UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Sistemas Informáticos y de Computación

Universitat Politècnica de València

Aplicación de prácticas de DevOps en el desarrollo de una aplicación móvil programada con Flutter

TRABAJO FINAL DE MASTER (TFM)

Master MUITSS

Autor: Andrés Sen Moya

Tutor: Tanja Ernestina Jozefina Vos

Curso 2020-2021

Resumen

Adecuación de una aplicación desarrollada con Flutter a las prácticas de DevOps. Para ello se realizará un conjunto de test automáticos para la parte de CI (Continuous Integration) con las herramientas Flutter Driver (Scripted) y TESTAR (Scriptless) estudiando la complementariedad de estas dos herramientas. También se realizará la configuración para que haga un despliegue automático para la parte de CD (Continuous Delivery).

Palabras clave: DevOps, desarrollo continuado, integración continua, entrega continua, Flutter, TESTAR.

Resum

Adequació d'una aplicació desenvolupada amb Flutter a les pràctiques de DevOps. Per a això es realitzarà un conjunt de test automàtics en la part de CI (Continuous Integration) amb les eines Flutter Driver (Scripted) i TESTAR (Scriptless) estudiant la complementarietat d'aquestes dues eines. També es realitzarà la configuració perquè faça un desplegament automàtic en la part de CD (Continuous Delivery).

Paraules clau:DevOps, desenvolupament continuat, integració contínua, lliurament continu, Flutter, TESTAR.

Abstract

Adaptation of an application developed with Flutter to DevOps practices. For this purpose, a set of automatic tests will be performed for the part of Continuous Integration with Flutter Driver (Scripted) and TESTAR (Scriptless) studying the complementarity of these two tools. The configuration will also be made to make an automatic deployment for the Continuous Delivery part.

Keywords: DevOps, Continuous Development, Continuous Integration, Continuous Delivery, Flutter, TESTAR

Índice de figuras

Índice de tablas

Índice general

CAPÍTULO 1

Introducción, motivación y objetivos

1.1 Introducción

En este trabajo se estudiará la complementariedad de dos diferentes enfoques de testing de sistema en un entorno DevOps. El estudio empieza a partir de una aplicación desarrollada en el Framework multiplataforma Flutter. Para utilizar metodologías DevOps se ha de automatizar ciertas partes existentes en el proceso del software como son el testear una aplicación o el publicarla. Después de añadir metodologías DevOps se realizará un estudio de la complentariedad de dos herramientas de testing automatizado. Estas herramientas son las utilizadas en una de las partes a desarrollar para la adición de DevOps. Este estudio se centrará en dos enfoques las herramientas para el testing a nivel de sistema: scripted y las scriptless.

1.2 Tecnologías

1.2.1. Flutter

Flutter [10] es un Framework multidesarrollo, esta desarrollado por Google y que usa el lenguaje de programación DART. Con Flutter se pueden desarrollar al mismo tiempo aplicaciones Android e iOS y además con la versión 2 del framework, también se puede desarrollar para web, Linux, Windows y Mac OS. A pesar de encontrarse en otras plataformas en este proyecto se trabajará con una aplicación desarrollada para Android e iOS.

Figura 1.1: Logotipo de Flutter

1.2.2. DevOps

DevOps [7] es una metodología ágil para el desarrollo del software. Uno de los principios clave de DevOps es la automatización de las tareas y, además, esta metodología se basa en integrar la parte de desarrollo, que consiste en escribir código o hacer testing, y la parte de operaciones, que consiste en montar el sistema y desplegarlo. Un conjunto de prácticas DevOps son las llamadas CI/CD que significan *continuos integration* o integración continua en español que pertenece a la parte de desarrollo y *continuos delivery* o en español entrega continua que se encuentra en la parte de operaciones.

En este trabajo se utilizarán las herramientas necesarias para llevar a cabo esta metodologia automatizando las tareas que sean posibles del desarrollo. Estás herramientas se introducirán en el capítulo 2.

1.2.3. Testing

Los tests son muy útiles a la hora de probar software ya que con ellos se pueden detectar errores. Hay varios tipos de tests, los test unitarios que se centran en probar pequeñas partes del sistema o incluso métodos, los tests de integración que son los que prueban el sistema con todas o la mayoría de sus funcionalidades, test de sistema para probar las propiedades no funcionales del software como por ejemplo la seguridad o el rendimiento y por último el testing de aceptación que es donde se prueban que esten todas las características requeridas por el usuario final y por el cliente. Otro tipo de test son los tests de regresión, estos test se caracterizan en que se pasan después de haber modificado alguna parte del software, este tipo de test esta muy ligado a la metodología de DevOps ya que con ella este tipo de test se puede decir que se pasa continuamente, como se ha mencionado en el apartado anterior, la parte de integración continua se basa en pasar test continuamente y, sobre todo, después de cada modificación.

Como se ha dicho en el apartado 1.1, además de implementar las metodologías DevOps se hará un estudio de dos enfoques de herramientas de testing automatizado, Scripted y Scriptless.

Scripted - El testing basado en scripts consiste en realizar una serie de testing definidos mediante código, con el fin de guiar a la herramienta de testeo que testear [6], esto es útil si se quiere comprobar que ciertas características del sistema funciona correctamente. Con este enfoque siempre se sigue un mismo camino disminuyendo la precisión de estas herramienta ya que los errores generados en caminos alternativos no se detectan.

Scriptless - En contraposición al testing basado en scripts, se encuentra el testing scriptless en este tipo de testing no se necesita programar los tests que debe hacer la herramienta ya que su funcionamiento se basa en un agente que decide donde pulsar [19].

1.2.4. Despliegue

Una parte fundamental del desarrollo del software es el despliegue. En el contexto de aplicaciones móviles el despliegue es la publicación del sistema software en forma de aplicación en alguna de las tiendas de aplicaciones disponibles. Esta parte suele ser realizada por los desarrolladores de la propia aplicación y a veces se convierte en una tarea

bastante pesada para estos lo que ocasiona que se publiquen pocas versiones y no se hagan constantemente. Esto para la metodología DevOps no debe ser así si no que se ha de publicar versiones continuamente y por eso se propone que se automatice este proceso haciendo que los desarrolladores no deban preocuparse por la publicación de las aplicaciones ya que estas se publicarían automáticamente de forma continuada, son forme el sistema software fuera creciendo.

1.3 Motivación

Actualmente soy desarrollador de Flutter y con el fin de mejorar mis competencias decidí realizar este trabajo. Las metodología DevOps me parece muy útil y siempre he querido llevarla a la práctica. En este proyecto junto el hecho de ganar experiencia en Flutter y en DevOps llevando a la practica la metodología DevOps en Flutter. Y además de hacer esto estudiare la complementariedad de dos herramientas testing, cosa que me beneficiara personalmente ya que ganare experiencia en herramientas de testing, algo que es buscado por las empresas para un aseguramiento de la calidad. En resumen, veo este proyecto como una oportunidad para mejorar mis capacidades tecnológicas y aprender tecnologías novedosas como lo son Flutter y DevOps.

1.4 Objetivos

¿Existe una complementariedad entre herramientas Scriptless y Scripted para la parte de continuos integration de metodologías DevOps?

Para poder estudiar la complementariedad primero se debe tener el concepto muy claro. Según la RAE [17] el termino «completar» significa «Añadir a una magnitud o cantidad las partes que le faltan». Por tanto se puede inducir que dos herramientas de testing complementarias significa que una herramienta consigue solucionar, facilitar o mejorar ciertos aspectos de la otra. Dentro del ámbito del testing se puede decir que dos herramientas son complementarias si, por ejemplo, una de ellas detecta un tipo de errores que la otra no detecta tan fácilmente o que una es mucho menos costosa de implementar frente a la otra. De estos dos ejemplos se pueden extraer las medidas necesarias para concluir que estas dos herramientas son complementarias, y justo esto mismo es lo que se va a abordar en este trabajo.

CAPÍTULO 2
Estado de la cuestión

En este capitulo se analizarán las herramientas necesarias para llevar a cabo este trabajo con el fin de elegir que herramientas se adaptan mejor a las necesidades de este trabajo y a la tecnología que se usará.

2.1 Entorno DevOps

Para llevar a cabo las practicas de DevOps necesitamos herramientas que automaticen las tareas tanto de integración continua como de entrega continua. Actualmente existen muchas herramientas para ejecutar estas tareas automáticamente, a continuación se harán un análisis de las herramientas mas utilizadas.

En primer lugar se encuentra Jenkins [15], esta herramienta esta programada en Java por el desarrollador Jenkins CI y es open source. Otra herramienta es Atlassian Bamboo [4], desarrollada por Atlassian y tiene una licencia privada. Por ultimo se va a analizar la herramienta GitLab CI/CD [12], desarrollada por GitLab. En la tabla 2.1 se puede observar la comparación de las características clave para el desarrollo de este trabajo.

Herramienta	Código abierto	Precio	Integración con repositorio
Jenkins	Si	Gratuito, solo costes de servidor externo	A través de Plugin
Atlassian Bamboo	No	Gratis hasta 10 trabajos	A través de Plugin
GitLab	Si	Gratuito, funciones extra de pago	Por defecto

Tabla 2.1: Comparación herramientas para desarrollar un entorno DevOps

Después de esta comparación se ha optado por utilizar GitLab CI/CD ya que es la mejor opción en todas las características clave y se implementará en el capitulo 7.

2.2 Integración continua

En la parte de integración continua necesitamos herramientas de testing automatizado. En este proyecto se va a estudiar la complementariedad de dos herramientas con dos enfoques diferente al testing: testing scripted y scriptless. Siempre desde un enfoque de tests de sistema e de integración. Como en este proyecto se trabaja con el Framework de desarrollo Flutter, necesitamos utilizar herramientas compatibles o adaptar las ya existentes.

2.2.1. Testing Scripted

Una de las herramientas para el scripted testing disponible para Flutter es Flutter Driver[11] que es la recomendada en la documentación de Flutter a pesar de ello se analizará otra herramienta. Esta herramienta es Appium [3] que aunque de por si no es compatible con Flutter, existe una librería de JavaScript diseñada por el usuario truongsinh de GitHub[18]. En la tabla 2.2 se hace una comparativa de las principales características de estas dos herramientas.

Herramienta	Compatibilidad con Flutter	Facilidad de ejecución	Lenguaje de programación de los scripts
Flutter Driver	Si, por defecto	Solo es necesario ejecutar un comando	Dart
Appium	Si, a través de un paquete externo	Es necesario ejecutar un servidor appium	JavaScript

Tabla 2.2: Comparación herramientas de testing scripted

La herramienta elegida es Flutter Driver debido a que tiene mejores opciones que Appium, los scripts para hacer uso de esta herramienta se describirán en el capitulo 4.

2.2.2. Testing Scriptless

Para el testing scriptless existen diversas herramientas, una de ellas es TESTAR [19], desarrollada en Java y de código abierto. Otra herramienta es Firebase Test Lab [9] que pertenece a la suite de Firebase y está desarrollada por Google. Por ultimo existe otra herramienta denominada Monkey [16]. Las principales características de estas herramientas se analizan en la tabla 2.3.

Aunque esta herramienta no ha sido usada anteriormente para realizar tests de aplicaciones Flutter, es parte de este trabajo el adaptarla a este Framework de desarrollo. Otras herramienta scriptless disponible es que tiene como funcionalidad añadida que se ejecuta la aplicación en múltiples dispositivos aunque esta función no es gratuita.

A pesar de que TESTAR no ha sido usada anteriormente para realizar tests de aplicaciones Flutter, se ha escogido esta herramienta debido a que es parte de este trabajo el adaptarla a este Framework de desarrollo como se verá en el capitulo 5.

Herramienta	Configuración del numero de acciones en cada ciclo	Número de ciclos	Ejecución en múltiples dispositivos
Firebase Test Lab	No	Uno	Si
Monkey	Si	Uno	No
TESTAR	Si	Configurable	No

Tabla 2.3: Comparación herramientas de testing scriptless

2.3 Entrega continua

Para la entrega continua se necesita una herramienta que permita subir aplicaciones de forma automatizada. A continuación se han analizado dos herramientas que llevan a cabo esta función. Fastlane[8] que es de código abierto y esta desarrollada por Google y App Center[2] que es de código privado y esta desarrollada por Miscrosoft. En la tabla 2.4 se puede observar una comparación de las características principales de estas dos herramientas.

Herramienta	Precio	Compatibilidad con Flutter
App Center	30 días gratis luego 40$ al mes.	A través de Plug-in
Fastlane	Gratuito	Ejecutando Fastlane en cada sistema operativo

Tabla 2.4: Comparación herramientas de entrega continua

La herramienta escogida para desarrollar este trabajo es Fastlane ya que es gratuita. En el capitulo 6 se mostrará como se ha utilizado.

CAPÍTULO 3

Diseño del estudio de complementariedad

En este capitulo se describirá la aplicación que se usará en este trabajo, además se hará el diseño del estudio de la complementariedad de los dos enfoques de testing automatizado para ello se definirá el concepto de complementariedad, se especificará que es lo que se va a estudiar exactamente y se hablará de como se realizará.

3.1 Aplicación bajo prueba

La aplicación que se utilizará en este proyecto se llama My Notes. Esta aplicación ha sido desarrollada para una entrevista de trabajo con el fin de demostrar unos conocimientos en Flutter. Se ha decidido utilizar esta aplicación debido a que es una aplicación sencilla sin muchas características para así hacer un mejor estudio. Esta aplicación consiste en una herramienta donde escribir notas. La aplicación cuenta con dos pantallas, una es donde aparece una lista de notas que se corresponde con la figura 3.1 y la otra es para crear o modificar una nota que se corresponde con la figura 3.2. En la pantalla de la lista de notas se pueden buscar notas, cambiar la vista y crear una nota nueva y en la pantalla de editar nota se puede editar el titulo, el contenido de la nota, cambiar el color de la nota, ajustar la prioridad, guardar los cambios de la nota y eliminar la nota.

