ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA INFORMÁTICA

Curso Académico 2020/2021

Trabajo Fin de Grado

CONSTRUCCIÓN DE DATASET DE PRUEBAS END-TO-END DE PROYECTOS GITHUB

Autor: Jorge Contreras Padilla

Directores: Francisco Gortázar Bellas
Michel Maes Bermejo

Resumen

En este proyecto se aborda la elaboración de un DataSet de proyectos que contengan pruebas end-to-end, aplicando minería de repositorios sobre la plataforma GitHub.

Para ello se elabora un heurístico en base a una serie de criterios encargados de localizar pruebas de este tipo. Este heurístico se implementa mediante el lenguaje de programación Python y se puede consultar en la siguiente url: https://github.com/jorcontrerasp/BuscadorE2eGitHubRepos.

Palabras clave

- Testing.
- End-to-end tests.
- Integration tests.
- E2e.

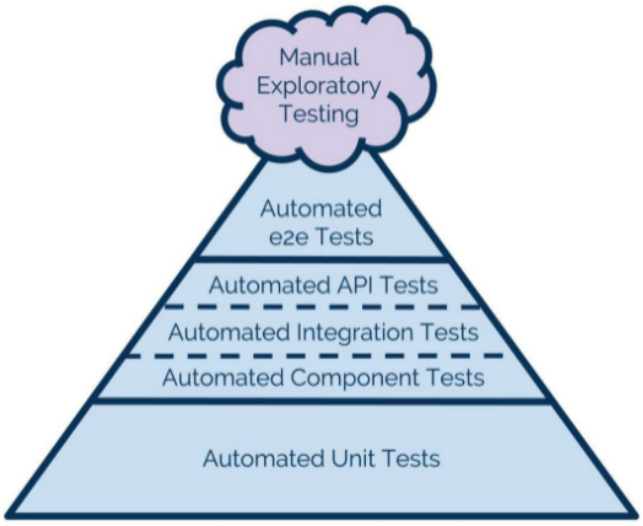
Índice

1. Introducción

La importancia del software y los costes asociados a un fallo motivan la necesidad de probar lo que se está construyendo por lo que desarrollar una batería de pruebas de forma minuciosa y bien planificadas a la hora de implementar cualquier desarrollo software es de suma importancia, ya que es la forma de garantizar que nuestro negocio está asegurado en cierto modo frente a diferentes fallos que se puedan dar.

Mike Cohn propuso una clasificación de tests automatizados conocida como la pirámide de tests o la pirámide de Cohn.
En la parte inferior de la pirámide se encuentran los tests unitarios los cuales representan la parte más extensa de la pirámide de automatización. Estos tests brindan un feedback muy específico y rápido.

En el otro extremo se encuentran los tests de usuario que prueban la aplicación punta a punta a través de la interfaz de usuario. Estos son también conocidos con el nombre de end-to-end. Generalmente se querrán tener pocos tests de este tipo en cualquier desarrollo software debido a su fragilidad y su alto coste de mantenimiento.



Esto ha provocado que apenas se tenga mucha información sobre la implementación de este tipo de tests de integración.

En este trabajo se pretende generar un DataSet con proyectos de código abierto “open source” alojados en la plataforma de GitHub concretamente con pruebas de integración y/o end-to-end (e2e).

Una vez obtenido el conjunto de repositorios se podrán realizar estudios más completos sobre este tipo de pruebas y verificar la existencia de algún tipo de convenio entre los programadores a la hora de estructurar las pruebas e2e dentro de la jerarquía de ficheros del proyecto.

Objetivos:

- Construir un DataSet de repositorios GitHub con pruebas end-to-end.
- Expandir el lenguaje de programación Python.
- Aprendizaje en técnicas de “research” como la minería de repositorios.
- Afianzar conocimientos sobre pruebas de software y en concreto sobre pruebas end-to-end.

2. Tecnologías y herramientas utilizadas.

2.1. Git

Git es un sistema de control de versiones distribuido, de código abierto y gratuito diseñado para manejar todo relacionado con desarrollo de software, desde proyectos pequeños a muy grandes, con velocidad y eficiencia.

Desarrollado inicialmente por Linus Torvalds, el famoso creador del kernel del sistema operativo Linux, en 2005, hoy en día es, con diferencia, el sistema de control de versiones moderno más utilizado del mundo.

2.2. API de GitHub

GitHub es una compañía sin fines de lucro que ofrece un servicio de hosting de repositorios almacenados en la nube utilizando el sistema de control de versiones Git.
Además, cuenta con una API REST disponible para cualquier desarrollador que quiera implementar alguna aplicación relacionada con el servicio que ofrece.

En este caso, la aplicación programada para buscar proyectos que puedan tener pruebas end-to-end utiliza el lenguaje de programación Python y, concretamente, la librería “PyGithub” que permite utilizar la versión 3 de la API ya mencionada.

2.3. Python

Es un lenguaje de programación cuya filosofía hace hincapié en la legibilidad del código. Sus principales características son las siguientes:

- Multiparadigma: ya que más que forzar a los programadores a adoptar un estilo de programación, permite varios estilos, soportando la orientación a objetos, la programación imperativa y la funcional.
- Interpretado.
- Dinámico: permitiendo que una variable pueda tomar valores de distinto tipo.
- Multiplataforma.

Se utiliza en el trabajo para implementar la aplicación que se encargará de recoger repositorios de GitHub y aplicarles los criterios que conforman el heurístico de búsqueda de pruebas end-to-end.

2.3.1. Librerías y herramientas Python utilizadas

Para desarrollar la aplicación se han utilizado una serie de librerías y herramientas propias del lenguaje Python, especificadas a continuación:

- **PyGithub**: librería que facilita el uso de la API de GitHub v3. Permite la gestión de diferentes recursos de GitHub (repositorios, perfiles de usuario, organizaciones, etc.) desde scripts de Python.
- **Pandas**: iniciada en 2008, es una librería que pretende ser el bloque de construcción fundamental de alto nivel para realizar análisis de datos prácticos del mundo real en Python. Además, tiene el objetivo más amplio de convertirse en la herramienta de análisis/manipulación de datos de código abierto más potente y flexible disponible en cualquier idioma.

  Nos va a permitir manejar DataFrames y convertir la información que queramos tanto en formato excel como en formato csv para su posterior estudio.

- **PyMysql**: el paquete PyMySQL contiene una biblioteca cliente de MySql sobre Python, basada en PEP 249. Nos va a permitir conectar una aplicación escrita en Python con una base de datos donde poder almacenar la información que queramos.
- **Pickle**: importando esta librería en el proyecto vamos a poder almacenar la información que queramos en un fichero binario de Python. En este caso, una vez obtenidos los repositorios a analizar se almacenarán en un fichero de este tipo para poder reutilizar esos repositorios en posteriores ejecuciones.
- **Tkinter**: el paquete tkinter es la interfaz estándar de Python. Nos permitirá implementar una interfaz de usuario básica con la que poder lanzar el proceso, realizar pruebas y consultar los repositorios almacenados en base de datos.
- **Pillow**: es la biblioteca de imágenes de Python. Va a ser la herramienta auxiliar de la biblioteca tkinter y nos va a permitir agregar capacidad de procesamiento de imágenes al intérprete de Python.
- **Subprocess**: herramienta que nos va a permitir llevar a cabo la ejecución de diferentes comandos de manera sencilla, como si fuese una ventana "cmd" o un terminal de ejecución.
- **Base64**: este módulo proporciona funciones para codificar datos binarios en caracteres ASCII imprimibles y decodificar dichas codificaciones en datos binarios.

2.4. MySQL

Se trata de un sistema de bases de datos relacional desarrollado bajo licencia dual. Inicialmente desarrollado por MySQL AB fundada por David Axmark, Allan Larsson y Michael Widenius, fue adquirida por Sun Microsystems en 2008 y, a su vez, comprada por Oracle Corporation en 2010.

Es muy utilizado a nivel profesional en aplicaciones web como Joomla, Wordpress, Drupal o phpBB en plataformas LAMP y por herramientas de seguimiento de errores como Bugzilla.

Ideal para utilizar en aplicaciones web con una baja concurrencia de modificación de datos e intensivo en lectura de datos, ya que se trata de una base de datos muy rápida en lectura cuando se utiliza el motor no transaccional MyISAM. No obstante, puede provocar problemas de integridad de los datos si la concurrencia del entorno es alta.

En este proyecto se utiliza MySQL como herramienta para almacenar toda la información que el buscador requiera, así como los datos de configuración que gestionan la búsqueda y los repositorios GitHub que se vayan encontrando en cada ejecución.

2.5. Java

Se trata de un lenguaje de programación, desarrollado originalmente por James Gosling, y una plataforma informática que fue comercializada por primera vez en 1995 por Sun Microsystems. Es rápido, seguro y fiable y actualmente uno de los lenguajes de programación más populares en uso.

En cuanto a su filosofía, Java se creó con cinco objetivos principales:

- Orientación a objetos.
- Independencia de la plataforma: “write once, run everywhere”.
- Soporte para trabajo en red.
- Ejecución de código en sistemas remotos de forma segura.
- Facilidad de uso.

En este trabajo se utiliza el lenguaje de programación Java con el objetivo de implementar una herramienta encargada de ejecutar un fichero “tarea.h” cada x tiempo.
De esta forma se puede ejecutar de forma desatendida el programa que desarrolla la búsqueda y aplicación de criterios sobre los repositorios GitHub, y de esta forma obtener información de forma masiva.

2.6. PyCharm Community

PyCharm se trata de un IDE de Python para desarrolladores creado por la empresa JetBrains. Cuenta con dos versiones del producto, una versión simplificada orientada a la comunidad (PyCharm Community Edition) y otra versión más profesional (PyCharm Professional Edition). Para implementar el programa con el que se va a realizar la búsqueda de repositorios en GitHub se ha utilizado la versión "Community", la cual consta de:

- Editor de Python inteligente.
- Depurador gráfico y ejecutor de pruebas.
- Navegación y refactorización.
- Inspecciones de código.
- Compatibilidad con VCS.

2.7. NetBeans

NetBeans es un entorno de desarrollo integrado libre y gratuito sin restricciones de uso, hecho principalmente para el lenguaje de programación Java.

Desde su lanzamiento por Sun Microsystems, actualmente administrado por Oracle Corporation, en diciembre del 2000 con la versión 3.1, se han ido desarrollando multitud de versiones. Hasta el día de hoy con la versión 12.x iniciada en 2020.

