ADO.NET Connected Layer

CSIS 3540

Client Server Systems

Class 06

Topics

- Connecting to a Database
- Creating a database in Visual Studio
- Reading and Querying a database
- Insertion and Deletion
- Basic Transaction processing

ADO

- ADO – Active Data Objects
  - Terminology left over from COM/OLE/ActiveX
  - .NET added to use more modern methods
- Three faces of ADO
  - Connected Layer (this lecture)
  - Disconnected Layer
  - Entity Framework
- All can be used, but EF is the 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.

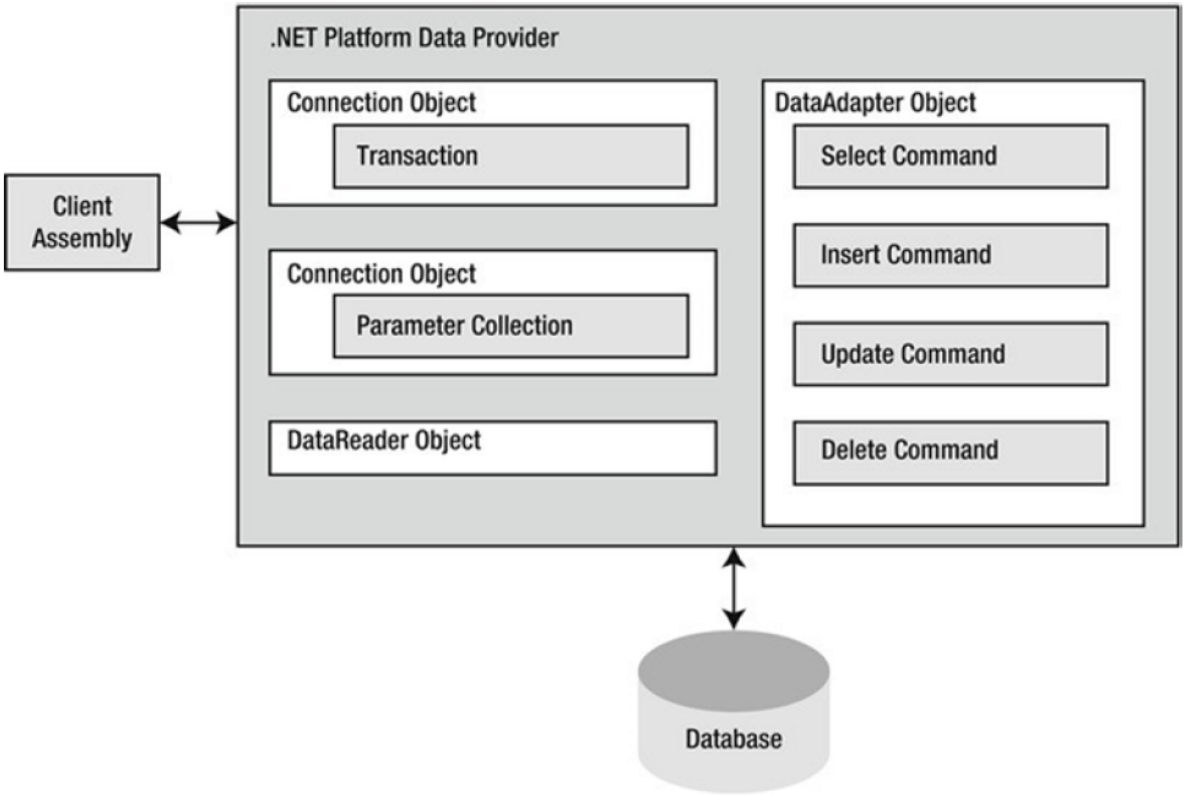


Figure 21-2. *ADO.NET data providers provide access to a given DBMS*

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)
`DataView`	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();
}
```

IDbCommand

- 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

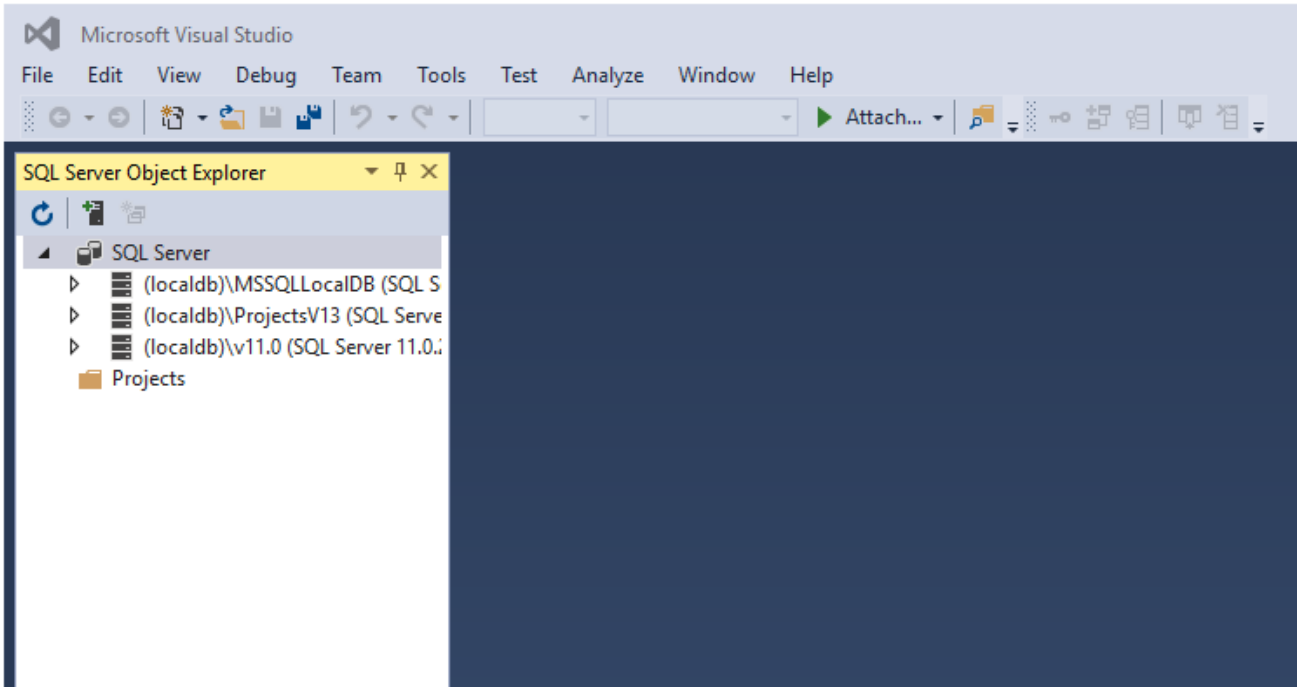
```
class SimpleConnectionProgram
{
        static void Main(string[] args)
        {
            WriteLine("**** Very Simple Connection Factory *****\n");

