Los microservicios en el EDGE

Los **microservicios en el Edge** se refieren a pequeñas aplicaciones o servicios independientes que se ejecutan directamente en los dispositivos Edge, como los microcontroladores ESP32-Wroom. Estos microservicios están diseñados para realizar tareas específicas de procesamiento de datos, toma de decisiones y control de actuadores localmente, sin necesidad de depender de la comunicación constante con la nube.

Características Clave de los Microservicios en el Edge:

1. **Independencia y Descentralización:**
    - Cada microservicio se encarga de una función específica (por ejemplo, recolección de datos de un sensor, filtrado de datos anómalos, activación de un sistema de riego).
    - Estos servicios operan de manera autónoma, lo que significa que pueden continuar funcionando incluso si la conexión a la nube se pierde o es intermitente.
2. **Procesamiento en Tiempo Real:**
    - Dado que los microservicios se ejecutan en el Edge, pueden procesar datos en tiempo real. Esto es crucial para aplicaciones como la automatización del riego en un cultivo, donde las decisiones deben tomarse inmediatamente según las condiciones ambientales actuales.
3. **Filtrado y Normalización de Datos:**
    - Los microservicios pueden incluir lógicas de filtrado para eliminar lecturas de sensores que sean anómalas o no confiables.
    - La normalización de datos también se realiza en el Edge, transformando los datos crudos en un formato estandarizado antes de ser procesados o transmitidos a otras partes del sistema.
4. **Reducción de Latencia:**
    - Al procesar los datos localmente, los microservicios en el Edge reducen la latencia que normalmente se asocia con el envío de datos a la nube para su procesamiento.
    - Esto es especialmente útil en aplicaciones críticas donde la rapidez en la toma de decisiones es vital.
5. **Optimización de Recursos:**
    - Estos microservicios están diseñados para ser ligeros y eficientes, optimizando el uso de recursos limitados en

dispositivos Edge, como la memoria, el procesamiento y la energía.

- Además, pueden implementar estrategias para ahorrar energía, como activar sensores solo en momentos críticos o reducir la frecuencia de recolección de datos durante periodos de estabilidad.

Ejemplos de Microservicios en el Edge para el Proyecto de Vivero Automático:

1. **Microservicio de Recolección de Datos:**
   - Este servicio se encarga de leer continuamente los valores de los sensores conectados al ESP32-Wroom, como los de humedad del suelo, temperatura y luz. Los datos recolectados son luego filtrados para eliminar lecturas que puedan ser causadas por errores momentáneos.
2. **Microservicio de Control de Riego:**
   - Basado en los datos del sensor de humedad del suelo, este microservicio decide cuándo activar o desactivar el sistema de riego. Por ejemplo, si la humedad cae por debajo de un umbral predefinido, el microservicio activará la bomba de agua para regar las plantas.
3. **Microservicio de Control de Luz:**
   - Este servicio se encarga de gestionar la iluminación en el vivero. Si el sensor de luz detecta que la intensidad lumínica es baja, el microservicio encenderá las luces LED para asegurar que las plantas reciban la cantidad de luz necesaria para su crecimiento.
4. **Microservicio de Control de Ventilación:**
   - Este microservicio monitorea la temperatura y humedad del aire, y activa un ventilador si la temperatura es demasiado alta, ayudando a mantener un ambiente óptimo para las plantas.

Beneficios de Utilizar Microservicios en el Edge:

- **Resiliencia:** Al no depender constantemente de la nube, el sistema puede continuar funcionando eficazmente incluso durante fallos de conectividad.
- **Escalabilidad:** Los microservicios pueden ser desarrollados, desplegados y actualizados de manera independiente, lo que facilita la expansión y mejora del sistema.

- **Eficiencia:** Al procesar datos localmente y solo enviar los datos procesados o relevantes a la nube, se reduce el consumo de ancho de banda y se optimiza el uso de recursos.

Entonces, los microservicios en el Edge permiten un procesamiento local eficiente, reducen la dependencia de la nube y mejoran la capacidad de respuesta del sistema en tiempo real.

