Universitat de les Illes Balears

Escola Politècnica Superior

Memoria del Trabajo de Fin de Grado

Sistema de pruebas automáticas para aplicaciones de Smart TV

Pablo González Maya

Grado de Ingeniería Informática

Año académico 2019-2020

DNI del alumno: 43212897Z

Trabajo tutelado por Antoni Oliver y Antoni Bibiloni
Departamento de Ciencias Matemáticas e Informática

Se autoriza a la Universidad a incluir este Trabajo en el Repositorio Institucional para su consulta en acceso abierto y difusión en línea, con finalidades exclusivamente académicas y de investigación	Autor		Tutor	
	Sí	No	Sí	No
	X		X	

Palabras clave del Trabajo: testing, Smart TV, televisor, Arduino, IR.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE FRAGMENTOS DE CÓDIGO

GLOSARIO

API: siglas en inglés de Interfaz de Programación de Aplicaciones. Se trata de un conjunto de definiciones y protocolos para desarrollar software, permitiendo que los distintos módulos o servicios puedan comunicarse entre ellos sin necesidad de conocer su funcionamiento interno. [1]

App: aplicación o herramienta informática.

Comando: cada una de las instrucciones que conforman un test. En él se indica qué es lo que se va a testear, de qué forma y qué acciones realizar tanto si se desarrolla de manera satisfactoria como si no.

Dashboard: panel de trabajo en el que se facilitan instrumentos de gestión a nivel visual.

Endpoint: en el contexto de API, se trata de cada uno de los puntos sobre los que se establece la comunicación. Es decir, es la localización o URL desde la que el sistema accede a los recursos necesarios para realizar una función y posteriormente ofrecer el resultado. [2]

Frecuencia: es la característica que la radiación infrarroja que determina las repeticiones de la señal por unidad de tiempo. En este trabajo, se utiliza para definir la cadena de caracteres hexadecimales que se transmiten a través de dicha tecnología.

Fetch: acceder a un *endpoint*, recopilar la información necesaria y devolverla. Un sinónimo a nivel más práctico podría ser *recoger*.

Hospitality: en el sector del turismo, se trata de la unión del desarrollo de nuevas estrategias de administración, la capacidad de innovación de una empresa y la especialización para conseguir mejorar la experiencia de los clientes.

IR: radiación infrarroja, un tipo de radiación electromagnética.

JSON: es un objeto de datos formado por conjuntos de pares clave-valor.

Log: es la secuencia de mensajes que emite un sistema con información sobre todos los acontecimientos que tienen lugar en el mismo. Algunos de los datos más frecuentes que incluyen son datos sobre el momento en que se encuentra (estado del sistema, fecha u hora) y otras informaciones o errores que puedan tener lugar.

OpenSource: modelo de desarrollo de software basado en la libre colaboración y gratuidad.

Promise: en programación, es el objeto de entrada o salida de una operación asíncrona, en el que se indica si la ejecución ha finalizado correctamente o si en su caso, ha fallado.

SQL: del inglés *Structured Query Language*, se trata de un lenguaje de programación utilizado muy comúnmente para realizar consultas y administrar bases de datos relacionales.

Televisión: transmisión de imágenes a distancia.

Televisor: aparato receptor de televisión.

URL: del inglés *Uniform Resource Locator*, se refiere la cadena de caracteres estándar para la localización de recursos en internet.

Websocket (también utilizado como socket): es la tecnología de transmisión de mensajes bidireccional entre cliente y servidor a través del protocolo TCP.

AGRADECIMIENTOS

En primer lugar, quiero agradecer a mis tutores de Trabajo de Fin de Grado, Antoni Oliver y Antoni Bibiloni por la gran ayuda que me han ofrecido en la elaboración del trabajo, y a SMY Hotels, por la oportunidad de llevar a cabo este proyecto innovador sobre tecnologías con una gran visión de futuro.

Gracias también a Miquel Mascaró Portells por toda la ayuda recibida en relación a las placas Arduino y sus componentes. Así como también, agradecer a Miquel Antoni Llambias toda su disposición y orientación debido a la continuidad que ejerce mi proyecto sobre su TFG.

Finalmente, agradezco a mi familia y amigos todo el apoyo que me han dado a lo largo del proyecto, especialmente en los momentos de mayor carga de trabajo.

ABSTRACT

Este proyecto tiene como objetivo diseñar un sistema de testing automático para aplicaciones de Smart TV. La necesidad aparece debido a la alta demanda en la actualidad de este tipo de dispositivos destinados al turismo, y el creciente desarrollo de aplicaciones para los mismos.

El desarrollo tiene lugar en el laboratorio multimedia LTIM de la Universitat de les Illes Balears, donde se realizan todas las pruebas gracias a que se dispone de un servidor especializado y varios modelos de televisores sobre los que ejecutar las pruebas.

El proyecto consta de tres fases. La primera consiste en la puesta en marcha del receptor de frecuencias por parte del mando original del televisor. En la segunda se realiza el desarrollo de la WebApp de forma que se crea una interfaz de usuario para gestionar los distintos modelos y tests, y se pone en marcha la emisión de frecuencias desde el servidor hacia los televisores. Finalmente, en la tercera fase se plantea el tratamiento de imágenes como solución a la comprobación de la respuesta visual por parte de la aplicación de Smart TV.

En el proyecto se han utilizado diferentes tecnologías, tanto a nivel de Hardware como pueden ser las placas Arduino para hacer la función de receptores y emisores que simulen un mando a distancia; como a nivel de Software, donde se han utilizado tecnologías basadas en JavaScript y programación reactiva, bases de datos Postgres e incluso transmisión de mensajes a través de Socket.io.

La aplicación está dirigida a usuarios que se encarguen de gestionar las pruebas, para los que se han diseñado diferentes módulos, de forma que se puedan gestionar los modelos de televisores que se dispongan, los botones que se quieren simular de cada uno de ellos, los emisores Arduino que se utilizarán para realizar las pruebas, un diseñador de tests visual y un comparador de las ejecuciones de dichos tests.

Se ha realizado además un conjunto de pruebas sobre el sistema para comprobar el correcto funcionamiento tanto para la creación de tests como de la correcta ejecución de los mismos. Finalmente, se ofrece a los futuros usuarios de la plataforma diversos manuales sobre su instalación y uso.

1. INTRODUCCIÓN

El objetivo de este proyecto es el desarrollo de una plataforma de pruebas automáticas sobre aplicaciones para Smart TV. Está enfocado principalmente a testear plataformas desarrolladas sobre soluciones *hospitality*¸ es decir, aquellas aplicaciones que se encargan de brindar facilidades a los clientes principalmente del sector turístico, y en este caso, aplicadas a los televisores de las habitaciones de los hoteles.

Esta plataforma deberá permitir simular un mando de televisor que sea autónomo, de forma que, a base de la inserción de unas reglas y comandos, los tests pueden ejecutarse sin necesitar más interacción humana que la necesaria para realizar la configuración inicial.

El desarrollo tiene lugar en el laboratorio LTIM (Laboratori de Tecnologies d'Informació i Multimèdia de la UIB) y con ayuda de la tecnología ofrecida por SMY Hotels. La instalación hardware necesaria para el proyecto, consiste principalmente en un servidor y dos placas Arduino. En la Figura 1 encontramos un diagrama del sistema, como podemos observar, el servidor aloja una API, de forma que permite recibir y enviar información desde y hacia la WebApp, que será la parte visible para el usuario final, así como también establece la conexión con la Base de Datos, y da servicios a ambos Arduinos. En cuanto a los Arduinos, uno realiza la función de receptor para poder recoger las frecuencias de los mandos de TV y otro hará la función de emisor que será el encargado de enviar los comandos descritos en los tests.

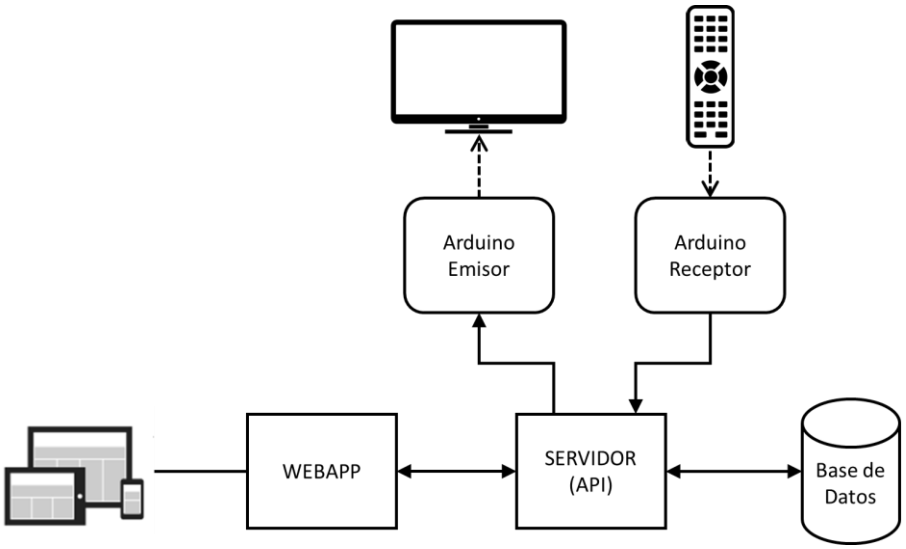

Figura 1: Diagrama de módulos del proyecto

En la *WebApp* encontramos un *Dashboard* con distintos apartados para poder configurar los mandos, los emisores, los tests y las ejecuciones que se hayan hecho. Así, podrán tenerse distintos modelos de televisores sobre los que aplicar las pruebas y se podrán realizar comparaciones de los resultados de los distintos tests que se hayan llevado a cabo.

1.1. ESTADO DEL ARTE

En la actualidad, los televisores convencionales han evolucionado adaptándose a los constantes avances en tecnología. Las Smart TVs están a la orden del día gracias a sus precios cada vez más competitivos y que cada vez ofrecen más posibilidades a los usuarios, ya que al igual que ha ocurrido con los *smartphones* (ya no sólo se utilizan para llamar), los televisores ya no se utilizan solo para ver canales de televisión.

El mundo de la hostelería no se queda atrás, y la instalación de Smart TVs en hoteles es cada vez más frecuente. Estos dispositivos son un nuevo canal de información y comunicación con el cliente, convirtiéndose en una nueva herramienta de marketing y de comercialización de servicios. [3] Tanto es así que en un estudio realizado por PFK Hospitality Research [4] [5], se asegura que un 80% de los encuestados desearían poder conectar sus móviles a los televisores del hotel, así como a un 65% les gustaría poder navegar por internet a través del televisor y un 61% preferiría acceder a información sobre los servicios disponibles en el hotel, como el horario de desayuno, actividades o excursiones, a través del televisor.

Y al igual que aumenta el uso de las Smart TV, también lo hace el desarrollo de aplicaciones para cubrir estas nuevas necesidades. La cadena de hoteles SMY Hotels ya dispone de su propia interfaz que interactúa directamente con el cliente a través de cada televisor de cada habitación. Al estar conectadas a la red interna del hotel, se pueden mostrar mensajes personalizados a los huéspedes, ofrecer la información meteorológica en tiempo real, publicitar las distintas actividades del hotel o incluso permitir hacer compras y videoconferencias. [6]

Así como para el desarrollo de software convencional sí encontramos muchas formas de testear la calidad de las aplicaciones y reducir el número de errores, en el ámbito de las Smart TVs no encontramos ninguna forma eficaz de testear.

Para evaluar estas *apps* se podrían utilizar tecnologías ya existentes como, por ejemplo, efectuar tests de velocidad de carga, tests de usabilidad y accesibilidad, o incluso afrontarse a comprobaciones gramaticales y ortográficas de todo aquello que se muestra en la pantalla. Sin embargo, por ahora no se puede encontrar ninguna tecnología que permita simular los distintos comportamientos del usuario aplicados al uso específico de un televisor.

Por ello, lo más cercano a lo que se pretende conseguir es actuar directamente sobre el televisor con un mando a distancia. Probar un número bajo combinaciones puede ser insuficiente para poder asegurar que la aplicación no producirá errores. Por tanto, resulta necesario hacer un testeo intensivo que pueda durar horas o incluso días, para lo cual, se hace inviable contratar personal que se encargue de pulsar botones de un mando repetidamente.

Gracias a las placas Arduino, es posible preparar un sistema compuesto de sensores de IR, tanto receptores como emisores, que simulen un mando a distancia y que se controle mediante un ordenador o servidor. Esto se consigue debido a las características que nos ofrecen estos microcontroladores que se crearon con intención de ofrecer algo asequible, sencillo de programar y multiplataforma. Nos ofrecen múltiples conexiones tanto de entrada como de salida, así como distintos componentes que podemos encontrar en el mercado que se ajustan a cada modelo de Arduino. Además, al pertenecer a un proyecto OpenSource se puede encontrar infinidad de información y librerías publicadas por la comunidad de desarrolladores y está siendo uno de los pilares para dar paso al Internet of Things. [7] Por todo ello, se nos presenta como una opción eficaz y que no requiere una gran curva de aprendizaje ni un gran desembolso económico para desarrollar el proyecto.

2. DESARROLLO

En esta sección se detallan los aspectos referentes al desarrollo del proyecto, tales como la metodología y tecnologías utilizadas, así como el análisis y diseño previos del sistema.

2.1. DEFINICIÓN DEL PROYECTO

Se plantean tres fases para llevar a cabo el desarrollo del proyecto.

1) Configuración del receptor IR

Consiste en la configuración inicial de un dispositivo Arduino encargado de recibir señales IR del mando original del televisor. Las frecuencias recogidas se envían a un servidor para ser almacenadas en una base de datos. En esta fase se recogen los datos por línea de comandos, y sólo es necesario comprobar la llegada de dichos datos al servidor.

2) Interfaz gráfica y emisor IR

En primer lugar, se desarrolla una interfaz gráfica inicial para la plataforma, en la que se puedan determinar qué modelos se van a manejar y los botones a almacenar de cada uno.

A continuación, se realiza la configuración del Arduino encargado de simular la emisión de frecuencias hacia un televisor. Además, se define la interfaz gráfica que se ofrece al usuario encargado de gestionar los tests. En dicha interfaz se pueden configurar los distintos modelos a utilizar junto con los botones deseados, así como el diseño de comandos de test para realizar las pruebas sobre los televisores. En caso de que varios modelos compartan dichos botones, los tests se podrán reutilizar entre ellos. En el momento en el que el usuario ponga en funcionamiento el test, el Arduino emisor debe enviar cada comando bajo las condiciones definidas en la plataforma, y comprueba el mensaje con el resultado de la acción proveniente del televisor para determinar si se ha completado satisfactoriamente o si debe realizar alguna otra acción, como un reintento de emisión o abortar la ejecución del test. Todos los resultados de las ejecuciones quedarán guardados en la base de datos para poder ser consultados con posterioridad.

3) Tratamiento de imágenes

Una validación añadida que se puede llevar a cabo en el sistema de pruebas es la comparación de imágenes entre las que aparecen en el televisor y una imagen esperada que se haya definido previamente en la plataforma. Este desarrollo se puede plantear de dos maneras. La primera consiste en comparar el código HTML/CSS que se está

mostrando en la pantalla con una plantilla de lo que se espera que aparezca en el televisor. Mientras que de la segunda manera se utiliza una cámara enfocada hacia la pantalla del televisor y con la que se realiza una comparación de imágenes que pueda confirmar la información mostrada (por ejemplo: si en la pantalla aparece un 1, corresponderá con el canal 1). Esta forma se mejora la comprobación a través de *logs* ya que no sólo nos ayuda a saber si el televisor cree estar en la pantalla esperada, sino que también la imagen se corresponde con la que el usuario final debe percibir.

2.2. METODOLOGÍA

Se sigue un desarrollo iterativo, a base de ir perfilando los requisitos y objetivos del proyecto periódicamente. Según se avanza en el desarrollo, se puede ir probando la aplicación para detectar si el funcionamiento está siendo correcto o no. En este caso el prototipado es de tipo vertical [8] [9], ya que se da más prioridad a la funcionalidad de la aplicación que al diseño de esta. De esta forma se pueden ir haciendo pruebas de las funcionalidades deseadas y centrarnos en alcanzar las metas propuestas, sin tener que prestar demasiada atención al diseño, pero permitiendo que pueda cambiarse con facilidad en un futuro. Así se pueden ir probando las diferentes casuísticas y errores que pueden aparecer al usar tanto la aplicación web como las emisiones de las frecuencias en los tests.

Además, se ha hecho uso de un tablero Kanban a través del que se han podido ir detallando las distintas tareas y tener una visión sencilla del transcurso del desarrollo, así como poder poner plazos límite para la implementación de cada tarea y poner más atención en aquellas con más prioridad. En este caso se ha utilizado la herramienta online Trello. [10]

Finalmente, en cuanto al desarrollo, se han utilizado herramientas como GitLab y Google Drive para prevenir que de forma inesperada se pudieran perder todos los avances realizados en el proyecto o la memoria respectivamente.

2.3. TECNOLOGÍAS

A continuación, se detallan las tecnologías utilizadas en el proyecto tanto a nivel de Hardware como de Software.

2.3.1. Hardware

En cuanto a los recursos materiales utilizados para llevar a cabo el desarrollo del proyecto se han utilizado diversos dispositivos que se detallan a continuación.

Ordenador Portátil	
Propósito	Programación
Características principales	Procesador Intel i5 Memoria RAM de 12GB Conectividad WiFi, Ethernet y USB
Precio aproximado	600€

Arduino Uno	
Propósito	Receptor de frecuencias
Características principales	Microcontrolador: ATmega328 Voltaje operativo: 5V Memoria flash de 32KB Conexión Ethernet Sensor instalado: diodo receptor IR
Precio aproximado	30€

Arduino Yun	
Propósito	Emisor de frecuencias
Características principales	Microcontrolador: Atheros AR9331 Voltaje operativo: 5V Memoria flash de 32MB Conexión WiFi Sensor instalado: diodo emisor IR
Precio aproximado	50€

Servidor LTIM	
Propósito	Alojar la plataforma web + API
Características principales	Servidor virtual
Precio aproximado	Se reutiliza de proyectos anteriores

Televisor LG	
Propósito	Realizar pruebas
Características principales	Tecnología Hospitality Reproducción de aplicaciones Smart TV basadas en HTML5
Precio aproximado	500€

2.3.2. Software

2.3.2.1. Node.js

Node.js es un entorno de Javascript, que a su vez se trata de un lenguaje de programación interpretado y orientado a objetos [11]. Node.js además permite manejar eventos asíncronos en aplicaciones de red, lo cual quiere decir que, en contraposición a la programación secuencial, se permite que las funciones devuelvan el control al programa principal antes de haber finalizado completamente su operación. De esta forma, una vez

se inicia la app en el servidor, ésta creará un servidor web al que se podrá acceder a través de una IP seleccionada (hostname + puerto). [12]. Además, este asincronismo nos permite ejecutar funciones que no se bloquean, en contraste con los hilos de los Sistemas Operativos, donde resulta necesario controlar los recursos que consumen estos subprocesos con tal de que la modificación que realiza uno no afecte a la actividad de otro. [13] Una de las peculiaridades de su funcionamiento es que permite realizar la ejecución de Javascript en el servidor, de forma que podremos utilizarlo no sólo para la WebApp sino también para otras aplicaciones como la API y la gestión de sockets.

