Universidad Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA EN SISTEMAS AUDIOVISUALES Y MULTIMEDIA

TRABAJO DE FIN DE GRADO

PRUEBAS AUTOMÁTICAS EN UNA PLATAFORMA EDUCATIVA WEB DE ALTA DISPONIBILIDAD

Autor: Felicidad Abad Quintanilla
Tutores: Dr. Jose María Cañas Plaza y David Valladares

Curso académico: 2022 / 2023

Agradecimientos

Me gustaría empezar agradeciendo a todos aquellos que me han ayudado a llegar hasta aquí.

En primer lugar a mi familia. Mamá, gracias por ser tan luchadora y haber hecho lo imposible por salir adelante y darnos todas las oportunidades que hemos tenido. A mi hermano, Alejandro, aunque no siempre te apetezca hablar se que me apoyas incondicionalmente.

Quiero agradecer también a los que ya no están. Abuelos, papá, gracias por ayudarme a ser la persona que soy hoy en día. Espero que estéis orgullosos.

No puedo olvidarme de agradecer a mis tutores; Jose María y David. Muchas gracias por vuestra infinita paciencia conmigo, por todos los conocimientos que me habéis ayudado a obtener y por la oportunidad de colaborar en un proyecto con tanto futuro.

Por último, agradecer también a mis amigos por estar durante todo este largo proceso.

Resumen

La velocidad de avance de la tecnología y el ritmo constante e imparable con el que se requiere realizar modificaciones o nuevos desarrollos, han hecho que los desarrolladores cada vez dispongan de menos tiempo para dedicar a tareas repetitivas y las pruebas necesarias para comprobar el funcionamiento de sistemas software antes de desplegarlos en entornos productivos.

Por este motivo nacen y se popularizan los sistemas de CI/CD. Estos sistemas permiten la automatización de tareas repetitivas y pruebas de casos, lo que a su vez hace que la velocidad a la que se desarrolla software sea mayor y elimina posibles fallos humanos. Utilizar algún sistema de CI/CD en un proyecto software ofrece agilidad, robustez y capacidad de reacción, pues los errores pueden detectarse antes de que lleguen a suceder en producción, teniendo tiempo para solucionarlos.

Este TFG se centra en añadir distintos tipos de pruebas automatizadas a la plataforma de robótica educativa Unibotics. Se ha creado la configuración e insfraestructura necesaria para que un evento, en este caso un despliegue, lance las pruebas de forma automática sin necesidad de que intervenga ninguna persona.

Además, se ha añadido un contenido nuevo a la plataforma, un ejercicio de tratamiento de imagen en la parte offline de Unibotics. El objetivo de este ejercicio es permitir a los usuarios introducirse en el mundo del procesamiento de imagen con la biblioteca OpenCV a través de una serie de retos.

Se ha usado Django como entorno para el desarrollo web y todas las pruebas y automatización se han realizado con Selenium, ya que ofrece gran cantidad de recursos con los que trabajar y además es multilenguaje y multiplataforma. El lenguaje en que se ha desarrollado este TFG es Python, y en menor medida también se ha usado OpenCV, GitHub Actions y el módulo *subprocess* para automatizar el levantamiento y apagado de contenedores, entre otras funciones.

Índice general

Índice de figuras

Capítulo 1

Introducción

Este Trabajo de Fin de Grado (TFG) gira en torno a la implementación de pruebas automatizadas en la plataforma web Unibotics. Estas pruebas evitarán a los desarrolladores repetir las pruebas manuales en cada uno de los despliegues de la plataforma, permitiendo ahorrar tiempo y tareas repetitivas a la vez que contribuyen a la robustez de la plataforma.

Para ilustrar el contexto en el que se encuadra este TFG, a lo largo de este capítulo se realiza una introducción a las tecnologías web, sistemas de CI/CD y a las plataformas de robótica educativa.

1.1. Tecnologías Web

Las tecnologías web son el conjunto de herramientas y estándares que se usan para crear y presentar el contenido de Internet. Estas permiten la creación de páginas y aplicaciones web accesibles y que se pueden usar desde distintos navegadores que a su vez pueden estar instalados sobre distintos sistemas operativos.

Las aplicaciones web son programas que se ejecutan en un navegador (Google Chrome, Mozilla Firefox, etc) y ofrecen servicios a sus usuarios a través de Internet, sin necesitar instalaciones particulares en las máquinas de usuario, lo que las convierte en una herramienta cómoda de utilizar.

A la vez que Internet ha evolucionado, también lo han hecho las aplicaciones web, pasando de ser unas aplicaciones que en sus inicios sólo ofrecían la posibilidad de rellenar formularios a ofrecer funcionalidades tan variadas como: comercio online, aplicaciones web multimedia donde se puede consumir audio y vídeo o portales web que ofrecen chats.

Figura 1.1: Aplicación web de Netflix

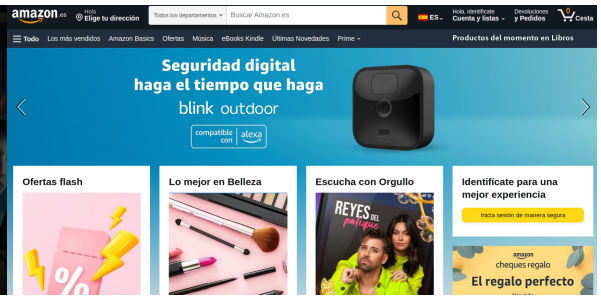

Figura 1.2: Aplicación web de Amazon

Estas aplicaciones web usan una arquitectura cliente-servidor (ver Figura 1.3), donde el cliente es el navegador Web y el servidor se encarga de servir contenidos y código a ejecutar en el navegador.

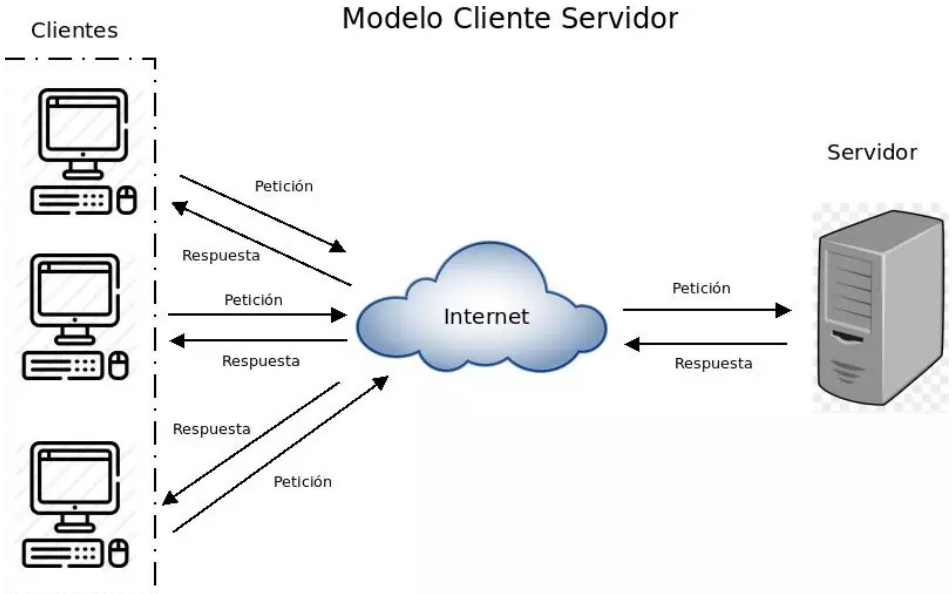

Figura 1.3: Arquitectura cliente-servidor Fuente

Normalmente estas aplicaciones web funcionan de la siguiente manera; Como primer paso, el cliente accede a la aplicación web especificando una dirección, una URL, esto genera una solicitud HTTP al servidor. Este servidor recibe la solicitud y se encarga de procesarla, para ello debe saber qué se está solicitado y en consecuencia realizar todas las acciones necesarias, como puede ser realizar cálculos, acceder a bases de datos, etc.

Cuando el servidor ha procesado la petición, genera una respuesta en el formato que sea adecuado y contenga todo lo necesario para que el navegador del cliente sea capaz de renderizar la página. Después, envía la respuesta de vuelta al cliente usando el protocolo HTTP.

Y finalmente, una vez el navegador recibe la respuesta, muestra el contenido y apli-

ca todo lo necesario para mostrárselo al usuario. A partir de aquí, el usuario ya puede interactuar nuevamente con la aplicación web. Alguna interacción puede generar nuevas solicitudes al servidor, y se repite el ciclo hasta que se deje de utilizar la aplicación web.

La comunicación entre cliente y servidor, hace uso del protocolo HTTP. El protocolo HTTP (Hypertext Transfer Protocol) es un protocolo que usan navegadores y servidores para intercambiar datos en la Web. Estos mensajes tienen una estructura definida (ver Figura 1.4):

- **Linea de inicio**: Esta línea de inicio es única, exclusiva para las peticiones y está formada por tres elementos que ayudan a describir la petición. El primero de ellos; el *verbo* que describe la acción a realizar, un GET significa que se quiere recuperar información, un POST que se envían datos al servidor, un PUT que se va a modificar información, etc. El segundo elemento es la *URL*, que se refiere al recurso solicitado. Y por último, la versión de HTTP.
- **Linea de estado**: Es un campo exclusivo de las respuestas HTTP, estas líneas llevan un código que indica si la petición se ha procesado correctamente (código 200) o ha ocurrido algún error en el servidor (código entre los 500) o en el cliente (código entre los 400).
- **Cabeceras**: Son campos adicionales que dan algo de información adicional o sirven para controlar ciertos comportamientos de la solicitud. Un ejemplo puede ser la cabecera Content-Type, que identifica el tipo de contenido que se envía en el cuerpo de la petición o respuesta (JSON, HTML, JavaScript, etc).
- **Cuerpo**: Es una sección adicional, ya que además no todos los métodos de una petición requieren un cuerpo. El cuerpo contiene la información que se envía al servidor o que este devuelve.

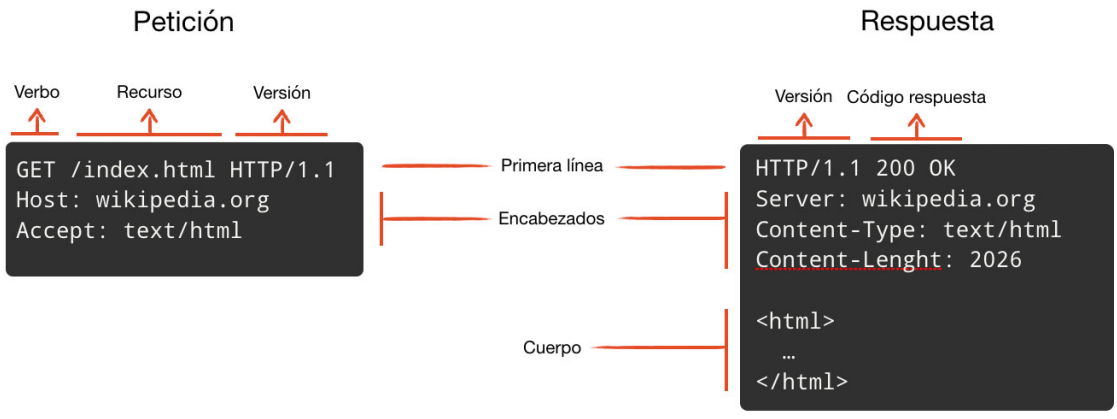

Figura 1.4: Estructura mensajes HTTP Fuente

Para que estas aplicaciones web sean posibles, ha sido necesario el desarrollo de tecnologías web tanto del lado de cliente, como del lado del servidor.

1.1.1. Tecnologías Web del lado cliente

Las tecnologías del lado cliente son las que se ejecutan en el navegador del usuario y permiten la interacción entre el usuario humano y la web. Entre las principales tecnologías del lado cliente se encuentran:

- **HTML5**: *HyperText Markup Language*. Usado para definir la estructura de una página mediante etiquetas que representan distintas partes de la web: títulos, párrafos, imágenes, enlaces, etc.
- **CSS**: *Cascading Style Sheets*. Se usa para definir la apariencia de la web.
- **JavaScript**: Es el lenguaje de programación que permite añadir funcionalidad a las páginas, como puede ser efectos visuales o validación de formularios entre otras muchas.

1.1.2. Tecnologías Web del lado servidor

Las tecnologías web del lado servidor son aquellas tecnologías que se utilizan para desarrollar la funcionalidad de una aplicación web. Estas se ejecutan en los servidores y procesan las solicitudes de los clientes, realizan operaciones en las bases de datos, generan la respuesta y la envían al cliente. Algunas de estas tecnologías son:

- **Lenguajes de programación del lado servidor**: Entre los más comunes están Python, Java, JavaScript, PHP o C. Con ellos se escribe código que se ejecuta en el servidor y tiene distintas funcionalidades.
- **Entornos del lado servidor**: Simplifican el desarrollo de las aplicaciones web al facilitar las tareas más comunes cuando se programa en un servidor web. Por ejemplo: Django en Python, Express.js en Node.js, Laravel en PHP o Ruby on Rails.
- **Bases de datos**: En ellas se almacena o recupera información necesaria para la aplicación web de forma estructurada. Existen sistemas de gestión de bases de datos, entre los que destacan: MySQL, MongoDB o SQLite.

Las tecnologías utilizadas en el lado de servidor tienen más variedad que aquellas en lado cliente. Todas tienen sus ventajas e inconvenientes y se suele elegir la más adecuada en función del objetivo del proyecto que se esté desarrollando.

1.2. Sistemas CI/CD

A la vez que evoluciona Internet, también lo hacen las aplicaciones web que podemos encontrar en ella, evolucionando desde formularios a, por ejemplo, plataformas de compra en línea. Las aplicaciones web deben evolucionar también, adaptándose a las novedades. Es por ello que los cambios en las plataformas web deben ser continuos y a un ritmo que permita a las empresas satisfacer la demanda y ofrecer productos seguros y de calidad. Es por eso que los sistemas de CI/CD (Continous Integration/Continous Delivery) se han vuelto imprescindibles.

Normalmente una vez un desarrollo es solicitado, este pasa por varios despliegues antes de llegar hasta el entorno que da servicio a los usuarios finales. Estos desarrollos y despliegues pueden llegar a ser procesos lentos, pues se gasta mucho tiempo en tareas y pruebas repetitivas para asegurar la compatibilidad con lo anterior, además de propensos a errores, como por ejemplo puede ser saltarse una prueba concreta, o no configurar alguna pieza software correctamente en el despliegue. Por ello se hace complicado mantener el ritmo entre entradas y nuevas peticiones de desarrollos. En busca de una solución para estos problemas, nacen los sistemas de CI/CD.

Estos sistemas de CI/CD tienen un objetivo; automatizar partes repetitivas y agilizar todos los pasos por lo que atraviesa el producto de software. Empezando por la integración del código, pasando por la definición y realización de pruebas automáticas y llegando hasta el despliegue en producción de estos desarrollos.

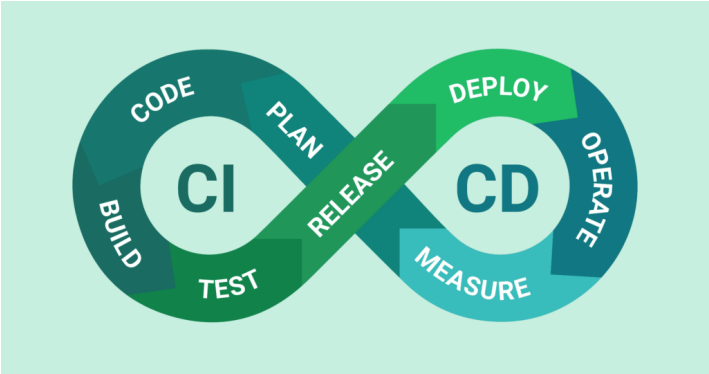

Figura 1.5: Esquema flujo CI/CD Fuente

Estos sistemas de CI/CD se definen como el conjunto de herramientas y procesos que permiten implementar CI/CD en cualquier proyecto de desarrollo de software. Normalmente, estos sistemas siguen un flujo de trabajo con pasos concretos, que son los siguientes:

- **Integración Continua (CI)**: La integración continua es una filosofía a la hora de programar, en la que se propone realizar cambios pequeños en el código de manera regular y en espacios cortos de tiempo, en lugar de realizar cambios más

grandes espaciados en el tiempo. Estos cambios y nuevos desarrollos se suben a un repositorio, donde el sistema detecta los cambios y los compila con el objetivo de verificar la integridad del código.

- **Pruebas automáticas**: Una vez se ha compilado el código, suele existir una serie de pruebas automáticas para asegurar que el código cumple con los requisitos pedidos. Estas pruebas pueden ser de varios tipos: pruebas unitarias, pruebas de integración o pruebas de rendimiento sin limitarse a sólo un tipo, es decir, puede existir cualquir combinación que se considere necesaria.

- **Despliegue continuo (CD)**: Después de que se hayan superado las pruebas, el sistema procede a desplegar la aplicación en el entorno que corresponda en ese momento, que puede ser producción u otros entornos previos. En este despliegue se incluye tanto el código, como la configuración que sea necesaria añadir o modificar.

- **Supervisión Continua**: Una vez la aplicación ya está desplegada, existe la posibilidad de monitorizarla en busca de problemas, con el objetivo de recoger información sobre los mismos o sobre el rendimiento de la aplicación.

Estas herramientas no sólo permiten detectar con rapidez posibles problemas, sino que también pueden tener la capacidad de responder ante los mismos y así ahorrar tiempo a desarrolladores y evitar errores humanos en las tareas repetitivas.

Entre las herramientas más populares de CI/CD, pueden encontrarse: Jenkins, GitHub Actions o Azure DevOps. A alto nivel, estas herrmientas funcionan de la siguiente manera:

- **Jenkins**: Se trata de un servidor de automatización. En él los desarrolladors pueden definir flujos de trabajo que se llaman *pipelines*, y estos pipeslines se crean mediante un lenguaje exclusivo para Jenkins (DSL) o con ayuda de una interfaz gráfica. El flujo básico consiste en que Jenkis comprueba cada cierto tiempo si hay cambios en repositorio del código fuente y, en caso de haberlos, se compila el código y se prepara un *build*. Si no hay error, el código se despliega en un servidor para pruebas y tras ellas se avisa al *committer* de los resultados. En caso de error, se envía también una notificación al desarrollador responsable o al equipo.

- **GitHub Actions**: Se trata de una plataforma de automatización de GitHub. Permite automatizar tareas como pruebas, compilación y despliegue, entre otros. Es cómodo de definir por los usuarios, y además ofrece acciones predefinidas que proporciona la comunidad de desarrolladores, que pueden ser tareas básicas o flujos complejos. Como punto fuerte, es posible ver el estado de los flujos y además recibir alertas o informes sobre cada uno de los pasos.

- **Azure DevOps**: El sistema de Azure es un conjunto de servicios que incluen: (1)*Pipelines*: para definir y configurar flujos de trabajo con archivos YAML, (2) Compilación continua; cada cambio en repositorio puede iniciar el proceso de compilación, (3) Pruebas automatizadas, (4) Despliegue continuo en diferentes

entornos, pudiendo definir los pasos de un despliegue. Además, se adapta bastante bien al entorno de desarrollo al ser integrable en servicios como Docker o Kubernetes. Por último, también ofrece monitorización para recopilar datos sobre el rendimiento en producción.

Estos sistemas de CI/CD se han convertido con el paso del tiempo en una parte importante del desarrollo de software. En el capítulo tres se explorará más uno de estos sistemas.

1.3. Plataformas de robótica educativa

La robótica se encarga de diseñar, construir y programar máquinas que pueden realizar tareas de forma autónoma o controlada por humanos. A estas máquinas se les llama *robots*. El objetivo al crear estos robots es que puedan encargarse tanto de una tarea simple como de algunas especialmente complejas, como pueden ser las que se realizen en entornos médicos o industriales. Para poder realizar estas tareas, los robots suelen estar equipados con sensores que captan información del exterior, un sistema capaz de ejecutar acciones basadas en la información que reciben y actuadores que les permiten interactuar con dicho entorno.

Hoy en día es posible encontrar robots en muchos aspectos de la vida cotidiana, como pueden ser los robot Roomba (ver Figura 1.6) que mucha gente tiene en casa. Fuera del entorno familia, pueden encontrarse robots como Aeo (ver Figura 1.7), un robot capaz de realizar servicios de entrega, cuidar personas mayores o incluso aumentar la seguridad. O robots aún más enfocados al ocio, como Dog-e (ver Figura 1.8) un perro robot que se puede entrenar, enseñar trucos y se comporta de forma similar a un animal real al hacer saber al usuario cuando quiere jugar o tiene "hambre". Otro ejemplo, pueden ser los muy conocidos robots de Boston Dynamics (ver Figura 1.9)

Figura 1.6: Robot roomba.

Figura 1.7: Robot Aeo.

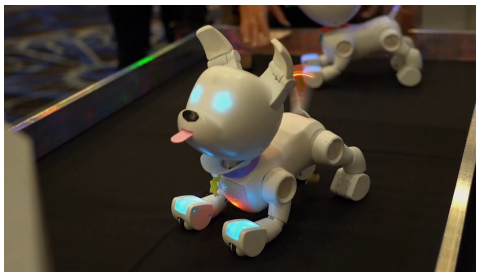

Figura 1.8: Robot Dog-e.

Figura 1.9: Robot Spot de Boston Dynamic.

Junto con al avance de la robótica, nacen las plataformas de robótica educativa para cubrir la necesidad de acercar la robótica a todo tipo de estudiantes, para ayudar a afianzar o aumentar conocimientos, y los más pequeños, con el objetivo de despertar su interés por este tipo de estudios

Estas plataformas permiten a los estudiantes conocer y explorar el mundo de la robótica y programación de forma práctica, es decir, los propios estudiantes pueden construir o programar sus robots. Ser capaces de ver la aplicación de lo que están haciendo es un aliciente a la hora de comprender y asentar los conocimientos, además de para atraer gente a las carreras STEM (Science, Technology, Engineering and Mathematics). Al no existir una única forma de resolver los retos la mayoría de las veces, también se fomenta la creatividad, pues invita a los usuarios a realizar mejoras en su solución.

Estas plataformas pueden tener diferentes formatos, como por ejemplo el físico para LEGO Mindstorms, que combina piezas de LEGO con la robótica e incluso una unidad de control en la que se puede programar para permitir al usuario crear su propio robot. En este caso, para programar la unidad de control se usa un software específico que se llama "LEGO Mindstorms Education EV3 Software"que permite programar de forma visual mediante bloques con los que el usuario crea instrucciones (ver Figura 1.10).

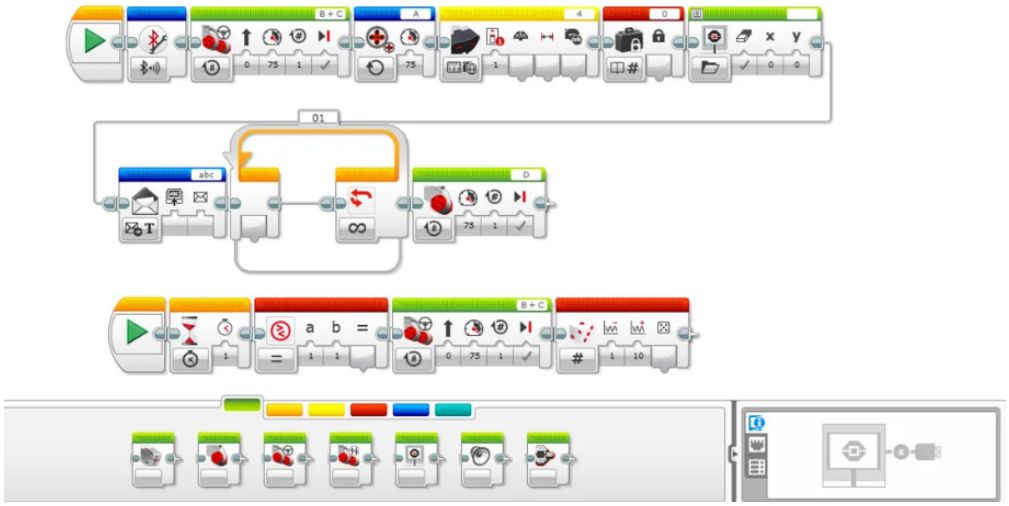

Figura 1.10: Interfaz de LEGO Mindstorms Education EV3 Software

También existen plataformas *online*, que permiten a los usuarios usarlas sin necesidad de tener que realizar instalaciones y en muchos casos de forma gratuita. Entre estas plataformas *online* destaca Scratch, creada por el grupo Lifelong Kindergarten del MIT Media Lab y diseñado especialmente para niños o principiantes en el mundo de la programación, permitiéndoles aprender de forma accesible y entretenida. El código fuente se crea combinando bloques que representan diferentes comandos (ver Figura 1.11). Otro punto a destacar es que Scratch permite y fomenta la colaboración, pues los usuarios pueden compartir sus proyectos y explorar los de otras personas, de forma que los usuarios aprenden unos de otros.

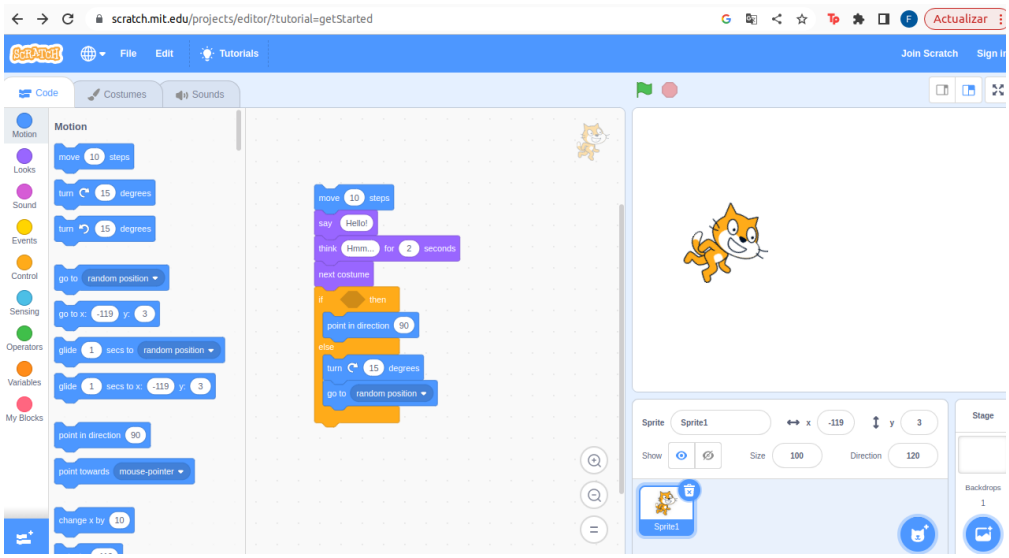

Figura 1.11: Interfaz de Scratch

Existen más opciones de plataformas de robótica educativa de programación *online*, como *Code.org* donde también se utiliza programación por bloques y la colaboración entre usuarios. *Tynker*, otra plataforma con programación basada en bloques que además tiene una extensión para Minecraft, de forma que los usuarios pueden personalizar elementos del juego. O *EdPy*, donde el usuario escribe código Python que se ejecuta en un robot llamado Edison.

Durante el desarrollo de este TFG se ha trabajado con la plataforma web Unibotics (ver Figura 1.12), dirigida a estudiantes de ingeniería que ya poseen una base de conocimiento de programación o usuarios que deseen ampliar conocimientos. Unibotics es una plataforma online, gratuita y multi-sistema operativo que permite programar robots simulados en distintos escenarios.