Librerias, Shields y Microservicios.

Las librerías proporcionadas para los shields y módulos que se conectan a los microcontroladores, como el ESP32-Wroom, facilitan la creación de microservicios al ofrecer funciones y métodos predefinidos para interactuar con los sensores, actuadores y otros componentes del sistema.

¿Cómo Proporcionan las Librerías los Recursos para Crear Microservicios?

1. **Abstracción de Hardware:**
    - Las librerías ofrecen una abstracción del hardware, lo que significa que no necesitas preocuparte por los detalles de bajo nivel de cómo funciona un sensor o actuador. Por ejemplo, una librería para un sensor de humedad del suelo podría incluir métodos como readHumidity() o getMoistureLevel(), que devuelven directamente los datos que necesitas.
2. **Interfaz de Comunicación Simplificada:**
    - Las librerías facilitan la comunicación con los dispositivos conectados a través de interfaces como I2C, SPI o UART. Esto permite a los microservicios acceder y gestionar los datos de los sensores sin necesidad de programar complejas rutinas de comunicación.
    - Por ejemplo, una librería para un sensor de temperatura que utiliza I2C podría proporcionarte métodos que te permiten leer la temperatura simplemente llamando a una función, sin preocuparte por cómo se envían y reciben los bits.
3. **Funciones de Procesamiento Incorporadas:**
    - Algunas librerías incluyen funciones de procesamiento básico, como el filtrado de datos, promedio de lecturas, o incluso detección de valores anómalos. Esto facilita la implementación de lógicas de filtrado y normalización dentro de los microservicios en el Edge.
4. **Control de Actuadores:**

- Librerías para controladores de actuadores, como motores o bombas, permiten a los microservicios activar o desactivar estos componentes de manera sencilla. Por ejemplo, una librería para controlar una bomba de agua podría incluir métodos como turnOnPump() y turnOffPump(), que el microservicio de riego puede utilizar según sea necesario.

5. **Gestión de Energía:**
   - Algunas librerías proporcionan funciones para gestionar el consumo de energía, como la posibilidad de poner sensores o el propio microcontrolador en modos de bajo consumo cuando no están en uso, lo que es esencial para dispositivos Edge que operan en entornos con recursos limitados.
   -

Ejemplo: "ESP32Servo.h"

La **ESP32 Servo Library** es una biblioteca diseñada para controlar servomotores utilizando el microcontrolador ESP32, replicando fielmente la funcionalidad de la clásica biblioteca Servo de Arduino. Esta librería se adapta a las características específicas del ESP32, como el manejo avanzado de los temporizadores PWM (Pulse Width Modulation), que permiten controlar la posición del servomotor ajustando el ancho del pulso.

Principales Funciones de la Biblioteca:

1. **attach(pin):**
   - Asocia un pin GPIO del ESP32 con un canal PWM disponible para controlar un servomotor. Los pines recomendados para esta operación son 2, 4, 12-19, 21-23, 25-27, 32-33.
   - Devuelve el número del canal asignado o 0 en caso de fallo.
2. **attach(pin, min, max):**
   - Similar a la función attach estándar, pero permite especificar los valores mínimos y máximos en microsegundos para el pulso PWM que controla el servo.
   - Los valores de min y max están forzados a un rango mínimo de 500us y un máximo de 2500us.
3. **write(angle):**
   - Establece el ángulo del servo en grados (0 a 180). También acepta valores en microsegundos, dependiendo del rango proporcionado en attach.
   - Los valores fuera del rango permitido son ajustados automáticamente a los límites superiores o inferiores.

4. **writeMicroseconds(us):**
    - Establece directamente el ancho del pulso en microsegundos. Esta función permite un control más preciso del servo, especialmente útil en aplicaciones que requieren posicionamiento exacto.
5. **read():**
    - Obtiene el último valor de ángulo que fue establecido para el servo.
6. **readMicroseconds():**
    - Recupera el último valor del ancho del pulso en microsegundos que fue escrito al servo.
7. **attached():**
    - Indica si un servo está actualmente conectado a un pin.
8. **detach():**
    - Desconecta el servo del pin asignado, liberando el canal PWM para su reutilización.

