



UNIVERSIDAD
DE MÁLAGA



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DE COMPUTADORES

**Disvise: Arquitectura distribuida para la gestión de
notificaciones basadas en el microcontrolador ESP32**

**Disvise: Distributed architecture for notification
management based on ESP32 microcontroller**

Realizado por
David Gómez Delgado

Tutorizado por
Alberto Gabriel Saldero Hidalgo

Departamento
Lenguaje y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2017

Fecha defensa: 7 de julio de 2017

Resumen

Realización de un proyecto basado en el microcontrolador ESP32 conectado vía WIFI con peticiones http a una API REST con el objetivo de conseguir un dispositivo portable de avisos para coordinación de un equipo en un espacio controlado, así como poder guardar un registro de todas las peticiones para futuros estudios de coordinación.

La aplicación sigue una arquitectura distribuida, contando con un servidor principal al que se conectan los distintos dispositivos de aviso. Para la comunicación se ha diseñado una API Rest que actuará como intermediaria entre el dispositivo generador del aviso, los dispositivos basados en el microcontrolador ESP32 y una base de datos, como canal de comunicación hemos optado por usar WiFi.

El dispositivo con microcontrolador ESP32 será la segunda pieza importante para la arquitectura distribuida, encargada de enviar y principalmente recibir peticiones, para ello necesitamos conexión WiFi para poder conectar a la API Rest, cosa que nos permite hacer el microcontrolador ESP32, como segunda función principal, el microcontrolador deberá mostrar en una pantalla la información que ha recibido de la API Rest.

Palabras clave:

ESP32, microcontrolador, API, base de datos, programa cargable.

Abstract

The project involves the development of a portable alert device designed for team coordination within a controlled environment. It is based on the ESP32 microcontroller, which communicates via Wi-Fi through HTTP requests to a RESTful API. The primary objective is to facilitate real-time alerts and maintain a comprehensive log of all requests for subsequent coordination analysis.

The application adopts a distributed architecture, comprising a central server to which multiple alert devices are connected. Communication is handled through a specifically designed REST API that serves as an intermediary between the alert-generating device, the ESP32-based microcontroller devices, and a database. Wi-Fi has been selected as the communication channel.

The ESP32 microcontroller serves as a crucial component within this distributed system. Its primary role is to receive, and occasionally send, requests to the REST API over a Wi-Fi connection, a capability inherently supported by the ESP32. Additionally, the microcontroller is tasked with displaying on-screen the information received from the API, thus fulfilling its secondary but essential function within the system architecture.

Keywords:

ESP32, microcontroller, API, data base, flashable program.

Índice

Resumen	1
Abstract	2
Índice	1
Introducción	1
1.1 Motivación	1
1.2 Objetivos y tecnologías a usar	4
1.3 Metodología	5
1.4 Estructura de la memoria	6
Fase de análisis y diseño	7
2.1 API REST	7
2.1.1 Requisitos funcionales	7
2.1.2 Requisitos no funcionales	8
2.1.3 Actores del Sistema	9
2.1.4 Diagrama de Casos de Uso	10
2.1.5 Diagrama de Clases	11
2.1.4 Diagrama de secuencia	14
2.1.5 Base de datos	15
2.2 Microcontrolador ESP32	16
2.2.1 Requisitos funcionales	16
2.2.2 Requisitos no funcionales	17
2.2.3 Actores del sistema	18
2.2.4 Diagrama de casos de uso	18
2.2.5 Diagrama de secuencia	19
2.2.6 Diagrama de estados	22
Implementación	25
3.1 API REST	25
3.2 Microcontrolador ESP32	26
Conclusión	29
4.1 Objetivos cumplidos	29
4.2 Líneas Futuras	30
Referencias	31
Manual de Instalación	33
Requerimientos	33
Constantes y variables a modificar	34

Manual de Usuario 37

1

Introducción

1.1 Motivación

Muchas empresas y organizaciones están organizadas en distintos puestos dependientes entre ellos para poder cumplir su función. Cada puesto, conformado por distintas personas, debe coordinarse en determinados momentos, lo que en ocasiones de cierta urgencia genera momentos de caos debido a los retrasos debidos a tener que hacer el aviso en persona o por un teléfono móvil. Algunas empresas emplean redes sociales para coordinar a los trabajadores. Esta solución, que está lejos de ser la adecuada, es inviable en la mayoría de los casos, pues en muchas organizaciones se tiene prohibido el uso de móviles en los puestos de trabajo.

En puestos como cajero y reponedor de supermercados los avisos suelen ser realizados a través de un micrófono, con una calidad de audio pésima. A este problema se le suman los casos en el que la persona avisada está ocupado con otra labor y no

puede avisar de vuelta para que avisen a otra persona. Otro problema generado es el tiempo perdido de tener que parar la labor del puesto de trabajo para coger el micrófono, dar el aviso y continuar trabajando.

En este trabajo pretendemos dar solución a este problema desarrollando un sistema de avisos basado en dispositivos desarrollados ad hoc. Estos dispositivos, basados en el microcontrolador ESP32, estarán conectados a la red de la organización de forma inalámbrica y notificarán a los trabajadores los avisos de forma individualizada. Los trabajadores podrán utilizar también el dispositivo para responder al aviso. Todos los avisos serán gestionados desde una aplicación que implementará un sistema de colas que tendrá en cuenta en todo momento la situación de los trabajadores en la organización.

Con un sistema embebido con el microcontrolador ESP32 podemos obtener mejores tiempos de reacción para acudir a los avisos, teniendo en cuenta también, como aspecto añadido, conseguimos mejoras en la protección de datos del trabajador y un ambiente más adecuado para personas con autismo e hipersensibilidad sensorial evitando el ruido alto que pueden llegar a generar los altavoces.

Otra ventaja añadida del uso del microcontrolador ESP32, es que es económico y se le pueden añadir fácilmente funciones de vibración, pantalla y más extensiones para poder personalizar a gusto del presupuesto disponible y funcionalidad requerida.

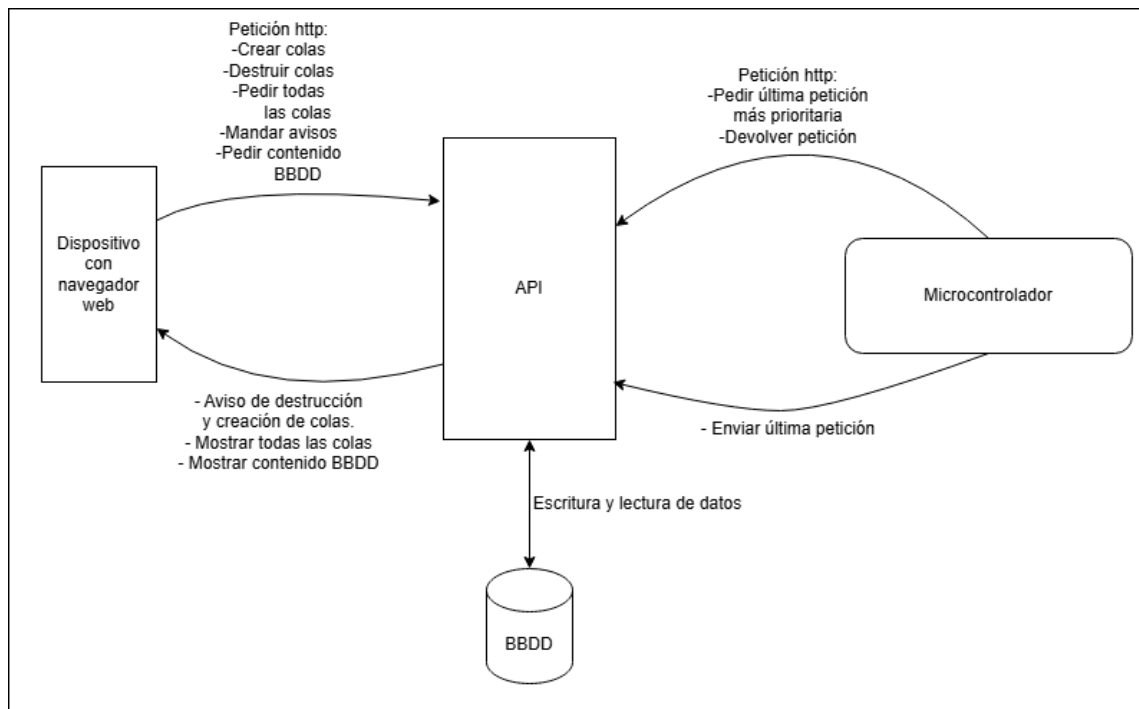


Figura 1.1 Diagrama de la idea de flujo del dispositivo.

1.2 Objetivos y tecnologías a usar

Para obtener el Sistema Distribuido necesitaremos diseñar varios elementos seguidamente enumerados:

1. La API REST, encargada de recibir peticiones URL desde un dispositivo móvil o sobremesa como un ordenador o un microcontrolador, así como enviar confirmación de haber realizado los procesos correctamente al dispositivo origen.
2. Gestionar las peticiones en colas con RabbitMQ ya que aseguran la entrega de los mensajes y proporciona prioridad entre mensajes, dos aspectos muy importantes para nuestro sistema (León, 2018).
3. Queremos conectar el microcontrolador ESP32 a la API REST como dispositivo principal que consumirá y mostrará los avisos almacenados en RabbitMQ, dado que su bajo consumo energético y conexión WI-FI sumado a su bajo costo, son características suficientes para elegir el uso de este dispositivo (Hercog, 2023).
4. La creación de una base de datos ayudará a tener una gestión de todas las peticiones realizadas, eligiendo como base de datos SQLite ya al ser una base de datos relacional podemos tener claves únicas según necesidades futuras.
5. Querremos una interfaz web con HTML, CSS y JavaScript, ya que, son actualmente lenguajes muy potentes y que evolucionan constantemente (Córcoles, 2020).