            // 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,

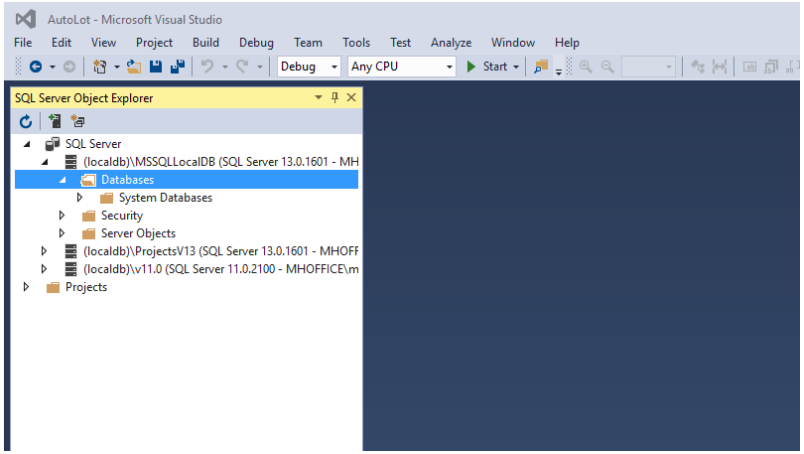
Creating the AutoLot database

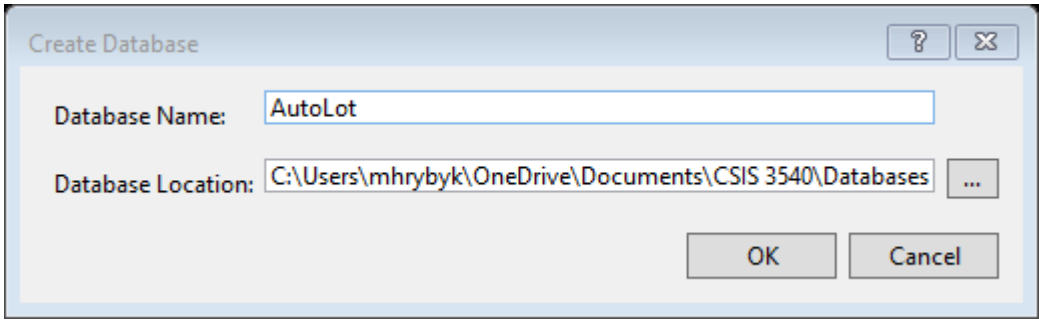
- Bring up Visual Studio, open SQL Server Object Explorer
  - Under View menu



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.

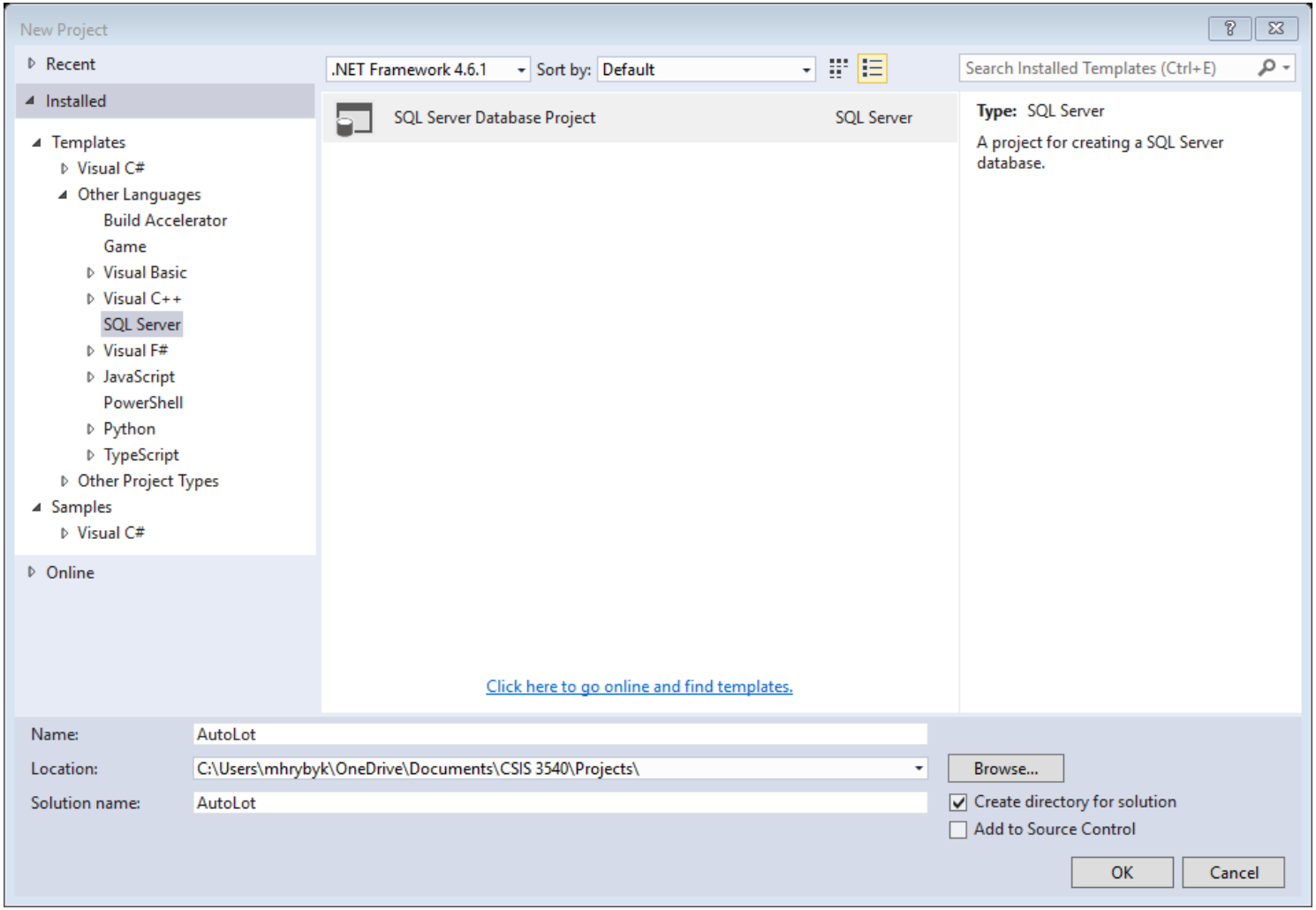




Now create a DB project

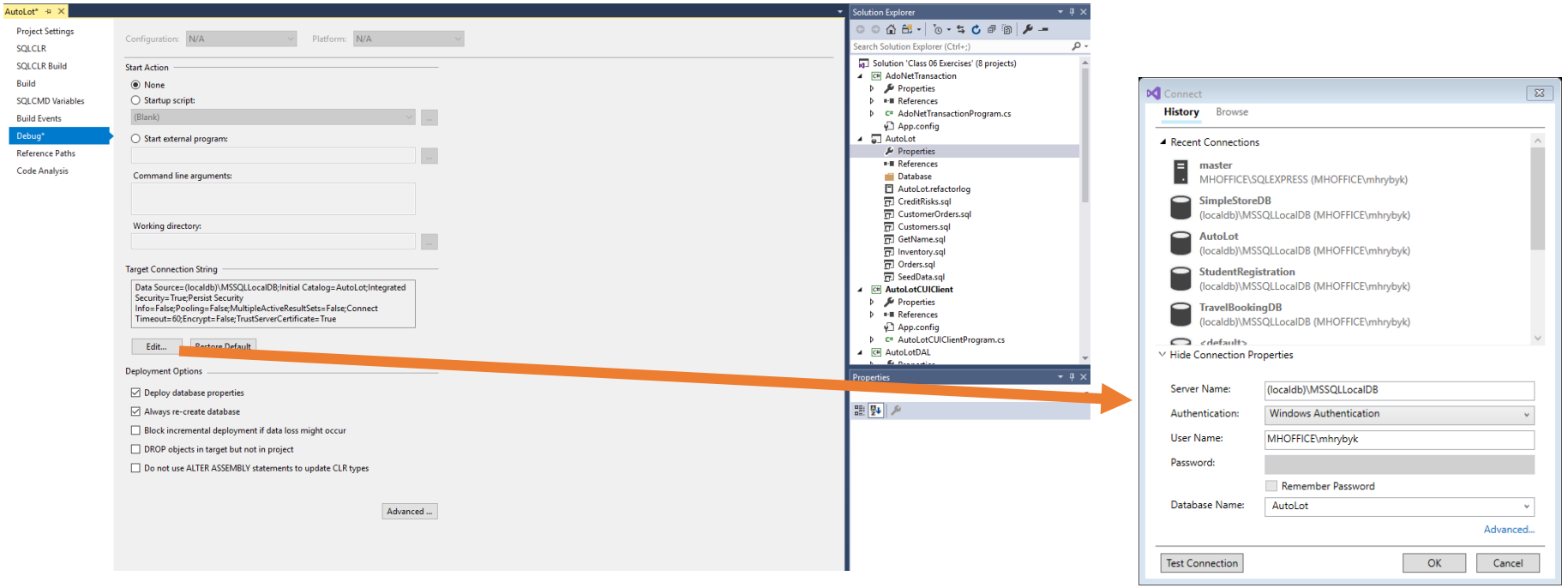
- Open VS
- Create a new project
- Select SQL Server Database project
- We will create SQL code to create tables, views, stored procedures, etc. within the project