La principal característica de NetBeans es la modularidad, ya que todas las funciones del IDE son provistas por módulos.
Este entorno de desarrollo integrado se utiliza en este trabajo para implementar el proyecto auxiliar "ScriptLapseExe".

2.8. DBeaver

Es una aplicación de software cliente de SQL y una herramienta de administración de bases de datos. Ofrece un editor, el autocompletado de código y el resaltado de sintaxis. Cuenta con tres versiones: Community Edition, software libre y de código abierto lanzada en 2011, Eclipse Plugin Edition, lanzada en 2012, y Enterprise Edition, lanzada en 2014.

Se utiliza en este trabajo para realizar toda la gestión relacionada con la base de datos del buscador: almacenaje de repositorios, búsquedas y datos de configuración.

3. Contexto del trabajo

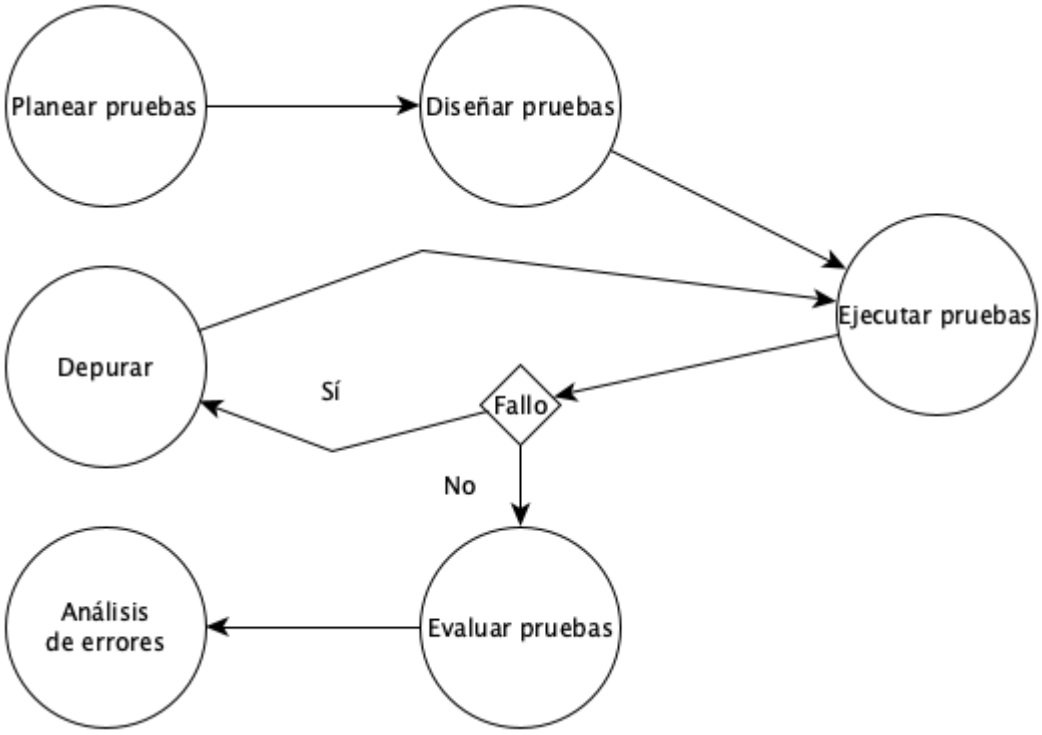
3.1. Principios y características de las pruebas software

Toda prueba software debe tener una serie de características y seguir una serie de principios para que se considere una prueba de calidad. Se podrían destacar las siguientes:

1) La prueba es el proceso de ejecución de un programa con el fin de descubrir un error, con poca cantidad de tiempo y esfuerzo.
2) Un buen caso de prueba es aquel que tiene una alta probabilidad de encontrar un error no descubierto hasta entonces.
3) Una prueba tiene éxito si descubre un error no detectado con anterioridad.
4) Se pueden usar como indicador de fiabilidad y de calidad del software.
5) No garantizan la ausencia de fallos.
6) A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente.
7) Plan de prueba (planificación anterior a la realización): las pruebas deberían planificarse mucho antes de que empiecen.
8) Principio de Pareto: el 80% de todos los errores descubiertos durante las pruebas se les puede hacer un seguimiento hasta un 20% de todos los módulos del programa.
9) Se deben incluir tanto entradas correctas como incorrectas.
10) Deberían empezar por “lo pequeño” y progresar a “lo grande”.
11) Imposibilidad de hacer unas pruebas exhaustivas.
12) Gran efectividad, siempre y cuando se realicen por equipos independientes.
13) Una buena prueba no debe ser redundante.
14) Una buena prueba no debe ser ni demasiado sencilla ni demasiado compleja.
15) Se deben documentar detalladamente.

3.2. Proceso de pruebas de software

El proceso que define los pasos a seguir para ejecutar pruebas software de forma correcta y llevar a cabo la corrección de errores producidos es el siguiente:



3.3. Tipos de pruebas software

A nivel genérico, existen dos tipos de pruebas:

- **Manuales**: son llevadas a cabo por personas que navegan e interactúan con el software. Son costosas ya que requiere contar con un profesional que configure el entorno de desarrollo y, además, están expuestas a errores "humanos", como por ejemplo errores tipográficos u omisión de pasos durante la prueba.
- **Automatizadas**: son realizadas por máquinas que ejecutan ficheros de script previamente configurados. Estos scripts se encargan de verificar si los métodos del programa devuelven un resultado esperado e incluso de asegurar que una secuencia de acciones complejas en la interfaz de usuario se realiza correctamente. Son rápidas, pero dependen de si el fichero de script está bien programado o no y, además, son un componente fundamental en la integración continua de cualquier aplicación.

Y a nivel específico existen otros muchos tipos, a tener en cuenta las pruebas unitarias, de integración, de sistema y de validación.

3.3.1. Pruebas unitarias

Las pruebas unitarias se centran en probar piezas/unidades individuales de una aplicación de software. Existen diferentes enfoques para la construcción de casos de pruebas unitarios, aplicable a cualquier producto de ingeniería: el **enfoque de caja blanca** también conocido con el nombre de caja de cristal, el **enfoque de caja negra** también denominadas pruebas de comportamiento, la **conjetura de errores** cuyo principal objetivo es enumerar una lista de errores comunes para elaborar los casos de prueba en función de ella y, por último, el **enfoque aleatorio**, que se encarga de simular la entrada de datos de la prueba.

3.3.2. Pruebas de integración

La prueba de integración es una técnica sistemática para construir la estructura del programa.
El objetivo es coger los módulos probados mediante la prueba de unidad y construir una estructura de programa que esté de acuerdo con lo que dicta el diseño.
Existen diferentes enfoques de integración, a tener en cuenta, la **integración descendente** "top-down" y la **integración ascendente** "bottom-up".

Un tipo de prueba de integración son las pruebas **end-to-end (e2e)**, que es el tipo de prueba que se va a estudiar más en profundidad en este trabajo puesto que el principal objetivo es recopilar pruebas de este tipo implementadas en proyectos GitHub.

3.3.3. Pruebas end-to-end

La prueba end-to-end es una técnica que busca validar que la combinación de diferentes funcionalidades de un sistema bajo pruebas se comporta como debe ser y no emergen errores al combinar los componentes involucrados replicando el comportamiento de los usuarios con el software, en un entorno de aplicación completo.

Estas pruebas verifican que los flujos que sigue un usuario trabajen como se espera, y pueden ser tan simples como cargar una página web, iniciar sesión, o mucho más complejas, verificando notificaciones vía email, pagos en línea, etc. Es decir, se prueban escenarios que combinan diferentes casos de uso/funcionalidades tal y como lo haría un usuario. Por ejemplo, en una aplicación de gestión de páginas web (CMS) un usuario primero debe autenticarse en el CSME, luego debe ir al módulo de edición de páginas, seleccionar una página existente, agregar el contenido, y al final publicar la página.

El e2e testing permite identificar errores que emergen por:

- la integración de diferentes subsistemas y componentes, y
- flujo de datos y cambios de estados cuando se ejecutan diferentes funcionalidades.

Las pruebas end-to-end son muy útiles, pero son costosas de realizar; y pueden ser difíciles de mantener cuando son automatizadas.

Por tanto, es recomendable tener unas pocas pruebas end-to-end, que resulten claves para nuestra aplicación, y confiar en mayor medida en las pruebas a bajo nivel (como pruebas unitarias y pruebas de integración) para detectar rápidamente aquellos cambios que impactan negativamente sobre nuestra aplicación.

Para implementar pruebas end-to-end se pueden usar herramientas de Record-and-replay, APIs de automatización, y herramientas de pruebas basadas en modelos (estas últimas son más de corte académico):

- **Record-and-replay (R&R)**: como su nombre indica, se orientan a la grabación de escenarios que se ejecutan sobre la aplicación, y luego pueden ser reproducidos automáticamente. La fase de grabación (recording) genera un script con las instrucciones a reproducir; el script puede estar en un lenguaje propio de la herramienta usada, o puede tener instrucciones de una API para automatización.

  Son fáciles de grabar y reproducir. Sin embargo, no funcionan bien si las instrucciones en el script están acopladas a ubicaciones en la pantalla y esta cambia su tamaño.

  Algunos ejemplos de herramientas para R&R son Selenium (web), RERAN (Android), Espresso Test Recorder (Android) y XCode UI Test Recorder (iOS).
- **APIs de automatización**: librerías y frameworks que permiten la automatización de pruebas de GUI, a través de APIs que permiten al programador/tester codificar las pruebas.

  Las pruebas codificadas con estas APIs tienen también el problema de ser frágiles si hay cambios drásticos en la GUI. Adicionalmente, es necesario aprender las APIs, con lo cual se debe tratar con la curva de aprendizaje.

  En el caso de uso de APIs de automatización es súper importante asegurarse del manejo de tiempos de espera entre eventos. Algunas herramientas proporcionan la funcionalidad para detección automática de los tiempos de espera, sin embargo, en otras

hay que inyectar los tiempos de forma manual en el código de la prueba; esto puede llevar a pruebas "flaky" que se pueden romper fácilmente cuando los tiempos programados en el código no son los mismos en ejecución.
Algunos ejemplos de estas APIs son Espresso (Android), UIAutomator (Android), Appium (Android), XCTest (iOS), Cypress io (web), Nightwatch (web), protractor (web), CasperJS (web).

En este trabajo se van a estudiar más en profundidad las herramientas de Selenium y Rest Assured puesto que van a conformar uno de los criterios de búsqueda del heurístico.