La arquitectura de la aplicación es una arquitectura MVVM [1] (Model View ViewModel) que cuenta en la parte de Model con una persistencia utilizando el módulo Hive [14] de Flutter y con un único tipo de objeto que es Note. Otra parte es la parte de View donde se encuentra toda la interfaz de la aplicación. Por ultimo existe la parte de ViewModel que es donde se encuentra toda la parte lógica de la interfaz y además actúa de intermediario entre la parte de Model y la parte de View.

[1] MVVM es una arquitectura de software cuyas siglas significan Model View ViewModel. Estas siglas son los tres componentes que forman esta arquitectura de software

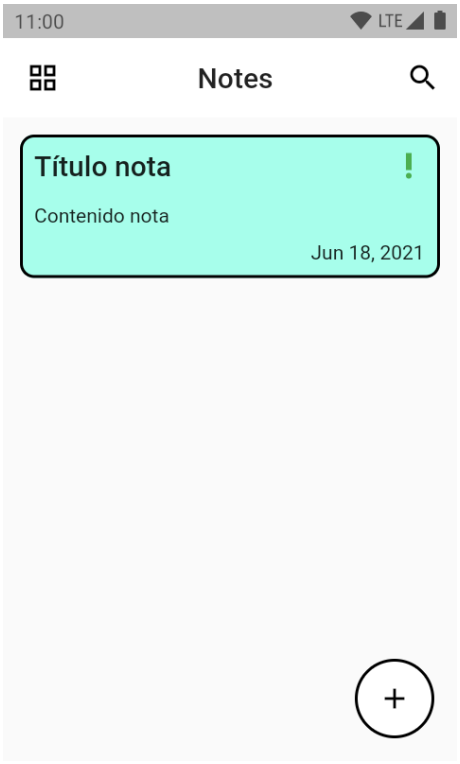


Figura 3.1: Pantalla de la lista de las notas

Figura 3.2: Pantalla de la edición de las notas

Casos de uso de la aplicación

Las características de esta aplicación vienen definidas por casos de uso que se nombraran brevemente a continuación:

1. Crear nota: Se puede crear una nota con los campos vacíos que no aparecerá en la lista a menos que se guarde.

2. Modificar nota: Se puede modificar la nota que tendrá los siguientes atributos: prioridad, color, título y contenido.

3. Borrar nota: Se podrá eliminar cualquier nota, siempre pidiendo confirmación del usuario.

4. Guardar nota: Se puede guardar los cambios realizados en la nota, en el caso de modificación si se quiere salir de la pantalla de editar nota habrá que mostrar un aviso.

5. Buscar nota: Poder buscar una nota tanto por su nombre como por su contenido

6. Cambiar vista de la lista de las notas: Se podrá cambiar la vista de las notas a una vista mas compacta.

3.2 ¿Cómo estudiar la complementariedad?

Una vez definido el sistema que se va a testear se procede a la definición de como realizar el estudio.

3.2.1. Enfoques de testing a estudiar

En primer lugar se especificarán los enfoques del testing que se usaran en este estudio que será sobre lo que se realizará el estudio. En este caso se realizará sobre dos herramientas de testing con dos enfoques diferentes. El primer enfoque será un enfoque Scripted donde se usará la herramienta Flutter Driver como se decidió en el capitulo 2 y un enfoque Scriptless para el que se utilizará la herramienta TESTAR según se especifico en el mismo capítulo.

3.2.2. ¿Cómo medir complementariedad entre los dos enfoques de testing?

Habiendo definido el término complementariedad en el apartado 1.4. Se procede a especificar como llevar a cabo esta complementariedad, para ello se ha decidido comparar las dos herramientas de diferentes formas con el fin de observar en que ámbitos destaca mas una u otra. Para ello se van a realizar tres comparaciones, una relativa al esfuerzo realizado en llevar a cabo las soluciones para cada enfoque, otra es respecto a que partes del código recorren mediante la utilización de diferentes criterios de cobertura y la ultima relativa a los diferentes tipos de errores que pueden encontrar cada una mediante la introducción de errores en el código. Estas tres comparaciones se explicaran mas en profundidad en la próxima sección.

3.2.3. Comparación de los enfoques

Esfuerzo realizado

Una parte importante de cualquier herramienta es el esfuerzo necesario para utilizarla. Por este motivo se va a comparar estos dos enfoques con el esfuerzo realizado para implementar cada una de ellas. Para ello se medirán atributos como el tiempo que ha durado o la cantidad de lo que se ha tenido que implementar o hacer antes y durante el desarrollo de cada herramienta.

Criterios de cobertura

Para saber que partes del sistemas son testeadas se necesita tener uno o varios criterios de cobertura. Los criterios se definen a partir de un modelo de testing. En este trabajo se ha desarrollado un diagrama de flujo como se ve en la figura 3.3.

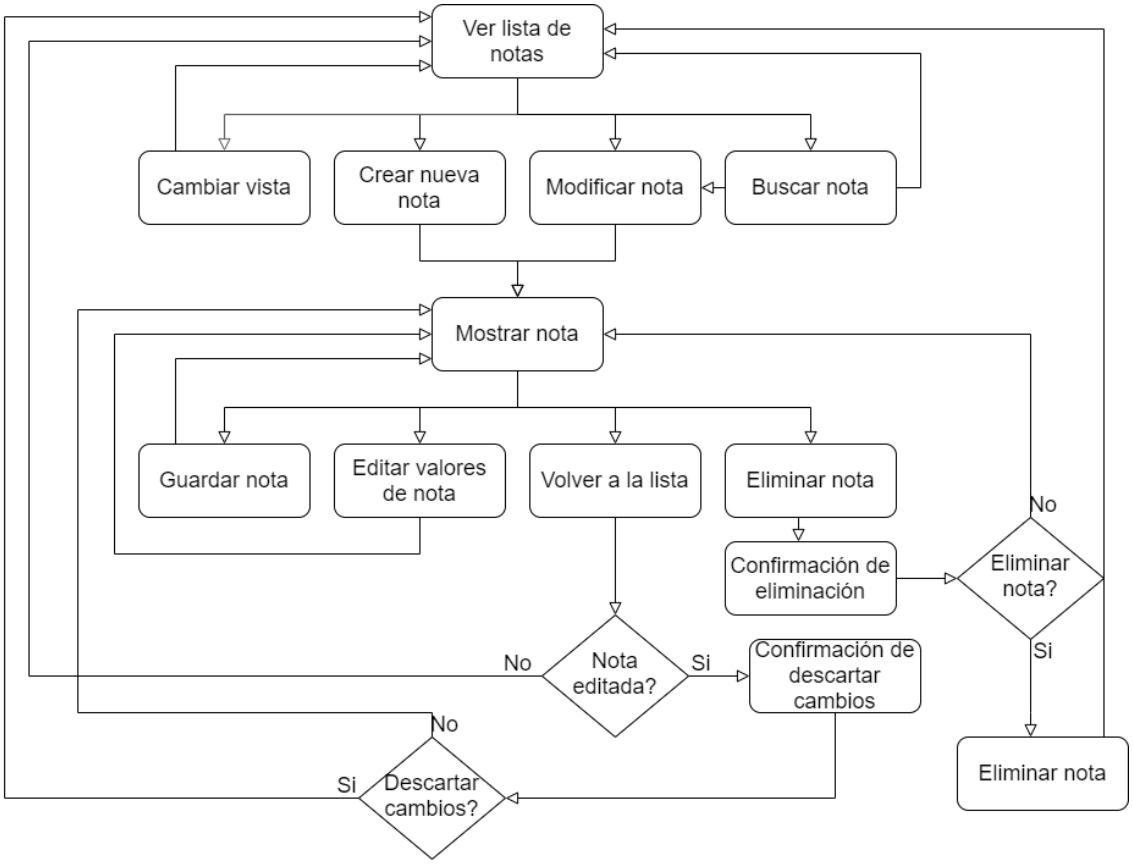


Figura 3.3: Modelo de testing – diagrama de flujo

Se han seleccionado algunos criterios de cobertura que mejor se adaptan al sistema actual [20], estos criterios de cobertura son los siguientes:

1. Cobertura de nodo: Este criterio se centra en medir que nodos[2] se testean, por nodo se entiende como acción o estado. Por ejemplo un nodo del sistema que se puede observar en la figura 3.3 es el de Modificar nota.

2. Cobertura de transiciones: Con este criterio se mide la cantidad de transiciones[3] por las que pasa la ejecución. Por transición se entiende como el camino de un estado a otro. Por ejemplo en la figura 3.3 una de las transiciones es la flecha que conecta el estado de ver lista de notas con cambiar vista.

3. Cobertura de caminos *Prime*: Que un camino sea *prime* quiere decir que va de un nodo a otro sin pasar dos veces por el mismo, a excepción del primer y ultimo nodo que pueden ser el mismo. Dentro del conjunto de caminos *prime* no puede haber un camino que se pueda incluir dentro de otro. Este criterio de cobertura mide todos los caminos *prime* por los que pasa la ejecución de un sistema.

Además de estos criterios de cobertura se analizó el utilizar otros criterios de cobertura pero se desestimo por diversas razones. Los criterios de cobertura que se analizaron en utilizar y las razones por la que se desestimarion son estas:

[2]Nodos son los estados en los que se encuentra un sistema software. Se les suele llamar también nodos porque a partir del diagrama de flujo se puede obtener un grafo.
[3]Una transición es cuando se pasa de un nodo a otro.

1. Cobertura de caminos: Este criterio de cobertura se refiere a la cantidad de caminos posibles que se ejecutan, se descartó esta cobertura debido a que según el diagrama de flujo de la aplicación que se ve en la figura 3.3 existen caminos infinitos y por tanto es imposible obtener esta cobertura.

2. Cobertura de condición múltiple: En este criterio se miden todas las combinaciones posibles de las condiciones que existen en el sistema. Este criterio se descartó porque como se puede ver en el diagrama de la figura 3.3 hay tres condiciones que generan un total de cinco combinaciones posibles esto ocasiona que el criterio de cobertura sea muy similar a utilizar el criterio de cobertura de caminos *Prime* y, por tanto, no aporta valor suficiente para esta comparación.

3. Cobertura de caminos *round-trip*: Que un camino sea *round-trip* quiere decir que empieza y acaba en el mismo estado. Sin pasar por un estado mas de una vez, a excepción del primer y ultimo estado, se diferencia de los caminos *Prime* ya que estos últimos pueden no empezar y acabar en el mismo estado. Este criterio de cobertura mide todos los caminos *round-trip* por los que pasa la ejecución de un sistema. Este criterio se ha descartado debido a su similitud con la cobertura de caminos *Prime* ya que en la lista de caminos *Prime* están contenidos todos los caminos *round-trip*.

Introducción de errores en el código

Con el fin de saber si las herramientas a analizar pueden detectar errores en el código se ha decidido introducir los errores mas comunes en Flutter [5] que se puedan aplicar a la aplicación a testear. Se ha decidido esto ya que sería lo mas imparcial para no favorecer a una u otra herramienta.

Los errores son los siguientes:

1. Llamar setState en el método build. SetState es un método que actualiza la interfaz para cambiarla, mientras que el método build es el que monta la interfaz. Entonces si se intenta actualizar la interfaz mientras se monta la interfaz se genera un error.

2. String mal escrito. Al programar se es muy propenso a escribir alguna letra mal y en los Strings es mas delicado ya que no se detecta en el momento de la compilación del programa. Y si ese String es del que depende la base de datos de la aplicación puede llegar a funcionar mal.

3. Un InputDecorator no puede tener un ancho ilimitado. Un InPutDecorator es un widget que se encuentra dentro del widget del campo de texto y es el encargado de proporcionar a este la interfaz. Cuando se introduce en un widget que no tiene un ancho determinado como puede ser una fila, este genera un error.

4. RenderFlex Overflowed. Este error ocurre básicamente cuando hay un widget que ocupa mas del espacio disponible que dispone.

CAPÍTULO 4

Implementación del enfoque scripted

En este capitulo tratará la implementación de los scripts que ejecutaran Flutter Driver para una posterior integración con un entorno DevOps de la que se hablará en el capitulo 7.

4.1 La herramienta Flutter Driver

La herramienta Flutter Driver realiza tests de integración de aplicaciones desarrolladas en Flutter. Esta herramienta lo que hace es que mediante unos scripts se va guiando la ejecución de la aplicación en un dispositivo móvil o emulador.

Para realizar estos test sobre la aplicación primero es necesario crear una carpeta en la raíz del proyecto llamada *test_driver* esta carpeta contendrá dos archivos, *app.dart* y *app_test.dart*

4.1.1. Archivo app.dart

Este archivo será donde se encuentra el método main [1] dentro de este método se inicializarán los tests y posteriormente se llamará al main de la aplicación principal. Esto se hace únicamente para activar los tests y que se ejecuten. Como se puede ver en la figura 4.1 en este archivo se habilita la extensión para los tests invocando al método *enableFlutterDriverExtension* y seguidamente se ejecutará el main de la aplicación principal.