Create a SQL Server Project: AutoLot

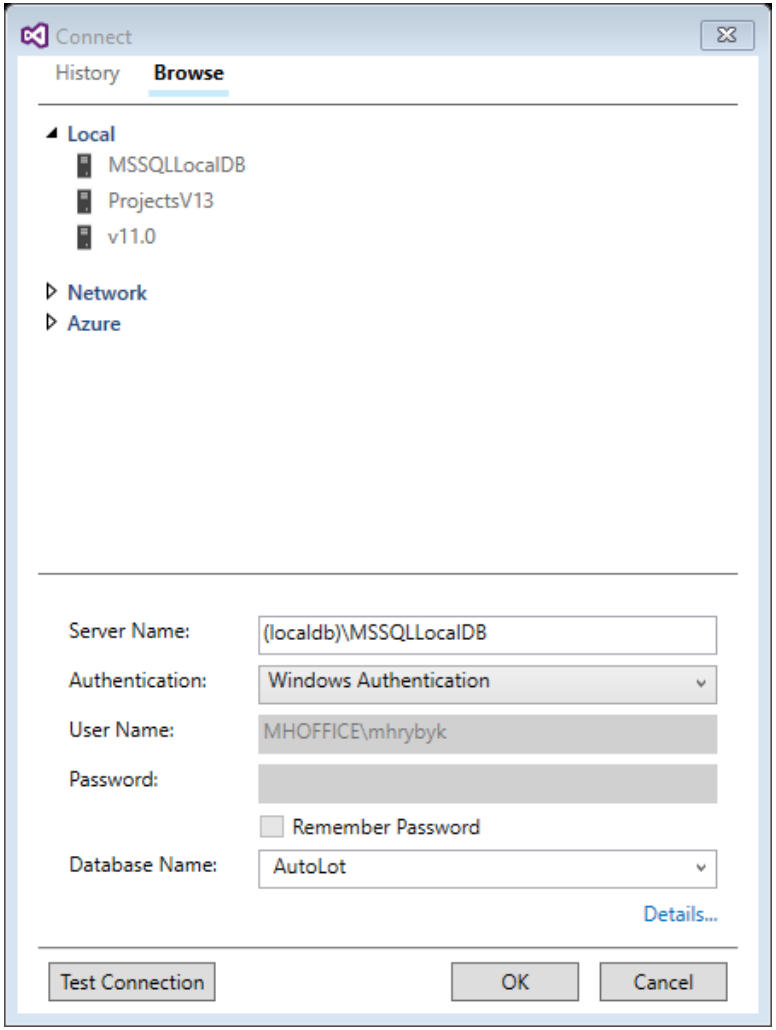


Double click on AutoLot Properties

- 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)
  - Probably best to Always re-create the Database
- We now have an Autolot database project which holds sql files, and can modify the AutoLot database.



