ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA DEL SOFTWARE
Curso Académico 2023/2024
Trabajo Fin de Grado

PRUEBAS AUTOMÁTICAS DE APLICACIONES DE VIDEOCONFERENCIA CON SELENIUM EN GITHUB ACTIONS

Autor: Andrea Patricia Acuña Padrón

Tutor: Micael Gallego Carrillo

©2023 Autora Andrea Patricia Acuña Padrón
Algunos derechos reservados
Este documento se distribuye bajo la licencia "Atribución-CompartirIgual 4.0 Internacional" de Creative Commons, disponible en
https://creativecommons.org/licenses/by-sa/4.0/

Resumen:

El presente Trabajo de Fin de Grado se sumerge en el dinámico mundo del software, focalizándose en la automatización de pruebas para aplicaciones de videoconferencia, un tema de gran actualidad dada la dependencia creciente de la comunicación virtual en nuestra sociedad. El título "Pruebas Automáticas de Aplicaciones de Videoconferencia con Selenium en Github Actions" esboza un proyecto que vincula la teoría y práctica de la ingeniería software con las necesidades actuales de eficiencia y calidad.

El siguiente trabajo destaca cómo el avance continuo de la industria del software demanda una mayor eficiencia en los procesos de prueba, resaltando el rol fundamental de la automatización. Sin embargo, no descuida la persistente relevancia del testing manual, ni la importancia de una buena formación en programación para quienes llevan a cabo estas tareas automatizadas.

La descripción técnica detalla cómo se han definido, diseñado e implementado las pruebas, asegurando que cubran las necesidades específicas de las aplicaciones de videoconferencia. Además, se tocan temas como la adaptación a distintas plataformas y la importancia de generar reportes comprensibles.

En relación a las tecnologías y herramientas, se selecciona Selenium por ser un estándar de facto en la automatización de pruebas para aplicaciones web, y GitHub Actions por su capacidad para automatizar flujos de trabajo, resultando en un proceso de integración y entrega continuos que es esencial en el ciclo de vida del desarrollo de software moderno.

Finalmente, el actual proyecto no solo enfrenta y resuelve retos técnicos sino que también enfatiza la mejora continua a través del refinamiento de las pruebas.

Índice:

Capítulo 1: Introducción y motivación..5

1.1. La evolución de la industria del software y el imperativo de la automatización........................5

1.2. Definición y relevancia de la automatización de pruebas en software......................................5

1.3. Limitaciones de la automatización y la coexistencia con el testing manual.............................. 7

1.4 La Importancia de la Formación en Programación para la Automatización de Pruebas.............8

Capítulo 2: Objetivos.. 11

Capítulo 3: Tecnologías, Herramientas y Metodologías...............................13

3.1 Tecnologías y Herramientas.. 13

3.1.1 Evolución de Java y su Naturaleza Multiplataforma.. 13

3.1.2 Selenium..15

3.1.2.1 Principales características de Selenium..16

3.1.2.2 Justificación del uso de Java con Selenium...17

3.1.2.3 Usos de Selenium.. 18

3.1.2.4 Desventajas de Selenium.. 19

3.1.2.5 Identificación de los elementos y el DOM...20

3.1.3 Github Actions.. 24

3.1.3.1 Componentes de Github Actions..25

3.1.3.2 Comparativa con otras herramientas CI/CD..26

3.1.4 Extent Report.. 27

3.1.5 Librerías.. 29

3.2 Metodología empleada... 31

3.2.1. Metodología Iterativa..31

3.2.2 Blog...33

3.2.3 Repositorio Github..33

Capítulo 4: Descripción informática...35

4.1 Definición de requisitos de pruebas..35

4.2 Diseño..38

4.3 Implementación.. 43

4.3.1 Configuración de drivers..44

4.3.2 Localización de elementos...48

4.3.3 Sincronizaciones/Esperas..52

4.3.4 Captura de video.. 54

4.3.5 Generación de un reporte... 56

4.3.6 Parametrización de datos.. 59

4.3.7 Ejecución en Github Actions.. 61

4.4 Refinamiento.. 64

4.4.1 Creación del reporte.. 64

4.4.2 Conflicto de versiones entre el driver y chrome.. 67

4.4.3 Captura de video. Falso positivo...68

4.4.4 Mezcla de espera implícita y explícita..68

Capítulo 5: Conclusiones y trabajo futuro.. 72
5.1 Conclusiones...72
5.1.1 Cumplimiento de objetivos.. 72
5.1.2 Conclusiones personales.. 72
5.2 Trabajo futuro... 73
Capítulo 6: Bibliografía...75

Capítulo 1: Introducción y motivación

1.1. La evolución de la industria del software y el imperativo de la automatización

La industria del software ha experimentado un auge inigualable en las últimas décadas. En un mundo cada vez más interconectado y digitalizado, el software se ha convertido en el núcleo de casi todas las operaciones. Desde la realización de transacciones bancarias y la gestión de nuestra información personal hasta la facilitación de nuestras comunicaciones diarias, el software desempeña un papel preponderante.

El ciudadano promedio confía ciegamente en estos sistemas informáticos que están omnipresentes en su vida cotidiana. Sin embargo, esta dependencia masiva del software también exige que los mismos sean fiables, seguros y eficientes. La confianza en estos sistemas no es simplemente una cuestión de comodidad. En muchos casos, es una cuestión de seguridad y bienestar.

Dado el ritmo vertiginoso con el que evoluciona la industria del software, los desarrolladores y las empresas enfrentan el desafío de producir soluciones de alta calidad en tiempos reducidos y a costos competitivos. No es suficiente con que un software funcione, también debe ser robusto, escalable y, sobre todo, seguro.

Es aquí donde la automatización de pruebas se vuelve crucial. A través de la automatización, es posible garantizar que las aplicaciones funcionen como se espera, sin fallos críticos y vulnerabilidades que puedan ser explotadas, a un costo menor en lo que a tiempo se refiere. Además, al automatizar pruebas repetitivas, se libera tiempo y recursos para centrarse en aspectos más complejos y críticos del desarrollo, como la innovación y la optimización.

La automatización no solo agiliza el proceso de desarrollo, sino que también reduce el margen de error humano, asegurando una mayor consistencia y confiabilidad en los productos finales. En un mundo donde la tecnología y el software son esenciales para el funcionamiento de la sociedad, la importancia de la automatización en la producción de software de calidad no puede ser subestimada.

1.2. Definición y relevancia de la automatización de pruebas en software

A medida que el panorama tecnológico se expande y la industria del software continúa su marcha implacable hacia adelante, el mandato es claro: entregar productos de software de calidad superior en el menor tiempo posible y con costes optimizados. Para cumplir con estas expectativas crecientes, la industria ha reconocido y abrazado el valor de la automatización de pruebas.

La automatización de pruebas de software se refiere al empleo de programas y herramientas especializadas para ejecutar pruebas en una aplicación de software, simular acciones de usuario y verificar si la aplicación funciona según lo previsto. A través de este enfoque, se pueden programar múltiples escenarios y caminos de usuario para comprobar la integridad, la funcionalidad y la seguridad de un software. Además, en este apartado entra en juego la integración continua, gracias a esta es posible que las pruebas automatizadas se ejecuten de manera automática a través de un disparador previamente configurado. De esta manera es posible que las pruebas se ejecuten de manera automática cada vez que se realice un cambio en el código, probando así que el software siga correcto tras realizar subidas de código, reduciendo así la intervención humana. Una vez ejecutadas, estas pruebas generan informes detallados, identificando posibles fallos y sus causas.

El antiguo paradigma de pruebas manuales, aunque necesario en ciertos escenarios, presentaba desafíos inherentes. La intervención humana es susceptible al error y a la inconsistencia, lo que puede llevar a omisiones o interpretaciones erróneas. Las pruebas manuales también pueden ser intensivas en tiempo y recursos, retrasando la entrega del software y aumentando los costos.

En contraste, la automatización de pruebas presenta ventajas significativas:

- **Eficiencia mejorada**: Las pruebas automatizadas son repetibles y se pueden ejecutar en múltiples configuraciones y plataformas, garantizando una cobertura más amplia y consistente.

- **Rapidez**: Como se mencionó anteriormente, una máquina puede ejecutar pruebas a una velocidad que ningún humano podría alcanzar, permitiendo la validación rápida de grandes porciones de código.

- **Reducción de errores humanos**: Al minimizar la intervención humana, se disminuye la posibilidad de omisiones o interpretaciones erróneas en el proceso de prueba.

- **Entrega acelerada**: La capacidad de realizar pruebas de manera rápida y efectiva facilita ciclos de desarrollo más cortos y permite a las organizaciones llevar productos al mercado en tiempos récord.

- **Reusabilidad del código**: Muchos casos de prueba en las pruebas automatizadas se repiten, como podría ser el login de una aplicación. Este código se debe reutilizar siempre que sea posible, haciendo así la tarea de automatizar menos compleja.

En resumen, en la era actual donde la agilidad y la calidad son esenciales, la automatización de pruebas no es solo una herramienta técnica, sino una estrategia crítica que potencia la capacidad de las empresas para competir y destacar en un mercado saturado y en constante evolución.

1.3. Limitaciones de la automatización y la coexistencia con el testing manual

Si bien la automatización de pruebas ha revolucionado el ámbito del aseguramiento de la calidad, no está exenta de limitaciones. La premisa de que todos los tests son automatizables es errónea. Hay situaciones y escenarios en los que la automatización podría no ser adecuada o incluso contraproducente. Por ello, es crucial distinguir entre los tests que pueden y deben ser automatizados y aquellos que requieren la sutileza y el discernimiento humano.

Los falsos positivos y falsos negativos representan una gran preocupación en la automatización. Un falso positivo indica que un test ha pasado cuando no debería haberlo hecho, lo que puede llevar a la liberación de un software con defectos. Por otro lado, un falso negativo podría resultar en tiempo y recursos desperdiciados en la búsqueda de un defecto inexistente. Estos riesgos reiteran la importancia de un enfoque equilibrado que combine pruebas manuales y automáticas.

Las pruebas manuales siguen siendo esenciales en ciertos escenarios, tales como:

- **Tests exploratorios**: Donde los testers buscan activamente defectos sin un plan de pruebas preestablecido. Es una técnica altamente adaptativa que se basa en el instinto y en la capacidad del tester para interactuar con la aplicación como lo haría un usuario real. Estas pruebas son particularmente útiles cuando se enfrenta a un nuevo software o a una funcionalidad que no ha sido bien documentada. A través de la exploración, los testers pueden descubrir defectos que las pruebas predefinidas podrían haber pasado por alto.

- **Pruebas de usabilidad**: Evaluar la experiencia del usuario y la facilidad de uso no puede ser realizado de manera efectiva por una máquina. Aunque la automatización puede evaluar ciertos aspectos técnicos de la usabilidad, como el tiempo de carga de una página, la verdadera evaluación de la facilidad de uso, la intuitividad y la satisfacción del usuario solo puede ser realizada por seres humanos. Las pruebas de usabilidad permiten a las empresas recibir retroalimentación directa de los usuarios, lo que puede llevar a mejoras significativas en el diseño y la funcionalidad.

- **Escenarios complejos o cambiantes**: En situaciones donde los requisitos cambian con frecuencia o son inherentemente complejos, automatizar puede ser más costoso que realizar pruebas manuales. En estas situaciones, invertir tiempo y recursos en la creación de pruebas automatizadas puede resultar contraproducente. Cada vez que cambian los requisitos, las pruebas automatizadas tendrían que ser revisadas o reescritas, lo que puede aumentar significativamente los costos y el tiempo de desarrollo.

Sin embargo, hay tests que son claramente beneficiados por la automatización:

- **Sanity testing**: En el ámbito del desarrollo, es común hacer pequeños ajustes o correcciones en el código. Sin embargo, incluso el cambio más mínimo puede tener repercusiones inesperadas. Este tipo de prueba es una forma rápida de asegurarse de que los cambios realizados no hayan causado errores graves en las partes fundamentales del software. Al ser pruebas ágiles y de corta duración, su automatización permite obtener resultados rápidos y decidir si el software puede pasar a etapas de pruebas más exhaustivas o si requiere correcciones.

- **Data Driven Testing (DDT)**: Es esencial en escenarios donde se necesita verificar el comportamiento de una función o módulo con múltiples conjuntos de datos. Al alimentar el software con diferentes conjuntos de datos de entrada, se puede asegurar que el sistema maneja correctamente una amplia variedad de situaciones. La automatización brilla en esta área porque hacer esto manualmente no solo sería extremadamente tedioso, sino que también sería propenso a errores.

- **Pruebas tediosas**: Aquellas que son repetitivas y que consumen mucho tiempo son ideales para la automatización, ya que reducen la posibilidad de error humano y aumentan la eficiencia. Automatizar estas pruebas repetitivas garantiza que cada ejecución se realice exactamente de la misma manera, asegurando precisión y consistencia. Además, la automatización puede hacerlo a una velocidad que ningún humano podría igualar, lo que significa que se pueden completar más pruebas en menos tiempo.

- **Pruebas de larga duración**: Los tests que requieren horas de ejecución son candidatos perfectos para la automatización, liberando al equipo de pruebas para centrarse en otras tareas.

En conclusión, aunque la automatización de pruebas es una herramienta poderosa, no debería ser vista como un reemplazo total del testing manual. Más bien, ambas modalidades deben coexistir, aprovechando sus respectivas fortalezas para garantizar la entrega de software de la más alta calidad.

1.4 La Importancia de la Formación en Programación para la Automatización de Pruebas

Tras la explicación introductoria sobre los conceptos más relevantes de la automatización de pruebas surge una pregunta: ¿Quién debería encargarse de la automatización de pruebas? Mientras que las herramientas modernas, como UFT (Unified Functional Testing) [17] y Cucumber [18], ofrecen facilidades para aquellos sin un trasfondo técnico, la realidad es que una formación sólida en programación puede ser fundamental para maximizar la eficacia de la automatización.

Las soluciones de "grabar y reproducir" como se puede apreciar en UFT (Unified Functional Testing), prometen una automatización sencilla, permitiendo a los usuarios crear scripts simplemente realizando acciones en una interfaz de usuario. Sin embargo, esta simplicidad es engañosa. Estos scripts generados automáticamente a menudo son frágiles, un cambio menor en la interfaz o en la funcionalidad puede hacer que fallen. Además, estos scripts pueden ser ineficientes, llevando a pruebas más lentas o a resultados imprecisos.

Por otro lado, herramientas como Cucumber, que emplean el lenguaje Gherkin, ofrecen una representación legible por humanos de los escenarios de prueba. Aunque esto puede hacer que la creación de pruebas parezca accesible para aquellos sin formación técnica, la verdad es que la implementación real de estos escenarios en scripts funcionales todavía requiere un conocimiento sólido de la programación.

Un profesional formado en programación aporta varios beneficios al proceso de automatización. Primero, tienen una comprensión más profunda de las estructuras de datos, los algoritmos y las mejores prácticas de codificación. Esta formación les permite escribir scripts más robustos, eficientes y mantenibles. Además, pueden anticipar y mitigar problemas potenciales en el código, evitando retrabajos y fallas costosas. También es más probable que entiendan el diseño y la arquitectura del software, permitiéndoles crear pruebas que aborden áreas críticas de la aplicación.

Mientras que la perspectiva de un tester no técnico es invaluable para evaluar la usabilidad y la experiencia del usuario, no es sustituto de la habilidad técnica requerida para escribir scripts de pruebas efectivos. La automatización no se trata simplemente de replicar las acciones del usuario, sino de asegurar que el código detrás de esas acciones funcione de manera confiable y consistente.

Por lo tanto, aunque las herramientas modernas han hecho que la automatización parezca más accesible, es esencial que las organizaciones reconozcan la importancia de la formación técnica en el proceso de automatización y valoren a los profesionales que poseen una combinación de habilidades técnicas y una perspectiva centrada en el usuario.

Capítulo 2: Objetivos

Tras establecer un marco teórico sobre la automatización en el primer capítulo, este segmento se centra en detallar los objetivos específicos que guían el presente proyecto en relación con la temática previamente discutida.

Es innegable la creciente relevancia de la automatización en el contexto contemporáneo. En el ámbito del software, donde la calidad no es solo deseable sino esencial, la automatización de pruebas emerge como una herramienta fundamental. Dentro de este escenario, el proyecto se concentra en la implementación de pruebas automatizadas dirigidas a la plataforma web de videoconferencias, OpenVidu [21]. Desarrollada por un equipo de la Universidad Rey Juan Carlos, la iniciativa detrás de automatizar pruebas para OpenVidu radica en potenciar la calidad, eficiencia y confiabilidad del producto software.

En esencia, el proyecto involucra la creación y gestión de scripts, además de su correspondiente ejecución mediante GitHub Actions, una reconocida herramienta para la integración y despliegue continuos. Este enfoque proactivo posibilita la identificación temprana de errores, proporcionando retroalimentación casi inmediata al equipo de desarrollo. Así, cada vez que se introduzca nuevo código, será posible verificar que no comprometa funcionalidades previamente establecidas. La automatización de este proceso garantiza agilidad y elimina la necesidad de intervención manual extensa por parte de un equipo de pruebas.

Uno de los pilares de este proyecto es explorar y analizar la calidad y eficiencia de las pruebas en un ecosistema de integración y entrega continuas. Adicionalmente, se persigue una profunda investigación sobre la concepción de casos de pruebas que, alineados con las discusiones del capítulo anterior, sean ideales para una automatización eficiente. Posteriormente, se proporcionará un detallado desglose de los casos seleccionados..

Finalmente, un objetivo crucial es examinar la habilidad de los scripts de pruebas automatizadas para identificar inconsistencias y vulnerabilidades en el código fuente de los tutoriales de la web de OpenVidu. Esto permitiría, por ejemplo, discernir si un botón presenta tiempos de respuesta fluctuantes, lo que podría derivar en fallas esporádicas. Al detectar tales incongruencias, el equipo de desarrollo puede profundizar en el análisis, potencialmente descubriendo problemas subyacentes más significativos.

