

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. Para partir de la misma base y que sea más sencillo el seguimiento de este ejercicio, lo primero que haremos será renombrar el paquete a **arquitectura_usuario**


1. Creando el paquete `data`

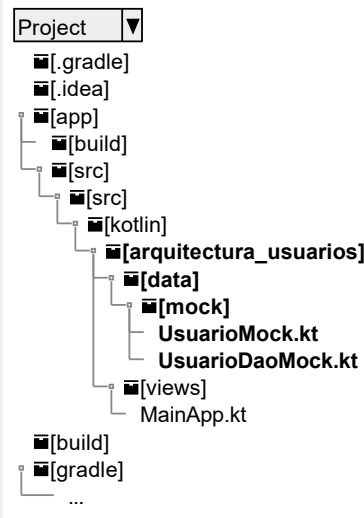
Vamos a comenzar creando los elementos necesarios para la manipulación de los datos, paquete `data`. Como ya hemos visto en el tema, el paquete `data` lo crearemos directamente dentro del paquete base **arquitectura_usuario**.

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`, por lo que el siguiente paso será crear el paquete `mocks` dentro del paquete `data`:

1. En un archivo nuevo llamado `UsuarioMock.kt` se creará la **data class UsuarioMock**, con las propiedades inmutables `login` y `password` de tipo `String`.
2. En un archivo nuevo llamado `UsuarioDaoMock.kt` se creará la Clase **UsuarioDaoMock**, clase en la que crearemos una lista mutable con los usuarios para pruebas y a la que además 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)
```

 **Tips:** En la siguiente imagen puedes ver la estructura de carpetas afectada hasta el momento.



2. Creamos el paquete `models`

Como hemos visto en el tema, para cumplir con la arquitectura de capas propuesta, deberemos tener las clases Modelo necesarias que independicen las fuentes de datos de la lógica de la aplicación, por lo que deberemos crear el paquete `models` (que situaremos a nivel de `data` cumpliendo la arquitectura propuesta), dentro de este paquete se creará la **data class `Usuario`** correspondiente a `UsuarioMock` y que en un principio tendrá las mismas propiedades que esta última.

3. Creando el repositorio en el paquete `data`

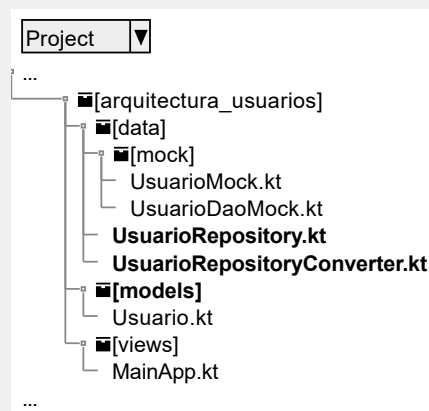
Volvemos a `data` y deberemos crear las clases `Repositorio` que nos hagan de puente entre los Mocks y los Modelos, en nuestro caso solo tenemos `UsuarioMock` y el tipo modelo correspondiente `Usuario`. Por lo que deberemos hacer lo siguiente:

1. Crearemos el archivo **UsuarioRepository.kt** dentro del paquete `data`, con la clase `UsuarioRepository` que tendrá una *propiedad inmutable* de tipo `UsuarioDaoMock` a través de la cual podremos acceder a los métodos de esta clase. Para ello crearemos los métodos paralelos a los de la clase DAO, pero esta vez los datos de entrada y salida serán de tipo `Usuario`. Por ejemplo, tendremos el siguiente método en `UsuarioRepository`:

```
fun get():List<Usuario> = proveedorUsuarios.get().toUsuarios()
```

2. También dentro de `data` crearemos el archivo **RepositoryConverter.kt** en el que incluiremos todos los métodos que sirven para mapear entre `Usuario` y `UsuarioMock`, como se ha explicado en el punto 3 de *estructurando la capa de datos* del tema 2.3 de Arquitectura.

💡 **Tips:** Los anteriores elementos habrán quedado como en la siguiente imagen:



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 `ui.features` dentro de `arquitectura_usuarios`, 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:

- i. Fichero `UsuarioUiState.kt` con la **data class `UsuarioUiState`**, que será igual que la data class `Usuario` del paquete `models`, pero con una propiedad más **`estaLogeado`** de tipo `Boolean`, que servirá para controlar si el usuario, en determinado momento, ha entrado correctamente al sistema.
- ii. 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:
 - **`AñadeUsuario`** al que le llega un login y password para posteriormente crear el usuario.
 - **`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`.
 - **`SaleSistema`** que cambia la anterior variable a false.
 - **`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).
 - **`ModificaUsuario`** al que le llega un password y si el usuario está logeado, se encargará de modificar el usuario con el nuevo password.

⚠ Aviso: Aunque se ha explicado la funcionalidad de los eventos de la interfaz sellada, deberemos tener en cuenta que su codificación la realizaremos en la parte correspondiente a la vista, que se explica posteriormente.

- iii. Dentro de **`features.usuario`** también crearemos el fichero `UsuarioViewModel.kt` con 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 extraídos de la fuente de datos que hemos preparado con anterioridad.

En esta clase tendremos la función **`onUsuarioEvent`** para gestionar los posibles eventos de la interfaz sellada **`UsuarioEvent`**.

• 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.kt`** dentro de **`ui.feature`**, donde alojaremos las siguientes funciones:

- i. **`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))  
    }  
    ...  
}
```



Para hacer posible la lógica del programa sin usar las clases de Android (recuerda que estamos en una aplicación de 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

- i. `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`.
- ii. `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
1. Sign in
2. Login
...
1. Sign in
2. Login
3. Change password
4. Logout
5. List
6. Exit
2
Introduce login pepe
Introduce la contraseña pepe12
Usuario: pepe logeado
1. Sign in
2. 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`. De la siguiente manera:

```
val usuarioViewModel = UsuarioViewModel()
usuarioScreen(usuarioViewModel::onUsuarioEvent)
```

💡 **Tips:** Al final la arquitectura te habrá quedado como la siguiente, resaltados los últimos elementos añadidos:

