

ADO.NET

Connected Layer

CSIS 3540

Client Server Systems

Class 06

©Michael Hrybyk and others
NOT TO BE REDISTRIBUTED

Topics

- Connecting to a Database
- Creating a database in Visual Studio
- Reading and Querying a database
- Insertion and Deletion
- Using DataSet and DataTable
- Basic Transaction processing
- Backing up database tables to XML

ADO

- ADO – Active Data Objects
 - Terminology left over from COM/OLE/ActiveX
 - .NET added to use more modern methods
- ADO layers
 - Connected Layer (this lecture)
 - Fundamental building block
 - Disconnected
 - Deprecated, superseded by ...
 - Entity Framework
 - Newest and most effective

ADO Objects

Table 21-1. *The Core Objects of an ADO.NET Data Provider*

Type of Object	Base Class	Relevant Interfaces	Meaning in Life
Connection	DbConnection	IDbConnection	Provides the ability to connect to and disconnect from the data store. Connection objects also provide access to a related transaction object.
Command	DbCommand	IDbCommand	Represents a SQL query or a stored procedure. Command objects also provide access to the provider's data reader object.
DataReader	DbDataReader	IDataReader, IDataRecord	Provides forward-only, read-only access to data using a server-side cursor.
DataAdapter	DbDataAdapter	IDataAdapter, IDbDataAdapter	Transfers DataSets between the caller and the data store. Data adapters contain a connection and a set of four internal command objects used to select, insert, update, and delete information from the data store.
Parameter	DbParameter	IDataParameter, IDbDataParameter	Represents a named parameter within a parameterized query.
Transaction	DbTransaction	IDbTransaction	Encapsulates a database transaction.

802

ADO Objects and Databases

Figure 21-2 shows the big picture behind ADO.NET data providers. Note how the diagram illustrates that the *Client Assembly* can literally be any type of .NET application: console program, Windows Forms application, WPF application, ASP.NET web page, WCF service, Web API service, .NET code library, and so on.

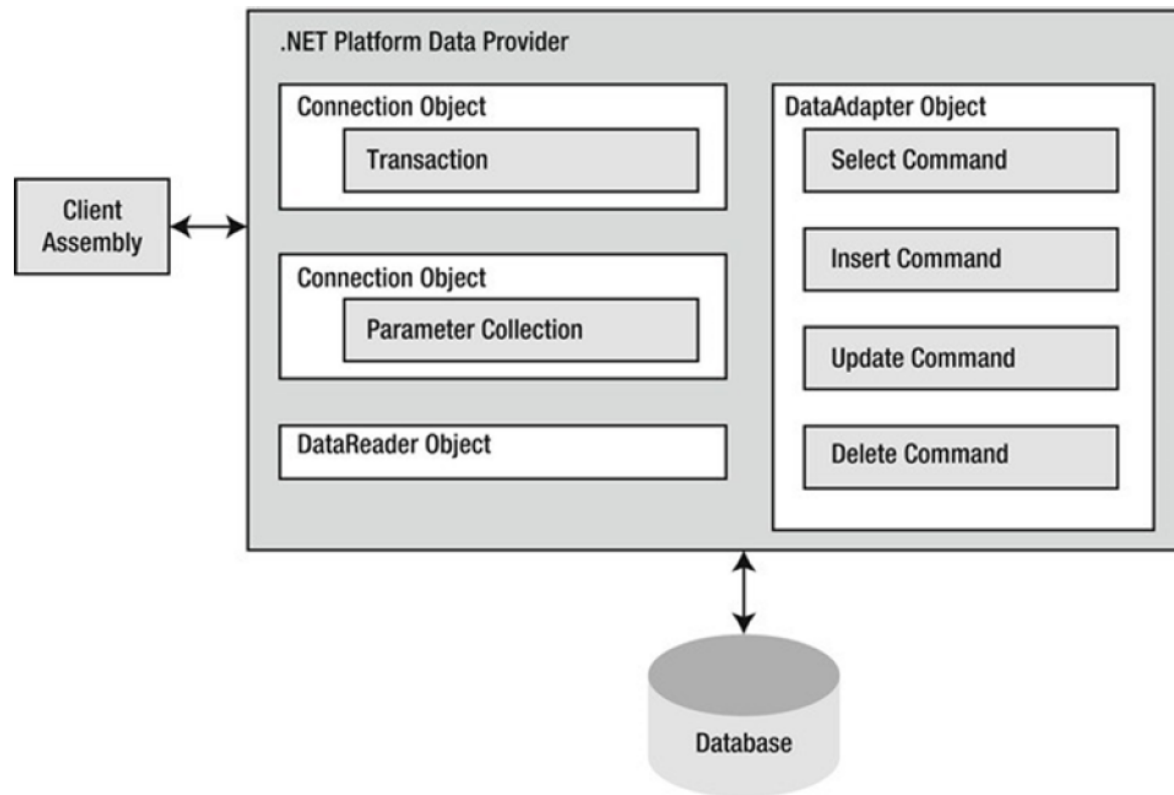


Figure 21-2. ADO.NET data providers provide access to a given DBMS

Data providers

- ADO tries to be database independent
- A data provider is a set of types defined in a given namespace that understand how to communicate with a specific type of data source.
- ADO allows the programmer to define data providers
 - All data providers are expected to implement core methods
- Benefits
 - one can program a specific data provider to access any unique features of a particular DBMS.
 - a specific data provider can connect directly to the underlying engine of the DBMS in question without an intermediate mapping layer standing between the tiers.
- Regardless of which data provider you use, each defines a set of class types that provide core functionality.

.NET data providers

The Microsoft-Supplied ADO.NET Data Providers

Microsoft's .NET distribution ships with numerous data providers, including a provider for Oracle, SQL Server, and OLE DB/ODBC-style connectivity. Table 21-2 documents the namespace and containing assembly for each Microsoft ADO.NET data provider.

Table 21-2. *Microsoft ADO.NET Data Providers*

Data Provider	Namespace	Assembly
OLE DB	System.Data.OleDb	System.Data.dll
Microsoft SQL Server LocalDb	System.Data.SqlClient	System.Data.dll
ODBC	System.Data.Odbc	System.Data.dll

Note While an Oracle provider is still being shipped with the .NET Framework, the recommendation is to use the Oracle-supplied Oracle Developer Tools for Visual Studio. In fact, if you open Server Explorer and select New Connection and then Oracle Database, Visual Studio will tell you to use the Oracle Data Tools and provide a link where they can be downloaded.

ADO.NET namespaces

Table 21-3. *Select Additional ADO.NET-Centric Namespaces*

Namespace	Meaning in Life
Microsoft.SqlServer.Server	This namespace provides types that facilitate CLR and SQL Server 2005 and later integration services.
System.Data	This namespace defines the core ADO.NET types used by all data providers, including common interfaces and numerous types that represent the disconnected layer (e.g., DataSet and DataTable).
System.Data.Common	This namespace contains types shared between all ADO.NET data providers, including the common abstract base classes.
System.Data.Sql	This namespace contains types that allow you to discover Microsoft SQL Server instances installed on the current local network.
System.Data.SqlTypes	This namespace contains native data types used by Microsoft SQL Server. You can always use the corresponding CLR data types, but the SqlTypes are optimized to work with SQL Server (e.g., if your SQL Server database contains an integer value, you can represent it using either int or SqlTypes.SqlInt32).

Note that this chapter does not examine every type within every ADO.NET namespace (that task would require a large book all by itself); however, it is quite important that you understand the types within the System.Data namespace.

Core Members of System.Data namespace

Table 21-4. *Core Members of the System.Data Namespace*

Type	Meaning in Life
Constraint	Represents a constraint for a given DataColumn object
DataColumn	Represents a single column within a DataTable object
DataRelation	Represents a parent-child relationship between two DataTable objects
DataRow	Represents a single row within a DataTable object
DataSet	Represents an in-memory cache of data consisting of any number of interrelated DataTable objects
DataTable	Represents a tabular block of in-memory data
DataTableReader	Allows you to treat a DataTable as a fire-hose cursor (forward only, read- only data access)
DataRowView	Represents a customized view of a DataTable for sorting, filtering, searching, editing, and navigation
IDataAdapter	Defines the core behavior of a data adapter object
IDataParameter	Defines the core behavior of a parameter object
IDataReader	Defines the core behavior of a data reader object
IDbCommand	Defines the core behavior of a command object
IDbDataAdapter	Extends IDataAdapter to provide additional functionality of a data adapter object
IDbTransaction	Defines the core behavior of a transaction object

You use the vast majority of the classes within System.Data when programming against the disconnected layer of ADO.NET. In the next chapter, you will get to know the details of the DataSet and its related cohorts (e.g., DataTable, DataRelation, and DataRow) and how to use them (and a related data adapter) to represent and manipulate client-side copies of remote data.

IDbConnection

- Provider methods and properties
- ConnectionString – arguments to hand to connection provider
- Open/Close
- CreateCommand – create a command object that can be filled in with SQL statements, for example
- SqlConnection class implements this interface. You will be using this!

```
public interface IDbConnection : IDisposable
{
    string ConnectionString { get; set; }
    int ConnectionTimeout { get; }
    string Database { get; }
    ConnectionState State { get; }
    IDbTransaction BeginTransaction();
    IDbTransaction BeginTransaction(IsolationLevel il);
    void ChangeDatabase(string databaseName);
    void Close();
    IDbCommand CreateCommand();
    void Open();
}
```

IDbTransaction

- Returned from IDbConnection BeginTransaction() methods
- Commit/Rollback
 - Catch an exception, if thrown, roll back the entire transaction
 - Ensures atomicity

```
public interface IDbTransaction : IDisposable
{
    IDbConnection Connection { get; }
    IsolationLevel IsolationLevel { get; }
    void Commit();
    void Rollback();
}
```

SqlCommand

- From IDbConnection.CreateCommand()
- Use CommandText to set up SQL statement
- ExecuteNonQuery() for things like delete/insert/...
- Then ExecuteReader() for connected layer invocation of select
 - Will return a set of records.

```
public interface IDbCommand : IDisposable
{
    IDbConnection Connection { get; set; }
    IDbTransaction Transaction { get; set; }
    string CommandText { get; set; }
    int CommandTimeout { get; set; }
    CommandType CommandType { get; set; }
    IDataParameterCollection Parameters { get; }
    UpdateRowSource UpdatedRowSource { get; set; }
    void Prepare();
    void Cancel();
    IDbDataParameter CreateParameter();
    int ExecuteNonQuery();
    IDataReader ExecuteReader();
    IDataReader ExecuteReader(CommandBehavior behavior);
    object ExecuteScalar();
}
```

IDataReader

- The next key interface to be aware of is IDataReader, which represents the common behaviors supported by a given data reader object.
- When you obtain an IDataReader-compatible type from an ADO.NET data provider, you can iterate over the result set in a forward-only, read-only manner.
 - Use Read() and NextResult()

```
public interface IDataReader : IDisposable, IDataRecord
{
    int Depth { get; }
    bool IsClosed { get; }
    int RecordsAffected { get; }
    void Close();
    DataTable GetSchemaTable();
    bool NextResult();
    bool Read();
}
```

IDataRecord

- returned from IDataReader (iteration)
- Actual data record, but must be cast
- Can use methods below to do so.

```
public interface IDataRecord
{
    int FieldCount { get; }
    object this[ int i ] { get; }
    object this[ string name ] { get; }
    string GetName(int i);
    string GetDataTypeName(int i);
    Type GetFieldType(int i);
    object GetValue(int i);
    int GetValues(object[] values);
    int GetOrdinal(string name);
    bool GetBoolean(int i);
    byte GetByte(int i);
    long GetBytes(int i, long fieldOffset, byte[] buffer, int bufferoffset, int length);
    char GetChar(int i);
    long GetChars(int i, long fieldoffset, char[] buffer, int bufferoffset, int length);
    Guid GetGuid(int i);
    short GetInt16(int i);
    int GetInt32(int i);
    long GetInt64(int i);
    float GetFloat(int i);
    double GetDouble(int i);
    string GetString(int i);
    Decimal GetDecimal(int i);
    DateTime GetDateTime(int i);
    IDataReader GetData(int i);
    bool IsDBNull(int i);
}
```

How it works - SimpleConnection

```
class SimpleConnectionProgram
{
    static void Main(string[] args)
    {
        WriteLine("**** Very Simple Connection Factory ****\n");

        // Read the provider key.
        string dataProviderString = ConfigurationManager.AppSettings["provider"];

        // demonstrate the use of App.config settings
        WriteLine(ConfigurationManager.AppSettings["Message"]);

        // Get a specific connection.

        IDbConnection myConnection = GetConnection(dataProviderString);

        // Open, use and close connection...

        WriteLine($"Your connection is a {myConnection?.GetType().Name ?? "unrecognized type"}");

        ReadLine();
    }

    static IDbConnection GetConnection(string dataProvider)
    {
        switch(dataProvider)
        {
            case "SqlServer":
                return (new SqlConnection()); // we will use this one a lot!!
            case "OleDb":
                return (new OleDbConnection());
            case "Odbc":
                return new OdbcConnection();
            default:
                return null;
        }
    }
}
```

