



Explore | Expand | Enrich

IT Software Solutions for Business (ITSSB) Test Projects

Ethnus
Mar 2025

ITSSB Test Projects.....	3
College Level (2.5 hours).....	3
State Level (3.5 hours).....	8
National Level (5 hours).....	12



Explore | Expand | Enrich

ITSSB Test Projects

College Level (2.5 hours)

- **Objective:** Create a simple .NET Windows Forms application to perform basic Create, Read, Update, and Delete (CRUD) operations on a single database table (e.g., 'Products').
- **Scenario:** "Innovate Startups" needs a very basic tool for internal staff to manage their initial product list.
- **Prerequisites:**
 - Windows PC with Visual Studio (Community Edition sufficient) and .NET Desktop Development workload installed.
 - SQL Server Express LocalDB (usually installed with VS) OR SQLite and DB Browser for SQLite.
 - SQL Server Management Studio (SSMS) if using LocalDB.
 - Provided SQL script: `college_schema.sql` (Creates a simple database like `InnovateDB_College` and a `Products` table with columns `ProductID` (int, PK), `ProductName` (varchar), `Price` (decimal), `StockQuantity` (int). Includes sample `INSERT` statements).
- **Detailed Steps/Modules:**
 - **Module 1: Database Setup (Approx. 30 mins)**
 - 1.1. **If using SQL Server LocalDB:** Open SSMS, connect to `(localdb)\MSSQLLocalDB`. Create a new database named `InnovateDB_College`. Open the provided `college_schema.sql` file and execute it against the `InnovateDB_College` database to create the `Products` table and insert sample data.
 - 1.2. **If using SQLite:** Use DB Browser for SQLite to create a new database file (e.g., `InnovateDB_College.sqlite`). Open the `college_schema.sql` file (it might need slight syntax adjustments for SQLite - e.g., `INTEGER PRIMARY KEY` instead of `INT IDENTITY PRIMARY KEY`) and execute it to create the table and data. Place the

`.sqlite` file in a known location accessible by your application (e.g., near the executable).

- 1.3. **Verify:** Browse the `Products` table using SSMS or DB Browser to ensure the table exists and contains the sample data. Note the connection details (Server name/DB name for LocalDB, or the full path to the `.sqlite` file).

- **Module 2: Windows Forms UI Design (Approx. 45 mins)**

- 2.1. Create a new C# Windows Forms Application project in Visual Studio (e.g., `InnovateProductManager`).
- 2.2. Design the main form (`Form1.cs` or rename it). Add the following controls from the Toolbox:
 - Labels and TextBoxes for `ProductID` (read-only), `ProductName`, `Price`, `StockQuantity`. Name the TextBoxes appropriately (e.g., `txtProductID`, `txtProductName`, `txtPrice`, `txtStockQuantity`).
 - Buttons for "Load Products", "Add New", "Update Selected", "Delete Selected". Name them (e.g., `btnLoad`, `btnAdd`, `btnUpdate`, `btnDelete`).
 - A `DataGridView` control to display the list of products. Name it `dgvProducts`. Configure its columns later or allow auto-generation.

- **Module 3: Database Connection & Read Operation (Approx. 45 mins)**

- 3.1. **Connection String:** Define the database connection string in your C# code (or `App.config`).
 - *LocalDB Example:*

```
Server=(localdb)\MSSQLLocalDB;Database=InnovateDB_College;Integrated Security=True; (or
Trusted_Connection=True;)
```
 - *SQLite Example: Data*

```
Source=C:\path\to\your\InnovateDB_College.sqlite;Version=3;
```

