UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Diseño e implementación del entorno y arquitectura para la ejecución de casos de test para una aplicación de C++ contenerizada

Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

AUTOR/A: Grimalt López, Ángel

Tutor/a: López Patiño, José Enrique

CURSO ACADÉMICO: 2023/2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

NO PONER PORTADA

Escuela Técnica Superior de Ingeniería de Telecomunicación
Universitat Politècnica de València
Edificio 4D. Camino de Vera, s/n, 46022 Valencia
Tel. +34 96 387 71 90, ext. 77190
www.etsit.upv.es

VLC/ CAMPUS
VALENCIA, INTERNATIONAL CAMPUS OF EXCELLENCE

EMAS
Gestión ambiental verificada
REG.NO. ES-CV- 000030

Resumen

Actualmente, la demanda de profesionales en el ámbito de desarrollo de software supera a lo que puede ofrecer el mercado, por lo que es interesante automatizar todos los procesos posibles así poder optimizar los recursos de un equipo. Además, automatizar procesos ofrece una serie de ventajas aparte de ahorrar tiempo, como pueden ser minimizar los errores humanos, o facilitar a nuevos integrantes o no especialistas la realización de ciertos procesos que requerirían de una formación para poder realizar de forma eficaz. Analizando todos estos beneficios cada vez disponemos de más herramientas para poder llevar esta tarea de automatizar procesos a cabo, de las cuales vamos a dar uso en este TFG

En este TFG se explicará primero el diseño del entorno en el cual se realizará todo el proceso de despliegue de la aplicación para poder ejecutarla correctamente. A continuación, el proceso de idear una batería de casos de test funcionales que servirán para comprobar que el comportamiento de la aplicación es el esperado y garantizar la calidad, y finalmente definiremos un flujo de trabajo por el cual estos procesos se realicen de forma automática.

Resum

Actualment, la demanda de professionals en l'àmbit de desenvolupament de software supera allò que pot oferir el mercat, per la qual cosa és interessant automatitzar tots els processos possibles així poder optimitzar els recursos d'un equip. A més, automatitzar processos ofereix una sèrie d'avantatges a més d'estalviar temps, com ara minimitzar els errors humans, o facilitar a nous integrants o no especialistes la realització de certs processos que requeririen una formació per poder realitzar de manera eficaç. Analitzant tots aquests beneficis cada cop disposem de més eines per poder portar aquesta tasca d'automatitzar processos a terme, de les quals donarem ús en aquest TFG

En aquest TFG s'explicarà, primer, el disseny de l'entorn en el qual es realitzarà tot el procés de desplegament de l'aplicació per poder executar-la correctament, a continuació, el procés d'idear una bateria de casos de test funcionals que serviran per comprovar que el comportament de l'aplicació és l'esperat, i finalment definirem un flux de treball pel qual aquests processos es facin automàticament.

Abstract

Currently, the demand for professionals in the field of software development exceeds what the market can offer, making it interesting to automate all possible processes to optimize team resources. Additionally, automating processes offers several advantages besides saving time, such as minimizing human errors or facilitating the execution of certain processes by new or non-specialist members that would otherwise require training to perform effectively. By analyzing all these benefits, we now have more tools available to carry out the task of automating processes, which we will use in this Final Degree Project (TFG).

In this TFG, we will first explain the design of the environment in which the entire application deployment process will be carried out to ensure it runs correctly. Next, we will describe the process of creating a battery of functional test cases that will serve to verify that the application's behavior is as expected. Finally, we will define a workflow through which these processes are automatically performed.

RESUMEN EJECUTIVO

La memoria del TFG del Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación debe desarrollar en el texto los siguientes conceptos, debidamente justificados y discutidos, centrados en el ámbito de la ingeniería de telecomunicación

CONCEPT (ABET)	**CONCEPTO (traducción)**	**¿Cumple? (S/N)**	**¿Dónde? (páginas)**
1. IDENTIFY:	1. IDENTIFICAR:		
1.1. Problem statement and opportunity	1.1. Planteamiento del problema y oportunidad	Si	6
1.2. Constraints (standards, codes, needs, requirements & specifications)	1.2. Toma en consideración de los condicionantes (normas técnicas y regulación, necesidades, requisitos y especificaciones)	Si	9-13
1.3. Setting of goals	1.3. Establecimiento de objetivos	Si	8
2. FORMULATE:	2. FORMULAR:		
2.1. Creative solution generation (analysis)	2.1. Generación de soluciones creativas (análisis)	Si	9-13
2.2. Evaluation of multiple solutions and decision-making (synthesis)	2.2. Evaluación de múltiples soluciones y toma de decisiones (síntesis)	Si	9-13
3. SOLVE:	3. RESOLVER:		
3.1. Fulfilment of goals	3.1. Evaluación del cumplimiento de objetivos	Si	56 -57
3.2. Overall impact and significance (contributions and practical recommendations)	3.2. Evaluación del impacto global y alcance (contribuciones y recomendaciones prácticas)		56 – 57

Índice

Índice de figuras

Índice de código

Capítulo 1. Introducción

En el exigente mundo del desarrollo de software y las telecomunicaciones, debido a la velocidad con las que las nuevas tecnologías aparecen y las antiguas dejan de ser útiles, la velocidad, además de la calidad es una obligación para poder estar a la altura de los competidores.

Un retraso en una entrega, un bug no localizado, o un nuevo miembro del equipo sin los conocimientos necesarios pueden provocar perder la confianza con un cliente o el fracaso de un proyecto, debido a la alta competitividad que existe.

Una forma eficaz de resolver estos problemas es la automatización de procesos y el uso de nuevas herramientas que nos permitan mejorar y optimizar el flujo de trabajo de un equipo, ya que estas nos permiten realizar tareas repetitivas de forma automática, reduce la intervención humana, evitando fallos puntuales que puedan suceder y nos permite mantener una regularidad en todo el proceso de desarrollo, evitando en gran medida arrastrar errores en este, garantizando calidad en cada cambio del propio código

A continuación, comentaremos algunos de los beneficios que surgen de la adopción de un flujo de trabajo automatizado en un equipo de desarrollo de software:

1. **Incremento en la eficiencia:** La automatización elimina tareas manuales que consumen un tiempo considerable, cosa que permite que los desarrolladores puedan enfocar su tiempo en tareas como la resolución de problemas de un carácter más creativo y optimización de código. Además, tener el despliegue de una aplicación automatizado, permite a los equipos poder entregar nuevas funcionalidades y versiones de una forma mucho más ágil, sin tener que sacrificar calidad.
2. **Calidad de código**: La automatización de las buenas prácticas como podrían ser compilación o la ejecución de casos de prueba funcionales nos permiten estabilidad en el código y la detección de bugs de una forma rápida. Esta calidad de código constante nos permite tener un producto mucho más estable y robusto.
3. **Minimizar el error humano**: Las personas tienden a realizar errores, especialmente con la ejecución de tareas repetitivas. Automatizar estas tareas nos permite mantener la integridad de la aplicación.
4. **Colaboración y revisión de código**: Estos métodos de flujo de trabajo automáticos también incluyen herramientas para facilitar la revisión de código, permitiendo a todos los miembros del equipo ver que cambios se están realizando, poder colaborar y compartir su opinión sobre ellos. Esto facilita transmitir conocimientos entre diferentes miembros del equipo y ayuda a tener un ambiente de trabajo más cooperativo.
5. **Escalabilidad y optimización de recursos**: A medida que crece un proyecto de software, la complejidad de este puede aumentar exponencialmente, cosa que provoca que manejar estos de forma manual pueda convertirse en todo un reto. La automatización nos permite evitar esto, ya que esta puede ir a la par que el crecimiento del proyecto, permitiendo ahorrarse el uso de más recursos.
6. **Centrarse en innovación**: Al automatizar las tareas repetitivas y mundanas, los desarrolladores pueden centrarse en aspectos más innovadores y creativos del desarrollo. Pueden invertir su tiempo en otras tareas como explorar nuevas tecnologías o resolver problemas.
7. **Consistencia en el ciclo de vida del software**: La automatización asegura que los procesos y pasos en el ciclo de vida del software se sigan de manera consistente en cada iteración. Esto ayuda a que no se deje ningún proceso sin hacer debido a descuidos o mala planificación.
8. **Flexibilidad del equipo**. La automatización de procesos nos permite no requerir de la persona experta en un área para realizar este proceso en cada iteración, ya que solo se ha necesitado de el para definir el proceso automático, esto permite poder afrontar una pérdida de un miembro (ya sea temporal o definitiva) de una mejor manera

En conclusión, la adopción de un flujo de trabajo automatizado en un equipo de desarrollo de software brinda una multitud de beneficios que impulsan la eficiencia, la calidad del código y la productividad en general. Al aprovechar la automatización para manejar tareas repetitivas, los equipos pueden centrarse en brindar valor, responder rápidamente a las demandas del mercado y producir productos de software confiables de manera constante.

Adoptar la automatización no es solo una cuestión de conveniencia, sino que se ha convertido en un paso necesario para los equipos de desarrollo modernos que luchan por la excelencia y el éxito en un panorama tecnológico que crece de forma vertiginosa.

Capítulo 2. Objetivos

Este trabajo de fin de grado se enfoca en el estudio integral de la automatización de un proceso de testing, que implica la creación de un entorno donde desplegar aplicaciones y ejecutar casos de pruebas funcionales de manera automatizada. El objetivo principal de este proyecto es proporcionar un ejemplo concreto de cómo automatizar completamente un proceso de testing, una tarea que muchas empresas aún realizan manualmente.

Este objetivo principal se puede desgranar en los siguientes objetivos:

Introducción a la Importancia de las Pruebas de Software: Destacar la importancia de las pruebas de software en el desarrollo de aplicaciones, explicando cómo ayudan a detectar errores, garantizar la calidad y mejorar la confiabilidad de los sistemas.

Diseño de un Entorno de Pruebas: Detallar el proceso de diseño y configuración de un entorno de pruebas adecuado para evaluar la aplicación. Esto podría incluir la selección de herramientas y recursos necesarios.

Desarrollo de Casos de Prueba Funcionales: Explicar cómo diseñar casos de prueba funcionales que evalúen las características clave de la aplicación y cómo ejecutar estas pruebas de manera manual.

Automatización de Casos de Prueba: Mostrar el proceso de automatización de casos de prueba funcionales definidos anteriormente.

Implementación de GitLab CI/CD: Explicar en detalle cómo configurar un pipeline de CI/CD en GitLab para automatizar las pruebas, la integración y la entrega continua. Destacar la importancia de esta automatización en la detección temprana de errores y la aceleración del proceso de desarrollo.

Documentación Detallada: Proporcionar documentación detallada paso a paso de todo el proceso, incluyendo ejemplos y capturas de pantalla.

Énfasis en la Calidad y la Seguridad: Destacar la importancia de garantizar la calidad del software y la seguridad de los datos a lo largo de todo el proceso de desarrollo, haciendo hincapié en las mejores prácticas y la mitigación de riesgos.

Evaluación de Resultados: Evaluar la efectividad de la automatización de pruebas y la implementación de GitLab CI/CD en términos de ahorro de tiempo, detección de errores, mejora de la calidad y eficiencia en el desarrollo.

Capítulo 3. Herramientas y terminología

En este capítulo vamos a tratar las diferentes herramientas que vamos a utilizar a lo largo de todo este proyecto, explicando el propósito de cada una y por qué nos hemos decantado por esa opción.

Tenemos que diferenciar 3 partes en nuestro proyecto, ya que en cada una necesitaremos unas herramientas diferentes que cumplan con su propósito:

- Configurar un entorno para el despliegue de la aplicación
- La ejecución y desarrollo de Casos de test automáticos
- Regresión utilizando GitLab CICD

Teniendo en cuenta estos tres puntos vamos a empezar por el primero.

3.1 Entorno y despliegue de la aplicación

En esta primera parte, tendremos que encontrar una solución a los siguientes problemas: Donde vamos a desplegar nuestra aplicación, y que necesitamos para ello.

3.1.1 Máquina virtual

Una máquina virtual es un entorno de software que emula una computadora física y permite ejecutar un sistema operativo y aplicaciones en un "ordenador virtual" dentro de un sistema anfitrión. Esta es la opción que vamos a elegir para empezar a crear nuestro entorno, ya que nos permite ahorrarnos la necesidad de adquirir una nueva máquina. Utilizaremos Ubuntu en su versión 20.04 como sistema operativo.[1]

3.1.2 Docker

Después de tener claro en el entorno en el que vamos a trabajar, necesitamos valorar que tecnología utilizar para el despliegue de nuestra aplicación. Nuestra aplicación requiere de dos contenedores para funcionar, uno el cual será el despliegue de nuestra propia aplicación, y el otro consiste en una Base de datos Hbase. Teniendo estos requerimientos en cuenta, optamos por utilizar Docker, que es la opción más utilizada en estos casos, aparte de tener los beneficios de ser Open Source (publica) y gratuita.

Docker es una plataforma de código abierto que se utiliza para la contenerización de aplicaciones. La contenerización es una tecnología que permite empaquetar una aplicación y todas sus dependencias, incluidas las bibliotecas y configuraciones, en un contenedor.

Nos decantamos por Docker por facilidad de despliegue, configuración y escalabilidad, y por la existencia de soluciones de Hbase server contenerizadas disponibles por la comunidad. Podríamos utilizar otras tecnologías de como Kubernetes,que consiste en el despliegue de un entorno con diferentes “pods” (contenedores) similar a Docker, pero debido al alto nivel de complejidad a la hora de configurarlo, nos hemos decantado por Docker.[2]

3.2 Desarrollo y ejecución de casos de tests automáticos

A continuación, después de tener un entorno en el cual podemos desplegar nuestra aplicación, tenemos que analizar cómo vamos a ejecutar nuestras pruebas de test automáticas, valorar las ventajas de diferentes herramientas y escoger una que cumpla nuestros requisitos.

3.2.1 Robot framework - Python

Para la elección de una herramienta de ejecución de test automáticos, hay que valorar diferentes características. Entre las cuales se encuentran:

- **Facilidad de uso:** Debe ser fácil de aprender y utilizar.
- **Flexibilidad:** Debe ser lo suficientemente flexible para adaptarse a diferentes tipos de aplicaciones y tecnologías.
- **Reutilización:** Debe permitir la reutilización de código para optimizar el desarrollo.
- **Mantenibilidad:** Debe facilitar la mantenibilidad de las pruebas a medida que la aplicación evoluciona, permitiendo realizar cambios de manera eficiente
- **Integración con CICD:** Debe integrarse sin problemas en pipelines de Integración Continua/Entrega Continua (CI/CD) para ejecutar pruebas de manera automática y continua.
- **Generación de informes:** Debe proporcionar informes detallados sobre la ejecución de pruebas, incluyendo resultados, estadísticas y capturas de pantalla en caso de errores.

Teniendo en cuenta todos estos factores, nos decantaremos por utilizar la herramienta robot framework, que cumple con todos los requisitos que buscamos, a continuación, una definición de esta herramienta.

Robot Framework es un marco de automatización de pruebas de código abierto que se utiliza para crear pruebas de aceptación y pruebas de sistema. Es especialmente útil en la automatización de pruebas de software, pero también se utiliza en otros contextos de automatización, como pruebas de hardware y procesos de negocios.

Las principales características de Robot Framework incluyen:

- **Lenguaje de dominio específico (DSL):** Robot Framework utiliza un lenguaje de alto nivel que es legible por humanos, lo que facilita la escritura de casos de prueba y la comprensión de los informes de resultados.
- **Soporte para bibliotecas de prueba:** Puede extender las capacidades de Robot Framework utilizando bibliotecas de prueba. Hay bibliotecas disponibles para diversas tareas, como automatización de pruebas web, pruebas de aplicaciones móviles, pruebas de bases de datos y más.
- **Formato tabular**: Los casos de prueba se escriben en formato tabular, lo que facilita la organización y la lectura.
- **Informes detallados**: Robot Framework genera informes detallados después de la ejecución de las pruebas, lo que facilita la identificación de problemas.
- **Reutilización de código**: Puede reutilizar código y casos de prueba existentes para evitar la duplicación de esfuerzos.
- **Soporte multiplataforma**: Robot Framework es compatible con sistemas operativos Windows, macOS y Linux.
- **Integración con otras herramientas:** Se puede integrar con herramientas de gestión de versiones, sistemas de generación de informes y sistemas de automatización de CI/CD (Integración continua y Despliegue Continuo).

Robot Framework se basa en Python para su ejecución, lo que permite a los usuarios aprovechar las ventajas de Python para personalizar y extender sus casos de prueba según sea necesario. En resumen, Robot Framework es una herramienta versátil para la automatización de pruebas que utiliza Python como su lenguaje subyacente y se utiliza en diversos entornos

de desarrollo de software y automatización de procesos. Facilita la creación de pruebas automatizadas de manera eficiente y proporciona informes detallados para ayudar a identificar problemas en una aplicación o sistema.

Existen multitud de frameworks para la ejecución de tests, como podrian ser pytest (python) o TestNG (java), pero nos decantamos por utilizar RobotFranework debido a todos los motivos mencionados anteriormente, destacando su ideal relación con otras herramientas como Gitlab CICD. [3]

3.3 Regresión utilizando GitLab CI/CD

Para finalizar, nos toca analizar la última de las 3 etapas, la regresión utilizando GitLab CI/CD, que consistirá en lanzar de manera automática todos los procesos anteriores, sin la necesidad de nosotros tener que lanzar nada manualmente. Utilizando esta herramienta podremos definir cuando queremos que se ejecuten estos procesos a través de Jobs dentro de las pipelines de nuestros proyectos.

