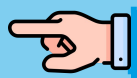
Testing

Mock y gradle

{desafío} latam_

Implementar tests unitarios y de instrumentación para la verificación del buen funcionamiento de los componentes de un proyecto Android.

- Unidad 1:
  Acceso a datos en Android
- Unidad 2:
  Consumo de API REST
- Unidad 3:
  Testing
- Unidad 4:
  Distribución del aplicativo Android



Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Qué es un mock*
- *Distintos mocks (clases, métodos)*
- *Configuración del gradle*
- *BuildVariants de testing*

¿Has escuchado el término Prueba Unitaria o Unit Test?

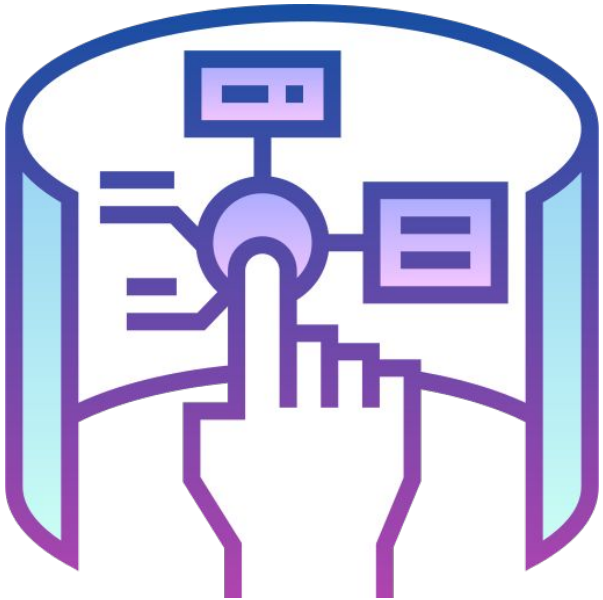
{desafío} latam_

/* Qué es un mock */

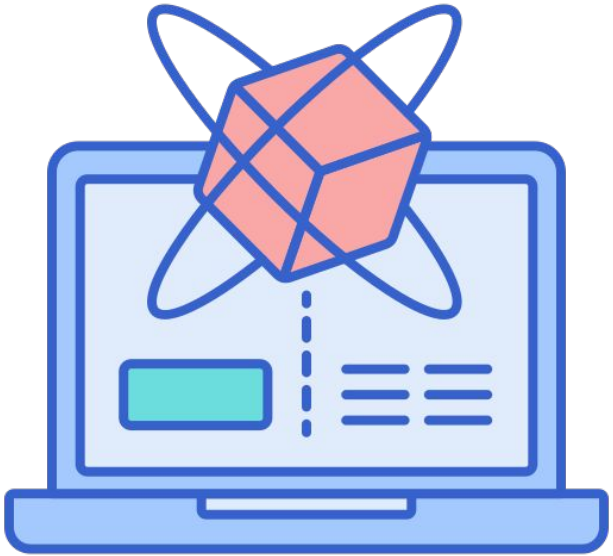
Qué es un mock

Un Mock es un simulacro, un doble de prueba que simula el comportamiento de un objeto real de forma controlada. El propósito de usar un objeto simulado es aislar el comportamiento del objeto que se está probando del comportamiento de cualquiera de sus dependencias.

Los objetos simulados se utilizan normalmente en las pruebas unitarias, en las que la atención se centra en probar una única unidad de código aislada del resto del sistema. Mediante el uso de un objeto simulado para simular el comportamiento de una dependencia, la unidad que se está probando se puede evaluar en un entorno controlado sin necesidad de interactuar con la dependencia real.



Qué es un mock



Por ejemplo, supongamos que estamos probando una clase que realiza llamadas a una API. Puedes crear un objeto simulado que simule el comportamiento de la API, lo que le permite probar la clase de forma aislada **sin realizar ninguna llamada de red**.

Hay varias bibliotecas disponibles para crear objetos simulados en las pruebas de Android, como Mockito y MockK. Estas bibliotecas brindan una manera conveniente de crear objetos simulados y definir su comportamiento en tu código de prueba.

/* Distintos mocks (clases, métodos) */

Distintos mocks (clases, métodos)

- **Dummy:** un objeto dummy es un objeto simulado que no hace nada, pero se usa para satisfacer las dependencias del código que se está probando. Por ejemplo, si un método toma un objeto como parámetro pero no lo usa, puede pasar un objeto ficticio para satisfacer la firma del método.

