Generador de Documentación Dinámica en Entornos Regulados

Samer Bujana Escalona
Ingeniería Informática, Mención en Tecnologías de la Información
Universidad Autónoma de Barcelona
Barcelona, Cataluña, España
samerernesto.bujana@autonoma.cat

***Abstract* — Este documento narra la ideación y el desarrollo de un sistema que será usado internamente en la empresa corporativa Roche como herramienta para automatizar los procesos de documentación de sus proyectos mediante la utilización de scripts de Python incluidos en *pipelines*.**

***Index Terms*—Continuous, Documentation, Information Search and Retrieval, Integration, Pipeline, Software development, Software Engineering Process, Systems development, System integration and implementation, Test documentation**

I. Introducción

ROCHE es una farmacéutica multinacional con muchos proyectos de gran calibre en desarrollo. Es importante que cada uno de ellos disponga de una buena documentación para cada *release* del producto que se desarrolla en dicho proyecto. Esta tarea resulta tediosa porque actualmente la documentación se genera manualmente, esta contiene la trazabilidad a los requerimientos y los artefactos producidos en la *release* actual. Otro de los inconvenientes que surge es que los diferentes proyectos de la empresa utilizan distintos sistemas que almacenan los requisitos en formatos y de formas distintas. Para intentar paliar dichos inconvenientes, se plantea el desarrollo de un servicio que se pueda ejecutar en cualquier servicio de integración continua para obtener la trazabilidad, información sobre los requisitos, pruebas y sus resultados y generar la documentación correspondiente para el proyecto según se define en el *Quality Management System* de la empresa.

II. Objetivo

Crearemos un sistema que trabajará conjuntamente con los sistemas de información utilizados por la empresa para recoger la información y requerimientos necesarios de ciertas tareas en el momento de ejecutar el *pipeline* de un proyecto para crear la documentación de manera automatizada. Desarrollaremos un sistema que trabajará estrechamente con los sistemas de información utilizados por la empresa: Jira[1] y HP ALM[2], aunque actualmente, el segundo software ha sido adquirido por Micro Focus y las versiones más actuales ya no pertenecen a HP; de donde recuperaremos la información y requerimientos necesarios. Para automatizar nuestro sistema nos aprovecharemos del servicio de integración continua de la herramienta GitLab[3] que, además, será donde se encontrarán alojados nuestros repositorios. El propósito de nuestro servicio será acceder y obtener diferentes datos alojados en cada uno de los sistemas de información para cruzarlos y generar un *output* de documentación. Este *output* se publicará en algunas de las herramientas comentadas previamente para mantenerlas correctamente documentadas.

Lo primero que haremos al organizar el proyecto será averiguar cuál es el lenguaje más adecuado para el desarrollo del sistema, ya que buscamos uno que nos permita trabajar con cierta facilidad pero que también sea sencillo combinarlo e implementarlo a los proyectos que tenemos activos. Los dos candidatos principales son *Python* y *Java*. Posteriormente, investigaremos cómo conectarnos a los sistemas de información con los que trabajaremos y cómo recuperaremos los datos. Finalmente desarrollaremos el sistema que generará la documentación con un formato concreto en el servicio de integración continua y subiremos estos *outputs* como resultado de la ejecución del *pipeline* e posiblemente a los sistemas de información mencionados anteriormente.

Este proyecto no se desarrollará directamente sobre proyectos ya existentes de la empresa, usaremos unos repositorios de prueba. Una vez este proyecto, que nos sirve como *proof of concept*, haya completado su desarrollo y sepamos la viabilidad de su implementación a gran escala, podremos iniciar su despliegue a un nivel más extendido en la empresa.

III. Organización del proyecto

Hay que tener en cuenta que aunque el proyecto y las tareas de desarrollo del sistema sigan una esquemática lineal, el segundo, será más semejante a una estructura en espiral, como las utilizadas en proyectos que disponen de un equipo que sigue la metodología *Scrum*[4]. Se desarrollará el sistema imitando el funcionamiento de la organización por *sprints* como en la metodología mencionada, por lo tanto, cada cierto tiempo se revisará el progreso, se analizará y determinará si se avanza por buen camino, si es necesario revertir alguna de las tareas o incluso si es necesario reestructurar la organización.

El proyecto se compone de tres fases: una inicial que contendrá las tareas relacionadas con el informe inicial, una de desarrollo

con todas sus tareas pertinentes y una final relacionada con las entregas finales del trabajo de fin de grado.

IV. Estado del arte

En otros proyectos la documentación y el cruce de datos se realizan de manera totalmente manual, por tanto: una persona busca en Jira los IDs de las instancias o *backlogs* correspondientes a una *release* concreta y luego se localizan estos IDs en el código; se crea una matriz de trazabilidad que indica el ID del test correspondiente a cada instancia de Jira, y en el caso de que el test en el código carezca de ID se coloca el código del test directamente. Además del documento de la matriz de trazabilidad se genera otro que contiene los tests, datos particulares de estos y también el resultado de sus ejecuciones.
Hay que tener en consideración que en cada proyecto el proceso de documentación se puede realizar de manera distinta en función de las características y las necesidades de este, pero si nuestro trabajo resulta práctico y funcional se pretende implementar en todos aquellos proyectos con una estructura similar y, es posible que en algunos casos se llegue a modificar ligeramente para adaptarlo a otros.