Modify Debug Properties



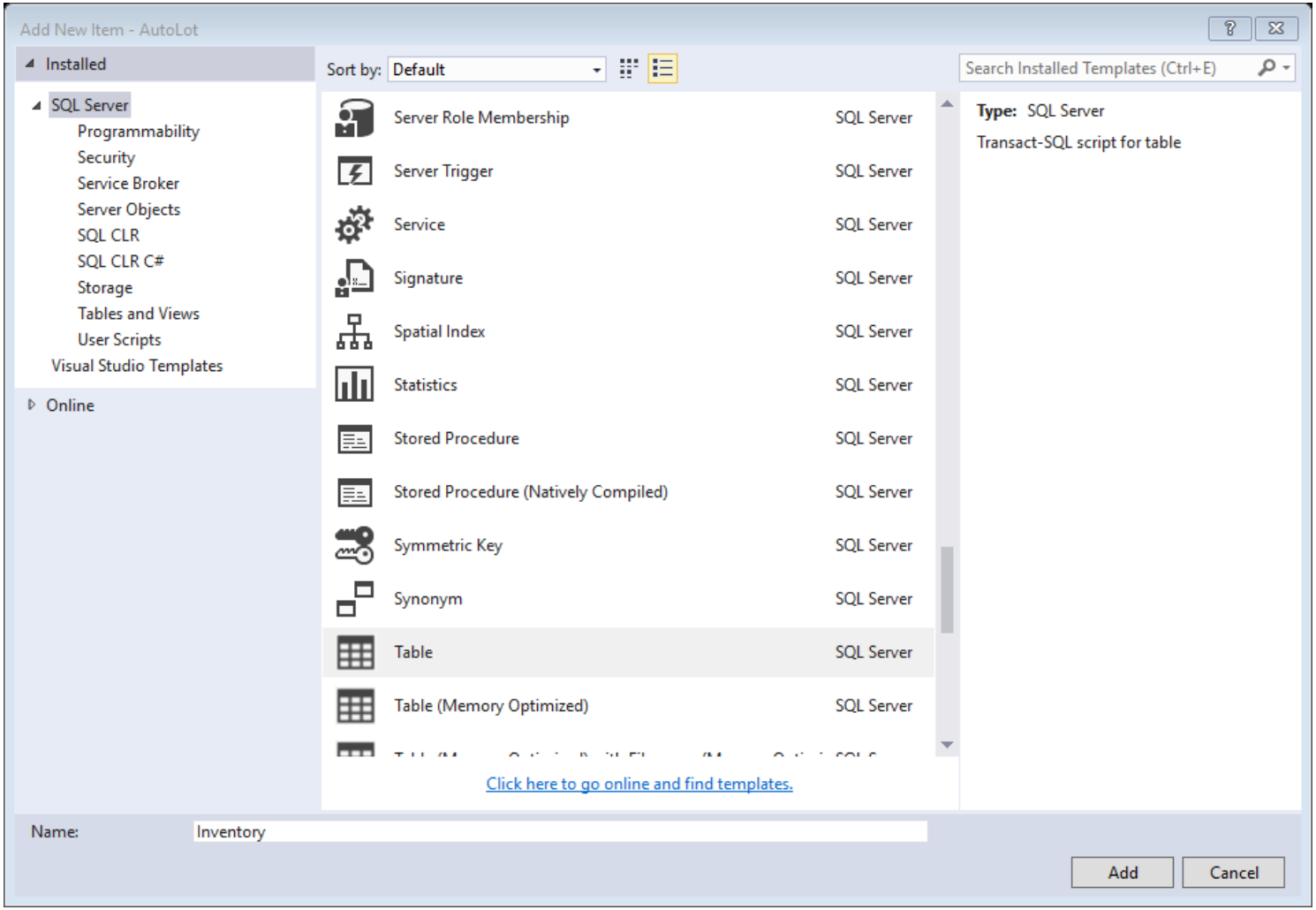
Data Connections

- 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

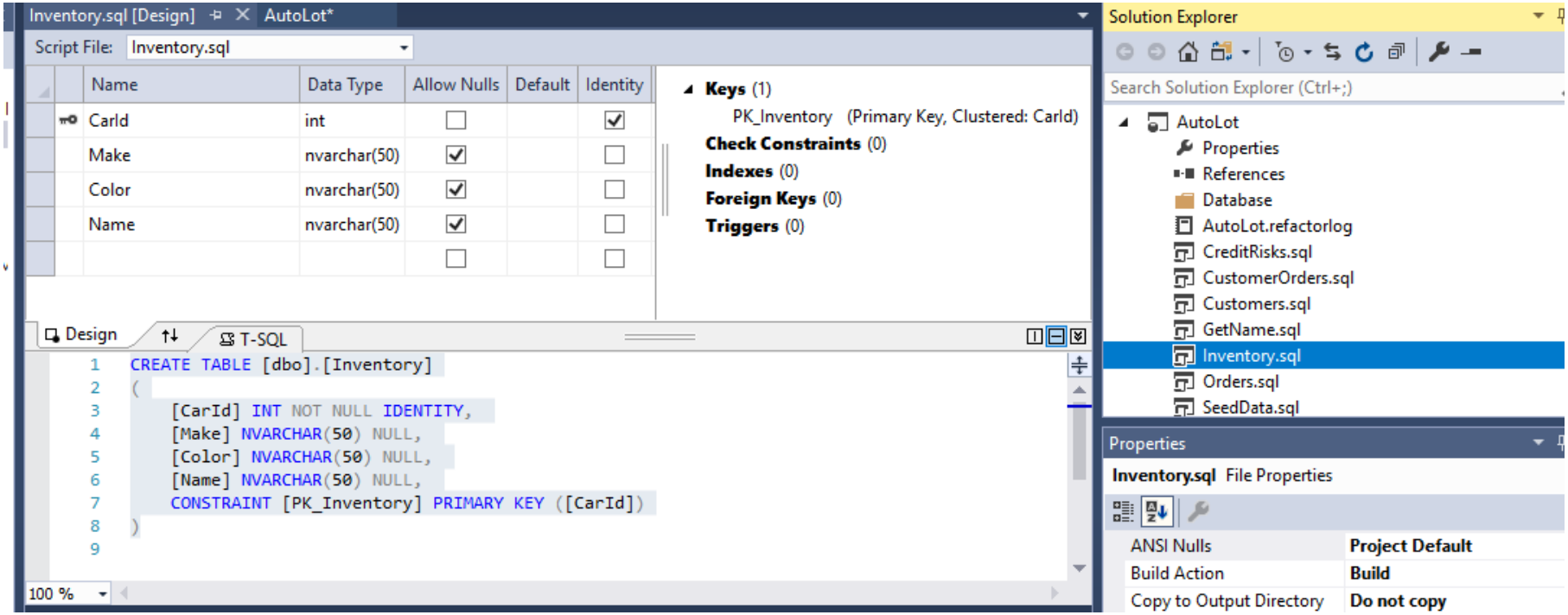
- In the AutoLot **PROJECT** add the following tables
  - Inventory
  - Customers
  - Orders

