🏗️ Disconnected Architecture in ADO.NET

📋 📚 What You'll Learn

- Understanding Disconnected Architecture
- Core Components (DataSet, DataTable, DataAdapter)
- How Data Flow Works
- Practical Examples
- Best Practices
- When to Use vs Connected Architecture

🎯 What is Disconnected Architecture?

ⓘ Definition

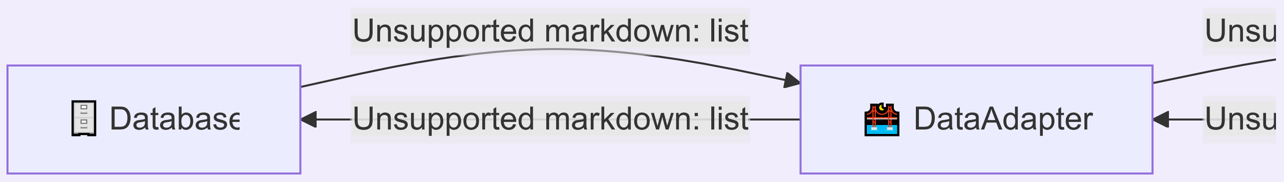
Disconnected Architecture is a data access model where the application retrieves data from the database, stores it in memory, and then **closes the connection**. The application can then work with this data **offline** without maintaining an active database connection. Changes are cached locally and synchronized back to the database when needed.

Key Concept

Unlike connected architecture (which maintains an open connection while reading data), disconnected architecture **minimizes database connections** by working with cached data in memory.

🔄 Architecture Flow

☰ The Complete Journey



📋 Step-by-Step Process

✓ Step 1: Connection Opens 🔌

The application establishes a connection to the database. This connection is **temporary** and will be closed soon.

✓ Step 2: Data Retrieval 📥

The DataAdapter executes SQL commands and retrieves data from the database into a DataSet using the `Fill()` method.

✓ Step 3: Connection Closes ❌

Once data is retrieved, the database connection is **immediately closed**. The data now lives in memory.

✓ Step 4: Offline Work 💻

The application works with the DataSet completely **offline**. Users can view, modify, add, or delete data in memory.

✓ Step 5: Synchronization 🔄

When ready, the connection reopens, and the DataAdapter's `Update()` method sends all changes back to the database.

✓ Step 6: Connection Closes Again ❌

After synchronization, the connection closes once more, completing the disconnected cycle.

🧩 Core Components

💾 DataSet

The In-Memory Database

An **in-memory cache** of data that can contain multiple DataTables. Think of it as a **mini-database in memory**.

Characteristics

- 📚 **Multiple Tables**: Can hold many tables at once
- 🔗 **Relationships**: Supports relationships between tables
- 📴 **Disconnected**: Works without database connection
- 💿 **Persistent**: Data stays in memory until cleared

📊 DataTable

The Table Structure

Represents a **single table** of in-memory data with rows and columns, similar to a database table.

☰ Characteristics

- 📋 **Rows & Columns**: Just like database tables
- 🔒 **Constraints**: Supports primary keys, unique constraints
- ✅ **Validation**: Can add validation rules
- 🔄 **Change Tracking**: Tracks additions, modifications, deletions

---

🌉 DataAdapter

✎ The Bridge

Acts as a **bridge** between the DataSet and the database, handling data retrieval and updates.

☰ Key Methods

- 📥 `Fill()` - Retrieves data from database to DataSet
- 📤 `Update()` - Sends changes from DataSet to database
- ⚙️ Auto-generates INSERT, UPDATE, DELETE commands
- 🔄 Handles batch operations

👁 DataView

✎ The Filtered View

Provides a **customizable view** of a DataTable, allowing filtering, sorting, and searching without modifying the original data.