2.3.2.2. PM2

Los procesos de Node.js pueden terminar inesperadamente o necesitar arrancarse bajo demanda mientras se realiza el desarrollo del proyecto, por ello, resulta interesante poder tener un control de estos procesos a través de un gestor, de ello se encarga PM2. Gracias a él se puede asegurar que la app estará en ejecución siempre que sea necesario (o que se reinicie en caso de que ocurra algún fallo inesperado), obtener el rendimiento sobre el uso de la aplicación, así como realizar algunos ajustes sobre la misma para mejorar su rendimiento. [14] A través del archivo *ecosystem.config.js* que podemos encontrar dentro del proyecto, se definen las directrices y datos básicos que verificará PM2 para su ejecución. En este caso se indica el nombre, la ubicación del script a ejecutar y la ruta base, como se puede observar en el Código 1.

```
Module.exports = {
    apps: [{
        name: 'testserver',
        script: 'dist/index.js',
        cwd: '/home/test/irtest/api/',
        watch: true
    }]
};
```

Código 1: Configuración de PM2

2.3.2.3. Postgres

Postgres es un sistema basado en SQL del cual destaca su gratuidad al tratarse de software libre. Además, es multiplataforma y muy escalable pudiéndose ajustar al número de CPUs y a la cantidad de memoria que disponga el sistema sobre el cual se ejecuta. Además, no tiene problemas en cuanto a concurrencia en el acceso y escritura de los datos, y tiene un soporte total de ACID, es decir: Atomicidad, Consistencia, Aislamiento y Durabilidad,

cuatro requisitos muy importantes a nivel de seguridad para confirmar que los datos no se perderán durante una transacción ni puedan ser interceptados. [15]

Por otro lado, también es interesante poder disponer de un gestor de base de datos de Postgres como puede ser PgAdmin [16], ya que se puede realizar toda la administración de la base de datos de forma visual. Su instalación se hace localmente en el dispositivo desde el que se quiera acceder a la base de datos. Para ello es necesario indicar los datos de acceso a la base de datos, y como en el caso de este desarrollo, al hacerlo dentro de la red privada del laboratorio LTIM es necesario también establecer un túnel SSH para acceder de forma segura.

2.3.2.4. ReactJS

La mayor parte del proyecto está programado en ReactJS, una biblioteca de Javascript que se basa en el paradigma de la programación orientada a componentes. [17] Es decir, la idea es tener una web de una sola página en la que se van mostrando dinámicamente los contenidos cargando únicamente los componentes que correspondan.

2.3.2.5. React-Bootstrap

En este proyecto se ha usado el framework React-Bootstrap [18], que es una adaptación de Bootstrap [19] para usar en el entorno React. Bootstrap es una librería de código abierto que permite aplicar diseños responsive de forma muy fácil, ya que dispone de muchos componentes prestablecidos de forma que se reduce considerablemente el esfuerzo en aplicar estilos CSS o el uso de jQuery en cualquier página web para que sea más agradable a la vista del usuario final.

Gracias a esta adaptación conseguimos reducir aún más las líneas de código necesarias para mostrar componentes como cuadros de texto, tablas, alertas, etc. Para cada componente del proyecto de React que se vaya a renderizar, se podrán importar únicamente aquellos componentes de React-Bootstrap que vayan utilizarse. [20]

2.3.2.6. Express

El paquete Express se trata de una infraestructura de aplicaciones web sobre Node.js. En ella se definen los métodos necesarios para crear una API a la que acceder a través del protocolo HTTP. En dichos métodos se indica el tipo de *method* (GET o POST), la vía de acceso (ruta sobre la que se aplicará el *method*) y la función que se ejecutará cuando se acceda. Este tipo de función se llama función de *middleware*, ya que es la que se encuentra

entre la capa de entrada de datos y el sistema en sí (todas las funciones internas, como llamadas a Base de datos o la comunicación con los Arduinos). [21]

2.3.2.7. Socket.io

La librería Socket.io se encarga de establecer comunicación bidireccional en tiempo real dentro del sistema. [22] Se basa en la emisión de mensajes desde un cliente al servidor o viceversa, como puede observarse en la Figura 2. Una vez se ha establecido la comunicación entre cliente y servidor, la emisión de los mensajes puede realizarse hacia todos los clientes que estén conectados o se puede enviar a un solo consumidor. Se puede asignar un nombre al canal de transmisión de forma que la escucha se abrirá para ese canal y los mensajes llegarán únicamente a los consumidores de ese canal. Esto nos permite gestionar la ejecución de tests simultáneos (cada Arduino emisor tendrá asignado un canal), así como también poder monitorizar en tiempo real las ejecuciones, sin necesidad de ir actualizando la página de visualización de éstas [23].

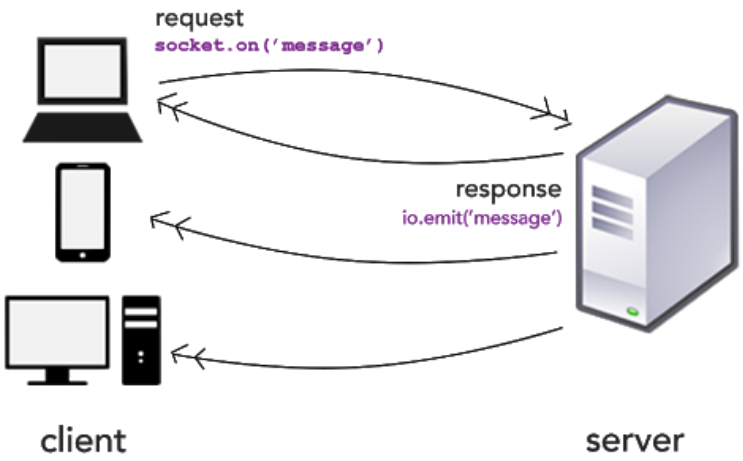

Figura 2: Diagrama de funcionamiento de socket.io

2.3.2.8. Arduino

Arduino utiliza un lenguaje de programación basado en C++ y su estructura básica está formada por dos bloques obligatorios: *setup()* y *loop()*. Al inicio del *sketch*, que es como se denomina a los programas para Arduino, se importan las distintas librerías y se declaran las variables necesarias. En el bloque *setup()* se realizarán las acciones que sólo queramos que se ejecuten una sola vez, al iniciarse el microcontrolador, mientras que en el bloque *loop()* introduciremos todo aquello que queramos que permanezca en ejecución constantemente en forma de bucle infinito. [24]

2.4. ANÁLISIS

En este apartado se lleva a cabo el análisis de los distintos usuarios que usarán el sistema, y los requisitos que se deben satisfacer para cumplir con las expectativas de éstos.

2.4.1. Usuarios

- Gestor de modelos: Persona encargada de configurar los nuevos modelos y mandos de los televisores dentro de la plataforma.
- Diseñador de tests: Persona que define los tests que se quieren hacer sobre cada televisor.
- Gestor de ejecuciones: Persona que ejecuta los tests y que podrá programar su ejecución: tiempo hasta el comienzo de la ejecución, frecuencia…

2.4.2. Requisitos de usuario

ID	RU1
Título	Añadir y gestionar modelos de televisor
Usuario	Gestor de modelos
Descripción	El usuario debe poder añadir, modificar o eliminar los distintos modelos de televisor con los que se desee trabajar.

ID	RU2
Título	Recoger y gestionar códigos
Usuario	Gestor de modelos
Descripción	El usuario debe poder recoger los códigos correspondientes a cada pulsación del mando original del televisor. Además, debe poder modificar o eliminar los códigos que ha recogido en el sistema.

ID	RU3
Título	Definir los tests
Usuario	Diseñador de tests
Descripción	El usuario debe poder definir los tests que dentro del sistema que se encargará de ejecutarlos posteriormente automáticamente. También deberá poder definir las acciones a realizar en caso de error.

ID	RU4
Título	Iniciar los tests
Usuario	Gestor de ejecuciones
Descripción	El usuario debe poder iniciar o programar el inicio de la ejecución de los tests que hayan sido previamente definidos.

ID	RU5
Título	Comprobar resultados
Usuario	Gestor de ejecuciones
Descripción	El usuario debe poder revisar y comparar los resultados de las ejecuciones de los tests realizados.

2.4.3. Requisitos del sistema

Es importante tener en cuenta los requisitos del sistema especialmente en este proyecto, ya que al tener diferentes módulos (receptor, emisor, API…) e incluso una parte externa (los *logs* que proceden de la app del televisor) es necesario tener bien claro qué debe hacer cada módulo y las relaciones entre ellos.

ID	RF1
Título	Gestionar códigos
Descripción	El sistema debe permitir añadir, modificar, eliminar y consultar códigos correspondientes a las señales de los botones del mando de cada modelo de televisión.

ID	RF2
Título	Gestionar tests
Descripción	El sistema debe permitir añadir, modificar, eliminar y consultar los tests que hayan sido introducidos por el usuario.

ID	RF3
Título	Copiar tests
Descripción	El sistema debe permitir hacer copias de un test asignado a un modelo de forma que ese mismo test pueda servir para otro modelo de TV.

ID	RF4
Título	Recibir peticiones de emisión IR
Descripción	El sistema debe poder recibir a través de peticiones HTTP la instrucción de realizar una emisión de una señal por IR.

ID	RF5
Título	Recibir el *log* del televisor
Descripción	El sistema debe poder recibir peticiones provenientes de la televisión con el estado de la misma a través de solicitudes HTTP.

ID	RF6
Título	Comprobar la ejecución de cada comando enviado a la TV
Descripción	El sistema debe poder comparar que todos los eventos que recibe la televisión (o que se han desencadenado) corresponden a los esperados en los tests. Además, debe tener indicaciones sobre qué hacer en caso de que estos no coincidan con lo esperado.

2.5. DISEÑO

En los próximos subapartados se explican las labores de diseño que se han llevado a cabo para definir con más precisión el sistema deseado. Se analizan las distintas entidades presentes en el sistema, así como los casos de uso y módulos que se extraen de éstos. Finalmente se muestra el diseño de los circuitos electrónicos que se deberán construir para poner en funcionamiento las placas Arduino.

2.5.1. Modelo de datos

En la Figura 3 se puede observar el modelo de datos utilizado para este proyecto. En él quedan plasmadas las relaciones entre entidades, utilizando la notación UML.

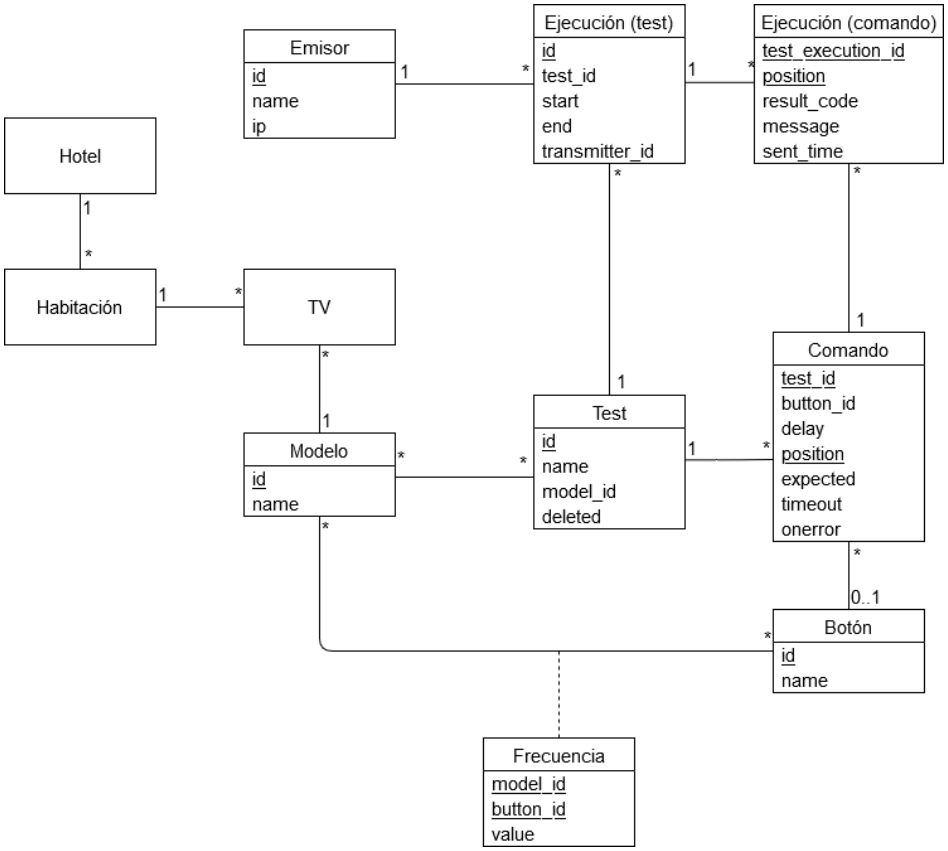

Figura 3: Modelo de datos del proyecto

2.5.2. Entidades y diccionario de datos

Las entidades Hotel, Habitación y TV no están directamente implicadas en el proyecto, ya que proceden de la implementación previa de la app del televisor [6]. Sin embargo, pueden ser de ayuda para poner en contexto las demás entidades.

Hotel: edificio o complejo constituido por un conjunto de habitaciones.

Habitación: habitáculo en el que se aloja el cliente final. Cada habitación se encontrará dentro de un hotel y puede tener uno o varios televisores.

TV: entendido como el televisor que hay en una única habitación y al cual le corresponde un identificador de modelo.

Modelo: indica el modelo asociado a la TV (estará formado principalmente por la marca del televisor y el modelo en sí). Esta propiedad corresponde con el atributo *name* y

además se le asignará un identificador único. Cada modelo tendrá una serie de botones definidos y los tests con los que se quiera interactuar sobre él.

Botón: se trata de cada uno de los botones que conforman el mando del televisor que se desea simular. Este estará definido por un nombre que corresponde con el atributo *name* y un identificador único. Podrá aparecer un mismo botón en varios modelos de televisión. Algunos ejemplos son: *power*, subir volumen, cambiar canal…

Frecuencia: es única para cada par modelo-botón. Consiste en el código que corresponde a la señal de IR que deberá emitir el mando virtual para realizar la acción deseada. Este código se almacena en el atributo *value* y además se deberá indicar para cada uno, el identificador tanto del modelo como del botón.

Comando: es cada una de las líneas que forman un test. Este comando deberá corresponderse con un único test indicado en el atributo *test_id* y su posición será única dentro del test. Además, se deberán indicar otros atributos necesarios para su ejecución como el identificador del botón que se desea emitir (*button_id*), el tiempo de espera hasta su emisión (*delay*), el tiempo de espera de respuesta por parte del televisor (*timeout*), el mensaje esperado (*expected*) y la acción a realizar en caso de que ocurra algún error (*onerror*).

Test: es el conjunto de comandos que deberán ejecutarse de forma secuencial para probar el funcionamiento de la aplicación del televisor. Los atributos que lo constituyen son el modelo sobre el que se debe aplicar (*model_id*), el nombre del test, la fecha de eliminación (si procede) y un identificador único.

Ejecución: las ejecuciones se generarán cada vez que se realice un test sobre un televisor. Estas se dividen en dos entidades, una para los tests ejecutados y otra para los comandos ejecutados en cada test. En el primer caso, se debe indicar el identificador del test que se realiza (*test_id*), el emisor que se utiliza (*transmitter_id*), el momento de inicio del test (*start*) y el momento en que tenga lugar la finalización (*end*). Por otro lado, para el caso de los comandos, se puede conocer de cada uno el test que lo ha ejecutado (*test_execution_id*), la posición dentro del mismo, el resultado (satisfactorio, respuesta incorrecta o sin respuesta), el mensaje que ha recibido por parte de la *app* del televisor y la fecha y hora del momento en que ha tenido lugar la ejecución del comando.

Emisor: se trata de la placa Arduino encargada de emitir las frecuencias en la ejecución de un test. Para su funcionamiento es necesario ponerle un nombre y la IP correspondiente, así como también se le asignará un identificador único.

2.5.3. Casos de uso

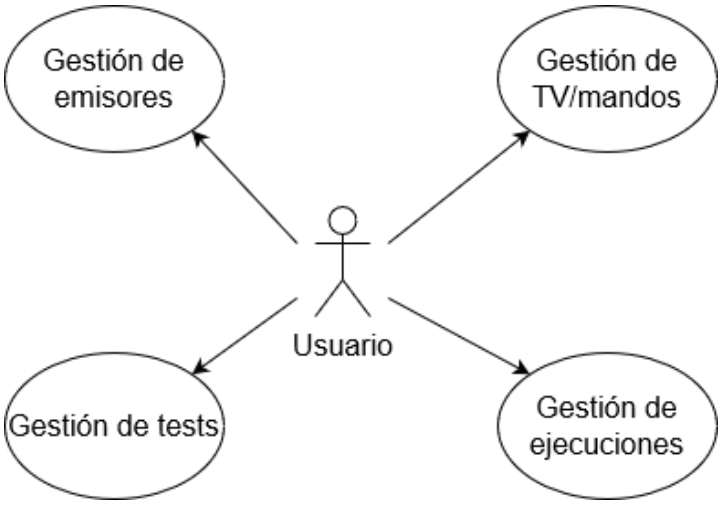

Figura 4: CU1: Home

En el primer caso de uso, al que hace referencia la Figura 4, observamos que el usuario podrá acceder a 4 opciones que se corresponden con los diferentes casos de usos que se detallan a continuación.

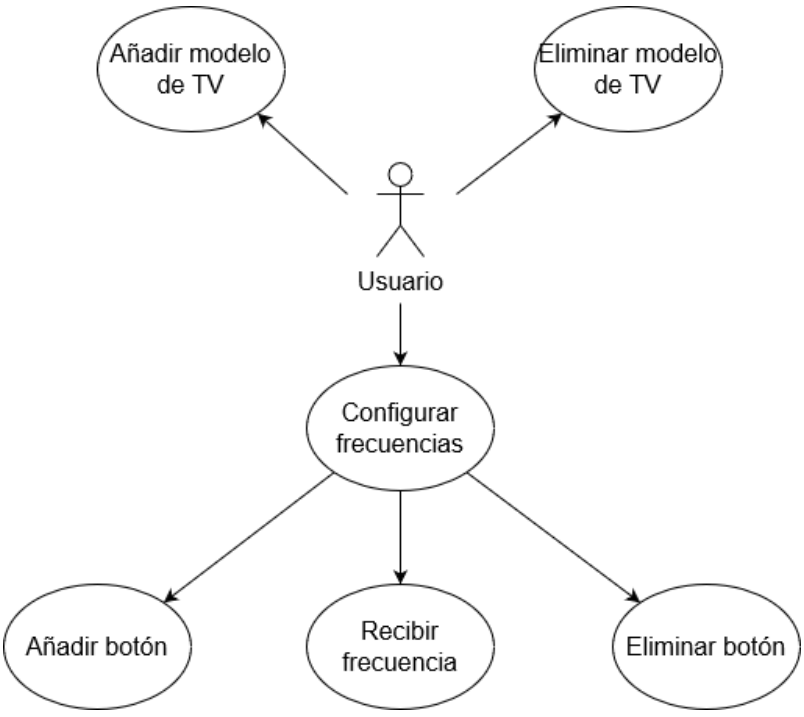

Figura 5: CU2: Configurar mandos

En el segundo caso de uso, al que hace referencia la Figura 5, el usuario podrá añadir o eliminar un nuevo modelo de televisor, así como configurar las frecuencias de uno de ellos. En este último caso, el procedimiento será añadir el botón que se quiera configurar y recibir la frecuencia correspondiente desde el Arduino receptor. Finalmente, también se da la opción de eliminar uno de los botones ya registrados.

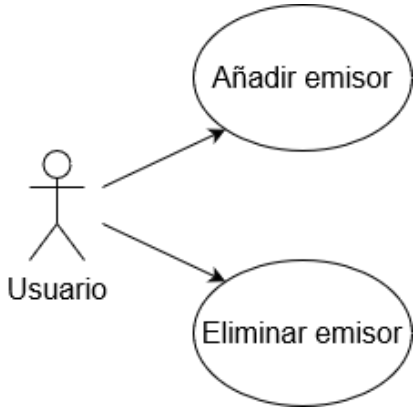

Figura 6: CU3: Gestión de emisores

El tercer caso de uso, al que hace referencia la Figura 6, contempla la gestión de emisores por parte del usuario. Desde este punto solo tendrá dos opciones, añadir un nuevo emisor o borrar uno ya existente.

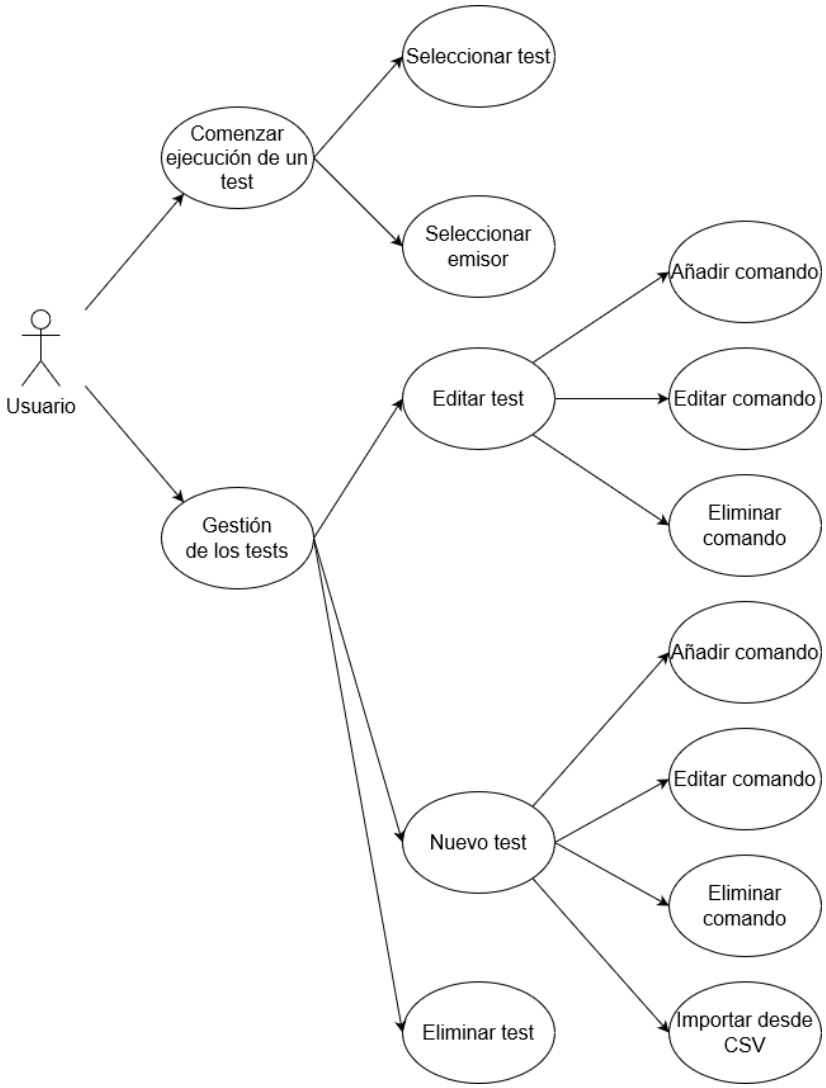

Figura 7: CU4: Gestión de tests

El cuarto caso de uso, al que hace referencia la Figura 7, se podría considerar el más relevante, ya que es el que permite al usuario gestionar los tests. El proceso habitual será crear un test añadiendo o editando los comandos que considere, lo guardará en base de datos y procederá a su ejecución seleccionando el emisor deseado. Además, el usuario también podrá editar o eliminar tests e incluso importar ficheros CSV para crearlos.

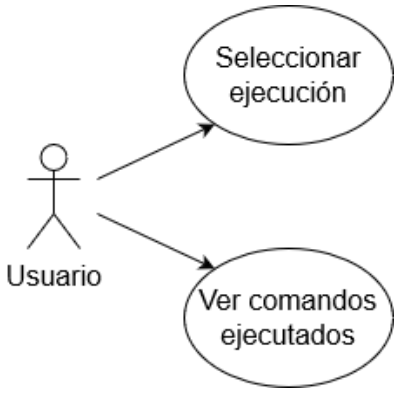

Figura 8: CU5: Gestión de ejecuciones

Por último, en el caso de uso al que hace referencia la Figura 8, el usuario podrá gestionar las ejecuciones de los tests, cuyo procedimiento consiste en seleccionar alguna de las ejecuciones, ya sea una ejecución pasada o una que esté teniendo lugar en el momento de la consulta). El usuario podrá observar los comandos que se han ejecutado, o que se están ejecutando, junto a su resultado y momento de ejecución.

2.5.4. Diseño modular del sistema

En esta sección se definen los distintos módulos que componen el sistema. Es decir, profundizaremos en la función de cada módulo que aparece en la Figura 1.

La parte del servidor consta de un sistema de API-REST en el que se definen diversos *endpoints* que utiliza la plataforma para interactuar con la Base de Datos y los Arduinos. Estos *endpoints* constituyen las funciones CRUD (crear, leer, actualizar y borrar) de las diversas entidades que encontramos en el Modelo de Datos de la Figura 3, por ejemplo, los modelos de televisor, los botones y frecuencias correspondientes a cada uno y los tests con sus comandos.

En cuanto a la parte a la que accederá el usuario, consiste en una *WebApp* tipo *Dashboard* desde donde se podrá controlar de forma visual toda la información que gestiona el servidor. Dispondrá de los siguientes apartados:

a) Gestión de TV/mandos: en este apartado el usuario podrá activar el reconocimiento de frecuencias. La parte visual consiste en una tabla en la que previamente se haya indicado el modelo de televisor, se debe indicar el nombre que se le va a dar a esa frecuencia, es decir, el nombre del botón, y para cada uno se puede activar la escucha a través de socket.io para recibir las frecuencias en tiempo real como podemos observar en la Figura 9. Gracias al Arduino receptor,

la frecuencia llegará a la API que se encargará de mostrar al usuario la frecuencia recibida a través del canal establecido de socket.io.

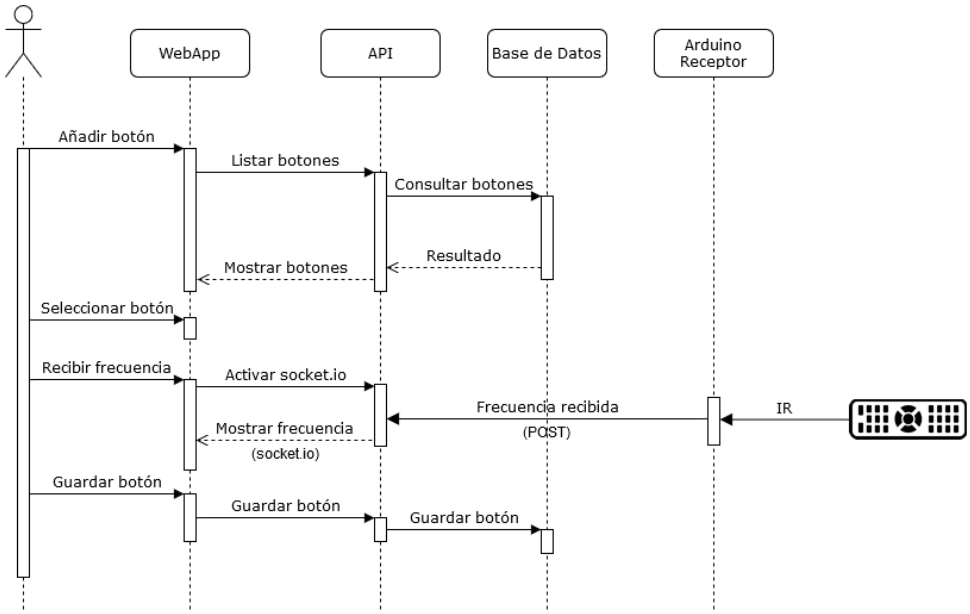

Figura 9: Diagrama de secuencia para la recepción de frecuencias

b) Gestión de emisores: en este apartado se pueden configurar distintos emisores que se encargarán de realizar los tests. En ellos se indicará el nombre e IP correspondiente. De esta manera, se podrán realizar varios tests simultáneamente usando diferentes Arduinos.

c) Gestión de tests: en este apartado se pueden crear, visualizar, editar, eliminar y poner en marcha los tests. Para diseñar un test, el usuario deberá haber introducido previamente todos los botones que quiera simular en dicho test. Cada uno de los botones, supondrá un comando del test, en el que se indicará además el tiempo de espera antes de la ejecución de dicho comando, que es el tiempo durante el cual se esperará una respuesta por parte del televisor; la respuesta esperada, y la acción a realizar en caso de que la respuesta no coincida o no haya llegado ninguna. En este último campo, se puede especificar si se desea continuar a pesar del error, reintentar la ejecución del comando o abortar el test.

   Una vez se haya puesto en marcha un test, indicando previamente el modelo de televisor sobre el que se va a realizar la prueba, se podrá visualizar el transcurso en tiempo real de su ejecución en el siguiente apartado.

d) Gestión de ejecuciones: en este apartado se pueden visualizar las ejecuciones de los tests tanto de los que están en marcha como los que ya están finalizados. De esta forma si se realizan distintas ejecuciones del mismo test se podrán comparar

los resultados. También están disponibles los resultados para cada uno de los comandos ejecutados, las frecuencias que se han emitido en cada caso y el momento exacto en el que se ha llevado a cabo, así como el mensaje que ha devuelto el televisor. En la Figura 10 se puede observar la secuencia de acciones que se realizan cada vez que se ejecuta un test.

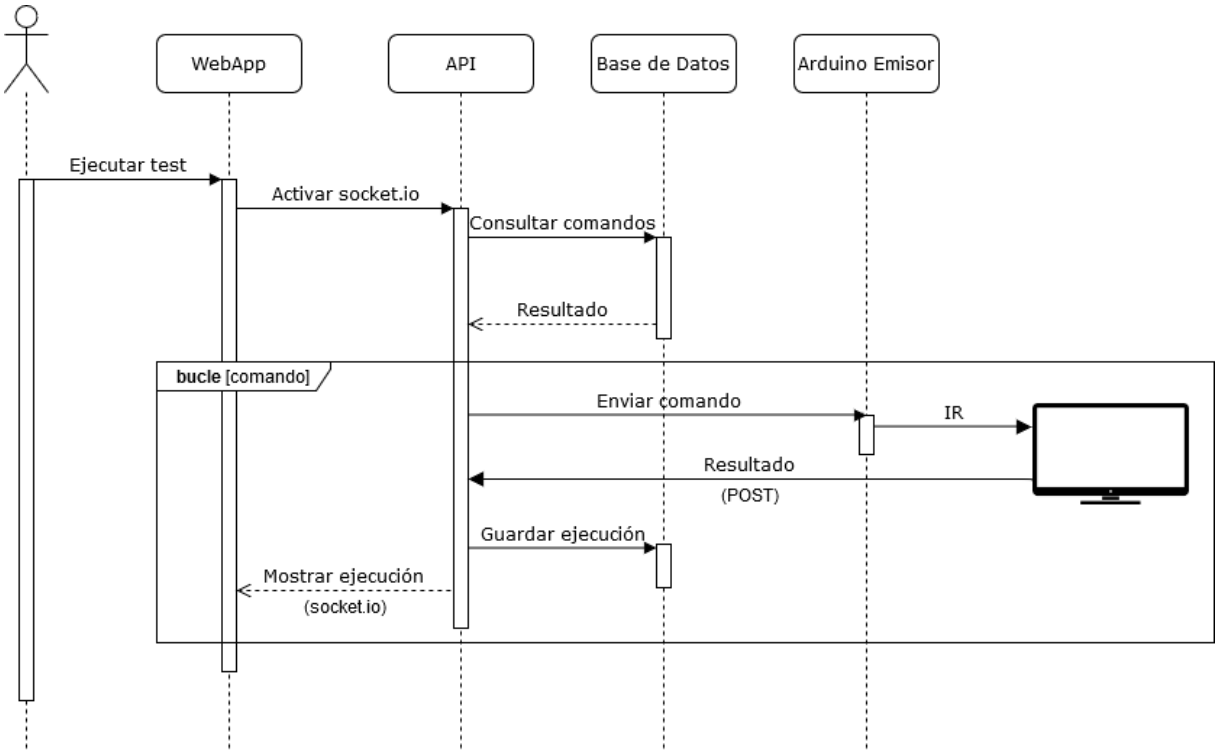

Figura 10: Diagrama de secuencia para la ejecución de tests

2.5.5. Circuitos Arduino

Esta sección describe el diseño de los circuitos utilizados en los Arduinos.

El Arduino encargado de emitir las señales IR que podemos observar en la Figura 11 utiliza un diodo LED emisor de infrarrojos, que en ese caso ya lleva integrada una resistencia de 220 Ohm para limitar la corriente.

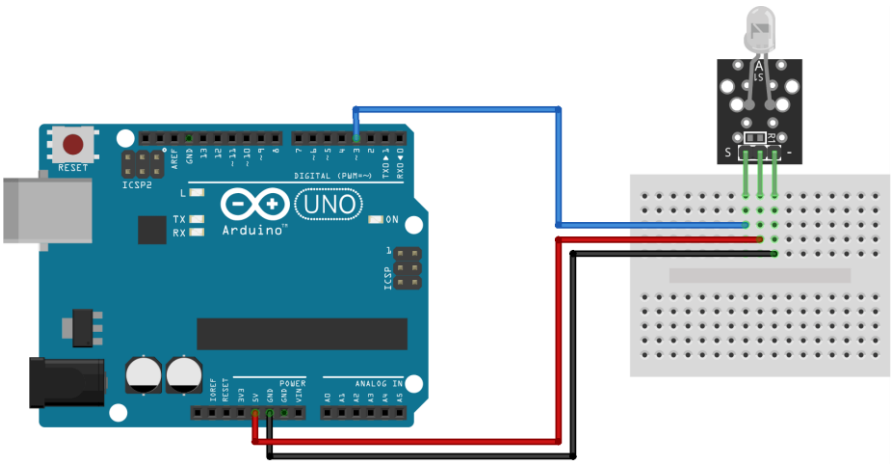

Figura 11: Arduino emisor

El Arduino que recoge las frecuencias del mando original del televisor que podemos observar en la Figura 12 utiliza un sensor IR que podrá estar alimentado por una tensión de hasta 5 V. Su interior contiene un demodulador que buscará señales moduladas a una frecuencia de aproximadamente 38 KHz (cuanto más alejada, más disminuirá la sensibilidad) y cuyo máximo color de LED será de 940 nm. [25]

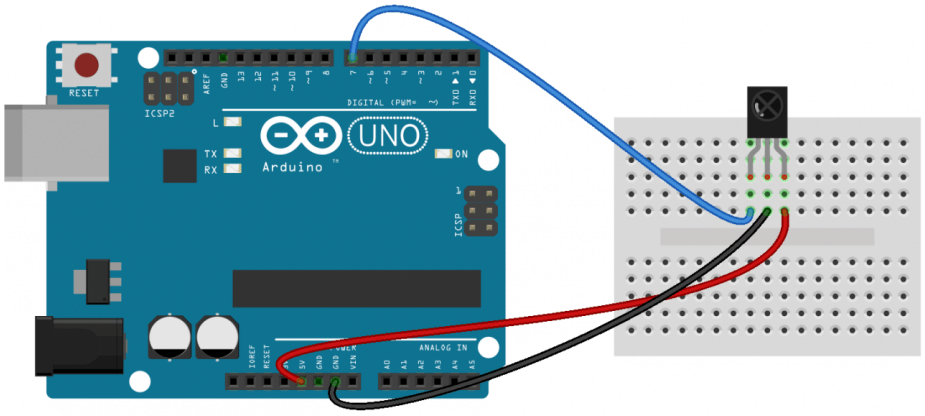

Figura 12: Arduino receptor

3. IMPLEMENTACIÓN

En esta sección se explica cómo se ha llevado a cabo la implementación del sistema a nivel de programación, tanto de los Arduinos como de la API y la *WebApp*. Estas dos últimas se han englobado en el mismo proyecto, pero su desarrollo tiene lugar en dos carpetas separadas: la carpeta *api* y la carpeta *webapp* respectivamente.

3.1. ARDUINOS (SOFTWARE)

Para el desarrollo del software que se ejecutará en los microcontroladores Arduino, se han utilizado los bloques *setup()* y *loop()* mencionados en el apartado 2.3.2.8 de este documento. Además, puede consultarse el código fuente completo utilizado en ambos Arduinos en el Anexo I.

3.1.1. Arduino emisor

En cuanto a la programación necesaria para poner en marcha el dispositivo, son necesarias varias librerías, y se seguirá el modelo indicado en la Figura 13.

Haciendo uso de las librerías *Bridge*, *YunServer* y *YunClient* podremos disponer de un endpoint a través del cual recibir mensajes como parámetro dentro de la URL.

En el bloque *setup* del código se inicializa la transmisión de datos entre la placa y los distintos componentes que haya conectados, en este caso, el emisor IR. También se realiza la llamada a las funciones *listenOnLocalhost()* y *begin()* del objeto *server*, de esta forma se permite la entrada de datos a través de la IP que haya asignado el router al Arduino. En este caso, el endpoint utilizado tendrá el nombre "emit" y será accesible desde la URL *http://[IP_del_Arduino]/arduino/emit/*. Para gestionar la llegada de códigos entrantes, iniciaremos el cliente a través de la funición *server.accept()* dentro del bloque *loop*. Como el dato que nos interesa va incluido en la URL, simplemente se deberá extraer la parte de la cadena de dicha URL, y se enviará a través del emisor IR conectado al PIN 3 utilizando el comando *irsend.sendNEC()* de la librería *IRremote*.

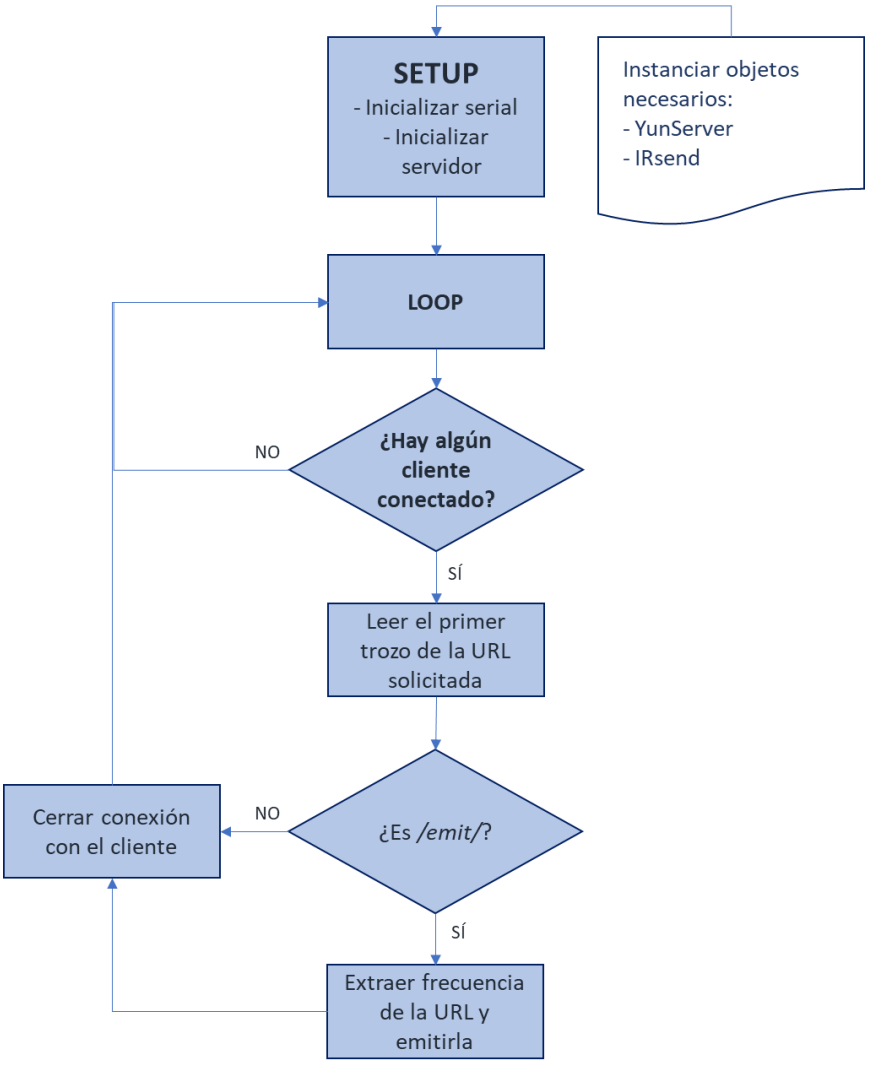

Figura 13: Diagrama de flujo del emisor

3.1.2. Arduino receptor

El código cargado en este Arduino consiste inicialmente en establecer la transmisión de datos, iniciar la conexión Ethernet (indicando la MAC del dispositivo, la IP deseada, el servidor DNS y la puerta de enlace) y habilitar el receptor de IR a través de la llamada al método *enableIRIn()* de la librería *IRrecv*. En el bloque *loop* se irá recibiendo constantemente la información que llega a través del receptor IR conectado al PIN 6. Como se puede ver en la Figura 14, cada vez que se recibe una frecuencia, es decir, cada vez que la función *decode()* detecte algún valor válido, se llama a la función *sendIR()*, que es la encargada de enviar ese contenido a través del cliente hacia el endpoint *http://[IP_del_servidor]:3501/hotel/test/ir?code=[código]*.

Para enviar estos datos, tendremos que indicar desde el Arduino tanto el método (POST, en este caso), la URL a la que se desea enviar y todas las cabeceras de la conexión. Esta

transmisión de datos se realiza utilizando el método *print()* del objeto *client* como muestra el Código 2.

```
client.print(String("POST ") + "/hotel/test/ir?code=");
client.print(results.value,HEX);
client.print(" HTTP/1.1\r\n");
client.print("Host: 192.168.2.88:3501\r\n");
client.print("Connection: close\r\n");
client.print("Content-Length: 0\r\n");
client.print("Content-Type: application/json;charset=UTF-8\r\n\r\n");
client.println();
```

Código 2: Comunicación Arduino-API

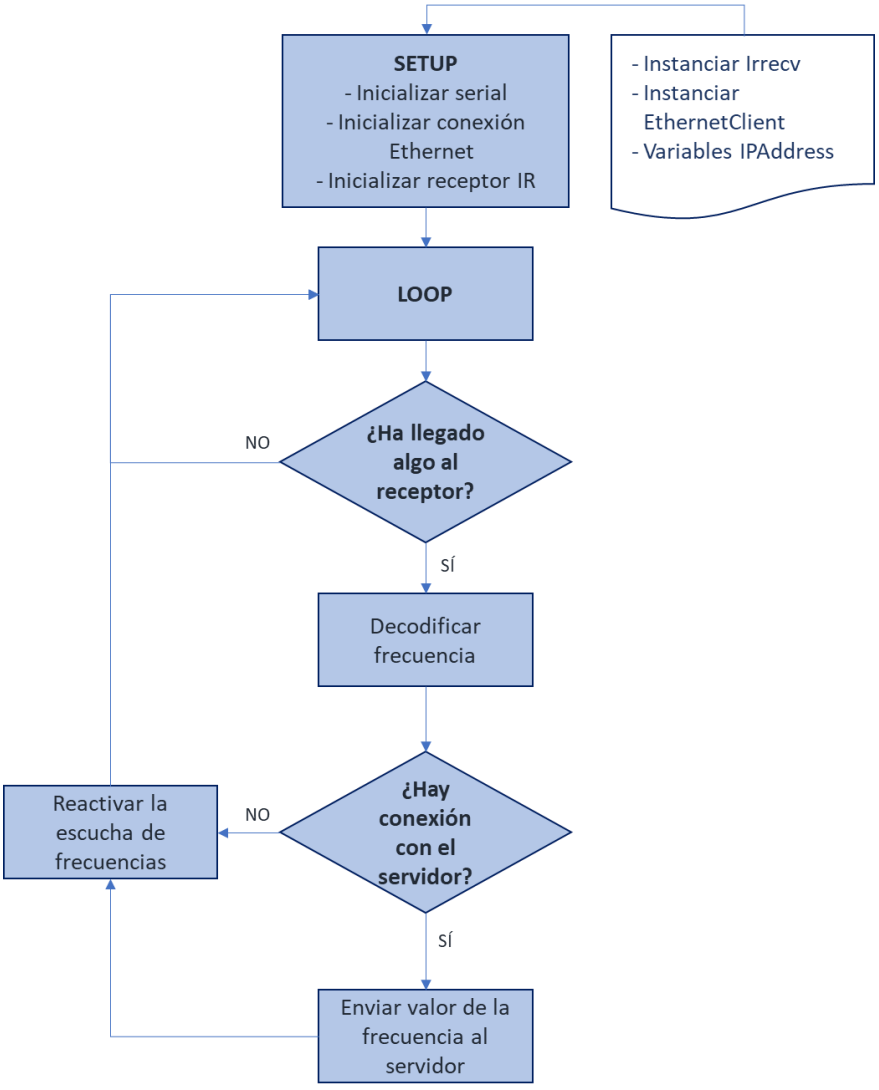

Figura 14: Diagrama de flujo del receptor

3.2. SERVIDOR

La parte del servidor del proyecto se encuentra en la carpeta */api/* ya que utilizaremos este tipo de interfaz de programación. Las respuestas a las consultas que se realizan a través de esta API se devuelven en formato JSON.

Módulo /api/

∟ config.js: primero encontramos un archivo de configuración en el que se define todo aquello relativo a la apertura del servidor para permitir conexiones entrantes. Por tanto, encontramos distintas variables globales como el host, puerto y proxy correspondiente para dar visibilidad a los endpoints del servidor, así como la configuración de la base de datos (puerto, nombre, usuario y contraseña) que se utilizará en diferentes puntos del servidor para establecer la conexión con la misma.

∟ index.js: se definen los distintos endpoints de entrada del servidor (GET/POST). Además, al comienzo de este archivo es donde se definen los paquetes/librerías que se van a usar para que esta parte del proyecto funcione.

∟ db.js: En este fichero se definen todas las funciones que harán consultas en la base de datos, ya sean inserciones, consultas, actualizaciones o borrados. La conexión con la base de datos se realiza a través del componente *Pool* de la librería *pg*. Para ello se crea un nuevo objeto (*new Pool()*) recuperando los datos del archivo de configuración (config.js). A continuación, definiremos cada función en la que se realizarán consultas tipo SQL a través del método (*pool.query()*). Podemos encontrar algunas de ellas más complejas en las que se realizan varias consultas para poder construir los objetos de datos a retornar con tal de que tengan todos los datos que necesitará posteriormente la parte de la *WebApp*.

∟ ir.js: Se trata del fichero en el que se definen las funciones que utilizarán los emisores o receptores IR. En el caso de la ejecución de tests, se recuperará la IP del Arduino emisor y los comandos que se deberán ejecutar. Al inicio de la ejecución registrará la nueva ejecución en la base de datos y entrará en el bucle que recorrerá cada comando a ejecutar. Para cada uno de ellos, realizará la petición a la URL del Arduino emisor indicando la frecuencia y, se encargará además de esperar a recibir los *logs* del televisor. Comprobará el mensaje que llegue con el que se haya indicado como esperado en el comando. De esta manera, registrará en base de datos si ha coincidido, si el mensaje que ha llegado no coincide o incluso si se ha acabado el tiempo de espera sin obtener una respuesta. Una vez finalizada la ejecución del test completo, registrará la fecha de finalización del test en base de datos.

3.3. WEBAPP

Módulo /webapp/

∟ index.js: en él se indica que se deberá renderizar aquello que haya dentro de la etiqueta *container* dentro de la "App" especificada.

∟ App: es un componente de React en el que se indican las distintas rutas a las que se puede acceder, con el componente que le corresponde.

∟ Menubar: la barra de navegación que aparece en la parte superior de todas las páginas del proyecto. Se centraliza para evitar repeticiones de código innecesarias.

∟ Components: en esta carpeta se encuentran todas las páginas que forman la "parte visible" del proyecto. Cada una consistirá en el código que se deberá renderizar, los estados iniciales con los que se trabajará y se llamará a las funciones que nos ayudarán a mostrar la información requerida en la pantalla. Cada uno de ellos se analiza en el siguiente apartado.

3.3.1. Componentes

Un componente es cada uno de los módulos que forman el *Dashboard* de la *WebApp*, que coinciden con los casos de usuario definidos en el apartado 2.4.2 de este documento.

AddModel.jsx

Este componente es el responsable de facilitar la gestión de modelos de TV guardados en base de datos. Se cargará al acceder a la URL /addModel. En primer lugar, hará un *fetch* de los modelos que haya en base de datos (endpoint /getModels del API) y los guardará en el estado (state). Una vez estén cargados (la variable *isLoaded* será true), se mostrará una tabla con los modelos guardados, el número de frecuencias/botones que están configurados para cada uno y las opciones de eliminar y añadir nuevos modelos. Cuando se elimina un modelo, se borra directamente de la base de datos y se muestra la lista actualizada. Sin embargo, cuando se añade un modelo o se edita el nombre de uno, los cambios se aplicarán cuando se haga clic en *Save changes*. En ese momento, se ejecutará un bucle que recorrerá el array de modelos buscando todos aquellos que tengan la propiedad *hasChanged* con el valor *true*, en ese caso verificará si tiene un *id* asignado, en caso afirmativo realizará un *UPDATE* en base de datos con el nuevo nombre del modelo, y en caso contrario realizará un *INSERT* del nuevo modelo.

AddFreq.jsx

Este componente es el responsable de facilitar la gestión de los botones y frecuencias de un modelo de televisor. Se cargará al acceder a la URL /config/*[modelo]*. En primer lugar, cargará los botones que ya existan en base de datos para el modelo en cuestión (desde el endpoint *getFreqs* y metiendo el resultado de la consulta en el array *freqs* del estado del componente), así como todos los nombres de botones que ya existan (desde el endpoint *getButtons* y metiendo el resultado de la consulta en el array *preButtons* del estado del componente). De esta forma, podremos añadir botones para ese modelo basándonos en un nombre de botón que ya exista o añadir un nuevo botón desde cero (mostrando un pequeño formulario pasando a *true* el estado *showNewButtonForm*). Una vez añadido el botón a la lista podremos recibir su frecuencia. Esto se consigue poniendo el id de ese botón al estado *listening*. En ese momento, el próximo mensaje que llegue a través del canal de socket.io llamado *messages* (a través del método *socket.on()*) se introducirá en el campo de frecuencia del id indicado. Y en ese momento se cerrará la comunicación a través de ese canal.

CreateTest.jsx

Este componente es el responsable de facilitar la creación de tests para un modelo concreto. El modelo deberá indicarse en la URL. Lo que hará este módulo es hacer un *fetch* a la URL */[modelo]/getFreqs* e introduce el resultado en el estado *buttons*. De esta forma, se renderizará una lista con todos los botones que hay configurados para ese modelo de televisor. Estos se podrán ir añadiendo al estado *selectedButtons* que serán los que formarán el test. Este array contendrá cada objeto *button* que tendrá los atributos necesarios para cada instrucción del test (nombre del botón, timeout, qué hacer en caso de error, retraso hasta su ejecución y el mensaje esperado por parte del televisor) y se mostrará visualmente en formato de tabla, en la que se podrán editar todos los campos y se podrá mover la posición de cada comando.

Además, tiene una opción extra que es *Importar desde CSV*, que utiliza la librería *react-csv-reader* [26] para poder subir un archivo CSV que contenga en la primera fila las cabeceras: name, timeout, onerror, delay, expected.

Al pulsar *Save* se realizará el guardado en base de datos a través de la llamada POST al endpoint */[modelo]/newTest* y aparecerá un mensaje avisando de que se ha guardado correctamente.

EditTest.jsx

Este componente es el responsable de facilitar la edición de tests. Para acceder al mismo hay que acceder a través de la URL */edittest/[modelo]/[ID del test]*. Este componente realizará un *fetch* de tres *endpoints* distintos para poder ser completamente funcional. El primero es */[modelo]/getFreqs* para sacar las frecuencias del modelo de televisor (es decir, los botones que podremos añadir o quitar del test) y meterlas en el array *buttons* del estado del componente, también */[modelo]/getTests* en cuyo resultado se buscará el nombre del test gracias a su ID y se introducirá en el estado *testName*. Finalmente, recuperará los comandos del test a través del endpoint */[modelo]/getCommands/[ID del test]* y los introducirá en el estado *selectedButtons*. En este último proceso también encontramos un proceso añadido, y es que en base de datos solo se dispone de la frecuencia del botón para cada comando, y visualmente resulta más *user-friendly* tener el nombre del mismo, por tanto, como previamente se han cargado las frecuencias con los nombres correspondientes para el modelo, se puede realizar una búsqueda en el estado *buttons* para añadir el nombre e id del botón en cada comando. Una vez esté todo cargado, se mostrarán las dos tablas que encontramos en CreateTest: una con la lista de botones disponibles para añadir al test, y otra con los comandos ya presentes en el test sobre los que podremos editar todas las opciones, eliminarlos o moverlos. Un dato importante a destacar es que al actualizar un test el procedimiento que se lleva a cabo en la base de datos es poner la fecha de borrado en el campo *deleted* del test actual y se crea un test nuevo. De esta forma podremos seguir consultando las ejecuciones pasadas de un test a pesar de estar modificado.

EditTransmitter.jsx

Este componente es el responsable de permitir la gestión de emisores (Arduinos) con los que se realizan los tests. Al cargar el componente se hará un *fetch* del endpoint */getTransmitters* de forma que una vez se hayan cargado en el array *transmitters* del estado, se mostrará una tabla en la pantalla desde la que podremos eliminar un emisor o añadir nuevos. Al añadir emisores, se podrá editar tanto su nombre como la IP correspondiente y se irán guardando en el array *newTransmitters* del estado. Una vez se hayan guardado en base de datos los nuevos emisores, pasarán al array de *transmitters* utilizando la función *concat* y en ese momento no se podrán modificar (únicamente borrar). Al borrar un emisor, se pedirá confirmación del usuario y en caso afirmativo, se eliminará directamente de base de datos y se eliminará también del array *transmitters*.

ExecutionCommands.jsx

Este componente es el responsable de permitir la visualización las ejecuciones que se han realizado de un test. Al cargar este componente se hace un *fetch* del endpoint */getExecutedCommands/[ID de la ejecución]* de donde nos llegarán todos los comandos ejecutados junto a los resultados de la ejecución de cada uno que se almacenarán en el estado *commands*. Si el test está en ejecución, además se activará la escucha de mensajes a través de socket.io y se irán incluyendo en el estado *testState*. Es importante destacar el hecho de que se revisa que el primer estado que llega a través de sockets no sea el mismo que el último estado recuperado de base de datos, ya que se podría dar el caso de que el usuario acceda a esta página después de que se haya guardado el envío del comando en base de datos pero aún no se hubiera enviado a través del canal de socket.io.

Home.jsx

Este componente por ahora únicamente contiene el menú de navegación (Menubar.jsx) de la plataforma web desde el que se puede acceder a la gestión de cualquier módulo de la web.

SelectTest.jsx

Este componente es el que permite gestionar los tests. Lo primero que realiza es un *fetch* a */getModels* para tener todos los modelos disponibles en el arrray *models*. En caso de que no haya ningún modelo seleccionado, se mostrará una lista de estos modelos para poder entrar en la gestión de los tests de ese modelo o seleccionar cuál de ellos se quiere ejecutar. Una vez se haya seleccionado el modelo, se hará un *fetch* de */[modelo]/getTests*. Este listado de tests se guardará en el array *tests* y se mostrará una lista con los mismos, se podrá entrar a la página de edición así como eliminarlos y se da también la opción de crear un nuevo test para el modelo seleccionado.

StartTest.jsx

Este componente es el responsable de dar comienzo a un test. Sólo se podrá acceder indicando en la URL el modelo de televisor sobre el que se quiere hacer el test. Al iniciarse, hará un *fetch* al endpoint */[modelo]/getTests* e introducirá el resultado con los diferentes tests que haya disponibles para el modelo en el array *tests* del estado. También hará un *fetch* al endpoint */getTransmitters* para poder tener todos los emisores que haya disponibles en el array *transmitters*. Una vez se haya cargado toda la información necesaria, se mostrará un desplegable tipo *select* para que el usuario seleccione el emisor

(el cual se quedará guardado en el estado *selectedTransmitter*), así como también aparecerá una tabla con los distintos tests junto a un botón de "Start" que lo que realizará al ser pulsado es un *fetch* del endpoint */[modelo]/startTest/[ID del test]/?ir=[ID del emisor]* y mostrará un mensaje indicando que se ha iniciado el test.

TestExecutions.jsx

Este componente es el responsable de mostrar la lista de ejecuciones que se han realizado. Es un módulo muy sencillo, ya que simplemente hace una llamada al endpoint */getExecutions* de forma que podrá guardar todas las ejecuciones en el array *executions*, los datos que tendremos de cada ejecución son el nombre, modelo, fecha de inicio y fin y su identificador. En la lista que se muestra se indicará la fecha en la que se comenzó a hacer el test, y en caso de que no haya terminado se mostrará el mensaje "On execution…". Junto a cada línea se mostrará un botón para ver más sobre la ejecución, que nos llevará al módulo *ExecutionCommands*.