Esta etapa básicamente consistirá en la aplicación de diferentes funcionalidades de la plataforma GitLab, así que empezaremos definiendo el concepto general de este, para posteriormente centrarnos en las características que nos interesará explotar.

GitLab es una plataforma de desarrollo de software que proporciona un conjunto completo de herramientas para la gestión del ciclo de vida del desarrollo de aplicaciones. Es conocido principalmente por ser un sistema de control de versiones, pero ofrece mucho más que eso. Algunas de las características y funcionalidades clave de GitLab incluyen:

- **Control de versiones**: GitLab ofrece un sistema de control de versiones distribuido basado en Git, lo que permite a los equipos de desarrollo gestionar y colaborar en el código fuente de sus proyectos de software de manera eficiente.
- **Gestión de proyectos**: Proporciona herramientas para la gestión de proyectos ágiles y tradicionales, lo que permite a los equipos planificar, realizar un seguimiento y colaborar en tareas y proyectos de desarrollo.
- **Integración continua (CI)**: GitLab CI/CD es una función que permite automatizar la construcción, prueba y entrega de aplicaciones, lo que mejora la calidad del software y acelera el proceso de desarrollo.
- **Registro de contenedores**: GitLab Container Registry permite a los desarrolladores almacenar y distribuir imágenes de contenedores de Docker directamente desde el repositorio GitLab.
- **Gestión de requisitos**: Permite definir y rastrear los requisitos del software, lo que facilita la colaboración entre los equipos de desarrollo y las partes interesadas.
- **Gestión de problemas y seguimiento de errores**: GitLab proporciona un sistema para gestionar problemas, realizar un seguimiento de errores y facilitar la colaboración en la resolución de problemas.
- **Colaboración y revisión de código**: Facilita la colaboración entre desarrolladores al ofrecer herramientas para la revisión y aprobación de solicitudes de extracción de código.
- **Seguridad y cumplimiento**: GitLab ofrece herramientas de seguridad, análisis de código estático y dinámico, y características de cumplimiento que ayudan a garantizar la calidad y la seguridad del software.

Después de esta definición general, vamos a explicar mas en detalle, diferentes conceptos que son necesarios para poder entender el concepto de la regresión que queremos implementar:

3.3.1 Flujo de trabajo en GitLab

Una de las ventajas que nos ofrece Gitlab, es poder implementar nuestro propio flujo de trabajo, utilizando las diferentes herramientas que nos ofrece según se adapten a nuestras necesidades.

A continuación, mostraremos un modelo de flujo de trabajo para poder comprender estas ideas:

1. **Creación del proyecto en GitLab**: El proceso comienza creando un proyecto en GitLab. En el cual será donde se recoja todo el código y archivos en el proceso de desarrollo
2. **Clonación del repositorio:** Los desarrolladores clonan (descargan) el repositorio del proyecto en sus propias máquinas locales para comenzar a trabajar en él.
3. **Creación de ramas**: Antes de realizar cambios en el código, los desarrolladores crean una nueva rama (branch) para cada tarea o característica que desean implementar. Esto

permite trabajar de forma aislada sin afectar la rama principal (por ejemplo, master o main) del proyecto.

4. **Desarrollo y confirmación de cambios**: Los desarrolladores realizan cambios en sus ramas locales y confirman (commit) sus modificaciones a medida que avanzan en sus tareas.
5. **Merge Requests - MR (solicitud de 'unión')**: Una vez que los desarrolladores han completado sus tareas, solicitan introducir el código de su rama en la rama principal del proyecto. En GitLab, esto se hace a través de la función "Merge Request". En la solicitud, se describen los cambios realizados y, si es necesario, se solicita la revisión de otros miembros del equipo.
6. **Revisión de código**: Otros miembros del equipo revisan el código propuesto en la solicitud de extracción y pueden dejar comentarios, solicitar modificaciones o aprobar la solicitud.
7. **Pruebas automáticas**: Si se ha configurado la integración continua (CI/CD), GitLab ejecutará automáticamente pruebas de código, construcciones y otros procesos definidos en el archivo de configuración CI/CD para garantizar que los cambios no rompan la funcionalidad existente.
8. **Ejecución de 'Merge Request'**: Una vez que la solicitud ha sido aprobada y las pruebas han pasado con éxito, se puede fusionar (merge) la rama de características en la rama principal del proyecto.
9. **Despliegue**: Si se ha configurado CI/CD, GitLab también puede gestionar la implementación automatizada del código en entornos de prueba y producción

Teniendo en cuenta estos conceptos, y explicada la idea de trabajo utilizando la plataforma GitLab, vamos a explicar en más detalle, los conceptos que se relacionan directamente con nuestro trabajo.

3.3.2 GitLab CI/CD & Pipelines

En este apartado vamos a definir en detalle los conceptos de GitLab CI/CD y *Pipeline*, que son fundamentales en la gestión de proyectos de desarrollo de software.

GitLab CI/CD se refiere a la Integración Continua (CI) y la Entrega Continua (CD) que se pueden lograr mediante la plataforma GitLab. La CI implica la automatización de pruebas y construcciones de código en un entorno compartido para detectar errores rápidamente. La CD se encarga de automatizar la entrega de software a entornos de prueba y producción de manera eficiente y confiable. En conjunto, CI/CD acelera el proceso de desarrollo y mejora la calidad del software.

Pipelines en GitLab son representaciones visuales de los pasos automatizados en un flujo de trabajo de CI/CD. Un pipeline es una serie de etapas que se ejecutan en orden, como la compilación de código, Tests unitarios, las pruebas de integración y la implementación en un servidor de prueba o producción. Los pipelines ayudan a los equipos a mantener un seguimiento de las tareas y a asegurarse de que el proceso de desarrollo se complete de manera eficiente y sin problemas.

En resumen, GitLab CI/CD y las *pipelines* son herramientas clave para automatizar y optimizar el desarrollo de software, lo que permite a los equipos desarrollar, probar y desplegar código de manera más eficiente y confiable. [4]

3.3.3 Relación entre commits y Pipelines

Explicados estos dos conceptos en los apartados anteriores, hay que explicar la relación entre estos y lo que provocan.

Entendiendo commit como un cambio en el código y una pipeline los pasos automatizados de un flujo de trabajo CI/CD, hay una relación directa entre estos dos conceptos, ya que siempre que haya un nuevo cambio en el código, se lanzara una *pipeline*, que generara una nueva versión de

la aplicación, y ejecutará los casos de test correspondientes. Esta sinergia es sobre la que se planteara todo nuestro proyecto, ya que, gracias a esta, podremos detectar fallos en la aplicación rápidamente, cosa que nos facilitara encontrar donde está el problema y poder solucionarlo de la forma más eficaz posible.

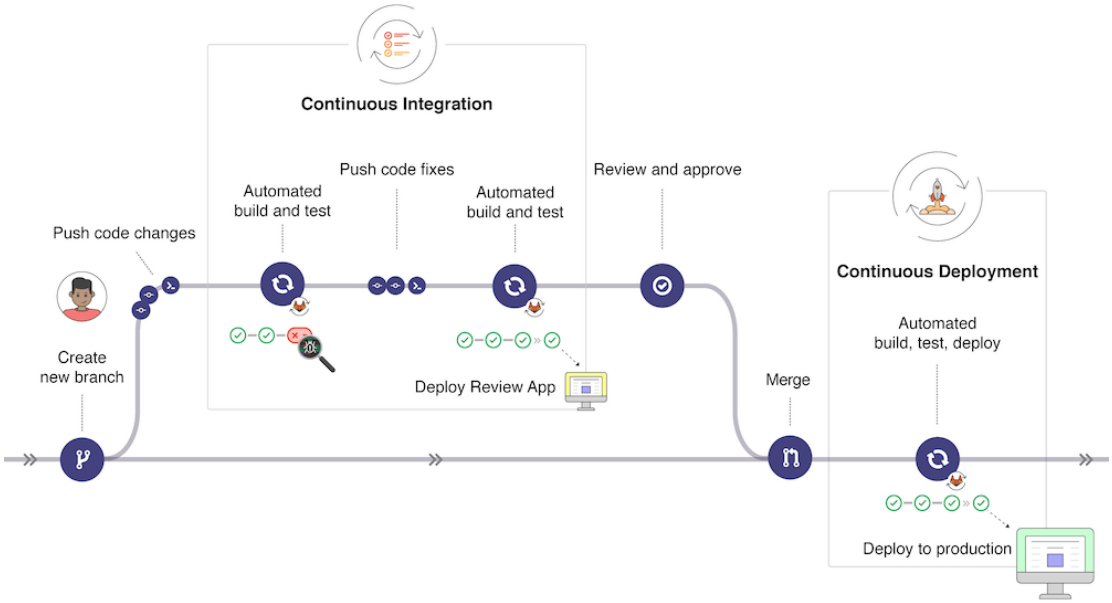


Figura 1. Representación de flujo de trabajo en Gitlab CI/CD

Capítulo 4. Desarrollo

En este capítulo, vamos a explicar paso por paso, todo el proceso de desarrollo de nuestro proyecto, para cumplir con los objetivos definidos en el capítulo 2, para lo cual nos apoyaremos en las tecnologías definidas en el capítulo 3. Este desarrollo parte desde el punto inicial, en el cual tenemos una máquina Windows, la cual tiene conexión a internet, y a su vez, acceso a una máquina virtual con Ubuntu 20.04 como sistema operativo.

4.1 Instalación de dependencias en la Máquina Virtual

En este aparto vamos a comentar como instalar todas las dependencias necesarias en nuestra máquina virtual, que por el momento solo tiene una instalación base del sistema operativo Ubuntu 20.04.

4.1.1 Python y Robot-framework

Para instalar Python en nuestra VM solo tendremos que seguir los siguientes pasos:

Para instalar Python utilizaremos el repositorio deadsnakes

sudo add-apt-repository ppa:deadsnakes/ppa

sudo apt-get update

sudo apt-get install python3.10 -y

sudo apt-get install python3.10-distutils -y

A continuación instalaremos curl y pip3

sudo apt install curl -y

sudo curl -sS https://bootstrap.pypa.io/get-pip.py | python3.10

Con esto ya deberíamos tener Python instalado en nuestra máquina, para confirmarlo podemos ejecutar el siguiente comando y nos devolverá la versión de nuestro programa:

python --version

Ahora vamos a instalar las librerías relacionadas con robotframework:

pip3 install robotframework==5.0

pip3 install robotframework-sshlibrary==3.8.0

pip3 install robotframework-jsonlibrary

pip3 install robotframework-kubelibrary==0.8.5

4.1.2 Docker

Para instalar Docker en nuestra VM seguiremos los siguientes pasos:

sudo apt update

sudo apt upgrade

sudo apt install apt-transport-https ca-certificates curl software-properties-common

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add –

sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"

sudo apt update

sudo apt install docker-ce

Para verificar que se ha instalado correctamente podemos ejecutar el siguiente comando al igual que hicimos con Python:

docker –version

Y nos devolverá que versión estamos utilizando

4.1.3 Giltab-runner

El servicio Gitlab-runner es el que nos permitirá que se puedan ejecutar los diferentes Jobs de la pipeline que definiremos más adelante en este proyecto, por lo que necesitaremos tenerlo instalado en nuestra máquina virtual

Para instalar el servicio de Gitlab-runner seguiremos el siguiente proceso:

curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh | sudo bash

sudo apt-get install gitlab-runner

sudo gitlab-runner register

En siguientes pasos crearemos el runner, pero ahora nos centraremos en el ‘set-up’ [5]

4.2 Despliegue de la aplicación

4.2.1 Descargar imágenes de docker.

Después de tener la VM preparada, vamos a desplegar todo lo necesario para poder ejecutar nuestra aplicación.

La aplicación que vamos a testear consiste en un Hbase-Sender, que es una aplicación de C++ que se encarga del manejo de una librería la cual tiene la función de insertar datos tablas dentro de un servidor de Hbase de una forma correcta.

Para ello necesitaremos desplegar las dos partes en forma de contenedor:

- Cliente (Hbase Sender)
- Un servidor Hbase

Para desplegar una aplicación de Docker contenerizada necesitamos la imagen de esta en nuestro registro de imágenes local

En nuestro caso, las imágenes para la aplicación y el servidor se encuentran en el registro de Gitlab por lo que deberemos hacer login en este desde nuestra máquina, para posteriormente hacer pull de nuestras imágenes.

El comando para poder descargar estas imágenes desde nuestro registro es el siguiente:

docker pull "registry_path"

En nuestro caso para descargar las dos imágenes que necesitamos ejecutaremos los siguientes comandos:

docker pull registry.gitlab.com/exfo/products/asa/msu/sdp/sender/hbase-sender:hbase-client

docker pull registry.gitlab.com/exfo/products/asa/msu/sdp/sender/hbase-sender:hbase-server

Con esto ya tendríamos las imágenes descargadas en nuestra máquina virtual, lo podemos corroborar mostrando las imágenes de nuestro registro local:

docker image ls

Que nos mostrara una lista con todas las imágenes.

Además de esto, necesitaremos tener el ejecutable de nuestra aplicación, que obtenemos compilando el código .

4.2.2 Despliegue de cliente.

Vamos a continuar con el despliegue del cliente.

Primero necesitamos saber qué comando necesitamos ejecutar para poder crear un contenedor de Docker a partir de una imagen, se puede hacer de diversas formas pero el mas básico (y el que utilizaremos) es el siguiente:

Docker run -d --name HbaseClient -it registry.gitlab.com/exfo/products/asa/msu/sdp/sender/hbase-sender:hbase-client /bin/bash

Vamos a desgranar las diferentes partes de este comando:

docker run: Es el comando principal de Docker que se utiliza para ejecutar contenedores.

-d: Esta bandera indica a Docker que ejecute el contenedor en segundo plano (en modo daemon). Esto significa que el contenedor se ejecutará en segundo plano y no bloqueará la terminal actual.

--name HbaseClient: Esta bandera especifica un nombre para el contenedor que se va a crear. En este caso, el nombre del contenedor será "HbaseClient".

-it: Estas dos banderas se usan juntas y tienen dos propósitos:

-i o --interactive: Mantiene el STDIN abierto incluso si no está conectado, permitiendo interactuar con el contenedor.

-t o --tty: Asigna un pseudo-TTY (terminal) al contenedor. Esto suele ser necesario para interactuar con la línea de comandos dentro del contenedor.

registry.gitlab.com/exfo/products/asa/msu/sdp/sender/hbase-sender:hbase-client: Esta es la imagen de Docker que se utilizará para crear el contenedor.

/bin/bash: Esta es la tarea que se ejecutará dentro del contenedor después de que se inicie. En este caso, se inicia un intérprete de Shell Bash dentro del contenedor. Esto permite interactuar con el contenedor después de que se inicie.

Para poder utilizar la aplicación tendremos que mover el ejecutable que tenemos en la VM dentro del contenedor del cliente:

docker cp HbaseClientTest hbaseClient:/home/HbaseClientTest

Después de ejecutar este comando ya tendremos nuestro cliente desplegado, a falta de encontrar un servidor.

Cabe recalcar, que la imagen utilizada contiene todas las dependencias para poder ejecutar la aplicación

4.2.3 Despliegue de servidor Hbase.

La segunda parte de nuestro despliegue consiste en el despliegue de nuestro servidor:

Seguiremos un proceso bastante similar al utilizado en el despliegue del cliente, pero con algunas variaciones.

Para empezar, necesitaremos crear un volumen en nuestra máquina virtual, que es donde nuestro servidor almacenara los datos.

En nuestro caso crearemos un directorio llamado data

mkdir data

A continuación, podremos ya desplegar nuestro servidor, el cual utilizara un comando ligeramente diferente al utilizado en el caso del cliente:

id=$(docker run --name=Hbase_server -h hbase-docker -d -v $PWD/data:/data registry.gitlab.com/exfo/products/asa/msu/sdp/sender/hbase-sender:hbase-server)

El cual consiste de diferentes partes:

id=: Esto indica que el resultado de la ejecución del comando se almacenará en una variable llamada id.

docker run: Comando principal de Docker para ejecutar un contenedor.

--name=Hbase_server: Esta opción establece el nombre del contenedor como "Hbase_server".

-h hbase-docker: Esta opción establece el nombre de host dentro del contenedor como "hbase-docker".

-d: Esta opción indica a Docker que ejecute el contenedor en segundo plano, en modo daemon.

-v $PWD/data:/data: Esta opción monta un volumen en el contenedor. $PWD/data es el directorio actual en el sistema local y /data es la ubicación dentro del contenedor donde se montará este directorio.

registry.gitlab.com/exfo/products/asa/msu/sdp/sender/hbase-sender:hbase-server: Esta es la imagen de Docker desde la cual se creará el contenedor

Ahora en este momento ya tenemos cliente y servidor desplegados, para poder utilizar nuestra aplicación solo nos faltaría un último paso: Crear una tabla de datos.

4.2.4 Crear Tabla de datos en el servidor Hbase.

Para finalizar con el despliegue de la aplicación necesitaremos crear dentro del propio servidor, una tabla a la cual insertaremos datos con nuestro cliente.

Cabe recordar que nuestra aplicación se encarga de introducir datos en las tablas de un servidor previamente creadas, éste no tiene la funcionalidad para crear las propias tablas.

Para crear una tabla en el servidor utilizaremos **Hbase Shell** , que incluye nuestro servidor , con la cual podemos realizar diversas acciones dentro de este.