Para lograr los objetivos además usaremos tecnologías como:

1. PlantUML para la realización de los diagramas a lo largo de la memoria a partir de código.
2. Plataformio para la realización del código en el microcontrolador sobre Visual Studio Code y su posterior compilación y carga en el dispositivo.
3. Maven sobre IntelliJ Idea Community para poder lanzar la API REST y la web HTML.
4. RabbitMQ para la gestión de colas en la API REST.

1.3 Metodología

Usaremos la metodología Scrum, este se basa en la gestión de proyectos de manera eficiente a través de un trabajo ágil. Dado que Scrum aumenta la productividad, la calidad del producto y la satisfacción del cliente. Estudios demuestran un aumento de hasta el 66 % en productividad y una reducción del 30 % en el tiempo de desarrollo. Esto junto con el flujo de trabajo en el que se realizan ciclos iterativos para desarrollo incremental y la adaptación constante a las necesidades que van surgiendo, es una metodología muy fiable para implementar (Gil, 2025).

Lo primero a realizar será la planificación del producto, en nuestro caso el programa instalable en microcontroladores para la gestión de avisos en un sistema distribuido.

Dividiremos el trabajo en diez ciclos los cuales seguirán una estructura similar, primero planificación del ciclo, luego su ejecución y una vez realizado se revisará. Los diez ciclos son:

Orden	Tarea	Tiempo (horas)
1	Creación de una API REST encargada de la generación de notificaciones a partir de una URL.	40
2	Creación de la aplicación consumidora de notificaciones.	25
3	Implementación de las funciones a realizar en el dispositivo ESP32 con el uso de temporizadores y botones.	60
4	Implementación de la aplicación consumidora en el dispositivo ESP32.	30
5	Enlazar los avisos consumidos con el dispositivo ESP32 con las funciones implementadas.	25
6	Implementar generación de peticiones desde el dispositivo hacia la API.	25
7	Implementación de una base de datos que recoja todas las notificaciones generadas.	25
8	Unión de la base de datos con la API REST.	26

9	Creación de una interfaz gráfica para realizar las peticiones.	30
10	Refactorización del proyecto.	10
	Total	296

Para cumplir los 5 puntos principales se incluye un ciclo de retrospectiva, en el cual lo enfocaremos a las futuras mejoras que podría tener el proyecto (Gil, 2025).

1.4 Estructura de la memoria

La memoria estará dividida en 5 puntos principales:

1. La fase de análisis y diseño, encontrando diagramas sobre la estructura del proyecto y requisitos funcionales y no funcionales, destacando que se dividirá en dos subproyectos a analizar y diseñar, la API y el código del microcontrolador.
2. La implementación del código, se mostrarán las partes del código más relevante para el funcionamiento de la API y el programa cargable en el microcontrolador.
3. Una conclusión donde se hará una revisión total de objetivos cumplidos y una visión futura del proyecto.
4. Un manual de instalación con la explicación de qué tecnologías necesitas y qué parámetros hay que modificar según la necesidad del consumidor.
5. Un manual de usuario que muestra la explicación de cómo usar el sistema distribuido.

2

Fase de análisis y diseño

2.1 API REST

2.1.1 Requisitos funcionales

Enfocaremos primero los requisitos funcionales, ya que será lo que nos aporte la visión clara del software. (López, I. D.). Por cada requisito vamos a definir una prioridad según el momento en el que necesitemos tener en funcionamiento el requisito. Nuestros requisitos serán:

- **RF1:** Recibir peticiones desde dispositivos externos para su gestión interna como encolar, desencolar, almacena y leer de base de datos. Esta será el requisito más prioritario puesto que sin recibir las peticiones no se podrá gestionar nada.

- **RF2:** Encolar y desencolar peticiones en colas que usarán el microcontrolador y la web para producir y el microcontrolador para consumir las peticiones. Siendo un requisito de baja prioridad dado que depende de tener la API y el microcontrolador funcionando pero necesario para obtener una consumición de mensajes justa.
- **RF3:** Gestionar peticiones entre distintos microcontroladores, un microcontrolador puede devolver una petición a la API para mandárselo a otro microcontrolador para simular que el usuario del primer dispositivo está ocupado. Es un requisito de muy baja prioridad ya que es un añadido a la posible gestión de peticiones a poder realizar.
- **RF4:** Registrar todas las peticiones en una base de datos, para llevar un control exhaustivo de todas las peticiones realizadas. No influye para el comportamiento final base de la aplicación, pero sí queremos tener un recurso para ver un histórico de las peticiones, por tanto es de prioridad baja.
- **RF5:** Obtener registro de todas las peticiones realizadas, para obtener un registro de todas las peticiones principales para futuros estudios o controlar el buen uso del sistema. Al igual que el requisito funcional número tres, tampoco es prioritario
- **RF6:** Crear y destruir colas de peticiones, cada dispositivo conectado al sistema distribuido deberá tener su cola adjunta, así como al momento de desconectarlos ya no será necesaria que la cola consuma recursos del servidor, aunque igualmente se permite la persistencia de las colas hasta que no sean eliminadas manualmente. Como otros requisitos que definen la gestión de colas, este también tiene una prioridad baja.

2.1.2 Requisitos no funcionales

Definiremos los requisitos que describen cómo debe comportarse el sistema como la seguridad, requisitos que aportan el fácil uso de la aplicación y la escalabilidad (López, I. D.). Cada requisito no funcional tiene una prioridad que depende de los requisitos funcionales y el momento en el que necesitamos cada requisito, siendo menos prioritarios los que no son necesarios para el funcionamiento principal. Los requisitos no funcionales son:

- **RNF1:** Sistema de seguridad de recepción de peticiones, queremos tener una protección programada en la API, para saber que el mensaje llega de un usuario válido deberá incluir un número como parámetro a la hora de enviar un aviso a la API y este debe coincidir con el número almacenado en la API. Requisito de baja prioridad que incluiremos cuando el proyecto esté funcionando.
- **RNF2:** Base de datos en MySQLite para una base de datos ligera y gratuita. Requisito de prioridad baja pero alta entre las bajas ya que sin la base de datos no podemos gestionar las operaciones.
- **RNF3:** Serializar mensajes con JSON para que sean mensajes ligeros al tener que enviarlos y leerlo en un microcontrolador.(Crockford, sf). Requisito de muy alta prioridad para poder comprobar lo antes posible que los mensajes se pueden consumir desde el microcontrolador.
- **RNF4:** Gestión de colas usando RabbitMQ, programa que me permite encolar según prioridad y en formato FIFO sin preocuparme de la lógica interna (León, 2018). Prioridad media baja, sin la aplicación no podremos gestionar las colas.
- **RNF5:** Spring Boot como framework base por su popularidad y soporte, siendo de los framework más potentes actuales para el desarrollo de API. Requisito más prioritario dado que es la base donde construiremos la API.
- **RNF6:** Encolar según prioridad y en formato FIFO, obteniendo un orden de mensajes justos en tiempo cuando se reciben y en prioridad. Este requisito tiene prioridad baja ya que necesitaremos tener en funcionamiento RabbitMQ.

2.1.3 Actores del Sistema

Estos son los elementos que interactuarán entre ellos para el funcionamiento de la aplicación y que se muestran en la Figura 2.1:

- **Usuario administrador:** encargado de crear y destruir colas, además mandará las peticiones para que sea encoladas. Finalmente podrá pedir un listado de todas las peticiones almacenadas en la base de datos.
- **Microcontroladores:** consumidores de peticiones así como podrán enviar el mismo mensaje consumido de vuelta al siguiente microcontrolador.

- **Base de datos SQLite:** almacenará cada petición realizada tanto por el administrador como por el dispositivo.

Aunque no sea un actor al ser la propia API la que contacta con RabbitMQ, la mencionamos también ya que es la aplicación encargada de gestionar todas las peticiones con prioridad y formato FIFO internamente junto con la API. (León, 2018).

2.1.4 Diagrama de Casos de Uso

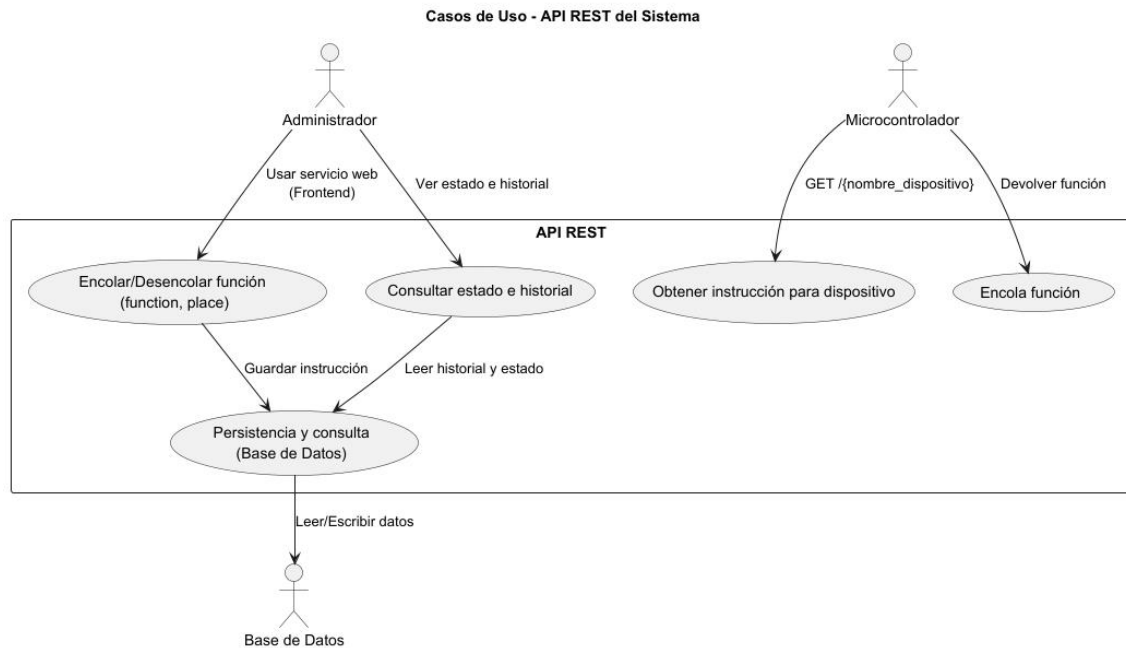


Figura 2.1 Diagrama de casos de uso API

En la Figura 2.1 podemos ver como la API interactúa con los actores del sistema mencionados anteriormente y la función que deben realizar, cada función está mostrado en los requisitos funcionales y no funcionales.