[1] El método main es el primer método que se ejecuta al iniciar una aplicación en Flutter

app.dart

```dart
import 'package:flutter_driver/driver_extension.dart';
import 'package:my_notes/main.dart' as app;

void main() {
  enableFlutterDriverExtension();

  app.main();
}
```

Figura 4.1: Contenido del archivo app.dart

4.1.2. Archivo app_test.dart

En este archivo será donde se encuentran todos los test scripts con los que se testeara la aplicación. La estructura de este archivo es la siguiente: en primer lugar hay un único método que es el main, dentro de este se encuentran los diferentes grupos que a su vez están compuestos por el método *setUpAll* que es el que se ejecuta al inicio de los test de ese grupo y será el encargado de conectar con FlutterDriver, el método *tearDownAll* que es el que se ejecuta al final de los tests, estos dos métodos se pueden ver en la figura 4.2. Además, de estos dos métodos en cada grupo se incluyen un conjunto de tests que serán los que se van ejecutando.

```dart
group('CRUD note', () {
  setUpAll(() async {
    driver = await FlutterDriver.connect();
  });

  tearDownAll(() async {
    driver.close();
  });
```

Figura 4.2: Métodos *setUpAll* y *tearDownAll*

Dentro de estos test es donde se en encuentra el código donde se le dice a la aplicación que acciones realizar. Para ejecutar una acción en primer lugar hay que crear un finder del widget que será el encargado de encontrar al widget. Esto se puede hacer a partir de una key[2] o de un texto que esté contenido en el widget. Una vez creado el finder se le pasa como argumento a los métodos que ejecuta driver para realizar acciones como se ve en la figura 4.3 donde *buttonAddNote* es un finder que se le pasa a las acciones como

[2]Una key es un nombre único para cada widget de Flutter que lo identifica.

argumento. Las acciones que se utilizaran son tap, escribir, obtener texto y además se utilizaran otras acciones en las que se incluyen timeouts como la de esperar a que este un widget determinado en pantalla y esperar a que un widget determinado deje de estar en pantalla.

```
test('open new note', () async {
  await driver.waitFor(buttonAddNote, timeout: const Duration(seconds: 4));

  await driver.tap(buttonAddNote);

  expect(await driver.getText(textTittle), "");
  expect(await driver.getText(textNote), "");
});
```

Figura 4.3: Ejemplo de test

Otra parte importante de dentro de los métodos de tests son los métodos expect donde se comprueba que dos valores son iguales, es necesario para comprobar que los outputs de la aplicación son los que se requieren que sean.

4.2 Implementación de los *scripts*

Para el desarrollo de estos test se ha basado en los casos de uso de la aplicación que se encuentran en la sección 3.1. Se ha creado un test para testear cada caso de uso a excepción del caso de uso de cambiar vista ya que por limitaciones de Flutter Driver no se podía comprobar que se cambiaba correctamente la vista. El *script* que se ha implementado cuenta con un total de 228 lineas y se ha dedicado un total de unas cinco horas. Los *scripts* se pueden ver en el apéndice A.

En la figura 4.4 se puede ver uno de los *tests* que se han implementado y que consiste en comprobar que se abre la pantalla de nueva nota, para ello se ejecuta un *wait for* que se mantiene a la espera del *widget* de añadir una nota, como se puede observar la variable *buttonAddNote* es un *finder* del botón de añadir nueva nota, y el atributo *timeout* es un objeto *Duration* para darle un *timeout* de 4 segundos. Una vez se detecta este *widget* se envía un *tap* y después mediante el *expect* se comprueba que los campos de texto de la nota estén vacíos.

```
test('open new note', () async {
  await driver.waitFor(buttonAddNote, timeout: const Duration(seconds: 4));

  await driver.tap(buttonAddNote);

  expect(await driver.getText(textTittle), "");
  expect(await driver.getText(textNote), "");
});
```

Figura 4.4: Ejemplo de test de Flutter Driver

CAPÍTULO 5

Implementación del enfoque scriptless

En este capitulo se tratará la implementación del testing scriptless. Como se ha definido en el apartado 1.2.3 estos tipos de test se basan en que no es necesario la implementación de scripts ya que las acciones a ejecutar son decididas por unos agentes. La herramienta que se va a utilizar es TESTAR. Actualmente esta herramienta no ofrece soporte con Flutter, por este motivo es necesario adaptar TESTAR a Flutter. En este capitulo se explicará todo el proceso que se ha llevado a cabo para realizar esta adaptación.

5.1 La herramienta TESTAR

TESTAR es una herramienta open source para el testing automático de aplicaciones a nivel de interfaz de usuario. Esta herramienta se distingue de las herramientas habituales debido a que no se necesita programar unos scripts que decidan que camino seguir y por este motivo se dice que TESTAR es una herramienta de testing scriptless.

El funcionamiento de TESTAR se basa en un bucle como se ve en la figura 5.1. En primer lugar se encuentra la parte de *Technical APIs/Frameworks* que es donde se inicia el SUT[1], a continuación se encuentra las partes de *Plugins for SUT interaction* y *Obtain GUI State* que es donde TESTAR obtiene el árbol de widgets de la aplicación a través de los plugins necesarios. En la siguientes partes es donde se obtienen las acciones y se ejecutan las que TESTAR decida en base a los ajustes del usuario. Después se encuentra la parte *Oracle Evaluate SUT* que es donde entra a trabajar la parte del oráculo.

Los oráculos son los encargados de detectar errores en la ejecución del SUT, ya sea mediante la detección de un texto de error o detectando que la aplicación se ha cerrado inesperadamente, entre otros. Después de la evaluación del oráculo este determina si ha habido fallo o no en caso de que hubiera fallo se guarda la secuencia seguida y se pararía la ejecución de la secuencia. En caso contrario se seguiría la ejecución de la secuencia, si la secuencia no contiene ninguna condición de stop, que puede ser por ejemplo que se han realizado un número determinado de acciones, se continua la ejecución volviendo a la obtención de el árbol de Widget ya que puede ser que al ejecutar una acción se haya cambiado de ventana. En el caso de que si que hubiera una condición de stop se generarían los informes y posteriormente se pararía el SUT y en el caso de que no se hubieran

[1]System Under Testing, en español se puede referir como sistema que se va a testear

ejecutado todas las secuencias requeridas por el usuario se volvería a empezar otra vez desde el inicio, en caso contrario ya habría acabado la ejecución de TESTAR.

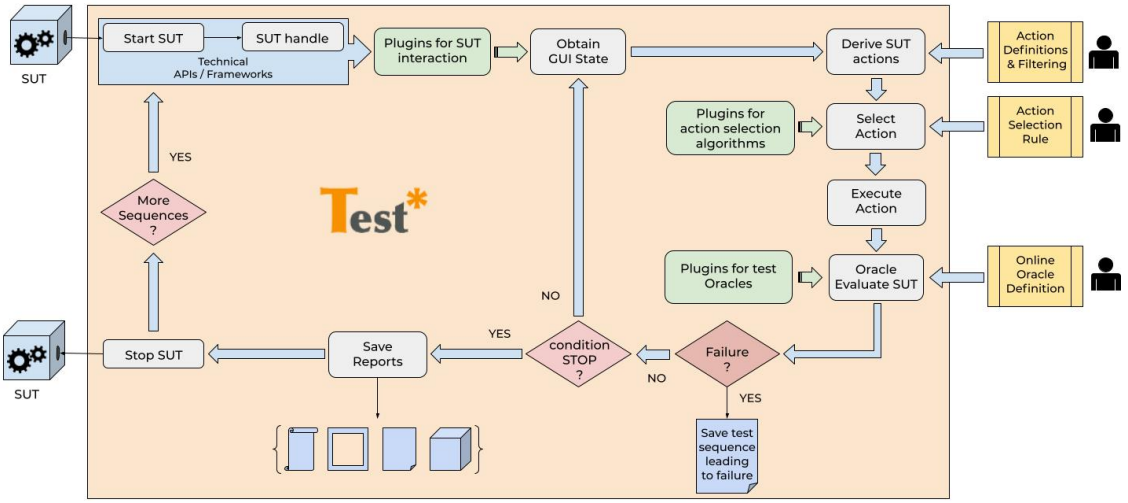


Figura 5.1: Flujo de TESTAR

En la figura 5.2 se puede ver un resumen de la ejecución de TESTAR donde se divide en tres partes que son obtener estado mediante un análisis de la interfaz, detectar acciones disponibles y seleccionar y ejecutar una acción.

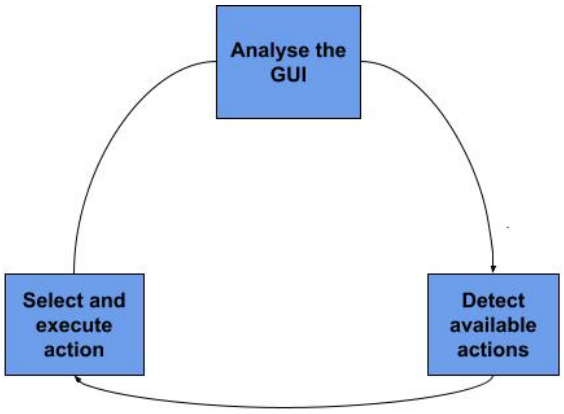


Figura 5.2: Simplificación del bucle de TESTAR

Toda la parte de comunicación con el SUT se hace a través de unos protocolos que es donde se definen los plugins que se van a usar para este motivo. El protocolo del que se partirá para la adaptación a Flutter es un protocolo para sistemas Android que ejecuta un servidor Appium para obtener el árbol de widgets, para realizar las acciones y para que los oráculos comprueben errores.

5.2 Obtener el estado con TESTAR de aplicaciones Flutter

Para utilizar TESTAR en aplicaciones desarrolladas en Flutter se ha decidido utilizar Android y no iOS debido a que todavia no está implementado para esta herramienta. Lo primero que se ha realizado es ejecutar TESTAR con el protocolo *android_generic* en modo *generate* para comprobar si funcionaba correctamente pero no fue así y se decidió ejecutar

el modo espía. De esta forma se encontró que había un problema con la resolución del emulador que se solucionó escalando la ventana del emulador.

El siguiente problema que se encontró fue que no se detectaban todos los elementos en los que se podía hacer clic como se ve en la figura 5.3 los colores de la nota son elementos que se les puede hacer clic sin embargo, no son detectados por TESTAR. Para hacer que TESTAR detectara los botones correctamente se decidió modificar el protocolo de android, mas concretamente el archivo *Protocol_android_generic.java* que es donde se comprueba si a un *widget* se le puede hacer clic o si se puede escribir en el. Se modificó el método *isClickable* añadiendo una nueva condición para comprobar si el elemento tenia la propiedad *isClickable* con valor verdadero como se puede observar en la figura 5.5 donde lo subrayado en color rojo es la parte que se ha añadido.

Una vez hecho el cambio se comprueba si se detectan todos los *widgets* y esta vez si que se detectan como se ve en la figura 5.4. A continuación comprobamos que los campos de texto se detecten como tal y funcionaba correctamente.

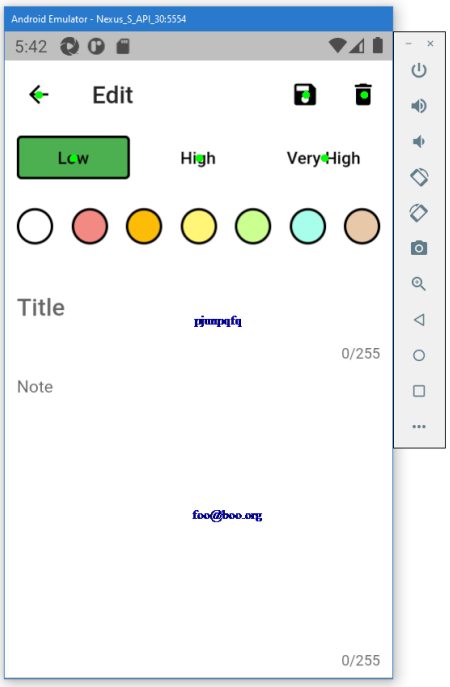


Figura 5.3: Emulador de Android con el modo espía de TESTAR sin ninguna adaptación a Flutter

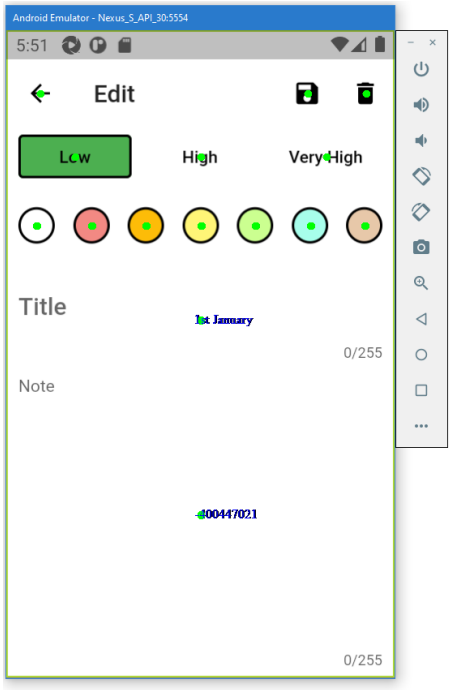


Figura 5.4: Emulador de Android con el modo espía de TESTAR y con la adaptación en la detección de *widgets*

```
@Override
protected boolean isClickable(Widget w) {
    return (w.get(AndroidTags.AndroidClassName, "").equals("android.widget.ImageButton")
    || w.get(AndroidTags.AndroidClassName, "").equals("android.widget.Button") || w.get(AndroidTags.AndroidClickable,false));
}
```

Figura 5.5: Método *isClickable* del protocolo de Android

5.3 Derivar, seleccionar y ejecutar acciones

Una vez hecho que TESTAR detecte todos los *widgets* de una aplicación Flutter se ejecuta el modo *generate* para generar casos de prueba y comprobar que funcione. A pesar de

detectar correctamente los *widgets* las acciones no se ejecutaban correctamente a través de Appium. Inspeccionando el output de TESTAR se encuentra que la propiedad *resource-id* en Flutter no existe y para ejecutar las acciones de introducir texto y de pulsar era necesario, se opta por buscar alternativas al *resource-id*, se encuentran algunas formas de realizar estas acciones como a travesa del atributo *content-desc* pero ocurre lo mismo que en *resource-id* y tampoco existe en Flutter, otra forma que se encuentra es a través del *xPath* pero no se puede acceder a el fácilmente.