Primero abrimos el Hbase Shell del servidor:

docker exec -it Hbase_server hbase shell

Despues de esto, podremos ejecutar comandos propios de hbase :

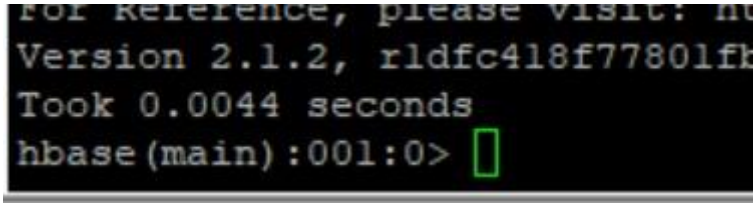


Figura 2. Hbase shell abierto correctamente

Como último paso, crearemos la tabla a la que llamaremos data_table (podríamos crearla con cualquier nombre ya que este parámetro es configurable en nuestro cliente).

create 'data_table','C'

El segundo parámetro ‘C’ es el nombre de las columnas, en nuestro caso, la aplicación tiene este valor por defecto, por lo que siempre utilizaremos este valor en cualquier tabla.

Con esto ya tendríamos la tabla creada, podemos confirmarlo ejecutando el siguiente comando:

list

El cual nos mostrara todas las tablas en el servidor.

4.2.5 Ejecución de la aplicación.

Después de realizar todos los pasos anteriores, nuestra aplicación está lista para poder ser utilizada. Para comprobar que todo ha ido correctamente, vamos a ejecutar un caso básico de nuestra aplicación.

Ahora mismo nuestra máquina virtual debería tener los dos contenedores desplegados:

Figura 3. Contenedores de servidor y cliente desplegados en la VM

El procedimiento para poder ejecutar nuestra aplicación es el siguiente:

Dentro de nuestro cliente se encuentra el ejecutable compilado, el cual pasándole los parámetros correspondientes será capaz de poder insertar datos en el servidor.

Los parámetros que requiere la aplicación son los siguientes:

- **nb-threads:** número de hilos (procesos en paralelo) en los que se enviaran las ráfagas.
- **nb-burst:** número de ráfagas que envia el sender**.**
- **nb-rows-in-a-burst:** número de filas que contiene una ráfaga.
- **nb-row-bytes-min:** número mínimo de bytes que contiene una fila.
- **nb-row-bytes-max:** número máximo de bytes que contiene una fila .
- **sleep-time-in-us-beetween-bursts:** tiempo que transcurre entre el envío de una ráfaga y la siguiente.
- **server-IP:** IP del contenedor en la cual se encuentra el servidor de Hbase.
- **table-name:** nombre de la tabla a la cual el sender enviara los datos.
- **print-stats:** parámetro que sirve que estadísticas debe mostrar la aplicación (en cada ráfaga, al final de toda la ejecución, etc).
- **continue-on-exception:** Si hay un fallo en el envio, seguir enviando hasta que finalice la ejecución o detener el envío**.**
- **logger:** que nivel de detalle se quiere de los logs de la aplicación (info,error,debug …).
- **memory_type:** memoria dinámica o fija.
- **batch_specification:** tamaño de ráfaga se necesita para poder realizar el envío
- **reconnect-on-fatal-ex:** En caso de desconexión con el servidor, decidir si intentar reconectarse o en su defecto, devolver un error.

Estos parámetros nos permiten modificar distintas características del envió, así como la cantidad de datos, quien los va a recibir, si queremos que muestre estadísticas/logs etc

Para nuestra primera sencilla prueba los únicos datos que realmente nos inportan son Server-IP y table-name, ya que, si estos no coinciden con los de nuestro cliente, el envío será erróneo.

Para obtener la IP de nuestro servidor ejecutaremos el siguiente comando:

docker inspect Hbase_server

Figura 4. Ip del servidor Hbase

cómo podemos ver, la IP de nuestro servidor es 172.17.0.4.

Teniendo este valor, y sabiendo el nombre de la tabla p que hemos creado anteriormente podemos ejecutar la aplicación de la siguiente forma

docker exec -w /home hbaseClient ./HbaseClientTest 1 2 1 45 45 0 172.17.0.4 data_table overall-stats continue-on-exception=no debug fixed=1,1 size=1 reconnect-on-fatal-ex=no'

Si todo ha ido correctamente, la aplicación nos devolverá el siguiente mensaje, que significa que el envío se ha realizado correctamente:

```
exfo@serspvl06:~$ docker exec -w /home hbaseClient ./HbaseClientTest 1 2 1 45 45 0 172
exception=no debug fixed=1,1 size=1 reconnect-on-fatal-ex=no
Data generation completed

===== Thread 0: Iteration 1 =====
Info hbase_batch_sender::send(span) Appended to data store: 1, 45
Info hbase_sender::send Created batch mutations from data: 1
Warning hbase_sender::send exception encountered: unsuccessful items, bytes: 1, 45
Info hbase_batch_sender::send_data_batch Successfully appended to errors store: 1, 45
Info hbase_batch_sender::send_errors_batch Retrying with errors store occupancy: 1/1
Warning hbase_sender::send exception encountered: unsuccessful items, bytes: 1, 45

===== Thread 0: Iteration 2 =====
Info hbase_batch_sender::send(span) Appended to data store: 1, 45
Info hbase_batch_sender::send_errors_batch Retrying with errors store occupancy: 1/1
Warning hbase_sender::send exception encountered: unsuccessful items, bytes: 1, 45
Info hbase_batch_sender::send_errors_batch Retrying with errors store occupancy: 1/1
Warning hbase_sender::send exception encountered: unsuccessful items, bytes: 1, 45

===== Thread 0: END OF ITERATIONS =====

=====
close thread idx = 0
Overall batch_sender_statistics = {sender_stats = {requested = {2, 90}, attempted = {4
4, 180}}, store occupancy = {1/1}, error store occupancy = {1/1}}
Stats:
Nb row: 2
Vol (Bytes): 90
Input data generation + Interface Duration (s): 1.10846
Interface only Duration(s): 1.10824
exfo@serspvl06:~$
```

Figura 5. Ejecucion exitosa del Hbase Sender

También podemos comprobar que en el servidor tenemos datos. Para ello tenemos que abrir el hbase Shell del servidor y ejecutar los siguientes comandos:

count 'data_table' Que nos devolverá el número de filas (rows) insertadas

scan 'data_table' Que nos mostrara el contenido de la tabla

```
hbase(main):004:0> count 'data_table'
2 row(s)
Took 0.1578 seconds
=> 2
hbase(main):005:0> scan 'data_table'
ROW                                COLUMN+CELL
 ]]\xFA3fe1.dg7f0_0_1681692777_fe column=C:f, timestamp=1713980
 1.dg7f0_0_1681692777_f3Hs         9\x86\xDD`\x00\x00\x00\x00q\x
                                   x11\x11\x00
 \xE2N\x89\x1Efe1.dg7f0_0_1804289 column=C:f, timestamp=1713980
 383_fe1.dg7f0_0_1804289383_2{#\x 9\x86\xDD`\x00\x00\x00\x00q\x
 C6                                x11\x11\x00
2 row(s)
```

Figura 6. Contenido de la tabla del servidor

Con esto podemos comprobar que la aplicación está desplegada y funciona correctamente.

4.3 Definición y automatización de casos de test

El proceso de validación y testeo no solo se trata de detectar errores o defectos en el software, sino también de garantizar que cumpla con los requisitos funcionales y no funcionales establecidos durante el proceso de desarrollo. Este proceso no sólo proporciona una medida de la calidad del software, sino que también ayuda a identificar posibles mejoras y optimizaciones que pueden ser implementadas antes de la entrega final al usuario.

Además, el testeo adecuado de una aplicación no solo implica la verificación de su correcto funcionamiento en condiciones normales, sino también la evaluación de su comportamiento bajo condiciones extremas y escenarios inesperados.

Teniendo en cuenta esta introducción sobre lo que representa un proceso de validación, y conociendo la aplicación que vamos a testear, en este capítulo diseñaremos diferentes casos de test para asegurar que el comportamiento de nuestra aplicación es el esperado, tanto cuando se utiliza correctamente, como incorrectamente.

Posteriormente estos casos de test que diseñemos serán automatizados.

4.3.1 Definición de casos de test

Estos son los pasos que debemos seguir a la hora de definir una batería de casos de test:

1. **Entender los Requisitos**
2. **Identificar Escenarios de Uso**
3. **Descomposición en Casos de Test**
4. **Definir Entradas y Salidas Esperadas**
5. **Considerar Casos de Test Negativos**

Cuando definimos un caso de test, lo hacemos con el objetivo de validar una funcionalidad del producto, ya sea una nueva funcionalidad, o un bug que ha sido corregido.

En los capítulos anteriores hemos explicado en que consiste nuestra aplicación, con un ejemplo práctico de uso.

Vamos a intentar definir 4 casos de test, incluyendo escenarios exitosos y también casos de test negativos (no exitosos), ya que también es importante que la aplicación se comporte como esperamos en casos no exitosos para evitar situaciones indeseadas ,como por ejemplo una caída de la aplicación.

Empezaremos planteando los escenarios a validar, los cuales desarrollaremos posteriormente.

Primer caso: Envío exitoso de datos a servidor Hbase (básicamente será recrear el ejemplo mostrado en el despliegue de la aplicación)

Segundo caso: Capacidad de modificar la cantidad de datos enviados (en este caso, analizaremos que los parámetros a la hora de elegir la cantidad de datos a enviar se están aplicando correctamente)

Tercer caso: envío de datos a un servidor de Hbase incorrecto (Validaremos que el comportamiento de la aplicación cuando envía a un servidor con una IP incorrecta es el esperado)

Cuarto caso: envío de datos a una tabla inexistente (Validaremos el comportamiento cuando se envía a un Servidor existente, pero la tabla no existe)

Teniendo estos 4 escenarios, nos dispondremos a redactar estos casos, paso por paso, intentando que sean de fácil comprensión, ya que buscamos que estos sean accesibles para miembros de otros equipos, además de futuros miembros del proyecto.

4.3.2 Casos de test

CASO 1. envío exitoso de datos a servidor Hbase

Prerrequisitos: Tener la aplicación desplegada (Sender y Servidor) y tabla creada en el servidor (data_table)

Comprobar la cantidad de datos en el servidor de Hbase *Scan 'data_table'*	El comando devuelve los datos dentro de la tabla
Obtener la ip del servidor *Docker inspect <name_server>*	Ip devuelta correctamente
Ejecutar Sender e insertar datos en el servidor: *docker exec -w /home hbaseClient ./HbaseClientTest 1 2 1 45 45 0 172.17.0.4 data_table overall-stats continue-on-exception=no debug fixed=1,1 size=1 reconnect-on-fatal-ex=no'*	Obtenemos respuesta de la aplicación, indicando que el envío se ha realizado correctamente
Comprobar que los datos se han insertado en el servidor *Scan 'data_table'*	El comando devuelve los datos dentro de la tabla y podemos ver que hay nuevos datos en la tabla

CASO 2. Capacidad de modificar la cantidad de datos enviados

Prerrequisitos: Tener la aplicación desplegada (Sender y Servidor) y tabla creada en el servidor (data_table)

Comprobar la cantidad de datos en el servidor de Hbase *Scan 'data_table'*	El comando devuelve los datos dentro de la tabla
Obtener la ip del servidor *Docker inspect <name_server>*	Ip devuelta correctamente
Ejecutar Sender e insertar datos en el servidor (insertando 2 filas): *docker exec -w /home hbaseClient ./HbaseClientTest 1 1 2 45 45 0 172.17.0.4 data_table overall-stats continue-on-exception=no debug fixed=1,1 size=1 reconnect-on-fatal-ex=no'*	Obtenemos respuesta de la aplicación, indicando que el envío se ha realizado correctamente
Comprobar que los datos se han insertado en el servidor *Scan 'data_table'*	El comando devuelve los datos dentro de la tabla y podemos ver que hay 2 nuevas filas en la tabla
Ejecutar Sender e insertar datos en el servidor (insertando 10 filas): *docker exec -w /home hbaseClient ./HbaseClientTest 1 1 10 45 45 0 172.17.0.4 data_table overall-stats continue-on-*	Obtenemos respuesta de la aplicación, indicando que el envío se ha realizado correctamente

exception=no debug fixed=1,1 size=1 reconnect-on-fatal-ex=no'	
Comprobar que los datos se han insertado en el servidor *Scan 'data_table'*	El comando devuelve los datos dentro de la tabla y podemos ver que hay 10 nuevas filas en la tabla
Ejecutar Sender e insertar datos en el servidor (insertando 2 rafagas de 6 filas): *docker exec -w /home hbaseClient ./HbaseClientTest 1 2 6 45 45 0 172.17.0.4 data_table overall-stats continue-on-exception=no debug fixed=1,1 size=1 reconnect-on-fatal-ex=no'*	Obtenemos respuesta de la aplicación, indicando que el envío se ha realizado correctamente
Comprobar que los datos se han insertado en el servidor *Scan 'data_table'*	El comando devuelve los datos dentro de la tabla y podemos ver que hay 12 nuevas filas en la tabla

CASO 3. Envió de datos a un servidor de Hbase incorrecto

Prerrequisitos: Tener la aplicación desplegada (Sender)

Ejecutar Sender e insertar datos en el servidor utilizando una IP incorrecta *docker exec -w /home hbaseClient ./HbaseClientTest 1 1 2 45 45 0 172.17.0.110 data_table overall-stats continue-on-exception=no debug fixed=1,1 size=1 reconnect-on-fatal-ex=no'*	Obtenemos respuesta de la aplicación , indicando que no existe un servidor de hbase en la ip utilizada en la petición *Data generation completed* *Thread 0: Exception occurred: TTransportException: Type: NOT_OPEN: What: connect() failed: No route to host: iostream error* *Overall batch_sender_statistics = {sender_stats = {requested = {0, 0}, attempted = {0, 0}, successful = {0, 0}, unsuccessful = {0, 0}}, store occupancy = {0/0}, error store occupancy = {0/0}}*

CASO 4. Envío de datos a una tabla inexistente

Prerrequisitos: Tener la aplicación desplegada (Sender y Servidor)

Obtener la ip del servidor *Docker inspect <name_server>*	Ip devuelta correctamente
Ejecutar Sender e insertar datos en el servidor utilizando una tabla inexistente: *docker exec -w /home hbaseClient ./HbaseClientTest 1 2 1 45 45 0 172.17.0.4 hbase_wrong overall-stats continue-on-exception=no debug fixed=1,1 size=1 reconnect-on-fatal-ex=no'*	Obtenemos respuesta de la aplicación, indicando que el envío se ha realizado incorrectamente: *Overall batch_sender_statistics = {sender_stats = {requested = {2, 512}, attempted = {2, 512}, successful = {0, 0}, unsuccessful = {2, 512}}*

4.3.3 Automatización del despliegue

Después de tener los 4 casos de test definidos vamos a proceder a la automatización de estos, ya que el objetivo de este proyecto es conseguir tener una batería de test que se puedan lanzar automáticamente, para así poder evitar, como hemos explicado en los capítulos anteriores, las desventajas de un trabajo manual, que deberán repetirse con cierta frecuencia.

Estos casos de test se van a lanzar desde nuestra maquina Windows, ya que nuestros test se conectarán por SSH a la VM en la que tendremos desplegada la aplicación.

Para la automatización utilizaremos *Robotframework* , herramienta que ya hemos explicado en capítulos anteriores, y la cual nos permitirá realizar todo lo que necesitamos para la automatización de tanto el despliegue de la aplicación , como lanzar posteriormente los casos de test sobre este despliegue.

Antes de empezar a escribir el código, tenemos que definir la estructura que vamos a seguir. Una característica que ofrece robotFramework, es que nos permite crear diferentes funciones (keywords) las cuales podremos utilizar recurrentemente en diferentes casos de test, lo cual nos ayudara a que todo nuestro código sea más fácil de entender y organizado.

Teniendo esto en cuenta la estructura que seguiremos de carpetas será la siguiente:

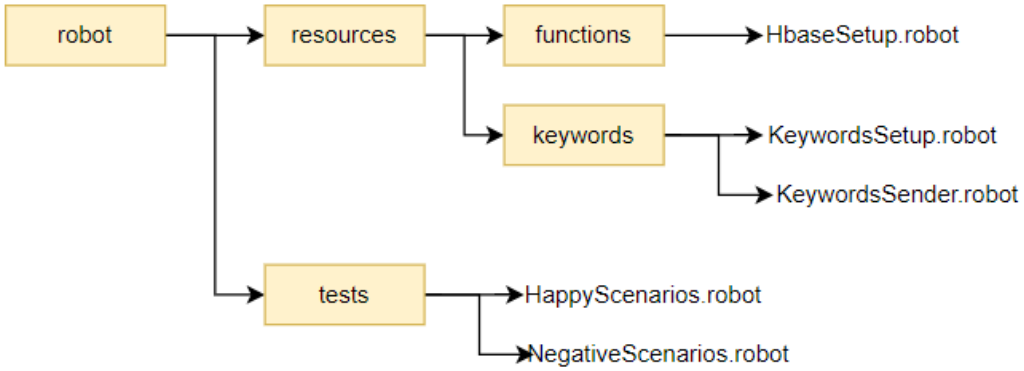


Figura 7. Diagrama de archivos para la automatización de los casos de test

En la cual podemos apreciar que separaremos los casos de test a correr, del resto de archivos, para poder diferenciar claramente que es lo que se va a ejecutar.

Hemos decidido seguir esta estructura, ya que nos permitirá poder escalar la cantidad de pruebas a medida que se implementen funcionalidades a la aplicación, a parte de que es una forma bastante intuitiva de ordenar los diferentes archivos, cosa que permitirá a un nuevo integrante del equipo entender el código.