El caso más común será:

1. El administrador mandará a la API una petición HTTP en el que envía un aviso, la API internamente registra la petición en la base de datos.
2. El microcontrolador pide obtener la instrucción a la API, este la desencola y se la devuelve el microcontrolador.

2.1.5 Diagrama de Clases

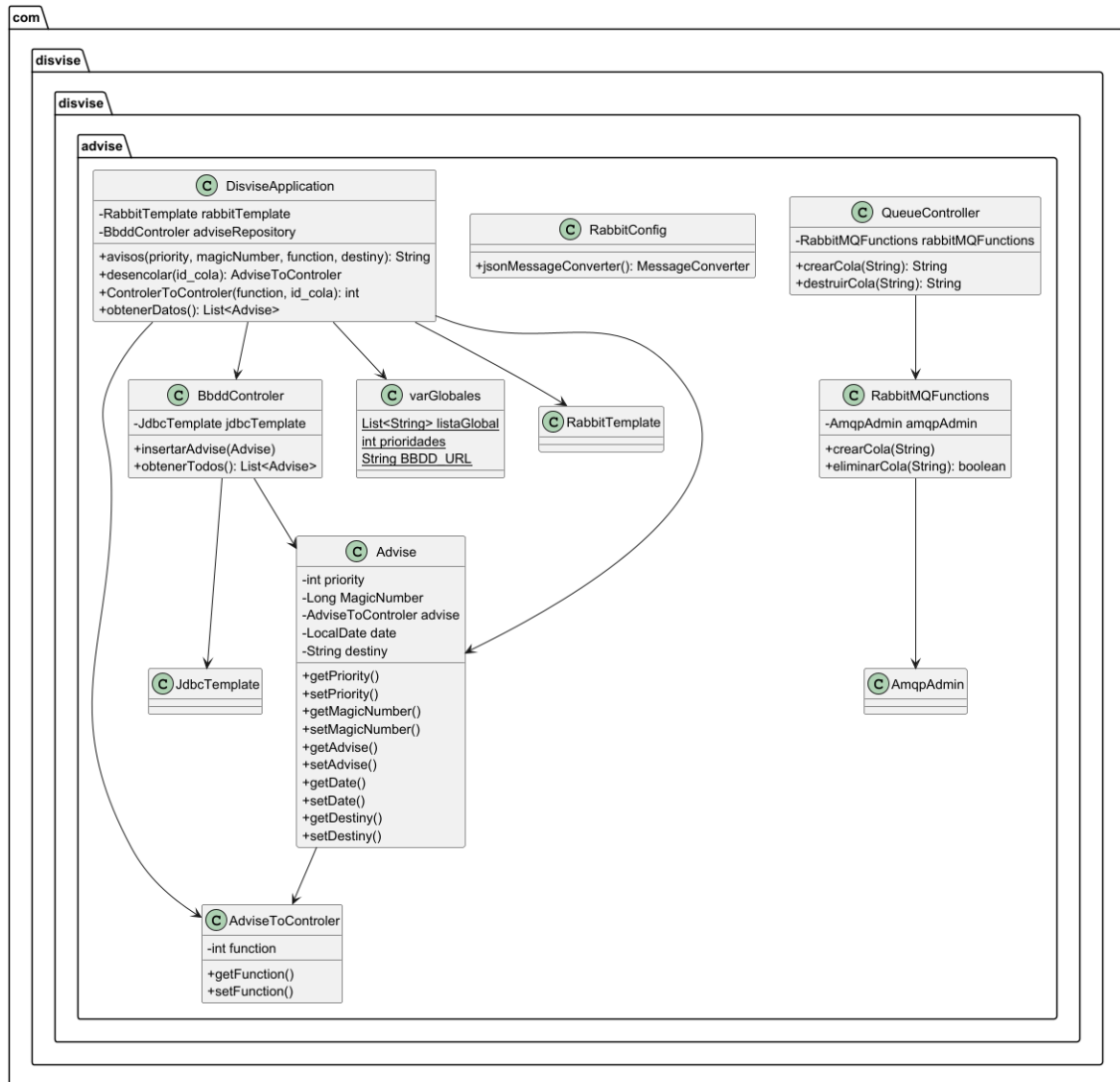


Figura 2.2 Diagrama de clases API

En la Figura 2.2 encontramos múltiples clases que en su conjunto conforman todo el comportamiento de la API, entre las que encontramos:

1. **DisviseApplication**, La clase principal de la API donde encontramos:
 - Función avisos, con endpoint `@GetMapping(path = "{priority}/{magicNumber}/{function}/{destiny}")`, encargada de revisar que el aviso es correcto y encolarla, para encolar necesitará la las funciones de la librería RabbitTemplate. Además hace uso de la clase BbddControler tanto para insertar la petición actual que acaba de recibir como obtener todos los datos.
 - Función desencolar, con endpoint `@GetMapping(path = "/api/{idCola}")`, haciendo uso de la librería RabbitTemplate buscará el aviso en las colas creadas en RabbitMQ a partir del parámetro "idCola".

- Función `ControlerToControler`, con endpoint `@GetMapping(path = "{function}/{id_cola}")` recibe un aviso y el id del microcontrolador origen, para poder enviar el aviso al siguiente microcontrolador libre.

-Función `obtenerDatos`, con endpoint `@GetMapping("/datos")`, hará uso de la clase `BbddControler` para acceder a la base de datos y obtener el listado de todas las peticiones realizadas.

2. **BbddControler**, haciendo uso de la clase `JdbcTemplate` y por tanto, de la librería `JdbcTemplate` que se encarga de la lógica de la base datos, en este caso tanto escribir usando `insertarAdvise` como leer con `obtenerTodos`.
3. **Advise**, clase que define la estructura del aviso así como las funciones para obtener y modificar cada parámetro del objeto.
4. **AdviseToControler**, clase que define el objeto que se envía al microcontrolador, pudiendo acotar en este los parámetros que queramos enviar, en este caso solo usamos el parámetro `function`.
5. **VarGlobales**, es una clase donde se indicarán las constantes y variables globales que pueda usar la aplicación.
6. **RabbitConfig** será la clase encargada de hacer la conversión del aviso a JSON.
7. **QueueControler** es la clase que contiene los dos últimos endpoints que se llamarán a partir de `@RequestMapping("/colas")`, estos dos endpoints son `@GetMapping("/crear")` que pide como parámetro un identificador para poder crear las colas con ese nombre y `@GetMapping("/destruir/{id}")` para eliminar colas según el identificador. Para su correcto funcionamiento, llama a las funciones de la clase `RabbitMQFunctions`.

8. **RabbitMQFunctions** es una clase con las funciones crearCola y eliminarCola, haciendo uso de la clase amqpAdmin para poder acceder a las funciones de RabbitMQ internas que crean y destruyen colas.

