Integración de MySQL para Monitoreo de Datos Ambientales

Diseño de Interfaces - Práctica 3

Dr. Rubén Estrada Marmolejo
Asignatura: Diseño de Interfaces
Práctica: 3 - Integración con Base de Datos
Universidad de Guadalajara
Centro Universitario de Ciencias Exactas e Ingenierías
6 de noviembre de 2025

Índice

1. Introducción al Proyecto

1.1. Objetivos de la Práctica

Objetivo

Al finalizar esta práctica guiada, el estudiante será capaz de:

- Integrar una base de datos MySQL en un proyecto existente
- Almacenar datos de temperatura y humedad provenientes de WebSockets
- Implementar consultas SQL para recuperación y análisis de datos
- Visualizar datos históricos mediante gráficos
- Diseñar una interfaz completa para monitoreo ambiental

1.2. Descripción General

Teoría

Contexto del Proyecto:

- **Práctica:** N° 3 - Integración con Base de Datos
- **Problema:** Monitoreo continuo de variables ambientales
- **Solución:** Sistema integrado con base de datos para persistencia
- **Tecnologías:** MySQL, WebSockets, Qt, C++
- **Aplicación:** Monitoreo ambiental en tiempo real con historial

2. Desarrollo de la Práctica

2.1. Configuración de la Base de Datos

Teoría

Estructura de la tabla MySQL:

```
CREATE TABLE mediciones (
    id INT AUTO_INCREMENT PRIMARY KEY,
    temperatura DECIMAL(4,2) NOT NULL,
    humedad DECIMAL(4,2) NOT NULL,
    fecha_hora TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

3. Requisitos Previos y Configuración

3.1. Preparación del Entorno de Desarrollo

Objetivo

Objetivo: Configurar el entorno de desarrollo con todos los componentes necesarios para la integración MySQL-Qt6

3.2. Script de Configuración Automática

Teoría

Descarga del Script:

- **Repositorio:** GitHub - qt-basic-examples
- **Script:** compilar-mysql-qt6.sh
- **URL:** `https://github.com/esmarr58/qt-basic-examples/blob/main/scripts/compilar-mysql-qt6.sh`

3.3. Paso 1: Descargar el Script

Descarga directa desde terminal:

```
wget https://raw.githubusercontent.com/esmarr58/qt-basic-examples/main/scripts/compilar-mysql-qt6.sh
```

O alternativamente:

```
curl -O https://raw.githubusercontent.com/esmarr58/qt-basic-examples/main/scripts/compilar-mysql-qt6.sh
```

3.4. Paso 2: Dar Permisos de Ejecución

Cambiar permisos del script:

```
chmod +x compilar-mysql-qt6.sh
```

Verificar permisos:

```
ls -la compilar-mysql-qt6.sh
```

Resultado esperado:

```
-rwxr-xr-x 1 usuario usuario 1234 fecha compilar-mysql-qt6.sh
```

3.5. Paso 3: Ejecutar el Script

Ejecutar el script:

```
./compilar-mysql-qt6.sh
```

O con permisos de superusuario si es necesario:

```
sudo ./compilar-mysql-qt6.sh
```

3.6. Componentes que Instala el Script

Teoría

Paquetes que serán instalados:

- **MySQL Server:** Servidor de base de datos
- **MySQL Client:** Cliente para conexiones
- **Qt6:** Framework de desarrollo
- **Qt6 MySQL Driver:** Controlador para MySQL
- **Dependencias:** Bibliotecas necesarias para la compilación
- **Desarrollo:** Headers y herramientas de desarrollo

3.7. Verificación de la Instalación

Ejercicio de verificación:

1. Verificar instalación de MySQL:

   ```
   mysql --version
   ```

2. Verificar instalación de Qt6:

   ```
   qmake6 --version
   ```

3. Verificar driver MySQL de Qt6:

   ```
   ls /usr/lib/x86_64-linux-gnu/qt6/plugins/sqldrivers/
   ```

3.8. Solución de Problemas Comunes

Problemas frecuentes y soluciones:

- **Script no ejecutable:** Verificar permisos con `chmod +x`
- **Falta de permisos:** Usar `sudo` si es necesario
- **Error de descarga:** Verificar conexión a internet
- **Dependencias faltantes:** El script debería instalarlas automáticamente
- **Problemas de compilación:** Verificar versión de Qt6 instalada

3.9. Configuración Post-Instalación

Configuración adicional requerida:

1. Iniciar servicio de MySQL:

   ```
   sudo systemctl start mysql
   ```

2. Habilitar MySQL al inicio:

   ```
   sudo systemctl enable mysql
   ```

3. Configurar contraseña de root:

   ```
   sudo mysql_secure_installation
   ```

4. Crear base de datos para la práctica:

   ```
   mysql -u root -p -e "CREATE DATABASE monitoreo_ambiental;"
   ```

3.10. Verificación Final

Verificación completa del entorno:

```
# Verificar MySQL
mysql --version

# Verificar Qt6
qmake6 --version

# Verificar drivers
ls /usr/lib/x86_64-linux-gnu/qt6/plugins/sqldrivers/ | grep mysql

# Verificar servicio MySQL
sudo systemctl status mysql

# Probar conexión a MySQL
mysql -u root -p -e "SHOW DATABASES;"
```

4. Descarga y Configuración de la Plantilla Base

4.1. Plantilla Base de la Práctica

Objetivo

Objetivo: Obtener y configurar la plantilla base del proyecto para la integración MySQL-Qt6

4.2. Descarga de la Plantilla

Teoría

Información de la plantilla:

- **Repositorio:** GitHub - qt-basic-examples
- **Archivo:** Practica2-v3.zip
- **URL:** `https://github.com/esmarr58/qt-basic-examples/blob/main/2025B/Practica2/Practica2-v3.zip`
- **Contenido:** Proyecto base con WebSockets y estructura para MySQL

4.3. Paso 1: Descargar la Plantilla

Descarga directa desde terminal:

```
wget https://github.com/esmarr58/qt-basic-examples/raw/main/2025B/Practica2/Practica2-v3.zip
```

O alternativamente desde el navegador:

- Abrir: `https://github.com/esmarr58/qt-basic-examples`
- Navegar a: 2025B/Practica2/
- Descargar: Practica2-v3.zip

4.4. Paso 2: Descomprimir la Plantilla

Descomprimir el archivo ZIP:

```
unzip Practica2-v3.zip
```

Si no tienes unzip instalado:

```
sudo apt install unzip
unzip Practica2-v3.zip
```

Verificar contenido extraído:

```
ls -la Practica2-v3/
```

4.5. Paso 3: Abrir el Proyecto en Qt Creator

Abrir proyecto en Qt Creator:

1. Abrir Qt Creator
2. Menú: `File → Open File or Project...`
3. Navegar a la carpeta `Practica2-v3`
4. Seleccionar el archivo `Practica2.pro`
5. Click en `Configure Project`

O desde terminal:

```
cd Practica2-v3
qtcreator Practica2.pro &
```

4.6. Características de la Plantilla Base

Teoría

Análisis del Proyecto - Practica2.pro:

- **Cliente WebSocket:** `websocketcliente.cpp/h` - Comunicación en tiempo real
- **Administrador de Cámara:** `administradorcamara.cpp/h` - Control de captura de video
- **Procesamiento de Video:** `vervideo.cpp/h` - Visualización y manejo de streams
- **Detección de Colores:** `detectorcolores.cpp/h` - Algoritmos de visión por computadora
- **Detección de Sonrisas:** `detectorsonrisas.cpp/h` - Procesamiento facial con OpenCV
- **Frame Worker:** `frameworker.cpp/h` - Procesamiento en hilos separados
- **Interfaz Principal:** `mainwindow.cpp/h/ui` - GUI completa con Qt Widgets
- **Aplicación:** `main.cpp` - Punto de entrada del programa

Teoría

Configuración Técnica del Proyecto:

- **Qt Modules:** core, gui, widgets, multimedia, websockets, network
- **OpenCV Integration:** Incluye 6 bibliotecas principales de OpenCV
- **C++ Standard:** Configurado para C++17
- **Computer Vision:** Procesamiento de imágenes y video en tiempo real
- **Real-time Communication:** WebSockets para comunicación bidireccional
- **Multimedia:** Captura y reproducción de audio/video

Teoría

Flujo de Procesamiento de la Aplicación:

1. **Captura:** `administradorcamara` obtiene video de la cámara
2. **Procesamiento:** `frameworker` maneja frames en hilos separados
3. **Detección:** `detectorcolores` y `detectorsonrisas` aplican algoritmos CV
4. **Visualización:** `vervideo` muestra el video procesado
5. **Comunicación:** `websocketcliente` envía/recibe datos en tiempo real
6. **Interfaz:** `mainwindow` coordina todos los componentes

Para la Práctica 3 - Integración MySQL:

- Se agregará: `QT += sql` en el archivo .pro
- Se incluirá: `LIBS += -lmysqlclient`
- Se crearán: Nuevos archivos para manejo de base de datos
- Se modificarán: `websocketcliente` para guardar datos en MySQL
- Se extenderá: `mainwindow` para mostrar datos históricos

5. Práctica 3: Integración MySQL con Qt

5.1. Objetivo de la Práctica

Objetivo

Objetivo General: Integrar una base de datos MySQL en el proyecto Qt existente para permitir el almacenamiento persistente de datos recibidos via WebSockets.

5.2. Paso 1: Inclusión de Headers

Paso 1: Agregar headers de MySQL al proyecto

En mainwindow.h - Agregar includes:

```
#include <QSqlDatabase>
#include <QSqlQuery>
#include <QSqlError>
#include <QDebug>
```

Teoría

Explicación de los Headers:

- **QSqlDatabase:** Maneja conexiones a bases de datos
- **QSqlQuery:** Ejecuta consultas SQL
- **QSqlError:** Maneja errores de base de datos
- **QDebug:** Para mensajes de depuración

Preguntas de Comprensión - Headers:

1. ¿Qué función cumple QSqlDatabase en la aplicación?
2. ¿Por qué es importante incluir QSqlError?
3. ¿Cómo ayuda QDebug en el desarrollo?
4. ¿Qué tipo de operaciones permite QSqlQuery?
5. ¿Por qué se incluyen estos headers en mainwindow.h?

Respuestas - Headers:

1. Gestiona la conexión y configuración de la base de datos
2. Permite detectar y manejar errores en operaciones SQL
3. Muestra mensajes informativos durante la ejecución
4. Ejecutar SELECT, INSERT, UPDATE, DELETE y CREATE
5. Porque mainwindow.h es la clase principal que coordina la aplicación

5.3. Función de Verificación de Base de Datos

Paso 2: Implementar función verificarOCrearBDYTabla()

Parte 1/6 - Declaración de función:

```
bool verificarOCrearBDYTabla() {
    // Datos de conexión (ajusta según tu entorno)
    QString host = "localhost";
    QString user = "admin";
    QString pass = "hola1234";
    QString dbName = "interfaces2025b";
```

Teoría

Parámetros de Conexión:

- **host:** Dirección del servidor MySQL
- **user:** Usuario de la base de datos
- **pass:** Contraseña del usuario
- **dbName:** Nombre de la base de datos a crear/usar

Parte 2/6 - Primera conexión:

```
// Paso 1: Conectarse sin seleccionar base de datos
QSqlDatabase db = QSqlDatabase::addDatabase(
    "QMYSQL", "conexion_inicial");
db.setHostName(host);
db.setUserName(user);
db.setPassword(pass);
```

Teoría

Conexión Inicial:

- Se usa `addDatabase()` para crear conexión
- `"QMYSQL"` especifica el driver MySQL
- `çonexion_inicial"` es nombre único de conexión
- Se configuran host, usuario y contraseña

Parte 3/6 - Verificación de conexión:

```
    if (!db.open()) {
        qWarning() << " Error al conectar al servidor MySQL:"
                   << db.lastError().text();
        return false;
    }
```

Teoría

Manejo de Errores:

- `db.open()` intenta establecer conexión
- Si falla, muestra error con `qWarning()`
- `lastError().text()` obtiene mensaje de error
- Retorna `false` indicando fallo

Parte 4/6 - Creación de base de datos:

```
// Paso 2: Crear la base de datos si no existe
QSqlQuery query(db);
if (!query.exec(QString("CREATE DATABASE IF NOT EXISTS %1")
                .arg(dbName))) {
    qWarning() << " Error al crear la base de datos:"
               << query.lastError().text();
    db.close();
    return false;
}
db.close();
```

Teoría

Creación de BD:

- `QSqlQuery` ejecuta comandos SQL
- `CREATE DATABASE IF NOT EXISTS` crea solo si no existe
- `.arg(dbName)` inserta nombre de BD dinámicamente
- Se cierra conexión inicial después de crear BD

Parte 5/6 - Conexión a BD específica:

```
    // Paso 3: Conectarse a la base de datos
    QSqlDatabase db2 = QSqlDatabase::addDatabase(
        "QMYSQL", "conexion_principal");
    db2.setHostName(host);
    db2.setUserName(user);
    db2.setPassword(pass);
    db2.setDatabaseName(dbName);

     if (!db2.open()) {
        qWarning() << " No se pudo abrir la base de datos:" << db2.lastError().text();
        return false;
    }
```

Teoría

Verificación de Conexión a BD Específica:

- **Conexión específica:** Ahora nos conectamos a la base de datos creada
- **setDatabaseName():** Especifica qué base de datos usar
- **db2.open():** Intenta abrir la conexión a la BD específica
- **Verificación crítica:** Si falla, mostramos el error y retornamos false
- **Mensaje de error:** `db2.lastError().text()` nos dice por qué falló

Posibles causas de error en esta parte:

- **Contraseña incorrecta** del usuario MySQL
- **Permisos insuficientes** del usuario para acceder a la BD
- **Problemas de red** con el servidor MySQL
- **Driver MySQL** no instalado correctamente en Qt

Parte 6/6 - Creación de tabla:

```
    // Paso 4: Crear la tabla si no existe
    QString crearTabla =
        "CREATE TABLE IF NOT EXISTS datos ("
        "id INT AUTO_INCREMENT PRIMARY KEY, "
        "timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP, "
        "dato FLOAT, "
        "tipoDato VARCHAR(50), "
        "sensorID INT"
        ");";

    QSqlQuery query2(db2);
    if (!query2.exec(crearTabla)) {
        qWarning() << " Error al crear tabla:"
                   << query2.lastError().text();
        db2.close();
        return false;
    }

    qDebug() << " Base de datos y tabla listas.";
    db2.close();
    return true;
}
```

Teoría

Estructura de la Tabla:

- **id:** Clave primaria auto-incremental
- **timestamp:** Fecha/hora automática
- **dato:** Valor numérico del sensor
- **tipoDato:** Tipo de medición (temperatura, humedad)
- **sensorID:** Identificador del sensor

Preguntas de Comprensión - Función BD:

1. ¿Por qué se hacen dos conexiones a la base de datos?
2. ¿Qué ventaja tiene usar `IF NOT EXISTS`?
3. ¿Para qué sirve el campo `timestamp`?
4. ¿Por qué se usa `AUTO_INCREMENT` en el id?
5. ¿Qué representa el campo `tipoDato`?

Respuestas - Función BD:

1. Primera para crear BD, segunda para operaciones normales
2. Evita errores si la BD o tabla ya existen
3. Registra automáticamente cuándo se insertó el dato
4. Genera IDs únicos automáticamente
5. Especifica el tipo de medición (ej: "temperatura", "humedad")

5.4. Integración en el Constructor

Paso 3: Llamar la función desde el constructor

En MainWindow constructor - Al final:

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    // ... configuración existente ...

    // Al final del constructor, después de toda configuración:
    if (verificarOCrearBDYTabla()) {
        qDebug() << "Sistema de base de datos inicializado";
    } else {
        qWarning() << "Falló inicialización de BD";
    }
}
```

Teoría

Ubicación en el Constructor:

- Se llama **al final** después de toda configuración
- Permite que primero se configuren WebSockets y UI
- Si falla, no afecta funcionalidad básica de la app
- Mensajes de debug informan del estado

Preguntas de Comprensión - Constructor:

1. ¿Por qué se llama la función al final del constructor?
2. ¿Qué pasa si la inicialización de BD falla?
3. ¿Por qué se usan mensajes de debug?
4. ¿La aplicación puede funcionar sin BD?
5. ¿Cómo afecta esto al usuario final?

Respuestas - Constructor:

1. Para no interferir con configuración existente
2. La aplicación sigue funcionando pero sin guardar datos
3. Para monitorear el estado durante desarrollo
4. Sí, pero sin persistencia de datos
5. Puede usar la app pero datos no se guardarán

5.5. Modificación del Archivo .pro

Paso 4: Actualizar Practica2.pro para MySQL

En Practica2.pro - Agregar:

```
QT += core gui
QT += multimedia websockets network
QT += sql  # <-- AGREGAR ESTA LÍNEA

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

CONFIG += c++17
```

Teoría

Importancia de QT += sql:

- Habilita módulos de base de datos de Qt
- Proporciona clases QSqlDatabase, QSqlQuery, etc.
- Sin esta línea, no compilarían los includes SQL
- Necesario para cualquier operación con BD

Preguntas de Comprensión - Archivo .pro:

1. ¿Qué función cumple `QT += sql`?
2. ¿Qué pasa si no se agrega esta línea?
3. ¿En qué parte del archivo .pro debe ir?
4. ¿Qué otros módulos SQL podrían usarse?
5. ¿Es necesario reiniciar Qt Creator después del cambio?