☰ Capabilities

- 🔍 **Filter**: Show only specific rows
- 📊 **Sort**: Order data by any column
- 🔎 **Search**: Find specific records
- 👀 **Multiple Views**: Different views of same data

💻 Code Examples

🚀 Basic Disconnected Data Access

☰ Complete Example

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

string connectionString = "Server=myServer;Database=myDB;...";

// Create DataAdapter with SQL query
SqlDataAdapter adapter = new SqlDataAdapter(
    "SELECT * FROM Customers", connectionString);

// Create DataSet to hold data
DataSet dataSet = new DataSet();

// Fill DataSet (connection opens, retrieves data, closes)
adapter.Fill(dataSet, "Customers");

// ✅ Now work with data OFFLINE - no database connection!
DataTable customersTable = dataSet.Tables["Customers"];
```

```
// 📖 Display data
foreach (DataRow row in customersTable.Rows)
{
    Console.WriteLine($"{row["CustomerID"]}: {row["Name"]}");
}

// ✏️ Modify data offline
customersTable.Rows[0]["Name"] = "Updated Name";

// ➕ Add new row offline
DataRow newRow = customersTable.NewRow();
newRow["Name"] = "New Customer";
newRow["Email"] = "new@example.com";
customersTable.Rows.Add(newRow);

// ❌ Delete row offline
customersTable.Rows[1].Delete();

// 🔄 Persist changes to database (connection opens, updates, closes)
SqlCommandBuilder builder = new SqlCommandBuilder(adapter);
adapter.Update(dataSet, "Customers");
```

📚 Working with Multiple Tables

Related Tables

```
DataSet dataSet = new DataSet();

// Fill multiple tables
SqlDataAdapter customerAdapter = new SqlDataAdapter(
    "SELECT * FROM Customers", connectionString);
customerAdapter.Fill(dataSet, "Customers");

SqlDataAdapter orderAdapter = new SqlDataAdapter(
    "SELECT * FROM Orders", connectionString);
orderAdapter.Fill(dataSet, "Orders");

// 🔗 Create relationship between tables
```

```
DataRelation relation = new DataRelation(
    "CustomerOrders",
    dataSet.Tables["Customers"].Columns["CustomerID"],
    dataSet.Tables["Orders"].Columns["CustomerID"]);
dataSet.Relations.Add(relation);

// 🔍 Navigate relationships
foreach (DataRow customer in dataSet.Tables["Customers"].Rows)
{
    Console.WriteLine($"Customer: {customer["Name"]}");

    // Get related orders
    DataRow[] orders = customer.GetChildRows(relation);
    foreach (DataRow order in orders)
    {
        Console.WriteLine($"  Order: {order["OrderID"]}");
    }
}
```

🔍 Using DataView for Filtering

Filter and Sort

```
DataTable customersTable = dataSet.Tables["Customers"];

// Create DataView with filter
DataView view = new DataView(customersTable);
view.RowFilter = "Country = 'USA' AND Age > 25";
view.Sort = "Name ASC";

// 📖 Display filtered data
foreach (DataRowView rowView in view)
{
    Console.WriteLine($"{rowView["Name"]} - {rowView["Country"]}");
}
```

```
// 🔎 Find specific row
DataRowView[] found = view.FindRows("Smith");
```

🔄 Advanced Data Manipulation

Complex Operations

```
DataTable dt = dataSet.Tables["Products"];

// ➕ Add new product
DataRow newProduct = dt.NewRow();
newProduct["ProductName"] = "New Product";
newProduct["Price"] = 99.99;
newProduct["Stock"] = 100;
dt.Rows.Add(newProduct);

// ✏️ Update existing product
DataRow[] products = dt.Select("ProductID = 5");
if (products.Length > 0)
{
    products[0]["Price"] = 149.99;
    products[0]["Stock"] = 50;
}

// ❌ Delete out of stock products
DataRow[] outOfStock = dt.Select("Stock = 0");
foreach (DataRow row in outOfStock)
{
    row.Delete();
}