2.1.4 Diagrama de secuencia

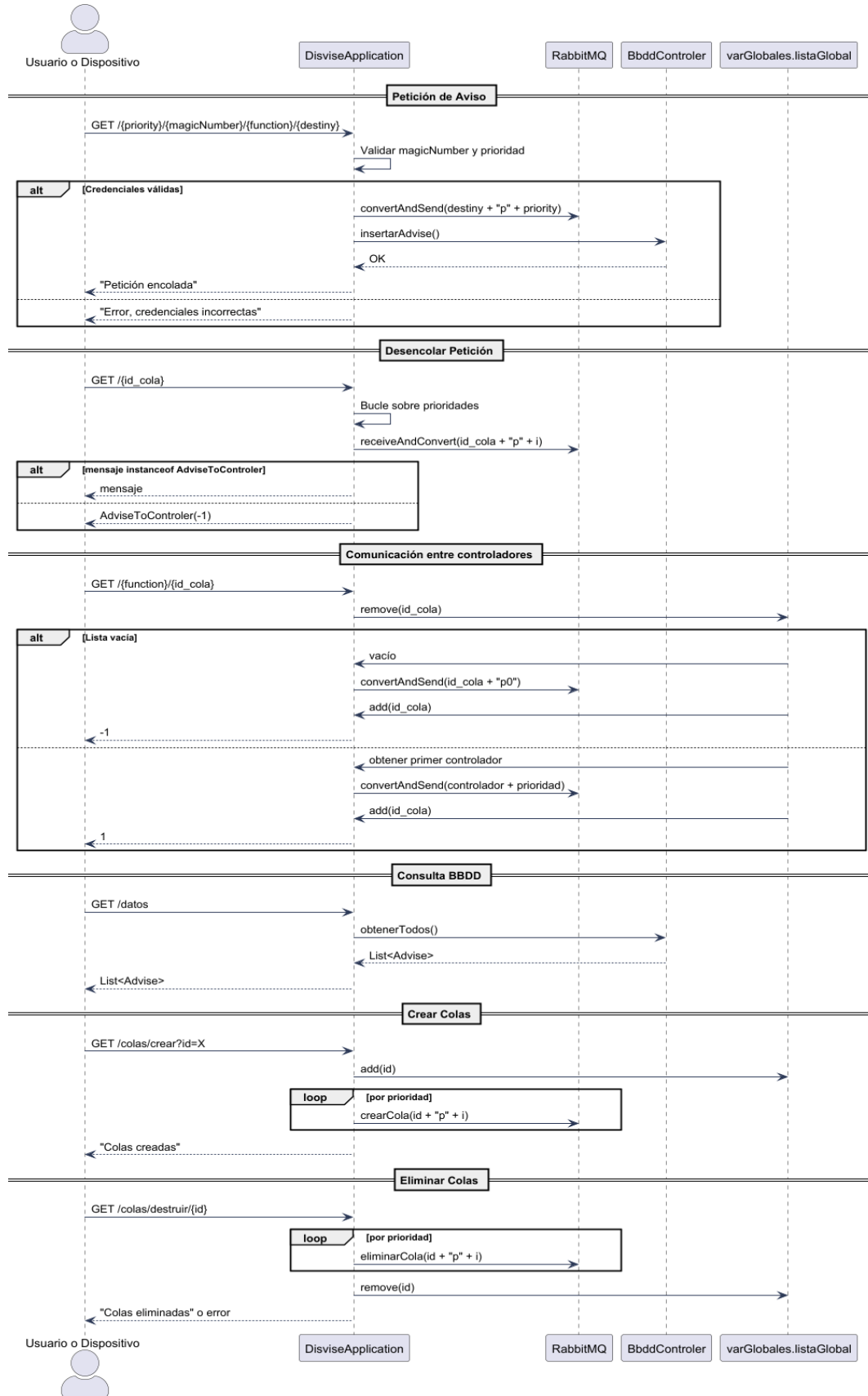


Figura 2.3 Diagrama de secuencia API

Con el diagrama de secuencia de la Figura 2.3 vemos más detalladamente el flujo de la API.

Para mandar un aviso se realiza primero una consulta http, la API internamente validará que el mensaje contiene un número en concreto a modo de seguridad, tras eso se comunicará con RabbitMQ para encolar el aviso y se conectará a la base de datos para insertar el elemento.

Por otro lado tenemos la función de desencolar petición es la función principal del microcontrolador que con el identificador que tienen el microcontrolador pedirá a la API el elemento, este a su vez lo consultará a RabbitMQ para recibir el objeto en JSON y a su vez mandar el objeto al microcontrolador.

Los microcontroladores también se pueden comunicar a través de otra consulta a la API, un microcontrolador mandará un objeto en JSON con un aviso y el identificador del microcontrolador de origen, la API se encargará de gestionar esta petición buscando el siguiente dispositivo libre y encolando la petición en RabbitMQ en su respectiva cola.

La API podrá acceder a la base de datos a través de una función de la librería JDBC.

Finalmente a través de funciones internas de la librería de RabbitMQ al realizar una consulta de crear más el identificador o destruir más el identificador, la API se conectará a RabbitMQ proporcionándole la información y este se comunicará con RabbitMQ para que se realice la operación.

2.1.5 Base de datos

La base de datos contendrá un registro de cada aviso realizado, por tanto tendrá que almacenar cada dato de la estructura. Al usar una base de datos relacional necesitaremos el uso de una clave única, sin embargo, al no necesitar realmente de clave única lo dejaremos como el número incremental como el número de elemento de entrada.

Siguiendo finalmente la estructura:

```
CREATE TABLE advise (  
  id INTEGER PRIMARY KEY AUTOINCREMENT,  
  priority INTEGER NOT NULL,  
  magic_number INTEGER,  
  advise STRING NOT NULL,
```

```
date TEXT,  
destiny TEXT  
);
```

Por cada zona a gestionar se deberá crear una tabla, con la intención de obtener un estudio de cada zona.

2.2 Microcontrolador ESP32

Tenemos un microcontrolador ESP-32 implementado en el dispositivo LilyGo T-Display S3, con el que se han realizado proyectos como el control inteligente de una casa, estudio en vivo del mercado de las cripto monedas conectado a una API y actualizando los precios por el dispositivo o monitorizar un huerto mandando alertas cuando las plantas necesitan atención (Zafeiriou, 2025). Con estos ejemplos presentes, el LilyGo T-Display S3 es un dispositivo idóneo para la consumición y producción de avisos a una API.

En la figura 2.4 se muestra el esquema del contenido final que debe aparecer en la pantalla del dispositivo.

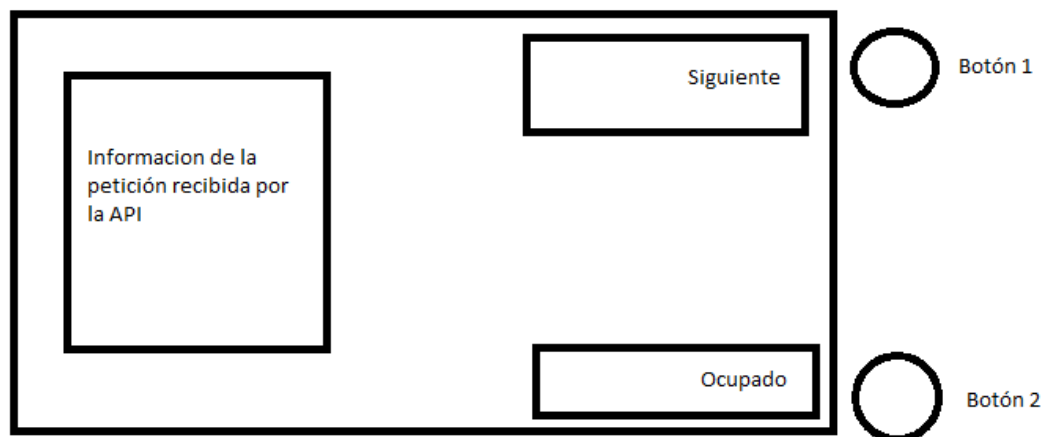


Figura 2.4 Esquema resultado del microcontrolador al consumir un aviso

2.2.1 Requisitos funcionales

Nuestros requisitos que necesitaremos para que el microcontrolador funcione mostrando información como la Figura 2.4 son:

- **RF1:** Conectar a una red WIFI, necesario para poder hacer peticiones a la API. Teniendo prioridad alta para el momento en el que el dispositivo ya pueda mostrar información.
- **RF2:** Mostrar información en la pantalla del dispositivo según la función recibida, con un ejemplo parecido a la Figura 2.4. Este requisito será de prioridad muy alta, antes de continuar con otros requisitos habrá que ver bien un funcionamiento básico del dispositivo.
- **RF3:** El dispositivo pedirá peticiones a la API cada diez segundos aproximadamente, cinco segundos mostrando la pantalla y 5 segundos extra estando suspendido. La prioridad será media alta, en cuanto el dispositivo pueda mostrar información y la API pueda dar avisos cuando se le pida, deberá pedir peticiones automáticamente constantemente.
- **RF4:** Uno de los botones debe pedir la siguiente petición. Con prioridad baja, añada velocidad a la consumición de peticiones.
- **RF5:** Uno de los botones debe poner el dispositivo en modo suspensión y reenviar el mensaje de vuelta a la API para la consumición de otro microcontrolador. El requisito será de prioridad baja, siendo una funcionalidad extra necesaria cuando el dispositivo funcione en su totalidad.

2.2.2 Requisitos no funcionales

Los siguientes requisitos definirán los comportamientos que ayudan a la facilidad de uso del programa y la escalabilidad nuevamente:

- **RNF1:** El dispositivo entrará en un modo de suspensión profunda para ahorrar batería, ya que es un dispositivo muy pequeño y la batería debe durar sesiones largas de mínimo media jornada laboral. Este requisito es de prioridad media alta ya que varias funciones del microcontrolador usan este funcionamiento, pero no es requisito funcional dado que con dispositivos con buena batería es innecesario este modo de suspensión.
- **RNF2:** El dispositivo esperará un tiempo prudencial a que se pulse algún botón, haciendo que el usuario tenga tiempo suficiente de reacción para ver el aviso. Con muy baja prioridad, este requisito solo se necesitará cuando los botones estén en funcionamiento.

- Tolerancia a errores WIFI o HTTP para que el dispositivo no se quede en bucle esperando una respuesta inexistente. Requisito de prioridad media, dado que queremos un control correcto durante las pruebas del dispositivo y poder centrarnos en el resto de requisitos.
- **RNF4:** Consultar a una API REST datos en JSON, ya que son mensajes muy livianos para gestionar sin problema en el microcontrolador. Es un requisito importante pero de prioridad media dado que será necesario en el estado en el que el dispositivo pueda conectarse a la API y la API pueda gestionar peticiones de manera básica.

