COMUNICACIÓN CON BASES DE DATOS

PARADIGMAS DE PROGRAMACIÓN II

Carlos Rojas Sánchez

Licenciatura en Informática

Universidad del Mar

Contenido

Conceptos generales

¿Qué es ADO.NET?

- Conjunto de clases que permite el acceso a datos desde C#.
- Soporta conexiones con SQL Server, MySQL, PostgreSQL, entre otros.
- Incluye:
  - `SqlConnection`
  - `SqlCommand`
  - `SqlDataReader`
  - `DataSet`, `DataTable`

Pasos para conectar con base de datos

1. Importar el espacio de nombres:
   - `using System.Data.SqlClient;`
2. Crear una cadena de conexión.
3. Abrir una conexión con `SqlConnection`.
4. Crear y ejecutar comandos con `SqlCommand`.
5. Leer resultados con `SqlDataReader`.
6. Cerrar la conexión.

Ejemplo de conexión y lectura

```
using System;
using System.Data.SqlClient;

class Program {
    static void Main() {
        string connStr = "Server=localhost;Database=miBase;User
            Id=sa;Password=12345;";
        using(SqlConnection conn = new SqlConnection(connStr)) {
            conn.Open();
            SqlCommand cmd = new SqlCommand("SELECT * FROM
                Usuarios", conn);
            SqlDataReader reader = cmd.ExecuteReader();

            while (reader.Read()) {
                Console.WriteLine($"ID: {reader["ID"]}, Nombre:
                    {reader["Nombre"]}");
            }
        }
    }
}
```

Buenas prácticas

- Siempre cerrar conexiones o usar `using`.
- Usar parámetros para evitar inyecciones SQL:

```
cmd.CommandText = "SELECT * FROM Usuarios WHERE ID = @id";
cmd.Parameters.AddWithValue("@id", 1);
```

- Manejar excepciones con `try-catch`.

Manejo de excepciones

```
try {
    conn.Open();
    // operaciones
} catch (SqlException ex) {
    Console.WriteLine("Error: " + ex.Message);
}
```

Drivers del lenguaje para conectividad con bases de datos

Drivers de C# para conectividad con bases de datos

- SQL Server – `System.Data.SqlClient`
  - Incluido en .NET Framework y .NET Core.
- MySQL – `MySql.Data` (MySQL Connector/NET)
  - Requiere instalar el paquete NuGet: `MySql.Data`.
- PostgreSQL – `Npgsql`
  - Requiere instalar el paquete NuGet: `Npgsql`.
- SQLite – `System.Data.SQLite`
  - Muy útil para aplicaciones pequeñas o locales.
- ODBC – `System.Data.Odbc`
  - Para conectarse mediante controladores ODBC genéricos.
- OLE DB – `System.Data.OleDb`
  - Para bases de datos Access u otras con soporte OLE DB.

Instalación de drivers con NuGet

- MySQL:

```
Install-Package MySql.Data
```

- PostgreSQL:

```
Install-Package Npgsql
```

- SQLite:

```
Install-Package System.Data.SQLite
```

Conectividad con MariaDB desde C#

- MariaDB es un fork de MySQL, por lo tanto es compatible con sus drivers.
- Se utiliza el mismo driver: `MySql.Data`.
- Puede instalarse desde NuGet:

```
Install-Package MySql.Data
```

- Requiere cadena de conexión compatible:

```
string connStr = "Server=localhost;Database=miBase;
User ID=root;Password=mipass;";
```

- También puede usarse el driver oficial: `MariaDBConnector`
  - Proyecto: `https://mariadb.com/downloads/connectors/net/`
  - Paquete NuGet: `MariaDBConnector`

Operaciones básicas con bases de datos

Conexión básica a base de datos

```
string connStr = "Server=localhost;Database=miBase;
User Id=sa;Password=12345;";
SqlConnection conn = new SqlConnection(connStr);
conn.Open();
Console.WriteLine("Conexión abierta.");
```

Conexión segura con using

```
string connStr = "Server=localhost;Database=miBase;
User Id=sa;Password=12345;";
using (SqlConnection conn = new SqlConnection(connStr)) {
    conn.Open();
    Console.WriteLine("Conectado correctamente.");
}
```