Database procedure

- Open a connection to the database (IDbConnection)
 - See App.Config for connection string
 - Can also be set in Properties->Debug for an SQL project
- Create a command (IDbCommand)
- Execute a command to read data or issue a non-query
 - ExecuteReader() method returns IDataReader
 - Result of a SELECT statement
 - Use IDataReader.Read() and IDataReader.Next()
 - Contains a set of records (IDataRecord) that you can iterate through
 - ExecuteNonQuery() method executes DDL, DML
 - CREATE, DROP, INSERT, DELETE, TRUNCATE
- For SQL Server we will be using SqlConnection, SqlCommand, SqlDataReader,

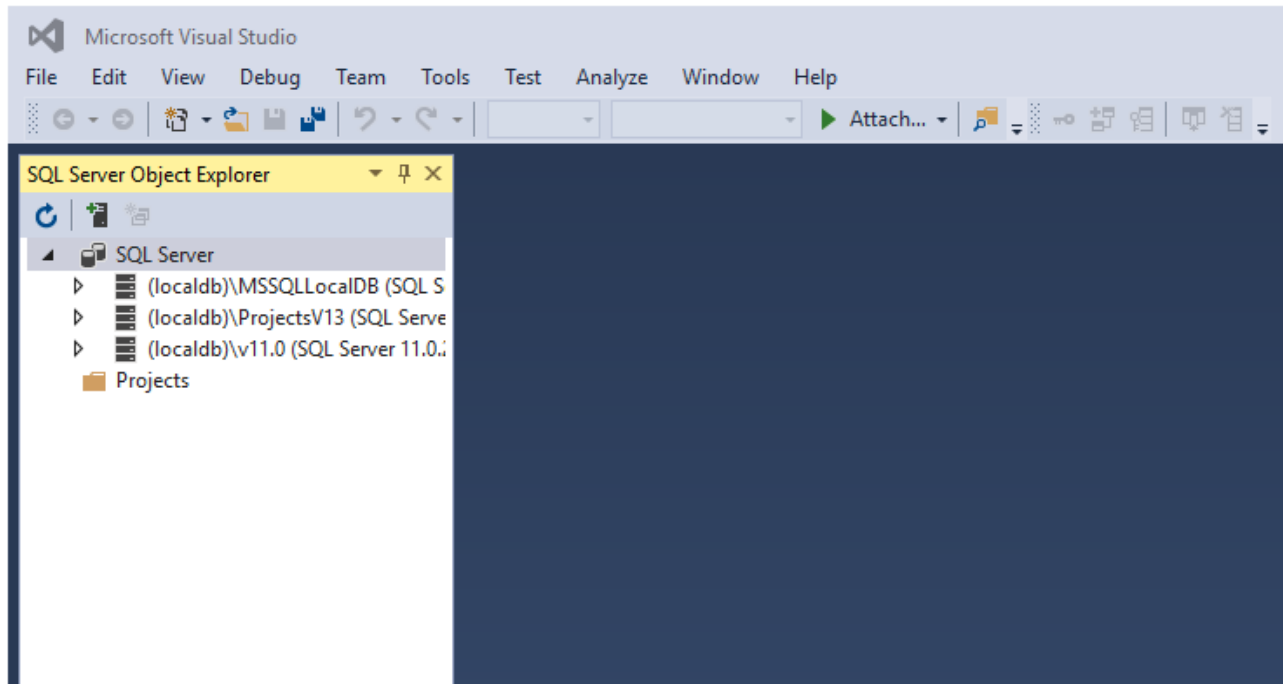
App.config

- Below is the standard used to connect to AutoLot database
- The key is AutoLotSqlProvider
 - This is associated with the connectionString
 - To change the database or the server, just change the connectionString in App.config
 - No need to recompile

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.7" />
  </startup>
  <connectionStrings>
    <add name="AutoLotSqlProvider" connectionString="Data Source=(localdb)\MSSQLLocalDB;Integrated
Security=SSPI;Initial Catalog=AutoLot"/>
  </connectionStrings>
</configuration>
```

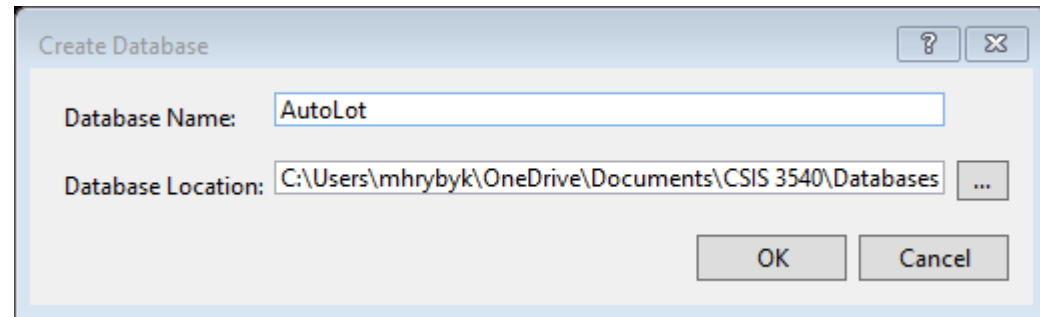
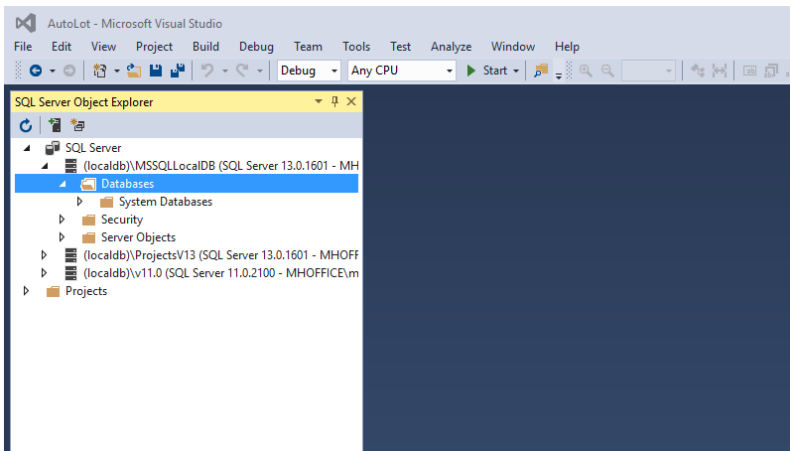
Creating the AutoLot database

- Bring up Visual Studio, open SQL Server Object Explorer
 - Under View menu



Expand (localdb)\MSSQLLocalDB

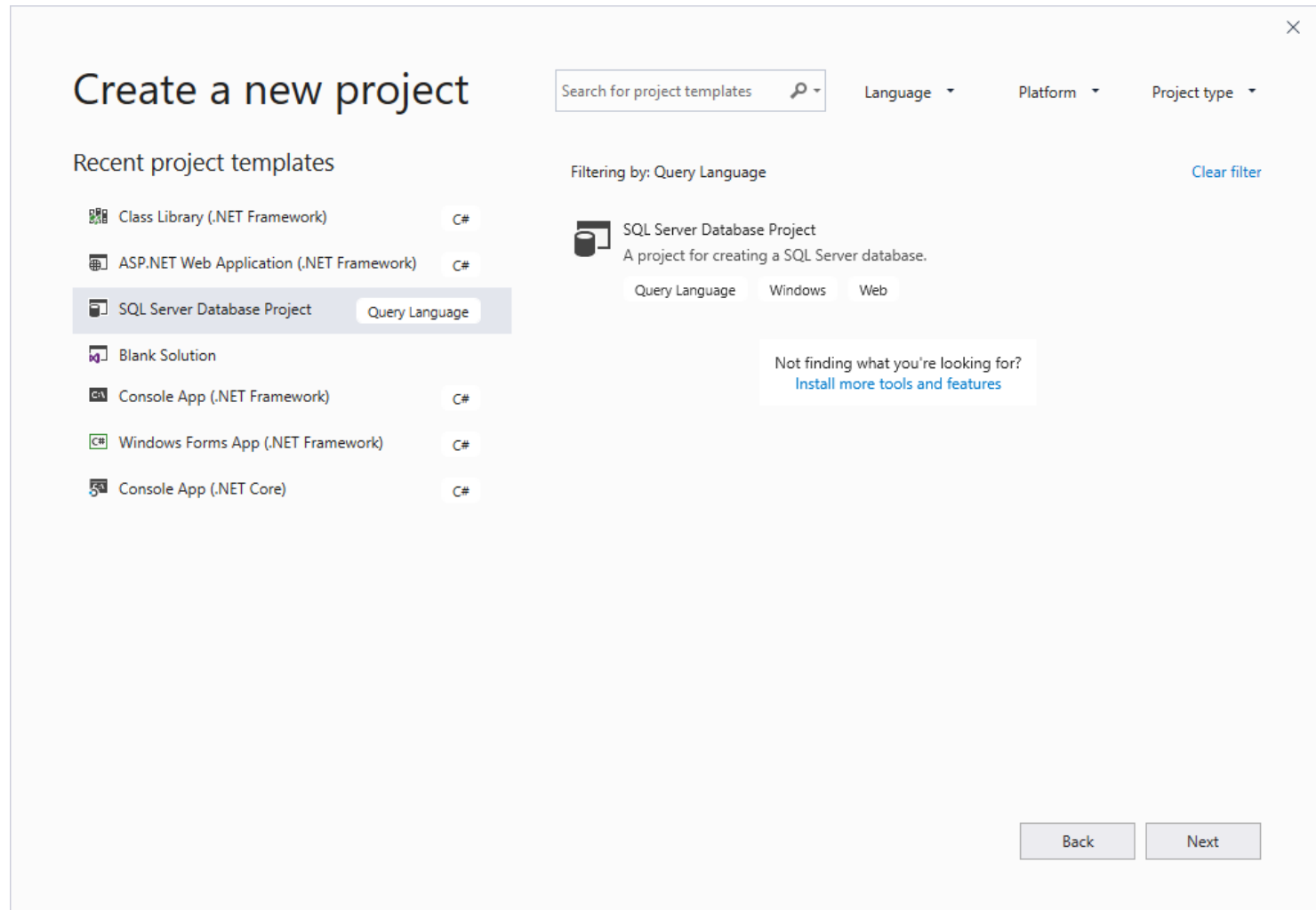
- Right click on Databases
- Add Database - AutoLot
 - Use a personal directory for location – USB or OneDrive (preferred)
 - Best to NOT store within the project.



Now create a DB project

- Open VS
- Create a new project
- Select SQL Server Database project
- We will create SQL code to create tables, views, stored procedures, etc. within the project

Create a SQL Server Project: AutoLot



Create a SQL Server Project: AutoLot

×

Configure your new project

SQL Server Database Project Query Language Windows Web

Project name

Location

...

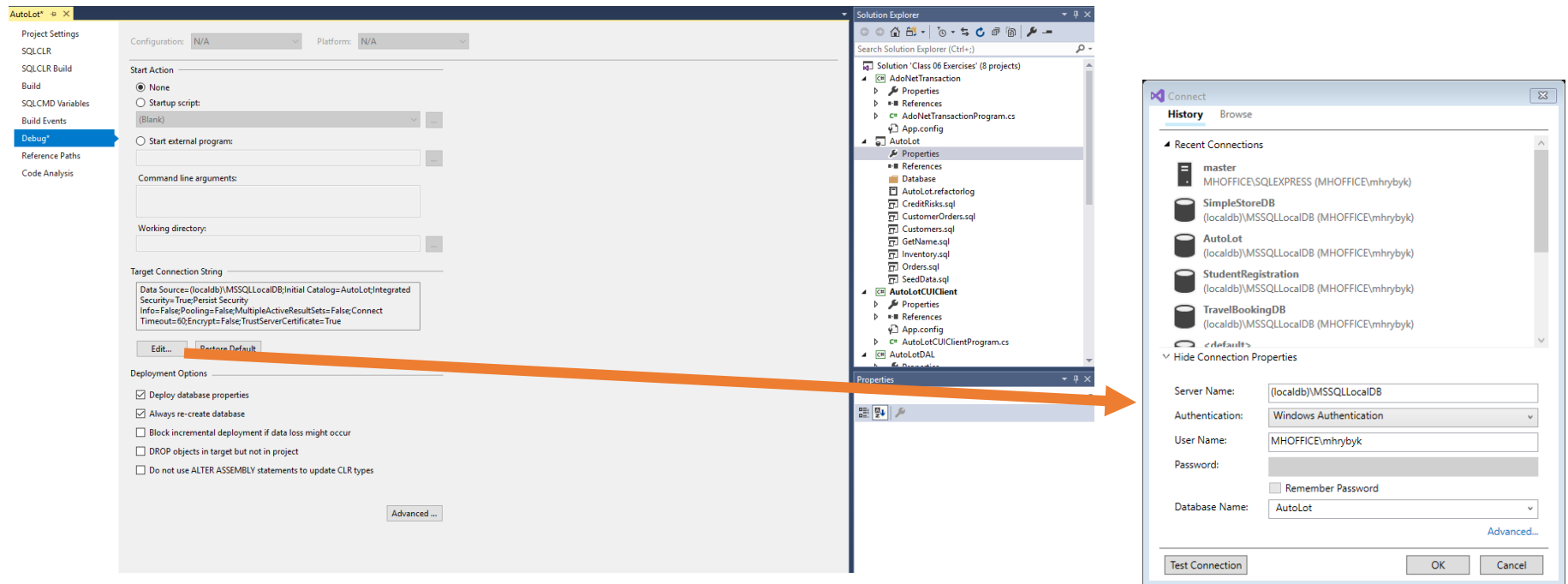
Solution name ⓘ

☐ Place solution and project in the same directory

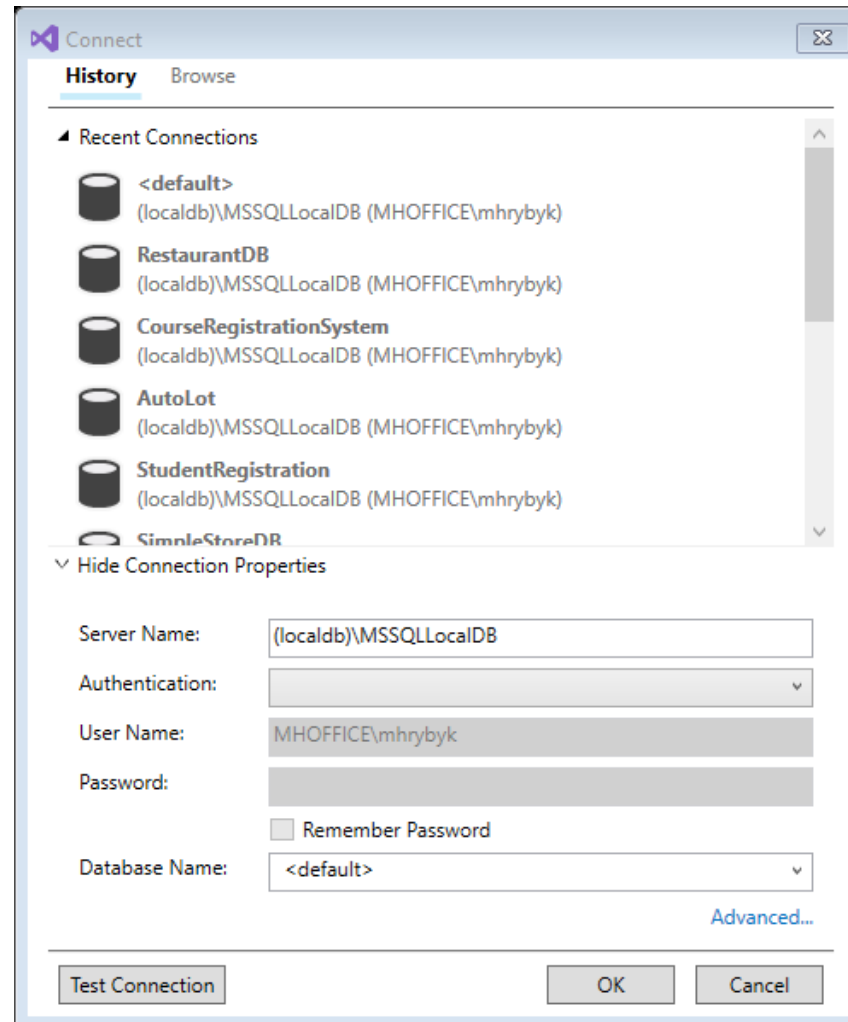
Back Create

Double click on AutoLot Properties

- Go to the Debug panel. Modify the connection string by clicking on Edit
 - Data Source=(localdb)\MSSQLLocalDB which is entered in the Server Name textbox from the Edit button.
- Whenever you modify a .sql file, and hit start, the statements in the file will be executed.
 - With some differences: CREATE will change to ALTER if need be
 - Make sure the AutoLot SQL project is Set As StartUp Project (right click on AutoLot Project)
 - Make sure Always re-create the Database is set
- We now have an Autolot database project which holds sql files, and can modify the AutoLot database.



Modify Debug Properties



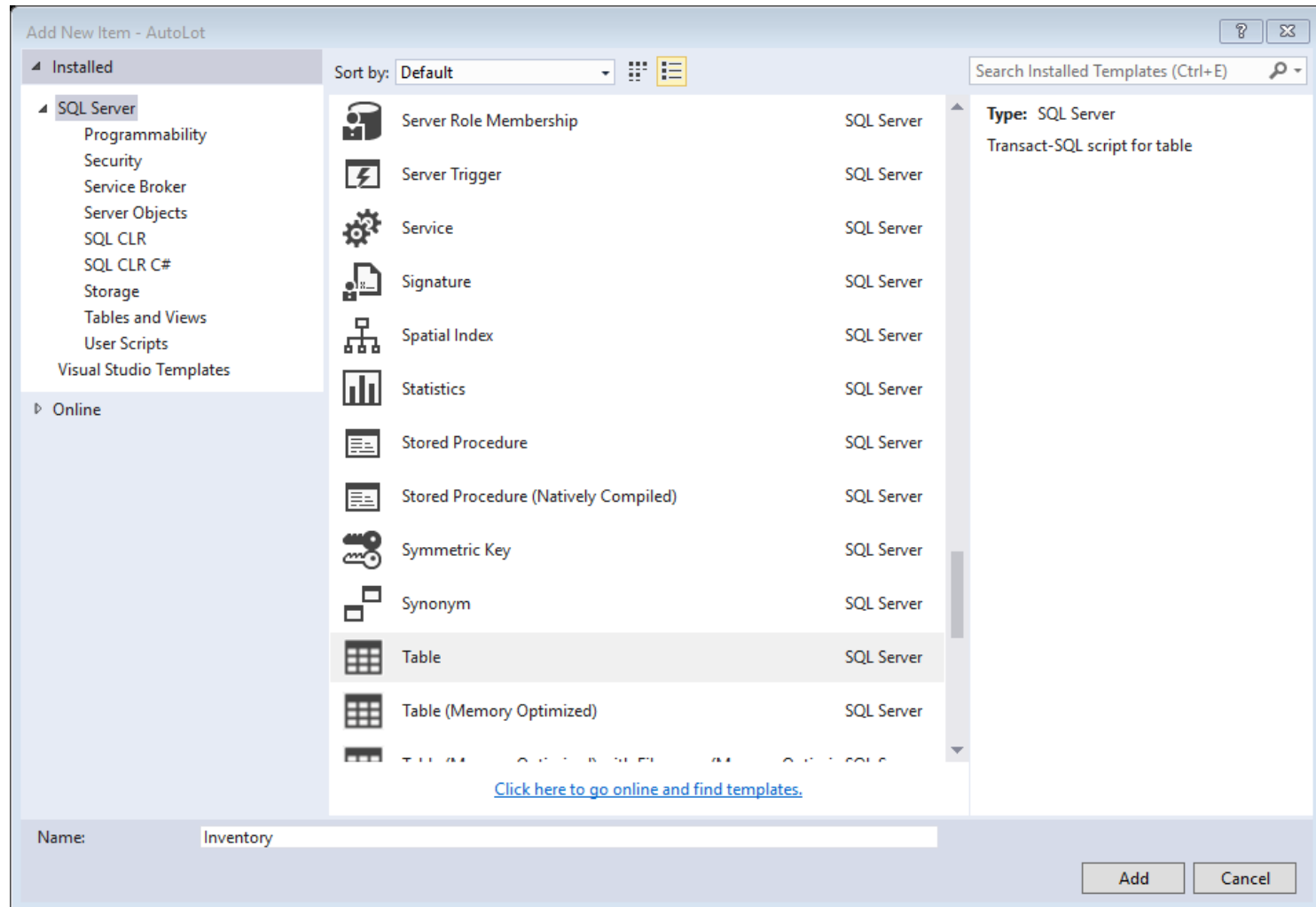
Data Connections

- Can also add a database as a connection OR as a database file
- Open up Server Browser and add a new data connection.
 - Can do this with existing AutoLot database

Now add tables, etc

- In the AutoLot **PROJECT** add the following tables
 - Inventory
 - Customers
 - Orders

Add the Inventory Table



Add Inventory fields

- Can use Designer or T-SQL pane
- Note use of Add Key/Constraint in upper right pane

The screenshot displays the SQL Server Enterprise Designer interface for the 'Inventory' table. The 'Design' tab is active, showing a table with four columns: CarId, Make, Color, and Name. The 'CarId' column is marked as the primary key. The 'T-SQL' tab is also visible, showing the corresponding CREATE TABLE script.

Name	Data Type	Allow Nulls	Default	Identity
CarId	int	<input type="checkbox"/>		<input checked="" type="checkbox"/>
Make	nvarchar(50)	<input checked="" type="checkbox"/>		<input type="checkbox"/>
Color	nvarchar(50)	<input checked="" type="checkbox"/>		<input type="checkbox"/>
Name	nvarchar(50)	<input checked="" type="checkbox"/>		<input type="checkbox"/>

Keys (1)
PK_Inventory (Primary Key, Clustered: CarId)

Check Constraints (0)
Indexes (0)
Foreign Keys (0)
Triggers (0)