2.2.3 Actores del sistema

Los elementos que interactúan entre sí en la Figura 2.5 para el funcionamiento del microcontrolador son:

- **Usuario:** es la persona física que usa el dispositivo, sus funciones serán inicializar el sistema y pulsar los botones para realizar los requisitos funcionales mencionados.
- **Servidor API:** Encargado de mandar y recibir avisos al microcontrolador vía WiFi por consultas http.
- **Sistema ESP32:** Gestionará toda la lógica interna del microcontrolador, como el estado en el que se encuentra, el manejo de las operaciones e interrupciones de los botones, entrar en estado de suspensión profunda, mostrar la función por pantalla como en la Figura 2.4 y mandar y recibir peticiones.

2.2.4 Diagrama de casos de uso

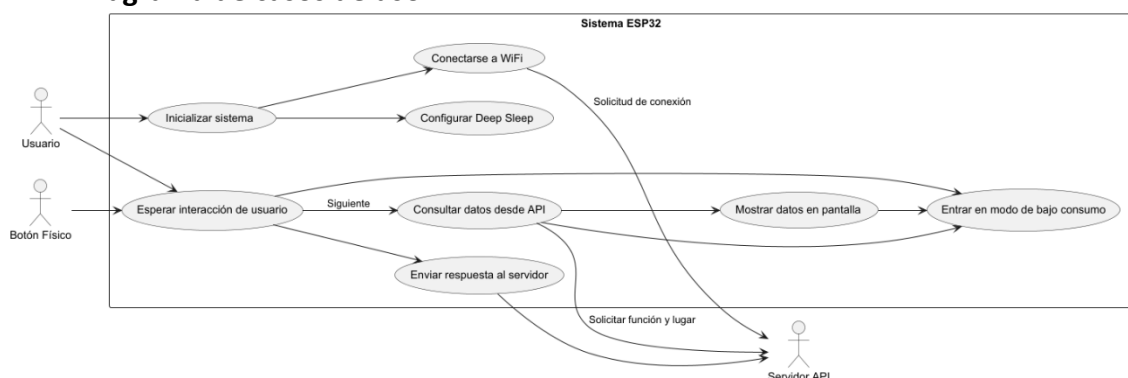


Figura 2.5 Diagrama de casos de uso microcontrolador

La Figura 2.5 muestra como los distintos actores del sistema realizan las funciones definidos en los requisitos funcionales.

El comportamiento básico es:

1. El usuario inicializa el dispositivo.
2. El dispositivo una vez es inicializado consulta a la API si existe una petición.
3. Si existe mostrará los datos por pantalla
4. El programa entrará a los 5 segundos en modo suspensión profunda.

2.2.5 Diagrama de secuencia

Diagrama de secuencia del inicializador del microcontrolador:

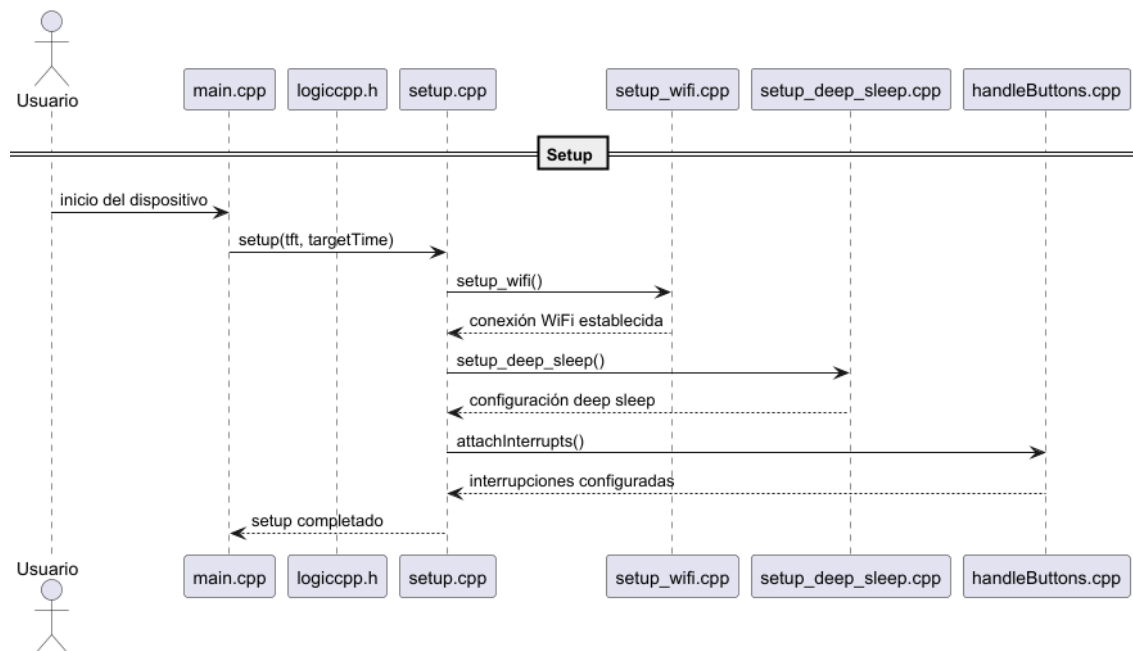


Figura 2.6 Diagrama de secuencia inicialización microcontrolador

Diagrama de secuencia para el bucle principal:

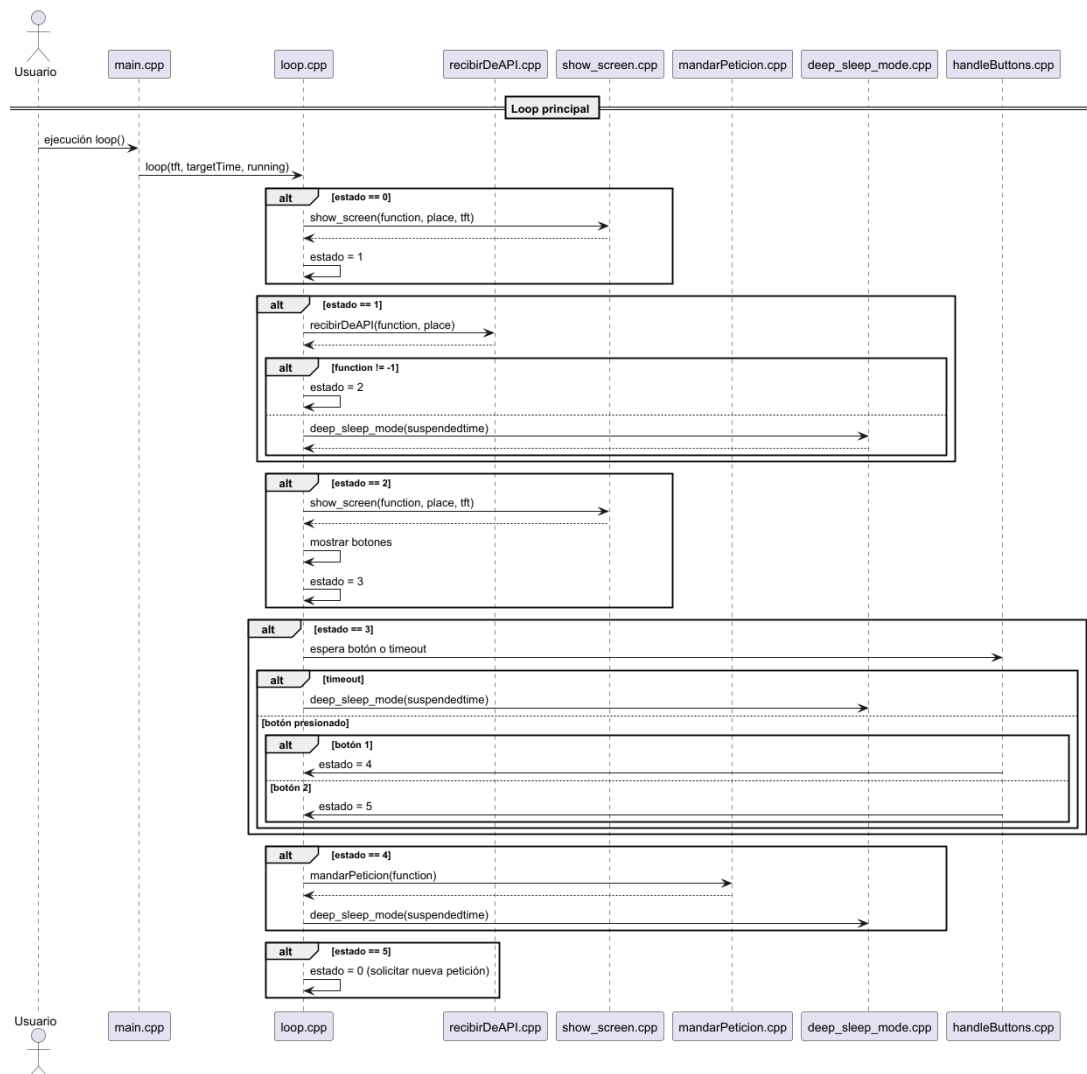


Figura 2.7 Diagrama de secuencia bucle principal microcontrolador

Aunque el programa del microcontrolador no esté definido con clases, la Figura 2.6 y 2.7 nos da una representación visual de cómo las funciones interactúan entre ellas dado que por cada fichero tenemos una única función encargada de la acción.

En el momento en el que el programa esté cargado en el microcontrolador iniciará la secuencia de la Figura 2.6, este iniciará la pantalla, el wifi y las variables necesarias. Inmediatamente después de terminar la configuración, mostrará un mensaje por pantalla de "Recibiendo petición" en un fondo negro. Continuando con el

bucle del programa pasamos al estado 1, haremos una llamada a la API para recibir un objeto en JSON, este objeto habrá que deserializarlo, según el número de la función se almacenará en una variable el nombre del lugar que posteriormente se mostrará en pantalla con la información que nos proporcione la función añadiendo que se mostrará ahora una indicación en cada botón según la función que desempeñen.

