasi pemuatan. Resource drawable loading_img disertakan dalam kode awal.

```
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.unit.dp
import androidx.compose.foundation.layout.size
import androidx.compose.foundation.Image


@Composable
fun LoadingScreen(modifier: Modifier = Modifier) {
    Image(
        modifier = modifier.size(200.dp),
```

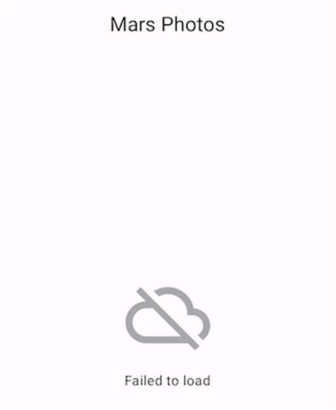
```
        painter = painterResource(R.drawable.loading_img),
        contentDescription = stringResource(R.string.loading)
    )
}
```

- Di bawah composable LoadingScreen, tambahkan fungsi composable ErrorScreen berikut sehingga aplikasi dapat menampilkan pesan error.

```
import androidx.compose.foundation.layout.Arrangement
import androidx.compose.foundation.layout.Column
import androidx.compose.foundation.layout.padding

@Composable
fun ErrorScreen(modifier: Modifier = Modifier) {
    Column(
        modifier = modifier,
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        Image(
            painter = painterResource(id =
R.drawable.ic_connection_error), contentDescription = ""
        )
        Text(text = stringResource(R.string.loading_failed),
modifier = Modifier.padding(16.dp))
    }
}
```

- Jalankan aplikasi lagi dengan mengaktifkan Mode Pesawat. Aplikasi tidak menutup sendiri sekarang, dan menampilkan pesan error berikut:



- Nonaktifkan Mode Pesawat di ponsel atau emulator

e. Mengurai respons JSON dengan kotlinx.serialization

Menambahkan dependensi library kotlinx.serialization

- Buka build.gradle.kts (Module :app).
- Di blok plugins, tambahkan plugin kotlinx serialization.

```
id("org.jetbrains.kotlin.plugin.serialization") version
"1.8.10"
```

- Di bagian dependencies, tambahkan kode berikut untuk menyertakan dependensi kotlinx.serialization. Dependensi ini menyediakan serialisasi JSON untuk project Kotlin.

```
// Kotlin serialization
implementation("org.jetbrains.kotlinx:kotlinx-serialization-
json:1.5.1")
```

- Temukan baris untuk pengonversi skalar Retrofit di blok dependencies lalu ubah untuk menggunakan kotlinx-serialization-converter:

```
// Retrofit with Kotlin serialization Converter

implementation("com.jakewharton.retrofit:retrofit2-kotlinx-
serialization-converter:1.0.0")
implementation("com.squareup.okhttp3:okhttp:4.11.0")
```

Menerapkan class data Mars Photo

- Klik kanan pada paket network dan pilih New > Kotlin File/Class.
- Di dialog pilih Class dan masukkan MarsPhoto sebagai nama class. Tindakan ini akan membuat file baru bernama MarsPhoto.kt dalam paket network.
- Jadikan MarsPhoto class data dengan menambahkan kata kunci data sebelum definisi class.
- Ubah tanda kurung kurawal {} menjadi tanda kurung (). Perubahan ini akan menimbulkan error karena class data harus memiliki setidaknya satu properti yang ditentukan.
- Tambahkan properti berikut ke definisi class MarsPhoto.
- Untuk membuat class MarsPhoto yang dapat diserialisasi dianotasi dengan @Serializable.

```
import kotlinx.serialization.Serializable

@Serializable
data class MarsPhoto(
    val id: String,  val img_src: String
)
```

Mengupdate MarsApiService dan MarsViewModel

- Buka network/MarsApiService.kt.
- Perhatikan error referensi yang belum terselesaikan untuk ScalarsConverterFactory. Error ini adalah hasil dari perubahan dependensi Retrofit di bagian sebelumnya.
- Hapus impor untuk ScalarConverterFactory.
- Pada deklarasi objek retrofit, ubah builder Retrofit untuk menggunakan kotlinx.serialization, bukan ScalarConverterFactory.

```
import
com.jakewharton.retrofit2.converter.kotlinx.serialization.asCon
verterFactory
import kotlinx.serialization.json.Json
import okhttp3.MediaType
```

```
private val retrofit = Retrofit.Builder()
        .addConverterFactory(Json.asConverterFactory("applicati
on/json".toMediaType()))
        .baseUrl(BASE_URL)
        .build()
```

- Perbarui antarmuka MarsApiService untuk Retrofit guna menampilkan daftar objek MarsPhoto, bukan String.

```
interface MarsApiService {
    @GET("photos")
    suspend fun getPhotos(): List<MarsPhoto>
}
```

- Buat perubahan yang serupa pada viewModel. Buka MarsViewModel.kt dan scroll ke bawah ke metode getMarsPhotos().
- Untuk mencetak jumlah foto yang diambil, update marsUiState

```
val listResult = MarsApi.retrofitService.getPhotos()
marsUiState = MarsUiState.Success(
   "Success: ${listResult.size} Mars photos retrieved"
)
```

- Pastikan Mode Pesawat dinonaktifkan pada perangkat atau emulator. Mengompilasi dan menjalankan aplikasi.

f. Ringkasan

Layanan web REST

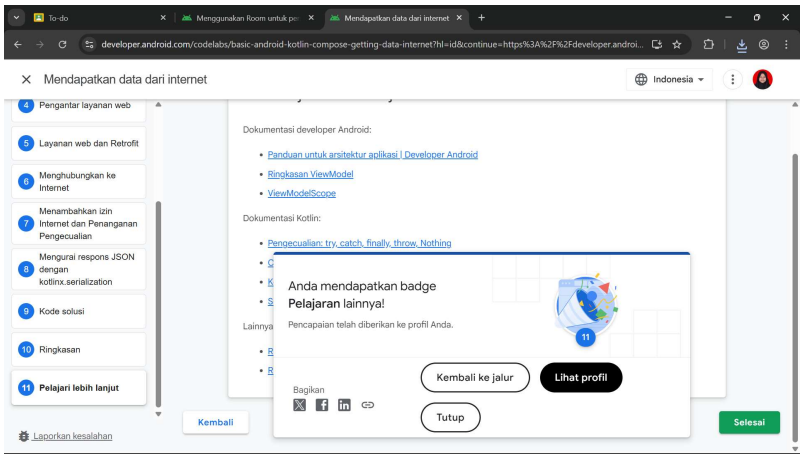
- Layanan web adalah fungsi berbasis software, yang ditawarkan melalui internet, yang memungkinkan aplikasi membuat permintaan dan mendapatkan data kembali.
- Layanan web umum menggunakan arsitektur REST. Layanan web yang menawarkan arsitektur REST disebut sebagai layanan RESTful. Layanan web RESTful dibuat menggunakan komponen dan protokol web standar.
- Membuat permintaan ke layanan web REST dengan cara standar melalui URI.