Funciones Específicas del ESP32:

1. **setTimerWidth(value):**
    - Permite ajustar el ancho del temporizador PWM, dentro de un rango de 16 a 20 bits. Esto es exclusivo del ESP32 y permite cambiar la resolución del temporizador, afectando la precisión con la que se puede controlar el servo.
2. **readTimerWidth():**
    - Recupera el ancho actual del temporizador PWM.

Valores por Defecto:

- **Ancho de pulso mínimo por defecto:** 544us
- **Ancho de pulso máximo por defecto:** 2400us
- **Ancho de temporizador por defecto:** 16 bits
- **Número máximo de servos:** 16 (equivalente al número de canales PWM disponibles en el ESP32)

Conclusión:

La **ESP32 Servo Library** es una herramienta poderosa que permite a los desarrolladores controlar servomotores con alta precisión y flexibilidad en el ESP32. Al aprovechar las capacidades avanzadas de PWM del ESP32, como la personalización del ancho de temporizador, esta

biblioteca proporciona un control más fino y adaptable para una variedad de aplicaciones y microservicios.

¿Qué es un Servo?

Un **servo** o **servomotor** es un dispositivo electromecánico que convierte señales eléctricas en movimiento mecánico controlado. A diferencia de un motor eléctrico estándar, un servo está diseñado para moverse a una posición específica y mantenerla, lo que lo hace ideal para aplicaciones que requieren control preciso de posición, velocidad y aceleración.

Componentes de un Servo:

Un servomotor generalmente está compuesto por tres partes principales:

1. **Motor de CC (Corriente Continua):**
    - Es el componente que proporciona la fuerza o torque necesario para mover el eje del servo.
2. **Sistema de Engranajes:**
    - Este sistema reduce la velocidad del motor de CC y aumenta el torque, permitiendo que el servo mueva cargas más pesadas con mayor precisión.
3. **Circuito de Control:**
    - Es un pequeño circuito electrónico integrado en el servo que recibe señales eléctricas (PWM) y determina cuánto debe moverse el motor para alcanzar la posición deseada.
4. **Potenciómetro:**
    - El potenciómetro está conectado al eje del servo y proporciona retroalimentación al circuito de control sobre la posición actual del eje. Esto permite al servo ajustar su posición de manera precisa.

¿Cómo Funciona un Servo?

El funcionamiento de un servo se basa en la modulación por ancho de pulso (PWM), una técnica en la que la duración del pulso determina la posición del eje del servo.

Proceso de Operación:

1. **Señal de Control (PWM):**
    - Un servo se controla mediante una señal PWM, que consiste en una serie de pulsos eléctricos. La duración del pulso, medida en microsegundos, determina la posición del eje del servo.

- En la mayoría de los servos, un pulso de 1.5 ms (1500 microsegundos) coloca el eje en la posición media (90°). Un pulso más corto (~1 ms) mueve el eje hacia un extremo (0°), y un pulso más largo (~2 ms) lo mueve hacia el otro extremo (180°).

2. **Retroalimentación:**
   - El potenciómetro en el servo mide la posición actual del eje y la compara con la posición deseada según la señal PWM recibida.
   - Si la posición actual no coincide con la deseada, el circuito de control ajusta el motor de CC para mover el eje a la posición correcta.
3. **Movimiento Preciso:**
   - El sistema de engranajes ayuda a que el motor mueva el eje con precisión y suficiente torque para mantener la posición incluso bajo carga.
4. **Mantenimiento de la Posición:**
   - Una vez que el eje alcanza la posición deseada, el circuito de control sigue monitoreando la señal PWM y ajusta el motor según sea necesario para mantener la posición, corrigiendo cualquier desalineación causada por fuerzas externas.