Inmediatamente de mostrar la información se realiza una espera, esperando a que pase un tiempo acotado o se pulse un botón.

Según el botón pulsado, uno de ellos será el responsable de reenviar la función a la API y desconectar el dispositivo más tiempo que el normal y el otro botón se encargará de volver a empezar el bucle sin necesidad de parar el dispositivo.

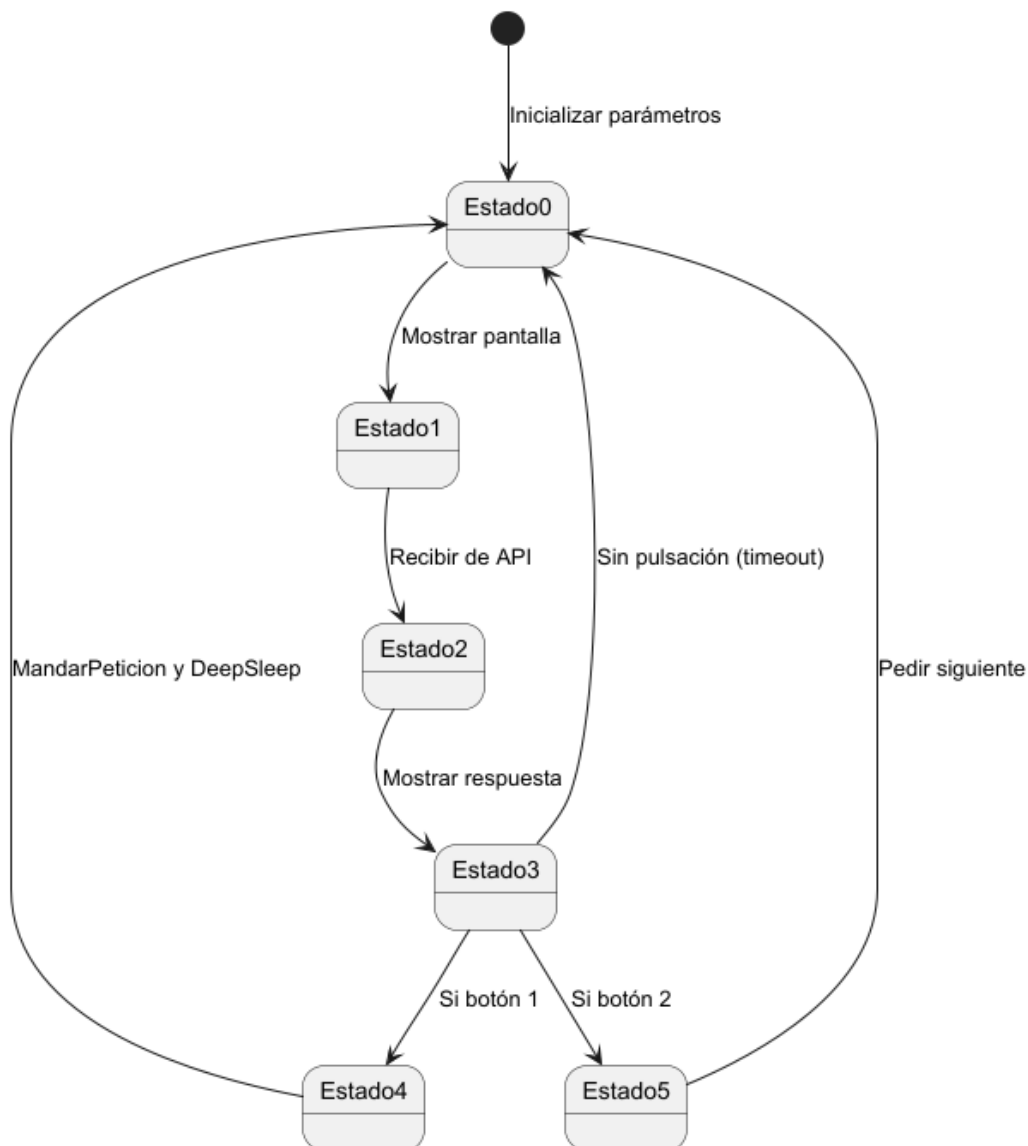


Figura 2.8 Diagrama de estados microcontrolador

Con la Figura 2.8 podemos estudiar el flujo del bucle principal del microcontrolador, lo que nos permite una fácil programación con el uso de switch case.

Una vez todos los parámetros estén inicializados se procederá a entrar al bucle en el siguiente orden:

- Estado 0: Mostrar por pantalla el mensaje "Recibiendo petición...", haciendo uso de la pantalla para mostrar al usuario que se va a proceder al acceso a la API para recibir la petición.

- Estado 1: Recibir de la API la petición y deserializar el mensaje que el microcontrolador usará para obtener los datos que quiere mostrar por la pantalla.
- Estado 2: Mostrar por pantalla las funciones de cada botón y la petición recibida en el Estado 1.
- Estado 3: En este momento el dispositivo se queda en espera unos segundos esperando un evento, los cuales son, finalizar un tiempo de 5 segundos para volver a suspenderse con el fin de ahorrar la batería del dispositivo, pulsar el botón de siguiente que te dará la siguiente petición o pulsar el botón de ocupado para devolver la petición a la api y suspender el dispositivo más tiempo.
- Estado 4: Se accede cuando se ha pulsado el botón de ocupado, por tanto, se tendrá que enviar a la API la petición recibida y suspenderemos más tiempo el dispositivo antes de pedir la siguiente función.
- Estado 5: Significa que hemos pulsado el botón de siguiente y podremos acceder más rápido a la siguiente petición volviendo al estado 0.

La suspensión que se realiza es una suspensión profunda, para así poder ahorrar más batería, cuando se realiza esta suspensión el sistema tiene que volver a inicializar todos los parámetros que no han sido guardados en la memoria caché y comenzará de nuevo el bucle.

3

Implementación

3.1 API REST

En la API encontramos cuatro funciones principales que definen el comportamiento de la Figura 2.3, dos de ellas encargadas de darle a RabbitMQ peticiones, una de origen del microcontrolador y otra de una petición http, una tercera función encargada de las consumiciones de peticiones del microcontrolador y finalmente la encargada de devolver como JSON la información de la base de datos.

Explicaremos la función principal de la recepción de la petición donde podremos observar varios conceptos principales así como algunas decisiones importantes como la gestión de prioridades en las colas.

```

@GetMapping(path =("/{priority}/{magicNumber}/{function}/{destiny}") //mapeamos para que esta funci
public String avisos(@PathVariable int priority, @PathVariable Long magicNumber,
                    @PathVariable int function, @PathVariable String destiny){
    if(magicNumber == varGlobales.magicNumber && priority < varGlobales.prioridades &&
        varGlobales.listaGlobal.contains(destiny) && priority <= varGlobales.prioridades
        && priority >= 0){
        AdviseToControler adviseToSend = new AdviseToControler(function);
        String newDestiny = destiny + "p" + priority;
        rabbitTemplate.convertAndSend(newDestiny, adviseToSend); //Mensaje en JSON

        //Insertar en la bbdd
        Advise advise = new Advise(priority, magicNumber, adviseToSend, LocalDate.now(), destiny);
        adviseRepository.insertarAdvise(advise);
        //Crear peticion para el microcontrolador y encolar
        return "Numero correcto, peticion encolada en " + destiny;
    }else{
        return "Error, credenciales incorrectas";
    }
}
}

```

Figura 3.1 Función encargada de encolar avisos.

En la figura 3.1 aparecen que se pasan 4 parámetros a la función, la prioridad, un denominado número mágico, el número de la función y el destino. El primer parámetro que tomaremos es el número mágico, un número puesto como medida de seguridad para asegurar que solo se procesarán avisos que conozcan ese valor. Seguidamente se encolará el aviso que, aunque Manuel León mencione en su artículo que es posible que RabbitMQ gestione colas con distintas prioridades, para poder tener mayor control sobre la generación de colas nosotros tendremos nuestras propias colas definidas por un identificador. El aviso encolado será del tipo "AdviseToControler", una clase definida por si en el futuro se requiere pasar más parámetros, en nuestro caso será solo uno, el número de la función. Finalmente se almacenará el aviso con todos los parámetros necesarios en la base de datos.

3.2 Microcontrolador ESP32

Nuestro programa cargado en el microcontrolador se centra en un bucle principal que sigue la estructura de la Figura 2.7, dentro del bucle encontramos tres funciones principales, una función encargada de recibir los datos de la API, otra que reenvía los datos a la API y para terminar la función que muestra el contenido por pantalla.

Mostraremos un fragmento de código importante para en el futuro poder implementar un aviso que tenga varios parámetros y no solo uno como en nuestro caso el parámetro de la función a realizar.

```
// Parsear con ArduinoJson
const size_t capacity = JSON_ARRAY_SIZE(1) + JSON_OBJECT_SIZE(1) + 16;
DynamicJsonDocument doc(capacity);
DeserializationError error = deserializeJson(doc, payload);

//A futuro solo necesitare place y function
if (!error && doc.size() > 0) {
    //Para implementar json con varios valores
    //JsonObject advise = doc[0];
    //function = doc["function"];
    function = doc["function"];
    Serial.println("Primer aviso:");
    Serial.printf("Function: %d", function);
    if(function == 25){
        strcpy(place, "Caja Registradora");
    }
} else {
    Serial.println("Error al parsear JSON o array vacío");
}
} else {
    function = -1;
    Serial.printf("Error HTTP: %d\n", httpCode);
}
```

Figura 3.2 Código encargado de deserializar el contenido JSON recibido de la API

La Figura 3.2 muestra la deserialización del aviso obtenido de la API, para poder obtener más de un parámetro será necesario declarar una capacidad suficiente para almacenar el aviso, obtenerlo y gestionarlo como un array de datos, si por ejemplo queremos mostrar el valor función y el valor de la prioridad se accederá al array por los nombres "function" y "priority" en vez de recorrer el array elemento a elemento. Actualmente con un elemento es suficiente para poder realizar toda la función del microcontrolador y al no tener el dispositivo mucha memoria es mejor realizar una o varias peticiones pequeñas que una muy grande.