- Untuk menggunakan layanan web, aplikasi harus membuat koneksi jaringan dan berkomunikasi dengan layanan. Kemudian, aplikasi harus menerima dan mengurai data respons ke dalam format yang dapat digunakan aplikasi.
- Library Retrofit adalah library klien yang memungkinkan aplikasi yang dapat membuat permintaan ke layanan web REST.
- Gunakan pengonversi untuk memberi tahu Retrofit apa yang harus dilakukan dengan data yang dikirimnya ke layanan web dan didapatkan kembali dari layanan web. Misalnya, ScalarsConverter memperlakukan data layanan web sebagai String atau primitif lainnya.
- Agar aplikasi dapat terhubung ke internet, tambahkan izin "android.permission.INTERNET" dalam manifes Android.
- Inisialisasi lambat mendelegasikan pembuatan objek saat pertama kali digunakan. Tindakan ini akan membuat referensi, tetapi tidak membuat objek. Saat objek diakses untuk pertama kalinya, referensi akan dibuat dan digunakan setiap kali setelahnya.

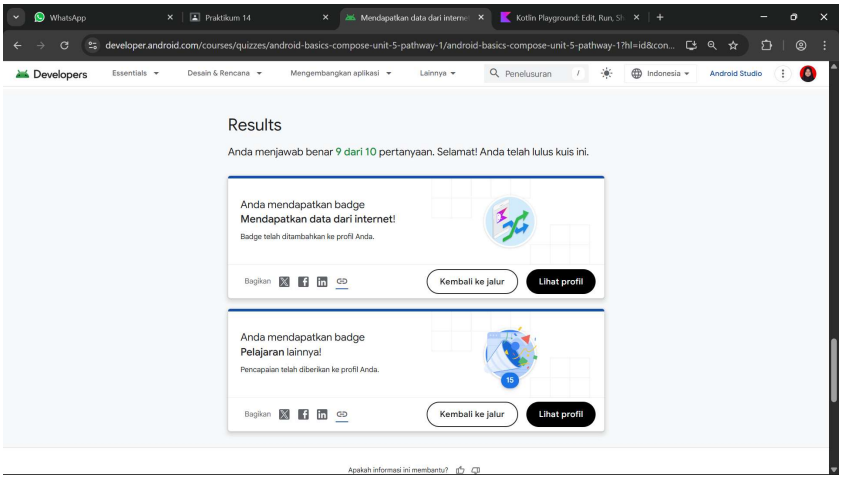
Penguraian JSON

- Respons dari layanan web sering kali diformat dalam JSON, format umum untuk mewakili data terstruktur.
- Objek JSON adalah kumpulan pasangan nilai-kunci.
- Koleksi objek JSON adalah array JSON.
- Kunci dalam pasangan nilai kunci dikelilingi oleh tanda kutip. Nilai dapat berupa angka atau string.
- Di Kotlin, alat serialisasi data tersedia dalam komponen terpisah, kotlinx.serialization. kotlinx.serialization menyediakan kumpulan library yang mengonversi string JSON menjadi objek Kotlin.
- Ada library Kotlin Pengonversi Serialisasi yang dikembangkan komunitas untuk Retrofit: retrofit2-kotlinx-serialization-converter.
- kotlinx.serialization mencocokkan kunci dalam respons JSON dengan properti dalam objek data yang memiliki nama yang sama.
- Agar dapat menggunakan nama properti yang berbeda untuk sebuah kunci, anotasikan properti tersebut dengan anotasi @SerialName dan value kunci JSON.

g. Badge



3. Kuis



B. Menggunakan Room untuk Persistensi Data

1. Mempertahankan Data dengan Room
   a. Sebelum memulai
   b. Ringkasan aplikasi
   c. Ringkasan aplikasi awal
   d. Komponen utama Room

      Tiga komponen Room berikut membuat alur kerja ini menjadi lancar.

      - Entity Room menampilkan tabel di database aplikasi. Hal ini dapat digunakan untuk memperbarui data yang disimpan dalam baris di tabel dan membuat baris baru untuk penyisipan.
      - DAOs Room menyediakan metode yang digunakan aplikasi untuk mengambil, memperbarui, menyisipkan, dan menghapus data dalam database.
      - Class database Room adalah class database yang menyediakan instance DAO yang terkait dengan database tersebut ke aplikasi.

   e. Membuat Entity item
      - Buka kode awal di Android Studio.
      - Buka paket data pada paket dasar com.example.inventory.
      - Di dalam paket data, buka class Kotlin Item yang mewakili entity database di aplikasi.

```
// No need to copy over, this is part of the starter code
class Item(
    val id: Int,
    val name: String,
    val price: Double,
    val quantity: Int
)
```

Class data

- Awali definisi class Item dengan kata kunci data untuk mengonversinya menjadi class data.

```
data class Item(
    val id: Int,
    val name: String,
    val price: Double,
    val quantity: Int
)
```

- Di atas deklarasi class Item, beri anotasi pada class data dengan @Entity. Gunakan argumen tableName untuk menetapkan items sebagai nama tabel SQLite.
- Anotasikan properti id dengan @PrimaryKey untuk menjadikan id sebagai kunci utama. Kunci utama adalah ID untuk mengidentifikasi setiap catatan/entri dalam tabel Item secara unik
- Tetapkan nilai default sebesar 0 pada id, yang diperlukan id untuk membuat nilai id secara otomatis.
- Tambahkan parameter autoGenerate ke anotasi @PrimaryKey untuk menentukan apakah kolom kunci utama harus dibuat secara otomatis. Jika autoGenerate disetel ke true, Room akan otomatis membuat nilai unik untuk kolom kunci utama saat instance entity baru dimasukkan ke dalam database. Ini memastikan bahwa setiap instance entity memiliki ID unik, tanpa harus menetapkan nilai ke kolom kunci utama secara manual

```
import androidx.room.PrimaryKey

@Entity(tableName = "items")
data class Item(
    @PrimaryKey
    val id: Int,

)
```

f. Membuat DAO item
   - Di paket data, buat antarmuka Kotlin ItemDao.kt.
   - Anotasi antarmuka ItemDao dengan @Dao.
   - Di dalam isi antarmuka, tambahkan anotasi @Insert.
   - Di bawah @Insert, tambahkan fungsi insert() yang menggunakan instance class Entity item sebagai argumennya.
   - Tandai fungsi dengan kata kunci suspend agar dapat berjalan di thread terpisah.