Empezaremos definiendo los dos archivos relacionados con el despliegue de la aplicación,

- En el archivo **HbaseSetup.robot**, se define la funcion que realizara el propio despliegue, utilizando las Keywords definidas en el siguiente archivo
- En el archivo **KeywordsSetup.robot** se definirán keywords/funciones para utilizarlas en el HbaseSetup.robot, estas serán definidas ya sea porque pensamos que pueden ser utilizadas múltiples veces, o por simple limpieza de código.

A continuación, se describirá el proceso, paso a paso de la definición del código, indicando el motivo por el cual se realizan dichas acciones/procesos.

Ya que empezamos desde zero, el procedimiento que seguiremos será ir añadiendo código en el archivo **HbaseSetup.robot y KeywordsSetup.robot** simultáneamente. Podríamos empezar definiendo las Keywords para posteriormente utilizarlas en el archivo principal, pero considero que de la otra forma, nos permite optimizar más el código y no crear nada innecesario.

El primer paso es el mismo que en todos los archivos .robot

```
*** Settings ***
Documentation      Robot Framework test script
Library     SSHLibrary     timeout=80 seconds
Library     String

Resource     ${EXECDIR}/robot/resources/Keywords/keywordsSetup.robot

*** Variables ***
${alias}                remote_host_1

*** Test Cases ***
```

Código 1. Primeras líneas del archivo HbaseSetup.robot

Primero empezamos definiendo las primeras líneas, que son necesarias en cualquier archivo .robot

Documentation es meramente informativo.

Importaremos las dos librerías: SSHLibrary y String, que nos permitirán utilizar algunas funciones que no vienen por defecto en robotframework

También importaremos como Resource, nuestro archivo keywordSetup.robot, que nos permitirá utilizar las Keywords que definamos

La única diferencia entre *Library y Resource* es que unas son librerías externas, y las otras son nuestras propias librerías.

También en el apartado variables, podemos definir las variables que vayamos a utilizar a lo largo de los casos de test, en este caso, solo necesitamos definir el alias de nuestra máquina.

A continuación, empieza el caso de test/función, El cual requiere de un título y el apartado [Tags]. donde definimos el tag/etiqueta la cual nos servirá para poder elegir que casos de test se ejecutaran en nuestra batería. Mas adelante se explicara como utilizarlas en el comando de ejecución de robot.

También definiremos la conexión a la máquina virtual donde se realizará el despliegue. Para esto utilizaremos la librería SSHLibrary:

```
Deploy containers
    [Tags]    Setup
    [Documentation]    The goal of this function is to deploy the containers needed for the test execution
    Open Connection     ${host}        alias=${alias}
    Login               ${username}    ${password}    delay=1
```

Código 2. Inicio de la función para desplegar los contenedores de Cliente y servidor

Como podemos ver, utilizamos las variables ${host} ${username} ${password} que no hemos definido. Estas las definiremos cuando ejecutemos el comando, ya que estas pueden variar en caso de que queramos correr los test en otra máquina, esto lo hemos hecho así para garantizar que podremos lanzar los casos de prueba en cualquier entorno que tenga las dependencias instaladas necesarias, a través de la configuración.

Después de la conexión con la VM, empezaremos desplegando el cliente, como hemos explicado en los apartados anteriores.

```
log    == Step 1: Create Client ==
${reference}=  Set Variable    registry.gitlab.com/exfo/products/asa/msu/sdp/sender/hbase-sender:hbase-client
${image}=    Get Image Id from name or tag    ${reference}
${client_name}=    Add sufix to String    string=RobotClient    sufix=${PIPELINE_ID}
${id_container}=    Create Client and return id    image=${image}    name=${client_name}
```

Código 3. Despliegue del contenedor del cliente

Podemos ver, que empezamos creando la variable ${reference} la cual es el path de la imagen que utilizaremos para desplegar el cliente.

Posteriormente, guardaremos en la variable ${image} el id de esta imagen (también podríamos utilizar directamente el nombre de la imagen en vez del id). Para este paso crearemos una keyword, ya que este paso se repetirá más adelante (y en posibles futuros trabajos).

Vamos a definir la primera Keyword, por lo que crearemos el archivo **KeywordsSetup.robot** de una forma similar al anterior:

```
** Settings ***
Documentation    File that contains all the keywords for Hbase requests

Library    SSHLibrary    timeout=100 seconds
Library    String
Library    Collections

*** Keywords ***
```

Código 4. Inicio del archivo KeywordsSetup.robot

Con esto ya podemos definir la primera Keyword:

```
Get Image Id from name or tag
    [Arguments]    ${reference}
    ${id_image}=    Execute Command    command=docker image ls -f "reference=${reference}" -q    sudo=True    sudo_password=${password}
    [Return]    ${id_image}
```

Código 5. Keyword para obtener el id de una imagen de Docker

También crearemos la variable ${client_name}, con la cual añadiremos al nombre base RobotClient un sufijo, que será el valor de la variable ${PIPELINE_ID}, que se definirá también en el comando de ejecución (Ejemplo: RobotClient_123), esto nos permitirá poder tener ejecuciones en paralelo, ya que los contenedores tendrán un nombre independiente (podríamos tener en el mismo entorno dos clientes y dos servidores).

Add Sufix to String es una Keyword que hemos definido nosotros:

```
Add sufix to String
    [Arguments]    ${string}    ${sufix}=${EMPTY}
    [Documentation]    The goal of this KW is to modify a string adding more characters to the end of it
    ${ret}=    Evaluate    "${string}"+"_"+"${sufix}"
    [Return]    ${ret}
```

Código 6. Keyword add Sufix to String

Con todo esto, ahora nos queda crear el cliente. Para esto utilizaremos una Keyword, que lo que hará es crear el cliente, y guardara en la variable ${id_container} su id, que podremos utilizar posteriormente.

Esta keyword, al ser de una complejidad más elevada , se desgranaremos en diferentes funciones como veremos a continuación.

```
Create Client and return id
    [Arguments]    ${image}    ${name}
    ${id_container}=    Create Container from image    image=${image}    name=${name}
    Copy executable form VM to Client    ${id_container}
    [Return]    ${id_container}
```

Código 7. Keyword para crear el contenedor del cliente

Para crear el cliente, primero necesitaremos desplegar un contenedor con la imagen de Docker correspondiente, y luego mover el ejecutable que se encuentra en la VM, dentro del cliente

```
34  Create Container from image
35      [Arguments]    ${image}    ${name}
36      ${id_container}=    Execute Command    command=docker run -d --name ${name} -it ${image} /bin/bash
        sudo=True    sudo_password=${password}
37      [Return]    ${id_container}
38
39  Copy executable form VM to Client
40      [Arguments]    ${id_client}
41      [Documentation]    This keyword is used to move the executable we got created on the previous job of the
        pipeline as an artifact
42      ...                to the client container where we are going to run the application
43      ${rc}    Execute Command    command=docker cp HbaseClientTest ${id_client}:/home/HbaseClientTest
        sudo=True    sudo_password=${password}    return_stdout=False    return_rc=True
44      Should be equal as integers    ${rc}    0    msg=${rc} =! 0 'command not successful
```

Código 8. Keywords utilizadas en el proceso para crear el contenedor del cliente

Podemos ver que lo que se realiza en estas dos funciones es ejecutar en la VM, los comandos explicados en el capítulo del despliegue

En la última línea comprobamos que el ${rc} (return code) es igual a 0, cosa que significa que el comando ha sido realizado correctamente, en el caso de que este no fuera 0, la ejecución pararía y tendríamos que ver cuál es el problema a través de los logs.

Es interesante añadir este tipo de checkeos en los casos automáticos, ya que nos permiten ver donde ha fallado la ejecución, cosa que nos permite encontrar el error rápidamente

Cabe destacar, que esta separación en diferentes Keywords no es necesaria, ya que podríamos poner el contenido de estas en lugar de donde son llamadas en el código principal, pero nos permite poder entender lo que hace un Caso de Test/función de una forma más intuitiva.

Con esto, ya tendríamos el despliegue del cliente listo, ahora procederemos con el despliegue del servidor:

```
log    == Step 2: Create Server ==
${server_name}=    Add sufix to String    string=RobotServer    sufix=${PIPELINE_ID}
${server_id}=    Create Server and return id    ${server_name}
${server_ip}=    Get ip from container id    ${server_id}
```

Código 9. Despliegue del servidor

Como podemos apreciar, para declarar el nombre del servidor, utilizamos la misma función que en el cliente. Al haber creado esta Keyword en nuestro archivo keywordsSetup.robot, la podemos reutilizar sin problemas.

Para crear el servidor, hemos creado una nueva Keyword, en la cual borramos el directorio donde se almacena el contenido del servidor (en caso de que existiera previamente) para crearlo de nuevo, a continuación, ejecutamos el comando necesario para desplegar el servidor:

```
63  Create Server and return id
64      [Arguments]    ${name}
65      Execute Command    command=rm -r TFGdata   sudo=True    sudo_password=${password}
66      Execute Command    command=mkdir TFGdata
67      Write    command=id=$(docker run --name=${name} -h hbase-docker -d -v $PWD/TFGdata:/data registry.gitlab.com/
        exfo/products/asa/msu/sdp/sender/hbase-sender:hbase-server)
68      ${server_id}=    Get id from container name    ${name}
69      [Return]    ${server_id}
```

Código 10. Keyword principal que desplegara el servidor

Posteriormente, obtendremos la ip de nuestro servidor con la siguiente Keyword:

```
71  Get ip from container id
72      [Arguments]    ${container_id}
73      ${container_ip}=    Execute Command    command=docker inspect --format '{{ .NetworkSettings.IPAddress }}' $
        {container_id}    sudo=True    sudo_password=${password}
74      [Return]    ${container_ip}
```

Código 11. Keyword que devuelve la Ip de un contenedor

Con esto ya tendríamos completa la automatización de la parte del despliegue del servidor, finalizaremos con la parte de Hbase, en la cual crearemos una tabla dentro del servidor.

```
log    == Step 3: Create HBase Table ==
Open Hbase Shell    ${server_id}
${table_name}    Set Variable    hbase_client
Create Hbase Table    ${table_name}
Exit from Hbase shell
```

Código 12. Creación de la tabla de Hbase en el servidor

Hemos utilizado 3 nuevas Keywords, estas sirven para Abrir el Hbase Shell. Crear la tabla, y salir de Hbase Shell, respectivamente:

```
Open Hbase shell
    [Arguments]    ${container_id}
    Write    docker exec -it ${container_id} hbase shell
    ${stdout}=    Read until    expected=hbase(main)

Create Hbase Table
    [Arguments]    ${name}
    Write    create '${name}','C'
    ${stdout}=    Read until    expected=Created table ${name}

Exit from Hbase shell
    Write    exit
    ${stdout}=    Read until    expected=${username}
```

Código 13. Defínción de 3 keywords relacionadas con acciones en Hbase Shell

Con todo esto, ya tendríamos lista la automatización de nuestro despliegue. A continuación vamos a comprobar que, efectivamente, nuestra función despliega cliente y servidor.

Para ello simplemente tendremos que ejecutar el siguiente comando, en la carpeta en la que tengamos nuestro código.

```
robot -v host:192.168.175.124 -v username:exfo -v password:PasswOrdastellia -v
PIPELINE_ID:123  --outputdir  robot/robot-results  --xunit  xunit.xml  -i  Setup
robot/resources/Functions
```

Vamos a explicar las diferentes partes de este comando:

- Primero empezamos con robot, que es lo que nos permite realizar ejecuciones con robotframework

- La flag – v nos permite declarar variables, por lo que las variables ${host} ${username} ${password} ${PIPELINE_ID} mencionadas anteriormente, se definen aquí. Esto nos permite flexibilidad en caso de querer utilizar otra VM, otro usuario etc.
- La flag –outputdir y – xunit nos permiten indicar donde se almacenaran los logs de la ejecución
- La flag -i nos sirve para indicar que Tag queremos que se ejecute, en caso de querer ejecutar todo, podríamos no incluir esta flag, y se ejecutarían todos los casos en la dirección que indiquemos
- Para acabar, incluiremos el path donde se encuentran nuestro caso de test (despliegue en este caso)

Si todo funciona correctamente, obtendremos la siguiente respuesta al ejecutar el comando:

```
anggri1@lvl0025 MINGW64 /c/gitlab/SDP/TFG-ANGEL/hbase-sender (TFG-Angel)
$ robot -v host:192.168.175.124 -v username:exfo -v password:Passw0rdastellia -v
 PIPELINE_ID:123 --outputdir robot/robot-results --xunit xunit.xml -i Setup robo
t/resources/Functions
==============================================================================
Functions
==============================================================================
Functions.HbaseSetup :: Robot Framework test script
==============================================================================
Deploy containers :: The goal of this function is to deploy the co... | PASS |
------------------------------------------------------------------------------
Functions.HbaseSetup :: Robot Framework test script                   | PASS |
1 test, 1 passed, 0 failed
==============================================================================
Functions                                                             | PASS |
1 test, 1 passed, 0 failed
==============================================================================
Output:  C:\gitlab\SDP\TFG-ANGEL\hbase-sender\robot\robot-results\output.xml
XUnit:   C:\gitlab\SDP\TFG-ANGEL\hbase-sender\robot\robot-results\xunit.xml
Log:     C:\gitlab\SDP\TFG-ANGEL\hbase-sender\robot\robot-results\log.html
Report:  C:\gitlab\SDP\TFG-ANGEL\hbase-sender\robot\robot-results\report.html
```

Figura 8. Ejecución del despliegue automatizado con robot

después de la ejecución, si accedemos a nuestra VM, deberíamos encontrar nuestros contenedores desplegados, y preparados para funcionar:

```
9c64264lafe7   registry.gitlab.com/exfo/p
cp   RobotServer_123
96f8ee09edlc   80a050e53253
     RobotClient_123
```

Figura 9. Cliente y servidor desplegados en la máquina virtual

También podremos ver como se ha generado una carpeta llamada robot-results, donde podremos encontrar los logs de la ejecución:

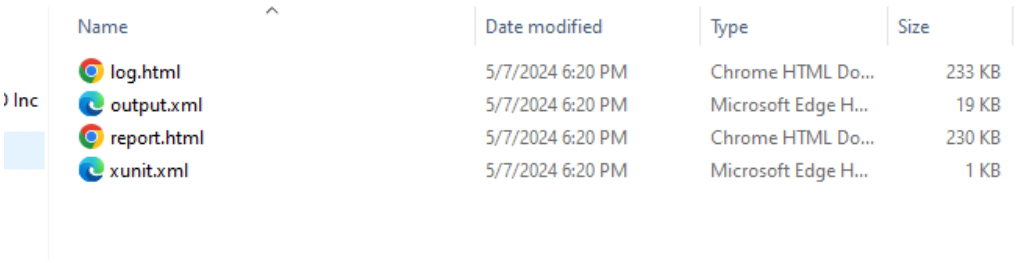
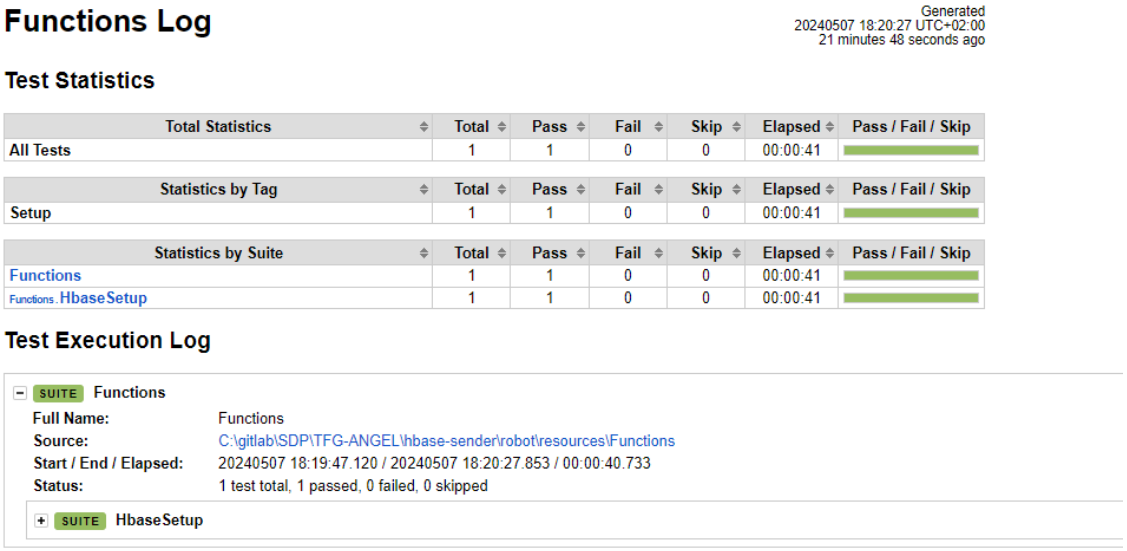


Figura 10. archivos generados tras la ejecución del caso en Robot

En caso de que esta ejecución fallara, podríamos ver en los logs generados, en que paso está fallando e intentar solucionarlo, esto lo analizaremos en el siguiente capitulo.



Functions Log

Generated
20240507 18:20:27 UTC+02:00
21 minutes 48 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	1	1	0	0	00:00:41	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Setup	1	1	0	0	00:00:41	

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Functions	1	1	0	0	00:00:41	
Functions.HbaseSetup	1	1	0	0	00:00:41	

Test Execution Log

SUITE Functions

Full Name:	Functions
Source:	C:\gitlab\SDP\TFG-ANGEL\hbase-sender\robot\resources\Functions
Start / End / Elapsed:	20240507 18:19:47.120 / 20240507 18:20:27.853 / 00:00:40.733
Status:	1 test total, 1 passed, 0 failed, 0 skipped

SUITE HbaseSetup

Figura 11. Ejemplo de logs generados tras una ejecución

4.3.4 Automatización de los casos de test

Teniendo el despliegue de la aplicación automatizado, procederemos a la automatización de los casos de test.