4

Conclusión

4.1 Objetivos cumplidos

Para conseguir el sistema distribuido necesitábamos tener varios dispositivos conectados entre sí, en nuestro sistema está compuesto por la API, el microcontrolador y el dispositivo con acceso al navegador web.

La API consigue encolar y desencolar peticiones en colas con prioridad usando RabbitMQ, las peticiones llegan correctamente por peticiones http y accede a la base de datos con lecturas y escrituras correctas.

El microcontrolador se conecta a la red WiFi, realiza peticiones http y recibe el objeto serializado en JSON, deserializa el objeto bien para realizar su función mostrando la información por pantalla, ambos botones funcionan y cumplen su función.

La base de datos almacena y devuelve todas las peticiones realizadas de origen distinto al microcontrolador.

Obtenemos un sistema distribuido aplicable a múltiples ámbitos como supermercados, hospitales, sector hostelería y cualquier ámbito que necesite notificar a una persona rápidamente, pudiendo ser muy determinante en situaciones críticas.

4.2 Líneas Futuras

El proyecto contiene posibles mejoras como que pueda soportar más tipo de funciones, considerando la opción de almacenar todas las funciones en un árbol para poder realizar búsquedas más rápidas para mejorar la reacción del dispositivo. Otra posible mejora es querer la gestión de prioridades también gestionada por RabbitMQ, confiando en el funcionamiento interno de esta aplicación. Para poder tener más seguridad que un solo número que comprueba al inicio, se podrá implementar el uso de ACL en red, sin embargo, esto no está reflejado en la memoria porque es una implementación que debe ser realizada según la empresa que haga uso de este sistema distribuido.

Referencias

León, M. (2018). *RabbitMQ, qué es y cuáles son sus funcionalidades. Funcionalidades de RabbitMQ.*

Hercog, D., Lerher, T., Truntič, M., & Težak, O. (2023). *Design and implementation of ESP32-based IoT devices. Sensors.*

López, I. D. (s.f.). *Requerimientos funcionales y no funcionales: La guía completa. Byspel.*

SQLite. (2025). *Documentación SQLite. Recuperado de <https://sqlite.org/docs.html>*

Córcoles Briongos, C. (2020). *¿Qué aprender para ser expert@ en desarrollo web? Blog de la UOC.*

Gil, C. (2025). *Qué es la metodología Scrum y cómo aplicarla en 5 fases. Blog de Sortlist.*

Spring Framework. (2025). *Manual JdbcTemplate. Recuperado de <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org.springframework.jdbc.core/JdbcTemplate.html>*

PlatformIO. (2025). *Manual PlatformIO. Recuperado de <https://docs.platformio.org/en/latest/what-is-platformio.html>*

Arduino. (2025). *Manual Arduino. Recuperado de <https://docs.arduino.cc/>*

Crockford, D. (s.f.). *JSON: JavaScript Object Notation.* Recuperado de <https://www.json.org/json-es.html>

Spring. (2025). *Spring Boot documentation.* Recuperado en junio de 2025, de <https://docs.spring.io/spring-boot/redirect.html>

Zafeiriou, S. (2025). *9 Exciting LilyGo T-Display S3 ESP32 Projects to Try in 2025.*

Gómez Delgado, David. (2025). *Disvise: Arquitectura distribuida para la gestión de notificaciones basadas en el microcontrolador ESP32.* Recuperado de <https://github.com/DavidGomezDelgado/Disvise>

Apéndice A

Manual de Instalación

Requerimientos

El sistema distribuido cuenta con varios componentes hardware y software que se conectarán entre ellos, siendo estos:

- Un servidor que corra la API REST usando maven, que se comunicará con la base de datos, el navegador web, el microcontrolador y RabbitMQ y esta a su vez realizará la comunicación intermedia entre cada componente.
- Un servidor dedicado a la base de datos o el mismo servidor de la API, este solo tendrá comunicación directa con la API.
- Un navegador web para realizar peticiones http y ver la interfaz gráfica, otro elemento con solo conexión directa con la API.
- Un microcontrolador con pantalla y dos botones como el Lilygo T-Display S3, otro elemento que solo se comunica con la API
- Un compilador de C++ y el programa platformIO para la compilación del programa cargable en el microcontrolador, con conexión única a la API.
- Un servidor de RabbitMQ lanzado de manera local o usando Docker o Kubernetes, también tendrá comunicación únicamente con la API.

Constantes y variables a modificar

```
const char* ssid = "";  
const char* password = "";
```

Figura A.1 Parámetros configurables para conexión WIFI

```
const char* api_url = "";
```

Figura A.2 Parámetro configurable dirección url de la API

```
const char* nombre_dispositivo = "display0";
```

Figura A.3 Parámetro nombre dispositivo microcontrolador

En el programa cargable del microcontrolador tendremos que modificar varios parámetros, siendo estos, los parámetros de conexión WiFi, el nombre de la red y la contraseña vistos en la Figura 6.1, la pantalla se tendrá que configurar comentando la línea `"#include <User_Setup.h>"` y descomentando la línea `"#include <User_Setups/Setup206_LilyGo_T_Display_S3.h>"` en el fichero `"User_Setup_Select.h"`, en caso de ser distinta se deberá tener en cuenta y configurar la pantalla deseada, y configurar el parámetro de la dirección de la API terminado en `"/"` para asegurar cuando se complete la dirección como podemos ver en la Figura 3.12 que se genera bien la dirección. Esta dirección deberá contener una dirección IP y el puerto si no se cuenta con un servidor DNS.

Para saber qué nombre tiene el dispositivo, se deberá modificar el parámetro de la Figura 6.3, para que en cuando se fuese a crear una cola, este nombre debe coincidir con el identificador que usamos en la función de la Figura 3.1.

```
public static String BBDD_URL = ".jdbc:sqlite:"; no usages
```

Figura A.5 Parámetro configurable dirección url de la base de datos

En la API se deberá modificar el valor que almacena la dirección física de la base de datos iniciando con `".jdbc:sqlite"` para que pueda ubicar la base de datos y realizar las consultas SQL.

```
public static int prioridades = 2;
```

Figura A.6 Constante del número máximo de prioridades

Se podrá modificar el nivel máximo de prioridad modificando la constante de la figura A.6, del fichero varGlobales.java marcando de ese modo cuántas colas de prioridad tendrá cada dispositivo.

```
public static long magicNumber = 123456;
```

Figura A.7 Constante del número de seguridad

Se encuentra en el fichero varGlobales.java en el proyecto de la API y se podrá modificar según el número que se requiera como parámetro de seguridad, para no recibir peticiones no deseables.

Apéndice B

Manual de Usuario

Se mostrará cada funcionalidad de la aplicación web, mostrando en las figuras siguientes el comportamiento que tiene la web al realizar una función así como su comportamiento interno al comunicarse la API con RabbitMQ, además se mostrará un resultado de lo que muestra el microcontrolador en una iteración completa.