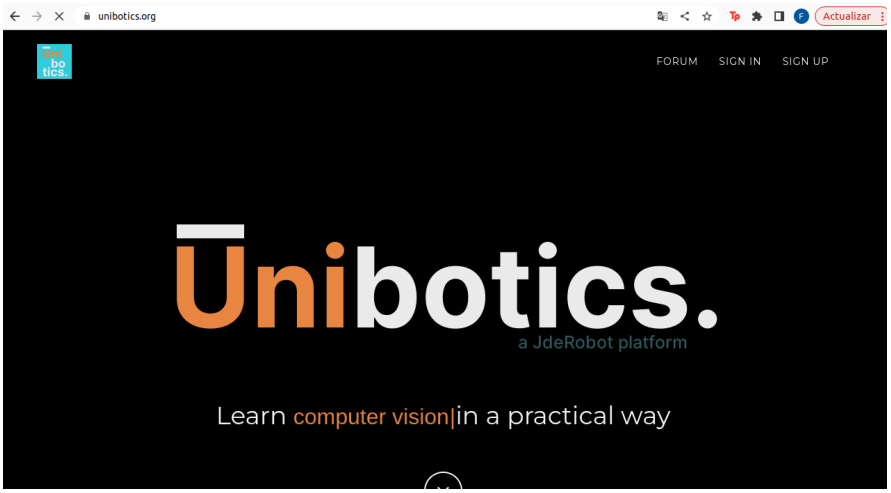

Figura 1.12: Página web de la plataforma Unibotics

Unibotics ofrece una variedad de ejercicios que abarcan distintos aspectos de la robótica, como puede ser la programación de robots de servicio, el procesamiento de imagen o el aprendizaje automático. Dentro de estos ejercicios, existen recursos para ayudar al usuario como documentación sobre el objetivo y teoría detrás del ejercicio, pistas sobre la forma de resolverlos e incluso vídeos demostrativos de la solución de referencia.

Además, dispone de torneos automatizados que pueden retransmitirse vía *streaming*, lo que puede atraer a usuarios. Estas competiciones junto con la capacidad de recibir una evaluación automática del código ayudan a fomentar el análisis y la buena organización a la hora de resolver un problema, además de la colaboración para lograr un código más eficiente.

En el capítulo 3 se extenderá la información sobre esta plataforma, y en los posteriores se explicará cómo se ha contribuido a su robustez en producción mediante la adición de pruebas automatizadas en los despliegues.

Capítulo 2

Objetivos y Planificación

2.1. Objetivos

Los objetivos de este Trabajo de Fin de Grado son principalmente dos. Primero, la implementación de pruebas automatizadas de la plataforma web Unibotics, capaces de detectar errores previos el despliegue a producción que permitan a los desarrolladores dar marcha atrás antes de que el error llegue a los usuarios. Y segundo, la implementación de un nuevo ejercicio de procesamiento de imagen, con pequeños retos de dificultad creciente.

Teniendo en cuenta estos dos objetivos principales, se puede dividir en tres subobjetivos concretos con alcance más pequeño:

1. **Implementación de pruebas unitarias de frontend**: El objetivo de estas pruebas es verificar el correcto funcionamiento y la existencia de todos los botones necesarios para el funcionamiento de la página web de cada ejercicio contenido en la plataforma.

2. **Implementación de pruebas integrales**: Que verifiquen el correcto funcionamiento del sistema mediante la simulación de acciones completas de un usuario humano desde iniciar sesión en la página, levantar el contenedor, preparar el ejercicio, insertar código fuente en la página y ejecutarlo hasta poder comprobar que se ejecuta correctamente.

3. **Implementación de un nuevo ejercicio de procesamiento de imagen**: Se tratará de un ejercicio introductorio basado en la biblioteca OpenCV con un video fijo como fuente de imágenes en lugar de utilizar la cámara del usuario en vivo. Además, se le especificarán a este unos retos que aumentan progresivamente el nivel de dificultad.

Para las pruebas automatizadas se realizará toda la configuración necesaria del entorno de pruebas, desarrollo y obtención de resultados de las mismas. Mientras que en el caso del ejercicio se desarrollará la infraestructura necesaria, documentación y material de video para integrarlo en RoboticsAcademy.

2.2. Metodología

La metodología seguida en este TFG se basa en la metodología *scrum*, entre cuyas características principales se encuentra que los requisitos y objetivos de un proyecto pueden cambiar con el tiempo y adaptarse a estos cambios, además de la colaboración entre los miembros del equipo, organización en *sprints* que son periodos de tiempo delimitados en los que se trabaja en un número de finito de tareas, entrega de resultados de forma frecuente para poder recibir retroalimentación y continuar mejorando.

Mientras se desarrollaban y surgían problemas o se completaban los distintos objetivos y durante toda la realización del Trabajo de Fin de Grado se han mantenido reuniones semanales con los tutores a través de Google Meet con el objetivo de repasar los objetivos de la semana, resolver dudas, recibir retroalimentación sobre los siguientes pasos a seguir o tratar de resolver puntos conflictivos. Además, también se ha contado con un canal de Slack donde se encuentran el resto de desarrolladores de la plataforma Unibotics para consultar dudas con ellos y recibir respuestas y soporte de estos.

Cada objetivo ha tenido distintas fases, y la primera de ellas siempre ha sido realizar el desarrollo en local para tratar de encontrar la mejor forma de cumplir con el objetivo. Una vez el desarrollo se encontraba funcionando en local, se adaptaba y probaba en el despliegue D1 "de desarrolladores"de Unibotics, y se volvía a comprobar el funcionamiento además de resolver posibles errores.

Después de que el desarrollo estuviese funcionando correctamente en el despliegue D1, se procedía a comentarlo en la reunión semanal y crear una incidencia en el repositorio de unibotics-webserver. A partir de esta incidencia y tomando su identificador, se crea una rama copia de la rama master cuyo nombre es el identificador anteriormente mencionado. De esta forma qué se está desarrollando en dicha rama es localizable para cualquier otro desarrollador del equipo. Sobre esta rama recién creada se realizan *çommits"* y *"push"* de los nuevos archivos y cambios que otros hubiesen podido necesitar. Una vez todo lo necesario se encuentra en el repositorio, se crea un parche contra la rama master y se asocia tanto a la incidencia como a uno de los revisores del equipo para su previa comprobación antes de fusionarlo con la rama master.

2.3. Plan de Trabajo

Se dividido el plan de trabajo en cuatro etapas:

1. **Primera etapa**:

    a) **Introducción y aterrizaje**: Durante esta primera etapa, fue esencial el repaso de las tecnologías utilizadas a lo largo de la carrera, el conocimiento de las nuevas tecnologías y el aterrizaje en la plataforma de Unibotics. Las tecnologías que se estudiaron y repasaron fueron:

- Python, lenguaje usado en Django y en el que se ha desarrollado casi al completo este TFG.
- Selenium para permitir la automatización de las pruebas tanto unitarias de frontend, como de integración.
- OpenCV para el desarrollo y resolución del ejercicio de introducción al procesamiento de imagen.
- Docker para entender el funcionamiento del contenedor RADI usado en Unibotics y la manera de automatizar distintas acciones que involucran al mismo.
- Unibotics como usuario para empezar a familiarizarse con la plataforma.

2. **Segunda etapa**:

   a) **Diseño de una prueba unitaria de frontend**: Seleccionar uno de los ejercicios disponibles para realizar sobre él una primera versión de prueba de frontend, con sólo las acciones de Selenium automatizadas.

   b) **Automatización de la primera prueba unitaria de frontend**: Partiendo de la prueba creada en la etapa anterior, automatizar completamente la misma para eliminar los parámetros a fuego (como puede ser la url sobre la que realizar pruebas), pues esta cambiará entre los distintos entornos, además de la creación de un usuario de pruebas que libre al desarrollador de introducir credenciales.

   c) **Incorporación de esta primera prueba al sistema CI/CD de Unibotics**: Para comprobar que el desarrollo realizado y probado en despliegue D1 es compatible con el resto de entornos, se integra y sube el primer test de frontend en los despliegues de Unibotics.

   d) **Ampliación de las pruebas al resto de ejercicios**: Tras comprobar que este desarrollo funciona en el despliegue a producción, se utiliza como base para replicar estas pruebas en el resto de los ejercicios existentes en la plataforma Unibotics, adaptándose a las particularidades que cada uno de ellos pueda tener.

3. **Tercera etapa:**

   a) **Desarrollo de la primera prueba de integración**: Estas pruebas se encargarán de verificar el funcionamiento del sistema completo desde la inyecciónde código fuente en un ejercicio por parte del usuario humano hasta su ejecución correcta, y han requerido desarrollar los siguientes puntos.

   - Automatización del levantamiento del contenedor RADI.
   - Automatización de inserción de código de usuario.
   - Inserción automática de archivos necesarios para conseguir la información de los ROS *topics* correspondientes a sensores y actuadores del robot.
   - Procesado y comprobación de los resultados obtenidos.

- Borrado de los archivos introducidos en el contenedor.
- Apagado automatizado del contenedor RADI una vez las pruebas finalizan.

b) **Incorporación de la primera prueba integral al sistema CI/CD de Unibotics**: Al igual que las de frontend, es necesario comprobar que el desarrollo realizado y probado en el despliegue D1 es compatible con el resto de despliegues, D2 y D3.

c) **Expansión de las pruebas al resto de ejercicios**: Tras verificar que la primera prueba integral incorporada funciona, se expanden las pruebas al resto de ejercicios ofrecidos en Unibotics, teniendo siempre en cuenta la diferencia entre el código que se debe insertar, los formatos de mensajes que intercambian los ROS *topics* y la comparativa de las posiciones que se debe hacer.

4. **Cuarta etapa**:

a) **Estudio del código fuente de RoboticsAcademy**: Para poder realizar un nuevo ejercicio es primero necesario saber cómo funciona la plataforma, que es la versión offline de Unibotics

b) **Desarrollo del ejercicio de procesamiento de imagen**: Creación de los archivos necesarios para el ejercicio, propuesta de los retos y grabación del material de vídeo con la solución de referencia.

Capítulo 3

Herramientas utilizadas

A lo largo de este capítulo se introducirán y explicarán brevemente todas las herramientas usadas durante el desarrollo de este trabajo, siendo los elementos principales Selenium, Django y OpenCV. También se tratarán otros elementos indispensables, tales como Docker, Github Actions y los lenguajes HTML y Python. Como punto final de este capítulo, se presentará la plataforma web Unibotics, que es la aplicación web en la que se ha contribuido.

3.1. Selenium

Selenium[1] es una herramienta para realizar automatización de pruebas de software. Es de código abierto y se utiliza normalmente para probar aplicaciones web. Esta herramienta permite simular las acciones que un usuario humano real podría realizar en un navegador web.

Figura 3.1: Logo de Selenium

La historia de Seleniun comienza en 2004, cuando el ingeniero Jason Huggins desarrolla la herramienta *JavascriptTestRunner* con el objetivo de automatizar las pruebas de una aplicación web de la empresa ThoughtWorks. Creyendo que tenía potencial al ser capaz de librar a los programadores de pruebas repetitivas, la lanzó poco después como una herramienta de código abierto bajo el nombre Selenium Core.

[1] https://www.selenium.dev/

Un par de años después, en 2006, otro ingeniero de la misma empresa, Paul Hammant, desarrolló una extensión que además de permitir desarrollar el código de las pruebas en varios lenguajes de programación, también permitía ejecutarlas en distintos navegadores. Fue llamado Selenium Remote Control, o Selenium RC.

En 2008 Selenium RC se fusionó con otro proyecto de código abierto: WebDriver, cuyo objetivo también era la automatización de pruebas. El resultado de esto fue Selenium WebDriver.

Hoy en día, Selenium se usa en empresas tan conocidas como Netflix, Google, HubSpot y una larga lista. Además, dentro del conjunto de Selenium, existen distintos componentes que ofrecen diferentes funcionalidades. Entre ellos destacan:

- **Selenium IDE (Integrated Development Environment)**: Extensión del navegador Firefox creada por el japonés Shinya Kasatani. Se trata de una herramienta fácil de usar y que no requiere conocimientos avanzados de programación. Selenium IDE permite grabar y reproducir pruebas, aunque su funcionalidad está limitada a pruebas básicas y no ofrece demasiado nivel de personalización.
- **Selenium Grid**: Este componente permite ejecutar pruebas de Selenium al mismo tiempo en varias máquinas y navegadores diferentes. Fue creado por Patrick Lightbody con el objetivo de minimizar el tiempo de pruebas y facilitar su ejecución pero, además de optimizar el proceso de pruebas, también permite comprobar la compatibilidad entre distintos navegadores.
- **Selenium WebDriver**: Es la herramienta más usada y más potente de Selenium, ya que permite interactuar con un navegador en tiempo real y realizar acciones complejas. Es compatible con varios lenguajes de programación como Python, Java, C++, PHP o Ruby entre otros. También puede ser utilizado en distintos navegadores (siempre que se descargue el controlador correspondiente) y distintos sistemas operativos.

Para la realización de este TFG se ha utilizado Selenium WebDriver en su versión 4.1.3 por lo que, la explicación sobre la arquitectura se centrará en la de esta versión.

La arquitectura de Selenium Webdriver consta cuatro componentes:

- **Librerías cliente:** Proporcionan los métodos y funciones necesarios para escribir el código y enviar comandos al navegador, son librerías que permiten a los programadores utilizar Selenium WebDriver en distintos lenguajes de programación.
- **Protocolo JSON Wire:** Define formatos JSON estandarizados para los comandos y respuestas que se producen por la comunicación entre las librerías de cliente de Selenium y los controladores de los navegadores. Este protocolo está basado en HTTP.

- **Controladores:** Actúan como intermediarios entre Selenium WebDriver y el navegador. Su función es interpretar los comandos enviados por Selenium y traducirlo a acciones en el navegador. Cada navegador (Google Chrome, Mozilla Firefox, etc) tiene su propio controlador.
- **Navegadores**: Los navegadores web sobre los que se realizan las pruebas.

Una vez se obtiene respuesta del navegador, esta sigue el flujo a la inversa. Es decir, se devuelve desde el navegador a través del controlador (ver Figura 3.2).

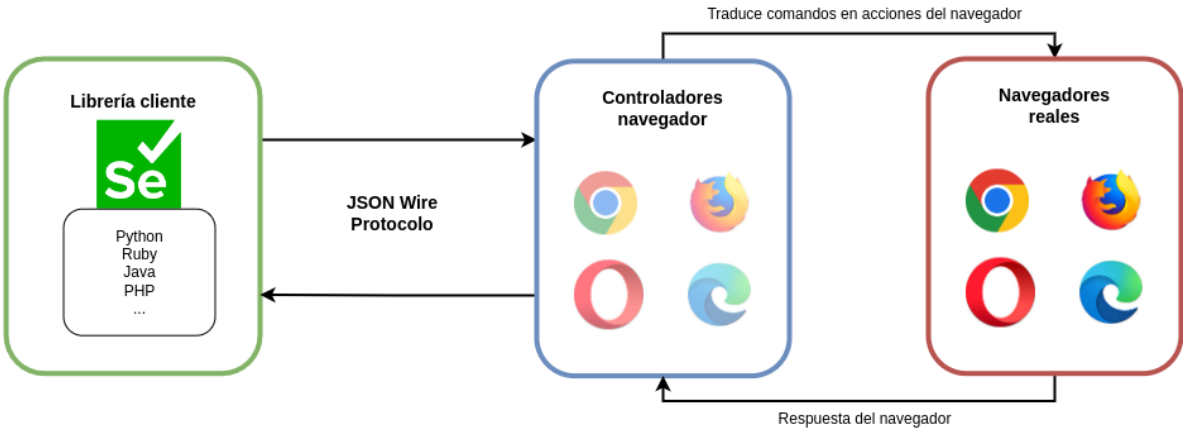

Figura 3.2: Diagrama arquitectura Selenium Webdriver

3.2. Django

Django[2] es un entorno de desarrollo web de alto nivel, de código abierto y gratuito escrito en Python. Al proporcionar una estructura y una serie de herramientas preconstruidas que ayudan a resolver tareas comunes, facilita la creación rápida de aplicaciones web complejas y robustas. Durante la realización de este TFG se ha usado la versión 3.2.12.

Su historia se remonta a otoño del 2003, cuando los programadores de The World Online eran responsables de varios sitios web de noticias locales. Debido a las peticiones de parte de directivos y periodistas para agregarles nuevas características a estas páginas o crear otras enteras con sólo unos días u horas de margen, dos de los desarrolladores de este equipo: Adrian Holovaty y Simon Willison, desarrollaron un entorno de desarrollo web que fuese capaz de ahorrarles tiempo, pues era la única forma de cumplir con las fechas y asegurarse de entregar un trabajo de calidad.

Después de que este entorno estuviese haciendo funcionar muchos de los sitios web de la empresa World Online, en Julio de 2005, decidieron liberarlos como software de

[2]https://www.djangoproject.com/

código abierto bajo el nombre de Django.

A día de hoy, es uno de los entorno más usados para el desarrollo web y aunque los desarrolladores originales aún aportan una guía centralizada, cuenta con miles de usuarios y contribuidores que ayudan a mejorarlo prácticamente a diario.

Figura 3.3: Logo de Django

Entre sus características destacan:

- **Rapidez**: Fue diseñado para ayudar a los desarrolladores a crear sus aplicaciones web con tanta rapidez como sea posible.
- **Seguridad**: Ayuda a los programadores a evitar los problemas más comunes de seguridad con, por ejemplo, las siguientes funcionalidades:
  - **Protección contra ataques XSS**: Los templates de Django ayudan a protegerse contra la mayoría de estos ataques en los que el usuario inserte scripts maliciosos en el navegador de otros usuarios gracias a que escapan caracteres peligrosos para HTML.
  - **Protección contra CRSF**: En estos ataques un usuario ejecuta acciones usando credenciales de un segundo usuario sin consentimiento ni conocimiento del mismo. Django incluye protección contra estos ataques siempre que se habilite al verificar un secreto en cada solicitud. De esta forma un tercero no puede copiar una solicitud al sitio web, ya que para hacerlo necesitaría conocer el secreto específico de ese usuario.
  - **Protección contra inyección SQL**: Django parametriza las consultas para proteger las *"querys"* y que no inserten código SQL en una base de datos, separando el código SQL de los parámetros proporcionados por el usuario.
  - **Protección contra clickjacking**: Un sitio malicioso envuelve a otro en un marco, por ello el usuario confiándose puede realizar acciones no deseadas. Django tiene una opción para evitar esto siempre que el navegador sea compatible
  - **Protocolo HTTPS**: Es importante utilizar este protocolo para evitar la fuga de información o que los atacantes puedan modificar datos

  Aparte de estas, ofrece más funcionalidades de seguridad como pueden ser: validación de las cabeceras Host o Referer, pero es importante recordar que ningún método es infalible o completamente seguro y los desarrolladores deben tomar precauciones adicionales.

- **Escalable**: Permite manejar grandes cantidades de tráfico y procesar solicitudes eficientemente al contar con herramientas que facilitan la escalabilidad, como pueden ser: caché, división de las bases de datos en particiones o la replicación de servidores.

- **Portátil**: Al estar escrito en Python, que es un lenguaje multiplataforma, no sólo puede usarse en diferentes sistemas operativos como Windows, Linux o macOS, sino que también se puede instalar en diferentes servidores web y entornos de producción. Por todo lo anterior, se adapta con facilidad a las necesidades o requisitos del usuario.

- **Mantenimiento**: Promueve el desarrollo de código reutilizable y eso facilita el trabajo, mantenimiento y velocidad de desarrollo.

- **Plantillas**: Se trata de archivos HTML que tienen marcadores y etiquetas específicas de Django y se utilizan para generar de forma dinámica el contenido de las páginas web.

Django se basa en el patrón **modelo-vista-controlador**. Este patrón permite diferenciar la lógica de negocio (modelo), la interfaz de usuario (vista) y el control de la aplicación (controlador) (ver Figura 3.4) al cumplir cada una de las capas una función específica:

- **Modelo**: En esta capa se define la estructura de los datos que se almacenan en base de datos. Los modelos son clases de Python y cada instancia de un modelo va a representar una fila en una tabla de la base de datos. Además, estos modelos facilitan métodos para poder interactuar con los registros de base de datos como por ejemplo leer, crear, modificar o incluso borrar.

- **Vista**: Esta capa se encarga de presentar los datos al usuario y recibir las acciones de este. Recibe la petición de parte del mismo y devuelve la respuesta que debe enviarse al navegador web utilizando los datos proporcionados por el modelo.

- **Controlador**: Se trata de una capa intermedia entre modelo y vista, que se asegura de que ambas partes estén sincronizadas y actualizadas. Recibe de la vista las entradas del usuario y después de procesarlas, actualiza el modelo y vista en consecuencia. Se podría decir que esta capa es el "cerebro" del patrón.

Usar este patrón modelo-vista-controlador permite desarrollar aplicaciones escalables y modularizadas, pues cada componente está separado de los otros de forma evidente. Además, permite reutilizar el código ya que los componentes pueden usarse en diferentes aplicaciones al ser independientes.

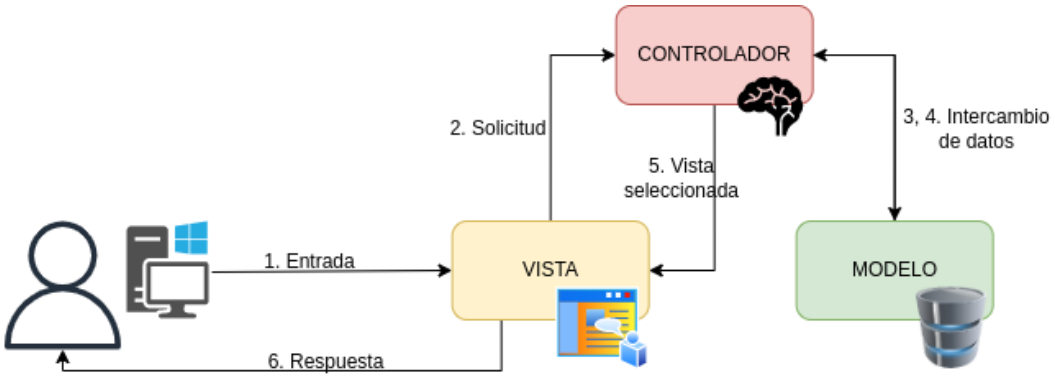

Figura 3.4: Diagrama patrón modelo-vista-controlador

3.3. Docker

Docker[3] permite que los desarrolladores empaqueten aplicaciones y dependencias en unidades llamadas **contenedores**. Cada contenedor es una unidad independiente que contiene entre otras cosas: código, bibliotecas y todo lo necesario para que una aplicación se ejecute. Además, estos contenedores son lo suficientemente ligeros como para poder moverlos entre servidores o entornos de ejecución.

Gracias a las funcionalidades de Docker es posible crear entornos de prueba y producción idénticos, lo que permite que las aplicaciones se ejecuten de la misma forma en todos los entornos por los que una integración pasa.

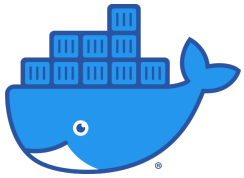

Figura 3.5: Logo de Docker

En este TFG se ha usado la versión 20.10.7 para crear las bases de datos MySQL, ElasticSearch y el contenedor RADI necesario para el funcionamiento de la plataforma Unibotics.

Docker está formado por los siguientes componentes:

- **Docker Daemon**: Es el servicio principal de Docker, se ejecuta en segundo plano en el sistema anfitrión y su función es gestionar los contenedores y las imágenes. Los usuarios pueden interactuar con él con peticiones al API de Docker o por línea de comandos.

[3]https://www.docker.com/

- **Docker Client**: Herramienta de línea de comandos, para que el usuario interactúe con el Docker Daemon y pueda enviarle peticiones para manejar contenedores e imágenes.
- **Imágenes de Docker**: Son la base para crear contenedores. Una imagen es una plantilla con todos los elementos necesarios para crear un contenedor y ejecutar una aplicación. Estos archivos se crean a partir de los Dockerfile, que contienen todo lo que debe incluirse en la imagen.
- **Contenedores de Docker**: Son instancias en ejecución de una imagen Docker. Cada uno de estos contenedores tienen su sistema de archivos y su sistema operativo diferente del anfitrión.
- **Docker Registry**: Repositorio para almacenar imágenes, puede ser público como Docker Hub o privado y administrado por una organización.
- **Docker Compose**: Define y ejecuta aplicación Docker multi-contenedor. Facilita también la creación y ejecución de aplicaciones completas al poder especificar la configuración de contenedores, red y almacenamiento dentro de un archivo YAML.

3.4. Github Actions

GitHub Actions[4] son un servicio de integración y entrega continua (CI/CD) que proporciona GitHub. Gracias a GitHub Actions es posible automatizar pruebas y despliegues al tener la posibilidad de crear flujos de trabajo personalizados.

Estos flujos se pueden desencadenar por eventos, como puede ser la creación de issues, fusión de ramas, etc. Para que sea posible ejecutar estos flujos de trabajo el propio GitHub proporciona máquinas Linux, Windows o MacOS, pero también es posible utilizar máquinas propias o en la nube.

Para entender un poco mejor cómo funciona GitHub Actions y su terminología, es necesario conocer también sus componentes según la documentación oficial de GitHub sobre las GitHub Actions[5]:

- **Workflow**: Un *workflow* es un proceso configurable y automatizado que va a lanzar uno o más *Jobs*. Estos flujos de trabajo pueden desencadenarse de forma manual o por eventos. Un mismo repositorio puede tener tantos *workflows* como se consideren necesarios, cumpliendo cada uno una tarea o incluso haciendo referencia a uno dentro de otro. Para encontrar estos ficheros dentro de un repositorio basta con mirar el directorio .github/workflows.
- **Eventos**: Una acción específica en el repositorio puede ser considerada un evento. Por ejemplo, fusionar y subir cambios en el código. Estos eventos pueden, o no,

[4]https://docs.github.com/es/actions
[5]https://docs.github.com/es/actions/learn-github-actions/understanding-github-actions

activar un *workflow*. En este enlace[6] pueden encontrarse todas los eventos que podrían desencadenar un workflow.

- **Jobs**: Es una tarea específica a realizar, como compilar un código por ejemplo. Estos *Jobs* vienen definidos en un archivo YAML que contiene los pasos necesarios para completar. Estos pasos se ejecutan secuencialmente. Dentro de un workflow puede haber de 1 a n Jobs y estos pueden ejecutarse secuencial o paralelamente, dependiendo de la configuración.

- **Actions**: Fragmentos de código de un *job* que se ejecuta dentro del mismo, por ejemplo: construir un contenedor Docker. Estas acciones son creadas por los usuarios de GitHub y se pueden compartir, reutilizar y personalizar para las necesidades de cada proyecto.

- **Runners**: Los *Runners* son los servidores donde se ejecutan los workflows cuando estos se activan. Existen dos tipos. Primero Hosted Runners (alojados) que proporciona el propio GitHub con sistemas operativos como Ubuntu Linux, Windows o macOS. Estos Runners son gratuitos y están disponibles para todos los usuarios. Segundo, también existen los Self-Hosted Runners (autosuficientes), estos pueden ser configurados en máquinas que tengan acceso a internet y cumplen unos requisitos mínimos, pero requieren configuración y mantenimiento.

En este TFG se han utilizado GitHub Actions para configurar un *workflow* y que los test automáticos se ejecuten en cada despliegue a producción de Unibotics.

3.5. Python

Python[7] es el lenguaje de programación en el que se ha desarrollado este TFG, al ser Django un entorno escrito sobre Python y haber usado la biblioteca de Selenium para este mismo lenguaje. La versión sobre la que se ha realizado el trabajo es la 3.10.6.