V. Desarrollo

A. Primera fase de desarrollo

1) Análisis del lenguaje del código

Hemos mantenido una reunión en la que se han decidido algunos aspectos sobre el proyecto, como qué lenguaje que se va a utilizar. Las dos opciones que se han sopesado son Java y Python. Ambas opciones son viables, pero para este proyecto necesitamos la versatilidad y simplicidad que nos proporciona Python. Este lenguaje interpretado es mucho más sencillo de utilizar y permite crear códigos altamente eficientes con facilidad, además, su implementación y ejecución también es menos compleja. El código que desarrollaremos se ejecutará en un *docker* contenido en un *pipeline* así que solo necesitaremos una máquina con Python para poder ejecutar el script, en cambio, si fuéramos a usar Java, necesitaríamos un conjunto de documentos que formen un proyecto que luego debe ser compilado, así que necesitaríamos añadir el lenguaje y por lo menos alguna otra herramienta como puede ser Maven[5] o Gradle[6] para gestionar, compilar y ejecutar el proyecto.
Otra de las ventajas de Python es que dispone de librerías de código abierto que nos pueden resultar muy útiles, solo tendremos que descargarlas en la máquina que vaya a ejecutar el código y luego importar la librería en el script.

2) Preparación inicial

Antes de la implementación del código en el sistema de integración continua lo desarrollaremos en local para tener una versión funcional y medianamente pulida del código hasta poder pasar a una fase de implementación más genérica.
Para poder ejecutar el script que vamos a crear tenemos que asegurarnos de que nuestra máquina dispone de Python, para ello deberemos instalarlo a través del terminal. La herramienta *pip*, que la necesitaremos para instalar librerías y otras herramientas, vendrá por defecto al instalar el lenguaje. Una vez instalado, simplemente necesitamos analizar cómo vamos a conectarnos a los sistemas de información con los que vamos a trabajar. Empezaremos con Jira, podemos acceder a este sistema de información a través de la web y disponemos de dos métodos para conectarnos a él: usando su REST API[7] y realizar peticiones HTTP, o utilizando una librería dedicada existente para Python[8][9]. Para poder usar la librería dedicada tenemos que instalarla usando la herramienta *pip* en la terminal con el siguiente comando: `pip install jira`.
Hemos analizado cómo realizar una conexión básica y cómo procesar la información, y hemos llegado a la conclusión de que el nivel de complejidad al usar la librería dedicada se reduce en comparación al usar la REST API. Si decidiéramos usarla, debemos tener en cuenta que tendremos que crear y gestionar las conexiones HTTP de manera manual, tanto las peticiones como las respuestas, y también tendremos que procesar las respuestas para extraer los datos; todo esto nos lo podemos ahorrar simplemente solicitando el dato que queremos usando la librería dedicada. También hay que tener en cuenta que gestionar un error nos resultará mucho más fácil con la librería dedicada ya que simplemente saltará el error, y para el caso de la petición HTTP seguiremos recibiendo una respuesta válida, pero no seremos conscientes de que ha saltado un error hasta que la procesemos y veamos en su contenido que la respuesta lleva un código identificador 400 y el mensaje de error en lugar de los datos esperados.
A pesar de sus diferencias, ambas opciones se asemejan, ya que la petición de los datos se reduce a lo mismo: JQL[10]. JQL o *Jira Query Language*, es la versión de SQL de Jira. Para solicitar datos contenidos en Jira utilizaremos este lenguaje para parametrizar una petición o *query*, dicha petición nos devolverá los datos cumpliendo con las condiciones y filtros que le hayamos indicado. Primero, crearemos un objeto con los datos básicos de autenticación, como son un usuario en el servidor dedicado de Jira y un token de acceso. De momento, trabajaremos con datos muy concretos para cerciorarnos de que funciona correctamente, y más adelante generalizaremos el código.
Para no desvelar datos privados algunos campos estarán vacíos o serán modificados.

```
from jira import JIRA

# dedicated Jira server
jiraOptions = {'server': ""}
# mail, token
jira = JIRA(options=jiraOptions, basic_auth=("",
↪   "")

for singleIssue in
↪   jira.search_issues(jql_str='project = TCCS
↪   AND "epic link" = "TCCS-68"'):
    # TCCS-68 is the epic of my project
    print('{}: {}:{}'.format(singleIssue.key,
    ↪   singleIssue.fields.summary,
    ↪   singleIssue.fields.reporter.displayName))
```

En este bloque de código hay representada la conexión más básica a nuestro proyecto. El propósito de estas líneas es que nos devuelva los datos de las instancias o *issues* de nuestro proyecto, concretamente su identificador, su nombre y la persona que la ha creado, siendo filtradas por una épica específica creada para este proyecto. Si hubiéramos utilizado

el método con la REST API tendríamos que haber importado las librerías pertinentes en lugar de la que se ha importado en este bloque, antes de la petición tendríamos que haber creado la base para la conexión HTTP, y después, enviar la petición y gestionar su respuesta para extraer los datos.
Por tanto, aunque a efectos prácticos no haya una gran diferencia entre ambos métodos, consideramos que la utilización de la librería dedicada es más sencilla debido a los pequeños detalles que los diferencian, como puede ser la cantidad de código necesario para un mismo resultado y la complejidad de este.

3) Progreso y revisión de la organización

Como podemos observar en la figura del diagrama de Gantt, a fecha de la entrega, estamos en el periodo temporal comprendido por la tarea 3.5, ahora es cuando empieza la mayor carga de trabajo a nivel de programación. Consideramos que la organización temporal establecida hasta la fecha es correcta.
Hemos modificado también la duración de la tarea 3.7, hemos alargado su tiempo ya que creemos que es posible que nos lleve un poco más de lo previsto. Confiamos en que si el desarrollo del proyecto progresa de manera constante sin demasiados obstáculos, como hasta ahora, la nueva organización prevista será adecuada, y la entrega del segundo informe coincidirá con la fase final de testing del código.