Add the Inventory Table

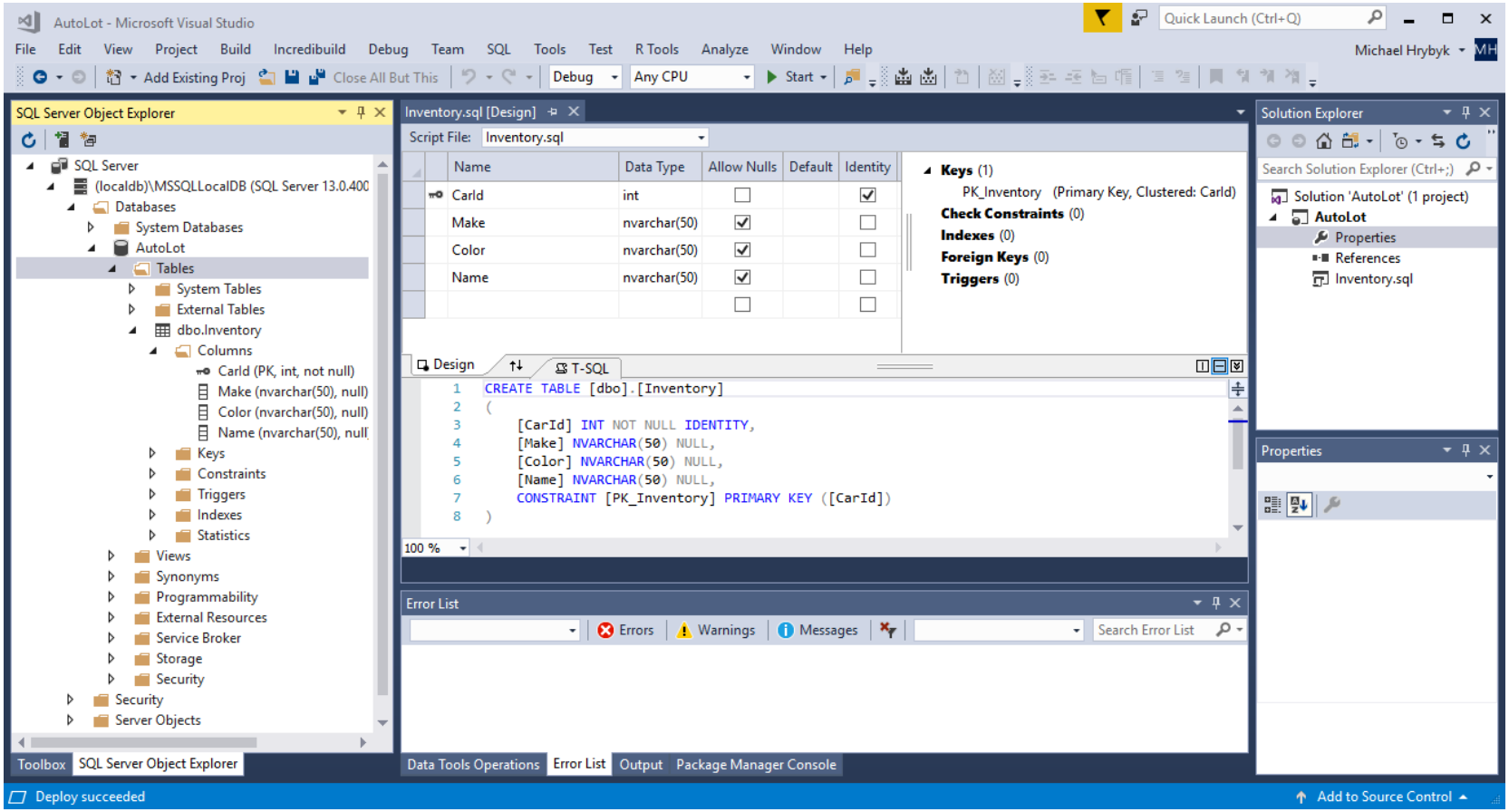


Add Inventory fields

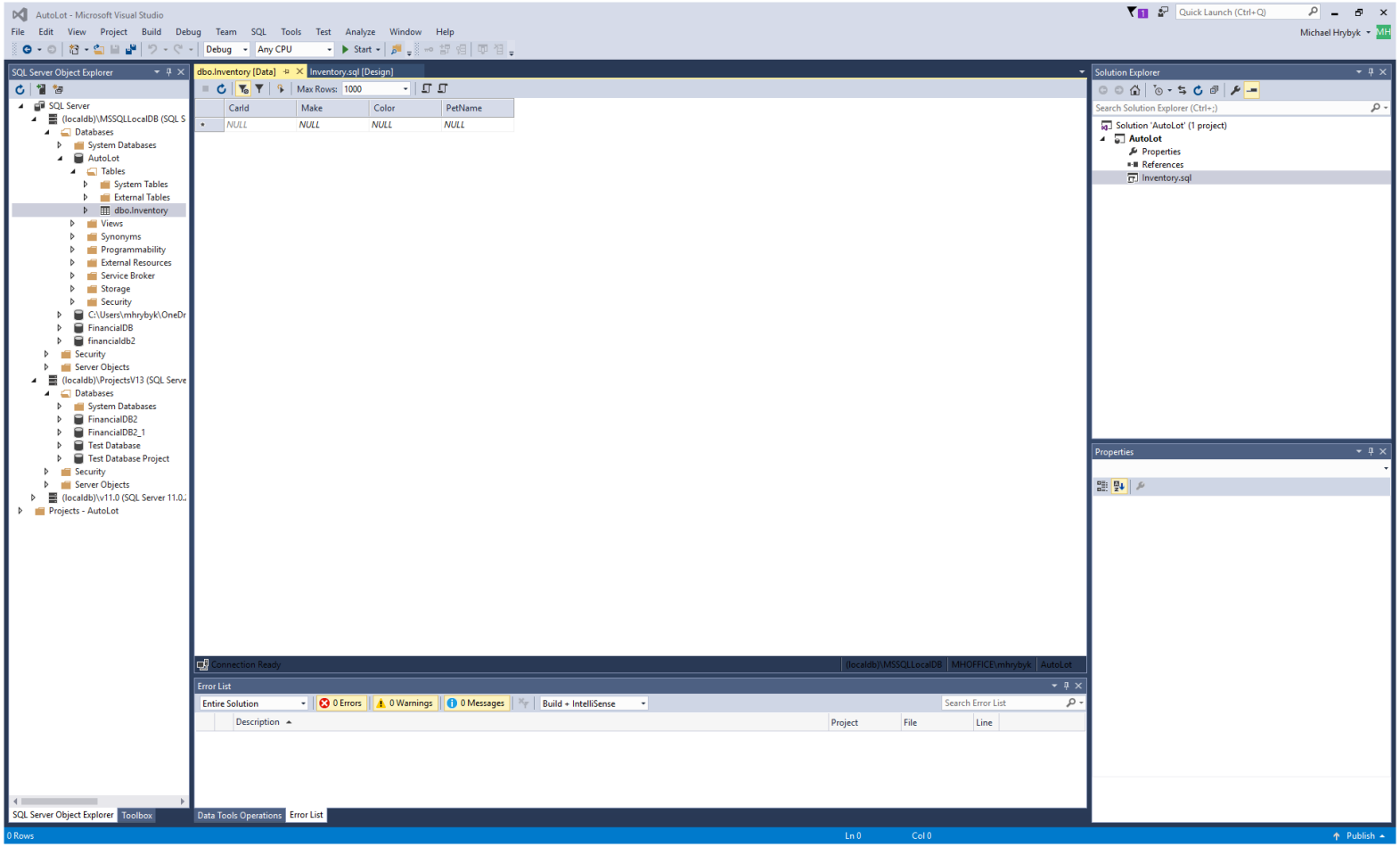
- Can use Designer or T-SQL pane
- Note use of Add Key/Constraint in upper right pane