Extra: **Menubar.jsx**

Este componente se utiliza en todos los módulos anteriores para mostrar el menú superior de navegación. Consiste simplemente en un componente Navbar con los distintos enlaces que se quieren mostrar al usuario para que pueda navegar a través de la aplicación.

3.3.2. Funciones recurrentes

Una función muy usada en los componentes citados es *fetch()*. Ésta se encarga de realizar una llamada a una URL pasada como parámetro. Se trata de una función asíncrona, por lo que la respuesta viene en forma de *Promise*. Una vez llega la respuesta, generalmente en formato JSON, se introducen estos datos en el estado del componente que ha hecho la llamada, y además se utiliza el valor del estado *isLoaded* para comprobar que se ha completado la carga de datos. De esta forma, el método *render()* mostrará el contenido en el momento en que *isLoaded* tenga el valor *true*.

En la mayoría de componentes esta carga se realiza dentro de la función *componentDidMount()* que es el encargado de establecer estados iniciales al componente, antes de que se realice el *render()*. Un ejemplo puede observarse en el Código 3.

```
componentDidMount() {
    fetch("http://localhost:3501/hotel/test/getModels")
        .then(res => res.json())
        .then(
            (result) => {
                this.setState({
                    isLoaded: true,
                    models: result
                });
            },
            (error) => {
                this.setState({
                    isLoaded: true,
                    error
                });
            }
        )
}
```

Código 3: Consulta fetch

Una librería que también se utiliza recurrentemente en este proyecto es *react-confirm-alert*. Esta se utiliza para mostrar un mensaje de confirmación que se superpone a la pantalla que se esté mostrando. Es especialmente útil para tener la confirmación por parte del usuario de que se quiere eliminar algún registro (modelo de TV, botón, test…).

Su implementación únicamente requiere importar el módulo junto a su hoja de estilos como muestra el Código 4.

```
import {confirmAlert} from 'react-confirm-alert';
import 'react-confirm-alert/src/react-confirm-alert.css';
```

Código 4: Importación del componente de confirmación

Posteriormente únicamente será necesario llamar la función *confirmAlert()* indicando los parámetros del mensaje que se quiera mostrar y las acciones a realizar cuando se pulse cualquiera de sus botones, como se observa en el Código 5.

```
confirm(i, index) {
    confirmAlert({
        title: 'Delete',
        message: 'Are you sure to delete this component?',
        buttons: [
            {
                label: 'Yes',
                onClick: () => {
                    // ejecución al pulsar Yes
                }
            },
            {
                label: 'No',
            }
        ]
    });
}
```

Código 5: Mensaje de confirmación

En esta misma línea, también hay módulos HTML que nos pueden interesar mostrar u ocultar atendiendo a las acciones que realice el usuario sobre cada página. En ReactJS es muy útil utilizar una vez más los estados del componente, de forma que se puede establecer un booleano (por ejemplo llamado *showForm*). Como en el ejemplo del Código 6, si utilizamos un contenedor de tipo <Overlay> podremos usar este estado en el parámetro *show* que se mostrará en función del estado correspondiente.

```
<Overlay show={this.state.showNewButtonForm}>
    //Contenido del componente
</Overlay>
```

Código 6: Overlay con parámetro show

4. PRUEBAS

Llegados a este punto en el que el desarrollo queda finalizado, podemos pasar a comprobar que se cumple una serie de funciones básicas que debe poder desempeñar la plataforma. Para ello, se comprueba un total de 10 puntos críticos bajo los que podremos aceptar la correcta consecución de los objetivos más básicos del proyecto.

4.1. CONEXIÓN DE LA WEBAPP CON LA BASE DE DATOS

Para realizar esta prueba, entraremos en cualquiera de los distintos apartados de gestión que disponemos, por ejemplo, de gestión de modelos de TV, y podremos comprobar que los datos mostrados en la pantalla se corresponden con los que contiene la base de datos como se observa en la Figura 15.

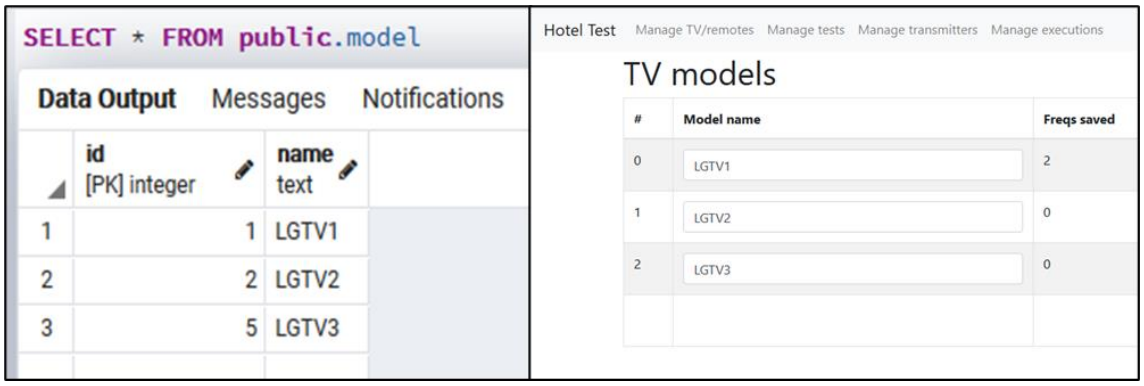

Figura 15: Base de datos (izquierda) y web (derecha)

Como se puede observar, los resultados en cuanto a los nombres de los modelos coinciden con lo esperado. El id de cada modelo dentro de la Base de Datos, al ser un dato interno, el usuario no puede acceder visualmente al mismo. Y finalmente, en la columna *Freqs saved* podemos ver como se ha hecho la unión con la tabla de *Frecuencia* y se ha hecho un recuento de las correspondientes a cada modelo.

4.2. RECEPCIÓN DE FRECUENCIAS EN EL SERVIDOR

Para llevar a cabo esta comprobación, verificaremos el *log* que nos mostrará el servidor al recibir una frecuencia a través del *endpoint* al que atacará el Arduino. Desde el servidor tendremos el comando *pm2 monit* en ejecución, cuya visualización corresponde con la mostrada en la Figura 16. De esta forma simplemente deberemos tener el Arduino debidamente conectado tanto a la corriente como a la red local. Apuntando con un mando de televisor hacia el receptor IR, podemos observar cómo instantáneamente se recibe la decodificación de la frecuencia en el terminal.

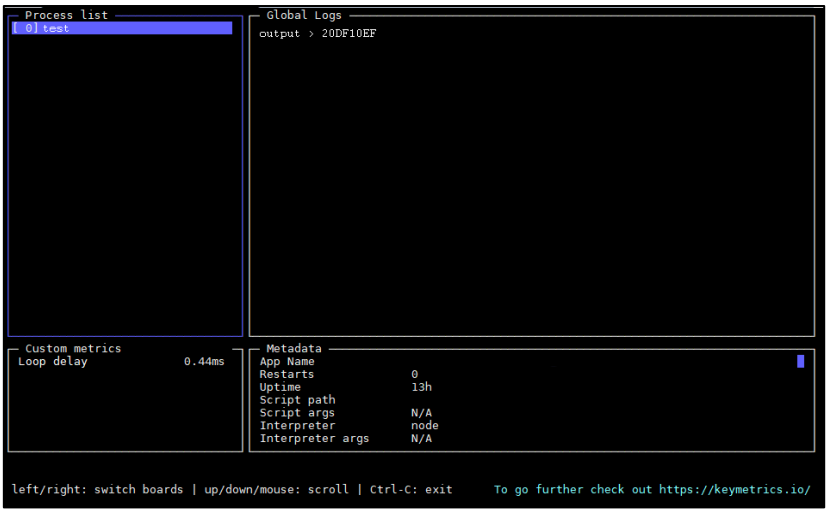

Figura 16: Instrucción pm2 monit en ejecución

4.3. RECEPCIÓN DE FRECUENCIAS EN LA WEBAPP

Para validar esta prueba, añadiremos un nuevo botón a la lista de frecuencias de un modelo y pulsaremos en *Receive freq* para que comience la escucha.

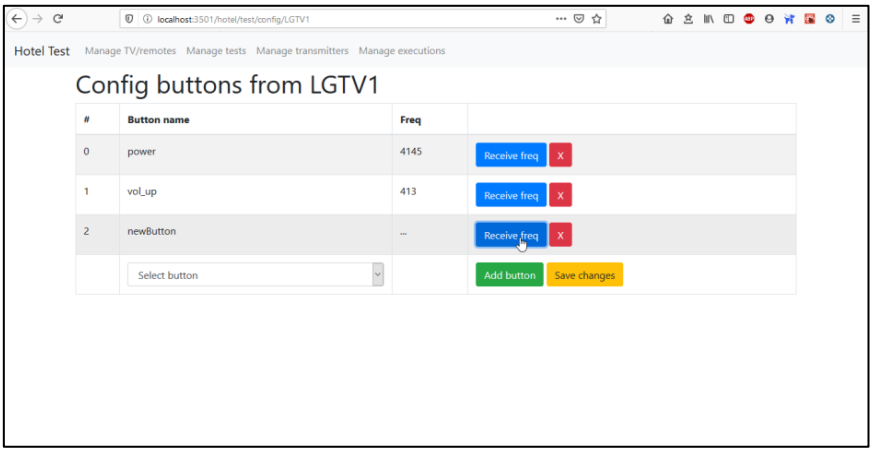

Figura 17: Receive freq

Como se observar en la Figura 17, aparecerán puntos suspensivos en la columna *Freq* para indicar que se ha activado la escucha de códigos. En cuanto se realiza la misma acción que en la Prueba 2 vemos como aparece automáticamente el valor en el campo correspondiente. Si probamos a pulsar otro botón del mando, veremos que ese código será ignorado ya que la *WebApp* habrá abandonado la escucha.

4.4. EMISIÓN DE FRECUENCIAS

La forma más directa de realizar esta comprobación consiste en poner en marcha un test básico que encienda y apague un televisor. Situaremos el Arduino con el emisor IR apuntando hacia el televisor y daremos inicio al test desde la WebApp. Desde el servidor,

como se muestra en la Figura 18, se puede observar que el *log* nos indica la posición del comando dentro del test y la frecuencia a emitir por parte del Arduino.

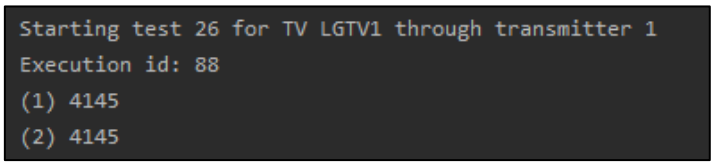

Figura 18: Log del servidor

Además, en el Arduino podemos observar también el parpadeo del diodo LED incrustado en la placa que nos indica que ha recibido los datos.

4.5. EJECUCIÓN DE COMANDOS EN TIEMPO REAL

La plataforma está preparada para consultar la ejecución de un test en tiempo real, por lo que accediendo al apartado de *Test executions* podemos comprobar el transcurso de cómo los mensajes van llegando a través de socket.io y cómo React.js los renderiza en la tabla de la Figura 19 que visualiza el usuario.

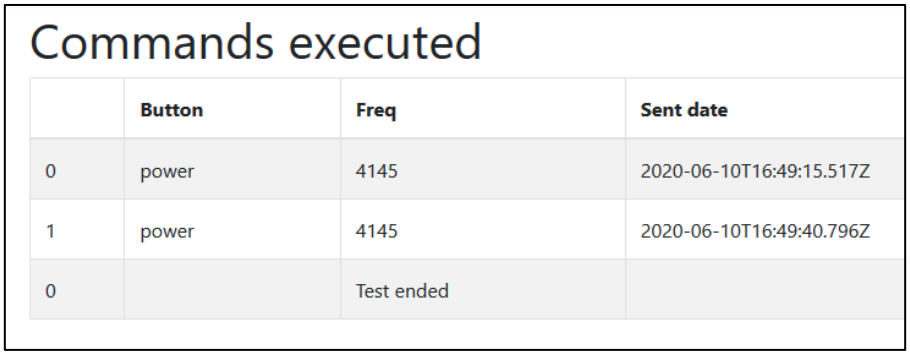

Figura 19: Tabla de comandos ejecutados

Ejecutando el mismo test que en la prueba 4, queda comprobado que aparece cada comando en el momento que es enviado al Arduino y muestra además la fecha y hora en que se ha realizado. Al finalizar el test aparece el mensaje *Test ended* a modo de confirmación (este mensaje no aparece posteriormente, si la consulta de los comandos ejecutados se hace una vez ha terminado el test, como podremos observar en la siguiente prueba).

4.6. GUARDADO DE COMANDOS EJECUTADOS

Además de la emisión a través de socket.io, los comandos también tienen que guardarse en la base de datos para poder ser consultados posteriormente. Es importante indicar en este punto que, si entramos a consultar un test en ejecución, se realiza por un lado la

consulta a la base de datos de los comandos ya ejecutados, y además se abre el canal para recibir los mensajes de socket.io para poder seguir el transcurso en directo.

En este caso, entramos en la ejecución que se realizó en la prueba 5 y vemos en la Figura 20 que los comandos se han guardado correctamente, y que la fecha y hora coinciden exactamente.

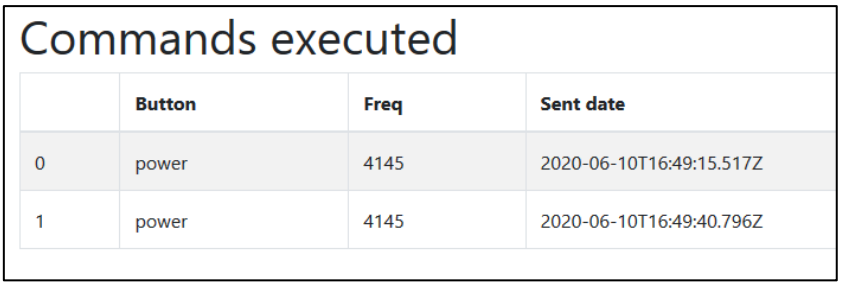

Figura 20: Comandos ejecutados

4.7. ELIMINACIÓN DE UNA FRECUENCIA

En caso de que se desee eliminar una de las frecuencias guardadas para un modelo de televisor, deberemos acceder al listado de las mismas y pulsar en el botón de eliminar. En este punto aparecerá la pantalla de confirmación de la eliminación de la Figura 21, como podemos comprobar en caso de Cancelar no ocurrirá nada, pero en caso de Aceptar, observaremos como esa fila desaparece de la tabla, y lo mismo ocurre en la Base de Datos.

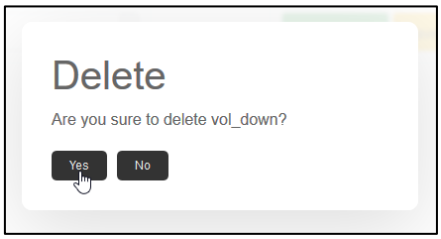

Figura 21: Confirmación de eliminación

4.8. ELIMINACIÓN DE UN TEST

La eliminación de tests es un caso especial, ya que en realidad al eliminar un test lo que se efectúa es su inactivación, de forma que el usuario no tendrá opción de editarlo o ejecutarlo, pero sí se le permitirá seguir viendo las ejecuciones que se han realizado del mismo. Esto es interesante de cara a poder comparar en el futuro el resultado entre tests.

Para proceder a la comprobación, tras eliminar un test de la lista, podemos ver en la Figura 22 que en la base de datos sigue apareciendo el test con la fecha y hora en que se eliminó

en el campo *deleted*, y que en la lista de ejecuciones realizadas de la Figura 23 seguimos viendo ejecuciones de dicho test.

```
SELECT * FROM test WHERE name='4volup'
```

Data Output | Messages | Notifications

	id [PK] integer	name text	model_id integer	deleted timestamp without time zone
1	16	4volup	1	2020-05-22 19:58:54.969

Figura 22: Consulta de un test eliminado en BD

Commands executed

	Button	Freq	Sent date
0	power	4145	2020-04-26T16:17:30.863Z
1	vol_up	413	2020-04-26T16:17:37.063Z
2	vol_up	413	2020-04-26T16:17:41.800Z
3	vol_up	413	2020-04-26T16:17:46.082Z
4	power	4145	2020-04-26T16:17:59.083Z

Figura 23: Ejecución del test 4volup

4.9. CREACIÓN DE UN TEST A PARTIR DE CSV

Con tal de facilitar la creación de tests, la plataforma ofrece la posibilidad de importar los comandos desde un fichero CSV. En esta prueba vamos a verificar que todos los datos que se indican en el fichero se establecen correctamente en los campos correspondientes de la *WebApp*.

Creamos el fichero CSV de la Figura 24 con cualquier herramienta de edición de texto plano y lo importamos desde la web. Como se puede observar en la Figura 25, la importación se ha realizado correctamente y aún podemos seguir editando los campos desde la propia web antes de guardarlo definitivamente.

test-TV.csv

```
1 delay;expected;name;onerror;timeout
2 2;ok;power;0;10
3 1;ok;vol_up;0;10
4 1;ok;vol_up;0;10
5 2;ok;vol_up;0;10
6 2;ok;power;0;10
```

Figura 24: Fichero CSV

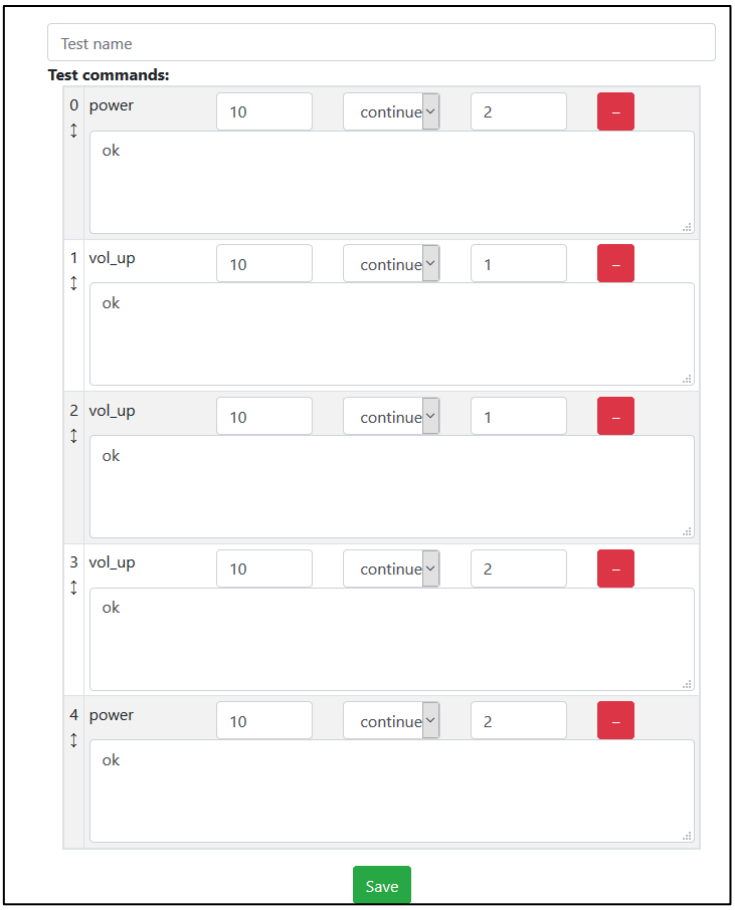

Figura 25: Test importado en la WebApp

4.10. DISEÑO DE TESTS

Siguiendo la prueba anterior, comprobamos también la edición de los comandos de un test. Podemos comprobar en la Figura 26 como efectivamente podemos editar todos los valores de cada comando y eliminarlos. En el caso de añadir, se introducirán al final de la tabla, pero gracias al botón ↕ podremos mover cada comando hacia la posición que deseemos sin que los valores se vean afectados.