B. Segunda fase de desarrollo

1) Finalización de conexiones

En la primera fase de desarrollo ya conseguimos realizar una conexión al sistema de información Jira, en esta segunda hemos proseguido desarrollando el código. Hemos analizado como conectarnos a GitLab, y hemos replanteado los siguientes pasos a causa de las diferentes opciones que se nos presentaban.
La manera de conectarnos a GitLab recuerda mucho al método seguido para conectarnos a Jira: es necesario instalar la librería de GitLab para Python con la herramienta *pip*, tenemos que indicar en el código la URL corporativa del servidor que aloja GitLab y un token privado de acceso. Una vez establecida la conexión podemos solicitar prácticamente cualquier elemento alojado en este servidor, hasta incluso podemos exportar un proyecto.

```
import gitlab
import time
import os

gl = gitlab.Gitlab('', private_token='')
# server, token

# project = gl.projects.get('')
# select project by name

project = gl.projects.get( )
# select by numerical ID

project.pprint()

print("-----------------------------------")

for commit in
↪   project.commits.list(ref_name='main'):
    # branch name
    commit.pprint()

export = project.exports.create()
export.refresh()
while export.export_status != 'finished':
    time.sleep(1)
    export.refresh()

with open('./export.tgz', 'wb') as f:
    export.download(streamed=True,
    ↪   action=f.write)
```

En este ejemplo seleccionamos un proyecto, usando su ID numérico, que hemos ocultado, e imprimimos en pantalla los datos de este proyecto. En la segunda parte del código realizamos una exportación del mismo.

2) Opciones para el cruce de datos

Una vez finalizado este script inicial nos reunimos el conjunto de personas involucradas en el proyecto: el mánager encargado de supervisar y dar soporte, un compañero para dar soporte y yo. El siguiente paso es recoger los datos que nos interesan para poder cruzar la información, tendremos que buscar en el código del proyecto deseado los identificadores de las instancias de Jira para recuperar los tests incluidos en el proyecto para resolver dichas instancias, pero para ello tenemos diferentes opciones disponibles:

- Buscar en el código un *string* que concuerde con el identificador de la instancia de Jira mediante las herramientas provistas por la librería de GitLab para Python.
- Exportar o descargar el código del proyecto y buscar el identificador con las herramientas básicas de Python. Una alternativa similar sería crear un objeto dinámico que contuviera todos los datos del proyecto.
- Aprovechar los artefactos[11] generados en los *stages* anteriores de nuestro *pipeline* antes de las etapas específicas de nuestra funcionalidad, solo funcionará con aquellos proyectos a los que se la añadamos.

El problema de la primera opción es que, aunque resulta muy útil y sencilla, requiere que la búsqueda de un elemento en el servidor tenga un *scope*[12] y para poder usar el *scope* que busca en el código necesitamos incluir ElasticSearch[13]. Si dispusiéramos de esta herramienta, sería equivalente a realizar una búsqueda con el buscador proporcionado en la interfaz web de GitLab.
La segunda opción es viable, pero en comparación a lo que implica la tercera, o la primera, es más compleja, tediosa y larga.
Analizando las opciones, llegamos a la conclusión que la mejor opción a nuestra disposición es la última. Teniendo en cuenta que en cada proyecto se están generando un *output* de archivos que contienen parte de la documentación que necesitamos, no nos hace falta buscar en el código los mismos datos otra vez, sino que simplemente podemos pasar este *output* como un conjunto de datos para nuestro *stage* y ahorrar trabajo de cómputo. Para poder proseguir con el desarrollo de esta funcionalidad, ha sido necesario crear un repositorio de prueba en GitLab. Contábamos con hacer ciertas tareas más adelante, pero hemos tenido que reorganizar, lo comentaremos en la siguiente sección.

Hemos creado un repositorio de pruebas y en él hemos alojado los dos scripts con los códigos de conexión. Esta ejecución tendrá que ser automatizada hasta cierto punto, por tanto, será necesario un *pipeline*, para crearlo tendremos que añadir un archivo con el nombre `.gitlab-ci.yml`[14]. Este archivo contiene los diferentes datos, fases y configuraciones con los que dotaremos a nuestro *pipeline*. Cuando GitLab detecta la existencia del archivo en el proyecto genera el *pipeline* automáticamente.

```yaml
stages:
  - build
  - test

default:
  image: python:latest
  before_script:
    - pip install --upgrade jira
    - pip install --upgrade python-gitlab

build:
  stage: build
  script:
    - python --version
    - pip list

test:
  stage: test
  needs: [build]
  # rules:
    # when: manual
  script:
    - echo "Este es el stage de test"
    - python --version
    - pip list
```

En este *pipeline* estamos definiendo dos *stages*: *build* y *test*. Como podemos observar, antes de las dos etapas que hemos mencionado hay una etapa llamada *default*, esta se encarga de ejecutar su contenido antes de cada etapa existente. Para poder ejecutar los scripts que hemos incluido hace falta que la imagen que vamos a utilizar tenga Python, es por ese motivo que incluimos la línea `image: python:latest`, en ella indicamos que se realice un *pull* de una imagen básica y oficial de Docker Hub con la última versión de Python[15]. En los dos *stages* siguientes estamos comprobando que Python está instalado en cada etapa, además, comprobamos que en la lista de librerías se incluyen las dos que nos interesan. La línea `needs: [build]` define que para que se ejecute la etapa *test* es necesario que la etapa *build* haya finalizado, por tanto, se está forzando una ejecución secuencial.
Para crear los artefactos[11] con el formato de reporte Allure[16] necesario para simular los datos de entrada que recibiríamos en un proyecto real tenemos que crear un script que contenga tests que generen un *output* en dicho formato. Ahondaremos en ello en la siguiente fase de desarrollo.