- **3.2. Load Products Logic:** Create an event handler for the `btnLoad` button's `Click` event.
- **3.3.** Inside the handler, write ADO.NET code (using `System.Data.SqlClient` for SQL Server or `System.Data.SQLite` for SQLite - add NuGet package if needed):
 - Create a `SqlConnection` / `SQLiteConnection` object using the connection string.
 - Create a `SqlCommand` / `SQLiteCommand` with the SQL query:


```
SELECT ProductID, ProductName, Price, StockQuantity FROM Products.
```
 - Create a `SqlDataAdapter` / `SQLiteDataAdapter` and a `DataTable`.
 - Open the connection, fill the `DataTable` using the adapter, and close the connection (ideally using `using` statements for connection and command objects).
 - Set the `DataSource` property of the `dgvProducts` `DataGridView` to the filled `DataTable`.
 - Make the `ProductID` column in the `DataGridView` read-only.
- **Module 4: Create, Update, Delete Operations (Approx. 60 mins)**
 - **4.1. Display Selected Product:** Add an event handler for the `dgvProducts`'s `SelectionChanged` or `CellClick` event. When a row is selected, populate the TextBoxes (`txtProductName`, `txtPrice`, `txtStockQuantity`, `txtProductID`) with the data from the selected row. Disable `txtProductID`.
 - **4.2. Add New Logic:** Create an event handler for `btnAdd`.
 - Read values from `txtProductName`, `txtPrice`, `txtStockQuantity`. Perform basic validation (e.g., ensure price/quantity are valid numbers).
 - Create a SQL `INSERT` command with parameters (`INSERT INTO Products (ProductName, Price, StockQuantity) VALUES (@Name, @Price, @Qty)`).

- Create connection and command objects. Add parameters (`cmd.Parameters.AddWithValue(...)`).
- Open connection, execute the command (`cmd.ExecuteNonQuery()`), close connection.
- Refresh the DataGridView by calling the `btnLoad`'s click handler logic again. Add basic error handling (e.g., `try-catch`).
- **4.3. Update Selected Logic:** Create an event handler for `btnUpdate`.
 - Read values from all TextBoxes, including `txtProductID`. Validate input.
 - Create a SQL `UPDATE` command with parameters (`UPDATE Products SET ProductName = @Name, Price = @Price, StockQuantity = @Qty WHERE ProductID = @ID`).
 - Create connection, command, add parameters (including `@ID` from `txtProductID`).
 - Execute the command and refresh the grid. Handle errors. Ensure a product is selected first.
- **4.4. Delete Selected Logic:** Create an event handler for `btnDelete`.
 - Get the `ProductID` from `txtProductID` (or the selected grid row). Add a confirmation message box (`MessageBox.Show("Are you sure?", "Confirm Delete", MessageBoxButtons.YesNo)`).
 - If confirmed 'Yes', create a SQL `DELETE` command (`DELETE FROM Products WHERE ProductID = @ID`).
 - Create connection, command, add `@ID` parameter.
 - Execute and refresh the grid. Handle errors. Ensure a product is selected.

● **Deliverables:**

- Submit the complete Visual Studio project folder (including `.sln`, `.csproj`, `.cs`, `.designer.cs` files, and the `.sqlite` file if used).

- OR Submit the compiled executable along with any required files (like the `.sqlite` DB).
- **Verification:**
 - Run the application.
 - Verify "Load Products" displays initial data in the grid.
 - Verify selecting a row populates the text boxes.
 - Verify "Add New" inserts a new product into the database and grid.
 - Verify "Update Selected" modifies the product details in the database and grid.
 - Verify "Delete Selected" removes the product after confirmation.

Explore | Expand | Enrich

State Level (3.5 hours)