```
1 CREATE TABLE [dbo].[Inventory]
2 (
3     [CarId] INT NOT NULL IDENTITY,
4     [Make] NVARCHAR(50) NULL,
5     [Color] NVARCHAR(50) NULL,
6     [Name] NVARCHAR(50) NULL,
7     CONSTRAINT [PK_Inventory] PRIMARY KEY ([CarId])
8 )
9
```

Solution Explorer

- AutoLot
 - Properties
 - References
 - Database
 - AutoLot.refactorlog
 - CreditRisks.sql
 - CustomerOrders.sql
 - Customers.sql
 - GetName.sql
 - Inventory.sql**
 - Orders.sql
 - SeedData.sql

Properties

Inventory.sql File Properties

Property	Value
ANSI Nulls	Project Default
Build Action	Build
Copy to Output Directory	Do not copy

Database publish

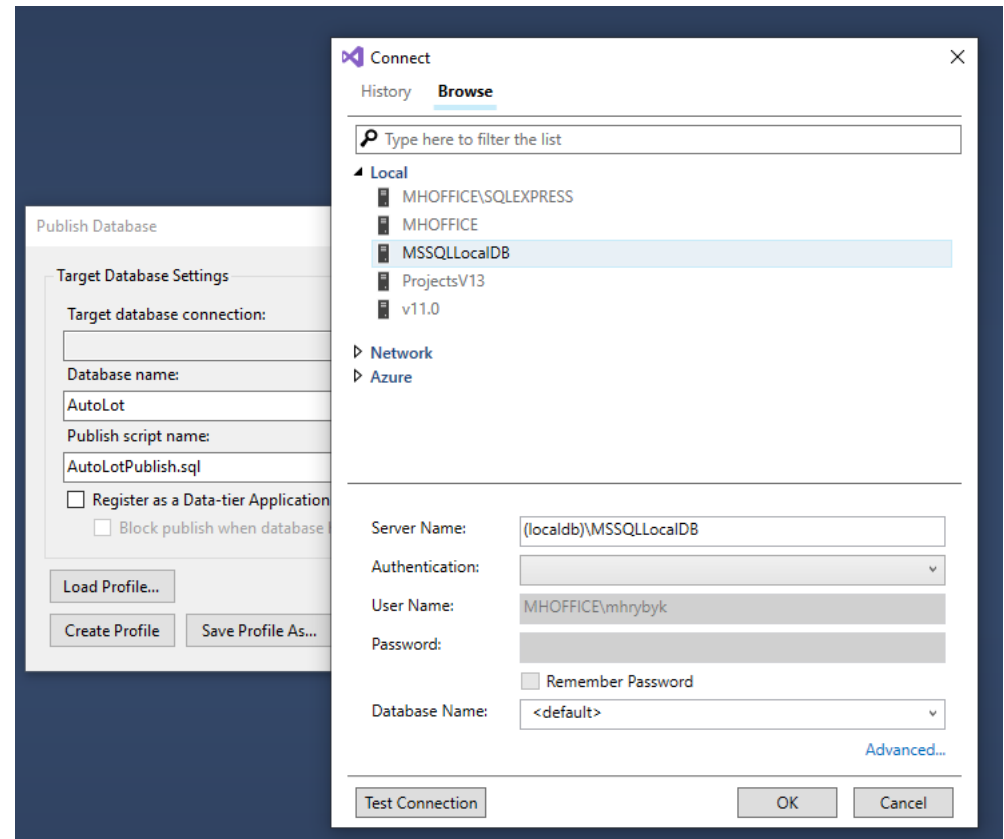
- Right click on AutoLot project
 - Select Publish
- Make sure the database name is correct in the form that pops up
- Name the script AutoLot.sql
- Click edit the target database connection

The screenshot shows the 'Publish Database' dialog box. The title bar is 'Publish Database' with a help icon and a close icon. The main area is titled 'Target Database Settings'. It contains the following fields and controls:

- 'Target database connection:' with an empty text box, an 'Edit...' button (highlighted with a blue border), and a 'Clear' button.
- 'Database name:' with a text box containing 'AutoLot'.
- 'Publish script name:' with a text box containing 'AutoLot.sql'.
- A checkbox labeled 'Register as a Data-tier Application' which is unchecked.
- A checkbox labeled 'Block publish when database has drifted from registered version' which is unchecked.
- An 'Advanced...' button.
- A 'Load Profile...' button.
- A row of buttons at the bottom: 'Create Profile', 'Save Profile As...', 'Generate Script', 'Publish', and 'Cancel'.

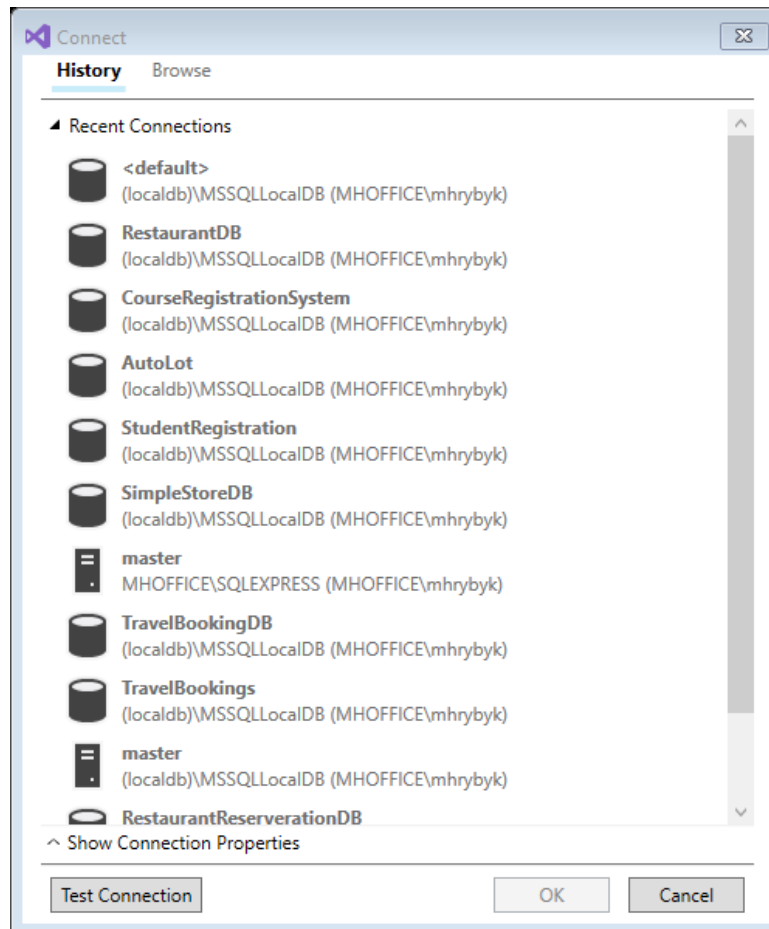
Database Publish

- Edit the target database connection
- Choose MSSQLLocalDB – no need for database name
- Click Create Profile, and the script will appear under the project
- Click Publish to create the database
- Any time you want to update or recreate the database, just click on the publish script within the project



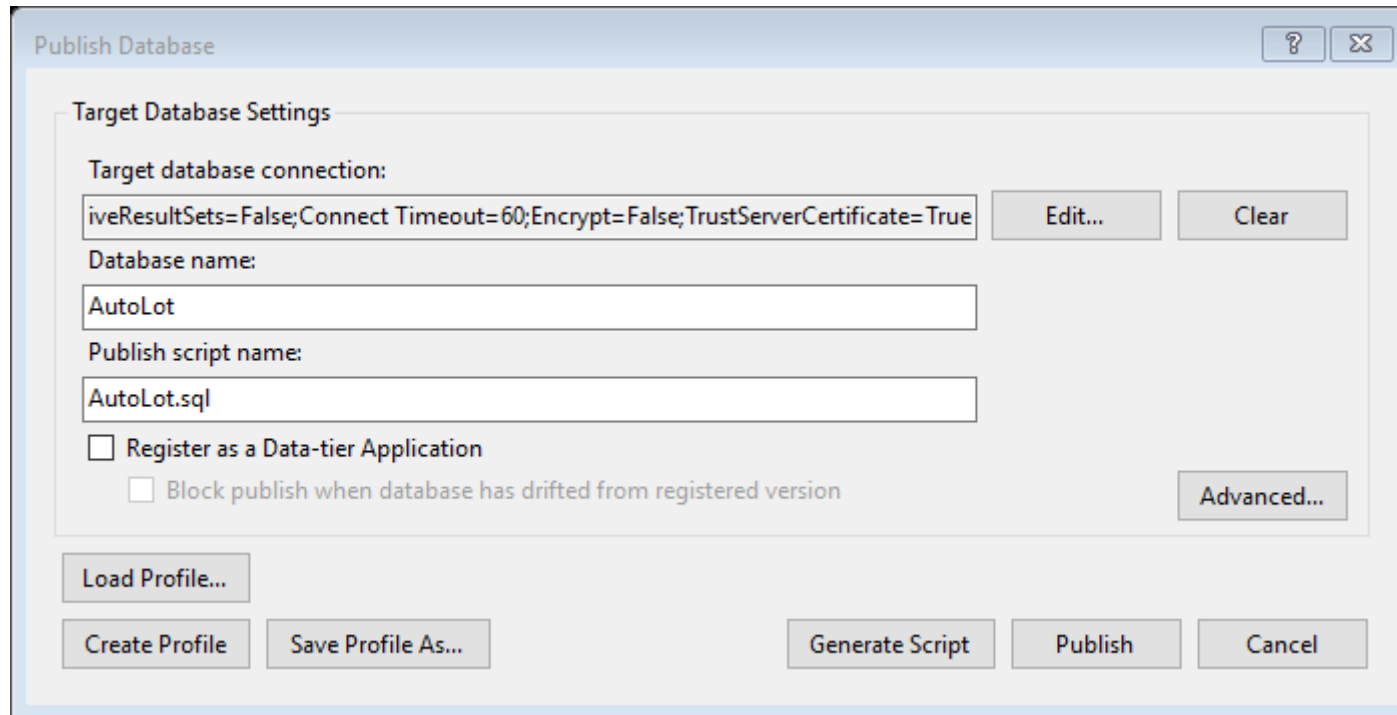
Database Publish

- Select default, or (localdb)\MSSQLLocalDB



Database Publish

- Click Create Profile then Publish and the database will be created.
- In Advanced, make sure you always check Recreate Database

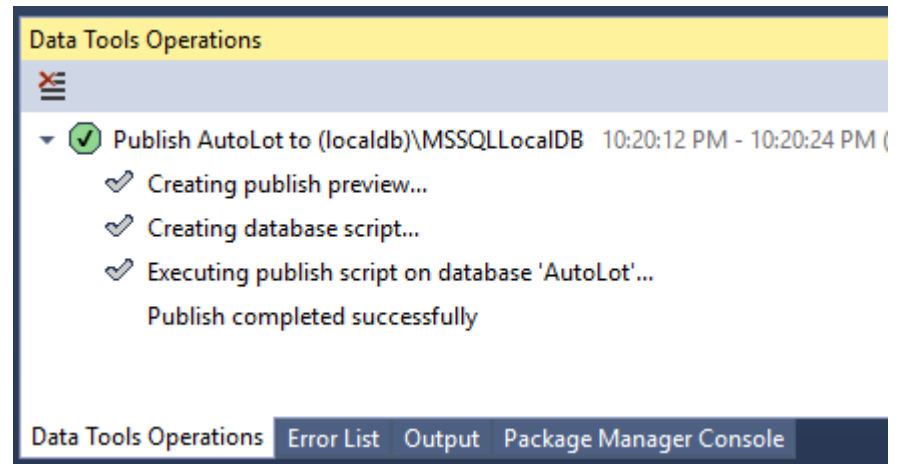
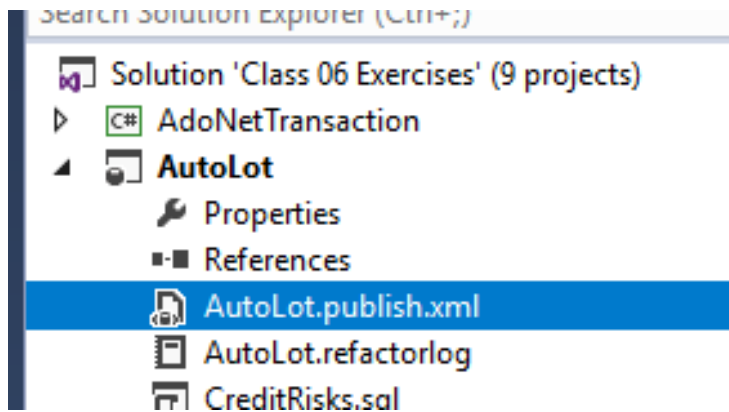


The screenshot shows the 'Publish Database' dialog box. The title bar is 'Publish Database' with a help icon and a close icon. The main area is titled 'Target Database Settings'. It contains the following fields and controls:

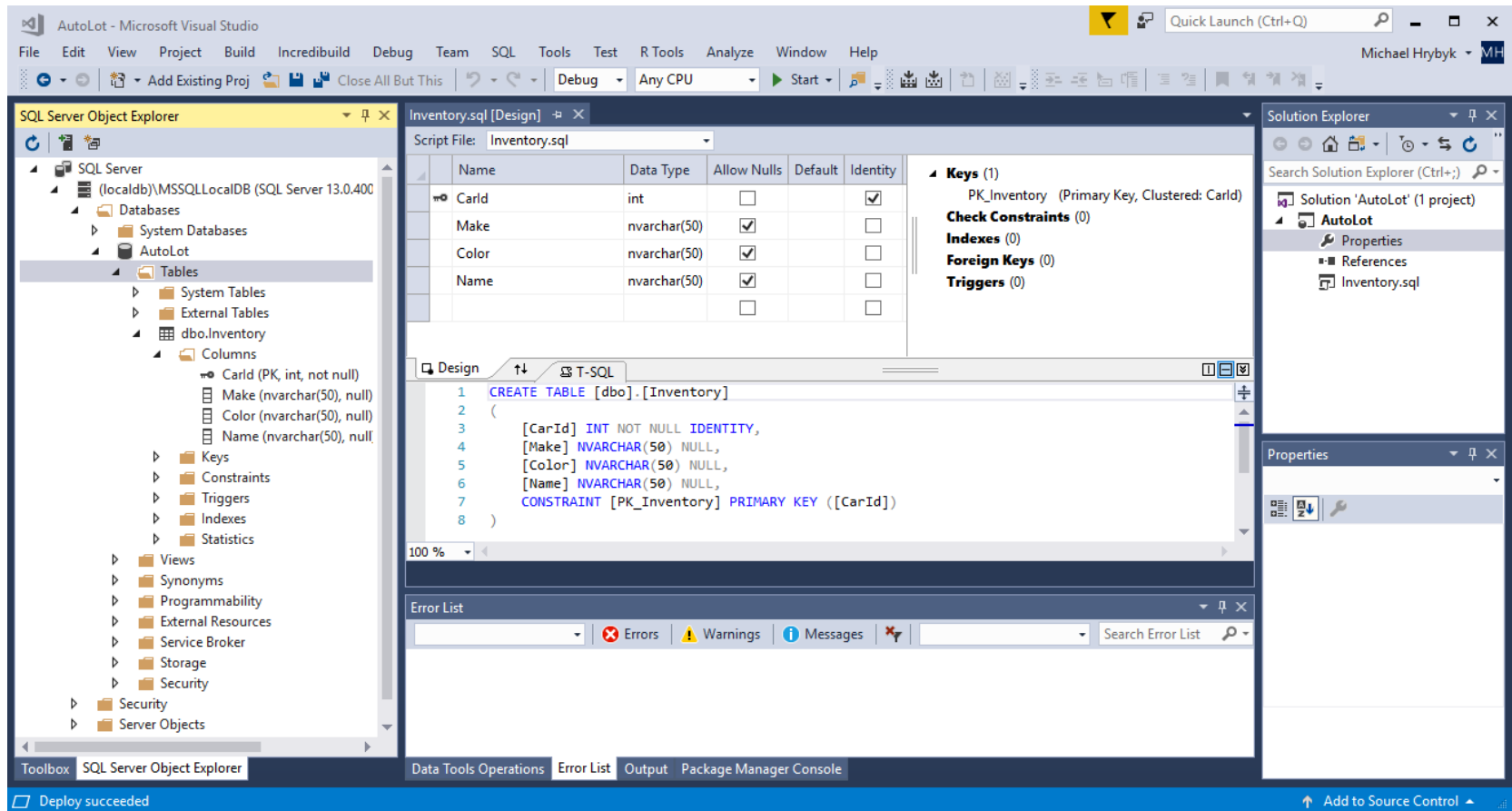
- 'Target database connection:' text label.
- A text box containing the connection string: 'iveResultSets=False;Connect Timeout=60;Encrypt=False;TrustServerCertificate=True'. To its right are 'Edit...' and 'Clear' buttons.
- 'Database name:' text label.
- A text box containing 'AutoLot'.
- 'Publish script name:' text label.
- A text box containing 'AutoLot.sql'.
- A checkbox labeled 'Register as a Data-tier Application'.
- A checkbox labeled 'Block publish when database has drifted from registered version'.
- An 'Advanced...' button.
- A 'Load Profile...' button.
- A row of four buttons at the bottom: 'Create Profile', 'Save Profile As...', 'Generate Script', and 'Publish'.
- A 'Cancel' button.

Database Publish

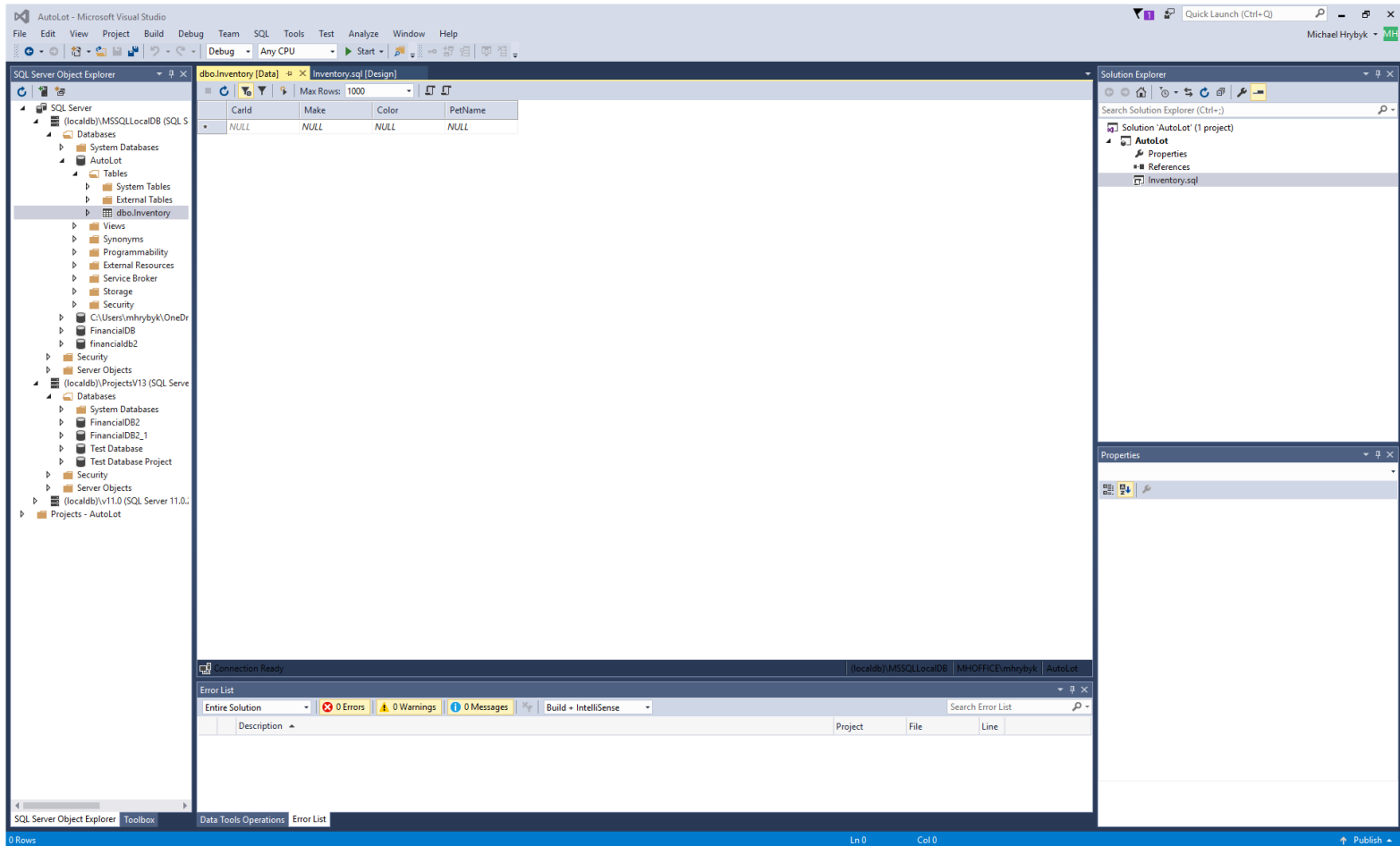
- AutoLot.publish.xml will appear under the AutoLotProject
- To update or recreate the database, just click on this file, and the Publish Dialog will appear again – but already filled in!



OR ... Hit Start, and refresh LocalDB



Expand AutoLot and right click on dbo.Inventory, choose Show Data



Enter the data and click refresh

The screenshot shows the Microsoft Visual Studio IDE with the SQL Server Object Explorer on the left and the SQL Server Enterprise Manager (SSE) window on the right. The SSE window displays the data for the `dbo.Inventory` table. The table has columns: `CardId`, `Make`, `Color`, and `PetName`. The data is as follows:

CardId	Make	Color	PetName
1	VW	Black	Zippy
2	Ford	Rust	Rusty
3	Saab	Black	Mel
4	Yugo	Yellow	Clunker
5	BMW	Black	Bimmer
6	BMW	Green	Hank
7	BMW	Pink	Pinky
NULL	NULL	NULL	NULL

Create Customer table and hit Start

The screenshot displays the SQL Server Enterprise Designer interface for a project named 'AutoLot'. The main window shows the 'Customers.sql [Design]' tab, which is split into two panes. The left pane shows the table design with columns: CustId (int, NOT NULL, IDENTITY), FirstName (nvarchar(50), NULL), and LastName (nvarchar(50), NULL). The right pane shows the 'Keys' section with a primary key 'PK_Customers' on the CustId column. Below the design panes is the 'T-SQL' tab, which contains the following SQL script:

```
1 CREATE TABLE [dbo].[Customers]
2 (
3     [CustId] INT NOT NULL IDENTITY,
4     [FirstName] NVARCHAR(50) NULL,
5     [LastName] NVARCHAR(50) NULL,
6     CONSTRAINT [PK_Customers] PRIMARY KEY ([CustId])
7 )
8
```

The 'Solution Explorer' on the right shows the project structure: 'Solution 'AutoLot' (1 project)' containing 'AutoLot' (Properties, References), 'Customers.sql', and 'Inventory.sql'. The 'Properties' window at the bottom right shows the 'Customers Table' properties, including the table name, data compression, description, filestream file, identity column (CustId), replication status (False), lock escalation (Table), and regular data space (Filegroup).

Table Design:

Name	Data Type	Allow Nulls	Default	Identity
CustId	int	<input type="checkbox"/>		<input checked="" type="checkbox"/>
FirstName	nvarchar(50)	<input checked="" type="checkbox"/>		<input type="checkbox"/>
LastName	nvarchar(50)	<input checked="" type="checkbox"/>		<input type="checkbox"/>

Keys (1)

- PK_Customers (Primary Key, Clustered: CustId)

Check Constraints (0)

Indexes (0)

Foreign Keys (0)

Triggers (0)

Add data to Customer table

The screenshot shows the Microsoft Visual Studio IDE with the SQL Server Object Explorer on the left and the SQL Server Enterprise Manager (SSE) window on the right. The SSE window displays the data for the `dbo.Customers` table in the `AutoLot` database. The table has four columns: `CustId`, `FirstName`, and `LastName`. The data is as follows:

CustId	FirstName	LastName
1	Dave	Bremmer
2	Matt	Walton
3	Steve	Hagen
4	Pat	Walton
NULL	NULL	NULL

Create Orders table (including relationships)

The screenshot displays the SQL Server Enterprise Designer interface for creating a table named 'Orders' in the 'dbo' schema. The 'Design' view is active, showing a table with three columns: 'OrderId' (int, NOT NULL, IDENTITY, PRIMARY KEY), 'CustId' (int, NOT NULL, FOREIGN KEY to Customers), and 'CarId' (int, NOT NULL, FOREIGN KEY to Inventory). The 'T-SQL' view is also visible, showing the corresponding CREATE TABLE statement.

Name	Data Type	Allow Nulls	Default	Identity
OrderId	int	<input type="checkbox"/>		<input checked="" type="checkbox"/>
CustId	int	<input type="checkbox"/>		<input type="checkbox"/>
CarId	int	<input type="checkbox"/>		<input type="checkbox"/>

Keys (1)
PK_Orders (Primary Key, Clustered: OrderId)

Check Constraints (0)

Indexes (0)

Foreign Keys (2)
FK_Orders_Customer (CustId)
FK_Orders_Inventory (CarId)

Triggers (0)

```
1 CREATE TABLE [dbo].[Orders]
2 (
3     [OrderId] INT NOT NULL IDENTITY,
4     [CustId] INT NOT NULL,
5     [CarId] INT NOT NULL,
6     CONSTRAINT [PK_Orders] PRIMARY KEY ([OrderId]),
7     CONSTRAINT [FK_Orders_Customer] FOREIGN KEY ([CustId]) REFERENCES [Customers]([CustId]),
8     CONSTRAINT [FK_Orders_Inventory] FOREIGN KEY ([CarId]) REFERENCES [Inventory]([CarId])
9 )
```

Solution Explorer
Solution 'AutoLot' (1 project)
AutoLot
Properties
References
Customers.sql
Inventory.sql
Orders.sql

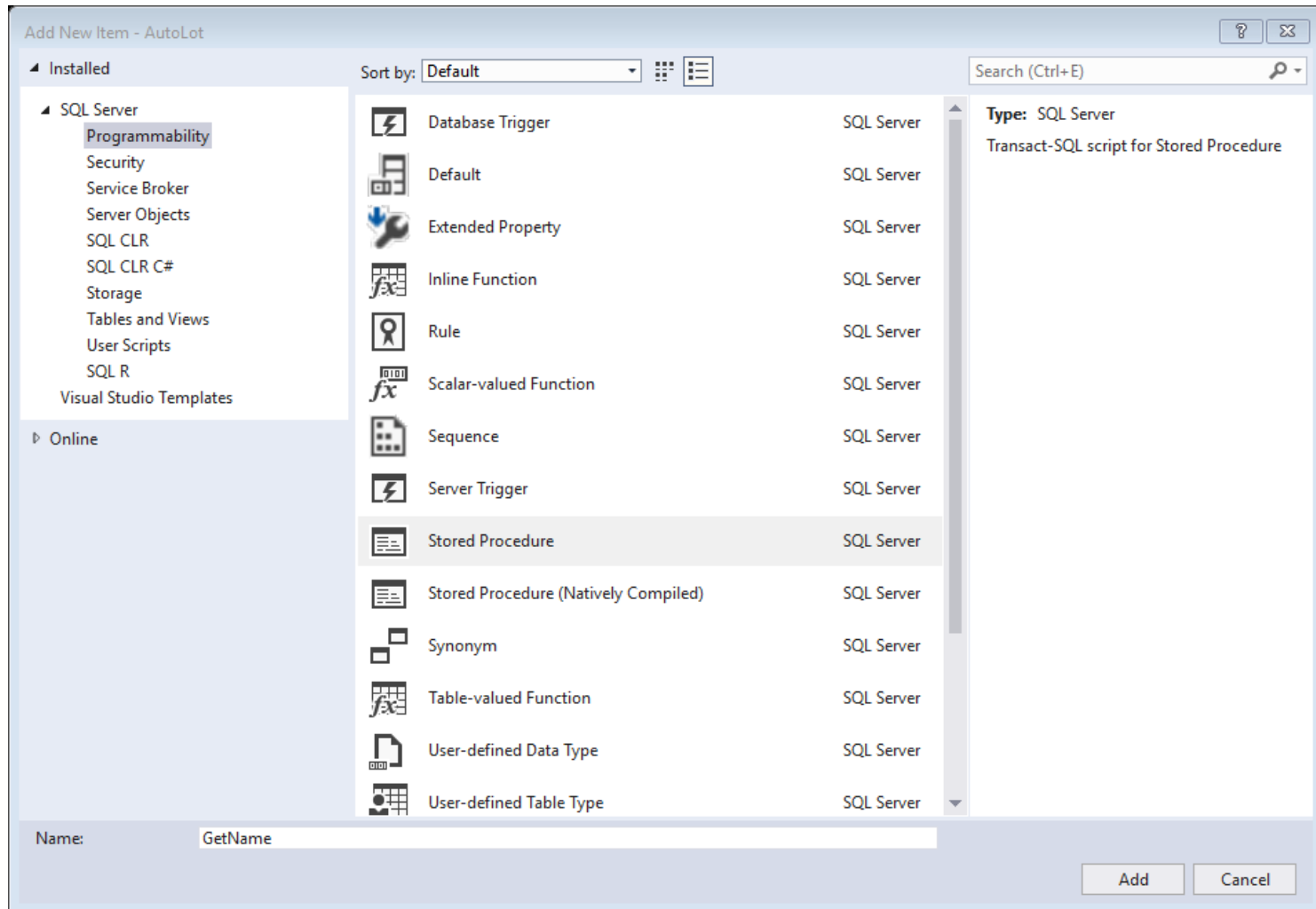
Properties
Orders Table
(Name) Orders
Data Compress
Description
Filestream File
Identity Column OrderId
Is Replicated False
Lock Escalation Table
Regular Data Sp Filegroup

Add data to Orders

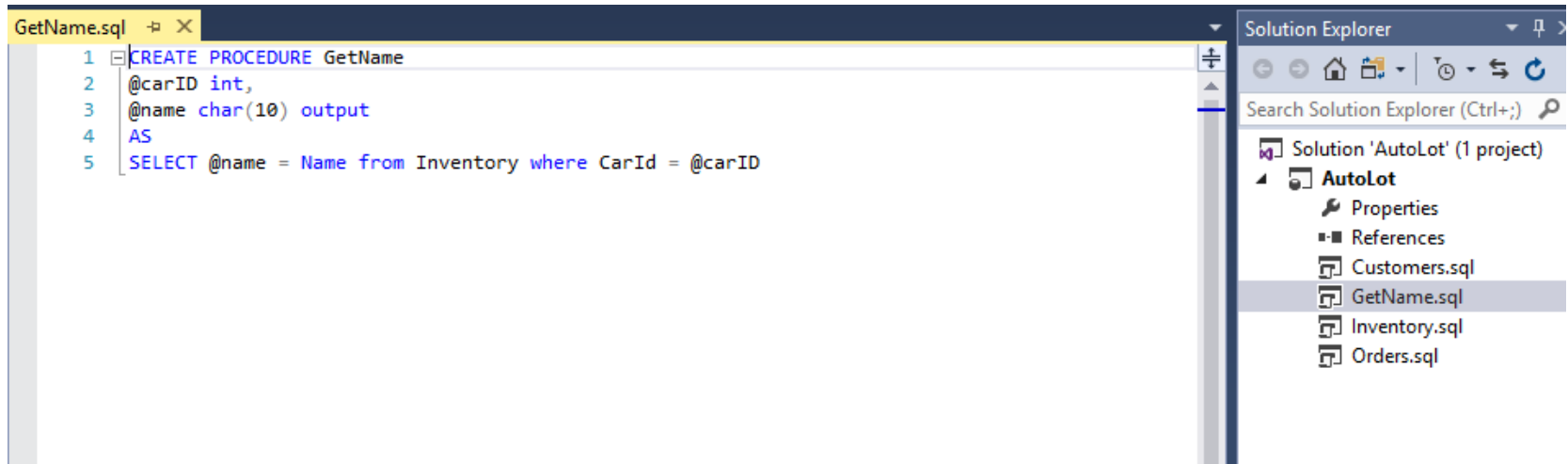
The screenshot shows the Microsoft Visual Studio IDE with the SQL Server Object Explorer on the left and the SQL Server Enterprise Manager on the right. The Object Explorer shows the hierarchy of the SQL Server instance, with the 'dbo.Orders' table selected. The SQL Server Enterprise Manager shows the 'dbo.Orders' table data, which is displayed in a grid. The grid has columns for 'OrderId', 'CustId', and 'CarId'. The data is as follows:

OrderId	CustId	CarId
1	1	5
2	2	1
3	3	4
4	4	7
•	NULL	NULL

Create Stored Procedure



Edit, Save and hit Start



Using AutoLot

- Create a new AutoLot database in SQL Server
 - In Class 06 Exercises, AutoLot project has already been created
 - Remove any existing AutoLot database in SQL Server Object Explorer
 - Create a new one
- AutoLot PROJECT has already been created
 - See SQL scripts
 - Note SeedData script automatically initializes tables
 - Need to enable post-build action
- Publish AutoLot using included profile
 - Will create the database and seed the data
 - Tables generated with data already inserted

DBConnection base class

■ **Note** Look up the `ConnectionString` property of your data provider's connection object in the .NET Framework 4.6 SDK documentation to learn more about each name-value pair for your specific DBMS.

After you establish your connection string, you can use a call to `Open()` to establish a connection with the DBMS. In addition to the `ConnectionString`, `Open()`, and `Close()` members, a connection object provides a number of members that let you configure additional settings regarding your connection, such as timeout settings and transactional information. Table 21-5 lists some (but not all) members of the `DbConnection` base class.

Table 21-5. *Members of the `DbConnection` Type*

Member	Meaning in Life
<code>BeginTransaction()</code>	You use this method to begin a database transaction.
<code>ChangeDatabase()</code>	You use this method to change the database on an open connection.
<code>ConnectionTimeout</code>	This read-only property returns the amount of time to wait while establishing a connection before terminating and generating an error (the default value is 15 seconds). If you would like to change the default, specify a <code>Connect Timeout</code> segment in the connection string (e.g., <code>Connect Timeout=30</code>).
<code>Database</code>	This read-only property gets the name of the database maintained by the connection object.
<code>DataSource</code>	This read-only property gets the location of the database maintained by the connection object.
<code>GetSchema()</code>	This method returns a <code>DataTable</code> object that contains schema information from the data source.
<code>State</code>	This read-only property gets the current state of the connection, which is represented by the <code>ConnectionState</code> enumeration.

ConnectionStringBuilder

Working with ConnectionStringBuilder Objects

Working with connection strings programmatically can be cumbersome because they are often represented as string literals, which are difficult to maintain and error-prone at best. The Microsoft-supplied ADO.NET data providers support *connection string builder objects*, which allow you to establish the name-value pairs using strongly typed properties. Consider the following update to the current `Main()` method:

```
static void Main(string[] args)
{
    WriteLine("***** Fun with Data Readers *****\n");

    // Create a connection string via the builder object.
    var cnStringBuilder = new SqlConnectionStringBuilder
    {
        InitialCatalog = "AutoLot",
        DataSource = @"(local)\SQLEXPRESS2014",
        ConnectTimeout = 30,
        IntegratedSecurity = true
    };

    using(SqlConnection connection = new SqlConnection())
    {
        connection.ConnectionString = cnStringBuilder.ConnectionString;
        connection.Open();
        ShowConnectionStatus(connection);
    }
    ...
    }
    ReadLine();
}
```

In this iteration, you create an instance of `SqlConnectionStringBuilder`, set the properties accordingly, and obtain the internal string using the `ConnectionString` property. Also note that you use the default constructor of the type. If you so choose, you can also create an instance of your data provider's connection

Connection Status

- See output from AutoLotDataReader project

```
static void ShowConnectionStatus(SqlConnection connection)
{
    // Show various stats about current connection object.
    WriteLine("***** Info about your connection *****");
    WriteLine($"Database location: {connection.DataSource}");
    WriteLine($"Database name: {connection.Database}");
    WriteLine($"Timeout: {connection.ConnectionTimeout}");
    WriteLine($"Connection state: {connection.State}\n");
}
```

Now let's write some code to access!

- Open AutoLotDataReader project.
- Notice the connection string.
- Process
 - Open a connection to the DB (it is actually a virtual connection – named pipe)
 - Use the connection string
 - Create a SQL command
 - Read the results (DataReader)

SimpleSqlCommand()

- Note ExecuteReader()
 - SqlCommand, then SqlDataReader
- Then a while loop to read each record
- Note use of **using**
 - frees up any resources when code block exits (eg, closes connection)

```
static void SimpleSqlCommand()
{
    WriteLine("***** Fun with Data Readers V1 *****\n");
    // Create and open a connection.
    using (SqlConnection connection = new SqlConnection())
    {
        // build the connection string by hand

        connection.ConnectionString =
            @"Data Source=(localdb)\MSSQLLocalDB;Integrated Security=SSPI;" +
            "Initial Catalog=AutoLot";
        connection.Open();

        WriteLine("++++ INVENTORY ++++");
        // Create a SQL command object.
        string sql = "Select * From Inventory";
        SqlCommand myCommand = new SqlCommand(sql, connection);

        // Obtain a data reader a la ExecuteReader().
        using (SqlDataReader myDataReader = myCommand.ExecuteReader())
        {
            // Loop over the results.
            while (myDataReader.Read())
            {
                WriteLine($"-> Make: {myDataReader["Make"]}, Name: {myDataReader["Name"]}, Color: {myDataReader["Color"]}");
            }
        }

        ShowConnectionStatus(connection);
    }
    ReadLine();
}
```


MultipleSQLCommands()

- Builds the connection string with SqlConnectionStringBuilder

```
static void MultipleSQLCommands()
{
    WriteLine("***** Fun with Data Readers V2 *****\n");
    // Create a connection string via the builder object.
    var connectionStringBuilder = new SqlConnectionStringBuilder
    {
        InitialCatalog = "AutoLot",
        DataSource = @"(localdb)\MSSQLLocalDB",
        ConnectTimeout = 30,
        IntegratedSecurity = true
    };

    // Create an open a connection.
    using (var connection = new SqlConnection())
    {
        connection.ConnectionString = connectionStringBuilder.ConnectionString;
        connection.Open();
        ShowConnectionStatus(connection);
    }
}
```

MultipleSQLCommands()

- For multiple tables, can create a command with two select statements
 - Although this is rarely done
 - Concatenates the results
- Note use of GetName() and GetValue() iterating over FieldCount.
 - This will be different for each table

```
// Create a SQL command object. Note two select statements

string sql = "Select * From Inventory;Select * from Customers";

WriteLine("++++ INVENTORY and CUSTOMERS ++++");

using (SqlCommand myCommand = new SqlCommand(sql, connection))
{
    //iterate over the inventory & customers
    // Obtain a data reader a la ExecuteReader().
    // note use of getname and getvalue for each field

    using (SqlDataReader myDataReader = myCommand.ExecuteReader())
    {
        do
        {
            while (myDataReader.Read())
            {
                WriteLine("***** Record *****");
                for (int i = 0; i < myDataReader.FieldCount; i++)
                {
                    WriteLine($"{myDataReader.GetName(i)} = {myDataReader.GetValue(i)}");
                }
                WriteLine();
            }
        } while (myDataReader.NextResult());
        ShowConnectionStatus(connection);
    }
}
```

Using App.config

- Walk through SimpleDataProviderFactory example
- Use of xml in configuration
 - key/value pairs
- Eliminates the need to hard code in connection strings for databases.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <!-- Which provider? -->
    <add key="provider" value="System.Data.SqlClient" />
    <!-- Which connection string? -->
    <add key="connectToAutoLot" value="Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=AutoLot;Integrated Security=True"/>
    <add key="connectToLocalAutoLot" value="Data Source=(localdb)\MSSQLLocalDB;AttachDbFilename='C:\Users\mhrybyk\OneDrive\Documents\CSIS 3540\Exercises\Class 06\AutoLot\AutoLot\Database\AutoLot.mdf';Initial Catalog=AutoLot;Integrated Security=False;"/>
  </appSettings>
  <connectionStrings>
    <add name="AutoLotDBSQLProvider" connectionString="Data Source=(localdb)\MSSQLLocalDB;Integrated Security=SSPI;Initial Catalog=AutoLot"/>
  </connectionStrings>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6"/>
  </startup>
</configuration>
```

SQL Statements

- Non-Queries
 - Insert Into TableName (FieldsName1, ...) Values (Values1, ...)
 - Use of @ for Parameters, where Fields are specified by type
 - Delete from TableName where <expression>
 - EG, CarId = 3
 - Update TableName Set var = value <expression>
 - Truncate Table TableName
 - All followed by ExecuteNonQuery()
- Queries
 - Select * from TableName <expression>
 - Followed by ExecuteReader()
- Scalars
 - Select statements using COUNT, AVERAGE, SUM, etc
 - Followed by ExecuteScalar(), cast to result type (int, for example)
- Stored Procedure
 - <procedure_name> <parameter_list>
 - must set the CommandType property to the value CommandType.StoredProcedure.

Data Access Layer

- Sometimes best to create a separate data access layer as a class library which other code can use
- AutoLotDAL project (class library) contains
 - DataOperations
 - AutoLotDataAccessLayer
 - Models
 - Car and Customer
- AutoLotDataAccessLayer extends DataTableAccessLayer
 - Used to get configuration information from App.config
 - Open/close connection to DB
 - Get DB table data into a DataTable
- **See AutoLotCUIClient**
 - **CUI means console user interface**
 - **Uses the custom AutoLotDataAccessLayer**
 - **Study this in detail**
- To use AutoLotDAL in an application, make sure you add a reference to it in the project
 - Adds the DLL to the project

Joins and Views

- Used as a simple query
 - Where $x = y$
- Or FROM table INNER JOIN table2 ON $x = y$
- See AutoLot **CustomerOrders** View and DAL code.
 - Can treat it as a table
 - As we will see, just as easy to use linq and internal code.

```
CREATE VIEW [dbo].[CustomerOrders]
AS SELECT [Customers].*, [Inventory].*
FROM [Orders]
inner join [Customers] on [Customers].CustId = [Orders].CustId
inner join [Inventory] on [Inventory].CarId = [Orders].CarId
```

Using SqlDataTableAccessLayer

- Can load the result of a query into a DataTable
- DataTable has Columns, Rows, and other properties that can be accessed.
 - Each Column has a Name property
- Much easier than using IDataReader Read() and Next() methods.

```
public DataTable GetDataTable(string tableName)
{
    DataTable dataTable = new DataTable
    {
        TableName = tableName
    };

    LoadDataTable(dataTable);
    return dataTable;
}

public void LoadDataTable(DataTable table)
{
    table.Clear();

    var sqlCommand =
        new SqlCommand("Select * From [" + table.TableName + "]", sqlConnection);

    Debug.WriteLine("LoadDataTable: " + sqlCommand.CommandText);

    using (SqlDataReader dataReader = sqlCommand.ExecuteReader())
    {
        // Fill the DataTable with data from the reader. This
        // set up the schema in the DataTable as well.

        table.Load(dataReader);
    }

    table.AcceptChanges();
}
```