3) Revisión de la organización

Como podemos ver en lo comentado en las secciones anteriores, ha habido un replanteamiento y una reorganización de las tareas en esta fase de desarrollo, esto se debe a que como hemos modificado la manera en la que trabajaremos con los datos de entrada, necesitamos, antes de lo planeado, un repositorio y un conjunto de pruebas que nos permitan simular cómo será una situación real, es por este motivo que las tareas 8 y 9, que estaban previstas de ser realizadas más adelante, pasan a ejecutarse paralelamente con la 7, empezando y finalizando las impares de manera simultánea, podríamos incluso fusionarlas y considerarlas la misma tarea.

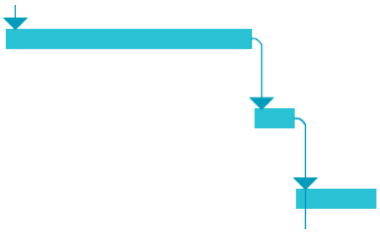


Figura 1: Tareas 7, 8 y 9 antes de la reorganización.

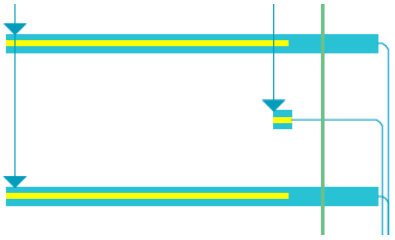


Figura 2: Tareas 7, 8 y 9 después de la reorganización.

Debido a esta reorganización, se atrasa unos días la fase de testeo de la herramienta y, por ende, la finalización total del proyecto. Por lo que progreso del trabajo respecta, se está progresando adecuadamente y no prevemos muchos obstáculos que impidan la finalización del proyecto en la nueva fecha establecida.
Otro cambio en la planificación del proyecto es que debido a la precariedad de HP ALM dentro de la empresa a día de hoy se ha decidido que no hará falta interactuar con este sistema, ya que además se está organizando una migración de datos a otro sistema.

C. Tercera y última fase de desarrollo

En la fase de desarrollo anterior hemos analizado la dirección que estábamos tomando con el proyecto y hemos decidido cambiar el método que usará nuestra herramienta para resolver el problema.
Al principio, el planteamiento ingeniado era que se realizara una conexión a Jira para recuperar los identificadores de las instancias y otra para conectarnos al repositorio alojado en GitLab y finalmente buscar dichos identificadores en el código para generar el *output* deseado. En una de las múltiples revisiones de progreso del proyecto se mencionó que en algunos *stages* de los *pipelines* de los proyectos se generan artefactos como output de los tests, así que, el cambio que implementamos es que en lugar de conectarnos al servidor propietario y buscar en el código del repositorio, recuperaremos los artefactos y buscaremos en ellos los datos que necesitamos.

1) Adaptación del entorno de trabajo

No se está produciendo la herramienta sobre un repositorio existente ya funcional, por tanto es necesario simular una versión simplificada en nuestro repositorio adaptado. Para ello

tendremos que crear un script de Python que contenga tests enlazados con instancias de Jira y generen un *output* con formato Allure. El fichero con los tests contiene el siguiente código:

```python
import sys
import allure
import pytest

class TestClass:
    @allure.story('TCCS-83: Test Assert True')
    @allure.link(link, 'TCCS-83')
    @allure.title("Assert that True is True")
    def test_true(self):
        # Returns True or False.
        assert True == True

    @allure.story('TCCS-84: Test Assert String')
    @allure.link(link, 'TCCS-84')
    @allure.title("Assert various string operations")
    def test_strings_a(self):
        # Returns True if the string contains 4 a.
        assert 'a'*4 == 'aaaa'
        assert 'a' in 'abcd'

    @allure.story('TCCS-87: Test Upper Method')
    @allure.link(link, 'TCCS-87')
    @allure.title("Assert upper method")
    def test_upper(self):
        # Returns True if the string is in upper case.
        assert 'foo'.upper() == 'FOO'

    @allure.story('TCCS-88: Test isupper Method')
    @allure.link(link, 'TCCS-88')
    @allure.title("Assert isupper method")
    def test_isupper(self):
        # Returns True if the string is in uppercase
        # else returns False.
        assert 'FOO'.isupper() == True
        assert'Bar'.isupper() != True
if __name__ == '__main__':
    pytest.main(['--alluredir=./allure-results'])
```

Este código contiene una clase con tests que han sido definidos con el paquete de *pytest*[17], y cada uno de los tests está etiquetado con las herramientas del paquete especializado de Allure y nos permitirá darle el formato deseado al resultado. Para la correcta ejecución de este script es necesario instalar los dos paquetes mencionados con la herramienta *pip*. Como *output* obtenemos unos ficheros con extensión *json* con nuestro formato y cada uno de ellos contiene diferentes datos de un test concreto, se almacenan todos en un directorio nuevo con el nombre *allure-results*. El siguiente bloque muestra uno de los archivos que obtenemos como resultado y que será uno de los artefactos.

```json
{
    "name": "Assert that True is True",
    "status": "passed",
    "start": 1654008838795,
    "stop": 1654008838795,
    "uuid": "uuid",
    "historyId": "historyId",
    "testCaseId": "testCaseId",
    "fullName": "test_class.TestClass#test_true",
    "labels":
    [
        {
            "name": "story",
            "value": "TCCS-83: Test Assert True"
        }, "resto de etiquetas"
    ],
    "links":
    [
        {
            "type": "TCCS-83",
            "url": "link",
            "name": "link"
        }
    ]
}
```