Se trata de un lenguaje de programación de alto nivel, creado en 1991 por Guido Van Rossum. Entre sus características más destacadas está su sintaxis sencilla, lo cual lo convierte en un lenguaje amigable y fácil de leer. Además, permite programar utilizando varios estilos, como pueden ser la programación orientada a objetos, programación funcional o programación estructurada.

Entre sus ventajas podemos listar:

- Sintaxis sencilla y fácil de recordar.

- No necesita compilar antes de ejecutar el código, las acciones se leen en tiempo de ejecución.

[6] https://docs.github.com/es/actions/using-workflows/events-that-trigger-workflows
[7] https://www.python.org/

- Tiene muchas librerías y entorno, lo que permite ahorrar tiempo y esfuerzo.
- Es multiplataforma, por lo que se puede usar en distintos sistemas operativos.
- Es gratuito y de código abierto, con una gran comunidad para resolver dudas y que aportan funcionalidades.
- Tipado dinámico, no es necesario declarar el tipo de una variable antes de inicializarla. Además, dependiendo del valor que tenga esta en un momento y otro, puede cambiar de tipo.

Es esta versatilidad la que hace que Python sea un lenguaje algo más lento que otros de sus competidores y consuma más memoria. A pesar de ello, las ventajas superan las desventajas para el desarrollo de este TFG.

3.6. HTML5

HTML[8] (Hypertext Markup Language), no es un lenguaje de programación, sino un lenguaje de marcado. Es una forma de definir la estructura de un documento o web mediante etiquetas, además, una de sus principales características es que estos archivos no se ejecutan sino que se leen, representan y son invisibles para el usuario que navega por la página web. La palabra *Hypertext* de su nombre, referencia un elemento fundamental de la Web: los enlaces que conectan páginas web entre sí o distintas partes dentro de una misma web.

El HTML puede considerarse la pieza más básica de una web, ya que como se explica anteriormente, ayuda a definir la estructura de la misma. Al archivo HTML lo suelen acompañar dos más: CSS para dar estilo a la página web, y JavaScript, para dotar de funcionalidad a la misma, de ejecución de cierto código fuente.

La forma de diferenciar los diferentes componentes de una web, son las etiquetas. HTML utiliza etiquetas como pueden ser <head>, <head>, <body> entre muchas otras. Es muy importante que cada etiqueta tenga su cierre.

Un ejemplo de un HTML básico y su estructura podría ser la siguiente figura:

[8] https://dev.w3.org/html5/spec-LC/

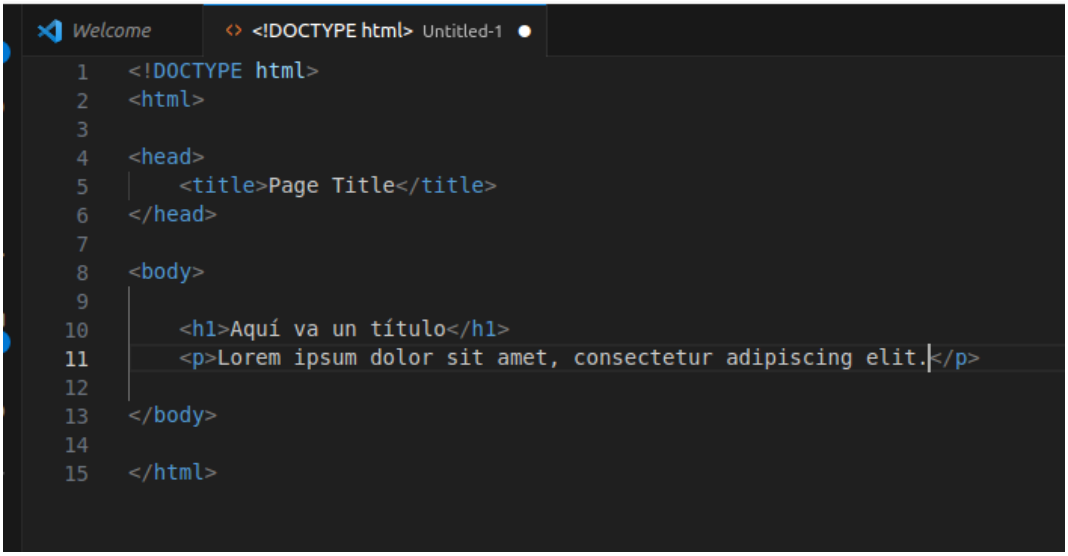

Figura 3.6: Ejemplo de HTML básico

Las etiquetas que conforman el ejemplo (ver Figura 3.6) pueden encontrarse en cualquier página web, y cada una de ellas tiene el siguiente significado:

- **<!DOCTYPE html>**: Es ligeramente diferente del resto de etiquetas y rompe la regla mencionada anteriormente al no tener etiqueta de cierre. Se usa para indicar al resto de programadores el tipo de documento.
- **<html></html>**: Es el elemento padre el documento y el más importante, engloba al resto de etiquetas de la página web
- **<head></head>**: Contienen datos que no se quieren mostrar el usuario, por ejemplo referencias a archivos CSS o JavaScript.
- **<body></body>**: Dentro de esta etiqueta se incluye todo el cuerpo de la página.
- **<h1></h1>**: Se utiliza para definir un título y según la importancia del mismo, se usan etiquetas desde h1 a h6, siendo h1 el tamaño más grande y la mayor importancia.
- ** **: Para escribir párrafos.

Además de estas que se muestran en el ejemplo hay otras con importancia, como para realizar divisiones dentro del contenido, <a>para insertar enlaces, <img>para imágenes o
para saltos de línea.

Conocer la estructura de una página web, las etiquetas y convenciones establecidas a la hora de crear una web ha sido esencial para la realización de este TFG, ya que Selenium interactúa con dichas etiquetas para localizar elementos en la página.

3.7. YAML

YAML[9] (YAML Ain't Markup Language) es un formato de datos legible que fue diseñado para ser fácilmente legible por humanos y utilizado en el intercambio de datos.

La estructura de un archivo YAML se basa en la combinación clave-valor y su sintaxis es sencilla. Para facilitar su entendimiento y editabilidad, los datos se organizan de forma jerárquica, usando sangría y los tipos de datos (cadenas, números, listas) se representan también de forma intuitiva (ver Figura 3.7)

```yaml
nominees:
    nominee1:
        name: Harry
        surname: Styles
        nationality: English
        records_count: 1
        records:
            record1:
                title: Fine Line
                year: 2019
    nominee2:
        name: Namjoon
        surname: Kim
        nationality: Korean
        records_count: 2
        records:
            record1:
                title: Mono
                year: 2018
            record2:
                title: Indigo
                year: 2022
    nominee3:
        name: Beyonce
        surname: Knowles
        nationality: American
        records_count: 4
        records:
            record1:
                title: Renaissance
                year: 2022
            record2:
                title: Lemonade
                year: 2016
```

Figura 3.7: Ejemplo de YAML con información sobre varios artistas

YAML no sólo se usa para intercambiar datos, también para establecer configuración de software o en herramientas de automatización como pueden ser Kubernetes o

[9] https://yaml.org/

GitHub Actions. En este último por ejemplo se utilizan para configurar flujos de trabajo.

Aunque es bastante menos popular que otros formatos como XML o JSON, entre sus ventajas destacan las siguientes:

- Ligero y sencillo de representar.
- Fácilmente legible por los usuarios.
- Se modifica con facilidad en cualquier editor de texto.
- Apropiado para realizar configuraciones.
- Los lenguajes de programación más importantes lo soportan.

Durante el desarrollo de este TFG hay dos puntos donde se ha necesitado hacer uso de YAML: para configurar la ejecución de los tests en GitHub Actions ante el evento de desplegar a producción y también para procesar la respuesta devuelta por las pruebas de integración sobre la posición del robot.

3.8. OpenCV

OpenCV[10] (Open Source Computar Vision Library) es una biblioteca de código abierto que contiene una serie de algoritmos y funciones para que los programadores tengan la posibilidad de procesar imágenes y vídeos eficientemente.

Se usa también para una gran variedad de aplicaciones, como pueden ser: visión artificial, robótica, reconocimiento de objetos e incluso en realidad aumentada.

Figura 3.8: Logo de OpenCV

Nace en el año 1999 y aunque originalmente estaba escrito en C++, también puede usarse actualmente en otros lenguajes como Java o Python. Con el paso del tiempo se ha popularizado su uso hasta convertirse, actualmente, en una de las bibliotecas más

[10]https://opencv.org/

populares en cuanto a visión artificial se refiere.

Entre las funcionalidades que ofrece, se encuentran:

- **Procesamiento de imágen 2D estáticas**: Con esta biblioteca es posible cargar y manipular imágenes: ponerles filtros, detectar los bordes, ajustar brillo y contraste, manejar sus píxeles directamente entre otras muchas funciones.
- **Reconocimiento de objetos**: Ofrece algoritmos para detectar y reconocer objetos en imágenes. De entre ellos se puede destacar el reconocimiento de caras, detección de bordes o el clasificar objetos usando aprendizaje automático.
- **Seguir movimiento**: Permite seguir objetos basándose por ejemplo en el flujo óptico o mediante ciertas características de dicho objeto.
- **Reconstrucción tridimensional**: La biblioteca OpenCV tiene algoritmos que pueden realizar una reconstrucción tridimensional de objetos a partir de imágenes, también tiene funciones para calibrar una cámara.

Al tratarse de una biblioteca gratuita y de código abierto, cuenta con una gran cantidad de usuarios. Y en esta característica se basa precisamente una de sus mayores ventajas: Al tener una amplia comunidad de usuarios y desarrolladores, la biblioteca se actualiza con regularidad y está en constante desarrollo, ya que estos desarrollan mejoras, ponen a disposición de otros nuevas funcionalidades, tutoriales y ejemplos.

El nuevo ejercicio de procesamiento de imagen se ha desarrollado con la versión 4.5.4 de OpenCV.

3.9. Unibotics

Unibotics[11], «A ROS based openweb platform for intelligent Robotics Education» (Unibotics web, https://unibotics.org/), es una plataforma online educativa principalmente sobre robótica. En ella estudiantes, profesores y curiosos pueden encontrar un amplio catálogo de ejercicios y documentación que les ayude a aumentar, reforzar o asentar sus conocimientos sobre robótica, inteligencia artificial e incluso procesamiento de imagen.

Los usuarios pueden acceder a todo este contenido de forma gratuita y sin que sea necesario realizar complejas instalaciones para poder utilizar la plataforma. Todo lo que deben hacer es descargar la imagen RADI y ejecutarla en un contenedor Docker. Este contenedor se conectará a través de WebSockets al navegador para atender las peticiones que el usuario realice. Para tener una idea de la arquitectura de la página antes de explicarla, se recomienda mirar la Figura 3.9.

[11] https://unibotics.org/

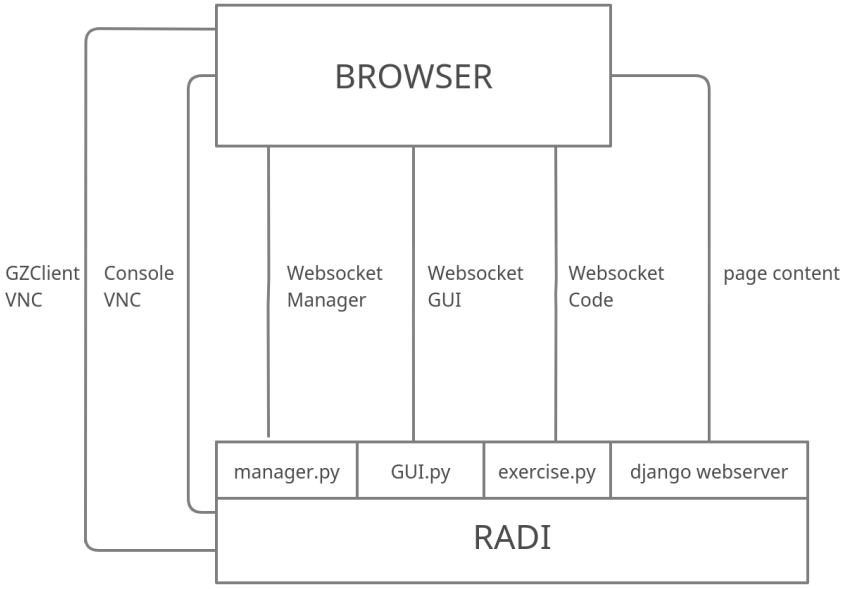

Figura 3.9: Arquitectura de Unibotics (Fuente)

Como primer paso, el usuario debe entrar en la página web https://unibotics.org/ y crear su usuario o hacer log in. Esto le redirigirá a una parrilla donde se muestran todos los ejercicios disponibles. Cada uno de ellos tiene una breve explicación del mismo acompañado de una pequeña imagen a modo de avance (ver Figura 3.10).

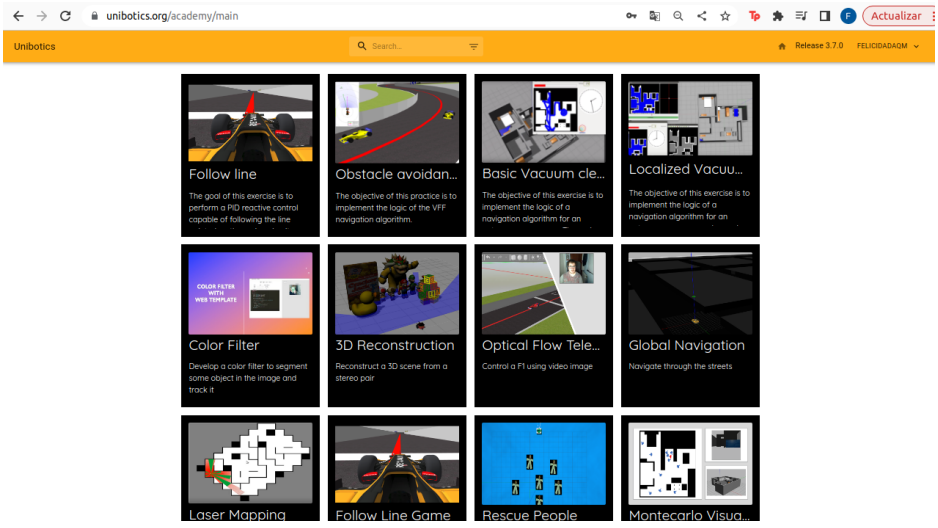

Figura 3.10: Parrilla de ejercicios de la página web Unibotics

Una vez el ejercicio ha sido seleccionado, la página que se muestra al usuario cambia por la de dicho ejercicio concreto. En ella puede encontrarse todo lo necesario para programar el robot de modo que resuelva la tarea concreta de ese ejercicio, para ejecutar ese programa y todos los elementos con los que el usuario debe interactuar, como el

editor de código, los botones de cargar en el robot, pausa, reseteo, etc, sino también lo necesario para comprender la teoría detrás de dicho ejercicio (ver Figura 3.11)

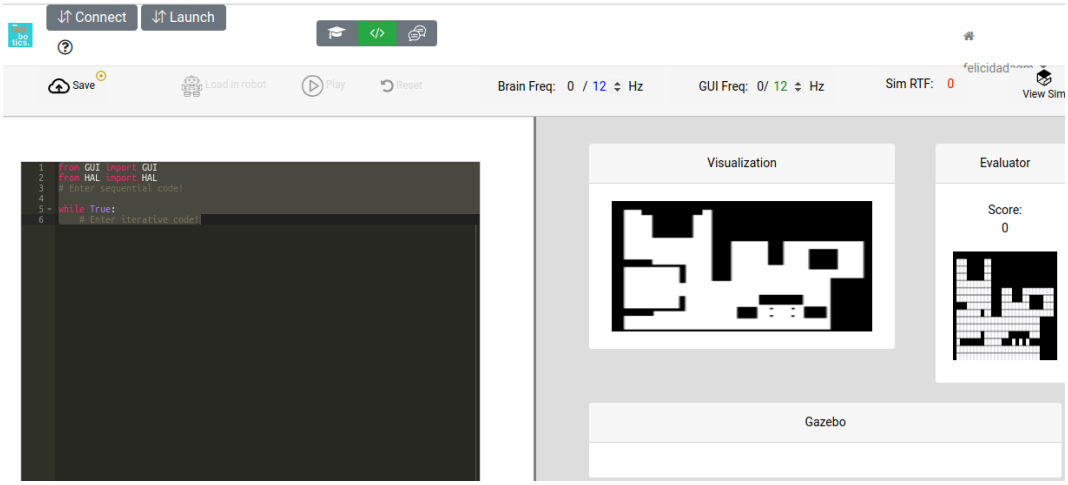

Figura 3.11: Página ejercicio Vacuum Cleaner

Primero, cuando se lanza el contenedor, comienza la ejecución del *manager.py*, que queda escuchando las peticiones del usuario. Segundo, una vez este selecciona un ejercicio y desea "lanzarlo" pulsando el botón "launch", el WebSocket envía el identificador de dicho ejercicio y gracias a él, el manager consigue las instrucciones necesarias para poder lanzar el simulador con el mundo de dicho ejercicio.

Segundo, después de esto, el usuario puede empezar a programar sobre el editor. Cuando desee probar su código sólo debe cargar el código en el robot (en caso de ser necesario) y pulsar el botón de Play. Además, en caso de que necesite algo más de contexto sobre los conocimientos de dicho ejercicio o una pista sobre cómo resolverlo, puede clickar el botón que le redirige a la documentación de usuario (ver Figura 3.12).

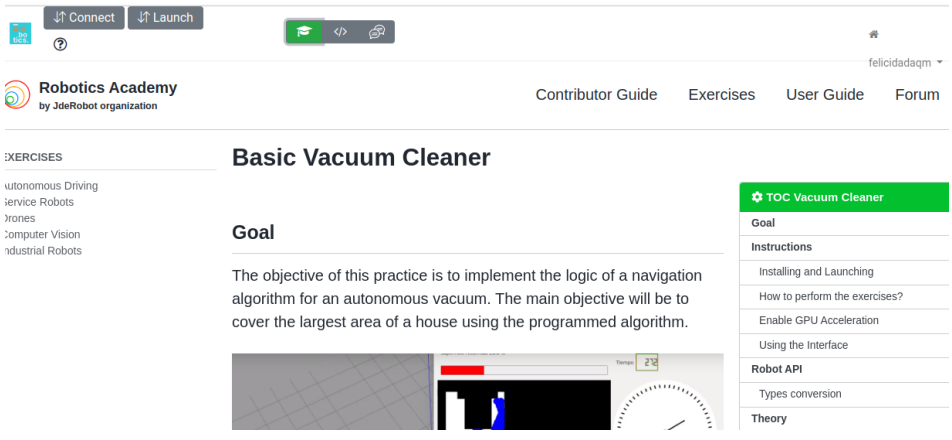

Figura 3.12: Teoría correspondiente a un ejercicio

De cara a los desarrolladores, Unibotics cuenta con 3 tipos de despliegue, que son los siguientes:

- **Despliegue de desarrolladores D1**: Se trata de un despliegue en local en la máquina del desarrollador, tanto el servidor web como la base de datos dummy se encuentran en esta misma.
- **Despliegue test, D2**: Este entorno es parecido al de producción, siendo su objetivo poder probar y detectar fallos antes de que suban en un entorno lo más parecido posible al final. En este caso, el servidor web y la base de datos se ejecutan en los ordenadores de los laboratorios de la URJC.
- **Despliegue de producción, D3**: El último despliegue es realizado por alguno de los desarrolladores con más veteranía en el proyecto. Los cambios del repositorio se fusionan con la rama master y tanto el servidor como la base de datos se ejecutan en un servidor AWS en Irlanda. Uno de los objetivos de este TFG es contribuir a la robustez de la plataforma web mediante la implementación de pruebas automáticas con Selenium que se realizan en el proceso de despliegue D3.

Capítulo 4

Automatización de pruebas en Unibotics con Selenium

Este capítulo recopila todo el desarrollo y el proceso de las pruebas automáticas con Selenium para la plataforma Unibotics. Se ha dividido en varias secciones y para poder abordar los diferentes aspectos de las mismas, siendo las primeras referentes a los puntos comunes: definición de los casos de pruebas y sus objetivos, las funcionalidades ofrecidas por Django y Selenium para la realización de estos tests y la configuración que ha sido necesaria crear para el entorno de pruebas. Por último, los dos últimas secciones harán referencia a las partes más específicas de cada tipo de pruebas.

4.1. Caso de uso de un usuario humano.

La plataforma web Unibotics[1] está pensada para estudiantes que tengan el objeto de reforzar o adquirir conocimientos sobre robótica. Se trata de una plataforma web con alta disponibilidad, que permite a los usuarios usarla en el momento que deseen. Para usarla, los usuarios deben acceder a la página principal de Unibotics, localizable por la URL: https://unibotics.org/, o simplemente utilizar un buscador para encontrarla.

En la portada pueden encontrarse las características principales de las páginas (ver Figura 4.1), redes sociales donde localizar la organización JdeRobot (ver Figura 4.2) y los botones para registrarse, iniciar sesión o acceder al foro.

Los usuarios deben registrarse o iniciar sesión en la página web. Una vez identificados, se les redirigirá de forma automática a una la página donde se muestra la parrilla de todos los ejercicios disponibles (ver Figura 3.10). En esta parrilla, se muestra el título de cada ejercicio, una descripción sobre el mismo y un pequeño avance. El usuario seleccionará un ejercicio pulsando sobre el mismo.

[1] https://unibotics.org/

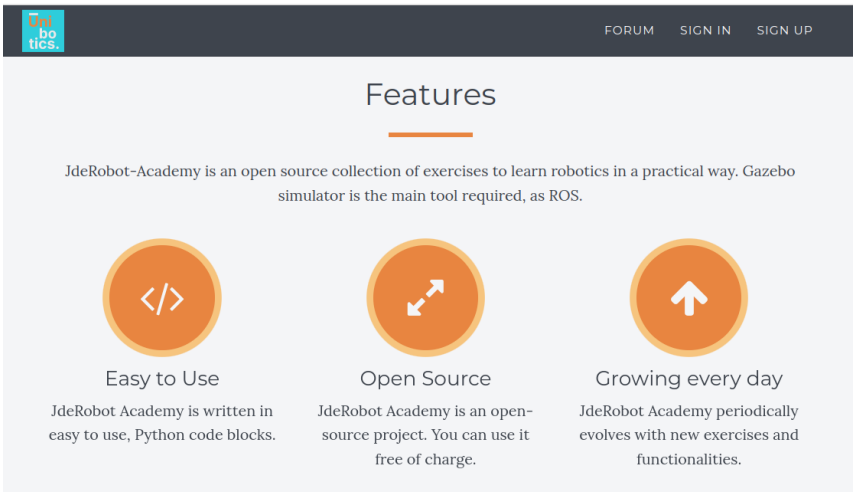

Figura 4.1: Características de la plataforma web en la página de portada.

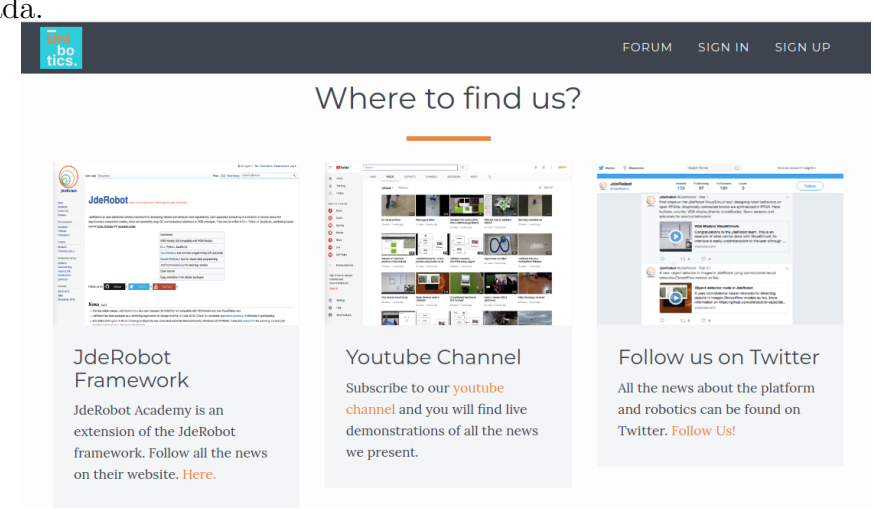

Figura 4.2: Dónde encontrar la plataforma en distintas redes.

Una vez situado en la página web del ejercicio seleccionado, se abre una ventana emergente con las instrucciones a seguir para poder realizar el ejercicio (ver Figura 4.12). Estas instrucciones suelen ser: 1. Descargar la última versión de la imagen RADI, 2. Iniciar con contenedor Docker usando dicha imagen y mapeando los puertos que se indiquen en las instrucciones, 3. Volver a la página del ejercicio, pulsar el botón *"Connect"* y esperar a que se torne verde. Si estos pasos se completan satisfactoriamente, la página ya está conectada con el contenedor RADI. Pueden encontrarse instrucciones más detalladas sobre los requerimientos previos a utilizar un ejercicio en el apartado de documentación de cada ejercicio concreto (ver Figura 4.3)

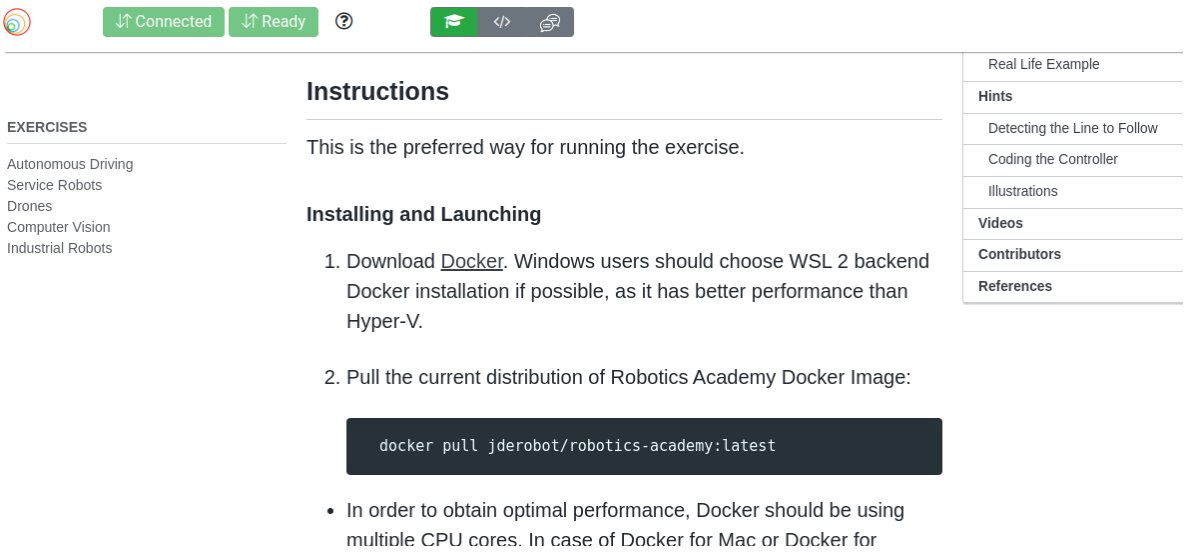

Figura 4.3: Documentación de las instrucciones para usar un ejercicio.

Teniendo comunicación con el contenedor, el usuario humano debe pulsar el botón *"Launch"* para que el contenedor RADI prepare todo lo necesario para usar el ejercicio. Tras esperar unos segundos, el texto de este botón cambiará por *"Ready"* y su color pasará a ser verde. El usuario ya puede empezar a programar el código fuente que será insertado después en el robot simulado.

Cada vez que se desee ejecutar dicho código, se debe pulsar el botón *"load in robot"* y esperar hasta que el desplegable que pide esperar mientras el código carga en el robot desaparezca. Con el código cargado, los botones *"Play"* (que cambia su valor a *"Stop"* mientras se ejecuta el código en el robot simulado) y *"Reset"* se encargan de controlar la simulación. El usuario tiene más acciones disponibles con el resto de botones de la página, como pueden ser: guardar el código, ver la consola donde puede imprimir mensajes de depuración, ver el mundo del robot simulado, teleoperar el robot con las flechas del teclado, realizar un torneo o evaluar la eficacia de su código fuente.

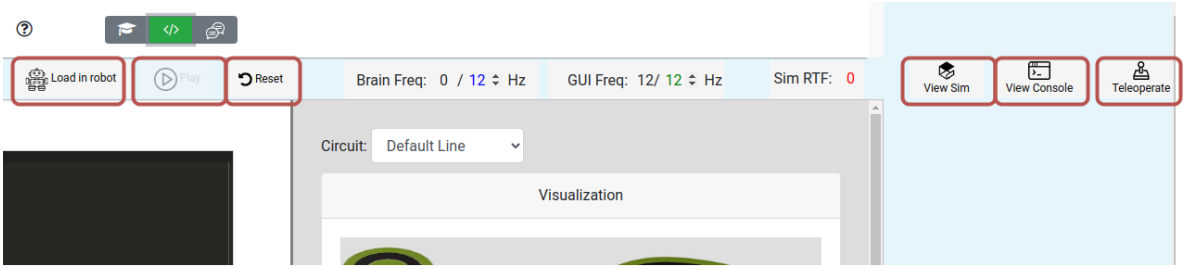

Figura 4.4: Ejemplo de botones existentes en la página web de un ejercicio concreto.

4.2. Definición de los casos de prueba.

Unibotics es una plataforma gratuíta, de código abierto y en crecimiento, por ello el número de desarrolladores que hacen aportes al proyecto crece naturalmente. Estos desarrolladores, y los revisores de código que revisan los parches antes de fusionarlas en la rama master, deben asegurarse de que cada nueva funcionalidad, mejora o arreglo es compatible con aquello que ya está en producción en un proyecto cada vez más grande.

Al crecer la plataforma, también lo hace todo lo que es necesario probar y revisar con cada despliegue, requiriendo una gran inversión de tiempo. Es por ello que surge la necesidad de crear una base de pruebas automatizadas que sean capaces de detectar los fallos que podrían provocar que la plataforma no funcionase correctamente.

Como las pruebas que se quieren implementar son un número considerable, se llega a la conclusión de que lo más eficiente sería dividirlas en pruebas unitarias de frontend y pruebas integrales. Ambos tipos de pruebas tendrán el mismo objetivo, comprobar que todo funciona correctamente, sin embargo se distinguen por el alcance de cada una de ellas

La finalidad de las pruebas unitarias es comprobar el funcionamiento de pequeñas partes del código, como pueden ser funciones o métodos que no tienen dependencias con otras partes del código y pueden probarse de forma aislada. Como prueban pequeñas funcionalidades, suelen ser pruebas rápidas a la hora de ejecutarse.

Por otro lado, las pruebas de integración tienen la finalidad de testear la interacción entre varias partes del sistema, comprueban que estas estén bien implementadas y cohesionadas. Al probar partes más complejas del sistema, no sólo tardan más tiempo en ejecutarse que las pruebas unitarias, sino que también requieren simular entornos más complejos para poder realizarse.

Aunque en la primera versión del desarrollo las pruebas se lancen de forma manual, es requisito indispensable que estas pruebas terminen siendo automatizadas, es decir, que el mismo script se encargue de preparar todo el entorno necesario para ejecutarlas ante un evento, que será el despliegue a producción (D3) de Unibotics. De esta forma, si las pruebas no devuelven un resultado satisfactorio se podrá echar marcha atrás antes de que el error llegue a producción y revisarlo con más calma en entornos previos.

Para el desarrollo de estas pruebas son necesarios, principalmente, los siguientes componentes. La relación entre los mismos se muestra en la figura 4.5 y se explicará con más detalle en las siguientes secciones:

- **Selenium:** Para automatizar las pruebas, se encarga tanto de preparar el entorno de pruebas, como de lanzar los contenedores y pruebas necesarias.
- **Django**: Ofrece funcionalidades compatibles con Selenium para facilitar la implementación de pruebas.

- **GitHub Actions**: Donde se configurará el evento que lanzará todos los *script* de pruebas automatizadas.

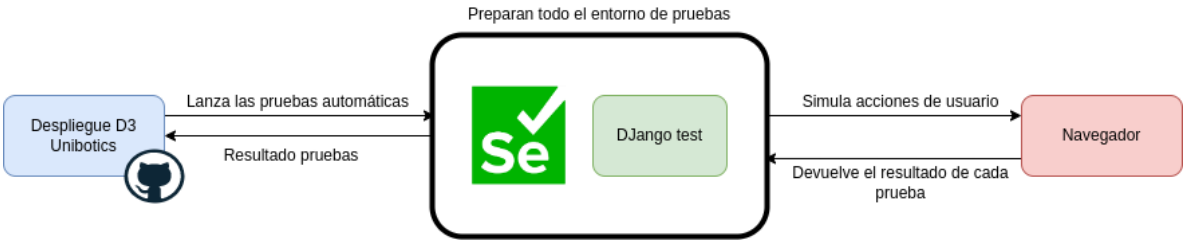

Figura 4.5: Esquema simplificado del flujo de pruebas.

4.3. Módulo Django Test

El propio entorno Django proporciona un módulo que facilita la automatización comentada en la sección anterior, que libera a los desarrolladores de realizar acciones manuales. Este módulo es *django.test*, y está basando en la librería unittest de Python. Aunque el nombre de esta librería haga entender que es una librería exclusiva para pruebas unitarias, lo cierto es que también permite crear pruebas de integración.

El módulo *django.test* proporciona un conjunto de clases y herramientas que ayudan a la escritura y automatización de pruebas de las aplicaciones web. Entre las clases base más usadas, se pueden encontrar:

- **TestCase**: La clase base principal, permite crear y usar subclases de la misma para comprobar que el código se comporta de la forma esperada. Para no contaminar la base de datos real, TestCase crea una base de datos limpia y cada función de prueba tiene su propio "espacio seguro" dentro de esta base de datos, lo que quiere decir que cada función realiza los cambios que necesite en este espacio seguro y al terminar se borra sólo eso, sin afectar a la base de datos completa.

  Se adjunta un ejemplo en el que se realizan dos pruebas: la primera de ella para verificar que la función *addition* devuelve el resultado esperando y la segunda de ellas, se relaciona con la base de datos: crea una entrada con el nombre *FirstEntry*, recupera todos los objetos de la tabla en la variable *queryset*, comprueba la longitud de la misma y por último comprueba que el nombre del primero del array se corresponde con *FirstEntry*.

Listing 4.1: Ejemplo de un caso de prueba con la clase base TestCase.

```
from django.test import TestCase
from myapp.models import MyModel

def addition(num1, num2):
return num1 + num2

class FirstTestCase(TestCase):
    def test_add(self):
        result = addition(2, 2)
        self.assertEqual(result, 4)

        MyModel.objects.create(name="FirstEntry")
        queryset = MyModel.objects.all()
        self.assertEqual(queryset.count(), 1)
        self.assertEqual(queryset[0].name, "FirstEntry")
```

- **SimpleTestCase**: Muy similar a la anterior, con la diferencia de que no tiene la posibilidad de interactuar con una base de datos. Por ello puede ser una clase más rápida que TestCase en casos concretos. Se adjunta un ejemplo de un test simple creado con esta clase SimpleTestCase que comprueba que la función *addition* devuelve el resultado esperado.

Listing 4.2: Ejemplo de un caso de prueba con la clase base SimpleTestCase.

```
from django.test import SimpleTestCase

def addition(num1, num2):
    return num1 + num2

class FirstSimpleTestCase(SimpleTestCase):
    def test_add(self):
        result = MathUtils.add(2, 2)
        self.assertEqual(result, 4)
```

- **TransactionTestCase**: Esta clase también se parece a TestCase, ambas son capaces de interactuar con la base de datos, pero con una diferencia. Mientras que TestCase crea una base de datos y pequeños espacios para cada función de prueba, TransactionTestCase crea y destruye la base de datos para cada clase. Es decir, crea una base de datos limpia, ejecuta todo el conjunto de pruebas que tenga esa clase, y borra la base de datos antes de pasar a la siguiente. Es por ello que hace que las pruebas se ejecuten en un entorno aún más aislado, a cambio de una penalización en tiempo.

Listing 4.3: Ejemplo de un caso de prueba con la clase base TransactionTestCase.

```
from django.test import TransactionTestCase
from myapp.models import MyModel

def addition(num1, num2):
    return num1 + num2

class MyTransactionTestCase(TransactionTestCase):
    def test_add(self):
        result = MathUtils.add(2, 2)
        self.assertEqual(result, 4)

    def test_database(self):
        MyModel.objects.create(name="FirstEntry")
        queryset = MyModel.objects.all()
        self.assertEqual(queryset.count(), 1)
        self.assertEqual(queryset[0].name, "FirstEntry")
```

- **LiveServerTestCase**: Se utiliza para realizar pruebas de integración que no sólo involucran base de datos, sino también un servidor web. Cuando se usa esta clase, Django inicia un servidor web en un puerto libre y aleatorio de la máquina.

  Se adjunta un ejemplo de código para esta clase base, donde se carga la página web que ha iniciado previamente Django. Una vez en la página web, se busca el elemento *name-input*, se escribe un nombre y a continuación se busca y clicka el botón para enviar este datos.

Listing 4.4: Ejemplo de un caso de prueba con la clase base LiveServerTestCase.

```
from django.test import LiveServerTestCase
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import
   expected_conditions as EC

class LiveServerFirstTest(LiveServerTestCase):
    def setUp(self):
        self.selenium = webdriver.Firefox()

    def tearDown(self):
        self.selenium.quit()

    def test_send_name(self):
        self.selenium.get(self.live_server_url)

```

```
17              input_element =
                    self.selenium.find_element(By.ID,
                    'name-input')
18              input_element.send_keys('Juan')
19
20              submit_button =
                    self.selenium.find_element(By.ID,
                    'submit-button')
21              submit_button.click()
```

De entras las cuatro clases base presentadas, la que más encaja con la idea de pruebas automátizas en una página web es LiveServerTestCase. Por ello es la que se eligió para empezar a desarrollar las pruebas automáticas. Esta clase base proporciona distintas formas de interactuar con el servidor, entre las que han resultado más útiles para el desarrollo de este TFG se encuentran:

- **Crear y destruir el servidor:** Se encarga de iniciar el servidor antes de realizar las pruebas, y apagarlo al finalizarlas.
- **Proporciona la URL del servidor web:** Se obtiene la URL de la web de la variable *live_server_url*, por lo que ya no es necesario preocuparse por cómo pasársela al script de testing.
- **Aserciones:** Incluye funciones para comprobar las respuestas del servidor, como *self.assertContains()* que comprueba si una cadena determinada se encuentra en la respuesta, o *self.assertEquals()*, que comprueba si la respuesta es igual a un valor.
- **Base de datos de prueba:** Crea una base de datos de prueba temporal al ejecutarse y la destruye al terminar.

Aunque las características citadas anteriormente han sido muy útiles de cara a la automatización de las pruebas, no son suficientes para cubrir todos los requisitos que se plantearon. La finalidad es crear pruebas que sean capaces de simular acciones de un usuario humano, ya que de esa forma se podrán realizar comprobaciones en profundidad siguiendo el proceso que un usuario haría al utilizar la página web. Es por ello, que además de usar las funcionalidades que ofrece Django test, ha sido necesario complementarlas con Selenium gracias a la flexibilidad que esta herramientas ofrece a la hora de personalizar pruebas.

4.4. Configuración del entorno de pruebas

Para que las pruebas sean capaces de ejecutarse de forma automática, es importante ocuparse antes de preparar toda la configuración necesaria. En esta sección se explicarán estas configuraciones que han sido necesarias, tanto para automatizar Selenium como las referentes a las páginas web de Unibotics.

Primero se han realizado las configuraciones relacionadas con Selenium. Para que Selenium pueda interactuar con un navegador, es necesario tener instalados los controladores del mismo. La finalidad de los controladores es traducir las instrucciones de Selenium a acciones para realizar las mismas en el navegador.

Se han instalado controladores para los dos principales navegadores: ChromeDriver en su versión 100.0.4896.60 para Google Chrome y la versión GeckoDriver 0.31.0 para Firefox. Es importante consultar la versión del navegador primero y descargar una versión de controlador que sea compatible con el mismo.

Una vez descargados, es necesario añadirlos al PATH del sistema. También existe la opción de proporcionar la ruta completa a este controlador en el código, pero esto haría que sólo funcionase en una máquina que tuviese el controlador en la misma ruta que la máquina local en la que se ha desarrollado el código. Para evitar esto, y que pueda ejecutarse en la máquina proporcionada por GitHub Actions, se ha optado por la primera opción.

Para indicarle al sistema operativo dónde se encuentran los controladores se añade al archivo de configuración .bashrc la línea: export *PATH="$PATH:ruta-controlador"* y se ejecuta el comando *"source /.bashrc"* en la shell para cargar los cambios.

```
export PATH="$PATH:/home/felicidad/Documentos/geckodriver-v0.31.0-linux64"
source /opt/ros/noetic/setup.bash
```

Figura 4.6: Archivo de configuración .bashrc con la ruta del driver de Firefox.

Una vez que se han realizado los cambios en el PATH, Selenium podrá localizar y utilizar los controladores correspondientes. Para que esto ocurra, también es necesario añadir la configuración adecuada en el script de Selenium, de forma que pueda escoger el controlador en función del navegador en el que se realizarán las pruebas.

Esta configuración se realiza mediante dos métodos que utiliza la clase LiveServerTestCase. Estos métodos son los siguientes:

- **setUp( ):** La finalidad de este método es preparar el entorno de pruebas y configurar todo lo necesario antes de realizar las pruebas. Las principales tareas de las que se encarga son: configura el navegador; inicializa el controlador correspondiente, y prepara el entorno de prueba; carga de la página web, configurar variables, crear usuarios, etc. Hay que destacar que es un método que debe existir

en cada bloque de prueba y que no es necesario invocarlo de forma explícita, pues se ejecutará solo.

- **tearDown( ):** Este método realiza tareas de limpieza cuando ha terminado la ejecución de cada bloque de pruebas, con el objetivo de restaurar el estado inicial del sistema. Entre sus tareas principales están: libera recursos; cierra el navegador web, y libera la memoria asociada, restaura el estado inicial; si se han modificado datos en una base de datos, este método los devuelve a su estado inicial y cerrar conexiones/sesiones: cierra las conexiones con base de datos o los inicios de sesión que se hayan hecho durante las pruebas. Al igual que *setUp()*, *tearDown()* es un método que debe existir por cada bloque de pruebas y que no es necesario invocar.

La configuración que se ha realizado en el método *setUp()* para Selenium, es la siguiente:

Listing 4.5: Configuración para Selenium del método setup.

```
1     def setUp(self):
2         opts = Options()
3         #opts.add_argument(env.str("SELENIUM_DISPLAY",
              "--headless"))
4         if env.str("SELENIUM_BROWSER", "firefox") == "chrome":
5             self.selenium = webdriver.Chrome(options=opts)
6         else:
7             self.selenium = webdriver.Firefox(options=opts)
```

Primero se crea una instancia del objeto Options, donde se guardarán las opciones seleccionadas para las pruebas. Después se puede ver una línea comentada que añade la variable *"SELENIUM_DISPLAY"* dentro del objeto *opts* con el valor *"–headless"*. Esta variable indica si las pruebas se ejecutarán con interfaz gráfica donde el *tester* humano pueda seguir las acciones que se realizan visualmente, o no. Si el valor es *-–headless*, quiere decir que no habrá interfaz gráfica.

Se llega a un if que comprueba el valor de la variable *"SELENIUM_BROWSER"* en la que se encontrará el nombre del navegador sobre el que se van a realizar las pruebas. Si este valor es *çhrome"*, se selecciona el controlador de Chrome. Por el contrario y por defecto, siempre se usará Firefox.

Con esto queda terminada la configuración referente a Selenium. Pero aún queda pendiente la configuración necesaria para poder usar las páginas web de Unibotics de forma automática.

Como se ha visto en la sección anterior, la clase *LiveServerTestCase* crea una base de datos limpia, eso significa que no tendrá ningún usuario o ejercicio almacenado y es necesario crearlos. Para cumplir este objetivo, se crea un archivo llamado *operations.py*.

Listing 4.6: Archivo operations.py.

```
1   from academy.models import User, Exercise
2   import json
3
4   def create_user(username="testing_user", password="12345",
        role="Betatester", docker_ip="127.0.0.1"):
5       test_user = User.objects.create_user(username=username,
6                                            password=password,
7                                            role=role,
8                                            docker_ip=docker_ip)
9       test_user.save()
10      return test_user
11
12
13  def create_exercise(exercise_id="follow_line",
14                      name="Follow␣line",
15                      description="Achieve␣the␣fastest␣lap␣in␣
                            Nurburgring",
16                      assets=json.dumps({"notebook": ""})):
17      test_exercise =
            Exercise.objects.create(exercise_id=exercise_id,
            name=name, description=description, assets=assets)
18      test_exercise.save()
19      return test_exercise
```

Para ser capaces de crear un usuario y el ejercicio, es necesario conocer los modelos que se importan en el archivo *operations.py*. Estas clases son *User* y *Exercise* (ver Figura 4.7) y muestran los argumentos necesarios para crear una instancia de cada uno de ellos.

```
33  class Exercise(models.Model):
34      exercise_id = models.CharField(max_length=40, blank=False, unique=True)
35      name = models.CharField(max_length=40, blank=False, unique=True)
36      description = models.CharField(max_length=400, blank=False)
37      assets = models.CharField(max_length=2000, default=json.dumps({"notebook": ""}))
38
39      def __str__(self):
40          """Representation of the name."""
41          return self.name
42
43
44  class User(AbstractUser):
45      """
46      This is the model used to store the users' data.
47      """
48      ROLES = (
49          ('admin', 'Admin'),
50          ('Betatester', 'Betatester'),
51          ('profesor', 'Profesor'),
52          ('alumno', 'Alumno')
53      )
54      role = models.CharField(max_length=40, choices=ROLES, blank=True)
55      docker_ip = models.CharField(max_length=40, blank=True, help_text="The current machine the user is using.")
56      institution = models.CharField(max_length=40, blank=True, help_text="The institution you belong")
57      online_docker = models.BooleanField(default=False,
58                                          help_text="If True, everytime the user starts a simulation the webserver will connect the user to a machine of the farm.")
59      active_exercise = models.CharField(max_length=40, blank=True, help_text="The current exercise the user is in.")
60
```

Figura 4.7: Definición de las clases User y Exercise.

Una vez conocidas estas clases, los argumentos que cada una de ellas necesita y el tipo de cada uno de estos, se crean dos funciones que se invocarán posteriormente desde el archivo de Selenium que ejecuta las pruebas. Aunque el nombre de estas funciones permite entender bastante sobre su objetivo (*create_user* y *create_exercise*), se explicarán brevemente:

- **create_user**: Hace uso del modelo User para crear un usuario nuevo. A este modelo se le pasan cuatro parámetros: *Username, password, role y docker_ip*. En caso de no hacerle llegar alguno de ellos al invocarlo, se pondrá el valor que aparece por defecto en la definión de la función. *Role* podría considerarse el parámetro más importante de esta función, pues identifica los permisos que tendrá el usuario, en este caso, se le dan los de *Betatester* pues no necesita acceder a otras funcionalidades como las que puede tener el rol Admin.

- **create_exercise**: Esta función hace uso del modelo Exercise para crear un ejercicio. Se le pasan también cuatro parámetros que en este caso son: *exercise_id, name, description y assets*. Tiene el mismo comportamiento que la función *create_user* y en caso de no pasarle alguno de los cuatro parámetros se pondrá el valor que aparece por defecto en la definición de la función. En este caso, es importante que los parámetros exercise_id y name existan y tengan el valor correcto por cada bloque de pruebas, ya que estos parámetros son a partir de los cuales se crea la entrada del ejercicio en base de datos y es este mismo que se cree el que aparecerá en la página por cada bloque de pruebas.

Una vez probadas estas funciones, se importan en el archivo principal de Selenium y se usan dentro del método setUp. El método completo, para un ejemplo del bloque de pruebas del ejercicio *Follow Line*, queda como se muestra en el siguiente cuadro de código:

Listing 4.7: Archivo operations.py.

```
1  def setUp(self):
2          opts = Options()
3          #opts.add_argument(env.str("SELENIUM_DISPLAY",
               "--headless"))
4          if env.str("SELENIUM_BROWSER", "firefox") == "chrome":
5              self.selenium = webdriver.Chrome(options=opts)
6          else:
7              self.selenium = webdriver.Firefox(options=opts)
8          self.password = "test1234"
9          self.user = create_user(username="testing_user",
               password=self.password)
10         self.exercise =
               create_exercise(exercise_id="follow_line",
               name="Follow␣Line")
```

Con la función setUp completa, queda finalizada la configuración del entorno de pruebas. A partir de ahora, sólo es necesario lanzar las pruebas desde el terminal de la máquina para que se realicen.

4.5. Pruebas Unitarias de frontend

Tras haber introducido las partes generales en el desarrollo de pruebas automáticas, esta sección profundiza en aquellas cuestiones específicas de las pruebas unitarias de frontend.

Al tratarse de pruebas más directas e involucrar menos piezas en el desarrollo de las mismas, estas pruebas fueron las primeras que se empezaron a diseñar y desarrollar, siendo sus objetivos los siguientes:

- Comprobar que la parrilla sirve todos los ejercicios disponibles de programación de robots en la plataforma Unibotics.
- Comprobar que en la página del ejercicio, existen, son clickables y funcionan todos los botones básicos además de los específicos que pueda tener cada uno de los ejercicios.

Realizar las comprobaciones anteriormente descritas implica que por cada ejercico haya unos 9 casos de prueba. Para no aumentar demasiado el tiempo de ejecución, se decide que estas pruebas se realizarán por bloqueas, conteniendo cada bloque todas las pruebas de un ejercicio concreto. De esta forma, se consiguen mejoras, pues no será necesario tener que borrar partes de la base de datos con cada prueba de todos los ejercicios, sino cada vez que se inicie el bloque de pruebas de un ejercicio nuevo.

4.5.1. Acceder a la portada de Unibotics e Inicio de sesión

En la sección 4.3 se explica cómo Django almacena la url de la página web que levanta dentro de la variable *live_server_url*, gracias a ello sólo es necesario utilizar primeramente el método get que utiliza Selenium para abrir el navegador y dirigirse a dicha página.

A partir de ahora, es importante conocer una de las herramientas de desarrollador del navegador para ayudar a localizar los elementos que se quieren probar. Pulsando la tecla F12 o botón derecho a inspeccionar, es posible ver el HTML de la página en que se navega actualmente. Además, al pasar el ratón sobre una sección de dicho HTML se muestra en la página el elemento al que se refiere con un recuadro (ver Figura 4.8). Se ha utilizado esta funcionalidad como ayuda para poder ubicar los elementos con los que Selenium debe interactuar a lo largo del desarrollo de las pruebas.

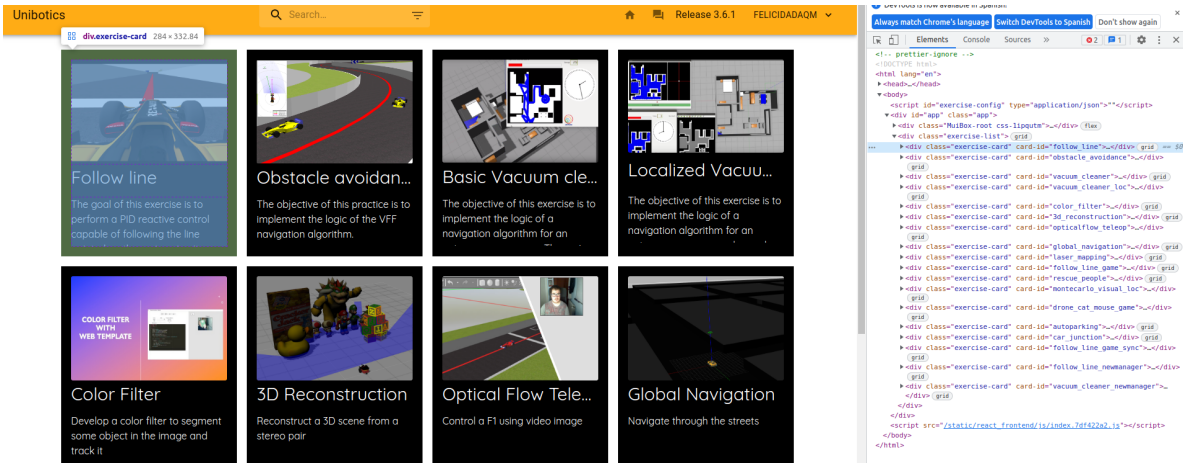

Figura 4.8: Herramienta desarrollador en navegador Google Chrome.

Tras aterrizar en la página de portada de Unibotics, es necesario iniciar sesión con el usuario y contraseña que se han establecido anteriormente con el método *setUp()*. Para conseguir llegar hasta la página de inicio de sesión, Selenium localiza por XPath el elemento '//a[@href=/academy/login"]', que se trata del enlace hacia la página de log in y pulsa sobre él.

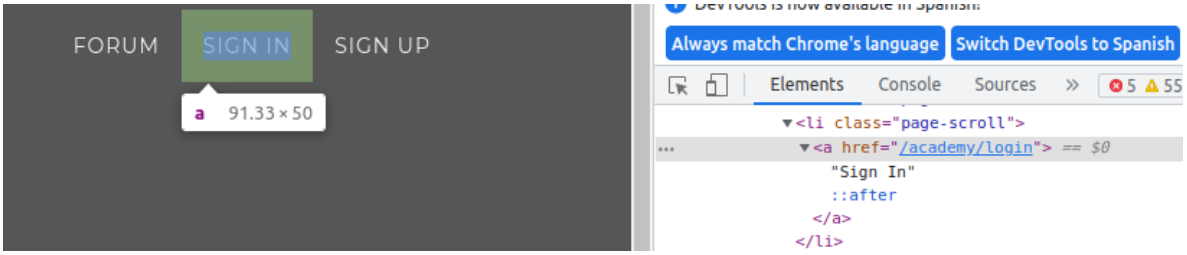

Figura 4.9: Botón inicio de sesión en Unibotics.

Una vez estando en la página de inicio de sesión, se localizan los elementos *username* y *password* para introducir los mismos valores que el del usuario creado con el método setUp (ver Figura 4.10). Estos dos cuadros de texto se localizan por su id, pues el valor de esta etiqueta debe ser único para cada elemento dentro del HTML. Para introducir el valor de las variables *username* y *password* en un *input* de texto, Selenium ofrece la función *send_keys*. Por último, se busca el botón de log in y se pulsa.