Como se puede apreciar en la Figura 7, crearemos una carpeta llamada Tests, la cual contendrá dos archivos .robot (**HappyScenarios.robot y NegativeScenarios.robot**).

Esto nos será útil para poder ordenar nuestros casos de prueba en base a la funcionalidad que prueban (en este caso escenarios exitosos y no exitosos), que nos permitirá poder añadir nuevas funcionalidades en nuestra batería de tests.

También crearemos un nuevo archivo en la carpeta Keywords, llamado KeywordsSender.robot

Además de la organización, tener el código en diferentes archivos nos permitirá poder ejecutar, por ejemplo, los casos de test relacionados solo con una funcionalidad, sin tener que ejecutar el resto de casos.

Para la definición de estos archivos, seguiremos el mismo procedimiento, en un archivo definiremos el caso de test, y en otro definiremos las diferentes keywords a medida que las vayamos utilizando.

Vamos a empezar creando el archivo HappyScenarios.robot, donde se encontrarán los dos primeros casos, y el archvo KeywordsSender.robot :

```
*** Settings ***
Documentation      Robot Framework test script
Library    SSHLibrary    timeout=100 seconds
Library    String

Resource    ${EXECDIR}/robot/resources/Keywords/keywordsSetup.robot
Resource    ${EXECDIR}/robot/resources/Keywords/keywordsSender.robot

*** Variables ***
${alias}              remote_host_1
${server_name}    RobotServer_${PIPELINE_ID}
${client_name}=    RobotClient_${PIPELINE_ID}
${table_name}    hbase_client
```

Código 14. creación del archivo HappyScenario.robot

```
** Settings ***
Documentation    File that contains all the keywords for Hbase requests
Resource    ${EXECDIR}/submodule/test-automation/resources/Keywords/keywordsToolbox.robot

Library    SSHLibrary    timeout=100 seconds
Library    String
Library    Collections
```

Código 15. Creación del archivo KeywordsSender.robot

Como vemos, es similar a lo realizado en el despliegue, lo único que en este caso también importaremos las keywords de KeywordsSender.robot, que son las que implementaremos para estos casos.

También en el apartado variables, definiremos el nombre el cliente y servidor, para poder reutilizarlos en las diferentes ejecuciones.

Con esto claro, vamos a pasar al primer caso de test, que consiste en una ejecución sencilla de la aplicación.

Empezaremos definiendo el nombre del Test, las Tags, y los parámetros que se utilizaran posteriormente para ejecutar la aplicación:

```
*** Test Cases ***

Basic Test
    [Tags]    Pipeline    T1
    ${nb_threads}    Set Variable    1
    ${nb_burst}    Set Variable    1
    ${nb_rows_burst}    Set Variable    1
    ${nb_row_bytes_min}    Set Variable    45
    ${nb_row_bytes_max}    Set Variable    45
    ${sleep}    Set Variable    0
    ${stats}    Set Variable    overall-stats
    ${continue_on_exception}    Set Variable    continue-on-exception=no
    ${log_level}    Set Variable    debug
    ${memory_selection}    Set Variable    dynamic=1,1
    ${trigger}    Set Variable    size=1
    ${reconnect_on_fatal_ex}    Set Variable    reconnect-on-fatal-ex=no
```

Código 16. Definición de variables en el primer caso de test

Las tags utilizadas son: Pipeline (esta tag la utilizaremos para correr todos los casos de test, por lo que será añadida de la misma forma en los siguientes casos, y T1, para poder ejecutar este test independientemente.

Luego las variables utilizadas son las mismas que indicamos en la definición del caso.

A continuación, nos conectaremos a la VM y guardaremos en variables, los ids del cliente y servidor, así como la ip del servidor.

```
log    == Login to Common Environment ==
Open Connection     ${host}        alias=${alias}
Login               ${username}    ${password}    delay=1
log    == Step 1: Get Server and client Data ==
${server_id}=    Get id from container name    container=${server_name}
${client_id}=    Get id from container name    container=${client_name}
${server_ip}=    Get ip from container id    ${server_id}
```

Código 17. conexión con la VM y obtener identificadores de cliente y servidor

Podemos apreciar que todas estas Keywords son las que hemos creado anteriormente, por lo que podemos ejecutarlas de nuevo sin problema, siempre que las hayamos importado.

Este es uno de los motivos por los cuales nos decantamos por seguir esta estrategia y no otra , que funcionalmente nos llevaría al mismo resultado, pero de una forma mucho menos eficiente.

El siguiente paso es comprobar los datos que contiene nuestro servidor, para poder comparar el antes y el después de la inserción de datos.

```
log    == Step 2: Get rows before running aplication ==
${pre_rows}    Return Rows from a table    ${server_id}    ${table_name}
```

Código 18. obtención de las filas en Hbase previas a la ejecución del caso

La keyword utilizada básicamente lo que hace es abrir el Hbase Shell, un count de la tabla que incluyamos como argumento, y cerrar el Shell:

```
Return Rows from a table
    [Arguments]    ${server_id}    ${table_name}
    Open Hbase shell  ${server_id}
    write    count '${table_name}'
    ${stdout}=    Read until    expected=row(s)
    ${rows}    Get lines containing string    ${stdout}    row(s)
    ${rows}    remove string    ${rows}    row(s)
    Exit from Hbase shell
    RETURN    ${rows}
```

Código 19. Keyword para obtener el número de columnas de una tabla de Hbase

Las dos Keywords *Get lines containing string y remove string* son Keywords de la libreria String, que nos permiten quedarnos solo con el número de filas de toda la respuesta.

Teniendo esto, procederemos a la ejecución de la aplicación.

```
42      ${response}=    Run Application    id_container=${client_id}    path=/home    nb_threads=${nb_threads}
        nb_burst=${nb_burst}    nb_rows_burst=${nb_rows_burst}    nb_row_bytes_min=${nb_row_bytes_min}
        nb_row_bytes_max=${nb_row_bytes_max}    sleep=${sleep}    server_ip=${server_ip}    table_name=${table_name}
        stats=${stats}    continue_on_exception=${continue_on_exception}    log_level=${log_level}    memory_selection=$
        {memory_selection}    trigger=${trigger}    reconnect_on_fatal_ex=${reconnect_on_fatal_ex}
43      Check logs    output=${response}    type=Info
```

Código 20. ejecución de la aplicación en el Test 1

Podemos apreciar como la Keyword *Run Application* tiene muchos argumentos, eso es necesario ya que nuestra aplicación requiere de ellos.

Aquí está la definición de la keyword:

```
11  Run Application
12      [Arguments]    ${id_container}    ${path}    ${nb_threads}    ${nb_burst}    ${nb_rows_burst}    $
        {nb_row_bytes_min}    ${nb_row_bytes_max}    ${sleep}    ${server_ip}    ${table_name}    ${stats}    $
        {continue_on_exception}    ${log_level}    ${memory_selection}    ${trigger}    ${reconnect_on_fatal_ex}    $
        {successful}=True
13      [Documentation]    Inject Data to a specific container
14      ${parameters}=    Set Variable    ${nb_threads} ${nb_burst} ${nb_rows_burst} ${nb_row_bytes_min} $
        {nb_row_bytes_max} ${sleep} ${server_ip} ${table_name} ${stats} ${continue_on_exception} ${log_level} $
        {memory_selection} ${trigger} ${reconnect_on_fatal_ex}
15      ${response}=    Execute Command    command=docker exec -w ${path} ${id_container} ./HbaseClientTest $
        {parameters}    sudo=True    sudo_password=${password}
16      IF    '${successful}' == 'True'
17          ${check_execution}=    Does String Contains    ${response}    close thread idx = 0
18          Should be true    ${check_execution}    msg=Application Run failed
19      END
20      [Return]    ${response}
```

Código 21. definición de la keyword utilizada para ejecutar la aplicación, con todos sus argumentos

Primero, juntamos todos los argumentos en una variable que llamaremos *${parameters}* y ejecutaremos el comando tal y como está definido en los casos de test.

Después de esto, comprobaremos que la respuesta contiene *"close thread idx = 0"*, esto nos indicara que la ejecución ha finalizado correctamente, este chequeo provocara que cualquier ejecución errónea, provoque un error en el test.

Para finalizar esta Keyword devolverá la respuesta, ya que la analizaremos en los siguientes pasos de este caso de test. La almacenaremos en la variable ${response}

También hemos utilizado la Keyword *Check logs*, que nos permite checkear si estos se encuentran en la respuesta.

Ejemplo de logs:

```
Info hbase_batch_sender::send(span) Appended to data store: 10, 450
Info hbase_sender::send Created batch mutations from data: 10
Info hbase_sender::send Successfully Data sent: 10, 450
```

Figura 12. Logs generados por la aplicación

```
Check logs
    [Arguments]    ${output}    ${type}
    [Documentation]    This KW checks different type of logs are displayed
    ...                ${type} can be: Debug,Info,Error,Warning or no_log
    IF    '${type}' == 'no_log'
        ${check}    Does String contains    str=${output}    value=Info hbase_batch_sender
        should not be true    ${check}
        ${check}    Does String contains    str=${output}    value=Error hbase_batch_sender
        should not be true    ${check}
        ${check}    Does String contains    str=${output}    value=Error hbase_sender
        should not be true    ${check}
    ELSE
        ${check}    Does String contains    str=${output}    value=${type} hbase_sender
        should be true    ${check}
    END
```

Código 22. Definición de la keyword para comprobar si los logs son los esperados

Esta función comprobara, si los logs se encuentran o no en la respuesta, ya que esto dependerá como se ejecute la aplicación, para esto utilizamos la Keyword *"Does String Contains"* que nos devolverá un booleano (TRUE Si se cumple y FALSE si no).

```
Does String contains
    [Arguments]    ${str}    ${value}
    [Documentation]    TRUE if ${str} contains at least one occurence of ${value}
    ${lines}=    Get Lines Containing String    ${str}    ${value}
    ${nb_lines}=    get length    ${lines}
    ${does_contain}=    Evaluate    (${nb_lines} > 0)
    [Return]    ${does_contain}
```

Código 23. Keyword para comprobar si un string contiene el string del argumento

En nuestro caso queremos que se compruebe que los logs aparecen, por lo que el argumento type en nuestra prueba será *info.*

Para acabar, tendremos que automatizar la comprobación de que la respuesta de la aplicación es la esperada (si hemos intentado insertar 2 filas, que así sea en la respuesta) y posteriormente, ver que estos datos se encuentran en el servidor.

```
44      log    == Step 4: Check output ==
45      ${inserted_rows}=    Check Data Output    response=${response}    nb_burst=${nb_burst}    nb_rows_burst=$
        {nb_rows_burst}    nb_row_bytes_min=${nb_row_bytes_min}    nb_row_bytes_max=${nb_row_bytes_max}
46      Check Stats Output    response=${response}    min_bytes=${nb_row_bytes_min}    max_bytes=$
        {nb_row_bytes_max}    requested_rows=${inserted_rows}    attempted_rows=${inserted_rows}    successful_rows=$
        {inserted_rows}    unsuccessful_rows=0    nb_threads=${nb_threads}    nb_burst=${nb_burst}    stats=${stats}
47      log    == Step 5: Check data is inserted on the server table ==
48      ${expected_rows}    evaluate    ${inserted_rows} + ${pre_rows}
49      Check hbase count is expected    ${table_name}    ${expected_rows}    ${server_id}
50      [Teardown]    Hbase Teardown    ${server_id}    ${table_name}
```

Código 24. 4º paso el T1, donde se valida la respuesta.

Para la primera comprobación, definiremos una nueva Keyword llamada *Check Data Output*, que tendrá como argumentos la respuesta de la aplicación y algunos de los parámetros utilizados, los cuales utilizaremos para comprobar que efectivamente, la respuesta de la aplicación entra dentro de los valores esperados:

```
22  Check Data output
23      [Arguments]    ${response}    ${nb_burst}    ${nb_rows_burst}    ${nb_row_bytes_min}    ${nb_row_bytes_max}
24      ${expected_rows}=    Evaluate    ${nb_burst} * ${nb_rows_burst}
25      ${min_bytes}=    Evaluate    ${expected_rows}*${nb_row_bytes_min}
26      ${max_bytes}=    Evaluate    ${expected_rows}*${nb_row_bytes_max}
27      ${rows_inserted}=    Check Sent Data    response=${response}    expected_rows=${expected_rows}    min_bytes=$
        {min_bytes}    max_bytes=${max_bytes}
28      [Return]    ${rows_inserted}
```

Código 25. Keyword utilizada para comprobar que los datos en el servidor son los esperados.

Podemos ver como en la primera parte de la función, calculamos los valores esperados, para posteriormente compararlos con los datos de la respuesta, esta segunda tarea la realizaremos utilizando la keyword *Check Sent Data*.

```
Check Sent Data
    [Arguments]    ${response}    ${expected_rows}    ${min_bytes}    ${max_bytes}
    [Documentation]    This keyword goal is to check the output of the application
    stats parameter to 'no_stats'
    ...                And Returns the number of rows inserted
    ${lines}=    Get Lines Containing String    ${response}    Nb row:
    ${rows}=    Fetch From Right    ${lines}    :
    ${rows}=    Replace String    ${rows}    ${SPACE}    ${EMPTY}
    Should be true    '${expected_rows}' == '${rows}'
    ${lines}=    Get Lines Containing String    ${response}    Vol
    ${vol_bytes}=    Fetch From Right    ${lines}    :
    ${vol_bytes}=    Replace String    ${vol_bytes}    ${SPACE}    ${EMPTY}
    Should be true    '${min_bytes}' <= '${vol_bytes}'
    Should be true    '${vol_bytes}' <= '${max_bytes}'
    [Return]    ${rows}
```

Código 26. Definición de la keyword Check Sent data

Si alguno de los valores de la respuesta no coincide con lo esperado, el test fallara gracias a las comprobaciones realizadas. Todas estas KW provienen de las librerías importadas (String Library)

Cabe destacar que, esta KW al final nos devuelve el número de filas (rows) insertadas, ya que comprobaremos en el siguiente paso, que coincide con el número de nuevas filas en la tabla de Hbase y efectivamente, se han insertado las filas que la aplicación debe.

Para ello primero calculamos el número de filas esperadas (antes del test + las insertadas con esta ejecución), almacenado en la variable *${expected_rows}*, que será utilizado como argumento en la siguiente keyword.

```
Check hbase count is expected
    [Arguments]    ${table_name}    ${expected_rows}    ${server_id}
    ${rows}    Return Rows from a table    ${server_id}    ${table_name}
    should be true    ${expected_rows} == ${rows}    msg= rows are not the expected
```

Código 27. Keyword que compara el valor obtenido de la tabla con el que se ha introducido como input

Esta Keyword obtiene las filas de Hbase (reutilizando la keyword *Return Rows from a table)* y comparando que este valor y el utilizado como argumento son iguales.

Con esta última comprobación, ya estaría el primer caso de test automatizado, ya que se realizan todas las comprobaciones que definimos en el caso de test 1.

Para poder mejorar este caso, añadiremos un teardown, que es una función que siempre se ejecutara al final del caso de test, sea este exitoso, o falle en algún paso intermedio.

Este teardown lo que hará será limpiar la tabla de Hbase, con la finalidad de dejar el servidor funcional en caso de algún problema en la inserción.

```
Hbase Teardown
    [Arguments]    ${server_id}    ${table_name}
    Open Hbase Shell    ${server_id}
    Reset Hbase Table    ${table_name}
    Exit from Hbase shell
```

Código 28. Teardown, función que se ejecuta al final de un caso de test, independientemente de si pasa o falla

Podemos ver como este último paso consiste en acceder al Hbase Shell, reniciar la tabla (borrarla y volverla a crear, y salir de el mismo Shell.)

```
Reset Hbase Table
    [Arguments]    ${name}    ${sleep}=0
    Delete Hbase Table    ${name}
    Sleep    ${sleep}
    Create Hbase Table    ${name}
```

Código 29. Reset de la tabla de Hbase, para limpiar lo ocurrido en la ejecución anterior

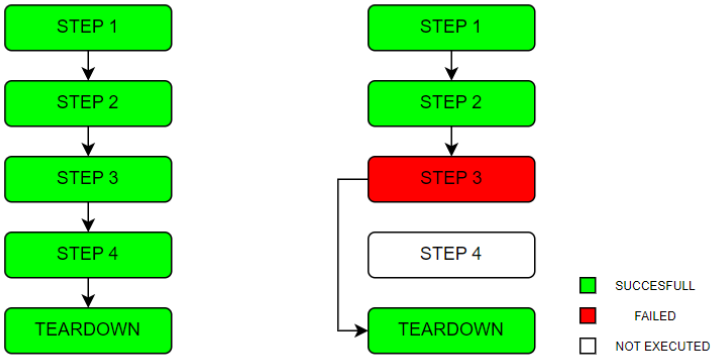


Figura 13. Ejemplo de ejecución del Teardown en casos exitosos y no exitosos

Con esto concluye la automatización del primer caso de prueba.

Para confirmar que el código es correcto, procederemos a ejecutar el caso, como hicimos con el despliegue de la aplicación.