// 💾 Save all changes
SqlCommandBuilder builder = new SqlCommandBuilder(adapter);
adapter.Update(dataSet, "Products");
```

✨ Benefits of Disconnected Architecture

✓ 🚀 Scalability

Minimizes database connections, allowing the system to handle **more concurrent users** efficiently.

✓ ⚡ Performance

Reduces network traffic and database load by working with **cached data in memory**.

✓ 💾 Offline Capability

Applications can work **without continuous database connectivity**, perfect for mobile or distributed scenarios.

✓ 🔄 Batch Updates

Multiple changes can be **accumulated** and sent to the database in a **single operation**.

✓ 🎯 Flexibility

Data can be manipulated, filtered, and sorted **without hitting the database**.

✓ 🔗 Relationships

Supports **complex relationships** between multiple tables in memory.

✓ 📊 Data Binding

Easy integration with **UI controls** for data display and manipulation.

✓ 🛡️ **Transaction Control**

Changes can be **reviewed before committing** to the database.

⚖️ Connected vs Disconnected Architecture

🔴 Connected Architecture (DataReader)

Characteristics

- ✅ Uses **DataReader**
- ✅ Connection stays **open**
- ✅ Forward-only reading
- ✅ Less memory usage
- ✅ Faster for simple reads
- ❌ Locks database resources
- ❌ Not scalable for many users
- ❌ No offline work

🟢 Disconnected Architecture (DataSet)

✓ **Characteristics**

- ✅ Uses **DataSet**
- ✅ Connection opens **briefly**
- ✅ Random access to data
- ✅ Highly scalable
- ✅ Offline capability
- ✅ Multiple tables support

❌ More memory usage
❌ Slower for simple reads

🎯 When to Use Which?

♨ **Decision Guide**

Use Connected (DataReader) 📖

- Reading data **once**, sequentially
- Simple **reports** or displays
- **Large datasets** that don't fit in memory
- **Real-time** data requirements

Use Disconnected (DataSet) 💾

- Need to **manipulate** data
- **Offline** work required
- **Complex** applications with relationships
- **Multiple users** accessing system
- **Batch** updates needed

📊 Comparison Table

Feature	🔴 Connected	🟢 Disconnected
🔌 Connection	Stays Open	Opens Briefly
📖 Data Access	Forward-Only	Random Access
⚡ Speed	Faster	Moderate
💾 Memory	Low Usage	Higher Usage
👥 Scalability	Limited	High

Feature	🔴 Connected	🟢 Disconnected
📴 Offline Work	❌ No	✅ Yes
🔄 Updates	Direct	Batch
📚 Multiple Tables	❌ No	✅ Yes
🔗 Relationships	❌ No	✅ Yes

⚠️ Important Considerations

🔍 Concurrency Issues

⚠ Problem

Since data is cached, **multiple users** might work with **stale data**. Implement optimistic concurrency control to handle conflicts.

```
// Handling concurrency conflicts
try
{
    adapter.Update(dataSet, "Customers");
}
catch (DBConcurrencyException ex)
{
    Console.WriteLine("⚠️ Concurrency conflict detected!");
    // Handle conflict: refresh data or notify user
    dataSet.Clear();
    adapter.Fill(dataSet, "Customers");
}
```

💾 Memory Management

⚠ Consideration

DataSets can consume **significant memory** for large datasets. Consider **pagination** or loading only necessary data.

```
// Load only necessary data
string query = "SELECT TOP 100 * FROM Products WHERE Category =
@category";
SqlDataAdapter adapter = new SqlDataAdapter(query, connectionString);
adapter.SelectCommand.Parameters.AddWithValue("@category",
"Electronics");
```

🔄 Data Synchronization

⚠ **Best Practice**

Ensure proper **error handling** when synchronizing changes back to the database. Use **transactions** for data integrity.

```
using (SqlConnection connection = new
SqlConnection(connectionString))
{
    connection.Open();
    SqlTransaction transaction = connection.BeginTransaction();

    try
    {
        adapter.SelectCommand.Transaction = transaction;
        adapter.Update(dataSet, "Customers");
        transaction.Commit();
    }
    catch
    {
        transaction.Rollback();
        throw;
    }
}
```

🎓 Best Practices

1 Dispose Resources Properly

♨ Memory Management

Always dispose of DataAdapters and connections using `using` statements to prevent memory leaks.

```
using (SqlDataAdapter adapter = new SqlDataAdapter(query,
connectionString))
{
    // Work with adapter
} // Automatically disposed
```

---

2 Load Only Needed Data

♨ Performance

Don't retrieve entire tables. Use **WHERE clauses** to filter data at the database level before loading into memory.

```
// ❌ BAD - Loads entire table
string query = "SELECT * FROM Customers";

// ✅ GOOD - Loads only needed data
string query = "SELECT * FROM Customers WHERE Country = 'USA' AND
Status = 'Active'";
```

---

3 Use Transactions

Data Integrity

Wrap batch updates in **transactions** to ensure all-or-nothing data integrity when synchronizing changes.