3.3.3.1. Selenium

Selenium es un entorno de pruebas que se utiliza para comprobar si el software que se está desarrollando funciona correctamente. Esta herramienta permite: grabar, editar y depurar casos de pruebas que se pueden automatizar.

Comenzó a desarrollarse en 2004 por Jason Huggins y poco a poco se fueron uniendo varios especialistas. Un gran detalle es que este software es de código abierto (bajo licencia apache 2.0) y puede ser descargado y usado sin pagar.

Las principales características de Selenium son:

- Las acciones serán ejecutadas punto a punto, si así se considera.
- A la hora de escribir el código tiene la opción de autocompletar.
- Se puede referenciar a objetos DOM: nombre, ID o con XPath.
- Ejecutar test complejos que ahorran muchas horas de trabajo.
- Gran depuración y puntos de verificación.
- Almacenamiento en varios formatos de los tests realizados.

Su objetivo principal es comprobar que un software funcione correctamente. La mayoría de los usuarios que trabajan con Selenium son programadores QA o Engineer QA, que trabajan no solo comprobando que todo funcione correctamente, sino también evitando problemas futuros, ya que ahí es donde reside la mejor característica de Selenium para algunos Ingenieros QA.

Hoy en día, Selenium tiene un conjunto de herramientas de software y cada una muestra una perspectiva diferente. Muchos programadores deciden utilizar uno o dos a la vez para automatizar su proyecto, pero es mejor conocer todas las opciones y entender para qué sirve Selenium:

- **Selenium IDE**: permite editar, grabar y depurar lo que se muestra en el navegador.
- **Selenium Remote Control**: sistema cliente/servidor que permite utilizar el navegador web de forma local o en otro ordenador.
- **Selenium WebDriver**: sucesor de Remote Control, permite utilizar un navegador de forma local o en remoto. No requiere un servidor especial ya que se inicia una instancia en el navegador y se controla de esta manera.
- **Selenium Grid**: permite ejecutar pruebas con varios navegadores a la vez, incluso con diferentes versiones y diferentes sistemas operativos. Además, puede reducir ampliamente el tiempo que tarda un paquete de pruebas en completarse.

3.3.3.2. Rest Assured

REST Assured es un Framework escrito en Java y diseñado para simplificar las pruebas sobre servicios basados en REST.

Ofrece un DSL descriptivo (lenguajes específicos de dominio), que muestra una unión a un punto de conexión HTTP y da los resultados esperados. Este framework soporta las operaciones POST, GET, PUT, DELETE, OPTIONS, PATCH y HEAD y contiene herramientas para invocarlas y verificarlas.

Rest Assured es utilizado para las pruebas funcionales de servicios 'REST', y nos da la posibilidad de validar las respuestas HTTP recibidas del servidor.

Actualmente, sus principales ventajas son:

- Se puede comprobar el estado del código, del mensaje e, incluso, se puede ver el cuerpo de la respuesta. Además, resulta muy fácil concatenar llamadas y, en todo momento, se tiene el control del código.
- Son fáciles de integrar con pruebas de todo tipo: funcionales, unitarias, de integración, etc.
- Totalmente codificado en Java e integrable con cualquier librería/framework de pruebas por separado como jUnit, TestNG o con Maven/Gradle.
- Se puede integrar fácilmente con Jenkins.
- Es posible combinar con pruebas automatizadas de UI y no requiere de herramientas externas para ejecutarse.

3.3.4. Pruebas de sistema

Las pruebas del sistema deben enfocarse en requisitos que puedan ser tomados directamente de casos de uso, reglas y funciones de negocios. El objetivo de estas pruebas es verificar el ingreso, procesamiento y recuperación apropiado de datos, y la implementación apropiada de las reglas de negocios. Este tipo de pruebas se basan en técnicas de caja negra, es decir, verificar el sistema (y sus procesos internos), la interacción con las aplicaciones que lo usan vía GUI y analizar las salidas o resultados.

3.3.5. Pruebas de validación

Como en otras etapas de la prueba, la validación permite descubrir errores, pero su enfoque está en el nivel de requisitos, sobre cosas que son necesarias para el usuario final.

La validación se consigue cuando el software funciona de acuerdo con las expectativas razonables del cliente, definidas en la especificación de requisitos del software, la cual debería contener una sección denominada “criterios de validación”.

4. Metodología

Un repositorio software contiene una gran cantidad de información histórica y valiosa sobre el desarrollo general del sistema software que trata (estado, progreso y evolución del proyecto). La minería de repositorios es uno de los campos interesantes y de más rápido crecimiento dentro de la ingeniería del software. Esta técnica de "research" se centra en extraer y analizar los datos heterogéneos disponibles en el repositorio de software para descubrir información interesante, útil y procesable sobre el sistema.

Utilizando herramientas y técnicas de minería de datos bien establecidas, profesionales e investigadores pueden explorar el potencial de estos valiosos datos para comprender y administrar mejor sus proyectos y también para producir un sistema de software altamente confiable entregado a tiempo y dentro del presupuesto estimado.

Y por ello, la minería de repositorios, concretamente sobre la plataforma GitHub (explicado en el apéndice de este trabajo), es la técnica que se va a emplear para la obtención de los resultados deseados.

En aras de poder detectar posibles candidatos de proyectos prometedores que pudiesen contener pruebas punta a punta o end-to-end se investigan en profundidad algunos repositorios GitHub que contienen el tipo de prueba comentado anteriormente. Con esto lo que se pretende es recopilar una serie de criterios que puedan conformar un heurístico que funcione en la detección de pruebas end-to-end, es decir, criterios que cuando se aplican a esta lista inicial de proyectos detecten al menos uno.

A continuación, se validan los criterios seleccionados aplicándolos a una lista de 100 proyectos. Cada criterio puede proporcionar un resultado positivo (el proyecto satisface el criterio y puede contener pruebas end-to-end bajo dicho criterio) o negativo (ese criterio no se cumple en el proyecto).

Una vez aplicados los criterios a una lista de 100 proyectos aleatorios, estos se inspeccionan y se comparan los resultados obtenidos por el heurístico con los resultados obtenidos manualmente, marcando el resultado proporcionado por cada criterio para un proyecto en específico como verdadero positivo (el criterio fue positivo y se encontraron pruebas end-to-end), verdadero negativo (el criterio fue positivo y no se encontraron pruebas), falso positivo (el criterio fue positivo y no se encontraron pruebas) o falso negativo (el criterio fue negativo y se encontraron pruebas).

Con esta información se podan los criterios originales y se eligen los criterios considerados como más prometedores en el conjunto de 100 proyectos, es decir, aquellos que proporcionaron más verdaderos positivos, ya que el principal objetivo de este trabajo es encontrar pruebas end-to-end.

Por tanto, los pasos específicos de la metodología seguida son los siguientes:

1) Seleccionar algunos proyectos de código abierto con pruebas end-to-end.
2) Estudiar los proyectos seleccionados para:
    a) comprobar si realmente incluyen pruebas end-to-end.
    b) anotar cualquier información relevante (ubicación y nombre de los casos de pruebas, herramientas de integración continua, etc.).
3) En base a la información recopilada en el paso anterior, se preparan un conjunto de criterios que puedan identificar al menos uno de los proyectos seleccionados.
4) Seleccionar 100 proyectos de GitHub.
5) Verificar cada criterio en cada uno de los proyectos, registrando si el criterio fue positivo o negativo.
6) Verificar manualmente todos los casos positivos, es decir, los criterios que resultaron ser exitosos en cada proyecto, para detectar falsos positivos.
7) Podar el conjunto de criterios, seleccionando sólamente aquellos que son realmente efectivos.
8) Ejecutar los criterios seleccionados finalmente sobre dos consultas diferentes: 900 proyectos populares Java y 900 proyectos populares implementados en cualquier lenguaje de programación.

4.1. Selección inicial de criterios

Se seleccionan una serie de repositorios GitHub con los que se va a realizar un estudio previo a la implementación del heurístico que nos va a permitir localizar posibles pruebas end-to-end. Esta selección se va a basar en la repercusión que tiene un repositorio en concreto o bien por el elevado número de estrellas que tenga o bien por la cantidad de bifurcaciones “forks” que se hayan realizado sobre el repositorio.

De los proyectos seleccionados se van a estudiar los siguientes parámetros:

- Finalidad.

- Lenguaje.
- Localización de carpetas en las que se podrían encontrar pruebas end-to-end
- Herramientas de CI (integración continua) utilizadas.

Los proyectos seleccionados son los siguientes:

Repositorio	**Finalidad**	**Lenguaje**	**Localización e2e**	**Herramientas CI**
Redis	Implementación de una caché	C	Carpetas tests e integration	GitHub Actions
RabbitMQ	Broker de mensajería	Erlang	No encontrado	Travis (no ejecuta tests)
Envoy	Servidor proxy	C++	Carpeta integration	CircleCI Bazel (Google) Azure Pipelines Jenkins
Traefik	Servidor proxy HTTP	Go	Carpeta integration	Travis SemaphoreCI
Redmine	App web de gestión de proyectos	Ruby	Carpetas system e integration	Jenkins
Ingress-nginx	Controlador Ingress para Kubernetes	Go	Carpeta e2e	GitHub Actions
Hazelcast	Implementa una caché	Java	No encontrado	No encontrado
Vuejs	framework para la construcción de UI en la web	JavaScript	Carpeta e2e	CircleCI
Angular	Plataforma de desarrollo para la construcción de app webs mediante Typescript	JavaScript	Carpetas e2e e integration	CircleCI
Consul	Service Mesh	Go	Carpeta integration	CircleCI
Kafka	Espejo de Apache Kafka	Java	“integrationTest” y carpeta docker	Jenkins

Apache Camel	Integra varios sistemas consumiendo/produciendo datos	Java	Carpeta itest y swagger	Jenkins GitHub Actions
Casandra	Espejo de Apache Cassandra	Java	Carpeta distributed	Jenkins CircleCI
Apache Flink	Framework de procesamiento de flujo	Java	Carpeta end-to-end-test	Azure Pipelines
Vscode	Visual Studio Code	JavaScript	Carpeta integration	Azure Pipelines
ElasticSearch	Motor de búsqueda REST	Java	Carpeta test	Jenkins
Grafana	Plataforma de monitorización	JavaScript	Carpeta e2e	CircleCI
Django	Framework de construcción de apps web	Python	Carpeta tests	Tox
Express	Framework web para Node	JavaScript	Carpeta acceptance	Travis

Se comprueba que la mayoría de los proyectos están escritos en una gran cantidad de lenguajes (Java, Go, C, Ruby, etc.) y tienen carpetas test (tanto en singular "test" como en plural "tests"): "test/integration", "test/system", "test/e2e", además de otras carpetas que también han resultado ser contenedoras de pruebas end-to-end como "itest", "acceptance", "distributed", "end-to-end-test", "docker" o "swagger".