En conclusión, este proyecto no solo busca implementar pruebas automatizadas para una herramienta como OpenVidu sino también trascender en el ámbito de la calidad del software. A través de la metodología y herramientas elegidas, se busca establecer un estándar elevado de precisión, eficiencia y confiabilidad, beneficiando tanto a los desarrolladores como a los usuarios finales.

Capítulo 3: Tecnologías, Herramientas y Metodologías

En este capítulo se explicará la teoría de las diferentes tecnologías y herramientas utilizadas para el correcto desarrollo del proyecto así como la metodología personal que se ha llevado a cabo.

3.1 Tecnologías y Herramientas

Java, creado por Sun Microsystems en 1995, es un lenguaje de programación orientado a objetos que se ha consolidado como uno de los lenguajes más populares y versátiles del mundo. Para el desarrollo del proyecto, se ha utilizado principalmente Java, no solo por su conocida aplicación en el desarrollo de páginas web, sino también por su amplio alcance en muchas otras áreas como el desarrollo de videojuegos, computación en la nube, macrodatos, inteligencia artificial e Internet de las cosas (IoT). La plataforma independiente y la robustez de su arquitectura hacen de Java una elección ideal para una variedad de aplicaciones. Específicamente, en este proyecto, se ha buscado llevar la automatización de pruebas a un nivel superior para asegurar la calidad y eficiencia del software en desarrollo. Al integrar Java con Selenium, un controlador de navegadores ampliamente utilizado, se ha ofrecido la capacidad de crear y ejecutar scripts de prueba, interactuando con páginas web de manera eficiente y precisa. Sin embargo, la simple ejecución de pruebas no fue suficiente para ellos. Fue esencial poder evaluar rápidamente los resultados y detectar posibles fallos. Es por eso que se ha integrado Extent Report, del que se hará un mayor hincapié más adelante. Finalmente, para garantizar que las pruebas se ejecuten automáticamente y sin intervención manual, se ha hecho uso de Github Actions, del que se hablará más adelante.

3.1.1 Evolución de Java y su Naturaleza Multiplataforma

Ilustración 1 - Logotipo Java

Java [22], con logotipo [**Ilustración 1 - Logotipo Java**], es un lenguaje de programación orientado a objetos y multiplataforma. El término "Multiplataforma" denota que un programa desarrollado en Java puede ejecutarse en cualquier dispositivo, ya sea con un sistema operativo Windows, Mac o cualquier otro.

La principal razón detrás de esta versatilidad es la Máquina Virtual de Java (Java Virtual Machine o JVM). El flujo de trabajo en Java es el siguiente:

- El código se escribe y se guarda con una extensión .java.

- Este código se compila con el compilador javac, transformándolo en bytecode, y resulta en un archivo con extensión .class.

- La JVM, dependiendo del sistema operativo, interpreta este bytecode y lo traduce a un código máquina específico.

Tomemos Minecraft como ejemplo. Cuando se instala, se solicita al usuario tener, como mínimo, la versión 8 de Java, ya que el juego se desarrolló con esa especificación. De este modo, se puede jugar en cualquier sistema, siempre y cuando se tenga la JVM adecuada. En caso contrario, si el usuario tiene una versión inferior a la 8, el juego no correría adecuadamente dando un fallo en el mismo.

La compatibilidad de versiones es esencial en el desarrollo y ejecución de programas. Al intentar correr un programa en una Máquina Virtual de Java (JVM) que no coincide con la versión de Java con la que fue creado, es probable que se presenten errores o malfuncionamientos. Los desarrolladores enfrentan este reto de compatibilidad de manera regular, y hay varias razones que subrayan su complejidad:

- **Actualizaciones frecuentes**: Con el avance tecnológico, los lenguajes y sistemas se actualizan regularmente. Estas actualizaciones pueden introducir nuevas características, pero también pueden descontinuar o modificar características antiguas.

- **Dependencia de bibliotecas**: Los programas a menudo dependen de bibliotecas externas. Si una biblioteca se actualiza y cambia su forma de funcionar o su interfaz, el programa que depende de ella puede dejar de funcionar correctamente.

- **Diferentes entornos**: Los desarrolladores a menudo tienen que asegurarse de que su software funcione en múltiples sistemas operativos, dispositivos y configuraciones. Cada uno de estos entornos puede tener sus propias peculiaridades que afecten la ejecución del programa.

- **Mantenimiento del código**: A medida que un programa crece y evoluciona, mantener la compatibilidad con versiones anteriores puede requerir un esfuerzo significativo. En ocasiones, los desarrolladores tienen que decidir entre implementar una nueva característica y mantener la compatibilidad.

- **Expectativas del usuario**: Los usuarios esperan que, al actualizar su sistema o software, los programas que ya tenían continúen funcionando sin problemas. Satisfacer esta expectativa es crucial para mantener la confianza y satisfacción del usuario.

Por todas estas razones, la compatibilidad es un aspecto que los desarrolladores deben tener en cuenta constantemente, requiriendo pruebas exhaustivas y, a menudo, adaptaciones y correcciones en el código para garantizar que los programas funcionen de manera óptima en diferentes escenarios y versiones.
Este enfoque en la compatibilidad y la adaptabilidad ha sido un factor en la evolución de los lenguajes de programación. En las últimas décadas, Java ha experimentado un crecimiento significativo en popularidad. Mientras que en tiempos anteriores, lenguajes como Pascal o Cobol dominaban la educación y la industria, hoy en día, universidades y empresas muestran una inclinación marcada hacia lenguajes como Java, Python o C. La robustez y versatilidad de Java lo han posicionado como una elección predilecta para muchos programadores y educadores en el mundo tecnológico actual.

3.1.2 Selenium

Ilustración 2 - Logotipo Selenium

Selenium [2], con logotipo [**Ilustración 2 - Logotipo Selenium**], es una herramienta de código abierto, con licencia bajo Apache 2.0, especialmente diseñada para automatizar acciones en navegadores web. Esta herramienta permite a desarrolladores y testers replicar acciones humanas en la web, como hacer clics o rellenar formularios, todo de forma automatizada.

Este entorno de pruebas software nació en 2004, proporcionando una solución bastante robusta a los desafíos inherentes de las pruebas en aplicaciones web. A lo largo de casi dos décadas, Selenium ha experimentado numerosas actualizaciones, ampliando sus capacidades y adaptándose a las cambiantes necesidades de la industria. Esta constante evolución lo ha consolidado como una de las plataformas líderes en el ámbito de las pruebas de software. Otros entornos que se destacan podrían ser: Appium [19], Cypress [20], UFT (Unified Functional Testing), entre otras.

Una de las principales fortalezas de Selenium radica en su versatilidad. La herramienta es compatible con una variedad de lenguajes de programación, permitiendo a los profesionales elegir aquel con el que se sientan más cómodos. Además, Selenium es capaz de trabajar con

la mayoría de los navegadores web populares, asegurando que las aplicaciones sean probadas en diferentes entornos para garantizar su funcionamiento.

Gracias a las capacidades de Selenium, las organizaciones pueden asegurar que sus aplicaciones y sitios web funcionen de manera óptima en diferentes escenarios y plataformas. Esto es crucial en el actual ecosistema digital, donde garantizar una experiencia de usuario consistente y sin errores puede ser determinante para el éxito de un proyecto. Con Selenium, las empresas tienen una herramienta potente para garantizar la calidad y eficacia de sus soluciones web.

3.1.2.1 Principales características de Selenium

Selenium se ha posicionado como una herramienta de automatización software bastante robusta y querida por la comunidad. Algunas de las razones de esto puede deberse a las principales características de este entorno:

- **Interfaz con múltiples lenguajes de programación**: Selenium proporciona compatibilidad con varios lenguajes de programación, incluidos Java, C#, Python, Ruby y Kotlin. Esto permite a los desarrolladores y testers elegir el lenguaje con el que están más cómodos para escribir sus scripts de prueba.

- **Soporte para múltiples navegadores**: Selenium ofrece la capacidad de ejecutar pruebas en una variedad de navegadores populares, incluyendo Chrome, Firefox, Safari, Edge, entre otros. Esto garantiza que las aplicaciones funcionen correctamente en todos los navegadores previstos, así como tener la posibilidad de crear pruebas que instancian diversos navegadores y poder observar la interacción entre estos.

- **Plataforma cruzada**: Con Selenium, los usuarios pueden desarrollar y ejecutar pruebas tanto en Windows, Linux y macOS. Esto es crucial para asegurarse de que las aplicaciones web sean consistentes en diferentes sistemas operativos.

- **Paralelización de pruebas**: Con herramientas como Selenium Grid, los usuarios pueden ejecutar pruebas en diferentes máquinas y navegadores simultáneamente. Esto ayuda a mejorar la eficiencia y velocidad de las pruebas.

- **Gratuito**: Con Selenium, cualquiera que quiera automatizar las pruebas de su web puede hacerlo de manera gratuita.

3.1.2.2 Justificación del uso de Java con Selenium

La combinación de Selenium con Java ha ganado una notable popularidad en la comunidad de pruebas automatizadas. Pero, ¿por qué muchos prefieren esta combinación sobre otros lenguajes?

- **Madurez y estabilidad**: Java es uno de los lenguajes de programación más antiguos y establecidos. A lo largo de los años, ha desarrollado una vasta biblioteca y una comunidad sólida. Esta madurez se traduce en estabilidad cuando se trata de pruebas automatizadas, un aspecto crítico para las empresas.

- **Amplia comunidad y gran soporte**: Debido a su larga existencia y popularidad, Java tiene una de las comunidades más grandes. Esto significa que, ante cualquier problema o duda con Selenium y Java, es probable que alguien ya se haya enfrentado contra ese desafío y compartido una solución o una mejor práctica.

- **Portabilidad**: Java es conocido por su lema "Escribe una vez, ejecuta en cualquier lugar". Esta naturaleza multiplataforma de Java hace que las pruebas automatizadas escritas para un sistema operativo puedan ser fácilmente portadas a otro.

- **Integración**: Muchas herramientas de integración continua y desarrollo, como Jenkins, están escritas en Java, lo que facilita la integración de pruebas Selenium escritas en Java.

- **Actualizaciones regulares**: Java recibe actualizaciones regulares, lo que garantiza que las pruebas automatizadas se beneficien de las últimas características y mejoras de seguridad.

Contrastando con otros lenguajes que pueden ser usados con Selenium:

Python es un lenguaje ampliamente popular para usar con Selenium debido a su sintaxis simple y legible. Es especialmente atractivo para quienes buscan escribir scripts de prueba de manera rápida. Sin embargo, en contextos empresariales donde la integración con herramientas orientadas a Java es crucial, este último podría presentar ventajas significativas. Por otro lado, C# destaca como una elección excelente en entornos que se inclinan predominantemente hacia Microsoft. En situaciones donde una organización ya está arraigada en herramientas y tecnologías basadas en .NET, utilizar Selenium con C# parece ser una transición lógica, aunque Java sigue ofreciendo una portabilidad superior en variados sistemas operativos. En cuanto a Ruby, a pesar de que, en combinación con el framework de pruebas Capybara, puede resultar bastante potente, su comunidad no es tan extensa como la de Java o Python. Esto puede llevar a una disponibilidad reducida de soporte y recursos cuando se enfrentan problemas específicos.

En resumen, aunque existen varios lenguajes que se pueden usar con Selenium, la elección de Java se justifica por su estabilidad, comunidad, portabilidad y facilidad de integración.

3.1.2.3 Usos de Selenium

Como se ha explicado previamente, Selenium es un framework de pruebas automatizadas para aplicaciones web. No obstante, no todos los tipos de pruebas pueden someterse a esto, aunque sí una gran variedad de ellas. Es por ello que a continuación se explicarán algunos usos que tiene Selenium:

- **Pruebas de regresión**: Las pruebas de regresión garantizan que el código nuevo no haya introducido defectos en las funcionalidades existentes. Selenium es ampliamente utilizado para este tipo de pruebas, dada su capacidad para automatizar y repetir las mismas pruebas una y otra vez.

- **Pruebas funcionales**: Estas pruebas se centran en la funcionalidad de las aplicaciones, verificando que trabajen según las especificaciones y requisitos. Selenium permite automatizar estos procesos, proporcionando feedback rápido y consistente a los equipos de desarrollo.

- **Pruebas de integración**: Estas pruebas verifican que diferentes módulos o servicios de una aplicación trabajen juntos correctamente. Selenium puede ser utilizado para automatizar estas pruebas, especialmente cuando se integra con otros servicios y aplicaciones web.

- **Pruebas en Múltiples Sistemas Operativos**: Con Selenium, es posible ejecutar pruebas en diferentes sistemas operativos, garantizando la funcionalidad de la aplicación en diversos entornos.

- **Creación de Data-Driven Tests**: Estas pruebas utilizan datos de entrada desde archivos externos, como hojas de cálculo o bases de datos, para probar el mismo flujo de trabajo con diferentes conjuntos de datos.

- **Pruebas End-to-end (E2E)**: Este tipo de pruebas simulan acciones del usuario en una aplicación web de principio a fin para asegurarse de que todos los flujos y funcionalidades de la aplicación funcionen como se espera en diferentes plataformas y navegadores. Estas pruebas son esenciales para garantizar que la aplicación sea robusta y libre de errores críticos antes de su lanzamiento al público o antes de actualizaciones importantes

- **Pruebas de UI (User Interface)**: Este tipo de pruebas se encarga de probar los elementos que se ven presentes en la interfaz de usuario con la finalidad de comprobar

que estos funcionen correctamente tal y como lo haría un usuario real. Dentro de estas interacciones nos encontramos los elementos los cuales son capturados por el driver de selenium, ya sea un botón, como un desplegable o una caja de texto o formulario.

3.1.2.4 Desventajas de Selenium

A pesar de su popularidad y versatilidad, Selenium tiene algunas desventajas que es importante tener en cuenta al considerar herramientas de automatización de pruebas. Algunas de las desventajas más notorias pueden ser:

- **Curva de aprendizaje**: Para aquellos que son nuevos en la automatización de pruebas o en la programación en general, Selenium puede presentar una curva de aprendizaje pronunciada. No obstante, si se tienen nociones sobre programación e informática, no debería considerarse un impedimento.

- **Configuración inicial**: Establecer un entorno de pruebas funcional con Selenium puede ser complicado..

- **Tiempo de ejecución**: Las pruebas automatizadas con Selenium pueden ser relativamente lentas en comparación con algunas otras herramientas modernas, especialmente cuando se ejecutan en un solo hilo o máquina.

- **No soporta aplicaciones móviles nativas**: Aunque Selenium puede trabajar con aplicaciones web móviles, no es adecuado para la automatización de aplicaciones móviles nativas. Para esto, herramientas como Appium son más apropiadas.

- **Limitaciones con aplicaciones de un solo componente (SPA)**: Las aplicaciones web modernas, especialmente las SPAs construidas con frameworks como Angular, React o Vue, pueden presentar desafíos debido a su dinamismo y tiempos de carga asíncronos.

- **Dependencia de plugins**: Para algunas funciones, Selenium puede depender de plugins o extensiones de terceros, lo que puede agregar complejidad y puntos de falla potenciales a la configuración.

- **Mantenimiento**: Dado que las interfaces web pueden cambiar con frecuencia, los scripts de prueba creados con Selenium pueden requerir un mantenimiento regular para asegurarse de que sigan siendo relevantes y funcionales con las versiones actualizadas de una aplicación.

- **Limitaciones en la detección de elementos**: En ocasiones, Selenium puede enfrentar problemas para interactuar con ciertos elementos, especialmente si están ocultos o presentan comportamientos complejos.

3.1.2.5 Identificación de los elementos y el DOM

Con la finalidad de poder comprender esta herramienta a un nivel más profundo y entender así correctamente su funcionamiento, es importante hablar de cómo se realiza la identificación de los elementos.

La identificación de elementos se refiere al proceso de localizar y seleccionar componentes específicos dentro de una página web con el propósito de interactuar con ellos durante las pruebas automatizadas. Estos elementos son los distintos componentes de una página web que corresponden a las etiquetas y atributos del DOM (Modelo de Objeto de Documento) de la página web.

El DOM, Document Object Model , es una interfaz de programación que proporciona una representación estructurada y lógica de documentos, permitiendo su manipulación. En términos web, el DOM, al transformar documentos como HTML o XML, crea una estructura en forma de árbol donde cada etiqueta HTML, como <p>, <div> o <a>, se representa como un nodo específico en dicho árbol. Esta representación es especialmente dinámica, permitiendo a los desarrolladores realizar modificaciones en tiempo real utilizando lenguajes de programación, siendo JavaScript uno de los más comunes. Sin embargo, lo que resalta de la API del DOM es su universalidad; no se limita únicamente a JavaScript, sino que admite interacciones con diversos lenguajes de programación. Para mantener una experiencia web uniforme, el World Wide Web Consortium (W3C) ha estandarizado el DOM, esperando que los navegadores lo implementen de forma coherente. Una de las fortalezas principales del DOM es permitir la manipulación directa del contenido, estructura y estilo de un documento. Esto facilita la creación de páginas web vivaces y adaptables, donde el contenido puede cambiar en función de las acciones del usuario o incluso animarse. Además, no solo se centra en el contenido, sino que también se entrelaza con los estilos. Al interactuar con CSS, los estilos definidos se aplican directamente a los nodos del DOM, influenciando así el diseño visual que los usuarios ven en sus navegadores.

El dinamismo que presenta el DOM es una característica muy importante que está intrínsecamente relacionada cuando se habla de crear un script de automatización con Selenium. Esta relación se puede ver en diversas partes:

- **Cambio constante del contenido web**: Las páginas web modernas suelen ser dinámicas, lo que significa que el contenido, la estructura y el estilo pueden cambiar en función de diversas acciones o eventos. Con Selenium, se pueden escribir scripts de automatización que interactúen con una página tal y como lo haría un usuario real.

Si el DOM de una página cambia, es crucial que los scripts de Selenium sean lo suficientemente robustos como para adaptarse o, al menos, identificar estos cambios y reportarlos.

- **Localización de elementos**: Selenium se basa en la identificación de elementos del DOM para interactuar con ellos (por ejemplo, hacer clic en un botón o ingresar texto en un campo). Si el DOM es dinámico y cambia, puede afectar la capacidad de Selenium para localizar y trabajar con estos elementos.

- **Esperas y asincronía**: Debido al dinamismo del DOM, a menudo es necesario que los scripts de Selenium esperen a que ciertos elementos aparezcan, desaparezcan o cambien su estado. Las funciones de espera, como las esperas explícitas en Selenium, son esenciales para manejar la asincronía y los cambios en el DOM.

- **Validación de cambios**: Uno de los propósitos de la automatización de pruebas es validar que los cambios en el software funcionen correctamente. Dado que el DOM puede reflejar estos cambios (nuevos elementos, estilos modificados, contenido actualizado), Selenium puede ser utilizado para verificar que estos cambios en el DOM sean los esperados y que no introduzcan errores.

- **Interacción con componentes dinámicos**: Muchas páginas web modernas utilizan tecnologías como AJAX (Asynchronous JavaScript and XML) para cargar y modificar contenido dinámicamente sin necesidad de recargar toda la página, es decir, de manera asincrónica. El proceso asincrónico no solo implica esperar a que los elementos estén disponibles, sino también gestionar estados transitorios. Es común que, al ejecutar una llamada AJAX, surjan indicadores temporales en la interfaz, tales como spinners o notificaciones de "Cargando". Para Selenium, reconocer y manejar eficientemente estos estados intermedios se vuelve esencial para asegurar una interacción continua y para prevenir posibles fallos en la prueba.

Selenium proporciona múltiples estrategias de localización para identificar estos elementos. A continuación, se enumeran algunos de los elementos comunes que se pueden identificar y las estrategias típicas para localizarlos

- **ID**:
    - Descripción: Esta estrategia utiliza el atributo id del elemento para su identificación.
    - Características: Esta técnica es particularmente rápida y eficaz, especialmente cuando el ID es único en el contexto de la página.

- **Name**:
  - Descripción: Esta estrategia utiliza el atributo name del elemento para su identificación.
  - Características: Es una opción viable cuando el atributo name es distintivo.

- **Class Name**:
  - Descripción: Esta estrategia utiliza el atributo class del elemento para su identificación.
  - Características: Es especialmente útil cuando se trata de agrupaciones de elementos que comparten una misma clase.

- **Tag Name**:
  - Descripción: Esta estrategia utiliza la identificación del elemento por medio de su etiqueta para su identificación.
  - Características: Aunque puede no ser específica, es útil para categorías generales de elementos, como <a>, <button>, entre otros.

- **Link Text**:
  - Descripción: Esta estrategia consiste en localizar un enlace mediante el uso de su texto completo para la localización del elemento.
  - Características: Es altamente efectiva para elementos <a>, pero depende de la singularidad del texto del enlace.

- **Partial Link Text**:
  - Descripción: Esta estrategia utiliza una porción del texto del enlace para la identificación del elemento.
  - Características: Provee flexibilidad, pero requiere que el fragmento del texto sea distintivo.

- **XPath**:
  - Descripción: Esta estrategia se apoya en expresiones XPath para la identificación de elementos. Pueden ser absolutas o relativas.
  - Características: Posee una alta versatilidad y es especialmente útil para elementos que carecen de atributos id o name únicos. Puede utilizarse de forma compleja, permitiendo la navegación a través de la estructura jerárquica del documento.

Al seleccionar una estrategia de localización en Selenium, es esencial considerar varios factores, tales como la velocidad de respuesta, la estabilidad o la complejidad, para garantizar la precisión y eficiencia de las pruebas automatizadas.

- **Velocidad**:
  - ID y Name: Son las formas más rápidas de localizar elementos. Los navegadores están optimizados para buscar estos atributos de manera eficiente. Utilizar el ID es particularmente rápido ya que se espera que sea único dentro de una página.

  - Class Name: Aunque es rápido, puede ser menos preciso si varias instancias del mismo nombre de clase existen en la página.

  - Tag Name: Su velocidad es moderada, pero su precisión puede no ser óptima, especialmente si se usa para etiquetas comunes como <div> o <a>.

- **Estabilidad**:

  - ID: Suele ser la estrategia más estable, ya que el ID de un elemento raramente cambia a menos que haya una revisión significativa en el diseño o funcionalidad de la página.

  - Name: También es estable, especialmente para campos de formularios y otros elementos que requieren interacción del usuario.

  - Class Name y Tag Name: Pueden ser menos estables si el diseño o la estructura de la página cambian frecuentemente.

  - CSS Selector y XPath: Dependiendo de cómo se construyen, pueden ser susceptibles a cambios en la estructura de la página. Los selectores que dependen fuertemente de la jerarquía pueden romperse si hay cambios en la disposición de los elementos.

- **Complejidad**:

  - ID, Name, Class Name y Tag Name: Son simples y directos. Estas estrategias no requieren un conocimiento avanzado para ser utilizadas eficientemente.

  - CSS Selector: Aunque es más complejo que los anteriores, permite una gran flexibilidad. Los selectores pueden ir desde simples, como #id o .class, hasta más avanzados que representan relaciones jerárquicas o estados específicos del elemento.

  - XPath: Es la estrategia más compleja. Aunque proporciona una potente herramienta para navegar a través de la estructura del documento, requiere un entendimiento más profundo y puede volverse complicado, especialmente cuando se utilizan funciones avanzadas de XPath.

En conclusión, al elegir una estrategia de localización en Selenium, es esencial sopesar estos factores. Mientras que la velocidad es crítica para pruebas que requieren una ejecución rápida, la estabilidad garantiza que las pruebas no se rompan con pequeños cambios en la aplicación. La complejidad, por otro lado, afecta el mantenimiento y la legibilidad de los scripts de prueba. Por todo esto es vital el equilibrio adecuado según las necesidades y posibilidades específicas del proyecto.

3.1.3 Github Actions

Ilustración 3 - Logotipo Github Actions

GitHub Actions [7], con logotipo [**Ilustración 3 - Logotipo Github Actions**], es una plataforma de integración continua y entrega continua (CI/CD) proporcionada por GitHub. Permite a los desarrolladores automatizar, personalizar y ejecutar flujos de trabajo directamente en sus repositorios de GitHub. A través de GitHub Actions, los usuarios pueden definir eventos específicos dentro de sus repositorios, como push o pull request, para desencadenar automáticamente flujos de trabajo.

Para continuar el concepto y las prácticas de Github Actions, se explicarán el concepto CI/CD, pues es importante que se entienda antes de continuar.

La integración continua (CI, Continuous Integration) Es una práctica que implica la integración automática y frecuente del código de diferentes desarrolladores en un repositorio compartido. A medida que se incorporan los cambios, estos se validan mediante pruebas automatizadas para detectar errores lo más pronto posible. Los objetivos principales son detectar y corregir errores rápidamente, mantener un código base limpio y funcional y automatizar las pruebas para validar la calidad del código.

Por otro lado, el término CD en el ámbito de GitHub Actions puede referirse a dos prácticas distintas: "Continuous Delivery" y "Continuous Deployment". La diferenciación entre ambos depende, en gran medida, de cómo un usuario decida configurar su flujo de trabajo en la plataforma. A pesar de la dualidad anteriormente mencionada, cuando se aborda el tema desde una perspectiva terminológica estricta y dentro del contexto más adoptado, "Continuous Delivery" emerge como la interpretación principal asociada a CD en GitHub Actions. La Entrega continua permite que los cambios en el código, desde nuevas características hasta correcciones, estén listos para ser desplegados en producción en cualquier momento de manera segura. Por otro lado, el despliegue continuo es una extensión

de la Entrega Continua donde no solo se preparan los cambios para ser desplegados, sino que se despliegan automáticamente en producción sin intervención humana, siempre que pasen todas las pruebas y controles de calidad.

3.1.3.1 Componentes de Github Actions

Cuando ocurre un evento en el repositorio, como la iniciación de una solicitud de cambios o el reporte de un problema, se puede activar un proceso de GitHub Actions. Este proceso alberga tareas que pueden realizarse en secuencia o simultáneamente. Cada tarea opera dentro de una máquina virtual individual o un contenedor. Además, estas tareas comprenden etapas que pueden ejecutar un código definido o una función preestablecida, la cual facilita el proceso general.

- **Flujos de Trabajo (Workflows)**: Un flujo de trabajo es una serie de pasos automatizados definidos que se ejecutan en respuesta a un evento o manualmente si así se desea. Los flujos de trabajo se definen utilizando archivos YAML en el directorio .github/workflows/ del repositorio.

- **Eventos (Events)**: Son los desencadenantes de los flujos de trabajo. Pueden ser acciones que ocurren directamente dentro de GitHub (como hacer un push, abrir un pull request, etc.) o eventos externos que llegan a través de la API de GitHub.

- **Trabajos (Jobs)**: Los trabajos son un conjunto de pasos que se encuentran en el flujo de trabajo. Estos pasos son ejecutados siguiendo un orden establecido y puesto que son ejecutados dentro de un mismo runner, son capaces de compartir datos entre ellos. Dentro de un flujo de trabajo, se definen trabajos que se ejecutan en máquinas virtuales separadas. Cada trabajo puede tener dependencias en otros trabajos a pesar de no ser lo establecido por defecto, lo que permite definir un orden de ejecución.

- **Acciones (Actions)**: Son piezas reutilizables de código que pueden ser compartidas entre diferentes flujos de trabajo. Por ejemplo, hay acciones para verificar el código fuente, para desplegar aplicaciones en plataformas específicas o para interactuar con servicios externos. Se pueden usar acciones proporcionadas por la comunidad, por organizaciones o crear propias. La finalidad de las acciones consisten principalmente en reducir la cantidad de código repetitivo.

- **Runners (Ejecutores)**: Son servidores en los que se ejecutan los trabajos de tus flujos de trabajo. GitHub proporciona runners para los sistemas operativos de: Microsoft Windows, macOS, o Ubuntu Linux. Sin embargo, también es posible la configuración propia para así usar runners propios si se tienen requisitos específicos.

3.1.3.2 Comparativa con otras herramientas CI/CD

El mundo de la Integración Continua y Entrega Continua (CI/CD) está repleto de herramientas que ofrecen una variedad de funcionalidades para automatizar y optimizar el proceso de desarrollo de software. GitHub Actions es uno de los entrantes más recientes a este espacio, pero ha ganado rápidamente tracción gracias a su integración directa con GitHub. No obstante, no es la única herramienta y tampoco la más popular.

Por ese motivo, a continuación se va a hacer una comparación con otras herramientas CI/CD y se explicará el motivo detrás de la utilización de Github Actions frente a las otras herramientas

- **Jenkins**:

Jenkins [8], con su reconocida madurez y flexibilidad, ha dominado el mercado de CI/CD durante muchos años, respaldado por una vasta gama de plugins desarrollados por su comunidad. Su característica auto-hospedada ofrece a las organizaciones un control absoluto sobre su infraestructura, siendo vital para aquellas con demandas rigurosas de seguridad y cumplimiento. Sin embargo, esta fortaleza también es su debilidad; Jenkins puede ser complicado en términos de configuración y mantenimiento, y su interfaz, aunque completamente funcional, puede parecer anticuada en comparación con herramientas más recientes. Por otro lado, GitHub Actions emerge como una solución moderna, ofreciendo una integración directa con GitHub, simplificando el proceso para los proyectos alojados en dicha plataforma. La facilidad de uso de GitHub Actions, con su enfoque en archivos YAML y una amplia biblioteca de acciones predefinidas, contrasta con las complejidades de Jenkins. A diferencia de Jenkins, GitHub Actions es un servicio basado en la nube gestionado por GitHub, lo que reduce las preocupaciones sobre el mantenimiento y escalabilidad, adaptándose automáticamente a las necesidades del proyecto y liberando a los usuarios de las cargas administrativas típicas de las soluciones auto-hospedadas.

- **CircleCI**:

CircleCI [10], conocido por su robustez y eficiencia, ha sido una herramienta esencial en el paisaje del CI/CD, apreciado por su rapidez y características de integración. Al igual que Jenkins, CircleCI ofrece opciones tanto en la nube como auto-hospedadas, lo que da a las organizaciones un grado de flexibilidad según sus requisitos. Sin embargo, algunas de las características que lo hacen potente también pueden introducir complejidades en cuanto a configuración y mantenimiento. Además, a pesar de su interfaz moderna, puede requerir una curva de aprendizaje para los nuevos usuarios.

- **Travis CI**:

Travis CI [9], una solución independiente que ganó reconocimiento en el ámbito de la integración continua, fue adquirida eventualmente por Idera, marcando su posición en el mercado durante muchos años, en especial dentro de proyectos de código abierto. Sin embargo, en 2019, GitHub, una subsidiaria de Microsoft, lanzó GitHub Actions, integrándose de forma nativa dentro de su ecosistema, eliminando la necesidad de configuraciones externas que sí eran requeridas en Travis CI. Aunque Travis CI disfrutó de una amplia popularidad, los recientes cambios en su modelo de precios, combinados con la flexibilidad y las características de GitHub Actions, llevaron a muchos proyectos a migrar, posicionando a GitHub Actions como una herramienta en alza dentro de la comunidad de desarrolladores.

En conclusión, aunque Travis CI y GitHub Actions tienen funcionalidades similares en términos de CI/CD, GitHub Actions tiene la ventaja de una integración más profunda con GitHub y una mayor flexibilidad gracias a su diseño basado en acciones y flujos de trabajo.

3.1.4 Extent Report

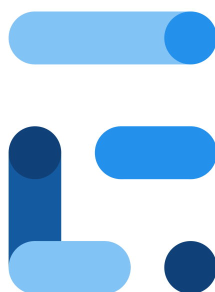

Ilustración 4 - Logotipo Extent Report

En el extenso y dinámico universo del aseguramiento de calidad de software, Extent Reports [11], con logotipo [**Ilustración 4 - Logotipo Extent Report**], emerge como una herramienta vital y de alto impacto que facilita la comunicación efectiva de los resultados de las pruebas automatizadas. Este framework de generación de informes, ampliamente reconocido y adoptado en la industria, empodera a los equipos de desarrollo y QA (Aseguramiento de Calidad) al proporcionar una visión clara y detallada del éxito o el fracaso de las pruebas ejecutadas, permitiendo, así, un análisis profundo. ExtentReports no solo sirve como un compendio de los resultados de las pruebas, sino que también enriquece los informes con una gama de funciones que permiten una exploración detallada del desempeño de las pruebas, al ofrecer gráficos, logs y una visualización amigable de los datos. La rica interfaz gráfica y la capacidad de personalización permiten a los equipos crear informes atractivos, cohesivos y, sobre todo, informativos que se convierten en una parte integral del flujo de trabajo de calidad, facilitando así la identificación de áreas de mejora y la elaboración de estrategias correctivas eficaces en los proyectos de software.

A continuación se hará una breve comparativa de las ventajas y las desventajas que supone el uso de esta herramienta

- **Ventajas**:

    - Informes Visuales: Extent Reports produce informes detallados y visualmente atractivos de forma sencilla, que facilitan la comprensión y el análisis de los resultados de las pruebas.

    - Personalización: Permite la personalización de informes según las necesidades específicas de un equipo o proyecto, incluyendo estilos, logos y detalles adicionales.

    - Integración: Ofrece la capacidad de integrarse con herramientas de pruebas populares como TestNG, JUnit, entre otros, lo que facilita su implementación en diversos proyectos.

    - Registro Detallado: Los informes pueden contener logs detallados de cada prueba, lo que facilita la identificación de errores o fallos.

    - Soporte para Screenshots: Es posible adjuntar capturas de pantalla a los informes, lo que es útil para documentar errores visuales o problemas de interfaz.

    - Multiplataforma: Funciona tanto en proyectos basados en Java como en .NET, ampliando su alcance y utilidad.

- **Desventajas**:

    - Curva de Aprendizaje: Aunque su configuración básica puede ser sencilla, aprovechar todas sus características avanzadas requiere una cierta curva de aprendizaje.

    - Dependencia de Versiones: Las nuevas versiones del framework pueden introducir cambios que no son compatibles con versiones anteriores, lo que puede causar problemas si no se realiza un mantenimiento adecuado del código.

    - Rendimiento: Generar informes detallados puede influir en el tiempo que toma ejecutar las pruebas, especialmente si se están capturando muchas capturas de pantalla o se están registrando muchos detalles.

- Necesidad de Almacenamiento: Los informes generados, especialmente si incluyen capturas de pantalla, pueden consumir una cantidad considerable de espacio de almacenamiento.

- Limitaciones en la Versión Gratuita: Aunque existe una versión gratuita de ExtentReports, algunas características avanzadas solo están disponibles en la versión de pago.