Respuestas - Archivo .pro:

1. Habilita funcionalidad de bases de datos en Qt
2. Error de compilación con includes SQL
3. Con los otros módulos QT +=
4. QPSQL (PostgreSQL), QSQLITE (SQLite), etc.
5. Sí, o al menos recompilar el proyecto

6. Resumen Práctica 3 - Parte 1

🎓 **Logros de esta sesión:**

- Configurados headers para MySQL en mainwindow.h
- Implementada función de verificación/creación de BD
- Integrada llamada en constructor de MainWindow
- Actualizado archivo .pro con módulo SQL
- Estructura de BD lista para almacenar datos

◎ Objetivo

◎ Próximos pasos:

- Modificar WebSocketClient para guardar datos en BD
- Implementar consultas para recuperar datos históricos
- Crear interfaz para visualizar datos almacenados
- Agregar gráficos con datos de la base de datos

Recordatorio importante:

- La función se ejecuta al iniciar la aplicación
- Verifica y crea automáticamente la estructura necesaria
- Los datos se almacenan persistentemente en MySQL
- La aplicación es más robusta y profesional

7. Creación de la Clase InsertarDatos

7.1. Archivo de Cabecera insertar_dato.h

Paso 5: Crear clase InsertarDatos para manejo de base de datos

insertar_dato.h - Parte 1/6:

```
#pragma once
#include <QObject>
#include <QSqlDatabase>
#include <QSqlQuery>
#include <QSqlError>
#include <QDateTime>
#include <QJsonObject>

class InsertaDatos : public QObject {
    Q_OBJECT
```

Teoría

Includes y Herencia:

- **#pragma once:** Evita inclusión múltiple del archivo
- **QObject:** Clase base para sistema de señales y slots
- **QSqlDatabase:** Maneja conexiones a base de datos
- **QSqlQuery:** Ejecuta consultas SQL
- **QSqlError:** Maneja errores de base de datos
- **QDateTime:** Manejo de fechas y horas
- **QJsonObject:** Procesamiento de datos JSON

insertar_dato.h - Parte 2/6:

```
public:
    explicit InsertaDatos(QObject* parent = nullptr);

    // Configurar antes de mover al hilo
    void setParametrosConexion(const QString& host,
                               int port,
                               const QString& user,
                               const QString& pass,
                               const QString& dbName = "interfaces2025b");
```

Teoría

Constructor y Configuración:

- **explicit:** Evita conversiones implícitas no deseadas
- **parent = nullptr:** Constructor por defecto sin padre
- **setParametrosConexion:** Configura conexión antes de usar hilos
- **Parámetros:** host, puerto, usuario, contraseña y nombre BD
- **dbName por defecto:** ïnterfaces2025b"si no se especifica

insertar_dato.h - Parte 3/6:

```
public slots:
    // Llama estos slots DESPUÉS de moveToThread
    bool abrir();         // abre conexión y prepara INSERT
    void cerrar();        // cierra conexión
    void asegurarBDyTabla(); // crea BD y tabla si no existen
```

Teoría

Slots Públicos:

- **abrir():** Abre conexión y prepara consulta INSERT
- **cerrar():** Cierra la conexión a la base de datos
- **asegurarBDyTabla():** Crea BD y tabla automáticamente
- **Importante:** Llamar después de `moveToThread()`
- **Hilos:** Operaciones de BD deben estar en el mismo hilo

insertar_dato.h - Parte 4/6:

```
    // Inserciones
    void insertar(double dato,
                  const QString& tipoDato,
                  int sensorID,
                  const QDateTime& timestamp = QDateTime());

    void insertarDesdeJson(const QJsonObject& obj);
```

Teoría

Métodos de Inserción:

- **insertar():** Inserta datos individuales con parámetros
- **dato:** Valor numérico del sensor (temperatura, humedad)
- **tipoDato:** Tipo de medición ("temperatura", "humedad")
- **sensorID:** Identificador único del sensor
- **timestamp:** Fecha y hora (opcional, usa DEFAULT si vacío)
- **insertarDesdeJson():** Inserta desde objeto JSON

insertar_dato.h - Parte 5/6:

```
signals:
    void insertOK(int idGenerado);
    void errorSQL(QString descripcion);
```

Teoría

Señales de la Clase:

- **insertOK:** Se emite cuando inserción es exitosa
- **idGenerado:** ID auto-incremental generado por MySQL
- **errorSQL:** Se emite cuando hay error en operación SQL
- **descripcion:** Mensaje de error detallado
- **Comunicación:** Permiten notificar a otras partes del sistema

insertar_dato.h - Parte 6/6:

```
private:
    QString m_host = "localhost";
    int     m_port = 3306;
    QString m_user = "root";
    QString m_pass = "";
    QString m_db   = "interfaces2025b";
    QString m_connName;

    QSqlQuery m_insert;

    bool prepararInsert();
};
```

Teoría

Variables Privadas:

- **m_host, m_port:** Configuración de servidor MySQL
- **m_user, m_pass:** Credenciales de acceso
- **m_db:** Nombre de la base de datos
- **m_connName:** Nombre único para la conexión
- **m_insert:** Objeto QSqlQuery para consultas INSERT
- **prepararInsert():** Método privado para preparar consulta

Preguntas de Comprensión - Clase InsertarDatos:

1. ¿Por qué la clase hereda de QObject?
2. ¿Qué ventaja tiene usar `explicit` en el constructor?
3. ¿Por qué los slots deben llamarse después de `moveToThread()`?
4. ¿Qué diferencia hay entre `insertar()` y `insertarDesdeJson()`?
5. ¿Para qué sirven las señales `insertOK` y `errorSQL`?
6. ¿Por qué se usa `QSqlQuery m_insert` como miembro privado?