El procedimiento es similar al anterior. El comando que ejecutaremos es el siguiente

```
robot -v host:192.168.175.124 -v username:exfo -v password:PasswOrdastellia -v
PIPELINE_ID:123  --outputdir  robot/robot-results  --xunit  xunit.xml  -i  T1
robot/tests
```

Podemos ver que la única diferencia está en el directorio donde se encuentran los tests (robot/tests) y el tag que utilizaremos, que en este caso será T1.

```
==============================================================================
Basic Test                                                            | PASS |
------------------------------------------------------------------------------
Tests.HappyScenarios :: Robot Framework test script                   | PASS |
1 test, 1 passed, 0 failed
==============================================================================
Tests                                                                 | PASS |
1 test, 1 passed, 0 failed
==============================================================================
```

Figura 14. ejecución exitosa del primer caso de test

Con esto podemos concluir con el primer caso de test. Para los siguientes se seguirá el mismo proceso.

Dado que los siguientes casos tienen pasos similares, no entraremos tanto en detalle en cosas explicadas en este primer caso.

Ahora procederemos a la automatización del segundo caso, en el cual validaremos la capacidad de la aplicación de poder enviar más o menos datos dependiendo de los parámetros introducidos.

Para ello empezaremos de forma similar al caso 1, conectándonos a la VM, definiendo parámetros y obteniendo la información de cliente y servidor.

```
Send different amounts of data
    [Tags]    Pipeline    T2
    ${nb_threads}    Set Variable    1
    ${nb_burst}    Set Variable    1
    ${nb_rows_burst}    Set Variable    2
    ${nb_row_bytes_min}    Set Variable    45
    ${nb_row_bytes_max}    Set Variable    45
    ${sleep}    Set Variable    0
    ${stats}    Set Variable    overall-stats
    ${continue_on_exception}    Set Variable    continue-on-exception=no
    ${log_level}    Set Variable    debug
    ${memory_selection}    Set Variable    dynamic=1,1
    ${trigger}    Set Variable    size=1
    ${reconnect_on_fatal_ex}    Set Variable    reconnect-on-fatal-ex=no
    log    == Login to Common Environment ==
    Open Connection    ${host}    alias=${alias}
    Login    ${username}    ${password}    delay=1
    log    == Step 1: Get Server and client Data ==
    ${server_id}=    Get id from container name    container=${server_name}
    ${client_id}=    Get id from container name    container=${client_name}
    ${server_ip}=    Get ip from container id    ${server_id}
    ${table_name}    Set Variable    hbase_client
```

Código 30. Inicio del 2 caso de test

Se puede apreciar que el inicio de este caso de test es muy similar al anterior, la única diferencia es que definiremos una etiqueta diferente (T2)

El objetivo de este test, tal como se define en el capitulo 4.3.2, es validar que nuestra aplicación es capaz de enviar diferente cantidad de datos en base a los parámetros utilizados, por lo que enviaremos datos en 2 ocasiones, validando la cantidad enviada en cada iteración.

Empezaremos con la primera ejecución de la aplicación, insertando 2 filas y comprobando que estas se encuentran en el servidor.

El código es muy similar al utilizado en el primer caso, por lo que podemos reutilizar las keywords creadas en el primer caso.

```
74      log    == Step 2: Get rows before running aplication ==
75      ${pre_rows}    Return Rows from a table    ${server_id}    ${table_name}
76      log    == Step 3: Run Application ==
77      ${response}=    Run Application    id_container=${client_id}    path=/home    nb_threads=${nb_threads}
        nb_burst=${nb_burst}    nb_rows_burst=${nb_rows_burst}    nb_row_bytes_min=${nb_row_bytes_min}
        nb_row_bytes_max=${nb_row_bytes_max}    sleep=${sleep}    server_ip=${server_ip}    table_name=${table_name}
        stats=${stats}    continue_on_exception=${continue_on_exception}    log_level=${log_level}    memory_selection=$
        {memory_selection}    trigger=${trigger}    reconnect_on_fatal_ex=${reconnect_on_fatal_ex}
78      Check logs    output=${response}    type=Info
79      log    == Step 4: Check output ==
80      ${inserted_rows}=    Check Data Output    response=${response}    nb_burst=${nb_burst}    nb_rows_burst=$
        {nb_rows_burst}    nb_row_bytes_min=${nb_row_bytes_min}    nb_row_bytes_max=${nb_row_bytes_max}
81      Check Stats Output    response=${response}    min_bytes=${nb_row_bytes_min}    max_bytes=$
        {nb_row_bytes_max}    requested_rows=${inserted_rows}    attempted_rows=${inserted_rows}    successful_rows=$
        {inserted_rows}    unsuccessful_rows=0    nb_threads=${nb_threads}    nb_burst=${nb_burst}    stats=${stats}
82      log    == Step 5: Check data is inserted on the server table ==
83      ${expected_rows}    evaluate    ${inserted_rows} + ${pre_rows}
84      Check hbase count is expected    ${table_name}    ${expected_rows}    ${server_id}
85      ${pre_rows}    Set variable    ${expected_rows}
```

Código 31. Inserción de dos filas en la tabla de Hbase

Ahora, siguiendo el caso de test que hemos definido, tenemos que modificar los parámetros, para poder insertar 10 filas en vez de 2.

```
log    == Step 6: Set NB rows to 10 and repeat the prevoius steps ==
${nb_rows_burst}    Set Variable    10
${trigger}    Set Variable    size=10
```

Código 32 Modificar los parámetros de la aplicación, para insertar 10 filas de datos

Después de esta actualización de las variables, volveremos a ejecutar la aplicación, de la misma forma que en los casos anteriores.

```
89      log    == Step 7: Run Application ==
90      ${response}=    Run Application    id_container=${client_id}    path=/home    nb_threads=${nb_threads}
        nb_burst=${nb_burst}    nb_rows_burst=${nb_rows_burst}    nb_row_bytes_min=${nb_row_bytes_min}
        nb_row_bytes_max=${nb_row_bytes_max}    sleep=${sleep}    server_ip=${server_ip}    table_name=${table_name}
        stats=${stats}    continue_on_exception=${continue_on_exception}    log_level=${log_level}    memory_selection=$
        {memory_selection}    trigger=${trigger}    reconnect_on_fatal_ex=${reconnect_on_fatal_ex}
91      Check logs    output=${response}    type=Info
92      log    == Step 8: Check output ==
93      ${inserted_rows}=    Check Data Output    response=${response}    nb_burst=${nb_burst}    nb_rows_burst=$
        {nb_rows_burst}    nb_row_bytes_min=${nb_row_bytes_min}    nb_row_bytes_max=${nb_row_bytes_max}
94      Check Stats Output    response=${response}    min_bytes=${nb_row_bytes_min}    max_bytes=$
        {nb_row_bytes_max}    requested_rows=${inserted_rows}    attempted_rows=${inserted_rows}    successful_rows=$
        {inserted_rows}    unsuccessful_rows=0    nb_threads=${nb_threads}    nb_burst=${nb_burst}    stats=${stats}
95      log    == Step 9: Check data is inserted on the server table ==
96      ${expected_rows}    evaluate    ${inserted_rows} + ${pre_rows}
97      Check hbase count is expected    ${table_name}    ${inserted_rows}    ${server_id}
```

Código 33. inserción de 10 filas de datos en el servidor Hbase

Finalizaremos de la misma forma que en el primer caso, ejecutando el Teardown.

```
[Teardown]    Hbase Teardown    ${server_id}    ${table_name}
```

Código 34. Teardown Hbase

Con esto, concluimos con la automatización del segundo caso de prueba.

Como podemos apreciar, gracias a la definición de Keywords en el primer caso, la definición de este segundo es mucho más rápida, de ahí la importancia de intentar 'modular' las diferentes acciones que se realizaran a lo largo de nuestros casos de test.

Como en el anterior caso, podemos comprobar que el caso de test funciona correctamente.

```
robot -v host:192.168.175.124 -v username:exfo -v password:Passw0rdastellia -v
PIPELINE_ID:123  --outputdir  robot/robot-results  --xunit  xunit.xml  -i  T2
robot/tests
```

```
==============================================================================
Send different amounts of data                                        | PASS |
------------------------------------------------------------------------------
Tests.HappyScenarios :: Robot Framework test script                   | PASS |
1 test, 1 passed, 0 failed
==============================================================================
Tests                                                                 | PASS |
1 test, 1 passed, 0 failed
```

Figura 15. ejecución exitosa del segundo caso de test

Ahora que tenemos más de un caso de test, podemos correrlos en el mismo comando, utilizando un 'tag' común, que como podemos apreciar en nuestro caso es pipeline.

```
robot -v host:192.168.175.124 -v username:exfo -v password:PasswOrdastellia -v PIPELINE_ID:123 --outputdir robot/robot-results --xunit xunit.xml -i Pipeline robot/tests
```

```
------------------------------------------------------------------------------
HappyScenarios :: Robot Framework test script
==============================================================================
Basic Test                                                            | PASS |
------------------------------------------------------------------------------
Send different amounts of data                                        | PASS |
------------------------------------------------------------------------------
HappyScenarios :: Robot Framework test script                         | PASS |
2 tests, 2 passed, 0 failed
------------------------------------------------------------------------------
```

Figura 16. ejecución de los 2 casos de test

Con esta última ejecución, concluye la automatización de los dos casos exitosos, donde se valida el correcto funcionamiento de la aplicación.

A continuación pasaremos a la segunda parte de nuestra batería de casos de prueba, en los cuales comprobaremos que nuestra aplicación se comporta de acuerdo a lo esperado en escenarios no exitosos. Estos test son igualmente importantes, ya que un comportamiento inesperado en la aplicación podría llegar a ser fatal (como por ejemplo corromper o borrar una base de datos en un caso extremo).

Para esto empezaremos creando el archivo NegativeScenarios.robot de la misma forma que el primer archivo.

```
*** Settings ***
Documentation      Robot Framework test script
Library    SSHLibrary    timeout=100 seconds
Library    String

Resource    ${EXECDIR}/robot/resources/Keywords/keywordsSetup.robot
Resource    ${EXECDIR}/robot/resources/Keywords/keywordsSender.robot

*** Variables ***
${alias}              remote_host_1
${server_name}    RobotServer_${PIPELINE_ID}
${client_name}=    RobotClient_${PIPELINE_ID}
${table_name}    hbase_client


*** Test Cases ***
```

Código 35. creación del archivo NegativeScenarios.robot

Procedemos a el desarrollo del 3 caso de prueba definido en los capítulos anteriores, el cual consiste en la validación del comportamiento de la aplicación cuando la IP del servidor no existe, en la cual la aplicación nos devolverá un error indicando precisamente, que la IP no es correcta.

Empezaremos definiendo las variables a utilizar y conectándonos a la VM.

```
Wrong Parameter input - IP address
    [Tags]    Pipeline    T3
    ${nb_threads}    Set Variable    1
    ${nb_burst}    Set Variable    1
    ${nb_rows_burst}    Set Variable    1
    ${nb_row_bytes_min}    Set Variable    45
    ${nb_row_bytes_max}    Set Variable    45
    ${sleep}    Set Variable    0
    ${stats}    Set Variable    overall-stats
    ${continue_on_exception}    Set Variable    continue-on-exception=no
    ${log_level}    Set Variable    debug
    ${memory_selection}    Set Variable    dynamic=1,1
    ${trigger}    Set Variable    size=1
    ${reconnect_on_fatal_ex}    Set Variable    reconnect-on-fatal-ex=no
    log    == Login to Common Environment ==
    Open Connection    ${host}    alias=${alias}
    Login    ${username}    ${password}    delay=1
```

Código 36. Inicio de la definición del tercer caso de test

Seguiremos con la obtención de los Ids del cliente y servidor, a parte de definir una ip errónea en la variable *${worng_ip}*.

```
log    == Step 1: Get Server and client Data ==
${server_id}=    Get id from container name    container=${server_name}
${client_id}=    Get id from container name    container=${client_name}
${wrong_ip}=    Set Variable    172.17.0.198
${table_name}    Set Variable    hbase_client
```

Código 37. obtención de ids y definición de una IP inexistente

Como siguiente paso, ejecutaremos la aplicación y validaremos que el error que devuelve es el esperado. Para este paso se definirá la Keyword *"Run Application and Expect Failure"*.

```
40      log    == Step 2: Run Application and expect error ==
41      Run Application and expect failure    id_container=${client_id}    path=/home    nb_threads=${nb_threads}
        nb_burst=${nb_burst}    nb_rows_burst=${nb_rows_burst}    nb_row_bytes_min=${nb_row_bytes_min}
        nb_row_bytes_max=${nb_row_bytes_max}    sleep=${sleep}    server_ip=${wrong_ip}    table_name=${table_name}
        stats=${stats}    continue_on_exception=${continue_on_exception}    log_level=${log_level}
        expected_error=IP    memory_selection=${memory_selection}    trigger=${trigger}    reconnect_on_fatal_ex=$
        {reconnect_on_fatal_ex}
```

Código 38. Llamada a la KW "Run application and expect failure" en el Test 3

Podemos apreciar que esta Keyword tiene como parámetros todo lo necesario para poder ejecutar la aplicación, a parte del expected_error, ya que esto nos permitirá en un futuro comprobar otro tipo de error en el futuro.

```
Run Application and expect failure
    [Arguments]    ${id_container}    ${path}    ${nb_threads}    ${nb_burst}    ${nb_rows_burst}    ${nb_row_bytes_min}    ${nb_row_bytes_max}    ${sleep}    ${server_ip}    ${table_name}    ${stats}    ${continue_on_exception}    ${log_level}    ${expected_error}    ${memory_selection}    ${trigger}    ${reconnect_on_fatal_ex}
    [Documentation]    Execute the Sender Application and expect an error in its execution (expected_error defined on the last argument):
    ...                - expected_error = IP
    ${parameters}=    Set Variable    ${nb_threads} ${nb_burst} ${nb_rows_burst} ${nb_row_bytes_min} ${nb_row_bytes_max} ${sleep} ${server_ip} ${table_name} ${stats} ${continue_on_exception} ${log_level} ${memory_selection} ${trigger} ${reconnect_on_fatal_ex}
    IF    '${expected_error}' == 'IP'
        ${stdout}    ${stderr}=    Execute Command    command=docker exec -w ${path} ${id_container} ./HbaseClientTest ${parameters}    sudo=True    sudo_password=${password}    return_stderr=True
        log    ${stdout}
        log    ${stderr}
        ${nb_what_lines}=    Count Lines Matching Regexp    ${stdout}    ^Thread\\s\\d*:\\sException\\soccurred:\\sTTransportException:\\sType:\\sNOT_OPEN:\\sWhat:\\sconnect\\(\\)\\sfailed.*$
        Should be true    ${nb_what_lines}==${nb_threads}    msg=Expected error msg not displayed
    END
```

Código 39. Definición de la keyword "Run Application and expect error"

Como se puede apreciar, teniendo en cuenta lo utilizado en los casos anteriores lo que hace esta función es ejecutar la aplicación con los parámetros introducidos, y dependiendo del tipo de error esperado (en nuestro caso IP) comprobar que el error que genera la aplicación es el esperado.

Esta comprobación se ha realizado utilizando expresiones regulares (Regexp), que son expresiones regulares básicas que pueden utilizarse para hacer coincidir patrones simples en cadenas de texto.

```
Count Lines Matching Regexp
    [Arguments]    ${str}    ${regexp}
    [Documentation]    Count of the no. of lines in ${str} matching ${regexp}
    ${lines}=    Get Lines Matching Regexp    ${str}    ${regexp}
    ${nb_lines}=    Get Line Count    ${lines}
    [Return]    ${nb_lines}
```

Código 40. definición de KW que comprueba las líneas coincidentes con la expresión regular del argumento

Así finalizaría la implementación del tercer caso, no haría falta el Teardown ya que no se ha insertado nada en Hbase.

De la misma forma que hemos hecho con los casos anteriores, ejecutaremos el caso de test para comprobar que todo funciona correctamente.

```
robot -v host:192.168.175.124 -v username:exfo -v password:PasswOrdastellia -v PIPELINE_ID:123 --outputdir robot/robot-results --xunit xunit.xml -i T3 robot/tests
```

```
==============================================================================
Wrong Parameter input - IP address                                    | PASS |
------------------------------------------------------------------------------
Tests.NegativeScenarios :: Robot Framework test script                | PASS |
1 test, 1 passed, 0 failed
==============================================================================
Tests                                                                 | PASS |
1 test, 1 passed, 0 failed
==============================================================================
```

Figura 17. Ejecución del tercer caso de test

Para finalizar con este capítulo, implementaremos el cuarto y último caso, en el cual se intentará enviar datos a un Servidor correcto, pero en una tabla incorrecta, por lo que tendremos que comprobar que la aplicación a intentado enviarlos, pero no con éxito.

Empezaremos de la misma forma que en los tres anteriores.

```
Wrong Parameter input - Wrong Hbase table_name
    [Tags]    Pipeline    T4
    ${nb_threads}    Set Variable    1
    ${nb_burst}    Set Variable    1
    ${nb_rows_burst}    Set Variable    2
    ${nb_row_bytes_min}    Set Variable    256
    ${nb_row_bytes_max}    Set Variable    256
    ${sleep}    Set Variable    0
    ${stats}    Set Variable    overall-stats
    ${continue_on_exception}    Set Variable    continue-on-exception=no
    ${log_level}    Set Variable    debug
    ${memory_selection}    Set Variable    dynamic=1,1
    ${trigger}    Set Variable    size=1
    ${reconnect_on_fatal_ex}    Set Variable    reconnect-on-fatal-ex=no
    log    == Login to Common Environment ==
    Open Connection    ${host}    alias=${alias}
    Login    ${username}    ${password}    delay=1
    log    == Step 1: Get Server and client Data ==
    ${server_id}=    Get id from container name    container=${server_name}
    ${client_id}=    Get id from container name    container=${client_name}
    ${server_ip}=    Get ip from container id    ${server_id}
    ${table_name}    Set Variable    hbase_wrong
```

Código 41. Inicio del 4 caso de Test

Podemos ver como hemos utilizado un nombre de tabla erróneo para provocar el escenario deseado.