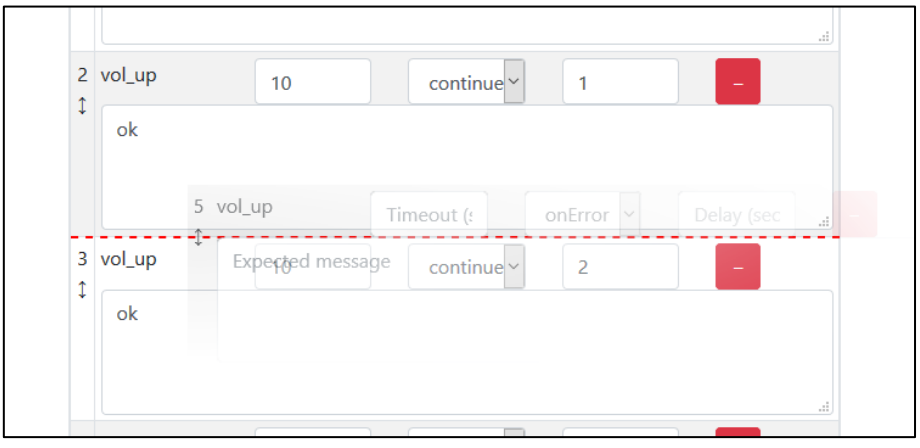

Figura 26: Edición de la posición de un comando

5. MANUALES

Se han definido una serie de manuales tanto de instalación como de uso para facilitar la labor de los usuarios que deseen instalar y utilizar el sistema.

5.1. MANUAL DE INSTALACIÓN DEL ENTORNO DE DESARROLLO

Para poder empezar a programar el proyecto y utilizar todos los entornos descritos en el apartado 2.3.2, debemos instalar una serie de librerías.

El primer componente a instalar es Node.js. Al disponer el servidor de sistema operativo Linux, basta con descargar e instalar el correspondiente paquete oficial de instalación. [27]

```
sudo apt-get install nodejs
sudo apt-get install npm
```

Una vez instalado podemos utilizar el comando *npm* para instalar paquetes de Node y/o librerías necesarias para el desarrollo de la aplicación.

En primer lugar, se procederá a instalar la librería React Bootstrap. Lo haremos de la manera indicada anteriormente, aunque es importante remarcar que se debe ejecutar el comando desde la carpeta en la que se desarrolla la parte *frontend* del proyecto, es decir, en la carpeta *webapp*.

```
npm install react-bootstrap bootstrap --save
```

Otros módulos que se deben instalar para su uso posterior a lo largo del desarrollo son los siguientes:

El módulo express en la carpeta *api*, para poder establecer las rutas HTML que se usarán dentro de la *WebApp*.

```
npm install express--save
```

En la misma carpeta, el módulo pg, para poder acceder a la base de datos Postgres desde la parte de la *api*.

```
npm install pg --save
```

Tanto en la carpeta *api* como en *webapp* necesitaremos el módulo de socket-io.

```
npm install socket.io --save
```

Nuevamente en la carpeta *api*, instalaremos el módulo Swagger para la documentación de los endpoints de la API que veremos en el apartado 5.3.

```
npm install swagger --save
```

Así como el módulo xmlhttprequest para poder realizar peticiones HTTP.

```
npm install xmlhttprequest --save
```

Finalmente, para poder comenzar con la programación de los Arduinos, debemos proceder a la instalación de Arduino IDE, cuya versión oficial estable se puede encontrar en su web oficial www.arduino.cc. Al ser un programa de código abierto su descarga y uso es totalmente gratuito.

5.2. MANUAL DE USO DE LA WEBAPP

La web *Dashboard* está diseñada para ser utilizada en cualquier navegador. Además, al utilizar Bootstrap, es responsive de forma que se adapta a pantallas más pequeñas como móviles o tablets. El desarrollo inicial se ha realizado en inglés con tal de utilizar un idioma lo más universal posible.

Una vez se tiene el servidor en marcha, podemos entrar a la web de la plataforma para poder iniciar las pruebas. Desde esta primera pantalla, que podemos observar en la Figura 27, visualizaremos el menú desde el que poder acceder a cualquiera de los apartados disponibles.

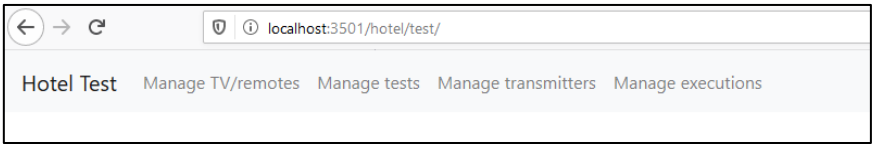

Figura 27: Home

Haciendo clic sobre el botón *Manage TV/remotes* podremos acceder a la gestión de modelos de televisor y sus correspondientes botones. El listado que muestra la Figura 28, ha recuperado los datos necesarios desde las tablas Modelo y Frecuencia de la base de datos.

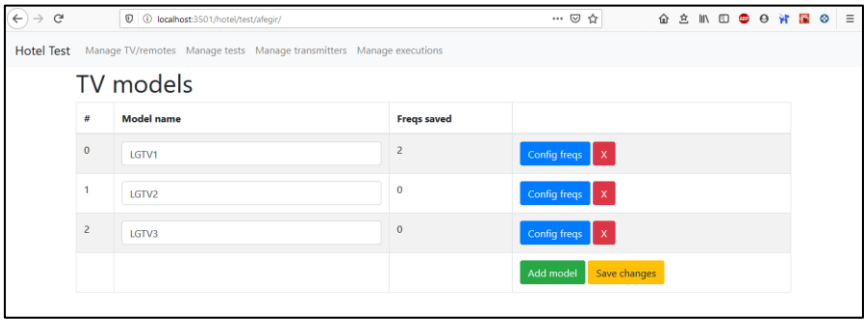

Figura 28: AddModel

Al pulsar en el botón *Add model* el usuario podrá añadir un nuevo modelo, como se muestra en la Figura 29, y este quedará registrado en la base de datos cuando pulse el botón *Save changes*, así como también actualizará los posibles cambios que pueda haber en los nombres de los modelos ya existentes.

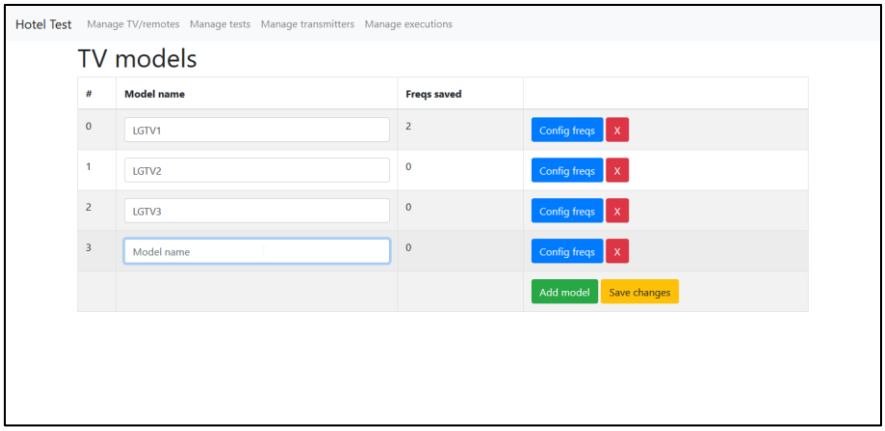

Figura 29: AddModel (Add)

También, como se observa en la Figura 30, si se desea eliminar uno de los modelos simplemente se debe pulsar el botón rojo marcado con el símbolo X.

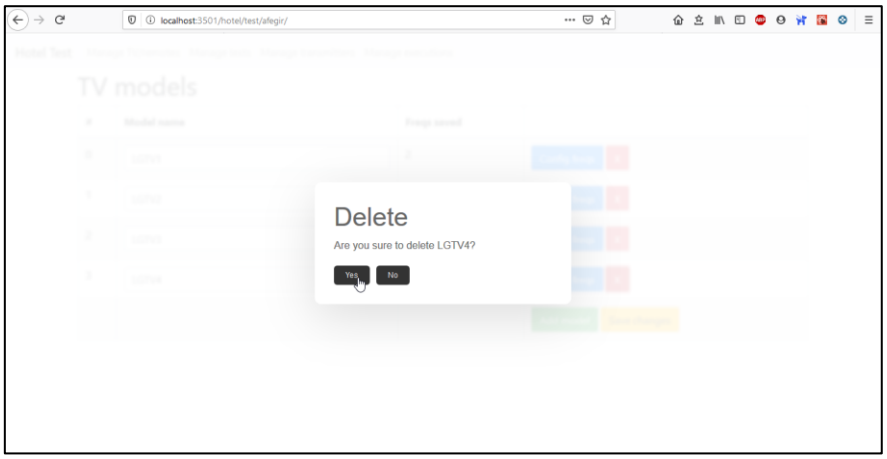

Figura 30: AddModel (Delete)

Este mismo procedimiento servirá para otros componentes que se puedan encontrar en la plataforma.

Al entrar a configurar los botones de uno de los modelos de televisor, se nos mostrará una lista con los botones que hay actualmente almacenados en la base de datos, tal y como se muestra en la Figura 31.

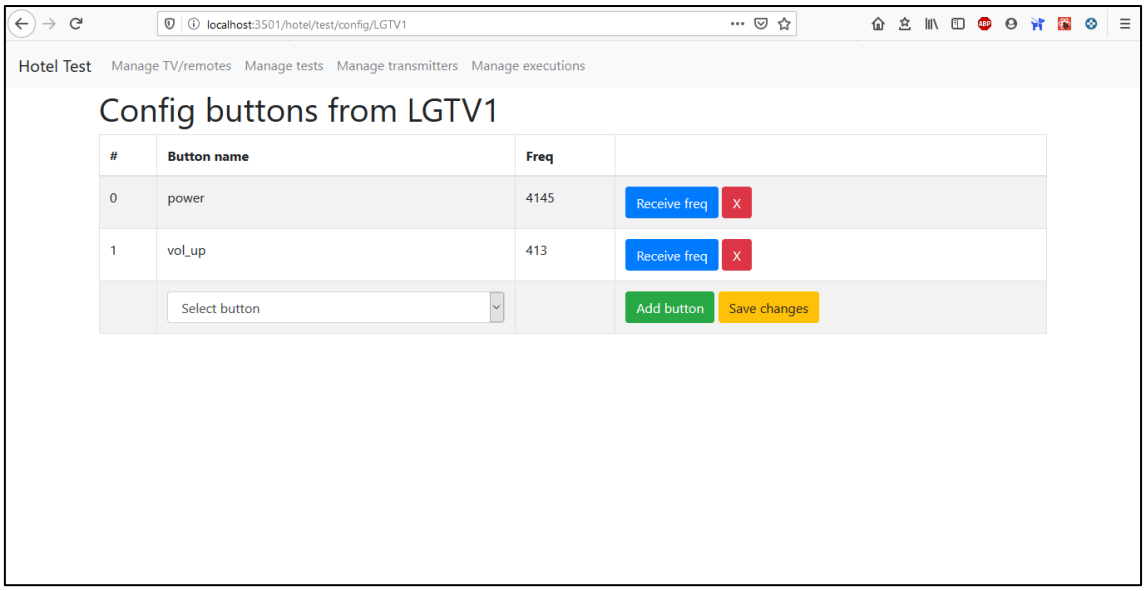

Figura 31: AddFreq

Al añadir un nuevo botón, el usuario debe indicar el nombre y éste quedará guardado para futuras ocasiones. Como se puede observar en la Figura 32, aparecerá un desplegable con todos los botones disponibles.

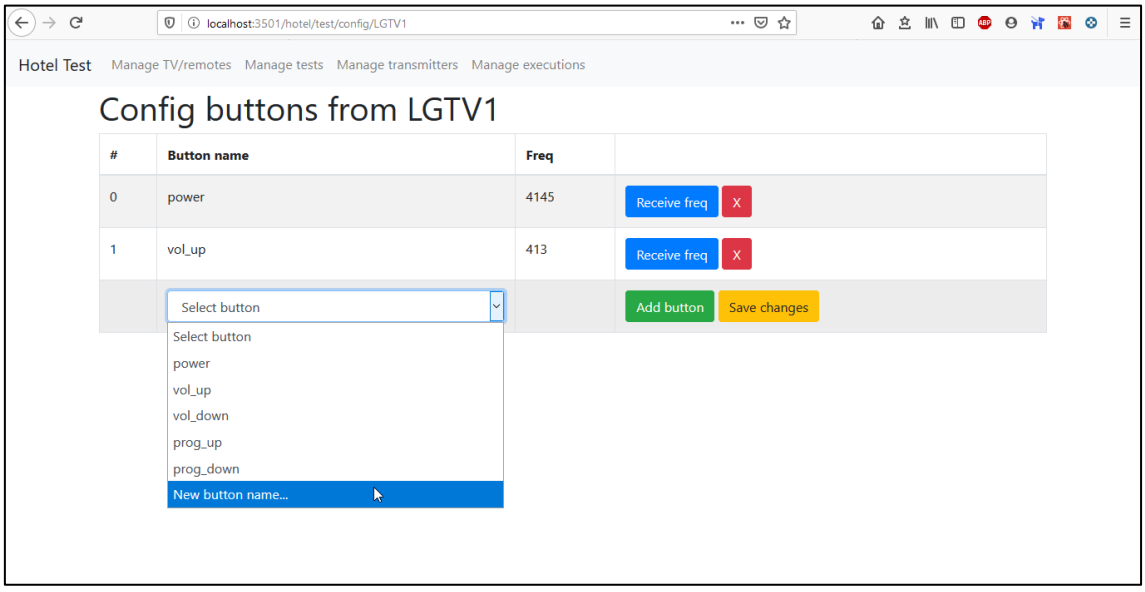

Figura 32: AddFreq (New Button)

Al pulsar el botón *Receive freq* se activará la escucha, tal y como se muestra en la Figura 33. Cuando el Arduino reciba una señal del mando del televisor, esta aparecerá

inmediatamente en el campo correspondiente. Finalmente, se debe pulsar en *Save changes* para guardar todos los cambios realizados.

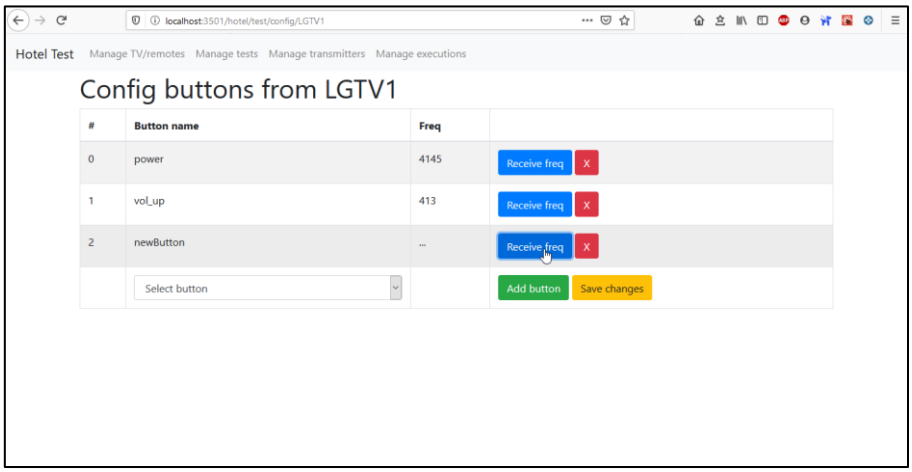

Figura 33: AddFreq (Receive freq)

Para gestionar tests, lo primero que se hará es seleccionar el modelo de televisor de forma que posteriormente se mostrarán los tests ya existentes para ese modelo como ocurre en la Figura 34 y Figura 35.

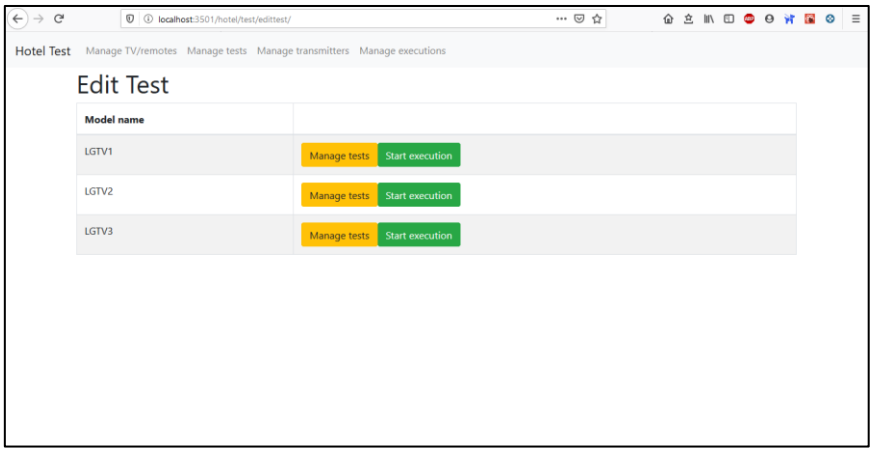

Figura 34: Select test

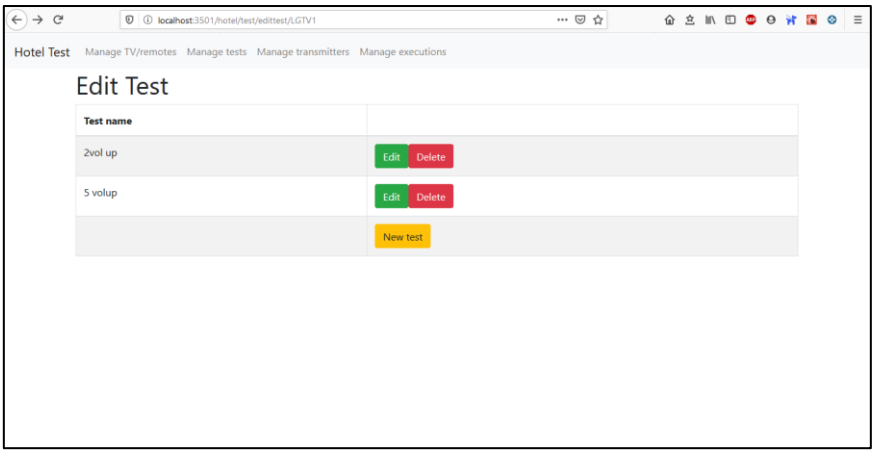

Figura 35: Edit test

Las ventanas para crear un test nuevo y editar uno ya existente son prácticamente idénticas. En ambas aparecen a la izquierda los botones de los que se tiene la frecuencia guardada, como se observa en la Figura 36, y a través de los botones que los acompañan se podrán ir añadiendo o eliminando comandos al test. Al pulsar *Save* el test queda guardado en base de datos y se muestra el mensaje de la Figura 37. Como punto extra, en la pantalla de crear un nuevo test, se da la opción a importar desde un fichero CSV.