Ahora que ya tenemos una simulación simplificada del *input* que tendremos en un caso real, podemos proceder con el desarrollo del código que cruzará los datos y generará los documentos con la matriz de trazabilidad y los resultados de los tests.

```python
import os
import sys
import getopt
from jira import JIRA

argList = sys.argv[1:]
options = "hr:"
release = None

try:
    arguments, values = getopt.getopt(argList, options)

    for currentArgument, currentValue in arguments:
        if currentArgument == "-h":
            print("python fetch_jira_keys.py -r <release>")
            sys.exit()
        elif currentArgument == ("-r"):
            release = currentValue
except getopt.error as e:
    print(str(e))

if release is not None:
    jiraOptions = {'server': ""}
    # mail, token
    jira = JIRA(options=jiraOptions, basic_auth=("", ""))
    f = open("jirakeys.txt", "w")

    jql_string = 'project = TCCS AND fixVersion = ' + release + ' ORDER BY Rank ASC'
    # DynamicDocGen-0.0.1

    for singleIssue in jira.search_issues(jql_str=jql_string):
        # TCCS-68 it's the epic of my project
        f.write('{}: {}\n'.format(singleIssue.key, singleIssue.fields.summary))

    f.close()
else:
    print("Release is missing.")
```

Hemos modificado el código de la conexión a Jira y recuperación de datos de este sistema para que el resultado se escriba en un documento nuevo al que podremos acceder más adelante cuando queramos generar la documentación. Este nuevo documento que genera la conexión a Jira contiene el

identificador junto con el nombre de las instancias de una *release* de un proyecto concreto. Además hemos cambiado el código para que a través de un parámetro indiquemos cual es la *release* con la que nos interesa trabajar.

```
import glob
import json
import pandas

file = open("jirakeys.txt", 'r')
issues = file.readlines()
issues = [line[:-1] for line in issues]
table = dict.fromkeys(issues)

for jsonPath in glob.glob('allure-results/*-result.json'):
    with open(jsonPath, encoding="UTF-8") as jsonFile:
        allureResult = json.load(jsonFile)
        allureName = allureResult['labels'][0]['value']

    if allureName in table:
        table[allureName] = allureResult

# this is to filter those issues that do not have tests
table = {x: table[x] for x in table if table[x] is not None}
issues = list(table.keys())
#issuesSplit = [line.split(": ") for line in issues]

f = open("traceability_matrix.md", "w")
f.write("# Traceability Matrix\n")
f.write("| Jira ID | Title | Test Case ID | Full Name |\n")
f.write("| ::---:: | ::---:: | ::---:: | ::---:: |\n")
for jiraData in issues:
    f.write("| " + jiraData.split(": ")[0] + " | " + "[" + jiraData.split(": ")[1] + "](" + table[jiraData]['links'][0]['url'] + ")" + " | " + table[jiraData]['testCaseId'] + " | " + table[jiraData]['fullName'] + " |\n")
f.close()

f = open("test_results.md", "w")
f.write("# Test Results\n")
f.write(pandas.DataFrame
        .from_dict(table).to_markdown())
f.close()
```

En el código anterior abrimos el archivo generado por el encargado de recuperar las instancias de Jira de una *release* concreta. Para evitar que se tengan que abrir todos los archivos múltiples veces se crea un diccionario vacío en contenido en el que como *keys* tendremos un *string* que contendrá el identificador y el nombre de cada instancia, el valor de cada *key* se irá llenando a medida que vayamos leyendo los artefactos y será el objeto *json* correspondiente. Es importante tener en cuenta que por defecto los artefactos se mantienen entre *stages* y no nos interesan todos los tipos que hay, así que dentro de la carpeta *allure-results* filtraremos los archivos y solo nos interesarán aquellos que su ruta siga la siguiente expresión regular: `allure-results/*-result.json`, que significa que son objetos de resultado del test. Una vez tenemos el diccionario lleno con todos los objetos posibles, eliminaremos las entradas que no tengan contenido, ya que no todas las instancias de Jira serán cubiertas por tests. Como *output* generaremos dos documentos: la matriz de trazabilidad; el primer archivo que escribimos, que será una tabla que combinará dos datos de Jira con dos datos del objeto resultado del test Allure; y una tabla de los datos y resultados, correspondiente al segundo archivo que creamos en el código, que introducirá todos los datos de los objetos *json* en una tabla. Los documentos generados son de tipo *Markdown*[18]. Una vez ejecutemos el proyecto, se creará un artefacto descargable que contendrá los dos archivos *Markdown* con la documentación que antiguamente tenía que realizarse manualmente.