Una vez realizado el estudio de los proyectos anteriormente descrito, se procede a la identificación del heurístico con el que encontrar las pruebas que nos interesan para rellenar el DataSet, en base a los siguientes criterios de búsqueda:

1) Criterio 1: búsqueda de "integration".
2) Criterio 2: búsqueda de "system".
3) Criterio 3: búsqueda de "e2e".
4) Criterio 4: búsqueda de "itest".
5) Criterio 5: búsqueda de "acceptance".
6) Criterio 6: búsqueda de "distributed".
7) Criterio 7: búsqueda de "end-to-end-test".
8) Criterio 8: búsqueda de "docker".

9) Criterio 9: búsqueda de “swagger”.
10) Criterio 10: búsqueda de ficheros test que:
    a) Empiezan o acaban en IT.
    b) Contengan e2e, system o integrationTest.
11) Criterio 11: búsqueda de ficheros pom.xml y build.xml en cuyo contenido se utilicen “selenium” o “rest-assured”.
12) Criterio 12: búsqueda de ficheros de CI:
    a) Jenkinsfile.
    b) .travis-ci.yml.
    c) .circle-ci.
    d) .github/workflows/pipeline.yml.
    e) .azure-pipelines/pipelines.yml.
    f) .gitlab-ci.yml.

Por ello se implementa un programa en Python utilizando la API de GitHub, y de esta forma poder aplicar los diferentes criterios establecidos en el heurístico.

5. Validación experimental y resultados

5.1. Experimento preliminar

Tras la implementación del programa encargado de aplicar el heurístico se realizan pruebas para corroborar la exactitud con la que los criterios se aplican correctamente sobre los diferentes repositorios analizados. Para ello se utilizan 100 repositorios Java.
Para obtener estos 100 repositorios Java se utilizan los siguientes filtros en la query de búsqueda:

- Lenguaje: Java.
- Stars: >=500.
- Forks: >=300.
- Created: <2015-01-01.
- Pushed: >2020-01-01.
- Archived: False.
- Public: True.

De esta búsqueda se obtienen algo más de 700 repositorios, de los cuales se eligen 100 de forma aleatoria.

Se obtienen dos tablas con resultados relativos a este experimento preliminar: la **tabla A** con el número de ocurrencias, tanto obtenidas de forma automática por el heurístico como de forma manual, de cada criterio, y la **tabla B** con los porcentajes correspondientes al conteo realizado en la tabla A.

Como se puede ver en los resultados, no siempre coincide el resultado positivo detectado automáticamente por el heurístico con el detectado de forma manual, incluso en algunos casos, como por ejemplo los números 5 “acceptance”, 6 “distributed”, 9 “swagger”, 11 “dependencies” o 12 “ci-files”, llegando a poder descartar por completo el criterio.

Otros criterios como el número 4 “itest” o el número 8 “docker” se descartan del heurístico a pesar de haber obtenido resultados satisfactorios debido a que dichos resultados satisfactorios son fruto de la casualidad y no porque efectivamente se hayan encontrado pruebas end-to-end por el motivo del criterio en cuestión.

Tabla A.

Nº	Criterio	Ocurrencias	
		Automático	Manual
1	integration	29	11
2	system	46	1
3	e2e	10	3
4	itest	25	1
5	acceptance	2	0
6	distributed	6	0
7	end-to-end-test	0	0
8	docker	17	2
9	swagger	3	0
10	test name	65	2
11	dependencies	3	0
12	ci-files	3	0
	Totales	62	13

Tabla B.

Nº	Criterio	% Automáticos	% Manual	% Automáticos (manual)
1	integration	29%	11%	37,93%
2	system	46%	1%	2,17%
3	e2e	10%	3%	30%
4	itest	25%	1%	4%
5	acceptance	2%	0%	0%
6	distributed	6%	0%	0%
7	end-to-end-test	0%	0%	Sin resultados
8	docker	17%	2%	11,76%
9	swagger	3%	0%	0%
10	test name	65%	2%	3.08%
11	dependencies	3%	0%	0%
12	ci-files	3%	0%	0%

Una vez analizados los resultados de esta primera ejecución sobre 100 repositorios Java, se llega a la conclusión de que los criterios que más consistencia tienen y los que finalmente van a conformar el heurístico son los siguientes:

- Criterio 1: “integration”.
- Criterio 3: “e2e”.
- Criterio 10: “test name”.

5.2. Experimento final

Una vez establecido el heurístico con los criterios más fiables se siguen haciendo pruebas, esta vez con un volumen mayor de repositorios con respecto a la ejecución anterior: 900 repositorios Java y 900 repositorios de cualquier lenguaje, obteniendo los siguientes resultados:

5.2.1. 900 repositorios Java

Criterio	Ocurrencias
integration	298
e2e	45
test name	121
Totales	339

5.2.2. 900 repositorios (lenguajes varios)

Criterio	Ocurrencias
integration	237
e2e	69
test name	76
Totales	266

Con respecto a los resultados obtenidos en este experimento se generan dos tablas informativas que se pueden consultar en la sección de anexos (punto 9 de este trabajo). La **tabla 1** con

información del experimento diferenciando entre lenguajes de programación y la **tabla 2** diferenciando entre lenguajes de programación y criterios.

Para estos 1800 proyectos no se lleva a cabo una validación manual. Por tanto, las cifras reales pueden ser considerablemente menores.

Se puede observar la gran variedad de lenguajes que se utilizan en los repositorios que componen la plataforma GitHub, obteniendo casi 50 diferentes en este experimento masivo. A tener en cuenta los primeros lenguajes que aparecen ordenando la lista por el número de repositorios encontrados por el heurístico con el tipo de pruebas que se busca (e2e): JavaScript, Java, Go, Python, Ruby, TypeScript, C++, PHP y C.

La mayoría de estos lenguajes son bastante conocidos y se suelen utilizar en el mundo profesional para diversas implementaciones.

En el experimento realizado exclusivamente sobre proyectos Java se obtienen 339 repositorios positivos, siendo el 37’7% del total de repositorios Java analizados y, en el experimento realizado sobre repositorios de múltiples lenguajes se obtienen 266, siendo el 29’55% del total de repositorios de este segundo experimento final.

Analizando los resultados del experimento multilenguaje, llama la atención que el lenguaje con mayor número de repositorios con pruebas e2e encontradas sea JavaScript en vez de Java, el cual quedaría en el segundo puesto, ya que el programa, en principio, ha sido entrenado principalmente sobre proyectos Java. No obstante, estos resultados son entendibles puesto que JavaScript, al igual que Java, es un lenguaje de programación muy utilizado en la implementación de aplicaciones web, que son las que, en teoría, podrían tener un mayor número de pruebas e2e.

El lenguaje Go, desarrollado por Google, se trata de un lenguaje de programación concurrente y compilado, inspirado en la sintaxis de C y actualmente en alza, por eso mismo es por lo que se han encontrado tantos repositorios que utilizan este lenguaje y, además, con pruebas e2e.

En cuanto a Python y Ruby, generalmente se utilizan para implementaciones de la parte del servidor en aplicaciones con arquitecturas cliente-servidor. Además, Python permite la

implementación de aplicaciones web mediante el framework Django. Las aplicaciones web como las implementaciones del lado del servidor suelen tener pruebas e2e, por ese motivo estos dos lenguajes están dentro de los lenguajes en los que más pruebas e2e se han encontrado.

A pesar de ser uno de los lenguajes que aparecen en primer lugar en la lista, el lenguaje de programación C, con 38 repositorios de 900 (4'22% del total), solamente ha encontrado 7 repositorios con pruebas e2e. Este lenguaje es un lenguaje de programación de bajo nivel, utilizado en sistemas operativos y, por este motivo, no se suelen utilizar frecuentemente pruebas e2e.

Hay otros lenguajes en los que no se ha encontrado absolutamente nada. Un claro ejemplo es TeX, utilizado para escritura de documentos al igual que LaTeX, y es por eso por lo que no se hayan encontrado pruebas del tipo con el que se quiere construir el DataSet.

También se puede ver que el criterio 1 “integration” tanto en el primer experimento final (900 repositorios Java) como en el segundo experimento final (900 repositorios de varios lenguajes) es el criterio del que más resultados se han obtenido, ganando incluso al criterio 3 “e2e” que, en a priori, se pensaba que iba a ser más prometedor ya que se trata del acrónimo del tipo de pruebas que se buscan.

Concluyendo con el análisis de los datos, se verifica que la implementación de pruebas end-to-end no suele ser muy habitual entre los programadores, ya que, de 1800 repositorios utilizados para este experimento final, solamente 605 han resultado positivos para el heurístico, es decir, solamente ha resultado ser positivo el 33’61% del total.

6. Descripción informática

6.1. Uso de la aplicación

Para poder aplicar el heurístico se desarrolla una aplicación con interfaz de usuario para que cualquiera que quisiese utilizarla pueda hacerlo de forma trivial y sencilla: https://github.com/jorcontrerasp/BuscadorE2eGitHubRepos.