Inserción sin parámetros (no recomendable)

```
string sql = "INSERT INTO Usuarios (Nombre) VALUES ('Ana')";
SqlCommand cmd = new SqlCommand(sql, conn);
cmd.ExecuteNonQuery();
```

Inserción segura con parámetros

```
string sql = "INSERT INTO Usuarios (Nombre) VALUES (@nombre)";
SqlCommand cmd = new SqlCommand(sql, conn);
cmd.Parameters.AddWithValue("@nombre", "Ana");
cmd.ExecuteNonQuery();
```

Consulta sin parámetros

```
string sql = "SELECT * FROM Usuarios";
SqlCommand cmd = new SqlCommand(sql, conn);
SqlDataReader reader = cmd.ExecuteReader();
while (reader.Read()) {
    Console.WriteLine(reader["Nombre"]);
}
reader.Close();
```

Consulta segura con parámetros

```
string sql = "SELECT * FROM Usuarios WHERE ID = @id";
SqlCommand cmd = new SqlCommand(sql, conn);
cmd.Parameters.AddWithValue("@id", 1);
SqlDataReader reader = cmd.ExecuteReader();
if (reader.Read()) {
    Console.WriteLine(reader["Nombre"]);
}
reader.Close();
```

Actualización sin parámetros (no segura)

```
string sql = "UPDATE Usuarios SET Nombre='Luis' WHERE ID=1";
SqlCommand cmd = new SqlCommand(sql, conn);
cmd.ExecuteNonQuery();
```

Actualización con parámetros

```
string sql = "UPDATE Usuarios SET Nombre=@nombre WHERE ID=@id";
SqlCommand cmd = new SqlCommand(sql, conn);
cmd.Parameters.AddWithValue("@nombre", "Luis");
cmd.Parameters.AddWithValue("@id", 1);
cmd.ExecuteNonQuery();
```

Eliminación sin parámetros (no recomendable)

```
string sql = "DELETE FROM Usuarios WHERE ID=2";
SqlCommand cmd = new SqlCommand(sql, conn);
cmd.ExecuteNonQuery();
```

Eliminación segura con parámetros

```
string sql = "DELETE FROM Usuarios WHERE ID=@id";
SqlCommand cmd = new SqlCommand(sql, conn);
cmd.Parameters.AddWithValue("@id", 2);
cmd.ExecuteNonQuery();
```

Buenas prácticas en el acceso a bases de datos

- Usar `using` para cerrar conexiones automáticamente.
- Siempre validar y sanitizar la entrada del usuario.
- Evitar construir cadenas SQL manualmente.
- Preferir comandos parametrizados para toda consulta.
- Cerrar siempre `SqlDataReader` o usar `using`.
- Controlar errores con bloques `try-catch`.
- Limitar los permisos del usuario de base de datos.

¿Por qué usar parámetros en las consultas?

- Evitan **inyección SQL**, una de las vulnerabilidades más graves.
- Mejoran la legibilidad del código.
- Permiten que el motor de base de datos reutilice planes de ejecución.
- Separan los datos del código.
- Facilitan el mantenimiento y depuración.

Peligro de no usar parámetros (inyección SQL)

```
// Supongamos que 'input' viene de un TextBox
string input = "1; DROP TABLE Usuarios; --";
string sql = "SELECT * FROM Usuarios WHERE ID = " + input;
SqlCommand cmd = new SqlCommand(sql, conn);
cmd.ExecuteNonQuery(); // ¡Puede eliminar la tabla!
```

Consecuencia: el atacante puede ejecutar múltiples comandos maliciosos.

Prevención de inyección con parámetros

```
string input = "1; DROP TABLE Usuarios; --"; // Intento
    malicioso
string sql = "SELECT * FROM Usuarios WHERE ID = @id";
SqlCommand cmd = new SqlCommand(sql, conn);
cmd.Parameters.AddWithValue("@id", input);
cmd.ExecuteReader(); // Solo busca el ID, no ejecuta código
    malicioso
```

Resultado: se busca literalmente el texto, sin ejecutar comandos maliciosos.

Diferencia entre `using` y `try-catch`

- **`using`** se utiliza para liberar automáticamente recursos, como conexiones y lectores.
- **`try-catch`** se utiliza para capturar y manejar errores en tiempo de ejecución.