Aplicaciones Comunes de los Servos:

- **Robótica:** Control de articulaciones, brazos robóticos, y mecanismos de movimiento.
- **Modelismo:** Control de la dirección y aceleración en coches, aviones y barcos a escala.
- **Automatización:** Control de válvulas, mecanismos de apertura/cierre, y otros sistemas que requieren movimientos precisos.
- **Electrónica de Consumo:** Cámaras de vigilancia (control de pan y tilt), sistemas de automatización del hogar, etc.

Ventajas de los Servos:

- **Precisión:** Los servos pueden moverse a posiciones específicas y mantenerlas con alta precisión.
- **Torque:** Gracias a los sistemas de engranajes, los servos pueden proporcionar un alto torque relativo a su tamaño.

- **Retroalimentación:** Los servos reciben retroalimentación continua de su posición, lo que les permite corregir errores en tiempo real.

Ejemplo Visual de Funcionamiento:

Imagina que un servo está conectado a la rueda delantera de un coche a control remoto. Cuando el usuario gira el volante en el control remoto, se envía una señal PWM al servo. Según la duración del pulso, el servo ajusta la rueda a un ángulo específico, permitiendo que el coche gire en la dirección deseada. Si la rueda es empujada fuera de su posición por una fuerza externa, el servo corregirá automáticamente la posición para mantener el ángulo correcto.

En resumen, un servo es un componente necesario en sistemas que requieren control preciso de posición y es muy utilizado en robótica y automatización, entre otras áreas.

Framework A y la POO

El lenguaje de programación utilizado en Arduino se basa en C/C++, un lenguaje de programación de propósito general ampliamente utilizado en sistemas embebidos y microcontroladores debido a su eficiencia y control a bajo nivel. Aunque no es estrictamente un lenguaje independiente, el entorno de desarrollo de Arduino proporciona una serie de bibliotecas y funciones que facilitan la programación de microcontroladores, encapsulando muchos detalles complejos y permitiendo que los desarrolladores se centren en las funcionalidades específicas de sus proyectos.

Lenguaje C en Arduino:

El lenguaje C es conocido por su eficiencia y control a bajo nivel, lo que lo hace ideal para programar microcontroladores como los utilizados en Arduino. En el contexto de Arduino, el lenguaje C se utiliza para realizar operaciones como:

- **Control de Hardware:**
    - Manipulación directa de los pines de entrada/salida (I/O) del microcontrolador.
    - Control de temporizadores, interrupciones, y otros periféricos del microcontrolador.
- **Manejo de Memoria:**
    - Uso eficiente de la memoria RAM limitada disponible en los microcontroladores.
    - Control manual sobre la asignación y liberación de memoria, lo que es crítico en sistemas embebidos.

- **Interacción con Sensores y Actuadores:**
    - Lectura de datos de sensores, procesamiento y toma de decisiones en tiempo real.
    - Control de motores, LEDs, pantallas y otros dispositivos externos conectados al microcontrolador.

Programación Orientada a Objetos (POO) en Arduino:

Aunque el lenguaje de Arduino está basado en C/C++, que soporta tanto la programación estructurada como la programación orientada a objetos (POO), el enfoque en Arduino suele ser más estructurado y procedimental. Sin embargo, es posible y a veces ventajoso usar la POO en proyectos de Arduino, especialmente cuando se trabaja en proyectos más complejos que requieren modularidad, reutilización de código y una mejor organización.

Conceptos Clave de la POO en Arduino:

1. **Clases y Objetos:**
    - **Clase:** Es una plantilla o modelo a partir del cual se crean objetos. Una clase define atributos (variables) y métodos (funciones) que describen el comportamiento y las propiedades del objeto.
    - **Objeto:** Es una instancia de una clase. Los objetos representan entidades del mundo real en el código y se utilizan para interactuar con otros objetos y funciones.