La interfaz de usuario de la aplicación cuenta con los siguientes elementos:

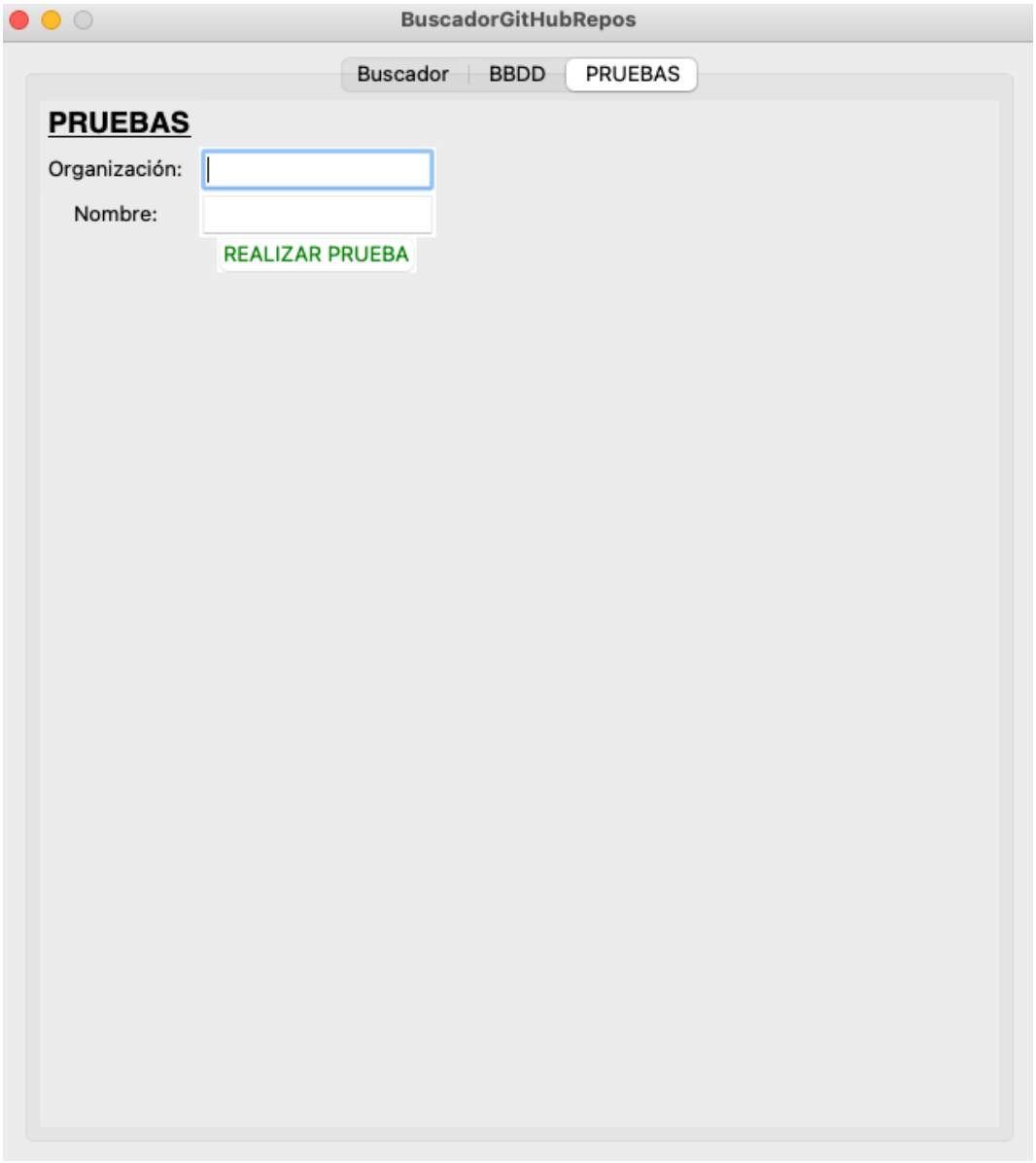
- **Primera pestaña**: pestaña de la interfaz de usuario que va a ejecutar el proceso de búsqueda en base a los parámetros y datos de configuración modificables en pantalla.
    - **Credenciales**:
        - Usuario: identificador del usuario de GitHub.
        - Token: token generado por el usuario. Se utilizará para crear el objeto generador que lanzará consultas hacia GitHub para obtener los repositorios deseados.
    - **Filtros query**:
        - Lenguaje: restricción del lenguaje de los repositorios que se van a buscar.
        - Stars: restricción de estrellas de los repositorios que se van a buscar.
        - Forks: restricción de forks de los repositorios que se van a buscar.
        - Created: restricción de fecha de creación de los repositorios que se van a buscar.
        - Pushed: restricción de la fecha de lanzamiento de los repositorios que se van a buscar.
        - Archived: restricción de si están o no archivados los repositorios que se van a buscar.
        - Public: restricción de si son o no públicos los repositorios que se van a buscar. En este caso siempre estará marcada esta opción ya que se va a trabajar con repositorios "open source".
    - **Variables de configuración**:
        - Actualizar BD: si se marca esta opción se actualizarán los datos relacionados con la búsqueda y con cada repositorio en base de datos.
        - Buscar repos en LOCAL: si se marca esta opción se clonan los proyectos que se van a utilizar en una carpeta "repositories", y una vez clonados,

la búsqueda se realiza sobre dichos ficheros en local. Al finalizar el proceso se borra la carpeta "repositories" y se genera un fichero zip a modo de snapshot con todos los repositorios clonados inicialmente.

- Generar lista repos ('.pickle'): si se marca esta opción se genera un nuevo fichero binario de Python con extensión ".pickle" con todos los repositorios que se van a utilizar en el proceso. En el caso de no marcar la opción se utiliza un fichero "repos.pickle" existente en la carpeta del proyecto.
- Randomizar repositorios: si se marca esta opción, de todos los repositorios obtenidos inicialmente para realizar la búsqueda, se utiliza un número N de forma aleatoria.
- Clonar repositorios resultantes: si se marca esta opción se clonan en local aquellos repositorios que hayan cumplido algún criterio del heurístico.
- Generar Excel: si se marca esta opción se guarda el resultado de la búsqueda en un fichero en formato Excel.
- Generar Csv: si se marca esta opción se guarda el resultado de la búsqueda en un fichero en formato Csv.
- Escribir en LOG: marcando esta opción se escriben ficheros de log relacionados con el análisis de cada repositorio al que se le aplica el heurístico.
- Botón ejecutar: botón encargado de llamar a la función exe().

- **Segunda pestaña**: esta pestaña nos va a permitir realizar consultas a la base de datos de forma visual. Para ello consta de una serie de campos (atributos de los repositorios almacenados en base de datos), un botón “CONSULTAR BD”, encargado de llamar a la función consultarBD(), una caja de texto donde se visualizan los repositorios encontrados según las restricciones seleccionadas y un botón “Limpiar”, encargado de llamar a la función limpiarResultados() que nos permitirá limpiar el contenido de la caja de texto con los repositorios resultados de la consulta.

- **Tercera pestaña**: pestaña que permite la ejecución de pruebas sobre un repositorio concreto (organización y nombre del repositorio). Para ello el botón "REALIZAR PRUEBA" llamará a la función ejecutaPrueba().



6.2. Estrategias seguidas

Para la implementación del programa se tienen en cuenta dos estrategias distintas:

1) Uso de la API de GitHub para búsqueda y análisis.
2) Uso de la API de GitHub para búsqueda, almacenamiento de los repositorios y análisis de estos en local.

En primera instancia, la estrategia que se sigue es la de utilizar la API de GitHub tanto para obtener los repositorios de la plataforma como para obtener la información relativa a dichos repositorios en su recorrido. Sin embargo, siguiendo esta estrategia aparece el principal problema encontrado en la realización de este trabajo, descrito a continuación.

Resulta que la API de GitHub nos permite crear llamadas para obtener los datos necesarios sobre los distintos repositorios que contiene la plataforma, y de esta manera poder integrar nuestra aplicación con GitHub, pero con una limitación de 60 peticiones a la hora, número insuficiente para realizar una búsqueda masiva que devuelva una cantidad aceptable de repositorios para tener un DataSet medianamente completo.

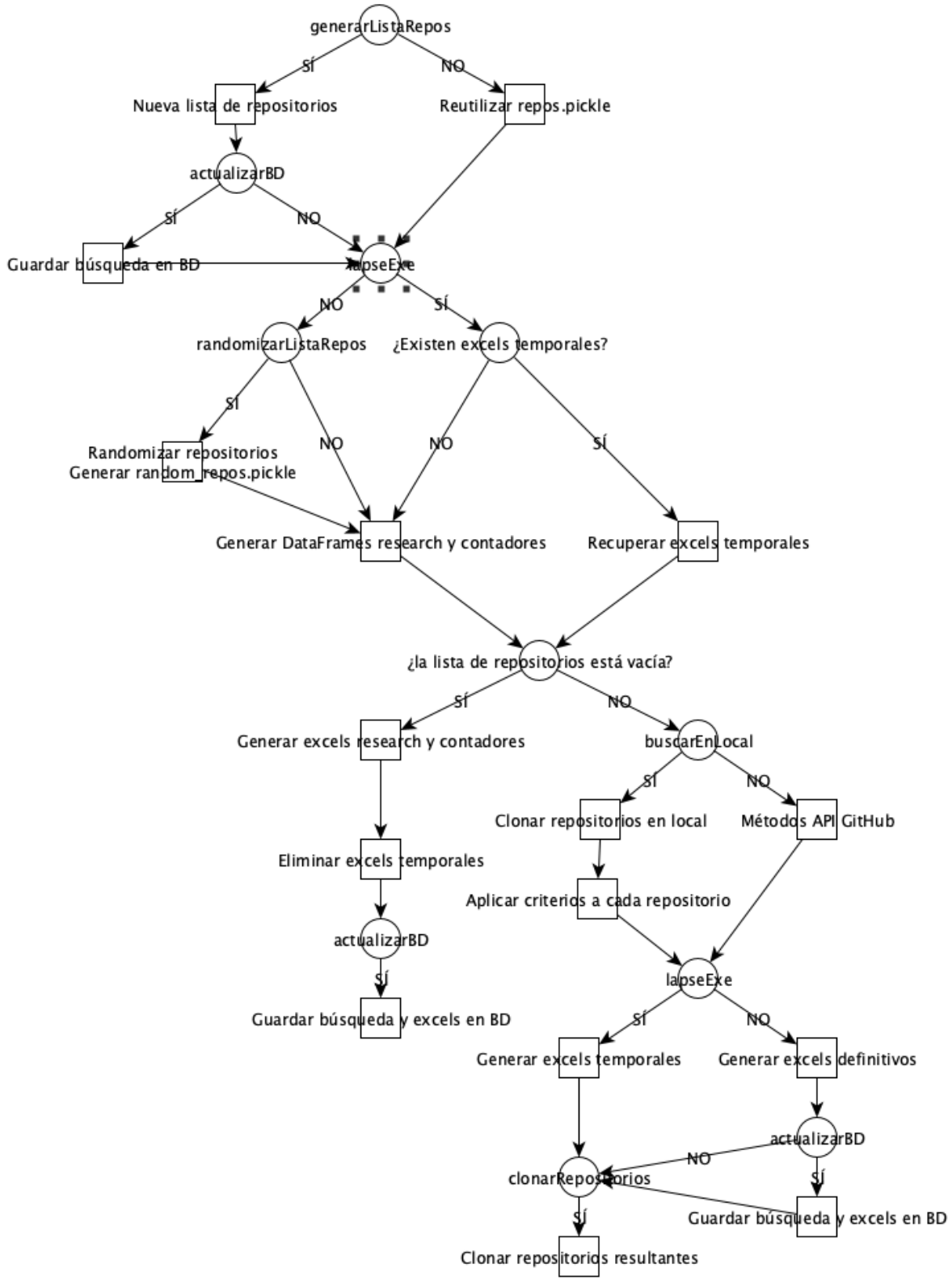
No obstante, la API de GitHub nos permite aumentar esta cantidad de 60 peticiones por hora a 5000 peticiones por hora mediante la generación de un "token" de autenticación, accediendo a los ajustes de desarrollador (ver generación de token de autenticación en el apartado de anexos).