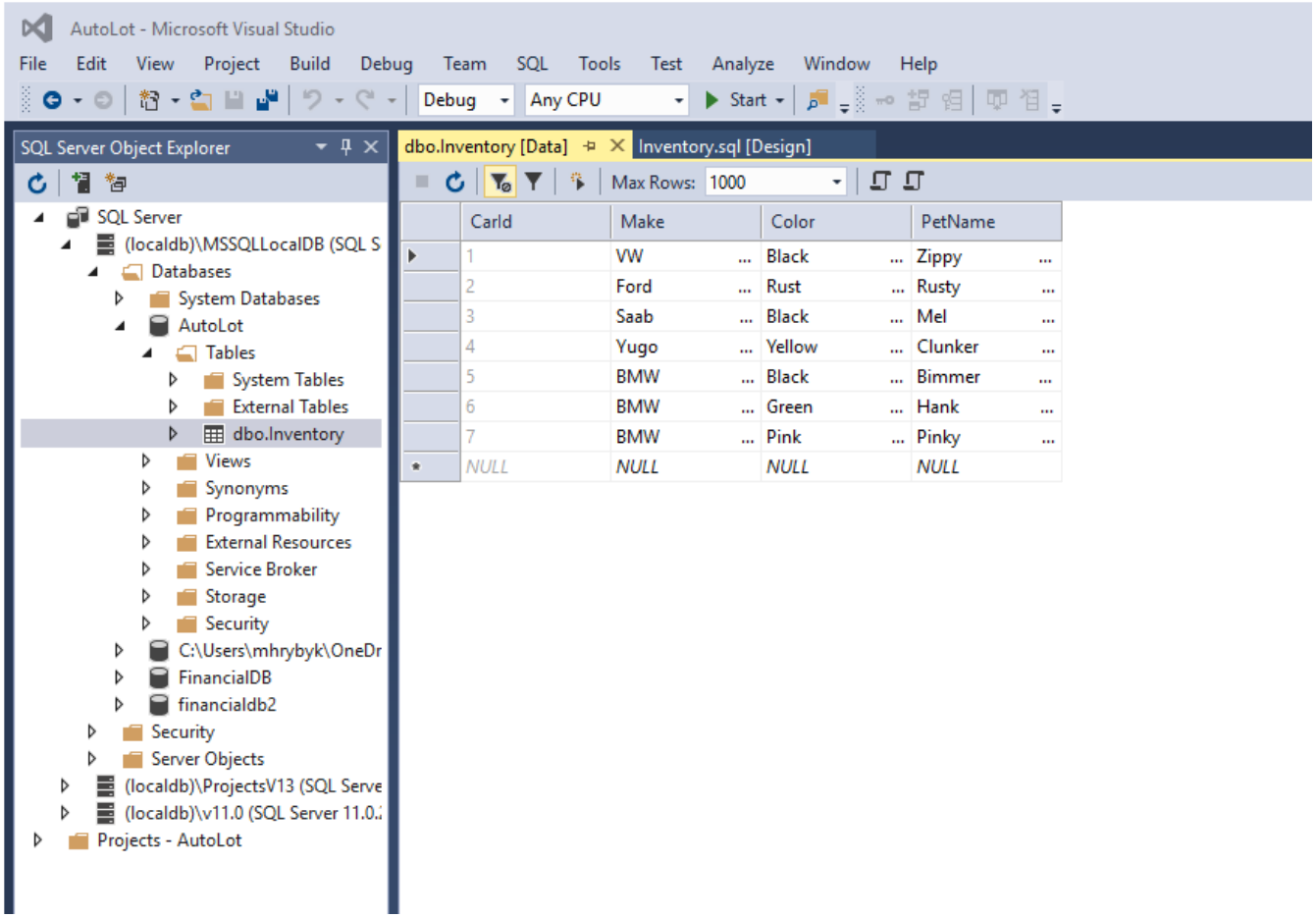
Hit Start, and refresh LocalDB



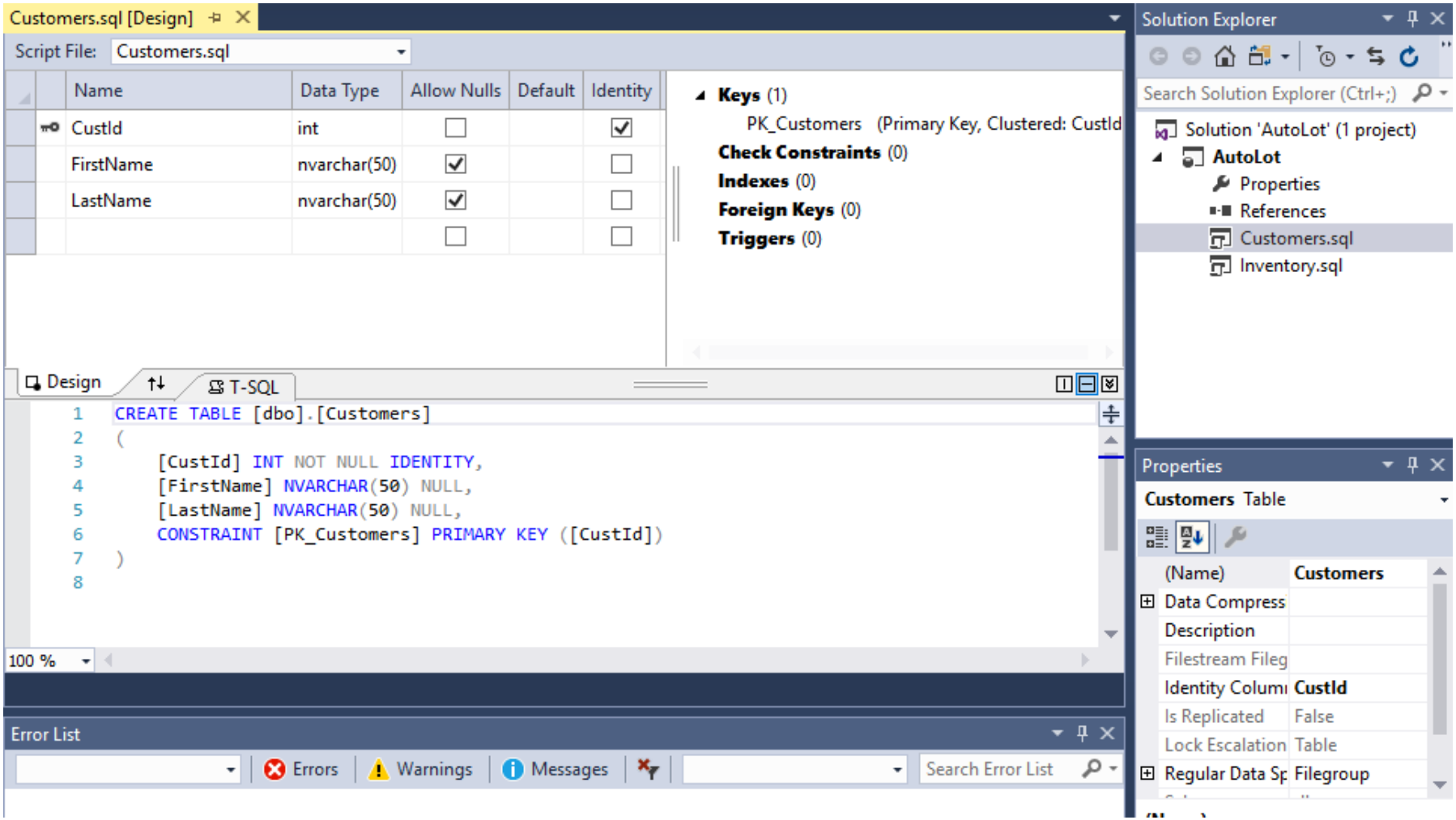
Expand AutoLot and right click on dbo.Inventory, choose Show Data



