UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

Graduado en Ingeniería Informática

Implementación de un sistema de seguridad e iluminación para hogares utilizando ESP32

Home security and lighting system using ESP32

Realizado por
Bruno Biondi Chaves

Tutorizado por
Cristian Martín Fernández

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, septiembre 2025

Abstract

In our world, the trend for the past twenty or so years has been to better our lives through technology, not only with new innovations that completely change how we live them but also by modernizing systems and integrating them better with the rest of our new devices. All the tools that we developed to work with each other without our direct involvement are what we call the Internet of Things (IoT).

Throughout this study, we intend to integrate a security and lighting system into our already existing technological environment. With sensors controlled by an ESP32 development board and a connected Android application, we want to centralize all of our functions in a way that allows users to monitor, control, and easily interact all the individual parts of our system.

In this project, we will exhibit the design, development, and implementation of the physical aspects of the system as well as the software infrastructure that allows our users to make the most of it.

Our work here can be interpreted as a cheaper and more interesting alternative to pre-made security systems, providing a more accessible solution to home protection. By creating it ourselves we will also benefit from increased privacy and data control and technical freedom to add any functionalities that our particular home situation might need in the future.

This approach demonstrates practical IoT implementation for cost-effective home automation that prioritizes user control and flexibility. The integrated platform showcases DIY smart home potential while establishing a foundation for future scalability, making advanced security and automation accessible to more users while maintaining data privacy and system control.

Keywords: IoT, Security, Lighting, ESP32, Android Application

Resumen

En nuestro mundo, la tendencia desde hace unos 20 años ha sido mejorar nuestras vidas a través de la tecnología, no solo con nuevas innovaciones que pueden cambiar completamente cómo vivimos sino también modernizando sistemas existentes e integrándolos mejor con el resto de nuestros dispositivos. Todas las herramientas que desarrollamos para trabajar entre sí sin nuestra intervención directa es lo que llamamos el Internet de las Cosas (IoT).

A lo largo de este trabajo, pretendemos integrar un sistema de seguridad e iluminación en nuestro entorno tecnológico ya existente. Con sensores controlados por una placa de desarrollo ESP32 conectada con una aplicación Android, queremos centralizar todas nuestras funciones de manera que permita a los usuarios monitorear, controlar e interactuar fácilmente con todas las partes individuales de nuestro sistema.

En este proyecto, exhibiremos el diseño, desarrollo e implementación de los aspectos físicos del sistema, así como la infraestructura de software que permite a nuestros usuarios aprovecharlo al máximo.

Nuestro trabajo aquí puede interpretarse como una alternativa más barata y modular a los sistemas de seguridad prefabricados, proporcionando una solución más accesible para la protección del hogar. Al crearlo nosotros mismos también nos beneficiaremos de mayor privacidad y control de datos, y libertad técnica para agregar cualquier funcionalidad que nuestra situación particular del hogar pueda necesitar en el futuro.

Este enfoque demuestra una implementación práctica de IoT para automatización del hogar rentable que prioriza el control del usuario y la flexibilidad. La plataforma integrada muestra el potencial del hogar inteligente DIY mientras establece una base para escalabilidad futura, haciendo la seguridad avanzada y automatización accesible a más usuarios mientras mantiene la privacidad de datos y el control del sistema.

Palabras Clave: IoT, Seguridad, Iluminación, ESP32, Aplicación Android

Índice

1 Introducción

1.1. Motivación

Por Internet de las Cosas (IoT)[11] nos referimos a todos los dispositivos electrónicos que a través de internet u otras tecnologías se comunican entre sí, compartiendo datos recogidos por sensores físicos y permitiéndonos obtener una representación digital del mundo real. En las últimas décadas el IoT ha transformado muchas industrias y ha redefinido las maneras que tenemos de interactuar con el mundo físico desde los dispositivos digitales.

La importancia del Internet de las Cosas viene de su capacidad para informar decisiones en tiempo real y automatizar procesos que involucran el mundo físico y el digital. Su implementación ha convertido nuestro mundo en uno mucho mas interconectado, inteligente, eficiente y cómodo.

Hoy en día con los cambios en patrones de vida tras la pandemia, todos pasamos más tiempo en casa que nunca lo que ha otorgado al IoT doméstico una importancia y demanda nunca vista. Los hogares inteligentes han pasado de ser un lujo tecnológico a una necesidad práctica para muchos que puede mejorar la calidad de vida de sus habitantes. En 2024, se estima aproximadamente 100 Millones de hogares contienen un "Amazon Alexa"[8] (Figura 1), lo que indica que cada vez la población está más cómoda utilizando dispositivos digitales para cubrir funciones que antiguamente eran analógicas.

Figura 1: *Echo Dot*, altavoz "inteligente", fuente: Amazon

Los beneficios de los avances del IoT son numerosos. Económicamente optimizan el consumo de recursos gracias a automatización, lo que también reduce su impacto medioambiental. La relativa sencillez del IoT ha democratizado el acceso a muchas tecnologías avanzadas que antes solo estaban al alcance de grandes corporaciones. Un ejemplo puede ser este proyecto para detectar escapes de gas domésticos realizado en la India [12].

La motivación para implementar el Internet de las Cosas en un sistema de seguridad e iluminación surge de las limitaciones que traen consigo estos sistemas tradicionalmente. Las implementaciones convencionales y disponibles en el mercado actualmente suelen ser muy costosas [13] y dependen de actores externos, por lo que es difícil confiar en ellos plenamente y ofrecen poca personalización al usuario.

Gracias al IoT podemos obtener todos los beneficios de un sistema de seguridad e iluminación automatizado por una fracción del coste, con mejoras adicionales y manteniendo el control total sobre nuestros datos en todo momento.

1.2. Objetivos

Los objetivos de este proyecto son:

1. Diseñar, desarrollar e implementar un sistema de seguridad e iluminación automatizado que mantenga protegida la casa y permita la actuación remota de luces y sensores.

2. Utilizar como corazón del sistema un dispositivo IoT, en este caso una placa de desarrollo ESP32, que gracias a sensores de movimiento y luz y a un lector RFID, formará la capa de percepción del proyecto. A este dispositivo también estarán conectadas las luces LED que iluminarán la casa y que el usuario podrá manipular libremente.

3. Contar con un sistema de seguridad doméstico que permita monitorear y gestionar de manera centralizada la información de los dispositivos, garantizando comunicación fluida y control confiable desde un único punto de acceso.

4. Permitir que toda la interacción entre el sistema y el usuario se realice por medio de una aplicación Android, desde la cual se podrá monitorear todas las habitaciones del hogar con dispositivos IoT, encender y apagar luces y sensores, programar períodos de

activación o desactivación de estos, añadir o eliminar habitaciones y otras funciones. La aplicación se comunicará con la aplicación web y esta, a su vez, con el dispositivo IoT.

Es nuestro deber para los usuarios que todo el sistema sea intuitivo de usar y que mantenga sobre todo máxima funcionalidad y eficiencia. Los tiempos de respuesta de la aplicación deberán de ser mínimos y el sistema debe de ser fácil de usar y accesible para los usuarios.

1.3. Metodología de trabajo

Antes de comenzar a trabajar en un proyecto como este, es importante tener una metodología de trabajo establecida, es decir una serie de reglas y pasos a seguir para plantear correctamente el proyecto y poder controlar el progreso que vayamos haciendo.

Existen varias metodologías adaptadas a distintos tipos de proyectos. Se diferencian entre sí en las longitudes, productos finales y orden de cada ciclo de trabajo. Inicialmente planeábamos utilizar una metodología *Ágil* de desarrollo iterativo o *Scrum*, pero rápidamente descubrimos que los requisitos del sistema tenían que ser definidos en su totalidad desde el principio y que el alcance del proyecto era lo suficientemente limitado para poder diseñarlo utilizando el **Desarrollo en cascada**.

En esta metodología de trabajo las fases ocurren consecutivamente, es decir deben completarse una tras otra en el orden de la figura 2, en teoría sin posibilidad de retroceso. El proceso empieza con la fase de "Requisitos", donde definimos todos los requisitos funcionales y no funcionales que debe cumplir nuestro sistema final. Es importante que no dejemos huecos en esta fase, pues añadir requisitos más adelante implicaría muchos más cambios en la implementación que si estuviésemos utilizando una metodología ágil. Por otro lado, con los requisitos bien definidos el trabajo posterior se hace mucho más claro y focalizado.

Una vez definidos, podemos crear diagramas de casos de uso o de secuencia para indagar más en las acciones a llevar a cabo para cumplir dichos requisitos. Con estas herramientas podemos pasar al diseño, que en nuestro caso fue identificar todos los elementos participantes en el sistema final, y las interacciones entre estos. En esta fase también definiremos qué patrones de diseño vamos a utilizar, como el *MVC* y la división de clases en nuestro código. Para nuestro proyecto, las fases de implementación y verificación han sido realizadas conjuntamente, pues teniendo las funcionalidades del sistema definidas de manera clara podemos compartimentar

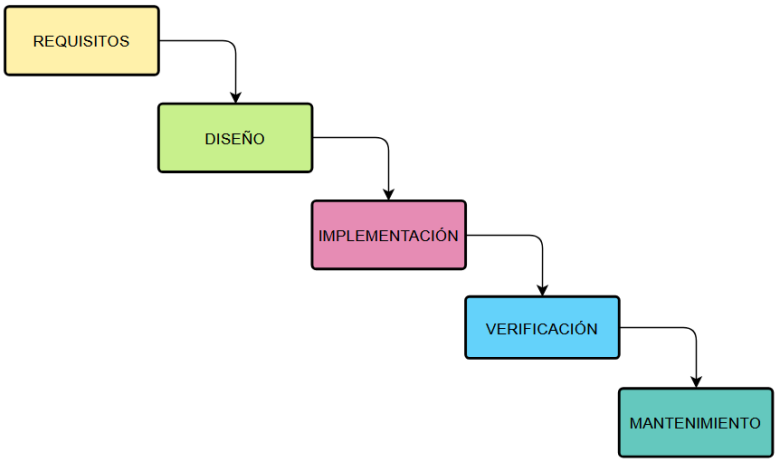


Figura 2: Metodología de desarrollo en cascada

cada nueva funcionalidad al código, ya sea del backend o frontend, y sus tests correspondientes para ir probando según se programa. Todo el código de todas las partes del proyecto, desde el frontend hasta el esp32, está disponible en el repositorio de GitHub.

La fase de mantenimiento en nuestro proyecto se manifestará principalmente como un corto período de corrección de bugs menores al final del desarrollo, monitoreo del sistema para controlar el uso de recursos y la creación de los manuales de instalación y de usuario.

1.4. Estructura del documento

1. **Tecnologías utilizadas**

   Esta sección cubre todas las tecnologías empleadas en el proyecto desde los componentes hardware individuales y los frameworks para el desarrollo software hasta los protocolos de comunicación del dispositivo IoT.

2. **Requisitos Identificados**

   En esta sección se listan todas las funciones que debe de llevar a cabo el sistema, las interacciones del usuario con este y cómo debe funcionar.

3. **Especificación**

   Se definen de manera detallada y precisa las interacciones del usuario con el sistema. Contiene el modelo de dominio y las descripciones y diagramas de los casos de uso, que

permiten un mejor entendimiento del flujo de trabajo y serán muy útiles a la hora de desarrollar con interacciones concretas en mente.

4. **Diseño del dispositivo IoT**

   Esta sección muestra el proceso de diseño e implementación del dispositivo IoT ESP32 con los diagramas necesarios.

5. **Backend Spring**

   En esta sección se detallan los pasos tomados para diseñar el servidor Spring Boot con servicios REST que recibirá los inputs del usuario y actuará de puente comunicador entre nuestro software y hardware mediante MQTT. Se explica además la estructura del backend de nuestro sistema, con un diagrama de clases, y el funcionamiento de las entidades con respecto a el servidor Spring Boot con servicios REST.

6. **Frontend Android**

   Está sección describe el proceso de desarrollo de la aplicación en Android que el usuario utilizará para controlar el sistema de seguridad. Se explicará tanto el diseño de la interfaz de usuario como las conexiones con el servidor central.

7. **Conclusiones**

   Se presentan las conclusiones extraídas durante el desarrollo del proyecto, con una valoración final y posibles extensiones futuras que se le puedan hacer a este.

2

Tecnologías Utilizadas

2.1. Entornos de trabajo

- **Visual Studio Code:** Todo el código desarrollado para el servidor Spring Boot se realizó dentro de VSCode, con extensiones para inicializar el proyecto Spring Boot y utilizando el control de versión de git integrado en la aplicación. Desde VSCode podemos ejecutar el servidor Spring con Maven y dejarlo corriendo localmente desde la terminal. También nos ofrece la funcionalidad de correr tests automáticamente si los hemos configurado. El lenguaje de programación utilizado en este entorno es **Java** (Figura 3).

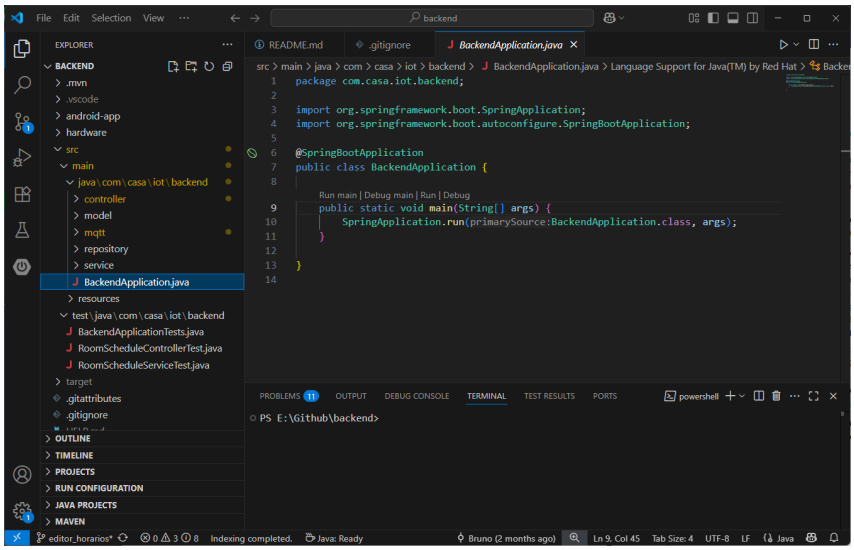


Figura 3: Vista del proyecto en VSCode

- **MySQL Workbench:** La creación del esquema de base de datos MySQL, perfiles de usuario y el monitoreo de las tablas se realizó a través de este programa. El lenguaje de

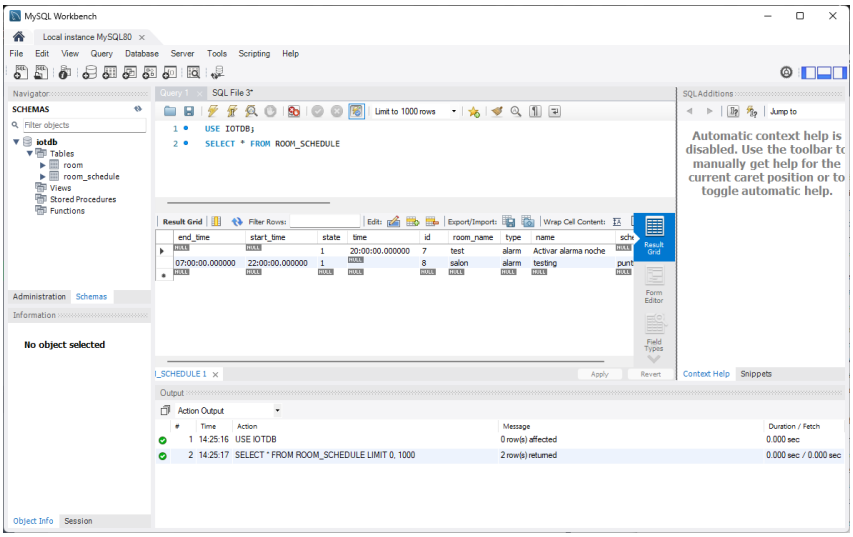


Figura 4: Workspace de MySQL Workbench

programación utilizado en este entorno es **SQL** (Figura 4).

- **Android Studio:** Este entorno (Figura 5) es el oficial de la plataforma móvil de Google, basado en IntelliJ IDEA. Se utilizó para hacer los componentes visuales y técnicos de la aplicación Android. Una de las características más valiosas de Android Studio es su emulador de dispositivo móvil integrado, que permite simular diferentes configuraciones de hardware, versiones de Android y tamaños de pantalla. Gracias a este emulador se pudo probar fácilmente cada iteración de la aplicación durante el desarrollo, verificando la funcionalidad y accesibilidad del diseño sin necesidad de un dispositivo físico.

  También el entorno nos facilita un editor visual para el Layout de la aplicación, lo que agilizó mucho la creación de las interfaces, y Gradle, su sistema para "buildear" que se ocupa de la gestión de dependencias, y de la compilación y despliegue de la aplicación. El lenguaje de programación utilizado en este entorno es **Kotlin** para toda la lógica de la aplicación y **XML** para el diseño de interfaces de usuario en archivos de layout [1].

- **Arduino IDE** Para la programación del dispositivo IoT ESP32[2] se utilizó este entorno (Figura 6) con la librería para ESP32 de *Espressif Systems* [7]. Desde este programa con el código adecuado podemos acceder a todas las funcionalidades que ofrece el kit de desarrollo. Arduino IDE agiliza el proceso de compilación y carga del código escrito, permitiéndonos pasar los cambios que hayamos hecho con un sólo botón y monitorear

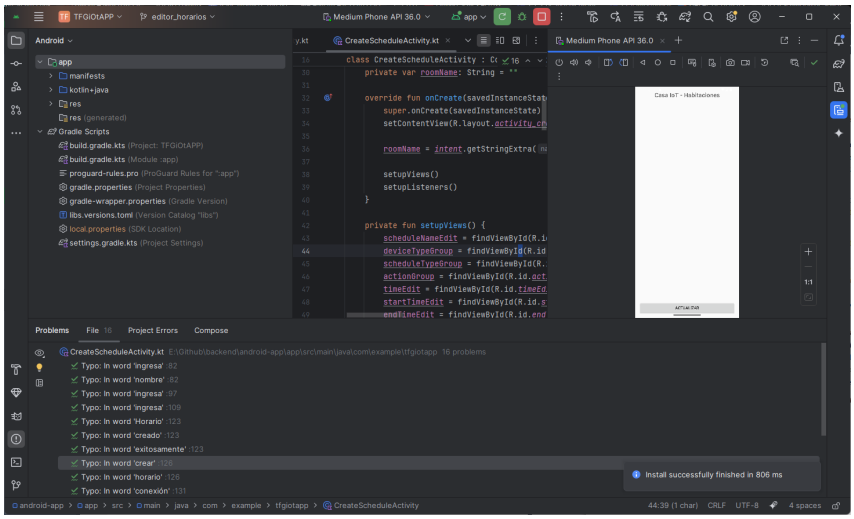


Figura 5: Proyecto en Android Studio con emulador

la ejecución en tiempo real a través de la consola serial.

2.2. Hardware IoT

Nuestro sistema de seguridad e iluminación se basa alrededor de la placa de desarrollo ESP32. Para el proyecto compramos un kit completo en internet que contenía todos los componentes que necesitábamos y más. Para este proyecto usaremos el microcontrolador en sí, sensores PIR para detectar movimiento, un buzzer que sirva de alarma contra intrusión y LEDs para simular las luces del hogar.

- **ESP32:** este microcontrolador desarrollado por Espressif Systems es el corazón físico de nuestro sistema. Su bajo precio y bajo consumo lo hacen la opción perfecta para un proyecto de IoT de funcionamiento continuo y responsivo como el nuestro. El ESP32 integra un procesador dual-core, funciones WiFi y Bluetooth y cuenta con múltiples pines GPIO programables digitales y analógicos. Podemos utilizar Arduino IDE para programar el funcionamiento de la placa y establecer el contacto MQTT con el servidor. Figura 7.

- **Sensor de movimiento HC-SR501:** utilizamos este sensor infrarrojo pasivo para detectar movimiento en nuestro hogar. Al ser un PIR detecta la radiación infrarroja (calor) que emiten los objetos y genera un cambio de voltaje en respuesta que nuestro microcontrolador entenderá como movimiento detectado. Se puede ajustar el tiempo de

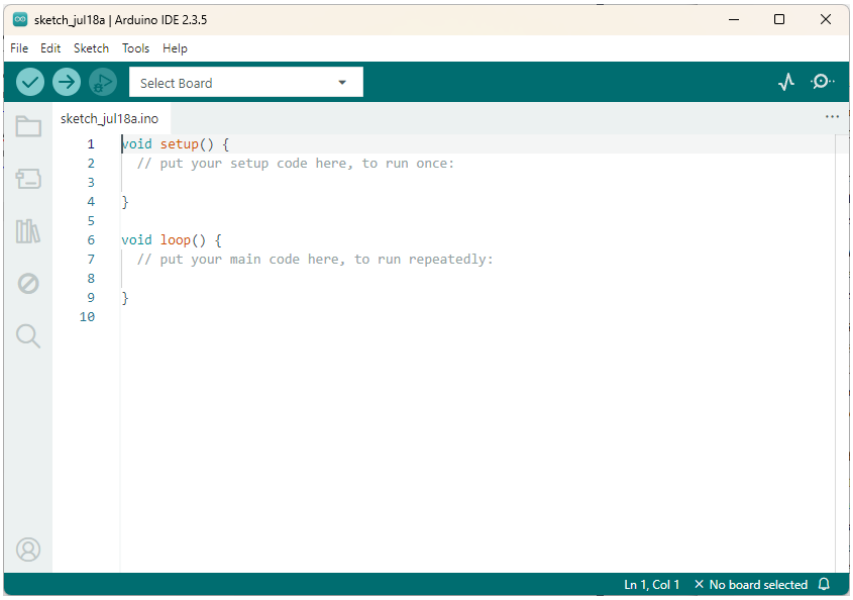


Figura 6: Sketch inicial en Arduino IDE

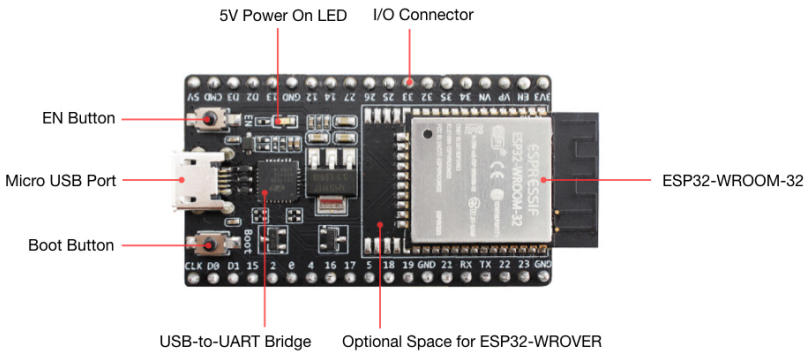


Figura 7: ESP32-DevKitC V4 con ESP32-WROOM-32, *Espressif Systems*

Figura 8: Sensor HC-SR501

Figura 9: RFID-RC522 con tarjeta y tag incluido

activación según necesitemos y tiene un alcance de unos 3 metros. Figura 8.

- **RFID-RC522:** módulo lector/escritor para tarjetas RFID. Incluye una tarjeta RFID para escribir y un pequeño tag. Funciona generando un campo magnético que detecta los objetos compatibles por inducción, lee su UID (identificador) y envía los datos al microcontrolador. Figura 9.

- **Buzzer activo:** con este zumbador activo podemos emitir un sonido continuo cuando se cumplen ciertas condiciones, esto servirá de alarma antirrobo si uno de nuestros sensores PIR detecta movimiento. Figura 10.

Figura 10: Buzzer activo genérico

Figura 11: LEDs bicolor

Figura 12: Logo de Spring

- **LEDs:** estos *Light Emitting Diodes* simularán las luces de nuestra casa en el proyecto. El usuario podrá controlar el estado de estos y programarlos desde la aplicación. Es un componente muy común en proyectos IoT debido a su bajo coste y su eficiencia energética. Figura 11.