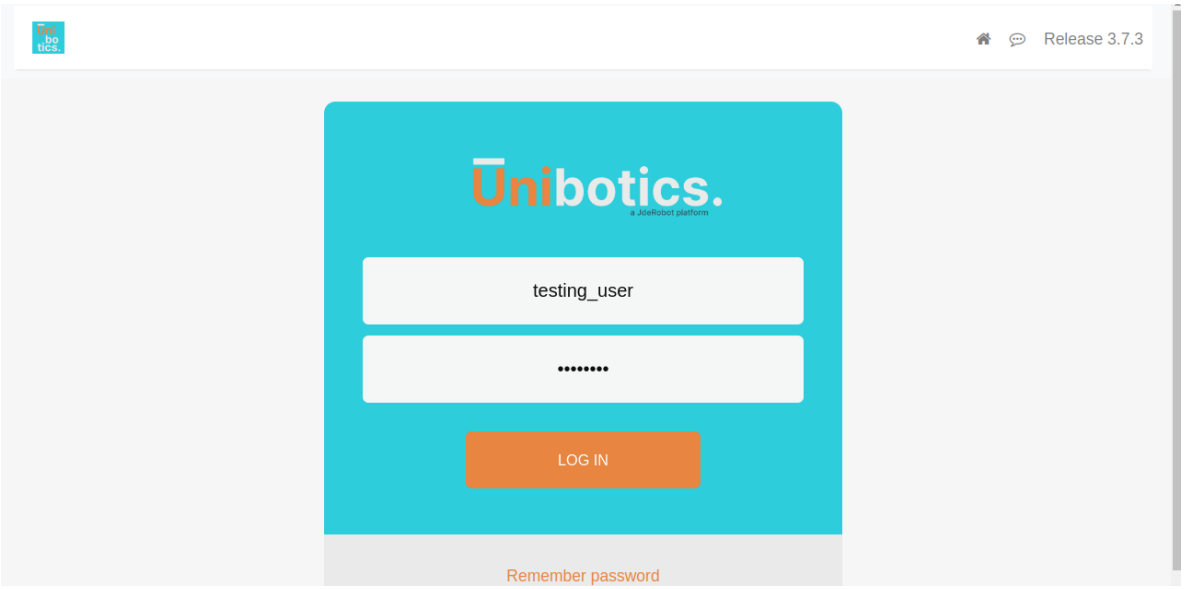

Figura 4.10: Introducción usuario y contraseña Unibotics.

4.5.2. Seleccionar ejercicio concreto.

Tras iniciar sesión, la página automáticamente se redirige a la parrilla de ejercicios. Como ya se ha visto en la sección 4.3, para el entorno de pruebas se crea una base de datos vacía, por lo que no habrá ningún ejercicio más que el que se dé de alta para cada bloque de pruebas. Si durante la ejecución de las pruebas la interfaz gráfica está activada (ver sección 4.4), podrá observarse este único ejercicio en la parrilla (ver Figura 4.11).

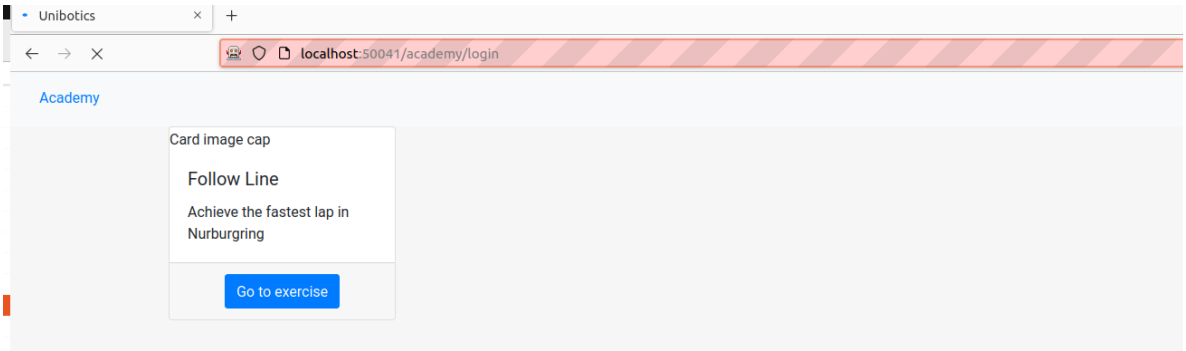

Figura 4.11: Parrilla de Unibotics con un único ejercicio.

Tras localizar el botón para entrar al ejercicio, se abrirá una ventana emergente (ver Figura 4.12). En esta ventana el usuario puede encontrar información sobre los pasos que necesita seguir para realizar el ejercicio.

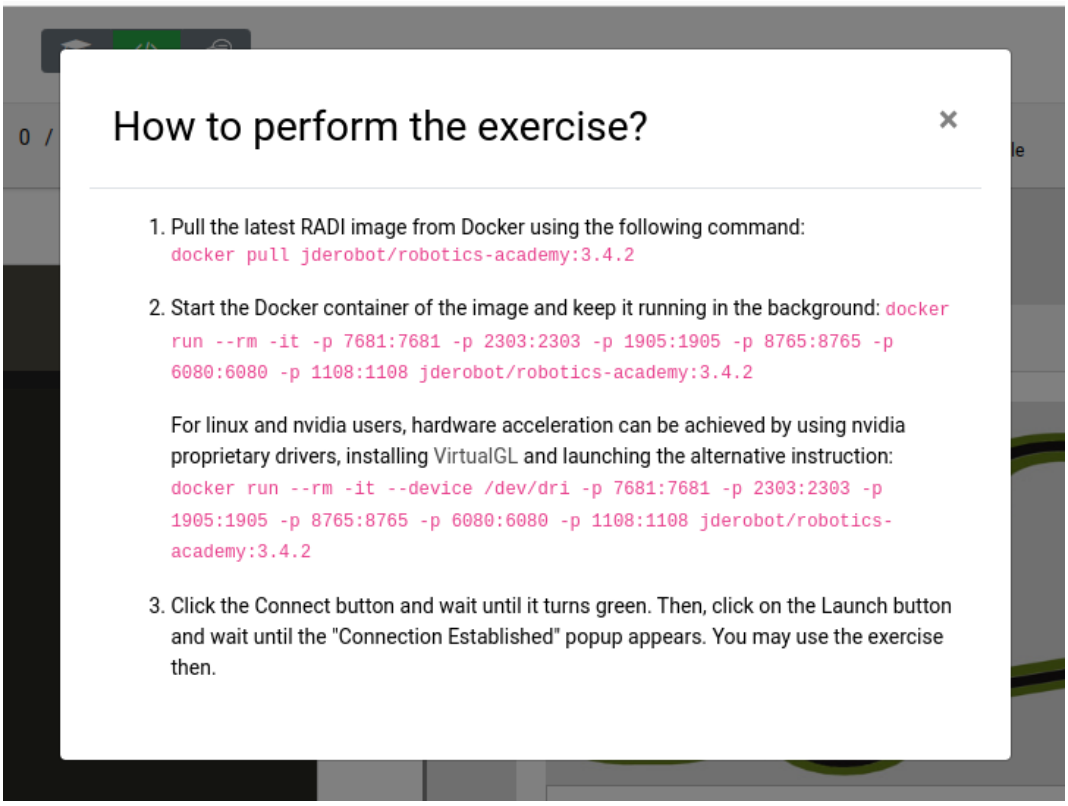

Figura 4.12: Ventana emergente al entrar a un ejercicio con instrucciones de lanzado del contenedor docker

Surgen dos puntos importantes y dos problemas a partir de esta ventana emergente. El primero de ellos es que es necesario esperar a que este elemento aparezca antes de proseguir con la ejecución de pruebas, pues oscurece el resto de la página web y hace que Selenium no sea capaz de encontrar elementos bajo este, que son los que realmente le interesan.

El segundo problema es que esta ventana emergente carece de algún identificador que ayude a localizarlo e interactuar con él por lo que, a pesar de que tenía una X que la cerraba si se pulsaba, no era posible localizarla con los métodos anteriores. Esta ventana emergente también se cerraba en caso de hacer click sobre una parte de la pantalla donde no se encontrase, por lo que se tomó esta forma como último recurso para poder cerrarla mientras no exista un identificador para este elemento.

4.5.3. Realización de las pruebas unitarias

Hasta este punto, se han realizado todos los pasos necesarios para llegar hasta la página web específica del ejercicio de forma automática. Una vez estando en esta página, queda desarrollar los casos de prueba para cada uno. Para ello, ha sido necesario una revisión previa que permita conocer los botones que cada ejercicio tiene y poder crear los casos de prueba necesarios para cada uno.

A cotinuación, se detalla una lista con las pruebas que se han añadido y lo que cada una de ellas verifica:

- **test_exercise_launch:** La función del botón launch es pasarle al contenedor RADI la orden *"open"* para que este, junto con el identificador del ejercicio, pueda preparar y levantar todo lo necesario para que el usuario pueda usar el ejercicio a continuación. Como en estas pruebas unitarias de frontend no se está validando la funcionalidad de la página sino que el frontend tenga las propiedades correctas, sean visibles, etc, sólo se comprueba que el botón existe y puede pulsarse.

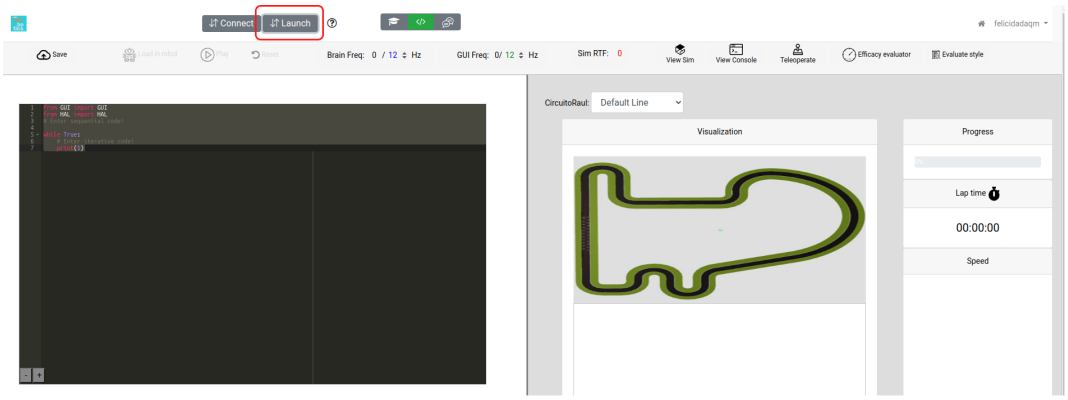

Figura 4.13: Botón Launch en la página de un ejercicio.

- **test_exercise_theory:** Al pulsar en este botón, la visualización de la página cambia, sin modificar el HTML o cambiar de url, y pasa a mostrar la página de la documentación de dicho ejercicio (ver Figura 4.10). Este botón se localiza por su id *open-theory*, cuyo valor es único en todo el HTML de la página. Se han programado acciones de Selenium que comprueben que este botón no sólo existe, sino que es posible pulsarlo.

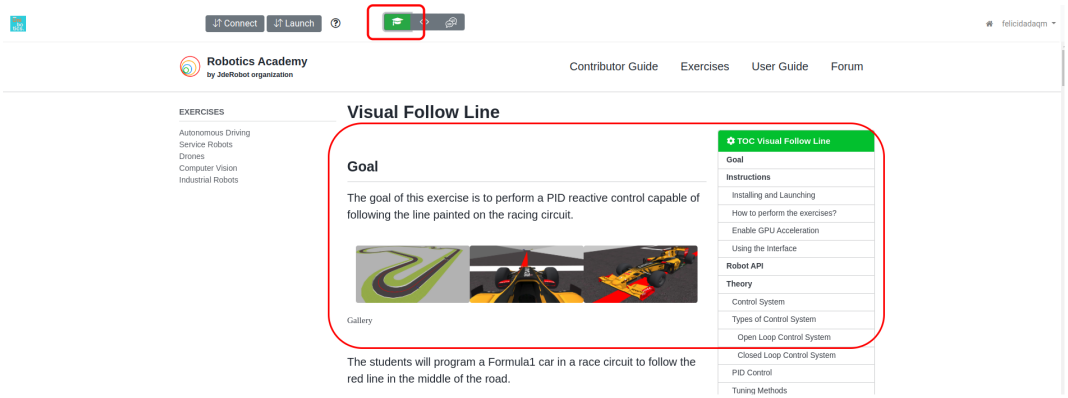

Figura 4.14: Visualización de documentación del ejercicio.

- **test_exercise_forum:** Este botón se identifica por el id *open-forum* y es similar al anterior, con la diferencia de que en lugar de mostrar la documentación del ejercicio, muestra el foro de Unibotics. En este foro, aparte de anuncios de los desarrolladores de la plataforma donde se dan noticias sobre el mantenimiento, o la página en si, los usuarios pueden crear posts con errores que han encontrado o sus dudas (ver Figura 4.11)

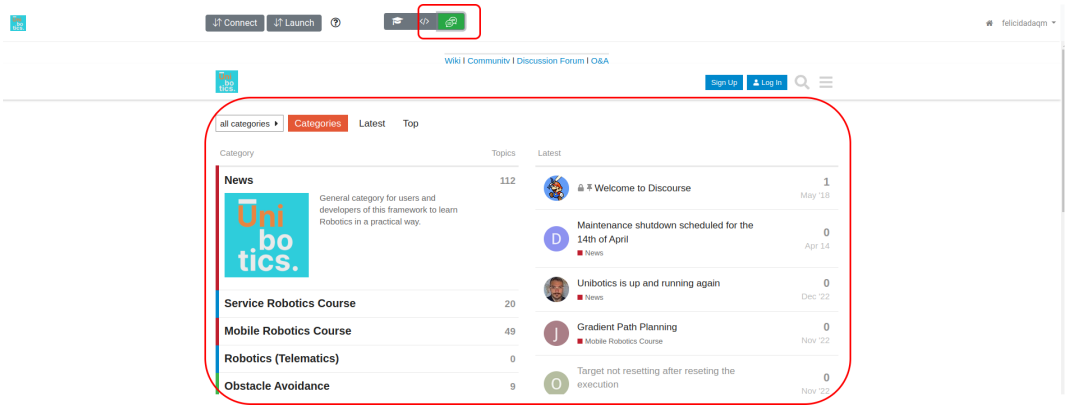

Figura 4.15: Visualización del foro de Unibotics.

- **test_exercise_exercise:** Al cargar la página de un ejercicio, por defecto, se encuentra en esta vista. En ella pueden encontrarse dos divisiones de la página: a la izquierda el editor, y a la derecha la visualización del robot, la consola, etc (ver Figura 4.12). En este *test*, se quiere probar que es posible volver a esta vista y que el botón existe y funciona. Por ello, como primer paso, el test automático pasa a la vista del foro y después intenta regresar de nuevo al ejercicio.

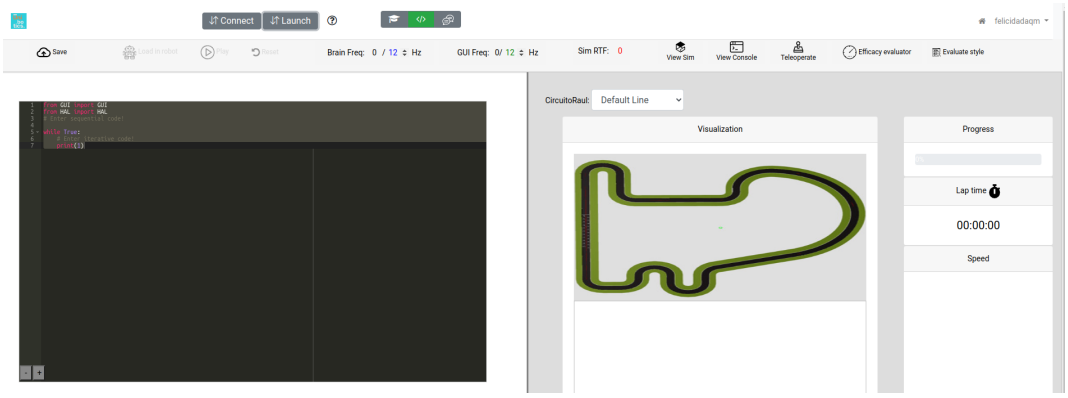

Figura 4.16: Visualización del ejercicio.

- **test_exercise_console:** En la división derecha de la pantalla, debajo de la visualización, se ofrece una consola al usuario en la que se pintan los mensajes de depuración. Esta consola se consigue utilizando el protocolo VNC, que hace posible la visualización y control de un escritorio de un ordenador desde otro. En este caso, la consola que se muestra es la del contenedor RADI. El identificador de este botón es *"console"*.

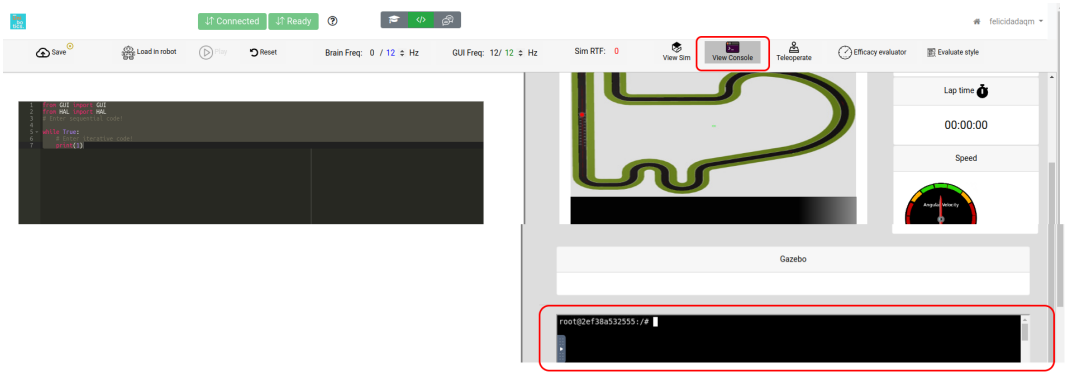

Figura 4.17: Consola en el ejercicio.

- **test_exercise_save:** Es muy probable que algunos usuarios no tengan tiempo para terminar un ejercicio en el momento en que empiezan a hacerlo, o que a pesar de terminarlo deseen retomarlo después para mejorar su código. Por eso se ofrece la funcionalidad de salvar el código de este usuario para que cuando vuelva a entrar en el ejercicio aparezca directamente este código guardado siempre que se use el mismo usuario. El identificador de este botón es *"save"*.

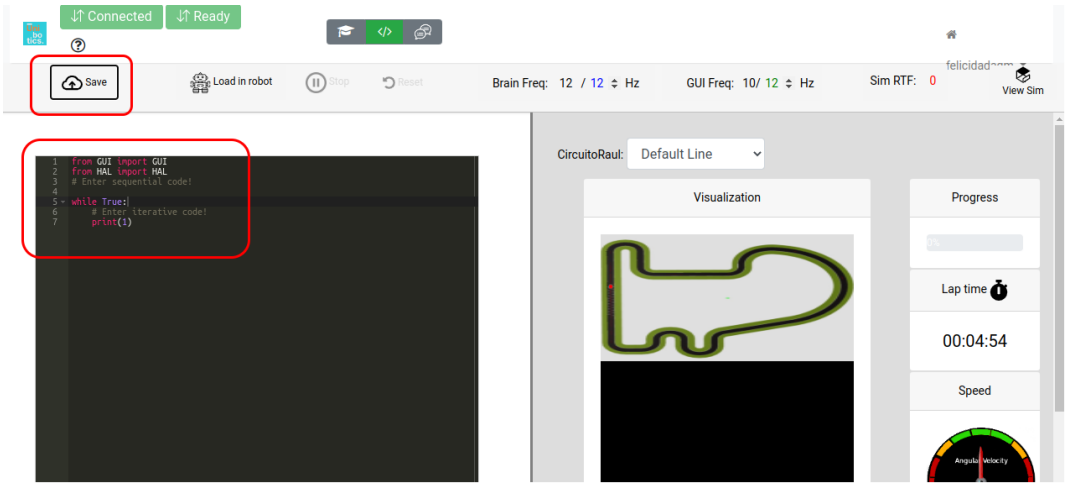

Figura 4.18: Código salvado gracias al botón save.

- **test_exercise_gazebo**: Este botón se localiza mediante el id *"gazebo-button"*, y su objetivo es mostrar el mundo en cada uno de los ejercicios con robot. Es una prueba que no se encuentra en todos los ejercicios, pues no todos utilizan ROS.

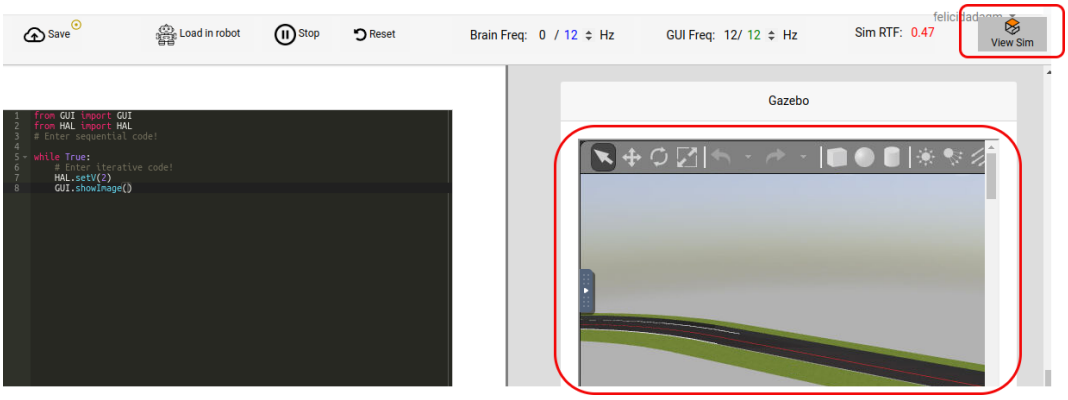

Figura 4.19: Botón Gazebo y su visualización.

- **test_exercise_teleop_button**: No todos los ejercicios disponen de este botón que se localiza por el id *"teleop-button"*, pero los que sí lo hacen, permiten que el robot sea dirigido también con las flechas del teclado.

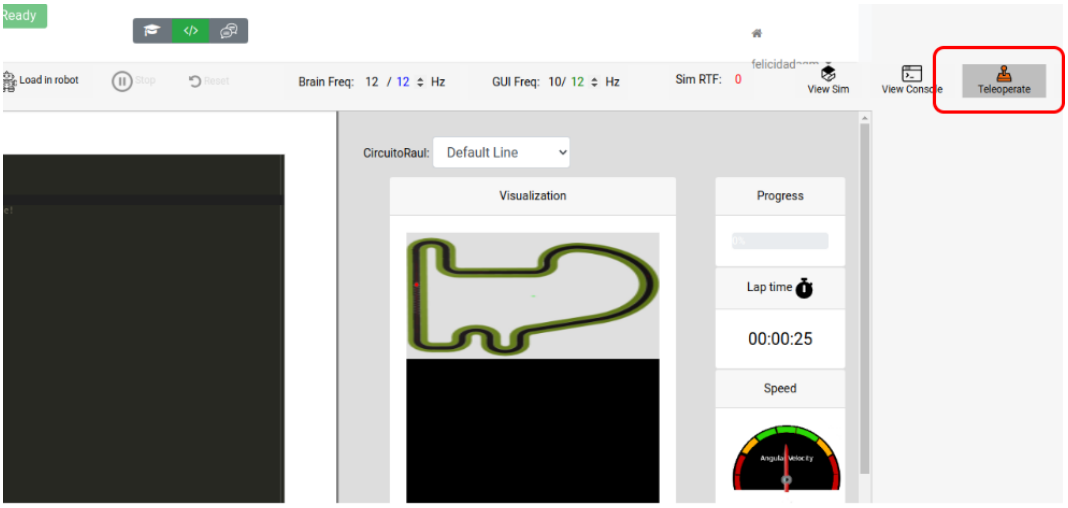

Figura 4.20: Botón teleop.

- **test_exercise_efficacy_button y test_exercse_evaluate_style**: Estos dos botones que se localizan con los ids *efficacy* y *evaluate* respectivamente, se encargan de evaluar la eficacia y el estilo del código de usuario.

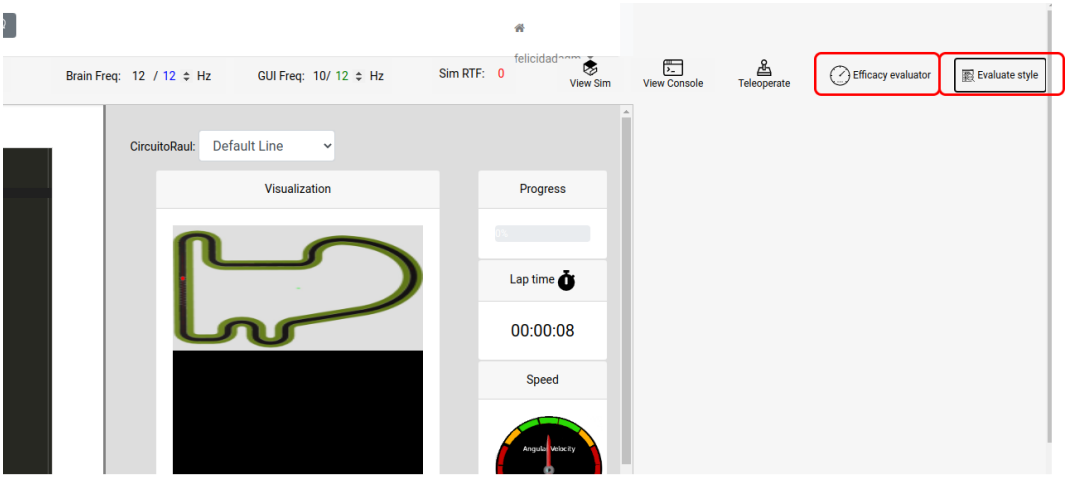

Figura 4.21: Botones efficency y evaluate style.

El archivo de pruebas se ejecuta al completo siempre que no sea cortado de forma manual e independientemente de si las pruebas dentro de cada bloque dan error o terminan correctamente. Aunque es posible controlar por consola el resultado de cada prueba, pues imprime un valor en función de si ha ido bien o mal, lo mejor es comprobar el resultado de las mismas una vez han terminado todas.

En caso de error en una prueba unitaria, se imprime una E por consola, y una vez terminan todas las pruebas se puede comprobar en la salida el resultado. Esta salida es como la de la Figura 4.22, donde se indica el número total de pruebas ejecutadas, cuántas de las mismas han fallado y el tiempo que han tardado en ejecutarse.

```
----------------------------------------------------------------------
Ran 10 tests in 565.623s

FAILED (errors=7)
Destroying test database for alias 'default'...
```

Figura 4.22: Resultado en consola de pruebas fallidas.

Además, también se puede encontrar el motivo por el que estas pruebas han fallado (ver Figura 4.23).

```
selenium.common.exceptions.NoSuchElementException: Message: Unable to locate element: [id="efficacy1234"]
Stacktrace:
WebDriverError@chrome://remote/content/shared/webdriver/Errors.jsm:188:5
```

Figura 4.23: Traza en consola del motivo del error.

En caso contrario, cuando una prueba unitaria termina con un resultado correcto, se imprime un punto (.) en la consola. Y una vez todo el conjunto de pruebas ha terminado, se imprime un resultado parecido al del caso de error, en el que se indica: número de pruebas, tiempo que se ha tardado en ejecutarlas y en esta ocasión un OK como resultado (ver Figura 4.24)

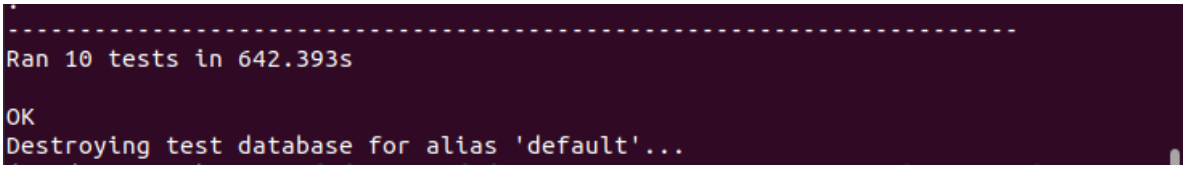

Figura 4.24: Resultado en consola de pruebas correctas.

4.6. Pruebas integrales

Esta sección profundiza en aquellas particularidades de las pruebas automáticas de integración, desde la definición de las pruebas hasta los resultados de las mismas, pasando por la automatización del levantamiento del contenedor RADI y el envío de código de usuario al editor.