Respuestas - Clase InsertarDatos:

1. Para usar el sistema de señales y slots de Qt
2. Evita conversiones implícitas no deseadas del compilador
3. Porque las operaciones de BD deben estar en el mismo hilo
4. `insertar()` usa parámetros directos, `insertarDesdeJson()` usa objeto JSON
5. Para notificar éxito o error en las operaciones de base de datos
6. Para reutilizar la misma consulta preparada y mejorar rendimiento

8. Implementación de InsertarDatos

8.1. Archivo insertar_dato.cpp

insertar_dato.cpp - Parte 1/6:

```
#include "insertar_datos.h"
#include <QUuid>
#include <QVariant>
#include <QSqlQuery>

InsertaDatos::InsertaDatos(QObject* parent)
    : QObject(parent)
{}
```

Teoría

Includes y Constructor:

- **QUuid:** Genera identificadores únicos para conexiones
- **QVariant:** Maneja tipos de datos genéricos
- **QSqlQuery:** Ejecuta consultas SQL
- **Constructor:** Llama al constructor de QObject padre
- **Inicialización:** Variables miembro se inicializan en header

insertar_dato.cpp - Parte 2/6:

```
void InsertaDatos::setParametrosConexion(const QString& host,
                                         int port,
                                         const QString& user,
                                         const QString& pass,
                                         const QString& dbName)
{
    m_host = host;
    m_port = port;
    m_user = user;
    m_pass = pass;
    m_db   = dbName;
}
```

Teoría

Configuración de Parámetros:

- **Almacenamiento:** Guarda parámetros en variables miembro
- **m_host:** Dirección del servidor MySQL
- **m_port:** Puerto de conexión (default 3306)
- **m_user, m_pass:** Credenciales de autenticación
- **m_db:** Nombre de la base de datos
- **Uso:** Se llama antes de mover a hilo secundario

insertar_dato.cpp - Parte 3/6:

```
bool InsertaDatos::abrir() {
    // Conexión única por hilo
    m_connName = QStringLiteral("ins_%1").arg(
        QUuid::createUuid().toString(QUuid::WithoutBraces));
    if (QSqlDatabase::contains(m_connName))
        QSqlDatabase::removeDatabase(m_connName);
```

Teoría

Gestión de Conexiones Únicas:

- **m_connName:** Genera nombre único para la conexión
- **QUuid::createUuid():** Crea identificador único universal
- **WithoutBraces:** Formato sin llaves
- **QSqlDatabase::contains:** Verifica si conexión ya existe
- **removeDatabase:** Elimina conexión previa si existe

insertar_dato.cpp - Parte 4/6:

```
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL", m_connName);
    db.setHostName(m_host);
    db.setPort(m_port);
    db.setUserName(m_user);
    db.setPassword(m_pass);
    db.setDatabaseName(m_db);
    db.setConnectOptions(
        "MYSQL_OPT_RECONNECT=1;MYSQL_OPT_CONNECT_TIMEOUT=5");
```

Teoría

Configuración de Conexión:

- **addDatabase:** Crea nueva conexión con nombre único
- **setHostName, setPort:** Configura servidor y puerto
- **setUserName, setPassword:** Credenciales de acceso
- **setDatabaseName:** Especifica base de datos a usar
- **setConnectOptions:** Opciones avanzadas de MySQL

insertar_dato.cpp - Parte 5/6:

```
    if (!db.open()) {
        emit errorSQL(db.lastError().text());
        return false;
    }
    return prepararInsert();
}
```

Teoría

Apertura y Verificación:

- **db.open():** Intenta abrir conexión a base de datos
- **!db.open():** Si falla la conexión
- **emit errorSQL:** Emite señal con descripción del error
- **lastError().text():** Mensaje de error específico
- **prepararInsert():** Prepara consulta INSERT si conexión exitosa

insertar_dato.cpp - Parte 6/6:

```
void InsertaDatos::cerrar() {
    if (!m_connName.isEmpty() &&
        QSqlDatabase::contains(m_connName)) {
        QSqlDatabase db = QSqlDatabase::database(m_connName);
        if (db.isOpen()) db.close();
        QSqlDatabase::removeDatabase(m_connName);
    }
}
```

Teoría

Cierre de Conexión:

- **Verificación:** Comprueba que existe conexión activa
- **QSqlDatabase::contains:** Verifica si la conexión está registrada
- **database(m_connName):** Obtiene referencia a la conexión
- **db.close():** Cierra la conexión si está abierta
- **removeDatabase:** Elimina la conexión del registro

insertar_dato.cpp - Parte 7/6:

```
void InsertaDatos::asegurarBDyTabla() {
    // 1) Crear BD si no existe (conexión sin DB)
    const QString c1 = QStringLiteral("tmp_%1").arg(
        QUuid::createUuid().toString(QUuid::WithoutBraces));
    {
        QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL", c1);
        db.setHostName(m_host);
        db.setPort(m_port);
        db.setUserName(m_user);
        db.setPassword(m_pass);
```

Teoría

Creación de Base de Datos:

- **asegurarBDyTabla:** Garantiza que BD y tabla existan
- **Conexión temporal:** Usa conexión sin base de datos
- **Nombre único:** Genera identificador para conexión temporal
- **Configuración:** Mismo host, usuario y contraseña
- **Sin databaseName:** Para crear la base de datos

insertar_dato.cpp - Parte 8/6:

```
        if (db.open()) {
            QSqlQuery q(db);
            q.exec(QStringLiteral(
                "CREATE DATABASE IF NOT EXISTS `%1`").arg(m_db));
        }
        db.close();
        QSqlDatabase::removeDatabase(c1);
    }
```

Teoría

Ejecución Creación BD:

- **db.open():** Abre conexión al servidor MySQL
- **QSqlQuery q(db):** Crea consulta usando conexión temporal
- **CREATE DATABASE:** Crea BD si no existe
- **Backticks:** `` `%1` `` protege nombre de BD con caracteres especiales
- **Limpieza:** Cierra y elimina conexión temporal

insertar_dato.cpp - Parte 9/6:

```
    // 2) Crear tabla si no existe (con DB seleccionada)
    const QString c2 = QStringLiteral("tmp2_%1").arg(
        QUuid::createUuid().toString(QUuid::WithoutBraces));
    {
        QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL", c2);
        db.setHostName(m_host);
        db.setPort(m_port);
        db.setUserName(m_user);
        db.setPassword(m_pass);
        db.setDatabaseName(m_db);
```

Teoría

Segunda Conexión Temporal:

- **Conexión con BD:** Ahora especifica databaseName
- **Nombre diferente:** c2 para evitar conflictos
- **setDatabaseName(m_db):** Conecta a la base de datos creada
- **Misma configuración:** Host, puerto, credenciales iguales
- **Propósito:** Crear tabla dentro de la base de datos

insertar_dato.cpp - Parte 10/6:

```
        if (db.open()) {
            QSqlQuery q(db);
            q.exec(
                "CREATE TABLE IF NOT EXISTS datos ("
                " id INT AUTO_INCREMENT PRIMARY KEY, "
                " timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP, "
                " dato FLOAT, "
                " tipoDato VARCHAR(50), "
                " sensorID INT"
                ");"
                );
        }
        db.close();
        QSqlDatabase::removeDatabase(c2);
    }
}
```

Teoría

Creación de Tabla:

- **CREATE TABLE IF NOT EXISTS:** Crea tabla solo si no existe
- **id:** Clave primaria auto-incremental
- **timestamp:** Marca temporal automática
- **dato:** Valor numérico de la medición
- **tipoDato:** Tipo de dato (ej: "temperatura")
- **sensorID:** Identificador del sensor

insertar_dato.cpp - Parte 11/6:

```
bool InsertaDatos::prepararInsert() {
    QSqlDatabase db = QSqlDatabase::database(m_connName);
    m_insert = QSqlQuery(db);
    return m_insert.prepare(
        "INSERT INTO datos (timestamp, dato, tipoDato, sensorID) "
        "VALUES (?, ?, ?, ?);"
        );
}
```

Teoría

Preparación de Consulta INSERT:

- **QSqlDatabase::database:** Obtiene conexión activa
- **m_insert = QSqlQuery:** Asocia consulta a conexión
- **prepare():** Precompila consulta SQL para mejor rendimiento
- **Placeholders ?:** Marcadores para valores variables
- **Estructura:** timestamp, dato, tipoDato, sensorID

insertar_dato.cpp - Parte 12/6:

```
void InsertaDatos::insertar(double dato,
                            const QString& tipoDato,
                            int sensorID,
                            const QDateTime& timestamp)
{
    if (!m_insert.isActive() && m_insert.lastError().isValid()) {
        if (!prepararInsert()) {
            emit errorSQL(QStringLiteral(
                "No se pudo preparar INSERT: %1")
                .arg(m_insert.lastError().text()));
            return;
        }
    }
```

Teoría

Verificación de Consulta Activa:

- **m_insert.isActive():** Verifica si consulta está activa
- **lastError().isValid():** Comprueba si hay error previo
- **Re-preparar:** Si consulta no está activa, la re-prepara
- **Error handling:** Emite error si no puede preparar consulta
- **Early return:** Sale del método si hay error

insertar_dato.cpp - Parte 13/6:

```
    if (timestamp.isValid()) {
        m_insert.bindValue(0, timestamp);
    } else {
        // Enviamos NULL para que MySQL use DEFAULT
        m_insert.bindValue(0, QVariant(QVariant::DateTime));
    }
    m_insert.bindValue(1, dato);
    m_insert.bindValue(2, tipoDato);
    m_insert.bindValue(3, sensorID);
```

Teoría

Asignación de Valores:

- **timestamp.isValid():** Verifica si fecha es válida
- **bindValue(0, timestamp):** Asigna timestamp si es válido
- **QVariant(QVariant::DateTime):** NULL para timestamp automático
- **bindValue(1, dato):** Valor numérico del sensor
- **bindValue(2, tipoDato):** Tipo de medición
- **bindValue(3, sensorID):** ID del sensor

insertar_dato.cpp - Parte 14/6:

```
    if (!m_insert.exec()) {
        emit errorSQL(m_insert.lastError().text());
        return;
    }

    // Recuperar ID autogenerado
    QVariant lid = m_insert.lastInsertId();
    if (lid.isValid()) {
        emit insertOK(lid.toInt());
    } else {
```

Teoría

Ejecución y Recuperación de ID:

- **m_insert.exec():** Ejecuta consulta INSERT
- **Error handling:** Emite error si ejecución falla
- **lastInsertId():** Obtiene ID auto-generado por MySQL
- **lid.isValid():** Verifica si ID es válido
- **emit insertOK:** Notifica éxito con ID generado

insertar_dato.cpp - Parte 15/6:

```
        // Fallback si el driver no lo provee
        QSqlQuery q(QSqlDatabase::database(m_connName));
        if (q.exec(QStringLiteral("SELECT LAST_INSERT_ID()"))
            && q.next())
            emit insertOK(q.value(0).toInt());
        else
            emit insertOK(-1);
    }
}
```

Teoría

Fallback para ID:

- **LAST_INSERT_ID():** Función MySQL para obtener último ID
- **Nueva consulta:** Crea QSqlQuery para función MySQL
- **q.exec() q.next():** Ejecuta y avanza al primer resultado
- **q.value(0).toInt():** Obtiene ID desde resultado
- **emit insertOK(-1):** ID -1 si no se puede obtener

insertar_dato.cpp - Parte 16/6:

```
void InsertaDatos::insertarDesdeJson(const QJsonObject& obj) {
    const double  dato = obj.value(
        QStringLiteral("dato")).toDouble(
        std::numeric_limits<double>::quiet_NaN());
    const QString tipo = obj.value(
        QStringLiteral("tipoDato")).toString(
        obj.value(QStringLiteral("tipo")).toString());
```

Teoría

Extracción de Datos desde JSON:

- **obj.value("dato"):** Obtiene valor numérico del JSON
- **toDouble():** Convierte a double, usa NaN si no existe
- **quiet_NaN():** Valor "Not a Number"por defecto
- **tipoDato o tipo:** Busca en dos posibles campos
- **toString():** Convierte a string, string vacío si no existe

insertar_dato.cpp - Parte 17/6:

```
    const int     sid  = obj.value(
        QStringLiteral("sensorID")).toInt(
        obj.value(QStringLiteral("sensorId")).toInt(-1));
    const QString tsS  = obj.value(
        QStringLiteral("timestamp")).toString();

    QDateTime ts;
    if (!tsS.isEmpty())
        ts = QDateTime::fromString(tsS, Qt::ISODate);

    insertar(dato, tipo, sid, ts);
}
```