- **Stub:** un stub es un objeto simulado que devuelve valores predeterminados en respuesta a llamadas a métodos. Por ejemplo, si un método realiza una llamada de red para recuperar datos, puede usar un código auxiliar para devolver una respuesta predeterminada en lugar de realizar una llamada de red real.

- **Espía (Spy):** un espía es un objeto simulado que envuelve un objeto real y te permite observar su comportamiento. Por ejemplo, si desea probar un método que modifica el estado de un objeto, puede usar un espía para verificar que el estado se modifique correctamente.

Distintos mocks (clases, métodos)

- **Mock**: Un simulacro es un término genérico para cualquier doble de prueba que simula el comportamiento de un objeto real. En general, los simulacros se utilizan para aislar el comportamiento del objeto que se está probando del comportamiento de sus dependencias.

- **Fake:** un objeto fake es una implementación simplificada de un objeto real que proporciona una interfaz similar pero es más fácil trabajar con él en las pruebas. Por ejemplo, puede usar una implementación de base de datos falsa que almacene datos en la memoria en lugar de en el disco.

- **Doble:** un doble es un término genérico para cualquier objeto de prueba que reemplaza un objeto real en el código que se está probando. Esto incluye simulacros, stubs, dummies y fakes.

Distintos mocks (clases, métodos)

Estos diferentes tipos de simulacros se pueden usar en combinación para crear escenarios de prueba complejos que le permitan probar su código a fondo. Es importante elegir el tipo correcto de simulacro para cada situación a fin de crear pruebas efectivas que detecten errores y garanticen la confiabilidad de su aplicación.

/* Configuración del gradle */

Configuración del gradle

Configurar Gradle para realizar pruebas en un proyecto Android, debes agregar las siguientes dependencias a tu archivo `build.gradle`:

```
dependencies {
    // JUnit
    testImplementation 'junit:junit:4.13.2'

    // Android JUnit Runner
    testImplementation 'androidx.test:runner:1.4.0'

    // Android JUnit Rules
    testImplementation 'androidx.test:rules:1.4.0'

    // Espresso for UI testing
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
}
```

Configuración del gradle

Aquí hay una breve explicación de cada dependencia:

- **junit:** junit proporciona el marco de pruebas JUnit.
- **androidx.test:runner** proporciona un ejecutor de pruebas específico de Android para pruebas JUnit.
- **androidx.test:rules** proporciona reglas JUnit que simplifican el proceso de escritura de pruebas de Android.
- **androidx.test.espresso:** espresso-core proporciona Espresso, un marco para probar las interacciones de la interfaz de usuario en Android.

Una vez que hayas agregado estas dependencias, puedes crear y ejecutar pruebas en tu proyecto Android.

Demostración
"Primera prueba"

Primera prueba

Para crear una prueba, puedes crear un nuevo archivo Kotlin en el directorio de Test de tu proyecto y anotar los métodos de prueba con `@Test`. Aquí hay un ejemplo:

```kotlin
import org.junit.Test
import org.junit.Assert.assertEquals

class ExampleUnitTest {
    @Test
    fun addition_isCorrect() {
        val sum = 2 + 2
        assertEquals(4, sum)
    }
}
```

{desafío} latam_

Primera prueba

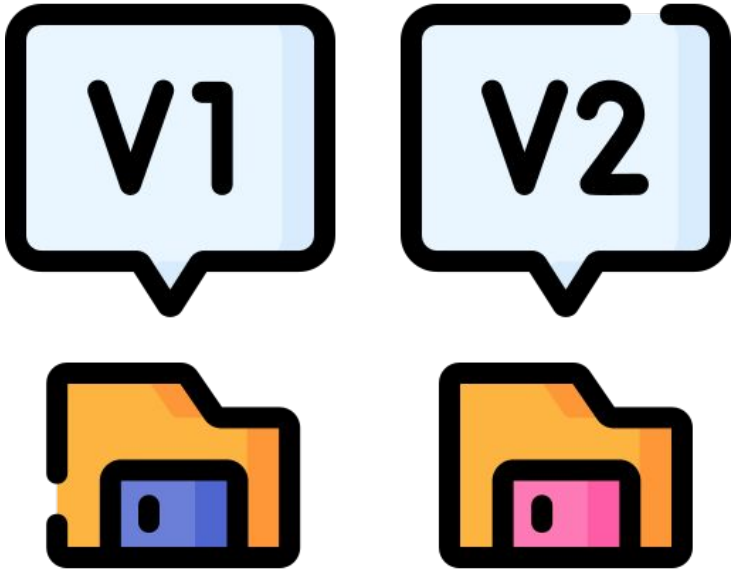
- Para ejecutar pruebas, puede usar la tarea de prueba de Gradle, que se puede ejecutar desde la línea de comando usando “`./gradlew test`”.