2.3. Tecnologías del Backend

- **Spring Boot:** para la creación del servidor se utilizó Spring Boot[14], un framework de código abierto que facilita y simplifica el proceso, configurando componentes y manejando injección de dependencias. Nos prepara el workspace para el proyecto en pocos pasos. Figura 12.

- **Spring Web:** módulo del framework Spring para crear controladores REST y manejar peticiones HTTP. Ofrece conversión de objetos entre formatos y etiquetas que hacen el servicio más sencillo y compacto.

- **Spring Data JPA:** nos proporciona interfaces para la generación de consultas SQL basadas en nombres de métodos, lo que simplifica notablemente la comunicación con la base de datos y permite la creación de entidades para nuestro modelo.

- **Spring Scheduling y MQTT:** módulos para la creación de tareas programadas y el manejo de comunicaciones MQTT respectivamente.

- **Eclipse Paho MQTT Client:** cliente Java para comunicación MQTT. Con él podemos conectar el servidor a brokers MQTT; suscribirnos a tópicos y publicar en ellos.
- **Maven:** herramienta de gestión de dependencias que descarga a nuestro equipo todas las dependencias del proyecto y prepara el código para su despliegue.
- **JUnit 5:** framework para testing, creación y ejecución de pruebas automatizadas. Spring además incluye MockMVC para crear mocks de objetos de nuestra aplicación para usar en los tests.
- **JaCoCo:** herramienta de análisis de cobertura de código que genera informes detallados sobre el porcentaje de líneas y ramas cubiertas por los tests.
- **MySQL:** utilizamos una base de datos MySQL para almacenar toda la información sobre habitaciones y acciones programadas en nuestra aplicación.

2.4. Tecnologías del Frontend

- **Android SDK:** kit de desarrollo que nos proporciona APIs para acceder a partes del dispositivo android como el GPS, sensores o notificaciones.
- **Material Design 3:** Sistema de diseño de Google que nos otorga gran cantidad de paquetes de componentes enfocados a la creacion de interfaces de usuarios modernas. En nuestra aplicación usamos botones, cartas y la navegación.
- **OkHttp:** cliente para la comunicación con la API REST mediante HTTP.
- **Gson:** librería para convertir objetos Java/Kotlin a JSON y viceversa.
- **RecyclerView:** widget android para mostrar listas de objetos de manera compacta y legible.
- **Gradle:** gestor de dependencias similar a Maven para el backend. Genera el .APK final y lo despliega en el dispositivo.
- **Firebase Cloud Messaging:** servicio de mensajería en la nube de Google que permite enviar notificaciones push a los dispositivos Android. En la aplicación se utiliza para recibir alertas de movimiento en tiempo real.

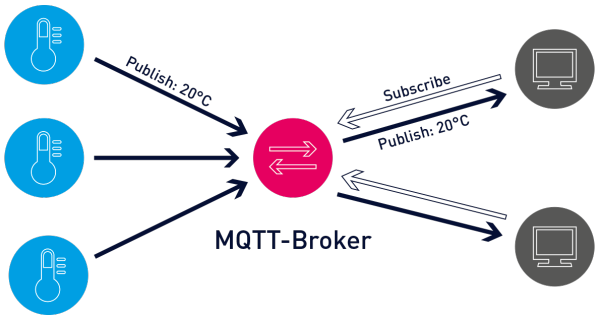


Figura 13: Estructura MQTT

2.5. Otros

- **MQTT (Message Queuing Telemetry Transport):** protocolo de mensajería especializado para IoT. Utiliza el modelo publicación/suscripción, en el que los dispositivos (sensores) o clientes publican datos a un tópico concreto, a los que solo podrán acceder dispositivos suscritos a el mismo tópico. Su implementación ligera y eficiente lo hace perfecto para comunicar nuestro backend y aplicación con el ESP32, y la división por tópicos mantiene los mensajes claros y divididos. Figura 13.

- **WiFi:** lo utilizamos para conectar el ESP32 con el backend de manera inalámbrica.

- **Arquitectura REST:** estructura de endpoints HTTP para operaciones CRUD. Es el corazón de las interacciones entre nuestra aplicación Android, que utiliza los endpoints para enviar y recibir información, y el backend.

- **MVC:** patrón de diseño utilizada en el desarrollo del backend que divide la lógica de negocio y datos (services y model) de las vistas y los controladores, para un trabajo más claro y organizado.

- **Git:** sistema de control de versiones para rastrear los cambios que hacemos en el código. En nuestro caso todo el código se ha contenido en un mismo repositorio en GitHub, con ramas creadas para añadir funcionalidades nuevas que una vez terminadas y probadas se pasan a la rama principal. VSCode incluye control de versión con Git, permitiéndonos hacer commits y manejar cambios sin tener que salir del entorno, lo que agiliza todo el proceso.

- **WiFiManager:** es una librería utilizada en el ESP32 para facilitar la configuración de la conexión WiFi y otros parámetros de red, como los datos del broker MQTT. Cuando el dispositivo no tiene credenciales guardadas o no puede conectarse a la red, WiFiManager crea automáticamente un punto de acceso y un portal cautivo al que el usuario puede acceder desde su móvil o PC e introducir el SSID y la contraseña del WiFi y los parámetros para la conexión al broker MQTT, lo que hace nuestro sistema más dinámico y nos permite instalarlo en otros sitios sin cambiar código.

3

Descripción general y requisitos

Este proyecto consiste en el desarrollo de un sistema integral de seguridad e iluminación doméstica utilizando el Internet de las Cosas. Para proporcionar toda la funcionalidad requerida el sistema está compuesto por un microcontrolador ESP32 con todo el hardware (LEDs, buzzer, sensores PIR, etc) conectado, un servidor REST para procesar toda la información que entra y sale, una base de datos MySQL para almacenar información del usuario y una aplicación Android como frontend del sistema para interactuar con este como se observa en la figura 14.

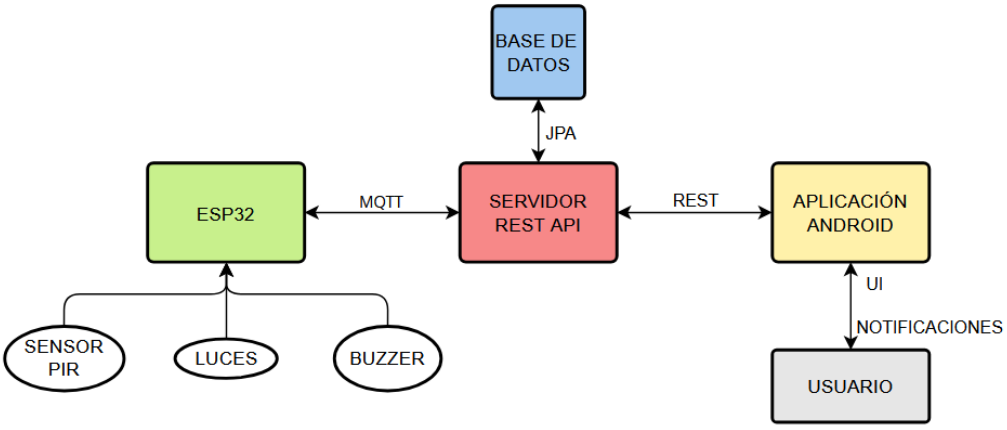


Figura 14: Vista general del sistema

3.1. Descripción general

Nuestro principal objetivo con este proyecto es el correcto despliegue del sistema de seguridad e iluminación en un entorno simulando una casa real. Instalaremos todo el hardware en un modelo de una casa construido con *LEGO* y probaremos todas las funcionalidades del

sistema a esa escala. El sistema debe ante todo proporcionar la seguridad esperada con los sensores PIR, avisando a los usuarios sobre posibles intrusos mediante notificaciones *Push* y permitiendo el control fiable de las luces del hogar desde la aplicación móvil.

El usuario podrá registrar una tarjeta RFID y utilizarla cuando tenga sensores de movimiento activados para desactivarlos sin necesidad de acceder a la aplicación. Otra funcionalidad es que los usuarios puedan crear eventos programados llamados "horarios" para ejecutar una acción a una hora determinada en una habitación específica.

Por otro lado, también tenemos el objetivo de crear un sistema de seguridad que a diferencia de las soluciones comerciales no requiere suscripciones mensuales, ofreciendo una alternativa open-source, sin costos recurrentes y personalizable.

Para el correcto funcionamiento del sistema necesitaremos implementar un servidor REST robusto con MQTT integrado para manejar la interacción bidireccional con los dispositivos IoT. A este se conectará una aplicación Android nativa con interfaz moderna e intuitiva que integre notificaciones push para alertas de movimiento.

3.2. Alcance

El proyecto abarca el desarrollo completo de un sistema IoT de seguridad e iluminación doméstica que integra tres componentes principales: un backend Spring Boot que actúa como servidor central, una aplicación Android nativa para el control remoto, y una placa ESP32 con sensores RFID, PIR y LEDs que simulan el sistema de iluminación real.

El alcance incluye todas las funcionalidades especificadas en los requisitos funcionales (RF-01 a RF-14), desde la gestión de habitaciones y control de luces hasta la programación de horarios, detección de movimiento, control RFID y sistema de notificaciones. Se desarrollará un prototipo funcional completo para demostrar la viabilidad técnica del concepto, incluyendo comunicación MQTT entre dispositivos, APIs REST para la aplicación móvil, y base de datos para persistencia de configuraciones. El proyecto se limita a una implementación de demostración con componentes básicos, pero el resultado final será un sistema completamente operativo que sirva como prueba de concepto para una implementación futura en un hogar real.

3.3. Suposiciones y dependencias

- **Suposiciones:** El proyecto asume que los usuarios finales disponen de una red WiFi doméstica estable con acceso a internet, dispositivos Android compatibles (versión 7.0 o superior), y la instalación correcta de los componentes IoT.

- **Dependencias:** El sistema depende de las librerías Spring Boot y el ecosistema Android para el correcto funcionamiento del backend y la aplicación móvil, así como de la disponibilidad del servicio de notificaciones push de Google (FCM). La comunicación IoT requiere un broker MQTT interno funcional y conectividad WiFi continua entre dispositivos ESP32 y el servidor, mientras que la persistencia de datos depende de la estabilidad de la base de datos MySQL configurada localmente.

3.4. Requisitos funcionales

- **RF-01 - Gestión de habitaciones:** el sistema permitirá al usuario crear, consultar y eliminar las habitaciones de su casa que contengan componentes de IoT compatibles.

- **RF-02 - Control de iluminación:** el sistema permitirá al usuario controlar, mediante la aplicación Android, las luces de la casa que tengan conectadas al sistema.

- **RF-03 - Control de sensores:** el sistema permitirá al usuario controlar, mediante la aplicación Android, los sensores de movimiento PIR que estén conectados al sistema.

- **RF-04 - Detección de movimiento:** el sistema correctamente identificará movimiento en las habitaciones con sensores presentes.

- **RF-05 - Notificaciones:** el sistema enviará notificaciones a los usuarios cuando se detecte movimiento en un sensor marcado como encendido.

- **RF-06 - Horarios puntuales:** el sistema permitirá a los usuarios programar acciones que serán ejecutadas a una hora concreta definida por estos.

- **RF-07 - Horarios de intervalo:** el sistema permitirá a los usuarios programar acciones que se llevarán a cabo en intervalos de hora concretos.

- **RF-08 - Ejecución de horarios:** cuando el sistema reconozca que hay una acción programada que tomar, esta se ejecutará correctamente y notificará al usuario.

- **RF-09 - Monitoreo en tiempo real:** el sistema permitirá a los usuarios monitorear todas las habitaciones registradas en la aplicación, mostrando el estado de las luces y los sensores en tiempo real.

- **RF-10 - Gestión tarjetas RFID:** el sistema permitirá a los usuarios registrar una tarjeta o objeto RFID que se asociará a su cuenta para el uso.

- **RF-11 - Desactivación por RFID:** el sistema permitirá a los usuarios desactivar las medidas de seguridad con el objeto RFID registrado.

- **RF-12 - Perfiles de usuario:** el sistema podrá crear de perfiles de usuario para mayor seguridad, autenticación y para permitirá designar roles como administrador, que ofreceran mayor control de la aplicación..

- **RF-13 - Modo 'vacaciones':** el sistema encenderá y apagará luces de manera semi-aleatoria para simular que hay alguien en casa para desalentar posibles criminales.

- **RF-14 - Logging:** el sistema mantendrá una lista de las habitaciones y los horarios presentes y eventos ocurridos en el día.

3.5. Requisitos no funcionales

- **RNF-01 - Rendimiento:** El sistema debe proporcionar respuesta a las interacciones del usuario en menos de 2 segundos y procesar los eventos que lanzan los componentes IoT en tiempo real. Este rendimiento debe mantenerse con al menos dos dispositivos conectados simultáneamente.

- **RNF-02 - Disponibilidad:** El sistema debe funcionar de manera continua con interrupciones mínimas, y ser capaz de recuperarse automáticamente de fallos temporales.

- **RNF-03 - Seguridad:** Todas las comunicaciones deberán de ir cifradas, MQTT utilizando TLS y las comunicaciones entre aplicación y servidor con HTTPS.

- **RNF-04 - Usabilidad:** La aplicación Android debe de ser fácil de instalar y utilizar, con una interfaz intuitiva y moderna.
- **RNF-05 - Compatibilidad:** El sistema debe funcionar en redes WiFi domésticas típicas, ser compatible con Android 7.0 o más y el backend debe de funcionar en Windows y Linux.
- **RNF-06 - Eficiencia energética:** El sistema debe optimizar el consumo energético de la placa ESP32 para operación continua, utilizando los recursos de memoria y procesamiento de manera eficiente.
- **RNF-07 - Escalabilidad:** El sistema debe soportar hasta 20 habitaciones registradas y el uso concurrente por parte de hasta 3 usuarios sin afectar al rendimiento de este.
- **RNF-08 - Mantenibilidad:** Deben de habilitarse actualizaciones OTA (inalámbricas) del microcontrolador ESP32.
- **RNF-09 - Privacidad:** Permitir eliminación completa de datos del usuario, mantener toda la información en red local sin dependencia de servicios cloud externos.

4

Especificación

La especificación de requisitos constituye el fundamento conceptual sobre el cual se construye todo el sistema de seguridad e iluminación desarrollado en este proyecto. Esta fase crítica del desarrollo establece de manera formal y detallada qué debe hacer el sistema y cómo debe comportarse, actuando como un contrato técnico entre las necesidades identificadas y la implementación real. Tras analizar las funcionalidades requeridas, se han definido catorce requisitos funcionales que abarcan desde la gestión básica de habitaciones y control de iluminación hasta características avanzadas como programación de horarios, control por RFID y modo vacaciones. Complementariamente, se han establecido nueve requisitos no funcionales que garantizan aspectos críticos como rendimiento, seguridad, usabilidad y compatibilidad del sistema.

Esta especificación no solo sirve como guía para el desarrollo de los componentes backend Spring Boot, aplicación Android y dispositivos ESP32, sino que también define los criterios de aceptación y las métricas de éxito que validarán la correcta implementación del sistema completo.

4.1. Modelo de dominio

El modelo de dominio (Figura 15) es una representación conceptual que identifica y define las entidades principales del negocio, sus atributos y las relaciones entre ellas dentro del contexto específico del proyecto. En el sistema IoT de seguridad e iluminación doméstica, este modelo incluye las entidades: User (usuarios del sistema), Room (habitaciones con sus estados de iluminación y sensores), Device (sensores PIR, LEDs), Schedule (horarios programados), Event (logs de eventos), y RfidCard (tarjetas RFID).

Utilizando este modelo podemos establecer una serie de “reglas”:

- Un usuario puede ser dueño y controlar varias habitaciones y cada habitación puede ser

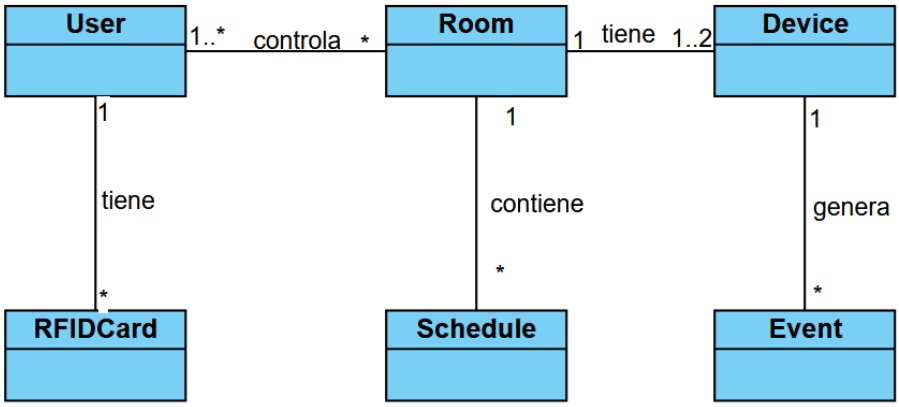


Figura 15: Modelo de dominio

controlada por varios usuarios si no se establece control de acceso.

- Una habitación puede tener hasta 2 dispositivos, una luz y un sensor de movimiento.
- Una habitación puede tener cualquier número de horarios programados.
- Un dispositivo hardware genera eventos y reporta su estado periódicamente.
- Un usuario puede tener varias tarjetas RFID.

4.2. Casos de uso

Los casos de uso describen las interacciones específicas entre usuarios y el sistema IoT, detallando cómo se ejecutan las funcionalidades de control de iluminación, detección de movimiento, programación de horarios y gestión de seguridad. Cada caso documenta el flujo completo desde la aplicación Android hasta los dispositivos ESP32, pasando por el backend Spring Boot, especificando precondiciones, pasos de ejecución, escenarios alternativos y post-condiciones de éxito. Esto traduce los requisitos funcionales RF-01 a RF-15 en experiencias de usuario que podremos utilizar como luz guía a la hora de implementar en el código.

4.2.1. CU-01: Gestionar Habitaciones

Campo	Descripción
Requisito	RF-01 (Gestión habitaciones)
Actor Principal	Usuario administrador
Precondición	Usuario es administrador
Flujo Principal	1. Usuario accede a la aplicación 2. Sistema muestra lista de habitaciones existentes 3. Usuario selecciona “Crear nueva habitación” 4. Usuario ingresa el nombre de la habitación 5. Sistema valida datos y crea habitación en base de datos 6. Sistema confirma creación exitosa
Postcondición	Nueva habitación disponible en el sistema
Flujos Alternativos	▪ **A1:** Nombre duplicado 1. Sistema valida nombre único 2. App muestra error “Habitación ya existe” 3. Usuario debe elegir nombre diferente ▪ **A2:** Datos inválidos 1. Sistema detecta el campo vacíos o caracteres no válidos 2. App resalta campos erróneos 3. Usuario corrige información requerida

Cuadro 1: Caso de Uso CU-01: Gestionar Habitaciones

4.2.2. CU-02: Controlar Iluminación

Campo	Descripción
Requisito	RF-02 (Luces)
Actor Principal	Usuario
Precondición	Habitación conectada y creada, dispositivo ESP32 conectado
Flujo Principal	1. Usuario selecciona la habitación en la aplicación 2. Sistema muestra estado actual de la iluminación 3. Usuario presiona switch para encender o apagar 4. App envía POST al backend 5. Backend publica mensaje MQTT al topic específico 6. ESP32 recibe comando y cambia estado del LED 7. ESP32 confirma cambio via MQTT 8. Sistema actualiza estado en base de datos 9. App refleja nuevo estado en la interfaz
Postcondición	Estado de iluminación modificado y sincronizado
Flujos Alternativos	▪ **A1:** Dispositivo offline 1. Sistema detecta falta de respuesta del ESP32 2. App muestra mensaje “Dispositivo no disponible” ▪ **A2:** Fallo de comunicación 1. Error en transmisión MQTT 2. Sistema registra error en logs

Continúa en la siguiente página...

Tabla 2 – continuación de la página anterior

Campo	Descripción

Cuadro 2: Caso de Uso CU-02: Controlar Iluminación

4.2.3. CU-03: Gestionar Sensores de Movimiento

Campo	Descripción
Requisito	RF-03 (Sensores de movimiento)
Actor Principal	Usuario
Precondición	Usuario autenticado, habitación configurada
Flujo Principal	1. Usuario navega a configuración de habitación 2. Sistema muestra estado actual del sensor de movimiento 3. Usuario cambia estado (Activar/Desactivar) 4. App envía comando al backend via API REST 5. Backend actualiza configuración en base de datos 6. Backend envía comando MQTT al ESP32 7. ESP32 configura sensor según nuevo estado 8. Sistema confirma cambio exitoso
Postcondición	Sensor configurado según preferencia del usuario

Continúa en la siguiente página...

Tabla 3 – continuación de la página anterior

Campo	Descripción
Flujos Alternativos	▪ **A1:** Dispositivo no responde 1. ESP32 no confirma recepción de comando 2. Sistema mantiene estado anterior ▪ **A2:** Fallo de comunicación 1. Error en transmisión MQTT 2. Sistema registra error en logs

Cuadro 3: Caso de Uso CU-03: Gestionar Sensores de Movimiento

4.2.4. CU-04: Detectar Movimiento

Campo	Descripción
Requisito	RF-04 (El sistema detectará movimiento cuando un sensor esté activo)
Actor Principal	Sensor PIR (ESP32)
Precondición	Sensor activado, sistema en funcionamiento
Continúa en la siguiente página...	

Tabla 4 – continuación de la página anterior

Campo	Descripción
Flujo Principal	1. Sensor PIR detecta movimiento en habitación 2. ESP32 procesa señal del sensor 3. ESP32 publica evento MQTT al backend 4. Backend recibe y procesa evento de movimiento 5. Sistema registra evento en base de datos 6. Backend evalúa si debe activar iluminación automática 7. Sistema prepara notificación para usuarios
Postcondición	Evento registrado, acciones automáticas ejecutadas
Flujos Alternativos	▪ **A1:** Sensor inactivo 1. ESP32 recibe señal pero sensor está desactivado 2. Sistema ignora evento de movimiento 3. No se genera registro ni notificación ▪ **A2:** Fallo de comunicación 1. Error en transmisión MQTT al backend 2. ESP32 almacena evento localmente 3. Sistema reintenta envío cuando conexión se restaure

Cuadro 4: Caso de Uso CU-04: Detectar Movimiento

4.2.5. CU-05: Recibir Notificaciones de Movimiento

Campo	Descripción
Requisito	RF-05 (El usuario recibirá una notificación push cuando se detecte movimiento)
Actor Principal	Sistema Backend
Precondición	Movimiento detectado, usuario con notificaciones habilitadas
Flujo Principal	1. Sistema recibe evento de movimiento desde ESP32 2. Backend verifica que sensor está activo 3. Sistema identifica usuarios a notificar 4. Backend genera contenido de notificación push 5. Sistema envía notificación via Firebase FCM 6. Servicio push entrega notificación a dispositivo Android 7. Usuario recibe alerta en tiempo real 8. Sistema registra envío de notificación en logs
Postcondición	Usuario notificado del evento de seguridad
Flujos Alternativos	▪ **A1:** Notificaciones deshabilitadas 1. Usuario ha desactivado el sensor 2. Sistema registra evento pero no envía alerta 3. Evento queda disponible en historial ▪ **A2:** Fallo de servicio push 1. Error en Firebase FCM o conectividad 2. Se registra fallo en logs para diagnóstico

Continúa en la siguiente página...

Tabla 5 – continuación de la página anterior

Campo	Descripción

Cuadro 5: Caso de Uso CU-05: Recibir Notificaciones de Movimiento

4.2.6. CU-06: Programar Horarios Puntuales

Campo	Descripción
Requisito	RF-06 (Programar horarios puntuales)
Actor Principal	Usuario
Precondición	Usuario autenticado, habitación existente
Flujo Principal	1. Usuario selecciona “Editar” a la derecha de “Horarios” 2. Usuario selecciona “Añadir nuevo horario” 3. Usuario configura el nombre del horario 4. Usuario selecciona el dispositivo a utilizar 5. Usuario elige el tipo de horario “Puntual” 6. Usuario selecciona la acción a tomar (encender o apagar) 7. Usuario introduce una hora concreta 8. Sistema valida configuración de horario 9. Backend guarda horario en base de datos 10. Sistema programa tarea automática con scheduler 11. Sistema confirma horario creado exitosamente
Postcondición	Horario programado y activo en el sistema

Continúa en la siguiente página...