- Tambahkan argumen onConflict dan tetapkan nilai OnConflictStrategy.IGNORE.
- Tambahkan fungsi baru dengan anotasi @Update yang menggunakan Item sebagai parameter.
- Serupa dengan metode insert(), tandai fungsi ini dengan kata kunci suspend.
- Tulis kueri SQLite untuk mengambil item tertentu dari tabel item berdasarkan id yang diberikan. Kode berikut memberikan contoh kueri yang memilih semua kolom dari items, dengan id yang cocok dengan nilai tertentu dan id adalah ID unik.
- Tambahkan anotasi @Query.
- Gunakan kueri SQLite dari langkah sebelumnya sebagai parameter string ke anotasi @Query.
- Tambahkan parameter String ke @Query yang merupakan kueri SQLite untuk mengambil item dari tabel item.
- Setelah anotasi @Query, tambahkan fungsi getItem() yang menggunakan argumen Int dan menampilkan Flow<Item>.
- Tambahkan @Query dengan fungsi getAllItems().
- Minta kueri SQLite untuk menampilkan semua kolom dari tabel item yang diurutkan dalam urutan menaik.
- Minta getAllItems() menampilkan daftar entity Item sebagai Flow. Room terus memperbarui Flow.

```
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Update
import kotlinx.coroutines.flow.Flow

@Dao
interface ItemDao {
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(item: Item)

    @Update
    suspend fun update(item: Item)

    @Delete
    suspend fun delete(item: Item)

    @Query("SELECT * from items WHERE id = :id")
    fun getItem(id: Int): Flow<Item>

    @Query("SELECT * from items ORDER BY name ASC")
    fun getAllItems(): Flow<List<Item>>
}
```

g. Membuat instance Database
   - Di paket data, buat class Kotlin InventoryDatabase.kt.
   - Di file InventoryDatabase.kt, buat class InventoryDatabase sebagai class abstract yang memperluas RoomDatabase.

- Anotasikan class dengan @Database.
- Tentukan Item sebagai satu-satunya class dengan daftar entities.
- Setel version sebagai 1. Setiap kali mengubah skema tabel database.
- Setel exportSchema ke false agar tidak menyimpan cadangan histori versi skema.
- Di dalam isi class, deklarasikan fungsi abstrak yang menampilkan ItemDao sehingga database mengetahui DAO.
- Di bawah fungsi abstrak, tentukan companion object, yang memungkinkan akses ke metode untuk membuat atau mendapatkan database dan menggunakan nama class sebagai penentu.
- Di dalam objek companion, deklarasikan variabel nullable pribadi Instance untuk database lalu inisialisasikan ke null.
- Anotasikan Instance dengan @Volatile.
- Di bawah Instance, saat masih berada di dalam objek companion, tentukan metode getDatabase() dengan parameter Context yang diperlukan builder database.
- Tampilkan jenis InventoryDatabase. Pesan error muncul karena getDatabase() belum menampilkan apa pun.
- Dalam getDatabase(), tampilkan variabel Instance. Atau, jika Instance adalah null, lakukan inisialisasi di dalam blok synchronized{}. Gunakan operator elvis (?:) untuk melakukannya.
- Teruskan this, objek pendamping.
- Di dalam blok yang disinkronkan, gunakan builder database untuk mendapatkan database.
- Di dalam blok synchronized, gunakan builder database untuk mendapatkan database. Teruskan konteks aplikasi, class database, serta nama untuk database-item_database ke Room.databaseBuilder().
- Tambahkan strategi migrasi yang diperlukan ke builder. Gunakan . fallbackToDestructiveMigration().
- Untuk membuat instance database, panggil .build(). Panggilan ini akan menghapus error Android Studio.
- Setelah build(), tambahkan blok also dan tetapkan Instance = it untuk mempertahankan referensi ke instance db yang baru dibuat.

```
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

@Database(entities = [Item::class], version = 1, exportSchema =
false)
abstract class InventoryDatabase : RoomDatabase() {

    abstract fun itemDao(): ItemDao

    companion object {
        @Volatile
        private var Instance: InventoryDatabase? = null

        fun getDatabase(context: Context): InventoryDatabase {
            // if the Instance is not null, return it, otherwise
```

```
create a new database instance.
            return Instance ?: synchronized(this) {
                Room.databaseBuilder(context,
InventoryDatabase::class.java, "item_database")
                    .build()
                    .also { Instance = it }
            }
        }
    }
}
```

h. Mengimplementasikan Repositori
   - Buka file ItemsRepository.kt pada paket data.
   - Tambahkan fungsi berikut ke antarmuka, yang memetakan ke implementasi DAO.
   - Buka file OfflineItemsRepository.kt pada paket data.
   - Teruskan parameter konstruktor jenis ItemDao.
   - Di class OfflineItemsRepository, ganti fungsi yang ditentukan di antarmuka ItemsRepository dan panggil fungsi yang sesuai dari ItemDao.

```
import kotlinx.coroutines.flow.Flow

class OfflineItemsRepository(private val itemDao: ItemDao) :
ItemsRepository {
    override fun getAllItemsStream(): Flow<List<Item>> =
itemDao.getAllItems()

    override fun getItemStream(id: Int): Flow<Item?> =
itemDao.getItem(id)

    override suspend fun insertItem(item: Item) =
itemDao.insert(item)

    override suspend fun deleteItem(item: Item) =
itemDao.delete(item)

    override suspend fun updateItem(item: Item) =
itemDao.update(item)
}
```

Mengimplementasikan class AppContainer

- Buka file AppContainer.kt pada paket data.
- Teruskan instance ItemDao() ke konstruktor OfflineItemsRepository.
- Buat instance database dengan memanggil getDatabase() pada class InventoryDatabase yang meneruskan konteks, lalu memanggil .itemDao() untuk membuat instance Dao.

```
override val itemsRepository: ItemsRepository by lazy {
    OfflineItemsRepository(InventoryDatabase.getDatabase(contex
t).itemDao())
}
```

i. Menambahkan fungsi simpan

Mengupdate ViewModel ItemEntry

- Perhatikan fungsi pribadi validateInput() di class ItemEntryViewModel.

```
private fun validateInput(uiState: ItemDetails =
itemUiState.itemDetails): Boolean {
    return with(uiState) {
        name.isNotBlank() && price.isNotBlank() &&
quantity.isNotBlank()
    }
}
```

- Buka class ItemEntryViewModel dan tambahkan parameter konstruktor default private dari jenis ItemsRepository.

```
import com.example.inventory.data.ItemsRepository

class ItemEntryViewModel(private val itemsRepository:
ItemsRepository) : ViewModel() {
}
```

- Update initializer untuk model tampilan entri item di ui/AppViewModelProvider.kt dan teruskan instance repositori sebagai parameter.

```
object AppViewModelProvider {
    val Factory = viewModelFactory {
        // Other Initializers
        // Initializer for ItemEntryViewModel
        initializer {
            ItemEntryViewModel(inventoryApplication().container
.itemsRepository)
        }
        //...
    }
}
```

- Buka file ItemEntryViewModel.kt dan di akhir class ItemEntryViewModel, lalu tambahkan fungsi penangguhan bernama saveItem() untuk menyisipkan item ke dalam database Room. Fungsi ini menambahkan data ke database dengan cara yang tidak memblokir.
- Di dalam fungsi, periksa apakah itemUiState valid, lalu konversikan ke jenis Item agar Room dapat memahami data.
- Panggil insertItem() pada itemsRepository dan teruskan data. UI memanggil fungsi ini untuk menambahkan detail Item ke database.

```
suspend fun saveItem() {
    if (validateInput()) {
        itemsRepository.insertItem(itemUiState.itemDetails.toIt
em())
    }
}
```

Pengaduan composable ItemEntryBody()

- Dalam file ui/item/ItemEntryScreen.kt, composable ItemEntryBody() diterapkan sebagian sebagai bagian dari kode awal. Lihat composable ItemEntryBody() dalam panggilan fungsi ItemEntryScreen().

```
ItemEntryBody(
    itemUiState = viewModel.itemUiState,
    onItemValueChange = viewModel::updateUiState,
    onSaveClick = { },
    modifier = Modifier
        .padding(innerPadding)
        .verticalScroll(rememberScrollState())
        .fillMaxWidth()
)
```

- Perhatikan bahwa status UI dan lambda updateUiState diteruskan sebagai parameter fungsi. Lihat definisi fungsi untuk mengetahui cara status UI diupdate.
- Lihat implementasi fungsi composable ItemInputForm() dan perhatikan parameter fungsi onValueChange. Memperbarui nilai itemDetails dengan nilai yang dimasukkan oleh pengguna di kolom teks. Saat tombol Save diaktifkan, itemUiState.itemDetails memiliki nilai yang perlu disimpan.

```
@Composable
fun ItemEntryBody(
    itemUiState: ItemUiState,
    onItemValueChange: (ItemUiState) -> Unit,
    onSaveClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier
            .padding(16.dp)
            .fillMaxWidth(),
        verticalArrangement = Arrangement.spacedBy(16.dp)
    ) {

        ItemInputForm(
            itemDetails = itemUiState.itemDetails,
            onValueChange = { details ->
                onItemValueChange(
                    itemUiState.copy(
                        itemDetails = details,
                        isEntryValid = validateInput(details)
                    )
                )
            },
            modifier = Modifier.fillMaxWidth()
        )

        Button(
            onClick = onSaveClick,
            enabled = itemUiState.isEntryValid,
            shape = MaterialTheme.shapes.small,
            modifier = Modifier.fillMaxWidth()
        ) {
            Text(text = stringResource(R.string.save_action))
        }
    }
```

```
}
```

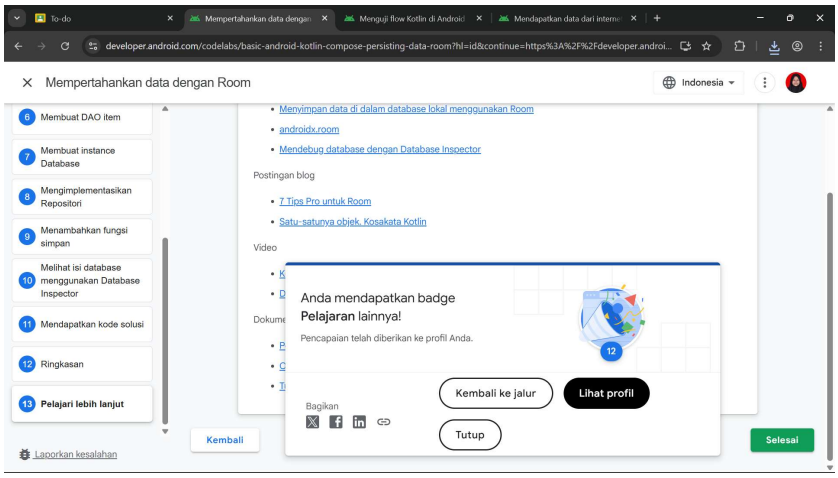
Menambahkan pemrorses klik ke tombol Save