En la imagen [**Ilustración 5 - Ejemplo Extent Report**] se puede observar un ejemplo del reporte generado

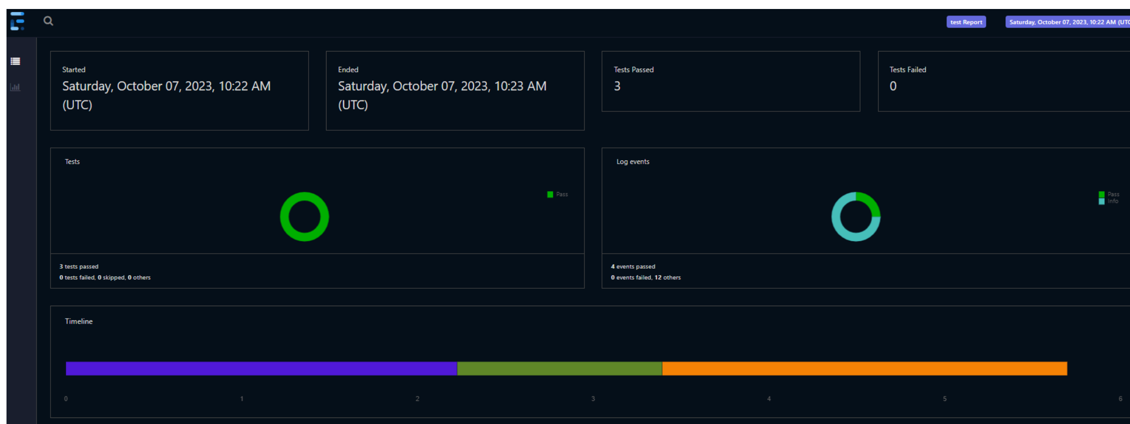

Ilustración 5 - Ejemplo Extent Report

En conclusión, Extent Reports es una herramienta poderosa y versátil para la generación de informes de pruebas automatizadas. Sin embargo, como con cualquier herramienta, es esencial entender sus características y limitaciones para aprovecharla al máximo y evitar posibles inconvenientes.

3.1.5 Librerías

A continuación se van a explicar brevemente las diferentes librerías de las cuales el proyecto depende. A través de la integración de estas, se busca no solo facilitar el proceso de desarrollo, sino también asegurar que las implementaciones sean robustas y confiables. Cada una de las librerías seleccionadas cumple un papel específico dentro del ciclo de vida del proyecto, contribuyendo a la optimización de las pruebas, la interacción con diferentes plataformas y la generación de informes detallados. En conjunto, estas herramientas conforman la columna vertebral tecnológica del proyecto, brindando una infraestructura sólida.

- **JUnit**:
  - Popular framework de pruebas unitarias para Java. Es utilizado para escribir y ejecutar pruebas automatizadas en aplicaciones Java.

- **JUnit Jupiter**:
    - Es el nombre de la nueva API y el modelo de programación en JUnit 5. Facilita la escritura y ejecución de pruebas unitarias en aplicaciones Java modernas.

- **Selenium API**:
    - Proporciona interfaces y clases para trabajar con Selenium, una herramienta popular para automatizar navegadores. Es la base para desarrollar pruebas de automación con Selenium.

- **Selenium Grid**:
    - Es una herramienta de Selenium que permite ejecutar pruebas en diferentes máquinas (físicas o virtuales) y navegadores de manera simultánea. Facilita la ejecución paralela y distribuida de pruebas.

- **Selenium Remote Driver**:
    - Permite controlar un navegador o una aplicación web en una máquina remota usando Selenium. Esencial para pruebas distribuidas y en la nube.

- **AssertJ Core**:
    - Es una biblioteca de aserciones para escribir afirmaciones en pruebas de Java de manera más legible y con autocompletado. Proporciona un conjunto rico de aserciones y es una alternativa a las aserciones tradicionales de JUnit.

- **Selenium Java**:
    - Es el conjunto de bindings de Java para Selenium. Permite escribir pruebas automatizadas para navegadores web en Java utilizando Selenium.

- **Apache Commons IO (desde Apache Directory Studio)**:
    - Proporciona utilidades para operaciones de entrada/salida, como leer y escribir archivos. Simplifica tareas comunes de IO en Java.

- **Selenium Devtools v102**:
    - Proporciona acceso a las Chrome DevTools a través de Selenium. Es útil para pruebas avanzadas y automatización que requieren interactuar con las herramientas de desarrollo de Chrome.

- **ExtentReports**:
    - Es una biblioteca de generación de informes para pruebas automatizadas. Permite crear informes detallados y estéticos sobre la ejecución de pruebas.

- **WebDriverManager**:
    - Es una biblioteca que permite manejar automáticamente los drivers de navegadores para Selenium WebDriver (como chromedriver). Se encarga de

descargar, almacenar en caché y configurar los drivers necesarios para las pruebas.

- **Apache POI**:
  - Es una biblioteca para leer y escribir archivos en formatos Microsoft Office, como Excel (XLS y XLSX). Es útil cuando las pruebas necesitan interactuar con datos en archivos Excel.

- **Apache POI OOXML**:
  - Es un componente de Apache POI que se ocupa específicamente de los formatos Office Open XML (OOXML) como XLSX de Excel.

- **io.github.bonigarcia:**
  - Es una librería gestionada por io.github.bonigarcia utilizada en proyectos Java para automatizar la gestión de los drivers de navegadores web, como ChromeDriver entre otros, que son necesarios para ejecutar pruebas automatizadas con Selenium WebDriver.

3.2 Metodología empleada

Para el correcto desarrollo de un proyecto, sin importar la naturaleza del mismo, es importante seguir una metodología de trabajo que dictamine cuál será la forma de trabajo. De esta manera, se puede asegurar que se realiza un avance seguro en el proyecto en el cual se está trabajando. Definir el método de trabajo previo al inicio del proyecto es crucial, ya que si no se realiza, se podría complicar innecesariamente la labor.

Para este proyecto se ha usado una metodología iterativa.

3.2.1. Metodología Iterativa

La metodología iterativa se refiere a un enfoque de desarrollo de proyectos, especialmente en el ámbito del software, donde el proceso se lleva a cabo a través de ciclos repetidos o iteraciones. Cada iteración es una versión pequeña del proceso de desarrollo completo, abarcando desde la definición de requisitos hasta las pruebas. La idea es que el proyecto se vaya refinando con cada ciclo basado en el feedback y los aprendizajes adquiridos en las iteraciones anteriores.

No obstante, dado el actual proyecto, es importante refinar la definición al ámbito del que se está hablando. Es por ello que en el ámbito de un proyecto basado en pruebas automatizadas la metodología iterativa adapta su definición. De esta manera, se puede definir la metodología iterativa con una serie de pasos:

- **Definición de Requisitos de Prueba**:
  - En la primera iteración, se definen los casos de prueba clave basados en los requisitos del software y/o en las necesidades que requiera el mismo en lo que automatizar se refiere.
  - Identificar áreas críticas, funciones y flujos de usuario que requieran pruebas automatizadas.

- **Diseño e Implementación**:
  - Diseñar scripts de prueba automatizados para los casos identificados utilizando herramientas y frameworks adecuados para la automatización.

- **Ejecución de Pruebas**:
  - Ejecutar los scripts automatizados en el ambiente de pruebas en local.
  - Documentar y registrar cualquier fallo o anomalía.
  - Repetir el proceso ejecutando en Github Actions.

- **Revisión y Feedback**:
  - Analizar los resultados.
  - Determinar si las pruebas cubren adecuadamente los requisitos previamente definidos.

- **Refinamiento**:
  - En base a los resultados, refinar y ajustar los scripts de prueba.
  - Añadir nuevos casos de prueba si se identifican áreas no cubiertas o si el software bajo prueba ha evolucionado.

- **Repetición**:
  - Con cada nueva iteración del software, repetir el proceso: ejecutar las pruebas, analizar, refinar y ajustar.

Recordando la naturaleza iterativa, es esencial revisar y ajustar continuamente. Esta adaptabilidad permite que las pruebas automatizadas se mantengan relevantes y efectivas a medida que el proyecto avanza.

Cabe mencionar que el presente proyecto se inició en noviembre de 2022. No obstante, el proyecto ha tenido periodos de inactividad. Estos periodos se han debido principalmente a épocas de intenso trabajo y poca disponibilidad.

3.2.2 Blog

Para la realización del proyecto, se ha llevado a cabo un blog [**Ilustración 6 - Blog Medium**] donde se han ido citando algunos de los avances y errores que se han ido encontrando a medida que se ha ido avanzando en el proyecto.

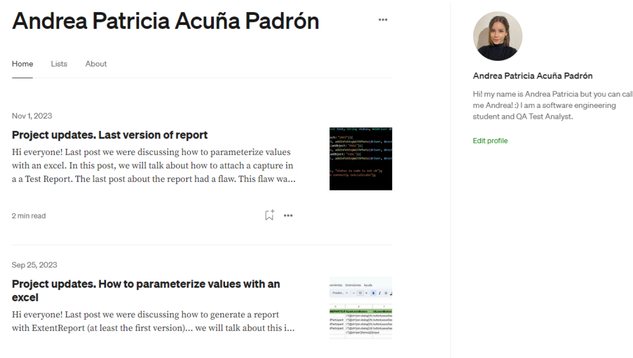

Ilustración 6 - Blog Medium

URL: https://medium.com/@acupadron

3.2.3 Repositorio Github

Para la realización del proyecto se ha hecho uso del siguiente repositorio Github, donde se pueden observar las ejecuciones en Github Actions así como los diversos commits y las fechas de los mismos. Pudiendo así ver el seguimiento del mismo.

URL: https://github.com/codeurjc-students/openvidu-tutorials-tests

Capítulo 4: Descripción informática

En este capítulo se hablará sobre la descripción informática seguida durante el proyecto así como una breve introducción teórica de cada apartado. Así, se observa la división en los siguientes apartados:

- **Definición de requisitos de pruebas**: En este apartado se hablará sobre los requisitos de pruebas especificados para el presente proyecto.

- **Diseño**: En este apartado se hablará sobre el diseño de los scripts y la elección de los mismos.

- **Implementación**: En este apartado se hablará más a detalle la implementación usada y algunas funciones del framework, Selenium.

- **Refinamiento**: En este apartado se hablará de problemas encontrados a la hora de ejecutar y refinar los scripts.

4.1 Definición de requisitos de pruebas

Los requisitos de prueba son las condiciones o especificaciones que una función o característica específica del software debe satisfacer para ser considerada adecuada o de acuerdo con lo previsto. Estos requisitos se derivan generalmente de los requisitos del software y sirven como base para diseñar y desarrollar casos de prueba automatizados. Además, estas condiciones son probadas manualmente en paralelo o anteriormente al desarrollo de los requisitos, de esta manera se comprueba que la automatización será satisfactoria y no hay ningún error o incidencia en el software.

Los requisitos de prueba desempeñan un papel esencial en la automatización porque:

- **Definen el alcance**: Ayudan a comprender qué aspectos del software se deben probar y qué no.

- **Aseguran la cobertura**: Aseguran que todas las funciones y características importantes del software estén cubiertas por los casos de prueba.

- **Facilitan la trazabilidad**: Permiten establecer una relación entre los requisitos originales del software y los casos de prueba, garantizando que todos los requisitos se estén probando adecuadamente.

- **Proporcionan criterios claros**: Ofrecen criterios específicos sobre lo que se considera un resultado exitoso o fallido en una prueba.

Cuando se automatizan pruebas, estos requisitos sirven como guía para desarrollar scripts de prueba que puedan ser ejecutados automáticamente, verificando si el software cumple con las especificaciones y expectativas definidas en los requisitos de prueba.

El proyecto consta de una serie de pruebas que se replican en los diversos "*Basic Tutorials*" que presenta el proyecto OpenVidu. Concretamente los siguientes: Hello World, openvidu-js, openvidu-angular, openvidu-react y openvidu-vue. Para cada tutorial disponible en la web, se han diseñado distintas pruebas. Estas, como ya se ha mencionado, necesitan una definición anticipada que detalle los pasos a seguir y el resultado esperado de cada paso para considerar la prueba como satisfactoria. Así, a continuación se mostrarán los diversos requisitos de pruebas:

T001_JoinSession	
Descripción: Unirse a la sesión y verificar que ambos navegadores están dentro de la sesión.	
Paso	Resultado esperado
Ir a la URL correspondiente	Página de inicio cargada con éxito
Configurar nombre de sesión y nombre de participante si corresponde en ambos navegadores	Nombre de sesión y participantes se muestran en pantalla correctamente
Click en botón Join en ambos navegadores	Ambos navegadores entran en pantalla de videoconferencia

Tabla 1 - Requisito_T001

T002_LeaveSession	
Descripción: verificación de que el video se está reproduciendo correctamente y ambos navegadores salen de la sesión correctamente	
Paso	Resultado esperado
Ir a la URL correspondiente	Página de inicio cargada con éxito
Configurar nombre de sesión y nombre de participante si corresponde en ambos navegadores y click en botón Join en ambos navegadores	Ambos navegadores entran en pantalla de videoconferencia
Capturar video en ambos navegadores	El video se está reproduciendo exitósamente
Click en botón leave	Página de inicio cargada con éxito

Tabla 2 - Requisito_T002

T003_SessionHeader	
Descripción: Se une a la sesión y verifica que el nombre de la sesión sea el esperado	
Paso	Resultado esperado
Ir a la URL correspondiente	Página de inicio cargada con éxito
Configurar nombre de sesión y nombre de participante si corresponde en ambos navegadores y click en botón Join en ambos navegadores	Ambos navegadores entran en pantalla de videoconferencia
Localizar el nombre de la sesión	El nombre de la sesión es el esperado

Tabla 3 - Requisito_T003

T004_ParticipantName	
Descripción: Se une a la sesión y verifica que el nombre del participante de un navegador sea correcto	
Paso	Resultado esperado
Ir a la URL correspondiente	Página de inicio cargada con éxito
Configurar nombre de sesión y nombre de participante si corresponde en ambos navegadores y click en botón Join en ambos navegadores	Ambos navegadores entran en pantalla de videoconferencia

Localizar el nombre del participante	El nombre del participante es el esperado

Tabla 4 - Requisito_T004

Una vez definidos los requisitos de prueba es importante saber cuáles se aplican a cada uno de los Basic Tutorials automatizados. Así tenemos la siguiente distribución:

- **Hello World**: T001_JoinSession, T002_LeaveSession y T003_SessionHeader
- **Openvidu-js**: T001_JoinSession, T002_LeaveSession y T003_SessionHeader
- **Openvidu-angular**: T001_JoinSession, T002_LeaveSession, T003_SessionHeader y T004_ParticipantName
- **Openvidu-react**: T001_JoinSession y T003_SessionHeader
- **Openvidu-vue**: T001_JoinSession, T002_LeaveSession, T003_SessionHeader y T004_ParticipantName

4.2 Diseño

El diseño de un script puede ser de diversas maneras. A la hora de elegir el diseño de un script o un conjunto de scripts de automatización es importante tener en cuenta la naturaleza del proyecto y las necesidades que tenga el mismo. Esto es fundamental por diversos motivos entre los que se destacan:

- **Fiabilidad y consistencia**: Un diseño establecido asegura que el test automatizado siempre se comporte de la misma manera cuando se ejecute. Esto garantiza resultados consistentes y confiables, lo que es esencial para evaluar de manera precisa el funcionamiento de una aplicación o sistema.
- **Facilita la identificación de problemas**: Cuando un test tiene un diseño establecido, es más fácil identificar y diagnosticar problemas cuando algo falla. Si el test se comporta de manera inesperada, se puede revisar el diseño para encontrar posibles errores o ajustes necesarios.
- **Mantenibilidad**: Un diseño bien estructurado hace que los tests sean más fáciles de mantener a lo largo del tiempo. Cuando los requisitos o características del software cambian, es más sencillo actualizar o extender los tests si se basan en un diseño establecido.
- **Reusabilidad**: Los tests automatizados con un diseño establecido son más fáciles de reutilizar en diferentes partes del proyecto o incluso en futuros proyectos. Esto ahorra

tiempo y esfuerzo, ya que no es necesario crear nuevos tests desde cero cada vez que se realice una prueba similar.

- **Escalabilidad**: Los tests automatizados diseñados de manera estable se pueden escalar más fácilmente para cubrir una amplia gama de escenarios y funcionalidades del software. Esto es especialmente importante en proyectos grandes y complejos.

Los scripts del presente proyecto no constan de un diseño único si no que utilizan un diseño mixto cogiendo lo necesario de diversos diseños, elaborando así un diseño propio y personalizado adaptado a las necesidades del proyecto.

Así, se pueden observar los siguientes diseños:

- **Framework Basado en Datos (Data-driven Framework)**:

Es un enfoque comúnmente utilizado en la automatización de pruebas que separa los datos de prueba de los scripts de prueba. Este enfoque permite ejecutar los mismos scripts de prueba con diferentes conjuntos de datos, lo que facilita la reutilización y la escalabilidad de las pruebas. Así, el script lee las variables desde una fuente externa, como un archivo Excel o un CSV al principio de cada prueba. Esto es característico de un enfoque basado en datos, donde los datos de entrada se externalizan y se leen desde una fuente externa.

En el proyecto esto se puede observar en la carpeta: "`test-input`" [**Ilustración 7 - Localización parámetros**], donde se observa la clara separación de datos:

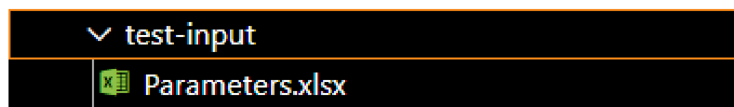

Ilustración 7 - Localización parámetros

Además, es importante que estos datos sean leídos en cada uno de los scripts de prueba. Así, y como ejemplo, podemos ver cómo estos datos son leídos por la función: *`readVariablesFromExcel`* en [**Ilustración 8 - @BeforeEach**], la cual se explicará en detalle en el punto 4.3 Implementación del presente capítulo, al inicio de cada prueba (*`@BeforeEach`*):

```java
/**
 * BeforeEach
 *
 * @author Andrea Acuña
 * Description: Execute before every single test.
 *              Configure the camera
 *              Set de url in each browser
 *              Read the variables from excel file
 */
    @BeforeEach
    void setup() {
        URL = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "URL");
        List<WebDriver> browsers = super.setUpTwoBrowsers();
        driverChrome = browsers.get(index: 0);
        driverFirefox = browsers.get(index: 1);
        driverChrome.get(URL);
        driverFirefox.get(URL);

        nameSession = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "NAMESESSION");
        NAMEPARTICIPANT = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "NAMEPARTICIPANT");
        XpathJoinButton = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "XpathJoinButton");
        xpathOtherCamera = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "xpathOtherCamera");
        xpathParticipant = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "xpathParticipant");
        idParticipant = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "idParticipant");
        idLeaveButton = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "idLeaveButton");
        idNameSession = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "idNameSession");
        idHeader = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "idHeader");
        idMainTitle = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "idMainTitle");
        idSelfCamera = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "idSelfCamera");
        idNameParticipant = readVariablesFromExcel(testLocation, testName: "OpenViduReactTest", ColValue: "idNameParticipant");
    }
```

Ilustración 8 - @BeforeEach

Este diseño le aporta al código varias propiedades de las cuales se puede sacar provecho:

- **Reutilización del Código**: Se escribe un único script de prueba que puede ser utilizado para diversos casos debido a que es posible ejecutarlo con múltiples conjuntos de datos. Esto evita repetir el mismo código para pruebas similares.