Tabla 6 – continuación de la página anterior

Campo	Descripción
Flujos Alternativos	▪ **A1:** Datos inválidos 1. Hora fuera de rango o formato incorrecto 2. Sistema muestra error de validación 3. Usuario corrige información antes de continuar

Cuadro 6: Caso de Uso CU-06: Programar Horarios Puntuales

4.2.7. CU-07: Programar Horarios de Intervalo

Campo	Descripción
Requisito	RF-07 (Programar horarios de intervalo)
Actor Principal	Usuario
Precondición	Usuario autenticado, habitación configurada
Continúa en la siguiente página...	

Tabla 7 – continuación de la página anterior

Campo	Descripción
Flujo Principal	1. Usuario selecciona “Editar” a la derecha de “Horarios” 2. Usuario selecciona “Añadir nuevo horario” 3. Usuario configura el nombre del horario 4. Usuario selecciona el dispositivo a utilizar 5. Usuario elige el tipo de horario “Intervalo” 6. Usuario introduce una hora de inicio y de fin 7. Sistema valida configuración de horario 8. Backend guarda horario en base de datos 9. Sistema programa tarea automática con scheduler 10. Sistema confirma horario creado exitosamente
Postcondición	Horario de intervalo activo y funcional
Flujos Alternativos	▪ **A1:** Intervalo inválido 1. Hora fin anterior a hora inicio 2. Sistema muestra error de validación temporal 3. Usuario corrige horarios antes de guardar

Cuadro 7: Caso de Uso CU-07: Programar Horarios de Intervalo

4.2.8. CU-08: Ejecutar Horarios Automáticos

Campo	Descripción
Requisito	RF-08 (Ejecución de horarios)
Actor Principal	Sistema Backend
Precondición	Horarios programados activos, sistema funcionando
Flujo Principal	1. Scheduler del backend detecta horario a ejecutar 2. Sistema verifica que horario sigue activo 3. Backend identifica habitación y acción objetivo 4. Sistema determina comando MQTT correspondiente 5. Backend publica comando al topic del ESP32 6. ESP32 ejecuta acción (enciende/apaga luz o sensor) 7. ESP32 confirma ejecución exitosa via MQTT 8. Sistema registra evento en log de actividades 9. Backend actualiza estado en base de datos
Postcondición	Acción programada ejecutada correctamente
Flujos Alternativos	▪ **A1:** Dispositivo offline 1. ESP32 no responde al comando MQTT 2. Sistema registra fallo de ejecución 3. Backend reintenta comando cada 5 minutos ▪ **A2:** Final de horario 1. El sistema detecta que un horario acaba de terminar 2. Se revierte el dispositivo afectado a su estado previo

Continúa en la siguiente página...

Tabla 8 – continuación de la página anterior

Campo	Descripción

Cuadro 8: Caso de Uso CU-08: Ejecutar Horarios Automáticos

4.2.9. CU-09: Visualizar Estado en Tiempo Real

Campo	Descripción
Requisito	RF-09 (Visualización en tiempo real)
Actor Principal	Usuario
Precondición	Usuario autenticado, dispositivos conectados
Flujo Principal	1. Usuario abre dashboard principal de la app 2. App solicita estado actual al backend via GET /rooms 3. Backend consulta estado de todas las habitaciones 4. Sistema retorna estado de luces y sensores 5. App actualiza interfaz con información actual 6. Usuario presiona “Actualizar” 7. Sistema retorna estado de luces y sensores
Postcondición	Usuario ve estado actualizado del sistema

Continúa en la siguiente página...

Tabla 9 – continuación de la página anterior

Campo	Descripción
Flujos Alternativos	▪ **A1:** Conexión lenta 1. Timeout en petición HTTP al backend 2. App muestra indicador de "Cargando..." 3. Sistema reintenta petición ▪ **A2:** Datos desactualizados 1. Dispositivo ESP32 no reporta estado reciente 2. Usuario puede forzar actualización manual

Cuadro 9: Caso de Uso CU-09: Visualizar Estado en Tiempo Real

4.2.10. CU-10: Gestión tarjetas RFID

Campo	Descripción
Requisito	RF-15 (Gestión de tarjetas RFID)
Actor Principal	Usuario autenticado
Precondición	Usuario autenticado en la aplicación, lector RFID operativo, conexión con backend y ESP32
	Continúa en la siguiente página...

Tabla 10 – continuación de la página anterior

Campo	Descripción
Flujo Principal	1. Usuario accede a la sección de gestión de tarjetas RFID en la app Android 2. App consulta al backend si el usuario ya tiene una tarjeta asociada 3. Sistema muestra el UID de la tarjeta si existe, o indica que no hay tarjeta registrada 4. Usuario pulsa "Registrar nueva" 5. App solicita al backend iniciar el proceso de registro RFID 6. Backend envía comando al ESP32 para activar el lector RFID 7. App muestra mensaje indicando que debe pasar una tarjeta por el lector en 30 segundos 8. Usuario acerca una tarjeta RFID al lector 9. ESP32 detecta el UID y lo envía al backend por MQTT 10. Backend asocia el UID a la cuenta del usuario y lo almacena en la base de datos 11. App muestra el nuevo UID asociado al usuario cuando este lo pide
Postcondición	Tarjeta RFID asociada al usuario y visible en la app

Continúa en la siguiente página...

Tabla 10 – continuación de la página anterior

Campo	Descripción
Flujos Alternativos	▪ **A1:** Usuario cancela el registro 1. Usuario pulsa "Cancelar" en la app durante el proceso 2. App solicita al backend cancelar el registro 3. Backend envía comando de cancelación al ESP32 4. Backend envía notificación push de cancelación a la app 5. App muestra mensaje de cancelación al usuario ▪ **A2:** Tiempo de espera agotado 1. Usuario no pasa ninguna tarjeta en 30 segundos 2. Backend cancela el proceso y notifica a la app 3. App muestra mensaje de error o timeout ▪ **A3:** Error de lectura 1. ESP32 no puede leer correctamente la tarjeta 2. Backend informa a la app del error 3. App muestra mensaje de error y permite reintentar

Cuadro 10: Caso de Uso CU-10: Gestión tarjetas RFID

4.2.11. CU-11: Desactivar Sistema con RFID

Campo	Descripción
Requisito	RF-10 (Desactivación por RFID)
Actor Principal	Usuario con tarjeta RFID

Continúa en la siguiente página...

Tabla 11 – continuación de la página anterior

Campo	Descripción
Precondición	Tarjeta registrada, lector RFID operativo
Flujo Principal	1. Usuario aproxima tarjeta RFID al lector RC522 2. ESP32 lee UID de la tarjeta 3. ESP32 envía UID al backend via MQTT 4. Backend verifica tarjeta en base de datos 5. Sistema identifica usuario propietario de la tarjeta 6. Backend ejecuta desactivación de todos los sensores activos 7. Sistema envía comandos MQTT a todos los ESP32 8. Dispositivos confirman desactivación de sensores 9. Sistema registra uso de tarjeta en logs 10. Backend notifica resultado a la app Android
Postcondición	Todos los sensores de movimiento desactivados

Continúa en la siguiente página...

Tabla 11 – continuación de la página anterior

Campo	Descripción
Flujos Alternativos	▪ **A1:** Tarjeta no reconocida 1. UID no existe en base de datos 2. Sistema registra intento de acceso no autorizado 3. Buzzer emite alerta sonora de rechazo ▪ **A2:** Lector con fallo 1. RC522 no puede leer tarjeta correctamente 2. Sistema registra error de hardware 3. Usuario debe intentar nuevamente o usar app

Cuadro 11: Caso de Uso CU-11: Desactivar Sistema con RFID

4.2.12. CU-12: Gestionar Perfiles de Usuario

Campo	Descripción
Requisito	RF-11 (Perfiles de usuario)
Actor Principal	Sistema
Descripción	Se pueden crear usuarios en la base de datos, cada uno con su usuario y contraseña, y asignar roles de administrador o usuario general. El sistema almacena estos perfiles y ofrece un servicio de autenticación para que solo usuarios que se hayan registrado en el backend puedan acceder a la aplicación y controlar los dispositivos.

Cuadro 12: Caso de Uso CU-12: Perfiles de Usuario

4.2.13. CU-13: Activar Modo Vacaciones

Campo	Descripción
Requisito	RF-12 (Modo "vacaciones")
Actor Principal	Usuario Propietario
Precondición	Usuario ADMIN, habitaciones configuradas
Flujo Principal	1. Usuario accede a la pantalla principal 2. Usuario pulsa "Modo vacaciones" 3. Sistema bloquea las funciones adicionales mientras esté activado el modo vacaciones 4. Mientras el modo está activado, el sistema simula actividad en el domicilio 5. Usuario pulsa "Desactivar modo vacaciones" 6. Usuario establece horarios de actividad simulada 7. Sistema restaura las funciones de la aplicación
Postcondición	Modo vacaciones activo simulando presencia

Cuadro 13: Caso de Uso CU-13: Activar Modo Vacaciones

4.2.14. CU-14: Consultar Logs del Sistema

Campo	Descripción
Requisito	RF-13 (Logging)
Actor Principal	Usuario Administrador
Precondición	Usuario con rol ADMIN

Continúa en la siguiente página...

Tabla 14 – continuación de la página anterior

Campo	Descripción
Flujo Principal	1. Usuario accede a “Logs de actividad” 2. Sistema muestra filtros disponibles (fecha, tipo, habitación) 3. Usuario aplica filtros deseados 4. Backend consulta eventos en base de datos 5. Sistema ordena eventos cronológicamente 6. App muestra lista paginada de eventos 7. Usuario puede ver detalles de eventos específicos 8. Sistema permite exportar logs seleccionados
Postcondición	Usuario accede al historial filtrado de actividades
Flujos Alternativos	▪ **A1:** Sin eventos en el período 1. Filtros no devuelven resultados 2. App muestra mensaje “No hay eventos” 3. Usuario puede modificar criterios de búsqueda ▪ **A2:** Error de consulta 1. Fallo en acceso a base de datos 2. Sistema muestra error temporal 3. Usuario puede reintentar consulta

Cuadro 14: Caso de Uso CU-14: Consultar Logs del Sistema

4.2.15. Diagrama de casos de uso

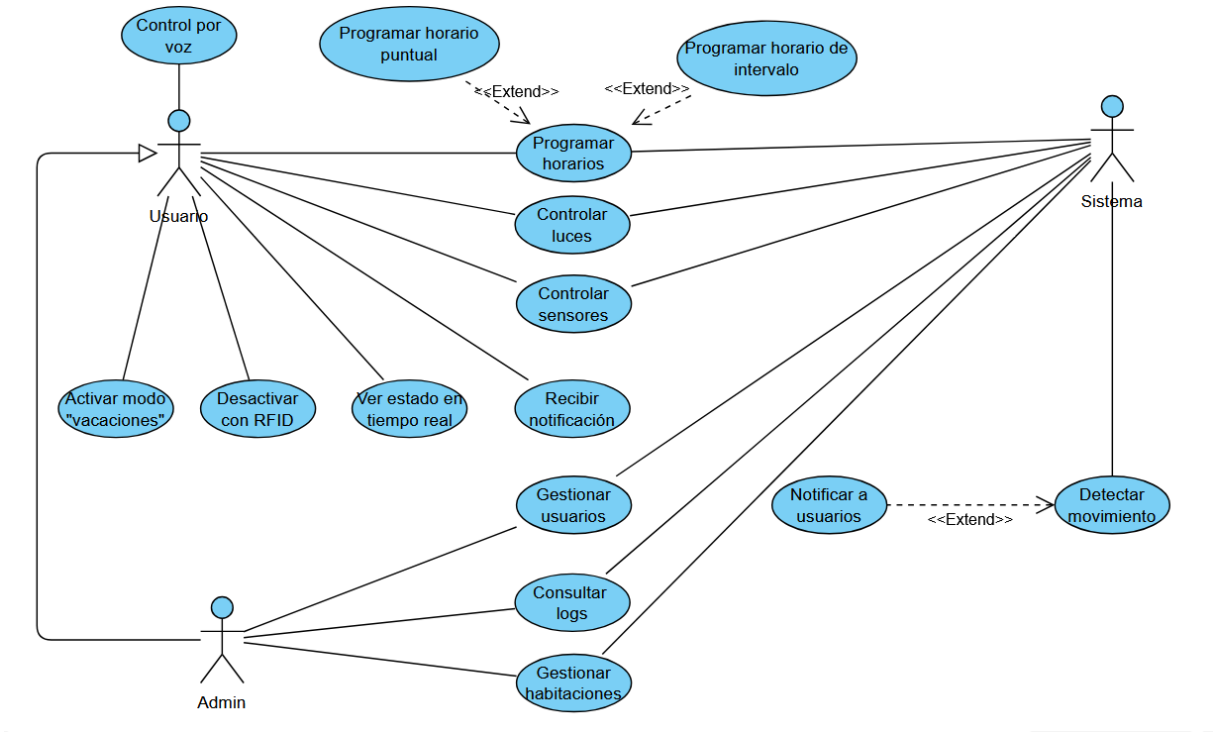


Figura 16: Diagrama de casos de uso

4.3. Diagramas de secuencia

A continuación vamos a representar algunas de las interacciones de los casos de usos anteriores en forma de diagramas de secuencia para identificar los actores y mensajes involucrados.

Controlar Iluminación

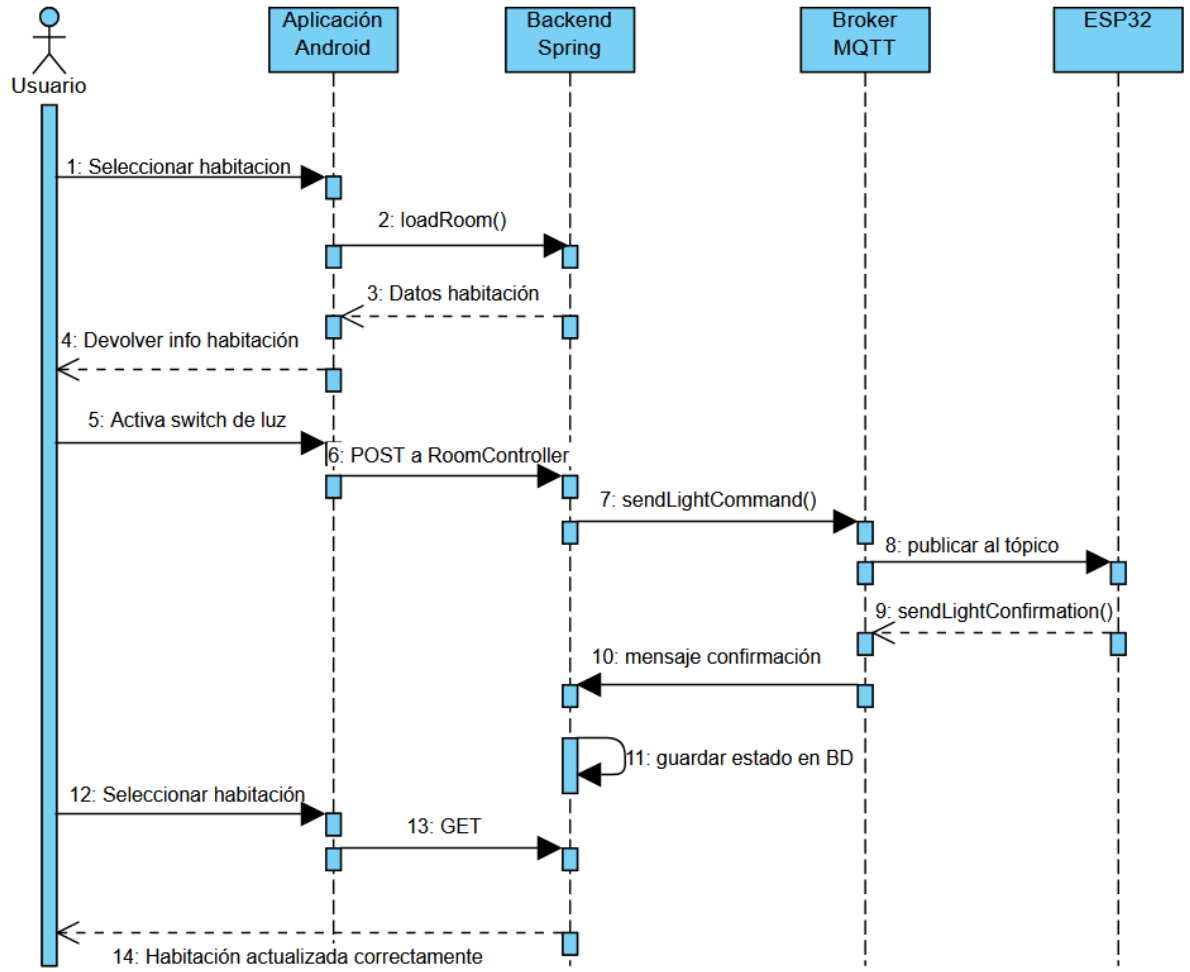


Figura 17: Diagrama de secuencia para el control de luces

- El usuario selecciona en la app una habitación para gestionar haciendo click en su tarjeta en la pantalla principal (1).
- La aplicación ejecuta un GET para la habitación seleccionada y el backend responde con datos sobre la habitación, que la aplicación muestra al usuario en la pantalla de la habitación. (2) (3) (4)

- El usuario acciona el switch de iluminación en la pantalla de detalles de la habitación. (5)
- La aplicación hace un POST al backend para cambiar el estado de la luz y este ejecuta el método para enviar el comando al dispositivo ESP32. (6) (7)
- El broker MQTT publica el comando para el dispositivo en el tópico de iluminación y el dispositivo toma la acción correspondiente. (8)
- Una vez ejecutada con éxito, el ESP32 envía confirmación mediante MQTT al backend para que este guarde el nuevo estado de la luz en la base de datos. (9) (10) (11)
- Si el usuario vuelve a seleccionar esta habitación o pulsa el botón de "Actualizar" recibirá la información correcta del backend. (12) (13) (14)

Detección de movimiento

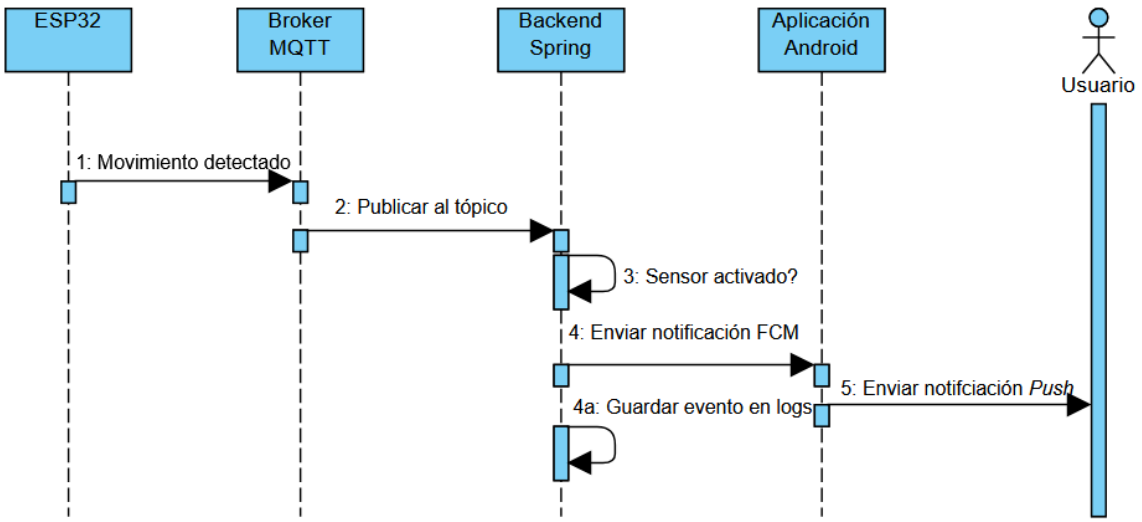


Figura 18: Diagrama de secuencia detección de movimiento

- Uno de los sensores PIR detecta movimiento en una habitación con los sensores marcados como "activos". (1) (2)
- El código del ESP32 envía mediante MQTT un mensaje al backend para que este tome acción. El backend comprueba que el sensor estaba activo antes de reaccionar. (3)
- El backend envía una notificación *push* al usuario para alertarle del movimiento y guarda el log de este evento en la base de datos. (4) (4a)
- La aplicación Android se encarga de mostrar la notificación *push* en el dispositivo del usuario con información sobre la sala dónde se detectó el movimiento y la hora. (5)

Registrar nueva tarjeta RFID

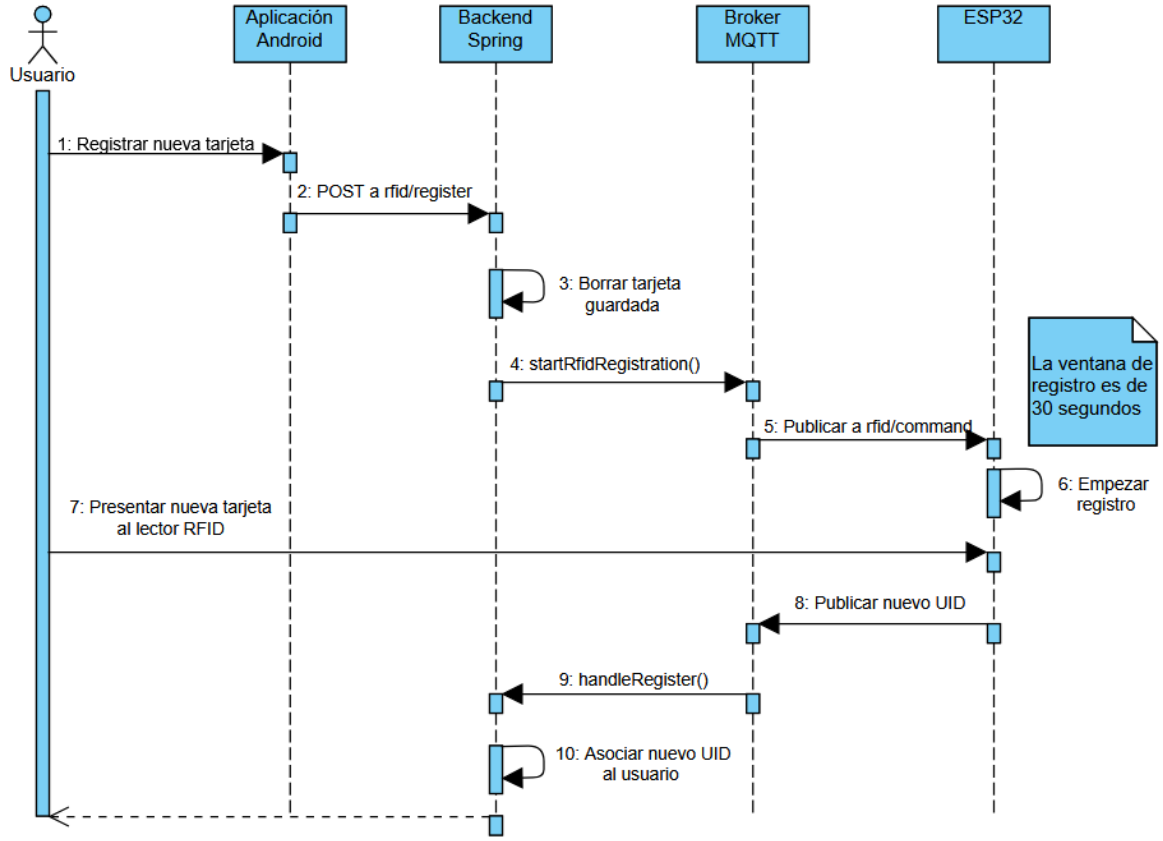


Figura 19: Diagrama de secuencia para el registro de una nueva tarjeta RFID

- El usuario desde el menú de la pantalla principal presiona el botón “Tarjeta RFID” y clickea para registrar una nueva. (1)
- La aplicación hace un POST al backend para iniciar el proceso de registro. (2)
- El backend borra la tarjeta anterior guardada para el usuario y utilizando MQTT, indica al dispositivo ESP32 que comience su proceso de registro. (3) (4) (5)
- El dispositivo activa el lector RFID, la primera tarjeta que se presente en un período de 30 segundos se considera la tarjeta a asociar al usuario. (6)
- El usuario presenta su nueva tarjeta RFID y el dispositivo pasa su nuevo UID por MQTT al backend para guardarla. (7) (8) (9)

- El backend recibe el UID y lo asocia en la base de datos con el usuario que inició el proceso de registro. (10)

5

Diseño del dispositivo IoT

Con los requisitos e interacciones de nuestro sistema definidos, es momento de diseñar y construir el dispositivo IoT que cubrirá las bases más importantes de las funciones principales. El cerebro pensante en el centro de este sistema es nuestro ESP32, que procesará todos los comandos enviados por el usuario y actuará en correspondencia. La placa recibirá y enviará mensajes al backend mediante MQTT a través de WiFi.