- Di ItemEntryScreen.kt, dalam fungsi composable ItemEntryScreen, buat val bernama coroutineScope dengan fungsi composable rememberCoroutineScope().
- Update panggilan fungsi ItemEntryBody() dan luncurkan coroutine dalam lambda onSaveClick.
- Lihat implementasi fungsi saveItem() di file ItemEntryViewModel.kt untuk memeriksa apakah itemUiState valid, mengonversi itemUiState menjadi jenis Item, dan menyisipkannya dalam database menggunakan itemsRepository.insertItem().
- Di ItemEntryScreen.kt, dalam fungsi composable ItemEntryScreen, di dalam coroutine, panggil viewModel.saveItem() untuk menyimpan item dalam database.
- Di lambda onSaveClick, tambahkan panggilan ke navigateBack() setelah panggilan ke viewModel.saveItem() untuk membuka kembali layar sebelumnya. Fungsi ItemEntryBody() terlihat seperti kode berikut:

```
ItemEntryBody(
    itemUiState = viewModel.itemUiState,
    onItemValueChange = viewModel::updateUiState,
    onSaveClick = {
        coroutineScope.launch {
            viewModel.saveItem()
            navigateBack()
        }
    },
    modifier = modifier.padding(innerPadding)
)
```

j. Ringkasan
   - Tentukan tabel sebagai class data yang dianotasi dengan @Entity. Tentukan properti yang dianotasi dengan @ColumnInfo sebagai kolom dalam tabel.
   - Tentukan objek akses data (DAO) sebagai antarmuka yang dianotasi dengan @Dao. DAO memetakan fungsi Kotlin ke kueri database.
   - Gunakan anotasi untuk menentukan fungsi @Insert, @Delete, dan @Update.
   - Gunakan anotasi @Query dengan string kueri SQLite sebagai parameter untuk kueri lainnya.
   - Gunakan Database Inspector untuk melihat data yang disimpan di database Android SQLite.

k. Badge



2. Membaca dan Memperbarui Data dengan Room
   a. Mengupdate status UI
      - Buka file ui/home/HomeViewModel.kt, yang berisi konstanta TIMEOUT_MILLIS dan class data HomeUiState dengan daftar item sebagai parameter konstruktor.
      - Di dalam class HomeViewModel, deklarasikan val yang disebut homeUiState dari jenis StateFlow<HomeUiState>.
      - Panggil getAllItemsStream() di itemsRepository dan tetapkan ke homeUiState yang baru saja deklarasikan.
      - Tambahkan parameter konstruktor jenis ItemsRepository ke class HomeViewModel.
      - Di file ui/AppViewModelProvider.kt, di penginisialisasi HomeViewModel, teruskan objek ItemsRepository seperti yang ditunjukkan.
      - Kembali ke file HomeViewModel.kt. Perhatikan error ketidakcocokan jenis. Untuk mengatasi hal ini, tambahkan peta transformasi seperti yang ditunjukkan di bawah.
      - Gunakan operator stateIn untuk mengonversi Flow menjadi StateFlow. StateFlow adalah API yang dapat diamati untuk status UI, yang memungkinkan UI mengupdate dirinya sendiri.

```
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.flow.stateIn

val homeUiState: StateFlow<HomeUiState> =
    itemsRepository.getAllItemsStream().map { HomeUiState(it) }
        .stateIn(
            scope = viewModelScope,
            started =
SharingStarted.WhileSubscribed(TIMEOUT_MILLIS),
            initialValue = HomeUiState()
        )
```

   b. Menampilkan data Inventaris

- Di file HomeScreen.kt, pada fungsi composable HomeScreen, tambahkan parameter fungsi baru dari jenis HomeViewModel dan lakukan inisialisasi.

```
import androidx.lifecycle.viewmodel.compose.viewModel
import com.example.inventory.ui.AppViewModelProvider


@Composable
fun HomeScreen(
    navigateToItemEntry: () -> Unit,
    navigateToItemUpdate: (Int) -> Unit,
    modifier: Modifier = Modifier,
    viewModel: HomeViewModel = viewModel(factory =
AppViewModelProvider.Factory)
)
```

- Dalam fungsi composable HomeScreen, tambahkan val yang disebut homeUiState untuk mengumpulkan status UI dari HomeViewModel. M Menggunakan collectAsState(), yang mengumpulkan nilai dari StateFlow dan mewakili nilai terbarunya melalui State.

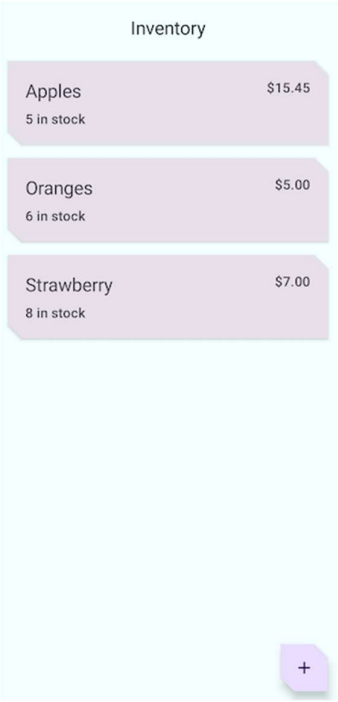
```
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue

val homeUiState by viewModel.homeUiState.collectAsState()
```

- Update panggilan fungsi HomeBody() dan teruskan homeUiState.itemList ke parameter itemList.

```
HomeBody(
    itemList = homeUiState.itemList,
    onItemClick = navigateToItemUpdate,
    modifier = modifier.padding(innerPadding)
)
```

- Jalankan aplikasi.



c. Menguji database
   - Dalam file build.gradle.kts (Module :app), perhatikan dependensi berikut untuk Espresso dan JUnit.
   - Beralihlah ke tampilan Project, lalu klik kanan pada src > New > Directory untuk membuat set sumber pengujian bagi pengujian.
   - Pilih androidTest/kotlin dari pop-up New Directory.
   - Buat class Kotlin bernama ItemDaoTest.kt.
   - Anotasikan class ItemDaoTest dengan @RunWith(AndroidJUnit4::class).

```
package com.example.inventory

import androidx.test.ext.junit.runners.AndroidJUnit4
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class ItemDaoTest {
}
```

   - Dalam class, tambahkan variabel var pribadi dari jenis ItemDao dan InventoryDatabase.

```
import com.example.inventory.data.InventoryDatabase
import com.example.inventory.data.ItemDao

private lateinit var itemDao: ItemDao
private lateinit var inventoryDatabase: InventoryDatabase
```