SqlDataTableAccessLayer Connections

- Gets the connection string
- Use the string to open and close connection.

```
protected SqlConnection sqlConnection = null;
public void OpenConnection(string connectionString)
{
    sqlConnection = new SqlConnection { ConnectionString = connectionString };
    sqlConnection.Open();
}

public void CloseConnection()
{
    sqlConnection.Close();
}

public string GetConnectionString(string key)
{
    if (key == null)
    {
        return null;
    }

    return ConfigurationManager.ConnectionStrings[key].ConnectionString;
}
```


GetDataTable()

- Given a table name, fill a DataTable object
- Load data from the DB into the object
- Set primary key(s) – see private method for how to do this
- Save changes in the object
- Only need to call when first referencing the database table

```
/// <summary>
/// Simple method to create and fill a DataTable object
/// from any sql table.
///
/// Primary key columns are set.
/// </summary>
/// <param name="tableName">Name of the sql table</param>
/// <returns>DataTable containing the sql data or null if table does not exist</returns>
public DataTable GetDataTable(string tableName)
{
    if (tableName == null)
        return null;

    DataTable dataTable = new DataTable
    {
        TableName = tableName,
    };

    // load the data from the database into the DataTable object
    LoadDataTable(dataTable);

    // get the primary keys from the database for the table, and load into the object

    SetPrimaryKey(dataTable);
    dataTable.AcceptChanges();

    return dataTable;
}
```

LoadDataTable()

- Loads data from the DB into a DataTable object
- Clears old data first.
- Executes a SELECT statement on the table and returns SqlDataReader result
- DataTable.Load() method then loads the result into the object

```
/// <summary>
/// Load records from a database table into an existing DataTable.
///
/// The DataTable must exist and have a TableName property set
/// to a table that exists in the SQL database
///
/// The DataTable is cleared before the records are added, so
/// any existing rows in the DataTable are removed.
/// </summary>
/// <param name="table">DataTable to be loaded</param>
public void LoadDataTable(DataTable table)
{
    table.Clear();

    var sqlCommand =
        new SqlCommand("Select * From [" + table.TableName + "]", sqlConnection);

    Debug.WriteLine("LoadDataTable: " + sqlCommand.CommandText);

    using (SqlDataReader dataReader = sqlCommand.ExecuteReader())
    {
        // Fill the DataTable with data from the reader. This
        // set up the schema in the DataTable as well.

        table.Load(dataReader);
    }

    table.AcceptChanges();
}
```

DataTables are event driven

- DataTable object has associated events
- Useful with GUI (Windows Forms, ...) and databases
- When DataTable changes, the underlying database table can be updated accordingly
- Process events from GUI and update the database

ColumnChanged	Occurs after a value has been successfully changed in a DataColumn .
ColumnChanging	Occurs when a value has been submitted for a DataColumn.
RowChanged	Occurs after a DataColumn value or the RowState of a DataRow in the DataTable has been changed successfully.
RowChanging	Occurs when a change has been submitted for a DataColumn value or the RowState of a DataRow in the DataTable.
RowDeleted	Occurs after a DataRow in the DataTable has been marked as Deleted.
RowDeleting	Occurs before a DataRow in the DataTable is marked as Deleted.
TableCleared	Occurs after a call to the Clear method of the DataTable has successfully cleared every DataRow.
TableClearing	Occurs after the Clear method is called but before the Clear operation begins.
TableNewRow	Occurs after a new DataRow is created by a call to the NewRow method of the DataTable.

Add DataTable methods to DAL

- DataTable consists of DataRow and DataColumn objects
 - DataTable.Rows and DataTable.Columns are collections of these objects.
- Each method takes a DataRow and operates on the related database table
 - InsertTableRow()
 - UpdateTableRow()
 - DeleteTableRow()
- These methods in general create a SQL command string (Insert, Update, Delete) then ExecuteNonQuery()
- DataRow keeps a property DataTable property that is a pointer to its parent table
- See **AutoLotDAL**

using statement and Debug class

- Note **using** statement followed by a code block will dispose of all objects when the block is finished executing.
 - Useful for closing connections automatically
- Note use of Debug.WriteLine, which displays its arguments in the Output window of Visual Studio.
 - Needs System.Diagnostics
 - Allows for tracing
- Code from LoadTable below

```
using System.Diagnostics;

Debug.WriteLine("LoadDataTable: " + sqlCommand.CommandText);

using (SqlDataReader dataReader = sqlCommand.ExecuteReader())
{
    // Fill the DataTable with data from the reader.
    table.Load(dataReader);
}
```

InsertTableRow() – create the command

- Command string consists of column names then the values for the row
- In this implementation, assume that the id column is the first one, so do not include it.
- Get the column names from DataTable object, contained in DataRow

```
public void InsertTableRow(DataRow row)
{
    // avoid potential race condition, as updating and id from the database
    // trigger a RowChangedEvent.Changed action below which should be ignored.

    if (row == null || row.RowState != DataRowState.Added)
        return;

    // carefully create the insert clause, followed by column names from the table
    // and values from the row

    DataTable table = row.Table;

    string insertClause = "Insert into [" + table.TableName + "]";

    // start columns with this
    string columns = "(";

    // start values with this
    string values = " Values (";

    // if a column has autoincrement set, do not include it in the insert clause
    // and remember if this was done, as we will have to get the id from the DB later
    bool identityColumn = table.Columns[0].AutoIncrement;

    for (int i = 0; i < table.Columns.Count; i++)
    {
        if (table.Columns[i].AutoIncrement == false || restoringFromBackup == true)
        {
            // include the column and its value in the insert clause
            columns += $"{table.Columns[i].ColumnName},";
            values += $"{row[i]}','";
        }
    }

    // the last column and value has an extra comma so trim it and add a close parenthesis
    columns = columns.TrimEnd(',', ' ') + ")";
    values = values.TrimEnd(',', ' ') + ")";

    // now put together the command

    string insertCommand = insertClause + columns + values;
```

InsertTableRow() – execute command

- Note that identity_insert is turned on if restoring from backup or if an identity column is being inserted directly.
- Need to AcceptChanges()
- Catching an exception requires RejectChanges()

```
using (SqlCommand sqlCommand = new SqlCommand(insertCommand, sqlConnection))
{
    try
    {
        // set identity_insert only if there is an identity column and
        // restoring from backup
        // for data restore purposes

        if (identityColumn == true && restoringFromBackup == true)
            SetIdentityInsert(table.TableName, true);

        Debug.WriteLine("InsertTableRow: " + insertCommand);
        sqlCommand.ExecuteNonQuery();

        if (identityColumn == true && restoringFromBackup == true)
            SetIdentityInsert(table.TableName, false);
    }
    catch (SqlException ex)
    {
        Debug.WriteLine("InsertTableRow: " + insertCommand);
        string id = row[0].ToString();
        row.RejectChanges();
        throw new Exception("Insert failed: " + ex.Message);
    }
}

table.AcceptChanges();
```

InsertTableRow() – get assigned id

- Second half of method requires getting the recently assigned id from the database
- Use sql command **select scope_identity ('tablename')**
- ExecuteScalar(), read response, and update DataTable column[0]
- Note use of SetField(), which is the only way to manually update a DataRow.
- This may NOT be safe, as other users could update the table and incorrect id could be returned.

```
string scopeIdentityCommand = "select scope_identity()";
Debug.WriteLine("InsertTableRow: " + scopeIdentityCommand);
using (SqlCommand sqlCommand =
    new SqlCommand(scopeIdentityCommand, sqlConnection))
{
    decimal id = (decimal)sqlCommand.ExecuteScalar();
    Debug.WriteLine("InsertTableRow: ID from DB " + id);

    // to update the id, we need to allow writing
    table.Columns[0].ReadOnly = false;

    // set the value of the id field with what was returned from the db
    // unfortunately, this triggers another RowChangedEvent, which should be ignored.
    // deal with this in the entry to this method to prevent a race condition.

    row.SetField(0, id.ToString());

    // do not use itemarray, as it is a COPY of the items, so nothing happens
    // row.ItemArray[0] = id.ToString();

    // now set it back to readonly
    table.Columns[0].ReadOnly = true;

    table.AcceptChanges();
}
```


UpdateTableRow()

- Used to update any element of a DataRow
- Might as well just update the whole row at once

```
public void UpdateTableRow(DataRow row)
{
    // create the update clause
    DataTable table = row.Table;

    string updateClause = "Update " + table.TableName + " Set";

    // create SET name = 'value' statements
    // do NOT update the id column in position 0

    string values = "";

    for (int i = 0; i < table.Columns.Count; i++)
    {
        if(table.Columns[i].AutoIncrement == false)
            values += $" {table.Columns[i].ColumnName} = '{row[i]}',";
    }

    values = values.TrimEnd(',');

    // use the original value for the id column in case we are changing the value of it
    string whereClause = $" where {table.Columns[0].ColumnName} = '{row[0], DataRowVersion.Original}]'";

    string updateCommand = updateClause + values + whereClause;

    Debug.WriteLine("UpdateTableRow: " + updateCommand);

    using (SqlCommand sqlCommand = new SqlCommand(updateCommand, sqlConnection))
    {
        try
        {
            sqlCommand.ExecuteNonQuery();
        }
        catch (SqlException ex)
        {
            string id = row[0].ToString();
            row.RejectChanges();
            throw new Exception("Update failed for id " + id + ": " + ex.Message);
        }
    }

    table.AcceptChanges();
}
```

DeleteTableRow()

- Construct command using ORIGINAL value
 - Current value of row does not exist (it has been deleted!)

```
public void DeleteTableRow(DataRow row)
{
    DataTable table = row.Table;

    // create the delete command

    string deleteClause = "delete from " + table.TableName;

    // a deleted table row has no data. one has to get the original column data instead

    string whereClause = " where " + table.Columns[0].ColumnName +
        " = '" + row[0, DataRowVersion.Original] + "'";

    string deleteCommand = deleteClause + whereClause;

    Debug.WriteLine("DeleteTableRow: " + deleteCommand);

    using (SqlCommand sqlCommand = new SqlCommand(deleteCommand, sqlConnection))
    {
        try
        {
            sqlCommand.ExecuteNonQuery();
        }
        catch (Exception ex)
        {
            row.RejectChanges();
            throw new Exception("Delete failed for id " +
                row[0, DataRowVersion.Original] + ": " + ex.Message);
        }
        row.AcceptChanges();
    }
    // if there are no rows left in the table, make sure
    // all the rows are cleared in the control as well
    if (table.Rows.Count == 0)
    {
        table.Rows.Clear();
        table.AcceptChanges();
    }
}
```

Console App using AutoLotDAL

- See **AutoLotCUI** and **AutoLotDAL**
- Add References to AutoLotDAL and DataTableAccessLayer libraries
- Make sure App.Config has correct connection string
- Create AutoLotDataAccessLayer() object and connect to the database.
- Console commands allow for CRUD operations
 - Can view or modify all tables
- Includes transaction processing with ProcessCreditRisk()

Displaying DataTable

- Use the Columns collection to display properties such as Column Name
- In each Column in the collection of Rows, display using ItemArray.
- Can also use array indices (see DisplayTable() method in
- Again, much easier than DataReader.
- DataTable will be the main constituent of other ADO objects, such as DataSet

```
private static void DisplayTable(DataTable dataTable)
{
    const int columnWidth = 10;

    // Print out the column names using collection
    foreach (DataColumn column in dataTable.Columns)
        Write($"{column.ColumnName,columnWidth}");
    WriteLine();

    // write the correct number of dashes to cover the columns
    WriteLine(new string('-', columnWidth * dataTable.Columns.Count)); //

    // Print the DataTable.
    foreach (DataRow row in dataTable.Rows)
    {
        foreach (object column in row.ItemArray)
            Write($"{column,columnWidth}");
        WriteLine();
    }
}
```

Displaying a database table

- Call GetDataTable(), which fills a DataTable object.
- Then call DisplayTable()

```
private static void DisplayAutoLotTable(AutoLotDataAccessLayer autoLotDAL, string tableName)
{
    if(tableName == null || tableName.Length == 0)
    {
        Console.WriteLine("null or blank table name - can't display the table");
        return;
    }

    // get the table

    DataTable table = autoLotDAL.GetDataTable(tableName);

    if (table == null)
    {
        Console.WriteLine("Can't find table " + tableName + " in the database");
        return;
    }

    // now display it

    DisplayTable(table);
}
```

Windows Forms App Using AutoLotDAL

- Using Windows Forms, associate DataGridView (or other controls) with a DataTable.
 - When the control changes, the underlying DataTable is changed as well.
 - To update the database, handle the DataTable events, NOT the control's events!
- Control.DataSource = a DataTable object is all that is needed.
 - Then handle DataTable events
- Add DataTable to DataSet
 - Keeps a list of all tables for XML backup

Creating a WinForms app using DataTables

- See **AutoLotWinFormsClient**
- Create DataGridView controls for each database table
- Create a DataTable for each database table, and fill it from the database (GetDataTable()).
- Set DataSource in the control to the DataTable object
- Associate RowChanged, ColumnChanged, and RowDeleting events with DAL handlers
- When a user makes changes via the control, DataTable will update (DataGridView will do this).
 - Catch the event and update the database