En nuestro circuito encontramos los siguientes componentes, que nos permitirán interactuar con el entorno como se estableció en los requisitos:

- **Sensor PIR HC-SR501:** se encarga de detectar movimiento en una sala. Solo enviará las detecciones cuando el sensor esté marcado como "ON" en la aplicación.
- **RFID-RC522:** este sensor RFID nos permite registrar nuevas tarjetas o utilizar las configuradas para desactivar las medidas de seguridad
- **LEDs:** simulan las luces del hogar, el usuario podrá apagarlos y encenderlos a voluntad.

5.1. MQTT

El protocolo MQTT es el pilar fundamental de la comunicación en nuestro proyecto de automatización y control IoT. Su propósito principal es proporcionar un canal eficiente, ligero y en tiempo real para el intercambio de mensajes entre el backend, nuestro dispositivo ESP32 y la aplicación móvil.

MQTT funciona bajo una arquitectura de publicador-suscriptor, donde todos los mensajes pasan a través de un intermediario central llamado broker. Los dispositivos o servicios que generan información (publicadores) envían mensajes a ciertos "tópicos", que corresponden con

cadenas jerárquicas que organizan y categorizan los mensajes. Los dispositivos o servicios interesados en recibir cierta información (suscriptores) se suscriben a esos tópicos específicos. El broker se encarga de distribuir los mensajes a todos los suscriptores de un tópico, sin tener que conocer su identidad. Esta estructura permite una comunicación desacoplada, flexible y escalable, ya que facilita la integración de nuevos dispositivos y la gestión eficiente de eventos y comandos en sistemas IoT como el nuestro. En el ESP32 podemos utilizar la librería "PubSubClient.h" para suscribir y publicar a tópicos de un broker desde el código.

En nuestro proyecto hemos aprovechado MQTT y hemos establecido una estructura clara de tópicos a los que enviar y por los que recibir mensajes sobre las distintas partes de nuestro sistema. Tenemos tres tópicos principales: rfid, <roomName>(el nombre de una habitación guardada en nuestro sistema) y REMOVE. En la figura 20 se muestra la estructura jerárquica de los tópicos MQTT.

- **<roomName>/** subtópico para controlar luces y sensores de cada habitación.
  - **mov/** subtópico para los sensores de movimiento. Se utiliza **command/** para encender o apagar el sensor, según el payload. **confirmation/** lo utiliza el ESP32 para informar al backend sobre la ejecución de una acción.
  - **lig/** subtópico para las luces del sistema. Se utiliza **command/** para encender o apagar la luz de una habitación, según el payload. **confirmation/** lo utiliza el ESP32 para informar al backend sobre la ejecución de una acción.
- **rfid/** subtópico para todas las comunicaciones sobre los sensores y tarjetas RFID.
  - **command/** utilizado para iniciar o cancelar el proceso de registro de nuevas tarjetas RFID.
  - **register/** enviado por el ESP32 para terminar el proceso de registro cuando detecta una tarjeta. Envía el UID a guardar en el payload.
  - **event/** el esp32 publica las detecciones de tarjetas RFID habituales a este tópico.
- **REMOVE/:** tópico para eliminar habitaciones, el nombre de la habitación a eliminar se pasa por el payload.

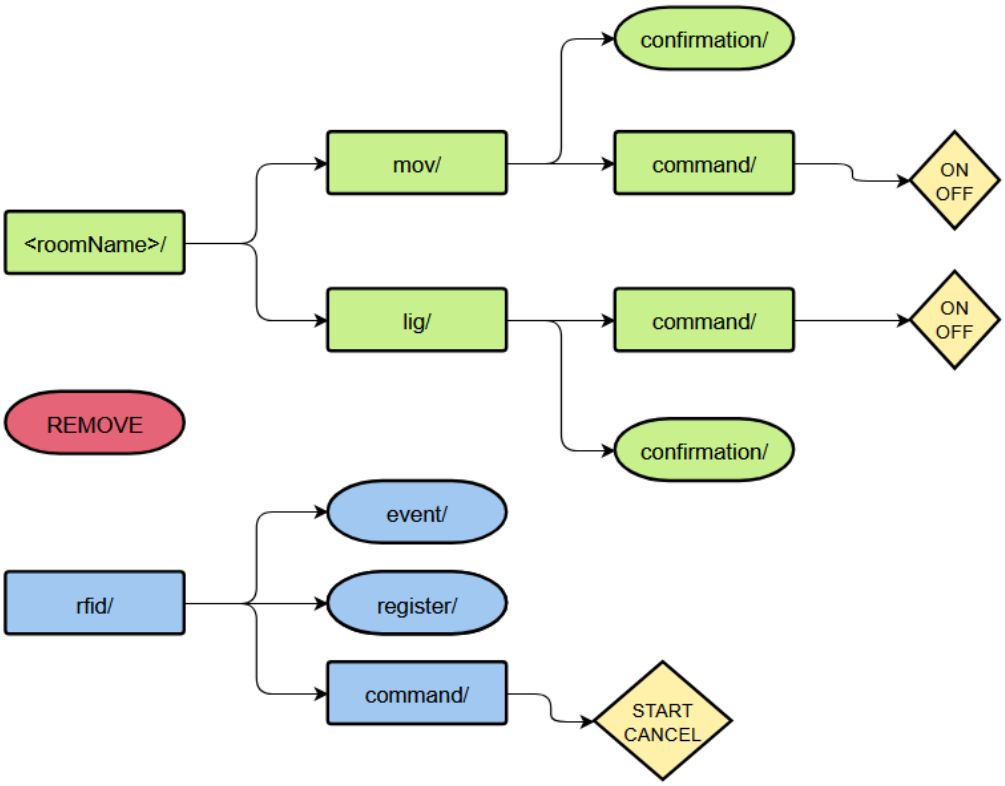


Figura 20: Estructura de tópicos MQTT

5.2. Diseño

En la sección 3 dimos una vista general del sistema en la figura 14. En esta sección vamos a detallar el diseño del dispositivo ESP32, las conexiones necesarias y el funcionamiento de los componentes. Para hacer el esquema de las conexiones visible en la figura 21, utilizamos la herramienta web Wokwi junto con los datasheets de cada componente, [6], [9] y [4].

En torno al ESP32 se conectan los diferentes sensores y actuadores que permiten la interacción con el entorno. Para la detección de presencia, se han incorporado dos sensores PIR, uno para la habitación del salón y otro para el cuarto, conectados a los pines GPIO 34 y 35 respectivamente. Estos sensores permiten detectar movimiento en ambas estancias y enviar eventos a través de sus pines y posteriormente al servidor cuando se produce una detección. Estos sensores están alimentados por la salida de 5v de nuestro ESP32.

El sistema también cuenta con dos indicadores luminosos, representados por LEDs de diferentes colores (amarillo para el salón y rojo para el cuarto), conectados a los pines GPIO 2 y 4. Estos LEDs permiten visualizar el estado de las luces en cada habitación, siendo controlados

directamente por el ESP32. Además, se ha añadido un buzzer conectado al pin GPIO 32, que puede se activará automáticamente cuando se detecte movimiento durante el modo vacaciones, para asustar a posibles intrusos. Se han empleado resistencias limitadoras en los LEDs para evitar daños por sobrecorriente.

Para la identificación de usuarios y desactivación del sistema de alarmas, se ha integrado un lector RFID MFRC522, conectado al ESP32 mediante la interfaz SPI. Los pines utilizados para esta comunicación son SDA (GPIO 5), RST (GPIO 22), SCK (GPIO 18), MOSI (GPIO 23) y MISO (GPIO 19). Este módulo permite registrar y detectar tarjetas RFID, facilitando la autenticación y el registro de eventos de acceso en el sistema. El lector requiere 3.3v de entrada que sacamos del pin 3v3 de nuestra placa.

Es importante remarcar que este diseño está hecho para la situación presupuesta de que tenemos una casa con dos habitaciones configuradas, *salon* y *cuarto*, cada una de ellas con luces y un sensor de movimiento. Cambiar los componentes en una habitación o añadir más conlleva retocar el diseño para adapatarlo. Aún así, el funcionamiento de las conexiones será el mismo en cuanto a los componentes individuales.

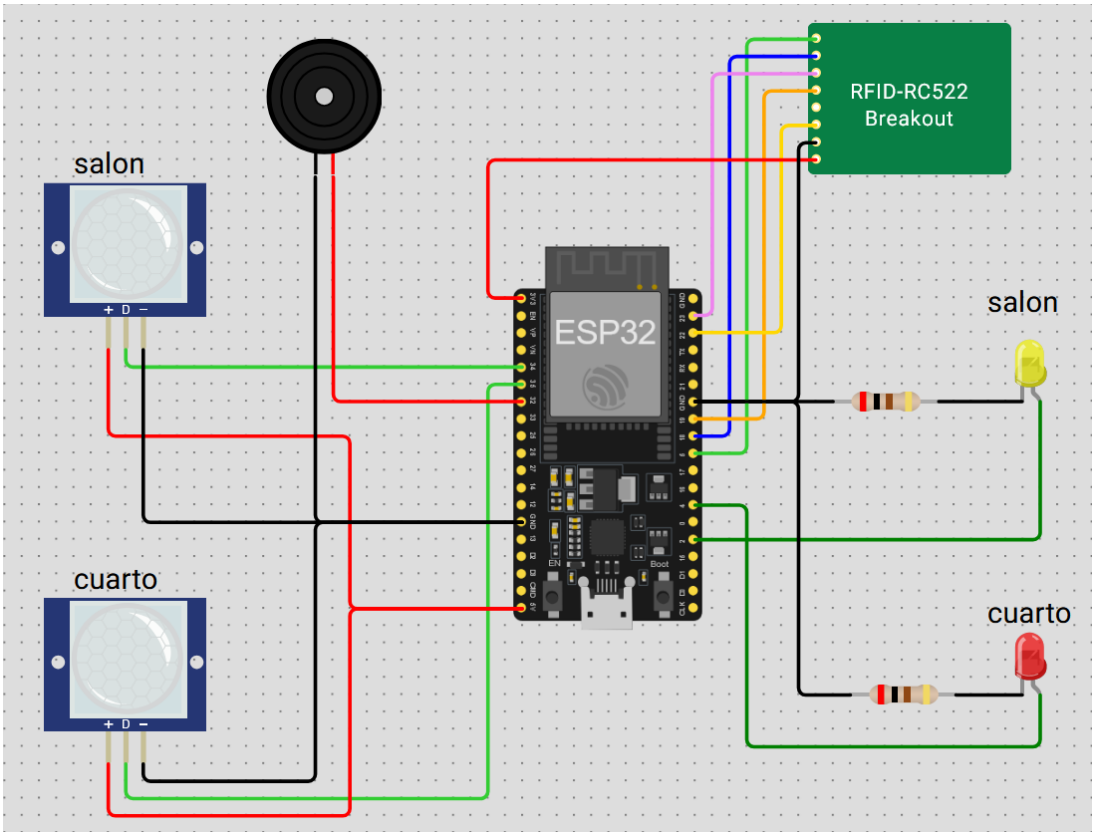


Figura 21: Esquemático del circuito principal

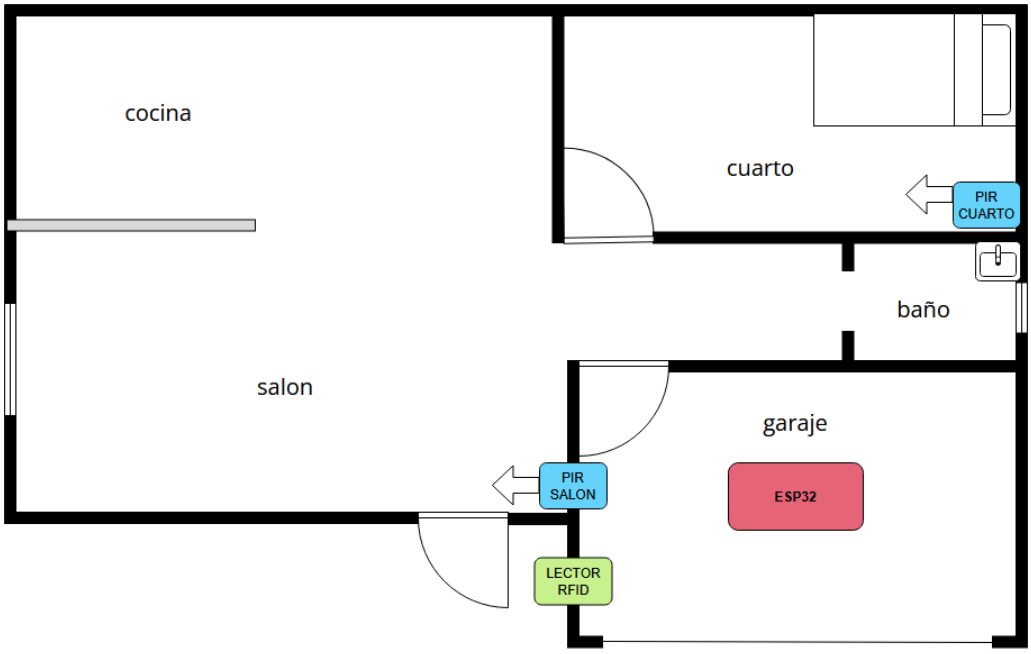


Figura 22: Plano de la casa

La casa de LEGO

Para crear un entorno en el que desplegar y utilizar nuestro dispositivo, hemos decidido construir una casa utilizando LEGO e instalar ahí los sensores de seguridad y luces junto con el dispositivo ESP32. Primero, esbozamos el plano de la casa (figura 22) para asegurarnos de la correcta distribución de las habitaciones.

En el plano tenemos la entrada principal abajo, a su derecha antes de entrar estará el lector RFID para desactivar las alarmas. Nada más entrar a mano derecha está el primer sensor de movimiento, este para proteger el salón. El sensor del cuarto está en la esquina inferior derecha de este. En el garaje colocaremos el dispositivo ESP32 y todo el cableado saldrá de ahí.

Con el plano hecho, podemos empezar a buscar las piezas necesarias y construir los muros principales de la casa (figura 23). Una vez comprobado que los componentes del sistema caben en su lugar pasamos a decorar la casa incluyendo cosas que veríamos en un hogar real como una cocina, sofá y demás (figura 24). Con esto, estamos listos para proteger este hogar remotamente.

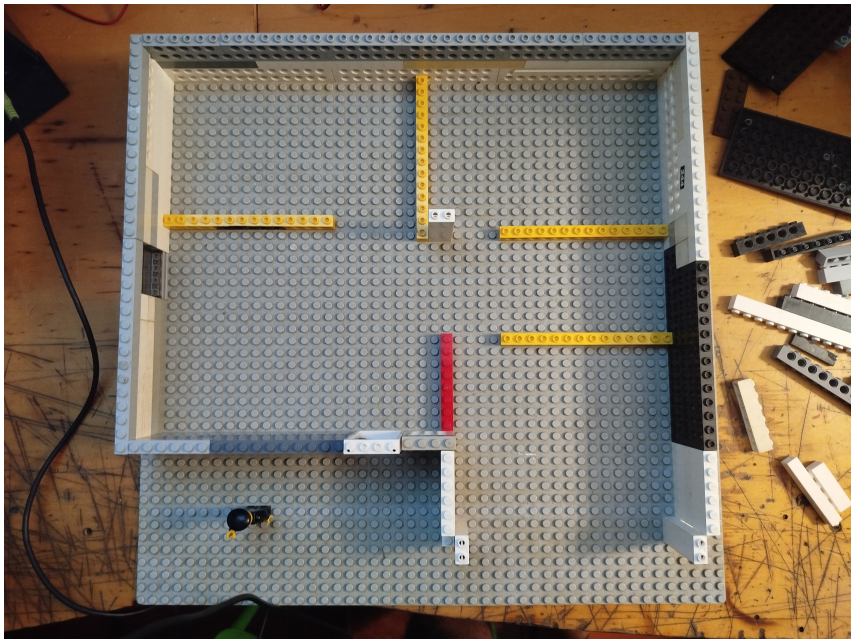


Figura 23: Cimientos de la casa

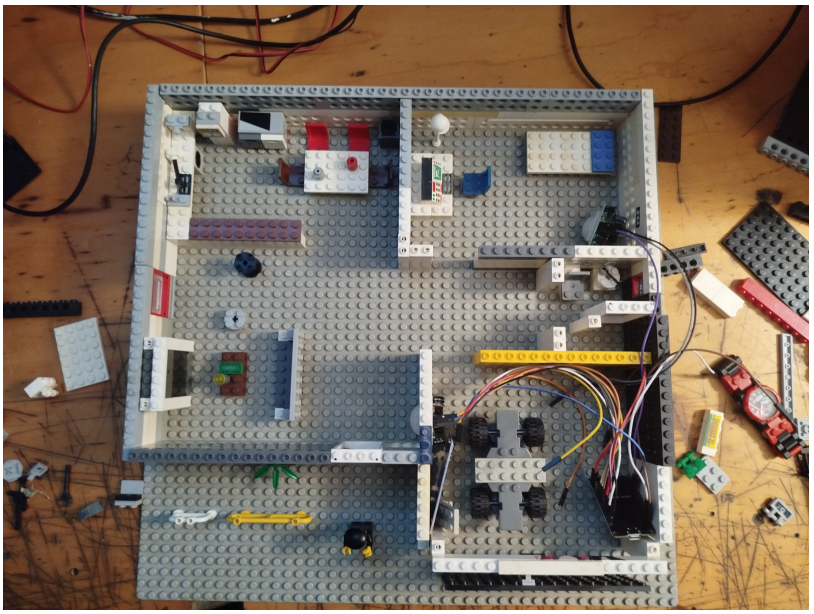


Figura 24: Casa de LEGO con el ESP32 instalado

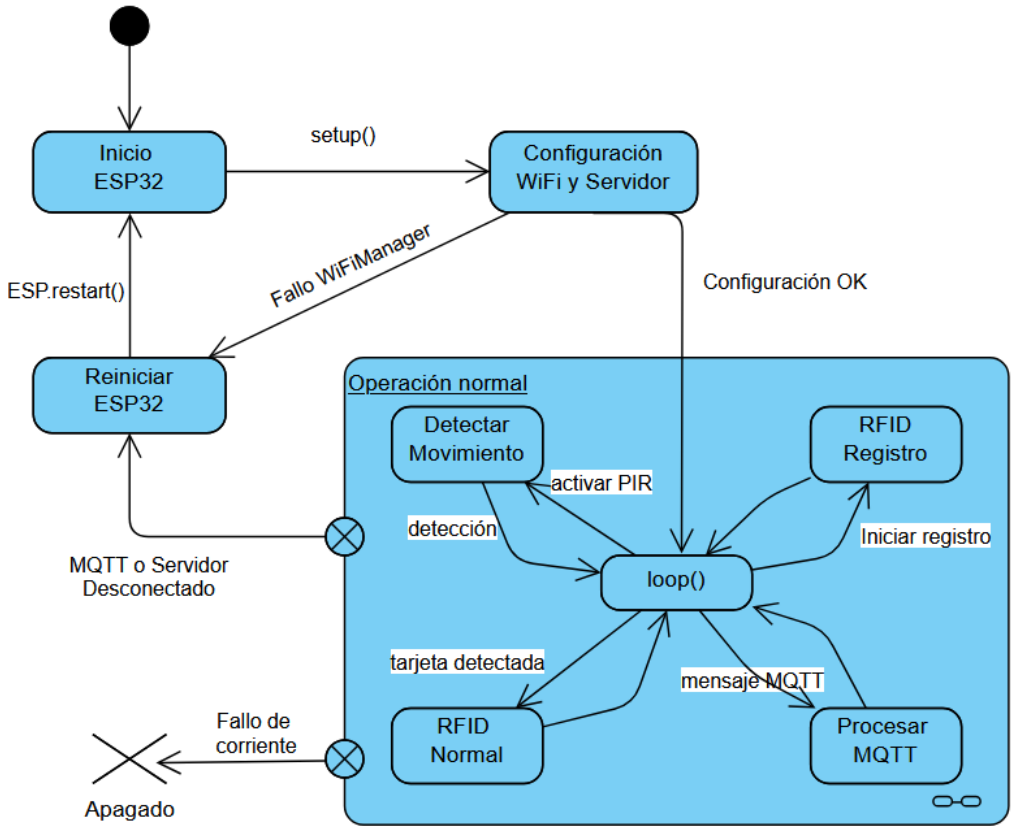


Figura 25: Diagrama de estados para el código del ESP32

5.3. Programación del Dispositivo IoT (ESP32)

La programación del ESP32 se ha realizado en C++ utilizando el entorno Arduino IDE, haciendo uso de librerías especializadas como *WiFiManager*, *PubSubClient*, *MFRC522* y *ArduinoJson* para facilitar la conectividad y la integración con el backend.

A diferencia de sistemas alimentados por batería que requieren modos de bajo consumo como el "deep sleep", en este proyecto el ESP32 debe permanecer activo de forma continua para garantizar la respuesta inmediata ante eventos de seguridad y control del hogar. Sin embargo, se ha optimizado el uso de recursos y la eficiencia mediante una arquitectura basada en eventos y el uso de interrupciones para los sensores de movimiento y el lector RFID.

El flujo de funcionamiento, representado en la figura 25 con un diagrama de estados, es el siguiente:

1. **Inicialización y Configuración**

Al arrancar, el ESP32 ejecuta la rutina *setup()*, donde inicializa los pines de los sensores, relés y periféricos, y configura la conexión WiFi. Si es la primera vez que se enciende o si se ha reseteado la configuración, el dispositivo crea una red WiFi propia ("CasaIoT-Setup") usando WiFiManager, permitiendo al usuario introducir las credenciales de su red y los parámetros del broker MQTT a través de un portal cautivo creado por el propio ESP32.

2. **Conexión a la Red y al Broker MQTT**
   Una vez conectado a la red WiFi, el ESP32 establece la conexión con el broker MQTT, cuyos parámetros pueden ser configurados dinámicamente y se almacenan en la memoria flash para futuros reinicios. El dispositivo se suscribe a los tópicos relevantes para recibir comandos de control desde el backend a través de MQTT.

3. **Bucle Principal y Gestión de Eventos**
   En el bucle principal (*loop*), figura 26, el ESP32 mantiene la conexión MQTT y gestiona los eventos de hardware. El método *onMqttMessage()* se ejecuta automáticamente cada vez que se publica un nuevo mensaje a un tópico al que nos hemos suscrito, lo que nos permite tomar acción correspondiente. Hemos dividido el código en distintos bloques que manejan cada parte del hardware:

   - **Sensores de Movimiento:** Con el método del loop *checkMovementSensors* se monitorizan los sensores PIR de cada habitación. Si se detecta movimiento y la alarma para esa habitación está activada, se envía un mensaje MQTT al backend notificando el evento.
   - **Control de Luces y sensores:** El dispositivo recibe comandos MQTT para cambiar el estado de luces y sensores, ejecuta la acción correspondiente y responde con una confirmación al backend.
   - **Lector RFID:** Al recibir un mensaje sobre RFID, el código comprueba si el proceso de registro de nuevas tarjetas está activado, en ese caso envía el UID detectado a el backend por el tópico */register*. Si está desactivado, enviará al backend el UID como un evento.

```
void loop() {
  if (!client.connected()) reconnectMqtt();
  client.loop();

  // Sensores de movimiento, si no hay encendido nos ahorramos
  if (salonAlarmState || cuartoAlarmState){
    checkMovementSensors();
  }

  // RFID
  handleRfidLogic();

  delay(100);
}

void onMqttMessage(char* topic, byte* payload, unsigned int length) {···
}
```

Figura 26: Bucle principal del código

Todos los parámetros de configuración (WiFi, MQTT, client ID) se almacenan en la memoria flash del ESP32 usando la librería *Preferences*, permitiendo que el dispositivo recupere su configuración tras un reinicio o corte de energía. Además, el sistema implementa reconexión automática tanto a la red WiFi como al broker MQTT en caso de desconexión.