Ejemplo de Clase en Arduino:

```
class Led {
private:
    int pin;  // Atributo: el pin al que está conectado el LED

public:
    // Constructor: Inicializa el pin
    Led(int p) {
        pin = p;
        pinMode(pin, OUTPUT);  // Configura el pin como salida
    }

    // Método: Encender el LED
    void on() {
        digitalWrite(pin, HIGH);
    }

    // Método: Apagar el LED
    void off() {
        digitalWrite(pin, LOW);
```

```
    }
};

// Crear un objeto Led conectado al pin 13
Led myLed(13);

void setup() {
    // Encender el LED al inicio
    myLed.on();
}

void loop() {
    // Alternar el estado del LED cada segundo
    myLed.on();
    delay(1000);
    myLed.off();
    delay(1000);
}
```

En este ejemplo:

- Led es una clase que encapsula el comportamiento de un LED.
- myLed es un objeto de la clase Led que controla un LED conectado al pin 13.
- Los métodos on() y off() encapsulan la lógica para encender y apagar el LED, proporcionando una interfaz clara y fácil de usar.

2. **Encapsulamiento:**

- El encapsulamiento se refiere a la ocultación de los detalles internos de una clase, exponiendo solo una interfaz pública. Esto se logra utilizando modificadores de acceso como private y public.
- En el ejemplo anterior, el atributo pin es privado (private), lo que significa que no puede ser accedido directamente desde fuera de la clase. Los métodos on() y off() son públicos (public), lo que permite que sean llamados desde fuera de la clase.

3. **Herencia:**

- La herencia permite crear nuevas clases basadas en clases existentes, reutilizando y extendiendo su funcionalidad. Esto es útil en proyectos grandes donde varias clases comparten propiedades o comportamientos comunes.
- Ejemplo: Una clase RGBLed podría heredar de Led y agregar funcionalidades adicionales para controlar el color de un LED RGB.

4. **Polimorfismo:**
    - El polimorfismo permite que una clase hija o derivada redefina los métodos de la clase base. Esto es útil cuando diferentes clases necesitan comportamientos específicos pero comparten una interfaz común.
    - Aunque el polimorfismo se usa menos en proyectos Arduino debido a la simplicidad de los sistemas embebidos, es un concepto fundamental en POO.

Ventajas de Usar POO en Arduino:

- **Modularidad:** Facilita la organización del código en módulos independientes, lo que mejora la legibilidad y el mantenimiento.
- **Reutilización de Código:** Las clases pueden ser reutilizadas en diferentes partes del proyecto o en otros proyectos, lo que reduce la duplicación de código.
- **Facilidad de Expansión:** Es más fácil agregar nuevas funcionalidades sin afectar el código existente, gracias a la herencia y el polimorfismo.
- **Abstracción:** La POO permite trabajar en un nivel de abstracción más alto, enfocándose en los objetos y sus interacciones en lugar de en los detalles de implementación.

Consideraciones al Usar POO en Arduino:

- **Limitaciones de Recursos:** Los microcontroladores Arduino tienen recursos limitados (memoria, procesamiento), por lo que es importante usar la POO de manera eficiente para no agotar estos recursos.
- **Simplicidad vs. Complejidad:** Para proyectos pequeños o simples, la programación estructurada puede ser más adecuada. La POO se recomienda para proyectos más complejos donde la modularidad y la organización del código son críticas.

Algunos puntos adicionales

1. Microservicios en el Contexto de IoT y Edge Computing:

- **Despliegue y Actualización de Microservicios:**
    - En un entorno IoT, los microservicios en el Edge deben ser fácilmente desplegables y actualizables. Esto es crítico para mantener la funcionalidad del sistema a medida que se agregan nuevas características o se corrigen errores. En el contexto de Arduino, esto podría implicar la carga de nuevo

firmware en los microcontroladores o la integración con plataformas de gestión de dispositivos que permiten actualizaciones over-the-air (OTA).

- **Comunicación entre Microservicios:**
  - Aunque en Arduino los microservicios suelen ser bastante simples y autónomos, en sistemas más avanzados pueden necesitar comunicarse entre ellos. Esto puede lograrse mediante buses de datos, protocolos de comunicación serial, o incluso mediante redes inalámbricas locales (como WiFi o LoRa) si los dispositivos están físicamente separados.