Ejemplo combinado

```
try {
    using (SqlConnection conn = new SqlConnection(connStr)) {
        conn.Open();
        // operaciones con base de datos
    }
} catch (SqlException ex) {
    Console.WriteLine("Error: " + ex.Message);
}
```

Diferencia entre `using` y `try-catch`

Resumen:

- `using` = asegura el cierre y liberación de recursos.
- `try-catch` = manejo de errores y control de flujo.
- Se deben usar juntos

Gestión de excepciones con monitoreo de recursos

Gestión de excepciones con monitoreo de recursos

Objetivo: garantizar que la aplicación sea confiable y que los recursos se liberen incluso en caso de error.

Ejemplo completo y seguro

```
try {
 using (SqlConnection conn = new SqlConnection(connStr)) {
  conn.Open();
  using (SqlCommand cmd = new SqlCommand("SELECT * FROM Usuarios
      ", conn))
  using (SqlDataReader reader = cmd.ExecuteReader()) {
   while (reader.Read()) {
    Console.WriteLine(reader["Nombre"]);
   }
  }
 }
} catch (SqlException ex) {
    Console.WriteLine("Error de SQL: " + ex.Message);
} catch (Exception ex) {
    Console.WriteLine("Error general: " + ex.Message);
}
```

Gestión de excepciones con monitoreo de recursos

Ventajas:

- Libera recursos automáticamente con `using`.
- Captura errores específicos y generales.
- Evita fugas de memoria o conexiones abiertas.

Estructura robusta de acceso a base de datos

Uso combinado de:

- `using`: para liberar recursos como conexiones y lectores.
- `try-catch`: para capturar errores y evitar que el programa se bloquee.
- `finally` (opcional): ejecutar acciones incluso si hay errores.

Estructura robusta de acceso a base de datos

Estructura típica

```
try {
 using (var conn = new SqlConnection(connStr)) {
  conn.Open();
  using (var cmd = new SqlCommand(query, conn)) {
   // Ejecutar comandos
  }
 }
} catch (SqlException ex) {
 // Error de base de datos
} catch (Exception ex) {
 // Otros errores
}
```

¿Por qué monitorear y liberar recursos?

- Las conexiones, lectores y comandos consumen recursos del sistema.
- Si no se cierran correctamente:
  - La aplicación puede volverse lenta.
  - El servidor puede rechazar nuevas conexiones.
  - Hay riesgo de pérdida de datos o corrupción.
- El uso de `using` garantiza el cierre inmediato.
- El uso de `try-catch` evita que errores detengan el programa.

Excepciones comunes en acceso a bases de datos (1/2)

- `SqlException`
  - Error en la consulta SQL, conexión, credenciales, etc.
  - **Ejemplo:** tabla inexistente, clave duplicada, timeout.
- `InvalidOperationException`
  - Uso incorrecto del objeto (conexión cerrada, lector ocupado).
  - **Ejemplo:** leer antes de abrir el lector.
- `FormatException`
  - Conversión de tipos inválida.
  - **Ejemplo:** intentar convertir texto a entero.

Excepciones comunes en acceso a bases de datos (2/2)

- `TimeoutException`
  - La operación tardó más de lo permitido.
  - **Ejemplo:** consulta muy lenta o red saturada.
- `IOException`
  - Problemas con el acceso a disco (en bases SQLite).
- `DbException` (genérica)
  - Clase base para excepciones de proveedor de datos.
  - Útil si se desea manejar errores de forma genérica.

Recomendaciones para manejar excepciones

- `SqlException` → mostrar mensaje y registrar el error.
- `InvalidOperationException` → revisar lógica del flujo.
- `FormatException` → validar datos de entrada.
- `TimeoutException` → optimizar la consulta, revisar red.
- `DbException` → manejar de forma genérica en código compartido.

Ejemplo

```
catch (SqlException ex) {
 Log(ex); Console.WriteLine("Error de base de datos.");
}
catch (FormatException ex) {
 Console.WriteLine("Entrada inválida.");
}
```

Herramientas para generación de reportes

¿Qué son los reportes?

- Son documentos que presentan información organizada, estructurada y legible.
- Se utilizan para analizar datos, tomar decisiones o comunicar resultados.
- En sistemas de software, los reportes reflejan la información almacenada en la base de datos.
- Pueden incluir: textos, tablas, gráficos, imágenes, cálculos, entre otros.