- **Flexibilidad**: hace referencia a la capacidad del framework para adaptarse fácilmente a cambios sin requerir grandes modificaciones. Así, Al cambiar los datos en la fuente de datos, se puede modificar el comportamiento de la prueba sin necesidad de alterar el código de la prueba. Así, se puede probar, por ejemplo, que con el cambio de una variable el script debe fallar, probando así un caso negativo.

- **Escalabilidad**: Se refiere a la capacidad del framework para manejar un aumento en la cantidad de datos de prueba o escenarios de prueba sin comprometer el rendimiento o requerir cambios significativos en la estructura existente. Así, es sencillo agregar más escenarios de prueba simplemente añadiendo más datos en la fuente de datos, sin necesidad de escribir más código.

- **Mantenimiento Reducido**: En caso de cambios en la aplicación o en las pruebas, es probable que solo se tenga que actualizar el código de la prueba en un lugar, manteniendo los datos intactos. En caso contrario, al ser necesario actualizar los datos, solo será necesario modificar el documento externo. Haciendo la labor de mantenimiento más sencilla.

- **Framework Orientado a Objetos**:

Es un enfoque que se basa en principios de programación orientada a objetos (POO) para diseñar y desarrollar pruebas automatizadas de manera eficiente y mantenible. Este planteamiento aprovecha las características de la POO, como la encapsulación, la herencia y el polimorfismo, para crear un diseño modular y reutilizable.

Este diseño se ve claramente reflejado con el siguiente diagrama de clases en UML [**Ilustración 9 - Diagrama Clases UML**]:

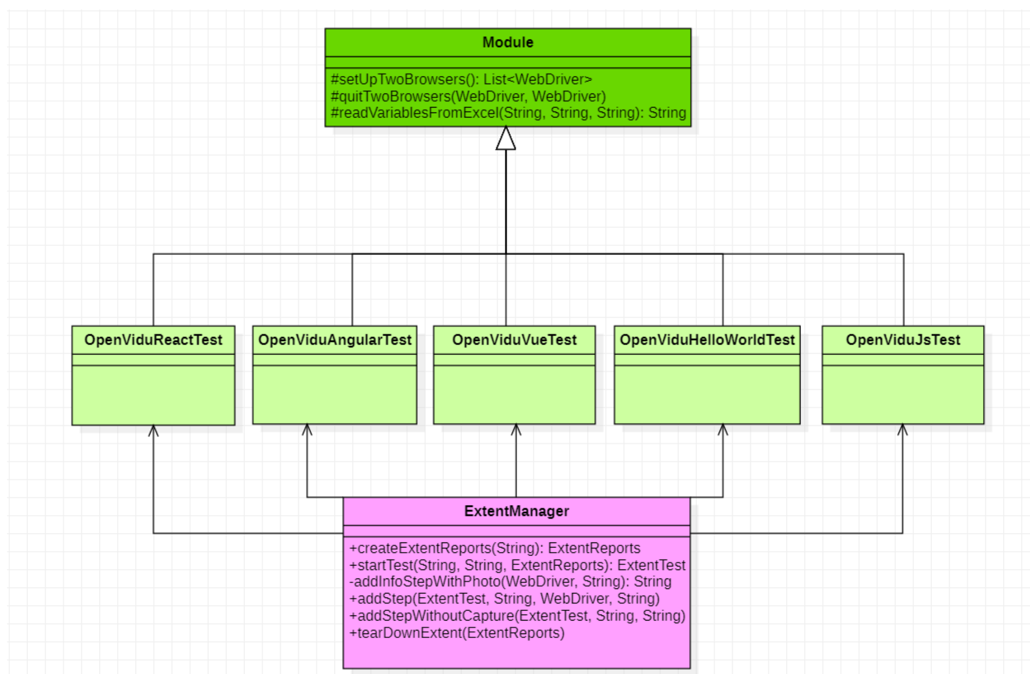

Ilustración 9 - Diagrama Clases UML

En el diagrama se puede observar cómo se representa la herencia de clases entre `Module`, la clase padre, y las diversas clases hijas, las cuales representan los scripts de automatización, para una óptima reutilización de código así como la encapsulación del mismo.

Por otro lado se puede observar la clase `ExtentManager`. Esta clase agrupa los diversos métodos para la correcta elaboración del reporte. Es por ello que cada una de las clases de scripts de automatización presentan un objeto `ExtentManager`.

- **Framework Modular o de Componentes**:

En un enfoque que se basa en la división del código. Lo que se pretende es dividir el proceso de prueba en distintas unidades o módulos independientes y reutilizables. Estos módulos se desarrollan y prueban individualmente y luego se combinan para formar pruebas más grandes y complejas.

Esto se ve reflejado en las dos clases externas a los scripts de automatización. Las clases: *`Module`* y `ExtentManager`.

En el caso particular de `Module` se puede encontrar un módulo encargado de la configuración y parametrización de los datos, [**Ilustración 10 - Module.Java Localización**]. Este módulo es independiente y reutilizable, de manera que se podría copiar y pegar en un proyecto ajeno facilitando el trabajo. Además, a la hora de mantener alguna funcionalidad relacionada con la configuración de los drivers o de la parametrización de los datos, habría que modificar exclusivamente dicha clase.

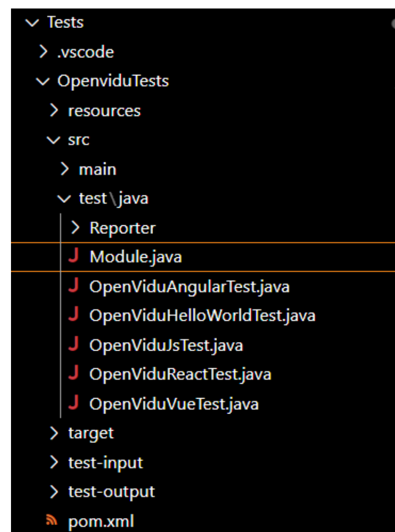

Ilustración 10 - Module.Java Localización

En el caso particular de `ExtentManager` se puede encontrar un módulo encargado de la generación del reporte y todo lo que este conlleva, [**Ilustración 11 - ExtentManager.Java Localización**]. Este módulo es independiente y reutilizable, de manera que se podría copiar y pegar en un proyecto ajeno facilitando el trabajo. Además, a la hora de mantener alguna funcionalidad relacionada con la configuración del reporte, la creación de un paso o con la imagen adjunta, habría que modificar exclusivamente dicha clase.

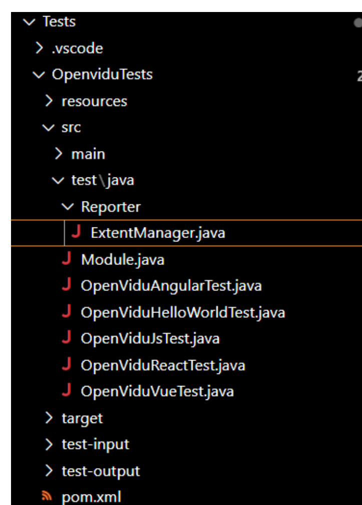

Ilustración 11 - ExtentManager.Java Localización

A continuación, se describen las ventajas de adoptar un diseño Framework Modular:

- **Reusabilidad del Código**: Una vez que un módulo ha sido creado y probado, puede ser reutilizado en diferentes pruebas o escenarios, lo que reduce la duplicación de esfuerzo y de código.

- **Mantenimiento Simplificado**: Si hay un cambio en una parte específica de la aplicación bajo prueba, solo el módulo correspondiente necesita ser actualizado, en lugar de modificar múltiples scripts de prueba.

- **Desarrollo Paralelo**: Como los módulos se pueden desarrollar de manera independiente, múltiples miembros del equipo pueden trabajar simultáneamente en diferentes módulos.

- **Modularidad y Estructura Clara**: El framework está claramente estructurado en módulos definidos, lo que facilita la comprensión y navegación a través del código.

- **Robustez**: Al aislar y probar módulos individualmente, se asegura que cada componente funcione correctamente antes de integrarlo en pruebas más amplias.

- **Integración con Otros Frameworks**: El diseño modular puede combinarse con otros enfoques, como el Data-driven Framework, para crear pruebas más robustas y flexibles.

- **Acelera la Identificación de Defectos**: Si surge un defecto durante la ejecución de una prueba, es más fácil identificar el problema cuando las pruebas están modularizadas, ya que se puede rastrear hasta un módulo específico.

En resumen, el diseño Framework Modular o de Componentes se basa en el principio de dividir y conquistar. Descompone las pruebas en piezas manejables que se pueden desarrollar, probar y mantener de manera independiente. Es especialmente útil en proyectos grandes donde la claridad, la reusabilidad y la eficiencia son esenciales.

4.3 Implementación

En el presente apartado se procederá a hacer una explicación de las partes más fundamentales del código. Concretamente se hará una explicación de la configuración de los drivers, la localización de elementos, las sincronizaciones, la captura de video, la generación de un reporte y la parametrización de datos.

4.3.1 Configuración de drivers

Un driver en Selenium es un componente específico que actúa como intermediario entre el código Selenium y un navegador concreto. Permite que Selenium interactúe con el navegador, enviando comandos para realizar acciones, como hacer click en un botón o escribir un texto en un formulario, y recibiendo información del navegador, como obtener texto o verificar estados.

Al escribir y ejecutar el script de prueba, el driver correspondiente solicita una nueva instancia del navegador especificado (por ejemplo, Chrome o Firefox) a través del driver correspondiente. Una vez que el driver inicia el navegador, establece una conexión con él. Para esto, el driver utiliza un protocolo llamado WebDriver, una API estandarizada. Cada acción realizada en el navegador se traduce en un comando que se envía desde el código de prueba al navegador a través del driver. Estos comandos se transmiten generalmente en formato JSON a través del protocolo HTTP. Al recibir un comando, el driver ejecuta la acción correspondiente en el navegador. Después de ejecutar el comando, el driver recopila cualquier resultado o información solicitada y la devuelve a tu código de prueba. Por ejemplo, si solicitas obtener el título de una página, el driver recoge esa información del navegador y la envía de vuelta a tu código. Una vez que todas las acciones y comandos han sido ejecutados y has obtenido la información deseada, puedes cerrar la instancia del navegador a través del driver.

Para el presente proyecto es importante que los drivers estén correctamente configurados. Así, se procede a explicar la configuración de los mismos:

- **Chrome:**

```
WebDriverManager.chromedriver().clearDriverCache().setup();

ChromeOptions options = new ChromeOptions();
options.addArguments("--headless", "--disable-gpu", "--ignore-certificate-errors","--disable-extensions","--no-sandbox","--disable-dev-shm-usage");
options.addArguments("--use-fake-ui-for-media-stream");
options.addArguments("--use-fake-device-for-media-stream");
options.addArguments("--remote-allow-origins=*");
driverChrome = new ChromeDriver(options);
browsers.add(driverChrome);
```

Ilustración 12 - Driver Chrome

Para un correcto funcionamiento en Github Actions la configuración debe ser la siguiente:

La primera línea de la imagen [**Ilustración 12 - Driver Chrome**] se puede observar que viene de la librería de io.github.bonigarcia:

`WebDriverManager.chromedriver().clearDriverCache().setup()`.
Dentro de esta línea es importante destacar los métodos correspondientes:

- **`WebDriverManager.chromedriver()`**: Esta llamada inicializa el gestor para el driver de Chrome. El WebDriverManager se encarga de descargar, configurar y manejar los binarios de los drivers automáticamente.

- **`clearDriverCache()`**: Este método borra cualquier versión previamente descargada del chromedriver. Así se verifica siempre estar utilizando la última versión..

- **`setup()`**: Este método configura el chromedriver para que esté listo para su uso. Esencialmente, ajusta la variable de sistema webdriver.chrome.driver con la ruta del binario del driver descargado o el existente.

La siguiente línea crea una nueva instancia de ChromeOptions, que permite personalizar el comportamiento del navegador Chrome:

```
ChromeOptions options = new ChromeOptions();
```

Posteriormente se procede a la configuración personalizada del navegador, esto se realiza añadiendo las opciones necesarias:

```
options.addArguments("--headless", "--disable-gpu",
"--ignore-certificate-errors","--disable-extensions","--no-sandbox
","--disable-dev-shm-usage");
options.addArguments("--use-fake-ui-for-media-stream");
options.addArguments("--use-fake-device-for-media-stream");
options.addArguments("--remote-allow-origins=*");
```

- **`--headless`**: Chrome se ejecuta en modo sin interfaz gráfica (headless). Es útil para ejecutar pruebas en entornos donde no hay interfaz gráfica disponible, como puede ser Github Actions.

- **`--disable-gpu`**: Deshabilita el uso de la GPU. Es comúnmente usado junto con el modo headless.

- **`--ignore-certificate-errors`**: Ignora los errores de certificados SSL/TLS. Útil para sitios con certificados no válidos o auto-firmados.

- **`--disable-extensions`**: Deshabilita todas las extensiones de Chrome.

- **`--no-sandbox`**: Desactiva el modo sandbox de Chrome.

- **`--disable-dev-shm-usage`**: La partición /dev/shm en algunos ambientes de VM es insuficiente, causando fallos o bloqueos en Chrome. Esta opción es necesaria para corregirlo.

- **`--use-fake-ui-for-media-stream`** y **`--use-fake-device-for-media-stream`**: Estos argumentos se usan para simular dispositivos multimedia durante las pruebas, evitando solicitudes de permisos para acceder a cámaras o micrófonos.

- **`--remote-allow-origins=*`**: Permite solicitudes desde cualquier origen cuando se utiliza el modo remoto.

Por último a destacar la línea:

```
driverChrome = new ChromeDriver(options);
```

La cual crea una nueva instancia de `ChromeDriver` con las opciones especificadas anteriormente. Así, `driverChrome` actuará como el controlador a través del cual se enviarán comandos al navegador.

- **Firefox**:

```
WebDriverManager.firefoxdriver().clearDriverCache().setup();

FirefoxOptions optionsF = new FirefoxOptions();
optionsF.setHeadless(true);
optionsF.addPreference("media.navigator.permission.disabled", true);
optionsF.addPreference("media.navigator.streams.fake", true);
driverFirefox = new FirefoxDriver(optionsF);
browsers.add(driverFirefox);
```

Ilustración 13 - Driver Firefox

La primera línea de la imagen [**Ilustración 13 - Driver Firefox**], se puede observar viene de la librería de bonigarcia: `bonigarcia.wdm.WebDriverManager`:

```
WebDriverManager.firefoxdriver().clearDriverCache().setup();
```

Dentro de esta línea es importante destacar los métodos correspondientes:

- **`WebDriverManager.firefoxdriver()`**: Esta llamada inicializa el gestor para el driver de Firefox. El WebDriverManager se encarga de descargar, configurar y manejar los binarios de los drivers (en este caso, geckodriver) automáticamente, evitando la necesidad de gestionar manualmente los archivos binarios y las rutas.

- **`clearDriverCache()`**: Limpia cualquier versión previamente descargada del geckodriver. Es útil para asegurarse de estar siempre utilizando la última versión o en caso de problemas con versiones anteriores.

- **`setup()`**: Configura el geckodriver para que esté listo para su uso. Ajusta la variable de sistema webdriver.gecko.driver con la ruta del binario del driver descargado o el existente.

La siguiente línea crea una nueva instancia de FirefoxOptions, que permite personalizar el comportamiento del navegador Firefox:

```
FirefoxOptions optionsF = new FirefoxOptions();
```

Posteriormente se procede a la configuración personalizada del navegador, esto se realiza añadiendo las opciones necesarias:

```
optionsF.setHeadless(true);
optionsF.addPreference("media.navigator.permission.disabled",
true);
optionsF.addPreference("media.navigator.streams.fake", true);
```

- **`setHeadless(true)`**: Configura Firefox para que se ejecute en modo sin interfaz gráfica (headless). Es útil para pruebas en entornos donde no hay una interfaz gráfica disponible, como en Github Actions.

- **`addPreference("media.navigator.permission.disabled", true)`**: Deshabilita los diálogos de permiso para acceder a dispositivos multimedia, como cámaras y micrófonos. De esta manera, Firefox no mostrará solicitudes de permisos para estos dispositivos.

- **`addPreference("media.navigator.streams.fake", true)`**: Configura Firefox para simular flujos de medios, evitando la necesidad de acceso real a dispositivos de cámara o micrófono.

Por último a destacar la línea:

```
driverFirefox = new FirefoxDriver(optionsF);
```

La cual crea una nueva instancia de FirefoxDriver con las opciones especificadas anteriormente. Así, driverFirefox actuará como el controlador a través del cual se enviarán comandos al navegador.

4.3.2 Localización de elementos

Para la localización de elementos se hará una breve descripción del código empleado para esto. Si se quiere hacer referencia a la teoría del mismo, se encuentra en el apartado 3.1.2.5 Identificación de los elementos y el DOM.

Las principales estrategias empleadas en el proyecto para la localización de los elementos han sido: Id y Xpath. Siempre y cuando fuese posible se ha optado por el Id debido a la velocidad que presenta el mismo a la hora de ejecutar el script, así como la estabilidad que suele presentar este tipo de atributo. Si esto no era posible, ya que el elemento no tuviese Id, entonces se ha optado por el uso del Xpath para la localización del elemento debido a la alta versatilidad que presenta.