Debido a la problemática de utilizar Appium para realizar acciones se decide utilizar Windows API para que ejecute las acciones haciendo que en lugar de mandar un *tap* a traves de Appium se hace clic con el ratón de Windows, para ello se modifica otra vez el archivo *Protocol_android_generic.java* y se cambian las acciones que utilizaban Appium por unas que utilicen Windows API. Este cambio si que ha funcionado correctamente, la problemática de Windows API son las limitaciones que tiene como que solo funciona en entornos Windows pero a pesar de esto se ha decidido utilizarla ya que no afecta al trabajo esta limitación.

5.4 Definir los oráculos

Para que TESTAR sepa identificar cuando hay algún error en la ejecución existen los oráculos. Estos se encargan de buscar en el sistema algo que denote un error del mismo por ejemplo la palabra *error* o *exception* como es lo que se hace en Android.

A continuación se comprueba si los oráculos definidos para Android funcionan en Flutter. Se llega a la conclusión de que no funcionan correctamente y se decide averiguar el porque. Se encuentra que para detectar los posibles errores TESTAR utiliza el atributo *AndroidText* que no existe en Flutter. Para encontrar un atributo que remplace al atributo *AndroidText* se ejecuta TESTAR es modo espía y se comprueba que el atributo *AndroidAccessibilityId* contiene el texto de los *widgets* de Flutter como se ve en la figura 5.6. Una vez obtenido el atributo necesario se cambia en el código de TESTAR por este nuevo.

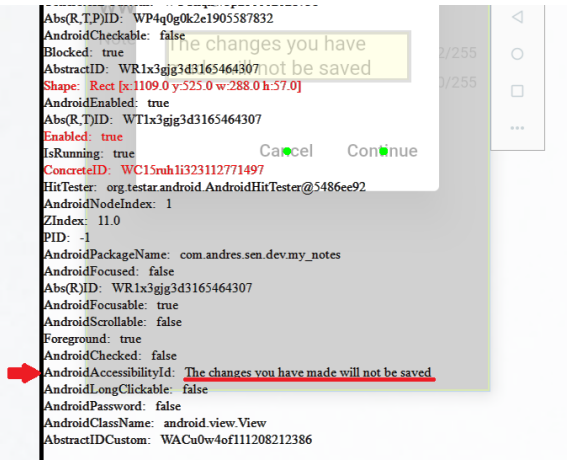


Figura 5.6: Captura del modo espía de TESTAR con un texto seleccionado

Para añadir precisión al oráculo se ha añadido otra funcionalidad que permite al oráculo detectar si la aplicación se ha cerrado espontáneamente. Para ello se utiliza el atributo AndroidPackageName de Appium el cual dice el nombre del paquete de la aplicación. Este atributo se comprueba para cada widget y se hace que si en el árbol de widgets se detecta un widget con el nombre del paquete similar al del sistema bajo testeo se tiene

la certeza de que no se ha parado la aplicación. En caso contrario no se detectaría ningún widget con el nombre del paquete que corresponde y se marcaría como que se ha detectado un error.

5.5 Informes generados

Una vez se ha completado la ejecución de TESTAR se obtienen los resultados mediante los informes que han sido generados. En estos informes están en HTML y abriéndolos con el navegador se muestra la ejecución de TESTAR en cada secuencia, incluyendo capturas de pantalla y donde se hace clic. Por ejemplo en la figura 5.7 se puede ver como la ejecución selecciona el botón de búsqueda y en la figura 5.8 el estado que sigue a esta acción. En el apéndice B se puede ver un extracto de informe.

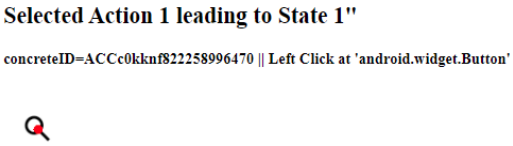


Figura 5.7: Captura de una acción

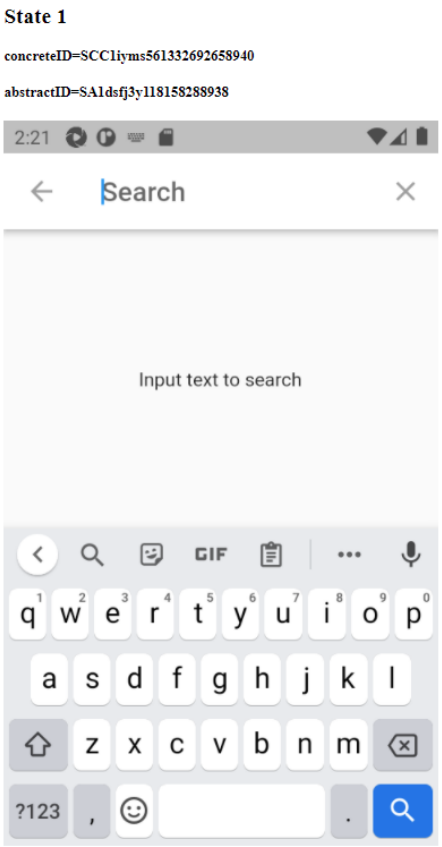


Figura 5.8: Captura de un estado

CAPÍTULO 6

Implementación de publicación automatizada

6.1 La herramienta Fastlane

Para la publicación automatizada de la aplicación se usará Fastlane. Esta herramienta se basa en lo que se denomina *lane* o carril en Español, los carriles se programan en el fichero Fastfile, del que se hablará a continuación, y cada uno es el encargado de realizar una serie de funciones. Para ejecutar un carril basta con lanzar el comando «fastlane lane_name» donde *lane_name* es el nombre del carril a ejecutar.

Para utilizar esta herramienta hay que ejecutar el comando «fastlane init» para que Fastlane cree los archivos necesarios que son cuatro:

1. Appfile que será el que contenga las diferentes configuraciones de Fastlane
2. Fastfile será donde se encuentre todo el código que se ejecutará
3. Readme.md un archivo de información que no será necesario editarlo
4. report.xml un archivo de *logs* que no se utilizará en este proyecto

En este proyecto la inicialización es distinta a como se haría un programación nativa debido a que los frameworks multiplataforma tienen una carpeta para cada sistema operativo, en este caso, Flutter, tiene la carpeta android para Android y la carpeta ios para iOS. Al tenerlo de esta forma es necesario inicializar fastlane en cada carpeta y no en la de raíz del proyecto como ocurre al programar de forma nativa. Para ello debemos ejecutar el comando «fastlane init» estando dentro de la carpeta de cada sistema operativo.

6.2 Configuración de Fastlane

Debido a que no se posee un ordenador con Mac OS no es posible montar la aplicación ni desplegarla para iOS por tanto, se centrará en la publicación automática de Android en Google Play.

Para comenzar con la configuración de Fastlane se necesitará una clave privada que permita acceder a la cuenta de desarrollador de Google Play para publicar la aplicación. Para obtener esa clave privada primero es necesario crear una cuenta de servicio en Google Cloud introduciendo un nombre cualquiera y la función de Usuario de cuentas de servicio como se puede ver en las figuras 6.1 y 6.2. Una vez creada la cuenta de servicio se procede a generar una clave privada en formato JSON como se ve en la figura 6.3.

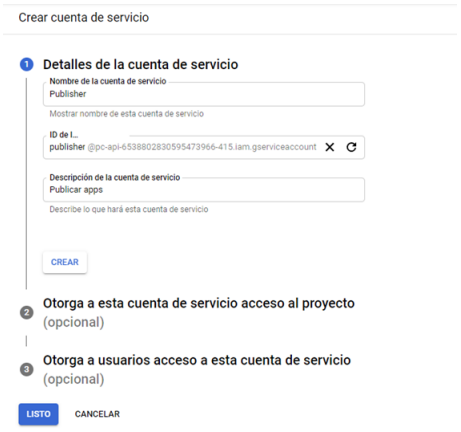


Figura 6.1: Creación cuenta de servicio paso 1

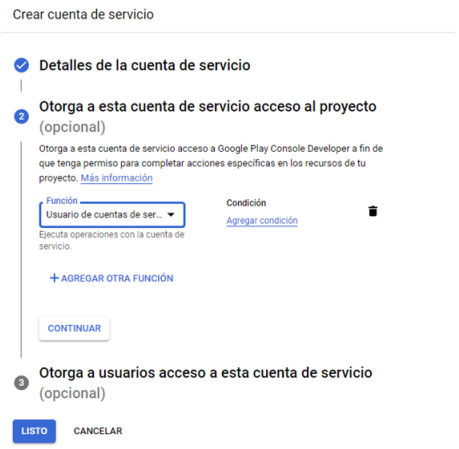


Figura 6.2: Creación cuenta de servicio paso 2

Figura 6.3: Captura de un estado

Cuando ya tenemos el archivo JSON con la clave privada inicializamos Fastlane. En este proyecto, como se ha dicho antes, se va a desarrollar únicamente el despliegue automatizado en Android, por ello solo se inicializará en Android, al ejecutar el comando «fastlane init» se detecta automáticamente que el proyecto es Android y se ha de introducir algunos datos como el nombre del paquete de Android que en este caso es com.andres.sen.dev.my_notes y el *path* donde se encuentra el archivo JSON con la clave privada obtenida con anterioridad.

Una vez inicializado se pueden ver todos los archivos que Fastlane ha creado como se aprecia en la figura 6.4. A continuación se procede a la configuración de Fastlane, para ello debemos editar el archivo Fastlane que se ha generado. Se añade un nuevo carril al que se le llamará *alpha* y que será el encargado de publicar la aplicación en Google Play de forma interna y de estado borrador ya que no es posible hacer una publicación automatizada en modo normal ya que no está aprobada por Google Play.

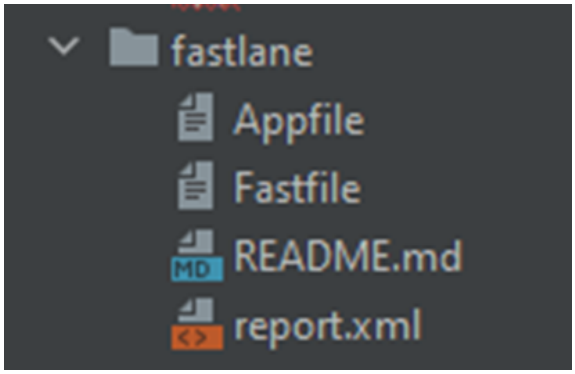


Figura 6.4: Archivos creado con la inicialización de Fastlane

En la figura 6.5 se puede observar el contenido del archivo Fastfile donde se encuentra la configuración de Fastlane que contiene el carril *alpha* y una descripción, en la implementación del carril se puede observar como en primer lugar monta la aplicación Flutter en formato Android App Bundle[1] ejecutando los comandos necesarios en la carpeta raíz y para eso es necesario cambiar el directorio de ejecución por lo que se ha dicho anteriormente de que Fastlane se encuentra en la carpeta del sistema operativo a montar y no en la carpeta raíz. Después de la parte de montar el *appBundle* de Android se procede a ejecutar la instrucción *upload_to_play_store* pasándole los argumentos: *track* con el valor *internal* para indicar que sera una publicación para uso interno sin que esté disponible de forma abierta [2], el argumento *release_status* para indicar que se hará en modo borrador con el valor *draft* y por último el argumento aab con la ruta del *bundle* construido.

[1]Android App Bundle o aab es un formato de aplicaciones Android alternativo a APK, este formato se diferencia de APK en que incluye todas las versiones para las diferentes arquitecturas del procesador de Android haciendo que la instalación en el dispositivo ocupe menos. Google recomienda el uso de este formato en lugar de APK por estos motivos [1].

[2]Además de el valor *internal* también existen otros que son: *production* que se corresponde con la versión disponible de forma abierta, beta que es igual que *production* pero indicando al descargar que puede ser una versión no estable y *alpha* que es una versión que esta disponible para un determinado grupo de personas.

```
default_platform(:android)

platform :android do

  desc "Deploy a new alpha build to Google Play"
  lane :alpha do
    build_number = number_of_commits()
    Dir.chdir "../.." do
      sh("flutter", "packages", "get")
      sh("flutter", "clean")
      sh("flutter", "build", "appbundle", "--build-number=#{build_number}")
    end
    upload_to_play_store(
        track: 'internal',
        release_status:'draft',
        aab: '../build/app/outputs/bundle/release/app-release.aab'
    )
  end

end
```

Figura 6.5: Contenido de archivo Fastfile

6.3 Ejecución de Fastlane

Una vez se ha configurado Fastlane correctamente se procede a ejecutarlo para comprobar su funcionamiento. Para ello se ejecuta el comando «fastlane android alpha». Cuando acaba la ejecución se puede observar como se ha completado la subida a Google Play como se ve en la figura 6.6.

Últimas versiones

Versión	Última versión	Canal	Estado de la versión
1.0.4	52	Prueba interna	Borrador

Figura 6.6: Versión en Google Play subida con Fastlane

CAPÍTULO 7

Implementación del entorno DevOps

En este capítulo se integrará lo desarrollado en los capítulos 4, 5 y 6 con el fin de implementar un entorno DevOps que ejecute automáticamente todas estas herramientas.

