Proyecto Usuarios con Arquitectura

Descargar estos ejercicios

Ejercicio

Para crear nuestro primer proyecto siguiendo la arquitectura explicada en los apuntes, vamos a realizar un ejercicio guiado que posteriormente ampliaremos. **La app va a seguir siendo de CONSOLA de Kotlin**, y tratará de gestionar la entrada a una aplicación mediante las típicas opciones para logearse.

1. Creamos el paquete `data`
   Vamos a comenzar creando los elementos necesarios para la manipulación de los datos, paquete `data`. Como todavía no tenemos conocimiento de acceso a BD con Android, vamos a usar la simulación de esta mediante colecciones, y como ya se ha explicado en el tema, la simulación se creará dentro del paquete `mocks` de `data`:
   1. Clase **UsuarioMock**, data class con `login` y `password` de tipo `String`.
   2. Clase **UsuarioDaoMock**, clase en la que crearemos una lista mutable con los usuarios para pruebas y a la que ademas le añadiremos los métodos que nos permitan manipular estos datos:

      ```
      get(): MutableList<UsuarioMock>
      get(login: String): UsuarioMock?
      insert(usuarioRemoto: UsuarioMock)
      update(usuarioRemoto: UsuarioMock)
      delete(login: String)
      ```

2. Creamos el paquete `models`
   Como hemos visto en el tema, en el paquete `data` deberemos tener una clase `Repositorio` que nos haga de puente entre UsuarioMock y el tipo modelo correspondiente, por lo que antes de nada pasaremos a crear en el paquete `models` (a nivel de `data` cumpliendo la arquitectura de usuario) la data class `Usuario` con las mismas propiedades que `UsuarioMock`.

3. Creando el repositorio en el paquete `data`

   Volvemos a `data` y:

   1. Creamos la clase **UsuarioRepository**, que tendrá una *variable inmutable* de tipo UsuarioDaoMock a través de la cual podremos acceder a los métodos del DAO creando los métodos paralelos, pero esta vez los datos de entrada y salida serán de tipo Usuario.
   2. Archivo **RepositoryConverter** en este archivo crearemos todos los métodos que sirven para mapear ente *Usuario* y *UsuarioMock*, además de los de mapeo entre *Usuario* y *UsuarioUiState* que veremos después.

4. Definiendo el paquete `ui.features`

   - **Lógica de la aplicación**

     Una vez terminada la parte de los datos y modelo, vamos a generar el código necesario de la UI. Como todavía no estamos trabajando la parte de vistas con Compose, no tendremos que crearnos las carpetas de componentes, pero si crearemos el paquete `features`, en el cual meteremos los diferentes archivos de la vista de usuario, como puede que con posterioridad la aplicación crezca en contenido, vamos a crear un paquete `usuario` dentro de features, donde tendremos lo referente a la parte de la aplicación que estamos resolviendo, y que constará de lo siguiente:

     1. La data class `UsuarioUiState`, que será igual que Usuario pero con una propiedad más `estaLogeado` de tipo Boolean, que servirá para controlar si el usuario, en determinado momento, ha entrado correctamente al sistema.
     2. el archivo `UsuarioEvent`, que como hemos visto en temas anteriores, va a contener una **interface sellada** con los eventos que pueden ocurrir en la aplicación:
        1. `AñadeUsuario` al que le llega un login y password para posteriormente crear el usuario.
        2. `EntraSistema` al que le llega un login y password y si coinciden con alguno de los usuarios del sistema, cambiará el estado de esta Logeado del usuarioUiState.
        3. `SaleSistena` que cambia la anterior variable a false.
        4. `MuestraUsuarios` que se encargará de mostrar la lista de usuarios del sistema (esto no tiene sentido en una app, pero lo usamos a modo de comprobación de que todo está correcto).
        5. `ModificaUsuario` al que le llega un password y si el usuario está logeado, se encargará de modificar el usuario con el nuevo password.
     3. La clase `UsuarioViewModel` encargada de toda la parte de la lógica de la aplicación, tendrá dos variables una de tipo `UsuarioRepository` y otra de tipo

`UsuarioUiState`. Esta última será la que guarde el valor del usuario logeado (si se logea un usuario satisfactoriamente, se guardará en esta variable. Si todavía no se ha logeado o se realiza logout, esta variable pasará a ser null). Mientras que `usuarioRepository` la utilizaremos para el tratamiento de los datos de los usuarios.

En esta clase tendremos la función `onUsuarioEvent` para gestionar los posibles eventos del tipo `LoginEvent` como se ha indicado en el punto anterior (intenta resolver esta parte, pide ayuda al profesor si no se te ocurre como).

- **Interfaz de usuario**

  Ahora nos quedaría hacer la parte de la aplicación que se encarga de la interacción con el usuario. Para ello crearemos un archivo `UsuarioScreen` donde alojaremos las siguientes funciones:

  1. **usuarioScreen** a la que le llegará una variable usuarioEvent de tipo `(UsuarioEvent) -> Unit` y que teniendo en cuenta el siguiente menú:

     ```
     1. Sign in
     2. Login
     3. Change password
     4. Logout
     5. List
     6. Exit
     ```

  Se encargará de gestionar los eventos posibles pidiendo los datos necesarios al usuario, por ejemplo, para el primer caso se podría codificar algo como los siguiente:

```
when (opcion) {
    "1" -> {
        print("Introduce login ")
        val login = readln()
        print("Introduce la contraseña ")
        val password = readln()
        usuarioEvent(UsuarioEvent.AñadeUsuario(login, password))
    }
    ...
}
```

Pista

Para hacer posible la lógica del programa sin usar las clases de Android (recuerda que estamos en una aplicaciónd e consola), vamos a utilizar una serie de funciones y propiedades que en un caso real no serían necesarias. Por tanto, dentro de `UsuarioScreen.kt` crearemos la función:

1. `mostrarInformacion` a la que le llegará la lista de usuarios y mostrará la salida por pantalla, esta función será llamada desde el evento correspondiente gestionado en `UsuarioViewModel`.
2. `actualizaEstado` a la que le llega `estadoUsuarioUiState: UsuarioUiState` y se encargará de actualizar una variable global de este archivo y que nos servirá para controlar si se ha logeado el usuario correctamente, a esta función la llamaremos desde la clase `UsuarioViewModel` en cada interacción con los eventos (de forma que utilizaremos la propiedad `usuarioUiState` del ViewModel para actualizar esta variable). Esto lo haremos porque la aplicación mostrará un mensaje antes de la salida del menú, indicando que el usuario está logeado o no lo está. Solo podremos cambiar la contraseña de un usuario logeado.

```
Usted no está logeado
3. Sign in
4. Login
...
5. Sign in
6. Login
7. Change password
8. Logout
9. List
10. Exit
2
Introduce login pepe
Introduce la contraseña pepe1234

Usuario: pepe logeado
11. Sign in
12. Login
```

5. Definiendo el paquete `views`
   En este paquete solo tendremos la función `main` que contendrá una variable inmutable de tipo viewModel y realizará la llamada a la función usuarioScreen.