Aunque el código está preparado para añadir nuevas habitaciones, sensores o funcionalidades ampliando las secciones correspondientes, cada cambio conlleva editar el código a mano y rehacer las conexiones. Más adelante podríamos idear una manera de adaptar y generar el código según las habitaciones configuradas en el backend.

6

Backend Spring

En esta sección de la memoria vamos a cubrir todo lo relacionado al diseño y desarrollo del servidor central de nuestro sistema de iluminación y seguridad. Nuestro ESP32 es capaz de recoger información muy útil sobre su entorno y adaptarse a este por su cuenta, pero necesita un sistema que lo controle y le otorgue propósito. Ahí entra nuestro backend como núcleo lógico y funcional de la aplicación. Este componente se encarga de gestionar todas las interacciones con los dispositivos IoT, la base de datos, los usuarios y la aplicación Android.

En el servidor se concentra toda la lógica pensante del proyecto, el orden en el que se van a llevar a cabo las acciones, los objetos modelados en el sistema, la comunicación en tiempo real y la exposición de endpoints REST para manejar todas estas funciones remotamente desde la aplicación.

En nuestro desarrollo siguiendo metodología en cascada, podemos hacernos una buena idea, gracias a los requisitos establecidos y el modelo de dominio, de las entidades, clases, endpoints y arquitectura general que va a seguir el servidor. El orden de ejecución de las tareas también está ya establecido de acuerdo con los flujos definidos en los casos de uso.

6.1. Inicialización con Spring

Elegimos Spring como framework en nuestro proyecto por varios motivos. El principal atractivo de Spring viene del concepto de "Inversión de control" [3], un principio de diseño de software que promueve la delegación de creación de objetos y gestión de dependencias a un componente externo, como sería Spring. Con este flujo de control establecido, podemos centrarnos en añadir funcionalidades al servidor sin perder tiempo intentando hacerla más modular o flexible. Spring, y en concreto Spring Boot, es uno de los frameworks más utilizados para la creación de aplicaciones Java, lo que se traduce en una amplia librería de documentación y componentes pre-existentes disponibles en internet para aprovechar.

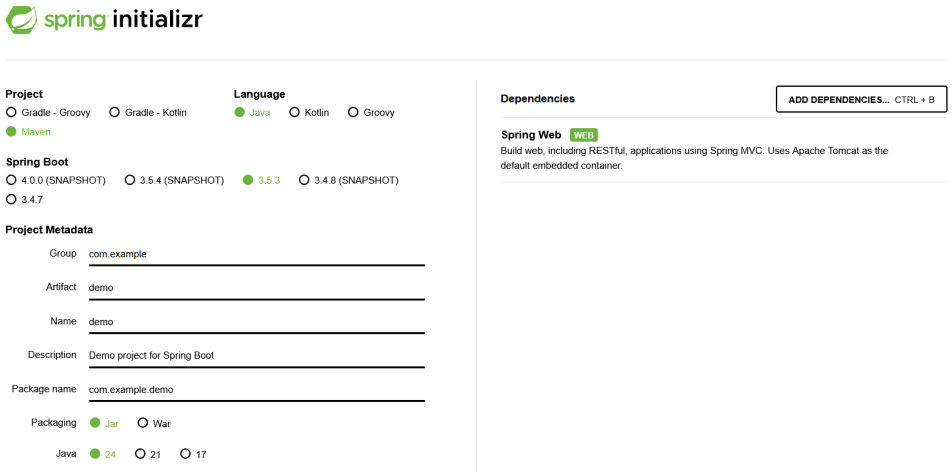


Figura 27: Servicio Web de Spring Initializr

Spring Boot

Spring boot simplifica aún más las cosas, proporciona configuración automática de componentes, un servidor web embebido (Apache Tomcat en nuestro caso) y el gestor de dependencias compatible con Maven para creación de ejecutables (.jar).

Para la creación del proyecto inicial, Spring ofrece una herramienta llamada Spring Initializr, que podemos usar desde la web o como una extensión en Visual Studio Code, que se ocupa de generar una estructura básica del entorno de desarrollo, desde el archivo de dependencias hasta la organización por paquetes según lo configuremos en el Initializr, como se ve en la figura 27. Lo primero que debemos seleccionar es el gestor de proyecto que queremos usar, en nuestro caso Maven. Después, seleccionamos el lenguaje de programación principal y la versión de Spring que queremos utilizar. Una vez personalizados los campos para nombre, descripción y extensión de archivo ejecutable a usar, podemos incluir algunas dependencias iniciales para nuestro proyecto. En la figura aparece incluido *Spring Web*, pero no hace falta incluirlo desde el principio, siempre podemos añadir nuevos componentes y dependencias más tarde. Tras pulsar generate, Spring creará nuestro proyecto inicial y lo descargará en formato zip para utilizarlo en el IDE de nuestro gusto.

Maven

Como ya hemos mencionado, Maven se encarga de resolver y descargar las dependencias especificadas en el archivo *pom.xml* de nuestro proyecto Spring. Concretamente descargará

e injectará las dependencias cuando se inicie un "ciclo de vida" que las requiera, como por ejemplo al compilar (mvn compile), iniciar tests (mvn test) o instalar (mvn install).

REST

La arquitectura REST nos permite conectar partes de un sistema mediante simples peticiones del protocolo HTTP. En este esquema de diseño, cada recurso o componente del sistema se expone a los usuarios o clientes mediante una URL única y se manipula utilizando los verbos de HTTP, estos son **GET** para recibir datos, **POST** para publicar datos, **PUT** para actualizar y **DELETE** para eliminar.

Con REST, todos los objetos se manipulan mediante URIs que contienen toda la información necesaria sobre la petición, facilitando la comprensión del código y manteniendo independencia entre cliente y servidor, por lo que podemos desarrollar nuestro backend sin tener que pensar demasiado en el frontend por ahora. Además, HTTP es un protocolo con más de 30 años de uso en páginas web, lo que nos da una base firme sobre la que hacer nuestro sistema.

6.2. Estructura del backend

A continuación, vamos a establecer la estructura de nuestro backend para el sistema de seguridad e iluminación. Como se mencionó en la sección de tecnologías utilizadas, vamos a seguir el patrón de diseño *Model-View-Controller* de la figura 28 que divide los aspectos de nuestra aplicación en tres partes interconectadas.

La *View* hace referencia a la presentación de los datos al usuario y el manejo de inputs de este, ese apartado lo cubriremos con la aplicación Android más adelante. El *Model* va a ser la representación de los datos que tratará la aplicación y interactuará con una base de datos para ocuparse de el almacenamiento, recuperación y modificación de los objetos que necesite nuestro sistema. Por último, en *Controller* se encuentra el código que hace de intermediario entre los dos anteriores, procesando los inputs que le pasa la *View*, entregándoselos al *Model* para que procese los datos y devolviendo la *View* oportuna al usuario según la acción que esté llevando a cabo. En nuestra implementación concreta, hemos además dividido el *Model* en dos partes, una que representa los objetos y repositorios para la base de datos, y otra llamada *Service*, que contiene toda la lógica de negocio y los métodos necesarios para trabajar con los

datos. Esta división hace el código más legible y establece claramente los roles de cada tipo de clase.

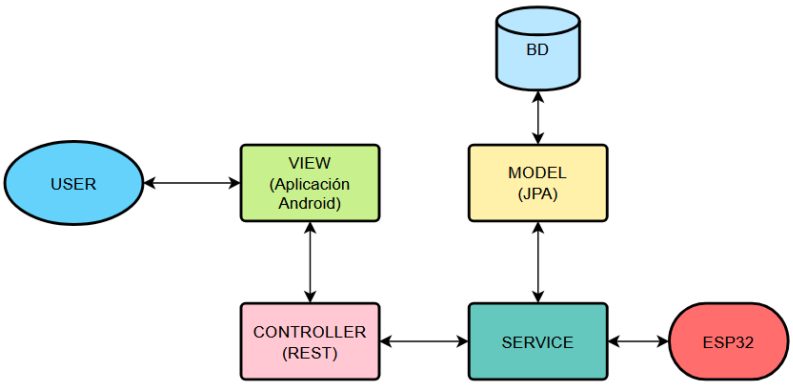


Figura 28: Patrón MVC modificado para nuestro sistema

6.3. Base de datos

Empezaremos nuestro desarrollo del backend definiendo los objetos y tablas que va a tener nuestra base de datos. Para facilitar el proceso, vamos a utilizar JPA (Java Persistence API), una especificación Java con interfaces y anotaciones que nos permiten gestionar los objetos de la base de datos desde código Java. JPA mapea las clases java que anotemos con tablas en la base de datos que conectemos. Podemos definir entidades, declarar claves primarias o foráneas, generar IDs automáticos, y más.

En Spring, JPA se integra perfectamente con Spring Data JPA, que proporciona repositorios que eliminan aún más código repetitivo, permitiéndonos definir métodos de consulta simplemente declarando sus firmas en interfaces.

Antes de empezar a crear entidades y tablas, debemos de tener instalado y corriendo *MySQL Server*, después desde MySQL Workbench podremos conectarnos a nuestra instancia local y crear los usuarios que queramos y más importante la base de datos que conectaremos al servidor Spring.

```
CREATE DATABASE iotdb;
```

Una vez creada la base de datos y configurada la instancia de *MySQL Server*, podemos ir a el archivo *application.properties* de nuestro proyecto Spring e introducir ahí los detalles de nuestra conexión con la base de datos. A partir de aquí cualquier entidad que creemos a partir

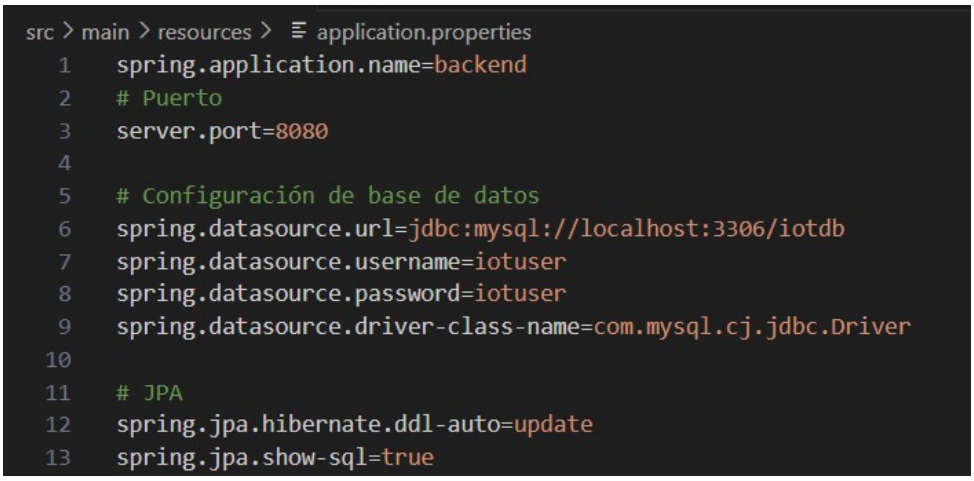


Figura 29: Conexión a la base de datos *IotDB* configurada en las propiedades de nuestra aplicación Spring

de una clase Java será guardada como una tabla directamente en la base de datos "IotDB". Figura 29

Con el método de creación establecido, podemos pasar a especificar las tablas de nuestro sistema. La primera y fundamental en el sistema es **Room**, en esta tabla se almacenan los datos de las habitaciones de nuestro hogar. Se utiliza el nombre de la habitación como clave primaria en la base de datos y tiene dos atributos booleanos, ambos haciendo referencia al estado de las luces y sensores de movimiento en la habitación.

La entidad **User** modela los usuarios que van a utilizar el sistema. Como clave primaria utilizamos un identificador único auto-generado con las anotaciones de JPA. Se distinguen dos tipos de usuarios según el rol que tengan: ADMIN y USER. Los administradores tienen permisos adicionales y acceso a funciones de gestión en la aplicación móvil. Además, se incluye una columna para guardar el UID de una tarjeta RFID del usuario.

RoomSchedule representa los horarios contenidos en cada una de las habitaciones. Está relacionada con **Room** mediante una clave foránea y puede manejar luces o sensores de movimiento. El sistema soporta dos tipos de programación: puntual (ejecutar una acción a una hora específica) e intervalo (mantener un estado durante un período de tiempo). Los horarios puntuales utilizan el campo *time*, mientras que los de intervalo usan *startTime* y *endTime*. Cada horario tiene un nombre opcional y un estado que define la acción a realizar.

Por último, la entidad **Event** nos sirve para registrar todos los eventos que registra nuestro

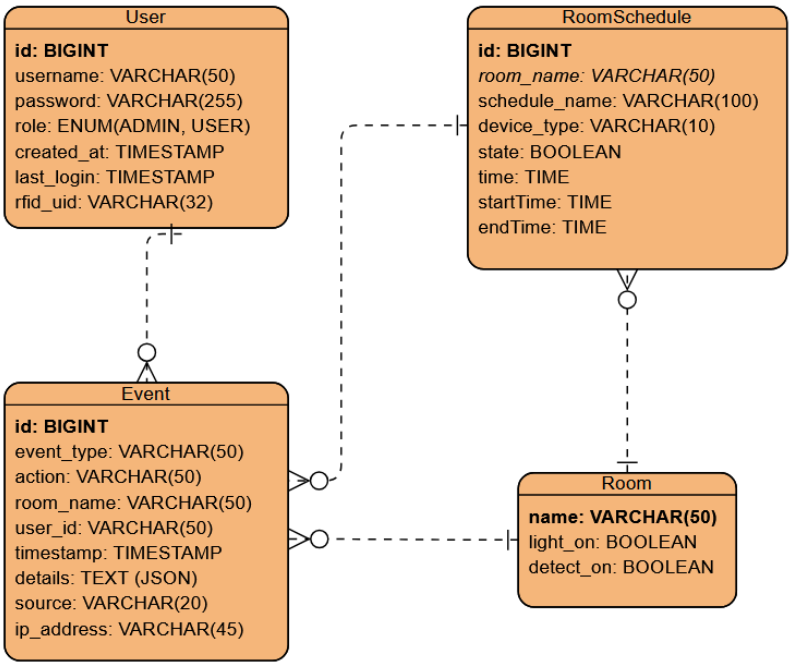


Figura 30: Diagrama Entidad/Relación de nuestra base de datos

sistema, clasificándolos según el tipo. Los tipos existentes son los siguientes: USER_ACTION (para acciones ejecutadas desde la app), SYSTEM_ACTION (para acciones del propio backend), MOVEMENT_DETECTED, SCHEDULE_EXECUTED y LOGIN_ATTEMPT. En cada objeto **Event** se almacena información adicional sobre el evento en formato JSON, como por ejemplo qué usuario concreto lanzo el evento, cuándo se lanzo y en que habitación. Para optimizar las consultas frecuentes, la tabla incluye índices en timestamp, room_name, event_type y user_id.

Podemos utilizar un diagrama Entidad-Relación para representar estos objetos, figura 30.

6.4. Broker MQTT

El broker MQTT es el componente encargado de gestionar la comunicación en tiempo real entre el backend y los dispositivos IoT, como los ESP32. Se encarga de hacer llegar los mensajes adecuados a los clientes según los tópicos a los que estén suscritos. En nuestro sistema, el broker utilizado es Mosquitto[5], un servidor MQTT ligero y ampliamente adoptado en entornos IoT.

Archivos de configuración MQTT

Spring ofrece soporte para canales de llegada y salida MQTT si utilizamos la dependencia de *spring-integration-mqtt* [15]. La integración del broker MQTT en el backend se realiza principalmente a través de los siguientes archivos y clases:

- **MqttProperties**: clase de configuración que encapsula las propiedades del broker (host, puerto, clientId, autoStart) y permite su personalización desde *application.properties*.
- **MosquittoAutoStart**: componente que detecta la IP local e inicia automáticamente el proceso Mosquitto al arrancar el backend. Verifica si el broker está corriendo y lo lanza como servicio o ejecutable según el sistema operativo.
- **MqttConfig**: clase con los beans de Spring Integration para la conexión MQTT. Define los canales de entrada y salida, el cliente MQTT y los adaptadores para recibir y enviar mensajes.
- **MqttGateway:** clase que encapsula el envío de mensajes MQTT desde el backend a los dispositivos IoT.
- **MqttEventHandler:** clase que recibe y procesa los mensajes entrantes desde el broker MQTT, distribuyéndolos a los servicios correspondientes (por ejemplo, encender luces, procesar sensores, manejar eventos RFID) según tópicos.

Funcionamiento del broker en el sistema

Al iniciar el backend, el sistema detecta la IP local y configura el broker MQTT para que el dispositivo IoT pueda conectarse correctamente. Si el broker Mosquitto no está ejecutándose y la opción "autoStart" está habilitada, el backend lo inicia automáticamente, asegurando que la comunicación esté disponible sin intervención por parte del usuario. Una vez lanzado correctamente, los datos del broker se imprimen en la consola del servidor para poder configurar nuestro dispositivo.

El backend utiliza Spring Integration para gestionar los canales de comunicación MQTT. Los mensajes enviados por los dispositivos (por ejemplo, una detección de los sensores de movimiento) llegan al broker y son procesados por el *MqttEventHandler*, que los distribuye

a los servicios correspondientes. De igual forma, el backend puede publicar mensajes en los topics MQTT para controlar dispositivos o enviar notificaciones.

Esta arquitectura permite una comunicación bidireccional, eficiente y desacoplada entre el backend y los dispositivos IoT, facilitando la escalabilidad y la instalación del sistema en distintas ubicaciones.

6.5. Controladores REST

En esta sección vamos a explicar los distintos controladores REST creados para el sistema, su propósito y que Endpoints exponen a usuarios para uso. Cada controlador recibirá peticiones HTTP y llamará a los servicios a tomar ciertas acciones.

RoomController

Este controlador gestiona las habitaciones que vayamos añadiendo a nuestro sistema de seguridad e iluminación. Desde este código se manejan todas las peticiones con respecto a los dispositivos presentes en una habitación. En el constructor mantenemos referencia a las clases de Service, pues haremos llamadas a sus métodos en los endpoints:

- **GET /rooms:** devuelve un objeto List<T> con todas las habitaciones presentes en la base de datos.
- **GET /rooms/{roomName}:** obtiene el objeto de una habitación concreta.
- **POST /rooms/{roomName}/remove:** elimina una habitación del sistema.
- **POST /rooms/{roomName}/alarm:** permite activar o desactivar las notificaciones de movimiento de una habitación específica, según que le pasemos en los parámetros.
- **POST /rooms/{roomName}/light:** permite encender o apagar las luces de una habitación específica, según que le pasemos en los parámetros.
- **POST /rooms:** permite crear una nueva habitación.

RoomScheduleController

Este controlador gestiona los horarios programados por el usuario para cada habitación de nuestro sistema de seguridad e iluminación. Permite a los usuarios crear horarios de dos tipos: puntuales, que ejecutan una acción a una hora concreta, o de intervalo, que mantienen el estado de un dispositivo durante un intervalo de horas.

- **POST /rooms/{roomName}/schedules:** este endpoint sirve para crear un nuevo horario en una habitación específica. Es obligatorio concretar en los parámetros el dispositivo a controlar y el estado. Según el tipo de horario será necesario especificar la hora o intervalo y se puede enviar también un nombre personalizado para este.
- **GET /rooms/{roomName}/schedules:** devuelve los horarios creados para una habitación específica en una List<T>.
- **DELETE /rooms/{roomName}/schedules/{id}:** elimina un horario concreto del sistema.

VacationModeController

Este controlador expone a los usuarios la funcionalidad del "modo vacaciones", durante el que se inhabilitan el resto de funciones del sistema mientras que este simula la presencia de personas dentro de la casa cuando estén fuera.

- **POST /vacation-mode/activate:** activa el modo vacaciones y devuelve un mensaje de confirmación.
- **POST /vacation-mode/deactivate:** desactiva el modo vacaciones y devuelve un mensaje de confirmación.
- **GET /vacation-mode/status:** consulta al sistema si el modo vacaciones está activado, devuelve la respuesta en un JSON con "true" o "false".

AuthController

Este controlador gestiona la autenticación y autorización de usuarios en el sistema. Proporciona los endpoints necesarios para el login, registro de nuevos usuarios y validación de

permisos administrativos. Mantiene registro de todos los intentos de autenticación mediante el EventLogService para fines de seguridad y auditoría. Además incluye los endpoints relacionado con la funcionalidad RFID.

- **POST /auth/login:** autentica a un usuario mediante username y password. Registra tanto intentos exitosos como fallidos incluyendo la dirección IP y User-Agent.
- **POST /auth/register:** permite el registro de nuevos usuarios en el sistema. Acepta parámetros de username, password y rol (por defecto USER).
- **GET /auth/validate/username:** valida si un usuario específico tiene permisos de administrador. Devuelve información sobre el username y su estado administrativo en formato JSON.
- **POST /auth/delete-user:** permite a los administradores eliminar usuarios del sistema. Requiere el username del usuario a eliminar e incluye validaciones de seguridad para prevenir la eliminación del usuario administrador principal.
- **POST /auth/rfid/register:** inicia el proceso de registro de una nueva tarjeta RFID para un usuario. El backend envía la orden al dispositivo IoT para activar el lector y espera la lectura de una tarjeta.
- **POST /auth/rfid/cancel:** cancela el proceso de registro de tarjeta RFID en curso para el usuario indicado, notificando tanto al backend como al dispositivo IoT.
- **GET /auth/rfid/{username}:** consulta el UID de la tarjeta RFID actualmente asociada a un usuario. Devuelve el identificador si existe o un mensaje de error si no hay tarjeta registrada.

EventController

Este controlador proporciona acceso de consulta al sistema de logging y auditoría. Permite a los administradores revisar el historial de eventos. Todas las consultas soportan paginación para mejorar el rendimiento.

- **GET /events/recent:** obtiene los eventos más recientes del sistema. Permite especificar el número de horas hacia atrás a consultar (por defecto 24 horas).

- **GET /events:** consulta eventos en un rango de fechas específico. Si no se especifican fechas, consulta los últimos 7 días.
- **GET /events/room/roomName:** obtiene todos los eventos relacionados con una habitación específica en un rango de fechas determinado.
- **GET /events/user/userId:** consulta todos los eventos generados por un usuario específico.
- **GET /events/failed-logins/username:** obtiene los intentos de login fallidos para un usuario específico en las últimas horas especificadas, importante para detectar posibles ataques.
- **POST /events/cleanup:** endpoint informativo sobre el proceso de limpieza automática de eventos antiguos que se ejecuta diariamente.

MqttController

Este controlador permite enviar mensajes MQTT al broker, facilitando la integración y pruebas desde clientes HTTP. Es útil para enviar comandos o notificaciones al dispositivo IoT directamente desde el backend.

- **POST /sendMessage:** recibe un JSON con los campos "message" y "topic" y publica el mensaje en el tópico MQTT indicado.

SimulationController

Este controlador permite simular eventos y respuestas de los dispositivos IoT en el sistema, facilitando así las pruebas y el desarrollo sin necesidad de hardware físico. A través de sus endpoints, hemos podido probar lanzar manualmente eventos como detección de movimiento, cambios de estado en alarmas y luces, así como simular respuestas y errores de los dispositivos.

- **POST /simulation/movement-detected/roomName:** simula la detección de movimiento en una habitación concreta, enviando el evento correspondiente como si lo hubiera detectado un sensor real.

- **POST /simulation/device-response/alarm/roomName:** simula la respuesta de confirmación de un dispositivo al cambiar el estado de la alarma, permitiendo verificar la actualización de estado en el backend.