   - Tambahkan fungsi untuk membuat database dan menganotasinya dengan @Before agar dapat berjalan sebelum setiap pengujian.
   - Dalam metode, lakukan inisialisasi itemDao.

```
import android.content.Context
import androidx.room.Room
import androidx.test.core.app.ApplicationProvider
import org.junit.Before

@Before
fun createDb() {
    val context: Context =
ApplicationProvider.getApplicationContext()
    // Using an in-memory database because the information
stored here disappears when the
    // process is killed.
    inventoryDatabase = Room.inMemoryDatabaseBuilder(context,
InventoryDatabase::class.java)
        // Allowing main thread queries, just for testing.
        .allowMainThreadQueries()
        .build()
    itemDao = inventoryDatabase.itemDao()
}
```

- Tambahkan fungsi lain untuk menutup database. Anotasikan dengan @After untuk menutup database dan menjalankannya setelah setiap pengujian.

```
import org.junit.After
import java.io.IOException

@After
@Throws(IOException::class)
fun closeDb() {
    inventoryDatabase.close()
}
```

- Deklarasikan item di class ItemDaoTest yang akan digunakan database, seperti yang ditunjukkan dalam contoh kode berikut:

```
import com.example.inventory.data.Item

private var item1 = Item(1, "Apples", 10.0, 20)
private var item2 = Item(2, "Bananas", 15.0, 97)
```

- Tambahkan fungsi utilitas untuk menambahkan satu item, lalu dua item, ke database.

```
private suspend fun addOneItemToDb() {
    itemDao.insert(item1)
}

private suspend fun addTwoItemsToDb() {
    itemDao.insert(item1)
    itemDao.insert(item2)
}
```

- Tulis pengujian untuk menyisipkan satu item ke dalam database, insert(). Beri nama pengujian daoInsert_insertsItemIntoDB dan anotasikan dengan @Test.

```
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.runBlocking
import org.junit.Assert.assertEquals
import org.junit.Test
```

```
@Test
@Throws(Exception::class)
fun daoInsert_insertsItemIntoDB() = runBlocking {
    addOneItemToDb()
    val allItems = itemDao.getAllItems().first()
    assertEquals(allItems[0], item1)
}
```

- Tulis pengujian lain untuk getAllItems() dari database. Beri nama pengujian daoGetAllItems_returnsAllItemsFromDB.

```
@Test
@Throws(Exception::class)
fun daoGetAllItems_returnsAllItemsFromDB() = runBlocking {
    addTwoItemsToDb()
    val allItems = itemDao.getAllItems().first()
    assertEquals(allItems[0], item1)
    assertEquals(allItems[1], item2)
}
```

d. Menampilkan detail item

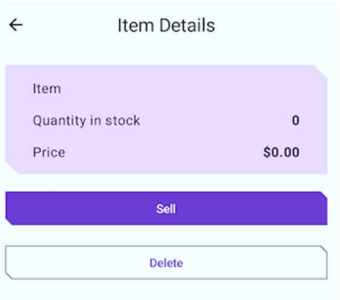
- Dalam fungsi composable HomeScreen, perhatikan panggilan fungsi HomeBody(). navigateToItemUpdate diteruskan ke parameter onItemClick, yang akan dipanggil saat mengklik item mana pun dalam daftar.

```
HomeBody(
    itemList = homeUiState.itemList,
    onItemClick = navigateToItemUpdate,
    modifier = modifier
        .padding(innerPadding)
        .fillMaxSize()
)
```

- Buka ui/navigation/InventoryNavGraph.kt dan perhatikan parameter navigateToItemUpdate di composable HomeScreen. Parameter ini menentukan tujuan untuk navigasi sebagai layar detail item.

```
HomeScreen(
    navigateToItemEntry = {
navController.navigate(ItemEntryDestination.route) },
    navigateToItemUpdate = {
        navController.navigate("${ItemDetailsDestination.route}
/${it}")
   }
```

- Klik item mana pun dalam daftar inventaris untuk melihat layar detail item dengan kolom kosong.



- Di UI/Item/ItemDetailsScreen.kt, tambahkan parameter baru ke composable ItemDetailsScreen jenis ItemDetailsViewModel dan gunakan metode factory untuk menginisialisasinya.

```
import androidx.lifecycle.viewmodel.compose.viewModel
import com.example.inventory.ui.AppViewModelProvider

@Composable
fun ItemDetailsScreen(
    navigateToEditItem: (Int) -> Unit,
    navigateBack: () -> Unit,
    modifier: Modifier = Modifier,
    viewModel: ItemDetailsViewModel = viewModel(factory =
AppViewModelProvider.Factory)
)
```

- Dalam composable ItemDetailsScreen(), buat val yang disebut uiState untuk mengumpulkan status UI. Gunakan collectAsState() untuk mengumpulkan uiState StateFlow dan mewakili nilai terbarunya melalui State. Android Studio menampilkan error referensi yang belum terselesaikan.

```
import androidx.compose.runtime.collectAsState

val uiState = viewModel.uiState.collectAsState()
```

- Untuk mengatasi error, buat val yang disebut uiState dari jenis StateFlow<ItemDetailsUiState> di class ItemDetailsViewModel.
- Ambil data dari repositori item dan petakan ke ItemDetailsUiState menggunakan fungsi ekstensi toItemDetails().

```
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.filterNotNull
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.flow.stateIn

val uiState: StateFlow<ItemDetailsUiState> =
        itemsRepository.getItemStream(itemId)
            .filterNotNull()
            .map {
                ItemDetailsUiState(itemDetails =
it.toItemDetails())
            }.stateIn(
                scope = viewModelScope,
                started =
```

```
SharingStarted.WhileSubscribed(TIMEOUT_MILLIS),
                initialValue = ItemDetailsUiState()
            )
```

- Teruskan ItemsRepository ke ItemDetailsViewModel untuk mengatasi error Unresolved reference: itemsRepository.

```
class ItemDetailsViewModel(
    savedStateHandle: SavedStateHandle,
    private val itemsRepository: ItemsRepository
    ) : ViewModel() {
```

- Di ui/AppViewModelProvider.kt, update penginisialisasi untuk ItemDetailsViewModel seperti ditunjukkan dalam cuplikan kode berikut:

```
initializer {
    ItemDetailsViewModel(
        this.createSavedStateHandle(),
        inventoryApplication().container.itemsRepository
    )
}
```

- Kembali ke ItemDetailsScreen.kt dan perhatikan bahwa error dalam composable ItemDetailsScreen() telah diselesaikan.
- Pada composable ItemDetailsScreen(), update panggilan fungsi ItemDetailsBody() dan teruskan argumen uiState.value ke itemUiState.