Aún generando este "token" de autenticación, el número de peticiones proporcionadas es insuficiente si queremos hacer todo el proceso de búsqueda y análisis con los métodos proporcionados por la API de GitHub, teniendo que estar tratando en cada iteración con un número muy pequeño de repositorios para poder ejecutar el proceso de selección de pruebas e2e de principio a fin.

Viendo la inviabilidad de realizar todo el proceso mediante la API de GitHub, tanto obtención de repositorios como ejecución del proceso de análisis de dichos repositorios, se decide cambiar de estrategia optando por la opción de obtener los repositorios utilizando los métodos proporcionados por la API, clonar dichos repositorios en local y una vez clonados ejecutar el heurístico desarrollado sobre dichos repositorios.

De este modo se consigue aprovechar el número de peticiones que nos proporciona la API ya que irían todas destinadas a la obtención de repositorios mediante la query inicial. Sin embargo, el principal problema ahora reside en la capacidad que pueda tener un disco duro en el que se esté ejecutando el proceso, problema que se resolverá con la ejecución de un script de copia de ficheros entre discos duros explicado en el punto 6.8 Proyecto "ScriptLapseExe" de este trabajo.

6.3. Flujo de ejecución



6.4. Implementación

El programa implementado en Python se compone de los siguientes ficheros:

- **Buscador-ui-py**: fichero encargado de montar una interfaz de usuario sencilla mediante la librería "tkinter" de Python.
- **Main.py**: fichero principal en el que se van a llevar a cabo todas las operaciones del programa.
- **Auxiliares.py**: fichero que contiene métodos auxiliares necesarios para poder realizar operaciones relativas a la búsqueda.
- **Criterios.py**: fichero que contiene una clase enumerada con los diferentes criterios que conforman el heurístico establecido para obtener los repositorios con pruebas de integración y end-to-end de GitHub:

  En este fichero se pueden diferenciar los siguientes grupos de funciones:
  - Funciones relativas a la búsqueda realizada en local.
  - Funciones relativas a la búsqueda realizada mediante la API de GitHub.
- **Configuración.py**: fichero que contiene los parámetros de configuración almacenados en la tabla BD_D_CONFIGURACION y BD_D_CONFIGURACIONTIPO.
- **BusquedaBD.py**: fichero que contiene la clase "BusquedaBD" que representa los registros almacenados en la tabla BD_D_BUSQUEDA de la base de datos. Cuenta con los métodos básicos de obtención de sentencias select, insert, update y delete, así como los getter y setter de los atributos de la clase coincidiendo con las columnas de la tabla antes mencionada.
- **RepoBD.py**: fichero que contiene la clase "RepoBD" que representa los registros almacenados en la tabla BD_D_REPO de la base de datos. Cuenta con los métodos básicos de obtención de sentencias select, insert, update y delete, así como los getter y setter de los atributos de la clase coincidiendo con las columnas de la tabla antes mencionada.
- **ConexionesBD.py**: fichero que contiene la configuración necesaria para conectar la aplicación con la base de datos.
- **FiltrosQuery.py**: fichero que contiene los filtros que se van a utilizar para llevar a cabo la búsqueda inicial de repositorios GitHub. Estos parámetros se cargarán directamente desde la base de datos.
- **ExecuteQuery.py**: fichero que contiene todo lo relacionado con la obtención de la conexión a la base de datos, así como la ejecución de sentencias sobre la misma.
- **Pruebas.py**: fichero encargado de aplicar el heurístico a un repositorio de manera particular. Solo consta de la función ejecutaPrueba() que, conociendo el nombre de la

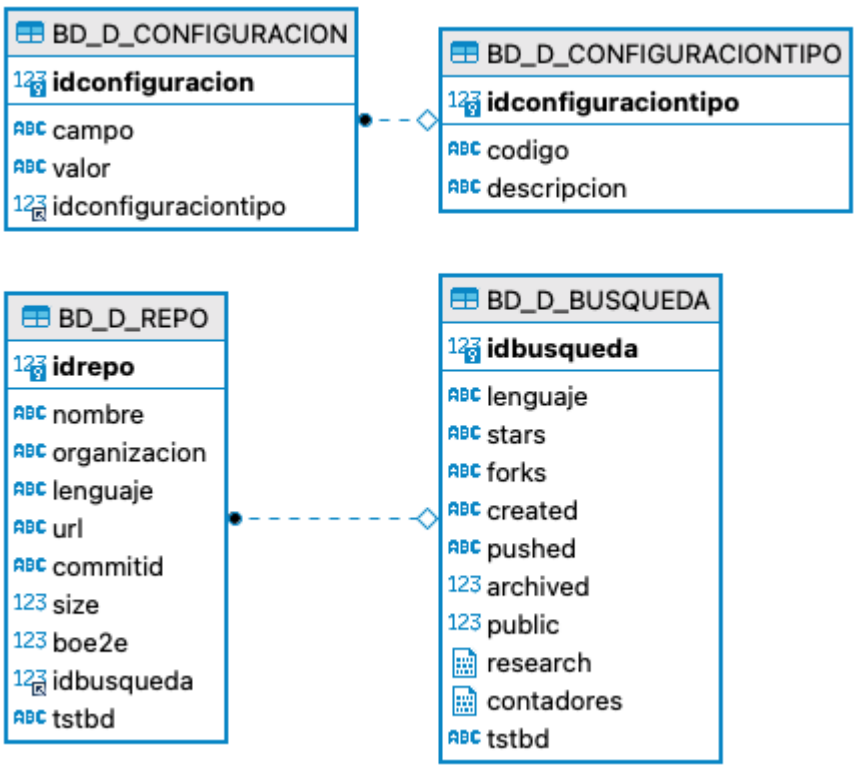
organización y del repositorio en concreto, se encarga de probar el heurístico sobre dicho repositorio.

6.5. Persistencia en base de datos

Para maquetar el resultado y tener la posibilidad de almacenar la información que devuelve el programa desarrollado se ha diseñado un modelo de base de datos formado por las siguientes tablas:

- **BD_D_CONFIGURACIONTIPO**: tabla en la que se va a almacenar información relativa a los conjuntos de configuraciones que puedan existir en el buscador.
- **BD_D_CONFIGURACION**: tabla en la que se va a almacenar toda la configuración propiamente dicha en base a los tipos de configuración que existen.
- **BD_D_REPO**: tabla en la que se van a ir almacenando todos los repositorios que se van a ir analizando en cada proceso de ejecución.
- **BD_D_BUSQUEDA**: tabla en la que se va a almacenar información relativa a cada ejecución del proceso.

El diagrama entidad-relación resultante es el siguiente:



6.6. Ficheros resultantes

En cada ejecución del proceso se van a generar una serie de directorios y ficheros en local, a destacar (siendo xxxxx la fecha en la que se lanza la ejecución):

- **contadores/contadores_xxxxx.xlsx**: fichero Excel con una serie de contadores de cuántos repositorios han cumplido qué criterio a modo de resumen.
  El formato que va a tener es el siguiente:

Criterio	N Encontrados

- **repos_snapshots/repositories_xxxxx.zip**: fichero en el que se van a almacenar todos los repositorios que se han ido clonando a lo largo de la ejecución. De esta forma se dispone de una copia del repositorio exacto con el que se ejecutó el proceso, ya que, al ser un repositorio público en funcionamiento, de un día a otro podría variar su estado y no ser igual al día en el que se le aplicó el heurístico.
- **repositories**: carpeta en la que irán los repositorios en los que el heurístico ha acertado, siempre y cuando se haya marcado la opción de clonar repositorios.
- **research/research_xxxxx.xlsx**: fichero Excel en el que se compone una matriz de repositorios/criterios para saber en cada repositorio qué criterio ha acertado y cuales son las rutas en las que ha dado lugar a que ese criterio concreto aplique.
  El formato que va a tener es el siguiente:

Nombre/Organización	GitHub URL	Lenguaje	Commit ID	Criterio 1	...	Criterio N

- **query-inicial_xxxxx.txt**: fichero de texto que contiene un resumen de los filtros iniciales que se han utilizado para realizar la búsqueda de repositorios.
- **random_repos_xxxxx.pickle**: si se marca la opción de randomizar repositorios se genera este fichero binario de Python que contiene los repositorios randomizados que se han utilizado.
- **repos_xxxxx.pickle**: fichero binario de Python con todos los repositorios resultado de la búsqueda inicial.

6.7. Dificultades y errores encontrados

En la realización de este trabajo se han ido encontrando diferentes errores que suponían un impedimento a la hora de continuar. Los más comunes han sido los siguientes:

- La API de GitHub cuenta con un número de peticiones limitado por hora, y en el momento en el que se utilizan más de los que esta API puede tratar se lanza la siguiente excepción:

  github.GithubException.RateLimitExceededException: 403 {"message": "API rate limit exceeded for user ID 18302763.", "documentation_url": "https://docs.github.com/rest/overview/resources-in-the-rest-api#rate-limiting"}

  Generando un token de autenticación podemos incrementar el número de peticiones por hora. Sin embargo, la solución considerada como óptima es la de utilizar la API de GitHub exclusivamente para la obtención de repositorios, realizando el análisis de estos repositorios clonados en local.

- Este otro error encontrado está relacionado con la existencia de algún "commit" de un repositorio una vez ya ha sido encontrado mediante la correspondiente llamada utilizando la API. La excepción que se obtiene en este caso es la siguiente:

  github.GithubException.GithubException: 404 {"message": "No commit found for the ref androidx-main", "documentation_url": "https://docs.github.com/v3/repos/contents/"}

  En este error se puede apreciar que, en un repositorio en concreto, a pesar de haberlo encontrado en la búsqueda inicial, a la hora de tratarlo no encuentra la existencia de algún "commit" en el mismo.

  Este error se podría considerar como puntual ya que una vez se obtenía la traza de la excepción, se probaba a ejecutar el proceso sobre ese mismo repositorio por separado obteniendo los resultados oportunos.