- **POST /simulation/device-response/light/roomName:** simula la respuesta de confirmación de un dispositivo al cambiar el estado de la luz, comprobando que el backend procese correctamente la confirmación.

- **POST /simulation/device-error/roomName:** simula un error en un dispositivo (ya sea alarma o luz), enviando un mensaje de error al backend para probar la gestión de fallos.

6.6. Archivos de configuración

SecurityConfig

El archivo *SecurityConfig* configura la seguridad de la aplicación Spring Boot. Define las reglas de acceso a los endpoints, los filtros de autenticación y autorización, y nos permite definir un encriptador de contraseñas para nuestro sistema.

AsyncConfig

El archivo *AsyncConfig* habilita la ejecución asíncrona de tareas en el backend. Permite que ciertos servicios, como el logging de eventos o el envío de notificaciones, se ejecuten en segundo plano sin bloquear el hilo principal de la petición HTTP. Esto mejora el rendimiento y la escalabilidad del sistema.

6.7. Servicios

La capa de servicios (Service) agrupa la lógica de negocio de nuestra aplicación, actuando como puente entre los controladores REST y las operaciones sobre la base de datos o el sistema de mensajería MQTT. Cada clase de servicio encapsula un conjunto de responsabilidades claramente delimitadas, lo que facilita el mantenimiento, la prueba unitaria y la extensión futura de funcionalidades.

RoomService

- Gestiona la creación, recuperación y eliminación de **Room** mediante **RoomRepository**.
- Se integra con el **MqttGateway** para notificar al sistema IoT sobre cambios en las habitaciones (por ejemplo, eliminación).

LightService, MovementService y SoundService

- Encapsulan el protocolo de mensajería MQTT: formatean comandos JSON y utilizan **MqttGateway** para publicar en los topics correspondientes (**lig/command**, **mov/command**, etc.).
- Procesan confirmaciones y eventos entrantes, actualizando el estado de las entidades **Room** en la base de datos.
- **SoundService** está preparado para gestionar futuramente el subsistema de audio.

RoomScheduleService y RoomScheduleExecutor

- **RoomScheduleService**: realiza operaciones CRUD sobre **RoomSchedule** a través de **RoomScheduleRepository** y proporciona métodos de consulta por hora puntual e intervalos.
- **RoomScheduleExecutor**: anotado con **@Scheduled**, consulta periódicamente los horarios almacenados y ejecuta las acciones de encendido/apagado de luces o alarmas. Además al terminar un horario de intervalo apagará el sensor o luz afectado.

MqttGateway & MqttEventHandler

- **MqttGateway**: interfaz anotada con **@MessagingGateway** que publica mensajes en el channel de salida configurado.
- **MqttEventHandler**: componente que recibe mensajes entrantes de MQTT, distribuye eventos y confirmaciones a los servicios de luz, movimiento y sonido.

VacationModeService

- Controla el estado del "modo vacaciones" mediante un **AtomicBoolean**.
- Al activarse, envía comandos de activación de sensores de movimiento en todas las habitaciones.
- Mediante un método anotado con **@Scheduled(fixedRate = 60000)**,que cada 60 segundos simula actividad aleatoria encendiendo y apagando luces para disuadir intrusos.

AuthService

- Gestiona la autenticación y autorización de usuarios mediante operaciones sobre **UserRepository**.
- Implementa validación de credenciales con hash de contraseñas para garantizar la seguridad.
- Registra todos los intentos de login (exitosos y fallidos) a través de **EventLogService** incluyendo la dirección IP y User-Agent del dispositivo.
- Proporciona métodos para validar permisos administrativos y crear nuevos usuarios con roles específicos.

EventLogService

- Centraliza el sistema de auditoría y logging de la aplicación mediante **EventRepository**.
- Registra automáticamente eventos críticos del sistema: autenticación, cambios de estado de dispositivos, activación de alarmas y acciones administrativas.
- Proporciona consultas avanzadas con paginación para análisis de actividad por usuario, habitación o rango temporal.
- Incluye funcionalidad de limpieza automática programada con **@Scheduled**.

NotificationService

- Gestiona el envío de notificaciones PUSH a usuarios.
- Se integra con **MqttEventHandler** para procesar eventos de los sensores de movimiento y generar alertas en tiempo real.
- Mantiene registro de notificaciones enviadas a través de **EventLogService**.

RFIDService

- Gestiona toda la lógica relacionada con el registro, detección y asociación de tarjetas RFID a usuarios del sistema.
- Permite iniciar y cancelar el proceso de registro de una nueva tarjeta RFID, coordinando la comunicación entre la app, el backend y el dispositivo IoT mediante **MqttGateway**.
- Procesa eventos recibidos por MQTT para la detección y registro de tarjetas.
- Actualiza la asociación entre usuarios y tarjetas RFID y desactiva los sensores de movimiento en todas las habitaciones cuando se detecta una tarjeta válida.

7

Frontend Android

Con nuestro dispositivo IoT en funcionamiento y conectado a nuestro backend Spring, podemos finalmente desarrollar la última capa de nuestro sistema, la aplicación Android. Desde aquí el usuario interactuará con todos los servicios que hemos puesto a disposición en el servidor y podrá acceder a todas las funcionalidades del sistema.

Como explicamos en la sección de 1.2 de la introducción, queremos que la aplicación sea intuitiva de usar y sobre todo eficiente y funcional, por lo que no va a tener un diseño muy complejo con muchos plugins y florituras.

Antes de empezar a explicar cada componente creado, es importante ver los distintos tipos de archivo que podemos encontrar en nuestro proyecto de Android Studio y la función de cada uno. Primero tenemos los archivos de configuración de proyecto: **build.gradle.kts** define las dependencias de nuestra aplicación y configuraciones de compilación similar a **Maven** en el backend, **settings.gradle.kts** sirve para estructurar el proyecto y **AndroidManifest.xml** declara los componentes y actividades que forman la aplicación, establece permisos y configuraciones de Android.

Los archivos escritos en Kotlin (.kt) son el núcleo funcional de la aplicación y según cómo los nombremos se ocuparán de distintas cosas. Las **Activities** representan "pantallas" principales y se ocupan del flujo de la interfaz de usuario, los **ApiService** es la lógica de negocio, ejecuta tareas y funciones en segundo plano. Los **Adapters** conectan objetos de nuestro backend a objetos en nuestra aplicación, traduciendo a formatos compatibles con Android. Estos objetos quedan representados en nuestra aplicación como **Classes** almacenadas en el paquete "Model" y tienen los mismos atributos y campos que sus homólogos en el backend Spring.

Para el aspecto visual de la aplicación tenemos los archivos de recursos. Los **Layout** definen la estructura visual de cada pantalla en formato .xml. Por suerte, Android Studio cuenta con un editor visual de archivos XML, por lo que la creación de interfaces ha sido un proceso

fácil ya que solo había que hacer un boceto de cada interfaz y ajustar los valores numéricos en el archivo para que quede con medidas profesionales. En esta carpeta también se almacenan otros objetos que queramos definir de la IU como textos predefinidos, colores y temas.

Diagrama de clases

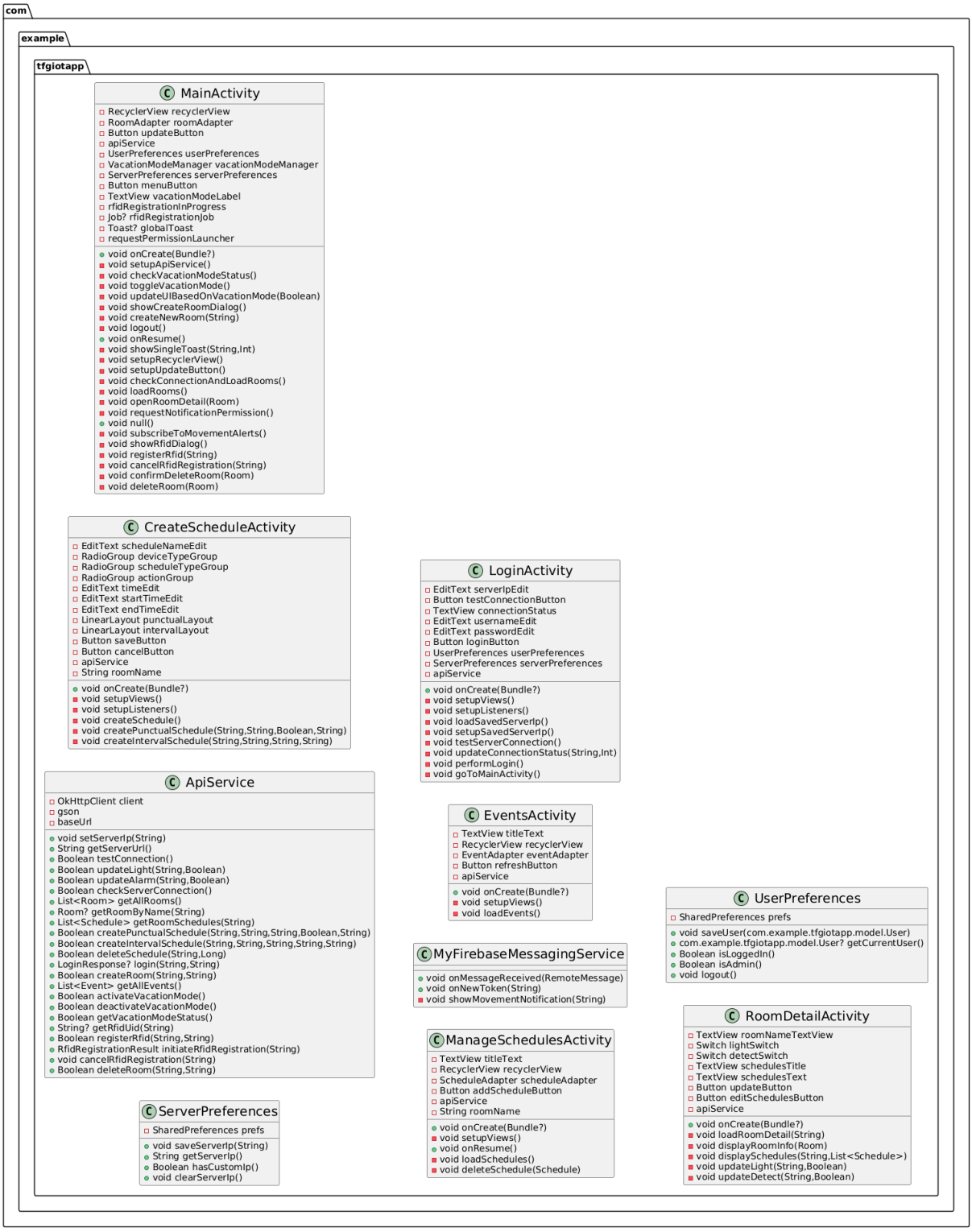


Figura 31: Diagrama de clases generado con plantUML

Vamos a explicar cada "pantalla" o **Activity** de la aplicación en orden y explicar que funciones ofrece al usuario.

Pantalla de autenticación

Lo primero que se encuentra el usuario al abrir la aplicación es esta pantalla de login, figura 32. Aquí deberá introducir la dirección IP del servidor Spring junto con sus sus credenciales (usuario y contraseña) para poder acceder a el menú de habitaciones. Puede utilizar el botón de "Probar" para confirmar que ha introducido la dirección del servidor correctamente. Cuando pulsa el botón de iniciar sesión, **ApiService.kt** envía la petición HTTP de autentificación a el controlador del backend, y según la respuesta que llegue de vuelta, permitirá al usuario entrar a la aplicación o informará al usuario de que error se ha producido según la excepción Java que llegue.

La pantalla de autenticación está implementada en *LoginActivity.kt*, donde se gestionan tanto la introducción y validación de la IP del servidor como el proceso de login. El botón de "Probar" ejecuta una petición de prueba usando *ApiService.testConnection()*, deshabilitando el botón mientras espera la respuesta. Las credenciales y la IP se almacenan usando las clases *UserPreferences* y *ServerPreferences*, permitiendo el acceso automático en futuros inicios de la app. Los mensajes de error se muestran mediante **Toast** según el tipo de excepción recibida.

La aplicación guarda la información del servidor en **ServerPreferences** y la del usuario en el archivo **UserPreferences** para evitar tener que iniciar sesión cada vez que se abre la app.

Pantalla principal

En estas pantallas de la figura 33 y 33a, se muestra a los usuarios logeados correctamente la lista de habitaciones presentes en nuestro sistema. Para cada habitación se muestra además el estado actual de sus luces y sensores de movimiento. En la parte de abajo de la pantalla aparecen un botón de actualizar, que sirve para enviar de nuevo el *GET* al endpoint de nuestro servidor que devuelve la lista de habitaciones. Si el usuario es administrador (figura 33b), al pulsar el botón superior "Menú", le apareceran varias opciones: Crear habitación, lista de eventos, modo vacaciones, tarjeta rfid y cerrar sesión. Además, verá un botón para eliminar ha-

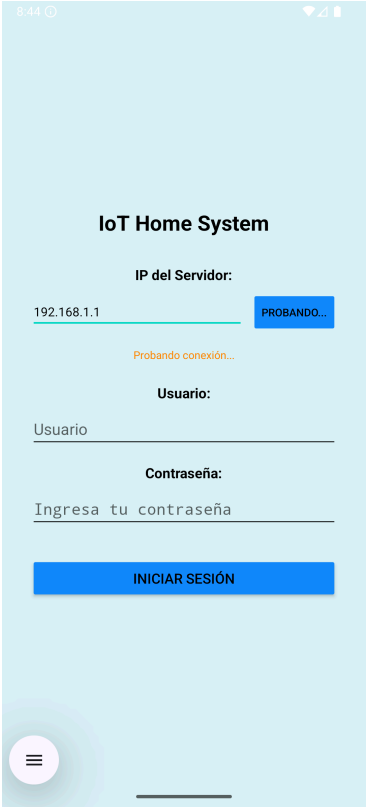
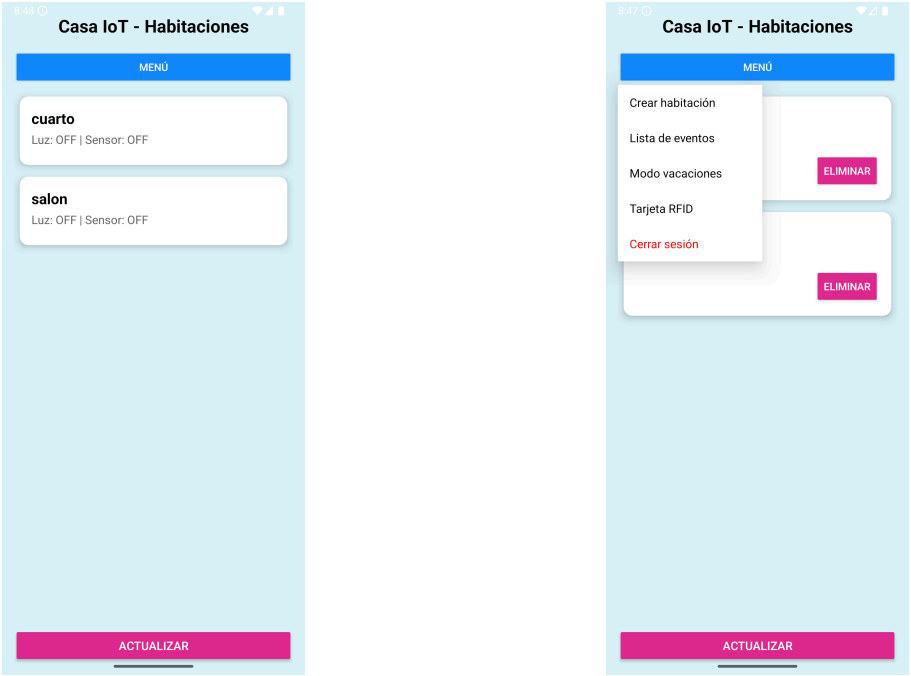


Figura 32: Layout para autenticación inicial

bitaciones a la derecha de cada una. El botón de cerrar sesión devuelve al usuario a la pantalla anterior y borra sus **UserPreferences**.

La pantalla principal está implementada en *MainActivity.kt*. La lista de habitaciones se muestra usando un *RecyclerView* con el adaptador *RoomAdapter*. Los botones de crear habitación, ver eventos y activar el modo vacaciones están disponibles solo para administradores (subfigura 33b), controlados mediante *UserPreferences.isAdmin()*. El botón de actualizar recarga la lista de habitaciones mediante una llamada a *ApiService.getAllRooms()*. El estado de los botones y la interfaz se actualiza dinámicamente según el estado del modo vacaciones usando *VacationModeManager*. El registro y visualización de tarjetas RFID se gestiona con diálogos y corrutinas, y la comunicación con el backend se realiza siempre usando la IP guardada en *ServerPreferences*.

Al pulsar el botón de “Crear nueva habitación” al administrador le aparecerá un cuadro de diálogo, figura 35a, en el que introducir el nombre para la nueva habitación, que será enviada con *POST* al backend y creada en el sistema. Se asume que todas las habitaciones tienen luces y sensor de movimiento configurados.



(a) Vista usuario

(b) Vista administrador

Figura 33: Pantalla principal, listado de habitaciones

El botón "Tarjeta RFID" permite consultar la tarjeta asociada al usuario y registrar una nueva (figura 34). Al pulsar "Registrar nueva", el sistema activa el sensor RFID durante 30 segundos para asociar la tarjeta al usuario; si no se detecta ninguna tarjeta o se cancela, no se guarda ningún cambio.

Si en su lugar pulsa el botón de "Ver todos los eventos" aparecerá una lista, figura 35b con todos los eventos loggeados en la base de datos.

Por último, el botón azul sirve para activar/desactivar el *Modo vacaciones* de nuestro sistema. Cuando este modo está activo, el usuario no puede utilizar ninguna función de la aplicación, salvo acceder a los logs para ver detecciones de movimiento. Los botones no accesibles aparecerán en gris, como se ve en la figura 36, y darán un mensaje apropiado hasta que el modo vacaciones se desactive.

Pantalla de habitación

Al pulsar sobre una de las habitaciones de la pantalla principal, aparece está ventana, figura 37, con los detalles de la habitación. Desde aquí el usuario puede enceder o apagar las

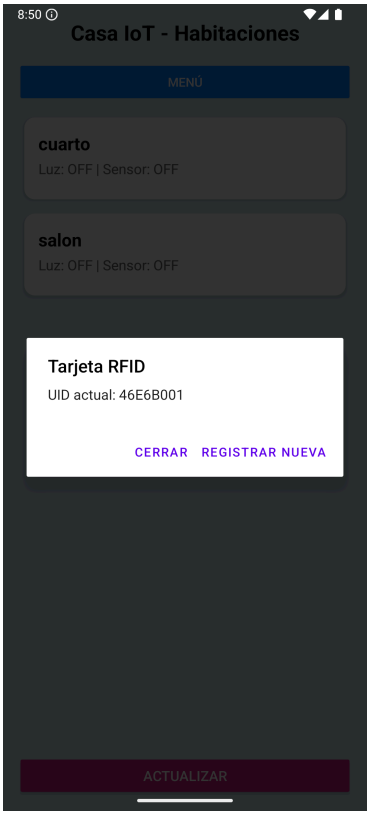
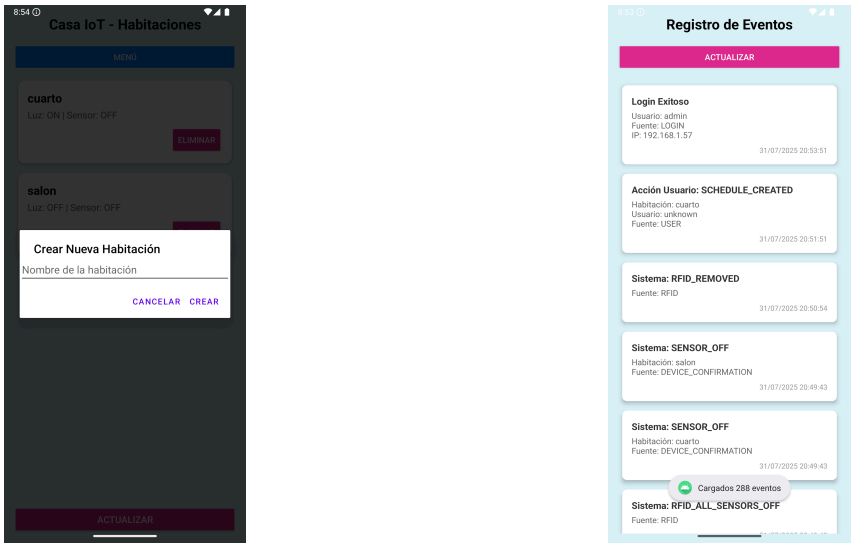


Figura 34: Pop-up para tarjetas RFID



(a) Cuadro de diálogo para crear nueva habitación (b) Lista de eventos

Figura 35: Funciones para administradores

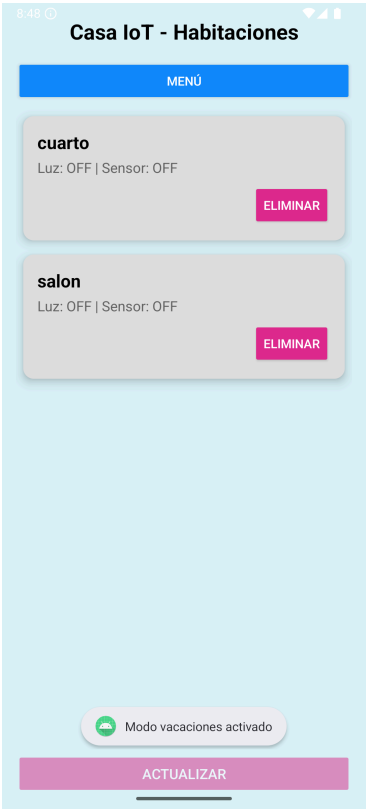


Figura 36: Pantalla principal con modo vacaciones activado

luces, activar o desactivar el sensor de movimiento de la habitación y gestionar los horarios existentes en esa habitación.

La pantalla de detalles de habitación está implementada en *RoomDetailActivity.kt*. Los switches de luces y sensores están sincronizados con el backend: al cambiar su estado, se envía una petición mediante *ApiService.updateLight()* o *ApiService.updateAlarm()*, y solo se actualiza el estado en la interfaz si el backend confirma la acción. Los horarios de la habitación se obtienen y muestran usando *ApiService.getRoomSchedules()*, y los cambios se reflejan en tiempo real en la interfaz.

Los switch que muestran el estado de luces y sensores están sincronizados con la base de datos, es decir, cuando el usuario envía un cambio (encender las luces del salón, por ejemplo), la aplicación hace el *POST* al backend, este recibe la petición y envía el comando MQTT al dispositivo IoT pero no se actualiza el estado en la base de datos hasta que el dispositivo IoT envíe una respuesta confirmando que se ha ejecutado el comando.

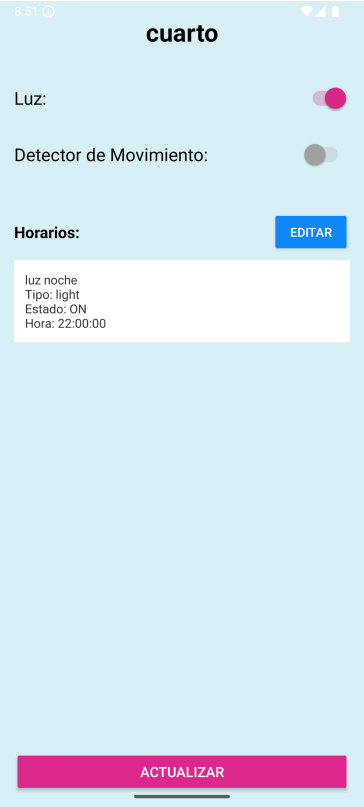


Figura 37: Detalles de habitación

Editor de horarios

A esta pantalla de la figura 38 se accede pulsando el botón "Editar" que aparece a la derecha de la lista de horarios. Aquí aparece la lista completa de horarios para la habitación actual, podemos ver los detalles de cada uno y eliminarlos.