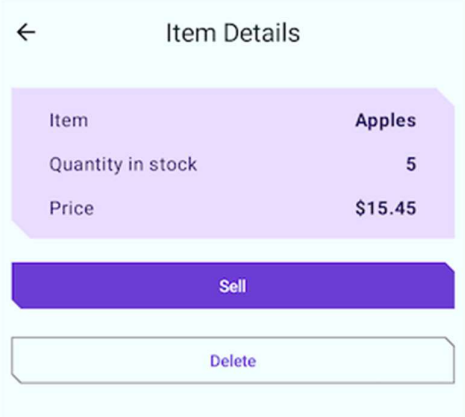
```
ItemDetailsBody(
    itemUiState = uiState.value,
    onSellItem = {  },
    onDelete = { },
    modifier = modifier.padding(innerPadding)
)
```

- Amati implementasi ItemDetailsBody() dan ItemInputForm(). Teruskan item yang saat ini dipilih dari ItemDetailsBody() ke ItemDetails()

```
@Composable
private fun ItemDetailsBody(
    itemUiState: ItemUiState,
    onSellItem: () -> Unit,
    onDelete: () -> Unit,
    modifier: Modifier = Modifier
) {
    Column(
       //...
    ) {
        var deleteConfirmationRequired by rememberSaveable {
mutableStateOf(false) }
        ItemDetails(
             item = itemDetailsUiState.itemDetails.toItem(),
modifier = Modifier.fillMaxWidth()
         )

      //...
    }
```

- Jalankan aplikasi.



e. Menerapkan item jual
   - Di ItemDaoTest.kt, tambahkan fungsi bernama daoUpdateItems_updatesItemsInDB() tanpa parameter. Anotasi dengan @Test dan @Throws(Exception::class).
   - Tentukan fungsi dan buat blok runBlocking. Panggil addTwoItemsToDb() di dalamnya.
   - Perbarui kedua entity dengan nilai yang berbeda, yang memanggil itemDao.update.
   - Ambil entity dengan itemDao.getAllItems(). Bandingkan dengan entity yang diupdate dan nyatakan.

```
@Test
@Throws(Exception::class)
fun daoUpdateItems_updatesItemsInDB() = runBlocking {
    addTwoItemsToDb()
    itemDao.update(Item(1, "Apples", 15.0, 25))
    itemDao.update(Item(2, "Bananas", 5.0, 50))

    val allItems = itemDao.getAllItems().first()
    assertEquals(allItems[0], Item(1, "Apples", 15.0, 25))
    assertEquals(allItems[1], Item(2, "Bananas", 5.0, 50))
}
```

Tambahkan fungsi di ViewModel

- Di ItemDetailsViewModel.kt, dalam class ItemDetailsViewModel, tambahkan fungsi bernama reduceQuantityByOne() tanpa parameter.
- Di dalam fungsi, mulai coroutine dengan viewModelScope.launch{}.
- Di dalam blok launch, buat val bernama currentItem dan tetapkan ke uiState.value.toItem().
- Tambahkan pernyataan if untuk memastikan quality lebih besar dari 0.
- Panggil updateItem() pada itemsRepository dan teruskan currentItem yang diupdate. Gunakan copy() untuk mengupdate nilai quantity sehingga fungsi terlihat seperti berikut:

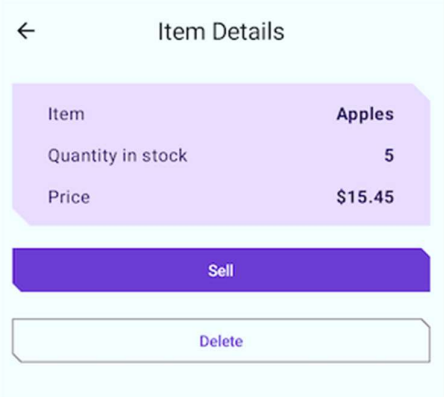
```
fun reduceQuantityByOne() {
    viewModelScope.launch {
```

```
        val currentItem = uiState.value.itemDetails.toItem()
        if (currentItem.quantity > 0) {
    itemsRepository.updateItem(currentItem.copy(quantity =
currentItem.quantity - 1))
        }
    }
}
```

- Kembali ke ItemDetailsScreen.kt.
- Pada composable ItemDetailsScreen, buka panggilan fungsi ItemDetailsBody().
- Di lambda onSellItem, panggil viewModel.reduceQuantityByOne().

```
ItemDetailsBody(
    itemUiState = uiState.value,
    onSellItem = { viewModel.reduceQuantityByOne() },
    onDelete = { },
    modifier = modifier.padding(innerPadding)
)
```

- Jalankan aplikasi.



Menghapus entity item

- Di ItemDaoTest.kt, tambahkan pengujian bernama daoDeleteItems_deletesAllItemsFromDB().

```
@Test
@Throws(Exception::class)
fun daoDeleteItems_deletesAllItemsFromDB()
```

- Luncurkan coroutine dengan runBlocking {}.
- Tambahkan dua item ke database dan panggil itemDao.delete() pada dua item tersebut untuk menghapusnya dari database.
- Ambil entity dari database dan pastikan daftar tersebut kosong. Pengujian yang sudah selesai akan terlihat seperti berikut.

```
import org.junit.Assert.assertTrue

@Test
@Throws(Exception::class)
fun daoDeleteItems_deletesAllItemsFromDB() = runBlocking {
    addTwoItemsToDb()
```

```
    itemDao.delete(item1)
    itemDao.delete(item2)
    val allItems = itemDao.getAllItems().first()
    assertTrue(allItems.isEmpty())
}
```

- Di ItemDetailsViewModel, tambahkan fungsi baru bernama deleteItem() yang tidak menggunakan parameter dan tidak menampilkan apa pun.
- Di dalam fungsi deleteItem(), tambahkan panggilan fungsi itemsRepository.deleteItem() dan teruskan uiState.value.toItem().

```
suspend fun deleteItem() {
    itemsRepository.deleteItem(uiState.value.itemDetails.toItem
())
}
```

- Di composable ui/item/ItemDetailsScreen, tambahkan val bernama coroutineScope dan tetapkan ke rememberCoroutineScope(). Pendekatan ini menampilkan cakupan coroutine yang terikat ke komposisi tempatnya dipanggil (composable ItemDetailsScreen).

```
import androidx.compose.runtime.rememberCoroutineScope

val coroutineScope = rememberCoroutineScope()
```