Por lo que podría ser un error de la propia API de GitHub provocado por la caída de algún servidor o por la pérdida de conexión con la API por algún motivo ajeno al proceso de búsqueda implementado.

- Otra restricción encontrada en la API de GitHub tiene que ver con la consulta del contenido de algún fichero concreto en ciertos repositorios. En el caso de acceder al contenido de un fichero demasiado grande se lanza la excepción descrita a continuación:

  github.GithubException.GithubException: 403 {"message": "This API returns blobs up to 1 MB in size. The requested blob is too large to fetch via the API, but you can use the Git Data API to request blobs up to 100 MB in size.", "errors": [{"resource": "Blob", "field": "data", "code": "too_large"}], "documentation_url": "https://docs.github.com/rest/reference/repos#get-repository-content"}

  Este caso se puede dar al aplicar el criterio número 11 del heurístico (búsqueda de los literales "selenium" o "rest-assured" en ficheros pom.xml y build.xml), por ejemplo, en el repositorio AnySoftKeyboard/AnySoftKeyboard en la ruta especificada a continuación se encuentra un fichero de 6.8MB: addons/languages/persian/pack/dictionary/prebuilt/PersianPrebuild.xml.

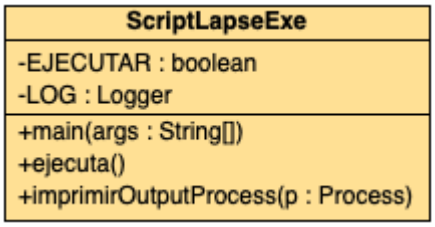
  Con el objetivo de corregir este error, en primer lugar, lanzando el proceso con la API de GitHub en su totalidad, tanto en obtención de repositorios como su posterior tratamiento y aplicación del heurístico, a la hora de obtener el contenido de algún fichero se usa el método "get_contents" y en el caso de fallar se obtiene el objeto BLOB del fichero cuyo contenido hay que consultar transformándolo más delante de base 64 a UTF8.

6.8. Proyecto "ScriptLapseExe"

ScriptLapseExe es un proyecto auxiliar implementado con el objetivo de poder ejecutar el heurístico diseñado de forma desatendida. En resumen, este proyecto auxiliar se va a encargar de ejecutar un fichero "tarea.sh" cada cierto tiempo y se utilizará para ejecutar el fichero "main.py" de la implementación del heurístico.
Se puede consultar en la siguiente url: https://github.com/jorcontrerasp/ScriptLapseExe.

El proyecto ScriptLapseExe está implementado en el lenguaje de programación Java y se compone de una única clase con los siguientes métodos:

- main(String[] args): método principal que se encarga de llamar a todos los métodos necesarios para poder ejecutar un script determinado por lapsos de tiempo.
- ejecuta(): método que se encarga de ejecutar el script propiamente dicho.
- imprimirOutputProcess(Process p): método que se encarga de imprimir por pantalla a modo de log lo que se va obteniendo como resultado de ejecutar el script.



Concretamente, el fichero "tarea.sh" que se utiliza para ser ejecutado por medio del proyecto ScriptLapseExe se compone de las siguientes líneas:

```
#!/bin/sh

cd BuscadorGitHubRepos
python3 main.py
cd ..
./hazCopiaSeg.sh
```

Y se va a apoyar, con el objetivo de no saturar el espacio local del disco duro en el que se esté ejecutando el proceso, del fichero "hazCopiaSeg.sh", ya que en cada ejecución se generan una serie de ficheros en local (ficheros de log y un fichero comprimido con los repositorios clonados a modo de snapshot) y podría darse el caso de que tras varias ejecuciones en cadena el disco duro local pueda quedarse sin espacio. Este script "hazCopiaSeg.sh" se va a encargar de borrar los ficheros generados en cada ejecución del disco en el que se esté ejecutando el proceso y copiarlos a otro disco duro distinto, mediante las instrucciones "cp" y "rm".

```
#!/bin/sh

cp -r BuscadorGitHubRepos/repos_snapshots/* /Volumes/hdd_samsung/research-tfg/repos_snapshots
cp -r BuscadorGitHubRepos/logs/* /Volumes/hdd_samsung/research-tfg/logs
rm -r BuscadorGitHubRepos/repos_snapshots/*
rm -r BuscadorGitHubRepos/logs/*
```

7. Conclusiones y trabajos futuros

Concluyendo con el proyecto, considero que he expandido mis conocimientos acerca de técnicas de research e investigación, como por ejemplo la minería de repositorios. Además, he afianzado mis conocimientos en el lenguaje de programación Python, dándome cuenta de que es un lenguaje muy interesante y con bastantes salidas en lo que respecta al mundo profesional.

En este trabajo se ha investigado la presencia de pruebas e2e en proyectos de código abierto en GitHub, con el objetivo de construir un conjunto de datos de pruebas e2e a partir de pruebas reales en proyectos activos.

Específicamente, se visualizaron 12 criterios que podrían revelar pruebas e2e en proyectos. En un experimento preliminar con 100 proyectos, se seleccionaron los criterios más prometedores con el objetivo de podar la lista de repositorios. A continuación, se realizaron dos análisis: uno en un conjunto de 900 proyectos Java, y otro sobre un conjunto de 900 proyectos de diferentes lenguajes.

Según los criterios seleccionados se han obtenido unos números suficientemente grandes para garantizar que muchas decenas de esos proyectos contendrán realmente pruebas end-to-end.

En futuras investigaciones se validarán los proyectos obtenidos y se incluirán aquellos con pruebas end-to-end en un conjunto de datos, el cual se pondrá a disposición de los investigadores en pruebas de software.

8. Bibliografía

[1] Sébastien Chazallet, «Python 3. Los fundamentos del lenguaje»

[2] Unmesh Gundecha, «Selenium Testing Tools Cookbook»

[3] Satya Avasarala, «Selenium WebDriver Practical Guide»

[4] Thom Garrett, Bernie Gauf, Elfriede Dustin, «Implementing Automated Software Testing»

[5] Rex Black, «Pragmatic Software Testing»

[6] Glenford Myers, «El arte de probar el software»

[7] GitHub, Api de GitHub disponible en línea en la siguiente url: https://pygithub.readthedocs.io/en/latest/examples/MainClass.html#search-repositories-by-language

9. Anexos

9.1. Tabla 1

Tabla con resultados del experimento final (lenguajes varios) desglosado por lenguajes:

Lenguaje	Total repos	% del total	repositorios con e2e (automático)	% con e2e (automático)
JavaScript	258	28,67	73	28,29
Java	79	8,78	34	43,04
Go	58	6,44	24	41,38
Python	71	7,89	23	32,39
Ruby	31	3,44	21	67,74
TypeScript	34	3,78	19	55,88
C++	51	5,67	17	33,33
PHP	35	3,89	14	40,00
C	38	4,22	7	18,42
C#	12	1,33	5	41,67
Sin lenguaje	71	7,89	4	5,63
Shell	24	2,67	4	16,67
Objective-C	23	2,56	3	13,04
Swift	15	1,67	3	20,00
Scala	5	0,56	3	60,00
HTML	21	2,33	2	9,52
Elixir	3	0,33	2	66,67
CoffeeScript	5	0,56	1	20,00
Rust	2	0,22	1	50,00
Assembly	1	0,11	1	100,00
Groovy	1	0,11	1	100,00
Lua	1	0,11	1	100,00
Mustache	1	0,11	1	100,00
Nim	1	0,11	1	100,00
OCaml	1	0,11	1	100,00
CSS	15	1,67	0	0,00
Vim script	10	1,11	0	0,00
Jupyter N.	5	0,56	0	0,00
Kotlin	4	0,44	0	0,00
SCSS	3	0,33	0	0,00
Clojure	2	0,22	0	0,00
Emacs Lisp	2	0,22	0	0,00
Haskell	2	0,22	0	0,00
Objective-C++	2	0,22	0	0,00

Batchfile	1	0,11	0	0,00
Crystal	1	0,11	0	0,00
Dockerfile	1	0,11	0	0,00
Erlang	1	0,11	0	0,00
Julia	1	0,11	0	0,00
Less	1	0,11	0	0,00
Markdown	1	0,11	0	0,00
Nunjucks	1	0,11	0	0,00
Perl	1	0,11	0	0,00
PowerShell	1	0,11	0	0,00
Roff	1	0,11	0	0,00
TeX	1	0,11	0	0,00
VimL	1	0,11	0	0,00
Totales	900	-	266	-

9.2. Tabla 2

Tabla con resultados del experimento final (lenguajes varios) desglosado por lenguajes y criterios:

Lenguaje	Total repositorios	Total C1	% C1	Total C3	% C3	Total C10	% C10
JavaScript	258	58	22,48	27	10,47	20	7,75
Java	79	33	41,77	10	12,66	10	12,66
Python	71	22	30,99	1	1,41	2	2,82
Sin lenguaje	71	4	5,63	1	1,41	2	2,82
Go	58	19	32,76	12	20,69	17	29,31
C++	51	17	33,33	3	5,88	8	15,69
C	38	5	13,16	2	5,26	2	5,26
PHP	35	14	40,00	1	2,86	0	0,00
TypeScript	34	17	50,00	9	26,47	9	26,47
Ruby	31	20	64,52	3	9,68	6	19,35
Shell	24	4	16,67	0	0,00	0	0,00
Objective-C	23	3	13,04	0	0,00	0	0,00
HTML	21	2	9,52	0	0,00	0	0,00
CSS	15	0	0,00	0	0,00	0	0,00
Swift	15	3	20,00	1	6,67	1	6,67
C#	12	5	41,67	2	16,67	1	8,33

Vim script	10	0	0,00	0	0,00	0	0,00
CoffeeScript	5	1	20,00	0	0,00	0	0,00
Jupyter N.	5	0	0,00	0	0,00	0	0,00
Scala	5	3	60,00	1	20,00	1	20,00
Kotlin	4	0	0,00	0	0,00	0	0,00
Elixir	3	2	66,67	0	0,00	1	33,33
SCSS	3	0	0,00	0	0,00	0	0,00
Clojure	2	0	0,00	0	0,00	0	0,00
Emacs Lisp	2	0	0,00	0	0,00	0	0,00
Haskell	2	0	0,00	0	0,00	0	0,00
Objective-C++	2	0	0,00	0	0,00	0	0,00
Rust	2	1	50,00	0	0,00	0	0,00
Assembly	1	1	100,00	0	0,00	0	0,00
Batchfile	1	0	0,00	0	0,00	0	0,00
Crystal	1	0	0,00	0	0,00	0	0,00
Dockerfile	1	0	0,00	0	0,00	0	0,00
Erlang	1	0	0,00	0	0,00	0	0,00
Groovy	1	1	100,00	0	0,00	0	0,00
Julia	1	0	0,00	0	0,00	0	0,00
Less	1	0	0,00	0	0,00	0	0,00
Lua	1	1	100,00	0	0,00	0	0,00
Markdown	1	0	0,00	0	0,00	0	0,00
Mustache	1	1	100,00	0	0,00	1	100,00
Nim	1	1	100,00	0	0,00	0	0,00
Nunjucks	1	0	0,00	0	0,00	0	0,00
OCaml	1	1	100,00	0	0,00	0	0,00
Perl	1	0	0,00	0	0,00	0	0,00
PowerShell	1	0	0,00	0	0,00	0	0,00
Roff	1	0	0,00	0	0,00	0	0,00
TeX	1	0	0,00	0	0,00	0	0,00
VimL	1	0	0,00	0	0,00	0	0,00
Totales	900	239	-	73	-	81	-

9.3. Minería de repositorios con GitHub

Para poder llevar a cabo sobre la plataforma GitHub la técnica de research conocida como minería de repositorios, MSR por sus siglas en inglés (Mining Software Repositories), se utiliza la biblioteca "PyGithub", la cual nos va a permitir manejar diferentes recursos de GitHub como repositorios, perfiles de usuario, organizaciones, etc. desde cualquier script Python.

Para instalar la biblioteca bastaría con ejecutar el comando *pip install pygithub* o clonarlo directamente desde el propio GitHub.

Una vez instalado ya se podría importar desde cualquier script Python mediante la siguiente instrucción:

```
from github import Github
```

Con la biblioteca ya importada en el script podemos generar un objeto "GitHub" y, partiendo de ese objeto, realizar consultas sencillas utilizando una serie de parámetros definidos por la librería.

```
usuario = "<usuario>"
token = "<token>"
g = Github(usuario, token)
```

Al ejecutar una consulta, no se realiza ninguna consulta o búsqueda como tal, sino que se obtiene un objeto generador y se comienza a ejecutar la consulta al iterar sobre dicho objeto generador.

```
query="<query>"
generator=g.search_repositories(query=query)
```

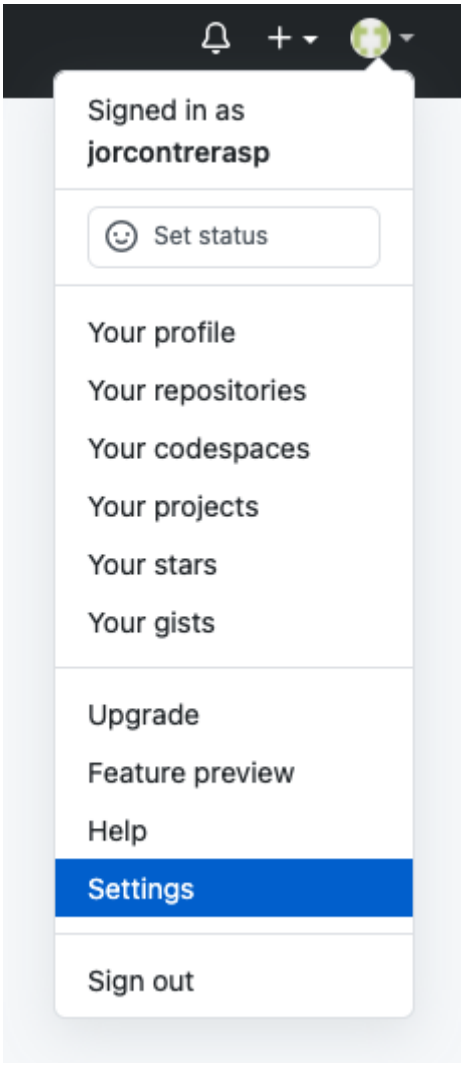
Listando el objeto generador "generator" devuelto por la función "search_repositories" de la API de GitHub, se obtienen todos los repositorios que satisfacen la query pasada como parámetro. Esta lista de repositorios va a ser una lista de objetos "Repository" de los que se van a poder obtener una infinidad de información relativa a cada uno de ellos.

9.4. Generación de token de autenticación

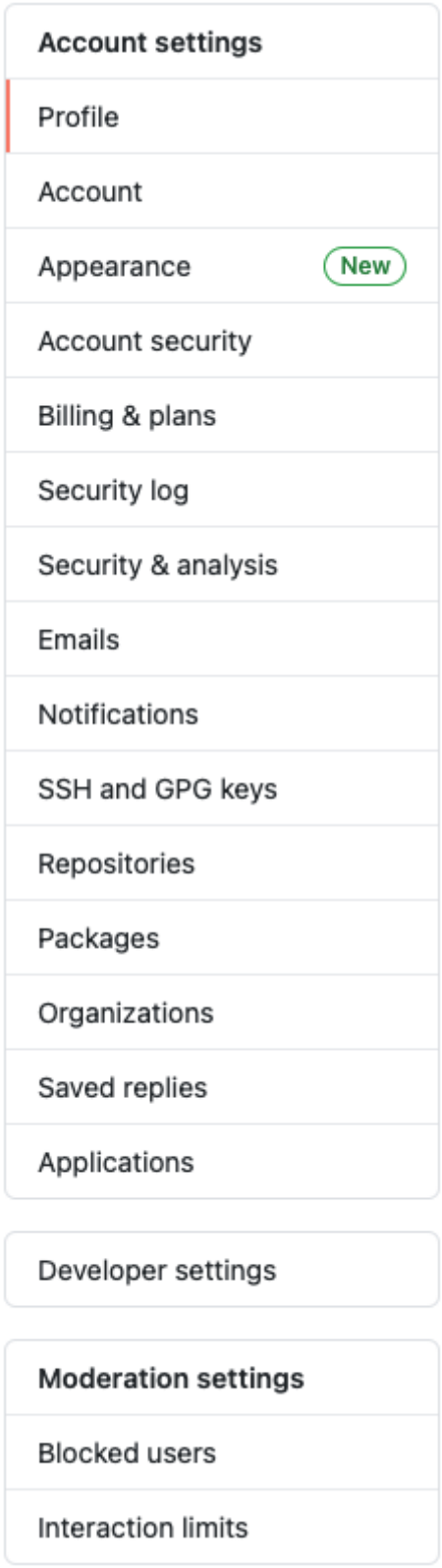
Generar un token de autenticación GitHub nos permite aumentar el número de peticiones por hora que disponemos, pasando de tener 60 a 1500.

Este token de autenticación lo podemos generar accediendo a los ajustes de desarrollador de GitHub siguiendo los siguientes pasos:

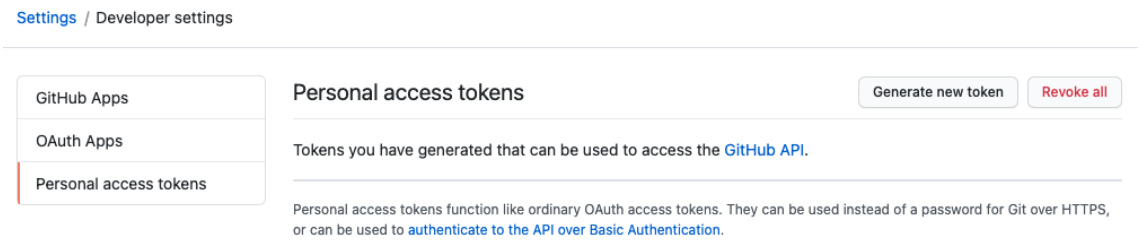
1) Desplegar ajustes de usuario pinchando en el icono redondo situado en la parte superior derecha de la pantalla y acceder a la opción “Settings”.



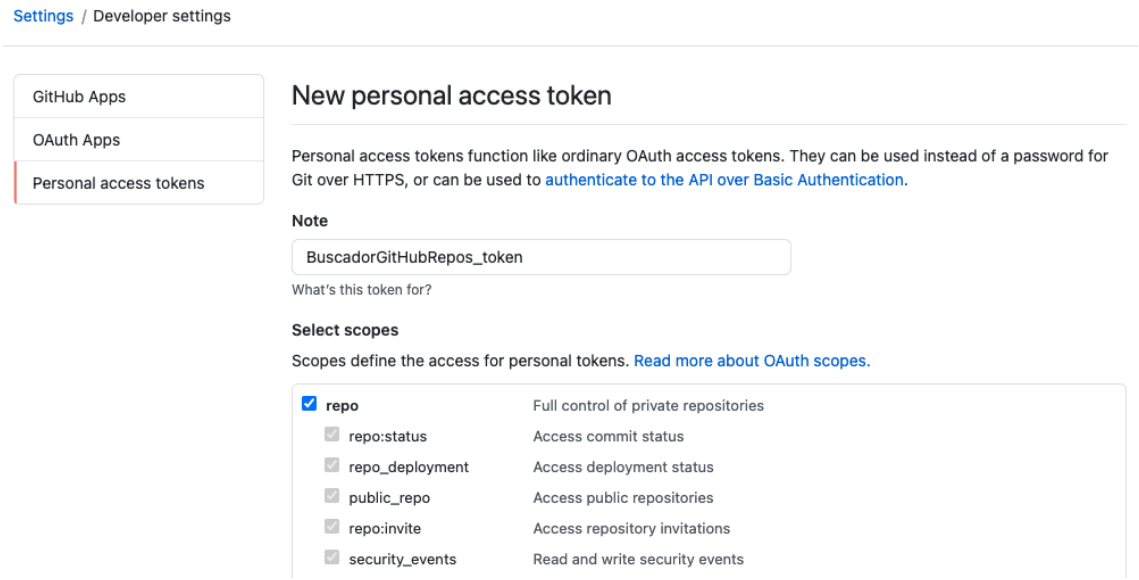
2) Una vez dentro de “Settings”, acceder a la opción de menú “Developer settings”.



3) Accediendo a la opción “Personal Access tokens” y pinchando el botón “Generate new token” nos permitirá rellenar los datos relacionados con el token que utilizaremos posteriormente para autenticarnos utilizando la API de GitHub.



4) Rellenar un nombre y seleccionar los permisos del token a generar. Para este caso seleccionar la opción “repo” sería más que suficiente.



5) Por último, pinchar en el botón de generar token.