Ahora ejecutaremos la aplicación

```
65      log    == Step 2: Run Application ==
66      ${response}    Run Application    id_container=${client_id}    path=/home    nb_threads=${nb_threads}
        nb_burst=${nb_burst}    nb_rows_burst=${nb_rows_burst}    nb_row_bytes_min=${nb_row_bytes_min}
        nb_row_bytes_max=${nb_row_bytes_max}    sleep=${sleep}    server_ip=${server_ip}    table_name=${table_name}
        stats=${stats}    continue_on_exception=${continue_on_exception}    log_level=${log_level}    memory_selection=$
        {memory_selection}    trigger=${trigger}    reconnect_on_fatal_ex=${reconnect_on_fatal_ex}    successful=False
```

Código 42. ejecución de la aplicación, a una tabla inexistente

En este caso, la aplicación no devolverá un error, ya que la conexión se realiza correctamente con el servidor, por lo que tendremos que comprobar que, en las estadísticas, esta inserción de datos no se ha realizado exitosamente.

Utilizaremos la Keyword *"Check Data output"* la cual utilizamos en los casos exitosos, pero en este caso, para confirmar que la aplicación nos dice que la inserción se ha intentado, pero sin éxito.

```
  log    == Step 3: Check Stats output ==
  ${attempted_rows}=    Check Data Output    response=${response}    nb_burst=${nb_burst}    nb_rows_burst=$
  {nb_rows_burst}    nb_row_bytes_min=${nb_row_bytes_min}    nb_row_bytes_max=${nb_row_bytes_max}
  Check Stats Output    response=${response}    min_bytes=${nb_row_bytes_min}    max_bytes=$
  {nb_row_bytes_max}    requested_rows=${attempted_rows}    attempted_rows=${attempted_rows}
  successful_rows=0    unsuccessful_rows=2    nb_threads=${nb_threads}    nb_burst=${nb_burst}    stats=$
  {stats}
```

Código 43. Comprobación de que el envío no ha sido exitoso

El mensaje que nos da la aplicación es el siguiente

```
Overall batch_sender_statistics = {sender_stats = {requested = {2, 512}, attempted =
{2, 512}, successful = {0, 0}, unsuccessful = {2, 512}}
```

Con esto ya tendriamos los 4 casos de test definidos, podemos comprobar que todos son correctos ejecutando de nuevo el comando de robot utilizando el tag Pipeline

```
==============================================================================
Basic Test                                                            | PASS |
------------------------------------------------------------------------------
Send different amounts of data                                        | PASS |
------------------------------------------------------------------------------
Tests.HappyScenarios :: Robot Framework test script                   | PASS |
2 tests, 2 passed, 0 failed
==============================================================================
Tests.NegativeScenarios :: Robot Framework test script
==============================================================================
Wrong Parameter input - IP address                                    | PASS |
------------------------------------------------------------------------------
Wrong Parameter input - Wrong Hbase table_name                        | PASS |
------------------------------------------------------------------------------
Tests.NegativeScenarios :: Robot Framework test script                | PASS |
2 tests, 2 passed, 0 failed
==============================================================================
Tests                                                                 | PASS |
4 tests, 4 passed, 0 failed
```

Código 44. ejecución de los 4 casos de test definidos en la misma ejecución.

Con esto damos por finalizado la implementación de los 4 casos de test, que en el siguiente capitulo serán implementados en una regresión utilizando Gitlab CI/CD

4.4 Regresión utilizando Gitlab CI/CD

En este capítulo se explicará el proceso realizado en el cual, a partir de todo lo desarrollado anteriormente, definimos una regresión, la cual nos permite poder correr la batería de casos de test funcionales cada vez que hay un cambio en el código de la aplicación, permitiéndonos así ver en qué momento se está introduciendo un bug, y poder solucionarlo

4.4.1 Desarrollo de la idea de Regresión

Tal como hemos mostrado en los capítulos anteriores, hemos basado nuestro testing en una aplicación previamente creada e introducida en nuestra maquina virtual.

Esto nos es útil cuando queremos comprobar que una versión o una entrega cumple con todos los requisitos funcionales, pero no nos permite saber, si hay algún error, cuando se ha producido.

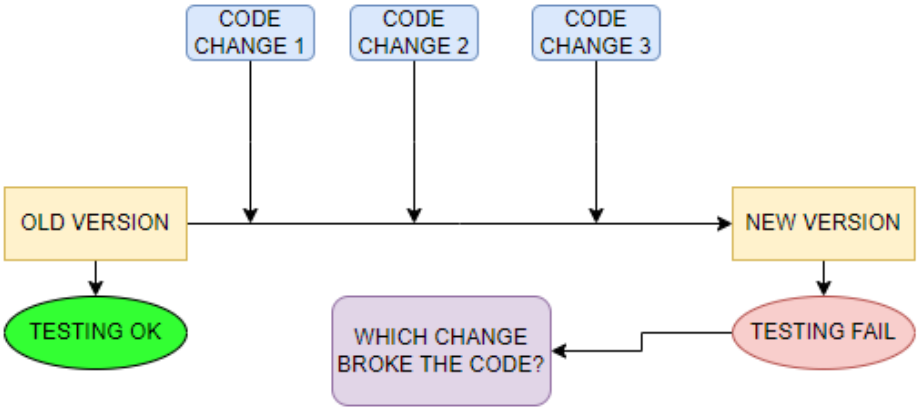


Figura 18. Diagrama donde la validación se realiza al final del desarrollo

Esto puede presentar problemas en un equipo de desarrollo, ya que no se encuentran los problemas hasta que se llega a la fase de validación de la aplicación. La idea propuesta para solucionar esto es utilizar una regresión, en la que cada cambio en el código es validado, y en caso de error en los test, no permitir incluirlo en la rama principal.

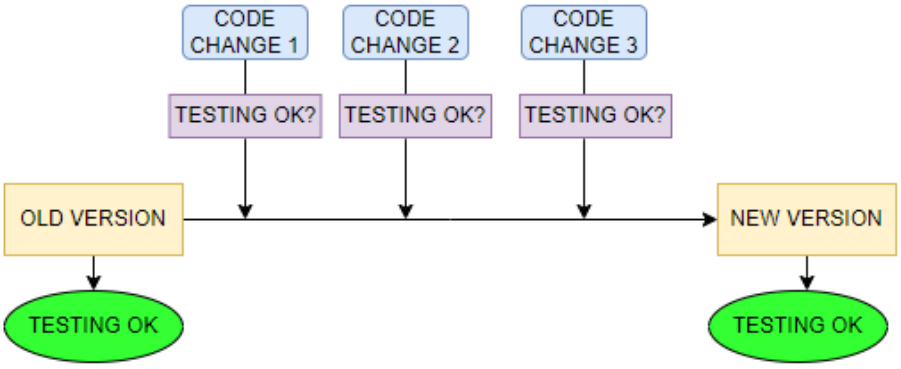


Figura 19. Diagrama donde la validación se realiza durante el desarrollo, validando cada cambio en la aplicación

Para poder plasmar esta idea de trabajo en el desarrollo de nuestra aplicación, utilizaremos Gitlab CI/CD, ya que como hemos explicado anteriormente, gracias a su sistema de pipelines, nos permite realizar estas acciones en cada cambio que introduzcamos en el código

4.4.2 Definición de etapas en la Pipeline

Teniendo la idea de lo que queremos conseguir, tenemos que particularizar esta idea de regresión en nuestro caso.

Podríamos separar este proceso en distintas etapas:

1. Compilar el código de nuestra aplicación para generar un nuevo ejecutable (el cual se utiliza para ejecutar la aplicación)
2. Sustituir el ejecutable que se encuentra en la máquina virtual , por el nuevo, para poder ejecutar los casos de test con los cambios de la aplicación
3. Desplegar el entorno de test, correr los casos de test y eliminar los contenedores de la VM

Estas etapas puestas en un diagrama se verían tal que así:

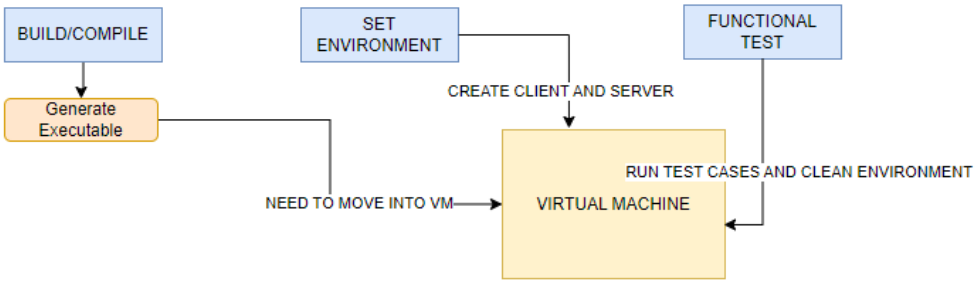


Figura 20. Etapas propuestas para la Pipeline que valida la aplicación en cada cambio del código.

4.4.3 Creación de gitlab runner

Un Gitlab runner es un agente, que nos permitirá ejecutar los diferentes Jobs o etapas que queramos definir en la pipeline (en nuestro caso específico, tal como podemos ver en la figura 20, las etapas **Build**, **Set environment** y **functional test**).

Dicho esto, vamos a proceder con crear un runner en nuestro entorno que nos permita ejecutar estas acciones que queremos realizar de forma automática.

El primer paso es instalar el servicio de gitlab-runner en nuestra máquina virtual, que realizamos en capítulos anteriores.

A continuación, tendremos que crear el runner desde la UI de gitlab.

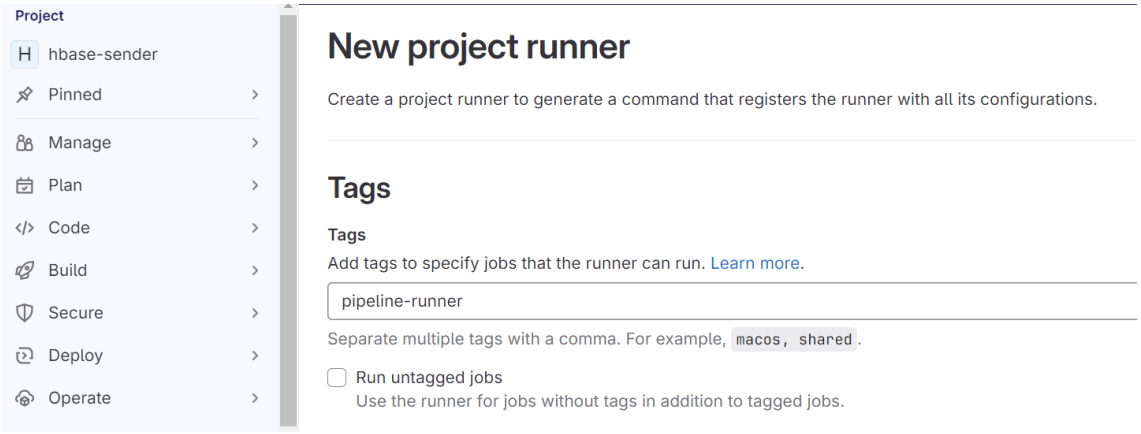


Figura 21. Ventana de Creacion de un runner en la UI de Gitlab

Utilizaremos el Tag "pipeline-runner" que luego necesitaremos para poder llamar al runner en la pipeline.

Cuando lo creemos, solo tendremos que ejecutar el comando que nos dará Gitlab para poder registrarlo en nuestra VM

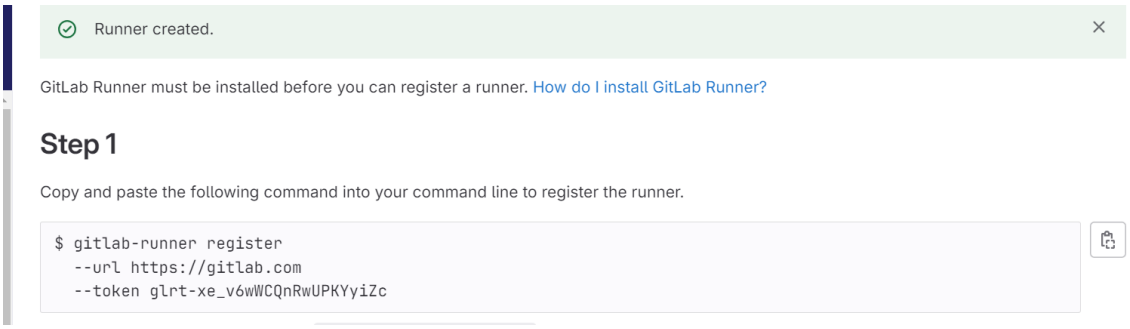


Figura 22. Runner creado generando el token para poder registrarlo en una VM

Con esto nuestro runner debería estar listo para ser utilizado. Para comprobar que no hay ningún problema de conectividad podemos ejecutar el comando "*sudo gitlab-runner verify*", que nos confirmara que la conexión es posible:

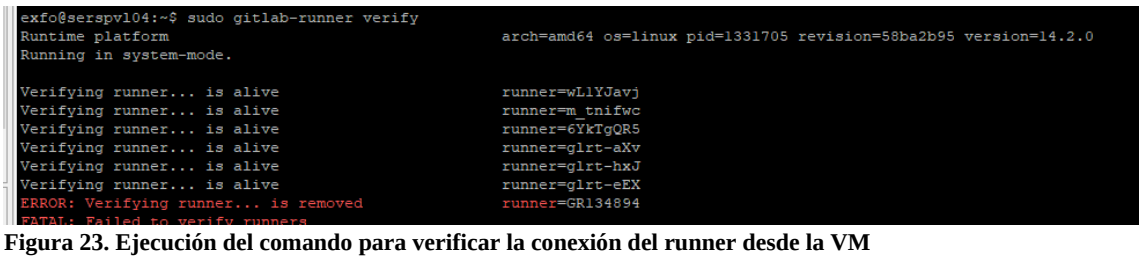


Figura 23. Ejecución del comando para verificar la conexión del runner desde la VM

Como podemos ver en este ejemplo, si la conexión es correcta, obtendremos el mensaje “is alive”, en caso contrario tendremos el mensaje “is removed”.

4.4.4 Definición de la Pipeline

Teniendo nuestro runner listo y claras las estapas de la pipeline, vamos a proceder a crear el archivo que nos permitirá que la pipeline se ejecute con cada cambio de código.

El archivo que crearemos se llamará “. *gitlab-ci.yml”,* este es un nombre estándar para todos los proyectos de Gitlab

La estructura de este tipo de archivo es la siguiente:

1. Al principio de este se definen los “stages”, que podrían definirse como etapas de la pipeline, en nuestro caso tendremos 2: Build y test
2. Posteriormente, se definirán los Jobs, que son los trabajos que queremos que realice el runner, estos Jobs se encuentran dentro de un stage. En nuestro caso tendremos 3 jobs.
   - **Compile** (stage build)
   - **Set-environment** (stage test)
   - **Robot-test** (stage test)

Teniendo esto claro vamos a pasar a definir cada job y explicar todos sus elementos:

Empezaremos definiendo los stages:

```
stages:
  - build
  - test
```

Código 45. Definición de los stages de la pipeline.

Después de este paso, ya podemos empezar con la definición de los distintos Jobs que se encuentran en estos stages:

```
compile:
  stage: build
  image: registry.gitlab.com/exfo/products/asa/msu/sdp/sender/hbase-sender:hbase-client
  variables:
    GIT_SUBMODULE_STRATEGY: recursive
    GIT_SUBMODULE_UPDATE_FLAGS: --remote
  script:
    - cmake .
    - make -j4
  artifacts:
    paths:
      - "build"
```

Código 46. Definición del Job compile

1. Podemos ver que empezamos con el nombre del job (compile) y el stage al que pertenece (build)
2. A continuación proporcionaremos la imagen que queremos que el runner utilice en este job (en este caso, utilizaremos la imagen que utilizamos en el cliente, ya que tiene las dependencias instaladas para poder compilar el código de la aplicación.
3. Posteriormente, sigue el script, que es donde le indicaremos que comandos debe ejecutar el runner, en este caso los dos comandos son los que se utilizan para compilar nuestra aplicacipon y generar el ejecutable en una carpeta build

4. Elegiremos el tag del runner que queremos que ejecute nuestro job, en nuestro caso utilizaremos el creado anteriormente para los 3 casos (pipeline-runner)
5. Para finalizar, tenemos los artifacts. Estos sirven para almacenar una carpeta en el runner para futuros trabajos/Jobs , en nuestro caso, como necesitamos el nuevo ejecutable para ejecutar los casos de test, almacenaremos la carpeta build .

Antes de empezar con la definición de los siguientes jobs, debemos tener una cosa en cuenta: Cada vez que se corra el job compile, tendremos un nuevo ejecutable, por lo que tendremos que actualizar el que se encuentra en la máquina virtual antes de desplegar nuestro entorno.

Para ello haremos una pequeña modificación en nuestra función, para poder mover el ejecutable desde el runner a nuestra máquina virtual:

Adaptaremos nuestra Keyword *"Create Client and return id"* para que el ejecutable se coja desde el runner:

```
Create Client and return id Pipeline
    [Arguments]    ${image}    ${name}
    ${id_container}=    Create Container from image    image=${image}    name=${name}
    ${runner_id}=    Get Runner id
    Copy executable form Runner to Client    ${runner_id}    ${id_container}
    [Return]    ${id_container}
```

Código 47. Nueva KW para obtener el ejecutable de la pipeline, siendo este el generado con los últimos cambios del código.