El editor de horarios está implementado en *ManageSchedulesActivity.kt* y *CreateScheduleActivity.kt*. La lista de horarios se muestra con un *RecyclerView* y el adaptador *ScheduleAdapter*. Al añadir o eliminar un horario, se realizan llamadas a *ApiService* para actualizar el backend y la interfaz se refresca automáticamente. La validación de campos y la gestión de errores se realiza mostrando mensajes mediante *Toast* en la interfaz.

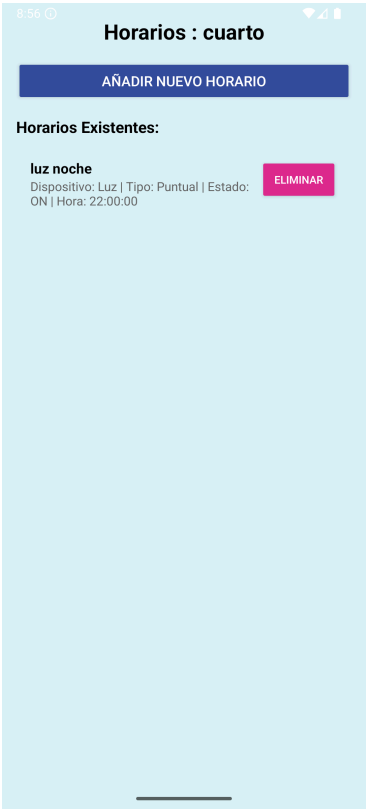
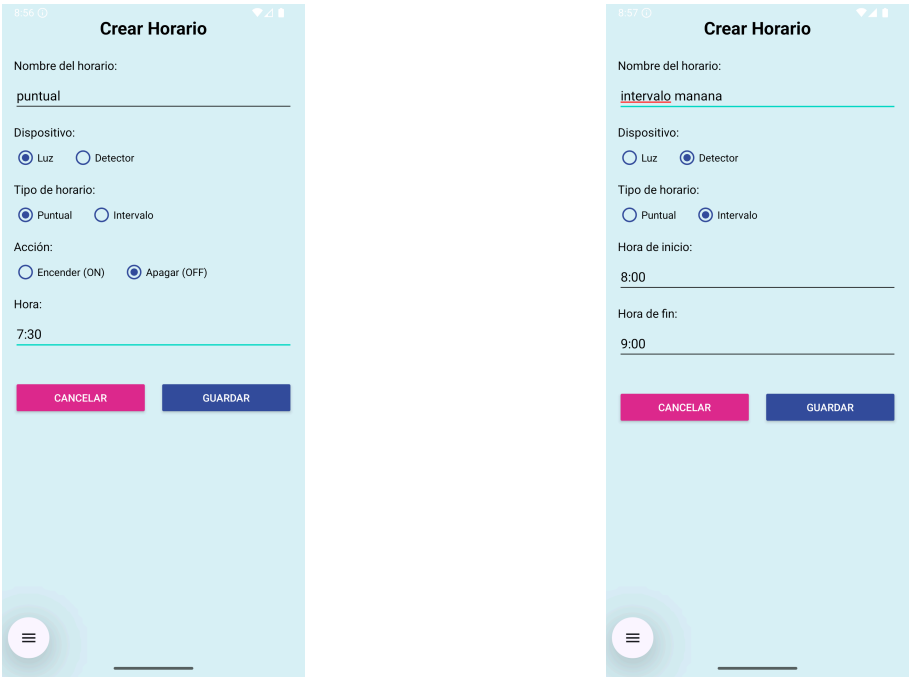


Figura 38: Editor de horarios

Al pulsar “Añadir nuevo horario”, el usuario puede crear y guardar un horario para la habitación desde la pantalla de la figura 39, eligiendo dispositivo, tipo y hora; si hay errores en los datos, la app lo notifica.



(a) Crear horario puntual

(b) Crear horario de intervalo

Figura 39: Ventana creación de horario

Con estas pantallas cubrimos toda la funcionalidad especificada en nuestros casos de uso. El usuario puede iniciar sesión, gestionar habitaciones, gestionar horarios, encender dispositivos, etc.

8
Testing

Para asegurar la fiabilidad y robustez del sistema desarrollado, se ha implementado una batería de pruebas automáticas sobre el backend utilizando los frameworks **JUnit 5** y **Mockito**. Además, se ha empleado la herramienta **JaCoCo** para el análisis de cobertura de código, permitiendo cuantificar el grado de protección frente a errores y facilitar la mejora continua.

8.1. Herramientas y metodología

El framework **JUnit 5** ha sido la base para la definición y ejecución de los tests unitarios y de integración. Gracias a la fácil integración con Spring Boot, ha permitido estructurar los casos de prueba de forma modular. Por otro lado, hemos utilizado **Mockito** para la creación de mocks de dependencias, especialmente en los servicios y repositorios, posibilitando el aislamiento de la lógica de negocio y la simulación de escenarios sin necesidad de acceder a recursos externos como la base de datos.

La herramienta **JaCoCo** se ha integrado en el ciclo de construcción mediante Maven, generando informes detallados sobre la cobertura alcanzada por los tests. Estos informes han sido fundamentales para identificar ramas y métodos no cubiertos, permitiendo así incrementar la calidad del software mediante la incorporación de nuevos casos de prueba.

8.2. Estructura y funcionamiento de los tests

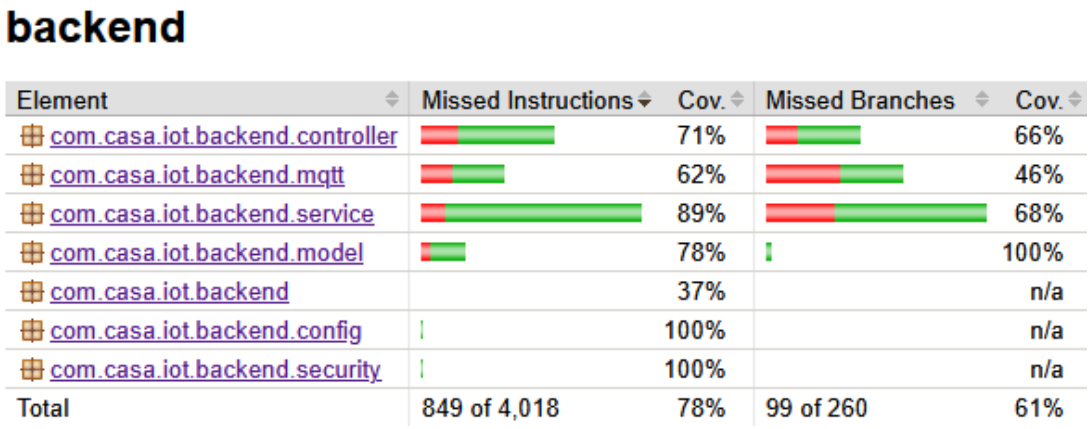
Los tests se han organizado en clases específicas para cada componente principal del sistema, siguiendo el patrón *Arrange-Act-Assert*. En la fase de *Arrange* se preparan los mocks y los datos de entrada; en la fase de *Act* se ejecuta el método bajo prueba; y en la fase de *Assert* se comprueba que el resultado o las interacciones son las esperadas.

Por ejemplo, en los tests de servicios como **EventLogService**, se verifica que al invocar métodos de registro de eventos, como el de detección de movimiento o la ejecución de horarios,

se realiza correctamente la persistencia en el repositorio y se generan los eventos esperados. Para los controladores, empleamos **MockMvc** para simular peticiones HTTP y validar tanto el código de respuesta como la estructura del JSON devuelto.

8.3. Cobertura obtenida

La integración de JaCoCo nos ha permitido alcanzar una cobertura superior al **85 %** en las clases de servicio y controlador, cubriendo los principales flujos de negocio y casos de error. En el informe de la figura 40 aparece un **71 %** de cobertura en los controladores debido a que el controlador de simulación no contiene pruebas, pues lo hemos utilizado únicamente al desarrollar y no es alcanzable por los usuarios. El resto de áreas del código sin cobertura son los flujos de error general en muchos métodos para capturar posibles excepciones, que aunque se podrían incluir en futuras iteraciones, no afectarán al funcionamiento de nuestro sistema de seguridad e iluminación.



backend

Element	Missed Instructions	Cov.	Missed Branches	Cov.
com.casa.iot.backend.controller		71%		66%
com.casa.iot.backend.mqtt		62%		46%
com.casa.iot.backend.service		89%		68%
com.casa.iot.backend.model		78%		100%
com.casa.iot.backend		37%		n/a
com.casa.iot.backend.config		100%		n/a
com.casa.iot.backend.security		100%		n/a
Total	849 of 4,018	78%	99 of 260	61%

Figura 40: Informe generado por JaCoCo

Los informes generados han facilitado la identificación de áreas no testeadas y la mejora progresiva de la batería de pruebas, que ha ido creciendo según íbamos desarrollando el código. Gracias a esto, hemos conseguido un backend más robusto y hemos asegurado el correcto funcionamiento, minimizando el riesgo de regresiones y facilitando la evolución futura del proyecto.

8.4. Otros tests

Con estas herramientas y tests, hemos cubierto la mayoría de la funcionalidad del sistema, pero aún debemos asegurarnos del correcto funcionamiento tanto de la aplicación Android como del dispositivo IoT. Este testing ha sido realizado mediante pruebas de usuario final, asegurándonos de que el dispositivo cumple los requisitos no funcionales establecidos en la sección 3.5.

Para asegurarnos del correcto rendimiento del dispositivo, comprobamos que el tiempo de respuesta entre un usuario tomando una acción en la app (por ejemplo, encender una luz concreta) es menor de 2 segundos con dos aplicaciones distintas conectadas al backend. Lo mismo ocurre con el tiempo entre que el dispositivo detecta movimiento en una habitación con el sensor activo y envía la notifiación push al usuario.

Para asegurar la disponibilidad del sistema y la usabilidad de la aplicación hemos implementado una interfaz de usuario clara y simple, que haciendo uso extensivo de verificación de conexión, corrutinas, gestión de errores y excepciones y feedback visual inmediato (a través de mensajes toast que informan sobre el estado de la aplicación, figura 41), minimiza el impacto de interrupciones en el backend en nuestra experiencia de usuario.

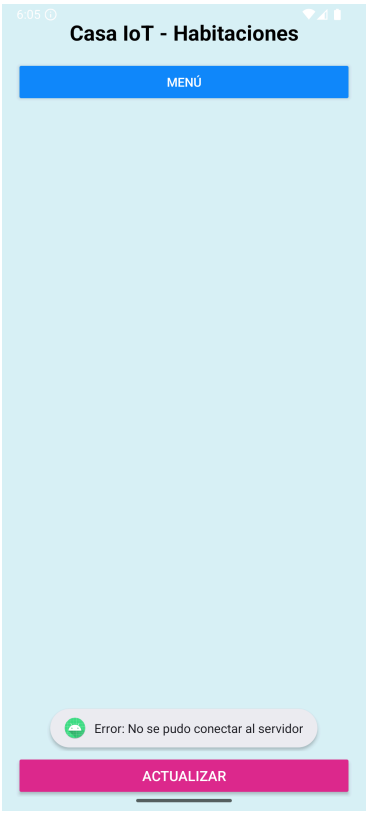


Figura 41: Mensaje de error de la aplicación

9

Conclusiones y perspectivas futuras

Con de este sistema IoT para la gestión de seguridad e iluminación doméstica hemos conseguido integrar de manera efectiva tecnologías modernas tanto en el backend (Spring Boot, MQTT, FCM) como en el frontend (Android, Material Design, notificaciones push). El resultado es una plataforma robusta, flexible y fácilmente extensible, capaz de cubrir las necesidades básicas y avanzadas de automatización en el hogar.

Hemos cubierto todos los objetivos principales que nos planteamos al inicio del desarrollo: la gestión centralizada de habitaciones y dispositivos, la programación de horarios automáticos, la integración de sensores de movimiento y tarjetas RFID para control de acceso, y la notificación en tiempo real de eventos críticos a los usuarios. La arquitectura modular y el uso de patrones de diseño como servicios desacoplados y mensajería asíncrona han facilitado la escalabilidad y claridad del código en el proyecto.

La experiencia de usuario en la aplicación Android se ha visto enriquecida gracias a la adopción de Material Design y la implementación simple y fácil para la gestión de habitaciones, horarios y tarjetas RFID. Las funcionalidades de gestión de eventos y habitaciones se han mantenido exclusivas para los usuarios administradores, permitiendo el uso por más personas del hogar sin preocupaciones sobre control de acceso. El uso de Firebase Cloud Messaging ha permitido una comunicación fiable entre el backend y los dispositivos móviles, asegurando que los usuarios estén siempre informados de cualquier evento relevante en su hogar cuando no estén en casa mediante notificaciones push.

Desde el punto de vista técnico, la integración de MQTT ha demostrado ser una solución óptima para la comunicación en tiempo real con los dispositivos IoT, permitiendo una respuesta inmediata ante eventos como la detección de movimiento o el uso de tarjetas RFID. El

backend, construido sobre Spring Boot, ha proporcionado una base sólida para la gestión de la lógica de negocio, la persistencia de datos y la seguridad.

Perspectivas futuras

El sistema desarrollado sienta las bases para múltiples líneas de mejora y expansión:

- **Integración de nuevos dispositivos:** En su estado actual, el sistema funciona con un único dispositivo ESP32 que cubre dos habitaciones. En futuras expansiones sería sencillo añadir más dispositivos que escuchen los mensajes de los tópicos y tomen las acciones necesarias. Además, se podrían implementar otros dispositivos IoT como sensores de temperatura o luz y actuadores para persianas o enchufes inteligentes. Todo gracias al diseño modular que hemos deseado desde el principio.

- **Interfaz web:** El desarrollo de un panel de control web permitiría gestionar el sistema desde cualquier navegador, facilitando el acceso multiplataforma y la administración remota.

- **ESP-NOW:** Este es un nuevo protocolo de comunicación inalámbrica creado por Espressif Systems específicamente para los ESP32. Espera suplantar la necesidad de conectarse a una red wifi convencional y ofrece baja latencia y seguridad por una conexión directa entre dispositivos [10].

- **Integración con asistentes de voz:** La conexión con plataformas como Google Assistant o Alexa permitiría controlar el sistema mediante comandos de voz, mejorando la accesibilidad y la comodidad.

- **Mejoras en seguridad:** Se pueden implementar mecanismos de autenticación multifactor, cifrado de extremo a extremo en las comunicaciones y detección de intrusiones basada en IA.

- **Escalabilidad y despliegue en la nube:** Migrar el backend a una infraestructura cloud permitiría soportar un mayor número de usuarios y dispositivos, así como ofrecer servicios de monitorización y mantenimiento centralizados.

- **Analítica y visualización de datos:** Incorporar dashboards de análisis de consumo, patrones de uso y estadísticas de seguridad aportaría valor añadido a los usuarios y facilitaría la toma de decisiones.

En conclusión, hemos cumplido nuestros objetivos y requisitos establecidos y hemos acabado con un producto mucho más modular, personalizable y asequible que otras soluciones de seguridad e iluminación disponibles en el mercado. Además, el proyecto establece una plataforma sólida sobre la que construir futuras mejoras e incluir aún más funciones domóticas junto con mejorar las existentes.

Apéndice A
Manual de Instalación

Requisitos previos

Antes de comenzar la instalación es importante asegurarnos de que contamos con todo el software y hardware necesario. Para una configuración mínima con solo una habitación necesitaremos al menos una placa ESP32, un sensor de movimiento PIR, un LED de cualquier color y un lector RFID RC522 con tarjetas compatibles.

Para poder correr el backend en nuestro sistema es importante que tengamos instalado **Java 17** o superior. Para crear la base de datos necesitaremos MySQL y por último necesitaremos haber descargado el broker de Mosquitto para que el código sea capaz de encontrarlo e iniciarlo. La aplicación solo requiere un móvil con una versión de Android igual o superior a 6.0.

Por último, para adaptar el código del ESP32 a nuestra configuración exacta necesitaremos instalar el entorno de Arduino IDE junto con las librerías utilizadas.

Instalación del Backend

El primer paso para tener el backend Spring en marcha es descargar el repositorio con todos los archivos del proyecto, disponible en la página de GitHub. Podemos utilizar git para clonar el repositorio a nuestra máquina o descargar el zip del proyecto desde la página, al final deberíamos tener un directorio con una estructura como la de la figura 42. Es recomendable abrir la carpeta del proyecto en Visual Studio Code para facilitar la edición de archivos y la compilación con extensiones como *Code Runner*.

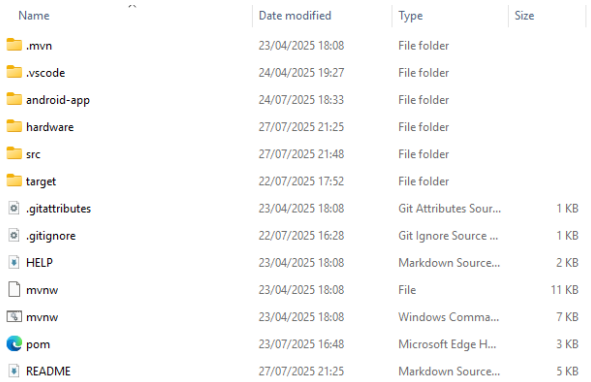


Figura 42: Repositorio del proyecto

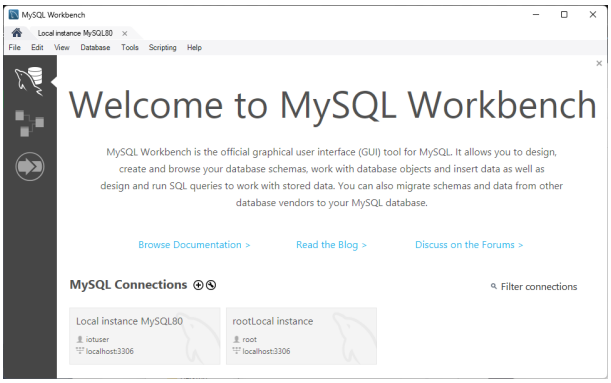


Figura 43: Conexiones disponibles en MySQL Workbench

Base de datos

Con el repositorio descargado, pasamos a crear la base de datos en MySQL. Este paso es necesario ya que Spring no tiene los permisos para crear la base de datos automáticamente, así que la crearemos a mano y la conectaremos posteriormente. Al abrir MySQL Workbench aparecerá por defecto la conexión a la instancia local de MySQL con el usuario “root”, como se ve en la figura 43. La contraseña para esta conexión será la que configuramos mientras instalabamos el entorno MySQL Workbench. Durante la instalación también Una vez nos hemos conectado al servidor, podemos ejecutar el siguiente código SQL en una página en blanco para crear la base de datos:

```
CREATE DATABASE iotdb;
```

Es importante crear además credenciales de usuario de administrador para darle a nuestro servidor Spring y que no utilice el root. Para ello solo hay que ejecutar las siguientes líneas

reemplazando “nuevo_usuario” y “nueva_contraseña”:
CREATE USER ’nuevo_usuario’@’ %’ IDENTIFIED BY ’nueva_contraseña’;
GRANT ALL PRIVILEGES ON IOTDB.* TO ’nuevo_usuario’@’ %’;
FLUSH PRIVILEGES;

Lo único que nos quedaría sería modificar el archivo *application.properties* en el repositorio con la siguiente configuración:
spring.datasource.url=jdbc:mysql://localhost:3306/iotdb
spring.datasource.username=’nuevo_usuario’
spring.datasource.password=’nueva_contraseña’

Firebase Cloud Messaging

Para la funcionalidad de enviar notificaciones push al usuario en caso de detección de movimiento, es necesario que incluyamos en los archivos del backend una clave para nuestra cuenta de servicio de Firebase en forma de json. Para obtener esta clave, tenemos que ir a la página de Firebase Console, crear o iniciar sesión en una cuenta de Google y crear un nuevo proyecto. Dentro del proyecto debemos activar **Cloud Messaging**, una vez activado podemos descargar el archivo json yendo a “Configuración de proyecto” >“Cuentas de servicio” y pulsando “Generar nueva clave privada”. Solo tenemos que moverlo a la carpeta src/main/resources del repositorio y renombrarlo como *firebase-service-account* para que el código pueda leerlo.

Mosquitto broker

El último componente necesario para la correcta funcionalidad del backend es el broker de MQTT Mosquitto. El servidor es capaz de iniciar el servicio si este no está corriendo pero es necesario que el usuario lo instale en su sistema manualmente previamente. Es preferible que al instalarlo en Windows se instale a la carpeta C:/Program Files/Mosquitto. Dentro de la carpeta instalada está el archivo *mosquitto.conf*, podemos abrirlo con cualquier editor de texto, eliminar la configuración por defecto y simplemente añadir estas dos líneas:
listener 1883 0.0.0.0

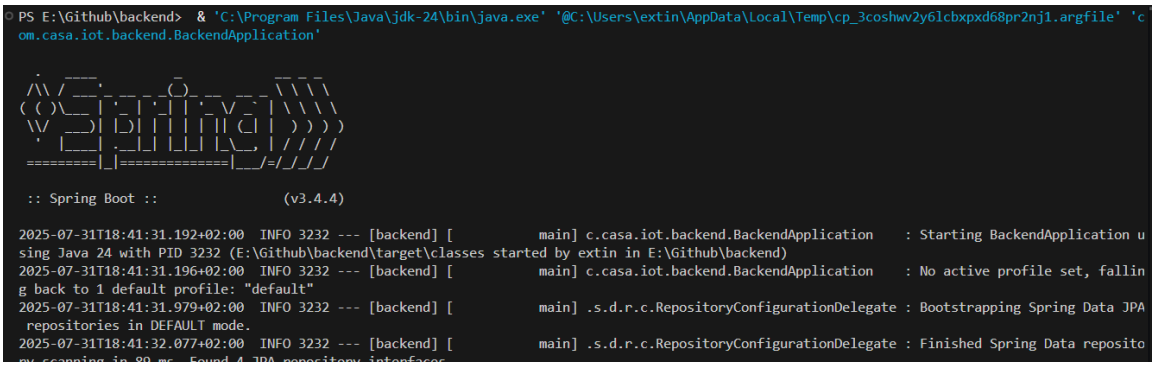


Figura 44: Output del servidor Spring en marcha

allow_anonymous true

Ejecución del backend

Con todo instalado, podemos poner en marcha nuestro servidor. En Visual Studio Code es tan sencillo como ir al archivo *BackendApplication.java* y pulsar el icono de "Run" en la esquina superior derecha. Si no tenemos VSCode, no es problema, con Maven podemos simplemente ir a la carpeta del directorio principal, donde aparece nuestro archivo *pom.xml* y abrir una terminal en ese directorio. Aquí ejecutamos el comando **mvn spring-boot:run** y automáticamente se compilará y lanzará el backend. En ambos casos podremos monitorear los logs del servidor a través de la terminal como se observa en la figura 44.

Podemos utilizar el endpoint */hello* para verificar que el backend responde yendo a la dirección *http://localhost:8080/hello* en cualquier browser.

Instalación de la Aplicación

Para utilizar la aplicación debemos generar el instalador .apk desde el entorno Android Studio. Una vez iniciado el IDE seleccionamos la opción para abrir un proyecto existente y seleccionamos la carpeta *android-app* de nuestro repositorio. Una vez cargado, Android Studio sincroniza automáticamente todas las dependencias y configura el entorno de compilación.

Desde aquí podemos utilizar el emulador incorporado para inciar la app en un dispositivo virtual pulsando el botón de ejecutar en la parte superior de la ventana. Esto iniciará el dispositivo e instalará la app, permitiéndonos probar toda la aplicación desde el propio entorno.