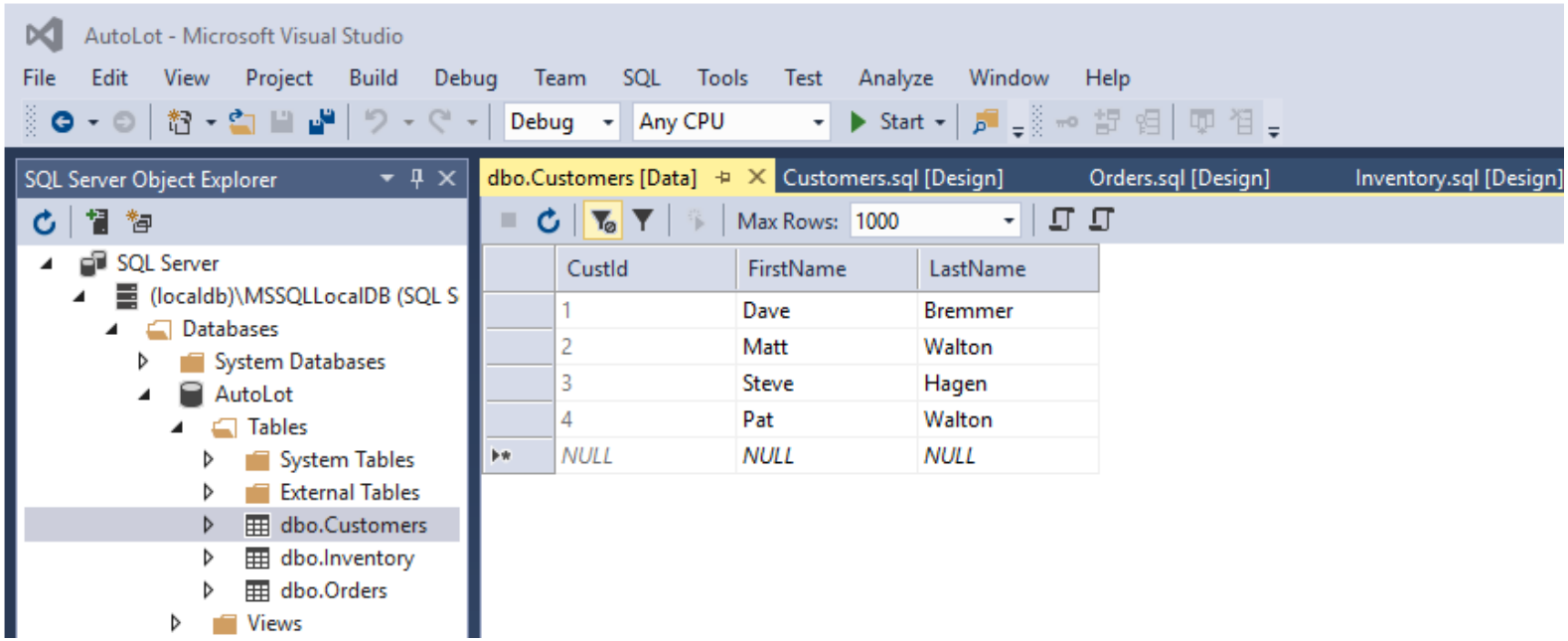
Enter the data and click refresh



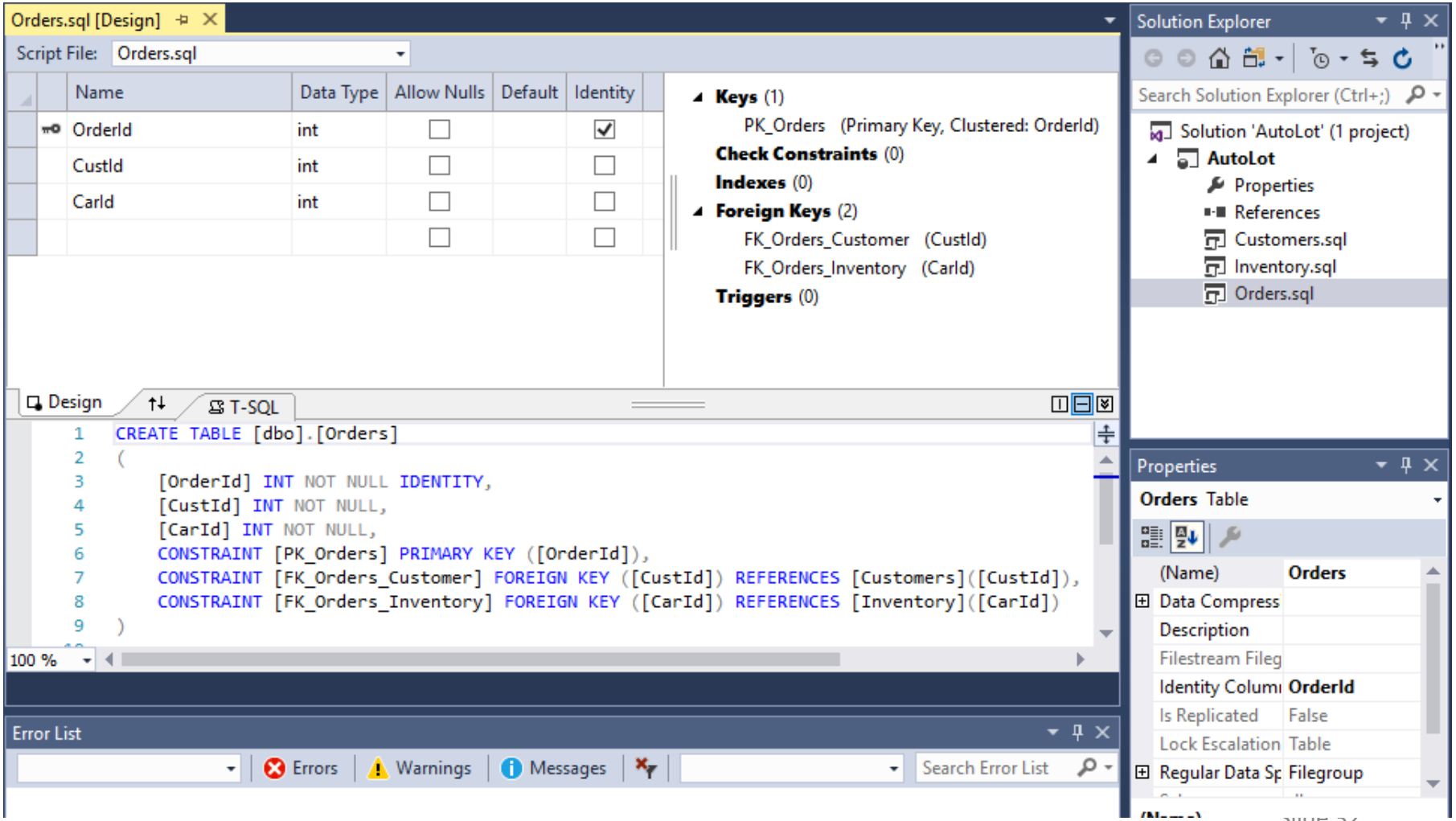
Create Customer table and hit Start



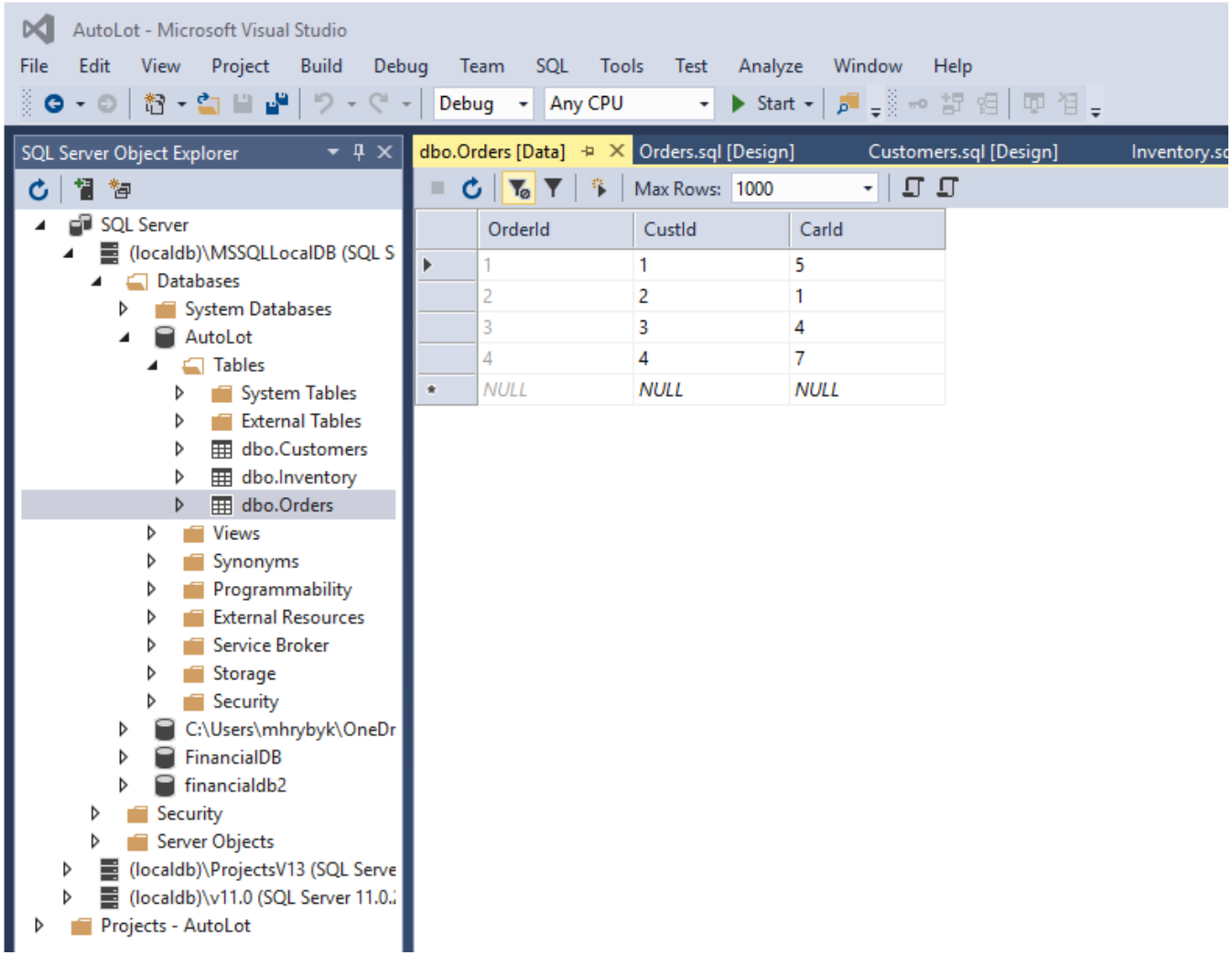
Add data to Customer table



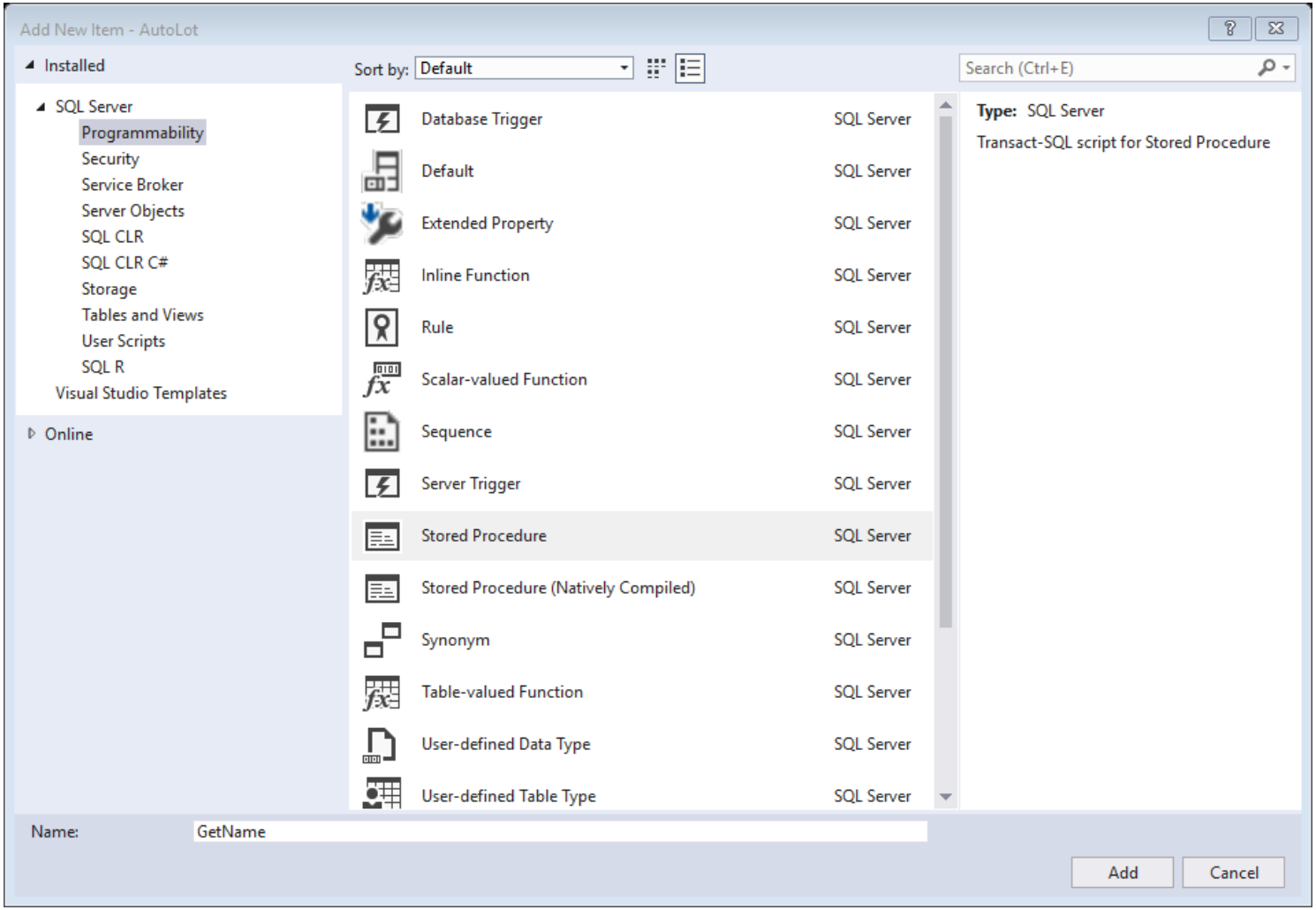
Create Orders table (including relationships)



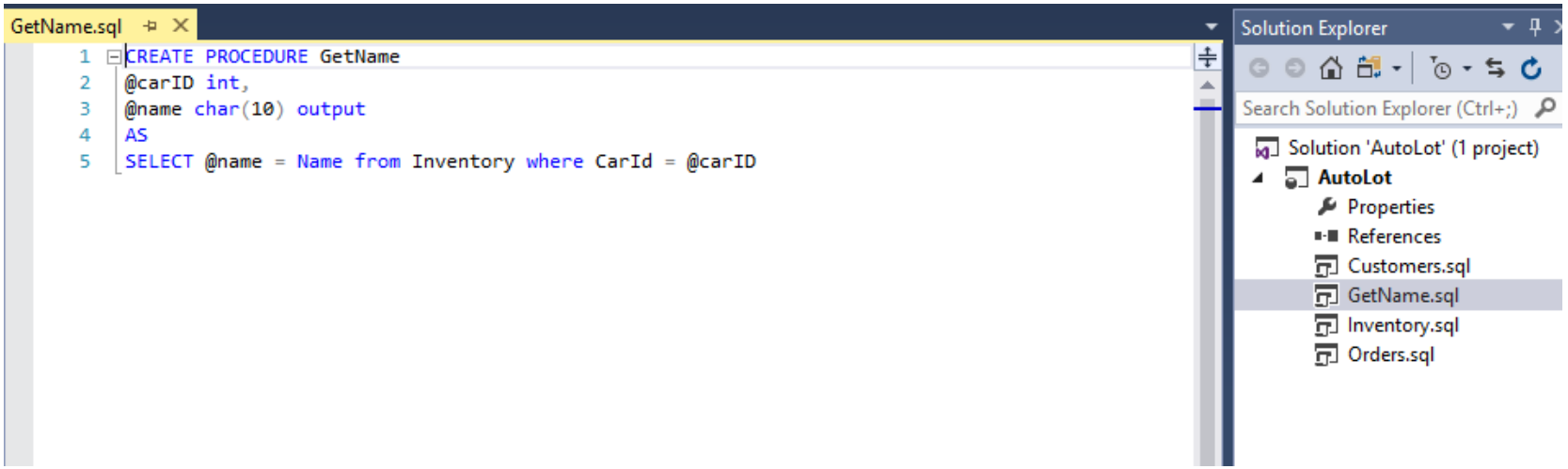
Add data to Orders



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
- Start AutoLot project
  - 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
`BeginTransaction()`	You use this method to begin a database transaction.
`ChangeDatabase()`	You use this method to change the database on an open connection.
`ConnectionTimeout`	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 `Connect Timeout` segment in the connection string (e.g., `Connect Timeout=30`).
`Database`	This read-only property gets the name of the database maintained by the connection object.
`DataSource`	This read-only property gets the location of the database maintained by the connection object.
`GetSchema()`	This method returns a `DataTable` object that contains schema information from the data source.
`State`	This read-only property gets the current state of the connection, which is represented by the `ConnectionState` 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)

Version 1 – read each record

- 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)

```
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"]}.");
                        }
            }
}
```

Version 2

- Similar, but reads all fields, printing out field name and value

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

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

            // Create a SQL command object.