Para hacer funcional el sistema en nuestra integración continua deberemos adaptar el fichero `.gitlab-ci.yml` para corresponderlo con las situaciones en las que trabajará.

```
stages:
  - build
  - test
  - docgen

variables:
 RELEASE:
   value: "" # this would be the default value
   description: "This is the release you want to generate documentation for"
   # makes this variable appear on the Run Pipeline

default:
  image: python:latest
  before_script:
    - pip install --upgrade jira
    - pip install --upgrade python-gitlab
    - pip install --upgrade pytest
    - pip install --upgrade allure-pytest
    - pip install --upgrade pandas
    - pip install --upgrade tabulate

build:
  stage: build
  script:
    - python --version
    - pip list

create_test:
  stage: test
  needs: [build]
  script:
    - python test_class.py
  artifacts:
    name: 'allure_results'
    paths:
      - ./allure-results

docgen:
  stage: docgen
  needs: [create_test]
  rules:
    - if: $RELEASE != ""
    # it will only execute this stage when added a value to the variable RELEASE manually.
  script:
    - python fetch_jira_keys.py -r $RELEASE
    - python docgen.py
  artifacts:
    name: 'traceability_${RELEASE}'
    paths:
      - ./traceability_matrix.md
      - ./test_results.md
```

En comparación con el archivo `.gitlab-ci.yml` origi-

nal, observamos ciertas diferencias: aparece una nueva sección llamada `variables`[19], en esta se encuentran un conjunto de variables que hemos predefinido y a las que podemos otorgar un valor predeterminado, para que cuando se ejecute el *pipeline* de manera manual aparezca esta variable; en la fase previa a los *stages* se han añadido diferentes comandos que se encargarán de instalar los paquetes que necesitaremos en nuestra máquina. La primera etapa se mantendrá igual y en la segunda encontraremos que hemos cambiado el nombre de la etapa de test; hemos añadido una última etapa que hemos llamado *docgen*, se encargará de buscar las instancias de Jira y crear la documentación. Hemos limitado esta etapa para que solo se ejecute cuando la variable `RELEASE` no esté vacía, por tanto, esta etapa no se ejecutará cuando se produzca un nuevo *commit* o *merge* del código, solo cuando se ejecute de manera manual el *pipeline*, que es cuando se permite dar un valor a la variable antes de su ejecución. Ahora, para recuperar las instancias, tendremos que indicar con un argumento en el comando de ejecución del código qué *release* nos interesa recuperar, esta será indicada cuando vayamos a ejecutar el *pipeline* de manera manual mediante la variable con el mismo nombre. Luego se generará un artefacto con los dos archivos de documentación generados.

2) Revisión de la planificación

No han habido más cambios en las tareas que teníamos planificadas, y en cuanto el margen temporal que hemos calculado es adecuado, ya que las tareas han finalizado cuando habíamos previsto.

D. Resultados

Este proyecto estaba enfocado como un *spike*, un proyecto de prueba que nos permitiera ver si realizar el sistema de documentación automática que deseábamos era viable. Actualmente disponemos de una versión funcional que puede adaptarse a otros proyectos para probar su funcionamiento. En las siguientes figuras podemos ver los resultados que obtenemos con nuestro proyecto, que sirven como ejemplo de cómo será el producto final para un caso real.

Traceability Matrix

Jira ID	Title	Test Case ID	Full Name
TCCS-88	Test isupper Method	[redacted]	test_class.TestClass#test_isupper
TCCS-87	Test Upper Method	[redacted]	test_class.TestClass#test_upper
TCCS-84	Test Assert String	[redacted]	test_class.TestClass#test_strings_a
TCCS-83	Test Assert True	[redacted]	test_class.TestClass#test_true

Figura 3: Matriz de trazabilidad obtenida como *output*.

E. Siguientes pasos

El sistema es perfectamente funcional a pequeña escala, el siguiente paso será implementarlo en proyectos de mayor escala. Además, se puede estudiar y revisar el código para intentar hacerlo más rápido y eficiente.

Test Results

	TCCS-88: Test isupper Method	TCCS-87: Test Upper Method	TCCS-84: Test Assert String	TCCS-83: Test Assert True
name	Assert isupper method	Assert upper method	Assert various string operations	Assert that True is True
status	passed	passed	passed	passed
start	1654706344576	1654706344574	1654706344572	1654706344570
stop	1654706344576	1654706344574	1654706344572	1654706344570
uuid	[redacted]	[redacted]	[redacted]	[redacted]
historyId	[redacted]	[redacted]	[redacted]	[redacted]
testCaseId	[redacted]	[redacted]	[redacted]	[redacted]
fullName	test_class.TestClass#test_isupper	test_class.TestClass#test_upper	test_class.TestClass#test_strings_a	test_class.TestClass#test_true
labels	[{'name': 'story', 'value': 'TCCS-88: Test isupper Method'}, {'name': 'suite', 'value': 'test_class'}, {'name': 'subSuite', 'value': 'TestClass'}, {'name': 'host', 'value': 'runner-vfiezbng-project-160207-concurrent-0'}, {'name': 'thread', 'value': '52-MainThread'}, {'name': 'framework', 'value': 'pytest'}, {'name': 'language', 'value': 'cpython3'}, {'name': 'package', 'value': 'test_class'}]	[{'name': 'story', 'value': 'TCCS-87: Test Upper Method'}, {'name': 'suite', 'value': 'test_class'}, {'name': 'subSuite', 'value': 'TestClass'}, {'name': 'host', 'value': 'runner-vfiezbng-project-160207-concurrent-0'}, {'name': 'thread', 'value': '52-MainThread'}, {'name': 'framework', 'value': 'pytest'}, {'name': 'language', 'value': 'cpython3'}, {'name': 'package', 'value': 'test_class'}]	[{'name': 'story', 'value': 'TCCS-84: Test Assert String'}, {'name': 'suite', 'value': 'test_class'}, {'name': 'subSuite', 'value': 'TestClass'}, {'name': 'host', 'value': 'runner-vfiezbng-project-160207-concurrent-0'}, {'name': 'thread', 'value': '52-MainThread'}, {'name': 'framework', 'value': 'pytest'}, {'name': 'language', 'value': 'cpython3'}, {'name': 'package', 'value': 'test_class'}]	[{'name': 'story', 'value': 'TCCS-83: Test Assert True'}, {'name': 'suite', 'value': 'test_class'}, {'name': 'subSuite', 'value': 'TestClass'}, {'name': 'host', 'value': 'runner-vfiezbng-project-160207-concurrent-0'}, {'name': 'thread', 'value': '52-MainThread'}, {'name': 'framework', 'value': 'pytest'}, {'name': 'language', 'value': 'cpython3'}, {'name': 'package', 'value': 'test_class'}]
links	[{'type': 'TCCS-88', 'url': [redacted] 'name': [redacted]}	[{'type': 'TCCS-87', 'url': [redacted] 'name': [redacted]}	[{'type': 'TCCS-84', 'url': [redacted] 'name': [redacted]}	[{'type': 'TCCS-83', 'url': [redacted] 'name': [redacted]}