Teoría

Finalización InsertarDesdeJson:

- **sensorID o sensorId:** Busca ID en dos formatos posibles
- **toInt(-1):** Valor -1 por defecto si no existe
- **timestamp:** Obtiene string de timestamp
- **QDateTime::fromString:** Convierte desde formato ISO
- **insertar():** Llama al método principal con datos extraídos

Preguntas de Comprensión - Implementación:

1. ¿Por qué se usan conexiones temporales en `asegurarBDyTabla()`?
2. ¿Qué ventaja tiene usar `QUuid` para nombres de conexión?
3. ¿Cómo maneja la clase el caso de timestamp no válido?
4. ¿Por qué hay un fallback para `lastInsertId()`?
5. ¿Qué formatos de JSON acepta `insertarDesdeJson()`?
6. ¿Cómo se manejan los errores de conexión en `abrir()`?

Respuestas - Implementación:

1. Para no interferir con la conexión principal durante creación
2. Garantiza nombres únicos evitando conflictos entre hilos
3. Envía NULL para que MySQL use CURRENT_TIMESTAMP por defecto
4. Por si el driver no soporta lastInsertId() nativamente
5. Acepta "tipoDato.º "tipo", "sensorID.º "sensorId
6. Emite señal errorSQL con descripción y retorna false

9. Integración en MainWindow.h

9.1. Modificación de mainwindow.h

Paso 6: Integrar InsertarDatos en MainWindow.h

mainwindow.h - Agregar forward declaration:

```
class InsertaDatos;
```

Teoría

Forward Declaration:

- **Declaración anticipada:** Indica que la clase existe
- **Evita include circular:** Previene dependencias circulares
- **Suficiente para punteros:** Adecuado para declarar punteros
- **Incluir en .cpp:** El include completo va en el archivo .cpp

mainwindow.h - Agregar variables privadas:

```
private:
    QThread       m_hiloDB;         //  hilo para MySQL INSERTs
    InsertaDatos* m_escritorDB = nullptr;  //  worker en ese hilo
```

Teoría

Variables para Sistema de Hilos:

- **m_hiloDB:** Objeto QThread para operaciones de base de datos
- **m_escritorDB:** Puntero al worker InsertarDatos
- **nullptr:** Inicialización segura del puntero
- **Arquitectura:** Worker en hilo separado para no bloquear UI
- **Responsabilidades:** Hilo maneja ejecución, worker maneja lógica BD

Ventajas de esta arquitectura:

- **UI responsiva:** Operaciones de BD no bloquean interfaz
- **Seguridad:** Conexiones de BD en hilo dedicado
- **Escalabilidad:** Fácil agregar más workers
- **Mantenimiento:** Separación clara de responsabilidades

Preguntas de Comprensión - MainWindow.h:

1. ¿Por qué se usa forward declaration en lugar de include?
2. ¿Qué ventaja tiene usar QThread para operaciones de BD?
3. ¿Por qué inicializar m_escritorDB con nullptr?
4. ¿Cómo se comunican MainWindow y InsertarDatos?
5. ¿Qué problema evita esta arquitectura?

Respuestas - MainWindow.h:

1. Para evitar includes circulares y reducir dependencias
2. Mantiene la interfaz responsive durante operaciones lentas de BD
3. Evita punteros no inicializados y permite verificación segura
4. A través del sistema de señales y slots de Qt
5. Evita que la interfaz se congele durante inserciones en BD

🎓 **Resumen - Integración en Header:**

- Forward declaration de InsertarDatos
- Variables para sistema de hilos
- Arquitectura worker-hilo definida
- Preparación para implementación en .cpp

Objetivo

Próximo Paso:

- Implementar la inicialización en MainWindow.cpp
- Configurar el sistema de hilos
- Conectar señales y slots
- Probar la integración completa

10. Integración en MainWindow.cpp

10.1. Paso 1: Incluir Header

Paso 1: Agregar include en MainWindow.cpp

MainWindow.cpp - Agregar include:

```
#include "inserta_datos.h"
```

Teoría

Inclusión del Header:

- Completa definición: Proporciona definición completa de la clase
- Necesario para implementación: Permite usar métodos y señales
- En .cpp: Lugar correcto para includes completos
- Separación: Header declara, .cpp implementa

10.2. Paso 2: Configurar Conexiones en Constructor

IMPORTANTE: Esta conexión YA EXISTE y se SOBREESCRIBE:

```
connect(m_ws, &WebSocketCliente::telemetria, this,
```

MainWindow.cpp - Parte 1/8: Configurar SENSOR_ID

```
// SensorID que quieras asociar a este stream
constexpr int SENSOR_ID_WS = 101;

connect(m_ws, &WebSocketCliente::telemetria, this,
    [this](const QJsonObject& in)
    {
        auto push = [&](const QString& tipo, double valor)
        {
            QJsonObject out;
            out["tipoDato"] = tipo;
            out["dato"]     = valor;
            out["sensorID"] = SENSOR_ID_WS;
```

Teoría

Configuración de Sensor ID:

- constexpr: Constante en tiempo de compilación
- SENSOR_ID_WS = 101: ID único para este stream WebSocket
- Lambda anidada: push para reutilizar código de inserción
- QJsonObject out: Objeto para enviar a base de datos
- Campos: tipoDato, dato, sensorID obligatorios

MainWindow.cpp - Parte 2/8: Timestamp opcional

```
            // Si quisieras timestamp explícito:
            // out["timestamp"] =
            //   QDateTime::currentDateTimeUtc()
            //   .toString(Qt::ISODate);

            QMetaObject::invokeMethod(m_escritorDB,
                "insertarDesdeJson",
                Qt::QueuedConnection,
                Q_ARG(QJsonObject, out));
        };
```

Teoría

Invocación Segura entre Hilos:

- QMetaObject::invokeMethod: Llama métodos entre hilos de forma segura
- Qt::QueuedConnection: Ejecución asíncrona en cola del hilo destino
- m_escritorDB: Worker en hilo separado
- insertarDesdeJson: Método que procesa el JSON
- Q_ARG: Macro para pasar argumentos de forma segura