La sintaxis base es la siguiente:

```
WebElement Nombre_Elemento = driver.findElement(By.xpath/Id(xpath_Name/Id_Name));
```

- **`findElement`**: Es un método del driver que se utiliza para buscar y obtener un elemento web en la página actual del navegador. Localiza y devuelve el primer elemento que coincide con el criterio de búsqueda especificado. Si no encuentra ningún elemento que coincida con el criterio de búsqueda, arrojará una excepción NoSuchElementException.

```
List<WebElement> Nombre_Elemento = driver.findElements(By.xpath/Id(xpath_Name/Id_Name));
```

- **`findElements`**: Es un método del driver que se utiliza para buscar y obtener un elemento web en la página actual del navegador. Localiza todos los elementos que coinciden con el criterio de búsqueda especificado. Si no encuentra ningún elemento que coincida con el criterio de búsqueda, Devuelve una lista vacía

De ambas maneras se asocia un elemento o elementos web dentro de la instancia del driver correspondiente a la variable: Nombre_Elemento

Una vez se tiene el elemento web capturado es posible obtener diversas propiedades o atributos del elemento así como realizar acciones sobre el elemento. A continuación se verán algunos usos de esto reflejado en el proyecto:

- **`Nombre_Elemento.getText();`** (Nombre_Elemento debe ser un WebElement)

Devuelve una cadena de caracteres, String, que representa el texto interno visible del elemento Nombre_Elemento. No recuperará el texto oculto por CSS. Es especialmente útil cuando se quiere validar el contenido de texto de un elemento, como un mensaje de error, un título, un párrafo, el texto de un botón, entre otros. Si el elemento no tiene texto, devolverá una cadena vacía. Ejemplo en: [**Tabla 5 - Ejemplo getText()**]

Ejemplo:

Dado el siguiente html:	
`<h1 id="titulo">Bienvenido a mi sitio web</h1>`	
Dado el siguiente código en Selenium:	
`WebElement tituloElemento = driver.findElement(By.id("titulo"));` `String textoTitulo = tituloElemento.getText();`	
variable: textoTitulo	**Valor**: “Bienvenido a mi sitio web”

Tabla 5 - Ejemplo getText()

- **`Nombre_Elemento.clear();`** (Nombre_Elemento debe ser un WebElement)

Se utiliza principalmente con campos de entrada de texto para eliminar cualquier texto presente en esos campos. Es útil cuando es necesario asegurarse de que un campo de entrada esté vacío antes de ingresar un nuevo valor. Por ejemplo, en formularios que podrían tener valores predeterminados. Ejemplo en: [**Tabla 6 - Ejemplo clear()**]

Ejemplo:

Dado el siguiente html:
`<input id="username" type="text" value="usuario_default"/>`
Dado el siguiente código en Selenium:
`WebElement campoUsuario = driver.findElement(By.id("username"));` `campoUsuario.clear();`
Resultado: La variable campoUsuario ya no tendrá el valor “usuario_default”

Tabla 6 - Ejemplo clear()

- **`Nombre_Elemento.sendKeys();`** (Nombre_Elemento debe ser un WebElement)

Envía una secuencia de caracteres, simulando la escritura en el teclado, al elemento especificado. Es usado comúnmente para llenar campos de texto en formularios o para enviar teclas especiales, como ENTER o TAB. Ejemplo en: [**Tabla 7 - Ejemplo sendKeys()**]

Ejemplo:

Dado el siguiente html:
<input id="username" type="text" />
Dado el siguiente código en Selenium:
WebElement campoUsuario = driver.findElement(By.id("username")); campoUsuario.sendKeys("mi_usuario");
Resultado: La variable campoUsuario tendrá reflejada en el GUI: “mi_usuario”

Tabla 7 - Ejemplo sendKeys()

- **`Nombre_Elemento.click();`** (Nombre_Elemento debe ser un WebElement)

Simula la acción de hacer clic en un elemento web. Este método es muy versátil y puede usarse para múltiples acciones, algunas pueden ser: Activar botones, seleccionar casillas de verificación, abrir enlaces, Expandir menús desplegables, etc. Si el elemento no es accesible (por ejemplo, está oculto, fuera del área visible o no interactivo), Selenium arrojará una excepción al intentar hacer clic en él. Ejemplo en: [**Tabla 8 - Ejemplo click()**]

Ejemplo:

Dado el siguiente html:
<button id="miBoton">Haz clic en mí</button>
Dado el siguiente código en Selenium:
WebElement boton = driver.findElement(By.id("miBoton")); boton.click();
Resultado: La variable boton habrá sido clickeada

Tabla 8 - Ejemplo click()

- **`Nombre_Elemento.getAttribute(Argumento);`** (Nombre_Elemento debe ser un WebElement)

Se utiliza para obtener el valor de un atributo específico de un elemento web. Cada elemento en una página HTML puede tener varios atributos que proporcionan información adicional sobre ese elemento.

Algunos de los usos que puede tener son: Obtener el valor actual de un campo de entrada (<input>), Leer la URL de un enlace (<a href="...">), Comprobar si una casilla de verificación está marcada (checked), etc.

Como argumento, la función necesita de el nombre del atributo cuyo valor se quiere recuperar. Ejemplo en: [**Tabla 9 - Ejemplo getAttribute()**]

Ejemplo:

<table>
<tr><td colspan="2">Dado el siguiente html:</td></tr>
<tr><td colspan="2"><code>Visitar sitio</code></td></tr>
<tr><td colspan="2">Dado el siguiente código en Selenium:</td></tr>
<tr><td colspan="2"><code>WebElement enlace = driver.findElement(By.id("miEnlace"));
String url = enlace.getAttribute("href");</code></td></tr>
<tr><td>Variable: url</td><td>Valor: https://www.ejemplo.com</td></tr>
</table>

Tabla 9 - Ejemplo getAttribute()

- **`Nombre_Elemento.isDisplayed();`** (Nombre_Elemento debe ser un WebElement)

Se utiliza para verificar si un elemento web es visible para el usuario en la página. Es una función útil cuando se quiere asegurar que un elemento no sólo está presente en el DOM de la página, sino que también está visible para el usuario.
Esta función devuelve true si el elemento es visible para el usuario y false si el elemento no está visible. No obstante, si el elemento no está presente en el DOM se lanzará una excepción. Ejemplo en: [**Tabla 10 - Ejemplo isDisplayed()**]

Ejemplo:

<table>
<tr><td colspan="2">Dado el siguiente html:</td></tr>
<tr><td colspan="2"><code><div id="mensajeError" style="display: none;">Ha ocurrido un error.</div></code></td></tr>
<tr><td colspan="2">Dado el siguiente código en Selenium:</td></tr>
<tr><td colspan="2"><code>WebElement mensaje = driver.findElement(By.id("mensajeError"));
boolean esVisible = mensaje.isDisplayed();</code></td></tr>
<tr><td>Variable: esVisible</td><td>Valor: false</td></tr>
</table>

Tabla 10 - Ejemplo isDisplayed()

- **`Nombre_Elemento.submit();`** (Nombre_Elemento debe ser un WebElement)

Se utiliza para simular la acción de enviar un formulario web. Es equivalente a hacer clic en un botón de tipo "submit" o presionar la tecla "Enter" dentro de un campo de texto de un formulario.

Es importante tener en consideración que si el elemento web sobre el cual se invoca submit() no forma parte de un formulario, la acción no tendrá ningún efecto y no se producirá ningún error. Ejemplo en: [**Tabla 11 - Ejemplo submit()**]

Ejemplo:

Dado el siguiente html:

```
<form id="loginForm" action="/login" method="post">
   <input type="text" id="username" name="username"/>
   <input type="password" id="password" name="password"/>
   <input type="submit" value="Iniciar sesión"/>
</form>
```

Dado el siguiente código en Selenium:

```
WebElement campoUsuario = driver.findElement(By.id("username"));
campoUsuario.sendKeys("mi_usuario");
WebElement campoContraseña = driver.findElement(By.id("password"));
campoContraseña.sendKeys("mi_contraseña");
campoContraseña.submit();
```

Resultado: El formulario habrá sido enviado

Tabla 11 - Ejemplo submit()

Estos son los ejemplos principales que podemos encontrar en el proyecto. No obstante, es importante mencionar que existen más ejemplos.

4.3.3 Sincronizaciones/Esperas

La automatización es un proceso cuya finalidad es eliminar la intervención humana. Es por ello que el script de automatización debe ser lo más robusto posible. Es aquí donde las esperas se convierten en las protagonistas.

Cuando uno navega en una página web es común hacer esperas, normalmente nímias para un humano, para poder continuar navegando correctamente. Esto se puede ver reflejado, por ejemplo, cuando se hace login en una página y, tras darle al botón de “entrar”, hay que hacer una espera a que la página cargue los datos correctamente. Este comportamiento innato en los

humanos también es importante replicarlo en la automatización. Para ello se crean las esperas.

Estas esperas se refieren al mecanismo mediante el cual el código de automatización se coordina con la velocidad de ejecución de la aplicación web, asegurando que las acciones solo se realicen cuando los elementos web estén listos para ser interactuados o cuando ciertas condiciones se cumplan. La sincronización es crucial en la automatización de pruebas web porque las aplicaciones web modernas a menudo cargan contenido y realizan operaciones de manera asincrónica, lo que puede resultar en intentos de interacción con elementos que aún no están disponibles o listos.

En Selenium hay dos tipos de esperas:

- **Espera implícita**:

Esta espera es una forma de configurar un tiempo de espera predeterminado que el WebDriver deberá usar al intentar encontrar elementos en la página. Durante este período de tiempo, el WebDriver intentará localizar el elemento hasta que lo encuentre o expire el tiempo de espera. Es una forma de decirle a WebDriver que espere un cierto período antes de lanzar una excepción si no puede encontrar el elemento de forma inmediata.

Este tipo de espera se define de manera global. Así, una vez configurado, afecta a todas las operaciones findElement y findElements que se realicen posteriormente.

En el presente proyecto se puede encontrar de la siguiente manera:

```
driverChrome.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
driverFirefox.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
```

No obstante, este tipo de espera tiene ciertas limitaciones. Entre las limitaciones cabe mencionar que no es adecuada para condiciones más complejas: si es necesario esperar que un elemento tenga cierto estado o que se cumpla una condición específica, la espera implícita no es suficiente. Además, afecta a todas las búsquedas de elementos: una vez que se establece, la espera implícita se aplica globalmente, por ello, no será posible elegir qué elementos tienen la espera y cuáles no. Es por ese motivo que el uso que se le ha dado a este tipo de espera en el proyecto no es muy notoria.

- **Espera explícita**:

Esta espera es un mecanismo que permite configurar condiciones específicas para que el WebDriver espere antes de proceder con la ejecución. En lugar de simplemente esperar un tiempo fijo como con la espera implícita, con la espera explícita, se puede especificar ciertas condiciones que deben cumplirse antes de seguir adelante.

Este tipo de espera se define de manera local. Por ese motivo y a diferencia de la espera implícita, solo afecta a la operación específica donde se aplica

En el presente proyecto esto se puede ver reflejado de múltiples formas. No obstante la base siempre es la misma, variando así el driver y la condición concreta que se espera:

```
WebDriverWait waitF = new WebDriverWait(driverFirefox, Duration.ofSeconds(30));
waitF.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(xpathOtherCamera
)));
```

No obstante, este tipo de espera tiene ciertas limitaciones. Entre las limitaciones cabe mencionar que su lógica puede ser más compleja que simplemente configurar una espera implícita ya que presenta una mayor planificación para decidir cuál es la mejor condición para cada situación.

Además de las esperas previamente definidas, también existe una manera de sincronizar elementos de forma manual dentro del código. Este patrón es útil para manejar elementos en páginas web que son dinámicas y cuyos estados pueden cambiar rápidamente:

```
WebElement joinButtonC = null;
while(repeat <= 5){
    try{
        joinButtonC = driverChrome.findElement(By.xpath(XpathJoinButton));
        joinButtonC.click();
        break;
    }catch(StaleElementReferenceException exc){
        exc.printStackTrace();
    }
    repeat++;
}
```

Ilustración 14 - Espera Manual

Esta forma de sincronizar elementos podría tratarse de una forma de espera forzosa. En el caso particular de [**Ilustración 14 - Espera Manual**]: si el botón no se puede interactuar debido a un cambio en el DOM que hace que la referencia al elemento sea obsoleta (`StaleElementReferenceException`), el error se captura, se imprime su traza de pila para fines de depuración y el bucle intenta de nuevo hasta que se alcanza el límite de intentos. Si el clic se realiza con éxito en cualquier momento, el bucle se rompe y se detiene.

4.3.4 Captura de video

La finalidad del presente proyecto es la confirmación del correcto funcionamiento de la web de videoconferencia OpenVidu. Es por ello que, el paso más importante para corroborar esto es confirmar que el video se está fluyendo adecuadamente.

Para poder realizar una captura del video correctamente se ha recurrido a la extracción de un atributo del propio elemento de vídeo que se quiere capturar. De esta manera, se tienen varias

estrategias dependiendo del tipo de aplicación que se esté probando: HelloWorld, Angular, JavaScript, React, y Vue.

Así, se destacan las siguientes formas:

- `driver.findElement(By.id/xpath(elemento)).getAttribute("currentTime")`

este método devolverá un String. Este String se corresponderá al tiempo que, en el momento de ejecutar el método, lleve la cámara reproduciendo vídeo.

El atributo currentTime es un atributo personalizado definido para ese elemento en la página web, no es un atributo estándar de HTML. Es por ello que esta forma de obtener el video sólo es posible para las aplicaciones: HelloWorld, React y Vue

- `driver.findElement(By.id/xpath(elemento)).getAttribute("readyState")`

El atributo readyState es un atributo personalizado definido para ese elemento en la página web, no es un atributo estándar de HTML. Es por ello que esta forma de obtener el video sólo es posible para las aplicaciones: Angular y JavaScript.

este método devolverá un número entre el 0 y el 4. Cada uno de los números representa un estado de vídeo, reflejando así el estado real del vídeo:

- 0, "HAVE_NOTHING": Indica que no se ha inicializado ninguna información sobre el recurso de medios o la red. En este estado, el elemento no ha cargado ningún dato aún, por lo que no hay información sobre la duración del medio, ni los datos de los frames han sido cargados.

- 1, "HAVE_METADATA": se ha cargado la información básica (metadatos) del recurso, como las dimensiones del video o la duración del medio, pero aún no se ha cargado ningún dato del cuadro actual que se podría reproducir.

- 2, "HAVE_CURRENT_DATA": los datos del recurso actual están disponibles, pero no hay suficiente datos cargados para reproducir el siguiente cuadro de medios sin tener que parar para cargar más datos. Puede que haya suficiente información para reproducir un poco del contenido, pero no para continuar una reproducción suave y sin interrupciones.

- 3, "HAVE_FUTURE_DATA": se han cargado suficientes datos para que la reproducción pueda empezar y continuar por un tiempo, es decir, hay datos suficientes que permitirían una reproducción fluida durante un tiempo inmediato, pero no se garantiza que la reproducción pueda completarse hasta el final sin tener que detenerse para cargar más datos.

- 4, "HAVE_ENOUGH_DATA": existen suficientes datos disponibles para una reproducción completa y sin interrupciones hasta el final del recurso de medios. En este estado, el medio puede reproducirse hasta el final sin detenerse, asumiendo que la velocidad de la red se mantenga al menos en el mismo nivel que cuando se alcanzó este estado.

Así, el estado mínimo permitido para verificar una reproducción correcta se ha establecido en el estado 3: "HAVE_FUTURE_DATA". No obstante, es importante recalcar que este tipo de datos podrá ser visualizado en el reporte generado.

4.3.5 Generación de un reporte

La generación de un reporte es un recurso muy necesario a la hora de automatizar pruebas. Para el presente proyecto se ha hecho uso de la herramienta Extent Report [3.1.4 Extent Report].

Para hablar de la implementación, primero se hará hincapié en la clase: ExtentManager.

```
public ExtentReports createExtentReports(String output) {

    String filePath = "test-output/" + output;

    ExtentReports extentReports = new ExtentReports();
    ExtentSparkReporter reporter = new ExtentSparkReporter(filePath);
    reporter.config().setReportName("test Report");
    reporter.config().setTimeStampFormat("EEEE, MMMM dd, yyyy, hh:mm a '('zzz')'");
    reporter.config().setTheme(Theme.DARK);
    extentReports.attachReporter(reporter);
    extentReports.setSystemInfo("Blog Name", "Automation Report");
    extentReports.setSystemInfo("Author", "Andrea P");

    return extentReports;
}
```

Ilustración 15 - createExtentReports

El método `createExtentReports` de [**Ilustración 15 - createExtentReports**] inicializa y configura una instancia de ExtentReports para la generación de informes de pruebas con Extent Report. Especifica la ruta del archivo de salida, personaliza el ExtentSparkReporter con un nombre de informe, un formato de marca de tiempo, y un tema oscuro para el estilo del informe. Además, adjunta el reporte a ExtentReports y agrega información del sistema como el nombre y el autor del informe. Esta configuración preparada es luego retornada para ser utilizada en el script.

```
public ExtentTest startTest(String testName, String desc, ExtentReports e) {
    ExtentTest test = e.createTest(testName, desc);
    return test;
}
```

Ilustración 16 - startTest

El método `startTest` de [**Ilustración 16 - startTest**] inicia un nuevo test utilizando ExtentReports, asignándole un nombre y una descripción. Toma tres parámetros: `testName` para el nombre del test, `desc` para la descripción del test, y `e` es la instancia de ExtentReports utilizada para crear el test. Dentro del método, se llama a `createTest` en la instancia de ExtentReports pasada, la cual devuelve un objeto ExtentTest. Este objeto ExtentTest representa una prueba individual o un nodo en el informe de Extent y es donde se pueden registrar los pasos del test, los resultados y las excepciones. El nuevo test creado se devuelve inmediatamente para su uso en el registro de las acciones y resultados de las pruebas.

```
private String addInfoStepWithPhoto(WebDriver driver, String description){
    TakesScreenshot screenshot = (TakesScreenshot) driver;
    String base64Screenshot = screenshot.getScreenshotAs(OutputType.BASE64);
    String base64 = "data:image/png;base64," + base64Screenshot;

    return "<div><img src=\"" + base64 + "\"/></div><div>" + description + "</div>";
}
```

Ilustración 17 - addInfoStepWithPhoto

El método `addInfoStepWithPhoto` de [**Ilustración 17 - addInfoStepWithPhoto**] es una función privada auxiliar que toma una captura de pantalla de la página actual en el navegador a través del WebDriver proporcionado y la devuelve como una cadena de texto en formato HTML que incorpora la imagen codificada en base64 junto con una descripción. La función comienza por realizar un casting del WebDriver a `TakesScreenshot` para utilizar su capacidad de tomar capturas de pantalla. Luego, obtiene la captura de pantalla en formato base64. Concatena esta información en una etiqueta img de HTML con el atributo src configurado para mostrar la imagen en base64, y añade la descripción proporcionada debajo de la imagen y lo devuelve.

```
public void addStep(ExtentTest test, String status, WebDriver driver, String description){

    if (status.equals(anObject: "INFO")){
        test.log(Status.INFO, addInfoStepWithPhoto(driver, description));
    }else if (status.equals(anObject: "PASS")){
        test.log(Status.PASS, addInfoStepWithPhoto(driver, description));
    }else if(status.equals(anObject: "FAIL")){
        test.log(Status.FAIL, addInfoStepWithPhoto(driver, description));

    }else{
        test.log(Status.FAIL, "Status in code is not ok");
        fail("The app is not correctly inicializate");
    }

}
```

Ilustración 18 - addStep

El método `addStep` de [**Ilustración 18 - addStep**] agrega un registro de un paso de prueba a un objeto ExtentTest, que es un componente de ExtentReports utilizado para la generación de informes de pruebas. El método toma un objeto ExtentTest, una cadena de texto status que representa el resultado del paso de la prueba, un WebDriver para tomar capturas de pantalla, y una descripción del paso de prueba.

La lógica dentro del método verifica el status del paso y realiza lo siguiente:

- Si status es "INFO", registra un paso informativo
- Si status es "PASS", registra un paso exitoso
- Si status es "FAIL", registra un paso fallido

Si el status proporcionado no es ninguno de los anteriores, registra un fallo en el informe indicando que el "Status in code is not ok" y lanza una excepción con el mensaje "The app is not correctly inicializate".

En cada uno de estos casos, el método `addInfoStepWithPhoto` es utilizado para obtener una representación HTML de la captura de pantalla y la descripción para ser incluida en el registro del informe. Esto enriquece el informe con evidencia visual para cada paso, lo cual es útil para el análisis detallado de los resultados de las pruebas.

```
public void addStepWithoutCapture(ExtentTest test, String status, String description){

    if (status.equals(anObject: "INFO")){
        test.log(Status.INFO, description);
    }else if (status.equals(anObject: "PASS")){
        test.log(Status.PASS, description);
    }else if(status.equals(anObject: "FAIL")){
        test.log(Status.FAIL, description);

    }else{
        test.log(Status.FAIL, "Status in code is not ok");
        fail("The app is not correctly inicializate");
    }

}
```

Ilustración 19 - addStepWithoutCapture

El método `addStepWithoutCapture` de [**Ilustración 19 - addStepWithoutCapture**] registra el resultado de un paso de prueba en un objeto ExtentTest sin adjuntar una captura de pantalla, a diferencia del método previo: `addStep`. Este método acepta tres argumentos: un objeto ExtentTest para registrar el paso, una cadena status que define el resultado del paso de la prueba, y una description que describe el paso de la prueba.

```
public void tearDownExtent(ExtentReports extentReports){
    extentReports.flush();
}
```

Ilustración 20 - tearDownExtent

El método `tearDownExtent` de [**Ilustración 20 - tearDownExtent**] se utiliza para finalizar y guardar todas las acciones que han sido registradas en la instancia de ExtentReports pasada como argumento. El método flush que se llama sobre extentReports es crítico porque asegura que todos los datos de las pruebas que han sido recopilados y registrados en la instancia de ExtentReports se escriban a su destino final.

Llamar a flush es un paso necesario para garantizar que el informe refleje todos los resultados de las pruebas. Si no se llama a este método al finalizar las pruebas, existe la posibilidad de que el informe final no esté completo o no se genere.

Una vez vista la lógica tras los diversos métodos de configuración de Extent Report, es importante saber cuándo usar cada uno de estos en los scripts de pruebas automatizadas.

- La creación del reporte (`createExtentReports`): debe ser creada una única vez al principio de cada script de prueba. Si se crea múltiples veces bajo la misma ruta ocasionará que se sobreescriba el mismo archivo, generando un informe incompleto.

- La creación de una prueba dentro del reporte (`startTest`): debe ser creada tantas veces como pruebas existan en la clase de prueba. Usualmente se realiza al principio de cada test.

- La creación de un paso dentro de una prueba (`addStep` , `addStepWithoutCapture`): debe ser utilizado cada vez que se quiera reportar la información de un paso del test.

- La finalización del reporte (`tearDownExtent`): debe ser utilizado una única vez tras la finalización de todas las pruebas que se quieran reportar. Usualmente se realizan en una etiqueta bajo la cual se garantiza una única ejecución tras todo el script. Un ejemplo podría ser: `@AfterAll`.

4.3.6 Parametrización de datos

La naturaleza del Data-driven Framework defiende la separación de datos con los pasos de ejecución. Por eso se ha encapsulado el método `readVariablesFromExcel` en la clase padre de los scripts y estos datos se cargan previamente a la ejecución de cada uno de los tests [**Ilustración 7 - @BeforeEach**].

Dicho método presenta la siguiente cabecera [**Ilustración 21 - Cabecera readVariablesFromExcel**]:

```
protected static String readVariablesFromExcel(String filePath, String testName, String ColValue)
```

Ilustración 21 - Cabecera readVariablesFromExcel

Se trata de una función protegida estática que, especificada la ruta del archivo, el nombre del test y la columna que se desea, devuelve un String. Dicho string será el valor dentro del excel que haga intersección entre la fila, delimitada por `testName`, y la columna, delimitada por `colValue`.

A continuación se dividirá el flujo del código y se irán explicando respectivamente.

```java
try (FileInputStream fileInputStream = new FileInputStream(filePath);
Workbook workbook = WorkbookFactory.create(fileInputStream)) {
org.apache.poi.ss.usermodel.Sheet sheet = workbook.getSheetAt(0);

int testNameColumnIndex = 0;
int colValueColumnIndex = -1;

Row firstRow = sheet.getRow(0);
int lastCellNum = firstRow.getLastCellNum();
```

Ilustración 22 - Inicio readVariablesFromExcel

El código de la imagen [**Ilustración 22 - Inicio readVariablesFromExcel**] utiliza un bloque try-catch para abrir un archivo Excel especificado por filePath. Una vez abierto, crea un objeto Workbook para representar el archivo y accede a su primera hoja, donde están almacenados los datos. Inicializa dos variables para representar los índices de las columnas que buscará más adelante. Finalmente, obtiene la primera fila del Excel, donde se buscará el `testName` y determina cuántas celdas contiene.

```java
for (int i = 0; i < lastCellNum; i++) {
    org.apache.poi.ss.usermodel.Cell cell = firstRow.getCell(i);
    if (cell.getStringCellValue().equals(colValue)) {
        colValueColumnIndex = i;
        break;
    }
}

if (colValueColumnIndex == -1) {
    System.out.println("No se encontró la columna: " + colValue + " en el archivo Excel.");
    return null;
}
```

Ilustración 23 - Medio readVariablesFromExcel

Posteriormente, en [**Ilustración 23 - Medio readVariablesFromExcel**] se itera sobre cada celda de la primera fila del archivo Excel hasta el último número de celda determinado previamente. Para cada celda, se verifica si su valor coincide con el valor `colValue`. Si encuentra una coincidencia, almacena el índice de esa celda en `colValueColumnIndex` y sale del bucle.

Si, después de revisar todas las celdas, no se encuentra ninguna coincidencia (es decir, `colValueColumnIndex` sigue siendo -1), se mostrará un mensaje indicando que no se encontró la columna con el valor especificado en `colValue` y devuelve null.

```java
int lastRowIndex = sheet.getLastRowNum();

for (int i = 1; i <= lastRowIndex; i++) {
    Row row = sheet.getRow(i);
    org.apache.poi.ss.usermodel.Cell testNameCell = row.getCell(testNameColumnIndex);

    if (testNameCell != null && testNameCell.getStringCellValue().equals(testName)) {
        org.apache.poi.ss.usermodel.Cell valueCell = row.getCell(colValueColumnIndex);

        if (valueCell != null) {
            return valueCell.getStringCellValue();
        } else {
            return null;
        }
    }
}

System.out.println("No se encontró la fila con el valor de testName: " + testName + " en el archivo Excel.");
```

Ilustración 24 - Fin readVariablesFromExcel

Posteriormente el código [**Ilustración 24 - Fin readVariablesFromExcel**] determina el índice de la última fila del archivo Excel. Después, itera sobre cada fila del documento, empezando desde la segunda, debido a que la primera contiene encabezados. En cada iteración, recupera la celda que se corresponde con el índice `testNameColumnIndex` y comprueba si su contenido coincide con `testName`. Si encuentra una coincidencia, busca en esa misma fila otra celda ubicada en el índice `colValueColumnIndex`. Si esta última celda existe, devuelve su contenido; de lo contrario, retorna null. Si, después de revisar todas las filas, no encuentra una celda con el valor `testName`, el programa imprime un mensaje indicando que no se halló una fila con ese valor específico en el archivo Excel.

4.3.7 Ejecución en Github Actions

El presente proyecto se ejecuta en una plataforma de integración continua y entrega continua, Github Actions.

Se ha creado un archivo .yml donde se describe el flujo de trabajo de GitHub Actions para la ejecución automática de pruebas en diferentes aplicaciones las cuales usan la plataforma OpenVidu para videoconferencias.

A continuación se irá desglosando el archivo `Deploy_OpenVidu.yml` paso por paso:

- **Trigger:**

```
on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
```

Ilustración 25 - Trigger Deploy_OpenVidu.yml

El flujo de trabajo se activará en dos situaciones, [**Ilustración 25 - Trigger Deploy_OpenVidu.yml**]:

- Cuando se realice un push a la rama master.
- Cuando se cree o actualice un pull request dirigido a la rama master.

- **Jobs**:

El flujo de trabajo consiste en varios trabajos que se ejecutan en paralelo. Cada trabajo representa pruebas para un tipo diferente de aplicación: HelloWorld, Angular, JavaScript, React, y Vue.

- **Pasos Comunes**:

Todos los trabajos tienen los siguientes pasos en común [**Ilustración 26 - Pasos Comunes Deploy_OpenVidu.yml**], los cuales corresponden al despliegue de la aplicación a probar:

- Realizar un checkout del código del repositorio.
- Iniciar un contenedor Docker de OpenVidu en modo desarrollo.
- Configurar JDK 11.
- Configurar Node.js versión 16.
- Ejecutar el backend de Java de OpenVidu usando Maven.

```
javaScriptTest:
  runs-on: ubuntu-latest
  steps:
  - uses: actions/checkout@v3
  - name: Execute OpenVidu Dev container
    run: docker run --detach=true -p 4443:4443 --rm -e OPENVIDU_SECRET=MY_SECRET openvidu/openvidu-dev:2.24.0
  - name: Set up JDK 11
    uses: actions/setup-java@v3
    with:
      java-version: '11'
      distribution: 'adopt'
  - name: Set up Node 16
    uses: actions/setup-node@v3
    with:
      node-version: '16'
  - name: Execute OpenVidu Tutorial Java Backend
    run: mvn --no-transfer-progress --batch-mode spring-boot:run &
    working-directory: ./openvidu-basic-java
```

Ilustración 26 - Pasos Comunes Deploy_OpenVidu.yml

- **Paso Específico - Run the client**:

Después de los pasos comunes, cada trabajo tiene un paso específico, La ejecución del cliente, relacionados con el tipo de aplicación que se está probando:

- HelloWorld: Sirve una página web estática y ejecuta pruebas [**Ilustración 27 -Run the client HelloWorld**].

```
name: Execute http-server to serve static web
run: |
  npm install --location=global http-server
  http-server openvidu-hello-world/web &
working-directory: .
```

Ilustración 27 -Run the client HelloWorld

- Angular: Ejecuta una aplicación Angular en modo de desarrollo y ejecuta pruebas [**Ilustración 28 -Run the client Angular**].

```
name: Execute Angular app in dev mode (ng-serve)
run: |
  npm install
  npx ng serve &
working-directory: ./openvidu-angular
```

Ilustración 28 -Run the client Angular

- JavaScript: Sirve una página web estática y ejecuta pruebas [**Ilustración 29 -Run the client JavaScript**].

```
name: Execute http-server to serve static web
run: |
  npm install --location=global http-server
  http-server openvidu-js/web &
working-directory: .
```

Ilustración 29 -Run the client JavaScript

- React: Ejecuta una aplicación React en modo de desarrollo [**Ilustración 30 -Run the client React**].

```
name: Execute React app in dev mode
run: |
  npm install
  npm start &
working-directory: ./openvidu-react
```

Ilustración 30 -Run the client React

- Vue: Ejecuta una aplicación Vue en modo de desarrollo [**Ilustración 31 -Run the client Vue**].

```
name: Execute Vue app in dev mode
run: |
  npm install --legacy-peer-deps
  npm run serve &
working-directory: ./openvidu-vue
```

Ilustración 31 -Run the client Vue

Cabe mencionar la importancia de no bloquear la terminal con ningún comando. Algunos comandos como "npm start" dejan la terminal bloqueada y no terminan hasta que se muere el comando, lo que ocasiona un error del flujo de ejecución al no poder continuar. Para evitar errores de este estilo, es importante ejecutar estos comandos en segundo plano añadiendo "&" al final del comando. De esta manera el comando no dejará bloqueada la terminal.

- **Ejecución de Pruebas**:

En cada job, después de configurar y ejecutar la aplicación, hay un paso para ejecutar pruebas específicas utilizando Maven: [**Ilustración 32 - Ejecución test Deploy_OpenVidu.yml**]. Estos pasos son iguales en cada job, sólo se diferencian en el nombre concreto del test que se va a ejecutar. En el siguiente ejemplo se trata del HelloWorld:

```
name: Execute tests
run: mvn test --no-transfer-progress --batch-mode -Dtest="OpenViduHelloWorldTest"
working-directory: ./Tests/OpenviduTests
```

Ilustración 32 - Ejecución test Deploy_OpenVidu.yml

- **Subida de Informes**:

Luego de la ejecución de las pruebas, se sube el informe de resultados de pruebas como un artefacto en GitHub Actions: [**Ilustración 33 - Reporte Deploy_OpenVidu.yml**]. Este paso se ejecuta siempre (`if: always()`), independientemente de si los pasos anteriores fueron exitosos o fallidos. Así es posible diagnosticar problemas cuando las pruebas fallan.
Este paso es común a todos los jobs, cambiando el informe html concreto a descargar.

```yaml
name: Upload Extent report
uses: actions/upload-artifact@v3
with:
  name: Extent report
  path: ./Tests/OpenviduTests/test-output/OpenViduHelloWordTestTestReport.html
if: always()
```

Ilustración 33 - Reporte Deploy_OpenVidu.yml

4.4 Refinamiento

Dentro del presente apartado se hará referencia a los errores más importantes encontrados a la hora de desarrollar el proyecto y cómo se han solucionado los mismos.