Ejemplos comunes:

- Listado de clientes, ventas del mes, resumen de inventario, facturas, etc.

Objetivos de los reportes

- **Organizar datos crudos** de forma comprensible.
- **Facilitar la toma de decisiones** con base en la información.
- **Automatizar la generación de documentos** (ej. facturas, informes).
- **Exportar o imprimir información** en formatos comunes: PDF, Excel, etc.

Importancia: Permiten visualizar los datos del sistema de forma útil y presentable.

Tipos de reportes

- **Listados simples:** filas con información básica (ej. usuarios, productos).
- **Reportes agrupados:** resumen por categorías (ej. ventas por zona).
- **Reportes gráficos:** incluyen barras, pastel, líneas.
- **Reportes dinámicos:** permiten aplicar filtros, ordenamientos.
- **Documentos formales:** facturas, boletas, tickets.

Reportes en aplicaciones C#

- Se generan a partir de los datos de la base de datos.
- Se pueden presentar al usuario en una ventana, exportar o imprimir.
- El diseño puede hacerse con herramientas visuales o con código.
- Dependiendo del tipo de reporte, se puede usar:
  - Diseñadores como Crystal Reports, RDLC.
  - Librerías como iTextSharp, ClosedXML, FastReport, etc.

Reportes en aplicaciones C#

- Se generan a partir de los datos de la base de datos.
- Se pueden presentar al usuario en una ventana, exportar o imprimir.
- El diseño puede hacerse con herramientas visuales o con código.
- Dependiendo del tipo de reporte, se puede usar:
  - Diseñadores como Crystal Reports, RDLC.
  - Librerías como iTextSharp, ClosedXML, FastReport, etc.

Ventaja: Automatización y profesionalismo en la presentación de datos.

Crystal Reports

- Herramienta visual para diseño y generación de reportes.
- Se integra con Visual Studio (requiere instalación adicional).
- Permite conectar a diversas bases de datos: SQL Server, MySQL, Oracle, etc.
- Genera reportes con tablas, gráficos, filtros y agrupaciones.
- Exporta a PDF, Excel, Word, etc.
- Ideal para aplicaciones empresariales con reportes complejos.

Licencia: Gratuita con limitaciones. Versión completa es de pago.

Microsoft Report Viewer (RDLC)

- Sistema de reportes local embebido en aplicaciones .NET.
- Los archivos .rdlc se diseñan visualmente en Visual Studio.
- No requiere servidor de reportes (a diferencia de SSRS).
- Compatible con WinForms, ASP.NET y Blazor.
- Exporta a PDF, Excel, Word directamente.
- Se puede usar con datasets locales o consultas SQL.

Licencia: Gratuita (de Microsoft).

iTextSharp / PdfSharp

- Librerías para crear y manipular archivos PDF desde C#.
- Permiten construir reportes de forma programática.
- Soportan texto, tablas, imágenes, estilos, y más.
- Ideal para reportes PDF personalizados (sin diseñador visual).
- Compatible con aplicaciones de consola, WinForms y web.
- `iTextSharp` tiene versión gratuita y versión comercial.

Licencia:

- iTextSharp: AGPL (uso comercial requiere licencia).
- PdfSharp: Open Source (MIT).

FastReport .NET

- Solución profesional para generación de reportes en C#.
- Incluye editor visual, diseñador en tiempo de ejecución, exportación variada.
- Compatible con WinForms, WPF, ASP.NET y Blazor.
- Soporta múltiples bases de datos y exportaciones (PDF, Excel, Word, etc.).
- Permite reportes interactivos con filtros, paginación y subreportes.
- Más ligero y rápido que Crystal Reports.

Licencia: Comercial (versión trial disponible).

ClosedXML / EPPlus (Excel)

- Librerías para crear archivos Excel (.xlsx) directamente desde C#.
- No requieren tener Microsoft Office instalado.
- Permiten crear hojas, insertar datos, aplicar estilos, fórmulas, etc.
- Útiles para reportes tabulares exportables.
- Se integran fácilmente en apps de consola, escritorio o web.
- EPPlus tiene mejor rendimiento en archivos grandes.

Licencia:

- ClosedXML: MIT (gratuita).
- EPPlus: Polyform Noncommercial (gratuito para uso no comercial).