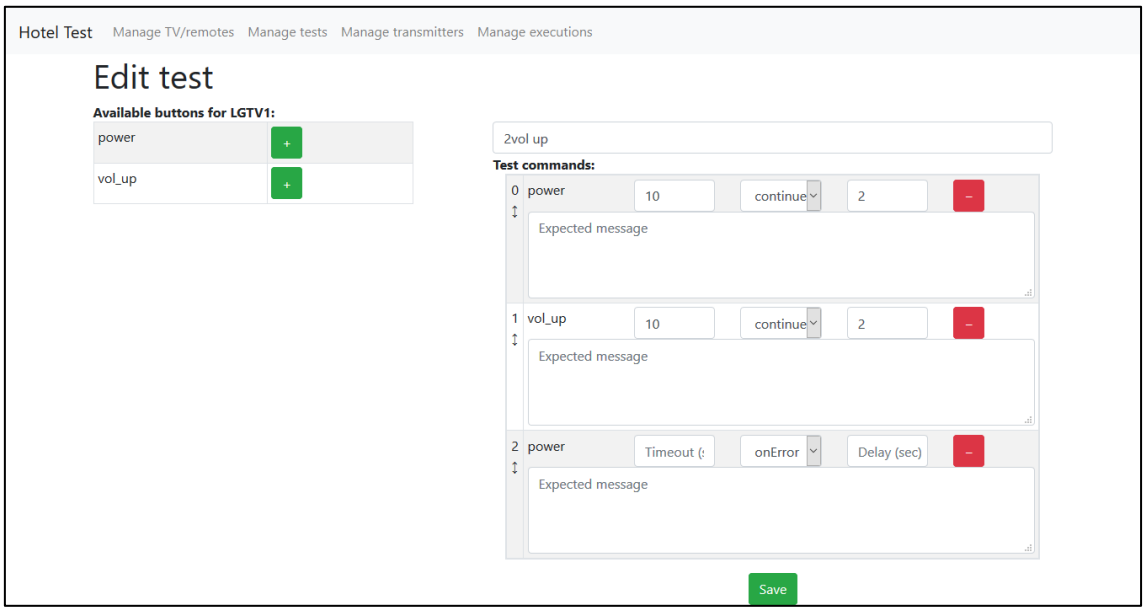

Figura 36: New/Edit test

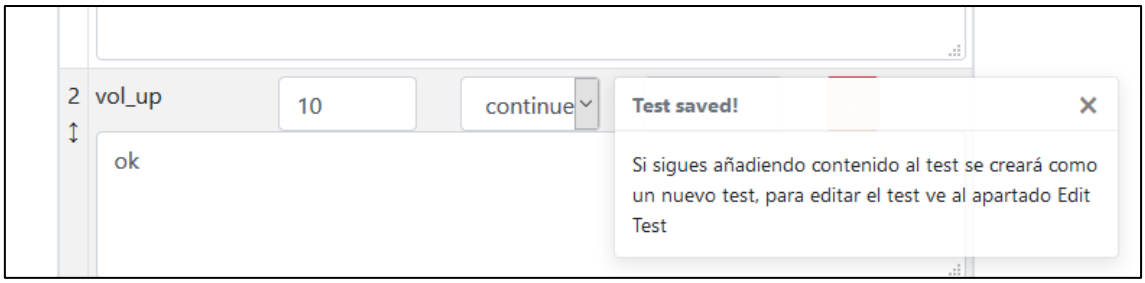

Figura 37: New/Edit test (Saved message)

Antes de realizar la ejecución de un test, se debe configurar al menos un emisor, en este caso la placa Arduino que dispone de emisor IR. Se pueden añadir todos los emisores que se deseen, siempre indicando debidamente el nombre y la IP, como aparece en la Figura 38 y Figura 39.

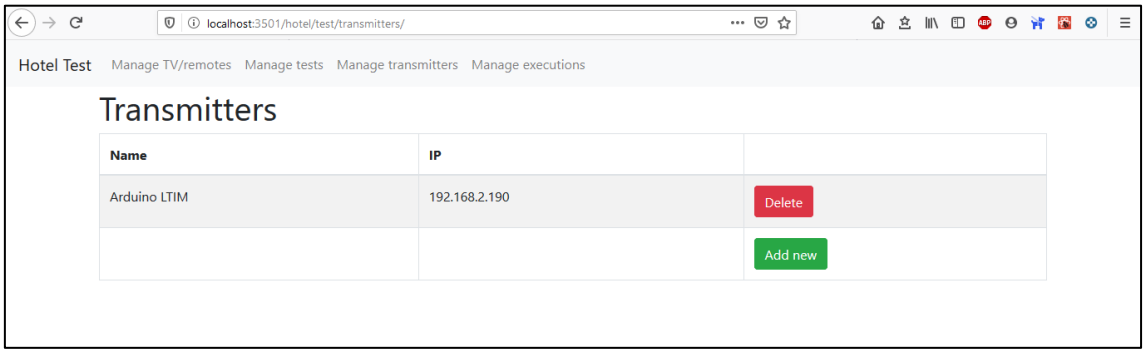

Figura 38: Manage transmitters

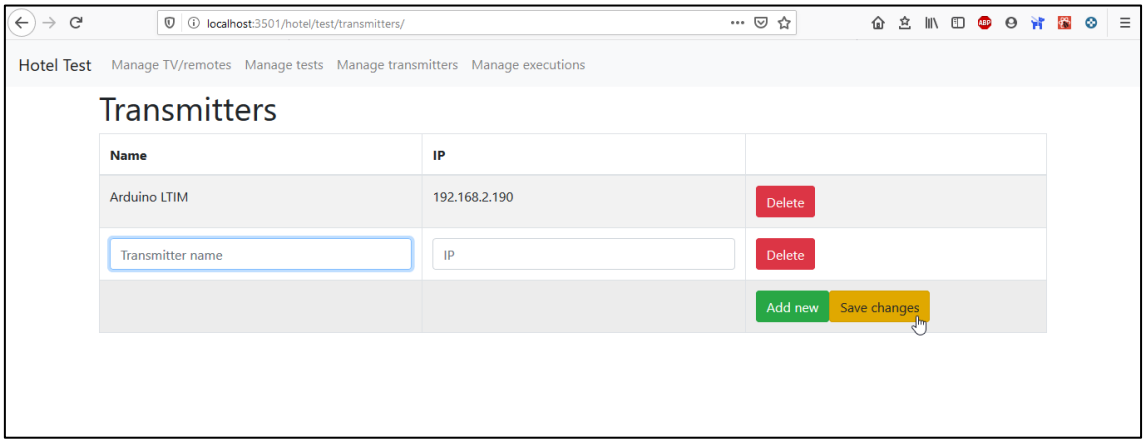

Figura 39: Manage transmitters (Add new)

Para comenzar la ejecución de uno de los tests que se hayan creado, se debe seleccionar el trasmisor a utilizar (es importante que el emisor IR del mismo esté apuntando hacia el televisor objetivo) como muestra la Figura 40. Al pulsar en *Start* el test comenzará

inmediatamente. En este punto podemos cambiar libremente de pantalla e incluso cerrar el navegador, ya que el test seguirá ejecutando de forma autónoma desde el servidor.

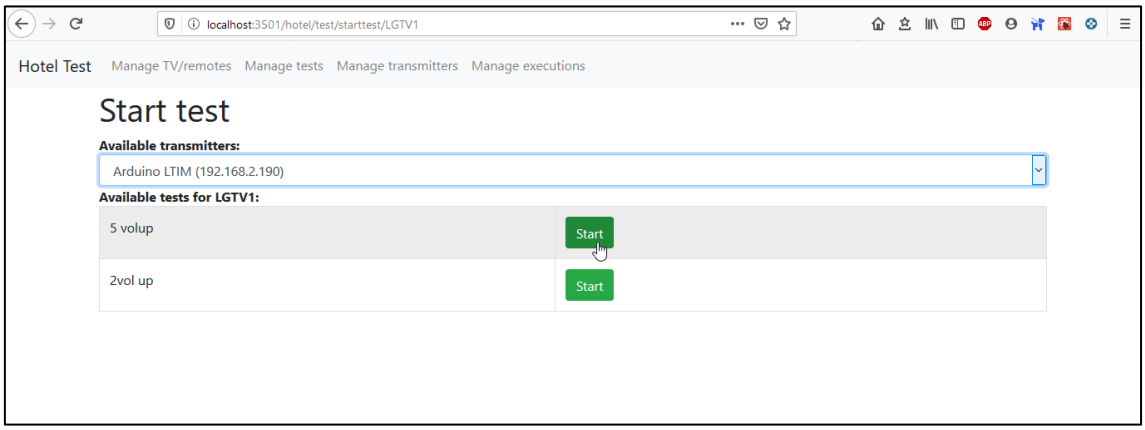

Figura 40: Start test

Finalmente, la web permite visualizar el transcurso de las ejecuciones que se hayan realizado e incluso las que estén en marcha en el momento de la consulta. Estas aparecerán ordenadas por fecha como se puede observar en la Figura 41, donde la primera de la lista es la más reciente. Pulsando el botón *See more* podremos consultar comando por comando cuál ha sido su resultado y el momento exacto en el que se ha enviado hacia el emisor, como se observa en la Figura 42.

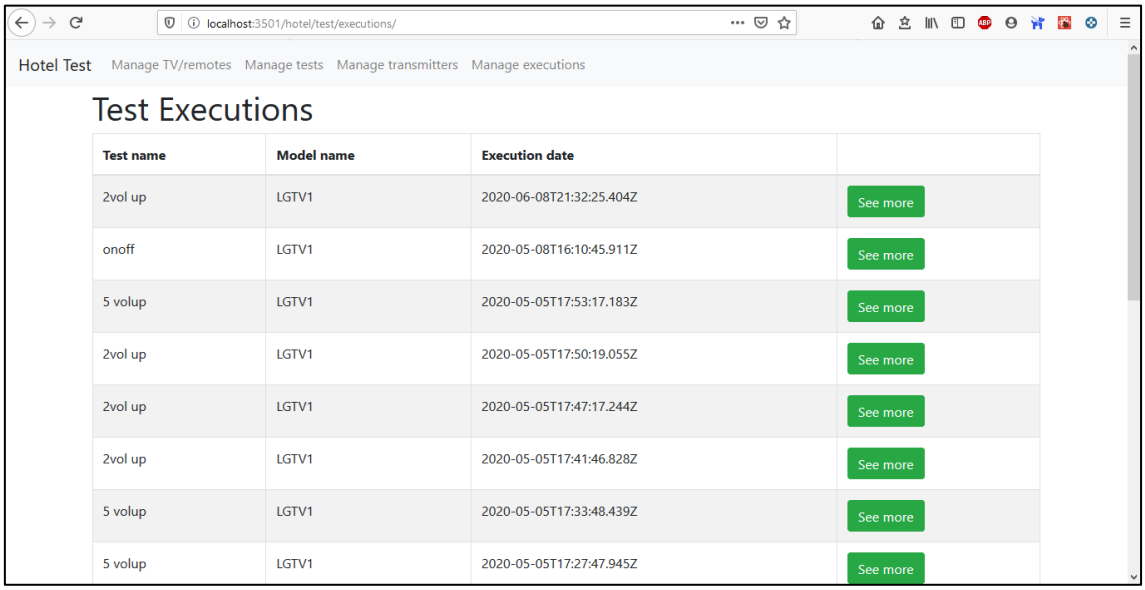

Figura 41: Test executions

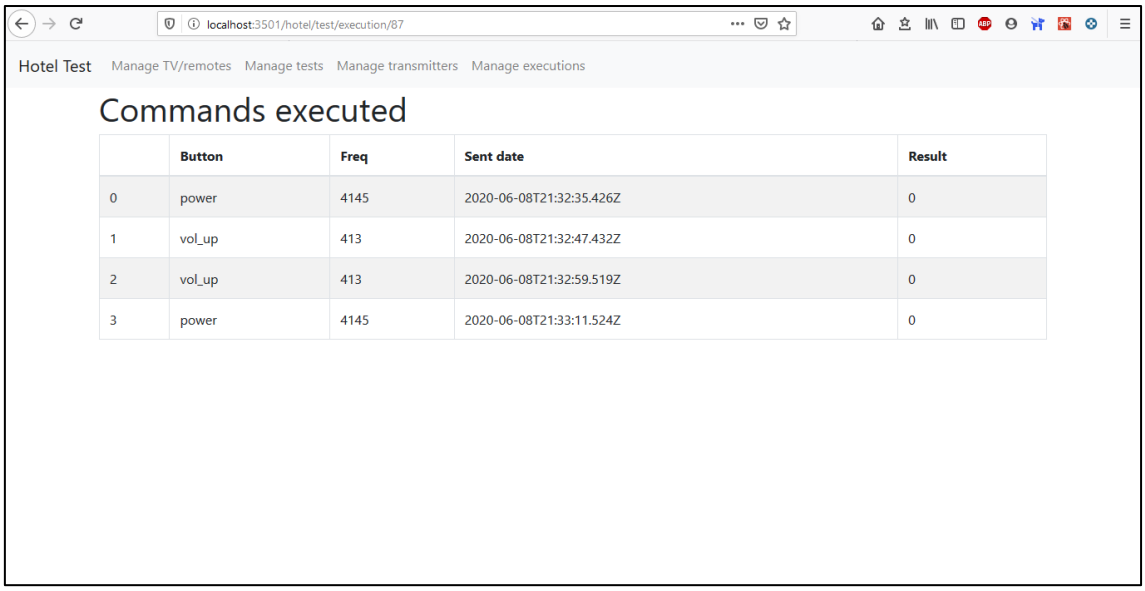

Figura 42: Commands executed

5.3. SWAGGER

Con el propósito de que la API sea más accesible, se ha creado una documentación interactiva que ayude a los usuarios que deseen usarla. Swagger es uno de los frameworks más extendidos para documentar APIs, y al tener una interfaz de usuario sencilla resulta realmente cómodo su uso y ahorrará todo el tiempo que en caso de no disponer de esta herramienta nos supondría tener que investigar a lo largo de todo el código de la carpeta API del proyecto.

La forma que tiene Swagger de interpretar nuestra API es a través de un fichero JSON o YAML en el que se indica el endpoint, la descripción del mismo y otros detalles como los parámetros y la respuesta esperada. [28]

En el caso de Node y Express, encontramos dos librerías que nos facilitan este trabajo, que son swagger-ui-express y swagger-jsdoc. Gracias a ellas el fichero JSON se irá generando automáticamente a través de los comentarios que se hagan en el código de cada endpoint definido en el fichero index.js. Un ejemplo lo encontramos en el Código 7.

```
/**
 * @swagger
 * /hotel/test/{model}/getFreqs:
 *    get:
 *      tags:
 *          - Frecuencias
 *      summary: Consultar frecuencias de un modelo.
 *      description: Consultar las frecuencias que hay almacenadas en la BD
de un modelo indicado en la URL.
 *      parameters:
 *      - name: "model"
 *        in: "path"
 *        description: "Nombre del modelo de TV"
 *        required: true
 *        type: "string"
 *      responses:
 *        200:
 *          description: modelos de TV.
 *          schema:
 *            type: array
 *            items:
 *              type: object
 *              properties:
 *                id:
 *                  type: integer
 *                name:
 *                  type: string
 *                freq:
 *                  type: string
 *                hasChanged:
 *                  type: Boolean
 */
```

Código 7: Definición de endpoint en Swagger

De esta forma simplemente tendremos un fichero swagger.js que será el encargado de definir a nivel general el nombre y versión de la aplicación, y el archivo en el que se encuentran todos los endpoints que queramos mostrar en Swagger (en este caso todos los de index.js).

Finalmente, en el archivo index.js bastará añadir el *endpoint* del propio swagger, que como se observa en el Código 8, en nuestro caso será */api-docs* en la raíz del servidor.

```
const swaggerUi = require('swagger-ui-express');
import * as specs from './swagger';
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(specs.default));
```

Código 8: Endpoint para Swagger

Es decir, entrando en el endpoint */api-docs* encontraremos con la pantalla que muestra la Figura 43 con todos los endpoints disponibles en la plataforma de testing, que para este proyecto se han organizado en los siguientes bloques:

- Arduino Receptor
- Modelos
- Botones
- Frecuencias
- Tests
- Ejecuciones
- Transmisores

En caso de querer consultar alguno de los endpoints, simplemente bastará con hacer clic encima y Swagger nos permitirá probarlo en tiempo real haciendo clic en el botón “Try it out”.

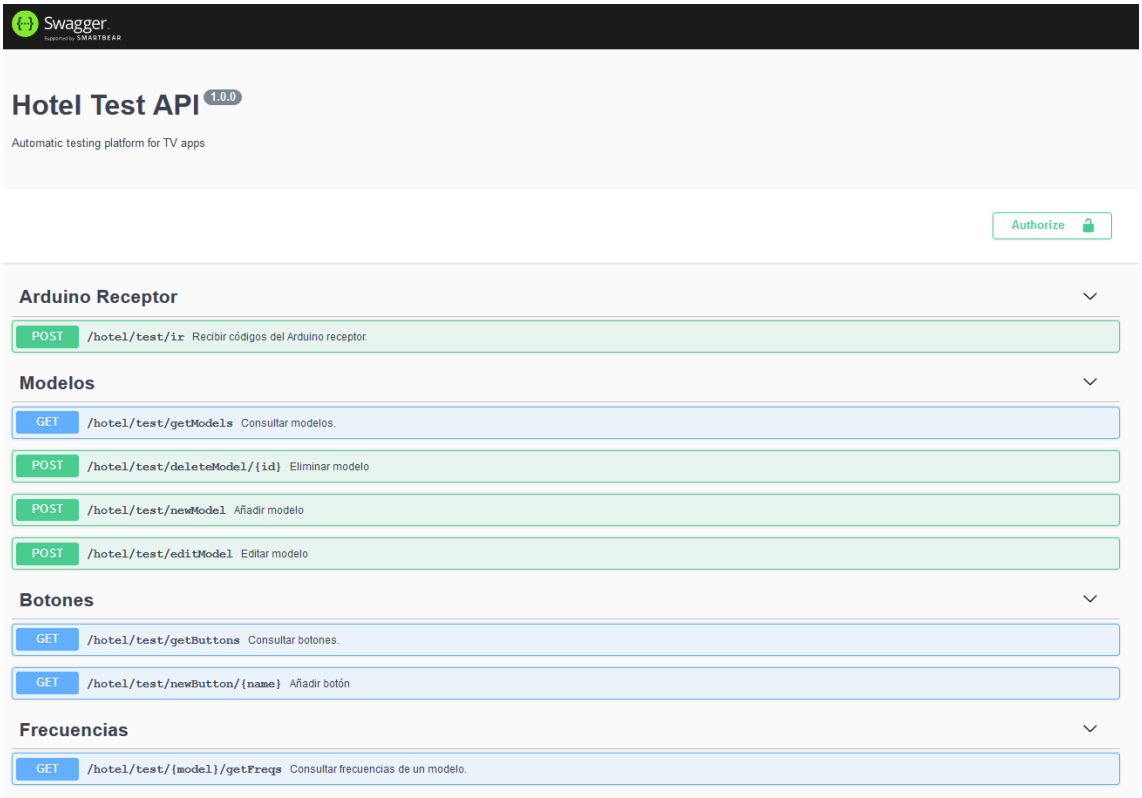

Figura 43: Pantalla principal de Swagger

5.4. MANUAL DE INSTALACIÓN DE LOS ARDUINOS

El dispositivo emisor con el que se han realizado las pruebas en el LTIM se trata de un Arduino Yun conectado a la red a través de Wifi. La primera vez que se pone en marcha este Arduino creará su propia red Wifi pública. [29] Una vez conectado desde cualquier dispositivo (ya sea ordenador o móvil) se deberá entrar en la URL arduino.local e introducir la contraseña *arduino* para entrar en su configuración. Una vez dentro, se puede

crear la conexión entre el Arduino y nuestra red Wifi de forma que este se le asignará una IP libre y podremos acceder a él a través de dicha IP.