7.1 La herramienta GitLab CI/CD

Como se ha comentado anteriormente la herramienta que se usara será GitLab CI/CD, esta herramienta lo que hace es cada vez que se detectan cambios en alguna rama se ejecuta un *script*. Este está programado en el archivo .gitlab-ci.yml que se debe crear en la carpeta raíz del proyecto. Este *Script* se basa en una serie de *stages* o etapas en español que se ejecutaran en orden y en el caso de que alguna falle no se ejecutarán las siguientes. Cada vez que se detectan cambios se genera una pipeline donde se ejecutarán todas las etapas que a su vez se convierten en *jobs* o trabajos en español que serán ejecutados por un *runner*.

```
flutter_build_android:
  stage: build
  before_script:
    - flutter packages get
    - flutter clean
  script:
    - flutter build apk
  artifacts:
    paths:
      - build/app/outputs/flutter-apk/app-release.apk
  tags:
    - flutter
```

Figura 7.1: Ejemplo de etapa

En la figura 7.1 se puede ver una etapa que es la de montar el archivo APK de la aplicación. Cada etapa se compone de un nombre que en la figura 7.1 el nombre es *build*, después se encuentra la parte de *before_script* donde se ejecutan todos los comandos antes de ejecutar el *script* del etapa. La siguiente parte como se puede ver en la figura 7.1

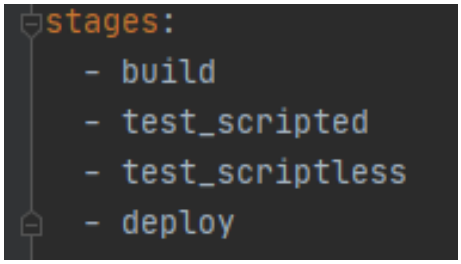


Figura 7.2: Orden de etapas

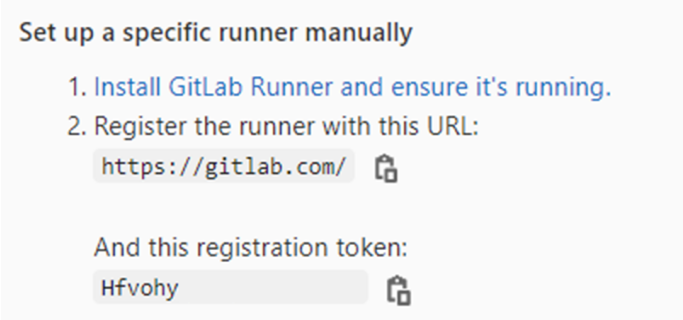


Figura 7.3: Url y token proporcionados por GitLab

es *script* que es donde se indican los comandos que se ejecutan en la etapa, después se encuentra la parte de *artifacts* que pueden ser archivos o directorios que se podrán descargar desde GitLab una vez terminada la ejecución de los *scripts*, además pueden ser usados por otras etapas. Por ultimo se encuentran las *tags* que son etiquetas del *stage*, son necesarias para saber en que tipo de *runner* ejecutarlas.

En la figura se puede ver el orden de las etapas que aparece al inicio de la figura 7.2.

Otra elemento de GitLab CI/CD son los *runners*, estos son los encargados de ejecutar los *scripts* según las etiquetas puesto que cada etapa será asignada a un *runner* que tenga las mismas etiquetas, por ejemplo un *runner* con las etiquetas flutter y TESTAR podrá ejecutar etapas que tengan las etiquetas flutter, TESTAR o ambas. Los *runners* pueden ser dockers, instancias compartidas u ordenadores personales como es el caso de este trabajo.

7.2 Inicialización del *runner*

Como se ha dicho anteriormente en este trabajo se utilizará un ordenador personal de *runner*, este ordenador cuenta con el sistema operativo Windows 10 y para inicializar el *runner* hay que decargarse el programa desde GitLab e instalarlo. Una vez instalado se procede a registrar el *runner* en el proyecto de GitLab, para ello necesitaremos la URL y un token que nos proporciona GitLab como se puede ver en la figura 7.3.

A continuación, se ejecuta el comando register para registrar el *runner* en el proyecto, al ejecutarlo se introducen la URL, el token, las etiquetas para que determinar que stages puede ejecutar, en este caso serán flutter y TESTAR y también se introduce el modo en el que se inicializará el script que será Shell ya que es se ejecuta en la ventana de comandos. En la figura 7.4 se puede ver todo este proceso en la ventana de comandos. Una vez registrado el *runner* se comprobará que aparece en GitLab como se ve en la figura 7.5.

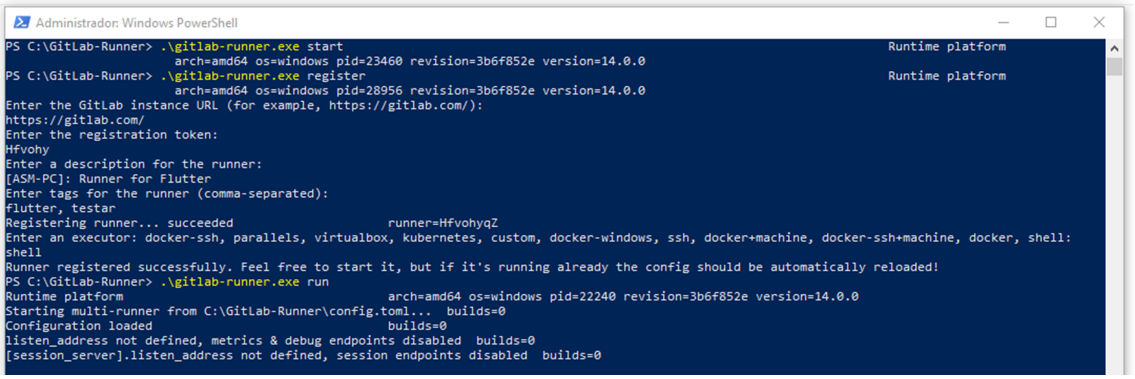


Figura 7.4: Ventana de comandos con la inicialización del *runner*

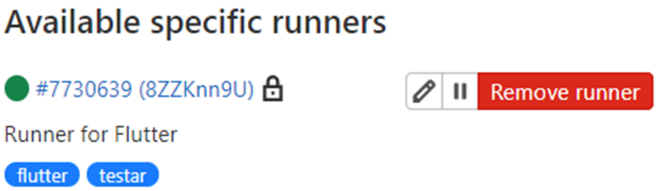


Figura 7.5: Runners disponibles para el proyecto

7.3 Creación del archivo .gitlab-ci.yml

En esta sección se creara el archivo donde se implementará el *script* que será el encargado de llamar a las herramientas de integración continua y de entrega continua.

El archivo que se creará se tendrá el nombre de .gitlab-ci.yml y será creado en la carpeta raíz del repositorio. Este archivo contendrá las diferentes etapas que serán un total de cuatro, la primera se llamara *build* y será la encargada de generar el archivo APK de la aplicación, la siguiente se dominará *test_scripted* y se ocupará de ejecutar los *tests* de Flutter Driver, a continuación se encuentra la etapa llamada *test_scriptless* que ejecutará TESTAR y por ultimo se encuentra la etapa llamada *deploy* que se encargará de ejecutar Fastlane para subir la aplicación a Google Play. En el apéndice C se encuentra el código del archivo.

7.4 Implementación de las etapas

A continuación se implementaran los comandos que se ejecutaran en las diferentes etapas ya comentadas previamente.

7.4.1. Etapa *build*

En esta etapa lo que se hará es montar la aplicación Android en un archivo APK que será subido como artifact para la descarga una vez se haya ejecutado esta etapa. Además este archivo también será de utilidad para otras etapas. En la figura 7.6 se puede ver esta etapa. Para montar una aplicación en Flutter es suficiente con ejecutar el comando

«flutter build apk» pero para asegurarnos de que se monta correctamente son errores se ejecutaran dos comandos previos a este en la parte de *before_script* estos dos comandos son «flutter packages get» que hace que se descarguen todos los módulos añadidos a la aplicación Flutter y «flutter clean» que elimina archivos innecesarios.

```
flutter_build_android:
  stage: build
  before_script:
    - flutter packages get
    - flutter clean
  script:
     - flutter build apk
  artifacts:
    paths:
      - build/app/outputs/flutter-apk/app-release.apk
  tags:
    - flutter
```

Figura 7.6: Etapa *build*

7.4.2. Etapa *test_scripted*

Esta etapa será la encargada de ejecutar los test de Flutter Driver para ello solo es necesario ejecutar el comando «flutter drive –target=test_driver/app.dart» donde el argumento *target* es el documento donde se encuentra el método *main*. Se puede ver el código de la etapa en la figura 7.7.

```
flutter_test_scripted:
  stage: test_scripted
  script:
    - echo running tests
    - flutter drive --target=test_driver/app.dart
  tags:
    - flutter
```

Figura 7.7: Etapa *test_scripted*

7.4.3. Etapa *test_scriptless*

En esta etapa será donde se ejecutará TESTAR, para ello se ha creado un archivo batch llamado run_testar.bat que se encuentra en el apendice D. Este archivo lo que hará es mover la aplicación montada en la etapa *build* a la carpeta de TESTAR. Depués se ejecutara el siguiente comando «testar sse=android_generic ShowVisualSettingsDialogOnStartup=false Sequences=2 SequenceLength=5 Mode=Generate» que será el encargado de arrancar TESTAR, el significado de los argumentos es el siguiente: *sse* es el protocolo a utilizar, *ShowVisualSettingsDialogOnStartup* es para indicar si se quiere o no que se muestre la interfaz gráfica, *Sequences* para indicar el número de secuencias que ejecutará TESTAR, *SequenceLength* es el número de acciones que se ejecutaran en cada secuencia y *Mode* es el modo de ejecución.

```yaml
flutter_test_scriptless:
  stage: test_scriptless
  script:
    - echo running tests
    - scripts/run_testar.bat
  artifacts:
    paths:
      - test_testar/output/
  tags:
    - testar
```

Figura 7.8: Etapa *test_scriptless*

```yaml
flutter_deploy_android:
  stage: deploy
  before_script:
    - cd android
  script:
    - fastlane android alpha
  tags:
    - flutter
```

Figura 7.9: Etapa *deploy*

Una vez ejecutado lo que hace el archivo batch es mover el output a la carpeta donde se ejecuta la etapa de GitLab CI/CD para que posteriormente se indique esta carpeta como artifact para poderse descargar desde GitLab. El código de esta etapa se puede ver en la figura 7.8.

7.4.4. Etapa *deploy*

Para la ultima etapa se encuentra el despliegue, en esta etapa lo que se hace es en primer lugar cambiar el directorio de la ejecución a Android y posteriormente ejecutar el comando «fastlane android alpha» para ejecutar el carril *alpha* de Fastlane.

7.5 Ejecución de GitLab CI/CD

Una vez implementadas todas las etapas se procede a la ejecución de todo el entorno DevOps. Para ello se hace algún cambio y se sube al repositorio de GitLab. Una vez subido se puede observar en la figura 7.10 como están las etapas en cola. Se ejecutan en orden y cada etapa necesita del runner previamente definido.

Durante la ejecución se pueden ver el output de la consola como se ve en la figura 7.11 que se muestra la consola de la etapa de testing scripted.

Una vez acabado correctamente se muestran todas las etapas completas correctamente como se ve en la figura 7.12. En el caso de que una etapa falle se muestra como en la figura 7.13 y como se puede apreciar las etapas siguientes no se ejecutarían. Además en

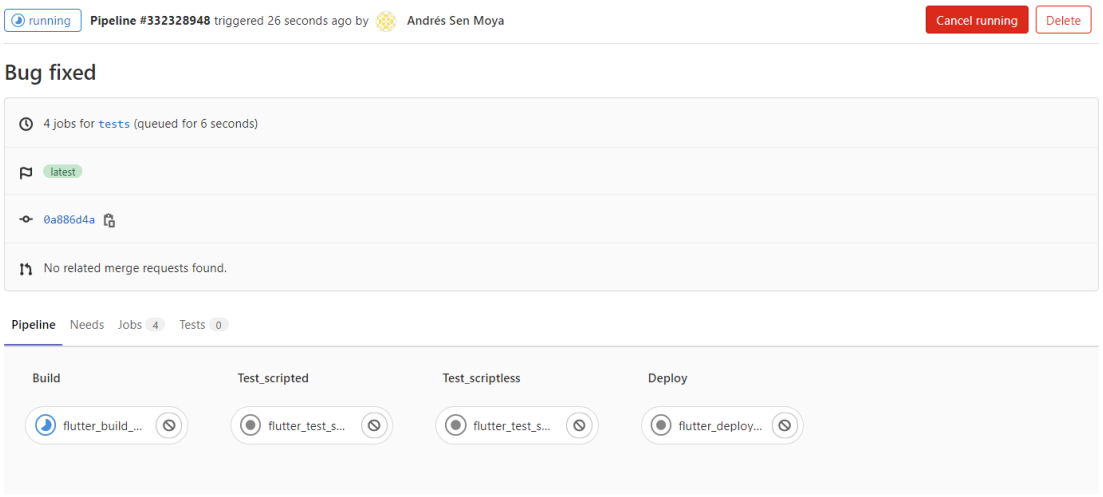


Figura 7.10: Etapas en cola

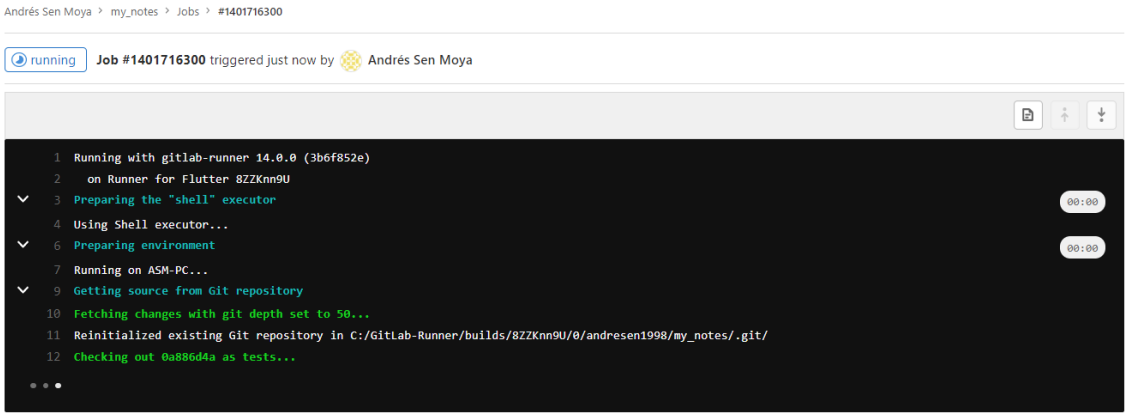


Figura 7.11: Consola de la ejecución de la etapa de testing scripted

el caso de producirse un fallo se envía automáticamente un correo electrónico como se ve en la figura 7.14 indicando que etapa ha fallado.

Una vez terminado todas las etapas se pueden descargar los artifacts como se ve en la figura 7.15. Donde se muestran el artifac de la etapa test scriptless que se corresponde con los informes generados por TESTAR y la etapa de build que se corresponde con el APK de la aplicación.

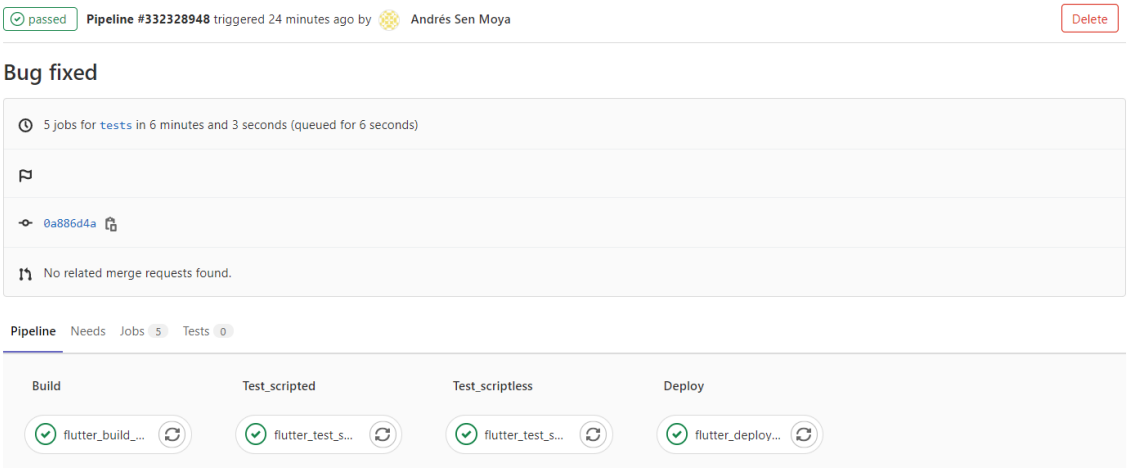


Figura 7.12: Etapas completadas correctamente

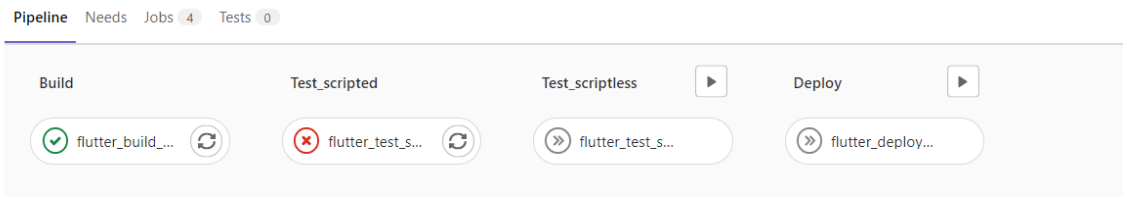


Figura 7.13: Etapas con fallo

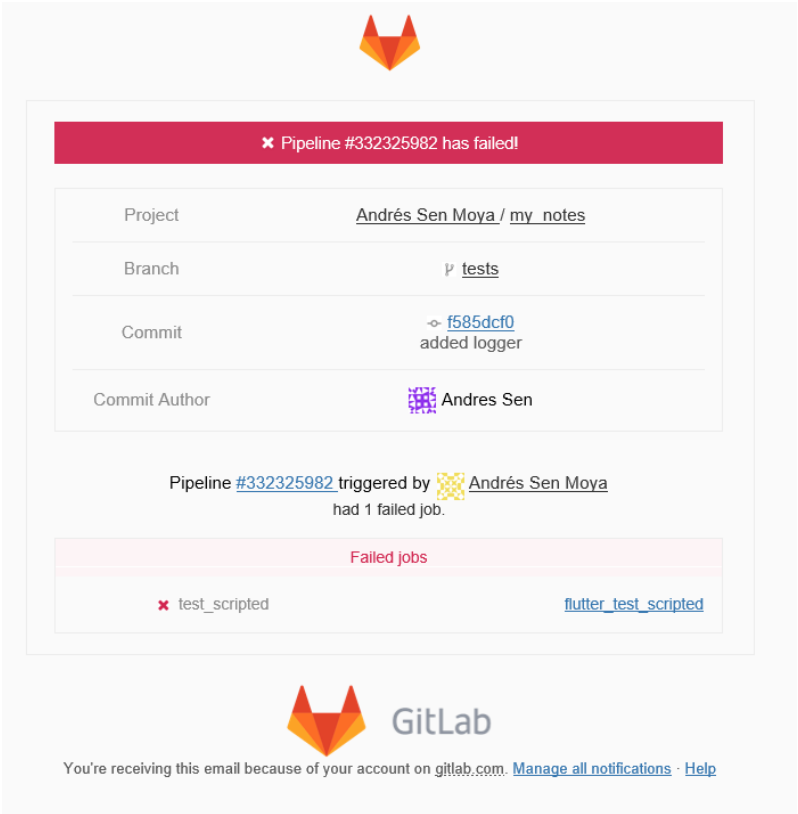


Figura 7.14: Correo electrónico en caso de fallo en alguna etapa

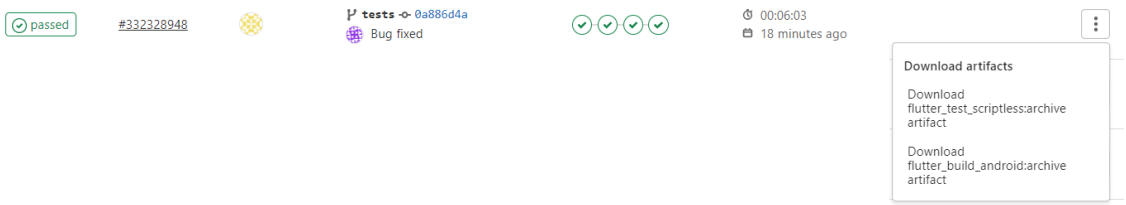


Figura 7.15: Artifacts generados por las etapas en GitLab

CAPÍTULO 8

Estudio de la complementariedad

8.1 Comparación de los enfoques por esfuerzo realizado para su utilización

Para medir el esfuerzo realizado se utilizaran diferentes métricas, estas son el tiempo que ha durado el desarrollo, que es lo que se ha tenido que implementar durante la implementación de cada enfoque y antes de esta. En la tabla 8.1 se puede observar esta comparativa.

Hay que tener en cuenta que en la comparación se hace sin tener el cuenta el tiempo requerido para adaptar Flutter a TESTAR ya que una vez adaptado puede ser utilizado por cualquier aplicación.

	Scripted	Scriptless
Duración desarrollo	5 horas	15 minutos
Desarrollo necesario puesta en marcha	Scripts de los tests	Nada
Preparación necesaria múltiple	Diagrama de flujo para definir los caminos que se ejecutarán	Nada

Tabla 8.1: Comparación de esfuerzo entre enfoque scripted y scriptless

8.2 Comparación de los enfoques mediante cobertura

8.2.1. Implementación de la infraestructura necesaria para medir la cobertura

Para medir la cobertura se ha decidió implementar un logger en Flutter. El logger añade todos los caminos y todas las transiciones que se van generando a unos archivos que se generan por cada ejecución del sistema y que se guardan en la memoria del dispositivo. Este logger esta incluido en el código del sistema y se llama desde las funciones del sistema, justo cuando se cambia de estado. Se ha implementado un método en el que se le pasa el estado al que se va a cambiar y a partir del estado anterior se detecta que transición se realiza.

Para cada criterio de cobertura se ha desarrollado un script distinto en python. Se ha desarrollado un total de 3 scripts:

1. Para medir la cobertura de nodo y de transición: Este script se encuentra en el apéndice E. Para este script se ha desarrollado un único método al que se le pasa dos argumentos el primero corresponde a la lista que se obtiene de la ejecución del sistema con el logger y donde se encuentran los estados o transiciones y el otro argumento es una lista con todos los estados o todas las transiciones posibles. Este método devuelve los estados o transiciones de la lista donde están todos que se encuentran en la lista de ejecución. Con esta lista ya podemos calcular el numero de nodos o transiciones por las que pasa la ejecución obteniendo así el porcentaje sobre el total y, por tanto, la cobertura de nodo y de transición.

2. Para medir la cobertura de caminos *Prime*: Para desarrollar este script lo primero que se ha realizado ha sido obtener una lista de todos los caminos *Prime*. Para hacer esto se ha utilizado la herramienta Prime-Path-Coverage [13] desarrollada por el usuario heshenghuan de GitHub. Una vez obtenida la lista con todos los caminos se ha desarrollado un script que comprobara que caminos *Prime* cubre la ejecución del sistema. Este script se encuentra en el apéndice F y se basa en tres métodos.

   El principal que es el que se encarga de llamar a los otros dos es el método getElementsOfArrayInAllElements y pasándole como argumentos en primer lugar la lista que contuviera todos los caminos *Prime*, y en segundo lugar se le pasaría como argumento la lista de caminos obtenida de la ejecución del sistema. Este método devuelve una lista con los caminos *Prime* que se encuentran dentro de la ejecución del sistema. En este método un camino se representa como una lista de estados. Con esta lista podemos obtener el porcentaje de cobertura de caminos *Prime*.

8.2.2. Porcentajes de cobertura

En la tabla 8.2 se muestran todos los porcentajes de cada cobertura para las diferentes herramientas. Respecto a TESTAR se ha ejecutado una única vez por cada profundidad. Estos porcentajes se han obtenido a partir de los documentos generados por el logger en cada ejecución y utilizando los scripts mencionados en la sección anterior.

	Flutter Driver	TESTAR prof. de 20	TESTAR prof. de 50	TESTAR prof. de 100	TESTAR prof. de 200
Cobertura de nodo	92,31 %	53,85 %	84,62 %	92,31 %	100 %
Cobertura de transición	76 %	40 %	72 %	76 %	92 %
Cobertura de camino *Prime*	5,26 %	4,09 %	9,36 %	8,19 %	16,37 %

Tabla 8.2: Porcentaje de cobertura según el criterio de cobertura y el enfoque

8.3 Comparación de los enfoques mediante la introducción de errores comunes

Para saber que errores son detectados por cada enfoque se introducen en el código. Una vez introducido el error se ejecutan ambas herramientas con la misma version de la aplicación, los errores no se han introducido todos juntos, si no que se ha creado una version distinta para cada error. A continuación se especifica las consecuencias de introducir cada error, explicando el porque de que una u otra herramienta lo detecte o no. En la tabla 8.3 se pueden ver de forma mas resumida que errores detecta cada herramienta. La herramienta TESTAR ha sido ejecutada con una profundidad de 50 un total de 3 veces.

1. Llamar setState en el método build. Al introducir este error en la aplicación se bloquea y no permite ejecutar ninguna acción, por este motivo se detecta en Flutter Driver y no en TESTAR ya que Flutter Driver quiere pulsar sobre un determinado botón y no puede mientras que TESTAR pulsa sobre los botones disponibles que en ese caso no es ninguno.

2. String mal escrito. Introduciendo este error la aplicación no guarda las notas correctamente provocando que no aparezcan en la lista de notas. Flutter Driver si que lo detecta ya que se espera que aparezca una nota en la lista sin embargo no aparece. Como en TESTAR no se han definido los oráculos adecuados para detectar este tipo de errores la ejecución de la aplicación no puede identificar que ha fallado ya que no espera que salga la nota en la lista.

3. Un InputDecorator no puede tener un ancho ilimitado. Este error provoca que la pantalla de edición de notas quede bloqueada al igual que ocurre con el error de llamar setState en el método build.

4. RenderFlex Overflowed. Este error se ha introducido en la barra de selección de color al editar la nota, provocando que la barra no pueda desplazarse. Como ninguna de las dos herramientas intenta desplazar la barra ninguna puede detectar este error.

	Scripted	Scriptless
Llamar setState en build	Si	No
String mal escrito	Si	No
InputDecorator con ancho ilimitado	Si	No
RenderFlex Overflowed	No	No

Tabla 8.3: Errores detectados por cada enfoque

CAPÍTULO 9
Resultados y conclusiones

9.1 Resultados

En primer lugar se hablará de la implementación de DevOps. En este trabajo se ha conseguido adaptar una aplicación ya programada en Flutter a metodología DevOps. Automatizando la parte de testing y de publicación. Sin embargo, solo se ha podido hacer en Android ya que no se disponía de un ordenador con sistema Mac OS para poder hacerlo en iOS además de que TESTAR todavía no es compatible con esta plataforma.

Respecto al estudio de la complementariedad. Se ha obtenido tres formas de comparar las diferentes herramientas con el fin de realizar este estudio. A continuación se hablará de los resultados obtenidos con cada comparativa.

1. Comparación de los enfoques por esfuerzo realizado para su utilización: Aquí se puede ver como TESTAR requiere de un menor esfuerzo para su implementación.

2. Comparación de los enfoques mediante cobertura: En esta comparación se puede ver como Flutter Driver tiene un mejor porcentaje de cobertura respecto a TESTAR cuando se ejecuta con menor profundidad, en cambio si se compara con las ejecuciones con mayor profundidad TESTAR es el enfoque que mejor cobertura de código tiene.

3. Comparación de los enfoques mediante la introducción de errores comunes: En esta comparación se observa como Flutter Driver detecta mejor los errores introducidos, esto puede deberse a que TESTAR no sabe si una aplicación se esta comportando de forma equivoca, sin embargo, Flutter Driver si que puede saber que una ejecución es la esperada.

9.2 Conclusiones

En la implementación de DevOps se han logrado todos los objetivos consiguiendo una implementación correcta de esta metodología.

Respecto al estudio de la complementariedad se puede concluir que ambas herramientas si son complementarias ya que, como se ha visto en el apartado anterior, donde destaca

uno de los enfoques el otro no, como se puede ver en la comparación del esfuerzo donde destaca TESTAR sin embargo en la comparación de introducción de errores destaca Flutter Driver.

9.3 Propuesta de trabajos futuros

Como trabajo futuro se puede hacer que la herramienta TESTAR con Flutter se pueda ejecutar en diferentes entornos sustituyendo la utilización de Windows API por otra. Además otra también se podría adaptar a iOS para que se pudiera ejecutar en este sistema operativo, ya que con Flutter se programa en ambos sistemas, sin embargo este trabajo se ha centrado en Android. Respecto a TESTAR también se puede mejorar la precisión en encontrar errores refinando los oráculos o utilizando algún mecanismo de selección de acciones en lugar de que se hagan de forma aleatoria.

Respecto al estudio de complementariedad se puede mejorar utilizando otras aplicaciones diferentes para así poder introducir otro tipo de errores. Además también se podrían introducir otro tipos de cobertura con el fin de mejorar este estudio.

APÉNDICE A

Scripts desarrollados para Flutter Driver

```
void main() {
  final buttonAddNote = find.byValueKey(buttonAddNoteKey);
  final buttonSearchNote = find.byValueKey(buttonSearchNoteKey);

  final buttonSave = find.byValueKey(buttonSaveKey);
  final buttonDelete = find.byValueKey(buttonDeleteKey);

  final noteSavedWarning = find.byValueKey(noteSavedWarningKey);

  final textTittle = find.byValueKey(textTittleKey);
  final textNote = find.byValueKey(textNoteKey);

  final confirmationDialogDeleteNote =
      find.byValueKey(confirmationDialogDeleteNoteKey);
  final confirmationDialogNoteIsModifiedNote =
      find.byValueKey(confirmationDialogNoteIsModifiedNoteKey);
  final confirmationDialogButtonMain =
      find.byValueKey(confirmationDialogButtonMainKey);
  final confirmationDialogButtonSecondary =
      find.byValueKey(confirmationDialogButtonSecondaryKey);

  const titleTextNewNote = "The title of the note";
  const contentTextNewNote = "The content of the note";

  const newTitleTextNewNote = "New modified";
  const newContentTextNewNote = "This is modified description";

  FlutterDriver driver;

  group('CRUD note', () {
    setUpAll(() async {
      driver = await FlutterDriver.connect();
    });

    tearDownAll(() async {
```

```
  driver.close();
});

test('open new note', () async {
  await driver.waitFor(buttonAddNote, timeout: const Duration(seconds: 4));

  await driver.tap(buttonAddNote);

  expect(await driver.getText(textTittle), "");
  expect(await driver.getText(textNote), "");
});

test('check note fields can write', () async {
  //Write the note
  await driver.tap(textTittle);
  await driver.enterText(titleTextNewNote);

  await driver.tap(textNote);
  await driver.enterText(contentTextNewNote);

  expect(await driver.getText(textTittle), titleTextNewNote);
  expect(await driver.getText(textNote), contentTextNewNote);
});

test('save note', () async {
  var noteFoundInList = true;

  //Save the note
  await driver.tap(buttonSave);

  //Wait note is saved
  await driver.waitFor(noteSavedWarning,
      timeout: const Duration(seconds: 4));

  await driver.waitForAbsent(noteSavedWarning,
      timeout: const Duration(seconds: 10));

  //Go to home Screen
  await driver.tap(find.pageBack());

  //Check if note is saved
  final noteElementByTitle = find.text(titleTextNewNote);
  final noteElementByContent = find.text(contentTextNewNote);
  try {
    await driver.waitFor(noteElementByTitle,
        timeout: const Duration(seconds: 1));

    await driver.waitFor(noteElementByContent,
        timeout: const Duration(seconds: 1));
  } catch (error) {
    noteFoundInList = false;
  }
```

```
  expect(noteFoundInList, true);
});
test('check warning modify note', () async {
  //Wait to be in home Screen
  await driver.waitFor(buttonAddNote, timeout: const Duration(seconds: 4));

  //Get the  note
  final noteElementByTitle = find.text(titleTextNewNote);

  //Edit the note
  await driver.tap(noteElementByTitle);
  await driver.waitFor(buttonSave, timeout: const Duration(seconds: 2));

  //Enter text
  await driver.tap(textTittle);
  await driver.enterText(newTitleTextNewNote);

  await driver.tap(textNote);
  await driver.enterText(newContentTextNewNote);

  //Check confirmation dialog
  await driver.tap(find.pageBack());

  await driver.waitFor(confirmationDialogNoteIsModifiedNote,
      timeout: const Duration(seconds: 2));

  await driver.tap(confirmationDialogButtonSecondary);
});

test('modify note', () async {
  var noteNotModifiedFoundInList = true;
  var noteModifiedFoundInList = true;

  //Save note
  await driver.tap(buttonSave);

  //Wait to note save
  await driver.waitFor(noteSavedWarning,
      timeout: const Duration(seconds: 4));

  await driver.waitForAbsent(noteSavedWarning,
      timeout: const Duration(seconds: 10));

  //Go to home screen
  await driver.tap(find.pageBack());

  //Check old note is not in list
  final noteElementByTitle = find.text(titleTextNewNote);
  final noteElementByContent = find.text(contentTextNewNote);
  try {
    await driver.waitFor(noteElementByTitle,
        timeout: const Duration(seconds: 1));
```

```
    await driver.waitFor(noteElementByContent,
        timeout: const Duration(seconds: 1));
  } catch (error) {
    noteNotModifiedFoundInList = false;
  }

  //Check new note is in list
  final noteModifiedElementByTitle = find.text(newTitleTextNewNote);
  final noteModifiedElementByContent = find.text(newContentTextNewNote);
  try {
    await driver.waitFor(noteModifiedElementByTitle,
        timeout: const Duration(seconds: 1));

    await driver.waitFor(noteModifiedElementByContent,
        timeout: const Duration(seconds: 1));
  } catch (error) {
    noteModifiedFoundInList = false;
  }

  expect(noteNotModifiedFoundInList, false);
  expect(noteModifiedFoundInList, true);
});
test('search note', () async {
  var noteFoundInList = true;
  await driver.tap(buttonSearchNote);
  await driver.waitFor(find.text("Input text to search"),
      timeout: const Duration(seconds: 3));
  //await driver.tap(searchBar);
  await driver.enterText(newContentTextNewNote);

  //Check note is show
  final noteElementByTitle = find.text(newTitleTextNewNote);
  final noteElementByContent = find.text(newContentTextNewNote);
  try {
    await driver.waitFor(noteElementByTitle,
        timeout: const Duration(seconds: 1));
  } catch (error) {
    noteFoundInList = false;
  }

  expect(noteFoundInList, true);
});
test('delete note', () async {
  var noteFoundInList = true;

  //Get the new note
  var noteElementByTitle = find.text(newTitleTextNewNote);

  //Edit the new note
  await driver.tap(noteElementByTitle);
  await driver.waitFor(buttonDelete, timeout: const Duration(seconds: 2));
```

```
      //Delete note
      await driver.tap(buttonDelete);

      //wait for delete note confirmation
      await driver.waitFor(confirmationDialogDeleteNote,
          timeout: const Duration(seconds: 2));

      //Tap confirmation to delete note
      await driver.tap(confirmationDialogButtonMain);

      //Wait to go home screen
      await driver.waitFor(buttonAddNote, timeout: const Duration(seconds: 4));

      //Check note is deleted
      noteElementByTitle = find.text(newTitleTextNewNote);
      final noteElementByContent = find.text(newContentTextNewNote);
      try {
        await driver.waitFor(noteElementByTitle,
            timeout: const Duration(seconds: 1));

        await driver.waitFor(noteElementByContent,
            timeout: const Duration(seconds: 1));
      } catch (error) {
        noteFoundInList = false;
      }

      expect(noteFoundInList, false);
    });
  });
}
```

APÉNDICE B

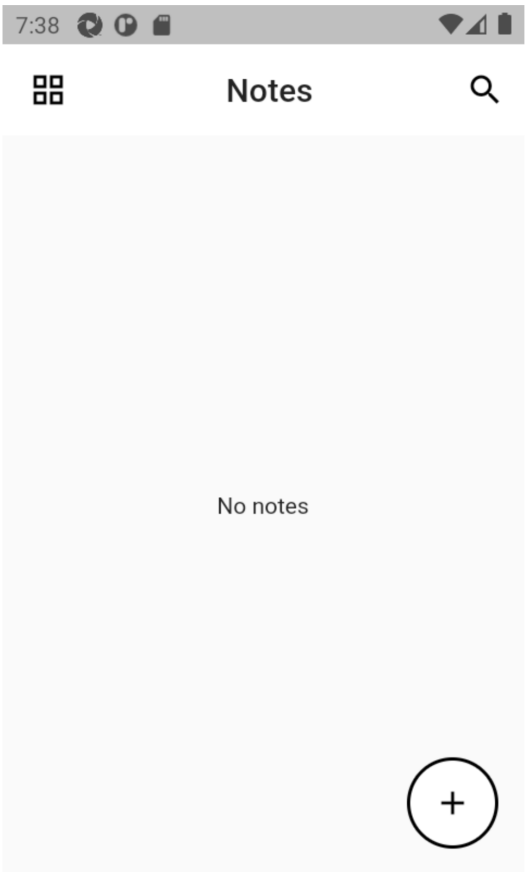
Extracto de informe de TESTAR

TESTAR execution sequence report for sequence 1

State 0

concreteID=SCC17m2ko6874091103420

abstractID=SAvlxgdy781549886454



Set of actions:

- **Left Click at 'android.widget.Button'** || Compound Action = Move mouse to (1269.0, 224.0). Press Mouse Button BUTTON1 Release Mouse Button BUTTON1 || ConcreteId=ACC1it5fe0823708708398 || AbstractId=AA16urxk5803682043865
- **Left Click at 'android.widget.Button'** || Compound Action = Move mouse to (1641.0, 880.0). Press Mouse Button BUTTON1 Release Mouse Button BUTTON1 || ConcreteId=ACCgm175282971634936 || AbstractId=AA4nnpb7801066408207
- **Left Click at 'android.widget.Button'** || Compound Action = Move mouse to (1671.0, 224.0). Press Mouse Button BUTTON1 Release Mouse Button BUTTON1 || ConcreteId=ACC1t5zpbh821948988878 || AbstractId=AA1h7m7hm801918030905

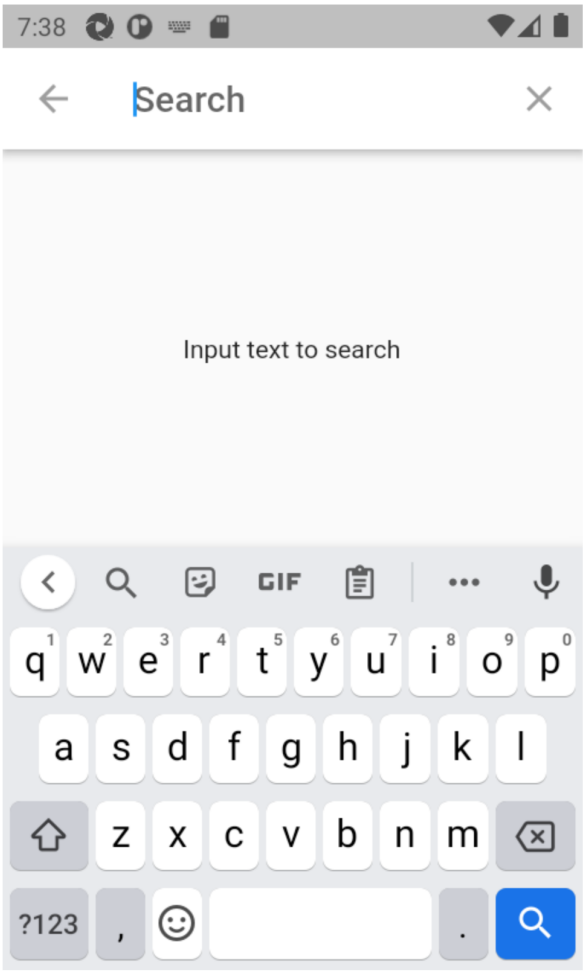
Selected Action 1 leading to State 1"

concreteID=ACC1t5zpbh821948988878 || Left Click at 'android.widget.Button'

State 1

concreteID=SCC1iyms561332692658940

abstractID=SA1dsfj3y118158288938



Set of actions:

- **Type 'www.foo.com' into 'android.widget.EditText** || Compound Action = Compound Action = Move mouse to (1473.0, 224.0). Press Mouse Button BUTTON1 Release Mouse Button BUTTON1 Press Key VK_CONTROL Press Key VK_A Release Key VK_A Release Key VK_CONTROL Type text 'www.foo.com' || ConcreteId=ACCzcy0b8fe821985950 || AbstractId=AA1czw82dd62160066023
- **Left Click at 'android.widget.Button'** || Compound Action = Move mouse to (1269.0, 224.0). Press Mouse Button BUTTON1 Release Mouse Button BUTTON1 || ConcreteId=ACC1mf1lkc822147139646 || AbstractId=AA8y1vw7812469283063
- **Left Click at 'android.widget.EditText'** || Compound Action = Move mouse to (1473.0, 224.0). Press Mouse Button BUTTON1 Release Mouse Button BUTTON1 || ConcreteId=ACCs18fxt823816047201 || AbstractId=AA1dlcs8s81262194856
- **Left Click at 'android.widget.Button'** || Compound Action = Move mouse to (1671.0, 224.0). Press Mouse Button BUTTON1 Release Mouse Button BUTTON1 || ConcreteId=ACC1wrvvht823604976606 || AbstractId=AAjaw5to81973805335

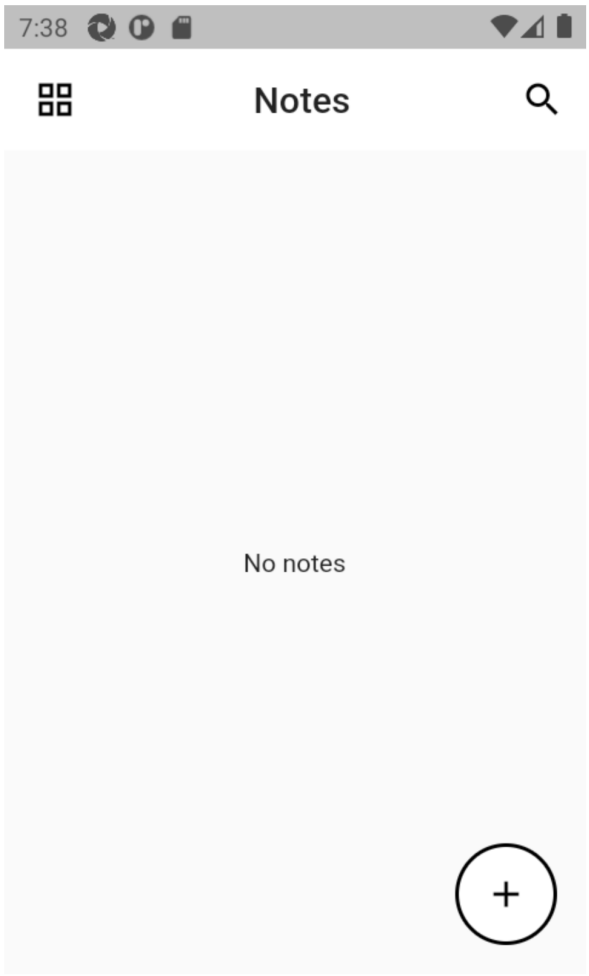
Selected Action 2 leading to State 2"

concreteID=ACC1mf1lkc822147139646 || Left Click at 'android.widget.Button'

State 2

concreteID=SCC1222wko14b2356430572

abstractID=SA1b1zmpz12c3157619116



Set of actions:

- **Left Click at 'android.widget.Button'** || Compound Action = Move mouse to (1671.0, 224.0). Press Mouse Button BUTTON1 Release Mouse Button BUTTON1 || ConcreteId=ACC1t5zpbh821948988878 || AbstractId=AA1h7m7hm801918030905
- **Left Click at 'android.widget.Button'** || Compound Action = Move mouse to (1269.0, 224.0). Press Mouse Button BUTTON1 Release Mouse Button BUTTON1 || ConcreteId=ACC1it5fe0823708708398 || AbstractId=AA16urxk5803682043865
- **Left Click at 'android.widget.Button'** || Compound Action = Move mouse to (1641.0, 880.0). Press Mouse Button BUTTON1 Release Mouse Button BUTTON1 || ConcreteId=ACCgm175282971634936 || AbstractId=AA4nnpb7801066408207

APÉNDICE C

Scripts GitLab CI/CD archivo .gitlab-ci.yml

```
stages:
  - build
  - test_scripted
  - test_scriptless
  - deploy

flutter_build_android:
  stage: build
  before_script:
    - flutter packages get
    - flutter clean
  script:
     - flutter build apk
  artifacts:
    paths:
      - build/app/outputs/flutter-apk/app-release.apk
  tags:
    - flutter

flutter_test_scripted:
  stage: test_scripted
  script:
    - echo running tests
    - flutter drive --target=test_driver/app.dart
  tags:
    - flutter

flutter_test_scriptless:
  stage: test_scriptless
  script:
    - echo running tests
    - scripts/run_testar.bat
  artifacts:
    paths:
      - test_testar/output/
```

```
  tags:
    - testar

flutter_deploy_android:
  stage: deploy
  before_script:
    - cd android
  script:
    - fastlane android alpha
  tags:
    - flutter
```

APÉNDICE D

Script para ejecutar TESTAR desde GitLab CI/CD

```
echo Running TESTAR
set run_dir=%CD%

echo Moving generated application to TESTAR directory
MOVE /Y .\build\app\outputs\flutter-apk\app-release.apk %TESTAR%\suts\calculator.apk

echo Running TESTAR
cd %TESTAR%
start /b /wait cmd /C "testar sse=android_generic ShowVisualSettingsDialogOnStartup=false
Sequences=2 SequenceLength=5 Mode=Generate"

echo Detecting last output generated by TESTAR
set datestr= %date:~4,2%%date:~7,2%%date:~10,4%
set currtime=%time: =0%
set timestr=%currtime:~0,2%%currtime:~3,2%%currtime:~6,2%
set currdatetime=%datestr%_%timestr%

FOR /F "delims=" %%i IN ('dir "%TESTAR%\output" /b /ad-h /t:c /o-d') DO (
SET last_folder=%%i
GOTO :found
)
goto :eof
:found

echo Moving TESTAR output to upload as artifact
mkdir "%run_dir%\test_testar\output\%last_folder%"
MOVE /Y "%TESTAR%\output\%last_folder%" "%run_dir%\test_testar\output\%last_folder%"
```

APÉNDICE E

Scripts en Python para medir cobertura de nodo y de transición

```
import pytest

#Obtiene los elementos dentro de los arrays
def getElementsOfArrayInAllElements(array,allElementsArray):
    elementsIn = []
    for element in allElementsArray:
        if element in array:
            elementsIn.append(element)
    return elementsIn

#Test
@pytest.mark.parametrize("testCase, array, allElements, expectedResult",[
    ("1",
    ["e1"],
    ["e1","e2","e3","e4","e5"],
    ["e1"]),

    ("2",
    [],
    ["e1","e2","e3","e4","e5"],
    []),

    ("3",
    ["e4","e2","e2"],
    ["e1","e2","e3","e4","e5"],
    ["e2","e4"]),

    ("4",
    ["e1","e2","e4","e2","e2"],
    ["e1","e2","e3","e4","e5"],
    ["e1","e2","e4"]),

    ("5",
    ["e1","e2","e3","e4","e5"],
    ["e1","e2","e3","e4","e5"],
```

```
    ["e1","e2","e3","e4","e5"]),

])

def testGetElementsOfArrayInAllElements(testCase,array,allElements,expectedResult):
    assert getElementsOfArrayInAllElements(array,allElements) == expectedResult
```

APÉNDICE F

Script en Python para medir cobertura de caminos

```
import pytest

#Devuelve los arrays que se encuentran dentro de bigArray
def getCombinationsIn(arrayList,bigArray):
    arraysIn = []
    for array in arrayList:
        if(conbinationIsIn(array,bigArray)):
            arraysIn.append(array)
    return arraysIn

#Comprueba si un array esta dentro de otro mas grande con varias posibilidades
def conbinationIsIn(array,bigArray):
    if len(array)==0 or len(bigArray)<len(array):
        return False
    positionsFirstElement = [i for i, e in enumerate(bigArray) if e == array[0]]
    for position in positionsFirstElement:
        if firstConbinationIsIn(array,bigArray[position : position+len(array)]):
            return True
    return False

#Comprueba si un array esta dentro de otro mas grande comprobando solo el primero
def firstConbinationIsIn(array,bigArray):
    if len(bigArray)<len(array):
        return False
    for pos in range(0,len(array)):
        if(array[pos]!=bigArray[pos]):
            return False
    return True


#Test
@pytest.mark.parametrize("testCase, arrayList, bigArray, expectedResult",[
    ("1",
    [["e4","e5"],["e2","e4"],["e1","e3","e5"],["e1","e2","e3","e4","e1"]],
    ["e1","e3","e4","e5"],
```

```
    [["e4","e5"]]),

    ("2",
    [["e4","e5"],["e2","e4"],["e1","e3","e5"],["e1","e2","e3","e4","e1"]],
    ["e1","e4","e2"]
    ,[]),

    ("3",
    [["e4","e5"],["e2","e4"],["e1","e3","e5"],["e1","e2","e3","e4","e1"]],
    ["e1","e3","e5","e1"],
    [["e1","e3","e5"]]),

    ("4",
    [["e4","e5"],["e2","e4"],["e1","e3","e5"],["e1","e2","e3","e4","e1"]],
    ["e1","e2","e3","e4","e1","e2","e4"],
    [["e2","e4"],["e1","e2","e3","e4","e1"]]),

    ("5",
    [["e4","e5"],["e2","e4"],["e1","e3","e5"],["e1","e2","e3","e4","e1"]],
    ["e2","e4","e1","e2","e3","e4","e1","e3","e5","e4","e5"],
    [["e4","e5"],["e2","e4"],["e1","e3","e5"],["e1","e2","e3","e4","e1"]]),

])

def testGetCombinationsIn(testCase,arrayList,bigArray,expectedResult):
    assert getCombinationsIn(arrayList,bigArray) == expectedResult
```

Bibliografía

[1] About Android App Bundles. https://developer.android.com/guide/app-bundle. [Online; accessed 27-06-2021].

[2] Visual Studio App Center | iOS, Android, Xamarin & React Native. https://appcenter.ms/. [Online; accessed 24-06-2021].

[3] Appium: Mobile App Automation Made Awesome. https://appium.io/. [Online; accessed 24-06-2021].

[4] Atlassian. Bamboo: servidor de integración continua y compilación. https://www.atlassian.com/es/software/bamboo. [Online; accessed 24-06-2021].

[5] Common Flutter errors. https://flutter.dev/docs/testing/common-errors. [Online; accessed 25-06-2021].

[6] R. Coppola, M. Morisio, and M. Torchiano. Scripted gui testing of android apps: A study on diffusion, evolution and fragility. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE, page 22–32, New York, NY, USA, 2017. Association for Computing Machinery.

[7] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. DevOps. *IEEE Software*, 33(3):94–100, May 2016. Conference Name: IEEE Software.

[8] - App automation done right. https://fastlane.tools/. [Online; accessed 24-06-2021].

[9] Firebase Test Lab | Test in the lab, not on your users. https://firebase.google.com/products/test-lab?hl=es-419. [Online; accessed 24-06-2021].

[10] Flutter - Beautiful native apps in record time. https://flutter.dev/. [Online; accessed 24-06-2021].

[11] Flutter testing. https://flutter.dev/docs/cookbook/testing/integration/introduction. [Online; accessed 24-06-2021].

[12] GitLab: Iterate faster, innovate together. https://about.gitlab.com/. [Online; accessed 24-06-2021].

[13] heshenghuan. heshenghuan/Prime-Path-Coverage. https://github.com/heshenghuan/Prime-Path-Coverage, June 2021. original-date: 25-03-2017.

[14] Hive | Dart Package - lightweight and blazing fast key-value database written in pure dart. strongly encrypted using aes-256. https://pub.dev/packages/hive. [Online; accessed 24-06-2021].

[15] Jenkins – an open source automation server which enables developers around the world to reliably build, test, and deploy their software. https://www.jenkins.io/. [Online; accessed 24-06-2021].

[16] UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey. [Online; accessed 24-06-2021].

[17] REAL ACADEMIA ESPAÑOLA. Diccionario de la lengua española. *Diccionario de la lengua española*, 23.ª ed.(versión 23.4 en línea), 2021.

[18] T. Tran-Nguyen. truongsinh/appium-flutter-driver. https://github.com/truongsinh/appium-flutter-driver, July 2021. original-date: 18-07-2019.

[19] T. E. J. Vos, P. Aho, F. Pastor Ricos, O. Rodriguez-Valdes, and A. Mulders. testar – scriptless testing through graphical user interface. *Software Testing, Verification and Reliability*, 31(3):e1771, 2021. e1771 stvr.1771.

[20] T. E. J. Vos and N. van Vugt-Hage. Structured software testing - Libro de texto para el curso - Pruebas de Software Estructuradas. page 263.