- **Objective:** Build a .NET WinForms application that manages related data from two database tables (e.g., 'Customers' and 'Orders'), implements data validation, and displays simple related information.
- **Scenario:** "Innovate Startups" needs a tool to view customer information and their associated orders. Basic data entry validation is required.
- **Prerequisites:**
 - Same as College Level (VS, .NET, DB engine, SSMS/DB Browser).
 - Provided SQL script: `state_schema.sql` (Creates tables like `Customers` (`CustomerID`, `Name`, `Email`) and `Orders` (`OrderID`, `CustomerID` (FK), `OrderDate`, `Amount`). Includes sample data and foreign key relationship).
- **Detailed Steps/Modules:**
 - **Module 1: Database Setup & Project Structure (Approx. 30 mins)**
 - 1.1. Set up the database (`InnovateDB_State`) using `state_schema.sql` (similar to College Step 1). Verify tables (`Customers`, `Orders`) and the foreign key relationship exist. Note connection details.
 - 1.2. Create a new C# Windows Forms Application project (e.g., `InnovateOrderViewer`).
 - **Module 2: UI Design for Related Data (Approx. 60 mins)**
 - 2.1. Design the main form. Consider using:
 - A `ComboBox` or `ListBox` control (`cmbCustomers` or `lstCustomers`) to display customer names.
 - A `DataGridView` (`dgvOrders`) to display orders for the selected customer.
 - `TextBoxes` (potentially read-only) to display details of the selected customer (`txtCustomerID`, `txtCustomerName`, `txtCustomerEmail`).

- TextBoxes/DateTimePicker for adding/editing Order details (`txtOrderID`, `dtpOrderDate`, `txtOrderAmount`). Make `txtOrderID` read-only.
- Buttons like "Load Customers", "Add Order", "Update Order", "Delete Order".
- 2.2. (Alternative UI): Use a `TabControl` with two tabs: "Customers" (showing customer grid/details) and "Orders" (showing order grid/details, perhaps filterable by customer). Choose one UI approach.
- **Module 3: Displaying Related Data (Approx. 60 mins)**
 - 3.1. **Load Customers:** Implement the "Load Customers" button logic. Fetch data from the `Customers` table (`SELECT CustomerID, Name FROM Customers ORDER BY Name`) and populate the `cmbCustomers` ComboBox (setting `DisplayMember` to "Name" and `ValueMember` to "CustomerID") or `lstCustomers` ListBox.
 - 3.2. **Display Customer Details:** Add an event handler for the `cmbCustomers SelectedIndexChanged` (or `lstCustomers SelectedIndexChanged`) event. When a customer is selected:
 - Get the selected `CustomerID` (from `SelectedValue`).
 - Fetch the full details for that customer (`SELECT Name, Email FROM Customers WHERE CustomerID = @ID`).
 - Display the details in the corresponding TextBoxes (`txtCustomerName`, `txtCustomerEmail`). Store the selected `CustomerID` (e.g., in `txtCustomerID` or a variable).
 - **Crucially:** Trigger the logic to load orders for this selected customer (call the function from step 3.3).
 - 3.3. **Load Orders for Selected Customer:** Create a function `LoadOrders(int customerId)` that:
 - Takes `customerId` as input.
 - Fetches orders for that customer (`SELECT OrderID, OrderDate, Amount FROM Orders WHERE CustomerID = @CustID`). Use parameters.

- Populates the `dgvOrders` DataGridView with the results. Clear previous orders first. Handle the case where a customer has no orders.
- **Module 4: Order Management & Data Validation (Approx. 75 mins)**
 - **4.1. Display Selected Order:** Add `SelectionChanged` event handler for `dgvOrders`. Populate the order detail controls (`txtOrderID`, `dtpOrderDate`, `txtOrderAmount`) when an order row is selected.
 - **4.2. Add New Order:** Implement `btnAddOrder` logic.
 - Ensure a customer is selected first (check `cmbCustomers.SelectedValue`).
 - Read `OrderDate` and `Amount` from controls.
 - **Data Validation:** Before saving, validate the inputs:
 - Ensure Amount is a valid decimal number and perhaps > 0 .
 - Ensure OrderDate is a valid date.
 - Display specific, user-friendly error messages if validation fails (e.g., using `MessageBox.Show` or `ErrorProvider` control). Prevent saving if invalid.
 - If valid, perform `INSERT INTO Orders (CustomerID, OrderDate, Amount) VALUES (@CustID, @Date, @Amount)`. Use parameters.
 - Refresh the `dgvOrders` for the current customer after insert.
 - **4.3. Update/Delete Order:** Implement `btnUpdateOrder` and `btnDeleteOrder` logic similar to College level (using `UPDATE...WHERE OrderID = @OrderID` and `DELETE...WHERE OrderID = @OrderID`), ensuring an order is selected. Include input validation for Update. Refresh the `dgvOrders`.
 - **4.4. Simple Reporting Element:** Add a Label (`lblTotalAmount`) that displays the sum of the 'Amount' for the currently displayed orders in `dgvOrders`. Update this label whenever orders are loaded or modified. Loop through the `DataTable` or `DataGridView` rows to calculate the sum.

- **Deliverables:**

- Submit the complete Visual Studio project folder OR compiled executable + necessary files.

- **Verification:**

- Run the application.
- Verify customers load correctly.
- Verify selecting a customer displays their details and filters the orders shown in the grid.
- Verify adding a new order fails with a clear message if validation rules (e.g., non-numeric amount) are broken.
- Verify adding/updating/deleting orders works correctly and updates the grid.
- Verify the total amount label updates correctly when the displayed orders change.

Explore | Expand | Enrich

National Level (5 hours)

- **Objective:** Develop a more robust .NET application managing multiple related tables (e.g., 'Suppliers', 'Products', 'InventoryTransactions') with search/filtering, basic transaction handling, potential use of a simple Data Access Layer pattern, and data export functionality.
- **Scenario:** "Innovate Startups" needs an inventory management tool to track products, suppliers, and stock movements (receiving stock, shipping stock). Data integrity and basic reporting are important.
- **Prerequisites:**
 - Same as State Level (VS, .NET, DB engine, SSMS/DB Browser).
 - Provided SQL script: `national_schema.sql` (Creates tables like `Suppliers (SupplierID, Name)`, `Products (ProductID, Name, SupplierID (FK), CurrentStock)`, `InventoryTransactions (TransactionID, ProductID (FK), ChangeQuantity (positive for received, negative for shipped), TransactionDate)`. Includes sample data and relationships).
- **Detailed Steps/Modules:**
 - **Module 1: Database Setup & Project Structure (Approx. 30 mins)**
 - 1.1. Set up the database (`InnovateDB_National`) using `national_schema.sql`. Verify tables and relationships. Note connection details.
 - 1.2. Create a new C# Windows Forms Application project (e.g., `InnovateInventory`).
 - 1.3. **(Optional) Data Access Layer (DAL):** Create separate C# classes for handling database interactions (e.g., `SupplierRepository.cs`, `ProductRepository.cs`, `InventoryRepository.cs`). These classes would contain the ADO.NET code (connection, command, data retrieval/modification logic) encapsulated in methods (e.g., `GetAllSuppliers()`, `GetProductsBySupplier(int supplierId)`, `AddInventoryTransaction(...)`). The UI forms

will call methods in these repository classes instead of having raw ADO.NET code directly in the form's code-behind. *If DAL not taught, keep ADO.NET code in forms but aim for cleaner separation.*

- **Module 2: UI Design - Multi-View & Search (Approx. 75 mins)**

- 2.1. Design the main UI, likely using a **TabControl** or separate forms/panels for different functions:
 - **Suppliers View:** Grid to display suppliers. Fields/Buttons to add/edit suppliers.
 - **Products View:** Grid to display products. ComboBox to filter by Supplier. TextBox for searching product name. Fields/Buttons to add/edit products (include ComboBox to select Supplier). Display **CurrentStock** (read-only here).
 - **Inventory Transactions View:** Grid to display transaction history. Fields/Controls to record a new transaction (ComboBox to select Product, NumericUpDown for Quantity, RadioButtons/ComboBox for Type 'Receive'/'Ship', Button to 'Record Transaction').