```
// ✅ Use transactions for batch updates
SqlTransaction transaction = connection.BeginTransaction();
try
{
    adapter.Update(dataSet, "Customers");
    transaction.Commit();
}
catch
{
    transaction.Rollback();
}
```

---

4 Implement Validation

Data Quality

Add **constraints** and **validation rules** to DataTables to maintain data quality before database updates.

```
DataTable dt = new DataTable("Customers");

// Add columns with constraints
DataColumn idColumn = new DataColumn("CustomerID", typeof(int));
idColumn.AutoIncrement = true;
idColumn.AutoIncrementSeed = 1;
dt.Columns.Add(idColumn);

DataColumn nameColumn = new DataColumn("Name", typeof(string));
```

```
nameColumn.AllowDBNull = false;
nameColumn.MaxLength = 100;
dt.Columns.Add(nameColumn);

// Set primary key
dt.PrimaryKey = new DataColumn[] { idColumn };
```

5 Handle Conflicts

Concurrency

Implement proper **concurrency conflict** resolution strategies for multi-user scenarios.

```
adapter.RowUpdated += (sender, e) =>
{
    if (e.Status == UpdateStatus.ErrorsOccurred)
    {
        Console.WriteLine($"⚠ Error updating row:
{e.Row["CustomerID"]}");
        e.Status = UpdateStatus.SkipCurrentRow;
    }
};
```

6 Consider Performance

Optimization

For **read-heavy operations** with large datasets, consider using DataReader (connected) instead.

```
// For simple reads: Use DataReader
using (SqlDataReader reader = command.ExecuteReader())
```

```
{
    while (reader.Read())
    {
        // Process data
    }
}

// For complex manipulation: Use DataSet
DataSet dataSet = new DataSet();
adapter.Fill(dataSet);
```

🎯 Common Use Cases

1️⃣ Desktop Applications 🖥️

☰ Scenario

Windows Forms or WPF applications that need to **work with data offline** and sync periodically.

```
// Load data at startup
private void LoadData()
{
    adapter.Fill(dataSet, "Customers");
    dataGridView.DataSource = dataSet.Tables["Customers"];
}

// Save changes on button click
private void SaveButton_Click(object sender, EventArgs e)
{
    adapter.Update(dataSet, "Customers");
    MessageBox.Show("✅ Changes saved successfully!");
}
```

2️⃣ Mobile Applications 📱

Scenario

Apps that **cache data locally** and synchronize when network connection is available.

3️⃣ Data Entry Forms 📝

Scenario

Complex forms where users make **multiple changes** before saving all data at once.

4️⃣ Reporting Systems 📊

Scenario

Applications that retrieve data, perform **calculations**, and generate reports without keeping connections open.

5️⃣ Web Applications 🌐

Scenario

Multi-tier web apps where the middle tier **caches data** and serves multiple clients efficiently.

🔄 Data State Tracking

ⓘ **DataRow States**

The DataSet tracks changes automatically using row states:

State	Description	Icon
Unchanged	Original data, no changes	⚪
Added	New row added	🟢
Modified	Existing row changed	🟡
Deleted	Row marked for deletion	🔴
Detached	Row created but not added	⚫

```csharp
// Check row state
foreach (DataRow row in dt.Rows)
{
    switch (row.RowState)
    {
        case DataRowState.Added:
            Console.WriteLine("🟢 New row");
            break;
        case DataRowState.Modified:
            Console.WriteLine("🟡 Modified row");
            break;
        case DataRowState.Deleted:
            Console.WriteLine("🔴 Deleted row");
            break;
        case DataRowState.Unchanged:
            Console.WriteLine("⚪ Unchanged row");
            break;
    }
}

// Get only modified rows
DataRow[] modifiedRows = dt.Select(null, null,
DataViewRowState.ModifiedCurrent);
```

📈 Performance Tips

🔥 **Optimization Strategies**

🚀 Use DataTable.BeginLoadData()

```
DataTable dt = new DataTable();
dt.BeginLoadData(); // Suspend constraints and events

// Load large amount of data
foreach (var item in largeDataSet)
{
    DataRow row = dt.NewRow();
    row["Column1"] = item.Value1;
    dt.Rows.Add(row);
}

dt.EndLoadData(); // Resume constraints and events
```

📊 Enable Batch Updates

```
adapter.UpdateBatchSize = 100; // Update 100 rows at a time
adapter.Update(dataSet, "Customers");
```

💾 Clear DataSet When Done

```
// Free memory when finished
dataSet.Clear();
dataSet.Dispose();
```

🎬 Complete Example: CRUD Operations

☰ **Full Implementation**

```
public class CustomerManager
{
    private string connectionString;
    private SqlDataAdapter adapter;
    private DataSet dataSet;

    public CustomerManager(string connString)
    {
        connectionString = connString;
        adapter = new SqlDataAdapter(
            "SELECT * FROM Customers", connectionString);
        dataSet = new DataSet();

        // Auto-generate commands
        SqlCommandBuilder builder = new SqlCommandBuilder(adapter);
    }

    // 📥 Load data
    public void LoadCustomers()
    {
        dataSet.Clear();
        adapter.Fill(dataSet, "Customers");
        Console.WriteLine("✅ Data loaded successfully!");
    }

    // ➕ Add customer
    public void AddCustomer(string name, string email, string phone)
    {
        DataTable dt = dataSet.Tables["Customers"];
        DataRow newRow = dt.NewRow();
        newRow["Name"] = name;
        newRow["Email"] = email;
        newRow["Phone"] = phone;
        dt.Rows.Add(newRow);
        Console.WriteLine("✅ Customer added (offline)");
    }

    // ✏️ Update customer
    public void UpdateCustomer(int customerId, string newEmail)
    {
        DataTable dt = dataSet.Tables["Customers"];
        DataRow[] rows = dt.Select($"CustomerID = {customerId}");

        if (rows.Length > 0)
        {
            rows[0]["Email"] = newEmail;
            Console.WriteLine("✅ Customer updated (offline)");
        }
    }

    // ❌ Delete customer
    public void DeleteCustomer(int customerId)
    {
        DataTable dt = dataSet.Tables["Customers"];
        DataRow[] rows = dt.Select($"CustomerID = {customerId}");

        if (rows.Length > 0)
        {
            rows[0].Delete();
            Console.WriteLine("✅ Customer deleted (offline)");
        }
    }

    // 💾 Save all changes
    public void SaveChanges()
    {
        try
        {
            int changes = adapter.Update(dataSet, "Customers");
            Console.WriteLine($"✅ {changes} change(s) saved to
database!");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"❌ Error saving: {ex.Message}");
        }
    }

    // 🔄 Discard changes
    public void DiscardChanges()
    {
        dataSet.RejectChanges();
        Console.WriteLine("↩️ Changes discarded!");
    }

    // 📊 Display customers
    public void DisplayCustomers()
    {
        DataTable dt = dataSet.Tables["Customers"];
        Console.WriteLine("\n📋 Customer List:");
        Console.WriteLine("".PadRight(50, '-'));

        foreach (DataRow row in dt.Rows)
        {
            if (row.RowState != DataRowState.Deleted)
            {
                Console.WriteLine($"{row["CustomerID"]}:
{row["Name"]} - {row["Email"]}");
            }
        }
    }
}

// 🎯 Usage
var manager = new CustomerManager(connectionString);
manager.LoadCustomers();
manager.AddCustomer("John Doe", "john@example.com", "123-456-7890");
manager.UpdateCustomer(5, "newemail@example.com");
manager.DeleteCustomer(3);
manager.DisplayCustomers();
manager.SaveChanges(); // Sync to database
```

📝 Summary

✓ **Key Takeaways**

🎯 Core Concepts

- ✅ Works with **cached data** in memory
- ✅ Connection opens **briefly** then closes

- ✅ **Highly scalable** design
- ✅ Enables **offline** scenarios
- ✅ Supports **complex relationships**

🧩 Core Components

- 💾 **DataSet** - In-memory cache
- 📊 **DataTable** - Table structure
- 🌉 **DataAdapter** - Bridge to database
- 👁️ **DataView** - Filtered view
- 🔗 **DataRelation** - Table relationships

⚖️ When to Use

- ✅ Complex data manipulation
- ✅ Offline work required
- ✅ Multiple users/scalability
- ✅ Batch updates
- ❌ Simple sequential reads
- ❌ Real-time data critical

> 💡 **Remember**
>
> *"Disconnected Architecture is all about **minimizing database connections** while maximizing **flexibility** and **scalability**. Perfect for modern applications that need to work offline and handle multiple concurrent users efficiently!"*

✏️ 👤 Author Information

Abdullah Ali

📞 **Contact: +201012613453**