InitializeDataGridViewAndDataSet()

- Called for each table
- Notice event handler ONLY needs eventargs (e)

```
public void InitializeDataGridViewAndDataSet(DataGridView dataGridView, DataSet dataSet, string tableName)
{
    // get the table filled with records from the db
    DataTable table = autoLotDAL.GetDataTable(tableName);

    // set the datasource to the table.
    // when the table changes, the control will update as well,
    // so make sure to handle relevant table change events

    // this autogenerates the column names, so no need to set them manually
    dataGridView.DataSource = table;

    // if we have an identity column, any time a row is added we want the
    // column to be set to -1
    if (table.Columns[0].AutoIncrement == true)
    {
        dataGridView.DefaultValuesNeeded += (s, e) => NewRowBeingAdded(s as DataGridView, e);
    }

    // handle insertion
    table.RowChanged += (s, e) => AutoLotTableRowChanged(e);

    // handle updates
    table.ColumnChanged += (s, e) => AutoLotTableColumnChanged(e);

    // handle deletes
    table.RowDeleted += (s, e) => AutoLotTableRowDeleted(e);

    // autosize the columns to fill out as much as possible
    dataGridView.AutoSizeColumnsMode = DataGridViewAutoSizeColumnsMode.Fill;

    // allow multiple select to allow for deletion of multiple rows
    dataGridView.MultiSelect = true;

    // add the table to the Tables collection.
    // This is only used for backup and restore
    dataSet.Tables.Add(table);
}
```


AutoLotTableRowChanged()

- Look at EventArgs.Action, and only invoke the DAL insert method if it is an Add, otherwise ignore
- Update the CustomerOrders control by reloading data, as this is a database view
 - It must be read as the view may have changed based on an insert

```
private void AutoLotTableRowChanged(DataRowChangeEventArgs e)
{
    if (e.Action == DataRowAction.Add)
    {
        try
        {
            autoLotDAL.InsertTableRow(e.Row);
        }
        catch (Exception ex)
        {
            MessageBox.Show("Insertion failed: " + ex.Message); ;
        }
        autoLotDAL.LoadDataTable(dataGridViewCustomerOrders.DataSource as DataTable);
    }
}
```

AutoLotTableColumnChanged()

- If a column changes in the table, update the database table
- If the row is detached, do nothing.
 - This means the row is in the process of being added. Kind of no man's land.
- Otherwise, update the database with the changed row
- Update any views as well.

```
private void AutoLotTableColumnChanged(DataColumnChangeEventArgs e)
{
    // if the row is in the process of being added (detached), don't update the cells
    // only do this if an existing cell is changed

    if (e.Row.RowState != DataRowState.Detached)
    {
        // if this is an identity column, it is only modified by the db in InsertTableRow()
        // so don't send an update back

        if (e.Column.AutoIncrement == false)
        {
            // just update the entire row even though just one column was changed
            // this could be optimized

            try
            {
                autoLotDAL.UpdateTableRow(e.Row);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
            autoLotDAL.LoadDataTable(dataGridViewCustomerOrders.DataSource as DataTable);
        }
    }
}
```

AutoLotTableRowDeleted()

- Very simple
 - DeleteTableRow and update view
 - Catch exceptions

```
private void AutoLotTableRowDeleted(DataRowChangeEventArgs e)
{
    try
    {
        autoLotDAL.DeleteTableRow(e.Row);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }

    autoLotDAL.LoadDataTable(dataGridViewCustomerOrders.DataSource as DataTable);
}
```

Putting it all together – Form method

- Create a DataSet
 - Used to hold all the tables
 - Will be only be used to backup the database
- Open the db connection
- Associate database tables with a DataTable and DataGridView

```
autoLotDataSet = new DataSet()
{
    DataSetName = "AutoLotDataSet",
};

// get the connection string from App.config
string connectionString = autoLotDAL.GetConnectionString("AutoLotSqlProvider");

autoLotDAL.OpenConnection(connectionString);

// set the form title
Text = "AutoLot Database Client";

// associate the datagridview controls with a database table
// this also adds each table to the dataset

// make sure Orders is added to dataset first. Add tables in the order
// in which they need to be cleared so database restore from backup works

InitializeDataGridViewAndDataSet(dataGridViewOrders, "Orders");
InitializeDataGridViewAndDataSet(dataGridViewInventory, "Inventory");
InitializeDataGridViewAndDataSet(dataGridViewCustomers, "Customers");
InitializeDataGridViewAndDataSet(dataGridViewCreditRisks, "CreditRisks");
```

NewRowBeingAdded()

- DataGridView event handler
- When a new row is being added in the control, display -1 in the first column
- Otherwise, it will use the number from DataTable AutoIncrement, which will be incorrect.
 - Won't actually be assigned, just appears odd to the user

```
private void NewRowBeingAdded(DataGridView dataGridView, DataGridViewRowEventArgs e)
{
    DataTable table = dataGridView.DataSource as DataTable;

    if (table.Columns[0].AutoIncrement == true)
    {
        e.Row.Cells[0].Value = -1;
    }
}
```

Putting it all together – Form method

- Set up the CustomerOrders view.
 - No events are handled for this table, as it is updated after each of the other tables' events
- Set up the CreditRisk button
- Set up the Backup Database and Restore Database buttons
- Make sure the db connection is closed with the form is closed

```
// associate the datagridview controls with a database table
// this also adds each table to the dataset

// make sure Orders is added to dataset last or database restore will crash

InitializeDataGridViewAndDataSet(dataGridViewInventory, autoLotDataSet, "Inventory");
InitializeDataGridViewAndDataSet(dataGridViewCustomers, autoLotDataSet, "Customers");
InitializeDataGridViewAndDataSet(dataGridViewCreditRisks, autoLotDataSet, "CreditRisks");
InitializeDataGridViewAndDataSet(dataGridViewOrders, autoLotDataSet, "Orders");

// CustomerOrders is a view, so don't let anyone edit it.
// Do not add it to DataSet.Tables

dataGridViewCustomerOrders.ReadOnly = true;
dataGridViewCustomerOrders.AllowUserToAddRows = false;
dataGridViewCustomerOrders.RowHeadersVisible = false;
dataGridViewCustomerOrders.AutoSizeColumnsMode = DataGridViewAutoSizeColumnsMode.Fill;
dataGridViewCustomerOrders.DataSource = autoLotDAL.GetDataTable("CustomerOrders");

// add button event handler for credit risk processing
buttonProcessCreditRisk.Click += (s, e) => ProcessCreditRisk();

// add button event handlers for database backup to xml
buttonBackupDatabase.Click += (s, e) => autoLotDAL.BackupDataSetToXML(autoLotDataSet);
buttonRestoreDatabaseFromBackup.Click += (s, e) => autoLotDAL.RestoreDataSetFromBackup(autoLotDataSet);

// ensure that the connection to the db is closed
this.FormClosing += (s, e) => autoLotDAL.CloseConnection();
```

Backup to XML

- Very simple
- Use DataSet.WriteXml()
 - Use arg to include the schema.
 - DataSetName needs to be set
- File saved in bin\Debug

```
public void BackupDataSetToXML(DataSet dataSet)
{
    if (dataSet == null)
        return;

    // writes the DataSet to an xml file including the schema
    // TODO: try/catch
    dataSet.WriteXml(dataSet.DataSetName + ".xml", XmlWriteMode.WriteSchema);
}
```

Sample XML backup file

```
<?xml version="1.0" standalone="yes"?>
<AutoLotDataSet>
  <xs:schema id="AutoLotDataSet" xmlns="" xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xs:element name="AutoLotDataSet" msdata:IsDataSet="true" msdata:UseCurrentLocale="true">
      ... the rest of the schema is here
    </xs:element>
  </xs:schema>
  <Inventory>
    <CarId>1</CarId>
    <Make>Honda</Make>
    <Color>White</Color>
    <Name>Civic</Name>
  </Inventory>
  <Customers>
    <CustId>1</CustId>
    <FirstName>John</FirstName>
    <LastName>Smith</LastName>
  </Customers>
  <Orders>
    <OrderId>1</OrderId>
    <CustId>2</CustId>
    <CarId>1</CarId>
  </Orders>
  <CreditRisks>
    <CustId>1</CustId>
    <FirstName>Mike</FirstName>
    <LastName>Jones</LastName>
  </CreditRisks>
</AutoLotDataSet>
```


Restore from XML

- A little trickier. Need to clear all the db tables first and reseed them to start their ids at 1
- Use DataSet.ReadXML, using the schema
- DataTable objects AND controls will automatically be updated!!
- Use restoringFromBackup flag so InsertTableRow() uses identity_insert

```
public void RestoreDataSetFromBackup(DataSet dataSet)
{
    if (dataSet == null)
    {
        Debug.WriteLine("RestoreDataSetFromBackup: Error - null dataset");
        return;
    }

    Debug.WriteLine("RestoreDataSetFromBackup: restoring from " + dataSet.DataSetName);

    // need to clear all of the dataset tables before restore
    // Tables are cleared in the order they are in the Tables
    // list, so make sure any table with dependencies is at the beginning
    // of the list, or else an exception will result.

    // so clear the tables in reverse order in which they were added

    for (int i = dataSet.Tables.Count; i > 0; i--)
    {
        DataTable table = dataSet.Tables[i - 1];
        Debug.WriteLine("Clearing Table " + table.TableName + " column[0] " +
            table.Columns[0].ColumnName + " autoincrement " +
            table.Columns[0].AutoIncrement);

        ClearDatabaseTable(table);
        table.Clear();
    }
    // to restore a table with identity set from xml, the
    // id's will be in the file already, so identity insert
    // needs to be turned on.

    // the InsertTableRow method checks to see if autoincrement
    // is on and if this is a restore.
    // ReadXML adds rows to the dataset tables, which fires
    // the RowChanged event calling InsertTableRow.
    // This will set identity_insert for
    // each insertion.

    restoringFromBackup = true;
    // Read the data from the file and add it to the dataset
    // TODO: exception handling
    dataSet.ReadXml(dataSet.DataSetName + ".xml", XmlReadMode.ReadSchema);

    // restore completed
    restoringFromBackup = false;
}
```

Clearing and reseed a DB table

- Use the DBCC CHECKIDENT and DELETE Sql commands

```
public void ClearDatabaseTable(DataTable table)
{
    // if identity is set, reseed the table
    if (table.Columns[0].AutoIncrement == true)
    {
        // reseed identity value should be 1 only if there were
        // no rows previously in the database

        int reseedStart = 0;

        if (table.Rows.Count > 0)
            reseedStart = 1;

        string reseedCommand = $"DBCC CHECKIDENT('{table.TableName}', RESEED, " +
            reseedStart + ")";

        Debug.WriteLine("ClearDatabaseTable: " + reseedCommand);
        using (SqlCommand sqlCommand = new SqlCommand(reseedCommand, sqlConnection))
        {
            sqlCommand.ExecuteNonQuery();
        }
    }

    // clear the table of all rows

    string deleteCommand = $"DELETE FROM {table.TableName}";

    Debug.WriteLine("ClearDatabaseTable: " + deleteCommand);

    using (SqlCommand sqlCommand = new SqlCommand(deleteCommand, sqlConnection))
    {
        sqlCommand.ExecuteNonQuery();
    }
}
```

Transactions

- ACID
 - *Atomic* (all or nothing)
 - *Consistent* (data remains stable throughout the transaction)
 - *Isolated* (transactions do not step on each other's feet)
 - *Durable* (transactions are saved and logged).
- Submit the set of transactions. If there are no exceptions, then commit(), else rollback()
- Catch exceptions and rollback()

Transactions

- See ProcessCreditRisks() in AutoLotDAL
 - Used in AutoLotCUIClient and AutoLotWinFormsClient
- Need to create CreditRisks table in AutoLot
- Removes a customer from Customers and moves to CreditRisks
 - Find customer id
 - If it exists, remove from Customers and insert into CreditRisks
 - Needs to be atomic
 - Use of commit, or if failuer, rollback

CreditRisks	
	CustID
	FirstName
	LastName

ProcessCreditRisk() in AutoLotWinFormsClient

- In the application, if a customer is selected and the credit risk button is clicked, move the customer to the CreditRisks table.
- Need to reload the Customers and CreditRisks table.
 - Note the use of DataSource in reverse – it is a DataTable!

```
private void ProcessCreditRisk()
{
    if (dataGridViewCustomers.SelectedRows.Count == 0)
    {
        MessageBox.Show("No customers selected");
        return;
    }

    // get each selected customer id and move them to CreditRisks

    foreach (DataGridViewRow row in dataGridViewCustomers.SelectedRows)
    {
        int customerId = (int)row.Cells[0].Value;
        if (autoLotDAL.ProcessCreditRisk(false, customerId) == false)
            MessageBox.Show($"ProcessCreditRisk failed for Customer {customerId}: id not found, or already present in Orders table");
    }

    autoLotDAL.LoadDataTable(dataGridViewCustomers.DataSource as DataTable);
    autoLotDAL.LoadDataTable(dataGridViewCreditRisks.DataSource as DataTable);
}
}
```