En esta clase de pruebas entran en juego todos los elementos del sistema, por lo que el trabajo que ha sido necesario desarrollar es bastante más complejo. Primero, se definió un objetivo claro para estas pruebas: Insertar código simulando a un usuario, ejecutarlo y comprobar que después de unos cinco segundos desde que comienza a ejecutarse dicho código la posición del robot ha cambiado en el simulador. De esta forma es posible comprobar que las modificaciones de código fuente que se realizan en el repositorio de Unibotics no han afectado a la funcionalidad de los ejercicios en producción.

Para lograr esto, el desarrollo de estas pruebas se dividió en varias partes:

1. Entrar en el ejercicio, algo que ya se había conseguido con las pruebas unitarias de frontend y puede reaprovecharse.
2. Levantar el contenedor RADI y el ejercicio pulsando el botón Launch y esperando a que esté todo listo.
3. Comprobar la posición inicial del robot.
4. Introducir código de usuario y ejecutarlo.
5. Comparar ambas posiciones para ver que el robot se ha movido. Si los valores de las coordenadas cambian, significa que el código de usuario se está ejecutando de forma correcta.

Las pruebas de integración, al igual que las de frontend, se realizarán por bloques donde cada bloque corresponde a un ejercicio concreto y contiene toda la lógica necesaria para realizar los pasos anteriores y devolver un resultado. En las siguientes secciones se explicarán estos pasos con mayor detalle.

4.6.1. Automatización levantamiento contenedor RADI.

Uno de los requisitos indispensables para que un usuario pueda realizar los ejercicios de Unibotics es tener instalado el contenedor RADI y lanzarlo como se explica nada más entrar a la página de cualquiera de los ejercicios disponibles (ver Figura 4.8). Como en las pruebas de integración el objetivo es probar de una forma lo más realista posible el sistema, también existirá este mismo requisito.

Con el objetivo de ahorrar tiempo en la ejecución de las pruebas, pues la imagen del contenedor puede ser bastante pesada, en la máquina en que se ejecuten estas pruebas ya existirá el contenedor RADI, siendo trabajo del archivo de Selenium iniciarlo para que el resto de pasos puedan ejecutarse.

Para cumplir con este propósito se ha utilizado el módulo *subprocess*, que permite ejecutar comandos en el sistema operativo y recoger el resultado de los mismos. Entre las funciones que ofrece este módulo, primero se probó con *Popen*, que ejecuta un comando en segundo plano, sin esperar el resultado de este. Sin embargo, en estas pruebas de integración necesitábamos estar seguros de que el contenedor RADI estuviese levantado en el momento en que se comienzan a ejecutar los bloques de prueba, algo que *Popen* no permite asegurar. Por este motivo se ha optado por usar la función *subprocess.run()*. Se trata de una función muy similar a *Popen*, pues también permite ejecutar un comando con la diferencia de que la función run espera a que la ejecución de este comando finalice antes de continuar con el resto del código. De esta forma, permite asegurar que el contenedor RADI esté levantado antes de ejecutar los distintos bloques de pruebas.

Con el uso de esta función al inicio de las pruebas, se cumple el objetivo de esta sección, pues el levantamiento del contenedor RADI será automático.

4.6.2. Entrar en el ejercicio y levantarlo.

Como se ha explicado en el apartado de las pruebas unitarias de frontend (ver sección 4.5), nada más entrar a la página el primer paso sería localizar el botón de log in, introducir los datos e iniciar sesión.

Una vez la sesión esté iniciada, se redirigirá automáticamente a la parrilla de ejercicios, donde sólo se encontrará el ejercicio que el bloque de pruebas cree con el módulo *setUp()*.

Tras entrar en la página del ejercicio, el usuario debe lanzar el mismo antes de poder utilizarlo, de forma que el contenedor RADI prepare e inicie todo lo necesario para

poder ejecutar el código enviado por el usuario cuando este desee ejecutarlo. Cuando se pulsa el botón *"Launch"* en la página de un ejercicio, el *WebSocket* que se comunica con el *manager.py* le envía a este un mensaje que contiene el identificador del ejercicio y la orden *"open"*.

Con esta orden el *manager.py* busca dentro de un fichero llamado *instructions.json* donde se encuentran las indicaciones de todo aquello que debe iniciar (ver Figura 4.25)

```
23        "color_filter": {
24            "instructions_host": "python3 /RoboticsAcademy/exercises/static/exercises/color_filter/exercise.py 0.0.0.0"
25        },
26        "drone_cat_mouse": {
27            "gazebo_path": "/RoboticsAcademy/exercises/static/exercises/drone_cat_mouse/launch",
28            "instructions_ros": ["python3 /RoboticsAcademy/exercises/static/exercises/drone_cat_mouse/launch/launch.py"],
29            "instructions_host": "python3 /RoboticsAcademy/exercises/static/exercises/drone_cat_mouse/exercise.py 0.0.0.0",
30            "instructions_guest": "python3 /RoboticsAcademy/exercises/static/exercises/drone_cat_mouse/exercise_guest.py 0.0.0.0"
31        },
```

Figura 4.25: Fichero con instrucciones para distintos ejercicios

Tras encontrar estas instrucciones e interpretarlas, el botón se volverá verde y su texto cambiará por *"Ready"*. Además de ello, se abrirá una alerta indicando que la conexión ha sido establecida (ver Figura 4.26).

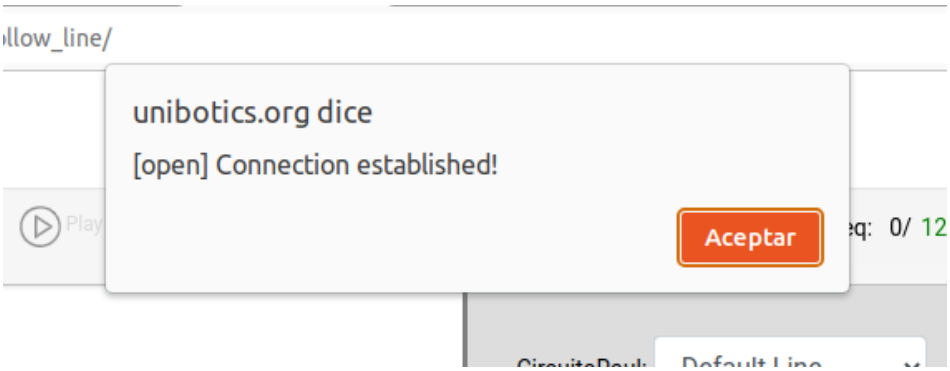

Figura 4.26: Alerta de conexión establecida.

El tiempo que tarda en completarse y aparecer esta alerta no es un tiempo fijo, sino que varía entre ejercicios, máquinas e incluso instantes de tiempo en la misma máquina. Es por ello que ordenar a Selenium esperar un tiempo fijo no era una buena opción, sino que ha sido necesario encontrar la forma de hacer que esperase hasta la aparición de la alerta, y aceptarla inmediatamente después.

Para lograr esto, ha sido necesario utilizar la librería *WebDriverWait* de Selenium. Esta librería permite que el código en ejecución espere hasta que se cumpla una condición en la página web antes de continuar realizando acciones. También ha sido necesario evitar que Selenium quede esperando un tiempo indefinido en caso de que ocurra algún error inesperado dentro del contenedor que no le permita preparar todo lo que cada

ejercicio necesita.

Un ejemplo de cómo se ha usado la librería en este trabajo es la siguiente línea de código: *alert = WebDriverWait(self.selenium, 60).until(EC.alert_is_present())*, y la forma en que funciona es la siguiente:

1. La parte a la izquierda del punto, ***WebDriverWait(self.selenium, 60)***, crea una nueva instancia de este librería, donde el primer parámetro es el controlador del navegador, y el segundo es el tiempo que se quiere esperar como máximo. Si en el tiempo especificado no se cumple la condición, esa prueba termina con un error por *timeout* y se continúa ejecutando las siguientes. En caso de no haber más, se devolverá un error directamente (ver Figura 4.22).

2. La parte de la derecha del punto, ***until(EC.alert_is_present())***, es la condición que Selenium va a esperar que se cumpla antes de continuar o dar error por *timeout*. En el caso del ejemplo, espera que una alerta se abra en el navegador.

Para ayudar a los desarrolladores, existen una serie de condiciones predefinidas para usar junto librería *WebDriverWait*. Esta lista de condiciones se puede obtener con el módulo de Selenium *"expected_conditions"*

Estando en la página del ejercicio, conectados al contenedor y teniendo este todo preparado para poder ejecutarlo, el objetivo de esta sección está completo.

4.6.3. Automatización de envío de código de usuario.

Para que el robot cambie de posición, es necesario que se ejecute código de usuario. Pero en unas pruebas automatizadas de integración, no va a haber ningún usuario real que interactúe con la página, inserte código y trate de ejecutarlo. Por ello debemos simularlo utilizando Selenium.

Lo primero que Selenium debe hacer es encontrar el editor ACE en el HTML e introducir líneas de código fuente del programa del robot. Este código introducido no pretende resolver el ejercicio ni debe ser perfecto, simplemente tiene el objetivo de mover el robot para poder comparar posteriormente la posición del mismo.

A la hora de realizar esta funcionalidad, han surgido varios problemas y puntos de bloqueo, que han llevado a probar diferentes alternativas hasta finalmente dar con la solución más óptima que permite realizar las pruebas.

La primera alternativa probada ha sido la función *send_keys* ofrecida por el propio Selenium. Esta función ya se ha usado en pruebas anteriores y su función es enviar texto a una *input* de la página web. Aunque para introducir, por ejemplo, el nombre de usuario y contraseña ha servido, en esta ocasión se devolvía un error indicando que no era posible interactuar con el elemento (ver Figura 4.27). Esto es porque el editor ACE no responde a las acciones de entrada de texto que normalmente usa el navegador, pues

este editor tiene unas acciones propias específicas.

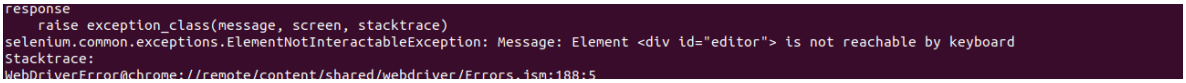

Figura 4.27: Error elemento no interactuable Selenium

La siguiente alternativa es la función *setValue()*. Esta función es proporcionada por ACE y su objetivo es reemplazar el contenido que hay en el editor por el texto que se le pase como parámetro. Sin embargo, aquí surgió un nuevo problema: la función *setValue()* es proporcionada por ACE, lo que significa que Selenium y su controlador no son capaces de interpretarlas directamente y convertirlas en acciones de navegador (ver Figura 4.28). Esta alternativa también ha tenido que ser parcialmente descartada.

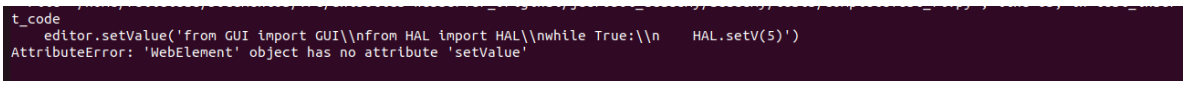

Figura 4.28: Error elemento web no puede interpretar setValue

Fue descartada sólo parcialmente porque ya se ha ubicado la función propia de ACE que es capaz de modificar el contenido y escribir un código dado, lo que queda pendiente es encontrar la forma de que el controlador de Selenium sea capaz de interpretarlo correctamente. Para ello existe la función *execute_script* que ofrece Selenium. Esta función ejecuta código JavaScript que interactua con los elementos de la página, modifica su contenido, etc. Por lo tanto, para solucionar el problema primero se ha localizado el editor ACE y a continuación se ha usado la función *execute_script* que tiene en su interior la función de ACE *setValue()*, como se muestra en el listado 4.8:

Listing 4.8: Función execute script Selenium.

```
1     self.selenium.execute_script("ace.edit('editor').setValue('from␣
          GUI␣import␣GUI\\nfrom␣HAL␣import␣HAL\\nwhile␣True:\\n␣␣␣␣
          HAL.setV(5)')")
```

Esta alternativa también presentaba un error al ser ejecutada (ver Figura 4.29), aún sabiendo que se trataba de las funciones correctas para el objetivo buscado.

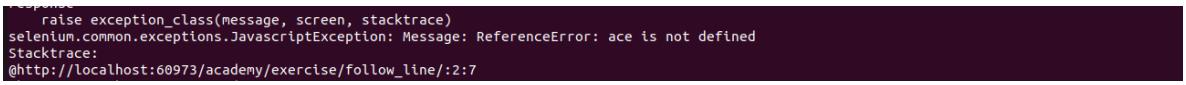

Figura 4.29: Error función execute script Selenium

Si las funciones estaban siendo usadas de manera correcta y era capaz de insertar código fuente probando directamente contra la url de producción, el problema se encontraba en otro punto.

La clase base *LiveServerTestCase* no envía los archivos estáticos al cliente cuando este solicita una págna web, lo que provoca que cosas como el CSS o el editor ACE no carguen de forma correcta (ver Figura 4.30). Al no cargar el editor correctamente, imposibilitaba el envío de código pues hacía que Selenium fuese incapaz de interactuar con él.

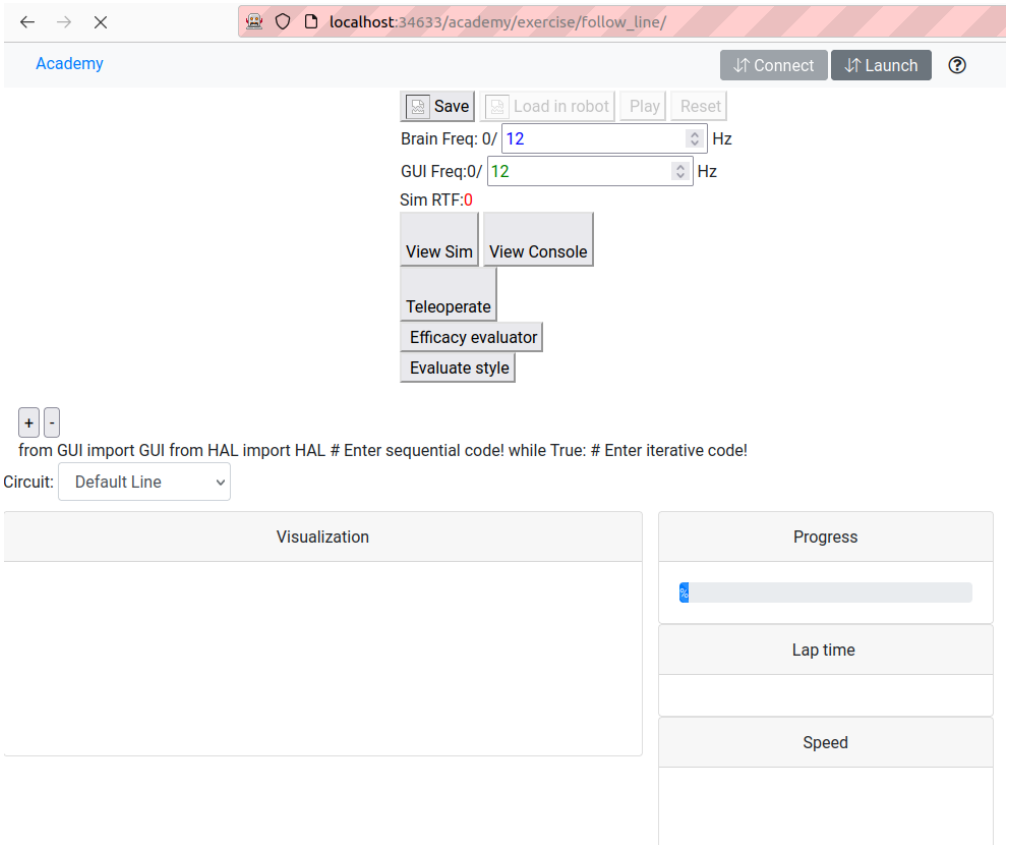

Figura 4.30: Carga de estáticos con LiveTestCaseServer.

Como solución, se sustituyó la clase *LiveTestCaseServer* por una de sus subclases: ***StaticLiveServerTestCase***. Entre estas clases existe una principal diferencia, y es la forma en que tratan los archivos estáticos. La clase *StaticLiveServerTestCase* se asegura de que los archivos estáticos se envíen y carguen de forma correcta (ver Figura 4.31). De esta forma, Selenium es capaz de localizar el editor sobre el que necesita ejecutar código.

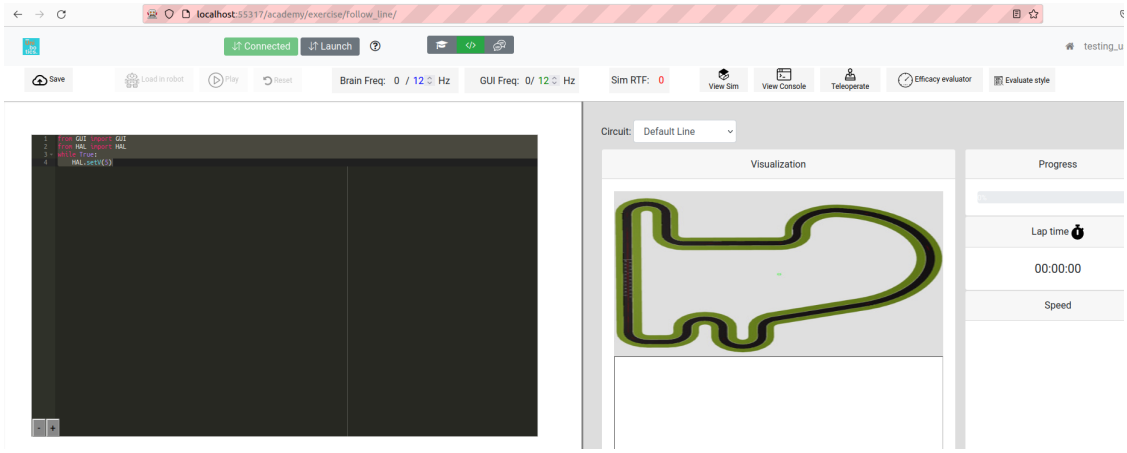

Figura 4.31: Carga de estáticos con StaticLiveServerTestCase.

Una vez insertado el código fuente en el robot, el siguiente paso es encontrar y pulsar el botón que carga el código en el robot. Para ello, se localiza mediante el id *"loadIntoRobot"* el botón. Tras pulsarlo, se abre una alerta que le indica al usuario que debe esperar mientras el código carga en el robot (ver Figura 4.32) y oscurece el resto de la página debajo de él, lo que hace que Selenium no pueda interactuar con ella. Al igual que en casos anteriores, el tiempo de carga no siempre va a ser el mismo, por ello se ha hecho que Selenium espere de forma dinámica a que este elemento desaparezca para poder continuar, con un tiempo máximo de 40 segundos.

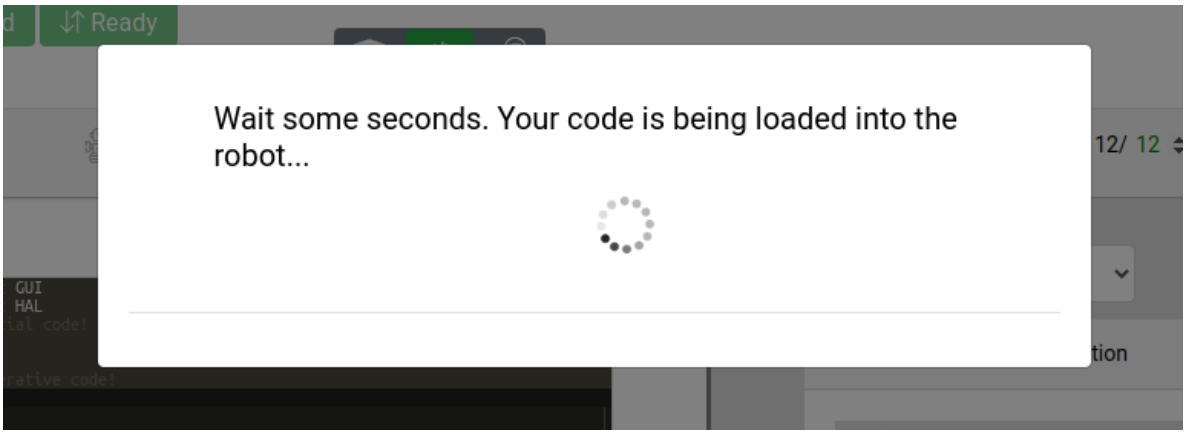

Figura 4.32: Alerta mientras carga código usuario

En cuanto este elemento desaparece, se pulsa el botón play, localizado por su id *"submit"*. El código empieza a ejecutarse en el robot.

4.6.4. Obtención automática de las coordenadas y resultados.

Para poder dar un resultado a cada prueba, es necesario comprobar y comparar la posición del robot en dos momentos distintos durante la prueba. La primera toma de coordenadas se hace cuando el contenedor ha terminado de preparar el ejercicio, es decir, se toman las coordenadas del robot en su punto inicial. La segunda, es cuando han pasado cinco segundos desde que el código fuente usado en la prueba ha cargado en el robot.

Ha sido necesario adquirir algo de conocimiento sobre los ROS topics para poder realizar esta parte del trabajo. A alto nivel, los ROS topics son canales que permiten que distintos nodos intercambien información, que tienen un nombre único y una estructura de mensajes pre-definida. Habrá un ROS topic por cada ejercicio en el que la solución necesita conocer esa posición continuamente.

Para conseguir localizar el topic que usa cada ejercicio, se ha hecho un estudio ejercicio por ejercicio para obtener tanto el nombre del topic como la estructura del mensaje que utiliza el mismo, con el fin de agrupar lo máximo posible el procesamiento de estos mensajes (ver Figura 4.33).

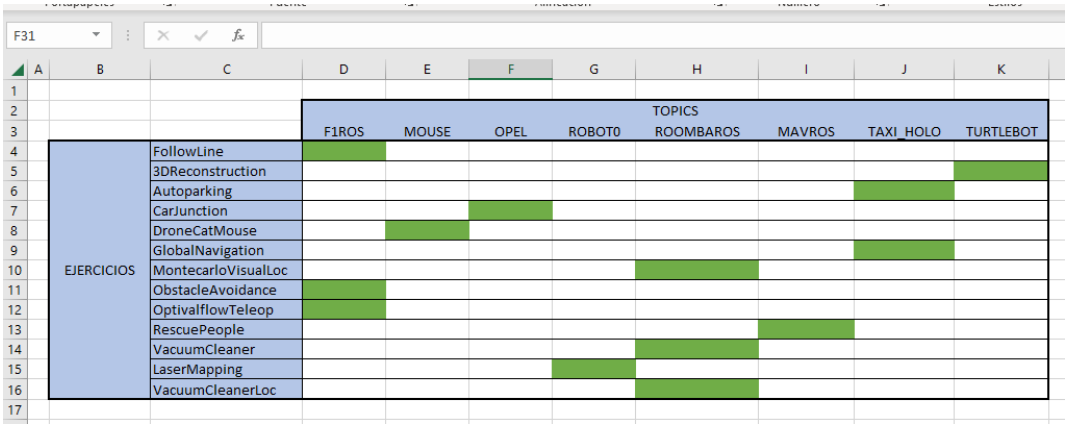

Figura 4.33: Tabla de análisis de topic usados y formatos por ejercicio

Es necesario conseguir dos de estos mensajes que se intercambian a través del ROS topic en los momentos especificados (inicio y tras la ejecución del código de prueba) para poder obtener las coordenadas del robot.

Si la máquina en la que se realizan las pruebas no tiene instalado ROS, o no tiene la configuración adecuada, no se podrán realizar las consultas a los topics desde fuera del contenedor. Para evitar tener que añadir configuración, y que sea posible tener todas las pruebas completas en un solo repositorio centralizadas, además de evitar tener que realizar modificaciones en la imagen de Docker con cada cambio en las pruebas, se crean unos archivos que Selenium se encarga de insertar dentro del contenedor en caliente con un subproceso. Una vez las pruebas han terminado, independientemente del resultado, Selenium también se encarga de eliminar el archivo que ha insertado. De esta forma,

estos no afectan ni modifican de forma permanente el contenedor RADI oficial.

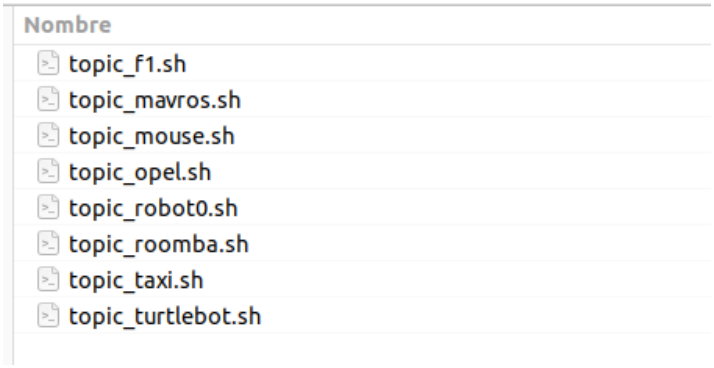

Figura 4.34: Lista de archivos que se intertan y borran del contenedor

Estos archivos que se insertan en caliente en el contenedor hacen uso del comando *rostopic echo* para suscribirse a un ROS topic y mostrar los mensajes que este publica en tiempo real. Se ha acompañado también de los parámetros *-n 1* para especificar el número de mensajes que se desean mostrar al ejecutar el comando.

Los archivos mencionados anteriormente se ejecutan dos veces mediante un subproceso con la función *run*, de forma que se espera a tener el resultado y el mensaje de ese topic antes de continuar. El resultado de este subproceso se devuelve en bytes, por lo que antes de empezar a tratarlo es necesario decodificarlo. Como todo este tratamiento previo a la comparación de posiciones para dar un resultado se repite en muchos de los bloques de prueba, se han creado funciones modelo a las que llamar. Estas funciones tienen como parámetro de entrada la respuesta del subproceso y decodifican el resultado del mismo, recortan el *string* resultante para poder quedarse sólo con la parte que interesa de cada estructura de mensaje y convertirlo a un formato YAML con la función *yaml.safe_load*. Esta función permite cargar datos YAML de forma segura, ya que evita que se ejecute código que puede estar en estos mismos YAML y ser peligroso.

Listing 4.9: Función que trata la respuesta del subproceso para recuperar posiciones.

```
1 def processResponse (data):
2     data = data.stdout.decode('utf-8')
3     findPoseIndex = data.index("pose")
4     findtwistIndex = data.index("twist")
5     pose_yaml =
          yaml.safe_load(data[findPoseIndex:findtwistIndex])
6
7     return pose_yaml
```

Por último se comparan los valores de las posiciones de ambos momentos con *assert*, una sentencia de Python que comprueba una condición dada. Por ejemplo *assert positions1 != positions2*. Si el resultado de esta aserción es *true*, significa que la posición

del robot ha cambiado y la prueba debe finalizar positivamente (ver Figura 4.35). Si por el contrario, el resultado es *false*, significa que la posición del robot no ha cambiado y la prueba debe devolver un error (ver Figura 4.36).

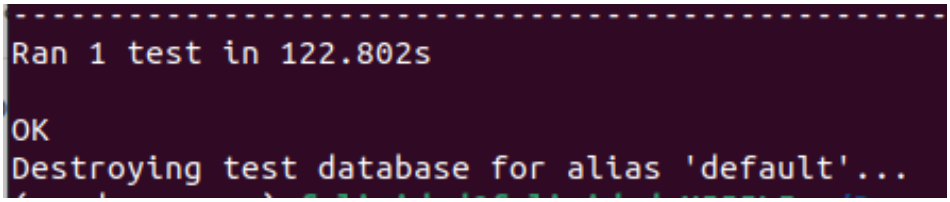

Figura 4.35: Resultado correcto de las pruebas de integración.