Donde en la nueva Función, se añade un paso, el cual es copiar el ejecutable desde los artifacts del runner a la VM.

```
47  Copy executable form Runner to Client
48      [Arguments]    ${runner_id}   ${id_client}
49      [Documentation]    This keyword is used to move the executable we got created on the previous job of the
        pipeline as an artifact
50      ...                to the client container where we are going to run the application
51      ${rc}    Execute Command    command=docker cp ${runner_id}:${RUNNER_PATH}/build/HbaseClientTest /tmp/
        HbaseClientTest    sudo=True    sudo_password=${password}    return_stdout=False    return_rc=True
52      Should be equal as integers    ${rc}    0    msg=${rc} =! 0 'command not successful
53      ${rc}    Execute Command    command=cp /tmp/HbaseClientTest HbaseClientTest    sudo=True    sudo_password=$
        {password}    return_stdout=False    return_rc=True
54      Should be equal as integers    ${rc}    0    msg=${rc} =! 0 'command not successful
55      ${rc}    Execute Command    command=docker cp HbaseClientTest ${id_client}:/home/HbaseClientTest
        sudo=True    sudo_password=${password}    return_stdout=False    return_rc=True
56      Should be equal as integers    ${rc}    0    msg=${rc} =! 0 'command not successful
```

Código 48. Proceso para obtener el ejecutable desde el runner

Para obtener el identificador del runner , se utiliza la siguiente keyword.

```
Get Runner id
    ${id}=    Execute Command    command=docker ps -f "name=runner" -q    sudo=True    sudo_password=${password}
    [return]    ${id}
```

Código 49. KW para obtener el id del runner

Con estos cambios nuestro código estará preparado para poder utilizar el ejecutable obtenido en el job anterior. Esto nos permite utilizar la versión generada con el código del repositorio, por lo que podemos ver si hay algún cambio en el funcionamiento con cada cambio realizado en el código de la aplicación.

Para finalizar, añadiremos una función en el archivo HbaseSetup.robot, para poder borrar los contenedores al final de los casos de test, ya que, sin este, una limpieza manual sería necesaria para el mantenimiento de la VM.

```
Kill containers
    [Tags]    Kill
    Open Connection     ${host}        alias=${alias}
    Login               ${username}    ${password}    delay=1
    log    == Step 1: set variables ==
    ${client_name}=    Add sufix to String    string=RobotClient    sufix=${PIPELINE_ID}
    ${server_name}=    Add sufix to String    string=RobotServer    sufix=${PIPELINE_ID}
    Remove containers    containers=${client_name} ${server_name}
```

Código 50. función para eliminar los contenedores del cliente y el servidor al finalizar la ejecución de la pipeline.

Utilizaremos la etiqueta “kill”, para poder ejecutarla en el comando de robot

Teniendo este nuevo código, estamos listos para definir el resto de la pipeline:

Seguimos con el job **set-environment.**

```
set-environment:
  stage: test
  image: registry.gitlab.com/exfo/products/asa/msu/sdp/test-automation/docker-robot:latest
  needs:
    - job: compile
  script:
    - RUNNER_PATH=${PWD}
    - mkdir -p robot/robot-results junit
    - robot -v RUNNER_PATH:$RUNNER_PATH -v host:$TEST_VM -v username:$ROBOT_USER -v password:$ROBOT_PASSWORD -v
    PIPELINE_ID:$CI_PIPELINE_ID -i Setup2  --outputdir robot/robot-results --xunit xunit.xml robot/resources/
    Functions/HbaseSetup.robot
  tags:
    - test-bench
  artifacts:
    paths:
      - ./robot/robot-results/
    expire_in: 72h
```

Código 51. definición del Job set-environment, perteneciente al stage test.

Empezaremos de la misma forma, definiendo el nombre del job y a continuación el stage y la imagen. En este caso la imagen utilizada tendrá instalada robotframework, pyhton y las dependencias necesarias para poder ejecutar los casos de test.

Imagen creada a partir del siguiente dockerfile, que simula las dependencias que hemos instalado en nuestro entorno Windows para ejecutar los casos de prueba.

```
FROM ubuntu:20.04

USER root

RUN apt-get autoremove && apt-get -y -f install && apt-get -y update && apt-get -y upgrade
RUN apt install software-properties-common -y

# Install Python 3.10.4 via deadsnakes repo
RUN add-apt-repository ppa:deadsnakes/ppa
RUN apt update
RUN apt install python3.10 -y
RUN apt install python3.10-distutils -y

# Install curl and pip3 to work with Python 3.10.4
RUN apt install curl -y
RUN curl -sS https://bootstrap.pypa.io/get-pip.py | python3.10

#############################################
# Install libraries and tests requirements
#############################################
RUN pip3 install robotframework==5.0
RUN pip3 install robotframework-sshlibrary==3.8.0

RUN apt update
RUN apt install -y git
```

Código 52. Dockerfile con las dependencias para ejecutar tests de robot

A continuación, indicaremos que el job 'depende' de otro, ya que necesitamos el artifact con el ejecutable, de ahí la línea *needs -job:compile*

Seguiremos con el script, que consiste en 3 comandos:

1. Guardaremos el path del runner en una variable, ya que se utilizará para mover el ejecutable desde el runner al cliente
2. Creamos el directorio para los logs de la ejecución
3. Ejecutamos el comando de robot de la misma forma que haríamos ejecutando un caso de test en nuestro entorno Windows

Para finalizar, definiremos el runner que utilizaremos, y donde se guardarán los artifacts (en este caso hemos añadido un tiempo de expiración, lo que significa que en 72 h los logs desaparecerán, para ahorrar espacio en nuestro entorno)

Con esto ya tendríamos el 2º job listo, pasaremos a definir el ultimo:

```
robot-test:
  stage: test
  image: registry.gitlab.com/exfo/products/asa/msu/sdp/test-automation/docker-robot:latest
  needs:
    - job: set-environment
  script:
    - RUNNER_PATH=${PWD}
    - mkdir -p robot/robot-results junit
    - robot -v RUNNER_PATH:$RUNNER_PATH -v host:$TEST_VM -v username:$ROBOT_USER -v password:$ROBOT_PASSWORD
    --outputdir robot/robot-results --xunit xunit.xml -i PipelineORkill robot/tests robot/resources/Functions/
    HbaseSetup.robot
  tags:
    - pipeline-runner
  artifacts:
    paths:
      - ./robot/robot-results/
    expire_in: 72h
```

Código 53. Definición del job robot-test, perteneciente al stage test

Como podemos ver, la estructura es casi idéntica:

1. En este caso, la dependencia está en que set-environment se ejecute primero, por lo que aplicaremos la condición *needs: job: set-environment*
2. Como podemos ver en la última línea del script, ejecutaremos los casos que contengan la etiqueta Pipeline o la etiqueta kill, y primero se ejecutaran los test de la carpeta robot/tests (la batería de casos de test automáticos) y después los casos del archivo HbaseSetup.robot (para eliminar los contenedores). De esta forma nos aseguramos de que siempre se corran los casos de test antes que el reinicio del entorno.

Con esto damos por finalizado el desarrollo de la pipeline y todas sus etapas.

4.4.5 Ejemplo de ejecución de la Pipeline

Con el ultimo capitulo, tenemos todo listo para poder realizar la regresión, solo tendremos que introducir nuestros archivos en el repositorio de Gitlab donde se encuentra nuestro proyecto, y en cada cambio en el código se lanzara una regresión.

El flujo de trabajo sería el siguiente:

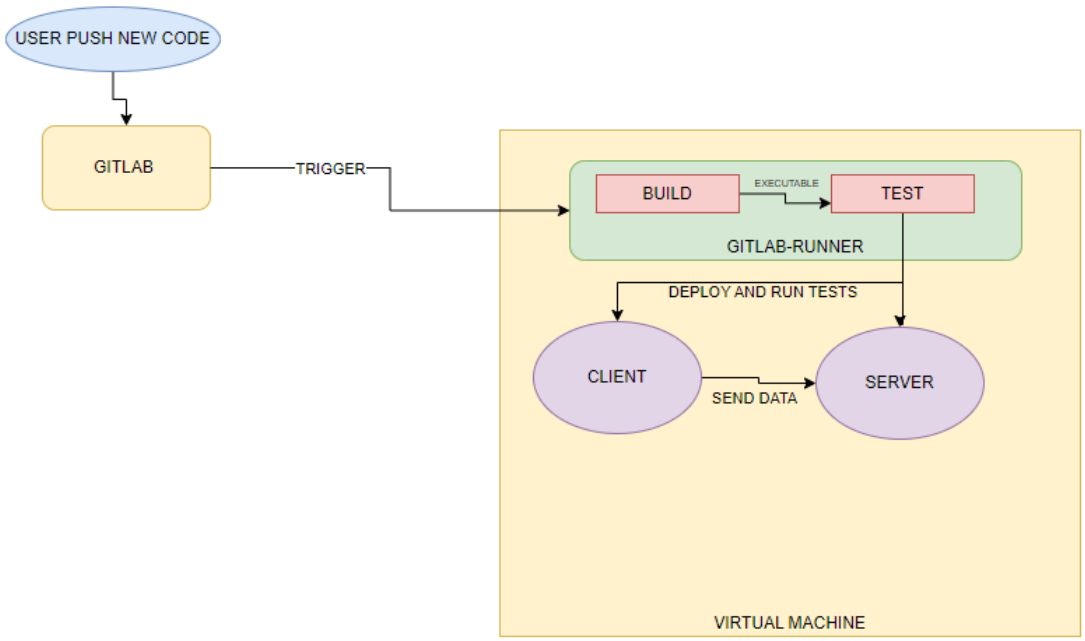


Figura 24. Flujo de trabajo que se realiza con un cambio en el código

Vamos a mostrar ahora este caso de uso desde la UI de Gitlab:

Primero el usuario introducirá un cambio en su rama de cualquier parte del código.

Automáticamente se lanzará una pipeline para comprobar si los casos de test siguen pasando o hay algún error

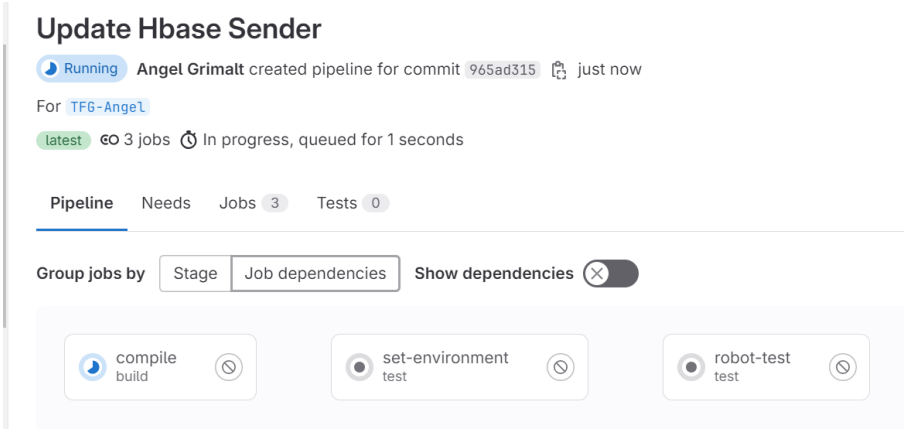


Figura 25. Pipeline es iniciada a causa de un cambio en el código

Cuando esta finalice hay dos opciones:

1. La pipeline pasa correctamente, por lo que la aplicación sigue portándose como debe, por lo que el desarrollador podría meter su código en la rama principal

2. La pipeline falla, por lo que el desarrollador tendrá que buscar con la ayuda de los resultados de los casos de test, en qué punto falla su aplicación.

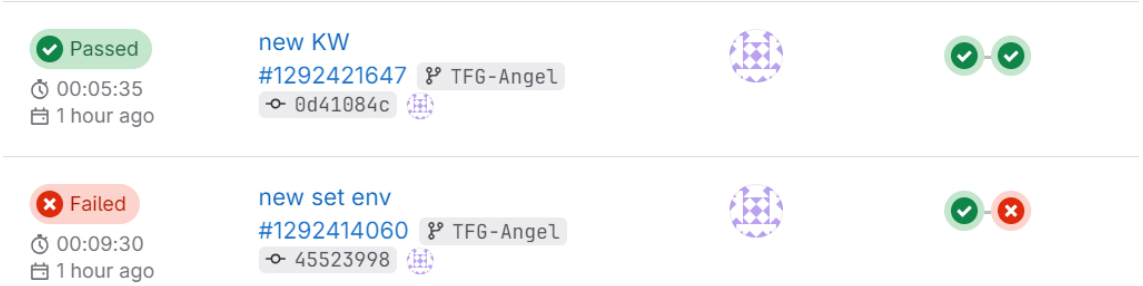


Figura 26. 2 ejecuciones de la Pipeline, una exitosa y otra fallida.

Capítulo 5. Conclusión

5.1 Evaluación del cumplimiento de objetivos

Para evaluar el cumplimiento de los objetivos, tomaremos como referencia, lo establecido en el capítulo 2 de este proyecto.

Como objetivo principal se planteó la creación de un proceso mediante el cual poder testear una aplicación de una forma automatizada.

Para ello se analizó todo lo que se consideró todo lo que una aplicación necesita para cumplir con los estándares de calidad y se definió todo lo necesario para poder asegurar esta calidad en la aplicación.

Este objetivo principal, que es un tanto general, se desgrano en diferentes puntos, que nos permiten evaluar de una forma mas precisa el cumplimiento de este:

- Se ha explicado y analizado la importancia de todo el proceso de validar y probar un producto/servicio en el desarrollo de aplicaciones, exponiendo todos los beneficios que esto ofrece, además de los problemas que la ausencia de este puede provocar.
- Se ha diseñado, y posteriormente, creado un entorno en el cual se puede ejecutar nuestra aplicación, paso imprescindible para poder probarla de una forma similar a la que será utilizada por un cliente
- Hemos diseñado unos casos de prueba, los cuales nos permitirán validar que el comportamiento de la aplicación es el esperado tanto en casos exitosos de uso, como en casos donde la aplicación no puede ser utilizada exitosamente.
- Hemos automatizado, utilizando la herramienta RobotFramework, los casos creados en el objetivo anterior, explicando paso a paso como codificar los diferentes pasos del caso de prueba. Además, se ha automatizado el despliegue de los contenedores necesarios para la ejecución de la aplicación.
- Todos estos procesos, han sido incorporados en un flujo de trabajo utilizando Gitlab CI/CD, explicando todos los beneficios que este ofrece sobre otras formas de trabajo y validación de aplicaciones.
- En este proyecto, se han explicado, paso a paso, todos los procedimientos realizados, incidiendo en el motivo por el cual se han tomado diferentes decisiones respecto a otras posibilidades.
- Se le ha dado mucha importancia al hecho d que una aplicación cumpla los estándares de calidad, centrando en esto todo el trabajo, ya que consideramos que entregar una aplicación sin calidad, es contraproducente, ya que además de no cumplir con las expectativas, si esto se convierte en una constante, puede repercutir en la confianza que otros clientes pueden tener hacia nuestro producto, haciéndoles buscar alternativas mas fiables.
- Hemos utilizado herramientas que permiten al usuario poder analizar los problemas y poder resolverlos de una forma eficaz, en el menor tiempo posible.

Para finalizar con la conclusión, me gustaría destacar que se han cumplido todos los objetivos marcados y en general, ha sido una experiencia gratificante, ya que considero que lo realizado en este proyecto puede ser de mucha utilidad en diferentes ámbitos del mundo del desarrollo.

5.2 Propuesta de trabajo futuro

Como propuesta de trabajo futuro, se abre un mundo de posibilidades. Este proyecto que hemos realizado, el cual, valida una aplicación simple con 4 casos de prueba, y es altamente escalable, podría evolucionar a un flujo de trabajo en el cual se valida una aplicación con diversas funcionalidades con una batería de test de más de 100 casos.

Inclusive, a parte de la validación de la aplicación, la base que hemos definido en GitLab podría ser utilizada para la automatización de más procesos, como podrían ser:

- un análisis de código, el cual nos recomiende ciertas mejoras posibles en el código.
- implementar otro tipo de pruebas, como podría ser, un análisis de vulnerabilidades.

Aunque, lo más interesante en mi opinión sería una implementación a gran escala, en el cual una solución formada por diferentes microservicios es desarrollada por distintos o el mismo equipo dentro de un grupo de trabajo o empresa, donde todos utilizan este flujo de trabajo desarrollado en este proyecto. Validando su aplicación de forma independiente, para después, implementar un "stage" en GitLab que despliegue estos microservicios en el entorno donde se encuentra el resto, y se compruebe que cada nueva versión generada por los diferentes equipos funciona correctamente y es compatible como sus versiones anteriores.

Capítulo 6. Bibliografía y referencias.

[1] Como desplegar Ubuntu en una máquina virtual https://ubuntu.com/tutorials/how-to-run-ubuntu-desktop-on-a-virtual-machine-using-virtualbox#1-overview
[2] Información sobre el despliegue de contenedores de docker https://docker-docs.uclv.cu/get-started/
[3] Robotframework documentation. https://robotframework.org/
[4] Get started with GitLab CI/CD https://docs.gitlab.com/ee/ci/
[5] How to install GitLab-runner on Ubuntu https://linux.how2shout.com/how-to-install-gitlab-runner-on-ubuntu-such-as-22-04-or-20-04/