El dispositivo receptor se trata de un Arduino Uno que se conecta a la red a través de Ethernet, por lo que será necesario indicar los datos de conexión (IP, puerta de enlace, DNS y MAC del dispositivo) dentro del propio *sketch* que se carga en el microcontrolador.

5.5. MANUAL DE INSTALACIÓN DEL SISTEMA

Es necesario realizar una serie de instalaciones de componentes para el correcto funcionamiento de la aplicación.

En caso de no tenerlo instalado previamente, se debe proceder a la instalación de Node.js, siguiendo los pasos indicados en la página 41.

Además, también es interesante disponer de PM2, que nos ayuda a tener un control sobre las ejecuciones que se están haciendo sobre Node.js. Para su instalación usaremos el comando

```
npm install pm2
```

Una vez instalado, a través del comando *pm2 start app.js* pondremos en marcha el proyecto. También podremos hacer uso de comandos básicos de Linux para ver los procesos (aplicaciones) en marcha: *list* | *ls* | *status* | *logs* | *monit*.

Finalmente, también necesitaremos una base de datos de Postgres, para lo cual usaremos el comando:

```
sudo apt-get install postgresql
```

Para la gestión de base de datos puede ser interesante también disponer de un gestor visual como pgAdmin. La base de datos inicial necesaria para el proyecto sigue el esquema indicado en la Figura 3 y cuyo código se detalla en el Anexo II.

6. CONCLUSIONES

Como conclusión a este trabajo, cabe destacar la oportunidad que he tenido para profundizar en la forma en que se desarrolla software de una forma más cercana al entorno laboral. El uso de entornos de desarrollo, plataformas de gestión de versiones y las decisiones sobre las distintas alternativas que he debido sopesar para poder comenzar con el desarrollo, me han demostrado que la definición inicial de un proyecto es uno de los pasos más importantes, ya que posteriormente te acompañan durante el resto del proyecto. Por todo ello, siento que he consolidado todos los conocimientos que he ido adquiriendo a lo largo de los cursos de este grado y que finalmente me habilitan para entrar a la vida laboral de una forma más firme y segura.

En cuanto a la plataforma desarrollada, hemos podido ver a lo largo del documento que el mundo de las pruebas no es algo poco relevante, sino que se le debe dedicar un tiempo para así conseguir productos de calidad y aumentar de esta forma la satisfacción del cliente final. Ha sido realmente interesante trabajar en un mercado tan emergente e innovador como son las Smart TVs, ya que, bajo mi punto de vista, todo aquello que lleve el prefijo Smart- es el futuro que cada vez tenemos más cerca.

También cabe destacar grandes ventajas del testing, no sólo en cuanto a la calidad o satisfacción del cliente, sino también por todo el tiempo que ahorra el hecho de que los tests sobre la aplicación se puedan realizar de forma autónoma, eliminando totalmente posibles errores humanos o restricciones horarias o económicas que podrían ser un impedimento para cualquier empresa que quiera producir software.

Sobre las funcionalidades implementadas, cabe destacar que, al tratarse de un proyecto acotado en el tiempo, estas tienen aún cierto margen de mejora. Es probable que se puedan encontrar algunos fallos que no estén contemplados, así como puede que haya funcionalidades útiles para el testing sobre televisores que no hayan sido contempladas en este proyecto, como, por ejemplo, realizar pruebas controlando el tiempo que se mantiene pulsado un botón, la simulación de alguna interrupción en la emisión de las frecuencias o una comparación automática de tests que se ejecuten de forma frecuente. Por otro lado, también hay algunas funcionalidades que se han citado en este documento pero que no han llegado a implementarse finalmente, como la copia de tests para distintos modelos de televisor, la posibilidad de programar la ejecución de los tests para que

comiencen de forma automática o el tercer módulo que se contemplaba en el apartado 2.1 sobre la comprobación de resultados a través del tratamiento de imágenes.

Como mejoras de cara al futuro, se podría contemplar la implementación de la funcionalidades mencionadas anteriormente y la depuración de posibles fallos en la plataforma. Es decir, aunque resulte redundante, se debería realizar un testing exhaustivo sobre la propia plataforma de testing.

También hay algunas mejoras que pueden realizarse en cuanto a la metodología de desarrollo, como una mejor planificación a través de Sprints usando la metodología Scrum, de forma que las iteraciones tuvieran mejor definidas las funcionalidades y el momento en el que debieran estar completadas. También hubiera sido interesante recibir más feedback por parte del cliente final para confirmar que se está llevando a cabo el proyecto cubriendo sus necesidades.

Por último, he de destacar que, a pesar de algunas dificultades, como la compaginación de la realización de este proyecto con una jornada laboral completa y la imposibilidad de seguir realizando pruebas presenciales con los Arduinos y televisores por culpa de la pandemia del COVID-19, ha sido muy gratificante trabajar en el mismo y observar día a día como las ideas iniciales que se plantearon fueron materializándose poco a poco.

7. BIBLIOGRAFÍA

[1] «RedHat,» [En línea]. Available: https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces. [Último acceso: Junio 2020].

[2] «SmartBear,» [En línea]. Available: https://smartbear.com/learn/performance-monitoring/api-endpoints/. [Último acceso: Junio 2020].

[3] O. Condés, «XatakaHome,» Agosto 2016. [En línea]. Available: https://www.xatakahome.com/curiosidades/poner-una-smart-tv-en-una-habitacion-de-hotel-es-innovacion. [Último acceso: Junio 2020].

[4] «Hotel Internet Services,» Octubre 2018. [En línea]. Available: https://www.hotelwifi.com/wp-content/uploads/2018/10/guestroom-entertainment_Survey_White-Paper.pdf. [Último acceso: Junio 2020].

[5] C. Hill, «MarketWatch,» Enero 2014. [En línea]. Available: https://www.marketwatch.com/story/7-disappearing-hotel-amenities-2014-01-08. [Último acceso: Junio 2020].

[6] M. A. Llambias Cabot, Sistema IPTV hospitality, TFG UIB, 2019.

[7] J. E. Crespo, «Aprendiendo Arduino,» Junio 2019. [En línea]. Available: https://aprendiendoarduino.wordpress.com/2019/06/15/por-que-usar-arduino/. [Último acceso: Junio 2020].

[8] R. Budde, K. Kautz, K. Kuhlenkamp y H. Züllighoven, «What is Prototyping?,» de *Prototyping. An Approach to Evolutionary System Development*, Springer Berlin Heidelberg, 1992.

[9] J. Nielsen, Usability Engineering, Cambridge, 1993.

[10] J. Mesh, «Trello,» Marzo 2020. [En línea]. Available: https://blog.trello.com/es/metodologia-kanban. [Último acceso: Junio 2020].

[11] «MDN web docs,» Junio 2020. [En línea]. Available: https://developer.mozilla.org/es/docs/Web/JavaScript. [Último acceso: Junio 2020].

[12] «Node.js,» [En línea]. Available: https://nodejs.org/es/about/. [Último acceso: Junio 2020].

[13] «Node.js,» [En línea]. Available: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/. [Último acceso: Junio 2020].

[14] W. Bravo, «Medium,» Marzo 2020. [En línea]. Available: https://medium.com/@wander.bravo/es-pm2-el-process-manager-para-tu-app-express-node-js-ejecut%C3%A1ndose-en-windows-a5974862fe2d. [Último acceso: Junio 2020].

[15] S. Borges, «Infranetworking,» Noviembre 2019. [En línea]. Available: https://blog.infranetworking.com/servidor-postgresql/. [Último acceso: Junio 2020].

[16] «pgAdmin,» [En línea]. Available: https://www.pgadmin.org/docs/. [Último acceso: Junio 2020].

[17] «ReactJS,» [En línea]. Available: https://es.reactjs.org/docs/. [Último acceso: Junio 2020].

[18] «React Bootstrap,» [En línea]. Available: https://react-bootstrap.github.io/. [Último acceso: Junio 2020].

[19] «Bootstrap,» [En línea]. Available: https://getbootstrap.com/docs/. [Último acceso: Junio 2020].

[20] «React Bootstrap,» [En línea]. Available: https://react-bootstrap.github.io/layout/grid/. [Último acceso: Junio 2020].

[21] «Express,» [En línea]. Available: https://expressjs.com/es/guide/routing.html. [Último acceso: Junio 2020].

[22] A. Rodríguez, «Hipertextual,» Agosto 2014. [En línea]. Available: https://hipertextual.com/archivo/2014/08/socketio-javascript/. [Último acceso: Junio 2020].

[23] D. Baah, «Spatialdevs,» Marzo 2016. [En línea]. Available: http://news.spatialdev.com/websockets-with-socket-io-taking-the-drag-out-of-client-server-web-communication/. [Último acceso: Junio 2020].

[24] J. E. Crespo, «Aprendiendo Arduino,» Enero 2017. [En línea]. Available: https://aprendiendoarduino.wordpress.com/2017/01/23/programacion-arduino-5/. [Último acceso: Junio 2020].

[25] «De Sensores,» [En línea]. Available: https://desensores.com/sensores-arduino/proyectos-basicos-con-arduino-para-principiantes/modulo-transmisor-receptor-ir-arduino/. [Último acceso: Junio 2020].

[26] «npm,» Mayo 2020. [En línea]. Available: https://www.npmjs.com/package/react-csv-reader. [Último acceso: Junio 2020].

[27] «npm,» [En línea]. Available: https://docs.npmjs.com/. [Último acceso: Junio 2020].

[28] «Chakray,» Enero 2017. [En línea]. Available: https://www.chakray.com/es/swagger-y-swagger-ui-por-que-es-imprescindible-para-tus-apis/. [Último acceso: Junio 2020].

[29] G. Baugues, «Twilio,» Febrero 2015. [En línea]. Available: https://www.twilio.com/blog/2015/02/arduino-wifi-getting-started-arduino-yun.html. [Último acceso: Junio 2020].

ANEXO I: CÓDIGO FUENTE DE LOS ARDUINOS

Arduino emisor

```
#include <IRremote.h>
#include <stdlib.h>

#include <Bridge.h>
#include <YunServer.h>
#include <YunClient.h>

YunServer server;
String msg;
IRsend irsend;

void setup () {

  Serial.begin(9600);
  Bridge.begin();
  server.listenOnLocalhost();
  server.begin();


}

void loop () {
  //*********Read new message from the client**************
  YunClient client = server.accept(); //check new clients

  if(client) {
    String command = client.readStringUntil('/');  //read the incoming data
    if (command == "emit") {
       String msg = client.readStringUntil('/');             // read the
incoming data
       //unsigned long freq = atol(msg.c_str());
       unsigned long freq = strtoul(msg.c_str(), NULL, 16);
       Serial.println(freq);
       irsend.sendNEC(freq, 32);
    }
    client.stop();
  }

}
```

Arduino receptor

```
#include <IRremote.h>
#include <SPI.h>
#include <Ethernet.h>

int RECV_PIN = 6;
IRrecv irrecv(RECV_PIN);
decode_results results;

// Network settings
byte mac[] = { 0x28, 0xAD, 0xBE, 0xEF, 0xB9, 0xED }; //MAC
```

```
IPAddress dnServer(192, 168, 2, 1);
IPAddress gateway(192, 168, 2, 1);
IPAddress subnet(255, 255, 255, 0);
IPAddress ip(192, 168, 2, 102);

// IP of the Target arduino with ir emitter
IPAddress server(192,168,2,88);
EthernetClient client;

void setup()
{
  Serial.begin(9600);
  // Start ethernet
  Ethernet.begin(mac, ip, dnServer, gateway, subnet);
  Serial.println("Inicio");
  irrecv.enableIRIn();
}

//sending the raw IR code over the ether net
void sendIR()
{
    //Ethernet.begin(mac, ip, dnServer, gateway, subnet);
    // Send over HTTP
    if (client.connect(server, 80))
    {
      Serial.print("Código leído: ");
      Serial.println(results.value,HEX);
      client.print(String("POST ") + "/hotel/test/ir?code=");
      client.print(results.value,HEX);
      client.print(" HTTP/1.1\r\n");
      client.print("Host: 192.168.2.88:3501\r\n");
      client.print("Connection: close\r\n");
      client.print("Content-Length: 0\r\n");
      client.print("Content-Type: application/json;charset=UTF-8\r\n\r\n");
      client.println();
      client.println();
      client.stop();
    }else{
      Serial.println("Fallo de conexión");
    }
}


void loop()
{
  if (irrecv.decode(&results))
  {
    sendIR();
    irrecv.resume(); // Receive the next value
  }
}
```

ANEXO II: CREACIÓN DE LA BASE DE DATOS

```
--
-- PostgreSQL database
--

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '', false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;

CREATE SEQUENCE public.button_id_seq
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1;


ALTER TABLE public.button_id_seq OWNER TO postgres;

SET default_tablespace = '';

SET default_table_access_method = heap;

CREATE TABLE public.button (
    id integer DEFAULT nextval('public.button_id_seq'::regclass) NOT NULL,
    name text
);

ALTER TABLE public.button OWNER TO postgres;

CREATE TABLE public.command (
    test_id integer NOT NULL,
    button_id integer NOT NULL,
    delay integer,
    "position" integer NOT NULL,
    expected text,
    timeout integer,
    onerror integer
);

ALTER TABLE public.command OWNER TO postgres;

CREATE TABLE public.command_execution (
    test_execution_id integer NOT NULL,
    "position" integer NOT NULL,
    result_code integer,
    message text,
    sent_time timestamp without time zone
);
```

```
ALTER TABLE public.command_execution OWNER TO postgres;

CREATE TABLE public.freq (
    model_id integer NOT NULL,
    button_id integer NOT NULL,
    value text
);

ALTER TABLE public.freq OWNER TO postgres;

CREATE TABLE public.model (
    id integer NOT NULL,
    name text
);

ALTER TABLE public.model OWNER TO postgres;

CREATE SEQUENCE public.model_id_seq
    AS integer
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1;

ALTER TABLE public.model_id_seq OWNER TO postgres;

ALTER SEQUENCE public.model_id_seq OWNED BY public.model.id;

CREATE TABLE public.test (
    id integer NOT NULL,
    name text,
    model_id integer NOT NULL,
    deleted timestamp without time zone
);


ALTER TABLE public.test OWNER TO postgres;

CREATE TABLE public.test_execution (
    id integer NOT NULL,
    test_id integer NOT NULL,
    start timestamp without time zone,
    "end" timestamp without time zone,
    transmitter_id integer
);


ALTER TABLE public.test_execution OWNER TO postgres;

CREATE SEQUENCE public.test_execution_id_seq
    AS integer
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1;
```

```
ALTER TABLE public.test_execution_id_seq OWNER TO postgres;

ALTER SEQUENCE public.test_execution_id_seq OWNED BY
public.test_execution.id;

CREATE SEQUENCE public.test_id_seq
    AS integer
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1;


ALTER TABLE public.test_id_seq OWNER TO postgres;

ALTER SEQUENCE public.test_id_seq OWNED BY public.test.id;

CREATE TABLE public.transmitter (
    id integer NOT NULL,
    name text,
    ip text
);

ALTER TABLE public.transmitter OWNER TO postgres;

CREATE SEQUENCE public.transmitter_id_seq
    AS integer
    START WITH 1
    INCREMENT BY 1
    NO MINVALUE
    NO MAXVALUE
    CACHE 1;

ALTER TABLE public.transmitter_id_seq OWNER TO postgres;

ALTER SEQUENCE public.transmitter_id_seq OWNED BY public.transmitter.id;


ALTER TABLE ONLY public.model ALTER COLUMN id SET DEFAULT
nextval('public.model_id_seq'::regclass);

ALTER TABLE ONLY public.test ALTER COLUMN id SET DEFAULT
nextval('public.test_id_seq'::regclass);

ALTER TABLE ONLY public.test_execution ALTER COLUMN id SET DEFAULT
nextval('public.test_execution_id_seq'::regclass);

ALTER TABLE ONLY public.transmitter ALTER COLUMN id SET DEFAULT
nextval('public.transmitter_id_seq'::regclass);

ALTER TABLE ONLY public.button
    ADD CONSTRAINT button_pkey PRIMARY KEY (id);

ALTER TABLE ONLY public.command_execution
    ADD CONSTRAINT command_execution_pkey PRIMARY KEY (test_execution_id,
"position");

ALTER TABLE ONLY public.command
```

```
    ADD CONSTRAINT command_pkey PRIMARY KEY (test_id, "position");

ALTER TABLE ONLY public.freq
    ADD CONSTRAINT freq_pkey PRIMARY KEY (model_id, button_id);

ALTER TABLE ONLY public.model
    ADD CONSTRAINT model_name_key UNIQUE (name);

ALTER TABLE ONLY public.model
    ADD CONSTRAINT model_pkey PRIMARY KEY (id);

ALTER TABLE ONLY public.test_execution
    ADD CONSTRAINT test_execution_pkey PRIMARY KEY (id);

ALTER TABLE ONLY public.test
    ADD CONSTRAINT test_pkey PRIMARY KEY (id);

ALTER TABLE ONLY public.transmitter
    ADD CONSTRAINT transmitter_name_key UNIQUE (name, ip);

ALTER TABLE ONLY public.transmitter
    ADD CONSTRAINT transmitter_pkey PRIMARY KEY (id);

ALTER TABLE ONLY public.command
    ADD CONSTRAINT command_button_id_fkey FOREIGN KEY (button_id) REFERENCES
public.button(id);

ALTER TABLE ONLY public.command_execution
    ADD CONSTRAINT command_execution_test_execution_id_fkey FOREIGN KEY
(test_execution_id) REFERENCES public.test_execution(id);

ALTER TABLE ONLY public.command
    ADD CONSTRAINT command_test_id_fkey FOREIGN KEY (test_id) REFERENCES
public.test(id);

ALTER TABLE ONLY public.freq
    ADD CONSTRAINT freq_button_id_fkey FOREIGN KEY (button_id) REFERENCES
public.button(id);

ALTER TABLE ONLY public.freq
    ADD CONSTRAINT freq_model_id_fkey FOREIGN KEY (model_id) REFERENCES
public.model(id);

ALTER TABLE ONLY public.test_execution
    ADD CONSTRAINT test_execution_test_id_fkey FOREIGN KEY (test_id)
REFERENCES public.test(id);

ALTER TABLE ONLY public.test_execution
    ADD CONSTRAINT test_execution_transmitter_id_fkey FOREIGN KEY
(transmitter_id) REFERENCES public.transmitter(id);

ALTER TABLE ONLY public.test
    ADD CONSTRAINT test_model_id_fkey FOREIGN KEY (model_id) REFERENCES
public.model(id);

--
-- PostgreSQL database complete
--
```