            string sql = "Select * From Inventory;Select * from Customers";

            using (SqlCommand myCommand = new SqlCommand(sql, connection))
            {
                  //iterate over the inventory & customers
                  // Obtain a data reader a la ExecuteReader().
                  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());
                  }
            }
      }
```

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 which other code can use
- **See AutoLotCUIClient**
  - **CUI means console user interface**
  - **Uses the custom DataAccessLayer**
- **Study this in detail**

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 DataTable

- 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.

```
/// <summary>
/// Simple method to get a datatable from any sql table
/// </summary>
/// <param name="tableName">Name of the sql table</param>
/// <returns>DataTable containing the sql data</returns>
public DataTable GetDataTable(string tableName)
{
    DataTable dataTable = new DataTable();

    string sql = "Select * From " + tableName;
    using (SqlCommand cmd = new SqlCommand(sql, sqlConnection))
    {
        SqlDataReader dataReader = cmd.ExecuteReader();
        // Fill the DataTable with data from the reader and clean up.
        dataTable.Load(dataReader);
        dataReader.Close();
    }
    return dataTable;

}
```

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 AutoLotCUIClient
- 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();
    }
}
```

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()
  - Defined in AutoLotDAL
  - Used in AutoLotCUI
- Need to create CreditRisks table in AutoLot
- Removes a customer from Customers and moves to CreditRisks
  - Needs to be atomic
  - Use of commit and rollback