2. Shields y su Importancia en la Expansión de Funcionalidades:

- **Modularidad y Expansión del Hardware:**
  - Los shields permiten una expansión modular del hardware en un proyecto Arduino, lo que facilita la adición de nuevas funcionalidades sin la necesidad de rediseñar el circuito base. Esta modularidad es clave para el prototipado rápido, donde se puede probar y ajustar diferentes configuraciones de hardware.
- **Interoperabilidad y Compatibilidad:**
  - Al utilizar shields, es importante considerar la interoperabilidad entre diferentes componentes. Algunos shields pueden compartir pines, lo que podría causar conflictos si no se gestionan adecuadamente. Utilizar shields con capacidad de configuración flexible (por ejemplo, para seleccionar diferentes pines) puede ser crucial en proyectos complejos.

3. Framework Arduino y Ecosistema de Desarrollo:

- **Ecosistema de Librerías y Comunidad:**
  - Arduino no solo es un entorno de desarrollo (IDE) y un conjunto de herramientas, sino también un ecosistema rico en librerías desarrolladas por la comunidad. Estas librerías, muchas de ellas disponibles en el Administrador de Librerías de Arduino, permiten extender las capacidades de los microcontroladores de manera rápida y efectiva.
  - La comunidad de Arduino también es un recurso invaluable para aprender, resolver problemas y compartir proyectos. Participar en foros, grupos y otros canales comunitarios puede ser de gran ayuda para desarrollar proyectos más complejos.
- **Plataformas y Herramientas Avanzadas:**

- Además de la IDE de Arduino, existen otras herramientas y entornos que pueden integrarse con Arduino, como PlatformIO, que ofrece características avanzadas como la integración con control de versiones, soporte para múltiples plataformas, y herramientas de depuración más robustas.

4. Programación Orientada a Objetos (POO) en Sistemas Embebidos:

- **POO y Gestión de Recursos:**
  - En sistemas embebidos como los que se desarrollan con Arduino, la POO debe usarse con consideración de las limitaciones de recursos. Es importante evitar la creación excesiva de objetos, el uso intensivo de memoria dinámica, y otros patrones de diseño que puedan consumir recursos innecesariamente.
  - Técnicas como el uso de static para variables y métodos dentro de una clase, o la reutilización de objetos existentes, pueden ayudar a gestionar mejor los recursos en un entorno de POO.
- **POO y Modularidad en Proyectos Complejos:**
  - A medida que los proyectos crecen en complejidad, la POO permite una mayor modularidad y claridad en el código. Por ejemplo, en un sistema IoT basado en Arduino, podrías tener clases para cada tipo de sensor, actuador, y lógica de control, lo que facilita la adición de nuevas funcionalidades y el mantenimiento del código a largo plazo.

5. Interacción entre Microservicios y POO en Arduino:

- **Microservicios Basados en Clases:**
  - En proyectos complejos, cada microservicio puede implementarse como una clase en C++, encapsulando no solo la funcionalidad, sino también los datos y el estado necesarios para su operación. Por ejemplo, un microservicio de control de riego podría ser una clase que maneja la lógica de riego, la comunicación con el sensor de humedad, y el control de la bomba, todo dentro de una estructura de clase bien definida.
- **Polimorfismo y Extensibilidad:**
  - Aprovechar el polimorfismo en POO puede permitir que diferentes microservicios compartan una interfaz común, facilitando la integración y la extensión del sistema. Esto es especialmente útil en proyectos donde diferentes tipos de sensores o actuadores deben ser manejados de manera uniforme, a pesar de sus diferencias específicas.

En resumen, combinar los principios de microservicios, el uso de shields, el framework Arduino, y la Programación Orientada a Objetos permite desarrollar sistemas IoT robustos, modulares y escalables. Estos conceptos no solo mejoran la estructura y mantenibilidad del código, sino que también optimizan el uso de recursos y facilitan la expansión del sistema a medida que crece en complejidad.