Añadir corrutinas e intents a nuestra agenda

Descargar estos apuntes

Vamos a partir de la solución del ejercicio 3 del tema 3.6 `3_formulario_add_contacto_agenda`. Si no dispones de una solución del mismo puedes pedírsela al profesor. Una vez copiado vamos a renombrarlo a `2_intents_agenda`

Realizando peticiones al repositorio de forma síncrona

Paso 1: Modificar el repositorio para que las peticiones sean asíncronas

En primer lugar vamos ha hacer que los métodos en `ContactoRepository` simulen un retardo aleatorio entre 0 y 1 segundo. Para ello vamos a crear un método privado `retardoAleatorio()` que nos lo genere.

```
private fun retardoAleatorio() = Thread.sleep((Math.random() * 1000).toLong())
```

Al introducir este retardo los métodos de CRUD ahora deberán ser asíncronos y ejecutarse en el contexto de un hilo secundario. Para ello vamos a utilizar `withContext(Dispatchers.IO)` siguiendo el siguiente esquema.

```
suspend fun get(): MutableList<Contacto> = withContext(Dispatchers.IO) {
    retardoAleatorio()
    dao.get().map { it.toContacto() }.toMutableList()
}
```

Cambia el resto de métodos de `ContactoRepository` para que sigan el mismo esquema.

Paso 2: Modificar los ViewModel para usar peticiones asíncronas

Vamos ahora a modificar `.ui.features.formcontacto.ContactoViewModel` para que use los métodos asíncronos del repositorio.

```
fun setContactoState(idContacto: Int) {
    viewModelScope.launch {
        ...
    }
}

    ...
    is ContactoEvent.OnSaveContacto -> {
        ...
        runBlocking {
            if (editandoContactoExistenteState) {
                contactoRepository.update(c)
            } else {
                contactoRepository.insert(c)
            }
        }
        ...
    }
    ...
```

> ✋ **Importante**: Fíjate que el `insert` y el `update` se ejecutan dentro de un bloque `runBlocking` . Esto es para crear un contexto de corrutina **bloqueante** para evitar efectos colaterales al volver posteriormente a la pantalla de de listar contactos. Esto sucede porque cómo esto pude tardar un tiempo, si lo hacemos totalmente asíncrono puede suceder que se cargue la lista antes de la inserción o actualización y no se vea reflejados los cambio. Por eso, hasta que no termine el proceso no saldremos de la pantalla bloqueado el hilo principal.
> Aunque no los vamos aver, usando `Flows` se solucionará el problema y podremos hacerlo de forma totalmente asíncrona.
> También, podríamos utilizar algún tipo de componente que nos indique espera como `CircularProgressIndicator` de Material Design.

Vamos ahora a modificar ahora `.ui.features.vercontactos.ListaContactosViewModel` para que use los métodos asíncronos del repositorio.

Marcaremos como suspendido el método privado `getContactos()` y su llamada en el constructor. Por esa razón ya no podremos llamarla directamente en la definición de la propiedad sino en una función `init` quedando un código similar al siguiente.

```
private var _listaContactosState by mutableStateOf(mutableListOf<ContactoUiState>())
val listaContactosState: List<ContactoUiState>
    get() = _listaContactosState

private suspend fun getContactos(): MutableList<ContactoUiState> =
contactoRepository.get().map { it.toContactoUiState() }.toMutableList()

init {
    viewModelScope.launch {
        _listaContactosState = getContactos()
    }
}
```

Por último, debes realizar las operaciones de borrado y actualización de contactos de forma asíncrona en su callback usando `viewModelScope.launch { ... }`.

Usando intents para llamar, enviar correo y cambiar el avatar

Paso 1: Cambiar el avatar al añadir o editar un contacto

Siguiendo los pasos vistos en el tema y ayudándonos de las utilidades definidas en `RegistroContratosCompose.kt` vamos primero a añadir los permisos necesarios a `AndroidManifest.xml` y después vamos a `.ui.features.formcontacto.FormContactoScreen` y en la función de composición `CabeceraFoto` vamos a definir registrar los contratos para hacer las operaciones definidas en los `OutlinedIconButton` para cambiar el avatar.

> 📌 **Nota**: Si ves que no se actualiza el componente ImagenContacto. Posiblemente es porque no se recompone al cambiar el valor de la propiedad `foto`. Para solucionarlo puedes usar `remember` indicándole que se recompute cuando cambie el valor de `foto`. Por ejemplo:
>
> ```
> val imagenSinFoto = rememberVectorPainter(image = Icons.Filled.Face2)
> var painterFoto = remember(foto) {
>     foto?.let { BitmapPainter(it) } ?: imagenSinFoto
> }
> ```

Paso 2: Añadir la funcionalidad de llamar y enviar correo

Para ello vamos a `.ui.features.vercontactos.ContactoListItem` donde definimos la tarjeta donde se muestran en la lista los datos de un contacto.

Fíjate que en la función de composición `AccionesContacto` donde aparecen los botones de icono con las posibles acciones a realizar sobre el contacto seleccionado nos faltan por definir dos callbacks para las acciones de llamar `onLlamarClicked` y enviar correo `onCorreoClicked`.

Siguiendo los pasos vistos en el tema y ayudándonos de las utilidades definidas en `RegistroContratosCompose.kt` vamos a definir dichos callbacks.