**UT-01: Desarrollo de Aplicaciones Móviles**

* **Vistas**:
* Android: sólo archivos exclusivos de Android (clases, recursos, gradle).
* Project: todos los archivos del proyecto, incluyendo configuraciones del entorno.
* **Invalidate caches**: permite borrar caches y puede resultar útil cuando se produce algún error de empaquetado y no se haya detectado la causa.
* **Gradle**: se encarga de la compilación y el empaquetado de la aplicación, agregando todas las librerías externas necesarias.

Como el proceso de compilación es muy lento se tiende a la modularización de las aplicaciones, pudiendo compilar a nivel de aplicación o a nivel de módulo, existiendo un archivo de configuración de gradle a nivel a aplicación y a nivel de cada módulo y todas se llaman “build.gradle”.

* **Versión code**: es un número entero y es obligatorio incrementarlo con cada nueva versión de la aplicación.
* **Versión name**: es la versión de la app que se muestra al usuario, y suele utilizarse el código de 3 cifras separadas por puntos de cualquier aplicación: version.update.bugfix
* **Dependencies**: permite incorporar librerías externas a nuestro proyecto.
* **Settings.gradle**: permite definir los repositorios desde los que se descargan las librerías: maven, google, etc.
* **Control de versiones**: el desarrollo de aplicaciones actual hace imprescindible realizar un control de versiones (GitHub) que se puede combinar con integración continua para que siempre esté disponible la última versión de la apk (Jenkins, GitHub Action).
* Master: rama principal con el código estable.
* Ramas: permiten clonar el contenido de la rama original para poder trabajar sobre ella sin interferir en el resto.

La versión de profesor de Github ofrece GitHub Action de forma gratuita.

En el repositorio podemos agregar al gitignore los archivos que no queremos agregar al proyecto, como la carpeta de configuraciones “.idea”. Podemos hacerlo de forma manual o utilizar la plantilla Android de GitHub dentro de la opción “Add .gitignore”.

Dentro de Android Studio activamos el control de versiones para un proyecto en “VCS” mediante la opción “Enable Versión Control Integration” (necesitaremos instalar GIT).

Con esta configuración podemos realizar un control de versiones local, pero lo más adecuado es hacer esa copia en un repositorio externo (nuestro repositorio en GitHub):

> git remote add origin url\_repositorio

Desde el icono de “commit” podemos ver los archivos que no se han sincronizado, con la vista (icono ojo), podemos mostrar la información por directorio o módulo (agrupado).

En el archivo “.gitignore” ponemos “/.idea/\*” para que no se sincronicen los archivos de configuración sin tener que ir agregando cualquier archivo de configuración nuevo que se vaya creando.

Por último, hacemos commit para que se guarden los cambios.

Cuando agreguemos algún archivo nos preguntará si queremos que los cambios se actualicen automáticamente, pero no es lo más adecuado porque puede que creemos alguna clase que no funcione como queremos que se sincronice, así que mejor poner la opción de que no pregunte de nuevo y cancelar, y seamos nosotros los que hagamos commit cuando deseemos.

Para crear una rama, desde la pestaña de git utilizamos “create branch” y establecemos el nombre de la rama.

Para subir los archivos a nuestro repositorio en github utilizamos la opción “push”, para ello tendremos que disponer de un token o generarlo en ese momento haciendo login en nuestra cuenta de github.

Token: ghp\_GG8p7TxMlvEXRkV43JOuo91C56zTcj09Fb5t

Podemos cambiar de una rama a otra en Android Studio mediante la opción “checkout”.

Para cambiar la rama principal en Github utilizaremos la opción “Settings”, “Branches”, “Default Branch”

Cuando vamos a fusionar los nuevos cambios de una rama sobre master es recomendable hacer un pull de master previamente por si hubiera cambios en la rama principal.

NOTA: podemos crear un repositorio vacío, sin utilizar gitignore en GitHub, y descargarnos una plantilla gitignore para aplicar directamente en el archivo de Android Studio antes de hacer la sincronización.

<https://github.com/github/gitignore/blob/main/Android.gitignore>

* **Readme**: podemos crear un archivo readme utilizando markdown (readme.md):

# titulo -> permite crear títulos h1(#), h2(##), etc

\*\* texto \*\* -> negrita

[nombre\_enlace](ruta\_enlace)

* **Emuladores**: permiten probar las aplicaciones sin necesidad de disponer de un dispositivo móvil, aunque no son fiables al 100%.