Figura 4: Tabla de resultados de los tests obtenida como *output*.

VI. Conclusiones

La documentación relativa a los tests necesaria después de la ejecución de un proyecto es sencilla de hacer, pero debido al alto volumen de tests que se implementan, esta tarea cada vez resulta más tediosa y extensa, y por tanto cada vez requiere de más tiempo o de más mano de obra que la que se sería la óptima, es por eso que una herramienta que genere la documentación de manera automática es más que práctica e incluso necesaria. Hemos analizado nuestras necesidades, cuál es el entorno en el que trabajamos, definido cuales son los *inputs* del sistema y cuáles son los *outputs* que debe generar. El propósito del proyecto era comprobar la viabilidad de la creación de la herramienta y su posterior implementación, y podemos concluir que el proyecto dispone de una alta viabilidad, que con una correcta implementación puede resultar muy práctico y que con un seguimiento adecuado puede mejorarse el producto final.

Referencias

[1] Atlassian. (2022) Una breve presentación de jira. https://www.atlassian.com/es/software/jira/guides/getting-started/overview.

[2] Micro-Focus. (2021) Alm: Application lifecycle management & quality center. https://www.microfocus.com/es-es/products/alm-quality-center/overview.

[3] GitLab. (2022) Unify the devops lifecycle with gitlab. https://about.gitlab.com/stages-devops-lifecycle/.

[4] Atlassian. (2022) ¿qué es scrum? https://www.atlassian.com/es/agile/scrum.

[5] Apache. (2022) What is maven? https://maven.apache.org/what-is-maven.html.

[6] Gradle. (2022) What is gradle? https://docs.gradle.org/current/userguide/what_is_gradle.html.

[7] Atlassian. (2022) Jira rest api examples. https://developer.atlassian.com/server/jira/platform/jira-rest-api-examples/.

[8] OpenSource. (2021) Jira python library. https://github.com/pycontribs/jira.

[9] ——. (2022) Python jira. https://jira.readthedocs.io/.

[10] Atlassian. (2022) Jql: Ponte en marcha con la búsqueda avanzada y jql. https://www.atlassian.com/es/software/jira/guides/expand-jira/jql.

[11] GitLab. (2022) Job artifacts. https://docs.gitlab.com/ee/ci/pipelines/job_artifacts.html.

[12] ——. (2022) Search api. https://docs.gitlab.com/ee/api/search.html.

[13] ——. (2022) Advanced search api, scope: Blobs. https://docs.gitlab.com/ee/api/search.html#scope-blobs.

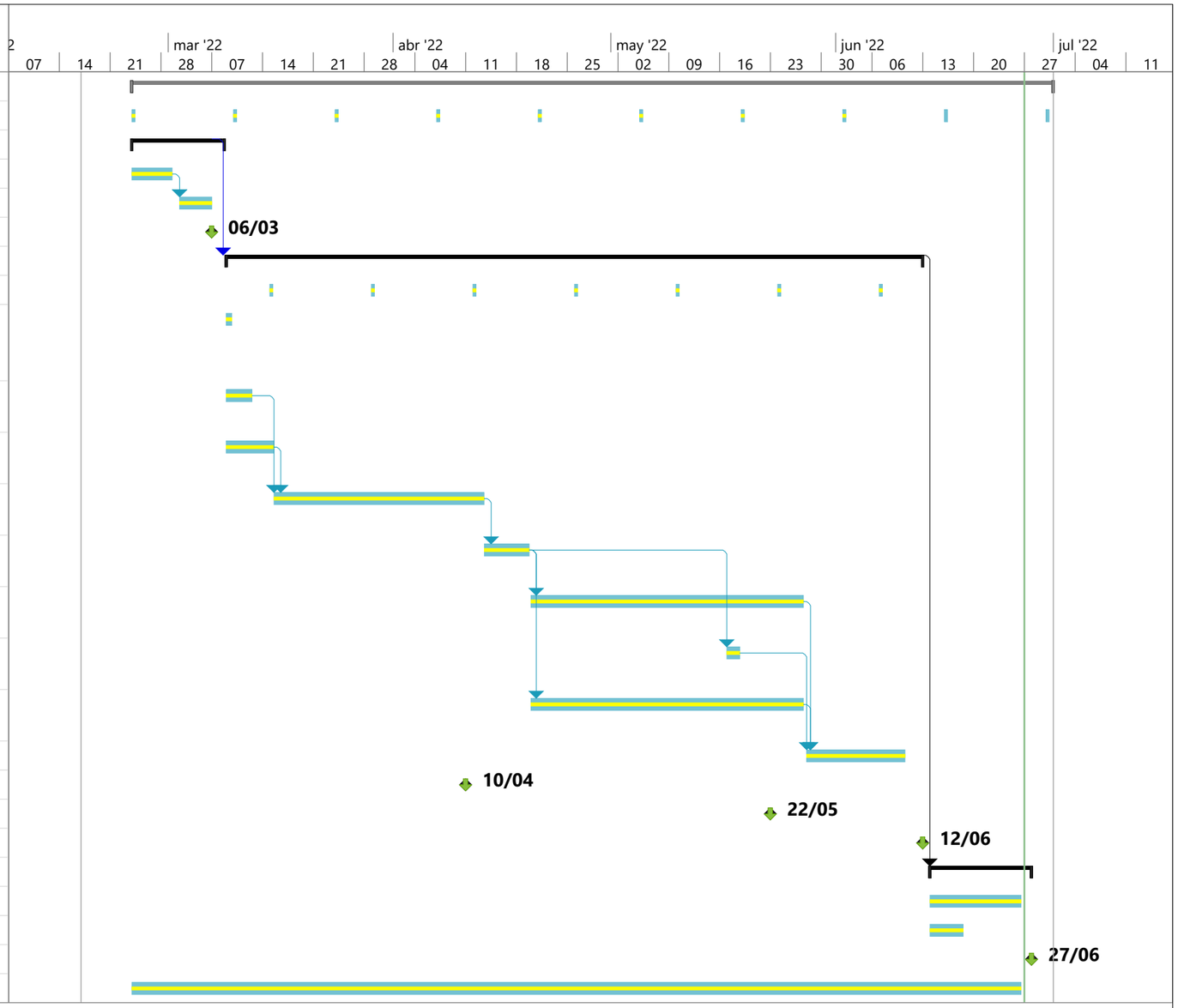
[14] ——. (2022) The .gitlab-ci.yml file. https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html.