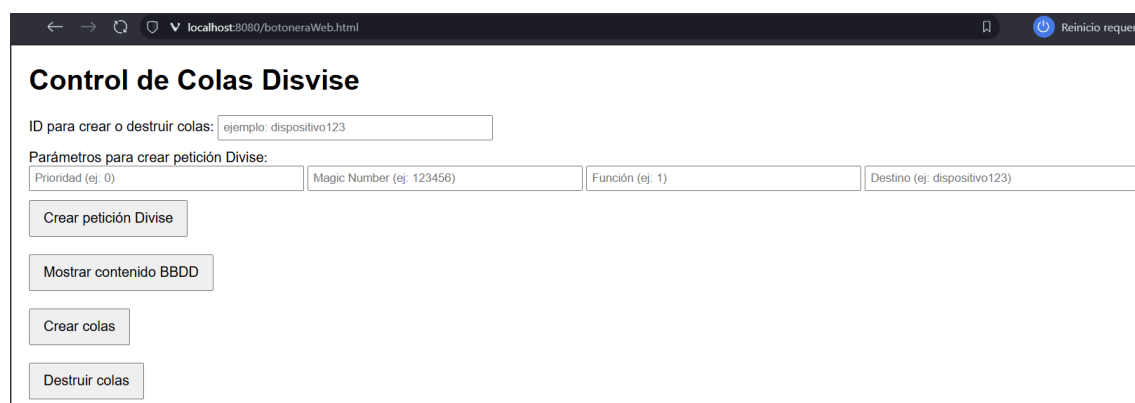


Figura B.1 Interfaz web base

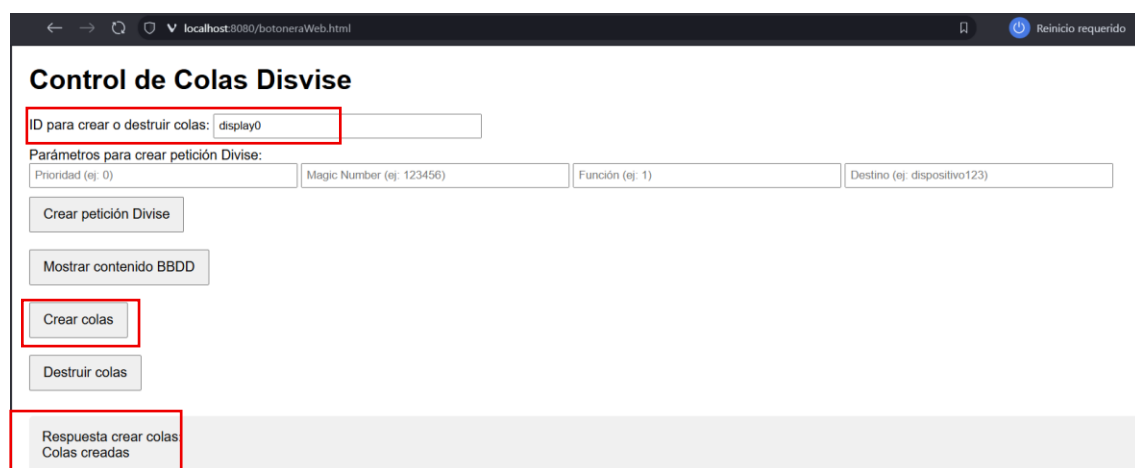


Figura B.2 Creación de la cola

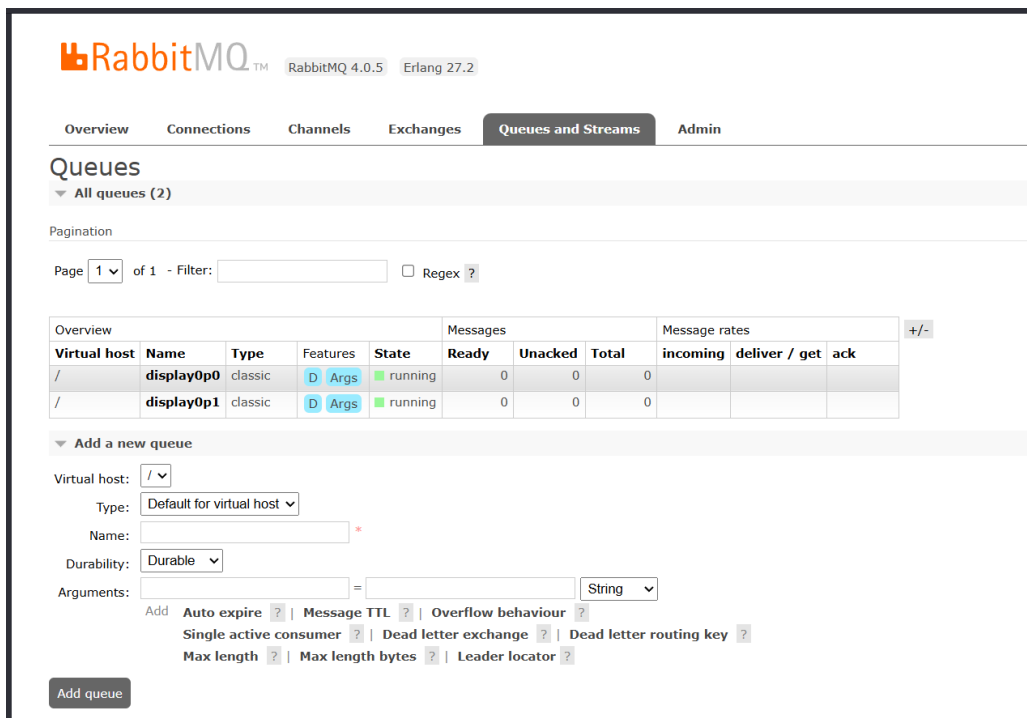


Figura B.3 Colas creadas en RabbitMQ

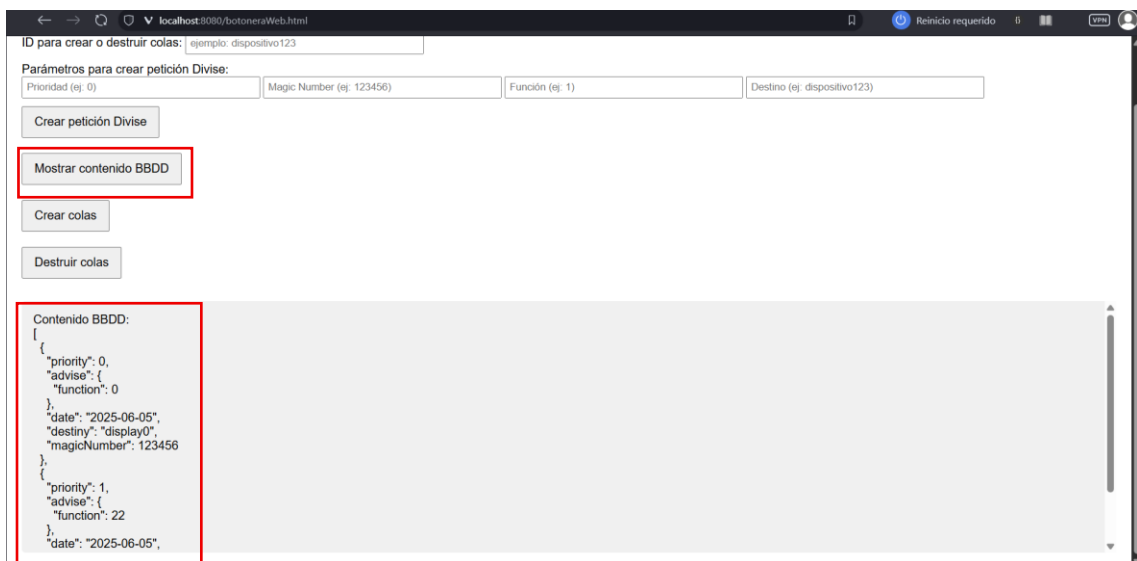


Figura B.4 Ejemplo mostrar base de datos

La Figura B.1 muestra la interfaz gráfica de la web, pudiendo hacer las cuatro operaciones principales, crear petición, mostrar el contenido de la base de datos, destruir colas y crear colas en RabbitMQ.

Para que el dispositivo pueda recibir peticiones, el parámetro introducido en "ID para crear o destruir colas" debe ser el mismo que en la Figura A.3. Al crear una cola se muestra en RabbitMQ las colas creadas siendo p0 la más prioritaria y p1 la menos prioritaria.

Con la opción "Mostrar Contenido BBDD" se podrá ver el contenido de toda la base datos.

Control de Colas Disvise

ID para crear o destruir colas:

Parámetros para crear petición Divise:

<input type="text" value="0"/>	<input type="text" value="123456"/>	<input type="text" value="25"/>	<input type="text" value="display0"/>
--------------------------------	-------------------------------------	---------------------------------	---------------------------------------

Respuesta crear petición Divise:
Numero correcto, peticion encolada en display0

Figura B.5 Encolar petición correcta

RabbitMQ ™ RabbitMQ 4.0.5 Erlang 27.2

Overview Connections Channels Exchanges **Queues and Streams** Admin

Queues

▼ All queues (2)

Pagination

Page of 1 - Filter: ☐ Regex ?

Overview					Messages			Message rates			+/-
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
/	display0p0	classic	D Args	■ running	1	0	1	0.00/s			
/	display0p1	classic	D Args	■ running	0	0	0				

▼ Add a new queue

Virtual host:

Type:

Name:

Durability:

Arguments: =

Add [Auto expire ?](#) | [Message TTL ?](#) | [Overflow behaviour ?](#) | [Single active consumer ?](#) | [Dead letter exchange ?](#) | [Dead letter routing key ?](#) | [Max length ?](#) | [Max length bytes ?](#) | [Leader locator ?](#)

Figura B.6 Petición encolada en RabbitMQ

Una vez se genere una petición, esta se encola automáticamente según su prioridad, no se encolará en caso de no acertar el número mágico, poner un número de prioridad

inválido o poner un nombre de dispositivo cuya cola no ha sido previamente creada, estas condiciones se puede ven en la Figura 3.1.

Control de Colas Disvise

ID para crear o destruir colas:

Parámetros para crear petición Divise:

<input type="text" value="0"/>	<input type="text" value="123456"/>	<input type="text" value="25"/>	<input type="text" value="display0"/>
--------------------------------	-------------------------------------	---------------------------------	---------------------------------------

Respuesta destruir colas:
Colas eliminadas

Figura B.7 Cola eliminada

RabbitMQ 4.0.5 Erlang 27.2

Overview Connections Channels Exchanges **Queues and Streams** Admin

Queues

▼ All queues (0)

Pagination

Page of 0 - Filter: ☐ Regex ?

... no queues ...

▼ Add a new queue

Virtual host:

Type:

Name: *

Durability:

Arguments: =

Add | | | | | | | |

Figura B.8 Cola eliminada en RabbitMQ

Se podrá eliminar colas para dispositivos que no estén en funcionamiento, eliminando también todas las peticiones que quedasen pendientes.

Microcontrolador Esp32



Figura B.9 Dispositivo mientras recibe información



Figura B.10 Dispositivo mostrando información

Durante la Figura B.8 el dispositivo está recogiendo el dato de la API y deserializándolo para finalmente mostrar por pantalla el resultado, este mismo bucle se repite variando ligeramente el final, si se pulsa "Siguiete", a los pocos segundos consumirá el siguiente aviso, si se pulsa en "Ocupado" el dispositivo devuelve la petición a RabbitMQ y se queda en suspensión más tiempo, si no se pulsa nada, el dispositivo se suspenderá unos 5 segundos y volverá a realizar la siguiente petición.