Diseño, cobertura y automatización de pruebas de software para una aplicación en Flutter

Daniel Felipe Jaramillo Álvarez

Informe de práctica para optar al título de Ingeniero de Sistemas otorgado por UdeA

Asesores
Robinson Coronado García, Especialista en desarrollo de software
Juan David Rivera Florez, Ingeniero de Sistemas

Universidad de Antioquia
Facultad de Ingeniería, Departamento de Ingeniería de Sistemas
Ingeniería de Sistemas
Medellín, Colombia
2024

Cita	(Jaramillo Álvarez, 2024)
Referencia **Estilo APA 7 (2020)**	[1] Jaramillo Álvarez D.F. (2024). *Diseño, cobertura y automatización de pruebas de software para una aplicación en Flutter* [Semestre de Industria]. Universidad de Antioquia, Medellín.

Tabla de contenido

Lista de figuras

Siglas, acrónimos y abreviaturas

AAA	Arrange, Act, Assert
Aprox.	Aproximadamente
IC	Integración contínua
IaC	Infraestructura como código (del inglés Infrastructure as Code)
S.	Segundos
TI	Tecnologías de la información
VSCode	Visual Studio Code

Resumen

Mis Aliados es un proyecto que hace parte de una alianza entre SURA y Bancolombia y cuenta con un conjunto de aplicaciones, incluyendo, entre otras cosas, con una aplicación móvil hecha en Flutter. El objetivo de la aplicación es servir como plataforma de contacto para que los trabajadores independientes en Colombia puedan ofrecer sus servicios y ser contratados por personas que estén necesitando estos servicios. Se plantea diseñar un plan de pruebas y su automatización para esta aplicación.

Palabras clave: aplicación móvil, pruebas, automatización, Flutter.

Abstract

Mis Aliados is a project that is part of an alliance between SURA and Bancolombia, and has a set of applications, including, among other things, a mobile application made in Flutter. The purpose of the application is to serve as a contact platform for independent workers in Colombia to offer their services and be hired by people who need these services. It is proposed to design a test plan and its automation for this application.

*Keywords***:** mobile app, testing, automation, Flutter.

Introducción

Este proyecto se enmarca dentro de **Mis Aliados**, un proyecto que se da como una alianza entre SURA y Bancolombia que busca conectar personas que trabajan de manera independiente[1] con personas que necesitan de estos servicios en su hogar. Cuando una persona necesita uno de estos servicios, puede solicitarlo ingresando a través de la página web de **Mis Aliados**. Por otro lado, cuando un independiente recibe una nueva solicitud de trabajo, debe realizar la gestión de ella a través de la aplicación móvil (disponible tanto para Android como para iOS).

Mis Aliados es un proyecto que cuenta con un sistema construído con diferentes tecnologías para backend, frontend (web y móvil) e IaC. Cuando el proyecto comienza a crecer es cuando más se hace notar la importancia de contar con pruebas de código.

El propósito de este documento es plasmar el trabajo realizado para incluir pruebas de diferente índole en una aplicación móvil hecha en Flutter y posteriormente poder ejecutarlas desde un ambiente de integración contínua (IC) para que se realicen en cada despliegue, dando tranquilidad a los desarrolladores de que no se ha inyectado errores funcionales dentro del código.

Este trabajo se hizo creando pruebas unitarias, de widgets, de integración y de interfaz gráfica (conocidas como pruebas doradas) para luego incluirlas en el *pipeline* de integración contínua en Codemagic que también se diseñó según las necesidades del proyecto.

[1] Es decir, personas que no cuentan con un contrato de trabajo establecido con una empresa. Estas personas realizan trabajos en diferentes sectores como: construcción, plomería, electricidad, carpintería, ebanistería, etc.

1 Objetivos

1.1 Objetivo general

Diseñar e implementar una batería de pruebas automatizadas en los diferentes escenarios dados por las historias de usuarios en rutas críticas de la aplicación, para así dar más confianza en los procesos de calidad en los despliegues de ambientes de test y producción.

1.2 Objetivos específicos

- Unificar despliegue contínuo en Codemagic.
- Diseñar conjunto de pruebas unitarias con un porcentaje del 35% en la cobertura de código.
- Implementar pruebas unitarias según fase de diseño.
- Implementar prueba de integración según fase de diseño y usando lenguaje Gherkin.
- Realizar pruebas *golden tests*.

2 Marco teórico

Cuando un producto de software es creado, es necesario garantizar que este producto funciona según lo esperado, tanto en su versión inicial como en versiones posteriores. Para garantizar este funcionamiento existen las pruebas de software, que pueden ser de muchos tipos, tales como: unitarias, de integración, de carga, de interfaz, etc.

Las pruebas unitarias se centran en validar individualmente las más pequeñas unidades de código, como funciones o métodos, para asegurarse de que se comporten como se espera. Las pruebas de integración verifican que los diferentes módulos o servicios funcionen correctamente cuando se interconectan. Las pruebas funcionales evalúan el sistema completo para asegurar que cumpla con los requisitos especificados, mientras que las pruebas no funcionales, como las pruebas de rendimiento, de carga y de estrés, evalúan el comportamiento del sistema bajo condiciones extremas.

En el contexto de las pruebas unitarias, se ha popularizado el patrón AAA (Arrange, Act, Assert) como una estructura para escribir pruebas más claras y comprensibles. El "Arrange" se refiere a la configuración inicial, donde se preparan los datos y se configuran las condiciones previas para la prueba. El "Act" implica llevar a cabo la acción real que se está probando, por ejemplo, invocar un método o función. Finalmente, "Assert" se encarga de verificar que el resultado obtenido sea el esperado. Este patrón ayuda a segmentar y organizar el código de prueba, proporcionando una estructura que es fácil de seguir y que asegura que todos los aspectos esenciales de una prueba se cubran de manera adecuada. (Microsoft, 2022).

Por otro lado, las pruebas de integración tienen como objetivo validar la interacción entre distintos módulos o componentes de un sistema. A diferencia de las pruebas unitarias, que se centran en unidades individuales de código, las pruebas de integración se enfocan en detectar problemas que pueden surgir cuando estas unidades interactúan entre sí. Estos problemas pueden incluir errores en la transferencia de datos, conflictos en la funcionalidad combinada y fallos en las interfaces. Es esencial realizar estas pruebas para asegurarse de que el sistema funciona de manera cohesiva como una entidad unificada. La complejidad de estas pruebas puede variar,

desde verificar la interacción entre dos módulos hasta evaluar la conectividad de sistemas enteros con múltiples componentes. (IEEE Computer Society, 2013).

La automatización de este tipo de pruebas se ha posicionado como una herramienta fundamental. Las pruebas automatizadas no solo garantizan que las nuevas características de una aplicación funcionen según lo previsto, sino que también verifican que las viejas características sigan funcionando correctamente. Esto es crucial para sistemas como el de **Mis Aliados**, que ha evolucionado a lo largo del tiempo y que integra diversos módulos.

En cuanto a Flutter (framework de desarrollo impulsado por Google), es una herramienta que ha ganado popularidad por su capacidad para desarrollar aplicaciones móviles nativas para múltiples plataformas a partir de un único código base. Aunque existen muchas herramientas para ayudar en la tarea de la automatización de pruebas, Flutter ofrece una variedad de herramientas y bibliotecas propias dedicadas específicamente para esta tarea.

En Flutter existen tipos de pruebas específicas para el mundo de este framework y de las aplicaciones móviles:

Pruebas de widgets: En Flutter, los widgets son elementos fundamentales de la interfaz de usuario. Las pruebas de widgets se enfocan en probar un único widget en aislamiento. Con la ayuda de herramientas propias del framework, se pueden realizar acciones simuladas sobre los widgets y verificar si están trabajando como se espera.

Golden tests: son un tipo de pruebas que permiten comparar la apariencia visual de un widget con una imagen de referencia guardada, también conocida como "imagen dorada" o "golden image". Estas pruebas son especialmente útiles para asegurar que la interfaz de usuario no cambie de manera no intencionada cuando se modifica el código.

Por otro lado, las prácticas de IC se han consolidado en el mundo del desarrollo como una estrategia esencial para garantizar la entrega de código de calidad. A través de la IC, cada cambio en el código es automáticamente probado y validado, permitiendo a los equipos de desarrollo identificar y solucionar problemas con mayor rapidez. Al integrar pruebas automatizadas en este

proceso, las empresas pueden asegurarse de que las actualizaciones o nuevas características no rompan la funcionalidad existente.

La cobertura de pruebas es una métrica que indica qué porcentaje del código fuente de una aplicación ha sido probado. Un porcentaje más alto de cobertura no necesariamente indica una mayor calidad del código, pero sí puede señalar áreas del código que no han sido probadas y que, por lo tanto, podrían ser propensas a errores.

Así mismo, es importante contar con mecanismos que permitan una mejor comunicación entre los equipos técnicos y los no técnicos, por lo que surgen lenguajes como Gherkin, que se utiliza para escribir pruebas legibles por humanos que describen el comportamiento de una aplicación sin detallar cómo se implementa dicha funcionalidad. Es decir, es un lenguaje enfocado en describir el qué y no el cómo. (Cucumber, 2008).

En Gherkin, las especificaciones se escriben en un lenguaje natural pero siguiendo una estructura específica, lo que permite que sean entendibles tanto para los desarrolladores como para los interesados no técnicos en un proyecto. Estas especificaciones luego se pueden utilizar para automatizar las pruebas de la aplicación, asegurando que cumpla con los comportamientos esperados.

3 Metodología

Scrumban es una metodología híbrida que combina elementos de Scrum y Kanban para crear un enfoque de gestión de proyectos y desarrollo de software más flexible y adaptativo. Conserva roles fundamentales de Scrum como el Scrum Master y el Product Owner, así como conceptos de iteraciones o sprints para el desarrollo. Sin embargo, incorpora el sistema visual de tablero y los límites de trabajo en curso de Kanban para mejorar la eficiencia del flujo de trabajo y adaptarse más fácilmente a cambios.

La metodología Scrumban es especialmente útil en proyectos que requieren la estructura y las disciplinas de Scrum pero que también necesitan la flexibilidad para manejar cambios y ajustes rápidos, típicos de un enfoque Kanban. Al fusionar lo mejor de ambos métodos, Scrumban permite a los equipos centrarse tanto en el desarrollo estructurado como en la entrega continua, al mismo tiempo que fomenta la mejora continua y la optimización del flujo de trabajo.

4 Resultados

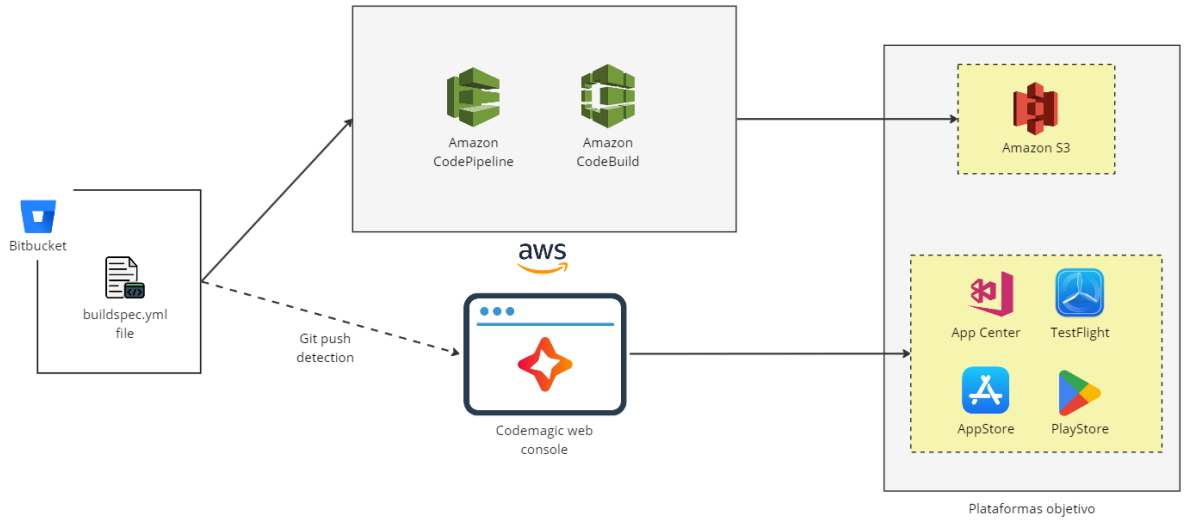
Los resultados obtenidos para este trabajo se pueden desglosar en 4 partes:

- Unificación del *pipeline* para despliegue contínuo
- Pruebas unitarias y de widgets
- Pruebas de integración
- Pruebas doradas (*golden tests*)

4.1 Unificación del *pipeline* para despliegue contínuo

Antes del inicio de este trabajo de prácticas, el sistema contaba con dos plataformas para realizar el trabajo: AWS y Codemagic. La configuración para AWS se tomaba de un archivo llamado *buildspec.yaml*, que residía en el repositorio del proyecto; por otro lado, la configuración para Codemagic se hacía a través de la consola web. Codemagic únicamente estaba asociado al repositorio para detectar cambios en el código de ciertas ramas (cuando se corría el comando *git push*) y así iniciar el lanzamiento de una nueva versión. En la **Figura 1** se puede observar el escenario de despliegue contínuo anteriormente descrito.

Figura 1
Estado inicial del despliegue contínuo de la aplicación



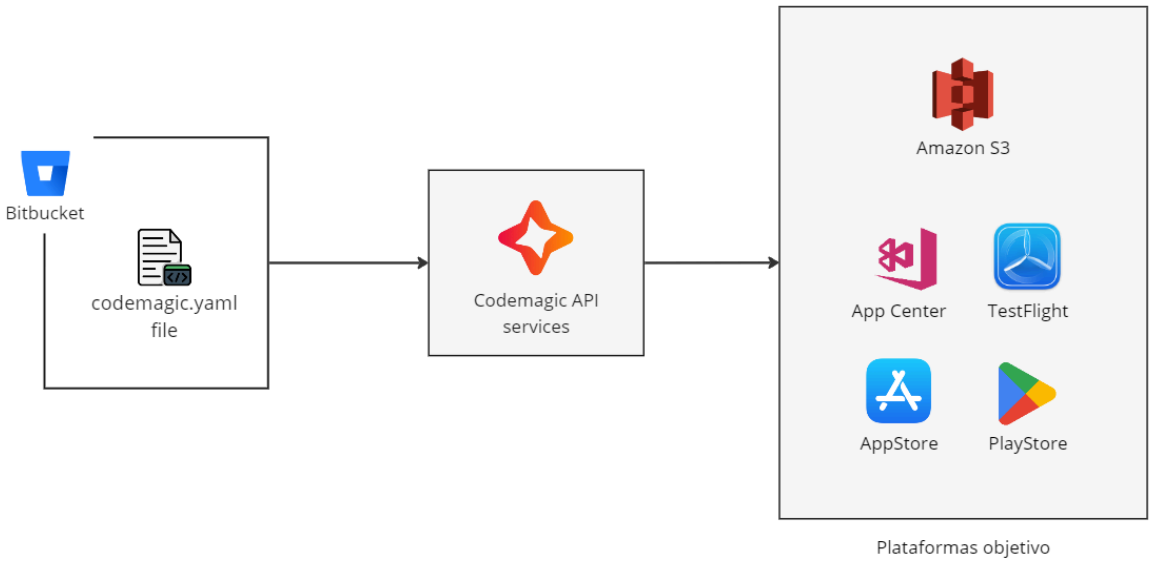
Nota. La imagen usada en este y otros diagramas para representar un archivo fue sacada de: https://icon-icons.com/icon/code-file-coding-paper-archive-document/219535

El objetivo planteado en este apartado se enfocó en dejar de depender de AWS para el despliegue contínuo de la aplicación. Nos estaba sucediendo que al momento de realizar algún cambio en la configuración del despliegue, por ejemplo cambiar la versión del SDK de Flutter, era necesario modificar diferentes archivos/configuraciones: el archivo *buildspec.yaml* para el despliegue web en AWS y la configuración en la consola de Codemagic para las tiendas. Este tipo de escenarios nos conllevan a, mínimo, las siguientes 2 dificultades:

1. Discordancias en la configuración que pueden terminar en despliegues fallidos.
2. Imposibilidad de automatizaciones debido a la necesidad de entrar a la configuración en consola web de Codemagic para realizar cambios necesarios.

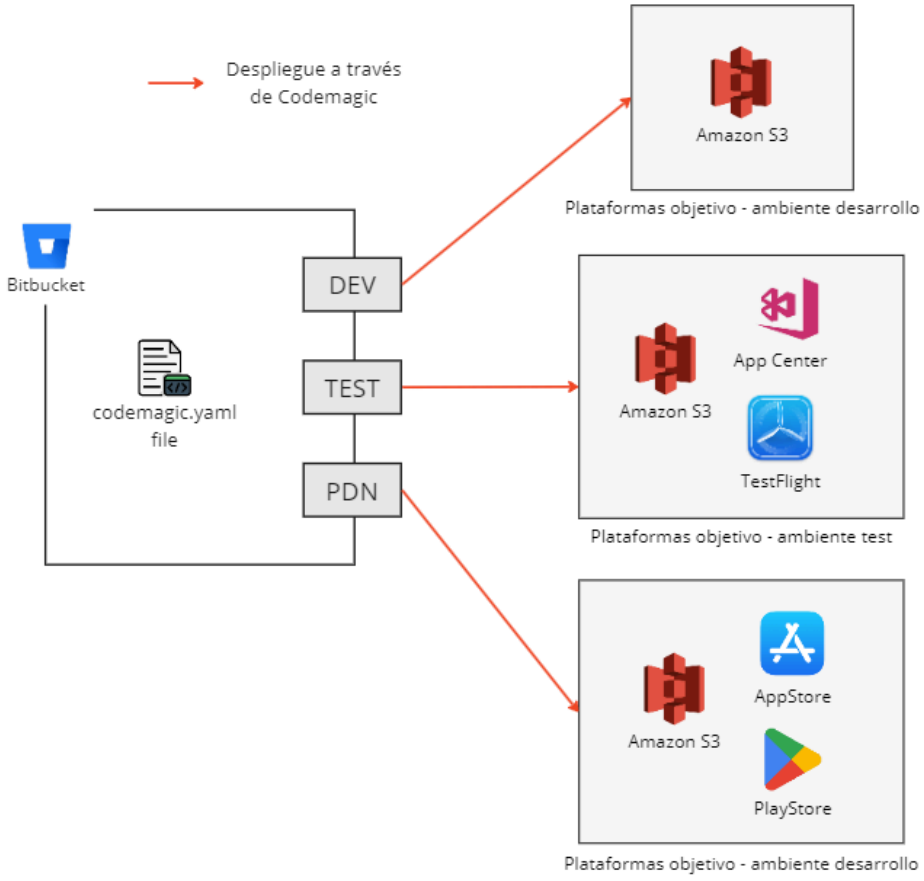
Para evitarnos este tipo de errores, nos enfocamos en centralizar el despliegue contínuo de la aplicación móvil únicamente en Codemagic a través de un único archivo llamado *codemagic.yaml* ubicado en el directorio raíz del proyecto, que es desde donde Codemagic toma la configuración necesaria para realizar la instalación de dependencias, construcción, pruebas y despliegue del proyecto a la plataforma deseada (la plataforma puede variar según el ambiente de despliegue; más detalles a continuación). En la **Figura 2** se puede ver que el flujo de despliegue una vez queda centralizado y, por tanto, simplificado.

Figura 2
Estado del despliegue contínuo luego de la refactorización



En Codemagic también se tuvieron en cuenta las plataformas a las que se debe desplegar según el ambiente: desarrollo, pruebas o producción. Para el ambiente de desarrollo únicamente es necesario desplegar la aplicación web alojada en S3; para el ambiente de pruebas se debe desplegar a la web, Android y iOS, usando AppCenter y TestFlight para estos últimos, respectivamente; y para el ambiente de producción también se debe desplegar en web, Android y iOS pero en las tiendas oficiales de cada sistema operativo. En la **Figura 3** se puede ver un diagrama que resume lo anteriormente explicado.

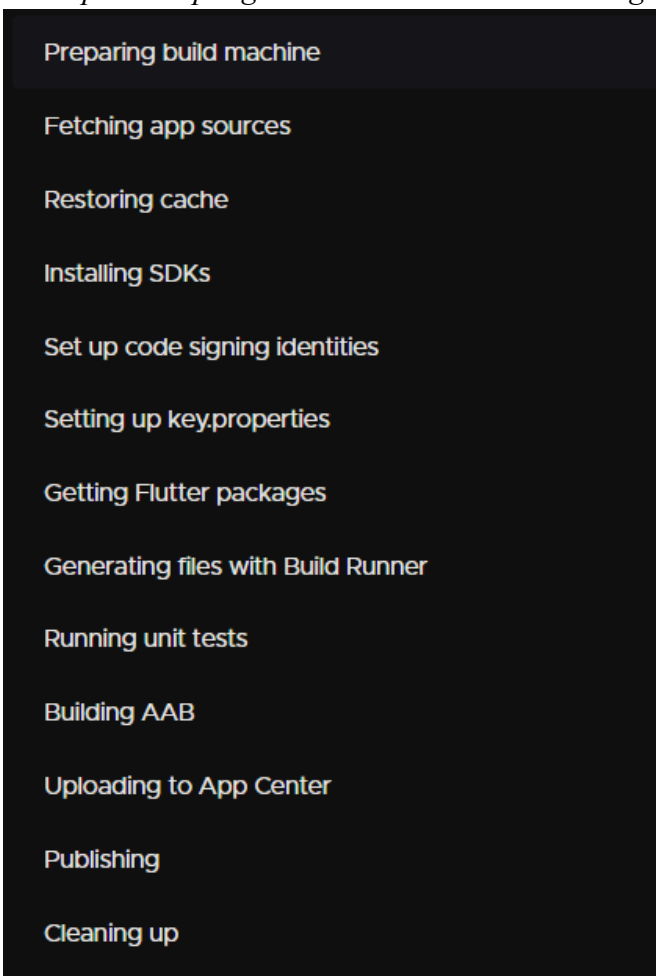
Figura 3
Plataformas objetivos según ambiente desplegado



Por último, en la **Figura 4** pueden verse desde la consola de Codemagic los pasos necesarios para realizar un despliegue a test en Android (es decir, se montará una nueva versión a App Center).

Figura 4

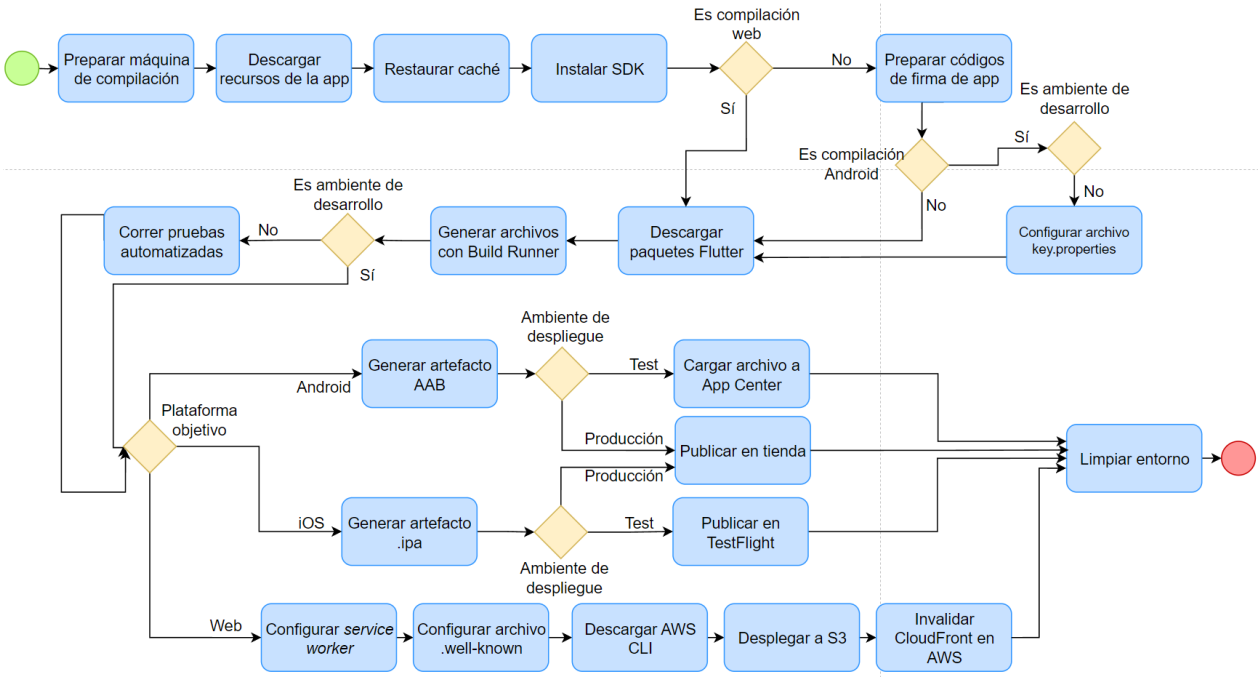
Pasos para despliegue desde consola en Codemagic



Recapitulando lo anteriormente dicho, en la **Figura 5** puede observarse el diagrama completo de flujo que se hace en Codemagic dependiendo del ambiente que se está desplegando.

Figura 5

Flujo completo de despliegue de aplicación en Codemagic

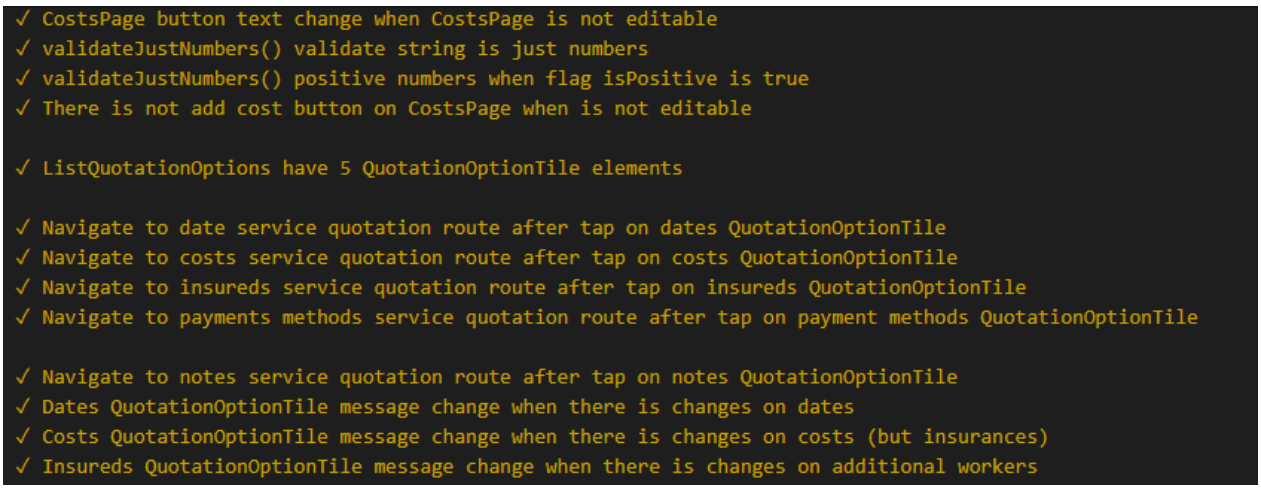


4.2 Pruebas unitarias y de widgets

Las pruebas unitarias y de widgets se ejecutan rápidamente (en Codemagic solamente toma aprox. 13 s; ver **Figura 9**) y no necesitan de un dispositivo para correr (al contrario de las pruebas de integración). Esto vuelve a las pruebas unitarias y de widgets métodos muy útiles de probar antes de un despliegue si la aplicación sigue funcionando según lo esperado. Estas pruebas pueden correrse tanto de forma local como configurarse para que sean corridos en el *pipeline*. Poder correr estas pruebas en el *pipeline* es de especial utilidad porque podemos configurar que si hay algún fallo en las pruebas entonces no sea desplegada la nueva versión de la app y así evitar que a nuestros usuarios llegue una versión con fallos. En la **Figura 6** y **Figura 9** pueden verse los resultados de correr las pruebas en un entorno local y en Codemagic, respectivamente.

Figura 6

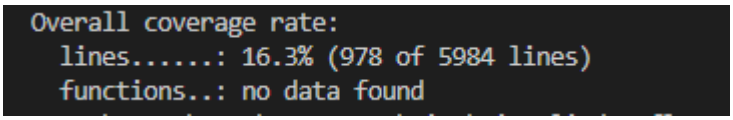
Consola en VSCode (en entorno local) mostrando algunos resultados de ejecutar las pruebas unitarias y de widgets



En la **Figura 6** se pueden observar algunos de los resultados de las pruebas realizadas exitosamente en un entorno local. Entre las pruebas se incluyen pruebas a elementos como botones, a funciones (*validateJustNumbers*), de navegación y de mensajes. Con estas pruebas realizadas se logra un **16.3%** en la cobertura de código, según se puede observar en la **Figura 7**[2].

Figura 7

Aparente cobertura de código con pruebas unitarias



[2] Para generar información de cobertura durante las pruebas unitarias y de widgets, debe correrse el comando ***flutter test --coverage***.

Sin embargo, hay una sutileza importante que puede pasarse por alto si no se observa con atención el número de líneas de código que se están tomando como base para calcular la cobertura (5984 líneas), que es un número considerablemente inferior a las tomadas como referencia en las pruebas de integración (ver **Figura 13)**. Esto sucede porque Flutter ignora los archivos de nuestro proyecto que no son accedidos **durante las pruebas**, por lo que gran parte de nuestro código queda sin hacer parte de la base sobre la que se calcula el porcentaje de código corrido durante las pruebas[3].

Para solucionar este error, recurrimos a crear un archivo "de prueba" que incluya el archivo con la función de entrada del proyecto (función *main*) para que podamos calcular la cobertura real de las pruebas. En la **Figura 8** se puede observar que una vez el porcentaje se calcula sobre la base de todo el código del proyecto, este llega a un **9.2%**.

Figura 8

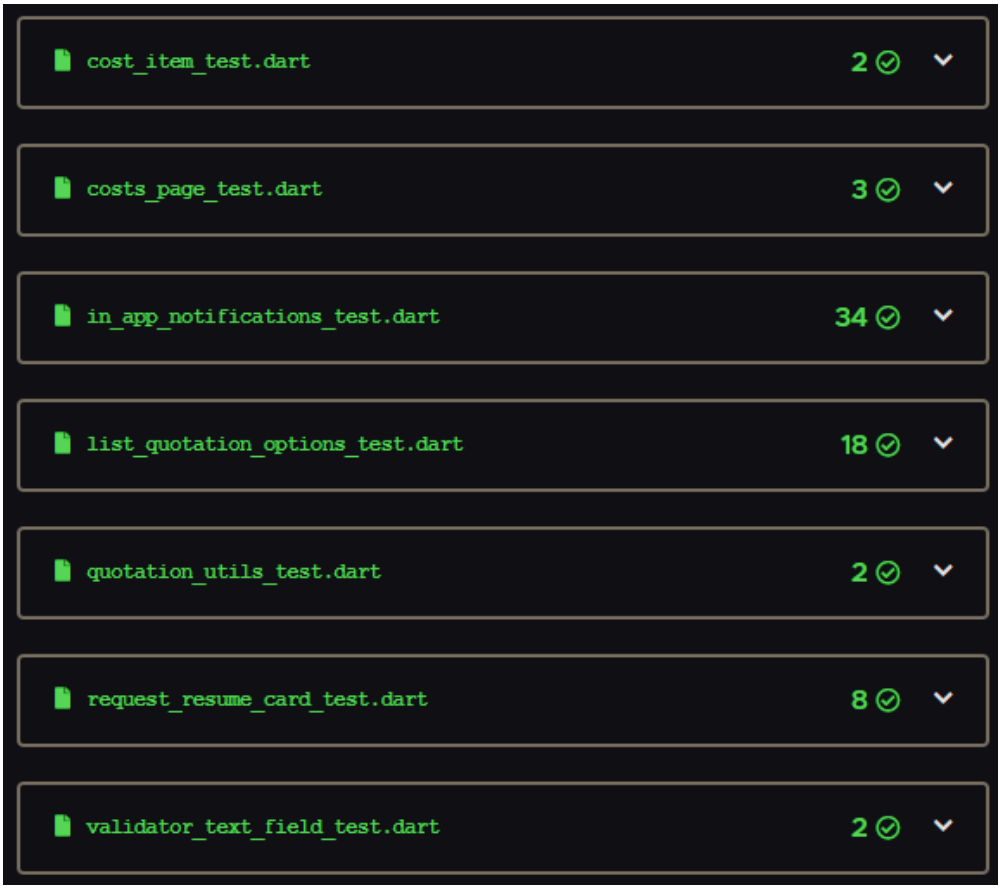
Aparente cobertura de código con pruebas unitarias

```
Overall coverage rate:
  lines.....: 9.2% (1059 of 11544 lines)
  functions..: no data found
```

[3] Ver https://github.com/flutter/flutter/issues/27997.

Figura 9

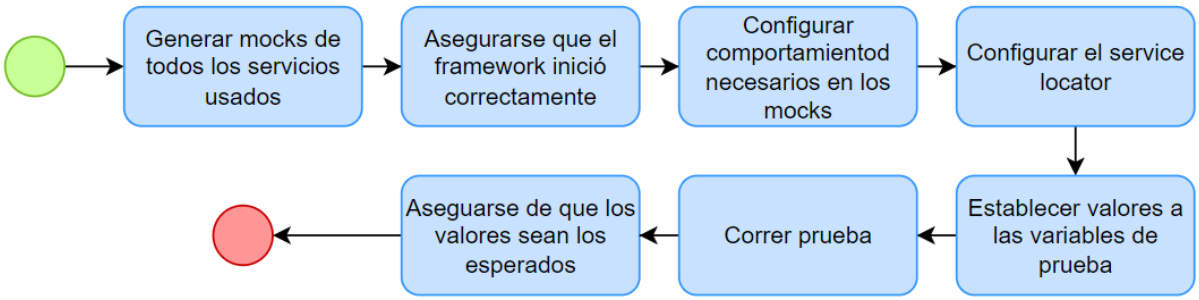
Consola en Codemagic mostrando el resultado de ejecutar las pruebas unitarias y de widgets



4.3 Pruebas de integración

Para correr las pruebas de integración es necesario realizar una serie de pasos que se pueden ver en la **Figura 10**.

Figura 10
Pasos necesarios para implementar y correr pruebas de integración



Los pasos se explican a continuación:

1. **Generar *mocks* de todos los servicios usados:** debido a que la aplicación utiliza un *service locator* para desacoplar los componentes, es necesario que este servicio cuente con las implementaciones necesarias al momento de realizar las pruebas. Para este propósito se usó la librería Mockito.
2. **Asegurarse que el framework inició correctamente:** es importante notar que esta comprobación se hace de manera diferente en la aplicación real y en las pruebas. En la aplicación real se hace mediante el objeto *WidgetsFlutterBinding*, mientras que en nuestro ambiente de pruebas usamos el objeto *IntegrationTestWidgetsFlutterBinding*, ambos encargados de cargar elementos del framework necesarios para cada ambiente.
3. **Configurar comportamientos necesarios en los mocks:** lo más probable es que el código que estamos probando haga un llamado a algunos de los métodos de los objetos que estamos simulando (con los *mocks*), por lo que es necesario configurar qué harán estos *mocks* al momento de que reciban este llamado. En el paquete *Mockito* este comportamiento se configura con el uso de la palabra clave *when*.
4. **Configurar el service locator:** el *service locator* hace parte fundamental de la arquitectura de nuestra aplicación, pues es la que se encarga de suplir de implementaciones concretas a las clases que tienen dependencias, disminuyendo así el acoplamiento entre componentes. La construcción del *service locator* en nuestro ambiente de pruebas se hace con los *mocks* generados anteriormente.

5. **Establecer valores a las variables de la prueba:** se deben establecer qué datos se usarán durante la prueba. Por ejemplo, en la prueba de integración que hacemos al proceso de autenticación (ver **Figura 11**), debemos establecer cuál será el correo y contraseña que se ingresan durante la prueba.
6. **Correr prueba:** este proceso puede tomar algún tiempo mientras se inicializa todo el entorno de pruebas, que debe ser lo más cercano posible a un entorno de ejecución real.
7. **Asegurarse de que los valores sean los esperados:** este paso a menudo se hace a medida que las pruebas corren. Sin embargo, también es posible separar por completo los procesos: primero correr la totalidad de las pruebas y según el estado final asegurarnos que los valores sean los esperados.

Por otro lado, para incentivar el uso de Gherkin dentro del equipo, se diseñaron los siguientes 3 casos de prueba para flujos relacionados con el inicio de sesión en la aplicación, mostrados a continuación:

1.

Feature: Inicio de sesión

Como independiente, quiero iniciar sesión en la aplicación para interactuar con la misma

Scenario: Inicio de sesión exitoso

Given un dispositivo en la pantalla de inicio de sesión

When se introduce un correo electrónico válido

And se introduce la contraseña correcta para dicho correo electrónico

And se hace clic en el botón "Iniciar sesión"

Then se redirige a la página principal de la aplicación

2.

Feature: Inicio de sesión

Como independiente, quiero poder saber si ingresé mis credenciales para inicio de sesión incorrectamente

Scenario: Inicio de sesión fallido con correo incorrecto

Given un dispositivo en la pantalla de inicio de sesión

When se introduce un correo electrónico inválido

And se introduce cualquier valor en el campo de contraseña

And se hace clic en el botón "Iniciar sesión"

Then se muestra un mensaje de error "Usuario o contraseña incorrectos"

3.

Feature: Inicio de sesión

Como independiente, quiero poder restablecer mi contraseña para poder volver entrar a mi cuenta

Scenario: Recuperación de contraseña

Given un dispositivo en la pantalla de inicio de sesión

When se hace clic en el enlace "¿Olvidaste tu contraseña?"

And se introduce un correo electrónico existente en el sistema en el campo de texto

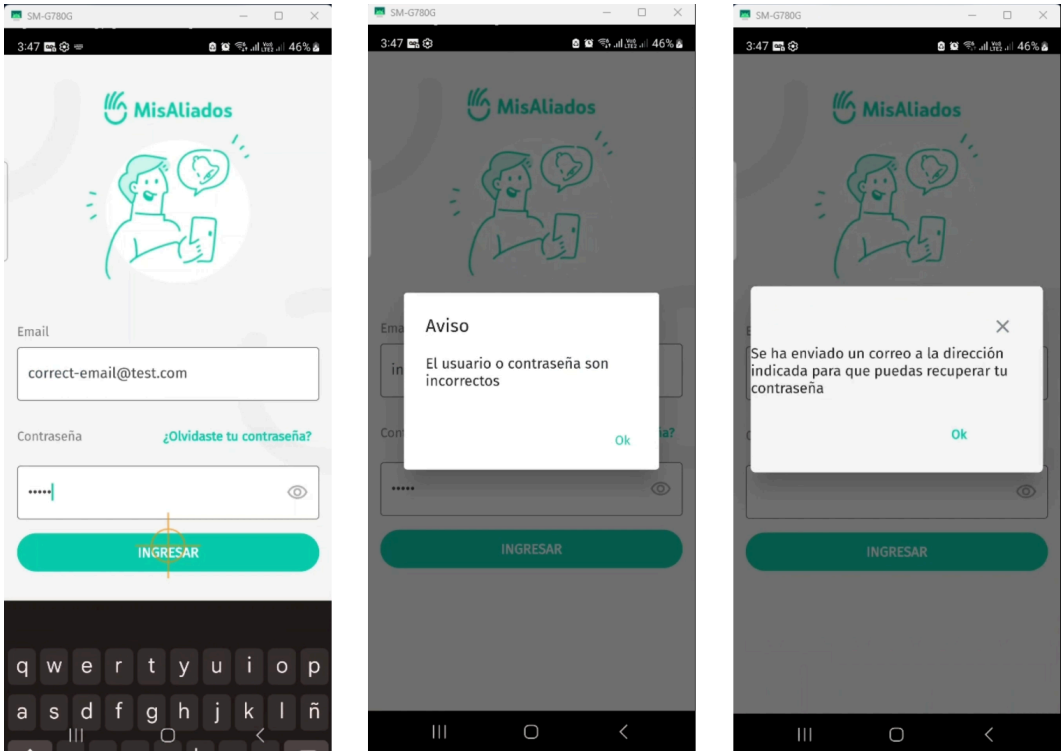
And se hace clic en el botón "Enviar"

Then se muestra un mensaje indicando que se ha enviado un correo de recuperación de contraseña a la dirección proporcionada

El resultado de implementar estos casos de prueba y correrlas se puede ver en la **Figura 11** y **Figura 12**. Además, en el siguiente link se puede ver un video de las pruebas del flujo de autenticación ejecutándose en un dispositivo real https://cyak.short.gy/integration-test-device.

Figura 11

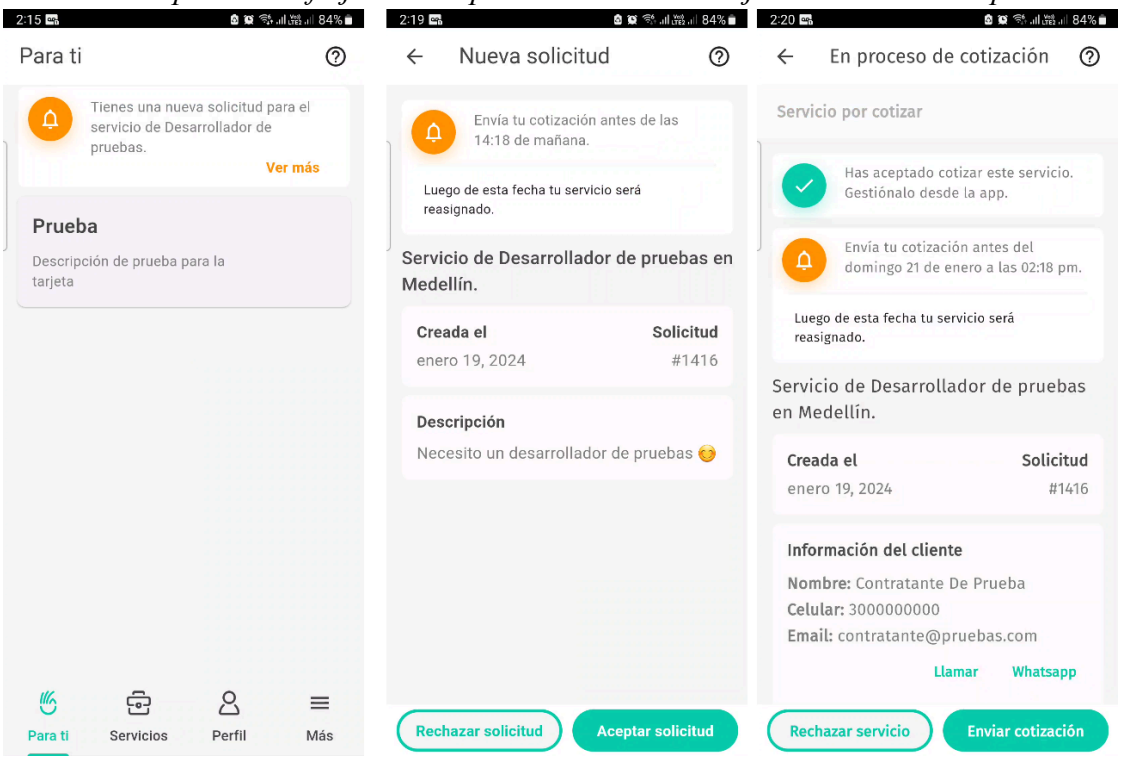
Secuencia de pruebas a flujos de inicio de sesión ejecutadas en un dispositivo real



En la **Figura 12** pueden verse 3 pantallas, a saber (de izquierda a derecha), cuando se está en la pantalla inicial (cuando se ha iniciado sesión), cuando se la al botón de "Ver más" en la tarjeta de notificación del servicio y la pantalla de detalles de la solicitud a la que se navega cuando se le da al botón de aceptar solicitud.

Figura 12

Secuencia de pruebas a flujos de aceptación de solicitud ejecutadas en un dispositivo real



Teniendo en cuenta que estas pruebas deben ejecutarse en dispositivos reales, su inclusión dentro del *pipeline* de IC deja de ser trivial y es necesario usar servicios (como Firebase, por ejemplo) que permitan facilitar el uso de las "granjas" de dispositivos para correr en ellos las pruebas. Sin embargo, siguen siendo de cierta utilidad para realizar las pruebas en entornos locales[4] y con estas dos pruebas de integración se logra una cobertura del **11.1%**, según se puede observar en la **Figura 13**.

Figura 13

Cobertura de código con pruebas de integración

```
Overall coverage rate:
  lines......: 11.1% (1279 of 11512 lines)
  functions..: no data found
```

[4] Para generar información de cobertura durante las pruebas de integración, debe correrse el comando ***flutter test integration_test --coverage***.

4.4 Pruebas doradas (*golden tests*)

Una *golden test* en Flutter es un tipo de prueba automatizada que se utiliza para verificar la correctitud visual de los widgets y la interfaz de usuario. Este tipo de prueba compara la representación actual de un widget o pantalla con una *golden image* (imagen dorada) de referencia, que es una captura de pantalla previamente aprobada y almacenada como estándar. Si la representación actual del widget coincide exactamente con la imagen de referencia, la prueba pasa. De lo contrario, la prueba falla, indicando que ha habido un cambio visual en el widget.

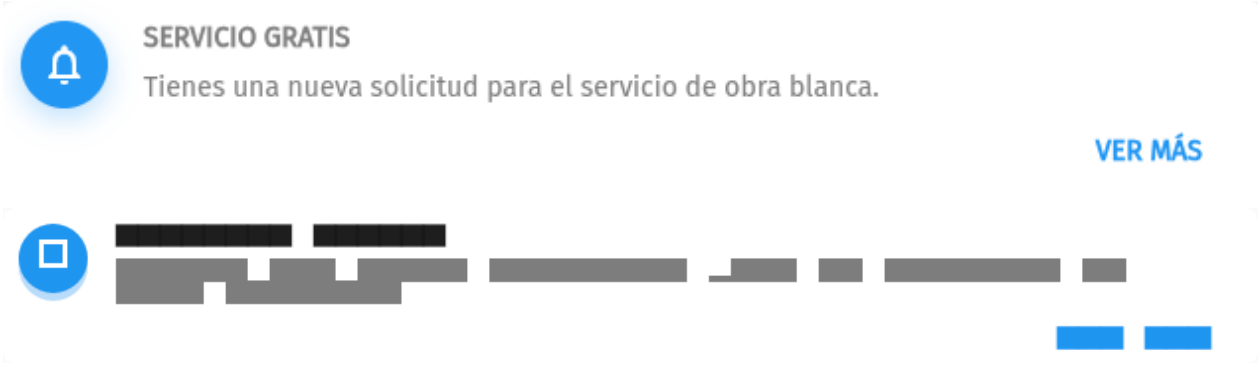
La configuración por defecto de estas pruebas en Flutter no permite renderizar imágenes, iconos ni una fuente entendible para humanos, pues usa la fuente **Ahem**, cuyos caracteres son todos exactamente cuadrados[5], lo que la hace especialmente útil en entornos de pruebas (Web-platform-tests.org, 2019). Sin embargo, esta configuración puede, en algunos casos, alterar la finalidad de las *golden test* puesto que, por ejemplo, alguna maquetación o widget que se quiera probar cuente con una imagen como parte integral del diseño, y una *golden image* sin la imagen no detecte errores debidos a un cambio inintencional de la misma.

Este tipo de pruebas se escriben de forma similar a las pruebas de widgets, que son más rápidas de correr que pruebas de integración pero más lentas que pruebas unitarias, debido a que los widgets deben ser "inflados". Esta velocidad de ejecución de las pruebas las hace unas buenas candidatas para que podamos probar una buena variedad pantallas y/o widgets que tengamos. En la **Figura 14** en la parte superior puede verse una carta existente en la aplicación que se está prueba con una *golden test*, en la parte inferior de la misma figura puede verse la *golden image* resultante que se usa como fuente de verdad al momento de determinar si la carta en sucesivas pruebas están bien.

[5] Salvo contadas excepciones, tales como la letra "p" o el carácter de espacio.

Figura 14

Una golden test realizada sobre un elemento muy específico de la interfaz (una carta)



Como se mencionó anteriormente, la flexibilidad de las *golden tests* permite probar elementos tan pequeños como una carta (como la vista en la **Figura 14**) o composiciones más complejas, como una pantalla completa. En la **Figura 15** puede verse una prueba realizada sobre la pantalla de inicio de sesión (esta pantalla puede verse en la **Figura 11**) y puede notarse que la *golden test* no renderiza las imágenes y los textos los muestra como cuadrados. Como este es un caso donde las imágenes forman parte integral del diseño se hace una configuración adicional para poder realizar renderizados en la prueba; el resultado de la imagen de prueba se puede ver en la **Figura 16.**

Figura 15

Golden test con la configuración por defecto realizado sobre una pantalla completa (pantalla de inicio de sesión)

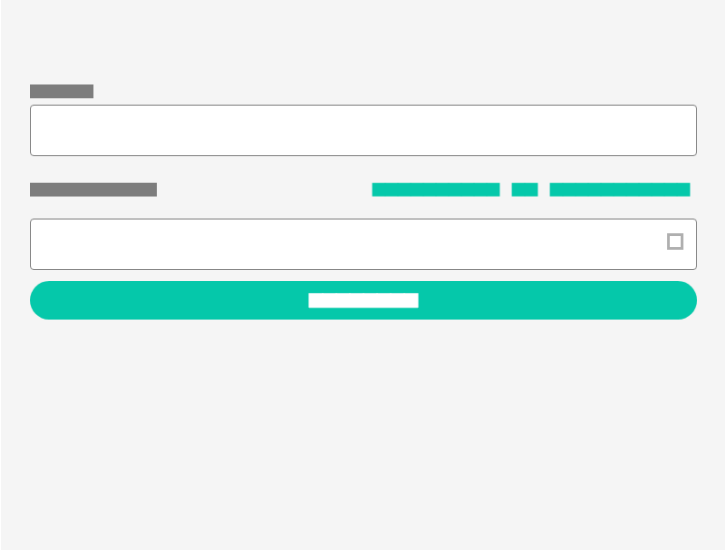
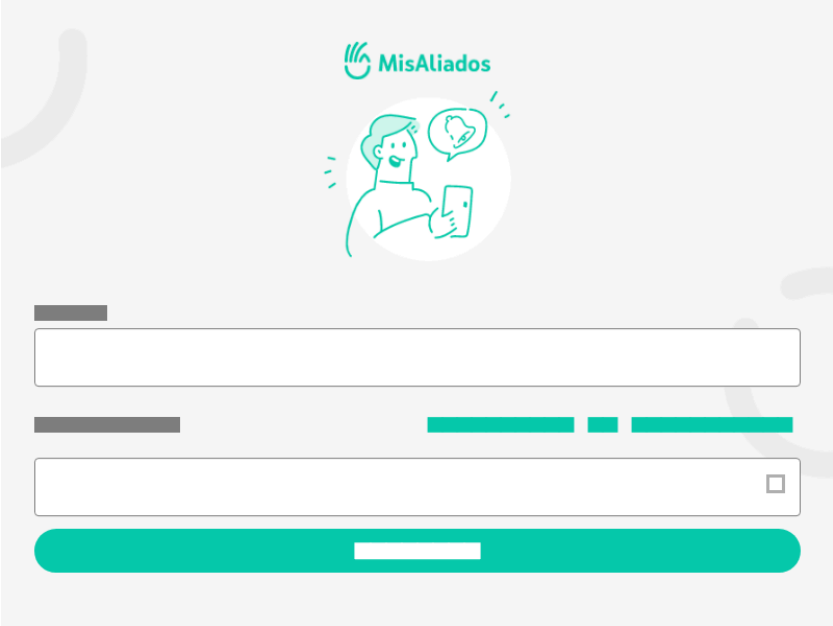


Figura 16

Golden test con una configuración adicional realizado sobre una pantalla completa (pantalla de inicio de sesión)



Este tipo de configuraciones adicionales en las *golden test* pueden ser tediosas de escribir a mano, por lo que existen paquetes como *Alchemist* o *Golden Toolkit* (desarrollada por eBay) que permiten ejecutar pruebas doradas con una configuración más avanzada, como añadir

fuentes, definir dimensiones y diferentes variaciones a las pruebas, pero para el alcance de este proyecto no fue necesario usarlos.

Otra característica muy interesante e importante de estas pruebas es que también se pueden ejecutar dentro de nuestros mecanismos de IC, heredando todas las bondades de las pruebas unitarias y de widgets. Sin embargo, al momento de realizar esta integración, Flutter cuenta con un error que causa que las *golden images* generadas desde Windows sean diferentes a las generadas por Mac's que cuenten con el chip M1 (ver https://github.com/flutter/flutter/issues/111739), que son el tipo de máquinas usadas en Codemagic, generando inconsistencias al momento de ver correr las pruebas, como se puede observar en la **Figura 17**.

Figura 17

Resultados al correr golden test en pipeline de IC en Codemagic

5 Análisis

Después de diseñar e implementar el sistema de pruebas para la aplicación móvil, queda claro la dificultad y esfuerzo que esto conlleva. Se deben tener en cuenta diferentes aspectos de la aplicación y deben ser probados en su mayoría para tener confianza en que el producto que llega al cliente final es de calidad.

Teniendo esto en cuenta, es importante destacar que en Flutter existen diferentes alternativas para probar nuestro código dependiendo del enfoque que se requiera. Por ejemplo, si queremos probar fragmentos pequeños de código, las pruebas unitarias son la mejor opción; sin embargo, si lo que buscamos es asegurarnos que nuestra interfaz gráfica se encuentre maquetada según nuestras expectativas, debemos usar *golden test*s. Una vez se conoce la utilidad de las *golden test* podría resultar tentador para probar diferente tipos de aspectos, como copies, pero es importante recordar que el costo computacional de estas pruebas es alto en comparación con las pruebas unitarias y de widgets (Poichet, 2021) y debe ser utilizado con mesura.

Por otro lado, aunque las pruebas son el eje central de un producto de software confiable, no se puede ignorar el trabajo requerido para incluir la ejecución dentro un pipeline de despliegue contínuo, puesto que es muy importante poder ejecutar nuestras pruebas cada que se hacen cambios en el código.

6 Conclusiones

Gracias a este trabajo se puede conocer el esfuerzo necesario para implementar las pruebas de código y lo que conlleva poder automatizarlas desde nuestros entornos de IC. Además, queda de relieve la importancia de las pruebas en los productos de software usando lenguajes de marcado como Gherkin para el entendimiento incluso de personas que no hagan parte del equipo de TI.

Por otro lado, aunque no se completó el objetivo del 35% de cobertura para las pruebas unitarias y de widgets, se logró un 20.3% en total, teniendo en cuenta las pruebas unitarias y las de integración. Es importante mencionar que aunque es posible que hayan algunos fragmentos de código que se ejecuten en ambos tipos de pruebas, es un porcentaje de código marginal puesto que las pruebas unitarias (y de widgets) y de integración se diseñaron para probar funcionalidades muy diferentes de la aplicación.

Teniendo en cuenta la variedad de formas que existen para probar el código hecho en Flutter, es importante reconocer la importancia y el caso de uso adecuados para cada una de ellas. Recapitulando lo visto, debido a que las pruebas unitarias y de widgets son las que más rápido se ejecutan, deben haber más cantidad; las de integración se ejecutan lentamente pero son muy útiles para probar flujos completos de nuestro sistema, pues son pruebas que se ejecutan en un entorno lo más parecido posible a un entorno real; por último, las *golden tests* también se ejecutan lentamente pero son muy útiles para probar que nuestra maquetación sea la correcta.

Por último, se logró una implementación satisfactoria de las pruebas dentro del *pipeline* de IC, dando tranquilidad de que no se van a realizar despliegues de artefactos defectuosos, y por lo tanto los usuarios de la aplicación no van a encontrar errores inesperados (al menos en los flujos probados). Sin embargo, quedan aún tareas por automatizar y mejorar, como pruebas de integración y *golden test* en el *pipeline* de IC.

Referencias

Cucumber. (2008). Gherkin Reference. Recuperado [2023] de https://cucumber.io/docs/gherkin/reference/

eBay. (2023). Golden Toolkit: A Flutter package for screenshot-based widget testing. Retrieved from https://pub.dev/packages/golden_toolkit

IEEE Computer Society. (2013). Pruebas de integración; Pruebas unitarias. En SWEBOK: Guide to the Software Engineering Body of Knowledge (Versión 3) (p. 86). https://ieeecs-media.computer.org/media/education/swebok/swebok-v3.pdf

Microsoft. (2022). Escribir las pruebas. En Conceptos básicos de las pruebas unitarias. Microsoft Learn. https://learn.microsoft.com/es-es/visualstudio/test/unit-test-basics?view=vs-2022#write-your-tests

Poichet, A. (2021). Flutter and the practical test pyramid with the BLOC pattern. Medium. Retrieved from https://medium.com/@a.poichet/flutter-and-the-practical-test-pyramid-with-the-bloc-pattern-6e6bf10dda9d

Web-platform-tests.org. (2019). The Ahem Font. Retrieved from https://web-platform-tests.org/writing-tests/ahem.html

Very Good Ventures & Betterment. (2023). Alchemist: A Flutter package for simplifying widget testing. Retrieved from https://pub.dev/packages/alchemist