[15] D. Inc. (2022) Python docker official image. https://hub.docker.com/_/python.

[16] Allure. (2020) Allure framework. https://docs.qameta.io/allure/.

[17] H. Krekel and pytest-dev team. (2015) pytest: helps you write better programs. https://docs.pytest.org/en/7.1.x/.
[18] M. Cone. (2022) Getting started: An overview of markdown, how it works, and what you can do with it. https://www.markdownguide.org/getting-started/.
[19] GitLab. (2022) Gitlab ci/cd variables. https://docs.gitlab.com/ee/ci/variables/.

Id	Nombre de tarea	Duración	Comienzo	Fin
0	**TFG Documentación automática**	**64 días?**	**jue 24/02/22**	**jue 30/06/22**
1	**1 Reuniones Seguimiento**	**91 días**	**jue 24/02/22**	**jue 30/06/22**
12	**2 Fase Inicial**	**7 días?**	**jue 24/02/22**	**mar 08/03/22**
13	2.1 Redacción Borrador Informe Inicial	3 días?	jue 24/02/22	mar 01/03/22
14	2.2 Redacción Informe Inicial	2 días?	mié 02/03/22	dom 06/03/22
15	2.3 Informe Incial	0 días	dom 06/03/22	dom 06/03/22
16	**3 Fase Desarrollo**	**48 días?**	**mié 09/03/22**	**dom 12/06/22**
17	**3.1 Reuniones Seguimiento con Mánager**	**45 días**	**mié 09/03/22**	**mar 07/06/22**
25	3.2 Establecer mi organización del proyecto en Jira para usarlo como conjunto de pruebas	1 día?	mié 09/03/22	mié 09/03/22
26	3.3 Estudiar como funciona Jira exactamente y cual es su estructura	2 días?	mié 09/03/22	sáb 12/03/22
27	3.4 Investigar como recuperar datos de Jira y GitLab con Python	3 días?	mié 09/03/22	mar 15/03/22
28	3.5 Empezar desarrollo del sistema de recuperación de datos de Jira	15 días?	mar 15/03/22	mié 13/04/22
29	3.6 Ánalisis de recuperación y cruce de datos de tests de GitLab y requerimientos de Jira	3 días?	mié 13/04/22	mar 19/04/22
30	3.7 Continuar desarrollo del sistema e incorporar cruce de datos	19 días?	mié 20/04/22	vie 27/05/22
31	3.8 Establecer formato deseado para el output de documentación e implementarlo	2 días?	mar 17/05/22	mié 18/05/22
32	3.9 Empezar la implementación del sistema en pipeline de GitLab	19 días?	mié 20/04/22	vie 27/05/22
33	3.10 Testing del código	7 días?	sáb 28/05/22	vie 10/06/22
34	3.11 Informe de progreso 1	0 días	dom 10/04/22	dom 10/04/22
35	3.12 Informe de progreso 2	0 días	dom 22/05/22	dom 22/05/22
36	3.13 Borrador Informe Final	0 días	dom 12/06/22	dom 12/06/22
37	**4 Fase Final**	**7 días?**	**mar 14/06/22**	**lun 27/06/22**
38	4.1 Redacción informe final	7 días?	mar 14/06/22	dom 26/06/22
39	4.2 Redacción póster	3 días?	mar 14/06/22	sáb 18/06/22
40	4.3 Entrega Final TFG	0 días	lun 27/06/22	lun 27/06/22
41	**5 Dosier del TFG**	**62 días**	**jue 24/02/22**	**dom 26/06/22**



Proyecto: TFG Documentación
Fecha: lun 27/06/22

Tarea	Resumen del proyecto	Tarea manual	solo el comienzo	Fecha límite
División	Tarea inactiva	solo duración	solo fin	Progreso
Hito	Hito inactivo	Informe de resumen manual	Tareas externas	Progreso manual
Resumen	Resumen inactivo	Resumen manual	Hito externo	