- **Module 3: Implementing Views & Relationships (Approx. 90 mins)**

- 3.1. Implement the **Suppliers View** with basic CRUD functionality for the **Suppliers** table (similar to College level but using DAL methods if implemented).
- 3.2. Implement the **Products View**:
 - Load Suppliers into the filter ComboBox.
 - Implement filtering: When supplier filter or search text changes, fetch and display matching products in the grid (e.g., `SELECT ... FROM Products WHERE (@SupplierID = 0 OR SupplierID = @SupplierID) AND ProductName LIKE @SearchText`). Use parameters. Remember to handle the "All Suppliers" case (e.g., pass 0 or null for @SupplierID).

- Implement Add/Edit Product functionality, ensuring the correct **SupplierID** (from a ComboBox populated with Suppliers) is saved. **CurrentStock** is not directly edited here.
- 3.3. Implement the **Inventory Transactions View** (Display only for now): Load Products into the 'Select Product' ComboBox. Display transaction history in the grid, perhaps join with Products table to show Product Name (`SELECT T.*, P.ProductName FROM InventoryTransactions T JOIN Products P ON T.ProductID = P.ProductID ORDER BY TransactionDate DESC`).
- **Module 4: Transaction Handling & Stock Update (Approx. 75 mins)**
 - 4.1. Implement the "Record Transaction" button logic in the Inventory Transactions View.
 - 4.2. Get selected **ProductID**, **Quantity**, and **TransactionType** ('Receive'/'Ship'). Validate inputs.
 - 4.3. **Transaction Logic:** This involves two steps that should succeed or fail together:
 - **Step A:** Insert a new record into the **InventoryTransactions** table (Quantity is positive for Receive, negative for Ship).
 - **Step B:** Update the **CurrentStock** in the **Products** table accordingly (`UPDATE Products SET CurrentStock = CurrentStock + @Change WHERE ProductID = @ProductID`). **@Change** would be the positive or negative quantity.
 - 4.4. **Implement using ADO.NET Transaction:**
 - Open a single **SqlConnection/SQLiteConnection**.
 - Begin a transaction (`SqlConnection tx = conn.BeginTransaction(); / SQLiteTransaction tx = conn.BeginTransaction();`).

- Create and execute the **INSERT** command (Step A), associating it with the transaction (`cmd.Transaction = tx;`).
- Create and execute the **UPDATE** command (Step B), associating it with the transaction.
- If both commands execute without error, commit the transaction (`tx.Commit();`).
- **Crucially:** Wrap the execution in a **try-catch** block. If any error occurs during execution, **rollback** the transaction (`tx.Rollback();`) and show an error message.
- Ensure the connection is closed properly in a **finally** block or using **using** statements.
- 4.5. Refresh the transactions grid and potentially update the displayed stock level in the Products view if it's visible.
- **Module 5: Simple Data Export (Approx. 30 mins)**
 - 5.1. In the **Products View**, add an "Export Products to CSV" button (`btnExport`).
 - 5.2. When clicked, fetch the *currently displayed/filtered* product data from the **Products** table (use the same logic/query as the product grid display).
 - 5.3. Use a **SaveFileDialog** control to ask the user where to save the **.csv** file.
 - 5.4. If a filename is chosen, write the product data (including headers like **ProductID, ProductName, CurrentStock, SupplierName**) to the selected file in CSV format (comma-separated values). You can manually build the string using **StringBuilder** and **File.WriteAllText** or use a simple CSV helper library if allowed. Include error handling for file writing.

● **Deliverables:**

- Submit the complete Visual Studio project folder OR compiled executable + necessary files.

- **Verification:**

- Run the application.
- Verify Supplier CRUD works.
- Verify Product filtering by supplier and name works. Verify Product CRUD works, including selecting a supplier.
- Verify recording an Inventory Transaction correctly inserts into **InventoryTransactions** AND updates **CurrentStock** in **Products**.
Test both Receive and Ship. Verify the transaction rollback works if an error is simulated (e.g., invalid ProductID during update - harder to force, but check code structure).
- Verify Transaction history displays correctly.
- Verify the "Export Products to CSV" button correctly generates a CSV file containing the filtered/displayed products with correct headers and data.

– End of Document –

Explore | Expand | Enrich