MainWindow.cpp - Parte 3/8: Procesar datos ESP32

```
        // Tu ESP32 manda: adc, temperatura, humedad,
        // dht_ok, tiempo_ms
        if (in.contains("temperatura"))
            push("temperatura",
                 in.value("temperatura").toDouble());
        if (in.contains("humedad"))
            push("humedad",
                 in.value("humedad").toDouble());
        if (in.contains("adc"))
            push("adc",
                 in.value("adc").toDouble());
```

Teoría

Procesamiento de Datos del ESP32:

- in.contains(): Verifica si el campo existe en JSON
- temperatura: Valor de temperatura del sensor
- humedad: Valor de humedad relativa
- adc: Lectura analógica-digital
- toDouble(): Conversión a tipo numérico
- Push: Envía cada dato individualmente a la BD

MainWindow.cpp - Parte 4/8: Más datos ESP32

```
        if (in.contains("dht_ok"))
            push("dht_ok",
                 in.value("dht_ok").toBool() ? 1.0 : 0.0);
        if (in.contains("tiempo_ms"))
            push("uptime_ms",
                 in.value("tiempo_ms").toDouble());
    });
```

Teoría

Datos Adicionales del ESP32:

- dht_ok: Estado del sensor DHT (booleano a 1.0/0.0)
- tiempo_ms: Tiempo de funcionamiento (uptime)
- uptime_ms: Nombre consistente en base de datos
- Operador ternario: Convierte booleano a numérico
- Estructura flexible: Fácil agregar nuevos tipos de datos

MainWindow.cpp - Parte 5/8: Configurar hilo BD

```
// === Hilo y worker para INSERTs en MySQL ===
// Ya tienes la BD/tabla con verificarOCrearBDYTabla(),
// así que no llamamos asegurarBDyTabla().
m_escritorDB = new InsertaDatos(); // sin parent;
                                   // se destruye con deleteLater
m_escritorDB->setParametrosConexion("localhost",
    3306, "admin", "hola1234", "interfaces2025b");
m_escritorDB->moveToThread(&m_hiloDB);
```

Teoría

Configuración del Worker de BD:

- new InsertaDatos(): Crea worker sin padre explícito
- deleteLater: Destrucción segura cuando hilo termina
- setParametrosConexion: Configura conexión a MySQL
- localhost:3306: Servidor MySQL local
- moveToThread: Mueve worker al hilo dedicado

MainWindow.cpp - Parte 6/8: Conexiones del hilo BD

```
connect(&m_hiloDB, &QThread::started, m_escritorDB,
    [this]() {
        if (!m_escritorDB->abrir()) {
            qWarning() << "InsertaDatos no pudo "
                          "abrir conexión:";
        } else {
            qDebug() << "InsertaDatos: conexión MySQL "
                        "abierta en hilo DB";
        }
    });
connect(&m_hiloDB, &QThread::finished, m_escritorDB,
    &QObject::deleteLater);
m_hiloDB.start();
```

Teoría

Gestión del Ciclo de Vida del Hilo:

- started: Cuando hilo inicia, abre conexión a BD
- abrir(): Intenta conectar a MySQL
- Mensajes: Debug para éxito, warning para error
- finished: Cuando hilo termina, destruye worker
- deleteLater: Destrucción segura en event loop
- start(): Inicia ejecución del hilo

MainWindow.cpp - Parte 7/8: Feedback de inserciones

```
// (Opcional) feedback de inserts
connect(m_escritorDB, &InsertaDatos::insertOK,
    this, [](int id){
        qDebug() << "INSERT OK id=" << id;
    });
connect(m_escritorDB, &InsertaDatos::errorSQL,
    this, [](const QString& e){
        qWarning() << "DB error:" << e;
    });
```

Teoría

Monitoreo de Operaciones de BD:

- insertOK: Señal emitida cuando inserción es exitosa
- id: ID auto-generado por MySQL
- errorSQL: Señal emitida cuando hay error
- e: Descripción del error de base de datos
- Debug: Para desarrollo y monitoreo
- Opcional: Puede deshabilitarse en producción

MainWindow.cpp - Parte 8/8: Destructor

```
MainWindow::~MainWindow()
{
    m_hiloDB.quit();
    m_hiloDB.wait();

    // Resto de limpieza existente...
    delete ui;
}
```

Teoría

Limpieza Segura de Hilos:

- quit(): Solicita terminación ordenada del hilo
- wait(): Espera a que el hilo termine completamente
- Evita crashes: Previene acceso a objetos destruidos
- deleteLater: Worker se destruye automáticamente
- Secuencia: Primero terminar hilos, luego destruir UI

Preguntas de Comprensión - MainWindow.cpp:

1. ¿Por qué se usa QMetaObject::invokeMethod en lugar de llamada directa?
2. ¿Qué ventaja tiene procesar cada dato individualmente?
3. ¿Por qué el worker no tiene parent en el constructor?
4. ¿Cómo se manejan los errores de conexión a MySQL?
5. ¿Por qué es importante usar wait() en el destructor?

Respuestas - MainWindow.cpp:

1. Para comunicación segura entre hilos diferentes
2. Mayor flexibilidad y facilidad para agregar nuevos tipos
3. Para que deleteLater funcione correctamente
4. Con señales errorSQL y mensajes de warning
5. Para asegurar que el hilo termine antes de destruir objetos

Resumen - Integración Completa:

- Inclusión de header en .cpp
- Configuración de conexión WebSocket sobreescrita
- Sistema de hilos para operaciones de BD
- Procesamiento de datos ESP32 a MySQL
- Monitoreo y feedback de operaciones
- Limpieza segura en destructor

Objetivo

Práctica 3 Completada:

- Sistema de base de datos MySQL integrado
- Arquitectura multi-hilo para operaciones de BD
- Recepción y almacenamiento de datos ESP32
- Interfaz responsive y sistema robusto
- Listo para producción y escalamiento