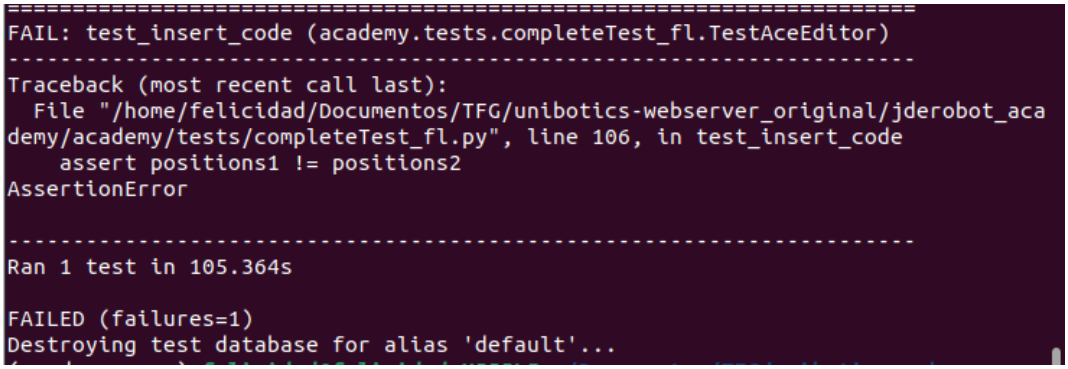

Figura 4.36: Error de las pruebas de integración.

Capítulo 5

Ejercicio introductorio al procesamiento de imagen

En este capítulo se recoge el desarrollo de un ejercicio nuevo para RoboticsAcademy. Este ejercicio de basa en el procesamiento de imagen, y a lo largo de este capítulo de explicará tanto el diseño del mismo, las piezas que lo componen y los retos escogidos con sus soluciones de referencia. Las tecnologías usadas principalmente han sido Python y la librería OpenCV.

5.1. Diseño del ejercicio

Este ejercicio está dirigido a usuarios que quieran introducirse en el mundo del procesamiento de imagen y utilicen la plataforma *RoboticsAcademy*, en la que se basa Unibotics y que es 100 % software libre. El objetivo principal es proporcionar una serie de retos de procesamiento de imagen 2D, de manejo de píxeles simples en un principio y que vayan aumentando de dificultad pero que a su vez permitan al usuario profundizar en el uso de la librería *OpenCV* y las múltiples funciones que esta ofrece para el procesamiento de imagen.

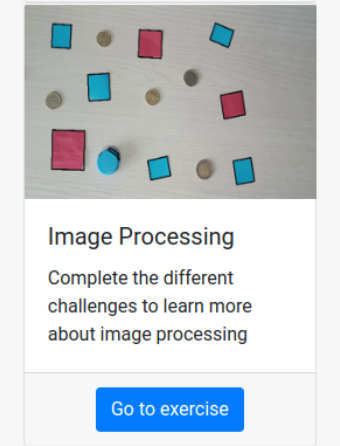

Figura 5.1: Teaser el ejercicio en la parilla de la página

Para asegurar la compatibilidad con los distintos sistemas operativos y que cualquier usuario pueda realizar este ejercicio, se ha optado por utilizar los fotogramas de un vídeo fijo como imágenes sobre las que realizar distintos tratamientos. Esto es debido a que los usuarios de Windows pueden encontrar serios problemas para utilizar su cámara real en vivos con los ejercicios en caso de no tener disponible un equipo con el sistema operativo Linux. A alto nivel, esto se produce porque Windows crea un entorno aislado en el que ejecuta los contenedores y este entorno está completamente separado del sistema operativo del anfitrión, este proceso es llamado "virtualización a nivel de sistema operativo". Por ello, desde ese entorno aislado no se permite el acceso a todos los recursos del host, como pueden ser la cámara USB u otros dispositivos de hardware.

El usuario deberá seguir los mismos pasos que en el resto de ejercicios de la plataforma y tener un contenedor RADI instalado y en ejecución antes de usar este nuevo ejercicio y deberá programar en Python un algoritmo para resolver los distintos retos propuestos.

La página del ejercicio se ha creado en HTML, siguiendo la base propuesta por el resto de los ejercicios de la plataforma (ver Figura 5.2). Además, también se proporciona al usuario una página de documentación donde pueden encontrarse: el objetivo del ejercicio, instrucciones sobre como usarlo, las principales funciones de los módulos *HAL* y *GUI* además de una lista con los retos propuestos, teoría para ayudar a comprender las funciones usadas y un apartado con pistas sobre la forma de resolver los retos.

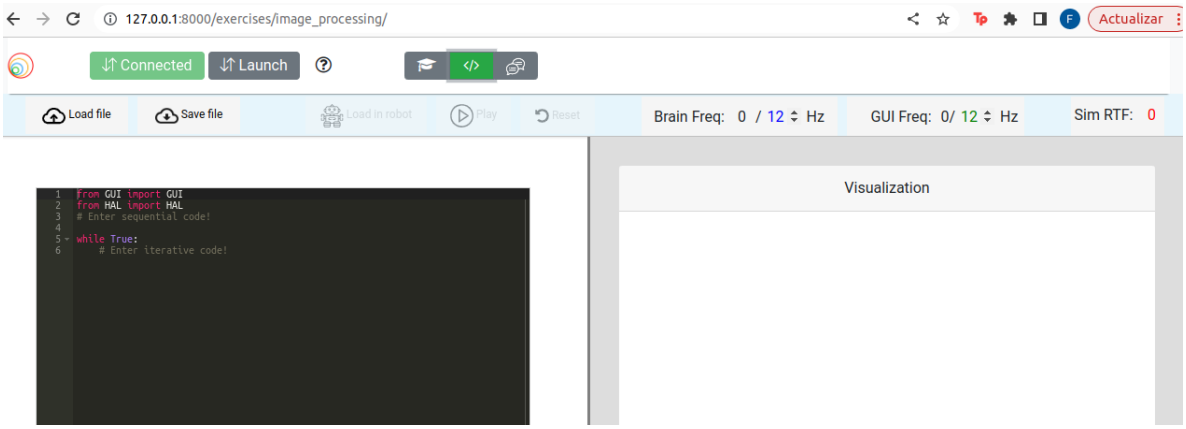

Figura 5.2: Página del ejercicio image processing

Una vez enviado el código fuente de usuario, el archivo *exercise.py* se encarga de procesarlo, dividiéndolo entre código secuencial (sólo se debe ejecutar una vez) y código iterativo (se debe ejecutar en bucle hasta cumplir una condición) para después ejecutar ambos.

El usuario podrá ver en el cuadro de visualización los fotogramas del video grabado. Se trata de un vídeo relativamente simple, donde hay objetos estáticos de distintas

formas y colores además de otro objeto que aparece y se mueve por la pantalla hasta desaparecer de vídeo. El objetivo es utilizar una entrada de imágenes fija e independiente del hardware y sistema operativo de usuario para poder realizar sobre él distintos procesamientos.

5.2. Retos propuestos y soluciones de referencia.

Como en muchos de los temas relacionados con la programación, no hay una sola forma correcta de resolver los retos propuestos. Cada usuario puede llegar a una solución por un camino distinto e igualmente válido si el objetivo se cumple. Sin embargo, se adjunta una solución de referencia a cada uno de los retos propuestos.

Estos retos se han ordenado del más simple al más complejo, y se expondrán en las siguientes secciones.

5.2.1. Transformar fotogramas a blanco y negro.

El primero de los retos consiste en aplicar a los fotogramas una transformación del espacio de color de la imagen[1]. El espacio de color que usa OpenCV por defecto es el espacio BGR (Blue, Green, Red) donde cada píxel de una imagen está compuesto por tres valores.

- Intensidad del color azul (B)
- Intensidad del color verde (G)
- Intensidad del color rojo (R)

Para pasar una imagen a blanco y negro, es necesario transformar esos tres valores que hacen referencia a la intensidad de tres canales distintos, en un sólo canal con un sólo valor de intensidad. Para realizar estas acciones, OpenCV ofrece la función *cv2.cvtColor()* que necesita dos parámetros: la imagen sobre la que realizar el cambio de espacio de color, y un código para representar el tipo de conversión que se va a realizar.

En este caso, para convertir al espacio de color de grises, el código que se utiliza es *cv2.COLOR_BGR2GRAY*. Este indica a la función que tiene que tomar los tres valores de cada píxel y hacer una media ponderada con ellos, teniendo como resultado un valor de 0 a 255 que será el nuevo valor de intensidad del pixel en escala de grises.

[1] https://imgur.com/a/Q5lgSG8

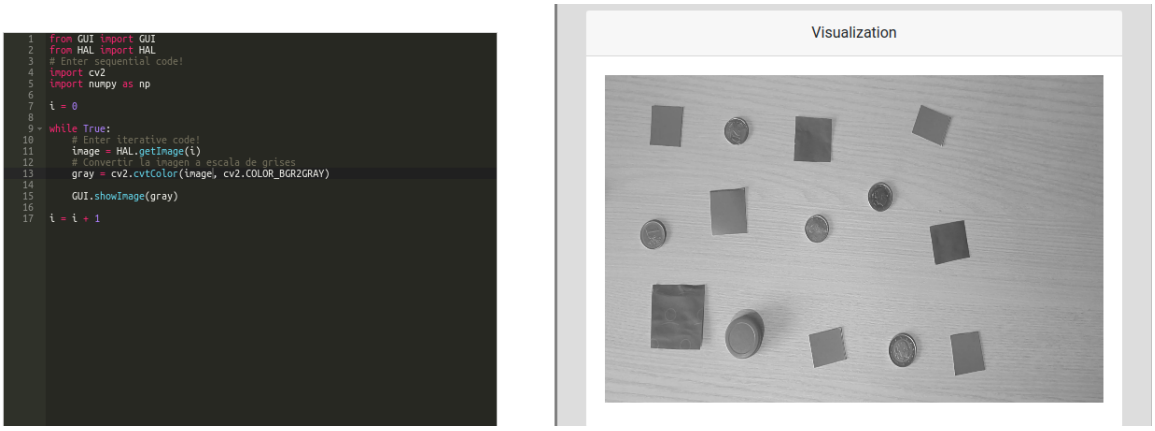

Figura 5.3: Conversión a escala de grises.

5.2.2. Aplicar un filtro de suavizado.

El segundo reto es suavizar cada uno de los frames del vídeo[2]. La propia librería OpenCV ofrece funciones que realizan tipos distintos de desenfoque.

Estos tipos de suavizado tienen diferencias notables y se utilizan con distintos propósitos dentro del mundo de procesamiento de imagen. En el enunciado no se especifica cúal es el objetivo posterior a este desenfoque, por lo que cualquiera podría ser válido, ya que el objetivo es que el usuario pueda ver la diferencia entre estos tres tipos de suavizado a la vez que los prueba.

Todas estas funcionen tienen un punto en común: los parámetros que necesitan. Siendo el primero de ellos la imagen sobre la que aplicar el suavizado, y el segundo el kernel. El kernel es una matriz, generalmente pequeña y en la mayoría de ocasiones de un tamaño impar (ver Figura 5.4). Esta última característica es muy importante, pues tener un tamaño impar asegura no sólo el tener un píxel central claramente definido, sino que en todas las direcciones este píxel tendrá la misma cantidad de vecinos.

El kernel se mueve sobre la imagen, pasando el valor central por cada uno de los píxeles de la imágen. Con todos los píxeles que caen bajo la matriz del kernel en cada movimiento del mismo, se realiza una operación. Por ejemplo: se multiplica por el valor del kernel y posteriormente se realiza la media del resultado de todos los píxeles bajo la matriz. Este resultado, se asigna al píxel central (ver Figura 5.5) y se pasa al siguiente, lo que consigue una imagen suavizada.

[2]https://imgur.com/a/Rk8dpo1

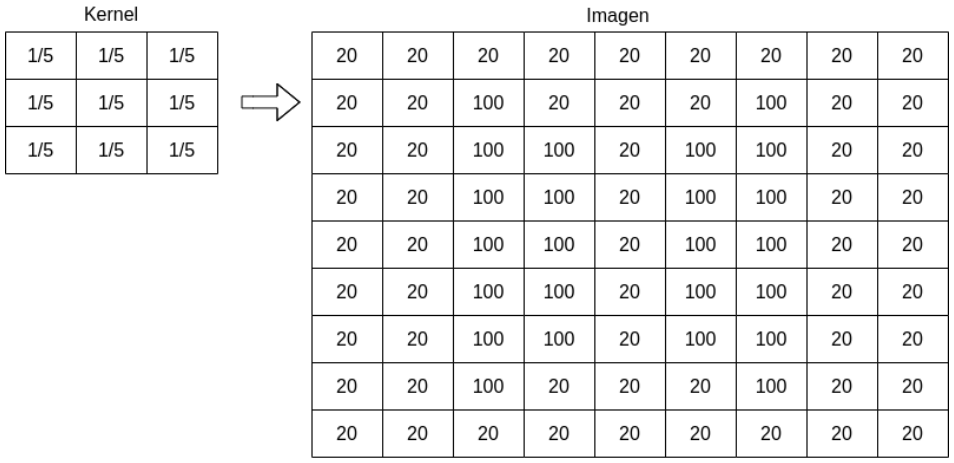

Figura 5.4: Ejemplo de kernel e imagen a convolucionar.

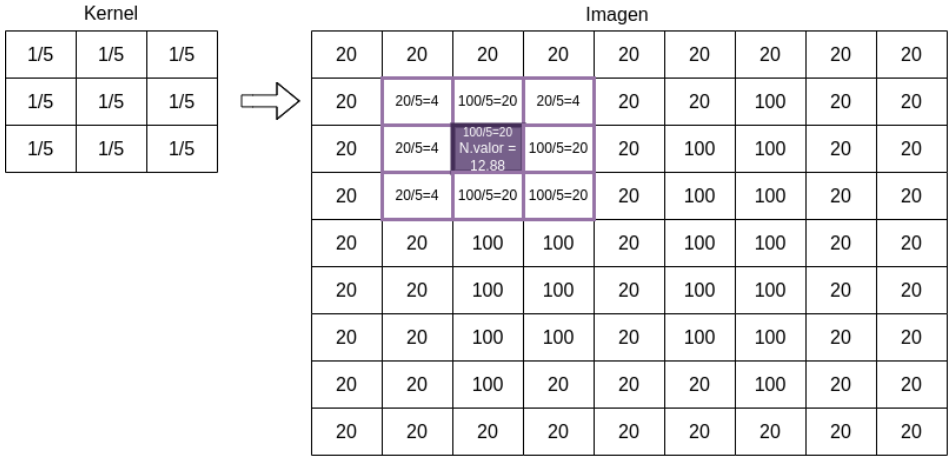

Figura 5.5: Ejemplo del kernel aplicado en una imagen.

Definir bien el tamaño de este Kernel también es importante, pues dependiendo de él la imagen resultará más o menos difuminada. Sabiendo esto, las principales opciones que pueden usarse para suavizar una imagen en OpenCV son:

- **Suavizado de media**: En OpenCV se trata da la función *cv2.blur()*. Se utiliza normalmente para suavizar una imagen de forma parecida. Lo que hace esta función es aplicar el kernel sobre cada pixel de la imagen y calcular la media de todos los píxeles que caen en la máscara. El resultado de esta media, será el nuevo valor del pixel central.

- **Suavizado Gaussiano**: Se usa con la función *cv2.GaussianBlur()*, y normalmente se usa para suavizar pequeños detalles de una imagen o deshacerse del ruido, a la vez que preserva los bordes de la imagen. En este caso, el Kernel aplica diferentes valores a cada uno de los píxeles de la máscara, teniendo más importancia aquellos que más cerca estén del central.

- **Suavizado de mediana**: Pasa usarla, es la función *cv2.medianBlur()*, y es especialmente útil para deshacerse del ruido sal y pimienta (puntos blancos y negros). Esta función, toma todos los valores de los píxeles que caen bajo la máscara y saca la mediana de los mismos, colocando este valor en el píxel central.

Figura 5.6: Comparativa de imagen normal (izquierda) e imagen suavizada (derecha).

5.2.3. Encontrar contornos de los objetos.

Encontrar y pintar los contornos de los objetos del vídeo es el tercer reto[3]. Como los objetos que aparecen en el vídeo son de distintas formas y colores, intentar realizar un filtro de colores no va a servir y se debe aplicar paso a paso un proceso para localizar los contornos.

El primer paso de la solución propuesta es convertir la imagen al espacio de color de escala de grises, para lo que puede servir el código desarrollado en la sección 5.2.1. Una vez la imagen se encuentra en escala de grises, el siguiente paso es convertir la imagen a una imagen binaria (ver Figura 5.7). En una imagen binaria los píxeles sólo pueden tener dos valores:

- Blanco (1), objeto de interés en la imagen.
- Negro (0), pertenece al fondo de la imagen.

Para diferenciar qué parte del fotograma pertenece al fondo y cuál pertenece a los objetos de la imagen, se establece un umbral. Si un píxel está por encima de ese umbral, en la imagen binaria aparecerá en blanco. Si un píxel está por debajo de este umbral, aparecerá en negro. En este caso, como el fondo es de un color más claro que los objetos, tendrán valores más altos de luminosidad por lo que además habrá que hacer la inversa de la imagen binaria.

Una vez la imagen binaria ya ha segmentado los objetos de interés en la imagen, se pueden dibujar y posteriormente tratar de perfeccionar y suavizar los bordes de los

[3]https://imgur.com/a/Ply4RAa

Figura 5.7: Imagen binaria

objetos para un resultado más limpio.

Con la imagen binaria ya generada, se usa la función *findContours* de OpenCV con el objetivo de detectar los contornos de este tipo de imagen. Esta función requiere tres parámetros:

1. **Imagen**: Se trata de la imagen de entrada. Esta debe ser una imagen binaria donde los objetos de interés son los blancos y el fondo en negro.

2. **Modo**: Es un código que identifica la forma de conseguir los contornos. Puede tener valores como *cv2.RETR_EXTERNAL* para localizar los contornos externos, *cv2.RETR_LIST* que encuentra todos los contornos sin tener en cuenta si son internos o externos respecto a otros. Otro posible valor de este código es *cv2.RETR_TREE* que localiza los contornos junto con la información de si son externos o internos.

3. **Método**: Es un código que identifica la aproximación de los contornos. Puede tener valores como *cv2.CHAIN_APPROX_NONE* que almacena cada punto del contorno o *cv2.CHAIN_APPROX_SIMPLE* que guarda los puntos iniciales y finales de cada contorno.

La función *findContours* devuelve dos valores. El primero de ellos, los una lista donde cada posición es una matríz con las coordenadas x e y de los puntos que forman el contorno. En segundo lugar, una matriz donde se encuentra la información sobre los contornos recuperados y su jerarquía.

Para pintar los contornos, se utiliza la función *drawContours* que recibe cinco parámetros de entrada. El primero, la imagen sobre la que se quieren pintar los contornos. El segundo, la lista de los contornos anteriormente recuperada con la función *findContours*. El tercero, un índice para saber qué contornos dibujar, si este índice tiene un valor positivo se pinta sólo esa posición de la lista de contornos, para pintar toda la lista se usa el valor -1. El cuarto, el color con el que se pintarán los contornos en formato BGR y por último, el quinto parámetro es el grosor del contorno.

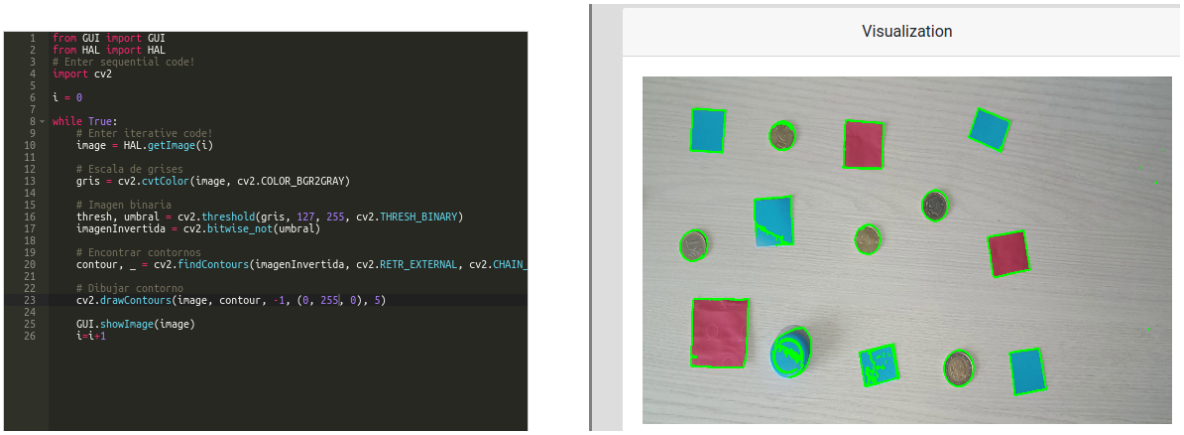

Figura 5.8: Contornos de los objetos en el fotograma.

5.2.4. Encontrar todas las figuras rectangulares.

El cuarto reto de este ejercicio consiste en encontrar todas las figuras rectangulares que existen en el vídeo[4]. En el video de entrada aparecen varias figuras con esta forma, de distintos colores.

En la solución propuesta (ver Listado 5.1), este reto puede empezar a desarrollarse a partir de los anteriores. Con la imagen binaria de los objetos (ver Figura 5.7), se pueden utilizar distintas operaciones morfológicas para limpiar la imagen y perfeccionar los contornos. Otra opción es aplicar primeramente un filtro de colores sobre los fotogramas, creando una máscara para cada uno de ellos y uniéndolas después para poder filtrar la imagen.

En primer lugar, se ha utilizado la operación de *apertura*, cuyo código en OpenCV es *cv2.MORPH_OPEN*. El objetivo de esta operación de apertura (erosión + dilatación), la primera elimina los objetos y detalles pequeños, mientras que la dilatación cierra pequeños huecos.

Después, con la operación de cierre (dilatación + erosión) cuyo código en OpenCV es *cv2.MORPH_CLOSE*, se intenta rellenar los pequeños huecos que hayan podido quedar y a continuación la erosión contrae los objetos y vuelve a eliminar detalles pequeños que pueda haber en la imagen.

[4]https://imgur.com/a/8VoWQpj

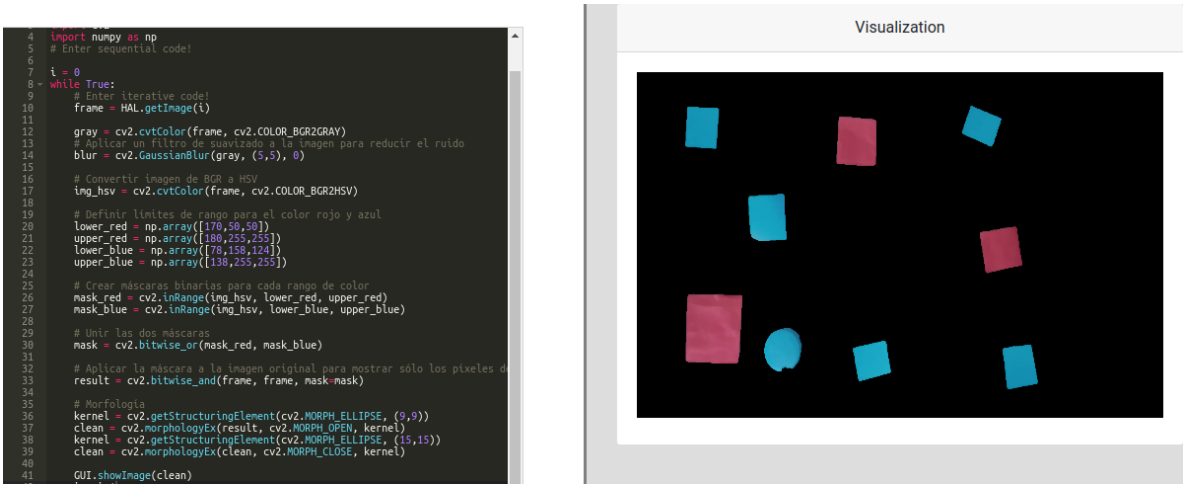

Figura 5.9: Resultado de las operaciones de apertura y cierre

Con la limpieza ya realizada, se buscan los contornos de los objetos del fotograma de la misma forma que se ha hecho en el apartado 5.2.3.

El siguiente paso ha sido crear una lista vacía llamada rectangles, en cuyo interior se almacenarán las figuras rectangulares que se encuentren. Para cada posición de la lista contours, se calcula el perímetro gracias a la función *cv2.arcLength()* que recibe dos parámetros; el primero el contorno para el que se quiere calcular el perímetro y el segundo es un booleano que indica si el contorno es cerrado o no. El resultado de calcular este perímetro se ha guardado en la variable *perimeter*.

La función *cv2.approxPolyDP()* pasándole un contorno, devuelve aproximadamente qué polígono puede ser, dando un número de vértices. Esta función, necesita recibir tres parámetros:

- **Contorno**: El contorno que se va a comprobar.
- **Precisión**: Para especificar la precisión de la aproximación, suele ser un porcentaje del perímetro del contorno a valorar. Cuanto más pequeño sea este valor, se obtendrá un polígono más detallado.
- **Closed**: Es un *booleano*, que indica si el contorno original está cerrado o no.

El resultado de los vértices del polígono se almacena en la variable *approx* y posteriormente se evalúa el valor de esta variable. Si se verifica que el polígono tine cuatro vértices, se añade a la lista de rectángulos anteriormente creada.

Como último paso, se recorre la lista de rectángulos y se pintan los contornos sobre el fotograma original (ver Figura 5.10).

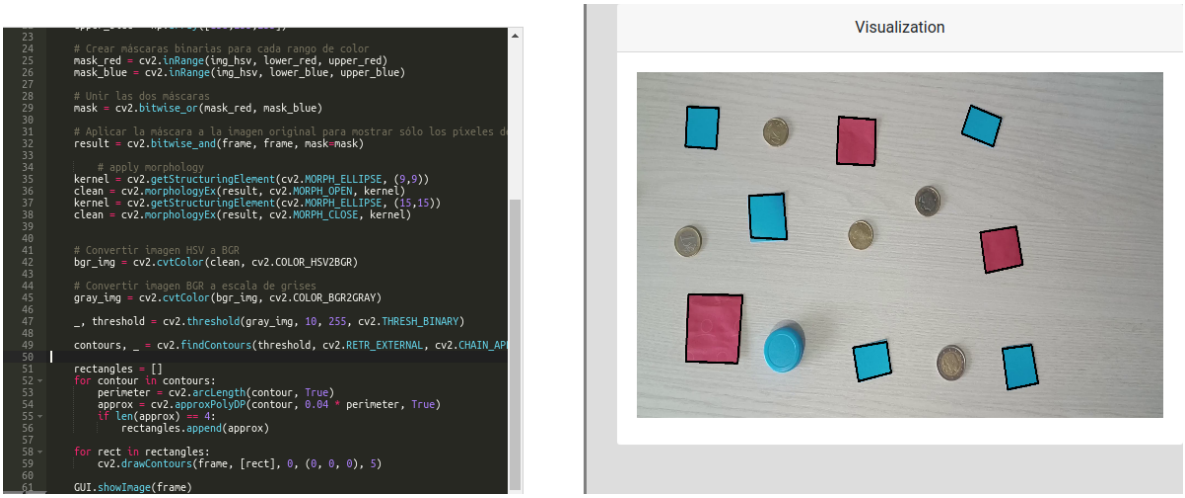

Figura 5.10: Resultado de los rectángulos encontrados en el vídeo

Listing 5.1: Código usado sobre el ejercicio para localizar figuras rectangulares.

```
from GUI import GUI
from HAL import HAL
import cv2
import numpy as np
# Enter sequential code!
i = 0

while True:
    # Enter iterative code!
    frame = HAL.getImage(0)

    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # Aplicar un filtro de suavizado a la imagen para reducir
       el ruido
    blur = cv2.GaussianBlur(gray, (5,5), 0)

    # Convertir imagen de BGR a HSV
    img_hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    lower_red = np.array([170,50,50])
    upper_red = np.array([180,255,255])
    lower_blue = np.array([78,158,124])
    upper_blue = np.array([138,255,255])

    mask_red = cv2.inRange(img_hsv, lower_red, upper_red)
    mask_blue = cv2.inRange(img_hsv, lower_blue, upper_blue)

    mask = cv2.bitwise_or(mask_red, mask_blue)

```

```
29      result = cv2.bitwise_and(frame, frame, mask=mask)
30
31      kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (9,9))
32      clean = cv2.morphologyEx(result, cv2.MORPH_OPEN, kernel)
33      kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,
            (15,15))
34      clean = cv2.morphologyEx(result, cv2.MORPH_CLOSE, kernel)
35
36      # convert to bgr color space
37      bgr_img = cv2.cvtColor(clean, cv2.COLOR_HSV2BGR)
38
39      # convert to gray color space
40      gray_img = cv2.cvtColor(bgr_img, cv2.COLOR_BGR2GRAY)
41
42      # convert to binary
43      _, threshold = cv2.threshold(gray_img, 10, 255,
            cv2.THRESH_BINARY)
44
45      contours, _ = cv2.findContours(threshold,
            cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
46
47      # Find rectangles borders
48      rectangles = []
49      for contour in contours:
50          perimeter = cv2.arcLength(contour, True)
51          approx = cv2.approxPolyDP(contour, 0.04 * perimeter,
                True)
52          if len(approx) == 4:
53              rectangles.append(approx)
54
55      # Draw rectangles
56      for rect in rectangles:
57          cv2.drawContours(frame, [rect], 0, (0, 0, 0), 5)
58
59      GUI.showImage(frame)
60      i = i + 1
```

5.2.5. Encontrar todas las monedas.

El quinto reto consiste en encontrar todas las monedas que aparecen en el vídeo[5]. En este caso, en la solución propuesta (ver Listado 5.2) el primer paso es realizar un filtro de color para eliminar de la imagen el color azul, ya que hay un círculo de ese color que no debe mezclarse con las monedas.

Una vez se han eliminado los objetos azules, ya es posible simplemente encontrar todos los círculos de la imagen para localizar y rodear las monedas. Se convierte la imagen a escala de grises, y se le aplica un filtro de suavizado para suavizar pequeños detalles en la imagen y que no se confundan con otros círculos.

A la hora de encontrar círculos, la propia librería OpenCV ofrece la función *HoughCircles*, que utiliza la transformada de Hough. Esta transformada representa cada círculo con tres valores: coordenadas del centro (x e y) y el radio del mismo (r).

La transformada de Hough circular busca en cada punto de la imagen cada posible círculo que podría pasar por él. Para hacerlo busca todas las combinaciones centro-radio que puedan generar un círculo que pase por el punto que se examina. Y por cada combinación que cumpla las condiciones, se suma uno a un contador de dichas coordenadas y al final se buscan los puntos que tengan un contador más alto. Esos puntos son los círculos que se la función va a detectar.

Esta función *cv2.HoughCircles* necesita una serie de parámetros, que son los siguientes:

- **image**: Debe ser la imagen sobre la que se quieren encontrar los círculos en escala de grises. En el caso de la solución de este ejercicio es el fotograma de entrada.
- **Método**: Qué método de detección se va a utilizar, en este caso el código es *cv2.HOUGH_GRADIENT.*
- **dp**: Inverso de la relación de resolución acumulativa. Este parámetro controla los píxeles o grupos de los mismos sobre los que se van a buscar estos círculos. Es decir, si el valor es uno, significa que se va a buscar píxel por píxel los círculos, mientras que si es un valor mayor a uno significa que se va a saltar varios píxeles a la hora de buscarlos.
- **minDist**: Distancia que debe haber como mínimo entre los círculos detectados. Si este valor es muy pequeño saldrán varios círculos alrededor de un sólo objeto.
- **param1**: Su objetivo es ajustar la calidad de la detección de círculos. Si este parámetro es un número pequeño, significa que el contador debe tener un número más pequeño para ser considerado círculo, lo que puede dar lugar a falsos positivos. Por el contrario, si este parámetro es un número más grande, se necesita que el

[5]https://imgur.com/a/I8HeHcP

valor del contador sea más alto para ser considerado círculo, por lo que será más selectivo.

- **param2**: Establece la sensibilidad del detector de círculos. Si tiene un valor muy bajo va a ser más sensible, es decir, sólo va a considerar círculos aquellos que estén muy claramente definidos. Al contrario, si este valor es alto se seleccionarán círculos con menos precisión, lo que puede encontrar algunos con bordes menos definidos o círculos que puedan estar medio ocultos.
- **minRadius**: Define el radio mínimo de los círculos que se van a buscar.
- **maxRadius**: Define el radio máximo de los círculos que se van a buscar.

La función necesita todos estos parámetros, sin embargo, los valores de los mismos deben irse ajustando dependiendo del objetivo que se busque en el tratamiento de cada imagen.

Por último, si ha sido detectado algún círculo se transforman las coordenadas de centro y el valor del radio en número enteros y se itera sobre los círculos que se han encontrado. Al iterar, se toman los valores de esas cordenadas y se ha usado la función *cv2.circle()* para dibujar círculos alrededor de las moneras detectadas. Esta función recibe varios parámetros, que en orden son:

- Imagen sobre la que se va a dibujar.
- Coordenadas del centro.
- Radio del círculo.
- Color en que se va a dibujar el círculo alrededor de las monedas.
- Grosor del contorno.

El resultado de aplicar esta función sobre los fotogramas del vídeo es el que se muestra en la Figura 5.11.

Listing 5.2: Código solución propuesta para el reto de localizar monedas.

```
from GUI import GUI
from HAL import HAL
import cv2
import numpy as np
# Enter sequential code!

i = 0

while True:
    # Enter iterative code!
    frame = HAL.getImage(i)
```

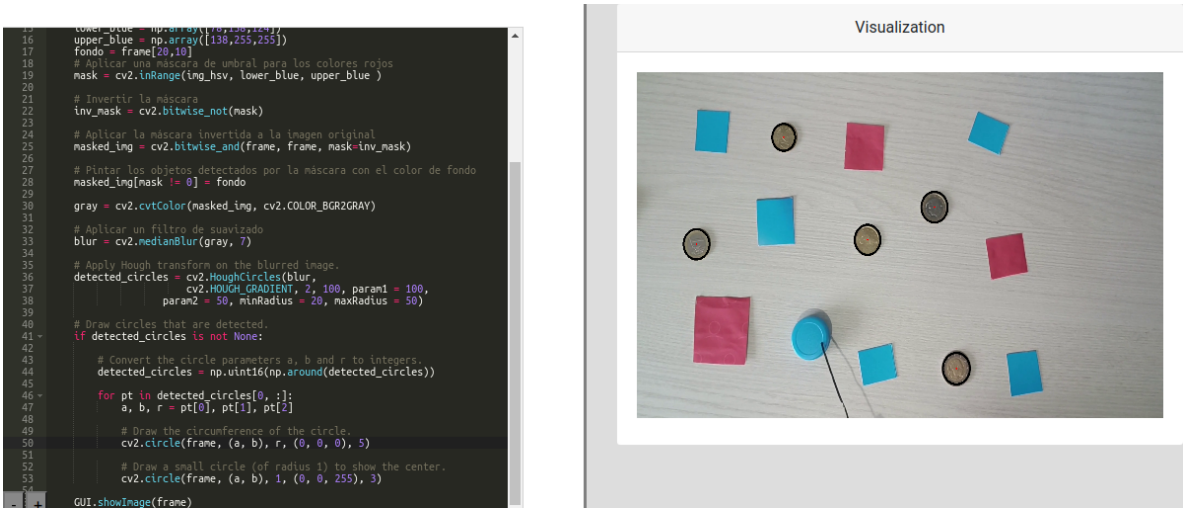

Figura 5.11: Resultado de las monedas encontradas en el vídeo.

```
12
13      img_hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
14
15      lower_blue = np.array([78,158,124])
16      upper_blue = np.array([138,255,255])
17      fondo = frame[20,10]
18      mask = cv2.inRange(img_hsv, lower_blue, upper_blue )
19
20      inv_mask = cv2.bitwise_not(mask)
21
22      # invert mask
23      masked_img = cv2.bitwise_and(frame, frame, mask=inv_mask)
24      masked_img[mask != 0] = fondo
25
26      gray = cv2.cvtColor(masked_img, cv2.COLOR_BGR2GRAY)
27
28      blur = cv2.medianBlur(gray, 7)
29
30      # Find circles
31      detected_circles = cv2.HoughCircles(blur,
32                         cv2.HOUGH_GRADIENT, 2, 100, param1 = 100,
33                      param2 = 50, minRadius = 20, maxRadius = 50)
34
35
36      if detected_circles is not None:
37
38          # Convert the circle parameters a, b and r to integers.
39          detected_circles =
               np.uint16(np.around(detected_circles))
40
```

```
41          for pt in detected_circles[0, :]:
42              a, b, r = pt[0], pt[1], pt[2]
43
44              # Draw the circumference of the circle.
45              cv2.circle(frame, (a, b), r, (0, 255, 0), 2)
46
47              # Draw a small circle (of radius 1) to show the
                    center.
48              cv2.circle(frame, (a, b), 1, (0, 0, 255), 3)
49
50      GUI.showImage(frame)
51      i = i + 1
```

5.2.6. Seguir el círculo en movimiento.

En el vídeo que se usa en este reto, aparece un círculo azul que va moviéndose alrededor de la pantalla hasta desaparecer de la misma. El último reto consiste en seguir el movimiento del mismo[6].

En este caso conseguir la solución requiere una mezcla de varios de los retos anteriores, pues comparte la forma con las monedas y el color con otras formas rectangulares del vídeo. Aunque hay multitud de formas de realizar un procesamiento adecuado para seguir el círculo azul a lo largo del vídeo, en este caso se propone una solución basada en filtros de color, operaciones morfológicas y detección de círculos (ver Listado 5.3).

En primer lugar, se transforma el espacio de color RGB a HSV (ver sección 5.2.1). Las siglas HSV significan Hue, Saturation y Value, y se trata de un espacio de color que se usa mucho en el procesamiento de imagen, también se trata del espacio de color que se asemeja más a la forma en que los seres humanos percibimos el color.

Los componentes del espacio de color son:

- **Hue (Matiz)**: En este espacio de color, la tonalidad se representa en un círculo de 360 grados. Este valor es el ángulo de dicho círculo y representa el color que percibimos, por ejemplo, de 0 a 120 grados se corresponde con los rojos.
- **Saturation (Saturación)**: Es la intensidad que tiene un color medido en porcentaje, donde 0 es el gris y 100 el color más intenso. Por lo tanto cuanto más alto sea este valor, más intensidad va a tener dicho color y cuanto más bajo sea, más apagado va a ser.
- **Value (valor)**: Representa el brillo del color medido en porcentaje, donde 0 corresponde al negro y 100 al blanco. Cuanto más alto sea este valor, más brillante va a ser el color, y cuanto más bajo sea, más oscuro.

[6] https://imgur.com/a/W4RqVUM

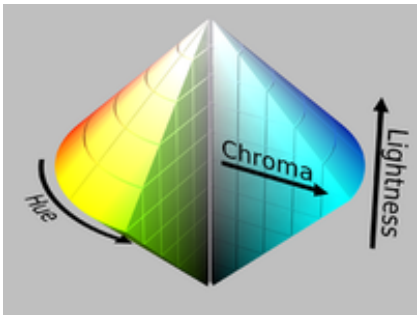

Figura 5.12: Representación espacio de color HSV. Fuente

Aunque el valor de un porcentaje vaya del 0 al 100, OpenCV usa la escala de 0 a 255, ya que normalmente los píxeles de una imagen se presentan con 8 bits. Además utilizar esta escala ayuda a tener más precisión, pues puede distinguir más tonalidades en la imagen.

En la solución propuesta al final se realiza un filtro para mantener los azules y posteriormente se aplican las operaciones morfológicas de apertura y cierre con el objetivo de eliminar objetos muy pequeños y cerrar posibles huecos en los ya detectados.

Con el pre-procesado de la imagen ya realizado, se pueden detectar los círculos existentes en la imagen. Para ello se utiliza la función Hough Circles (ver sección 5.2.5) cuyo resultado va a ser una lista con los círculos detectados. Se recorre después esta lista, y por cada una de sus posiciones se transforman los valores de las coordenadas del centro (x, y) y el radio a números enteros con la función *np.round()*.

Se crea también una máscara del tamaño del fotograma, que se inicializa con todos los píxeles a 0 con la función *np.zeros()*. Por cada uno de los círculos detectados por la transformada de Hough se dibuja un círculo en blanco sobre la máscara usando la función *cv2.circle()* con el mismo centro y radio que el encontrado. Se aplica la máscara a la imagen original usando la función *cv2.bitwise_and()* para limpiar los posibles píxeles que hayan quedado en blanco fuera de los círculos especificados en la lista.

El fotograma con la máscara aplicada (ver Figura 5.13) se convierte a escala de grises y gracias a la función *cv2.cvtColor* se localizan los contornos de los objetos en la imagen y se guardan sobre la variable contours.

Antes de proceder a dibujar un cuadro en torno al círculo en movimiento que le siga durante todo el vídeo, se crean una copia del fotograma original y se recorre la lista que hay en la variable contours uno a uno.

Para cada uno de los contornos en la lista, se calcula el rectángulo mínimo que envuelve lo puede envolver con la función *cv2.minAreaRect()*. Este rectángulo se ajusta al contorno e incluso va rotándose a medida que el objeto se mueva por el vídeo. Teniendo

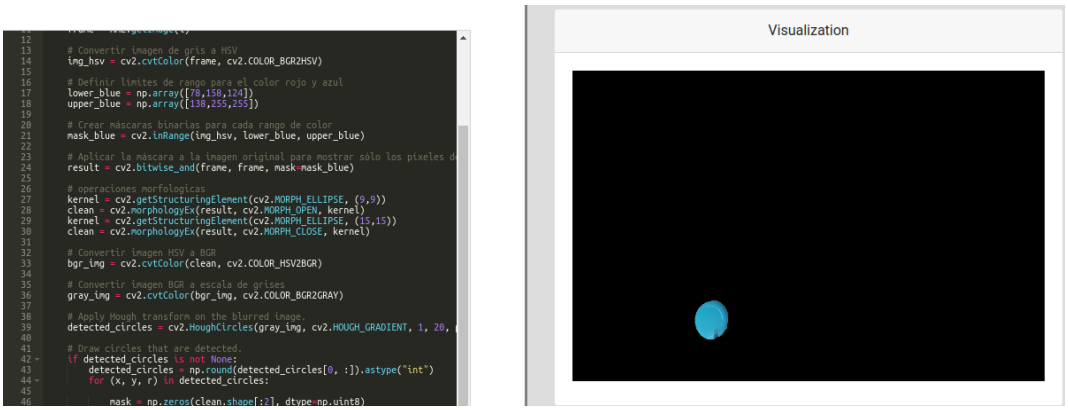

Figura 5.13: Resultado tras aplicar la máscara sobre la imagen original.

ya la posición en que debe pintarse ese rectángulo, se sacan los vértices del mismo con la función *cv2.boxPoints* y se convierten las coordenadas a valores interos.

Como paso final, se dibuja el rectángulo rotado en la copia del frame creada anteriormente con la función *cv2.drawContours* que necesita los vértices del rectángulo como argumento (ver Figura 5.14).

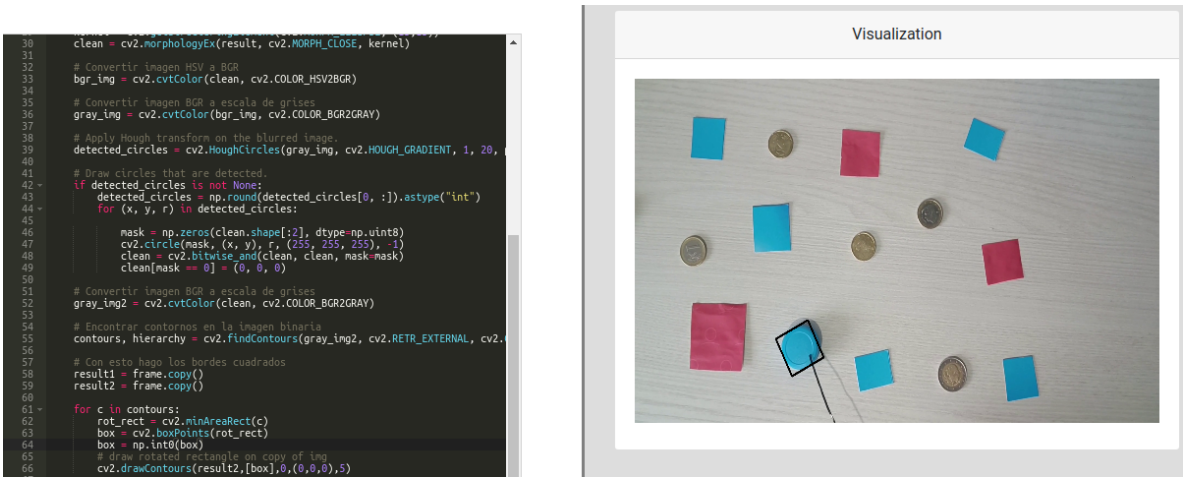

Figura 5.14: Resultado con la solución propuesta.

Listing 5.3: Código solución propuesta para seguir el objeto en movimiento.

```
from GUI import GUI
from HAL import HAL
import cv2
import numpy as np
# Enter sequential code!

i = 0

```

```
while True:
    # Enter iterative code!
    frame = HAL.getImage(i)

    # Convertir imagen de gris a HSV
    img_hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    lower_blue = np.array([78,158,124])
    upper_blue = np.array([138,255,255])

    #apply mask
    mask_blue = cv2.inRange(img_hsv, lower_blue, upper_blue)
    result = cv2.bitwise_and(frame, frame, mask=mask_blue)

    # Clean image
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (9,9))
    clean = cv2.morphologyEx(result, cv2.MORPH_OPEN, kernel)
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE,
       (15,15))
    clean = cv2.morphologyEx(result, cv2.MORPH_CLOSE, kernel)

    bgr_img = cv2.cvtColor(clean, cv2.COLOR_HSV2BGR)

    gray_img = cv2.cvtColor(bgr_img, cv2.COLOR_BGR2GRAY)

    # Find circles
    detected_circles = cv2.HoughCircles(gray_img,
       cv2.HOUGH_GRADIENT, 1, 20, param1 = 50, param2 = 30,
       minRadius = 1, maxRadius = 100)

    if detected_circles is not None:
        detected_circles = np.round(detected_circles[0,
           :]).astype("int")
        for (x, y, r) in detected_circles:

            mask = np.zeros(clean.shape[:2], dtype=np.uint8)
            cv2.circle(mask, (x, y), r, (255, 255, 255), -1)
            clean = cv2.bitwise_and(clean, clean, mask=mask)
            clean[mask == 0] = (0, 0, 0)


    gray_img2 = cv2.cvtColor(clean, cv2.COLOR_BGR2GRAY)

    contours, hierarchy = cv2.findContours(gray_img2,
       cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    result1 = frame.copy()
    result2 = frame.copy()
```

```
52
53      # Paint a rectangle around the object
54      for c in contours:
55          rot_rect = cv2.minAreaRect(c)
56          box = cv2.boxPoints(rot_rect)
57          box = np.int0(box)
58          # draw rotated rectangle on copy of img
59          cv2.drawContours(result2,[box],0,(0,0,0),5)
60
61      GUI.showImage(result2)
62      i = i + 1
```

Capítulo 6

Conclusiones

Una vez descrito todo el contenido del Trabajo de Fin de Grado, en este capítulo se recogen conclusiones sobre los objetivos satisfechos y se proponen líneas de trabajo a futuro, tomando como base los desarrollos realizados.

6.1. Recapitulación de objetivos

En el segundo capítulo de esta memoria se dividían los objetivos principales de este Trabajo de Fin de Grado en tres subobjetivos distintos, que se van a repasar para poder comprobar que los tres se han cumplido satisfactoriamente.

El primero de los subobjetivos consistía en la implementación de pruebas unitarias de frontend. Estas pruebas testean que existan los botones adecuados, que sean tanto visibles como clickables por el usuario y cumplan su función. Como se ha explicado en la sección 4.4 de esta memoria, este objetivo se ha completado mediante el desarrollo de pruebas automatizadas con Selenium y Django.test.

El segundo de los subobjetivos ha sido añadir pruebas de integración completas, cuyo objetivo es simular los pasos y acciones de un usuario real en la página web. Para ello ha sido necesario involucrar todas las piezas que componen Unibotics. Gracias al desarrollo de pruebas completas de integración con el uso de Selenium, Django.test, subprocesos, Docker y ROS Topics ha sido posible completar también este objetivo, cuyos detalles se explican en la sección 4.5

El tercer subobjetivo era la creación de un nuevo ejercicio para RoboticsAcademy. Este ejercicio se trata de una introducción al procesamiento de imagen para aquellos usuarios que deseen probar y comenzar a aprender sobre visión artificial. Con la creación de todos aquellos archivos que un ejercicio necesita (*exercise.py, hal.py, gui.py*, etc), asegurar la compatibilidad entre sistemas operativos al usar un vídeo ya pre-grabado, la creación de este vídeo de entrada y la propuesta de seis retos concretos para guiar al usuario junto a soluciones de referencia, este tercer y último subobjetivo también se

ha cumplido.

A parte de estos subobjetivos, a nivel personal adquirir conocimientos sobre distintas tecnologías que se usan día a día en el mundo laboral también era uno de los objetivos de este Trabajo de Fin de Grado. Antes de comenzar, no tenía ninguna experiencia con muchas de las herramientas usadas como Selenium, Docker o GitHub Actions. O en caso de tener experiencia, como es el caso de Python y Django se limitaba al ámbito académico. Tener la oportunidad de aprender sobre todas estas tecnologías y herramientas, además de realizar desarrollos que contribuyen a mejorar la robustez de Unibotics han supuesto un gran reto.

Este último objetivo puede considerarse cumplido al haber adquirido los conocimientos suficientes para terminar los desarrollos y ser capaz de realizar correcciones sobre las primeras versiones del desarrollo para mejorar control de errores o su eficiencia.

6.2. Trabajos futuros

El desarrollo de una primera versión productiva de pruebas automatizadas tanto de frontend como de integración ha contribuido a la robuztez de la plataforma Unibotics. A partir de estas primeras versiones, se pueden sacar varias líneas de trabajo como:

- **Mejora de la eficiencia en las pruebas automatizadas**. Actualmente el tiempo que tardan en completarse rondan entre 40-60 segundos para las pruebas de integración y 60-90 segundos para las pruebas unitarias de frontend. Por ello una posible línea de trabajo puede ser mejorar la eficiencia computacional de estas pruebas haciendo que se ejecuten en paralelo y no de forma secuencial, por ejemplo.
- **Ampliación de las pruebas**: Conforme la plataforma continúe evolucionando y añadiendo contenidos, será necesario realizar una ampliación de estas pruebas para ser capaces de contemplar nuevas funcionalidades.
- **Soporte a ejercicios Unibotics en REACT**: La plataforma se encuentra en una migración hacia REACT, por lo que una futura línea de trabajo podría ser modificar todas estas pruebas para que funcionen también con REACT.
- **Mejora del ejercicio de introducción al tratamiento de imagen**: Este ejercicio podría modificarse de forma que para el propio usuario sera posible elegir si desea usar su entrada de cámara cuando el sistema operativo es compatible, o si por el contrario desea usar el vídeo pre-grabado como fuente de imágenes de entrada.

Bibliografía

[1] Hostingplus. Funcionamiento y evolución de las aplicaciones web. https://www.hostingplus.com.es/blog/funcionamiento-y-evolucion-de-las-aplicaciones-web/

[2] Selenium. Documentación oficial de Selenium. https://www.selenium.dev/documentation/

[3] "¿Qué es Selenium?" https://sentrio.io/blog/que-es-selenium/

[4] Selenium. https://es.wikipedia.org/wiki/Selenium

[5] "¿Qué es Selenium? https://recluit.com/que-es-selenium/

[6] Django. Documentación oficial de Django. https://www.djangoproject.com/

[7] La historia de Django. https://uniwebsidad.com/libros/django-1-0/capitulo-1/la-historia-de-django

[8] Inazio. Entornos de desarrollo. Modelo - Vista - Controlador (con ejemplo de conversor). http://www.programandoapasitos.com/2015/05/entornos-de-desarrollo-modelo-vista.html

[9] Rafael D. Hernandez. El patrón modelo-vista-controlador: Arquitectura y frameworks explicados. https://www.freecodecamp.org/espanol/news/el-modelo-de-arquitectura-view-controller-pattern/

[10] Docker. Documentación oficial de Docker. https://docs.docker.com/get-started/

[11] ¿Qué es Docker? https://www.oracle.com/es/cloud/cloud-native/container-registry/what-is-docker/

[12] Entender las GitHub Actions. Documentación oficial de GitHub. https://docs.github.com/es/actions/learn-github-actions/understanding-github-actions

[13] HTML: Lenguaje de etiquetas de hipertexto. https://developer.mozilla.org/es/docs/Web/HTML

[14] ¿Qué es YAML? https://www.redhat.com/es/topics/automation/what-is-yaml

[15] Hugo Rodriguez. Qué es OpenCV?: ¡Descubre todo acerca de la visión artificial! https://www.crehana.com/blog/transformacion-digital/que-es-opencv/

[16] Documentación oficial de Unibotics-Webserver. https://github.com/JdeRobot/unibotics-webserver

[17] Universidad Europea. ¿Que es el lenguaje de marca o de marcado? https://universidadeuropea.com/blog/que-es-lenguaje-marca/

[18] Django Testing Docs. Documentación oficial de Django. https://django-testing-docs.readthedocs.io/en/latest/basic-unittests.html

[19] Django Tutorial Part 10: Testing a Django web application. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Testing

[20] Writting and running test. Documentación oficial de Django. https://docs.djangoproject.com/en/4.2/topics/testing/overview

[21] Ferin Patel. How to write tests in Django: Best practices for testing in Django. https://ordinarycoders.com/blog/article/django-testing

[22] RedHat. Tipos de virtualización. https://www.redhat.com/es/topics/virtualization/what-is-virtualization

[23] ComputerWeekly. Virtualización basada en contenedores (virtualización a nivel de sistema operativo) https://www.computerweekly.com/es/definicion/Virtualizacion-basada-en-contenedores-virtualizacion-a-nivel-de-sistema-operativo

[24] Hough Circle Transform. Documentación oficial de OpenCV. https://docs.opencv.org/4.x/da/d53/tutorial_py_houghcircles.html

[25] HSV Color Model in Computer Graphics. https://www.geeksforgeeks.org/hsv-color-model-in-computer-graphics/

[26] Modelo Cliente Servidor. https://blog.infranetworking.com/modelo-cliente-servidor/

[27] El protocolo HTTP. https://blog.makeitreal.camp/el-protocolo-http/

[28] Introducción al frontend y el backend. https://tinyurl.com/2b2tjwe8

[29] What is CI/CD and how does it work? https://www.linkedin.com/pulse/what-cicd-how-does-work-bestarion

[30] Qué es scrum y cómo empezar. https://www.atlassian.com/es/agile/scrum

[31] Contours. Getting Started. https://docs.opencv.org/4.x/d4/d73/tutorial_py_contours_begin.html