**UT-02: Arquitectura Android**

La arquitectura Android se compone de capas en la que se proponen 3 capas, pero se pueden añadir tantas capas como se necesiten:

* Presentación: abarca toda la vista de la aplicación, está muy ligada al SDK de Android.
* Dominio: abarca la lógica de negocio (casos de uso) y debe ser exclusivamente Kotlin.

Funciones como obtener listado de usuarios, eliminar usuario, etc.

* Datos: abarca los datos y la funcionalidad para la manipulación de los datos, utilizando el patrón repositorio.

Los datos pueden ser locales (BD o XML local) o remotos (API), y el desarrollo debe realizarse en modo offline y se utilizan caches para la obtención de los datos remotos.

* **Principios SOLID**:
* Single Responsability : cada clase/función debe realizar una única cosa: recuperar usuario.
* Open/Close: cada clase debe ser independiente del resto.
* Liscov: cualquier clase debe poder ser reemplazada por sus clases bases.
* Interface Segregation Principle: integridad de interfaces, es preferible tener muchas interfaces con pocas funciones.
* Dependency Inversion: evitar las dependencias entre modulos.
* **Estructura** **aplicación Android**:
* androidTest: incluye los test unitarios que requieren Android, estos test son muy lentos porque requiere lanzar emuladores, así que se suelen realizar en horario que no afecte al desarrollo.
* test: test unitarios que no requieren Android y por tanto son mucho más rápidos.
* main: incluye todo el código de la aplicación: clases (java), recursos (imágenes, iconos, archivos xml, carpeta xml para almacenar claves de APIS, etc).
* manifest.xml: define las propiedades de nuestra aplicación: nombre, icono, API objetivo, tema, pantallas, pantalla principal, permisos.
* **Arquitectura**
* app: clases de uso general como acceso a API, etc.
* features: funcionalidades de la aplicación (casos de uso).

Ejemplo: login (data, domain, presentation).

LoginActivity

LoginViewModel

SignInUseCase

LoginRepository

LoginDataRepository

LoginMockRemoteDataSource

**Presentation**

**Domain**

**Data**

* **Vistas**

Las propiedades con el espacio de nombres “tools” se utilizan solo para la vista previa pero no tendrán efecto sobre la vista de la aplicación.

Por ejemplo, un textView con la propiedad:

tools:text=”Hola”

Mostrará el texto “Hola” en el textview de la vista previa, pero no al ejecutar la aplicación.

Ctrl + Alt + Intro => formatear código (ordena propiedades por orden alfabético)

Ctrl + Alt + O => eliminar imports no usados

* **Implementación de la arquitectura**

Es conveniente utilizar el código top and bottom que consiste en dividir el código de los estados onCreate, onStart, etc en métodos e irlos llamando en orden sucesivo:

override fun onCreate(savedInstanceState: Bundle?) {  
 super.onCreate(savedInstanceState)  
 setContentView(R.layout.*activity\_login*)  
 setupView()  
}  
  
private fun setupView() {  
 val actionValidate = findViewById<Button>(R.id.*action\_validate*)  
 actionValidate.setOnClickListener **{** Log.d("@dev", "clicked!")  
 **}**}

Por cada Activity creamos una clase ViewModel que sobrevive a la vida del Activity y de esta manera mantener los objetos por ejemplo cuando se gira la pantalla.

Por cada funcionalidad (feature) creamos una clase Factory que se encarga de instanciar los objetos ViewModel, casos de uso, etc.

Cada caso de uso (domain) debe realizar una sola cosa, no debe tener ninguna referencia a Android, y contará con un único método público (invoke) y todos los que se necesiten privados.

Cuando una interfaz solo tenga un método a implementar se sustituye por una función lambda que recibe sus parámetros. Cuando hay un solo parámetro por defecto se llama it, y si hay varios hay que establecer el nombre de cada uno.

Todas las dependencias entre clases se van a pasar a través del constructor, que en kotlin se agrega como si fueran parámetros de la definición de la clase:

class LoginViewModel(private val singInUseCase: SignInUseCase: SignInUseCase) : ViewModel () {

**UT-03: Interfaz de Usuario**

* **Diseño**:
* XML (tradicional)
* Compose (nuevo)
* Mediante programación (poco usado)
* Imágenes de prueba

tools:src="@tools:sample/avatars"