4.4.1 Creación del reporte

La evolución del reporte fue lenta pero constante. La primera versión de evidencias se trataba de un método que realizaba una captura de pantalla y la guardaba en una carpeta donde se iban almacenando todas las capturas:

```java
/**
* method.
*
* @author Andrea Acuña
* Description: take a screenshot to create an evidence.
* Parameters:
*          - url1: the relative or absolute path to a evidence file of the chrome photo
*          - url2: the relative or absolute path to a evidence file of the firefox photo
*/
public void takePhoto(String url1, String url2, WebDriver c, WebDriver f) throws IOException{
    try {
        if(url1 != ""){
            File scrFileC = ((TakesScreenshot)c).getScreenshotAs(OutputType.FILE);
            FileUtils.copyFile(scrFileC, new File(url1));
        }
        if(url2 != ""){
            File scrFileF = ((TakesScreenshot)f).getScreenshotAs(OutputType.FILE);
            FileUtils.copyFile(scrFileF, new File(url2));
        }
    } catch (Exception e) {
        System.out.println("an error has occurred with the screenshot. Please preview the url");
    }
}
```

Ilustración 34 - Primera versión reporte

Posterior a esta primera versión: [**Ilustración 34 - Primera versión reporte**], se introdujo la versión del reporte sin capturas de pantalla con Extent Report, donde se mostraba exclusivamente el paso con información que se establecía manualmente. No obstante, al intentar crear esta versión, aparecieron problemas a la hora de la creación del propio reporte ExtentReports.

```
/**
 * BeforeEach.
 *
 * @author Andrea Acuña
 * Description: Execute before every single test. Configure the camera an set de url in each browser
 */
    @BeforeEach
    void setup() {
        List<WebDriver> browsers = super.setUpTwoBrowsers();
        driverChrome = browsers.get(0);
        driverFirefox = browsers.get(1);
        driverChrome.get(URL);
        driverFirefox.get(URL);

        extentReports = super.createExtentReports();
        TESTNAME = Thread.currentThread().getStackTrace()[2].getMethodName();
        test = super.startTest(TESTNAME, "");
    }
```

Ilustración 35 - Segunda versión reporte

El problema de la imagen [**Ilustración 35 - Segunda versión reporte**] en las líneas marcadas se trata de un problema de lógica. La creación del reporte no debe estar en un @BeforeEach. Esto ocasionará que el reporte se cree por cada prueba del script, por lo que este se reescribirá cada vez que se ejecute una prueba.

```
/**
 * BeforeTest.
 *
 * @author Andrea Acuña
 * Description: Execute before the grouf of tests. Configure the reporter
 */
    @BeforeTest
    void setupReporter() {
        extentReports = super.createExtentReports();
    }
```

Ilustración 36 - Tercera versión reporte

El problema de la imagen [**Ilustración 36 - Tercera versión reporte**] viene en la mezcla de dos frameworks que pueden interferir entre ellos: JUnit y TestNG. Esto se ve en el uso de las anotaciones, ya que `@BeforeTest` es una clara anotación de TestNG, mientras que otras anotaciones del test son de JUnit. Además, cabe destacar que también hubo problemas con la repetición de la línea de creación del reporte, lo que claramente es una mala práctica.

```
public OpenViduVueTest() {
    if (extentReports == null){
        extentReports = e.createExtentReports(reportLocation);
    }
}
```

Ilustración 37 - Última versión reporte

Finalmente en la imagen [**Ilustración 37 - Última versión reporte**] se puede ver la opción final para la creación del reporte. Cabe destacar la importancia de la comprobación a null, pues si esta comprobación no se hace, el reporte se reescribirá cada vez que el una prueba del test sea ejecutada, dejando así una única prueba, la última ejecutada, dentro del reporte.
Para la correcta visualización del reporte en Github Actions deberán seguirse los siguientes pasos:

El primer paso será dirigirse hacia el apartado Artifacts, situado dentro de una action, como se puede observar en la imagen [**Ilustración 38 - Artifacts Github**]

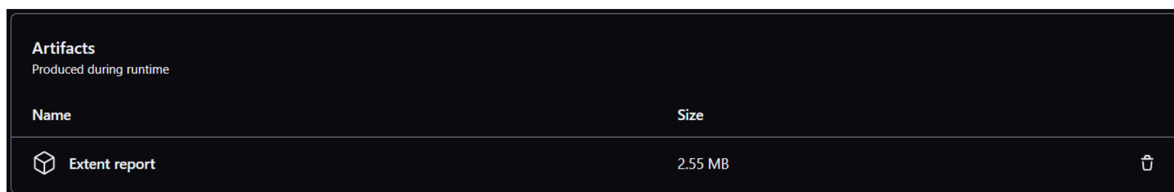

Ilustración 38 - Artifacts Github

posteriormente se procede a la descarga del archivo Extent Report y se descargará como un archivo comprimido: [**Ilustración 39 - Descarga zip reporte**]

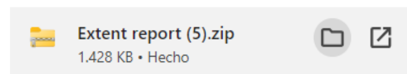

Ilustración 39 - Descarga zip reporte

Tras la descarga del archivo comprimido se procederá a descomprimirlo en la localización de preferencia. El resultado será la descarga de los siguientes archivos HTML: [**Ilustración 40 - Reportes descargados HTML**]

OpenViduAngularTestReport	20/01/2024 19:01	Archivo HTML	763 KB
OpenViduHelloWordTestTestReport	20/01/2024 19:01	Archivo HTML	332 KB
OpenViduJsTestTestReport	20/01/2024 19:01	Archivo HTML	812 KB
OpenViduReactTestTestReport	20/01/2024 19:01	Archivo HTML	142 KB
OpenViduVueTestTestReport	20/01/2024 19:01	Archivo HTML	561 KB

Ilustración 40 - Reportes descargados HTML

Finalmente se abrirá el archivo que se quiera visualizar con el navegador de preferencia, viéndose así un reporte con la siguiente apariencia: [**Ilustración 41 - Reporte abierto**]

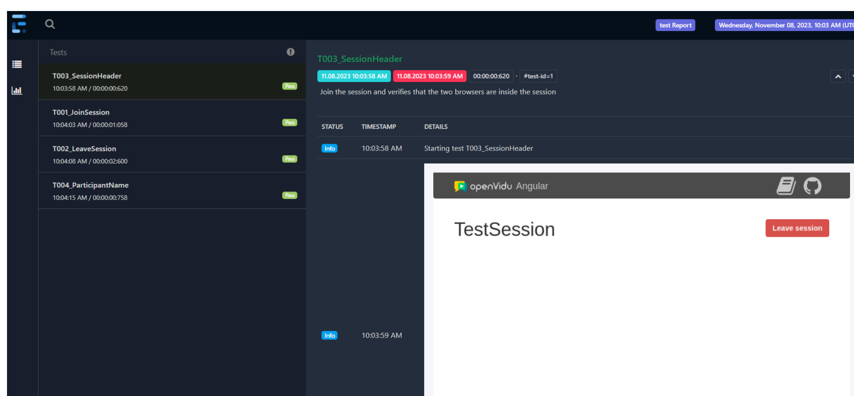

Ilustración 41 - Reporte abierto

4.4.2 Conflicto de versiones entre el driver y chrome

El presente proyecto se ha ejecutado múltiples veces en la máquina local antes de pasar a su ejecución en Github Actions. No obstante, a cada actualización que presentara algunos de los navegadores usados, el driver podía fallar por un conflicto de versiones.

Es por eso que, para evitar los múltiples conflictos de ese estilo que se presentaron, se decidió implementar una dependencia de bonigarcia para la gestión del webDriver. Así, con una única línea [**Ilustración 36 - Configuración chrome bonigarcia**] la prueba se volvía más robusta y solo sería necesario mantener la dependencia de bonigarcia actualizada en caso de ser necesario.

```
WebDriverManager.chromedriver().clearDriverCache().setup();
```

Ilustración 42 - Configuración chrome bonigarcia

La línea [**Ilustración 42 - Configuración chrome bonigarcia**] funciona de la siguiente manera:

- `WebDriverManager.chromedriver()` - Este método indica la configuración del WebDriver para Chrome.

- `.clearDriverCache()` - Este método elimina cualquier versión previamente descargada del ChromeDriver. Es útil si es necesario forzar la descarga de la última versión del WebDriver o si la versión cacheada está causando problemas.

- `.setup()` - Este método realiza las siguientes acciones:

    - Comprueba la versión del navegador instalado en la máquina.

    - Descarga la versión correspondiente del WebDriver si no está presente en el caché (o si el caché fue borrado por `clearDriverCache()`).

- Configura la propiedad del sistema adecuada (webdriver.chrome.driver para Chrome) con la ruta al WebDriver descargado. Esto permite que Selenium utilice este WebDriver sin necesidad de establecer manualmente la ruta al binario.

En conclusión, es una herramienta que simplifica la gestión del binario del WebDriver para diferentes navegadores como Chrome, Firefox, Edge, entre otros. Es decir, gestiona automáticamente la descarga del binario del WebDriver correcto para la versión del navegador que tienes instalada en tu máquina y lo configura para que pueda ser utilizado por Selenium.

4.4.3 Captura de video. Falso positivo

La versión previa a la definitiva a lo que captura de video se refiere, se trataba de un falso positivo. Si se ve la ejecución de Github Actions en dicha versión, se puede ver que el test pasa correctamente. No obstante, esto se trata de un falso positivo, una prueba que se marca como completada satisfactoriamente cuando realmente debería fallar.

```
// see if the video is playing properly
String currentTimeChrome= driverChrome.findElement(By.id(idSelfCamera)).getAttribute("duration");
String currentTimeFirefox = driverFirefox.findElement(By.id(idSelfCamera)).getAttribute("duration");

assertNotEquals(currentTimeChrome, "NaN");
assertNotEquals(currentTimeFirefox, "NaN");

e.addStep(test, "INFO", driverChrome, "Session configurated in Chrome with session name: " + NAMESESSION);
e.addStep(test, "INFO", driverFirefox, "Session configurated in Firefox with session name: " + NAMESESSION);
```

Ilustración 43 - Falso positivo en captura de video

No obstante, el principal problema viene de las líneas previas a las marcadas en la imagen: [**Ilustración 43 - Falso positivo en captura de video**]. La obtención del atributo "duration" devolvía null. El motivo detrás de esto era que dicho atributo no existe para los tests en los que se definía. Así, al ser el valor devuelto diferente de "NaN", este pasaba satisfactoriamente, sin hacer una comprobación real del video.

4.4.4 Mezcla de espera implícita y explícita

En el presente proyecto se habían mezclado este dos tipos de esperas. Esta práctica es polémica ya que el tiempo exacto de espera puede verse alterado al mezclar dos tipos de espera.

```java
driverChrome.manage().timeouts().implicitlyWait(Duration.ofSeconds(seconds: 10));
driverFirefox.manage().timeouts().implicitlyWait(Duration.ofSeconds(seconds: 10));

// see if the video is playing properly, moreover synchronize both videos
WebDriverWait waitC = new WebDriverWait(driverChrome, Duration.ofSeconds(seconds: 30));
waitC.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(xpathOtherCamera)));

WebDriverWait waitF = new WebDriverWait(driverFirefox, Duration.ofSeconds(seconds: 30));
waitF.until(ExpectedConditions.visibilityOfElementLocated(By.xpath(xpathOtherCamera)));
```

Ilustración 44 - Mezcla de esperas

En la imagen [**Ilustración 44 - Mezcla de esperas**] primeramente se observa la configuración de Espera Implícita: Tanto para `driverChrome` como para `driverFirefox`, se ha configurado una espera implícita de 10 segundos. Esto significa que, cuando se intente buscar un elemento con estos drivers, intentarán encontrar el elemento repetidamente durante hasta 10 segundos antes de lanzar una excepción de "elemento no encontrado". Posteriormente se hace uso de la Espera Explícita: se usa WebDriverWait para configurar una espera explícita de 30 segundos para cada driver, esperando que un elemento se vuelva visible en la página.
Esto da como resultado un comportamiento combinado: cuando se llama a waitC.until(...) o waitF.until(...), la espera explícita verifica la condición proporcionada (en este caso, la visibilidad del elemento) repetidamente hasta que se cumple o se agota el tiempo de espera, en este caso de 30 segundos.

Sin embargo, aquí es donde la interacción entre las esperas implícitas y explícitas se vuelve problemática: cuando la espera explícita verifica su condición y llama internamente a findElement(...), esta llamada interna está sujeta a la espera implícita. Por lo tanto, cada intento de verificación por parte de la espera explícita podría tomar hasta 10 segundos antes de decidir si la condición se cumple o no. Esta interacción puede llevar a tiempos de espera mucho más largos de lo esperado. Así, en el peor escenario, si el elemento nunca se vuelve visible, la espera total podría ser significativamente mayor al tiempo especificado para la espera explícita (debido a los múltiples intentos sujetos a la espera implícita).

Los problemas potenciales que está práctica puede traer son:

- Tiempos de Espera Extendidos: El tiempo de espera puede ser mucho mayor al esperado.

- Impredecibilidad: La interacción entre esperas puede llevar a comportamientos inconsistentes dependiendo del estado y la velocidad de carga de la aplicación web.

- Dificultad de Depuración: Si las pruebas fallan o tardan demasiado, puede ser difícil determinar la causa, ya que las interacciones entre las esperas complican la lógica.

Generalmente, es una buena práctica evitar mezclar esperas implícitas y explícitas en el mismo flujo de prueba. En lugar de eso, es preferible elegir uno de los enfoques según las necesidades de tus pruebas.

Capítulo 5: Conclusiones y trabajo futuro

5.1 Conclusiones

5.1.1 Cumplimiento de objetivos

Tras la finalización del proyecto: PRUEBAS AUTOMÁTICAS DE APLICACIONES DE VIDEOCONFERENCIA CON SELENIUM EN GITHUB ACTIONS, se ha concluido que se han cumplidos los objetivos de manera satisfactoria.

A lo largo de la realización del proyecto se han podido adquirir múltiples conocimientos relacionados con la automatización de pruebas así como de la realización de reportes e integración continua. Es por ello que se han visto satisfechos los objetivos planteados al principio del proyecto.

5.1.2 Conclusiones personales

La elaboración del presente trabajo me ha permitido seguir formándome en el ámbito de aseguramiento de la calidad, concretamente en la automatización de pruebas.

Actualmente me encuentro en el puesto de QA Automation, automatizando principalmente la parte GUI y API de aplicaciones web en el sector bancario. Es por ese mismo motivo que no había tenido un contacto previo con la automatización de aplicaciones de videoconferencia. Anterior al desarrollo del presente trabajo no tenía nociones sobre el control de las videoconferencias a nivel de automatización y gracias al desarrollo del trabajo de fin de grado he podido adquirir conocimientos nuevos y conocer más aplicaciones de la automatización y ver qué tan lejos puede llegar y qué tan potente puede ser la automatización de pruebas.

Además, previo a la realización del proyecto no tenía nociones sobre la herramienta Github Actions ni la generación de reportes partiendo de cero, siendo este el motivo por el cual he podido ampliar mi visión acerca tanto del CI/CD como de la creación de reportes en pruebas automáticas.

Por todas estas razones, gracias al desarrollo del proyecto: PRUEBAS AUTOMÁTICAS DE APLICACIONES DE VIDEOCONFERENCIA CON SELENIUM EN GITHUB ACTIONS, he podido aprender y me siento satisfecha con el resultado final.

5.2 Trabajo futuro

Como futuro trabajo sería sumamente práctico ampliar la automatización de pruebas a los diversos tutoriales que no se han visto involucrados en el presente proyecto. Además, sería interesante alcanzar un mejor y más alto nivel y automatizar tutoriales de dispositivos móviles, haciendo uso de herramientas como Appium [19].

Por otro lado, podría llevarse la automatización a otro nivel y usar otros sistemas operativos distintos a Linux, como podría ser Windows o Mac. De esta manera, se consigue ampliar la cobertura de las pruebas tras probar el comportamiento en más sistemas operativos.

Capítulo 6: Bibliografía

[1] SQA, S. A. (2022, Abril 13). Automatización de Pruebas. LinkedIn.
https://www.linkedin.com/pulse/automatizaci%C3%B3n-de-pruebas-sqa-s-a/?originalSubdomain=es

[2] Selenium. (n.d.). Selenium Documentation.
https://www.selenium.dev/documentation/

[3] Atlassian. (n.d.). Automated Testing.
https://www.atlassian.com/continuous-delivery/software-testing/automated-testing

[4] Microsoft Azure. (n.d.). What is Java Programming Language?
https://azure.microsoft.com/en-ca/resources/cloud-computing-dictionary/what-is-java-programming-language#:~:text=Learn%20more-,How%20does%20Java%20work%3F,runtime%20environment%20for%20Java%20apps.

[5] Naidu, R. (2021, Julio 14). Advantages of Using Selenium WebDriver with Java. LinkedIn.
https://www.linkedin.com/pulse/advantages-using-selenium-webdriver-java-ritesh-naidu/

[6] BrowserStack. (n.d.). Selenium Testing.
https://www.browserstack.com/selenium#what-is-selenium

[7] GitHub. (n.d.). GitHub Actions Documentation.
https://docs.github.com/es/actions

[8] Jenkins. (n.d.). Jenkins Documentation.
https://www.jenkins.io/doc/

[9] Travis CI. (n.d.). Travis CI Documentation.
https://docs.travis-ci.com/

[10] CircleCI. (n.d.). CircleCI Documentation.
https://circleci.com/docs/

[11] Extent Reports. (n.d.). Extent Reports Documentation for Java (Version 4).
https://www.extentreports.com/docs/versions/4/java/

[12] Santos, J. P. R. dos, da Silva, K. P., Gonçalves, B. P. G., Pinheiro, J. S. de S., Oliveira, J. M. L. de, & Alencar, D. B. de. (2020). Selenium as a Free Tool to Test for Java Web Application. International Journal of Advanced Engineering Research and Science, 7(4). Retrieved from https://journal-repository.theshillonga.com/index.php/ijaers/article/view/1835

[13] Zhu, H. N., Guan, K. Z., Furth, R. M., & Rubio-González, C. (2023). ActionsRemaker: Reproducing GitHub Actions. ICSE-Companion. IEEE. Retrieved from http://cdn.zhuhaonan.com/files/icse-23-actionsremaker.pdf

[14] Quadri, M. N. I. S. (2020). Framework for Automation of Software Testing for Web Application in Cloud-(FASTEST) Web Application. Solid State Technology, 63(5). Retrieved from https://www.researchgate.net/profile/Md-Islam-1446/publication/356823221_Framework_for_Automation_of_Software_Testing_for_Web_Application_in_Cloud-_FASTEST_Web_Application/links/61af0100b3c26a1e5d8e717d/Framework-for-Automation-of-Software-Testing-for-Web-Application-in-Cloud-FASTEST-Web-Application.pdf

[15] Kumar, A., & Saxena, S. (2015). Data driven testing framework using selenium WebDriver. International Journal of Computer Applications, 118(18). Retrieved from https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=65e283a565b98fc8d3e6aa4069f13ea32dda9a03

[16] BrowserStack. (n.d.). Best Test Automation Frameworks. https://www.browserstack.com/guide/best-test-automation-frameworks

[17] Micro Focus. (2023). User Guide for UFT: Introduction. https://admhelp.microfocus.com/uft/en/23.4/UFT_Help/Content/User_Guide/Ch_UFT_Intro.htm

[18] Cucumber. (n.d.). Cucumber Documentation. https://cucumber.io/docs/cucumber/

[19] Appium. (n.d.). Appium Documentation. https://appium.io/docs/en/2.4/

[20] Cypress. (n.d.). Why Cypress? Cypress Documentation. https://docs.cypress.io/guides/overview/why-cypress

[21] OpenVidu. (2022). OpenVidu: Plataforma de videoconferencia y transmisión en vivo. https://openvidu.io/

[22] Oracle. (n.d.). Documentación de Java. https://docs.oracle.com/en/java/