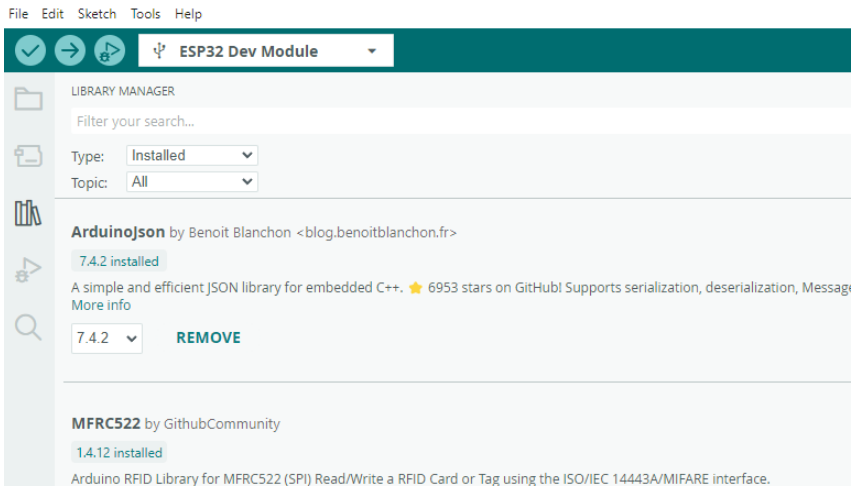


Figura 45: Gestor de librerías de Arduino IDE

Si deseamos instalarlo en un dispositivo móvil físico, tenemos que navegar al menú *Build* y seleccionar la opción *Build APK(s)*, que producirá el paquete instalable en la ruta "android-app/app/build/outputs/apk/debug" por defecto o donde nos indique la notificación. Con el .apk generado, solo tenemos que transferirlo al sistema de archivos de nuestro teléfono e instalarlo desde ahí.

Configuración del dispositivo IoT

Librerías

Una vez instalado Arduino IDE necesitamos descargar todas las librerías desde el Gestor de librerías incluido (figura 45). Necesitaremos introducir en la barra de búsqueda e instalar las siguientes: ArduinoJson, MFRC522, PubSubClient y WiFiManager.

Conexión con el ESP32

Antes de cargar el código tenemos que asegurarnos de instalar el paquete de soporte para la placa esp32 de Espressif Systems [7]. Para ello debemos dentro de Arduino IDE navegar a "File" >"Preferences" e introducir en el campo llamado "Additional Boards Manager URLs" el siguiente enlace:

https://espressif.github.io/arduino-esp32/package_esp32_index.json

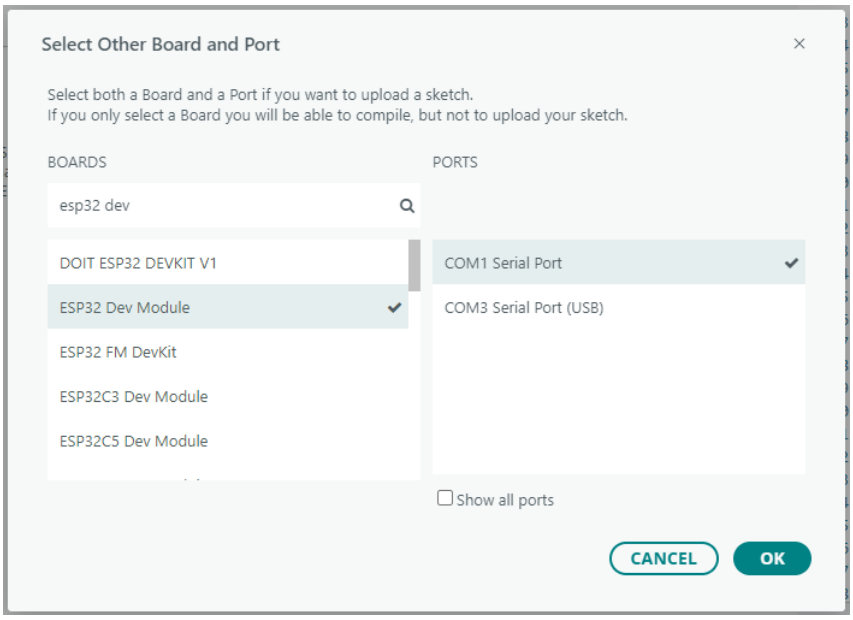


Figura 46: Configuración de puerto USB en Arduino IDE

Una vez añadido, navegamos a el gestor de placas en el menú de la izquierda y buscamos "esp32". Seleccionamos el que tiene a Espressif Systems como autor y listo. Esto instalará varias placas para configurar nuestros puertos junto con ejemplos de código para probar la conexión y componentes de nuestro hardware.

Ahora podemos configurar nuestro puerto USB con el esp32 conectado como se ve en la figura 46. Buscamos y seleccionamos "ESP32 Dev Module" y lo asignamos a el puerto con nuestra placa conectado. Para comprobar la conexión podemos cargar un sketch vacío, abrir el *Serial Monitor* desde el menú de tools y pulsar el botón de reset físico de la placa. Deberíamos una serie de mensajes sobre la flash de la placa.

Código

Es importante antes de pasar el código a nuestro ESP32 tener claro las habitaciones que queremos modelar. Vamos a explicar el proceso suponiendo que tenemos una placa esp32 con las conexiones descritas en la sección 5.2 de la memoria. El código de Arduino IDE disponible en el repositorio está creado para 2 habitaciones llamadas "cuarto" y "salon", cada una de ellas con una luz y un sensor PIR.

Asumiendo que hemos hecho las conexiones exactamente como describimos en el esquema, solo necesitamos introducir en el código los nombres de las dos habitaciones que contro-

```
14  Preferences preferences;
15
16  // Habitaciones (nombres configurables)
17  char habitacion1[16] = "salon";
18  char habitacion2[16] = "cuarto";
19
20  // Pines del hardware
```

Figura 47: Habitaciones introducidas en nuestro esp32

lará nuestro esp32 y todo funcionará perfectamente siempre y cuando estas dos habitaciones tengan el mismo nombre exacto que en nuestra aplicación móvil. Los nombres los introducimos en las líneas 17 y 18 del código .ino como se aprecia en la figura 47

Finalmente, pulsamos el botón con la flecha arriba a la izquierda para subir el código a la placa. Si todo ha salido bien, WiFiManager iniciará el punto de acceso "CasaIoT-Setup", y podremos acceder con la IP que aparezca en el Serial Monitor a un portal48 donde conectar el ESP32 a nuestra conexión WiFi e introducir la IP del broker MQTT, que será la máquina con el servicio de Mosquitto en ejecución.

Al introducir una configuración correcta, el esp32 imprimirá los detalles de la conexión y las habitaciones configuradas por el Serial Monitor como se ve en la figura 49.

Ejecución

Para probar que todo funciona, iniciamos el servidor Spring, conectamos y subimos el código a la placa esp32 y conectamos la aplicación Android al servidor. Creamos una habitación con el nombre configurado en el código Arduino y si todo ha ido bien, el esp32 encenderá y apagará los sensores y luces a petición de la App.

Una vez completados todos los pasos de este manual, habremos conseguido desplegar un sistema IoT funcional, con nuestro Servidor Spring corriendo, una base de datos, un broker MQTT Mosquitto, una aplicación móvil para controlarlo todo y un dispositivo ESP32 configurado y operativo. El sistema estará listo para gestionar habitaciones inteligentes, recibir notificaciones en tiempo real y monitorizar eventos.

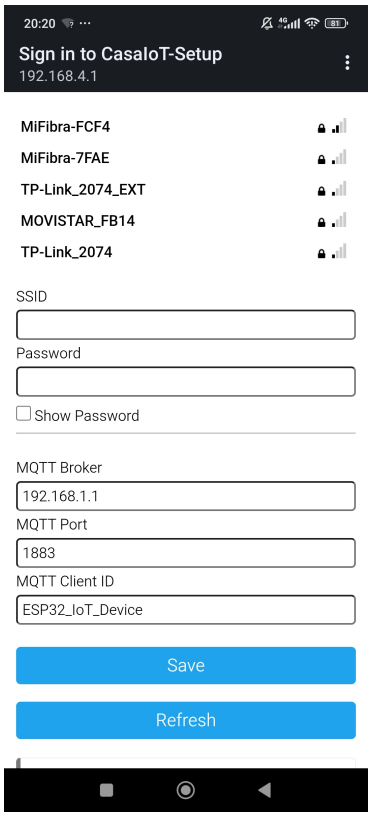


Figura 48: Portal cautivo de WiFi Manager

```
20:23:30.939 -> MQTT Broker: 192.168.1.57
20:23:30.939 -> MQTT Port: 1883
20:23:30.939 -> MQTT Client: ESP32_IoT_Device
20:23:30.939 -> Conectando a MQTT...conectado
20:23:31.242 -> Suscrito a tópicos MQTT
20:23:31.242 -> ESP32 IoT Device iniciado
20:23:31.282 -> Habitaciones: salon, cuarto
```

Figura 49: Detalles de la conexión de la placa esp32

Apéndice B

Manual de Usuario

Manual de usuario de la aplicación Android

Una vez instalada y configurada la aplicación móvil, el usuario puede gestionar y monitorizar todas las habitaciones inteligentes conectadas al sistema. A continuación se describen las funcionalidades principales y el uso recomendado de la app.

Inicio de sesión y configuración de servidor

Al abrir la aplicación, se muestra la pantalla de inicio de sesión (figura 50). El usuario debe introducir su nombre de usuario y contraseña previamente registrados en el sistema. Es posible configurar la IP del servidor desde la propia pantalla de login; para ello, basta con introducir la dirección y pulsar el botón *Probar conexión*. Si la conexión es correcta, la app lo indicará y permitirá continuar con el inicio de sesión. Toda la información introducida en esta pantalla se guarda en el dispositivo para evitar al usuario la necesidad de logearse cada vez que abre la aplicación.

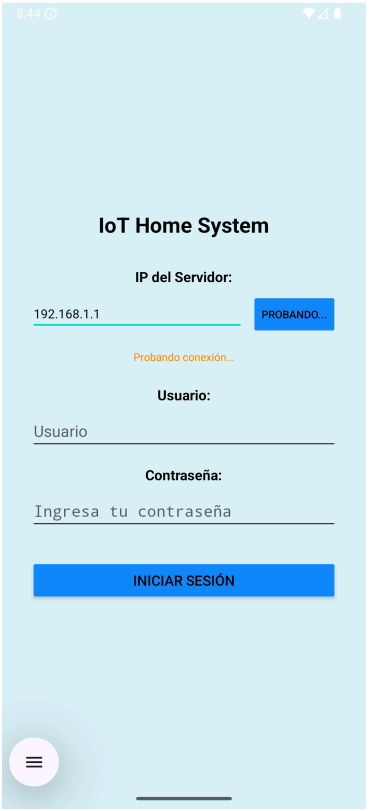
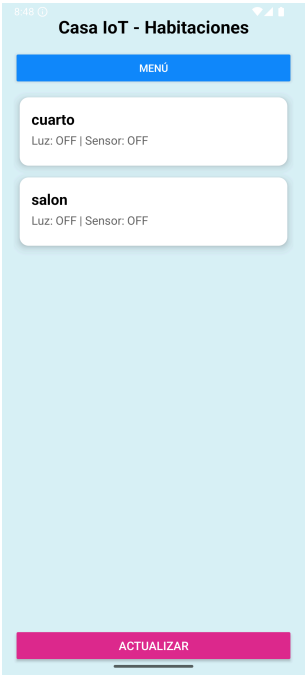


Figura 50: Pantalla de inicio de sesión

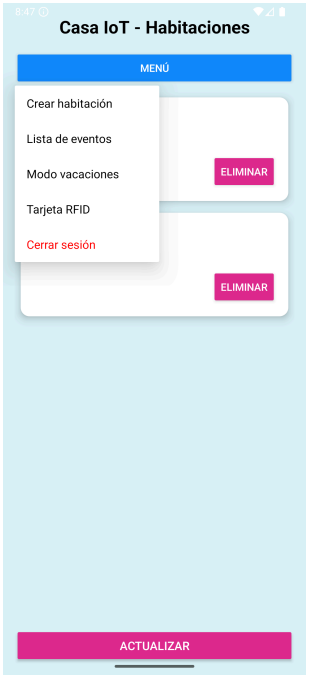
Pantalla principal y gestión de habitaciones

Tras iniciar sesión, se accede a la pantalla principal de la figura 51, donde se muestra el listado de habitaciones registradas en el sistema. El usuario puede actualizar el listado pulsando el botón correspondiente; si el servidor no está disponible, la app mostrará un mensaje de error. Desde el menú principal es posible crear nuevas habitaciones (si el usuario tiene permisos de administrador), acceder al historial de eventos, gestionar tarjetas RFID, activar o desactivar el modo vacaciones y cerrar sesión. Además, con cada habitación registrada aparece un botón de eliminar que un usuario administrador puede utilizar para borrar una habitación del sistema.

Para crear una habitación, se selecciona la opción en el menú, se introduce el nombre deseado y se confirma la acción. Al pulsar sobre una habitación, se accede a su detalle (figura 52), donde se puede consultar el estado de los dispositivos (luz, sensor de movimiento), modificar su estado y visualizar los horarios programados.



(a) Vista usuario



(b) Vista administrador

Figura 51: Pantalla principal, listado de habitaciones

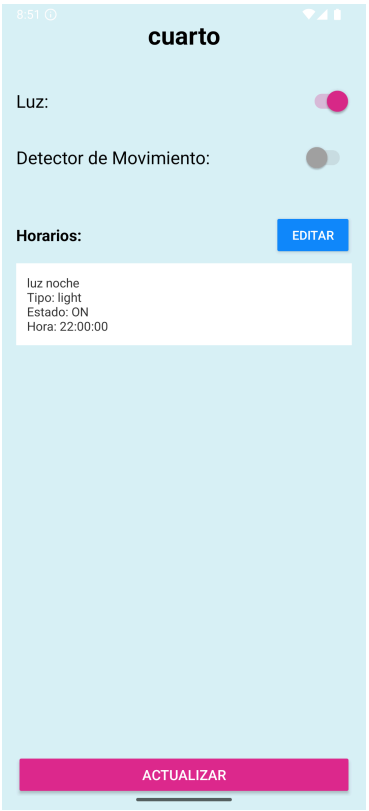


Figura 52: Detalles de habitación

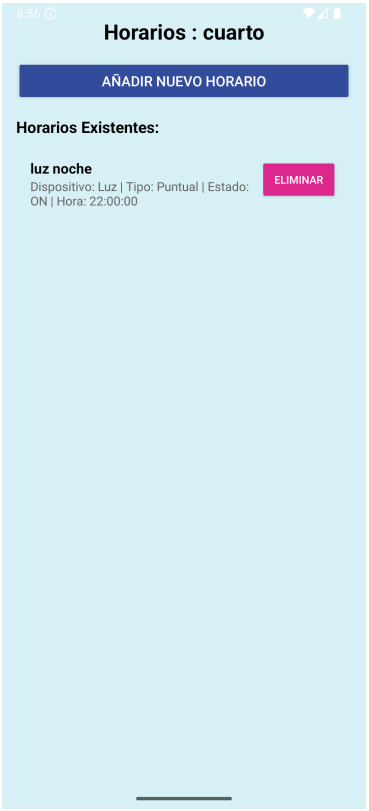


Figura 53: Editor de horarios

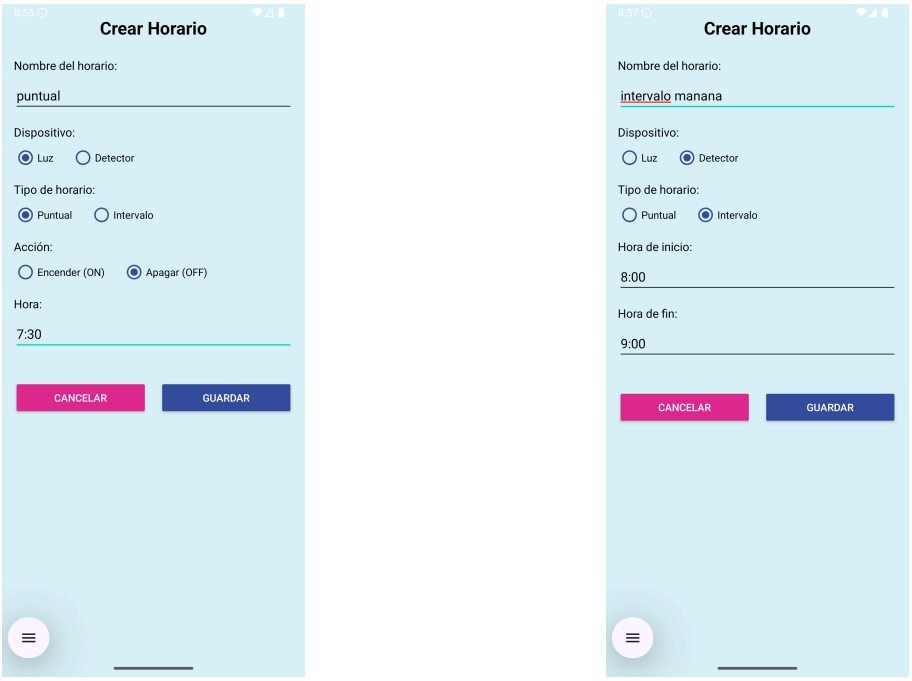
Control de dispositivos y programación de horarios

En el detalle de cada habitación, el usuario puede encender o apagar la luz y activar o desactivar el sensor de movimiento mediante los interruptores disponibles. Además, es posible añadir horarios de funcionamiento para los dispositivos con el botón "Editar" que aparece en el detalle (figura 53).

Cuando el usuario pulsa "Añadir nuevo horario" en la parte superior de la pantalla, se le muestra la siguiente página (figura 54) para crear un nuevo horario y guardarlo en la habitación. Deberá de introducir un nombre, elegir el dispositivo a controlar, el tipo de horario, la acción a tomar y la hora de ejecución o el intervalo horario, según el tipo elegido. Si el usuario introduce algún dato inválido en un campo se le notificará con un mensaje.

Eventos y notificaciones

La sección de eventos de la figura 55 permite a los administradores consultar el historial de acciones realizadas en el sistema, como cambios de estado, detección de movimiento o registros



(a) Crear horario puntual

(b) Crear horario de intervalo

Figura 54: Ventana creación de horario

de acceso. La aplicación envía notificaciones push al usuario en caso de eventos relevantes, como la detección de movimiento en alguna habitación, siempre que se hayan concedido los permisos necesarios.

Gestión de tarjetas RFID

El botón de "Tarjeta RFID" del menú principal abre un pop-up (figura 56) que, si la tiene, muestra el código *UID* de la tarjeta asociada al usuario actual y si no, un mensaje. Aquí el usuario puede pulsar el botón "Registrar nueva", esto activará el sensor RFID de nuestro sistema durante de tal manera que si se pasa una tarjeta esta será asociada al usuario de la aplicación en nuestra base de datos y le permitirá usarla para apagar las alarmas del hogar. Si en 30 segundos no se pasa ninguna tarjeta RFID o el usuario pulsa cancelar no se guardará ninguna tarjeta para el usuario.

Figura 55: Lista de eventos

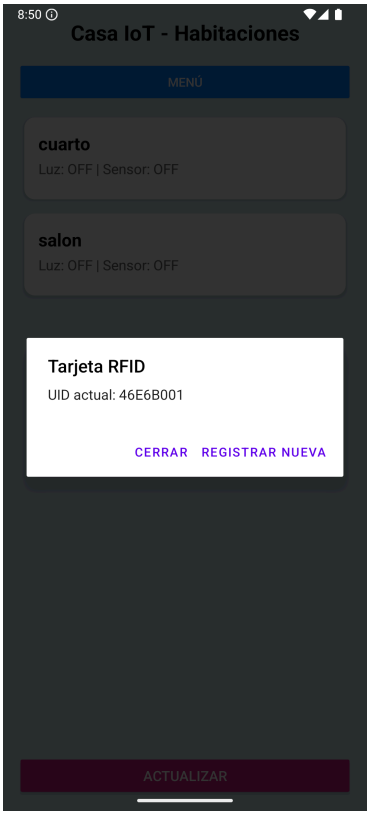


Figura 56: Pop-up para tarjetas RFID

Modo vacaciones

El modo vacaciones permite restringir ciertas acciones en el sistema, como la creación o modificación de habitaciones y dispositivos. El usuario puede activar o desactivar este modo desde el menú principal; la app mostrará un indicador visual (figura 57) cuando el modo esté activo y deshabilitará las funciones restringidas.

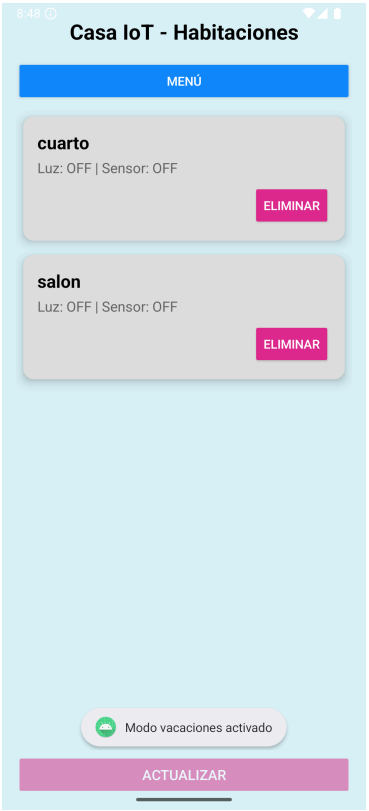


Figura 57: Pantalla principal con modo vacaciones activado

Cierre de sesión y solución de problemas

Para cerrar sesión, basta con seleccionar la opción correspondiente en el menú. Si en algún momento la app no muestra habitaciones o dispositivos, se recomienda comprobar la conexión al servidor y la configuración de la IP. Los mensajes de error informan al usuario sobre problemas de conectividad, permisos o funcionamiento del sistema.

Referencias

[1] Android Developers. *Estructura de compilación de Android*. 2024. URL: https://developer.android.com/build/android-build-structure?hl=es-419.

[2] Arduino. *Arduino Documentation*. 2024. URL: https://docs.arduino.cc/.

[3] Oscar Blancarte. *El concepto Inversión of Control*. 2016. URL: https://www.oscarblancarteblog.com/2016/12/01/concepto-inversion-of-control/.

[4] Digi-Key Electronics. *RF RC522 Datasheet*. 2024. URL: https://mm.digikey.com/Volume0/opasdata/d220001/medias/docus/5531/4411_CN0090%20other%20related%20document%20%281%29.pdf.

[5] Eclipse Mosquitto. *Mosquitto Documentation*. 2024. URL: https://mosquitto.org/documentation/.

[6] Espressif Systems. *ESP32-WROOM-32 Datasheet*. 2024. URL: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf.

[7] Espressif Systems. *Installing Arduino ESP32*. Espressif Systems. URL: https://docs.espressif.com/projects/arduino-esp32/en/latest/installing.html.

[8] Yahoo Finance. *Alexa is now in 100 million homes*. 2022. URL: https://finance.yahoo.com/news/alexa-now-100-million-homes-183412222.html.

[9] Handson Technology. *SR501 Motion Sensor Datasheet*. 2024. URL: https://www.handsontec.com/dataspecs/SR501%20Motion%20Sensor.pdf.

[10] Roberto Ivo Kokan. "ESP-NOW communication protocol with ESP32". En: 2021. URL: https://eprints.uklo.edu.mk/id/eprint/6312/1/Paper%20Slovenia%20-%202021%20Chalenges%20of%20the%20future%20-%20Roberto%20Ivo%20Kokan.pdf.

[11] Oracle. *Internet of Things (IoT) | Oracle España*. 2024. URL: https://www.oracle.com/es/internet-of-things/.

[12] K. S. Rajasekaran et al. “Advanced Domestic Alarms with IoT”. En: *International Journal of Electronics and Communication Engineering & Technology (IJECET)* 7.5 (2016), págs. 90-97. URL: https://d1wqtxts1xzle7.cloudfront.net/50264255/IJECET_07_05_009-libre.pdf.

[13] Securitas Direct. *Calculadora de Alarmas para Hogar | Securitas Direct*. 2024. URL: https://www.securitasdirect.es/alarmas/calculadora/index.html?camp=seo_hogar&&lpType=site.

[14] Spring Team. *Spring Framework Reference Documentation*. 2024. URL: https://docs.spring.io/spring-framework/reference/index.html.

[15] Spring Team. *Spring Integration Reference - MQTT Support*. 2024. URL: https://docs.spring.io/spring-integration/reference/mqtt.html.

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA