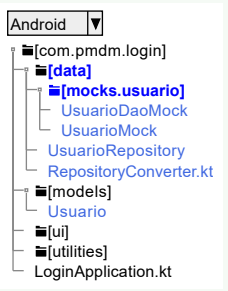
Aplicando ViewModel a Login

Descargar estos apuntes

Ejercicio

En el siguiente ejercicio partiremos del ejercicio 3 del Tema 3.3 donde definíamos los elementos composables para crear el LoginScreen. Ahora vamos a ir un paso más allá y añadir la clases necesarias para llevar la lógica de la aplicación al `ViewModel` además de incluir una fuente de datos que no teníamos en el ejercicio del que partimos. Para ello vamos a añadir toda la estructura de paquetes correspondientes a data y a model necesarios para seguir con la arquitectura que hemos propuesto en este curso.

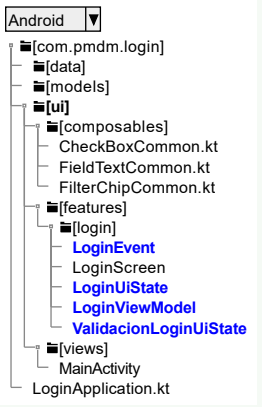
Paquete Data y Model



Como ya hemos visto en ejercicios anteriores, debemos separar las fuentes de datos de la lógica de empresa, por lo que nos crearemos los paquetes necesarios **y que ya conocemos sobradamente**, con la data class **UsuarioMock** (login y password de tipo string), con la clase **UsuarioDaoMock** (en la que tendremos una lista de usuarios con algunos usuarios para pruebas), la clase para el repository y la clase para los mapeados de datos. Además, dentro del paquete models crearemos nuestra clase modelo Usuario con las mismas propiedades que UsuarioMock.

Paquete Ui

Dentro del paquete `ui.features.login` vamos a añadir las clases necesarias para crear la lógica de negocia con la ayuda de ViewModel. Para ello comenzaremos creando la **interface sellada LoginEnvent** que permite centralizar los eventos que ocurren en el Screen, por ejemplo:



- LoginChanged
- PasswordChanged
- OnClickLogearse

En la **data class LoginUiState** añadiremos el boolean estaLogeado a las propiedades del modelo Usuario.

Vomos a crear una clase que se utilizará para encapsular los states de las validaciones y la llamaremos **data class ValidacionLoginUiState** que heredará de la **interfaz Validacion** que ya conocemos de ejercicios anteriores, y que para que tengáis una primera aproximación os pasamos el código:

```
data class ValidacionLoginUiState(
    val validacionLogin: Validacion = object : Validacion {},
    val validacionPassword: Validacion = object : Validacion {},
) : Validacion {
    override val hayError: Boolean
        get() = validacionLogin.hayError || validacionPassword.hayError
    override val mensajeError: String?
        get() = if (validacionLogin.hayError) validacionLogin.mensajeError
        else if (validacionPassword.hayError) validacionPassword.mensajeError
        else ""
}
```

> 💡 **Tips:** Tendremos que modificar la función LoginScreen para que tenga la siguiente entrada de argumentos:
> `fun LoginScreen(  usuarioUiState: LoginUiState,  validacionLoginUiState: ValidacionLoginUiState,  onLoginEvent: (LoginEvent) -> U`

Y por último crearemos la clase estrella de este ejercicio **LoginViewModel**, en la que tendremos la lógica de la aplicación y que nos hará de nexo entre las fuentes de datos y la interface de usuario. Recuerda que deberemos crear un objeto de tipo *UsuarioRepository* para acceder a la fuente de datos y poder usar las funciones *CRUD* de esta. En este ejercicio será mas bien de consulta, ya que solo podremos logearnos con alguno de los usuarios que ya tengamos en nuestros datos (no tenemos funcionalidad para crear nuevos usuarios).
En esta clase además, deberemos tener todas las variables **StateFul** necesarias, para seguir la lógica de **state-hoisting** (elevación de estado) vista en los temas. Y el método **onLoginEvent** para gestionar los eventos.
Además será aconsejable crear dos métodos validaPassword y validaLogin que se encargarán de cambiar los estados de la validación para las entradas de los TextField.

⚠️ **Aviso:** Para mostrar los avisos de entrar a la aplicación si hemos introducido un login y password correcto o en caso incorrecto indicar que no existe el login o password. Podemos usar un layout `Box` que envuelva todo nuestro contenido y un texto alineado en la parte inferior que se muestre o no con el mensaje correspondiente, puede desaparecer cuando detecte algún cambio en los `TextField`.