- Scroll ke fungsi ItemDetailsBody().
- Luncurkan coroutine dengan coroutineScope di dalam lambda onDelete.
- Di dalam blok launch, panggil metode deleteItem() di viewModel.

```
import kotlinx.coroutines.launch

ItemDetailsBody(
    itemUiState = uiState.value,
    onSellItem = { viewModel.reduceQuantityByOne() },
    onDelete = {
        coroutineScope.launch {
           viewModel.deleteItem()
    }
    modifier = modifier.padding(innerPadding)
)
```

- Setelah menghapus item, kembali ke layar inventaris.
- Panggil navigateBack() setelah panggilan fungsi deleteItem().

```
onDelete = {
    coroutineScope.launch {
        viewModel.deleteItem()
        navigateBack()
    }
```

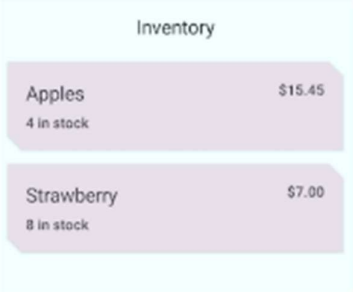
- Masih dalam file ItemDetailsScreen.kt, scroll ke fungsi ItemDetailsBody().
- Fungsi ini adalah bagian dari kode awal. Composable ini menampilkan dialog pemberitahuan untuk mendapatkan konfirmasi pengguna sebelum menghapus item dan memanggil fungsi deleteItem() saat mengetuk Yes.

```
@Composable
private fun ItemDetailsBody(
```

```
    itemUiState: ItemUiState,
    onSellItem: () -> Unit,
    onDelete: () -> Unit,
    modifier: Modifier = Modifier
) {
    Column(
        /*...*/
    ) {
        //...

        if (deleteConfirmationRequired) {
            DeleteConfirmationDialog(
                onDeleteConfirm = {
                    deleteConfirmationRequired = false
                    onDelete()
                },
                //...
            )
        }
    }
}
```

- Jalankan aplikasi.
- Pilih elemen daftar di layar Inventory.
- Di layar Item Details, ketuk Delete.
- Ketuk Yes pada dialog pemberitahuan, dan aplikasi akan membuka kembali layar Inventory.
- Pastikan entity yang dihapus tidak ada lagi di database aplikasi.



Mengedit entity item

- Di ItemDaoTest.kt, tambahkan pengujian bernama daoGetItem_returnsItemFromDB().
- Tentukan fungsi. Di dalam coroutine, tambahkan satu item ke database.
- Ambil entity dari database menggunakan fungsi itemDao.getItem() dan tetapkan ke val bernama item.
- Bandingkan nilai sebenarnya dengan nilai yang diambil, lalu nyatakan menggunakan assertEquals(). Pengujian yang telah selesai akan terlihat seperti berikut:

```
@Test
@Throws(Exception::class)
fun daoGetItem_returnsItemFromDB() = runBlocking {
    addOneItemToDb()
    val item = itemDao.getItem(1)
    assertEquals(item.first(), item1)
}
```

- Di ItemDetailsScreen.kt, scroll ke composable ItemDetailsScreen.
- Di FloatingActionButton(), ubah argumen onClick untuk menyertakan uiState.value.itemDetails.id, yang merupakan id dari entity yang dipilih.
- Di class ItemEditViewModel, tambahkan blok init.
- Di dalam blok init, luncurkan coroutine dengan viewModelScope.launch.
- Di dalam blok launch, ambil detail entity dengan itemsRepository.getItemStream(itemId).

```
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.flow.filterNotNull
import kotlinx.coroutines.flow.first


init {
    viewModelScope.launch {
        itemUiState = itemsRepository.getItemStream(itemId)
            .filterNotNull()
            .first()
            .toItemUiState(true)
    }
}
```

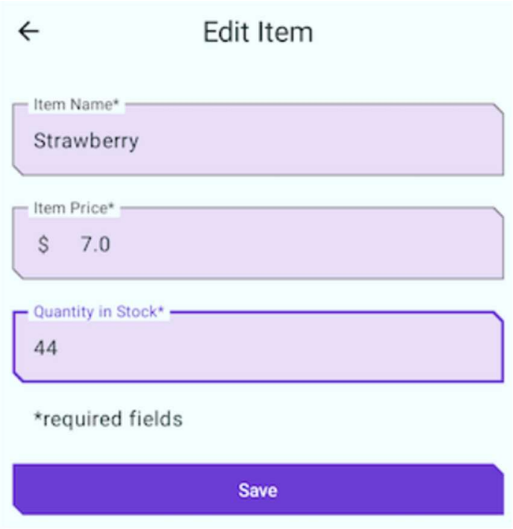
- Tambahkan parameter konstruktor ke class ItemEditViewModel:

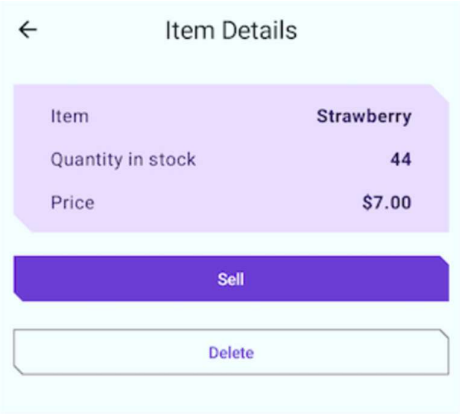
```
class ItemEditViewModel(
    savedStateHandle: SavedStateHandle,
    private val itemsRepository: ItemsRepository
)
```

- Pada file AppViewModelProvider.kt, di penginisialisasi ItemEditViewModel, tambahkan objek ItemsRepository sebagai argumen.

```
initializer {
    ItemEditViewModel(
        this.createSavedStateHandle(),
        inventoryApplication().container.itemsRepository
    )
}
```

- Jalankan aplikasi.


Edit Item
Item Name*
Strawberry
Item Price*
$ 7.0
Quantity in Stock*
44
*required fields
Save


Item Details
Item
Strawberry
Quantity in stock
44
Price
$7.00
Sell
Delete