- También puede ejecutar pruebas desde Android Studio abriendo el menú "Ejecutar" y seleccionando "Ejecutar 'Todas las pruebas'".

/* BuildVariants de testing */

BuildVariants de testing

Las BuildVariants o variantes de compilación en Android son versiones diferentes de la misma aplicación que se pueden compilar a partir del mismo código base del proyecto. Cada variante de compilación puede tener diferentes ajustes y configuraciones, como diferentes ID de aplicación, claves de firma, recursos y dependencias.



BuildVariants en Tests

En el contexto de las pruebas, las variantes de compilación le permiten ejecutar pruebas en diferentes configuraciones de su aplicación. Por ejemplo, es posible que desee ejecutar pruebas en una variante de compilación de depuración para asegurarse de que su aplicación se comporte correctamente durante el desarrollo y luego ejecutar pruebas en una variante de compilación de lanzamiento para asegurarse de que su aplicación esté lista para la producción.

Variantes de compilación

- Cuando creas un nuevo proyecto en Android Studio, el IDE genera automáticamente dos variantes de compilación: `debug` y `release`. La variante de `debug` está destinada a fines de prueba y depuración, mientras que la variante de `release` está destinada a la distribución.

- Para agregar variantes de compilación adicionales para la prueba, puedes modificar el archivo `build.gradle` de tu proyecto. Por ejemplo, para agregar una nueva variante de compilación llamada `staging` que usa una configuración diferente a la de depuración y lanzamiento, puede agregar el siguiente código:

Variantes de compilación

```
android {
    ...
    buildTypes {
        ...
        staging {
            // Customize the configuration for the staging build variant
            applicationIdSuffix ".staging"
            versionNameSuffix "-staging"
        }
    }
}
```

Ejercicio "Unit Test, promedio de notas"

{desafío} latam_

Unit Test, promedio de notas

Ejemplo de cómo probar una función que calcula la calificación promedio de una lista de estudiantes:

```
// La siguiente función calcula el promedio de notas de un lista de estudiantes
fun calculateAverageGrade(students: List<Student>): Double {
    if (students.isEmpty()) {
        throw IllegalArgumentException("List of students is empty")
    }

    var total = 0.0
    for (student in students) {
        total += student.grade
    }

    return total / students.size
}
```

Unit Test, promedio de notas

Para probar esta función, necesitaremos escribir una prueba unitaria que verifique si la función devuelve el resultado esperado para una entrada determinada. Aquí hay una prueba de ejemplo:

```
@Test
fun testCalculateAverageGrade() {
    // Arrange
    val students = listOf(
        Student("Alice", 80.0),
        Student("Bob", 90.0),
        Student("Charlie", 70.0)
    )

    // Act
    val result = calculateAverageGrade(students)

    // Assert
    assertEquals(80.0, result, 0.1)
}
```

Unit Test, promedio de notas - Explicación

1

En esta prueba, primero creamos una lista de objetos Student con sus nombres y calificaciones. Luego llamamos a la función de calculateAverageGrade() con esta lista como entrada. Finalmente, usamos el método assertEquals para comprobar si el resultado de la función coincide con el valor esperado (que es 80,0 en este caso).

2

Al ejecutar esta prueba, podemos verificar que la función de calculateAverageGrade() funciona correctamente para la entrada proporcionada. También podemos escribir pruebas adicionales para verificar si la función maneja los casos extremos (como una lista vacía de estudiantes) correctamente.

3

En general, probar funciones como esta garantiza que nuestro código funcione como se espera y ayuda a detectar errores o comportamientos inesperados al principio del proceso de desarrollo.

Si bien las pruebas unitarias pueden ser desafiantes, son una parte esencial del proceso de desarrollo de software ¿Podrías decir por qué lo son?

Próxima sesión...

- *JUnit y Mockito*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales