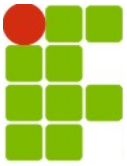

INSTITUTO FED. DE EDUCAÇÃO, CIÊNC. E TEC. DE PERNAMBUCO
CURSO: TEC. EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
DISCIPLINA: PROGRAMAÇÃO PARA DISPOSITIVOS MÓVEIS
PROFESSOR: RAMIDE DANTAS
ASSUNTO: TRABALHANDO EM BACKGROUND E NOTIFICAÇÕES

Prática 10

Obs.: esta prática é continuação da prática 09; use controle de versões.

Parte 0 – Correções e Arrumações

Passo 1 – Em `MainViewModel`, ajuste o `set()` da propriedade city como abaixo:

```
    private var _city = mutableStateOf<City?>(null)
    var city: City?
        get() = _city.value
        set(tmp) {
            _city.value = tmp?.copy()
    }
```

Essa modificação torna a recomposição (atualização da UI) acontecer de forma mais confiável.

Passo 2 – Na `HomePage`, faça a modificação abaixo:

```
@Composable
fun HomePage(...) {
    ...
//        if (viewModel.city == null ||
//            viewModel.city!!.forecast == null) return
        viewModel.city?.forecast?.let { forecasts ->
            LazyColumn {
                items(forecasts) { forecast ->
                    ForecastItem(
                        forecast = forecast,
                        onClick = { },
                        modifier = modifier
                    )
                }
            }
        }
    }
}
```

Essa modificação previne erros de `NullPointerException` que podem ocorrer quando clicamos nas notificações, apesar de que não deveria ser necessária.

Passo 3 – Rode e teste a aplicação.

Se nada quebrou, faça um novo *commit*.

Parte 1 – Adicionando atributo de monitoramento à cidade

Passo 1: Em `model.City`, adicione o atributo booleano `isMonitored`, com valor *default* `false`.

Esse campo indica se a cidade deve ser monitorada no background.

Passo 2: Na classe `db.fb.FBCity`, adicione o atributo booleano `monitored`.

Esse atributo corresponde ao atributo `isMonitored` de `model.City`.

Modifique `FBCity.toCity()` e `City.toFBCity()` para fazer esse mapeamento.

Passo 3: Em `db.fb.FBDatabase`, adicione o método `update()` como baixo:

```
    fun update(city: City) {
        if (auth.currentUser == null) throw RuntimeException("Not logged in!")
        val uid = auth.currentUser!!.uid
        val fbCity = city.toFBCity()
        val changes = mapOf( "lat" to fbCity.lat, "lng" to fbCity.lng,
                             "monitored" to fbCity.monitored )
        db.collection("users").document(uid)
            .collection("cities").document(fbCity.name!!).update(changes)
    }
```

Passo 4: Ainda em `db.fb.FBDatabase`, adicione os métodos `onUserSignOut()` e `onCityUpdated(city: City)` à interface `Listener`.

Passo 5: Ainda em `db.fb.FBDatabase`, modifique o bloco `init` como abaixo:

```
   init {
        auth.addAuthStateListener { auth ->
            if (auth.currentUser == null) {
                citiesListReg?.remove()
                listener?.onUserSignOut()
                return@addAuthStateListener
            }

            val refCurrUser = db.collection("users")
                                .document(auth.currentUser!!.uid)
            refCurrUser.get().addOnSuccessListener {
                it.toObject(FBUser::class.java)?.let { user ->
                    listener?.onUserLoaded(user.toUser())
                }
            }

            citiesListReg = refCurrUser.collection("cities")
                .addSnapshotListener { snapshots, ex ->
                    if (ex != null) return@addSnapshotListener

                    snapshots?.documentChanges?.forEach { change ->
                        val fbCity = change.document
                                          .toObject(FBCity::class.java)
                        when (change.type) {
                            DocumentChange.Type.ADDED ->
                                    listener?.onCityAdded(fbCity.toCity())
                            DocumentChange.Type.MODIFIED ->
                                    listener?.onCityUpdated(fbCity.toCity())
                            DocumentChange.Type.REMOVED ->
                                    listener?.onCityRemoved(fbCity.toCity())
                        }
                    }
                }
        }
    }
```

Passo 6: Em `repo.Repository`, adicione o método `onUserSignOut` a interface Listener do repositório.

Implemente também os métodos herdados de `FBDatabase.Listener` (passo anterior); simplesmente chamando os métodos correspondentes do *listener* do repositório (ver código do `onCityAdded()`).

Passo 7: Ainda em `repo.Repository`, adicione o método `update(city: City)`, que simplesmente chama o `update()` do `FBDatabase` criado anteriormente.

Passo 8: Em `MainViewModel`, modifique `onCityUpdated()` como abaixo:

```
    override fun onCityUpdated(city: City) {
        _cities.remove(city.name)
        _cities[city.name] = city.copy()

        if (_city.value?.name == city.name) {
            _city.value = city.copy(
                weather = if (city.weather != null) city.weather
                                                    else _city.value?.weather,
                forecast = if (city.forecast != null) city.forecast
                                                    else _city.value?.forecast
            )
        }
    }
```

Também será necessário implementar o `onUserSignOut()` do `Listener`. Dê uma implementação vazia.

Passo 9: Em `ui.HomePage`, adicione um ícone ao lado nome da cidade selecionada usando código a seguir.

```
    Icon(
        imageVector = icon,
        contentDescription = "Monitor?",
        modifier = Modifier.size(32.dp)
                           .clickable (enabled = viewModel.city != null) {
                               repo.update(viewModel.city!!
                                           .copy(isMonitored = !isMonitored))
                            }
    )
```

A variável `icon` deve ser setada previamente com o valor `Icons.Outlined.Favorite` se a cidade for monitorada (`isMonitored == true`) ou `Icons.Outlined.FavoriteBorder` caso contrário. O ícone adicionado é clicável e muda o status de monitoramento da cidade.

Passo 10: Adicione o ícone acima para cada cidade em `ui.ListPage`.

Se baseie no código do passo anterior porém sem a habilidade clicar. Faça os ajustes necessários na UI (posicionamento do ícone, tamanho, etc.).

Passo 11: Rode e teste o aplicativo.

Verifique se o campo "`monitored`" é criado a atualizado corretamente no Firebase. Se estiver tudo correto, faça um novo *commit*.

Parte 2 – Realizando o monitoramento no background e lançando notificações

Passo 1: Adicionado as dependências no `build.gradle.kts`:

```
...
dependencies {
    implementation(libs.androidx.work.runtime.ktx)
    ...
}
```

Passo 2: Crie a classe `monitor.ForecastWorker` com o código a seguir:

```
class ForecastWorker(context: Context, params: WorkerParameters) : Worker(context,
params) {
    companion object {
        private const val CHANNEL_ID: String = "WEATHER_APP"
    }

    override fun doWork(): Result {
        val cityName = inputData.getString("city") ?: return Result.failure()
        showNotification(cityName)
        return Result.success()
    }

    private fun showNotification(cityName: String) {
        val newIntent = Intent(this.applicationContext,
                               MainActivity::class.java)
        newIntent.addFlags(
            Intent.FLAG_ACTIVITY_SINGLE_TOP or
                Intent.FLAG_ACTIVITY_CLEAR_TOP)
        newIntent.putExtra("city", cityName)

        val pendingIntent = PendingIntent.getActivity(
            this.applicationContext, cityName.hashCode(), newIntent,
            PendingIntent.FLAG_UPDATE_CURRENT or PendingIntent.FLAG_IMMUTABLE)

        createNotificationChannel()

        val builder = NotificationCompat
            .Builder(this.applicationContext, CHANNEL_ID)
            .setSmallIcon(R.drawable.ic_launcher_foreground)
            .setContentTitle(cityName)
            .setContentText("Clique para ver previsão do tempo atualizada.")
            .setPriority(NotificationCompat.PRIORITY_DEFAULT)
            .setContentIntent(pendingIntent)
            .setAutoCancel(true)

        val notificationManager: NotificationManager =
            this.applicationContext
                .getSystemService(Context.NOTIFICATION_SERVICE)
                    as NotificationManager
        // ID = hashCode: para substituir ou remover notificações
        notificationManager.notify(cityName.hashCode(), builder.build())
    }

    private fun createNotificationChannel() {
        val name = "WeatherApp"
        val descriptionText = "WeatherApp Notifications"
        val importance = NotificationManager.IMPORTANCE_DEFAULT
        val channel = NotificationChannel(CHANNEL_ID, name, importance)
                    .apply {
            description = descriptionText
        }

        val notificationManager: NotificationManager = this.applicationContext
            .getSystemService(Context.NOTIFICATION_SERVICE)
                as NotificationManager
        notificationManager.createNotificationChannel(channel)
    }
}
```

Essa classe implementa o `Worker` que roda em *background* periodicamente. Neste caso, ele simplesmente lança uma notificação para o usuário. Ao clicar na notificação, a aplicação é aberta e a cidade da notificação é selecionada.

Passo 3: Crie a classe `monitor.ForecastMonitor` com o código a seguir:

```
class ForecastMonitor (context: Context) : Repository.Listener {
    private val wm = WorkManager.getInstance(context)
    private val nm = context.getSystemService(Context.NOTIFICATION_SERVICE)
            as NotificationManager

    private fun updateMonitor(city: City) {
        cancelCity(city)

        if (!city.isMonitored) return;
        val inputData = Data.Builder().putString("city", city.name).build()
        val request = PeriodicWorkRequestBuilder<ForecastWorker>(
            repeatInterval = 15, repeatIntervalTimeUnit = TimeUnit.MINUTES
        ).setInitialDelay(
            duration = 10, timeUnit = TimeUnit.SECONDS
        ).setInputData(inputData).build()
        wm.enqueueUniquePeriodicWork(city.name,
            ExistingPeriodicWorkPolicy.CANCEL_AND_REENQUEUE, request )
    }

    private fun cancelCity(city : City) {
        wm.cancelUniqueWork(city.name)
        nm.cancel(city.name.hashCode())
    }

    private fun cancelAll() {
        wm.cancelAllWork()
        nm.cancelAll()
    }

    override fun onUserLoaded(user: User) {  /* DO NOTHING */  }

    override fun onUserSignOut() = cancelAll()

    override fun onCityAdded(city: City) = updateMonitor(city)

    override fun onCityRemoved(city: City) = cancelCity(city)

    override fun onCityUpdated(city: City) = updateMonitor(city)
}
```

Essa classe é um `Listener` do repositório. Sempre que uma cidade precisar ser monitorada (`isMonitored == true`), um `ForecastWorker` é lançado em *background* usando o `WorkerManager` da plataforma Android. Será criado um Worker para cada cidade monitorada.

Passo 4: Crie a classe WeatherApp como a seguir:

```
class WeatherApp : Application() {
    override fun onCreate() {
        super.onCreate()
        val monitor = ForecastMonitor(this)
        val repo = Repository(monitor)
    }
}
```

Essa classe é classe principal da nossa aplicação Android (antes estávamos usando uma implementação *default*). Nesse caso, usamos essa classe para instanciar o `ForecastMonitor` e associar a um repositório, que irá notificar o monitor em caso de mudança das cidades.

Passo 5: Em `MainActivity`, adicione o trecho abaixo dentro de `setContent()`, depois de criar o repositório.

```
 DisposableEffect(Unit) {
    val listener = Consumer<Intent> { intent ->
        val name = intent.getStringExtra("city")
        val city = viewModel.cities.find { it.name == name }
        viewModel.city = city
        if (city != null) {
           repo.loadWeather(city)
           repo.loadForecast(city)
        }
     }
     addOnNewIntentListener(listener)
     onDispose { removeOnNewIntentListener(listener) }
}
```

Esse código trata o recebimento do Intent que é lançado ao clicar na notificação criada pelo `Worker`. Esse tratador seleciona a cidade monitorada no `ViewModel` e também lança a carga das condições climáticas atuais.

Passo 6: No AndroidManifest.xml, faça as seguintes modificações:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.POST_NOTIFICATIONS" />

    <application
        android:name=".WeatherApp"
        ...>
        ...
    </application>
</manifest>
```

Essas modificações adicionam a permissão de lança notificações ao usuário e também indicam explicitamente o uso da classe `WeatherApp` como principal do aplicativo.

Passo 7: Rode e teste o aplicativo.

Veja se ao fazer uma cidade ser monitorada, uma notificação aparece em cerca de 10 segundos (deve aparecer novamente notificação a cada 15 minutos). Clique na notificação para abrir o aplicativo.

Se estiver tudo certo, faça um novo *commit*.